Integrating Redis as a Cache for RabbitMQ Log Streaming to PostgreSQL

Tomasz Olejarczuk

21-06-2024

Contents

Integrating Redis as a Cache for RabbitMQ Log Streaming to PostgreSQL

1. Summary

This document details the implementation of Redis as a cache layer between RabbitMQ log streaming and PostgreSQL log storage within the Teamium project. Redis, a high-performance in-memory data store, is used to buffer log messages received from RabbitMQ before they are asynchronously synchronized with the PostgreSQL database. This approach improves the system's overall performance, reduces the load on the database, and ensures more efficient log handling.

2. Introduction

2.1. Background

The Teamium project utilizes RabbitMQ to transmit log messages generated by the MATLAB server in real time. These logs are then consumed by the API wrapper and stored in a PostgreSQL database. While this architecture works well, it can lead to performance bottlenecks when the volume of log messages is high, as direct writes to the database can become a limiting factor.

To address this, Redis is introduced as a cache. Log messages are first published to RabbitMQ, then consumed by the API wrapper and stored in Redis. A background service periodically reads these messages from Redis and writes them to PostgreSQL, effectively offloading the database and improving overall system responsiveness.

2.2. Objectives

The primary objectives of integrating Redis as a cache are:

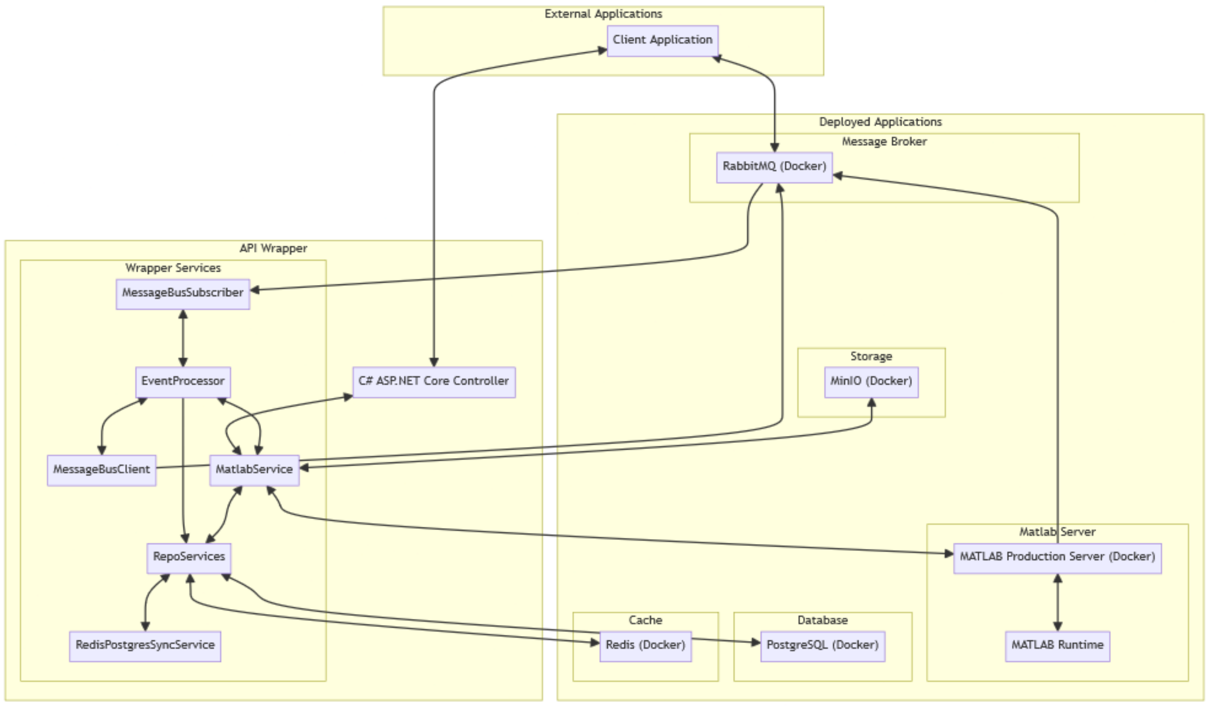
1. **Performance Optimization:** Improve the system's ability to handle a high volume of log messages by reducing the frequency of database writes.
2. **Database Load Reduction:** Decrease the load on the PostgreSQL database, allowing it to focus on other critical tasks.
3. **Fault Tolerance:** Provide a buffer for log messages in case the PostgreSQL database is temporarily unavailable.

3. Methodology

3.1. Architecture

The revised architecture incorporates Redis as a cache between RabbitMQ and PostgreSQL:

1. **MATLAB Logging:** Log messages are published to a RabbitMQ exchange.
2. **RabbitMQ Routing:** The exchange routes messages to a queue.
3. **API Wrapper Consumption and Caching:** The API wrapper consumes messages from the queue and stores them in Redis.
4. **Redis-PostgreSQL Synchronization:** A background service periodically reads messages from Redis and writes them to PostgreSQL.



3.2. Implementation

- **Redis Configuration:**

  - The Docker Compose file was updated to include a Redis container.
  - A Redis client library (StackExchange.Redis) was added to the C# API wrapper project.

- **API Wrapper (MessageBusSubscriber):**
  - Modified to store received log messages in Redis using a list data structure.

- **Redis-PostgreSQL Sync Service (`RedisPostgresSyncService`):**
  - Implements a background service that runs periodically (e.g., every minute).
  - Retrieves log messages from Redis in batches.
  - Writes the retrieved messages to the PostgreSQL database using Entity Framework Core.

4. Results & Insights

4.1. Key Outcomes

- **Improved Performance:** The introduction of Redis as a cache significantly improved the system's ability to handle a high volume of log messages. The API wrapper can quickly store messages in Redis without waiting for database writes, leading to faster response times and reduced latency.
- **Reduced Database Load:** By batching log writes to PostgreSQL, the load on the database was reduced, freeing up resources for other operations and improving overall system performance.
- **Reliable Log Persistence:** The asynchronous synchronization mechanism ensures that log messages are reliably persisted to PostgreSQL, even if there are temporary network issues or database unavailability.

5. Insights and Conclusion

The integration of Redis as a cache for RabbitMQ log streaming has proven to be a highly effective solution for optimizing log handling in the Teamium project. By decoupling log ingestion from database writes, we have achieved a more scalable, resilient, and performant system. This enhancement not only addresses current performance bottlenecks but also lays the groundwork for future expansion and the ability to handle even larger volumes of log data. The success of this integration highlights the importance of carefully considering caching strategies in distributed systems, especially when dealing with high-throughput data streams. By choosing the right tools and implementing them thoughtfully, we can significantly improve system performance and ensure a seamless user experience.