🐳 Docker Compose Example: Node.js + Sequelize + MySQL

This project demonstrates how to run **multiple containers together** using **Docker Compose**. It includes:

- A Node.js Express backend using Sequelize
- A MySQL database
- Communication between containers via a shared network
- Environment variables for config
- A persistent volume for the database

📦 Why Docker Compose?

You already know how to build and run individual Docker containers. But when your app needs more than one container — like a **backend and a database** — things start getting messy:

```
# Start backend container
docker run -p 3000:3000 my-backend

# Start DB container separately
docker run -e MYSQL_ROOT_PASSWORD=secret -d mysql
```

It works, **but it's fragile and hard to maintain**.

Docker Compose solves this by letting you define everything in one file: services, networks, env variables, volumes, and startup order.

> 🧠 Think of Compose as a **docker-based script** for running an entire app environment locally — reliably and consistently.

🧠 The "localhost" Problem (Why Networks Matter)

If you've only used Docker for a single container, `localhost` just works.

But here's the catch:

> When two containers run separately, `localhost` **in one container refers only to itself**, not to other containers.

For example, if your backend tries to connect to `localhost:3306` for MySQL, it will look **inside its own container**, not the DB container — and fail.

☑️ Solution: Docker Networks

Docker Compose **creates a shared private network**, where services can **talk to each other by name**.

In this setup:

- The backend connects to the database using `db` (not `localhost`)
- Docker handles the routing behind the scenes

🔍 What's Inside `docker-compose.yml`

```yaml
services:
  backend:            # Our Express API
    build: ./backend
    env_file:
      - ./backend/.env
    ports:
      - "3000:3000"
    depends_on:
      - db
    networks:
      - appnet

  db:                 # Our MySQL database
    image: mysql:8
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: rootpass
      MYSQL_DATABASE: mydb
      MYSQL_USER: user
      MYSQL_PASSWORD: userpass
    volumes:
      - dbdata:/var/lib/mysql
    networks:
      - appnet

volumes:
  dbdata:             # Named volume to persist MySQL data

networks:
  appnet:             # Custom network for containers to talk to each other
```

🧪 How to Run

```
docker compose up
```

Or if you want to recreate your images (after making changes):

```
docker compose up --build --force-recreate
```

- Access the backend at: http://localhost:3000
- The backend will try to connect to MySQL using the `db` hostname

- MySQL stores its data in a persistent volume, so even after stopping the containers, the data remains

To stop and remove everything:

```
docker compose down
```

🗀 What's in the Backend

- `app.js`: Starts the server and connects to MySQL via Sequelize
- `.env`: Stores config variables like DB name, user, and password
- `Dockerfile`: Builds the Node.js container

You can modify the Sequelize code to define models and create real endpoints (e.g. `/users`, `/products`, etc.).

🎲 Why Use `.env` Files?

Instead of hardcoding things like your database credentials inside the app, we store them in a `.env` file:

```
DB_HOST=db
DB_USER=user
DB_PASSWORD=userpass
DB_NAME=mydb
PORT=3000
```

Docker Compose passes this into the container so your code can access it safely via `process.env`.

🚀 Production Deployments — What Happens Later?

Docker Compose is **meant for local development**. It helps you spin up multiple services on your machine quickly.

But in production, you typically won't:

- Deploy the actual `docker-compose.yml`
- Manually expose ports
- Use `depends_on` for service order

Instead, you'll **replicate the setup using infrastructure tools**, like:

- Kubernetes (K8s)
- Docker Swarm (less common now)
- Azure App Services / Azure Container Apps / AKS
- AWS ECS / Fargate
- Terraform, Pulumi, etc.

> Some cloud providers now offer ways to **deploy Compose files directly**, but that's more for convenience — not a production best practice.

🧱 What You'd Add Later (as Your App Grows)

As your app evolves, you'll likely add:

- A **frontend** (React/Vite/etc.) as a separate service
- An **auth service** or API gateway
- **Redis** or **message queues** for caching or communication
- **nginx** or **Traefik** as a reverse proxy/load balancer
- **Adminer** or **phpMyAdmin** to inspect databases
- **Monitoring** tools like Prometheus/Grafana

And guess what?

You can add all of them in the same `docker-compose.yml` — it scales well for development use.

🧩 Next Step: Kubernetes

After Compose, we'll move to Kubernetes (K8s), where:

- Each container becomes a **Pod**
- Services are managed by **Deployments**
- Networking and scaling are more powerful
- Configuration uses **ConfigMaps** and **Secrets**

But now you'll already understand the core ideas.

> 🧠 If Docker is learning to sail a boat, Docker Compose is learning to sail with a crew. Kubernetes is learning to run a shipping company.

💬 Questions You Should Be Able to Answer

- Why doesn't `localhost` work between containers?
- What is the role of `depends_on`?
- What is the purpose of Docker volumes?
- How do environment variables get passed into a container?
- Why is Compose used in development and not production?

🛠 Troubleshooting Tips

- If you get module errors, rebuild with `--build`
- If the DB fails to connect, make sure `DB_HOST=db`
- Delete volumes if MySQL data gets corrupted:

  ```
  docker compose down --volumes
  ```

🧹 Cleanup

```
docker compose down --volumes --remove-orphans
```

👐 You're Ready

You're not just running containers now — you're orchestrating them.

Welcome to the world of **real-world backend infrastructure**. 🎉