Message Broker System

Introduction

This project is a hands-on implementation of an **event-driven system** using a **message broker** (RabbitMQ). The system is designed to process messages through a series of services, each performing a specific task, and demonstrates the use of microservices and asynchronous communication.

System Architecture

The system consists of four services:

1. **API Service**: A user-facing REST API that receives POST requests containing a message and a user alias.
2. **Filter Service**: Filters out messages containing specific stop words.
3. **Screaming Service**: Converts the message text to uppercase.
4. **Publish Service**: Sends the processed message via email to a specified recipient list.

Message Flow:

- The API Service receives messages and publishes them to the `filter_queue`.
- The Filter Service consumes messages from `filter_queue`, filters out unwanted messages, and publishes valid messages to `screaming_queue`.
- The Screaming Service consumes messages from `screaming_queue`, converts the text to uppercase, and publishes them to `publish_queue`.
- The Publish Service consumes messages from `publish_queue` and sends emails to the recipients.

Project Structure

```
message_broker_system/
├── .env
├── .env.example
├── docker-compose.yml
├── README.md
├── services/
│   ├── api_service/
│   │   ├── api_service.py
│   │   ├── Dockerfile
│   │   └── requirements.txt
│   ├── filter_service/
│   │   ├── filter_service.py
│   │   ├── Dockerfile
│   │   └── requirements.txt
│   ├── screaming_service/
│   │   ├── screaming_service.py
│   │   ├── Dockerfile
│   │   └── requirements.txt
│   └── publish_service/
```

```
│       ├── publish_service.py
│       ├── Dockerfile
│       └── requirements.txt
├── load_testing/
│   ├── load_test_rabbitmq.py
│   ├── test.py
│   └── report.md
```

Prerequisites

- **Docker** and **Docker Compose** installed on your machine.
- **Python 3.7+** installed for running the load testing script.
- An SMTP server or email testing service like **Mailtrap** for sending emails.
- Optional: **cURL** or **Postman** for testing the API endpoints.

Setup and Installation

1. Clone the Repository

```
git clone <repository-url>
cd message_broker_system
```

2. Configure Environment Variables

Create a `.env` file in the root directory based on `.env.example` file.

Note: Replace the email settings with your actual SMTP server details.

3. Build and Start the Services

```
docker-compose up --build
```

Running the Application

Once the Docker containers are up and running, the services will be accessible as per the configuration.

Access RabbitMQ Management UI (Optional)

You can access the RabbitMQ Management UI at http://localhost:15672 using the default credentials (`guest` / `guest`).

Testing the Application

1. Sending a Message Without Stop Words

Use `curl` or Postman to send a POST request to the API Service.

Using `curl`:

```
curl -X POST -H "Content-Type: application/json" \
-d '{"user_alias": "professor", "message_text": "Services should be separately deployable units!"}' \
http://localhost:5000/message
```

Expected Outcome

- The message passes through the Filter Service.
- The Screaming Service converts the message to uppercase.
- The Publish Service sends an email with the message content.

2. Sending a Message With Stop Words

```
curl -X POST -H "Content-Type: application/json" \
-d '{"user_alias": "student", "message_text": "I love bird-watching"}' \
http://localhost:5000/message
```

Expected Outcome

- The Filter Service detects the stop word (`bird-watching`) and filters out the message.
- No email is sent.

3. Checking the Email

If configured correctly, you should receive an email as per the message sent. If using Mailtrap, you can view the email in your Mailtrap inbox.

Load Testing

Running the Load Testing Script

1. Open a new terminal window.
2. Navigate to the `load_testing/` directory:

   ```
   cd message_broker_system/load_testing
   ```

3. Install the required dependencies:

```
pip install requests
```

4. Run the load testing script for both message brokers and pipes and filters system:

```
python load_test_rabbitmq.py
```

```
python load_test_pipes.py
```

Understanding the Load Test

- **Number of Threads**: The script uses 50 concurrent threads.
- **Requests per Thread**: Each thread sends 100 requests.
- **Total Requests**: 30 threads * 100 requests = 3,000 total requests.
- **Metrics Captured**:
  - Total time taken.
  - Requests per second.
  - Requests sent.

Viewing the Load Test Report

After the test completes, a `report.md` file is generated in the `load_testing/` directory containing the performance metrics.

Performance Report

Performance Comparison Summary

Metric	RabbitMQ-Based System	Pipes-and-Filters System
Workload	3000 text messages	16-second 60fps Full HD video
Total Time Taken	39.89 seconds	12.18 seconds
Throughput	75.20 requests per second	30 pipelines in 12.18 seconds
Overhead	Network + broker communication	Minimal (in-memory only)

1. **Latency**: The pipes-and-filters system achieves lower latency (0.41 seconds per pipeline) due to in-memory operations, while RabbitMQ incurs overhead from broker communication and message serialization.

2. **Throughput**: Pipes-and-filters processes computationally intensive tasks faster, whereas RabbitMQ offers moderate throughput suitable for asynchronous messaging workloads.

3. **Scalability**: RabbitMQ supports distributed scaling across machines, making it ideal for decoupled systems. Pipes-and-filters is limited to the resources of a single machine but excels in high-performance, real-time processing.

Summary

The RabbitMQ-based system is better for distributed and fault-tolerant applications, while the pipes-and-filters system is optimal for CPU-intensive, real-time workloads. The choice depends on workload requirements and system scalability needs.

Troubleshooting

- **RabbitMQ Connection Issues**: Ensure that the RabbitMQ service is running and accessible. The services are configured to retry connections if they fail.
- **Email Sending Issues**: Verify your SMTP server settings and credentials in the `.env` file.
- **Port Conflicts**: Ensure that the ports defined in `docker-compose.yml` are not in use by other applications.
