📄 Polyglot Networked Services Project Documentation

Polyglot Networked Services Project

This document details the architecture, implementation, and functionality of a Docker Compose project demonstrating a microservices pattern with language interoperability (Polyglot Persistence).

🌟 1. Project Overview & Objectives

The primary objective is to build a containerized environment where different backend services, written in fundamentally different languages (Python and C), can successfully and independently interact with a single, shared data source (MySQL).

Key Features Demonstrated:

- **Microservices Design:** Each server and client is a separate, dedicated container (service).
- **Polyglot Backend:** Successful database access and business logic execution in both Python and C.
- **Concurrency:** Use of threads (Python) and Pthreads (C) for autonomous background tasks.
- **Data Integrity:** Maintaining a single source of truth across diverse applications.

📐 2. Architectural Components

The entire system is defined and orchestrated by the docker-compose.yml file, placing all services on a single, internal Docker network.

Service	Technology	Port	Primary Role	User Handled
database	MySQL 8.0	3306 (Internal)	Centralized data persistence.	N/A
python-server	Python 3 + Flask/Socket	5001	Handles periodic updates and client requests.	python_user

Service	Technology	Port	Primary Role	User Handled
c-server	C (Native Sockets)	5000	Handles periodic updates and client requests.	c_user
python-client	Python 3 Client	N/A	Requests data from python-server.	N/A
c-client	C Client	N/A	Requests data from c-server.	N/A

Database Schema

The users table is the core entity, tracking two users handled by the separate backend servers.

Column	Type	Description
user	VARCHAR(50) (PK)	The username (python_user or c_user).
points	INT	The current points total (incremented every 30s).
datetime_stamp	DATETIME	Timestamp of the last update.

⚙️ 3. Implementation Details

A. The Server Logic (30-Second Update)

Both services implement identical logic in a dedicated background process:

Service	Concurrency Method	Database Access Method	Task
python-server	Python Threading	mysql-connector-python Library	UPDATE users SET points = points + 1 ... WHERE user = 'python_user';
c-server	POSIX Thread (pthread)	**External Shell Execution** (system())	Executes the mysql command-line client to run the UPDATE query.

The C server's use of system() is a critical point, as it successfully substitutes complex native database libraries with a robust shell command pattern.

B. Client Request Handling (Fetching Data)

When a client connects, the server performs a SELECT query to retrieve the current data.

The C Server's Parsing Solution:

Due to complex interactions between the C runtime and the shell output, reading data was challenging. The final, robust solution involves:

1. Executing a mysql command that uses CONCAT_WS to explicitly join points and timestamp with a reliable delimiter (e.g., a pipe |).

SELECT CONCAT_WS('|', points, DATE_FORMAT(datetime_stamp, '%%Y-%%m-%%d %%H:%%i:%%s')) ...

2. The C code uses file I/O (fgets) and string tokenization (strtok) on the pipe (|) delimiter to reliably separate the integer (points) from the datetime string.

This approach guarantees data integrity and correct logging output regardless of runtime environment variances.

🚀 **4. Deployment and Verification**

Deployment Steps

1. **Prerequisites:** Docker and Docker Compose installed.
2. **Build and Run:**

docker compose up -d --build --force-recreate

Verification (Logs are the Proof)

The seamless integration is visible in the combined server logs, where both services report successful operations.

1. **Server Monitoring (30-Second Updates):**

- docker compose logs -f python-server c-server
- **Expected Log:** [c_user Server] DB updated: Points incremented. (Repeats every 30s)

2. **Client Response (Data Retrieval):** When a client connects, the log confirms successful database access and correct data retrieval, including the accurate timestamp.

- docker compose logs -f c-client
- **Expected Log:** [c_user Server] Sent: User: c_user, Points: [N], Last Update: YYYY-MM-DD HH:MM:SS

HOW TO RUN?

🚀 **Getting Started: Prerequisites and Installation**

Prerequisites

You must have **Docker Desktop** installed, which includes both the Docker Engine and Docker Compose (or Docker Compose V2).

1. Install Docker Desktop

Follow the instructions for your operating system:

- **Windows:** Download and install Docker Desktop for Windows.
- **macOS:** Download and install Docker Desktop for Mac.
- **Linux:** Follow the instructions for your distribution (Docker Engine and Docker Compose).

After installation, ensure Docker Desktop is running. You can verify the installation via your terminal:

docker --version

docker compose version

2. Project Setup and Running

1. **Clone the Repository (Simulated):**

Bash

Replace this with your actual repository steps:

git clone <your-repository-url>

cd <your-repository-name>

2. **Build and Run the Containers:** This command tells Docker Compose to build the necessary images (for the C and Python servers) and start all six services in detached mode (-d). The --force-recreate ensures a fresh start.

Bash

docker compose up -d --build --force-recreate

The services will now be running in the background, starting their periodic database updates.