Docker Compose – Full DevOps Power in One File

What is Docker Compose?

> **Docker Compose** is a tool for defining and managing **multi-container** Docker applications using a simple YAML file.

Instead of running `docker run` commands multiple times, Compose lets you:

☑ Define containers, networks, volumes, and environment variables ☑ Start everything with `docker compose up` ☑ Manage dependencies, ports, and data easily

Basic Syntax of `docker-compose.yml`

```yaml
version: "3.9"          # Compose file version
services:               # Define all containers here
  service1:             # A named service (container)
    image: nginx        # Use this image
    ports:
      - "8080:80"       # host:container
```

Why Use Docker Compose?

Feature	Why It Helps
Declarative Setup	All infra defined in one YAML file
Built-in Networking	Services can talk by name (like DNS)
Volume Integration	Persistent data made easy
Auto-Dependency Mgmt	Start db before app, cache, etc.
Dev & Prod Configs	Easy environment switching

Networking in Docker Compose

☑ Auto Network Creation:

All services in a Compose file **automatically share a custom bridge network** with internal DNS!

Service Name	Container DNS Name
db	db

Service Name	Container DNS Name
`redis`	`redis`
`backend`	`backend`

🔧 You can ping other containers **by service name**.

🔍 Example:

```
services:
  web:
    build: .
    ports:
      - "3000:3000"
  api:
    image: node:alpine
    depends_on:
      - web
```

In `api`, you can make requests like:

```
fetch("http://web:3000")
```

🗃️ Volumes in Docker Compose

☑️ Define persistent data storage:

```
volumes:
  mydata:
```

Then attach to a service:

```
services:
  db:
    image: postgres
    volumes:
      - mydata:/var/lib/postgresql/data
```

💧 Compose **creates and manages** these volumes for you!

🧠 Understanding Docker Compose Networking + Port Mapping 🔌

🏗️ Sample `docker-compose.yml` Setup:

Your project folder is:

```
📂 myapp/
  └── docker-compose.yml
```

🔧 Compose File:

```yaml
services:
  web:
    build: .
    ports:
      - "8000:8000"

  db:
    image: postgres
    ports:
      - "8001:5432"
```

🚀 What Happens When You Run:

```
docker compose up
```

☑️ Docker Compose Automatically Does:

🔧 **Action**	💬 **What Happens**
🧱 Creates a network	Named `myapp_default` (based on folder name)
📦 Launches web container	Joins `myapp_default` as `web`
📦 Launches db container	Joins `myapp_default` as `db`
🧠 Enables DNS lookup	`web` can reach `db` by hostname `db`

🌐 Internal Networking (Container ↔ Container)

☑ Inside the `web` container, your app can connect to the database like this:

```
postgres://db:5432
```

🤔 Why? Because:

- Docker provides **internal DNS** to resolve service names
- The port `5432` is the **internal (container) port**, exposed by the Postgres container

---

🌍 External Networking (Host ↔ Container)

You've mapped container ports to host ports like this:

```
web:
  ports:
    - "8000:8000"     # Host: 8000 → Container: 8000

db:
  ports:
    - "8001:5432"     # Host: 8001 → Container: 5432
```

So from **your host machine**, you can:

- Access web on 👉 `http://localhost:8000`
- Connect to Postgres on 👉 `postgres://localhost:8001`

🤔 Important Concept: `HOST_PORT:CONTAINER_PORT`

Concept	Example	Meaning
`HOST_PORT`	`8001`	Port **on your machine**
`CONTAINER_PORT`	`5432`	Port **inside the container**
Mapping	`8001:5432`	Requests to `localhost:8001` go to Postgres in container on port `5432`

🧩 Real World Analogy

> 🛍 Think of containers as hotel rooms.

- Each has its own **room number** (container port)
- The **front desk** (your host machine) assigns a **guest-access number** (host port)

So:

- `5432` = actual DB server inside room

- `8001` = external phone number to reach that room from outside

---

🔒 Internal vs External Communication Recap

Context	URL Format	Who uses it?
🔁 Container-to-container	`postgres://db:5432`	Inside Docker network
🌍 Host-to-container	`postgres://localhost:8001`	From your laptop / browser

☑ Internal comms use **service names + container port** ☑ External comms use **localhost + host port**

📌 Final Notes

- Docker Compose **auto-creates a network** (unless you override it)
- You **don't need to expose ports** unless you want **outside access**
- Use internal ports (`CONTAINER_PORT`) when services talk to each other
- Expose **only the ports you need** to keep things secure 🔐

🧪 Bonus Tip: Inspect the Compose Network

```
docker network inspect myapp_default
```

This will show:

- Containers in the network
- Their IPs
- Connection metadata

🌐 Docker Compose Advanced Networking – A Complete Guide

🚀 1. Multi-Host Networking via **Overlay** (Swarm Mode)

> 💡 Overlay networking allows **containers on different Docker hosts** to communicate — as if they were on the same network!

🔦 Use Case:

☑ Deploying a **multi-host** microservice system ☑ Need backend containers on **host A** to talk to database on **host B**

🛠️ How It Works:

- Requires **Swarm mode** (`docker swarm init`)
- Docker uses **overlay driver** to create a virtual network across machines
- Compose uses this with no special setup, if Swarm mode is active

🧪 Example:

```
networks:
  my_overlay:
    driver: overlay
```

Then attach to services:

```
services:
  web:
    networks:
      - my_overlay
  db:
    networks:
      - my_overlay
```

☑ Compose will automatically connect services to the **multi-host overlay network**

🧠 Overlay vs Bridge (Single Host Only)

Feature	`bridge` **(default)**	`overlay` **(Swarm)**
Host Limit	1 machine	Multiple machines
Use case	Local dev	Distributed apps
DNS-based discovery	☑ Yes	☑ Yes
Needs Swarm mode?	❌ No	☑ Yes

🧱 2. Custom Networks in Compose

You're not limited to just the default network. You can **define and connect containers to specific networks** using the `networks:` key.

🧪 Example Topology

```
services:
  proxy:
```

```
    build: ./proxy
    networks:
      - frontend

  app:
    build: ./app
    networks:
      - frontend
      - backend

  db:
    image: postgres
    networks:
      - backend
```

Network Layout

```
frontend network:
  - proxy
  - app

backend network:
  - app
  - db

proxy 🔁 app 🔁 db
```

☑ `proxy` can't see `db`, but `app` can talk to both ☑ **Great for enforcing security boundaries** 🔐

---

3. Custom Network Drivers & Options

```
networks:
  frontend:
    driver: bridge
    driver_opts:
      com.docker.network.bridge.host_binding_ipv4: "127.0.0.1"

  backend:
    driver: custom-driver
```

- `bridge`: Standard Docker single-host network
- `custom-driver`: Plug in advanced/third-party networking solutions (e.g., macvlan, overlay, weave)

---

4. Rename Docker Network (Custom Name)

```
networks:
  frontend:
    name: custom_frontend
    driver: bridge
```

🔁 Instead of `projectname_frontend`, Docker will create `custom_frontend`.

☑ Useful in CI/CD or pre-defined environments.

🧾 5. Assigning Static IPs in Compose

Use `ipv4_address` under the **service's network attachment**.

```
services:
  web:
    networks:
      app_net:
        ipv4_address: 172.28.0.4

networks:
  app_net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.28.0.0/16
```

☑ Be careful — misconfiguring subnets or overlapping IPs can break your network.

⚙ 6. Customize the `default` Network

You can override the default Compose-generated network like this:

```
services:
  web:
    build: .
    ports:
      - "8000:8000"

networks:
  default:
    driver: custom-driver-1
```

☑ Still lets you use default behavior, but with custom driver/settings.

---

7. Use a Pre-Existing Docker Network

Want to connect to a network created **outside Compose**? Use `external: true`

```
networks:
  network1:
    name: my-pre-existing-network
    external: true
```

Compose will **connect** to the existing network, not recreate it.

Helpful when:

- Using shared infra (like reverse proxies)
- Reusing CI/CD networking
- Linking across Compose projects

Final Cheatsheet

Feature	Keyword	Description
Multi-host networking	`overlay` driver	Works with Swarm
Isolated networks	`networks:` per service	Scoped communication
Custom drivers	`driver: bridge`	Control behavior
Rename network	`name:`	Use meaningful names
Static IPs	`ipv4_address`	Predictable networking
Pre-existing networks	`external: true`	Connect to outside network
Customize default	`networks: default`	Override auto network

Final Takeaways

☑ Compose makes **networking declarative, secure, and powerful** ☑ Use **custom networks** to enforce boundaries and structure ☑ Use **overlay** for multi-host magic ☑ Use **external** to plug into existing infra ☑ DNS-based discovery simplifies microservice URLs

---

Custom Docker Builds in Compose

Use your own `Dockerfile` with build context:

```
backend:
  build:
    context: .                # current folder
```

```
    dockerfile: Dockerfile     # name of your Dockerfile
  ports:
    - "8000:8000"
```

🤔 Docker Compose will build the image **before running** the service.

🧪 Compose vs Manual Docker Commands

Task	Docker CLI	Docker Compose
Start container	`docker run`	`docker compose up`
Stop container	`docker stop`	`docker compose down`
Rebuild image	`docker build`	`docker compose build`
View logs	`docker logs`	`docker compose logs`

🧱 Full Project Example: E-Commerce Stack

```
# 📦 Project Name
name: e-commerce

services:
  # 🔧 Backend Service
  backend:
    build:
      context: .                 # Use current directory as build context
      dockerfile: Dockerfile    # Dockerfile to build the backend image
    container_name: backend     # Name of the backend container
    ports:
      - "8000:8000"             # Map host:container port
    depends_on:
      - db                      # Start db first
      - redis                   # Start redis first

  # 🗄️ PostgreSQL Database Service
  db:
    image: postgres:16          # Latest stable PostgreSQL
    container_name: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: postgres
    volumes:
      - postgres_data:/var/lib/postgresql/data

  # ⚡ Redis In-Memory Data Store
  redis:
    image: redis:7-alpine       # Lightweight Redis image
```

```
    container_name: redis
    volumes:
      - redis_data:/data

# 🗃️ Named Volumes for Persistent Storage
volumes:
  postgres_data:
  redis_data:
```

🔥 How It Works:

- `backend` is built from the local `Dockerfile`
- `db` & `redis` use official images
- Volumes persist data between restarts
- All services can talk to each other via names (`backend`, `db`, `redis`)

☑️ Start Everything:

```
docker compose up --build
```

⛔ Stop & Clean Everything:

```
docker compose down -v
```

🧰 Pro Tips

Tip 💡	Description
`depends_on`	Control **boot order** of containers (not readiness!)
`volumes:`	Use **named volumes** for clean reuse & backups
`env_file:`	Load `.env` for cleaner configuration
`profiles:`	Enable or disable services dynamically
`networks:`	Customize default networks if needed

🎓 Final Summary

Feature	Compose Benefit 🚀
Networking	🔗 Name-based access between containers

Feature	Compose Benefit 🚀
Volumes	📦 Persistent storage, managed cleanly
Builds	🛠️ Custom image building from source
Automation	🤖 Multi-container orchestration made easy
Reusability	♻️ YAML can be reused across environments

🧩 Example: Internal-only Docker Compose Network (No Exposed Ports) 🔒

☑️ Scenario:

You're building a simple **backend + database + Redis** stack that:

- Does **not need to be accessed from the host/browser**
- Only needs **container-to-container communication**
- Should remain **internal and secure** (no `ports:` exposed)

📦 `docker-compose.yml` (No Ports Exposed)

```yaml
version: "3.9"

services:
  # 🔧 Backend API
  backend:
    build:
      context: .
    container_name: backend
    depends_on:
      - db
      - redis
    environment:
      DB_HOST: db
      REDIS_HOST: redis

  # 🗄️ PostgreSQL Database
  db:
    image: postgres:16
    container_name: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: postgres
    volumes:
      - postgres_data:/var/lib/postgresql/data
```

```
  # ⚡ Redis Cache
  redis:
    image: redis:7-alpine
    container_name: redis
    volumes:
      - redis_data:/data

volumes:
  postgres_data:
  redis_data:
```

🧠 Key Points

💡 Concept	☑ Benefit
❌ No `ports:` field	Nothing is exposed to host machine
☑ Internal network	Docker Compose auto-creates a shared bridge
🔗 DNS by service name	`backend` can talk to `db`, `redis` via name
🔐 Security	Fully isolated, no public entry points

💬 How Services Communicate

Inside `backend` container, you can do:

```
// Node.js example (env used above)
const pg = require('pg');
const redis = require('redis');

const db = new pg.Client({
  host: process.env.DB_HOST, // "db"
  user: 'postgres',
  password: 'postgres',
  database: 'postgres'
});

const redisClient = redis.createClient({
  url: 'redis://redis:6379'
});
```

☑ Works seamlessly, because **Docker Compose provides built-in DNS resolution** for service names.

🚀 Run the Stack:

```
docker compose up --build
```

👀 Nothing exposed outside, but all services talk inside. Want to debug? Use:

```
docker exec -it backend sh
```

🧪 When to Use This Pattern

Use Case	Why it Fits
🧱 Internal APIs/microservices	No external access needed
👷 Workers/CRON jobs	Runs in background only
🔐 Security-sensitive apps	Reduce attack surface
🧪 Local-only testing	Don't expose unnecessary ports

🔐 Final Tip

If you later need external access (e.g., for testing): Just **add a single port** to the backend:

```
ports:
  - "8000:8000"
```

But for private, secure, container-to-container apps — **no ports is cleanest**. ☑️

🛠️ Docker Compose – Custom Docker Builds (Full Guide) 🧼

🧩 Overview

Docker Compose allows **two primary ways** to set up containers:

Method	Keyword	Use Case
🛠️ Build locally from Dockerfile	`build:`	During development
☁️ Pull prebuilt image	`image:`	CI/CD & production

☑ 1. **Local Dockerfile Build** – Using `build:`

When you want to **build your Docker image directly from your source code**:

```
services:
  backend:
    build:
      context: .                 # 🗀 Location of source code (root of app)
      dockerfile: Dockerfile     # 📝 File to use (default is 'Dockerfile')
    ports:
      - "8000:8000"
    container_name: backend
```

🗀 Folder Structure:

```
myapp/
├── backend/
│   ├── Dockerfile
│   ├── server.js
│   └── package.json
└── docker-compose.yml
```

🧠 Explanation:

Field	Meaning
`context:`	Folder Docker will send to the daemon for the build
`dockerfile:`	Custom name or path to Dockerfile
`build:`	Triggers a local build when you run `docker compose up --build`

🔁 You don't push/pull — just **edit** → **build** → **run** locally:

```
docker compose up --build
```

☁ 2. **Pull Image from Registry** – Using `image:`

In production or CI/CD, it's better to **pull prebuilt, versioned images** from Docker Hub or GitHub Container Registry.

```
services:
  backend:
```

```
    image: dpvasani56/myapp-backend:v1.0.3
    ports:
      - "8000:8000"
```

☑ Works only **if you've pushed this image earlier**:

```
docker build -t dpvasani56/myapp-backend:v1.0.3 .
docker push dpvasani56/myapp-backend:v1.0.3
```

🤔 This is great when:

- You want reproducible builds ☑
- You deploy to servers that don't have your source code ☑
- You want fast CI/CD ☑

🔁 Both Build & Image (Hybrid)

```
services:
  backend:
    image: dpvasani56/myapp-backend:latest
    build:
      context: ./backend
      dockerfile: Dockerfile
```

💡 In this case:

- Local build happens first
- Image is **tagged and stored locally** as `dpvasani56/myapp-backend:latest`
- Useful for **building locally but keeping consistent image tags**

🧪 Real Example: Backend + Frontend + DB

```
services:
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    image: dpvasani56/app-backend:dev
    ports:
      - "8000:8000"

  frontend:
    image: dpvasani56/app-frontend:latest   # Pulled from Docker Hub
```

```
  db:
    image: postgres:16
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

Push/Deploy Workflow

Step	Action
1	Build locally: `docker build -t dpvasani56/app-backend:dev .`
2	Push to registry: `docker push dpvasani56/app-backend:dev`
3	On server: `docker compose pull && docker compose up -d`

☑ Easy deployment ☑ No source code leak ☑ Fast start time

Custom Dockerfile Path or Name

```
build:
  context: ./src
  dockerfile: Dockerfile.prod
```

👉 You can place your Dockerfile anywhere and name it however you want.

Best Practices

Stage	Use `build:`	Use `image:`
Development 👨‍💻	☑ Yes	❌ Optional
Production 🚀	❌ Avoid	☑ Required
CI/CD 🧪	☑ Build & Push	☑ Pull
Collaboration 👥	☑ Share Compose file	☑ Share tagged image

Full Compose File with Both Options

```
version: "3.9"

services:
```

```
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    image: dpvasani56/my-backend:latest
    ports:
      - "8000:8000"

  frontend:
    image: dpvasani56/my-frontend:latest
    ports:
      - "3000:3000"

  postgres:
    image: postgres:16
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
    volumes:
      - pgdata:/var/lib/postgresql/data

volumes:
  pgdata:
```

📓 Summary Cheatsheet

Field	Use When?	Notes
`build:`	You have Dockerfile locally	Great for dev
`image:`	You push/pull from registry	Ideal for prod
`build` + `image`	Build locally with version tag	Best of both worlds
`context`	Define source code directory	Defaults to `.`
`dockerfile`	Use custom name/location	Optional
