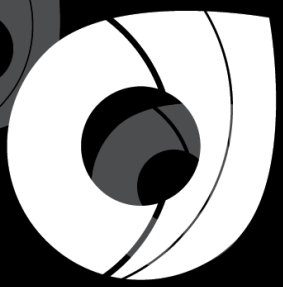
Corndel DevOps Engineering Programme

in association with Softwire

Module 5: Introduction to Immutable Infrastructure



Corndel Digital.

DA8L

Introduction

DevOps aims to break down traditional barriers between those who create software, and those who manage deployed applications. By sharing ownership, expertise and best-practices, the discipline aims to streamline the entire software lifecycle and continually update processes to leverage new tools and techniques.

The ways we deploy software into live environments are varied. Legacy applications might run on manually configured on-premises servers, and require manual updates, configuration adjustments and software deployments. More modern systems may make use of virtual machines - perhaps hosted in the cloud - but still entail an update process. Configuration management tools, introduced in the previous module, can help automate this, but do not completely remove the risks of configuration drift.

In this module we take things a step further, first introducing the concepts of **immutable infrastructure** and **infrastructure as code**. We'll review why these ideas have become hugely influential, and how your own applications can benefit from adopting these principles.

We will take a deep dive into **containers**, and see how this technology bundles an application, its environment and configuration into a standalone, immutable, system-agnostic package. We'll gain hands-on experience with **docker**, a popular piece of container software, and use it to create and deploy a simple Hello World application. You'll then apply these skills to move your own project into docker.

Finally, we take a step back and compare containers with the configuration management tools introduced in the previous module.

Table of Contents

Core Material

Additional Material

Chapter 1

Immutable Infrastructure

What is immutable infrastructure?

Immutability is the idea that once we create a thing, we don't change it.

When we talk about mutable and immutable infrastructure, we don't really mean that there's anything intrinsically different about the infrastructure itself; the terms relate to the approach we take when we provision or change that infrastructure. In this context, "infrastructure" means the machines that the software is running on, whether those are physical or virtual machines.

Mutable versus immutable infrastructure

At a high level, the difference can be summarised as follows:

- **Mutable:** ongoing upgrades, configuration changes and maintenance are applied to running servers. These updates are made in place on the existing servers. Changes are often made manually; for example, by SSH'ing into servers and running scripts. Alternatively, a configuration management tool might be used to automate applying updates.
- **Immutable:** once a server is set up, it is never changed. Instead, if something needs to be upgraded, fixed or modified in any way, a new server is created with the necessary changes and the old one is then decommissioned.

This represents a key conceptual difference in how servers and other infrastructure should be treated. Let's take an example scenario and see how it would play out with these two approaches. Imagine you decide to upgrade the version of the web server running on your production infrastructure. If you're taking a mutable approach to your infrastructure, you would upgrade the software on each server, thereby changing their configuration. If, on the other hand, you treat the infrastructure as immutable, you would instead set up entirely new servers with the upgraded version of the web server software, then replace the existing servers with these new ones, switching over external traffic to the new servers once they have been validated.

Key differences

Mutable Servers	Immutable Servers
Long-lived (years)	Can be destroyed within days
Updated in place	Not modified once created
Created infrequently	Created and destroyed often
Slow to provision and configure	Fast to provision (ideally in minutes)
Managed by hand	Built via automated processes

A Bit of History

The rise of immutable infrastructure is largely due to the development of two core technologies: **virtualisation** and **cloud computing**. Together, these allowed virtual machines to be created and destroyed with minimal effort. It became much cheaper to spin up new servers (both in terms of cost and effort), and this led to a different conceptual approach. In addition, the process could be automated, allowing new VM instances to be created with identical configurations.

Previously, on-premise server rooms were the standard setup for many organisations. This remains the case, particularly in certain slow-moving and highly regulated industries. **What other reasons do you think industries might have for wanting dedicated, on-premise server rooms?**

Data centres became increasingly popular, either the colocated ("colo") type, where the centre provides power and physical space for your own equipment, or where all equipment (including server hardware) is leased from the data centre.

Virtualisation first became widely available in the late 1990s, with VMware and IBM both providing popular virtual machine solutions.

Cloud computing in its modern form started to take off in the mid-2000s, with Amazon Web Services, then later Google Cloud Platform and Microsoft Azure becoming the dominant players. We will be going into more detail on popular cloud infrastructure platforms and the services they offer in later modules.

Benefits of immutable infrastructure

Despite the apparent inconvenience of having to effectively recreate (part of) your infrastructure each time you need to change something, there are a number of important benefits that immutable infrastructure offers:

- **Reproducibility / consistency** - It is guaranteed that multiple servers created from the same configuration will be identical. Software will run in the same way on each such server.
- **Avoids "configuration drift"** - When mutable servers are updated in place it is possible for configuration differences to emerge between different machines. This drift can result in differences in behaviour that can be difficult to track down. Immutable infrastructure ensures all servers are identical upon creation, eliminating configuration drift (or at least making it much easier to spot).
- **Faster horizontal scaling capability ("just in time" provisioning)** - Adding more servers to a running environment is as simple as provisioning new servers and adding them to the pool (normally a fast process). There is no need to do any updates or changes (normally a slow process) on the servers, they will be in the exact state needed following the provision.
- **Less manual effort thanks to automation** - Immutable infrastructure is easy to automate - provisioning is done based on a written configuration and the success of it can be automatically verified. There is no manual work needed on the servers themselves.
- **Environment consistency (test/staging environments identical to production)** - The same configuration can be used in all environments to avoid different behaviour between them due to infrastructure
- **Atomic deployments** - Deployments either complete successfully or - nothing changes. Status of deployments and server provisioning can be tested automatically based on known expected status
- **Allows for deployment strategies like *blue-green deployment* or *rolling releases*** - These deployment strategies lower the risk of environments ending up in a broken state. Rollback and recovery is simplified and faster, as traffic can be re-routed to the old instances again.

- *Blue-green deployment:* This strategy creates two identical environments (Blue and Green). Only one environment is live at any given time (a router directs all traffic to it). Updates can be done to the offline environment and, once complete, switch the router to point to the updated environment.
- *Rolling releases:* Infrastructure updates are applied to a cluster of servers incrementally, allowing performance in the live environment to be evaluated. A successful deployment is gradually rolled out to all servers, an unsuccessful one is rolled back. Updates are generally small and frequent.

Requirements for immutable infrastructure

Reproducible configuration

Immutable infrastructure requires that servers and other infrastructure components can be reliably created with identical setup. What is the best way to make an object reproducible? There are three basic steps:

- Document exactly what is required to create the object.
- Create scripts that will build and assemble the components into the object as described in the documentation.
- Automate the process.

Configuration scripts and setup documentation can (and should) be stored in source control (e.g. git). This workflow is known as **Infrastructure as Code** because it involves treating infrastructure configuration with the same good practices that are commonly applied to application source code.

Infrastructure as Code (sometimes abbreviated as **IaC**) is the concept of managing an environment (infrastructure) in the same way as different applications are managed. Developers create source-controlled code files defining the desired configuration and state of the infrastructure. An engine then uses these files to produce infrastructure satisfying the desired state. In this module we will learn about **Dockerfiles**, which are a good example of infrastructure as code. IaC is an important concept, and we'll come back to it in future modules.

Rapid provisioning of new infrastructure

For immutable infrastructure to be practical, new servers and other infrastructure need to be created and validated much faster than mutable infrastructure. Cloud computing platforms and other virtualised environments which are optimised for rapid creation of new instances meet this need.

Full automation of deployment pipeline

Creating new infrastructure by hand is time-consuming and prone to error. This issue is magnified when using immutable infrastructure, since new instances need to be created much more frequently. The only way this approach becomes feasible is if the process is fully automated. Developing an automated deployment pipeline can add a high upfront cost to a project, but this is quickly amortised over the lifetime of the service, as future releases are greatly simplified.

Stateless applications

When following an immutable infrastructure philosophy, it's essential that all servers deployed in the same state remain in the same state. This way, one can be replaced by an identical copy at any time, without loss of data or interruption of service.

This is only possible if the applications running on these servers are *stateless*. Data must be separated from application logic, with specialised database servers deployed separately.

This means that the service must be designed with a **persistent data layer**, which supports multiple application connections and needs to consider deployment/migration implications.

Pets versus Cattle

A common analogy that is used to highlight the difference between mutable and immutable infrastructure is that of "pets versus cattle". Mutable infrastructure is lovingly cared for, like a pet, with ongoing fixes and changes, careful maintenance, and treated as "special". Immutable infrastructure is treated with indifference – spun up, put into service, and removed once no longer required; each instance is essentially identical and no server is more special than any other.

You'll probably hear the "snowflakes versus phoenixes" analogy, too. Snowflakes are intricate, unique and very hard to recreate. Whereas phoenixes are easy to destroy and rebuild; recreated from the ashes of previously destroyed instances!

Chapter 2

Containers

Containers are isolated environments that allow you to separate your application from your infrastructure. They let you wrap up all the necessary configuration for that application in a package (called an image) that can be used to create many duplicate instances. **Docker** is the most popular platform for developing and running applications in containers, and has played a key role in the growth of immutable infrastructure.

In this section we discuss containers and benefits, before exploring docker in more detail.

Containers and why they're useful

Containers wrap up a piece of software in a complete file system that contains everything it needs to run: code, configuration, runtime, dependencies, system libraries — anything you can install on a server. This guarantees that it will always run the same, regardless of the environment it is running in, whether that's a private server, cloud platform or on your laptop. Think of a container as a self-contained package designed to run a specific application. The only software required on the underlying host machine is the software needed to run the container.

Like virtual machines (discussed in the previous module), containers allow you to package your application together with libraries and other dependencies, providing isolated environments for running your software services.

Containers have several key features:

1. *Lightweight* – with much smaller disk and memory footprints than virtual machines.
2. *Fast* – new containers start up in milliseconds.
3. *Isolated* – each container runs separately, with no dependency on others or the host system.
4. *Reproducible* – creating new containers from the same image, you can guarantee they will all behave the same way.

Together, these features make it much easier to run many duplicate instances of your application and guarantee their consistency. Since they take up significantly fewer resources than virtual machines, you can run many more containers on the same hardware, and start them quickly as needed. Containers are also able to run virtually anywhere, greatly simplifying development and deployment: on Linux, Windows and Mac operating systems; on virtual machines or bare metal; on your laptop or in a data centre or public cloud.

Docker is a popular open-source container format and runtime and is what we'll be using in this course; however, other container software platforms exist and we'll discuss them briefly in this module. Part of the reason for Docker's popularity is its broad ecosystem of available images (the blueprints from which containers are created). This means that there are usually "off-the-shelf" images already available for you to download and use for things like standard web server or database installations, which can then be tweaked to suit your needs.

The reproducibility that containers provide — guaranteeing that the same dependencies and environment configuration are available, wherever that container is run — also has significant benefits for local development work. In fact, it's possible to do most of your local development with the code being built and run entirely using containers, which removes the need to install and maintain different compilers and development tools for multiple projects on your laptop. This leads to the concept that everything can be containerised in the software development lifecycle: local development tooling, continuous integration and deployment pipelines, testing and production environments. We'll explore this concept further in the project exercise for this module, where we'll create container images to run and test our todo app in development, as well as a production-ready container image.

Terminology

Container: A self-contained environment for running an application, together with its dependencies, isolated from other processes. Containers offer a lightweight and reproducible way to run many duplicate instances of an application. Similar to virtual machines, containers can be started, stopped and destroyed. Each container instance has its own file system, memory, and network interface.

Image: A sort of "blueprint" for creating containers. An image is a package with all the dependencies and information needed to create a container. An image includes all the dependencies (such as frameworks) plus deployment and execution configuration to be used by a container runtime. Usually, an image is built up from base images that are layers stacked on top of each other to form the container's file system. An image is immutable once it has been created.

Tag: A label you can apply to images so that different images or versions of the same image can be identified. Docker image tags have the format

Volume: Most programs need to be able to store some sort of data. However, images are read-only and anything written to a container's filesystem is lost when the container is destroyed. Volumes add a persistent, writable layer on top of the

container image. Volumes live on the host system and are managed by Docker, allowing data to be persisted outside of the container lifecycle (i.e., survive after a container is destroyed). Volumes also allow for a shared file system between the container and host machine, acting like a shared folder on the container file system.

Container Software

Container Software is used to generate and manage multiple containers using OS-level virtualization. Containers are generated using predefined configurations or images. Container software typically contains a runtime, plus higher-level components to handle container management.

The most popular container software is Docker and it is widely used in the DevOps community. We will focus on Docker as part of this module and it will be detailed further in the next section.

Docker is not the only container software on the market, however. Alternatives you may encounter include:

- **LXC** - an acronym for Linux Containers. LXC is an open source tool for executing Linux-only containers on a Unix OS. LXC is designed to run whole operating systems in containers; this contrasts with the docker philosophy of a container as a single application. Early docker versions were based on LXC containers.
- **Fedora CoreOS** - the recently released community successor of CoreOS. The original CoreOS and its **rkt** container runtime were a security-focused alternative to Docker. Since accquisition by Red Hat, much of the core functionality of has been merged into Red Hat Openshift, a enterprise container orchestration tool (more on those below).

Containers are designed to scale. As well as the container software itself, there are numerous tools that can help you run containers in large numbers and/or spread across multiple host machines. This is known as **container orchestration**, and is an advanced topic we'll return to later. For now, it's worth knowing the basics, so you can understand how these technologies relate to Docker and other container runtimes.

- **Docker Swarm** - Docker's built-in container orchestration tool. It creates a single virtual host out of multiple host machines, and distributes docker workloads across them.
- **Kubernetes** - a popular container orchestration tool developed by Google. Kubernetes can be used with

multiple different container runtimes, although Docker remains the most popular.

- **Portainer** - an open-source tool that provides a clean web interface to facilitate container and cluster management.

As well as the tools discussed above, the main cloud infrastructure providers have their own offers. These include proprietary container orchestration tools, as well as managed versions of popular open-source tools. We'll discuss some of these services when introducing cloud infrastructure later in the course. Here we summarise the key offerings, simply to provide some context:

1. **AWS ECS** - Amazon's proprietary container management software. It can be used to run docker images in the cloud and at scale.
2. **AWS EKS** - EKS provides a managed kubernetes environment. EKS is a more complex and powerful alternative to ECS.
3. **Google Kubernetes Engine** - Similar to AWS EKS, the Google Cloud Platform provides managed kubernetes clusters.
4. **Azure App Service** - Microsoft's proprietary container management software.
5. **Azure Kubernetes Service** - A managed kubernetes environment hosted on Microsoft Azure. Similar in function to AWS EKS or Google Kubernetes Engine.

Chapter 3

Introduction to Docker

What is Docker?

Docker is an open source software program designed to make it easier to create, deploy and run applications by using containers.

It is by far the most commonly used container software, being part of many DevOps toolchains. It is used by both developers and system administrators. Developers find it helpful as they can run code without worrying about the infrastructure the application will be running on. They can also use local containers identical to the containers used in production for their development environment. It also helps new developers ramp up more quickly: they don't have to install project specific software or libraries on their local machine, everything will already be within the Docker container(s). For DevOps, Docker will make deployments and server configuration easier - they will not need to worry about manually installing software on the servers.

Docker is configured using Dockerfiles. These contain configuration code that instructs Docker to create **images** that will be used to provision new containers.

Docker consists of the following key components. We'll discuss each in detail:

- Docker objects (containers, images and services)
- The docker engine - software used to run and manage a container
- Docker registries - version control for docker images (similar to git)

Docker Concepts

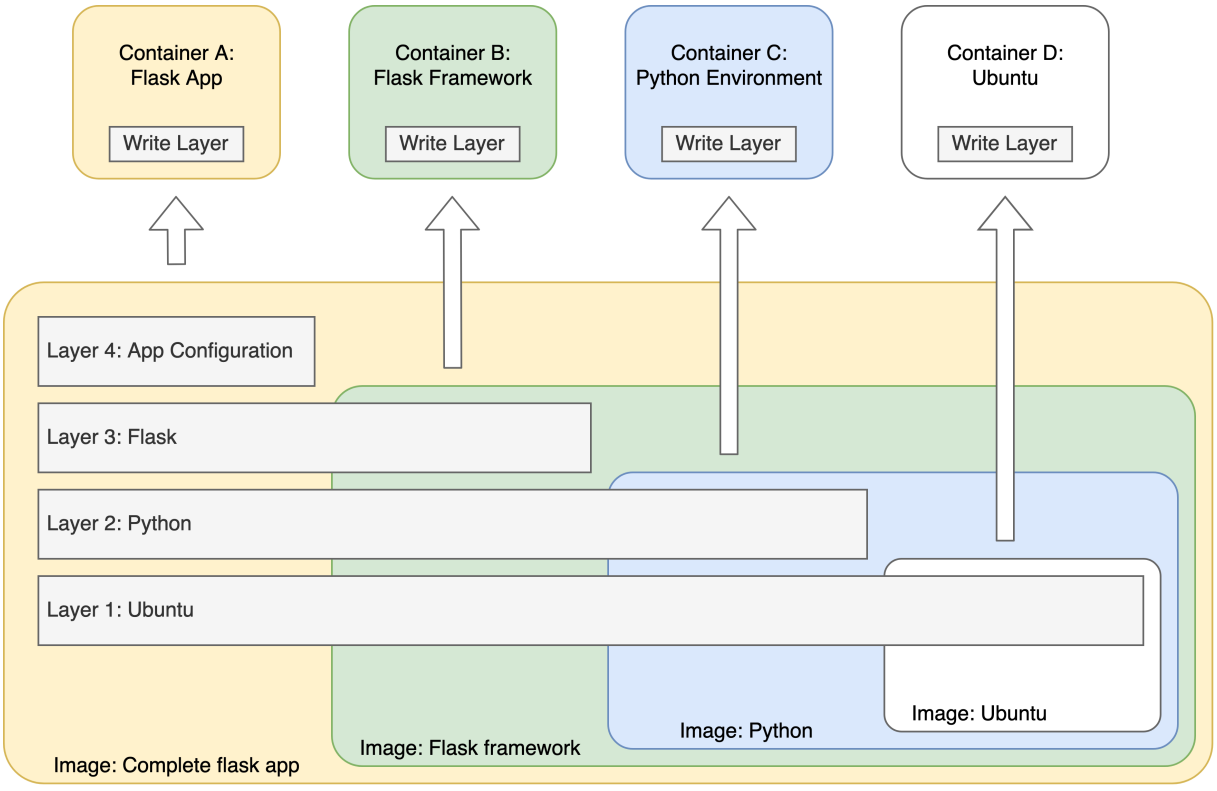
Images

If you've ever used virtual machines, you'll already be familiar with the concept of images. In the context of virtual machines, images would be called something like "snapshots". They're a description of a virtual machine's state at a specific point in time. Docker images differ from virtual machine snapshots in a couple of important ways, but are similar in principle. First, Docker images are read-only and immutable. Once you've made one, you can delete it, but you can't modify it. If you need a new version of the snapshot, you create an entirely new image.

This immutability is a fundamental aspect of Docker images. Once you get your Docker container into a working state and create an image, you know that image will always work, forever. This makes it easy to try out additions to your environment. You might experiment with new software packages, or make changes to system configuration files. When you do this, you can be sure that you won't break your working instance — because you can't. You will always be able to stop your Docker container and recreate it using your existing image, and it'll be like nothing ever changed.

The second key aspect of Docker images is that they are built up in layers. The underlying file system for an image consists of a number of distinct read-only layers, each describing a set of changes to the previous layer (files or directories added, deleted or modified). Think of these a bit like Git commits, where only the changes are recorded. When these layers are stacked together, the combined result of all these changes is what you see in the file system.

The main benefit of this approach is that image file sizes can be kept small by describing only the minimum changes required to create the necessary file system, and underlying layers can be shared between images.



Docker Images and Layers

The layered file system also allows programs running in a container to write data to their container's file system (remember that the file system layers of the image are read-only, since the image is immutable). When you create a new container, Docker adds a new writable layer on top of the underlying layers from the image. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. This also minimises the disk space required to create many containers from the same image - only the thin writable layer for each container needs to be created, while the bulk of the file system is shared from the read-only image layers.

Because of their immutability, Docker images are uniquely tagged so that you can choose a specific version/variant when creating a container. Image tags will often include a version number that corresponds to the version of the main application they provide. For example, images for Java or Python languages are typically tagged with the version of the language runtime that they provide (`python:2.7`, `python:3.6`, `openjdk:8`, etc.). You can think of an image tag as equivalent to a git branch or tag, marking a particular unique version of an application. If a tag is not specified then the default `latest` tag is used. This applies when building a new image or when creating a container from an existing image. Multiple tags can also refer to the same image, for example `my_app:2.3` and `my_app:latest` could both refer to the newest build you created for an image of your app.

There are a number of curated official images that are publicly available. These images are designed to:

- Provide essential base OS images (e.g. ubuntu, centos) that serve as the starting point for the majority of users.
- Provide drop-in solutions for popular programming language runtimes, data stores, and other services, such as Ruby, Python, MySQL or Elasticsearch.
- Exemplify Dockerfile best practices and provide clear documentation to serve as a reference for other Dockerfile authors.
- Ensure that security updates are applied in a timely manner.

One of the key benefits of Docker images is that they allow custom images to be created with only minimal size increase from existing base images.

Images are created in a layered manner, in which a new image can be created upon an existing image by adding another layer that contains the difference between the two. In contrast, the image files of different VMs are isolated from each other, so each must contain a full copy of all the files required by its operating system.

Containers

We've already discussed containers in general, and mentioned them in the context of docker images. In Docker, a container runs a *single* application, and a full service would typically consist of many containers talking to each other, and the outside world.

A container is made from an image. The image is a template. A container will add a writable layer on top of the image that stores all file modifications made within the container. A single image can be used to create many identical containers, and common layers will be shared between containers to reduce disk usage. You can make a container from the command line:

```
$ docker run <image_name>
```

Just like an application, you can supply containers with runtime parameters. These are different from build-time parameters included in images. Runtime parameters are often used to inject secrets or other environment-specific configuration. The example below uses the `--env` option create a container and add a `DB_USER` environment variable. This variable can be used by the application running in the container.

```
$ docker run --env DB_USER=<database_user> <image_name>
```

Docker containers may host long-lived applications that only terminate on an unhandled exception or shutdown command (e.g. a web server), or scripts that exit when complete. When a container stops running - either through an error or successful completion - it can either be restarted or discarded. We could restart the container ourselves using `docker restart`. We can also get the docker container to restart itself in case of unhandled exceptions, through `docker run --restart=always` or `docker run --restart=unless-stopped`. However, eventually all containers must die.

In this way, you can think of a container itself as ephemeral, even if the application it hosts is long-lived. At some point a container will stop, or be stopped, and another will be created to continue running the same application. For this to work without data loss, containerised applications must be *stateless*. They should not store persistent data

within the container, as the container may be destroyed at any time and replaced by another. Any data on the writable container layer is lost when this happens.

The distinction between an ephemeral container, and the persistent service that the container provides, is essential to understanding how containerised services can scale to meet demand. We will develop this concept further in later modules when we discuss container orchestration.

Stateless applications are necessary if we want to treat Docker containers as immutable infrastructure (which we certainly do, for all the scalability and maintainability reasons discussed previously). To achieve this, data storage should happen outside of the container - either through a network connection (e.g. a cloud database) or on the local filesystem stored outside the container. This can be achieved through use of **volumes** or **bind mounts**.

Volumes and Bind Mounts

Normally, a container only has access to the files copied in as part of the image build process. Volumes and bind mounts allow container to access external, shared file systems that persist beyond the container lifecycle. Because these file systems are external to the container, they are not destroyed when a container is removed. The same volume or bind mount can also be attached to multiple containers, allowing data sharing.

- A **bind mount** attaches an existing file or directory from the host machine to a container. This is useful for sharing pre-existing data or host configuration files with containers.
- A **volume** creates a new docker-managed file system, and attaches it to containers. Volumes are more flexible than bind mounts for general-purpose data storage, and more portable as they do not rely on host system file permissions.

Volumes are the preferred tool for storing persistent local data on containerised applications. A common use case is a containerised database server, where the database server itself exists within Docker, while the data files are stored on a volume. In another use case, a volume might be used to allow a code development container to access source files stored, and edited on, the host device file system.

You can create a bind mount or volume using the the `--mount` option:

```
# Mounts the host /shared/data directory to /data within the container.
$ docker run --mount type=bind,src=/shared/data,dst=/data <my_image>

# Attach a volume called 'my-data-volume' to /data within the container.
# If the volume already exists it will be re-used, otherwise docker will
create a new volume.
$ docker run --mount type=volume,src=my-data-volume,dst=/data <my_image>
```

You may also see volumes added using the `--volume` or `-v` options. This mounts a host directory to a container in a similar fashion to `--mount type=bind`. In general, `--mount` is preferred. You can read more about the differences in the docker documentation.

Networks and Port Binding

Docker containers usually need to communicate with each other, and also send and receive network requests over the wider network via the host machine. Networking docker containers is complex, and for all but the most simple use-cases you'll rely on container orchestration tools to help manage this at a higher level. We will introduce the basic docker networking principles here, and come back to orchestration tools in a later module.

Docker manages its own virtual network, and assigns each container a network interface and IP address within that network. You can configure how, and even if, docker creates this network. For further details on docker's low-level networking architecture please refer to the official documentation.

By default, docker creates a *bridge* network where all containers have unrestricted outbound network access, can communicate with other containers on the same docker host via IP address, and do *not* accept any requests from the external (i.e. host) network and wider internet.

To receive inbound network requests, docker needs to forward traffic on a host machine port to a virtual port on the container. This **port binding** is configured using the `-p` (publish) flag during `docker run`. The port binding example below specifies that incoming traffic to port 80 on the host machine (http traffic) should be redirected to port 5000 inside the container, where a web server may be listening.

```
# Map host machine port 80 to port 5000 inside the container.
$ docker run -p 80:5000 <my_image>
```

The `docker run` command can be used to publish multiple ports, limit port bindings to specific protocols and bind port ranges.

Port binding should not be confused with the `EXPOSE` directive found in Dockerfiles (more in that in a moment). `EXPOSE` directives do not actually publish a port. Rather, they form part of an image's documentation. Control over port publishing and binding resides solely with the `docker run` command.

The –P flag

Occasionally, a container will expose many ports and it won't be feasible, or desirable, to manually map them all using the publish option (–p) of `docker run`. Instead, you would like to automatically map every port documented in the Dockerfile.

You can use the –P flag on `docker run` to publish all ports documented by `EXPOSE` directives in the Dockerfile. Docker will randomly bind all documented container ports to high-order host ports. This is of limited use, as the user has no control over the host ports bound. It is generally preferable to specify port bindings explicitly using multiple –p flags.

Remember, –p and –P have very different behaviour.

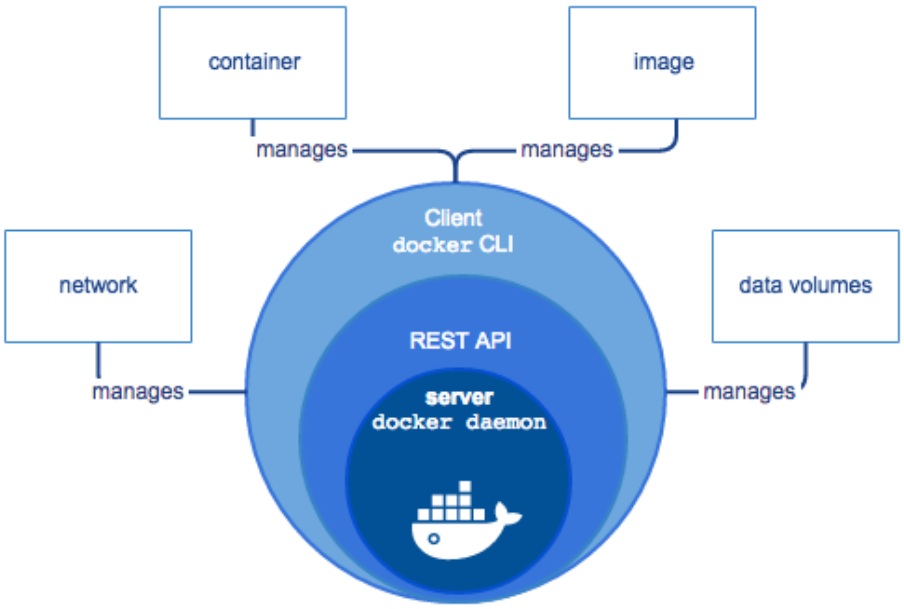
The Docker Engine

The Docker Engine provides most of the platform's key functionality and consists of several main components that form a client-server application:

1. A long-running program called a daemon process that acts as the server (the `dockerd` command).
2. A REST API specifying the interface that programs can use to talk to the daemon and instruct it what to do.
3. A command line interface (CLI) client (the `docker` command) that allows you to interact with the daemon via the REST API.

The daemon is responsible for creating and managing Docker objects, such as images, containers, networks and volumes, while the CLI client (and other clients) interact with the daemon via the REST API. The server-client separation is important, as it allows the docker daemon and docker CLI to run on separate machines if needed.

You can read more about Docker's underlying architecture here.



Docker Engine

Comparison with Virtual Machines

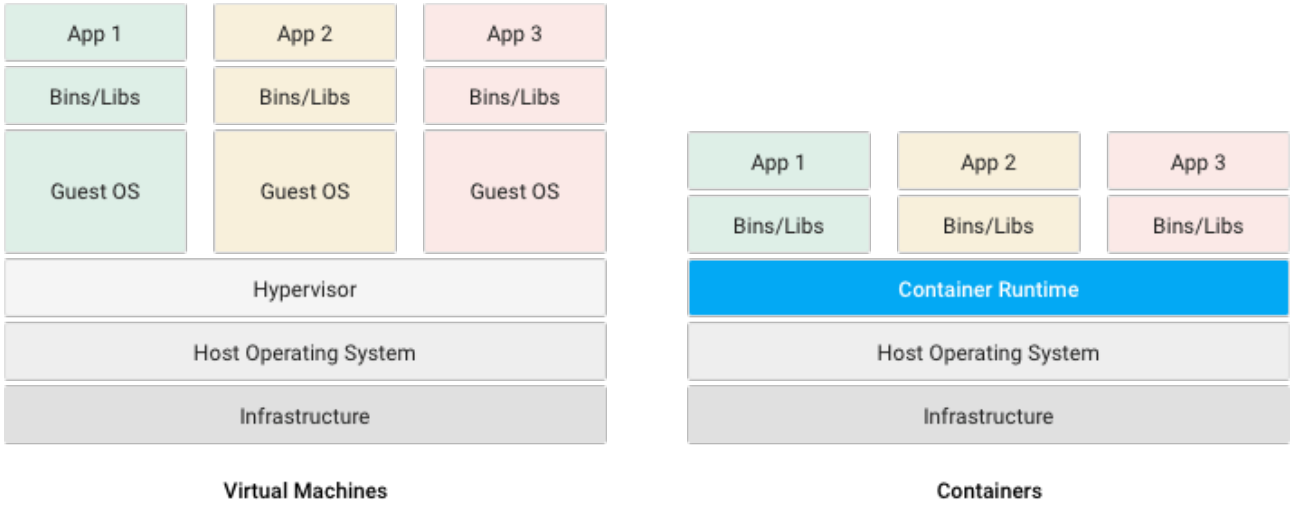
While a container might look superficially like a virtual machine (such as a dedicated Linux VM running on a Windows laptop), it works quite differently. The reason that containers are so lightweight in terms of disk and memory footprint is that they only contain the application binaries and dependent libraries needed to run the given application, rather than the full operating system contained in a VM. All containers share a common operating system kernel, the central component responsible for running or executing programs. The kernel takes responsibility for deciding which of the many running programs should be allocated to the processor at any given time, in addition to managing memory, devices and other resources.

By default, Docker uses a Linux kernel as the underlying OS that all containers share. For this reason, the vast majority of standard Docker images are Linux-based. Docker makes use of several key features of the kernel to create isolated environments for each container to run in, but each container is really just a set of isolated processes and file systems interacting with the common kernel. This is called OS-level virtualisation and is fundamentally different from the full virtualisation used to emulate a complete computer system (e.g. VirtualBox or VMware).

These concepts should be familiar from material covered in the previous module.

Because containers do not need to run their own operating system, they can be much smaller and are able to start up within milliseconds, compared to the seconds (or even minutes) it can take for VMs to start up.

The following image compares the main components/layers involved in running virtual machines versus containers.



VM vs Container Architecture

Chapter 4

Using Docker

Building Custom Images with Dockerfiles

While there is a healthy ecosystem of published Docker images you can use to run standard installations of many applications, what happens when you want to run your own application in a container? Fortunately, it's pretty straightforward to build your own custom images by creating a **Dockerfile**.

A Dockerfile is a recipe containing instructions on how to build an image by copying files, running commands and adjusting configuration settings. These instructions are applied on top of an existing image (called the base image). An image can then be built from the Dockerfile using the `docker build` command. The result is a new Docker image that can be run locally or saved to a repository, such as **Docker Hub** (see below).

When building an image, Docker effectively runs each of the instructions listed in the Dockerfile within a container created from the base image. Each instruction that changes the file system creates a new layer and the final image is the combination of each of these layers.

Typical tasks you might perform in a Dockerfile include:

1. Installing or updating software packages and dependencies (using `apt-get`, `pip` or other package manager)
2. Copying project code files to the image file system
3. Downloading other software or files (using `curl` or `wget`)
4. Setting file permissions
5. Setting build-time environment variables
6. Defining an application launch command or script

Dockerfiles are typically stored alongside application code in source control and simply named `Dockerfile` in the repository root. If you have multiple Dockerfiles, it's conventional to give them meaningful file extensions (e.g. `Dockerfile.development` and `Dockerfile.production`).

Here's a very simple Dockerfile, we'll cover the instructions in detail in the next section.

```
Dockerfile FROM alpine ENTRYPOINT ["echo"] CMD ["Hello World"]
```

Dockerfile instructions

Here are some of the most useful instructions for use in Dockerfiles:

1. **FROM** defines the base image used to start the build process. This is always the first instruction.
2. **RUN** executes a shell command in the container.
3. **COPY** copies the files from a source on the host into the container's own file system at the specified destination.
4. **WORKDIR** sets the path where subsequent commands are to be executed.
5. **ENTRYPOINT** sets a default application to be started every time a container is created from the image. Can be overridden at runtime.
6. **CMD** can be used to provide default command arguments to run when starting a container. Can be overridden at runtime.
7. **ENV** sets environment variables within the context of the container.
8. **USER** sets the UID (or username) used to run the container.
9. **VOLUME** is used to enable access from the container to a specified directory on the host machine. This is similar to using the `-v` option of `docker run`.
10. **EXPOSE** documents a network port that the application listens on. This can be used, or ignored by, those using the image.
11. **LABEL** allows you to add a custom label to your image (note these are different from tags).

For further details about Dockerfile syntax and other available instructions, it's worth reading through the full Dockerfile reference.

Take care when storing secrets

Just as with your source code, you should take care not to include secrets (API keys, passwords, etc.) in your Docker images, since they will then be easily retrievable for anyone who pulls that image.

This means that you should not use the ENV instruction to set environment variables for sensitive properties in your Dockerfile. Instead, these values should be set at runtime when creating the container, e.g., by using the `--env` or `--env-file` options for docker run.

`ENTRYPOINT` versus `CMD`

The `ENTRYPOINT` and `CMD` instructions can sometimes seem a bit confusing. Essentially, they are intended to specify the default command or application to run when a container is created from that image, and the default command line arguments to pass to that application, which can then be overridden on a per-container basis.

Specifying command line arguments when running `docker run <image_name>` will append them to the end of the command declared by `ENTRYPOINT`, and will override all arguments specified using `CMD`. This allows arguments to be passed to the entry point, i.e., `docker run <image_name> -d` will pass the `-d` argument to the entry point. You can override the `ENTRYPOINT` instruction using the `docker run --entrypoint` flag.

You can read a more detailed discussion of how and when to use `ENTRYPOINT` and `CMD` in this article.

Docker build

The `docker build` command instructs the docker daemon to build a new image and add it to the local repository. The image can subsequently be used to create containers with `docker run`, or pushed to a remote repository. To build an image, `docker build` requires two things:

1. **A Dockerfile** - By default, `docker build` will look for a file named "Dockerfile" in the current working directory. If the file is located elsewhere, or has a different name, you must specify the file path with the `-f` option.
2. **A Build Context** - Dockerfile COPY and ADD instructions typically move files from the host filesystem to the container filesystem. However, it's not actually that simple. Remember, the docker CLI and docker daemon are loosely coupled, and may be running on different host machines. The daemon (which executes the build command) has no access to the host filesystem. Instead, the CLI must send a collection of files (the "Build Context") to the daemon before the build can begin. These files may be sourced locally, or pulled from a URL. All files referenced in COPY and ADD instructions must be included in this context.

The build context is often a source of confusion to those new to Docker. A minimal `docker build` command is shown below,

and would typically be run from the root of a git repository:

```
$ docker build .
```

The command above requires an appropriately named Dockerfile file in the current directory. The build context is the one and only argument to docker build (`.`). This means that the whole of the current directory, and any subdirectories, will be sent to the docker daemon before the build process begins. After this transfer, the rest of the build process is performed as defined in the Dockerfile.

`docker build` can also be used to tag the built image in the standard `name:tag` format. This is done using the `--tag` or `-t` option.

Version your images

If you rely on just using the `latest` tag, you'll have no way of knowing which image version is actually running in a given container. Instead, use a tag that uniquely identifies when/what was built. There are many versioning strategies you could consider, but some possible unique identifiers include:

- timestamps
- incrementing build numbers
- Git commit hashes

For example, you could use both the commit SHA1 hash (to associate the image to a specific commit to help with debugging) and the environment name: `/$PROJECT/$ENVIRONMENT:$SHA1`.

```
$ docker build --tag web/prod:a072c4 .
```

You can add an unlimited number of the tags to the same image so you can be very flexible with your tagging approach.

Example: A Docker 'Hello World'

Let's see how all this works in practice, by creating a simple Docker image for a container that echoes "Hello World!" to the console, and then exits. To run through this, you'll need Docker installed locally.

Since all standard Linux images will already have the `echo` command available, we'll use one of the most common base images: `alpine`. This is based on the minimal Alpine Linux distribution and is designed to have a small footprint and to serve as a reasonable starting point for creating many other images.

Start by creating a new Dockerfile file (in whichever directory you prefer) and specifying `alpine` as the base image:

```
FROM alpine
```

Since we haven't specified a version (only the name of the base image), the `latest` tagged version will be used.

We need to instruct the container to run a command when it starts. This is called the entry point for the container and is declared in the Dockerfile using the `ENTRYPOINT` instruction. In our case, we want the container to use the echo command to print "Hello World" to stdout (which Docker will then display in our terminal):

```
ENTRYPOINT ["echo", "Hello World"]
```

Now we're ready to build an image from our Dockerfile. Start a terminal from the same directory as your Dockerfile and run the build command:

```
$ docker build --tag hello-world .
```

Check that your image is now available by listing all images in your local repository:

```
$ docker image ls
```

Finally, create a container from your image to see if it works:

```
$ docker run hello-world
```

Hopefully, you should see "Hello World" printed to your terminal!

If you wanted to make the message printed by the container customisable (such that running `docker run hello-world Greetings!` would print "Greetings!" instead). How would you modify your Dockerfile to achieve that? (Hint: `ENTRYPOINT` and `CMD` can help.)

Docker Hub and other repositories

Docker Hub is a cloud-based repository run and managed by Docker Inc. It's an online repository where Docker images can be published and downloaded by other users. There are both public and private repositories; you can register as an individual or organisation and have private repositories for use within your own organisation, or choose to make images public so that they can be used by anyone.

Docker Hub is the primary source for the curated official images that are used as base images for the vast majority of other custom images. There are also thousands of images published on Docker Hub for you to use, providing a vast resource of ready-made images for lots of different uses (web servers, database servers, etc.).

Docker comes installed with Docker Hub as its default registry, so when you tell Docker to run a container from an image that isn't available locally on your machine, it will look for it instead on Docker Hub and download from there if it's available. However, it's possible to configure Docker to use other registries to store and retrieve images, and a number of cloud- and privately-hosted image registry platforms exist, such as GitLab Container Registry, JFrog Artifactory and Red Hat Quay.

Regardless of whether you are using Docker Hub or an alternative image registry, the Docker commands to fetch and save images are the same. Downloading an image is referred to as pulling it, and for Docker Hub you do not need an account to be able to pull public images (similarly to GitHub). Uploading an image is known as pushing it. You generally need to authenticate with an account on the registry to be able to push images.

The `docker login` command is used to configure Docker to be able to authenticate against a registry. When run, it prompts you for the username and password you use to log in to the registry, and stores your login credentials so that Docker can access your account in the future when pulling or pushing images.

Docker containers as immutable infrastructure

How do docker containers fit in with the goals and requirements of immutable infrastructure? Let's look at the main aspects of immutable infrastructure and see how containers help:

Reproducible configuration	Containers created from the same image are guaranteed to be identical.
Rapid provisioning	New containers can be created in milliseconds.
Automated deployment	Images can be built from a Dockerfile and their deployment as new containers can be automated.
Stateless applications	Containers are intended to be ephemeral, with all app data saved to a mounted volume that can be shared between containers. (It's possible to have apps save data to their container file system instead, this goes against recommended best practice.)
Immutable versioning	Docker images are immutable and can be versioned using tags.

Containers achieve the immutable infrastructure requirements for the application servers that constitute your infrastructure (web servers, database servers, etc.). However, they constitute only one part of fully immutable infrastructure and other tools and platforms are required to fully realise that goal. Two key aspects that containers alone do not address are:

1. Automated provisioning of the underlying infrastructure (compute instances, load balancers, firewalls, network configuration, etc.).
2. Deploying containerised applications to that infrastructure (replacing old containers with new ones when new image versions are released, scaling container instances to handle load).

The first of those requirements can be achieved using automated configuration management and other infrastructure as code tools, which we'll cover later in this module. We'll discuss deployment platforms for containers — also known as container orchestration platforms — later in the course.

Chapter 5

Comparison with configuration tools

In this module we've introduced containers and Docker, presenting these technologies as ways to treat your application infrastructure and dependencies as code. We have advocated deploying stateless applications in immutable containers. It is useful at this stage to reflect on the comparison with the configuration management tools (e.g. Ansible, Chef) introduced in the previous module.

Automated configuration and deployment tools fill another gap in our immutable infrastructure requirements. By declaring the desired state of our infrastructure, these tools will automate its creation and are typically able to provision new systems much faster than could be done by hand. This allows us to automatically spin up a new server (or an entirely new environment) and ensure its consistency and reproducibility.

However, these general purpose automation tools do not enforce immutability. They're equally happy to make changes to existing infrastructure. This means that they must be used in specific ways — and with other tooling — to realise the full benefits of immutable infrastructure.

Finally, it's important to note that, in reality, mutability is a spectrum. Even when striving for fully immutable infrastructure, some applications in your system may not be designed to achieve that, particularly if they distribute state across many machines. This is especially true for legacy applications. These systems will require machines to be backed up and maintained, requiring us to continuously manipulate the state of these machines.

Unless you are designing a new infrastructure from scratch, making an existing system immutable can be a gradual and sometimes incomplete process, where compromises have to be made. Even in these cases, automation still has a key role to play. Automated configuration and deployment tools, such as Ansible and Chef can form the foundations upon which you can gradually transition to an immutable approach, based on containerised applications and fully immutable infrastructure.

Using Docker can reduce the overall workload for a DevOps team. Docker containers already encapsulate all the configuration and state for an individual service. This shifts the DevOps focus from low-level configuration to high-level architecture and interactions between different components. In simple terms: Docker handles individual server setup; this leaves the DevOps team to manage which versions of those containers to use and how to connect them.

Additional Material

Docker Best Practices

We've covered the basics of building your own Docker images and had a go at creating a simple example. When creating images for real-world projects, however, there are other considerations you need to keep in mind. For instance:

- Reducing build times
- Improving scalability by reducing image size
- Ensuring the security of running containers

In this section we'll cover some of the best practices that can help to improve the performance and security of your builds.

Choice of base image

Use images based on Alpine Linux since they come with only the packages you need. Resulting images will be smaller.

What are the benefits?

- Decreased hosting costs since less disk space is used
- Quicker build, download, and run times
- More secure (since there are fewer packages and libraries)
- Faster deployments

For example, the official Python image has a number of variants, including one based on Alpine.

Multi-stage builds

Take advantage of multi-stage builds to create a temporary image used for building artefacts that can then be copied over to the final production image. The temporary build image is discarded along with the original files, folders, and dependencies associated with it.

You can read more here.

Ignore files that should be excluded

Add a .dockerignore file to your project to exclude files that aren't needed when copying source code or artefacts to an image. This helps to reduce the size of your image and ensures that sensitive or unwanted files are not copied across by accident. It also speeds up the build as excluded files are not sent to the daemon as part of the build context.

The file syntax is the same as a .gitignore file, with any file or path listed in it being automatically excluded by the COPY or ADD instructions when copying files from the host machine to the container.

Take advantage of caching

Each instruction in a Dockerfile will create a new image layer that can later be re-used. Docker caches the steps in a Dockerfile to speed up subsequent builds. When a change is made to a step, all steps following it will be redone.

Avoid invalidating the cache by:

- Starting your Dockerfile with commands that are less likely to change
- Putting commands that are more likely to change (like `COPY .`) as late as possible
- Adding only necessary files (using a `.dockerignore` file to exclude project files that aren't needed by the container)

Consider the following example:

```
FROM python:3.8-alpine

WORKDIR /app

COPY sample.py . # What happens when a change is made to sample.py?

COPY requirements.txt .
RUN pip install -r /requirements.txt
```

The code in `sample.py` is likely to change much more often than the packages listed in requirements.txt, so moving the `COPY sample.py .` statement to the bottom will allow the package installation step to be cached and re-used.

Combine RUN instructions to minimise the number of layers

Docker will create a new layer for every RUN instruction in your Dockerfile, which increases the overall size of the image. To minimise this, combine RUN steps that are related. For example:

```
RUN apt-get update
RUN pip install --upgrade pip
```

will create two layers. However, both steps are effectively just updating package managers, so can be combined into one:

```
RUN apt-get update && pip install --upgrade pip
```

Things to note:

- RUN, COPY, and ADD steps will create new layers.
- Each layer contains the differences from the previous layer.
- Layers increase the size of the final image.

Note that careful use of layers that are common across multiple images can save overall disk space (since the same layer can be shared for all those images), however, this only applies when you are creating containers from different images on the same machine, which is less likely to be the case in production. The layers created by steps in your Dockerfile are also likely to be specific to that image only (for example, Python packages for a specific app), so minimising the number of layers in your builds is more likely to reduce disk space in general.

Some additional tips:

1. Put related commands (`apt-get update && apt-get install`) in the same RUN step.
2. Remove intermediate files in the same RUN step that created them.
3. Avoid using `apt-get upgrade` since it upgrades all packages to the latest version, including those from the base image. Updates to base image packages should come from a new version of that image, not a later upgrade (remember, images are immutable).

Further reading on best practices

The above advice covers the most important aspects to consider when creating images. If you want to do some further reading on the subject, Docker provide a much more in-depth guide to Dockerfile best practices, which you can find here (docs.docker.com/develop/develop-images/dockerfile_best-practices).

Troubleshooting Docker

When things go wrong, either with building images or running containers, there are a few commands and techniques that can help you diagnose what's causing the issue. As is generally the case with most troubleshooting, the two key approaches are:

- Examine log files for relevant error messages.
- Run steps interactively to explore the context in which they are failing.

The following tips cover some tools that can prove particularly useful.

Build failures

When you build an image, each instruction that is executed from the Dockerfile is printed to the terminal as "Step 1", etc. If the build fails, you should be able to identify which instruction failed and what the error message for that failing step was. What's more, Docker

creates intermediate images at each step of the build, which you can use to "jump into" a container just prior to the failing instruction and try out that instruction within an interactive shell, to get a better idea of what the error might be. This article gives a nice overview of how to do that.

Docker caches each step of the build process, which can also lead to some confusing build failures. You may run into a situation where you run an update in the build, Docker caches this update, and some time later your base distribution updates its sources again, leaving you with outdated sources, despite doing a package cleanup and update in your Dockerfile. If you run into issues installing or updating packages inside the container, you can try running apt-get clean && apt-get update inside of the running container to refresh stale package list caches. You can also force Docker to rebuild an image without using the existing cache by including the `--no-cache` option when running docker build.

Syntax errors and caching problems are the most common issues you are likely to encounter when building an image in Docker. As a general troubleshooting tip, pay close attention to the Docker build output to identify where typos could be causing commands to fail.

A few other things you can try when diagnosing issues with custom images:

- Add `RUN` instructions to call echo and other commands (ls, pwd, etc.) to help diagnose issues in your Dockerfile
- Run an interactive shell on your built container to inspect its filesystem and see if things were installed correctly
    - Most base images should come with bash installed, so you can run
      `docker run -it <image_name> /bin/bash`
      to open an interactive shell on your newly built image.
    - You will need to use the `--entrypoint` option if you have specified an `ENTRYPOINT` instruction in your Dockerfile, e.g.
      `docker run -it --entrypoint /bin/bash <image_name>`.
    - Notably, the `alpine` image does not come with bash installed!
      You can use `/bin/sh` instead though.
    - Use the `docker history` command to view details of the individual layers that make up an image, along with the instructions that created them.

Logging

The first port of call when trying to troubleshoot issues with a running container is to look at the logs.

Many programs log information to stdout, especially when things go wrong. Anything that gets written to stdout and stderr from the primary process (pid 1) running inside a container will get captured to a history file on the host machine, where it can be viewed with the `docker logs` command:

```
$ docker logs <container_name>
```

This history is available even after the container exits, as long as the container hasn't been removed with docker rm. The data is stored in a JSON file buried under `/var/lib/docker`. The log command takes options that allow you to follow this file (`--follow`), which will stream its contents as more data is written to it, as well as choose how many lines the command returns (`--tail 10`). By default the command returns all lines.

Get process stats

The `docker top` command is equivalent to the normal top Linux command and displays the current running processes within a container:

```
$ docker top <container_name>
```

There is also a `docker stats` command that displays a live stream of resource usage statistics (CPU, memory usage, etc.) for one or more containers running on your machine.

View container details with the `inspect` command

The docker inspect command returns information about a container or an image. Here's an example of running it on a hello-world container from a previous exercise.

```
$ docker inspect <container_name>
[
  {
    "Id":
"fdb3008e70892e14d183f8 ...
020cc34fec9703c821",
    "Created":
"2020-03-23T16:27:30.6958079Z",
    "Path": "/bin/sh",
    "Args": [
      "-c",
      "echo 'Hello World'"
    ],
    "State": {
    ... and lots, lots more.
  }
]
```

The bulk of the output has been skipped in the above example because there's a lot of it. Some of the more valuable bits of information included are:

- Current state of the container (in the "State" property).
- Path to the log history file (in the "LogPath" field).
- Values of set environment variables (in the "Config.Env" field).
- Mapped ports (in the "NetworkSettings.Ports" field).

One of the most valuable uses of inspect is to list the environment variables on a container, which can often be a source of hard-to-diagnose problems. Again, this command can be run on a container that has already exited, so you can use it to check the details of containers that are crashing/stopping unexpectedly.

Issues with Docker itself

There are times when Docker itself can misbehave or get into a bad state. If Docker commands become unresponsive, or containers fail to start (even ones from official images), then the first thing to try is to restart Docker Desktop via the "Restart" option in the Docker toolbar menu.

If restarting Docker Desktop and restarting your machine both fail to fix the problem, it's probably worth reading through the official troubleshooting guide for other things to try. As always, googling for any error messages you see might also turn up suggested fixes, too.

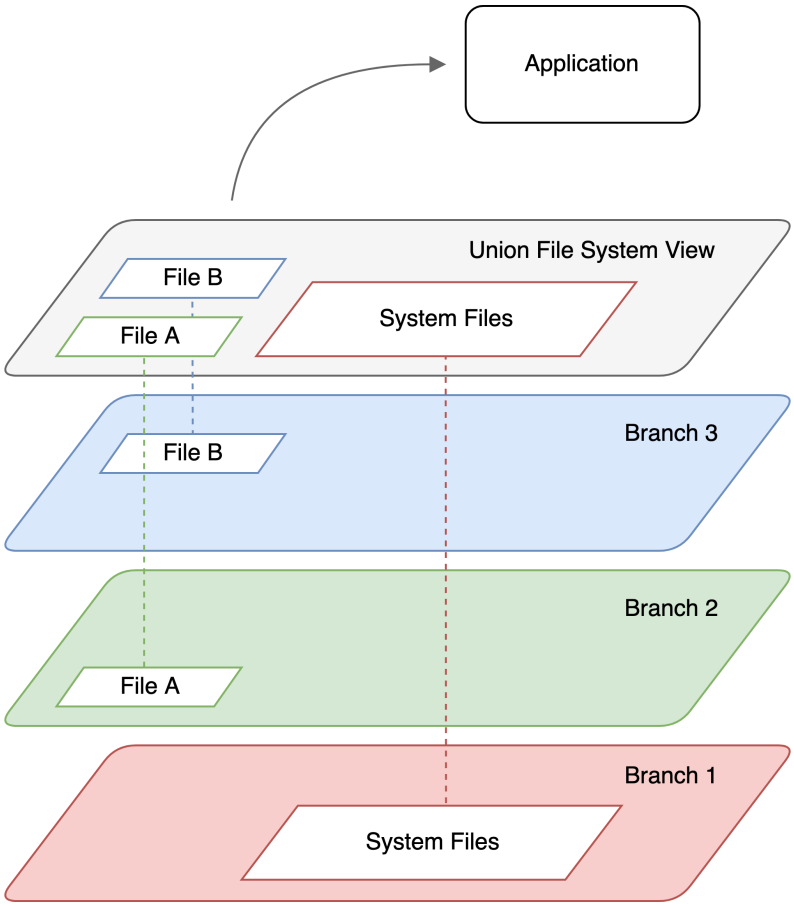
Union file systems

We've discussed how docker builds images out of layers. This approach has many advantages over a monolithic image architecture. It allows docker to re-use layers between images, saving on disk and network usage, as well as optimising the build process to only re-build layers that have changed. But how does it work?

Docker relies on a **union file system**. A union file system combines multiple file systems (**branches**) into one by overlaying them on top of each other. A client application can then read from the union file system as though it were a single file system, and the union file system will look through the stack of branches, in order, to file a matching entry.

Union file systems are read-only. Writing data to the file system creates a copy of that file in a new branch on top of the stack. This technic is called "copy-on-wite". Further file writes are added to this new branch. In the context of docker, this is the "container layer" mentioned previously.

You can think of each layer in a docker image as a branch in the union file system. This is the approach of **AUFS** - the previous default docker storage driver. In order to read a file, AUFS search through the branches (or layers) from top to bottom until it found the file. This is shown in the diagram below:



Union File System

OverlayFS is now the default storage driver for docker. It is similar to AUFS, but simpler and faster. Instead of storing all branches individually, each image layer combines all previous file system branches into a single branch (known as `lowerdir`). A second branch, `upperdir`, contains this layer's changes. This means that when a file is searched for, there are only two branches to look at, both inside the top layer of an image. This helps keep disk performance high, without compromising the layered structure of the union file system.