Event-Driven Messaging: Implementing RabbitMQ, Kafka, and Azure Service Bus in .NET

Nagib Sabbag Filho

Leaders.Tec.Br, 2(32), ISSN: 2966-263X, 2025.

e-mail: profnagib.filho@fiap.com.br

PermaLink: https://leaders.tec.br/article/9e4fd7

Received: 27 Sep 2025 / Accepted: 29 Sep 2025 / Published online: 01 Oct 2025

Abstract:

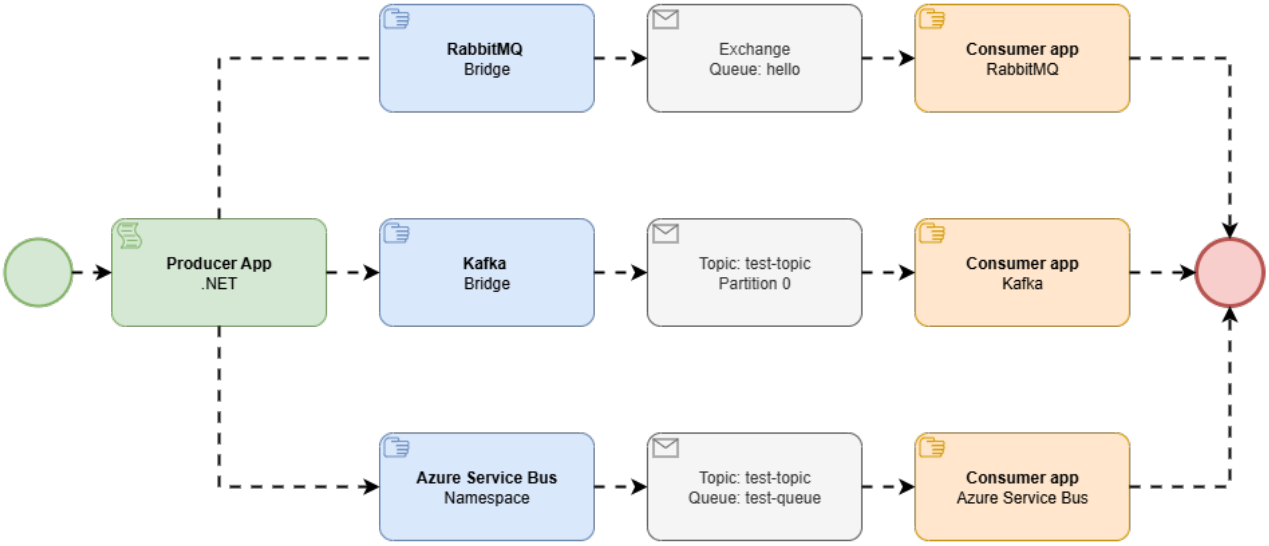
This article provides an introduction to event-driven messaging, highlighting its importance in distributed systems for asynchronous communication. Three main solutions are explored: RabbitMQ, Apache Kafka, and Azure Service Bus, all applicable in .NET. The text includes guidance on installation and practical examples of publishing and consuming messages for each solution.

Key words:

Messaging, Event-Driven, RabbitMQ, Kafka, Azure Service Bus, .NET, Microservices, Architecture, Asynchronous Communication, Messages, Pub/Sub, Message Queue, Scalability, Performance, Resilience, Implementation, Integration, Protocols, Topics, Event Processing, Distributed Applications, C#

Introduction to Event-Driven Messaging

Event-driven messaging is an architectural pattern that enables communication between different components of a system in an asynchronous manner (LAZZARI; FARIAS, 2023). This model is particularly useful in distributed systems, where scalability, resilience, and decoupling of components are essential. In this article, we will explore three of the main event-driven messaging solutions: RabbitMQ, Kafka, and Azure Service Bus, all implementable in .NET applications.



RabbitMQ

RabbitMQ is an open-source message broker that implements the AMQP (Advanced Message Queuing Protocol). It is widely used due to its simplicity and robustness (RABBITMQ, 2025).

Installing RabbitMQ

To install RabbitMQ, you can use Docker. Run the following command:

```
docker run -d --hostname rabbit --name rabbit -p 5672:5672 -
p 15672:15672 rabbitmq:3-management
```

After that, you can access the management interface via the browser at http://localhost:15672 with the default credentials (user: guest, password: guest).

Publishing and Consuming Messages

Next, we present an example of how to publish and consume messages using RabbitMQ in a .NET application.

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Text;

class Program
{
    static void Main()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        using var connection = factory.CreateConnection();
        using var channel = connection.CreateModel();

        channel.QueueDeclare(queue: "hello",
```

```
                             durable: false,
                             exclusive: false,
                             autoDelete: false,
                             arguments: null);

        string message = "Hello World!";
        var body = Encoding.UTF8.GetBytes(message);

        channel.BasicPublish(exchange: "",
                             routingKey: "hello",
                             basicProperties: null,
                             body: body);
        Console.WriteLine($" [x] Sent {message}");

        var consumer = new EventingBasicConsumer(channel);
        consumer.Received += (model, ea) =>
        {
            var body = ea.Body.ToArray();
            var message = Encoding.UTF8.GetString(body);
            Console.WriteLine($" [x] Received {message}");
        };
        channel.BasicConsume(queue: "hello",
                             autoAck: true,
                             consumer: consumer);

        Console.ReadLine();
    }
}
```

Apache Kafka

Apache Kafka is a distributed streaming platform that allows for publishing, subscribing, storing, and processing streams of records in real time (APACHE KAFKA, 2025). It is highly scalable and resilient.

Installing Kafka

To install Kafka, you can also use Docker. Run the following commands:

```
docker run -d --network=host --name zookeeper -e ZOO_MY_ID=1 -
e ZOO_SERVERS=1 zookeeper
docker run -d --network=host --name kafka -
e KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092 -
e KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT -
e KAFKA_ZOOKEEPER_CONNECT=localhost:2181 -
e KAFKA_LISTENERS=PLAINTEXT://0.0.0.0:9092 kafka
```

Publishing and Consuming Messages

To interact with Kafka in .NET, you can use the Confluent.Kafka library. Here is an example of how to publish and consume messages:

```
using Confluent.Kafka;
```

```
using System;
using System.Threading;

class Program
{
    public static void Main(string[] args)
    {
        var config = new ProducerConfig { BootstrapServers = "localhost:9092" };

        using (var producer = new ProducerBuilder<Null, string>(config).Build())
        {
            try
            {
                var dr = producer.ProduceAsync("test-
topic", new Message<Null, string> { Value = "Hello Kafka!" }).Result;
                Console.WriteLine($"Delivered '{dr.Value}' to '{dr.TopicPartitionOf
fset}'");
            }
            catch (ProduceException<Null, string> e)
            {
                Console.WriteLine($"Delivery failed: {e.Error.Reason}");
            }
        }

        var consumerConfig = new ConsumerConfig
        {
            BootstrapServers = "localhost:9092",
            GroupId = "test-consumer-group",
            AutoOffsetReset = AutoOffsetReset.Earliest
        };

        using (var consumer = new ConsumerBuilder<Ignore, string>(consumerConfig).B
uild())
        {
            consumer.Subscribe("test-topic");

            CancellationTokenSource cts = new CancellationTokenSource();
            Console.CancelKeyPress += (_, e) => {
                e.Cancel = true; // prevent the process from terminating
                cts.Cancel();
            };

            try
            {
                while (true)
                {
                    try
                    {
                        var cr = consumer.Consume(cts.Token);
                        Console.WriteLine($"Consumed message '{cr.Value}' at: '{cr.
TopicPartitionOffset}'.");
                    }
                    catch (ConsumeException e)
                    {
                        Console.WriteLine($"Error occurred: {e.Error.Reason}");
                    }
                }
```

```
            }
            catch (OperationCanceledException)
            {
                consumer.Close();
            }
        }
    }
}
```

Azure Service Bus

Azure Service Bus is a managed messaging service that provides queues and topics for asynchronous communication between applications (SABBAG FILHO, 2025). It is ideal for applications requiring high availability and scalability.

Configuring Azure Service Bus

To start using Azure Service Bus, you need to create a namespace in the Azure portal and obtain the connection string.

Publishing and Consuming Messages

Here is an example of how to interact with Azure Service Bus using the Azure.Messaging.ServiceBus library:

```
using Azure.Messaging.ServiceBus;
using System;
using System.Threading.Tasks;

class Program
{
    const string connectionString = "Your_Connection_String";
    const string queueName = "test-queue";

    static async Task Main()
    {
        await SendMessageAsync("Hello Azure Service Bus!");
        await ReceiveMessagesAsync();
    }

    static async Task SendMessageAsync(string messageContent)
    {
        var client = new ServiceBusClient(connectionString);
        var sender = client.CreateSender(queueName);

        var message = new ServiceBusMessage(messageContent);
        await sender.SendMessageAsync(message);
        Console.WriteLine($"Sent: {messageContent}");

        await sender.DisposeAsync();
        await client.DisposeAsync();
    }

    static async Task ReceiveMessagesAsync()
```

```
    {
        var client = new ServiceBusClient(connectionString);
        var processor = client.CreateProcessor(queueName, new ServiceBusProcessorOp
tions());

        processor.ProcessMessageAsync += MessageHandler;
        processor.ProcessErrorAsync += ErrorHandler;

        await processor.StartProcessingAsync();

        Console.WriteLine("Press any key to stop the processing...");
        Console.ReadKey();

        await processor.StopProcessingAsync();
        await processor.DisposeAsync();
        await client.DisposeAsync();
    }

    static Task MessageHandler(ProcessMessageEventArgs args)
    {
        Console.WriteLine($"Received: {args.Message.Body.ToString()}");
        return args.CompleteMessageAsync(args.Message);
    }

    static Task ErrorHandler(ProcessErrorEventArgs args)
    {
        Console.WriteLine($"Error: {args.Exception.Message}");
        return Task.CompletedTask;
    }
}
```

Comparison between RabbitMQ, Kafka, and Azure Service Bus

When choosing a messaging solution, it is important to consider the characteristics of each:

• RabbitMQ: Ideal for applications that require simple and fast communication. It supports persistent messages and various routing options.

• Kafka: Better for data streaming applications and large volumes of transactions. It is highly scalable and allows for real-time processing.

• Azure Service Bus: Great for cloud applications that require a managed service. It offers easy integration with other Azure services and supports transactions.

Conclusion

Event-driven messaging is a crucial component in modern software architectures. RabbitMQ, Kafka, and Azure Service Bus offer robust solutions for different needs. The choice of the right tool depends on the specific requirements of your project, including data volume, system complexity, and existing infrastructure.

References

• SABBAG FILHO, Nagib. Application of best practices in developing workers in .NET for consuming messages in

Azure Service Bus. Leaders Tec, v. 2, n. 17, 2025.

• LAZZARI, Luan; FARIAS, Kleinner. Uncovering the hidden potential of event-driven architecture: A research agenda. arXiv preprint arXiv:2308.05270, 2023.

• RABBITMQ. RabbitMQ Messaging Broker. Available at: https://www.rabbitmq.com/. Accessed on: Sep 2025.

• APACHE KAFKA. Documentation. Available at: https://kafka.apache.org/documentation/. Accessed on: Sep. 2025.

Nagib acts as a Microsoft MVP, has experience as a Systems Architecture Leader, Tech Manager, and University Professor. He is currently working at Mercado Livre. He holds technical and agile certifications, such as GitHub Copilot, PSM1, and Azure Fundamentals. He has a postgraduate degree from SENAC and from Mackenzie Presbyterian University, with an MBA in Software Technology from USP, as well as having participated in extension programs at MIT and the University of Chicago. He is the author of a peer-reviewed scientific article on chatbots, presented in person at the University of Barcelona. He actively participates in the technical community, with over 50 published articles, including more than 10 articles on iMasters. He was also a speaker in the .NET Architecture track at TDC São Paulo 2024 and in the Solutions Architecture track at TDC Floripa 2025. He is confirmed to speak at MVPConf 2025.