Activity 5: Multi-Container Applications with Docker Compose

🎯 Objective

- Learn how Docker Compose simplifies orchestration of multi-container applications.
- Containerize and orchestrate a real **3-tier demo application** (React frontend, Spring Boot backend, PostgreSQL database).
- Understand Compose concepts step by step:
    - Networks
    - Environment variables
    - Secrets
    - Volumes & persistence
    - Healthchecks & startup sequencing
    - Restart policies
    - Scaling
    - Override files
- Finally, build a **complete production-ready `docker-compose.yml`** from scratch.

✅ Prerequisites

- Completion of **Activity 4** (Networking in Docker).
- A running **Ubuntu 24.04 LTS EC2 instance** with Docker + Docker Compose installed.
- SSH access to EC2 instance (`public-ip` and `secret-key.pem`).
- Internet access on EC2 to pull images.

📂 Lab Setup

- Your lab directory contains:
    - `dockerlab_activity5_3TierApp.tar.xz` → compressed archive of the 3-tier app.
    - `system-clean-init.sh` → helper script that prepares your EC2 environment for this activity.

⚠️ **Important Note about `system-clean-init.sh`:** Running this script will **reset/clean your EC2 instance environment** (remove all containers, images, volumes) and then **extract the 3-tier demo app** into the correct working directory. Use it carefully.

🛠️ Run the setup script (in clab terminal):

```
Shell
bash system-clean-init.sh
```

Then log in to your EC2 instance and move into the extracted app folder:

```
Shell
ssh -i secret-key.pem ubuntu@<public-ip>
cd /home/ubuntu/docker_lab/dockerlab_activity5_3TierApp
```

📦 About the Application

This is a simple **3-tier demo app**:

- **Frontend** → React app served via Nginx.
- **Backend** → Spring Boot REST API that exposes course-related endpoints.
- **Database** → PostgreSQL storing courses, registrations, and grades.

The backend initializes schema & seed data at startup. The frontend consumes backend APIs and shows animated UI.

🐳 Task 1 — Understand the 3-Tier App

🎯 **Goal** Get familiar with the application codebase, its structure, and verify that each component (Postgres, backend, frontend) works when run manually using `docker build` and `docker run`.

📂 Project Tree

```
None
$ pwd
/home/ubuntu/docker_lab/dockerlab_activity5_3TierApp

$ tree -a
.
├── iitb-course-backend
│   ├── .dockerignore
│   ├── .vscode
│   │   ├── launch.json
│   │   └── settings.json
│   ├── Dockerfile
│   ├── README.md
│   ├── pom.xml
│   └── src
│       └── main
│           ├── java
│           │   └── org
│           │       └── iitb
│           │           └── demo
│           │               ├── CourseBackendApplication.java
│           │               ├── config
│           │               │   └── WebConfig.java
│           │               ├── controller
│           │               │   └── CourseController.java
│           │               ├── model
│           │               │   ├── Course.java
│           │               │   ├── Grade.java
│           │               │   └── Registration.java
│           │               └── repository
│           │                   ├── CourseRepository.java
│           │                   ├── GradeRepository.java
│           │                   └── RegistrationRepository.java
│           │
```

```
|           └── resources
|               ├── application.yml
|               ├── db
|               |   └── migration
|               |       └── V1__init_schema_and_data.sql
|               └── logback-spring.xml
└── iitb-course-frontend
    ├── .dockerignore
    ├── Dockerfile
    ├── README.md
    ├── docker-entrypoint.sh
    ├── package.json
    ├── public
    |   ├── env-config.template.js
    |   └── index.html
    └── src
        ├── App.js
        ├── api.js
        ├── components
        |   ├── CourseList.js
        |   └── CourseModal.js
        ├── index.css
        └── index.js

20 directories, 31 files
```

🛠️ Steps

Run the following commands step by step to manually build and run the app components.
👉 Replace `<EC2_IP>` with your EC2 public IP.

Shell

```
# (1) Run Postgres (persistent volume, exposed on host port 5432)
docker run -d --name postgres \
  -e POSTGRES_DB=demo_db \
  -e POSTGRES_USER=demo_user \
  -e POSTGRES_PASSWORD=demo_pass \
  -v pgdata:/var/lib/postgresql/data \
  -p 5432:5432 \
```

```
  postgres:15

# (2) Build and run backend (Spring Boot)
cd iitb-course-backend
docker build -t iitb-course-backend .

docker run -d --name backend \
  -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://<EC2_PRIVATE_IP>:5432/demo_db \
  -e SPRING_DATASOURCE_USERNAME=demo_user \
  -e SPRING_DATASOURCE_PASSWORD=demo_pass \
  iitb-course-backend

# (3) Build and run frontend (React + Nginx)
cd ../iitb-course-frontend
docker build -t iitb-course-frontend .

docker run -d --name frontend \
  -p 3000:80 \
  -e REACT_APP_API_BASE_URL="http://<EC2_PUBLIC_IP>:8080" \
  iitb-course-frontend

# (4) Verify endpoints
# From EC2 host:
curl -i http://localhost:8080/api/health
curl -i http://localhost:8080/api/courses

# From your laptop browser:
# - http://<EC2_PUBLIC_IP>:3000  (frontend UI)
# - http://<EC2_PUBLIC_IP>:8080/api/health
# - http://<EC2_PUBLIC_IP>:8080/api/courses

# (5) Inspect logs
docker logs -f backend
docker logs -f frontend
docker logs -f postgres

# (6) Cleanup when done
docker stop frontend backend postgres
docker rm frontend backend postgres
```

📌 Notes

- Here we use **published ports (`-p`)** so services are directly reachable on `<EC2_PUBLIC_IP>`.
- No Docker network is needed at this stage because all traffic flows via host ports.
- This task is for **understanding only** — *no test cases are provided*.

⚠️ Before starting Task 2, **stop all running containers** and rerun (in clab terminal):

```
Shell
bash system-clean-init.sh
```

This ensures a clean environment and avoids port conflicts.

📝 *Note* — Why Docker Compose?

When working with multi-container applications (like our **3-tier app** with frontend, backend, and database), manually running containers with `docker run` becomes **tedious and error-prone**. You would have to remember:

- Each container's `docker run` command.
- Port mappings, environment variables, volumes, and networks.
- Startup order (DB before backend, backend before frontend).

To solve this, **Docker Compose** was created. It lets you:

- Define all services in a **single YAML file (`docker-compose.yml`)**.
- Use simple commands like `docker compose up` and `docker compose down` to start/stop everything.
- Automatically create a network so services can talk by name.
- Handle configuration in a **reproducible** and **portable** way.

⚡ **Why it came to market:** Teams needed a **standardized tool** to manage multi-container apps easily, without custom shell scripts or manual orchestration. Compose filled that gap and became the de facto tool for local development and small deployments.

🐳 Task 2 — Compose Basics (Single-Service Postgres)

🎯 **Goal** Learn the basics of Docker Compose by running a single Postgres service. This task introduces the `docker-compose.yml` format and basic commands.

🛠️ Steps

1. Create a new file named `docker-compose.yml` inside the `dockerlab_activity5_3TierApp` folder with the following content:

```
None
services:
  postgres:
    image: postgres:15
    container_name: postgres
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: demo_db
      POSTGRES_USER: demo_user
      POSTGRES_PASSWORD: demo_pass
```

📖 Explanation of keywords

- **services** → Top-level key defining all containers (services) in the app.
- **postgres** → Name of the service (also acts as its DNS hostname on the Compose network).
- **image** → The Docker image to use (`postgres:15`).
- **container_name** → Explicitly names the container (otherwise Compose generates one).
- **ports** → Publishes container's port 5432 to host port 5432 so we can connect from outside.

- **environment** → Environment variables passed into the container:
  - `POSTGRES_DB` → Creates a database named `demo_db`.
  - `POSTGRES_USER` → Creates a user `demo_user`.
  - `POSTGRES_PASSWORD` → Password for that user.

🛠️ Run and Verify

1. Start the service in detached mode:

```
Shell
docker compose up -d
```

2. Check running containers:

```
Shell
docker compose ps
docker network ls
docker volume ls
```

3. View logs for Postgres:

```
Shell
docker compose logs postgres
```

4. Stop the service:

```
Shell
docker compose down
```

📌 Notes

- The `version` attribute is now **obsolete** in Docker Compose v2.
- At this stage we are not using **volumes**, **networks**, or **healthchecks**.
- Docker Compose automatically creates a **default network**, but since only one service is running, this doesn't matter yet.
- *No test cases are provided for this task* — it is for learning and practice only.

⚠️ Before moving on, ensure you stop services with `docker compose down` to keep your environment clean.

🐳 Task 3 — Custom Networks in Compose

🎯 **Goal** Learn how to define and use a **user-defined bridge network** in Docker Compose so services can communicate by name and be isolated from other containers.

🛠️ Steps

1. Create or update `docker-compose.yml` in `dockerlab_activity5_3TierApp` to declare a named network and attach the `postgres` service to it. Example minimal file:

```
None
services:
  postgres:
    image: postgres:15
    container_name: postgres
    ports:
      - "5432:5432"
    environment:
      POSTGRES_DB: demo_db
      POSTGRES_USER: demo_user
      POSTGRES_PASSWORD: demo_pass
    networks:
      - app-net

networks:
  app-net:
    driver: bridge
```

2. Start the service (the network will be created by Compose):

```shell
docker compose up -d
```

3. Verify the network exists and inspect its details:

```shell
docker network ls
docker network inspect dockerlab_activity5_3tierapp_app-net
# or inspect by the name shown in `docker network ls`
```

4. Run another container on the same network to test name resolution (example: lightweight `alpine` client):

```shell
docker run -it --rm --network dockerlab_activity5_3tierapp_app-net alpine sh
# inside the alpine shell, install tools and test DNS/HTTP:
apk add --no-cache curl iputils
ping -c 2 postgres
curl -sI http://postgres:5432 || true
```

> Note: Replace `dockerlab_activity5tierapp_app-net` above with the actual network name reported by `docker network ls`. Compose-derived network names are usually `<folder>_<network>` unless `name:` is provided explicitly under the `networks:` block.

📖 Explanation of keywords

- **networks:** → Top-level key defining networks that Compose will create.
- **app-net:** → Logical network name you choose (used by services to attach).
- **driver: bridge** → Creates a user-defined bridge network (default and suitable for single-host deployments).

- **services.networks** → Lists networks the service will join. Joining the same user-defined network enables **DNS-based service discovery** by service name (`postgres`).

📌 Verification Checklist

- ✅ `docker network ls` shows a new network for this Compose project.
- ✅ `docker network inspect <network>` lists the `postgres` container under `Containers`.
- ✅ From another container attached to the same network, `ping postgres` resolves and responds.

📌 Notes & Tips

- Compose automatically creates networks declared in the `networks:` section when you run `docker compose up`.
- If you want to **control the exact network name**, add a `name:` under the network definition:

```
None
networks:
  app-net:
    name: iitb-app-net
    driver: bridge
```

- Using a user-defined network is generally **preferable to the default bridge** because it provides built-in DNS and easier service discovery.
- For Task 3 we attached only Postgres to the custom network — later tasks will attach backend and frontend so they communicate internally by service name.
- *No test cases are provided for this task* — it is for learning and practice only.

⚠️ Before moving on, keep the Compose services running or stop them with:

```shell
docker compose down
```

🐳 Task 4 — Environment Variables in Compose

🎯 **Goal** Learn how to externalize configuration for containers using a `.env` file so you don't hardcode values in `docker-compose.yml`.

🛠️ Steps

1. Create a `.env` file at the project root (`dockerlab_activity5_3TierApp/.env`) with the following contents:

```
# .env
POSTGRES_DB=demo_db
POSTGRES_USER=demo_user
POSTGRES_PASSWORD=demo_pass
```

2. Update your `docker-compose.yml` to reference these variables:

```
services:
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
      - "5432:5432"
    networks:
      - app-net
```

```
networks:
  app-net:
    driver: bridge
```

3. Validate substitution and final config:

```
Shell
docker compose config
```

This shows the resolved configuration with variables substituted.

4. Start the service:

```
Shell
docker compose up -d
```

5. Verify Postgres is running with your custom values:

```
Shell
docker compose logs postgres
```

📖 Explanation of keywords & behavior

- **`.env` file** → Compose automatically loads variables from `.env` if present in project **root**.
- **`${VAR}` syntax** → Substitutes environment variables inside `docker-compose.yml`.
- **`docker compose config`** → Debug tool to render full YAML after substitution.

📌 Verification Checklist

- ✅ `.env` file exists and contains the database variables.
- ✅ `docker compose config` shows `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD` substituted.
- ✅ Postgres starts successfully and logs show the DB and user created with your values.

📋 Notes

- At this stage only **Postgres** is part of the Compose file — backend and frontend will come later.
- This task focuses purely on externalizing config for database service.
- If you want to override a value temporarily, export it in your shell before `docker compose up` (shell env has higher precedence):

```shell
export POSTGRES_PASSWORD="temp_pass"; docker compose up -d
```

- For reproducible labs, keep `.env` in the repo for default values but **never commit real production secrets**.
- *No test cases are provided for this task* — it is for learning and practice only.

⚠️ Before moving on, keep the Compose services running or stop them with:

```shell
docker compose down
```

🐳 Task 5 — Secrets in Compose (Compose-native)

🎯 **Goal** Secure sensitive values by using Docker Compose **secrets** (Compose can mount secret files into containers). Move the database password out of `.env` into a

secret file so it is not stored alongside other configuration. We will keep `POSTGRES_DB` and `POSTGRES_USER` in `.env` and **remove** `POSTGRES_PASSWORD` from `.env`.

🛠️ Steps

1. **Remove password from `.env`** (edit `dockerlab_activity5_3TierApp/.env`):

```
None
# .env (updated)
POSTGRES_DB=demo_db
POSTGRES_USER=demo_user
# POSTGRES_PASSWORD is intentionally removed from .env
```

2. **Create a directory and secret file on the host** (with strict permissions):

```
Shell
mkdir -p ./secrets
printf "demo_pass" > ./secrets/postgres_password.txt
chmod 400 ./secrets/postgres_password.txt
```

3. **Update `docker-compose.yml`** to use the secret and still supply DB name/user via `environment:`. Because the official `postgres` image expects `POSTGRES_PASSWORD` in the environment, we use a small wrapper that reads the mounted secret and exports `POSTGRES_PASSWORD` before starting Postgres:

```
None
services:
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      # DO NOT set POSTGRES_PASSWORD here; it will come from the secret
```

```
    secrets:
      - postgres_password
    networks:
      - app-net
    entrypoint: [ "sh", "-c" ]
    command: >
      "export POSTGRES_PASSWORD=$(cat /run/secrets/postgres_password) &&
       exec docker-entrypoint.sh postgres"

secrets:
  postgres_password:
    file: ./secrets/postgres_password.txt

networks:
  app-net:
    driver: bridge
```

4. **Start the stack**:

```
Shell
docker compose up -d
```

5. **Verify the secret is mounted and Postgres started**:

```
Shell
# Check mounted secret inside container
docker exec -it postgres sh -c "ls -l /run/secrets && cat
/run/secrets/postgres_password"

# Check Postgres logs for successful startup
docker logs postgres
```

6. **(Optional)** Remove the secret file from host after use if needed, or **keep it secure outside the repository**.

📖 Explanation of keywords & behavior

- **secrets:** — Top-level Compose key declaring secrets. Using `file:` tells Compose to use a local file as the secret source and mount it into containers at `/run/secrets/<name>`.
- **service.secrets:** — The service consumes the secret; at runtime the secret is mounted read-only to `/run/secrets/<name>`.
- **Why `environment:` still needed:** `POSTGRES_DB` and `POSTGRES_USER` must be provided so Postgres can initialize the database and user. We keep those in `.env` and reference them via `environment:`.
- **Why wrapper is used:** The official `postgres` image expects `POSTGRES_PASSWORD` via an environment variable. Since Compose mounts secrets to files, we use a minimal wrapper command that reads `/run/secrets/postgres_password` and exports `POSTGRES_PASSWORD` before invoking the standard entrypoint. This keeps the password out of `.env` and avoids leaking it in the process command or image layers.
- **Permissions:** Keep the host secret file with tight permissions (`chmod 400`) so it is not world-readable.

📌 Verification Checklist

- ✅ `.env` file has `POSTGRES_DB` and `POSTGRES_USER` but **no** `POSTGRES_PASSWORD`.
- ✅ `./secrets/postgres_password.txt` exists on host and has `chmod 400`.
- ✅ `docker compose up -d` starts the `postgres` service successfully.
- ✅ Inside the `postgres` container `/run/secrets/postgres_password` exists and contains the expected password.
- ✅ Postgres logs show database initialization and readiness.

📌 Notes & Best Practices

- Removing `POSTGRES_PASSWORD` from `.env` prevents accidental commit of secrets to VCS.
- Compose-native `secrets` (with `file:`) allow you to keep secret files out of the main repo while still mounting them at runtime.

- Do **not** commit `./secrets/postgres_password.txt` into source control. Add `./secrets/` to `.gitignore`.
- For images that natively read secrets, the wrapper is unnecessary — you could then read `/run/secrets/<name>` directly in the app. For the official `postgres` image, the small wrapper is the simplest safe approach.
- *No test cases are provided for this task* — it is for learning and practice only.

⚠️ Before moving on, keep the Compose services running or stop them with:

```
Shell
docker compose down
```

🐳 Task 6 — Add Backend to Compose

🎯 **Goal** Wire the Spring Boot backend into your Compose setup so it connects to Postgres (using the custom network and envs/secrets configured earlier). The backend will be built from the local source (`build:`) for this learning phase and will mount a volume for persistent logs.

🛠️ Steps

1. Ensure you are in the project root and `.env` + `./secrets/postgres_password.txt` exist (from previous tasks).
2. Update `docker-compose.yml` to add the `backend` service (merge this snippet into your existing Compose file):

```
None
services:
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
```

```
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
    secrets:
      - postgres_password
    networks:
      - app-net
    entrypoint: [ "sh", "-c" ]
    command: >
      "export POSTGRES_PASSWORD=$(cat /run/secrets/postgres_password) &&
       exec docker-entrypoint.sh postgres"
    volumes:
      - pgdata:/var/lib/postgresql/data

  backend:
    build:
      context: ./iitb-course-backend
    image: iitb-course-backend:local
    container_name: backend
    ports:
      - "8080:8080"               # expose backend for host testing
    environment:
      SPRING_DATASOURCE_URL:
${SPRING_DATASOURCE_URL:-jdbc:postgresql://postgres:5432/${POSTGRES_DB}}
      SPRING_DATASOURCE_USERNAME: ${POSTGRES_USER}
      # Do NOT set SPRING_DATASOURCE_PASSWORD here; backend will read DB
password from secret file if implemented
    secrets:
      - postgres_password
    networks:
      - app-net
    volumes:
      - backend-logs:/app/logs     # persist application logs (Logback
configured to write here)
    depends_on:
      - postgres

volumes:
  pgdata:
  backend-logs:

secrets:
  postgres_password:
    file: ./secrets/postgres_password.txt
```

```
networks:
  app-net:
    driver: bridge
```

3. Build and start services:

```
Shell
docker compose up --build -d
```

4. Watch startup logs (backend will apply migrations/seed data using the DB):

```
Shell
docker compose logs -f backend
```

5. Verify backend is reachable from host:

```
Shell
# health endpoint
curl -i http://localhost:8080/api/health

# list courses
curl -i http://localhost:8080/api/courses
```

6. Inspect persisted logs (inside container):

```
Shell
docker exec -it backend sh -c "ls -la /app/logs && tail -n 100 /app/logs/*.log
|| true"
```

7. Stop services when done:

```shell
docker compose down
```

📖 Explanation of snippet keywords

- **build.context** → Points Compose to the backend source directory so the image is built locally from `./iitb-course-backend`.
- **image** → Names the locally built image. Helpful for caching and later switching to Docker Hub images.
- **ports** → Maps backend port 8080 in container to 8080 on host, so you can curl it from EC2 or your laptop (`http://<EC2_IP>:8080`).
- **environment** → Passes DB connection details. `SPRING_DATASOURCE_PASSWORD` is excluded for security; backend must read it from the secret file.
- **secrets** → Mounts the Postgres password at `/run/secrets/postgres_password`. The backend must be configured (via wrapper or code) to use it.
- **volumes** → Persists logs at `/app/logs` so they survive container restarts.
- **depends_on** → Ensures `postgres` starts before `backend`. Does not wait for readiness — proper sequencing via healthchecks will be added in Task 8.

📌 Verification Checklist

- ✅ `docker compose up --build -d` starts `postgres` and `backend` successfully.
- ✅ Backend logs show successful DB connection and migrations.
- ✅ `curl http://localhost:8080/api/health` returns `ok`.
- ✅ `curl http://localhost:8080/api/courses` returns seeded course data.
- ✅ Backend logs are persisted in the `backend-logs` volume across restarts (`docker compose restart backend`).

📌 Notes

- We use `docker compose up --build -d` to ensure the backend image is **rebuilt each time** you change the source code. If you skip `--build`, Compose will reuse the last built image and you may not see your changes.
- Later, when we switch to prebuilt Docker Hub images in the **final exercise**, you'll only need `docker compose up -d`.
- If backend fails to authenticate, confirm it can read `/run/secrets/postgres_password`. As a fallback, you can temporarily add `SPRING_DATASOURCE_PASSWORD` in `.env` for debugging, but secrets must be used for the graded exercise.
- *No test cases are provided for this task* — it is for learning and practice only.

🐳 Task 7 — Add Frontend to Compose

🎯 **Goal** Integrate the React-based frontend into the Compose setup. The frontend will be built from local source (`build:`) and will connect to the backend API using an environment variable passed at container startup. Unlike Postgres, the backend and frontend **must** expose a port to the host so you can access it in your browser.

🛠️ Steps

1. Ensure you are in the project root (`dockerlab_activity5_3TierApp`).
2. Update `docker-compose.yml` to add the `frontend` service:

```
None
services:
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
    secrets:
      - postgres_password
    networks:
```

```
      - app-net
    entrypoint: [ "sh", "-c" ]
    command: >
      "export POSTGRES_PASSWORD=$(cat /run/secrets/postgres_password) &&
       exec docker-entrypoint.sh postgres"
    volumes:
      - pgdata:/var/lib/postgresql/data

  backend:
    build:
      context: ./iitb-course-backend
    image: iitb-course-backend:local
    container_name: backend
    ports:
      - "8080:8080"               # expose backend for outside api call
    environment:
      SPRING_DATASOURCE_URL:
${SPRING_DATASOURCE_URL:-jdbc:postgresql://postgres:5432/${POSTGRES_DB}}
      SPRING_DATASOURCE_USERNAME: ${POSTGRES_USER}
    secrets:
      - postgres_password
    networks:
      - app-net
    volumes:
      - backend-logs:/app/logs
    depends_on:
      - postgres

  frontend:
    build:
      context: ./iitb-course-frontend
    image: iitb-course-frontend:local
    container_name: frontend
    ports:
      - "3000:80"        # map container's Nginx port 80 to host port 3000
    environment:
      REACT_APP_API_BASE_URL: "http://<EC2_PUBLIC_IP>:8080"
    networks:
      - app-net
    depends_on:
      - backend

volumes:
  pgdata:
```

```
  backend-logs:

secrets:
  postgres_password:
    file: ./secrets/postgres_password.txt

networks:
  app-net:
    driver: bridge
```

3. Build and start all services:

```
Shell
docker compose up --build -d
```

4. Verify frontend logs:

```
Shell
docker compose logs -f frontend
```

5. Access the app in your browser using the EC2 public IP:

```
None
http://<EC2_PUBLIC_IP>:3000
```

6. The frontend should display the course list and interact with the backend API.

7. Stop services when done:

```
Shell
docker compose down
```

📖 Explanation of snippet keywords

- **build.context** → Builds the frontend image from `./iitb-course-frontend`.
- **image** → Names the local frontend image (helps with caching).
- **ports "3000:80"** → Maps host port 3000 to container port 80 (Nginx). This is **necessary** because the frontend must be accessible from your laptop browser.
- **environment.REACT_APP_API_BASE_URL** → Configures the frontend to call backend APIs at `http://<EC2_PUBLIC_IP>:8080`. This is needed because a React application is served to the client's browser, the JavaScript code runs **on the client's machine**. Any API calls in the code will be executed by the browser. The browser doesn't know what `'backend'` is, since it's a hostname that only exists within your Docker network..
- **jdbc:postgresql://postgres:5432/${POSTGRES_DB}}** → In contrast, for DB calls DNS resolves postgres to the specific Postgres container IP in the same network.
- **networks** → Puts all three services (`postgres`, `backend`, `frontend`) on the same custom network. This removes the need for exposing ports for **backend** and **postgres** — they communicate internally via service names (`backend`, `postgres`).
- **depends_on** → Ensures `backend` starts before `frontend`. Does not wait for readiness yet (healthchecks will be added later).

📌 Verification Checklist

- ✅ `docker compose up --build -d` starts `postgres`, `backend`, and `frontend` together.
- ✅ Logs show Nginx serving frontend build.
- ✅ Visiting `http://<EC2_IP>:3000` shows the React UI.
- ✅ Frontend fetches data from backend (`/api/courses`) without CORS issues.
- ✅ No explicit host port for **postgres** is needed since communication happens via the internal network.

📌 Notes

- **Frontend** and **Backend** must expose ports (`3000:80`, `8080:8080`) because your browser is outside Docker.

- **Postgres** does not need a port exposed — it only communicates with the backend through the Compose network.
- Keep using `docker compose up --build -d` while working with local source code. Later, when switching to prebuilt Docker Hub images in the **final exercise**, `--build` will not be required.
- *No test cases are provided for this task* — it is for learning and practice only.

🐳 Task 8 — Healthchecks & Startup Sequencing

🎯 **Goal** Make service startup robust by adding **healthchecks** for Postgres, backend, and frontend, and ensure Compose starts dependent services only after their dependencies are healthy. This avoids race conditions (backend trying DB before DB is ready, frontend calling backend that hasn't finished booting).

🛠️ Steps

1. Ensure you are in the project root (`dockerlab_activity5_3TierApp`).
2. Update `docker-compose.yml` to add `healthcheck` endpoints::

```
None
services:
  postgres:
    image: postgres:15
    container_name: postgres
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
    secrets:
      - postgres_password
    networks:
      - app-net
    entrypoint: [ "sh", "-c" ]
    command: >
      "export POSTGRES_PASSWORD=$(cat /run/secrets/postgres_password) &&
       exec docker-entrypoint.sh postgres"
    volumes:
      - pgdata:/var/lib/postgresql/data
```

```
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB} -h
localhost"]
      interval: 10s
      timeout: 5s
      retries: 10
      start_period: 10s

  backend:
    build:
      context: ./iitb-course-backend
    image: iitb-course-backend:local
    container_name: backend
    ports:
      - "8080:8080"               # expose backend for outside api call
    environment:
      SPRING_DATASOURCE_URL:
${SPRING_DATASOURCE_URL:-jdbc:postgresql://postgres:5432/${POSTGRES_DB}}
      SPRING_DATASOURCE_USERNAME: ${POSTGRES_USER}
    secrets:
      - postgres_password
    networks:
      - app-net
    volumes:
      - backend-logs:/app/logs
    depends_on:
      postgres:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:8080/api/health || exit 1"]
      interval: 10s
      timeout: 5s
      retries: 10
      start_period: 20s

  frontend:
    build:
      context: ./iitb-course-frontend
    image: iitb-course-frontend:local
    container_name: frontend
    ports:
      - "3000:80"        # map container's Nginx port 80 to host port 3000
    environment:
      REACT_APP_API_BASE_URL: "http://<EC2_PUBLIC_IP>:8080"
```

```
    networks:
      - app-net
    depends_on:
      backend:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost/ || exit 1"]
      interval: 15s
      timeout: 5s
      retries: 8
      start_period: 15s

volumes:
  pgdata:
  backend-logs:

secrets:
  postgres_password:
    file: ./secrets/postgres_password.txt

networks:
  app-net:
    driver: bridge
```

🛠️ Steps

1. Save the `docker-compose.yml` (merge with your existing file; ensure `.env` and `./secrets/postgres_password.txt` exist).
2. Start the stack:

```
Shell
docker compose up --build -d
```

3. Watch services and their health status:

```shell
docker compose ps
# For detailed health JSON:
docker inspect --format='{{json .State.Health}}' postgres | jq
docker inspect --format='{{json .State.Health}}' backend | jq
docker inspect --format='{{json .State.Health}}' frontend | jq
```

4. Tail logs if any service becomes unhealthy:

```shell
docker compose logs -f postgres
docker compose logs -f backend
docker compose logs -f frontend
```

📖 How `condition: service_healthy` helps

- `healthcheck` defines a command Docker runs periodically inside the container (e.g., `pg_isready` for Postgres, `curl` to the backend health endpoint). Docker tracks the container's health status (`starting`, `healthy`, or `unhealthy`).
- `depends_on` with `condition: service_healthy` tells Compose to wait to start the dependent service until the dependency's health status is `healthy`. In the snippet above:
  - `backend` depends on `postgres: { condition: service_healthy }` — Compose will delay starting the `backend` service until Postgres reports `healthy`.
  - `frontend` depends on `backend: { condition: service_healthy }` — Compose will delay starting the `frontend` service until the backend reports `healthy`.
- This sequence reduces startup races: backend won't attempt DB migration until DB is ready; frontend won't send requests to backend before it's ready.

📌 Verification Checklist

- ✅ `docker compose up --build -d` starts services.

- ✅ After a short period, `docker compose ps` shows services with `healthy` state.
- ✅ `docker inspect .State.Health` shows `"Status": "healthy"` for Postgres, backend, and frontend.
- ✅ Backend logs indicate migrations applied and readiness.
- ✅ Frontend UI is responsive once frontend becomes healthy.

📌 Notes & Caveats

- `condition: service_healthy` relies on Compose honoring that field. In many environments and Compose versions this is supported; if you observe `depends_on` not waiting, check your Docker Compose version and behavior.
- If your Compose version does **not** support `condition: service_healthy`, simply remove the `condition` block and keep plain `depends_on` (services will start in order, but not wait for health).

Example update:

```
None
backend:
  depends_on:
    - postgres

frontend:
  depends_on:
    - backend
```

- Tune `interval`, `retries`, and `start_period` to match observed startup times — overly aggressive healthchecks can mark services `unhealthy` prematurely.
- Avoid placing secrets or sensitive data directly in healthcheck commands.
- *No test cases are provided for this task* — it is for learning and practice only.

🐳 Task 9 — Restart Policies

🎯 **Goal** Learn how to configure container restart behavior in Compose using `restart:` policies so services recover automatically from failures or behave predictably after host reboots.

🛠️ Steps

1. **Add restart policies to your services** in `docker-compose.yml`. Common policies:

```
None
services:
  postgres:
    image: postgres:15
    restart: unless-stopped
    # ... (other config)

  backend:
    build: ./iitb-course-backend
    restart: on-failure
    # ... (other config)

  frontend:
    build: ./iitb-course-frontend
    restart: always
    # ... (other config)
```

2. **Meaning of common restart policies**

- `no` (default) — Do not automatically restart the container.
- `always` — Always restart the container if it stops. If the container is stopped **manually**, it will be restarted when **Docker daemon restarts**.
- `unless-stopped` — Like `always`, but does **not** restart the container if it was stopped via `docker stop` (useful for interactive debugging).
- `on-failure[:max-retries]` — Restart only if the container exits with a non-zero exit code. Optionally supply `:max-retries` (e.g., `on-failure:5`).

3. **Apply and observe behavior**

```shell
# Start services
docker compose up -d

# Simulate failures:
# Example: kill the backend process inside container (or stop the container)
docker exec -it backend sh -c "kill 1" || true
# Or stop a service from host
docker stop backend

# Observe restart behavior
docker ps --filter name=backend --format "table {{.Names}}\t{{.Status}}"
docker logs -f backend
```

4. **Test host reboot behavior (optional / caution)** Rebooting the host will show how `always` vs `unless-stopped` behave:

- With `always`, containers come back up after Docker daemon starts.
- With `unless-stopped`, containers that were not manually stopped will restart; containers manually stopped remain stopped.

📖 Explanation & Rationale

- Use **`on-failure`** for services that should automatically retry transient errors (e.g., temporary DB connection issues).
- Use **`unless-stopped`** as a sensible default for long-running services you want to persist across host reboots but still allow manual stop during maintenance.
- Use **`always`** for critical services that must be running regardless of how they were stopped (use carefully — this can complicate debugging).

📌 Verification Checklist

- ✅ Services configured with `restart:` behave as expected after container termination (they restart or remain stopped according to policy).
- ✅ After simulating a crash (`kill` or `docker stop`), `docker ps` shows the container restarted per policy.

- ✅ After a host reboot, containers with `always` or `unless-stopped` return to running state if they were not previously manually stopped (verify after reboot).

📌 Notes & Best Practices

- For production orchestration (Kubernetes/Swarm), restart semantics differ — restart policies here apply to single-host Docker Compose setups.
- Be careful with `restart: always` in development — it can make iterative debugging harder because containers automatically respawn. Use `unless-stopped` for a friendlier development experience.
- Combine restart policies with robust **healthchecks** (Task 8) so failing services don't repeatedly restart without addressing root causes.
- *No test cases are provided for this task* — it is for learning and practice only.

🐳 Task 10 — Scaling Backend

🎯 **Goal** Understand the idea of scaling services with multiple replicas in Docker Compose, its challenges in this 3-tier demo, and what can be achieved in real systems.

📖 Concept

Scaling allows you to run **multiple replicas of a service**. Docker's embedded DNS will return multiple IPs for the same service name, enabling internal load distribution across replicas.

⚠️ Challenges in our 3-tier app

1. **Container names** → You cannot use `container_name:` with scaling (each replica must have a unique name).
2. **Port publishing** → Only one container can bind to a host port (e.g., `8080`). When scaling replicas, they cannot all expose the same host port.

3. **Testing load distribution** → Our backend has no endpoint that returns replica-specific info (like container ID or hostname), so you cannot observe which replica served the request.
4. **Frontend dependency** → The frontend expects to call `backend:8080` inside the Docker network, so scaling is only useful internally unless a reverse proxy/load balancer is added.

💡 What scaling enables in real systems

- Run multiple replicas of the backend → distribute load and improve fault tolerance.
- Use an internal DNS round-robin (Compose built-in) or add a reverse proxy (NGINX, Traefik) for external load-balancing.
- Combine scaling with **stateless services** (no in-memory session state) to make replicas interchangeable.
- Useful for **high availability** — if one replica crashes, others continue serving.

📌 Notes

- Scaling in Compose is configured at runtime only:

```shell
docker compose up --build -d --scale backend=2
```

- The `deploy.replicas` field exists in the Compose spec but is **ignored** by plain Docker Compose (only works in Swarm/Kubernetes).
- For production-grade scaling and load balancing, pair this with a reverse proxy or move to an orchestrator (Swarm/K8s).

No test cases are provided for this task — it is for learning and practice only.

🐳 Task 11 — Logging Drivers

🎯 **Goal** Understand what Docker logging drivers are, and how they differ from application log files written to mounted volumes.

📖 Concept

- **Logging drivers** manage how Docker captures a container's **stdout/stderr** output.

```
None
services:
  backend:
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
```

    - This stores container logs in JSON files on the host and rotates them after 10 MB.

- Other drivers forward logs to external systems (`syslog`, `fluentd`, `none`).
- **Mounted log volumes** (like `/app/logs` in our backend) are **different**:
    - The application itself writes files to disk.
    - Rotation/retention is handled by the app's logging framework (Logback in our case), not by Docker.

📌 Notes

- Use **logging drivers** to control and forward container output (good for central log collection).
- Use **mounted volumes** for application-specific log files (good for persistence and analysis).
- In practice, many teams configure apps to log to **stdout** and let Docker drivers handle collection + forwarding.

No test cases are provided for this task — it is for learning and practice only.

🐳 Task 12 — Override Files (Dev vs Prod)

🎯 **Goal** Learn how to manage different configurations for **development** and **production** using Docker Compose override files, so you don't need to constantly edit the base `docker-compose.yml`.

📖 Concept

Docker Compose allows you to **combine multiple YAML files**. The rules are simple:

- **`docker-compose.yml`** → always the **base file**.
- **`docker-compose.override.yml`** → if present, it is **applied automatically** on top of the base when you run `docker compose up`.
- **Any other override file** (e.g., `docker-compose.prod.yml`, `docker-compose.dev.yml`) → used only when explicitly passed with `-f`.

👉 Order matters: later files **override or extend** settings from earlier files.

🛠️ Steps

1. **Create a base `docker-compose.yml`** (production-like defaults):

```
None
services:
  backend:
    image: <dockerhub-username>/iitb-course-backend:latest
    ports:
      - "8080:8080"
    restart: unless-stopped
```

2. **Create a `docker-compose.override.yml`** (development tweaks). Since Compose loads this file automatically, you don't need to pass `-f` in commands.

```
None
services:
  backend:
    build: ./iitb-course-backend
    volumes:
      - ./iitb-course-backend/src:/app/src   # mount local code
    environment:
      SPRING_PROFILES_ACTIVE: dev
    restart: "no"
```

✅ Here, `build:` from `override.yml` **replaces** the `image:` from the base file. ✅ The `volumes:` and `environment:` are **added** on top.

3. **Create a `docker-compose.prod.yml`** (explicit production overrides). This is not auto-loaded; you must specify it manually.

```
None
services:
  backend:
    environment:
      SPRING_PROFILES_ACTIVE: prod
    logging:
      driver: "json-file"
      options:
        max-size: "10m"
        max-file: "3"
```

4. **Run in development** (base + override automatically):

```
Shell
docker compose up --build -d
```

5. **Run in production** (base + prod override):

```
Shell
docker compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

📖 Which file overrides which?

- Default behavior:

```
None
docker-compose.yml  ← base
docker-compose.override.yml  ← applied automatically on top
```

- With custom files:

```
None
docker-compose.yml  ← base
docker-compose.prod.yml  ← applied on top when passed with -f
```

- When multiple files are passed: the **last one wins** for conflicting keys.

Example run order:

```
Shell
docker compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

Here, `docker-compose.prod.yml` overrides both the base and the default override.

📌 Verification Checklist

- ✅ Development run uses **override.yml** automatically → builds from source, mounts volumes, no auto-restart.
- ✅ Production run with `-f docker-compose.prod.yml` uses prebuilt images, enables logging driver, and restart policy.

📌 Notes

- Use **`docker-compose.override.yml`** for developer convenience (auto-applied).
- Use **`docker-compose.prod.yml`** (or `dev`, `staging`) for explicit environment configs.
- Compose never merges arrays like `ports:` — they are replaced. Keys like `environment:` and `volumes:` are merged.
- *No test cases are provided for this task* — it is for learning and practice only.

🏁 Task 13 — Final Exercise: Write a Complete Compose File

🎯 **Goal** Bring together all concepts from this activity by writing a complete `docker-compose.yml` from scratch. This file should run the **3-tier course app** using prebuilt images from Docker Hub, with all best practices applied.

🛠️ Instructions

1. **Clean your environment** (from your local clab terminal): Reset your lab environment to avoid conflicts:

```shell
# run the provided setup which resets EC2 and extracts the app
bash system-clean-init.sh
# now SSH into EC2
ssh -i secret-key.pem ubuntu@<EC2_PUBLIC_IP>
cd /home/ubuntu/docker_lab/dockerlab_activity5_3TierApp
```

2. **Write a new `docker-compose.yml`** in the project root. Follow these requirements **exactly** so that the autograder can validate:

    - **Services and images**
        - `course-db` → `postgres:15`
        - `course-backend` → `soumik13/iitb-course-backend:v2`

        - `course-frontend` → `soumik13/iitb-course-frontend:v2`
    - **Network**
        - All services must be attached to a custom network → `course-net`.
    - **Volumes**
        - `course-db-data` → for Postgres data.
        - `course-backend-logs` → for backend logs.
    - **Secrets**
        - Secret name: `course_db_password`.
        - File path: `./secrets/course_db_password.txt`.
    - **Environment variables**
        - Define `POSTGRES_DB` and `POSTGRES_USER` in `.env`.
        - Use the secret for `POSTGRES_PASSWORD`.
        - Backend must connect to DB using the Compose service name (`course-db`).
        - Frontend must use env variable `REACT_APP_API_BASE_URL` pointing to backend (`http://<EC2_PUBLIC_IP>:8080`).
    - **Ports**
        - Expose only:
            - `course-frontend` → `3000:80`
            - `course-backend` → `8080:8080`
            - `course-db` → **no host port**
    - **Add the following for each service:**
        - **Healthchecks** → ensure each service is actually "ready" (DB → `pg_isready`, backend → `/api/health`, frontend → `/`).
        - **Restart policies** → e.g., `unless-stopped` for DB/frontend, `on-failure` for backend.
        - **depends_on with service_healthy** → backend waits for DB to be healthy, frontend waits for backend to be healthy.

3. **Skeleton (fill in the TODOs yourself):**

```
None
services:
  course-db:
```

```
    # TODO: add image (postgres:15)
    # TODO: add environment variables (POSTGRES_DB, POSTGRES_USER)
    # TODO: mount secret for password
    # TODO: mount volume for db data
    # TODO: attach to custom network
    # TODO: add restart policy
    # TODO: add logging driver with rotation
    # TODO: add healthcheck using pg_isready

  course-backend:
    # TODO: add image (soumik13/iitb-course-backend:v2)
    # TODO: add environment variables (JDBC URL, DB username)
    # TODO: mount secret for db password
    # TODO: expose port 8080
    # TODO: mount volume for backend logs
    # TODO: attach to custom network
    # TODO: add depends_on with service_healthy for course-db
    # TODO: add restart policy
    # TODO: add logging driver with rotation
    # TODO: add healthcheck (curl /api/health)

  course-frontend:
    # TODO: add image (soumik13/iitb-course-frontend:v2)
    # TODO: add environment variable (REACT_APP_API_BASE_URL)
    # TODO: expose port 3000:80
    # TODO: attach to custom network
    # TODO: add depends_on with service_healthy for course-backend
    # TODO: add restart policy
    # TODO: add logging driver with rotation
    # TODO: add healthcheck (curl /)

volumes:
  # TODO: define course-db-data volume
  # TODO: define course-backend-logs volume

secrets:
  # TODO: define course_db_password secret with file path
./secrets/course_db_password.txt

networks:
  # TODO: define course-net network (bridge driver)
```

📌 Verification Checklist

- ✅ `course-db` is healthy.
- ✅ `course-backend` is healthy and responds at `http://<EC2_PUBLIC_IP>:8080/api/courses`.
- ✅ `course-frontend` loads at `http://<EC2_PUBLIC_IP>:3000` and fetches courses from backend.
- ✅ Data persists in `course-db-data`, and backend logs persist in `course-backend-logs`.
- ✅ Restart policies and logging drivers are correctly configured.
- ✅ Only required ports are exposed (3000, 8080).

🔐 Required file names & exact EC2 paths (must match exactly)

All files must be placed inside the extracted lab folder on the EC2 instance:

```
None
/home/ubuntu/docker_lab/dockerlab_activity5_3TierApp/docker-compose.yml
/home/ubuntu/docker_lab/dockerlab_activity5_3TierApp/.env
/home/ubuntu/docker_lab/dockerlab_activity5_3TierApp/secrets/course_db_password
.txt
```

Make sure `course_db_password.txt` has permission `chmod 400`.

🔁 Copy files back to your local clab workspace for submission

After verifying everything on EC2, **copy the three files** **`docker-compose.yml`,`.env`,`secret/course_db_password.txt`** from EC2 to your **local clab workspace** (so you can submit / keep copies).

📌 **Final Step** : submit it for testing.
The **autograder will run it and validate each checkpoint automatically.**

🧹 Cleanup

Stop all services and reset workspace:

```shell
# Stops and removes all containers, networks, and services from your project
# Also deletes associated named volumes and orphaned containers for a complete cleanup.
docker compose down --volumes --remove-orphans

# From clab run it
bash system-clean-init.sh
```

🛑 Stop / Terminate Instance

After completing the activity and saving your work:

1. Exit SSH session:

```shell
exit
```

2. Stop or terminate the EC2 instance from the AWS Management Console if no longer required.

📚 Appendix — Extra Compose Concepts

Here are some additional Docker Compose features that are not part of this lab but are useful to know for real-world projects:

Concept	What it Does	Why it's Useful
Profiles	Allows you to group services and start only selected ones.	Avoids always running optional services (e.g., monitoring, debugging).
External Networks & Volumes	Lets containers join pre-created networks or use existing volumes.	Enables multiple Compose projects or standalone containers to share data or talk over the same network.
Resource Limits	Restricts how much CPU/memory a service can consume.	Prevents one service from consuming all host resources (works in Swarm/K8s, not plain Compose).
Build Caching	Controls whether Docker uses cached layers during builds.	Ensures a fresh build when debugging or avoiding stale files.
Configs	Mounts non-sensitive config files (e.g., Nginx conf) into containers.	Cleanly manages config files without baking them into images. Useful for apps needing flexible configs.

🏆 Congratulations — You've completed Activity 5!

END OF DOCUMENT