Tutorial: Building a Resilient Event-Driven Notification System using Hybrid Messaging (Kafka & RabbitMQ) and Serverless Functions

Adrian Soim

1 Introduction

In modern microservices architectures, synchronous communication (blocking HTTP calls) often leads to performance bottlenecks and tight coupling. In the **SOA Task Manager** project, sending an email notification immediately after creating a task would degrade the user experience by forcing the user to wait for the email provider's response.

To solve this, I implemented an **asynchronous, event-driven architecture**. This tutorial demonstrates a hybrid messaging approach where **Apache Kafka** is used for high-throughput domain events (e.g., `TaskUpdated`), and **RabbitMQ** is used for task queues (e.g., `SendEmail`), resulting in a **Serverless function (OpenFaaS)** that handles the actual notification.

2 System Architecture

The system utilizes a "Bridge" pattern implemented within the `TaskService`. The flow of data is as follows:

1. **Producer:** The `TaskService` publishes a domain event to **Kafka** immediately upon data persistence.
2. **Orchestrator:** An internal component listens to Kafka, processes the business logic, and forwards a specific command to **RabbitMQ**.
3. **Consumer:** An **OpenFaaS** Python function, triggered by RabbitMQ, executes the notification logic.

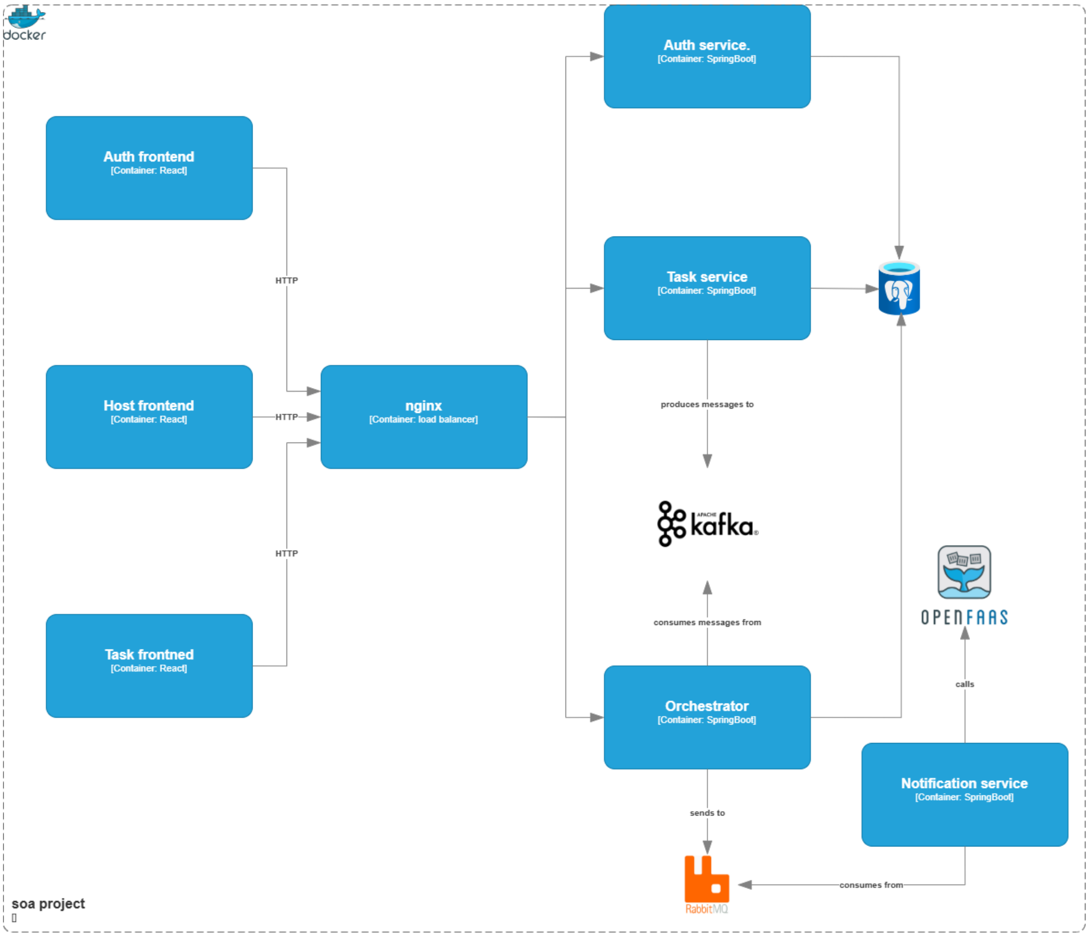


Figure 1: C4 Container Diagram illustrating the hybrid messaging flow.

3 Implementation Guide

3.1 Step 1: The Event Producer (Java Spring Boot)

First, we configure the `TaskController` to decouple the write operation from the notification logic. Instead of calling an email service directly, we inject a `KafkaTemplate` and publish a JSON payload to the `task-events` topic.

```java
@RestController
@RequestMapping("/tasks")
public class TaskController {

    private final KafkaTemplate<String, String> kafkaTemplate;
    private final ObjectMapper objectMapper;

    @PutMapping("/{id}")
    public Task updateTask(@PathVariable Long id, @RequestBody TaskDto dto) {
        Task task = taskService.update(id, dto);

        // Serialize and publish event asynchronously
        try {
            String payload = objectMapper.writeValueAsString(task);
            kafkaTemplate.send("task-events", payload);
            System.out.println("Event sent to Kafka: " + task.getId());
        } catch (Exception e) {
            e.printStackTrace();
        }

        return task;
    }
}
```

Listing 1: TaskController.java - Publishing to Kafka

3.2 Step 2: The Orchestrator Bridge (Kafka to RabbitMQ)

A unique feature of this system is the `Orchestrator` package. It acts as a bridge, allowing us to leverage Kafka for event sourcing (history) while using RabbitMQ for reliable task execution. This component listens to Kafka and, if the task status is `DONE`, pushes a job to RabbitMQ.

```java
@Component
public class TaskOrchestrator {

    private final RabbitTemplate rabbitTemplate;

    @KafkaListener(topics = "task-events", groupId = "orchestrator-group")
    public void handleTaskEvent(String message) {
        try {
            Task task = objectMapper.readValue(message, Task.class);

            // Business Logic: Only notify if task is COMPLETED
            if ("COMPLETED".equals(task.getStatus())) {
                System.out.println("Task DONE. Forwarding to RabbitMQ...");

                // Forward to the queue consumed by OpenFaaS
                rabbitTemplate.convertAndSend("email-jobs", message);
            }
        } catch (Exception e) {
            System.err.println("Error processing event: " + e.getMessage());
        }
    }
}
```

Listing 2: Orchestrator.java - The Bridge Logic

3.3 Step 3: The Serverless Consumer (Python/OpenFaaS)

The final component is a lightweight Python function deployed via OpenFaaS. It is completely decoupled from the Java backend. It receives the JSON payload from RabbitMQ and formats an HTML email.

```python
import json

def handle(req):
    try:
        # Parse the JSON received from RabbitMQ
        data = json.loads(req)
        task_id = data.get("id")
        status = data.get("status")

        # Generate HTML content dynamically
        html_content = f"""
        <html>
            <body>
                <h1>Update for Task #{task_id}</h1>
                <p>Status is now: <b style="color:green">{status}</b></p>
            </body>
        </html>
        """

        # In a real scenario, we would call an SMTP server here
        print(f"Sending email for task {task_id}")
        return html_content

    except Exception as e:
        return f"Error: {str(e)}"
```

Listing 3: handler.py - OpenFaaS Function

4 Deployment and Infrastructure

The entire system is containerized using Docker. The orchestration of the messaging infrastructure is defined in `docker-compose.yml`.

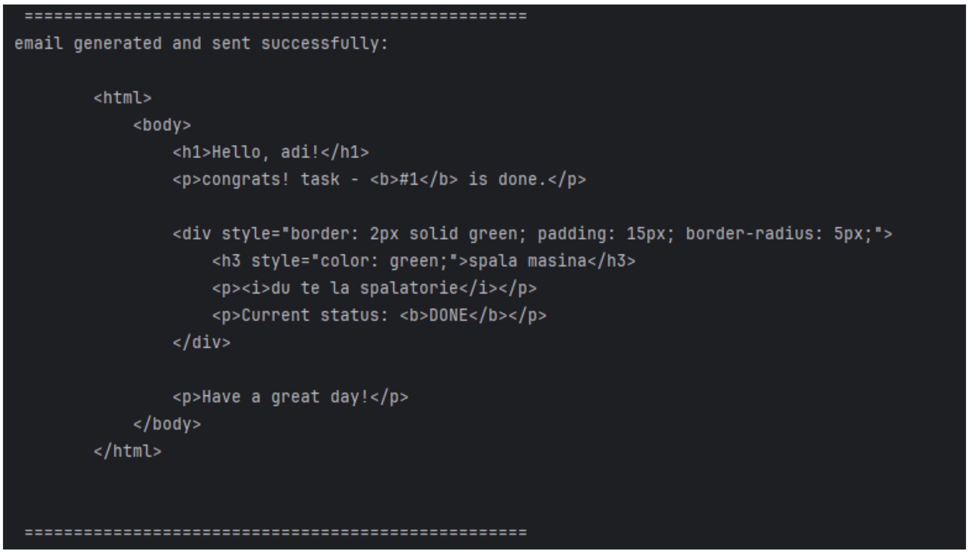
```
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest

  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on: [zookeeper]

  rabbitmq:
    image: rabbitmq:3-management
    ports:
      - "5672:5672"
      - "15672:15672" # Management UI

  gateway: # OpenFaaS Gateway
    image: openfaas/gateway:latest
    ports:
      - "8080:8080"
```

Listing 4: docker-compose.yml (Messaging Infrastructure)

5 Results and Conclusion

By implementing this architecture, we achieved significant system resilience.

- **Decoupling:** The frontend receives a response in milliseconds, regardless of how long the email takes to send.
- **Reliability:** If the OpenFaaS function is down, messages persist in RabbitMQ until the service recovers.
- **Scalability:** We can scale the notification function independently of the main Java monolith.



```
 ===================================================
email generated and sent successfully:

        <html>
            <body>
                <h1>Hello, adi!</h1>
                <p>congrats! task - <b>#1</b> is done.</p>

                <div style="border: 2px solid green; padding: 15px; border-radius: 5px;">
                    <h3 style="color: green;">spala masina</h3>
                    <p><i>du te la spalatorie</i></p>
                    <p>Current status: <b>DONE</b></p>
                </div>

                <p>Have a great day!</p>
            </body>
        </html>


 ===================================================
```

Figure 2: Logs showing the flow: Java - Kafka - Rabbit - Python.