Tutorial: Asynchronous and Event-Driven Microservices Architecture

Subscription Hub – Practical Implementation

1. Introduction

Modern distributed systems must be scalable, resilient, and capable of handling multiple operations simultaneously without blocking user interactions. Traditional monolithic architectures often struggle to meet these requirements, especially when systems grow in complexity and user traffic increases.

This tutorial presents a practical implementation of an **asynchronous, event-driven microservices architecture** using a real working system called **Subscription Hub**. The focus is on how different communication patterns—synchronous REST, asynchronous messaging, event streaming, and Function-as-a-Service (FaaS)—are combined to build a scalable backend system.

The tutorial is based on a fully functional public Git repository and demonstrates real-world technologies such as **RabbitMQ**, **Apache Kafka**, **JWT authentication**, **WebSockets**, and **Node.js microservices**.

2. Problem Statement

In a subscription-based platform, users expect fast feedback when performing actions such as subscribing to a service or making a payment. However, payment processing and fraud detection can be slow or unreliable operations if implemented synchronously.

The main challenges addressed in this tutorial are:

- Avoiding blocking operations during order creation
- Ensuring secure access to REST APIs
- Decoupling services to improve fault tolerance
- Broadcasting system events to multiple consumers
- Delivering real-time feedback to users

To solve these challenges, the system adopts an asynchronous, message-driven approach combined with event streaming and server-side notifications.

3. System Overview

Subscription Hub is composed of multiple independent microservices, each with a single responsibility. All client requests pass through an **API Gateway**, which acts as the single entry point into the system.

Main components:

- API Gateway (Nginx + Node.js)
- Authentication Service
- Orders Service
- Payments Service
- Notifications Service
- Fraud Detection Function (FaaS)
- MongoDB
- RabbitMQ
- Apache Kafka

Each service can be developed, deployed, and scaled independently, following microservices best practices.

4. Securing REST APIs with JWT

Security is a fundamental requirement in distributed systems. Subscription Hub secures all protected REST endpoints using **JSON Web Tokens (JWT)**.

Authentication Flow:

1. The user logs in through the web application.
2. The Authentication Service validates credentials.
3. A signed JWT token is issued.
4. The token is attached to subsequent HTTP requests.
5. The API Gateway validates the token before routing requests.

This approach ensures:

- Stateless authentication
- Reduced load on the authentication service
- Clear separation between security and business logic

JWT validation happens synchronously because it is a lightweight operation and must be completed before processing any request.

5. Fraud Detection as a Function-as-a-Service (FaaS)

Fraud detection is implemented as a **Function-as-a-Service**, following a serverless-style design.

Characteristics:

- Stateless
- Single responsibility
- Invoked only when needed
- Exposed via a simple HTTP endpoint

The Orders Service calls the Fraud Detection FaaS during order creation. Based on the order amount, the function returns a fraud score. Orders with a high fraud score can be rejected early, before payment processing begins.

This design allows the fraud logic to be:

- Easily replaced
- Independently scaled
- Developed without affecting other services

6. Asynchronous Communication with RabbitMQ

Synchronous communication between services can lead to cascading failures and poor performance. To avoid this, Subscription Hub uses **RabbitMQ** as a message broker.

Workflow:

1. The Orders Service publishes a payment request message to a RabbitMQ queue.
2. The Payments Service consumes messages asynchronously.
3. Order creation completes immediately without waiting for payment processing.

Benefits:

- Loose coupling between services
- Improved system responsiveness
- Better fault tolerance

If the Payments Service is temporarily unavailable, messages remain in the queue and are processed later.

7. Event Streaming with Apache Kafka

While RabbitMQ is used for command-based messaging, **Apache Kafka** is used for **event streaming**.

After processing a payment, the Payments Service publishes a **payment confirmation event** to a Kafka topic. Multiple services can consume the same event independently.

Kafka use cases in the system:

- Notifications delivery
- System monitoring
- Future extensions (audit logs, analytics)

Kafka enables the system to scale horizontally and react to events without tight coupling between producers and consumers.

8. Real-Time Notifications with WebSockets

To improve user experience, Subscription Hub provides real-time feedback using **WebSockets**.

Flow:

1. The Notifications Service consumes payment events from Kafka.
2. Events are pushed to connected clients via WebSocket connections.
3. Users receive instant updates without refreshing the page.

This approach demonstrates how event-driven systems can deliver real-time features efficiently.

9. End-to-End Flow Summary

The complete subscription creation flow is as follows:

1. User sends a POST request to create a subscription.
2. API Gateway validates JWT.
3. Orders Service calls Fraud Detection FaaS.
4. Order is stored in MongoDB.

5. Payment request is published to RabbitMQ.
6. Payments Service processes the payment.
7. Payment event is published to Kafka.
8. Notifications Service pushes updates via WebSockets.

This flow combines synchronous and asynchronous communication in a balanced and scalable manner.

10. Conclusion

This tutorial demonstrated how an asynchronous, event-driven microservices architecture can be implemented using modern technologies. By combining REST APIs, message brokers, event streaming, and serverless functions, Subscription Hub achieves scalability, resilience, and real-time communication.

The presented approach is suitable for real-world systems that require high availability and responsive user experiences, making it a practical example of distributed system design.