Llenard C. Diama
IV-BCSAD

Docker and Containerization: Study Plan

Phase 1: Foundations—The "Why" and The History

Before you dive into commands, you need to understand the problems Docker was built to solve. This section covers the history of process isolation and how we got from giant Virtual Machines (VMs) to lightweight containers.

The Road to Isolation
The initial process toward containerization began when operating systems became isolated from one another. Key historical steps include:

chroot (Change Root): Added in 1979, this command changes a process's root directory, preventing it and its descendants from accessing files outside that directory—an easy way to create a "jail".

FreeBSD Jails: A more advanced system that virtualizes a process's filesystem, users, and network. Each jail has its own IP address and appears to the process like a separate machine.

Solaris Zones: Full OS-level virtualization that offers entirely isolated environments for applications.

VMs vs. Containers: The Core Difference
Virtual Machines were once used to solve the problem of running applications in various environments.

Feature	**VM's**	**Containers**
What They Virtualize	The entire computer system, including the hardware.	The operating system, not the hardware.
How they work?	A hypervisor (like VirtualBox) loads and runs multiple guest operating systems.	They are lightweight, isolated processes that share the host OS kernel.
Pros	Total isolation and can run different operating systems (e.g., Linux on Windows).	Extremely light and fast—boot in seconds. Use much less RAM and disk space.
Cons	Resource-intensive; each VM needs a full OS,	All containers on a host must share the same OS

	consuming gigs of disk space and considerable RAM. Slow to boot.	kernel (i.e., you can't run a Windows container on a Linux host).

Analogy: Think of a VM as an individual house with its own foundation. A container is like an apartment within a single building, isolated from others but sharing the core infrastructure.

The Docker Architecture
Docker uses a client-server architecture.

Docker Daemon (dockerd): The long-running background process that manages Docker objects like images, containers, networks, and volumes. It accepts API requests from the client.

Docker Client (docker): The command-line interface you use to send instructions to the daemon.

Docker Registry: An external place for storing and distributing images, with Docker Hub being the default public registry.

Phase 2: Core Skills—Building, Running, and Managing

This is where you go hands-on, learning how to build and manage your own containerized applications.

Writing a Dockerfile
A Dockerfile is just a text file containing the step-by-step instructions for building a Docker image.

Example Instructions (for a simple Node.js app):

Dockerfile

```
<-------------------------------------------------------->
# Use an official Node.js runtime as a parent image
FROM node:18-alpine
# Set the working directory in the container
WORKDIR /usr/src/app
# Copy the current directory contents into the container
COPY . .
# Run the app when the container launches
CMD [ "node", "app.js" ]
<-------------------------------------------------------->
```

Essential Container Lifecycle Commands
Once you build your image with docker build -t myapp:v1 . , here's how you manage the running container:

<-->
docker run myapp:v1: Creates and runs a container from the image.

docker ps: Lists only the running containers (add -a to see all, including stopped ones).

docker stop <container_id>: Gracefully stops a running container.

docker rm <container_id>: Removes a stopped container.

docker logs <container_id>: Views the output of a container.

docker exec -it <container_id> bash: Gets an interactive shell inside a running container for debugging.
<-->

Phase 3: The Ecosystem—Networking, Storage, and Compose

Applications rarely exist alone. This phase addresses how containers store data and communicate with each other.

Data Persistence: Volumes vs. Bind Mounts
A container's filesystem is ephemeral—data is lost when the container is removed. To persist data, you use:

- Volumes: The recommended approach for production. Volumes are managed by Docker and stored in a special, safe area of the host filesystem. They are platform-independent and less risky.

- Bind Mounts: These directly map a file or directory from the host system into the container. They're typically used in development to instantly apply source code changes to a running container.

Docker Networking

- Bridge Network: The default Docker network. It's an internal, private network that allows containers within it to talk to each other using their container names as hostnames.

- Port Publishing: To let the outside world access a service in your container (e.g., a web server), you must publish its port: docker run -p 8080:80 nginx exposes the container's port 80 to the host's port 8080.

Docker Compose

- Docker Compose is a command-line tool used to define and run multi-container applications, such as a web app, database, and cache. You define all your services, networks, and volumes in a single YAML file.

- You can start the entire stack with a single command: docker-compose up -d.

Phase 4: The Real World—Orchestration & Security

When you move your application to run reliably on multiple servers for high traffic and scale, you shift from managing one container to managing a fleet.

Container Orchestration: The Conductor
A container orchestrator is your conductor. It deploys, manages, and scales your app across a cluster of servers. If one server fails, the orchestrator moves the containers to a healthy one.

Kubernetes (K8s): The undisputed de facto standard for orchestration. It offers powerful features like self-healing, rolling updates, and advanced networking. While it has a higher learning curve, it's the choice for heavy-duty production workloads.

Docker Swarm: Docker's built-in tool. It is much easier to use, feeling like a natural extension of your existing Docker commands. It's a great option for smaller teams or apps that need orchestration without the complexity of K8s.

Essential Security Practices
Creating an application is about getting it to run securely.

Manage Your Secrets: Never write database passwords or API keys directly into your Dockerfile. This is a severe security mistake. Instead, use a secrets management tool (like Kubernetes Secrets) that injects them securely at runtime.

Scan Your Images: Always scan your images for known security vulnerabilities before deployment. Tools like Docker Scout, Trivy, or Snyk can be automated to check for vulnerabilities in the base OS or application libraries.

Docker in CI/CD: The Automated Assembly Line

CI/CD (Continuous Integration/Continuous Deployment) is the automated pipeline from a developer's code to a live environment, and Docker powers it.

Build: An automated CI server checks out the code and uses the Dockerfile to create a clean Docker image with a unique version tag.

Test: The server runs the new image as a container and executes automated tests to confirm the code is good.

Push: The approved image is deployed to a secure container registry (like Docker Hub or AWS ECR).

Deploy: The CD phase tells the orchestrator (e.g., Kubernetes) there's a new version. The orchestrator then performs a smooth rolling update, replacing the old containers with the new ones with zero downtime.

Phase 4: The Real World—Orchestration & Security

When you move your application to run reliably on multiple servers for high traffic and scale, you shift from managing one container to managing a fleet.

Container Orchestration: The Conductor
A container orchestrator is your conductor. It deploys, manages, and scales your app across a cluster of servers. If one server fails, the orchestrator moves the containers to a healthy one.

- Kubernetes (K8s): The undisputed de facto standard for orchestration. It offers powerful features like self-healing, rolling updates, and advanced networking. While it has a higher learning curve, it's the choice for heavy-duty production workloads.
- Docker Swarm: Docker's built-in tool. It is much easier to use, feeling like a natural extension of your existing Docker commands. It's a great option for smaller teams or apps that need orchestration without the complexity of K8s.

Essential Security Practices
Creating an application is about getting it to run securely.

1. Manage Your Secrets: Never write database passwords or API keys directly into your Dockerfile. This is a severe security mistake. Instead, use a secrets management tool (like Kubernetes Secrets) that injects them securely at runtime.

2. Scan Your Images: Always scan your images for known security vulnerabilities before deployment. Tools like Docker Scout, Trivy, or Snyk can be automated to check for vulnerabilities in the base OS or application libraries.

Docker in CI/CD: The Automated Assembly Line

CI/CD (Continuous Integration/Continuous Deployment) is the automated pipeline from a developer's code to a live environment, and Docker powers it.

1. Build: An automated CI server checks out the code and uses the Dockerfile to create a clean Docker image with a unique version tag.

2. Test: The server runs the new image as a container and executes automated tests to confirm the code is good.

3. Push: The approved image is deployed to a secure container registry (like Docker Hub or AWS ECR).

4. Deploy: The CD phase tells the orchestrator (e.g., Kubernetes) there's a new version. The orchestrator then performs a smooth rolling update, replacing the old containers with the new ones with zero downtime.