

Containerization

Kay Ashaolu - Instructor

Aishwarya Sriram - TA

Chapter 4: Introduction to Docker

Docker Containers, Images & Compose

Docker vs. Virtual Machines

- **Virtual Machines (VMs):**
 - Emulate a complete operating system (OS)
 - Use a hypervisor to simulate hardware
- **Docker Containers:**
 - Share the host OS kernel (typically Linux)
 - Run as isolated processes with less overhead
- **Key Idea:** Avoid duplicating the full OS stack

How Virtual Machines Work

- **Architecture:**

- **Hardware:** CPU, RAM, GPU, etc.
- **Host OS:** e.g., macOS running on your computer
- **Hypervisor:** Manages VMs on top of the host OS
- **Guest OS:** e.g., Windows running inside a VM

- **Diagram Concept:**

Hardware → Host OS → Hypervisor → Guest OS

Inefficiencies in Virtual Machines

- Each VM runs its own complete OS, including:
 - Duplicate code for hardware interaction
 - Full OS libraries and tools
- **Result:** Increased resource consumption and slower startup times

Docker Containers: A More Efficient Alternative

- **Efficiency:**
 - Containers share the host's OS kernel
 - They run as isolated processes, not full OS instances
- **Advantages:**
 - Faster startup
 - Lower resource overhead
- **Trade-off:** Reduced flexibility (e.g., limited to Linux kernel for most cloud servers)

Why Linux for Docker?

- **Cloud Servers:** Most run Linux
- **Consistency:** Develop locally with Linux containers to mirror production
- **Local Development on Non-Linux Systems:**
 - Docker Desktop creates a Linux VM on macOS/Windows automatically

What is a Docker Image?

- **Definition:** A snapshot containing:
 - Source code
 - Libraries and dependencies
 - Tools and applications
- **Excludes:** The OS kernel (uses the host's kernel)
- **Sources:** Many prebuilt images available (e.g., from Docker Hub)

What is a Docker Container?

- **Runtime Instance:** A running image
- **Characteristics:**
 - Isolated process environment
 - Own storage and networking setup
 - Leverages the host OS kernel
- **Analogy:** Think of it as a lightweight, portable application environment

The Dockerfile: Defining an Image

- **Purpose:** Instruct Docker how to build an image
- **Key Commands:**
 - **FROM:** Specify the base image (e.g., `python:3.10`)
 - **RUN:** Execute commands (e.g., install dependencies)
 - **COPY:** Copy files/directories into the image
 - **CMD / ENTRYPOINT:** Define the container's startup command

Sample Dockerfile for a Flask App



```
1 FROM python:3.10
2 EXPOSE 5000
3 WORKDIR /app
4 RUN pip install flask
5 COPY . .
6 CMD ["flask", "run", "--host", "0.0.0.0"]
```

- **Highlights:**

- **Base Image:** Uses Python 3.10 (includes Debian tools)
- **Port Exposure:** Opens port 5000 for the Flask app
- **Work Directory:** Sets /app as the directory for app files
- **Command:** Starts Flask with external access enabled

Docker Image Layers & Caching

- **Layers:** Each Dockerfile command creates a new layer
- **Caching:**
 - Unchanged layers are cached to speed up rebuilds
 - Modifying one layer can invalidate the cache for subsequent layers
- **Best Practice:** Order Dockerfile commands to maximize cache reuse

Docker Desktop & CLI Tools

- **Docker Desktop:**

- Provides a GUI to manage images, containers, and volumes
- Automatically sets up a Linux VM on macOS/Windows

- **Docker CLI:**

- Command-line interface to build, run, and manage containers
- Common commands: `docker build`, `docker run`, `docker ps`, etc.

Running a Container with Docker Desktop

- **Workflow:**

1. **Build the Image:**

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

2. **Run the Container:**

Use Docker Desktop GUI or CLI commands

3. **Port Forwarding:**

Map host ports (e.g., host port 5005 to container port 5000)

- **Tip:** Visualize and manage containers easily via Docker Desktop

Port Forwarding in Docker

- **EXPOSE Command:**

Declares which port the container listens on (e.g.,
`EXPOSE 5000`)

- **Port Mapping at Runtime:**

`-p <host_port>:<container_port>`

- **Example:**

```
docker run -p 5005:5000 my-flask-app
```

Maps host port 5005 to container's port 5000

Running Containers via the CLI

- **Build the Image:**

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

- **Run the Container:**

```
docker run -p 5005:5000 my-flask-app
```

- **Run in Daemon Mode (Background):**

```
docker run -d -p 5005:5000 my-flask-app
```

Container Persistence & Data Storage

- **Ephemeral Storage:**
Data inside a container lasts until the container is deleted
- **Persistent Data:**
Use Docker volumes to maintain data across container lifecycles
- **Note:** Containers are disposable—treat them as stateless where possible

Introduction to Docker Compose

- **What It Does:**
Orchestrates multi-container applications with a single command
- **Why Use It:**
 - Simplifies starting multiple interdependent services (e.g., API + database)
 - Manages service configurations and dependencies

Docker Compose Configuration File

- **Filename:** `docker-compose.yml` (default name)
- **Structure:**
 - `version`: Specifies the Compose file format (e.g., "3")
 - `services`: Defines each container (e.g., `web`, `db`)
- **Advantage:** No need to type long `docker run` commands for each service

Sample docker-compose.yml for a Flask API



```
1 version: "3"
2 services:
3   web:
4     build: .
5     ports:
6       - "5000:5000"
7     volumes:
8       - ./app
```

- **Breakdown:**
 - **build:** Uses the Dockerfile in the current directory
 - **ports:** Maps host port 5000 to container port 5000
 - **volumes:** Synchronizes the current directory with /app in the container

Volume Mapping in Docker Compose

- **Syntax:**

```
volumes: - <host_directory>:  
          <container_directory>
```

- **Purpose:**

- Keeps local files synchronized with the container
- Ideal for rapid development and testing

- **Outcome:** Instant updates in the running container when code changes

Running Docker Compose

- **Command:**

`docker compose up`

- **What Happens:**

- Builds images (if not already built)
- Starts all services defined in `docker-compose.yml`
- Displays logs for each service (e.g., `web-1`)

- **Stopping Services:**

Use `Ctrl+C` or run `docker compose down`

Rebuilding Images with Docker Compose

- **When to Rebuild:**

After configuration or code changes that affect the image

- **Command Example:**



```
1 docker compose up --build --force-recreate --no-deps web
```

- **Flags Explained:**

- `--build`: Rebuilds the image
- `--force-recreate`: Forces container recreation
- `--no-deps`: Ignores linked service dependencies

Best Practices in Docker Development

- **Develop Consistently:**
Use Linux containers locally to match production environments
- **Optimize Dockerfiles:**
Order commands to maximize caching and reduce rebuild times
- **Stay Updated:**
Use the latest stable base images (e.g., Python 3.10/3.11)

Local vs. Cloud Deployment Considerations

- **Local Development:**
 - Often requires a VM (via Docker Desktop) on macOS/Windows
- **Cloud Deployment:**
 - Typically runs containers on native Linux servers
- **Key Point:**

Ensure that your local development environment mirrors the production setup to avoid unexpected bugs

Conclusion & Next Steps

- **Recap:**
 - Docker offers efficient containerization for backend services
 - Dockerfiles and Docker Compose streamline development and deployment

Questions?