# Project Documentation: Code Structure and Implementation Decisions

### Augusto Bastos 10/05/2024

## 1. Overview

This document explains the structure of the project codebase, decisions made during the implementation, the use of Redpanda on Docker, and the coding best practices that were followed.

## 2. Project Structure

The project's structure is designed to follow a clean, modular approach where each component is isolated for better maintainability. Below is the breakdown of the key files and directories:

```
.
├── Dockerfile
├── docker-compose.yml
├── requirements.txt
├── producer.py
├── consumer.py
├── init.py
├── test_docker_compose.py
└── README.md
```

### 2.1 Key Files

- **`Dockerfile`**: This file defines the Docker image and specifies the environment required to run the application, including the installation of dependencies like Python libraries and system utilities.
- **`docker-compose.yml`**: This YAML file orchestrates the multi-container Docker environment, defining services like PostgreSQL, Redpanda, and the application itself.
- **`requirements.txt`**: Lists the Python dependencies required for the application.
- **`producer.py`**: The producer service fetches real-time cryptocurrency data from the CoinGecko API and sends it to Redpanda.
- **`consumer.py`**: The consumer reads data from Redpanda and stores it in a PostgreSQL database.
- **`init.py`**: Handles any initialization logic before the main application starts, such as setting up the database schema.
- **`test_docker_compose.py`**: This file runs automated tests using Pytest to verify that the Docker environment and application components are functioning as expected.
- **`README.md`**: The documentation file that provides setup instructions, usage guidelines, and troubleshooting tips.

## 3. Why Use Redpanda in Docker?

Redpanda was chosen as the event streaming platform for the following reasons:

1. **Kafka Compatibility**: Redpanda is a Kafka-compatible event streaming platform but without the complexity and overhead of running a Kafka cluster. This makes it easier to set up and maintain.
2. **Performance**: Redpanda offers high performance in streaming data and is more resource-efficient compared to traditional Kafka setups.
3. **Docker Convenience**: Docker enables an isolated, repeatable environment that makes it easier to deploy Redpanda alongside other services like PostgreSQL. It eliminates dependency management issues and ensures consistency across different environments (development, staging, production).

### 3.1 Redpanda on Docker

- **Isolation**: By containerizing Redpanda, it runs independently of other services, ensuring that issues in one component don't affect the other.
- **Networking**: Redpanda communicates with the other containers (e.g., the Python app and PostgreSQL) over a Docker network. This allows for smooth integration and seamless data flow.
- **Resource Efficiency**: The `redpanda` image is configured to use minimal resources (`--overprovisioned --smp 1 --memory 1G`), which is critical for local development or small-scale applications.

## 4. How Things Were Built

### 4.1 Docker Setup

- **Dockerfile**: The Dockerfile builds a slim Python image, installs necessary system and Python dependencies, and defines a default command for running the producer service.
- **docker-compose.yml**: Docker Compose manages the services for PostgreSQL, Redpanda, and the app, making it easy to orchestrate the full stack.

### 4.2 Automated Testing

- Before running the actual application, tests are automatically executed via `pytest` to ensure that the system is functioning correctly. This step is added to the `docker-compose.yml` file through the following command:

  ```
  command: >
    sh -c "pytest test_docker_compose.py && echo 'Tests passed, running the app' && python init.py"
  ```

  This ensures that the application only runs if the tests pass, preventing runtime issues from going unnoticed.

### 4.3 Producer-Consumer Workflow

- **Producer (`producer.py`)**: This service fetches real-time data from the CoinGecko API, formats the data into a message (JSON), and publishes it to Redpanda.
- **Consumer (`consumer.py`)**: This service listens to the Redpanda stream, reads incoming messages, and writes the data into the PostgreSQL database for storage and analysis.

### 4.4 Data Ingestion Flow

1. The producer fetches OHLCV (Open, High, Low, Close, Volume) data for cryptocurrencies like Bitcoin, Ethereum, and Zcash from the CoinGecko API.
2. The producer sends the fetched data to the Redpanda broker.
3. The consumer listens for new data on Redpanda, processes the data, and inserts it into the PostgreSQL database.

### 4.5 Data Consistency

The project ensures data consistency through:

- **Transactions in PostgreSQL**: Each data insertion operation is wrapped in a transaction, ensuring that data integrity is maintained even in case of failures.
- **Redpanda Acknowledgments**: The consumer acknowledges messages from Redpanda only after successful database insertion, ensuring that no data is lost during processing.

# 5. Coding Best Practices

Several coding best practices were followed to ensure the quality and maintainability of the code:

### 5.1 Modular Code

Each service (producer and consumer) is separated into individual Python scripts. This modular structure allows for easy debugging, testing, and updates. If a future enhancement is needed, changes to one service can be made without affecting others.

### 5.2 Automated Testing

Using `pytest`, we run tests before starting the actual application to ensure that all components (Redpanda, PostgreSQL, and the app) are correctly set up and functioning. This proactive testing approach catches issues early in the development process.

### 5.3 Error Handling

Each component has error handling in place. For example:

- The producer handles API errors, ensuring that if CoinGecko is down, the service will retry.
- The consumer ensures data is only written to PostgreSQL after successful parsing of messages from Redpanda.

## 5.4 Environment Variables

Sensitive data like PostgreSQL credentials are stored in a `.env` file, and environment variables are used to access them. This ensures security by keeping credentials out of the codebase.

```
POSTGRES_USER=your_postgres_user
POSTGRES_PASSWORD=your_postgres_password
POSTGRES_DB=your_database_name
POSTGRES_HOST=postgres
KAFKA_BROKER=redpanda:9092
```

## 5.5 Dockerized Development

By containerizing the entire stack, the project achieves platform independence. Developers don't have to worry about installing specific versions of PostgreSQL or setting up Redpanda manually. Docker ensures that everything works as intended regardless of the development environment.

## 5.6 Documentation

Comprehensive documentation is provided, including this document, the `README.md` file, and code comments. Each module, its purpose, and its integration with the rest of the system are clearly explained, making it easier for other developers to contribute to or maintain the project.

### 6. Data Export Strategy

### 6.1 Manual and Automatic Data Export

Data export was designed to ensure that no valuable data is lost after the Docker container shuts down. The decision to export data both manually and automatically upon shutdown was made to:

- **Ensure Data Integrity**: Avoid the risk of data loss during development or unexpected container crashes.
- **Backup Critical Data**: Exported files (CSV and Parquet) serve as backups that can be used for future data recovery or analysis.

**Manual Data Export Command:**

To manually export the data:

bash

Copiar código

```
docker-compose run export_data
```

This command triggers the export_data.py script, which exports:

- **CSV File for OHLCV Daily Data**: The daily open, high, low, close, and volume data.
- **Parquet File for Minute-Level Data**: The cryptocurrency price data at the minute level.

**Automatic Export on Container Shutdown:**

Data is automatically exported when the container shuts down, ensuring that no data is lost during system shutdown. Files are saved to the data_export directory, which is mounted locally.

## 6. Conclusion

This project demonstrates a robust architecture for real-time cryptocurrency data ingestion using Redpanda and PostgreSQL, leveraging Docker for containerization and pytest for automated testing. The decision to use Redpanda was driven by its Kafka compatibility, performance, and resource efficiency, while the overall design ensures modularity, maintainability, and security.

By following coding best practices like modularity, error handling, and automated testing, this project ensures that the system is both reliable and easy to maintain in production environments.