Docker Notes

This document presents a detailed, step-by-step roadmap to learning Docker, starting from the foundational concepts of containers and images, moving through practical usage, and advancing to Docker Compose, networking, security, and orchestration. Each step builds on the previous concepts to provide a comprehensive understanding of Docker for both development and production environments.

Step 1: Docker Overview

What is Docker?

Docker is an open-source platform designed to automate the deployment, scaling, and management of applications using container technology. A container is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings.

Why use Docker?

- **Consistency:** Docker ensures your application runs the same way in development, testing, and production environments, eliminating the "works on my machine" problem.
- **Isolation:** Each container runs independently, so you can run multiple applications on the same host without conflicts.
- **Efficiency:** Containers are lightweight and use fewer resources than virtual machines.
- **Portability:** Docker containers can run on any system that supports Docker, including Windows, macOS, and Linux.

Containers vs Virtual Machines:

- **Virtual Machines (VMs):** Emulate entire operating systems, including the kernel. They are heavier and slower to start.
- **Containers:** Share the host OS kernel, making them lightweight and fast to start. They only contain the application and its dependencies.

Docker Engine Components:

- **Docker Daemon (dockerd):** The background service that manages containers and images.
- **Docker CLI (docker):** The command-line tool to interact with Docker.

Docker Notes

- **Docker REST API:** Allows external programs to communicate with the Docker daemon.

Install & Test:

To get started, install Docker Desktop from the official website. After installation, open a terminal and run:

```
    docker run hello-world
```

This command downloads a test image and runs it in a container. If you see a 'Hello from Docker!' message, your installation is successful.

Step 2: Docker Images & Containers

What is a Docker Image?

A Docker image is a read-only template used to create containers. It contains the application code, libraries, and dependencies required to run the application. Images are built from instructions in a Dockerfile.

What is a Docker Container?

A container is a running instance of a Docker image. It is isolated from the host system and other containers, but can communicate with them if needed.

Image Layers:

Docker images are made up of layers. Each instruction in a Dockerfile creates a new layer. Layers are cached and shared between images, making builds faster and more efficient.

Basic Commands:

```
    docker run nginx            # Run a container using the nginx image
    docker ps -a                # List all containers (running and stopped)
    docker stop <id>            # Stop a running container
    docker rm <id>              # Remove a stopped container
    docker rmi <image>          # Remove an image
```

Tip: Use `docker images` to list all images on your system.

Step 3: Writing Dockerfiles

What is a Dockerfile?

Docker Notes

A Dockerfile is a text file that contains a series of instructions on how to build a Docker image. Each instruction creates a new layer in the image.

Key Instructions:

- **FROM:** Sets the base image (e.g., FROM python:3.10)
- **RUN:** Executes commands in the image (e.g., RUN pip install flask)
- **CMD:** Sets the default command to run when the container starts
- **COPY/ADD:** Copies files from your project into the image
- **ENTRYPOINT:** Sets the main command for the container
- **ENV:** Sets environment variables
- **EXPOSE:** Documents which ports the container will listen on

Example: Multistage Build

This technique helps keep images small by separating build and runtime environments.

```
FROM node:18 AS builder
WORKDIR /app
COPY . .
RUN npm run build

FROM nginx
COPY --from=builder /app/dist /usr/share/nginx/html
```

Tip: Always start with an official base image and keep your Dockerfile simple and readable.

Step 4: Docker Build & Tag

Building Images:

Use the `docker build` command to create an image from your Dockerfile. The `-t` flag lets you tag your image with a name and version.

```
docker build -t myapp:1.0 .
```

Tags help you manage different versions of your images.

.dockerignore:

Create a `.dockerignore` file to exclude files and directories from the build context (like .git, node_modules, etc.). This makes builds faster and images smaller.

Inspecting Images:

```
    docker history <image>   # Shows the layers of an image
    docker inspect <image>   # Shows detailed information about an image
```

Tip: Use meaningful tags for your images, like `myapp:dev` or `myapp:prod`.

Step 5: Running Containers with Options

When running containers, you can use various options to control their behavior:

- **-d:** Run in detached mode (in the background)
- **--name:** Assign a custom name to the container
- **-p:** Map host ports to container ports (e.g., 8080:80)
- **-v:** Mount volumes for persistent data
- **--env:** Set environment variables
- **--rm:** Automatically remove the container when it exits
- **--cpus, --memory:** Limit CPU and memory usage

Example:

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

This runs an nginx container in the background, maps port 8080 on your computer to port 80 in the container, and names it 'web'.

Step 6: Docker Volumes & Bind Mounts

Docker Volumes:

Volumes are the preferred way to persist data generated by and used by Docker containers. Docker manages volumes, and they are stored outside the container's filesystem, so data isn't lost when the container is removed.

Bind Mounts:

Bind mounts map a file or directory from your host machine into the container. Useful for development, but less portable than volumes.

Commands:

```
    docker volume create myvol
    docker run -v myvol:/data ...
    docker volume inspect myvol
```

Tip: Use volumes for databases and important data. Use bind mounts for sharing source code during development.

Step 7: Docker Networking

Docker networking allows containers to communicate with each other, with the host, and with external networks. Understanding Docker networking is crucial for building real-world applications.

Types of Docker Networks:

- **Bridge (default):** The standard network for containers on a single host. Containers on the same bridge network can communicate using container names.
- **Host:** Removes network isolation between the container and the Docker host. The container shares the host's networking namespace. Useful for performance or when you need the container to use the host's IP.
- **None:** Disables networking for the container. The container has no access to the network.
- **Overlay:** Used in Docker Swarm or multi-host setups. Allows containers running on different Docker hosts to communicate securely.
- **Macvlan:** Assigns a MAC address to a container, making it appear as a physical device on the network. Useful for legacy applications.

Creating and Using Custom Networks:

Custom networks provide better isolation and enable containers to communicate by name (DNS resolution).

```
    docker network create mynet
    docker run --network=mynet --name=db postgres
    docker run --network=mynet --name=api myapi
```

Now, the 'api' container can connect to 'db' using the hostname 'db'.

Inspecting Networks:

```
    docker network ls           # List all networks
    docker network inspect mynet # View details about a network
```

Port Mapping:

To access a container from outside Docker, map container ports to host ports:

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

This maps port 8080 on your host to port 80 in the container.

Tips:

- Use custom bridge networks for multi-container apps on a single host.

- Use overlay networks for multi-host communication (requires Docker Swarm).

- Always use container names for communication within a custom network.

Tip: Proper networking setup is essential for scalable and secure Docker deployments.

Step 8: Docker Compose Basics

Docker Compose lets you define and run multi-container Docker applications using a YAML file (`docker-compose.yml`).

Example Structure:

```
    services:
      web:
        image: nginx
        ports:
          - "80:80"
      db:
        image: postgres
        environment:
          POSTGRES_PASSWORD: example
```

Commands:

```
    docker-compose up -d    # Start all services in the background
    docker-compose down     # Stop and remove all services
```

Tip: Compose is great for local development and testing.

Docker Notes

Step 9: Advanced Docker Compose

Advanced features in Docker Compose help manage complex applications:

- **depends_on:** Control the startup order of services.

- **restart:** Automatically restart containers if they fail.

- **healthcheck:** Monitor container health and status.

- **.env files:** Store environment variables outside the compose file.

- **Override files:** Use `docker-compose.override.yml` for local changes.

Example healthcheck:

```
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost"]
      interval: 30s
      timeout: 10s
      retries: 3
```

Tip: Use healthchecks to ensure your services are running as expected.

Step 10: Docker Registry & Docker Hub

Docker Hub is a public registry where you can share and store Docker images. You can also run your own private registry.

Push to Docker Hub:

```
    docker tag myapp myusername/myapp:v1
    docker push myusername/myapp:v1
```

Run a Private Registry:

```
    docker run -d -p 5000:5000 registry:2
```

Tip: Always use secure credentials and avoid pushing sensitive data to public registries.

Step 11: Dockerfile Optimization

Optimizing your Dockerfile makes your images smaller, faster, and more secure:

- Combine RUN commands to reduce layers.

- Use lightweight base images (like alpine).

- Clean up caches and temp files in the same RUN step.

- Use multistage builds to separate build and runtime environments.

Example:

```
FROM golang:alpine AS builder
WORKDIR /app
COPY . .
RUN go build -o myapp

FROM alpine
COPY --from=builder /app/myapp /myapp
ENTRYPOINT ["/myapp"]
```

Tip: Keep your images as small as possible for faster deployments.

Step 12: Docker Security Basics

Security is important when working with containers:

- Use a non-root user in your containers to reduce risk.

- Regularly scan your images for vulnerabilities.

- Use official images and keep them updated.

- Limit container capabilities and resources.

Example (non-root user):

```
RUN adduser --disabled-password myuser
USER myuser
```

Scan Images:

```
docker scan <image>
trivy image <image>
```

Tip: Never store secrets in your Dockerfile or images.

Step 13: Docker Networking Advanced

Advanced networking lets containers communicate across hosts and provides load balancing:

- Use DNS names to connect containers on the same network.

- Overlay networks (in Docker Swarm) allow containers on different hosts to communicate securely.

- Load balancing can be achieved by running multiple replicas of a service.

Example:

```
    docker exec <container> ping db
```

Tip: For production, consider using orchestration tools like Kubernetes or Docker Swarm.

Step 14: Docker Logs and Monitoring

Monitoring and logging are crucial for troubleshooting and performance:

- Use `docker logs` to view container output.

- Change the logging driver to integrate with external systems (e.g., syslog, json-file).

- Use tools like cAdvisor, Prometheus, and Grafana for metrics.

- Use the ELK stack (Elasticsearch, Logstash, Kibana) for centralized logging.

Examples:

```
    docker logs -f <container>
    docker run --log-driver=syslog ...
```

Tip: Always monitor your containers in production.

Step 15: Docker in CI/CD Pipelines

Docker is widely used in Continuous Integration/Continuous Deployment (CI/CD):

- Build and test images in CI pipelines for consistency.

- Use multi-platform builds with buildx for ARM/x86 compatibility.

- Store and deploy images from registries.

Example (GitHub Actions):

```
    jobs:
      build:
        runs-on: ubuntu-latest
```

```
      steps:
        - uses: actions/checkout@v2
        - run: docker build -t myapp .
```

Multi-platform build:

```
    docker buildx create --use
    docker buildx build --platform linux/amd64,linux/arm64 ...
```

Tip: Automate your builds and tests for reliable deployments.

Step 16: Container Orchestration Introduction

Orchestration tools like Kubernetes and Docker Swarm help manage containers at scale:

- Automatically restart failed containers (self-healing)

- Scale services up or down based on demand

- Load balance traffic between containers

- Perform rolling updates with zero downtime

Kubernetes is the industry standard for container orchestration.

Step 17: Docker BuildKit and Buildx

Docker BuildKit is a modern build subsystem for Docker images, offering better performance and features:

- Parallel build steps

- Improved caching

- Support for secrets and SSH forwarding

- Buildx extends BuildKit for cross-platform builds

Enable BuildKit:

```
    export DOCKER_BUILDKIT=1
    docker build .
```

Cross-platform build:

```
    docker buildx build --platform linux/amd64,linux/arm64 -t myapp .
```

Tip: Use BuildKit for faster and more secure builds.

Step 18: Multi-Stage Builds Deep Dive

Multi-stage builds are a powerful Docker feature that lets you use multiple `FROM` statements in a single Dockerfile. This allows you to separate the build environment from the runtime environment, resulting in smaller, more secure images.

Why Use Multi-Stage Builds?

- **Smaller Images:** Only the final artifacts are copied to the runtime image, so build tools and dependencies are left behind.
- **Improved Security:** The final image contains only what is needed to run the application, reducing the attack surface.
- **Cleaner Dockerfiles:** You can organize build steps logically and avoid manual cleanup.

How Multi-Stage Builds Work

Each `FROM` instruction starts a new stage. You can name stages and selectively copy files from one stage to another using the `COPY --from=<stage>` syntax.

Example: Building a Node.js App

```
FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx:alpine AS production
COPY --from=builder /app/build /usr/share/nginx/html
```

In this example:

- The first stage (`builder`) installs dependencies and builds the app.
- The second stage (`production`) copies only the build output into a minimal Nginx image.

Naming Stages

You can name stages for clarity and to reference them easily:

```
    FROM golang:alpine AS build-env
    # ...build steps...
    FROM alpine AS final
    COPY --from=build-env /app/myapp /myapp
```

Benefits

- **No build tools in production images** (e.g., compilers, package managers)
- **Faster deployments** due to smaller image size
- **Easier to maintain and update**

Tips

- Use as many stages as needed (e.g., test, lint, build, production)
- Clean up temporary files in the build stage to keep artifacts small
- Always use official base images for security

Tip: Multi-stage builds are recommended for all production Docker images.

Step 19: Docker Secrets & Configs

Docker Secrets:

Used to securely manage sensitive information (like passwords) in Docker Swarm. Secrets are encrypted and only available to services that need them.

Usage:

```
    echo 'mypassword' | docker secret create db_pass -
```

Access inside container: `/run/secrets/db_pass`

Docker Configs:

Used for non-sensitive configuration files, managed similarly to secrets.

Step 20: Docker Swarm Overview

Docker Swarm is Docker's built-in orchestration tool for clustering and managing containers:

- Service discovery and load balancing
- Rolling updates and scaling
- Secure communication between nodes

Basic Commands:

```
docker swarm init
docker service create --name web -p 80:80 nginx
docker node ls
```

Tip: Swarm is easier to start with than Kubernetes, but less powerful for large-scale production.

Step 21: Docker Storage Drivers

Storage drivers control how Docker stores images and container data on disk. The default is usually overlay2 on modern Linux systems.

Common drivers:

- overlay2 (default)
- aufs (older)
- devicemapper
- btrfs

Check your storage driver:

```
docker info | grep 'Storage Driver'
```

Tip: Use the default unless you have a specific need.

Step 22: Dockerfile Best Practices

Tips for writing efficient and secure Dockerfiles:

- Minimize the number of layers
- Order instructions to maximize build cache
- Use `.dockerignore` to exclude unnecessary files
- Pin base image versions to avoid unexpected changes

- Never store secrets in Dockerfiles
- Use official images when possible
- Add labels for metadata

```
    LABEL maintainer="you@example.com"
```

Tip: Review the official Dockerfile best practices guide.

Step 23: Docker and Kubernetes

Docker is used to build and run containers, while Kubernetes is used to orchestrate and manage containers at scale.

- Docker Desktop includes a local Kubernetes cluster for testing
- Kubernetes uses container runtimes (Docker, containerd, CRI-O)
- Learn about kubectl, manifests, pods, services, and deployments for Kubernetes

Tip: Master Docker basics before diving into Kubernetes.

Step 24: Troubleshooting & Debugging

Common troubleshooting steps:

- Use `docker exec -it <container> /bin/sh` to get a shell inside a running container
- View logs with `docker logs <container>`
- Remove unused images and volumes with `docker system prune`
- Use verbose output for builds: `docker build --progress=plain`
- Monitor resource usage with `docker stats`

```
    docker exec -it <container> /bin/sh
    docker logs <container>
    docker system prune
    docker stats
```

Tip: Always check logs first when debugging.

Step 25: Real World Use Cases

Docker is used in many scenarios:

- Microservices deployment

- Ensuring development and production environment parity
- Automating CI/CD pipelines
- Running hybrid cloud applications
- Reproducible data science and machine learning workflows
- Containerizing legacy applications for modernization

Tip: Start with small projects and gradually adopt Docker in your workflow.

Step 26: Next Steps & Learning Path

- Practice building Dockerfiles for your own projects
- Use Docker Compose for multi-container setups
- Learn Kubernetes basics for orchestration
- Explore cloud container services (AWS ECS, Google GKE, Azure AKS)
- Join Docker and Kubernetes communities for support
- Follow official documentation and tutorials
- Try advanced topics like image signing and vulnerability scanning

Tip: Consistent practice and real-world projects are the best way to learn.