# Event-Driven Architecture (EDA) with Worker Services in Microservices

Event-Driven Architecture (EDA) is a software design pattern in which services (or components) communicate with each other through events. In the context of microservices, EDA helps decouple services and ensures scalability and responsiveness by using asynchronous communication.

## Key Concepts

1. Event: A message indicating a change in state or an occurrence (e.g., 'OrderCreated', 'PaymentProcessed').  
2. Event Publisher: A service or component that raises an event (e.g., an Order Service publishing an 'OrderCreated' event).  
3. Event Consumer: A service or component that subscribes to and handles specific events.  
4. Message Broker: Middleware to facilitate communication between publishers and consumers (e.g., RabbitMQ, Kafka).  
5. Worker Service: A long-running background service in .NET Core used for consuming and processing events.

## Why Use Worker Services in EDA?

1. Asynchronous Processing: Handles events asynchronously, reducing the load on user-facing services.  
2. Scalability: Worker Services can scale independently to meet demand.  
3. Decoupling: Allows publishers and consumers to evolve independently.

## Components in EDA with Worker Services

1. Publisher Service: Raises events when specific actions occur.  
2. Message Broker: Routes events to appropriate consumers.  
3. Consumer Worker Service: Subscribes to events and processes them.

## Implementation Example: Order Processing System

This example demonstrates an order processing system using ASP.NET Core Web API for the publisher, RabbitMQ as the message broker, and a Worker Service as the consumer.

### Step 1: Set Up RabbitMQ

1. Install RabbitMQ using Docker:  
 docker run -d --hostname rabbit --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:management  
2. Access RabbitMQ Management UI at http://localhost:15672 (default credentials: guest/guest).

### Step 2: Create an Event Publisher

Implement an Order Service to publish events using RabbitMQ.

#### Code Example

using RabbitMQ.Client;  
 using System.Text;  
  
 public class OrderService  
 {  
 public void PublishOrderCreatedEvent(string orderId)  
 {  
 var factory = new ConnectionFactory() { HostName = "localhost" };  
 using var connection = factory.CreateConnection();  
 using var channel = connection.CreateModel();  
  
 channel.QueueDeclare(queue: "OrderQueue", durable: false, exclusive: false, autoDelete: false, arguments: null);  
  
 var message = $"{{ "OrderId": "{orderId}" }}";  
 var body = Encoding.UTF8.GetBytes(message);  
  
 channel.BasicPublish(exchange: "", routingKey: "OrderQueue", basicProperties: null, body: body);  
 Console.WriteLine($"OrderCreated event published: {message}");  
 }  
 }

### Step 3: Create a Worker Service (Consumer)

Create a Worker Service to consume messages from RabbitMQ and process them.

using RabbitMQ.Client;  
 using RabbitMQ.Client.Events;  
 using System.Text;  
  
 public class Worker : BackgroundService  
 {  
 protected override async Task ExecuteAsync(CancellationToken stoppingToken)  
 {  
 var factory = new ConnectionFactory() { HostName = "localhost" };  
 using var connection = factory.CreateConnection();  
 using var channel = connection.CreateModel();  
  
 channel.QueueDeclare(queue: "OrderQueue", durable: false, exclusive: false, autoDelete: false, arguments: null);  
  
 var consumer = new EventingBasicConsumer(channel);  
 consumer.Received += (model, ea) =>  
 {  
 var body = ea.Body.ToArray();  
 var message = Encoding.UTF8.GetString(body);  
 Console.WriteLine($"Order received: {message}");  
 // Add logic to update inventory here  
 };  
  
 channel.BasicConsume(queue: "OrderQueue", autoAck: true, consumer: consumer);  
  
 await Task.CompletedTask;  
 }  
 }

### Step 4: Test the System

1. Start RabbitMQ.  
2. Run the Publisher (Order Service) and Worker Service (InventoryWorker).  
3. Send a request to the Order API to publish an event:  
 POST /api/orders  
4. Verify the Worker Service logs the event and processes it.

## Scalability and Advanced Features

1. Horizontal Scaling: Deploy multiple instances of the Worker Service.  
2. Dead Letter Queues: Handle unprocessable messages.  
3. Retry Policies: Implement retry mechanisms for failed processing.  
4. Event Filtering: Use topics or routing keys to direct messages to specific consumers.