**Lab: Streaming to PostgreSQL with Python**

**Goal:** Build a resilient Python streaming application that attempts to write data to a PostgreSQL database, automatically retries on transient failures, and routes persistently failing messages to a Dead-Letter Queue (DLQ).

# **Purpose of the Lab**

This lab simulates a common real-world scenario where a custom application needs to interact with a downstream system (like a database) that may be temporarily unavailable. You will write a Python application that implements its own resilience logic to handle these failures gracefully. By first simulating a failure and observing the retry/DLQ behavior in your code, and then fixing the condition to see the successful outcome, you will gain practical experience in building fault-tolerant streaming applications.

# **Prerequisites**

* A Redpanda Cloud account with a running cluster and an rpk profile (e.g., rpk-cloud).
* Docker and Docker Compose installed on your local machine.
* Python 3 installed on your local machine.

# **Project Layout**

|  |
| --- |
| rp-python-postgres-lab/ ├── user\_profiles.jsonl ├── postgres\_sink.py ├── docker-compose.yml └── .env |

# **Part 1: Setting up a Local Database**

## Step 1: Prepare the Project

1. **Create the project directory:**

|  |
| --- |
| mkdir rp-python-postgres-lab cd rp-python-postgres-lab |

1. **Create a Docker Compose file for PostgreSQL:** Create docker-compose.yml.  
   **docker-compose.yml**

|  |
| --- |
| services:  postgres:  image: postgres:14  container\_name: postgres\_db  environment:  POSTGRES\_USER: redpanda\_user  POSTGRES\_PASSWORD: redpanda\_password  POSTGRES\_DB: redpanda\_db  ports:  - "5432:5432" |

1. **Start the database:**

|  |
| --- |
| docker compose up -d |

# **Part 2: Simulating and Handling Failures**

## **Step 2: Prepare Redpanda and the Python Environment**

1. **Get Cloud Credentials:** Create a new user (e.g., resilience-user) in the **Security -> Users** tab of the Redpanda Cloud UI and grant it **Allow All** permissions in the **ACLs** tab. Save the **Username**, **Password**, and your cluster's **Broker Address**.
2. **Create the topics on Redpanda Cloud:**

|  |
| --- |
| rpk topic create user-profiles-py --profile rpk-cloud rpk topic create user-profiles-py-dlq --profile rpk-cloud |

1. **Create a sample data file named user\_profiles.jsonl:  
   user\_profiles.jsonl**

|  |
| --- |
| {"profile\_id": 101, "email": "alice@example.com", "country": "US"} {"profile\_id": 102, "email": "bob@corp.com", "country": "CA"} |

1. **Set up the Python virtual environment:**

|  |
| --- |
| python3 -m venv venv source venv/bin/activate |

pip install kafka-python python-dotenv certifi psycopg2-binary python-snappy

1. **Create the Environment File (.env):**

|  |
| --- |
| # .env REDPANDA\_BROKERS="<YOUR\_BROKERS\_URL>" REDPANDA\_USER="<YOUR\_USERNAME>" REDPANDA\_PASS="<YOUR\_PASSWORD>" POSTGRES\_HOST="localhost" POSTGRES\_USER="redpanda\_user" POSTGRES\_PASSWORD="redpanda\_password" POSTGRES\_DB="redpanda\_db" |

Populate the file with your credentials.

1. **Produce the data to the user-profiles-py topic:**

|  |
| --- |
| rpk topic produce user-profiles-py --profile rpk-cloud < user\_profiles.jsonl |

## **Step 3: Create the Resilient Python Application**

Create a file named postgres\_sink.py. This script will consume from Redpanda and try to insert into PostgreSQL. **For our first run, we will not create the table in the database, forcing a failure.**

**postgres\_sink.py**

|  |
| --- |
| # postgres\_sink.py  import json, os, certifi, time  from kafka import KafkaConsumer, KafkaProducer  from dotenv import load\_dotenv  import psycopg2  # Load environment variables from .env file  load\_dotenv()  # --- Configuration ---  # Redpanda Config  BROKER\_URL = os.getenv("REDPANDA\_BROKERS")  USERNAME = os.getenv("REDPANDA\_USER")  PASSWORD = os.getenv("REDPANDA\_PASS")  SOURCE\_TOPIC = "user-profiles-py"  DLQ\_TOPIC = "user-profiles-py-dlq"  # Postgres Config  PG\_HOST = os.getenv("POSTGRES\_HOST")  PG\_USER = os.getenv("POSTGRES\_USER")  PG\_PASS = os.getenv("POSTGRES\_PASSWORD")  PG\_DB = os.getenv("POSTGRES\_DB")  # --- End Configuration ---  print("Starting resilient Python sink...")  print("Press Ctrl+C to stop.")  try:  consumer = KafkaConsumer(  SOURCE\_TOPIC,  bootstrap\_servers=BROKER\_URL, security\_protocol="SASL\_SSL",  sasl\_mechanism="SCRAM-SHA-256", sasl\_plain\_username=USERNAME, sasl\_plain\_password=PASSWORD,  group\_id="python-postgres-sink-group", auto\_offset\_reset="earliest",  ssl\_cafile=certifi.where(), api\_version=(2, 0, 2),  value\_deserializer=lambda v: json.loads(v.decode('utf-8')),  enable\_auto\_commit=False # <-- IMPORTANT: Disable auto-commit  )  producer = KafkaProducer(  bootstrap\_servers=BROKER\_URL, security\_protocol="SASL\_SSL",  sasl\_mechanism="SCRAM-SHA-256", sasl\_plain\_username=USERNAME, sasl\_plain\_password=PASSWORD,  ssl\_cafile=certifi.where(), api\_version=(2, 0, 2)  )    pg\_conn = psycopg2.connect(host=PG\_HOST, user=PG\_USER, password=PG\_PASS, dbname=PG\_DB)  pg\_cursor = pg\_conn.cursor()  for message in consumer:  payload = message.value  print(f"Received message: {payload}")    max\_retries = 3  for attempt in range(max\_retries):  try:  # Attempt to insert into the database  pg\_cursor.execute(  "INSERT INTO user\_profiles (profile\_id, email, country) VALUES (%s, %s, %s)",  (payload['profile\_id'], payload['email'], payload['country'])  )  pg\_conn.commit()  consumer.commit() # <-- IMPORTANT: Manually commit offset after success  print(f"Successfully inserted message for profile\_id {payload['profile\_id']}")  break # Exit the retry loop on success  except psycopg2.Error as e:  print(f"Attempt {attempt + 1}/{max\_retries} failed: {e}")  pg\_conn.rollback() # Rollback the failed transaction  if attempt < max\_retries - 1:  time.sleep(1) # Wait before retrying  else:  # If all retries fail, send to DLQ  print(f"All retries failed. Sending message for profile\_id {payload['profile\_id']} to DLQ.")  producer.send(DLQ\_TOPIC, value=json.dumps(payload).encode('utf-8'))  consumer.commit() # Commit offset even for DLQ'd messages to avoid reprocessing  except KeyboardInterrupt:  print("Handler stopped by user.")  except Exception as e:  print(f"An unexpected error occurred: {e}")  finally:  if 'consumer' in locals(): consumer.close()  if 'producer' in locals(): producer.close()  if 'pg\_conn' in locals(): pg\_conn.close()  print("Connections closed.") |

## **Step 4: Create the Database Table**

1. Stop the Python script with Ctrl+C in the first terminal.
2. Now, create the target table in the PostgreSQL database.

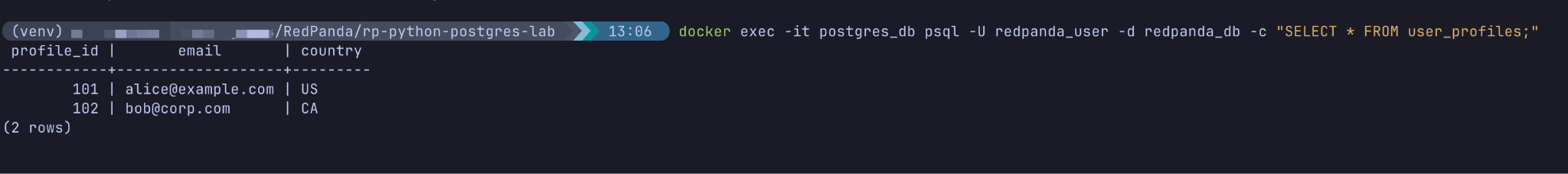
|  |
| --- |
| docker exec -it postgres\_db psql -U redpanda\_user -d redpanda\_db -c "CREATE TABLE user\_profiles (profile\_id INT PRIMARY KEY, email VARCHAR(255), country VARCHAR(2));" |

## **Step 6: Re-run the Application and Verify**

1. Run the application again. Because we are now manually committing offsets, it will correctly re-consume the original messages from the user-profiles-py topic.

|  |
| --- |
| python postgres\_sink.py |

1. **Observe the logs:** This time, you should see messages indicating successful insertion.
2. **Open a new terminal window.**
3. **Verify in the database:** Connect to the database and select the data.

**docker exec -it postgres\_db psql -U redpanda\_user -d redpanda\_db -c "SELECT \* FROM user\_profiles;"**  
  
**Expected output:** You should see the two user records now successfully inserted into the table.  
**

# **Cleanup**

1. Stop the script with Ctrl+C.
2. Stop and remove the database container: docker compose down.
3. Delete the Redpanda topics:

|  |
| --- |
| rpk topic delete user-profiles-py user-profiles-py-dlq --profile rpk-cloud |

1. Deactivate the virtual environment.

|  |
| --- |
| deactivate |