**Event Driven Microservice Guidelines**

Contents

[1 Document Control 3](#_Toc90479693)

[1.1 Change Record 3](#_Toc90479694)

[1.2 Reviewer 3](#_Toc90479695)

[1.3 Approver 3](#_Toc90479696)

[2 Document Purpose 3](#_Toc90479697)

[3 Event Driven Architecture 3](#_Toc90479698)

[4 Why use event driven architecture 4](#_Toc90479699)

[5 Design Considerations 4](#_Toc90479700)

[5.1 Messaging Framework 4](#_Toc90479701)

[5.1.1 Stream Processing 5](#_Toc90479702)

[5.1.2 Message Processing 5](#_Toc90479703)

[5.1.3 How to choose processing model 5](#_Toc90479704)

[5.1.4 How to Choose Message Broker 5](#_Toc90479705)

[5.1.5 Comparison of Message Brokers 6](#_Toc90479706)

[5.2 Event Sourcing 7](#_Toc90479707)

[5.3 Command Query Responsibility Segregation [CQRS] 7](#_Toc90479708)

[5.4 Discover Event Information 8](#_Toc90479709)

[5.5 Generic Events 8](#_Toc90479710)

[5.6 Create Events Wisely 8](#_Toc90479711)

[5.7 Exception Handling 8](#_Toc90479712)

[5.7.1 Business Exception 8](#_Toc90479713)

[5.7.2 System Exception 8](#_Toc90479714)

[6 Appendix 9](#_Toc90479715)

# Document Control

## Change Record

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Author** | **Version** | **Change reference** |
| 11/30/2021 | Prajeesh T S | 1.1 | Initial version |

## Reviewer

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Approval/Review Date** |
| **Anoop Jose** | Staff Software Architect |  |

## Approver

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Approval/Review Date** |
| **Anoop Jose** | Staff Software Architect |  |

# Document Purpose

This document provides details regarding the best standard’s for event driven micro service architecture.

# Event Driven Architecture

In event-driven architecture, when a service performs some piece of work that other services might be interested in, that service produces an *event*—a record of the performed action. Other services consume those events so that they can perform any of their own tasks needed as a result of the event. Unlike with REST, services that create requests do not need to know the details of the services consuming the requests.

Here’s a simple example: When an order is placed on an ecommerce site, a single “order placed” event is produced and then consumed by several microservices:

1. the order service which could write an *order* record to the database
2. the customer service which could create the *customer* record, and
3. the payment service which could process the payment.

Events can be published in a variety of ways. For example, they can be published to a queue that guarantees delivery of the event to the appropriate consumers, or they can be published to a “pub/sub” model stream that publishes the event and allows access to all interested parties. In either case, the *producer*publishes the event, and the *consumer*receives that event, reacting accordingly. Note that in some cases, these two actors can also be called the *publisher*(the producer) and the *subscriber*(the consumer).

# Why use event driven architecture

An event-driven architecture offers several advantages over REST, which include:

* **Asynchronous** – event-based architectures are asynchronous without blocking. This allows resources to move freely to the next task once their unit of work is complete, without worrying about what happened before or will happen next. They also allow events to be queued or buffered which prevents consumers from putting back pressure on producers or blocking them.
* **Loose Coupling** – services don’t need (and shouldn’t have) knowledge of, or dependencies on other services. When using events, services operate independently, without knowledge of other services, including their implementation details and transport protocol. Services under an event model can be updated, tested, and deployed independently and more easily.
* **Easy Scaling** – Since the services are decoupled under an event-driven architecture, and as services typically perform only one task, tracking down bottlenecks to a specific service, and scaling that service (and only that service) becomes easy.
* R**ecovery support** – An event-driven architecture with a queue can recover lost work by “replaying” events from the past. This can be valuable to prevent data loss when a consumer needs to recover.
* Modularity
* Reliability
* Performance
* Availability

# Design Considerations

## Messaging Framework

The way your events are produced and consumed is a key factor in the system

### Stream Processing

In stream processing, components emit events when they reach a certain state. Other interested components listen for these events on the event stream and react accordingly. Events are not targeted to a certain recipient, but rather are available to all interested components. Components can react to multiple events at the same time, and apply complex operations on multiple streams and events. Some streams include persistence where events stay on the stream for as long as necessary.

One of the most popular stream processing frameworks is Apache Kafka. Kafka is a mature and stable solution used by many projects. It can be considered a go-to, industrial-strength stream processing solution. Kafka has a large userbase, a helpful community, and an evolved toolset.

Event streams enable sending messages to multiple consumers and storing them for later retrieval, which gives room for more flexibility. In addition, keeping message logs can help consumers track various metrics and perform efficient and accurate data analyses.

### Message Processing

In message processing, a service creates a message and sends it to the destination. A subscribing service picks up the message from that destination. In AWS, we use SNS (Simple Notification Service) and SQS (Simple Queue Service). A service sends a message to a topic and a queue subscribing to that topic picks up that message and processes it further. Other than this AWS we have other framework like Active MQ and Rabbit MQ.

### How to choose processing model

The decision to choose processing model should be based on the type of the application, use cases such as payment processing fraud detection, anomaly detection, predictive maintenance, and IoTanalyticsall rely on immediate action on data. All of these use cases deal with data points in a continuous stream, each associated with a specific point in time. These are classic event stream processing examples because the order and timing of the data points help with identifying patterns and trends that represent an important insight for users.

Stream processing can improve any event-driven architecture and ensure flawless performance.

Ecommerce websites are the most common application of message queues. This is because they have an established routing logic that message brokers are familiar with to ensure decoupling and asynchronous task handling in event-driven architectures.

### How to Choose Message Broker

There are lot of message brokers are available in the market, most commonly used ones are, Apache Kafka, Rabbit MQ, Aws SNS, SQS, Azure service bus etc

#### Apache Kafka

Kafka is an open-source message broker developed and maintained primarily by the Apache software foundation with the assistance of the open-source community

**Features**

* High volume publish-subscribe messages and streams platform—durable, fast, and scalable.
* Durable message store—like a log, run in a server cluster, which keeps streams of records in topics (categories).
* Messages—made up of a value, a key, and a timestamp.
* **Communication**—can be synchronous or asynchronous

1. Rabbit MQ

RabbitMQ is a general-purpose message broker that supports protocols including MQTT, AMQP, and STOMP

**Features**

* **Communication**—can be synchronous or asynchronous
* Smart broker / dumb consumer model—consistent delivery of messages to consumers, at around the same speed as the broker monitors the consumer state**.**
* Mature platform—well supported, available for Java, client libraries, .NET, Ruby, node.js. Offers dozens of plugins.

### Comparison of Message Brokers

|  |  |  |  |
| --- | --- | --- | --- |
|  | Apache Kafka | Rabbit MQ | Amazon Kinesis |
| Platform | Open Source | Open Source | Paid |
| Installation | Can be installed and run in local machine | Can be installed and run in local machine | It is a cloud service, cannot be installed in local machine |
| Security | Kafka supports client-side security features like:  Encrypt data-in-transit between your applications and Kafka brokers.  Client authentication.  Client authorization. | RabbitMQ does not encrypt data at rest. Use a filesystem that offers encryption. | For data security, you can use server-side encryption with AWS KMS master keys to encrypt data stored in your data stream. |
| Message Order | Provides messages ordering because of its partitions. Messages are sent to topic by message key | Not supported |  |
| Message Lifetime | Kafka is a log which means the messages are always there, we can specify message retention policy. | It’s a queue messages are done away once consumed. | Stores data for 24 hours by default which can be increase up to 7 days by changing configuration. |
| Delivery Guaranty | Kafka partition guarantees that the messages either pass or fail. | Doesn’t guarantee | Doesn’t guarantee,  If you need guaranteed order of all data in the stream you can only have one shard. That, of course, doesn't scale very well |
| Message Priority | N/A | We can specify message priority and messages are consumed based on priority |  |
| Performance | Kafka leverages sequential disk I/O operations and thus demands less hardware.  Offers much higher performance. It uses sequential disk I/O to boost performance | RabbitMQ controls its messages almost in-memory, using a big cluster (30+ nodes)  Process a million messages per second but requires more resources (around 30 nodes) | Performance is good |
| Scale | Can send up to a million’s messages per second. | Based on configuration and resources. | Auto Scale |
| Use Case | Best fit in event driven development where data must flow between multiple components in the application. Using this distributed message  bus model gives a great deal of scalability.  Large amounts of data. | Best fit in job shaped applications,  Complex routing |  |
|  |  |  |  |

## Event Sourcing

In Event Sourcing, updates and deletes are never performed directly on the data; rather, state changes of an entity are saved as a series of events.

## Command Query Responsibility Segregation [CQRS]

The event sourcing will introduce an issue that state needs to be built from a series of events, queries can be slow and complex.Command Query Responsibility Segregation is a design solution that calls for separate models for insert operations and read operations.

## Discover Event Information

The great challenge in event drive architecture is to catalogue the events, include details like reason for event, description and who created it.

## Generic Events

Don’t use generic events, either in name or in purpose. Other teams to understand why our event exists, what it should be used for, and when it should be used. Events should have a specific purpose and be named accordingly.

## Create Events Wisely

Creating too many events will create unnecessary complexity between the services, increase cognitive load for developers, make deployment and testing more difficult, and cause congestion for event consumers.

## Exception Handling

Exception handling strategy consists of all or some of the following:

* Logging the exception
* Retrying the event for specified number of time and at specified retry intervals
* Moving the event to a dead letter queue (or stopping the processing of events), if all retries are exhausted
* Raising alerts
* In some cases, generating an event
* Correcting the cause of exception and replaying the event

Exceptions can be two types, Business exceptions and System Exceptions, treat them separately.

### Business Exception

Business exceptions are raised when validations or a business condition fails.

* Expected business exceptions are typically handled in the code. Handling could involve logging the exception, updating entities and their state, generating exception events, or consuming the exception and moving on.
* Exceptions due to invalid payloads (including serialization or de-serialization issues) will not be solved with retries. Such events are referred as poison pills in Kafka

### System Exception

System exceptions due to unavailability of components are temporary in nature. Hence, multiple retries should be configured. Another key configuration parameter is backoff multiplier. It is used to have exponentially increasing time interval between consecutive retries.

# Appendix

* Event Sourcing – [Event sourcing (microservices.io)](https://microservices.io/patterns/data/event-sourcing.html)
* CQRS - [Command Query Responsibility Segregation (CQRS) (microservices.io)](https://microservices.io/patterns/data/cqrs.html)
* Message brokers - [Kafka vs Kinesis | Top 5 Differences to Learn with Infographics (educba.com)](https://www.educba.com/kafka-vs-kinesis/)