What is Containerization?

Containers are a revolutionary approach to packaging applications. They bundle your application code together with all its dependencies—libraries, system tools, and runtime—into isolated, self-contained units. This ensures your application runs consistently, regardless of where it's deployed.

Unlike traditional virtual machines, containers are remarkably lightweight and portable. They share the host system's operating system kernel, which makes them faster to start and more efficient with resources. This makes containers ideal for modern development practices like continuous integration and continuous deployment (CI/CD) workflows, as well as microservices architectures.

> **Analogy:** Think of a container as a self-contained workspace—like a shipping container that holds everything needed for your journey. It works the same way everywhere, whether on your laptop, in testing, or in production.

Why Docker?

Industry Standard

Docker has become the de facto standard tool for containerisation, trusted by millions of developers worldwide.

Works Everywhere

Run your applications consistently across all environments—development, testing, staging, and production.

Eliminates Inconsistencies

Say goodbye to "works on my machine" problems. Docker ensures environmental parity across all systems.

Getting Started is Simple

With Docker, you can run a fully functional web server in seconds. The command below launches an nginx server instantly:

```
docker run nginx
```

This simplicity extends to complex applications. Docker abstracts away infrastructure concerns, letting you focus on building great software.

Docker Architecture

Understanding Docker's architecture is essential for effective containerisation. Docker follows a client-server model with several key components working together seamlessly.

Docker Client

The command-line interface (CLI) you use to interact with Docker. When you type docker commands, the client communicates with the Docker daemon to execute your requests.

Docker Daemon

The background service responsible for building, running, and managing containers. It listens for Docker API requests and handles the heavy lifting of container operations.

Docker Hub

A cloud-based registry service where you can find, share, and store Docker images. Think of it as GitHub for container images, with thousands of pre-built images available.

Images & Containers

Images are read-only templates containing your application and its dependencies—like blueprints. Containers are running instances of images—the actual executing applications.

The Docker Lifecycle

The Docker workflow follows a straightforward, repeatable process. Understanding this lifecycle is crucial for integrating Docker into your development pipeline effectively.

Write Dockerfile

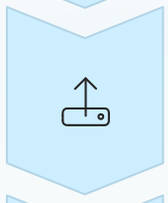
Create a text file defining your application's environment, dependencies, and configuration.

Build Image

Execute `docker build` to transform your Dockerfile into a reusable image.

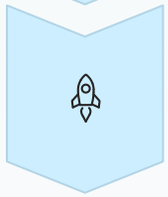
Run Container

Launch your application with `docker run`, creating a running container from the image.



Push to Registry

Share your image using `docker push`, making it available for deployment anywhere.



Deploy

Pull and run your containerised application on any Docker-enabled system, from cloud to on-premises.

This cycle promotes consistency and reproducibility. Once you've built an image, it can be deployed identically across all environments, eliminating configuration drift and deployment surprises.

Hands-On Goal

Theory is important, but nothing beats practical experience. In this workshop, we'll walk through the complete Docker workflow using a real-world example.

01

Build a Docker Image

We'll create a Docker image for a simple Node.js application, learning how to define the environment and dependencies.

02

Run Locally

Test the containerised application on your local machine, verifying it works as expected in its isolated environment.

03

Push to Docker Hub

Upload your image to Docker Hub, making it accessible for deployment from anywhere.

04

Deploy

Pull and run your container in a deployment environment, experiencing the true portability of Docker containers.

By the end of this hands-on exercise, you'll have experienced the complete containerisation workflow and gained confidence in using Docker for your own projects.

Writing a Dockerfile

Anatomy of a Dockerfile

A Dockerfile is a text document containing instructions for building a Docker image. Each instruction creates a layer in the final image, and understanding these instructions is key to creating efficient containers.

Let's examine a typical Dockerfile for a Node.js application. Each line serves a specific purpose in defining the container environment.

Example: Node.js Application

```
FROM node:18
WORKDIR /app
COPY package*.json .
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

1 FROM node:18

Specifies the base image. Here we're using the official Node.js 18 image as our foundation.

2 WORKDIR /app

Sets the working directory inside the container where subsequent commands will execute.

3 COPY & RUN

Copies package files first, installs dependencies, then copies the rest of the application code. This layering optimises build cache.

4 EXPOSE & CMD

EXPOSE documents which port the container listens on. CMD specifies the command to run when the container starts.

Building & Running Containers

With your Dockerfile written, you're ready to build and run your first container. These commands form the core of day-to-day Docker usage.

Build the Image

```
docker build -t my-node-app .
```

The `-t` flag tags your image with a name ("my-node-app"). The `.` tells Docker to look for the Dockerfile in the current directory. Docker will execute each instruction in your Dockerfile, creating layers and caching them for faster subsequent builds.

Run the Container

```
docker run -p 3000:3000 my-node-app
```

This command creates and starts a container from your image. The `-p 3000:3000` flag maps port 3000 on your host machine to port 3000 in the container, making your application accessible.

Verify It's Working

Open your web browser and navigate to **localhost:3000**. You should see your Node.js application running. Congratulations—you've just containerised and run your first application!

These three steps represent the fundamental Docker workflow you'll use repeatedly. As you become more comfortable, you'll add flags and options to customise behaviour, but this core pattern remains constant.

Docker Commands Cheatsheet

Mastering a handful of essential Docker commands will cover most of your daily needs. Here's a reference guide to the most important commands you'll use regularly.

Command	Description
docker ps	Lists all currently running containers. Add -a to see stopped containers as well.
`docker images`	Shows all Docker images stored locally on your system, including their sizes and creation dates.
docker exec -it <id> bash	Opens an interactive terminal session inside a running container, useful for debugging and exploration.
`docker logs <id>`	Displays the logs from a container, essential for troubleshooting and monitoring application output.
`docker stop <id>`	Gracefully stops a running container by sending a SIGTERM signal, allowing cleanup operations.
`docker rm <id>`	Removes a stopped container from your system. Use `docker rm -f` to force remove running containers.
`docker rmi <image>`	Deletes an image from local storage, freeing up disk space. Cannot remove images being used by containers.
`docker pull <image>`	Downloads an image from Docker Hub or another registry without running it immediately.

Pro Tip: You can use the first few characters of a container ID instead of the full ID in commands. Docker is smart enough to match partial IDs uniquely.

Docker Best Practices

Following best practices ensures your Docker containers are secure, efficient, and maintainable. These guidelines will help you avoid common pitfalls and create production-ready containers.

Use .dockerignore Files

Create a `.dockerignore` file to exclude unnecessary files from your build context. This speeds up builds and reduces image size by preventing things like `node_modules`, `.git`, and log files from being copied into images.

Keep Images Small

Use Alpine-based images when possible—they're minimal Linux distributions that dramatically reduce image size. For example, `node:18-alpine` is much smaller than `node:18` whilst providing the same functionality.

Never Hardcode Secrets

Don't embed passwords, API keys, or other sensitive data in Dockerfiles or images. Use environment variables, Docker secrets, or dedicated secret management tools instead. Remember: images can be inspected by anyone with access.

Pin Image Versions

Always specify exact image versions (e.g., `node:18.16.0`) rather than using `latest` tags. This ensures reproducible builds and prevents unexpected breaking changes when base images update.

Multi-Stage Builds

Use multi-stage builds to separate build-time dependencies from runtime dependencies. This technique can dramatically reduce final image size by including only what's needed to run the application, not build it.

Minimize Layers

Combine related commands in a single `RUN` instruction to reduce the number of layers. Fewer layers mean smaller images and faster builds. Use `&&` to chain commands together.

Introduction to Orchestration

Containers are brilliant for packaging and running individual applications, but what happens when you need to manage dozens or hundreds of containers across multiple servers? This is where orchestration comes in.

The Scaling Challenge

Managing containers manually becomes chaotic at scale. You need automation for deployment, scaling, load balancing, health monitoring, and recovery. Doing this by hand is simply not feasible.

Enter Kubernetes

Kubernetes (often abbreviated as K8s) is the leading container orchestration platform. It automates deployment, scaling, and management of containerised applications across clusters of machines.

Key Kubernetes Concepts

Pods

The smallest deployable unit in Kubernetes. A pod contains one or more containers that share storage and network resources.

Deployments

Define the desired state for your pods, including how many replicas to run. Kubernetes ensures this state is maintained automatically.

Services

Provide stable networking endpoints for accessing your pods, handling load balancing and service discovery automatically.

Kubernetes builds upon Docker's containerisation model, adding the orchestration layer needed for production-scale deployments. As you grow comfortable with Docker, Kubernetes becomes the natural next step in your containerisation journey.

Running Multiple Containers

In real-world applications, you'll often need to run multiple services that work together. Docker makes it straightforward to manage these interdependent containers, allowing each service to operate in its own isolated environment.

Nginx (Web Server)

```
docker run -d -p 8080:80 nginx
```

This command starts an Nginx web server container in detached mode (-d) and maps port 8080 on your host machine to port 80 inside the container. This Nginx instance can serve your frontend application.

MongoDB (Database)

```
docker run -d -p 27017:27017 mongo
```

Similarly, this command launches a MongoDB database container, running it in the background and exposing its default port 27017 both internally and externally. Your backend services can connect to this database.

These two containers are now running as isolated services. The Nginx container is ready to serve your web content, while the MongoDB container provides a data store. They can communicate seamlessly within Docker's internal networking, enabling you to build complex, multi-service applications.

Docker Networks

```
docker network create my-net
docker run -d --name app --network my-net my-node-app
docker run -d --name db --network my-net mongo
```

Why: Containers in the same network can talk by name.
Example: `mongodb://db:27017`

Introducing Docker Compose

Docker Compose is a powerful tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services, networks, and volumes, then launch everything with a single command.

It streamlines the development lifecycle, making it easy to set up, manage, and tear down complex application environments locally or in staging.

Defining Your Services with `docker-compose.yml`

```
version: "3.8" # Version of the Compose file format
services:
  web: # Service name for your web application
    build: . # Build the image from the Dockerfile in the current directory
    ports:
      - "3000:3000" # Map host port 3000 to container port 3000
  db: # Service name for your database
    image: mongo # Use the official MongoDB image from Docker Hub
    # Add optional environment variables, volumes, networks, etc.
```

The `docker-compose.yml` file is the heart of your Compose setup. It declaratively outlines each service required by your application, including their images, build instructions, port mappings, volumes, and network configurations.

Launching Your Application Stack

```
docker compose up -d
```

Once your `docker-compose.yml` is configured, a single command brings your entire application stack to life. The `-d` flag runs the services in "detached" mode, meaning they run in the background, freeing up your terminal.

Simplified Setup

Define an entire multi-service application in a single YAML file, eliminating lengthy command-line setups for each container.

Consistent Environments

Ensure that all developers and environments use the exact same configuration, reducing "it works on my machine" issues.

Efficient Development

Quickly start, stop, and rebuild your services, accelerating local development and testing workflows.

Docker Volumes

Containers are inherently ephemeral; any data stored within them is typically lost when the container is deleted or recreated. Docker Volumes provide a robust mechanism to persist data generated by Docker containers, ensuring your valuable information remains safe and accessible, independent of the container's lifecycle.

Create a Named Volume

```
docker volume create app-data
```

This command creates a named volume called `app-data`. Named volumes are managed directly by Docker and are the preferred way to persist data for services. They are isolated from the host filesystem's directory structure, making backups and migrations easier.

Run Container with Volume

```
docker run -d -v app-data:/data/db mongo
```

Here, we launch a MongoDB container in detached mode, mounting the `app-data` volume to the `/data/db` directory inside the container. All data written to this internal path will now be stored in the persistent volume, ensuring your database content remains intact even if the MongoDB container is removed or updated.

Environment Variables

Environment variables are a powerful way to configure your Docker containers dynamically. They allow you to pass configuration details, such as API keys or database connection strings, into your running applications without having to rebuild the container image.

Here's how to set an environment variable when running a container:

```
docker run -e NODE_ENV=production -p 3000:3000 my-node-app
```

This command launches a `my-node-app` container, setting the `NODE_ENV` environment variable to `production`. The application inside the container can then read this variable to adjust its behaviour accordingly, for example, by enabling different logging levels or connecting to a production database.

> **Tip:** For sensitive data like API keys or database credentials, never hardcode secrets directly in your commands or Dockerfiles. Instead, use `.env` files with `--env-file`, or Docker Secrets/Config management for enhanced security and best practice.

Kubernetes Basics

As you scale beyond individual containers and Docker Compose, Kubernetes steps in as the industry-standard platform for automating deployment, scaling, and management of containerised applications. Understanding its core components is key to distributed systems.

Pods

The smallest deployable unit in Kubernetes, a Pod encapsulates one or more containers, sharing storage, network resources, and a unique IP address. They are the fundamental building blocks for running workloads.

Deployments

A controller that manages a replicated set of Pods, ensuring that a specified number of Pod replicas are always running. Deployments provide declarative updates for Pods and ReplicaSets, enabling rolling updates and rollbacks.

Services

An abstract way to expose an application running on a set of Pods as a network service. Services define a logical set of Pods and a policy by which to access them, providing stable network access regardless of Pod changes.

Interacting with Kubernetes resources is primarily done via the `kubectl` command-line tool. Here are some essential commands:

View Running Pods

```
kubectl get pods
```

This command lists all Pods running in your currently selected Kubernetes namespace, providing details like status, restart count, and age.

Apply Configuration

```
kubectl apply -f deployment.yaml
```

This command applies a configuration defined in a YAML file (e.g., a Deployment or Service definition) to your Kubernetes cluster, creating or updating resources as specified.

Sample Kubernetes Deployment

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: node-app
  template:
    metadata:
      labels:
        app: node-app
    spec:
      containers:
      - name: node
        image: my-node-app
        ports:
        - containerPort: 3000
```

Exposing Deployment

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: node-service
spec:
  type: NodePort
  selector:
    app: node-app
  ports:
    - port: 80
      targetPort: 3000
      nodePort: 30080
```

Access: http://<minikube-ip>:30080

Verify Kubernetes Setup

```
kubectl get all
kubectl describe pod <pod-name>
kubectl logs <pod-name>
```

Goal: Debug running app in cluster.

Session Summary

Build & Run Docker Containers

We covered the fundamentals of containerization, including writing Dockerfiles, building images, and executing containers efficiently.

Docker Compose for Multi-Container Applications

We explored orchestrating multi-service applications using Docker Compose, simplifying setup and ensuring consistent development environments.

Deploy & Scale with Kubernetes

We introduced Kubernetes, understanding its core components like Pods, Deployments, and Services for robust, scalable container orchestration.

Next Day Preview: Automating Infrastructure with Terraform