Assignment #1: Docker Containers

The purpose of this assignment is to build and run a simple two-container stack using Docker and Docker Compose. One container will run a PostgreSQL database, while the other will host a lightweight Python application. The Python app will connect to the database, query a few rows, compute basic statistics, and then print and save the results. This exercise introduces you to the fundamentals of multi-container setups, including service networking, environment variables, and reproducible workflows. These are essential skills that you will continue to use and expand upon in the course project.

Learning goals:

- Run a database in a container and seed it with initial data.
- Connect from another container using service DNS, ports, and env vars.
- Query records and compute simple statistics.
- Package everything to run with one command.
- Use GitHub to manage your code and configuration.

What you will build:

- Service 1: PostgreSQL
  - Starts with a seeded table: `trips(city TEXT, minutes INT, fare NUMERIC)`
  - Data is inserted via an `init` script.
- Service 2: Python app
  - Connects to Postgres using `env` vars.
  - Runs a couple of read queries.
  - Computes simple stats.
  - Writes `out/summary.json` and prints a small table to `stdout`.

1. Installation Requirements

To complete this assignment, you must have the following tools installed and working on your machine:

- **Docker Desktop**: required to build images and run containers.
- **Docker Compose**: for defining and running multi-container applications.
- **GitHub Desktop**: for version control and to manage your assignment repository.
- **IDE**: such as VS Code, PyCharm, or any editor you prefer.

In addition:

- Ensure you can run basic Docker commands (docker build, docker run, docker ps) before starting.
- Verify that port 5432 (PostgreSQL default) is free on your system or adjust your Compose file accordingly.
- You should be comfortable creating and navigating directories, editing files, and running commands from the terminal.

Suggested GitHub repo layout:

```
.
├── app/                    # Python application
│   └── main.py             # Application entrypoint (queries DB + outputs summary)
├── db/                     # Database configuration
│   ├── Dockerfile.db       # Custom Postgres Dockerfile
│   └── init.sql            # Schema + seed data (auto-executed at startup)
├── out/                    # Output directory (ignored in grading, created at runtime)
├── compose.yml             # Defines how app + db run together
├── Dockerfile.app          # App container definition (Python + psycopg)
├── Makefile                # One-command run (build, up, down, clean)
├── run.sh                  # (optional) script alternative to Makefile
└── README.md               # Project description, commands, example outputs, troubleshooting
```

2. Setup the Database service container

Create the Dockerfile for the database: this file defines the database container. It starts from the official Postgres image and copies your initialization script into the special folder that Postgres checks on first startup. Any .sql files in that folder will be executed automatically, allowing you to preconfigure tables and seed data.

db/Dockerfile

```
FROM postgres:16
COPY init.sql /docker-entrypoint-initdb.d/
```

Create the initialization SQL script: this script sets up your schema and inserts a few rows of sample data. When the container runs for the first time, Postgres will automatically execute the script. You should expand it later for your final tests.

db/init.sql (example)

```
CREATE TABLE trips (
  id SERIAL PRIMARY KEY,
  city TEXT NOT NULL,
  minutes INT NOT NULL,
  fare NUMERIC(6,2) NOT NULL
);
INSERT INTO trips (city, minutes, fare) VALUES
('Charlotte', 12, 12.50),
('Charlotte', 21, 20.00),
('New York', 9, 10.90),
('New York', 26, 27.10),
('San Francisco', 11, 11.20),
('San Francisco', 28, 29.30);
```

How it works: when the `db` container starts for the first time, Postgres automatically executes any scripts in `/docker-entrypoint-initdb.d/`. This ensures your database is initialized with the correct schema and seed data without manual setup.

3. Setup the Python app service container

Create the Dockerfile for the app: This file defines the Python service container. It starts from a lightweight Python image, installs psycopg (a PostgreSQL driver), sets the working directory, copies your application code into the container, and specifies the default command to run.

`app/Dockerfile`

```
FROM python:3.11-slim
RUN pip install --no-cache-dir psycopg[binary]==3.1.19
WORKDIR /app
COPY main.py /app/
CMD ["python", "main.py"]
```

Create the Python application: This script connects to the PostgreSQL database using environment variables passed in via `compose.yml`. It retries the connection until the database is ready, executes a few queries (total trips, average fare per city, top N longest trips), and outputs the results both to the console and as a JSON file in `/out/summary.json`. Complete the missing parts.

`app/main.py`

```
import os, sys, time, json
import psycopg

# Environment variables with defaults
DB_HOST = os.getenv("DB_HOST", "db")
DB_PORT = int(os.getenv("DB_PORT", "5432"))
DB_USER = ...
DB_PASS = ...
DB_NAME = ...
TOP_N   = int(os.getenv("APP_TOP_N", "5"))
```

```
def connect_with_retry(retries=10, delay=2):
    last_err = None
    for _ in range(retries):
        try:
            conn = psycopg.connect(
                host=DB_HOST,
                port=DB_PORT,
                user=DB_USER,
                password=DB_PASS,
                dbname=DB_NAME,
                connect_timeout=3,
            )
            return conn
        except Exception as e:
            last_err = e
            print("Waiting for database...", file=sys.stderr)
            time.sleep(delay)
    print("Failed to connect to Postgres:", last_err, file=sys.stderr)
    sys.exit(1)

def main():
    conn = connect_with_retry()
    with conn, conn.cursor() as cur:
        # Total number of trips
        cur.execute("...")
        total_trips = cur.fetchone()[0]

        # Average fare by city
        cur.execute("""
            ...
            ...
            ...
            ...
        """)
        by_city = [{"city": c, "avg_fare": float(a)} for (c, a) in cur.fetchall()]

        # Top N trips by DEFINE YOUR QUERY
        cur.execute("""
            ...
        """, (TOP_N,))
        top = ...

    summary = {
        "total_trips": int(total_trips),
        "avg_fare_by_city": by_city,
        "top_by_minutes": top
    }

    # Write to /out/summary.json
    os.makedirs("/out", exist_ok=True)
    with open("/out/summary.json", "w") as f:
        json.dump(summary, f, indent=2)

    # Print to stdout
    print("=== Summary ===")
    print(json.dumps(summary, indent=2))

if __name__ == "__main__":
    main()
```

4. Compose the two services together

Compose the two services: Now that you have created the database and Python app containers, the next step is to define how they work together using Docker Compose. The `compose.yml` file specifies both services, their environment variables, volumes, and dependencies. This configuration allows you to launch the entire stack with a single command, so the database and application run together seamlessly. Complete the missing parts of the file.

compose.yml

```
services:
  db:
    build:
      context: ./db
      dockerfile: Dockerfile
    environment:
      POSTGRES_USER: appuser
      POSTGRES_PASSWORD: secretpw
      POSTGRES_DB: appdb
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U appuser -d appdb"]
      interval: 3s
      timeout: 3s
      retries: 10

  app:
    build:
      context: ./app
      dockerfile: Dockerfile
    depends_on:
      db:
        condition: service_healthy
    environment:
      ...
      APP_TOP_N: "10"
    volumes:
      - ./out:/out
```

Run the stack: Once the compose.yml file is set up, you can run both containers together with:

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

This command builds the images (if they are not already built), starts the PostgreSQL database service, and then runs the Python app. The app will connect to the database, perform its queries, and write the output to the `out/` folder. You should also see the JSON summary printed directly to your terminal.

5. One-command run

Write a Makefile: at this stage, you can simplify your workflow by using a Makefile for common Docker Compose commands. This ensures that building, starting, and cleaning up your containers can be done with a single, easy-to-remember command. In addition, your `README.md` should provide clear documentation so others (or you, later) can run the stack without confusion.

Makefile

```
COMPOSE=docker compose
build:
    $(COMPOSE) build app
up:
    $(COMPOSE) up --build
down:
    $(COMPOSE) down -v
clean: down
    rm -rf out && mkdir -p out
all: clean up
```

Tip: Make sure to use tabs (not spaces) before each command.

Test: Run make in your project folder. It should build the images, start both services, and produce the output JSON.

6. Document the work

Document: every project must include a clear `README.md` file at the root of the repository. The README explains what your stack does and provides all the information needed for someone else to build and run it without asking you questions. Think of it as your project's "user guide". It should include at least:

- What the stack does (2–3 sentences)
- Exact commands to run/stop
- Example output (copy/paste)
- Where outputs are written
- Troubleshooting (DB not ready, permission on `out/`, etc.)

7. Submission

Your final deliverables must be submitted on Canvas. Each student must turn in:

- The GitHub repository URL containing your assignment code, Dockerfiles, `compose.yml`, and README.
- A 1-page PDF that includes:
    - The exact commands you used to build and run the stack.

- A pasted summary from `stdout` (the JSON printed by the app).
- The contents of `out/summary.json`.
- A short reflection (3–5 sentences) describing what you learned and what you would improve.

The PDF file can be exported directly from your README or written separately. Keep it concise, professional, and easy to read.

8. Grading rubric

- Correct multi-container setup (20 pts): Compose runs; services reachable; app reads from DB; summary written to `/out`.
- Data model and seed (15 pts): Proper table + init data via `init.sql`.
- App correctness (25 pts): Queries execute; basic stats computed; error handling for failures.
- Reproducibility (10 pts): One-command run; clear README.
- Code & organization (10 pts): Clean repo structure; small Docker image; no secrets committed.
- Reflection (20 pts): Clear, specific takeaways.