Custom ELT project explained from basics

If you're new to tech, think of this project as a small factory that copies data from one place to another and prepares it for future use. It uses three tools working together: Docker (to run isolated mini-computers), PostgreSQL (a database), and a Python script (the worker that does the copying).

Foundations and key concepts

- **ELT vs ETL:**
  - **ETL (Extract, Transform, Load):** Pull data out, change/clean it, then load it into the destination.
  - **ELT (Extract, Load, Transform):** Pull data out, load it as-is into the destination, and transform later inside the destination database. Most modern data platforms prefer ELT because databases are powerful and can handle transformations efficiently.
- **Database (PostgreSQL):**
  - **What it is:** A structured place to store information (tables, rows, columns).
  - **Why PostgreSQL:** Reliable, widely used, free, supports SQL (the language for working with data).
- **Docker:**
  - **What it is:** A way to package an app (and all it needs) into a container, so it runs the same everywhere.
  - **Why it's useful here:** You don't install PostgreSQL or Python manually. Docker "spins up" containers that already have these tools with the exact configuration you define.
- **Docker Compose:**
  - **What it is:** A single file and command to start multiple containers that need to work together (like this whole mini-factory).
  - **Why it's useful:** One command manages the source database, destination database, and the ELT worker.

Repository structure and what each file does

- **docker-compose.yaml**
  - **Purpose:** Tells Docker how to start and connect three services:
    - **source_postgres:** The original database holding sample data.
    - **destination_postgres:** The empty database where we'll copy data.

    - **elt_script:** A container that runs the Python script which performs the ELT process.

- **elt_script/Dockerfile**
    - **Purpose:** Defines a container image for the Python environment. It:
        - **Installs Python and PostgreSQL client tools** (so the script can run system commands like pg_dump and psql).
        - **Copies the ELT Python script** into the container.
        - **Sets the default command** to run the script when the container starts.

- **elt_script/elt_script.py**
    - **Purpose:** Performs the ELT process. It:
        - **Waits** until the source database is ready and accepting connections.
        - **Extracts** data from the source by running `pg_dump` (creates a SQL file that represents the source data and schema).
        - **Loads** the data into the destination by running `psql` to execute that SQL file against the destination.
        - Optionally, you can add **Transform** steps later inside the destination database using SQL.

- **source_db_init/init.sql**
    - **Purpose:** Initializes the source database with sample tables and data:
        - **Creates tables**: users, films, film categories, actors, film_actors.
        - **Inserts sample rows** so you have realistic data to test the ELT process.

How the system works (step-by-step)

1) Start multiple containers with one command

- **What happens when you run `docker-compose up`:**
    - **source_postgres** starts a PostgreSQL database and runs `init.sql` to create tables and insert sample data.
    - **destination_postgres** starts an empty PostgreSQL database ready to receive data.
    - **elt_script** starts a Python process that will orchestrate copying data from source to destination.

2) The ELT process inside the Python script

- **Wait for readiness:**
  - The script keeps trying to connect to the source database until it's ready. Databases can take a few seconds to boot.
- **Extract (pg_dump):**
  - It runs `pg_dump` against the source database, producing a `.sql` file. This file is a full recipe: create the tables and insert all the data.
- **Load (psql):**
  - It runs `psql` against the destination database with that `.sql` file, recreating the tables and inserting all the data in the destination.
- **Transform (later):**
  - You can add SQL scripts in the destination to change data types, normalize tables, or aggregate metrics. ELT encourages doing this inside the destination database.

3) Accessing the databases

- **Ports:**
  - The source PostgreSQL is exposed at port **5433**.
  - The destination PostgreSQL is exposed at port **5434**.
- **Connecting with a client (e.g., psql, DBeaver):**
  - Host: `localhost`
  - Source: port 5433; Destination: port 5434
  - Use the username/password configured in `docker-compose.yaml` (often `postgres`/`postgres` unless changed).

The important files (sample content and explanations)

docker-compose.yaml (annotated outline)

Yaml Copy

```yaml
version: "3.9"
services:
  source_postgres:
    image: postgres:15
    container_name: source_postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
```

```
      POSTGRES_DB: sourcedb
    ports:
      - "5433:5432"            # expose container's 5432 as host 5433
    volumes:
      - ./source_db_init:/docker-entrypoint-initdb.d
      # Everything in this folder runs at startup; init.sql seeds the data.

  destination_postgres:
    image: postgres:15
    container_name: destination_postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: destdb
    ports:
      - "5434:5432"            # expose container's 5432 as host 5434

  elt_script:
    build: ./elt_script        # uses the Dockerfile in elt_script/
    container_name: elt_runner
    depends_on:
      - source_postgres
      - destination_postgres
    environment:
      SRC_HOST: source_postgres
      SRC_PORT: 5432
      SRC_DB: sourcedb
      SRC_USER: postgres
      SRC_PASSWORD: postgres
      DEST_HOST: destination_postgres
      DEST_PORT: 5432
      DEST_DB: destdb
      DEST_USER: postgres
      DEST_PASSWORD: postgres
    # The script uses these env vars to connect to both databases.
```

elt_script/Dockerfile (annotated)

Dockerfile Copy

```
# Use a lightweight Python base image
FROM python:3.11-slim

# Install PostgreSQL client tools (pg_dump, psql)
RUN apt-get update && apt-get install -y postgresql-client && rm -rf /var/lib/apt/lists

# Copy the ELT script into the image
WORKDIR /app
COPY elt_script.py /app/elt_script.py
```

```
# Optional: install Python dependencies if the script uses libraries
# COPY requirements.txt /app/requirements.txt
# RUN pip install -r requirements.txt

# Default command: run the ELT script
CMD ["python", "/app/elt_script.py"]
```

elt_script/elt_script.py (annotated outline)

Python Copy

```python
import os
import time
import subprocess

# Read connection details from environment variables (set in docker-compose)
SRC = {
    "host": os.getenv("SRC_HOST", "source_postgres"),
    "port": os.getenv("SRC_PORT", "5432"),
    "db":   os.getenv("SRC_DB", "sourcedb"),
    "user": os.getenv("SRC_USER", "postgres"),
    "pwd":  os.getenv("SRC_PASSWORD", "postgres"),
}
DEST = {
    "host": os.getenv("DEST_HOST", "destination_postgres"),
    "port": os.getenv("DEST_PORT", "5432"),
    "db":   os.getenv("DEST_DB", "destdb"),
    "user": os.getenv("DEST_USER", "postgres"),
    "pwd":  os.getenv("DEST_PASSWORD", "postgres"),
}

dump_file = "/app/dump.sql"

def wait_for_source():
    # Keep attempting a simple psql command until the source DB responds
    while True:
        try:
            subprocess.run(
                ["psql",
                 f"postgresql://{SRC['user']}:{SRC['pwd']}@{SRC['host']}:{SRC['port']}/
                 "-c", "SELECT 1;"],
                check=True)
            print("Source DB is ready.")
            return
        except subprocess.CalledProcessError:
            print("Waiting for source DB...")
            time.sleep(2)
```

```
def extract():
    # pg_dump exports schema + data into a single SQL file
    subprocess.run(
        ["pg_dump",
         f"postgresql://{SRC['user']}:{SRC['pwd']}@{SRC['host']}:{SRC['port']}/{SRC['db
         "-Fc", "-f", dump_file],  # custom format (-Fc) or use plain SQL without -Fc
        check=True
    )

def load():
    # For -Fc (custom format), use pg_restore; for plain SQL, use psql
    # Example for plain SQL:
    subprocess.run(
        ["psql",
         f"postgresql://{DEST['user']}:{DEST['pwd']}@{DEST['host']}:{DEST['port']}/{DES
         "-f", dump_file],
        check=True
    )

if __name__ == "__main__":
    wait_for_source()
    # If you used -Fc in extract, replace load() with pg_restore steps:
    # subprocess.run(["pg_restore", ...], check=True)
    extract()
    load()
    print("ELT complete.")
```

source_db_init/init.sql (annotated outline)

Sql

Copy

```
-- Create tables
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT UNIQUE NOT NULL
);

CREATE TABLE films (
    film_id SERIAL PRIMARY KEY,
    title TEXT NOT NULL,
    release_year INT
);

CREATE TABLE film_categories (
    category_id SERIAL PRIMARY KEY,
    category_name TEXT NOT NULL
);
```

```
CREATE TABLE actors (
    actor_id SERIAL PRIMARY KEY,
    actor_name TEXT NOT NULL
);

CREATE TABLE film_actors (
    film_id INT REFERENCES films(film_id),
    actor_id INT REFERENCES actors(actor_id),
    PRIMARY KEY (film_id, actor_id)
);

-- Insert sample data
INSERT INTO users (name, email) VALUES
('Alice', 'alice@example.com'),
('Bob',   'bob@example.com');

INSERT INTO films (title, release_year) VALUES
('Inception', 2010),
('Spirited Away', 2001);

INSERT INTO film_categories (category_name) VALUES
('Sci-Fi'),
('Animation');

INSERT INTO actors (actor_name) VALUES
('Leonardo DiCaprio'),
('Rumi Hiiragi');

INSERT INTO film_actors (film_id, actor_id) VALUES
(1, 1),
(2, 2);
```

Getting started: step-by-step for non-technical users

- **Prerequisites:**
  - **Install Docker Desktop** (Windows/macOS/Linux). On Windows 11, ensure WSL2 is updated.
  - **Install Docker Compose** (bundled with Docker Desktop).
- **Steps to run the project:**
  1. **Clone the repository:**
     - Open a terminal and run: `git clone <repo-url>` then `cd <repo-folder>`.
  2. **Start the system:**
     - Run: `docker-compose up` (add `-d` to run in background).

    - Docker downloads images and starts three containers.
3. **Wait until ready:**
    - The logs will show the source database initializing, then the ELT script running.
4. **Access the databases:**
    - Source DB: `localhost:5433`
    - Destination DB: `localhost:5434`
    - Use a database client (e.g., DBeaver or psql) with user/password from docker-compose.
5. **Stop everything:**
    - Run: `docker-compose down` (keeps volumes unless you add `--volumes`).

Why this is valuable for aspiring data engineers

- **Realistic workflow:** You learn how to orchestrate multiple services (two databases + a script) and make them talk to each other.
- **Reproducibility:** Anyone can run the exact same environment with one command—critical for collaboration.
- **Separation of concerns:** Source stays independent from destination; transformations happen later, closer to analytics needs.
- **Foundation for growth:** You can extend this with scheduling (Airflow), transformations (dbt or SQL), monitoring, and testing.

Common questions and simple answers

- **"What does 'spin up' mean?"**
    - **Start a container/service quickly** so it's up and running.
- **"Why not just export/import manually?"**
    - Manual steps are error-prone and hard to reproduce. ELT scripting makes it consistent and automatable.
- **"What if the source database isn't ready?"**
    - The script waits and retries until it is. This avoids failures from race conditions.
- **"Where is the data stored?"**

    - Inside each PostgreSQL container. You can persist data using Docker volumes if needed (configure in docker-compose).

- **"Can I transform data afterward?"**
    - Yes. Run SQL in the destination or add a transformation step after load.