**Synchronous Communication:**

In this exercise, we will explore how to deploy Web API in Kubernetes. First of all, create a .NET9 Web API application with the below code:

**Program.cs (Microservice B)**

using Microsoft.OpenApi.Models;

using System.Diagnostics;

using System.Text.Json.Serialization;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers().AddJsonOptions(options =>

{

options.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.Preserve;

});

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "Microservice B", Version = "v1" });

});

var app = builder.Build();

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Microservice B"));

app.Lifetime.ApplicationStarted.Register(() =>

{

var url = "https://localhost:7074/Swagger";

try

{

Process.Start(new ProcessStartInfo

{

FileName = url,

UseShellExecute = true

});

}

catch (Exception ex)

{

Console.WriteLine($"Could not open browser: {ex.Message}");

}

});

}

app.UseHttpsRedirection();

app.MapControllers();

app.Run();

**JustBController.cs**

using Microsoft.AspNetCore.Mvc;

namespace MicroserviceB.Controllers;

[Route("api/[controller]")]

[ApiController]

public class JustBController : ControllerBase

{

[HttpPost("process")]

public IActionResult ProcessData([FromBody] string message)

{

Console.WriteLine($"[B] Received: {message}");

return Ok($"[B] Processed: {message}");

}

}

**DockerFile**

# See https://aka.ms/customizecontainer to learn how to customize your debug container and how Visual Studio uses this Dockerfile to build your images for faster debugging.

# This stage is used when running from VS in fast mode (Default for Debug configuration)

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

USER $APP\_UID

WORKDIR /app

EXPOSE 80

# This stage is used to build the service project

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

ARG BUILD\_CONFIGURATION=Release

WORKDIR /src

COPY ["MicroserviceB.csproj", "./"]

RUN dotnet restore "./MicroserviceB.csproj"

COPY . .

WORKDIR "/src"

RUN dotnet build "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/build

# This stage is used to publish the service project to be copied to the final stage

FROM build AS publish

ARG BUILD\_CONFIGURATION=Release

RUN dotnet publish "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/publish /p:UseAppHost=false

# This stage is used in production or when running from VS in regular mode (Default when not using the Debug configuration)

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "MicroserviceB.dll"]

Dockerfile includes four stages, with some nuances tailored for Visual Studio (VS) integration and optimization. Let’s break down each stage and its significance in detail:

1. **base Stage**

FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS base

USER $APP\_UID

WORKDIR /app

EXPOSE 80

**Purpose**:

* **Runtime Environment Setup**: Prepares the base image for both **debugging** (fast mode in VS) and **production**.
* **Security**: Sets a non-root user via USER $APP\_UID to follow least-privilege principles.
* **Port Configuration**: Exposes port 80 for HTTP traffic.
* **Workdir**: Defines /app as the working directory for the app.

**Significance**:

* Serves as the foundation for the final stage.
* Optimized for VS debugging (e.g., hot reload support).
* Avoids redundancy by reusing this stage across configurations.

1. **build Stage**

FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build

ARG BUILD\_CONFIGURATION=Release

WORKDIR /src

COPY ["MicroserviceB.csproj", "./"]

RUN dotnet restore "./MicroserviceB.csproj"

COPY . .

RUN dotnet build "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/build

**Purpose**:

* **Dependency Restoration**: Restores NuGet packages *before* copying all source code to leverage Docker layer caching.
* **Build Process**: Compiles the app using the specified BUILD\_CONFIGURATION (default: Release).

**Significance**:

* Separates dependency restoration (dotnet restore) from the build (dotnet build) to optimize caching.
* The ARG parameter allows flexibility (e.g., switching between Debug and Release modes).
* Uses the **SDK image** to access build tools.

1. **publish Stage**

FROM build AS publish

ARG BUILD\_CONFIGURATION=Release

RUN dotnet publish "./MicroserviceB.csproj" -c $BUILD\_CONFIGURATION -o /app/publish /p:UseAppHost=false

**Purpose**:

* **Publishing**: Generates a self-contained deployment bundle.
* **Optimization**: The flag /p:UseAppHost=false skips generating an OS-specific executable (reduces size, as containers use dotnet CLI).

**Significance**:

* Isolates the publish step from the build to minimize the final image size.
* Prepares the app for deployment by including only runtime assets.

1. **final Stage**

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "MicroserviceB.dll"]

**Purpose**:

* **Production-Ready Image**: Uses the lightweight aspnet runtime image from the base stage.
* **Artifact Copy**: Pulls the published output from the publish stage.
* **Entrypoint**: Defines how to launch the app (executes the compiled DLL).

**Significance**:

* Final image excludes SDK and build tools, reducing size and attack surface.
* Inherits security settings (non-root user) and port configuration from base.

**Folder Structure:**

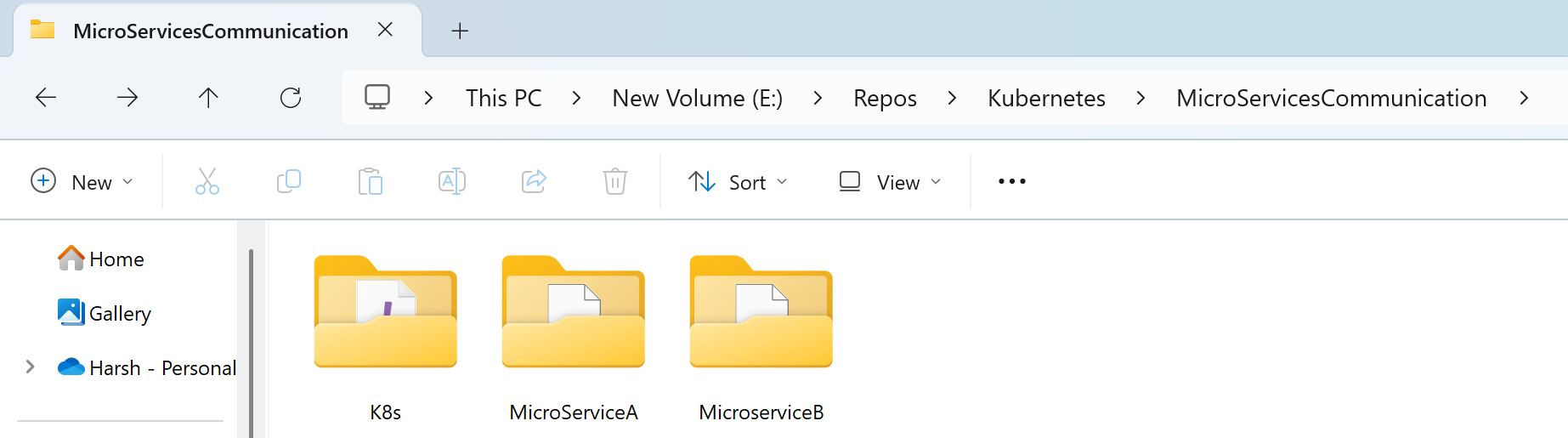
E:\Repos\Kubernetes\MicroServicesCommunication\MicroserviceB\MicroserviceB

The above folder contains the DockerFile and .csproj file which are required to build the docker image.

To run Kubernetes on local, you need to install the below tools:

1. Minikube
2. Kubectl
3. VirtualBox

Also, you need to create a deployment.yaml file that will deploy the services on your local. Here is the folder structure:



Command to build Docker Image:

docker build -t microservicea:latest .

Let's break down the command docker build -t microservicea:latest . and explain each component in detail:

1. **docker**

* **Role:** The Docker CLI (Command Line Interface) tool used to interact with the Docker daemon (background service).
* **Purpose:** Executes Docker commands like building images, running containers, etc.

1. **build**

* **Role:** The Docker subcommand to build a new image from a Dockerfile and a "build context" (files/directories used in the image).
* **Purpose:** Reads the Dockerfile in the current directory (or a specified path) and creates a layered image based on the instructions in the file.

1. **-t microservicea:latest (or --tag)**

* **Role:** The -t flag assigns a name and optionally a tag to the image.
* **Details:**
  + **microservicea:** The name of the image (e.g., a microservice named "microservicea").
  + **:latest:** The tag for the image. latest is a default tag used when no explicit tag is provided. It typically represents the most recent version of the image.
* **Purpose:** Tags the image for easy identification and versioning. Example: microservicea:1.0, microservicea:dev.

1. **. (the "build context")**

* **Role:** Specifies the build context, i.e., the directory where Docker will look for the Dockerfile and any files referenced in it (e.g., source code, configuration files).
* **Details:**
  + . refers to the current working directory.
  + Docker copies all files in this directory to the Docker daemon (unless excluded via .dockerignore).
  + The Dockerfile is expected to be in this directory by default. If it has a different name or path, use the -f flag (e.g., -f path/to/Dockerfile).

**Common Errors to Avoid:**

* Forgetting the . at the end (Docker won’t know where to find the build context).
* Using uppercase characters or special symbols in image names (Docker enforces lowercase names with underscores, hyphens, or numbers).

Definition of microserviceb-deployment.yaml file:

apiVersion: apps/v1

kind: Deployment

metadata:

  name: microserviceb

spec:

  replicas: 1

  selector:

    matchLabels:

      app: microserviceb

  template:

    metadata:

      labels:

        app: microserviceb

    spec:

      containers:

      - name: microserviceb

        image: microserviceb:latest

        env:

        - name: ASPNETCORE\_ENVIRONMENT

          value: Development

        - name: ASPNETCORE\_URLS

          value: "http://\*:82"

        imagePullPolicy: Never

        ports:

        - containerPort: 82

---

apiVersion: v1

kind: Service

metadata:

  name: microserviceb

spec:

  selector:

    app: microserviceb

  ports:

    - protocol: TCP

      port: 80

      targetPort: 82

What is the significance of --- in the above script? Do I need the whole script or something could be trimmed off?

The --- in a Kubernetes YAML file is a **document separator**. It allows you to define **multiple resources in the same file**. In your example, you're defining:

1. A Deployment named microserviceb
2. A Service named microserviceb

**Significance of ---**

* It separates two different Kubernetes resources within a single YAML file.
* It's not strictly required if you're applying just one resource, but it **is required** when you define **multiple resources** in the same file so that the Kubernetes parser knows where one ends and the next begins.

**Do you need the whole script?**

Yes, **both parts** are usually necessary for a functioning microservice:

* The Deployment defines the **pod and container(s)**.
* The Service exposes the pod so that other services or users can communicate with it (e.g., via port 80 in your case).

If you **only apply the Deployment**, there will be **no stable endpoint** (like a DNS name or load balancer) to access the microservice unless you do it manually (like using kubectl port-forward).

**You should keep both sections and the --- in between if:**

* You're applying both the Deployment and the Service in a single file.
* You want to ensure everything deploys and works together.

Can you describe each and every keyword of the above script in detail?

1. Deployment Configuration

apiVersion: apps/v1

* apiVersion: Specifies the Kubernetes API version you're using for this object.
* apps/v1 is used for managing Deployments, DaemonSets, and StatefulSets.

kind: Deployment

* kind: The type of Kubernetes object you're creating.
* Here, you're creating a Deployment, which manages replica sets and ensures the desired number of pod replicas are running.

metadata:

name: microserviceb

* metadata: Contains metadata information like the name, namespace, labels, etc.
* name: The unique name of the Deployment object in the cluster.

spec:

replicas: 1

* spec: Specifies the desired state of the object.
* replicas: Number of pod instances to run. Here, it is just 1 replica.

selector:

matchLabels:

app: microserviceb

* selector: Determines which pods belong to this Deployment by matching labels.
* matchLabels: Filters pods whose app label equals microserviceb.

template:

metadata:

labels:

app: microserviceb

* template: Describes the pod template to be used when creating pods.
* metadata.labels: Labels assigned to pods created by this Deployment.
* These labels must match the selector above for the Deployment to manage the pods.

spec:

containers:

- name: microserviceb

* spec (within template): Defines the container specs for the pod.
* containers: A list of containers (in this case, just one).
* name: A unique name for this container within the pod.

image: microserviceb:latest

* image: Docker image to use. microserviceb:latest means use the latest tag of the microserviceb image.

env:

- name: ASPNETCORE\_ENVIRONMENT

value: Development

- name: ASPNETCORE\_URLS

value: [http://\*:82](http://*:82)

* env: Specifies environment variables to set inside the container.
* ASPNETCORE\_ENVIRONMENT: Typically Development, Staging, or Production.
* ASPNETCORE\_URLS: Tells ASP.NET Core which URL/port to bind to. http://\*:82 binds to port 82 on all interfaces.

imagePullPolicy: Never

* imagePullPolicy: Controls when the image is pulled from a registry.
* Never means Kubernetes will not pull the image — it assumes it's already available locally on the node.
* Useful during local development/testing with local images.

ports:

- containerPort: 82

* ports: Specifies container ports that are exposed by the container.
* containerPort: Port inside the container that the app is listening on (port 82).

1. Service Configuration

---

* This is a YAML document separator. It lets you define multiple resources in the same file.

apiVersion: v1

* The API version for the Service resource.

kind: Service

* You're creating a Kubernetes Service.
* A Service allows network access to a set of pods.

metadata:

name: microserviceb

* Metadata for the service. The name will be used as the service DNS entry (e.g., microserviceb.default.svc.cluster.local).

spec:

selector:

app: microserviceb

* selector: Selects pods that match the given labels.
* Here, it matches pods with the label app: microserviceb (like the ones defined in the Deployment).

ports:

- protocol: TCP

port: 80

targetPort: 82

* ports: List of ports exposed by the service.
* protocol: Network protocol (typically TCP).
* port: Port on which the service is exposed internally.
* targetPort: Port on the pod that receives traffic from the service. It must match the container's listening port.

So:

* The service listens on port **80**, and routes that traffic to port **82** in the pod.

|  |  |
| --- | --- |
| Keyword | Description |
| apiVersion | API version to use for the resource |
| kind | Type of Kubernetes object |
| metadata | Identifiers and labels |
| spec | Specification of the desired state |
| replicas | Number of pods to run |
| selector | How to identify pods for management or exposure |
| template | Pod blueprint |
| containers | List of containers in a pod |
| env | Environment variables |
| imagePullPolicy | When the image should be pulled |
| containerPort | Port exposed by the container |
| --- | Separates multiple resources in the YAML file |
| Service | Creates a stable endpoint to access pods |

Definition of deployment.yaml file:

apiVersion: apps/v1

kind: Deployment

metadata:

  name: microservicea

spec:

  replicas: 1

  selector:

    matchLabels:

      app: microservicea

  template:

    metadata:

      labels:

        app: microservicea

    spec:

      containers:

      - name: microservicea

        image: microservicea:latest

        env:

        - name: ASPNETCORE\_URLS

          value: "http://\*:80"

        imagePullPolicy: Never

        ports:

        - containerPort: 80

---

apiVersion: v1

kind: Service

metadata:

  name: microservicea

spec:

  selector:

    app: microservicea

  ports:

    - protocol: TCP

      port: 80

      targetPort: 80

  type: NodePort

Q. The above is the example of synchronous communication between the two microservices. Let's say Microservice A makes an entry to the database and calls the Microservice B, which fails in its operation. Now I want that Microservice B should send a rollback response to Microservice A, which indeed should return the state of the system to the original one, how to meet such a scenario? Let me know if you need some more details from my side regarding the code or any other file. Or the scenario could be vice-versa.

A. To handle scenarios where **Microservice A** performs a database operation and then calls **Microservice B**, which might fail, you can implement the **Saga Pattern** with **compensating transactions**. This approach ensures data consistency across microservices without relying on distributed transactions.

**Understanding the Saga Pattern**

The Saga Pattern breaks a transaction into a series of steps, each managed by a different microservice. If a step fails, the system executes compensating actions to undo the previous steps.

**Implementing Compensating Transactions**

**In Microservice A:**

1. **Perform the initial database operation.**
2. **Call Microservice B.**
3. **If Microservice B responds with a failure, execute a compensating action to undo the database operation.**

app.MapGet("/send", async (IHttpClientFactory factory, MyDbContext dbContext) =>

{

// Step 1: Perform initial operation

var record = new MyRecord { Data = "Initial Data" };

dbContext.MyRecords.Add(record);

await dbContext.SaveChangesAsync();

// Step 2: Call Microservice B

var client = factory.CreateClient();

var response = await client.PostAsJsonAsync("http://microserviceb:7074/api/JustB/process", "Hello from A");

if (!response.IsSuccessStatusCode)

{

// Step 3: Compensating action

dbContext.MyRecords.Remove(record);

await dbContext.SaveChangesAsync();

return Results.Problem("Microservice B failed. Rolled back changes.");

}

var ack = await response.Content.ReadAsStringAsync();

return Results.Ok($"[A] Got response: {ack}");

});

**In Microservice B:**

Implement your processing logic and return appropriate HTTP status codes to indicate success or failure.

**Alternative: Using a Saga Orchestrator**

For more complex workflows, consider implementing a Saga orchestrator that manages the sequence of operations and their compensating actions. This can be done using:

* **Choreography:** Each service knows the next step and triggers it.
* **Orchestration:** A central orchestrator directs the sequence of steps.

In .NET, you can implement an orchestrator using tools like **MassTransit** or **NServiceBus**. Implementing the Saga Pattern with compensating transactions allows your microservices to maintain data consistency without tight coupling or distributed transactions. It's a robust approach for handling failures gracefully in a microservices architecture.

Q. I am unable to understand the Saga code that you have provided me above. The code is just checking the status response, and if it is not successful, then it simply deletes the data inserted in the previous step. Where is Saga in this picture?

A. You're absolutely right to question the earlier example. The code snippet provided demonstrates a simple rollback mechanism within a single service, not a full-fledged implementation of the Saga pattern. Let's delve deeper into what constitutes the Saga pattern and how it can be properly implemented in a microservices architecture.

**Understanding the Saga Pattern**

The Saga pattern is a design approach to manage distributed transactions across multiple microservices. Instead of relying on a single, atomic transaction, the Saga pattern breaks down a transaction into a series of local transactions. Each local transaction updates the data within a single service and publishes an event or message to trigger the next transaction. If a local transaction fails, the Saga executes a series of compensating transactions to undo the changes made by the preceding transactions.

**Implementing the Saga Pattern**

There are two primary approaches to implementing the Saga pattern:

1. **Choreography-Based Saga**:
   * Each service performs its local transaction and publishes an event.
   * Other services listen for these events and perform their transactions accordingly.
   * If a failure occurs, services listening to failure events execute compensating transactions.
2. **Orchestration-Based Saga**:

* A central orchestrator service manages the sequence of transactions.
* It invokes each service and decides whether to proceed or execute compensating transactions based on the responses.

**Example Scenario**

Let's consider a scenario where:

1. **Microservice A**: Creates an order.
2. **Microservice B**: Reserves inventory.
3. **Microservice C**: Processes payment.

**Orchestration-Based Saga Flow**:

1. The orchestrator instructs Microservice A to create an order.
2. Upon success, it instructs Microservice B to reserve inventory.
3. If inventory reservation is successful, it instructs Microservice C to process payment.
4. If payment processing fails, the orchestrator initiates compensating transactions:
   * Instructs Microservice B to release the reserved inventory.
   * Instructs Microservice A to cancel the order.

**Choreography-Based Saga Flow**:

1. Microservice A creates an order and publishes an "OrderCreated" event.
2. Microservice B listens for the "OrderCreated" event, reserves inventory, and publishes an "InventoryReserved" event.
3. Microservice C listens for the "InventoryReserved" event, processes payment, and publishes a "PaymentProcessed" event.
4. If payment processing fails, Microservice C publishes a "PaymentFailed" event.
5. Microservice B listens for the "PaymentFailed" event and releases the reserved inventory.
6. Microservice A listens for the "PaymentFailed" event and cancels the order.

**Compensating Transactions**

Compensating transactions are crucial in the Saga pattern. They are operations that undo the effects of previous transactions to maintain data consistency. For instance, if an order is created but payment fails, a compensating transaction would cancel the order. It's essential to design these compensating actions carefully to ensure they can handle failures and maintain idempotency.

**Tools and Frameworks**

In the .NET ecosystem, several tools can aid in implementing the Saga pattern:

* **MassTransit**: A distributed application framework for .NET that supports saga state machines and integrates with message brokers like RabbitMQ and Azure Service Bus.
* **NServiceBus**: Provides support for long-running processes and sagas, handling complex workflows and state management.
* **CAP (EventBus)**: A .NET library to achieve eventual consistency in distributed systems through event publishing and processing.

Q. Ok, let's go step by step. Can you provide me with an example of a choreography-based saga design pattern using the code that I have shared with you above?

A. Certainly! Let's walk through implementing a **Choreography-based Saga pattern** using your existing microservices setup. This approach ensures that each service autonomously handles its transactions and communicates state changes through events, promoting loose coupling and scalability.

**Understanding Choreography-Based Saga**

In a choreography-based saga:

* **Decentralized Coordination**: Each microservice performs its local transaction and publishes an event.
* **Event-Driven Communication**: Other services listen for these events and act accordingly.
* **Compensating Actions**: If a service encounters a failure, it publishes a compensating event to undo previous actions.

This pattern is particularly effective for simple workflows and when services need to remain decoupled.

**Implementing Choreography-Based Saga**

**1. Set Up an Event Bus**

To facilitate communication between services, we'll use a message broker like **RabbitMQ**. In .NET, libraries like **MassTransit** can simplify integration.

**Install MassTransit and RabbitMQ Client:**

dotnet add package MassTransit

dotnet add package MassTransit.RabbitMQ

Configure MassTransit in your services:

builder.Services.AddMassTransit(x =>

{

x.UsingRabbitMq((context, cfg) =>

{

cfg.Host("rabbitmq://localhost");

});

});

Ensure RabbitMQ is running locally or adjust the host settings accordingly.

**2. Define Events**

Create shared event contracts that services will publish and consume.

public record OrderCreatedEvent(Guid OrderId, string Product);

public record InventoryReservedEvent(Guid OrderId);

public record InventoryFailedEvent(Guid OrderId, string Reason);

public record PaymentProcessedEvent(Guid OrderId);

public record PaymentFailedEvent(Guid OrderId, string Reason);

**3. Microservice A (Order Service)**

**Publish OrderCreatedEvent after creating an order:**

app.MapPost("/create-order", async (IPublishEndpoint publishEndpoint) =>

{

var orderId = Guid.NewGuid();

// Save order to database with status 'Pending'

await publishEndpoint.Publish(new OrderCreatedEvent(orderId, "Sample Product"));

return Results.Ok($"Order {orderId} created.");

});

Handle compensating actions:

public class PaymentFailedConsumer : IConsumer<PaymentFailedEvent>

{

public async Task Consume(ConsumeContext<PaymentFailedEvent> context)

{

var orderId = context.Message.OrderId;

// Update order status to 'Cancelled' in database

}

}

**4. Microservice B (Inventory Service)**

**Consume OrderCreatedEvent and publish InventoryReservedEvent:**

public class OrderCreatedConsumer : IConsumer<OrderCreatedEvent>

{

public async Task Consume(ConsumeContext<OrderCreatedEvent> context)

{

var orderId = context.Message.OrderId;

// Attempt to reserve inventory

bool success = true; // Replace with actual logic

if (success)

{

await context.Publish(new InventoryReservedEvent(orderId));

}

else

{

await context.Publish(new InventoryFailedEvent(orderId, "Insufficient stock."));

}

}

}

**5. Microservice C (Payment Service)**

**Consume InventoryReservedEvent and publish PaymentProcessedEvent:**

public class InventoryReservedConsumer : IConsumer<InventoryReservedEvent>

{

public async Task Consume(ConsumeContext<InventoryReservedEvent> context)

{

var orderId = context.Message.OrderId;

// Attempt to process payment

bool success = true; // Replace with actual logic

if (success)

{

await context.Publish(new PaymentProcessedEvent(orderId));

}

else

{

await context.Publish(new PaymentFailedEvent(orderId, "Payment declined."));

}

}

}

**Handling Failures with Compensating Actions**

If any service encounters a failure:

* **Inventory Service**: Publishes InventoryFailedEvent if unable to reserve stock.
* **Payment Service**: Publishes PaymentFailedEvent if payment processing fails.
* **Order Service**: Listens to failure events and updates the order status accordingly, possibly triggering further compensating actions.

**Asynchronous Communication:**

The idea is now to change the code from synchronous communication to asynchronous communication. For that we need the following packages:

1. RabbitMQ
2. MassTransit

**RabbitMQ Overview**

RabbitMQ is an open-source message broker that implements the AMQP (Advanced Message Queuing Protocol) standard. It acts as a middleware for asynchronous communication between distributed systems.

**Key Features:**

* **Exchanges, Queues, Bindings:** Routes messages using rules (direct, topic, fanout, headers).
* **Persistence:** Messages can be stored on disk for reliability.
* **Clustering & High Availability:** Supports fault-tolerant deployments.
* **Protocols:** Supports AMQP, MQTT, STOMP, etc.
* **Use Case:** Decouple microservices, handle background jobs, or manage event-driven architectures.

**MassTransit Overview**

MassTransit is a .NET library for building distributed applications using messaging patterns. It abstracts the underlying transport (e.g., RabbitMQ, Azure Service Bus, Amazon SQS).

**Key Features:**

* **Abstraction Layer:** Write broker-agnostic code.
* **Message Patterns:** Simplifies pub/sub, request-response, sagas (state machines), retries, and error handling.
* **Automatic Serialization:** Handles JSON/XML/Binary serialization.
* **Middleware Pipeline:** Supports interceptors, filters, and custom behaviors.
* **Use Case:** Simplify complex messaging logic in .NET apps while supporting multiple transports.

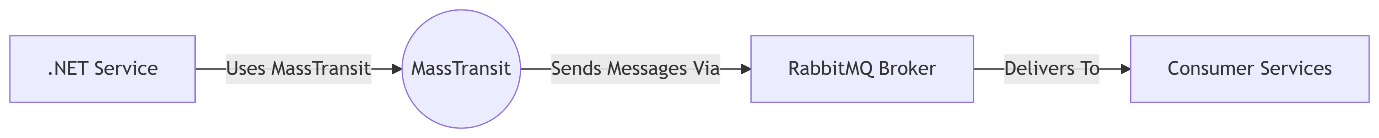
**Key Differences**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **RabbitMQ** | **MassTransit** |
| **Type** | Message Broker (infrastructure) | .NET Library (abstraction framework) |
| **Primary Role** | Transport layer for messages | Framework to *use* messaging systems |
| **Complexity** | Low-level (you manage queues/exchanges) | High-level (abstracts broker details) |
| **Code Focus** | Broker configuration & protocol specifics | Business logic & messaging patterns |
| **Retry/Error Handling** | Manual implementation | Built-in (automatic retries, dead-letter queues) |
| **Sagas** | Not supported natively | Built-in support for stateful workflows |
| **Multi-Broker Support** | Only RabbitMQ | RabbitMQ, Azure Service Bus, Amazon SQS, etc. |

**When to Use What?**

* Use RabbitMQ Directly:
  + You need fine-grained control over queues/exchanges.
  + Your project is small or language-agnostic.
  + You prefer lightweight integration without .NET dependencies.
* Use MassTransit:
  + You're building complex .NET applications with messaging.
  + You want built-in patterns (sagas, retries) and broker portability.
  + You need to reduce boilerplate code for error handling.

**Example Workflow**



**In Practice**:  
MassTransit uses RabbitMQ as a *transport*, handling routing, serialization, and reliability so you focus on business logic.

// MassTransit + RabbitMQ Example

services.AddMassTransit(x =>

{

x.UsingRabbitMq((ctx, cfg) =>

{

cfg.Host("localhost");

cfg.ReceiveEndpoint("order-queue", e =>

{

e.Consumer<OrderConsumer>();

});

});

});

Your code structure for Microservice A using MassTransit and RabbitMQ looks quite solid and clean. You're using modern practices with Minimal APIs, MassTransit for messaging, EF Core for data persistence, and DI for services.

**Working Aspects:**

1. RabbitMQ Publishing:
   * publishEndpoint.Publish(new OrderCreatedEvent(...)) in /create-order endpoint is correct. This publishes to the bus.
   * MassTransit is properly configured with a RabbitMQ host.
2. MassTransit Consumer Setup:
   * You've added PaymentFailedConsumer to listen to PaymentFailedEvent.
   * AddConsumer and UsingRabbitMq setup is appropriate.
3. Entity Framework:
   * OrderDbContext is correctly registered with SQL Server and contains the Orders DbSet.
4. Minimal API Endpoints:
   * /create-order and /send are syntactically and semantically correct.

**Redundant / Unused Code:**

Here are a few things that are not needed right now or can be cleaned up:

1. Unused Consumer: PaymentFailedConsumer

* You're not consuming anything yet in Microservice A (you're publishing).
* This consumer makes sense only if you're handling PaymentFailedEvent in Microservice A.
* If this is a future concern or handled by Microservice C, remove it for now to simplify things.

x.AddConsumer<PaymentFailedConsumer>(); // 🔁 Remove if not needed yet

1. **/send Endpoint Using HttpClient:**

* /send is unrelated to RabbitMQ.
* It's fine for integration testing with Microservice B, but it **is not necessary** for RabbitMQ flow.
* Remove if it's not needed at this stage.

app.MapGet("/send", ...) // 🔁 Optional — remove if irrelevant now

**Good Practices in Place:**

* You’ve correctly used IPublishEndpoint (good instead of using IBus directly).
* Used record types for events (👍 ideal for immutability in messaging).
* Added timeouts for MassTransitHostOptions (nice detail!).

**Missing / Recommended Additions:**

1. RabbitMQ URI Format Fix

This may be your biggest technical issue.

cfg.Host("rabbitmq://localhost");

This format is **deprecated** in MassTransit v8+. You should instead use:

cfg.Host("localhost", "/", h =>

{

h.Username("guest");

h.Password("guest");

});

This is critical or your app may fail to connect to RabbitMQ.

1. Consumer Configuration (If Kept)

x.AddConsumer<PaymentFailedConsumer>(cfg =>

{

cfg.UseMessageRetry(r => r.Interval(3, TimeSpan.FromSeconds(5)));

});

x.UsingRabbitMq((context, cfg) =>

{

cfg.Host("localhost", "/", h =>

{

h.Username("guest");

h.Password("guest");

});

cfg.ReceiveEndpoint("payment-failed-queue", e =>

{

e.ConfigureConsumer<PaymentFailedConsumer>(context);

});

});

Otherwise, your consumer won’t get registered to a queue.

Corrected Minimal Version of Program.cs (For Only Publishing)

If you're just **publishing** to RabbitMQ for now (no consuming):

using MassTransit;

using MicroServiceA.Data;

using Microsoft.EntityFrameworkCore;

using static MicroServiceA.SharedEvents.OrderEvents;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<OrderDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("OrderDatabase")));

builder.Services.AddMassTransit(x =>

{

x.UsingRabbitMq((context, cfg) =>

{

cfg.Host("localhost", "/", h =>

{

h.Username("guest");

h.Password("guest");

});

});

});

builder.Services.Configure<MassTransitHostOptions>(options =>

{

options.WaitUntilStarted = true;

options.StartTimeout = TimeSpan.FromSeconds(30);

options.StopTimeout = TimeSpan.FromMinutes(1);

});

var app = builder.Build();

app.MapPost("/create-order", async (IPublishEndpoint publishEndpoint) =>

{

var orderId = Guid.NewGuid();

await publishEndpoint.Publish(new OrderCreatedEvent(orderId, "Sample Product"));

return Results.Ok($"Order {orderId} created.");

});

app.Run();

If you plan to **consume PaymentFailedEvent** as well, let me know — I’ll share the full setup with queue binding and config.

Let me know how you want to proceed next (build Microservice B, add consumer config, etc.)

**Q. I am not too sure do I need to install RabbitMQ to my local system as well? This is the first time I am working on it.**

**Do You Need to Install RabbitMQ Locally?**

**Yes**, you need **a running RabbitMQ server** for your microservice to:

* **Publish** messages (like you're doing with OrderCreatedEvent)
* **Consume** messages (if you add consumers)

**3 Easy Ways to Run RabbitMQ Locally**

You can run RabbitMQ using **one of the following methods**:

Option 1: Use Docker (RECOMMENDED & EASIEST)

If you have Docker installed, just run this:

docker run -d --hostname rabbit-host --name rabbitmq \

-p 5672:5672 -p 15672:15672 \

rabbitmq:3-management

This does the following:

|  |  |
| --- | --- |
| **Port** | **Purpose** |
| 5672 | Messaging protocol (AMQP) |
| 15672 | RabbitMQ Management Dashboard |

Then open: <http://localhost:15672>

* **Username:** guest
* **Password:** guest

Option 2: Install RabbitMQ Locally (Windows/Linux/Mac)

**Steps (Windows):**

1. Install **Erlang** from:  
   https://www.erlang.org/downloads
2. Install **RabbitMQ Server** from:  
   https://www.rabbitmq.com/install-windows.html
3. Start the service from **RabbitMQ Command Prompt** or services.msc.
4. Visit: <http://localhost:15672>

This method is heavier than Docker and requires system config. Not recommended unless Docker is not an option.

**Option 3: Use RabbitMQ in the Cloud**

For advanced/prod use (not local dev), you can also:

* Use **CloudAMQP** (free RabbitMQ cloud instance)
* Use **AWS MQ**, **Azure Service Bus with AMQP**, etc.

For now, stick to Docker.

**How to Test If RabbitMQ Is Working**

Once RabbitMQ is running:

1. Visit http://localhost:15672
2. Login with guest / guest
3. Your MassTransit app (Microservice A) should connect automatically
4. When you hit your /create-order endpoint, a message should appear in the **exchange** and can be routed to a queue (if you configure a consumer).

**Checklist for You**

|  |  |
| --- | --- |
| **Task** | **Status** |
| Install Docker (or RabbitMQ manually) | Do this now |
| Start RabbitMQ |  |
| Open RabbitMQ Dashboard | Check localhost:15672 |
| Run Microservice A |  |
| Hit /create-order endpoint |  |
| Verify message shows in RabbitMQ | (Dashboard > Exchanges) |

**Q. Also tell me the EF migration commands**

**Prerequisites**

Make sure your project has:

* EF Core tools installed
* A valid connection string in appsettings.json or your environment
* The correct DbContext registered (you already have OrderDbContext)

**Install EF Core Tools (if not installed)**

Use this **only once** globally (or per project):

dotnet tool install --global dotnet-ef

If already installed, you can update it:

dotnet tool update --global dotnet-ef

**Common EF Core Commands**

1. **Add a Migration**

dotnet ef migrations add InitialCreate -p MicroServiceA.csproj

* InitialCreate is the migration name (can be anything).
* -p specifies the project file where the DbContext is located.

If your solution has multiple projects, you may also need -s (startup project):

dotnet ef migrations add InitialCreate -p MicroServiceA.csproj -s MicroServiceA.csproj

1. **Apply the Migration to the Database**

dotnet ef database update -p MicroServiceA.csproj -s MicroServiceA.csproj

This creates the database (if not already present) and applies all migrations.

1. **List Existing Migrations**

dotnet ef migrations list -p MicroServiceA.csproj -s MicroServiceA.csproj

1. **Remove Last Migration (if needed)**

dotnet ef migrations remove -p MicroServiceA.csproj -s MicroServiceA.csproj

Only works if the migration has **not been applied** yet (i.e., database update hasn't been run after adding).

Example Connection String (in appsettings.json)

{

"ConnectionStrings": {

"OrderDatabase": "Server=localhost;Database=OrdersDb;Trusted\_Connection=True;TrustServerCertificate=True"

}

}

Make sure it matches the one used in your Program.cs:

builder.Services.AddDbContext<OrderDbContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("OrderDatabase")));

**Troubleshooting Tips**

|  |  |
| --- | --- |
| **Issue** | **Solution** |
| Unable to create an object of type 'OrderDbContext' | Make sure it has a public constructor with DbContextOptions<OrderDbContext> |
| dotnet ef not found | Ensure dotnet-ef tool is installed (dotnet tool install --global dotnet-ef) |
| SQL errors | Check your connection string and ensure SQL Server is running locally |

**Q. I was able to create database as well. And changed the code as per your recommendation. How can I test this code? Tell me steps for that as well.**

You're testing this endpoint in Microservice A:

app.MapPost("/create-order", async (IPublishEndpoint publishEndpoint) =>

{

var orderId = Guid.NewGuid();

await publishEndpoint.Publish(new OrderCreatedEvent(orderId, "Sample Product"));

return Results.Ok($"Order {orderId} created.");

});

It should:

1. Create a new OrderCreatedEvent
2. Publish it to RabbitMQ
3. (Later) Microservices B/C can consume this event

**Step-by-Step Testing Guide**

Step 1: Make Sure RabbitMQ is Running

docker ps

If it's not running:

docker run -d --hostname rabbit-host --name rabbitmq \

-p 5672:5672 -p 15672:15672 \

rabbitmq:3-management

Then go to:  
👉 <http://localhost:15672>  
Login: guest / guest

Step 2: Run Microservice A

From inside MicroServiceA project folder:

dotnet run

Check the logs to ensure it connects to RabbitMQ. You should see something like:

MassTransit started

Connected to RabbitMQ

Step 3: Call the /create-order Endpoint

Use any of the following:

**Option A: Curl**

curl -X POST <http://localhost:5000/create-order>

(Adjust port if it’s different)

**Option B: Postman or Thunder Client**

* URL: http://localhost:5000/create-order
* Method: POST
* Body: None

**Option C: Browser**

Just navigate to:

<http://localhost:5000/create-order>

(POST won’t work from address bar, but some browsers may prompt)

Step 4: Check the Response

You should get something like:

Order e1b6dabc-00a3-4b6f-a184-xxxxxxxxxxxx created.

Step 5: Check RabbitMQ Management UI

1. Open: http://localhost:15672
2. Go to Exchanges
3. Look for an exchange name like ordercreatedevent (MassTransit auto-creates it)
4. If you had any consumers running (like B or C), queues would be bound and show messages

Optional: Add Temporary Consumer to Test It Works

To test the message actually *arrives*, add this dummy consumer in Microservice A:

x.AddConsumer<OrderCreatedEventTestConsumer>();

// Consumer class

public class OrderCreatedEventTestConsumer : IConsumer<OrderCreatedEvent>

{

public Task Consume(ConsumeContext<OrderCreatedEvent> context)

{

Console.WriteLine($"Received: {context.Message.OrderId} - {context.Message.Product}");

return Task.CompletedTask;

}

}

Then you’ll see the message printed in the console when you hit /create-order.

What's Next?

|  |  |
| --- | --- |
| **Task** | **Description** |
| Test /create-order | Make sure publish works |
| Verify in RabbitMQ UI | Message/exchange is created |
| Add Microservice B | Listen for OrderCreatedEvent and process it |

**Q. But where I have to write the code to inserting the record to the database**

Great question! You're currently publishing an OrderCreatedEvent to RabbitMQ — but that alone doesn't insert anything into your database.

There are **two typical places** to insert an order into the database:

Option 1: Insert Into Database *Before* Publishing the Event

Modify your /create-order endpoint like this:

app.MapPost("/create-order", async (OrderDbContext dbContext, IPublishEndpoint publishEndpoint) =>

{

var orderId = Guid.NewGuid();

var product = "Sample Product";

// Insert into DB

var order = new Order

{

OrderId = orderId,

Product = product,

Status = "Created"

};

dbContext.Orders.Add(order);

await dbContext.SaveChangesAsync();

// Then publish event

await publishEndpoint.Publish(new OrderCreatedEvent(orderId, product));

return Results.Ok($"Order {orderId} created and event published.");

});

Option 2: Insert Into Database in a Consumer (Event-Driven)

This is more decoupled — **the event gets consumed, and the consumer inserts the data**.

Create a New Consumer:

public class OrderCreatedEventConsumer : IConsumer<OrderCreatedEvent>

{

private readonly OrderDbContext \_dbContext;

public OrderCreatedEventConsumer(OrderDbContext dbContext)

{

\_dbContext = dbContext;

}

public async Task Consume(ConsumeContext<OrderCreatedEvent> context)

{

var message = context.Message;

var order = new Order

{

OrderId = message.OrderId,

Product = message.Product,

Status = "Created"

};

\_dbContext.Orders.Add(order);

await \_dbContext.SaveChangesAsync();

}

}

Register It in Program.cs:

builder.Services.AddMassTransit(x =>

{

x.AddConsumer<OrderCreatedEventConsumer>();

x.AddConsumer<PaymentFailedConsumer>();

x.UsingRabbitMq((context, cfg) =>

{

cfg.Host("rabbitmq://localhost");

cfg.ReceiveEndpoint("order-created-queue", e =>

{

e.ConfigureConsumer<OrderCreatedEventConsumer>(context);

});

});

});

Which One Should You Use?

|  |  |  |
| --- | --- | --- |
| **Approach** | **When to Use** | **Notes** |
| Insert before publishing | Simpler, good for single service setup | Transactional logic is in same place |
| Insert in consumer | Preferred in microservice environments | Loosely coupled, more scalable, but needs eventual consistency |

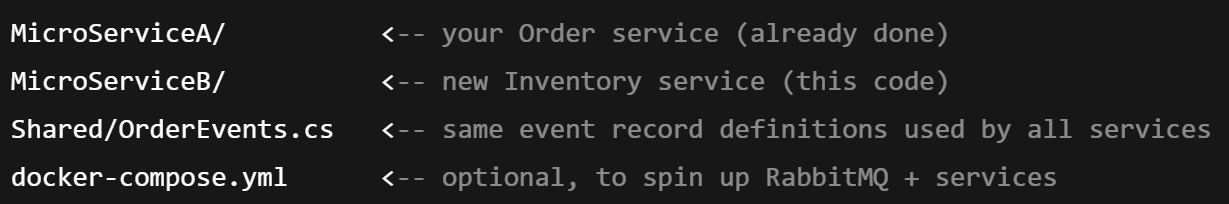
**Q. The create-order microservice is done, now which microservice should I create next?**

Below is a **complete, working template for “Microservice B – Inventory Service.”**  
It does two things:

1. **Consumes** OrderCreatedEvent that Microservice A publishes.
2. **Publishes** back either InventoryReservedEvent (success) or InventoryFailedEvent (out‑of‑stock).

You can keep the Swagger / controller skeleton you already had, but the real work now happens in the consumer.

Directory / Project layout (what you need)



The Shared class library just contains the OrderEvents record types so every microservice references the same definitions.

Shared ‑ OrderEvents.cs *(unchanged)*

public static class OrderEvents

{

public record OrderCreatedEvent(Guid OrderId, string Product);

public record InventoryReservedEvent(Guid OrderId);

public record InventoryFailedEvent(Guid OrderId, string Reason);

public record PaymentProcessedEvent(Guid OrderId);

public record PaymentFailedEvent(Guid OrderId, string Reason);

}

Reference this library from **both** projects:

dotnet add MicroServiceA reference Shared

dotnet add MicroServiceB reference Shared

Microservice B ‑ Program.cs

using MassTransit;

using Microsoft.OpenApi.Models;

using System.Diagnostics;

using System.Text.Json.Serialization;

using Shared; // <-- we need the events

using MicroServiceB.Consumers;

var builder = WebApplication.CreateBuilder(args);

// ------------------------------------------------------------------

// 1. MVC + Swagger (what you already had)

// ------------------------------------------------------------------

builder.Services.AddControllers()

.AddJsonOptions(o => o.JsonSerializerOptions.ReferenceHandler = ReferenceHandler.IgnoreCycles);

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(c => c.SwaggerDoc("v1", new OpenApiInfo

{

Title = "Microservice B – Inventory Service",

Version = "v1"

}));

// ------------------------------------------------------------------

// 2. MassTransit + RabbitMQ

// ------------------------------------------------------------------

builder.Services.AddMassTransit(x =>

{

// register the consumer

x.AddConsumer<OrderCreatedConsumer>();

x.UsingRabbitMq((context, cfg) =>

{

// \*\*\* IMPORTANT: use host in the modern format \*\*\*

cfg.Host("localhost", "/", h =>

{

h.Username("guest");

h.Password("guest");

});

// bind consumer to its queue

cfg.ReceiveEndpoint("inventory-service-queue", e =>

{

e.ConfigureConsumer<OrderCreatedConsumer>(context);

});

});

});

// optional but nice – same timeout options you used in A

builder.Services.Configure<MassTransitHostOptions>(opt =>

{

opt.WaitUntilStarted = true;

opt.StartTimeout = TimeSpan.FromSeconds(30);

opt.StopTimeout = TimeSpan.FromMinutes(1);

});

var app = builder.Build();

// Swagger convenience (unchanged)

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "Microservice B v1"));

app.Lifetime.ApplicationStarted.Register(() =>

{

var url = "https://localhost:7074/swagger";

try { Process.Start(new ProcessStartInfo { FileName = url, UseShellExecute = true }); }

catch { /\* swallow \*/ }

});

}

app.UseHttpsRedirection();

app.MapControllers();

app.Run();

Microservice B ‑ Consumers/OrderCreatedConsumer.cs

using MassTransit;

using Shared;

using static Shared.OrderEvents;

namespace MicroServiceB.Consumers;

/// <summary>

/// Pretend we have a stock-checking subsystem.

/// If product name contains "Fail" we simulate out‑of‑stock.

/// </summary>

public class OrderCreatedConsumer : IConsumer<OrderCreatedEvent>

{

private readonly IPublishEndpoint \_publishEndpoint;

public OrderCreatedConsumer(IPublishEndpoint publishEndpoint)

{

\_publishEndpoint = publishEndpoint;

}

public async Task Consume(ConsumeContext<OrderCreatedEvent> context)

{

var msg = context.Message;

Console.WriteLine($"[Inventory] Received OrderCreatedEvent for {msg.OrderId} – {msg.Product}");

var inStock = !msg.Product.Contains("Fail", StringComparison.OrdinalIgnoreCase);

if (inStock)

{

Console.WriteLine($"[Inventory] Stock reserved ✔️");

await \_publishEndpoint.Publish(new InventoryReservedEvent(msg.OrderId));

}

else

{

Console.WriteLine($"[Inventory] Stock unavailable ❌");

await \_publishEndpoint.Publish(new InventoryFailedEvent(msg.OrderId, "Not enough stock"));

}

}

}

**(Optional) Keep or remove the controller**

Your existing controller was only used by A’s “/send” HTTP test.  
Feel free to **delete** it or leave it for ad‑hoc testing. It doesn’t hurt anything.

How to Test the Flow End‑to‑End

|  |  |  |
| --- | --- | --- |
| **Step** | **What to do** | **Expected** |
| 1. | Run RabbitMQ (Docker) |  |
| docker run -d -p 5672:5672 -p 15672:15672 rabbitmq:3-management | Management UI at <http://localhost:15672> (guest/guest) |  |
| 2. | dotnet run in **MicroServiceB** | Console shows MassTransit bus started |
| 3. | dotnet run in **MicroServiceA** | Bus started |
| 4. | POST to http://localhost:5000/create-order (A) | A responds “Order … created and event published.” |
| 5. | Watch B’s console | You’ll see [Inventory] Received OrderCreatedEvent … and then either “Stock reserved” or “Stock unavailable.” |
| 6. | In RabbitMQ UI → Queues | inventory-service-queue appears and message count is 0 (consumed) |

What’s Next?

* **Microservice C (Payments)** – consume InventoryReservedEvent, charge payment, then publish PaymentProcessedEvent or PaymentFailedEvent.
* **Wire PaymentFailedConsumer (already in A)** – to cancel the order when payment fails.
* **Add real DB to B** – table Inventory with product quantities, etc.

**Q. Do I need this (Shared/OrderEvents.cs) in Microservice B as well?**

Yes, **you do need the OrderEvents.cs file in Microservice B** — but not as a copy.

**Why?**

Microservice B needs to **understand the structure of events** that Microservice A publishes (like OrderCreatedEvent) so it can:

* Consume events from RabbitMQ using MassTransit.
* Deserialize the message correctly.
* Respond with new events (InventoryReservedEvent, etc.).

**Q. Tell me the command to run the MicroServiceA from command prompt (CMD) through docker?**

1. **Create a Dockerfile** inside MicroServiceA/

# Stage 1: Build

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

WORKDIR /app

# Copy csproj and restore

COPY \*.sln .

COPY MicroServiceA/\*.csproj ./MicroServiceA/

COPY Shared/\*.csproj ./Shared/

RUN dotnet restore

# Copy everything else and build

COPY . .

WORKDIR /app/MicroServiceA

RUN dotnet publish -c Release -o out

# Stage 2: Run

FROM mcr.microsoft.com/dotnet/aspnet:8.0

WORKDIR /app

COPY --from=build /app/MicroServiceA/out ./

ENTRYPOINT ["dotnet", "MicroServiceA.dll"]

2. **Build the Docker Image**

From your **solution root folder**, run:

docker build -t microservicea -f MicroServiceA/Dockerfile .

3. **Run the Container**

docker run -d -p 5000:80 --name microservicea microservicea

|  |  |
| --- | --- |
| **Port Mapping** | **Meaning** |
| -p 5000:80 | Maps Docker's port 80 → host port 5000 |

After this, you can visit:

<http://localhost:5000/create-order>

4. (Optional) Connect to RabbitMQ from Docker

If RabbitMQ is also running in Docker, your container won't see localhost.

Modify your Program.cs:

cfg.Host("rabbitmq", "/", h =>

{

h.Username("guest");

h.Password("guest");

});

This assumes you're using docker-compose where the service is named rabbitmq.

5. (Better) Use docker-compose to run all together

If you want RabbitMQ + Microservice A + B to run as a single system, I can give you a docker-compose.yml file.

**Q. I can see that MicroServiceA is running in the docker dashboard yet when I am trying to send a post request to the URL http://localhost:8080/create-order, I am getting response as Connection was refused by the server. How can I debug this issue?**

Great question — let’s walk through **how to debug your Dockerized Microservice A**, since it’s running but rejecting your HTTP request (Connection refused).

Step-by-Step Debug Checklist

1. **Verify Port Binding**

Run this in your terminal:

docker ps

Check that your **container is publishing the correct port**, like:

|  |  |  |
| --- | --- | --- |
| **CONTAINER ID** | **IMAGE** | **PORTS** |
| abc123 | Microservicea | 0.0.0.0:8080->80/tcp |

If it shows 8080->80, then you're good.

If **not**, you probably forgot to map ports correctly when running the container.

If ports are not mapped