Exercise: Persistent API with Custom PostgreSQL and Docker Run

Juan David Guarnizo Hernandez, PhD
juan.guarnizo@tec.mx

TC2007B - AD2025

Goals

By the end of this comprehensive exercise you will be able to:

1. Create a custom PostgreSQL Docker image that includes a database schema.
2. Use an `init.db.sql` script to initialize a table on container startup.
3. Configure a GitHub Actions workflow to automatically build and publish the custom *database* image to the GitHub Container Registry (GHCR).
4. Modify a Flask API to use the `psycopg2` library for database persistence.
5. Create a health-check endpoint to validate database connectivity.
6. Read database connection parameters (host, user, pass, db) from environment variables.
7. Use the default Docker network to allow two separate containers to communicate by IP address.
8. Deploy and test a multi-container application (API and DB) using only `docker run` commands.

This exercise assumes you are running a Kali Linux virtual machine on your personal device. This exercise builds on the `app.py` and GitHub Actions workflow from the previous lab.

---

0) Software and Files Required

- **Kali Linux:** A running instance (Guest OS in a VM is recommended).
- **Docker:** Must be installed and running.
- **Accounts in:**
  - Github - https://github.com

System Installation

You must have Docker installed and running on your Kali Linux system.

```
sudo systemctl start docker
sudo systemctl enable docker
```

This exercise also uses `jq` to format JSON output, which you can install with:

```
sudo apt update
sudo apt install -y jq
```

Part 1: The Custom Database Container

First, we will create a dedicated, custom Docker image for our PostgreSQL database. This image will not just run Postgres; it will also automatically create our `news` table when it starts for the first time.

1.1) Create a New GitHub Repository

Create a **new, public** GitHub repository. Let's call it `my-db`. Clone this repository to your Kali VM.

```
git clone https://github.com/YOUR_USERNAME/my-db.git
cd my-db
```

1.2) Create the Initialization Script

Inside the `my-db` folder, create a file named `init.db.sql`. This SQL script will be executed by the Postgres image on its first run.

```
CREATE TABLE IF NOT EXISTS news (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) NOT NULL,
    content TEXT
);
```

init.db.sql

1.3) Create the Database Dockerfile

Now, in the same folder, create a file named `Dockerfile`.

```
FROM postgres:14-alpine
COPY init.db.sql /docker-entrypoint-initdb.d/
```

Dockerfile

Task: Build and Test the DB Image Locally

Let's build and run this image to prove it works.

1. Build the image:

```
docker build -t my-local-db:1.0 .

```

2. Run the image. We must provide the password as an environment variable.

```
docker run -d --name test-db -e POSTGRES_PASSWORD=mysecretpassword -p
    5432:5432 my-local-db:1.0

```

3. Wait a few seconds for it to initialize, then connect to it using `psql` (you may need to `sudo apt install postgresql-client`):

```
docker exec -it test-db psql -U postgres

```

4. Inside the `psql` prompt, list the tables with `\dt`. You should see your `news` table!

5. Type `\q` to exit `psql`, then stop and remove the test container:

```
docker stop test-db
docker rm test-db

```

Part 2: Automate DB Image with GitHub Actions

Now that our database image works locally, let's commit our two files (`init.db.sql` and `Dockerfile`) and set up GitHub Actions to publish it to the GitHub Container Registry (GHCR).

2.1) Commit and Push

Commit and push your files (`init.db.sql` and `Dockerfile`) to the `main` branch of your repository.

2.2) Set Up GitHub Actions

In your `my-db` GitHub repository, go to the **Actions** tab and create a new workflow. Name the file `.github/workflows/`

```yaml
name: CI-DB

on:
  push:
    branches: [ "main" ]

permissions:
  contents: read
  packages: write

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ghcr.io/${{ github.repository }}

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Check out the repo
        uses: actions/checkout@v4

      - name: Log in to GHCR
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build and push
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ env.IMAGE_NAME }}:latest
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

.github/workflows/docker-publish.yml

Commit this file and push it (or commit it directly in the GitHub UI). Go to the **Actions** tab and watch it run. Once it succeeds, go to your repository's main page. On the right-hand side, click **Packages** to confirm your new database image is there.

Part 3: Connecting the API and Testing the Connection

We will now modify our existing API from the previous lab to prepare it for the database. Go to the repository you used for the previous lab (e.g., `my-api`).

3.1) Update the Dockerfile

Your API now has a new dependency: the PostgreSQL driver for Python (`psycopg2`). Modify your `Dockerfile` to install it.

```
FROM python:latest
WORKDIR /app
COPY . .
RUN pip install Flask psycopg2-binary
EXPOSE 3000
CMD ["python", "app.py"]
```

Dockerfile

3.2) Update app.py for Initial Connection Test

Replace the content of your `app.py` with this code. This version adds the database connection logic and a new `/db-health` endpoint, but **keeps the original in-memory** news functionality for now.

```
from flask import Flask, jsonify, request, abort
import os
import psycopg2
import time

app = Flask(__name__)

def get_db_connection():
    db_host = os.environ.get('DB_HOST')
    db_name = os.environ.get('DB_NAME')
    db_user = os.environ.get('DB_USER')
    db_pass = os.environ.get('DB_PASS')

    retries = 5
    while retries > 0:
        try:
            conn = psycopg2.connect(
                host=db_host,
                database=db_name,
                user=db_user,
                password=db_pass)
            return conn
        except psycopg2.OperationalError:
            retries -= 1
            app.logger.warning("Database not ready, retrying...")
            time.sleep(5)

```

```
    app.logger.error("Could not connect to database.")
    return None

@app.route("/db-health", methods=["GET"])
def db_health_check():
    conn = get_db_connection()
    if conn is None:
        return jsonify({"status": "error", "message": "Database connection failed
    "}), 500

    conn.close()
    return jsonify({"status": "ok", "message": "Database connection successful"})

# -- Previous in-memory news code omitted --

if __name__ == "__main__":
    app.run(threaded=True, host='0.0.0.0', port=3000)
```

app.py (Connection Test Version)

Task: Push, Automate, and Test Connection

1. Commit and push your changed `app.py` and `Dockerfile` to your API repository.
2. Go to the **Actions** tab and verify your workflow builds and publishes the new image.
3. **Deploy the Database:** Run your database container from GHCR. (Replace `YOUR_GITHUB_USERNAME`).

```
export DB_PASS="my-super-secret-password-123"
export DB_USER="news_user"
export DB_NAME="news_db"

docker pull ghcr.io/YOUR_GITHUB_USERNAME/my-db:latest

docker run -d --name news-db \
    -e POSTGRES_PASSWORD=$DB_PASS \
    -e POSTGRES_USER=$DB_USER \
    -e POSTGRES_DB=$DB_NAME \
    ghcr.io/YOUR_GITHUB_USERNAME/my-db:latest

```

4. **Find the Database IP:**

```
export DB_HOST_IP=$(docker inspect -f '{{range .NetworkSettings.Networks
    }}{{.IPAddress}}{{end}}' news-db)
echo "Database IP is: $DB_HOST_IP"

```

5. **Deploy the API:** Run your new API image, passing in the database credentials and IP. (Replace `YOUR_API_REPO_NA`

```
docker pull ghcr.io/YOUR_GITHUB_USERNAME/YOUR_API_REPO_NAME:latest

docker run -d --name news-api -p 3000:3000 \
    -e DB_HOST=$DB_HOST_IP \
    -e DB_NAME=$DB_NAME \
    -e DB_USER=$DB_USER \
    -e DB_PASS=$DB_PASS \
    ghcr.io/YOUR_GITHUB_USERNAME/YOUR_API_REPO_NAME:latest

```

6. **Test the Connection:**

```
curl http://localhost:3000/db-health | jq

```

If successful, you should see the "Database connection successful" message.

7. **Test the In-Memory Endpoint:**

```
curl http://localhost:3000/news | jq

```

This should still return the in-memory data, proving the old endpoints are unaffected.

8. **Cleanup:** Stop and remove the test containers before proceeding.

```
docker stop news-api news-db
docker rm news-api news-db

```

Part 4: Migrating CRUD Endpoints to Database

Now that we know the connection works, we will replace the in-memory functions with real database queries.

Update app.py (Full Migration)

Replace the content of your `app.py` again, this time with the final version. This version removes the in-memory list and connects all `/news` endpoints to the database.

```
from flask import Flask, jsonify, request, abort
import os
import psycopg2
import time

app = Flask(__name__)

def get_db_connection():
    db_host = os.environ.get('DB_HOST')
    db_name = os.environ.get('DB_NAME')
    db_user = os.environ.get('DB_USER')
    db_pass = os.environ.get('DB_PASS')

    retries = 5
    while retries > 0:
        try:
            conn = psycopg2.connect(
                host=db_host,
                database=db_name,
                user=db_user,
                password=db_pass)
            return conn
        except psycopg2.OperationalError:
            retries -= 1
            app.logger.warning("Database not ready, retrying...")
            time.sleep(5)

    app.logger.error("Could not connect to database.")
    return None
```

```
30
31 @app.route("/db-health", methods=["GET"])
32 def db_health_check():
33     conn = get_db_connection()
34     if conn is None:
35         return jsonify({"status": "error", "message": "Database connection failed
    "}), 500
36
37     conn.close()
38     return jsonify({"status": "ok", "message": "Database connection successful"})
39
40 @app.route("/", methods=["GET"])
41 def index():
42     return jsonify({
43         "message": "Welcome to the News API (with Postgres)!",
44         "endpoints": {
45             "list_all_news": "GET /news",
46             "create_news": "POST /news",
47             "update_news": "PUT /news/<id>",
48             "delete_news": "DELETE /news/<id>",
49             "db_health": "GET /db-health"
50         }
51     })
52
53 @app.route("/news", methods=["GET"])
54 def list_news():
55     conn = get_db_connection()
56     if conn is None:
57         return jsonify({"error": "Database connection failed"}), 500
58
59     items = []
60     try:
61         with conn.cursor() as cur:
62             cur.execute("SELECT id, title, content FROM news ORDER BY id;")
63             rows = cur.fetchall()
64             for row in rows:
65                 items.append({"id": row[0], "title": row[1], "content": row[2]})
66     except Exception as e:
67         app.logger.error(f"Error listing news: {e}")
68         return jsonify({"error": str(e)}), 500
69     finally:
70         if conn:
71             conn.close()
72
73     return jsonify({"count": len(items), "items": items})
74
75 @app.route("/news", methods=["POST"])
76 def create_news():
77     if not request.json or not 'title' in request.json:
78         abort(400)
79
80     title = request.json['title']
81     content = request.json.get('content', "")
82
83     conn = get_db_connection()
84     if conn is None:
85         return jsonify({"error": "Database connection failed"}), 500
86
87     new_item = {}
88     try:
89         with conn.cursor() as cur:
```

```
            cur.execute(f"INSERT INTO news (title, content) VALUES ('{title}', '{content}') RETURNING id;")
            new_id = cur.fetchone()[0]
            conn.commit()
            new_item = {"id": new_id, "title": title, "content": content}
    except Exception as e:
        app.logger.error(f"Error creating news: {e}")
        conn.rollback()
        return jsonify({"error": str(e)}), 500
    finally:
        if conn:
            conn.close()

    return jsonify(new_item), 201

@app.route("/news/<int:item_id>", methods=["PUT"])
def update_news(item_id: int):
    if not request.json:
        abort(400)

    conn = get_db_connection()
    if conn is None:
        return jsonify({"error": "Database connection failed"}), 500

    updated_item = {}
    try:
        with conn.cursor() as cur:
            cur.execute(f"SELECT title, content FROM news WHERE id = {item_id};")
            item = cur.fetchone()
            if not item:
                abort(404)

            title = request.json.get('title', item[0])
            content = request.json.get('content', item[1])

            cur.execute(f"UPDATE news SET title = '{title}', content = '{content}' WHERE id = {item_id};")
            conn.commit()
            updated_item = {"id": item_id, "title": title, "content": content}
    except Exception as e:
        app.logger.error(f"Error updating news: {e}")
        conn.rollback()
        return jsonify({"error": str(e)}), 500
    finally:
        if conn:
            conn.close()

    return jsonify(updated_item)

@app.route("/news/<int:item_id>", methods=["DELETE"])
def delete_news(item_id: int):
    conn = get_db_connection()
    if conn is None:
        return jsonify({"error": "Database connection failed"}), 500

    try:
        with conn.cursor() as cur:
            cur.execute(f"DELETE FROM news WHERE id = {item_id} RETURNING id;")
            deleted = cur.fetchone()
            if not deleted:
                abort(404)
```

```
            conn.commit()
    except Exception as e:
        app.logger.error(f"Error deleting news: {e}")
        conn.rollback()
        return jsonify({"error": str(e)}), 500
    finally:
        if conn:
            conn.close()

    return jsonify({"status": "deleted", "id": item_id})

if __name__ == "__main__":
    app.run(threaded=True, host='0.0.0.0', port=3000)
```

app.py (Database Version)

Task: Push and Automate API Image

- Commit and push your fully migrated `app.py` file to your API repository.
- Verify the GitHub Action runs and publishes this final version.

Part 5: Deploying the Full Persistent Application

We will now deploy the final versions of our containers.

5.1) Run the Database Container

First, start your custom database container. (Stop and remove any old `news-db` containers first).

Important: You must replace `YOUR_GITHUB_USERNAME` with your actual GitHub username.

```
export DB_PASS="my-super-secret-password-123"
export DB_USER="news_user"
export DB_NAME="news_db"

docker pull ghcr.io/YOUR_GITHUB_USERNAME/my-db:latest

docker run -d \
    --name news-db \
    -e POSTGRES_PASSWORD=$DB_PASS \
    -e POSTGRES_USER=$DB_USER \
    -e POSTGRES_DB=$DB_NAME \
    ghcr.io/YOUR_GITHUB_USERNAME/my-db:latest
```

Check that it's running with `docker ps`.

5.2) Find the Database IP and Run the API Container

Now, find the IP address of the `news-db` container.

```
export DB_HOST_IP=$(docker inspect -f '{{range .NetworkSettings.Networks}}{{.
    IPAddress}}{{end}}' news-db)
echo "Database IP is: $DB_HOST_IP"
```

With the IP, run your final API container. (Stop/remove old `news-api` containers. Replace `YOUR_API_REPO_NAME`.)

```
docker pull ghcr.io/YOUR_GITHUB_USERNAME/YOUR_API_REPO_NAME:latest

docker run -d \
    --name news-api \
    -p 3000:3000 \
    -e DB_HOST=$DB_HOST_IP \
    -e DB_NAME=$DB_NAME \
    -e DB_USER=$DB_USER \
    -e DB_PASS=$DB_PASS \
    ghcr.io/YOUR_GITHUB_USERNAME/YOUR_API_REPO_NAME:latest
```

Check `docker ps` again. You should see both containers running.

Task: Test Persistence

Test your fully migrated API with `curl`.

```
curl -X POST http://localhost:3000/news -H "Content-Type: application/json" \
 -d '{"title": "Persistence Test", "content": "This should survive."}' | jq

curl "http://localhost:3000/news" | jq
```

Now, for the real test. Stop and **remove** the `news-api` container (but not the `news-db` container).

```
docker stop news-api
docker rm news-api
\end{D_HOST_IP} \
    -e DB_NAME=$DB_NAME \
    -e DB_USER=$DB_USER \
    -e DB_PASS=$DB_PASS \
    ghcr.io/YOUR_GITHUB_USERNAME/YOUR_API_REPO_NAME:latest
```

Once it's running, list the items again:

```
curl "http://localhost:3000/news" | jq
```

Does your "Persistence Test" item still exist? (It should!)