

Below is **Part 2**, a detailed, hands-on, step-by-step guide to build and run the sample microservices solution from scratch.

Prerequisites

- [.NET 9 SDK](#)
 - Docker & Docker Compose
 - (Optional) Visual Studio 2022 / VS Code
-

1. Create the Solution and Folder Structure

```
mkdir EShop.Microservices && cd EShop.Microservices
dotnet new sln -n EShop.Microservices
```

```
mkdir Services
cd Services
```

```
# Create projects
dotnet new webapi -o IdentityService
dotnet new grpc -o CatalogService
dotnet new webapi -o BasketService
dotnet new webapi -o OrderService
dotnet new webapi -o PaymentService
dotnet new webapi -o ApiGateway
```

```
cd ..
dotnet sln add Services/IdentityService/IdentityService.csproj \
    Services/CatalogService/CatalogService.csproj \
    Services/BasketService/BasketService.csproj \
    Services/OrderService/OrderService.csproj \
    Services/PaymentService/PaymentService.csproj \
    Services/ApiGateway/ApiGateway.csproj
```

Your tree now looks like:

```
EShop.Microservices.sln
```

- L Services
 - | IdentityService
 - | CatalogService
 - | BasketService
 - | OrderService
 - | PaymentService
 - L ApiGateway

Let's build a dedicated API Gateway project with Ocelot in .NET 9, end-to-end:

1. Create the Gateway Project

cd Services

cd ApiGateway

This scaffolds a minimal ASP.NET Core Web API called ApiGateway.

2. Add Ocelot and Authentication Packages

dotnet add package Ocelot # API Gateway routing

**dotnet add package
Microsoft.AspNetCore.Authentication.JwtBearer**

- Ocelot handles route-mapping, aggregation, load balancing.
 - JWT Bearer lets the gateway validate tokens (from your IdentityService).
-

3. Define Ocelot Routes

Create a file ocelot.json in the project root:

```
{  
  "Routes": [  
    {  
      "RouteId": "Catalog",  
      "UpstreamPathTemplate": "/catalog/{**catchAll}",  
      "UpstreamHttpMethod": [ "GET", "POST", "PUT",  
"DELETE" ],  
      "DownstreamPathTemplate": "/{**catchAll}",  
      "DownstreamScheme": "http",
```

```
"DownstreamHostAndPorts": [
  { "Host": "catalogservice", "Port": 80 }
],
"AuthenticationOptions": {
  "AuthenticationProviderKey": "Bearer",
  "AllowedScopes": [ "api.read" ]
},
{
  "RouteId": "Basket",
  "UpstreamPathTemplate": "/basket/{**catchAll}",
  "UpstreamHttpMethod": [ "GET", "POST", "PUT",
"DELETE" ],
  "DownstreamPathTemplate":
"/api/basket/{**catchAll}",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
```

```
{ "Host": "basketservice", "Port": 80 }  
  
],  
  
"AuthenticationOptions": {  
  
"AuthenticationProviderKey": "Bearer",  
  
"AllowedScopes": [ "api.read", "api.write" ]  
  
}  
  
},  
  
{  
  
"RouteId": "Order",  
  
"UpstreamPathTemplate": "/order/{**catchAll}",  
  
"UpstreamHttpMethod": [ "GET", "POST" ],  
  
"DownstreamPathTemplate":  
"/api/order/{**catchAll}",  
  
"DownstreamScheme": "http",  
  
"DownstreamHostAndPorts": [  
  
{ "Host": "orderservice", "Port": 80 }
```

```
],  
  
  "AuthenticationOptions": {  
  
    "AuthenticationProviderKey": "Bearer",  
  
    "AllowedScopes": [ "api.read", "api.write" ]  
  
  },  
  
  {  
  
    "RouteId": "Payment",  
  
    "UpstreamPathTemplate": "/payment/{**catchAll}",  
  
    "UpstreamHttpMethod": [ "GET", "POST" ],  
  
    "DownstreamPathTemplate":  
    "/api/payment/{**catchAll}",  
  
    "DownstreamScheme": "http",  
  
    "DownstreamHostAndPorts": [  
  
      { "Host": "paymentservice", "Port": 80 }  
  
    ],  
  
  ],  
  
}
```

```
"AuthenticationOptions": {  
  "AuthenticationProviderKey": "Bearer",  
  "AllowedScopes": [ "api.write" ]  
}  
,  
{  
  "RouteId": "Identity_Connect",  
  "UpstreamPathTemplate": "/connect/{**catchAll}",  
  "UpstreamHttpMethod": [ "GET", "POST" ],  
  "DownstreamPathTemplate": "/connect/{**catchAll}",  
  "DownstreamScheme": "http",  
  "DownstreamHostAndPorts": [  
    { "Host": "identityservice", "Port": 80 }  
  ]  
},  
{
```

```
"RouteId": "Identity_OtherApis",  
"UpstreamPathTemplate": "/auth/{**catchAll}",  
"UpstreamHttpMethod": [ "GET", "POST", "PUT",  
"DELETE" ],  
"DownstreamPathTemplate": "/{**catchAll}",  
"DownstreamScheme": "http",  
"DownstreamHostAndPorts": [  
  { "Host": "identityservice", "Port": 80 }  
  ]  
  }  
],  
"GlobalConfiguration": {  
  "BaseUrl": "http://localhost:5000",  
  "RequestIdKey": "OcRequestId"  
}  
}
```


Key fields

- **UpstreamPathTemplate**: how clients call the gateway.
- **DownstreamPathTemplate**: where Ocelot forwards the request inside the cluster.
- **Host**: the Docker service name or DNS host.
- **AuthenticationOptions**: apply JWT validation per route.

4. Configure Program.cs

Edit Program.cs to wire up Ocelot and JWT validation:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
```

```
using Microsoft.IdentityModel.Tokens;
```

```
using Ocelot.DependencyInjection;
```

```
using Ocelot.Middleware;
```

```
var builder = WebApplication.CreateBuilder(args);
```

```
// 1) Load ocelot.json
```

```
builder.Configuration.AddJsonFile("ocelot.json",  
optional: false, reloadOnChange: true);
```

```
// 2) Configure JWT Bearer
```

```
builder.Services
```

```
.AddAuthentication(JwtBearerDefaults.AuthenticationSch  
eme)
```

```
.AddJwtBearer("Bearer", options =>
```

```
{
```

```
options.Authority =  
builder.Configuration["IdentityServer:Authority"]
```

```
?? "http://identityservice:80";
```

```
options.RequireHttpsMetadata = false;
```

```
options.TokenValidationParameters = new  
TokenValidationParameters
```

```
{
```

```
    ValidateAudience = false
```

```
};
```

```
});
```

```
// 3) Register Ocelot
```

```
builder.Services.AddOcelot(builder.Configuration);
```

```
var app = builder.Build();
```

```
app.UseRouting();
```

```
// 4) Enable Authentication + Authorization
```

```
app.UseAuthentication();
```

```
app.UseAuthorization();
```

```
// 5) Start Ocelot pipeline
```

```
app.Map("/", () => Results.Redirect("/swagger")); // optional
```

```
await app.UseOcelot();
```

```
app.Run();
```

- **Authority**: the URL of your IdentityServer (replace with your service name & port).
- **RequireHttpsMetadata = false**: for local Docker (no TLS).
- We call **UseAuthentication()** before **UseOcelot()** so Ocelot enforces JWTs.

5. Add Docker Support

Dockerfile

Place this in Services/ApiGateway/Dockerfile:

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

WORKDIR /app

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

WORKDIR /src

COPY *.csproj ./

RUN dotnet restore

COPY . .

RUN dotnet publish -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=build /app/publish .

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

docker-compose.yml Entry

In your docker-compose.yml at repo root, add under `services::`

api-gateway:

build: Services/ApiGateway

container_name: api-gateway

ports:

- "5000:80"

depends_on:

- identityservice

- catalogservice

- basketservice

- orderservice

environment:

Tell the gateway where IdentityServer lives

IdentityServer__Authority: "http://identityservice:80"

- **ports:** maps host **5000** → container **80**.
 - **depends_on:** ensures Identity, Catalog, etc. start first.
 - **IdentityServer__Authority** sets our JWT validation authority.
-

6. Run & Verify

Bring up everything:

```
docker-compose up --build api-gateway identityservice  
catalogservice basketservice orderservice
```

1.

Obtain a JWT (from your IdentityService):

```
curl -X POST http://localhost:5001/connect/token \  
  
-d  
"grant_type=password&client_id=client&client_secret=se
```

cred&username=alice&password=password&scope=api.read"

2.

Call through the Gateway:

**curl http://localhost:5000/catalog/api/products **

-H "Authorization: Bearer <your_access_token>"

3. You should see the product list forwarded from CatalogService.

Test a protected route (e.g. Basket):

**curl -X POST http://localhost:5000/basket/alice/items **

**-H "Authorization: Bearer <token>" **

**-H "Content-Type: application/json" **

-d "\"1\""

4. If your token is valid and includes the required scope, the gateway will forward it to BasketService.

- Service names in **DownstreamHostAndPorts** must exactly match your Docker Compose service names.
- Make sure ocelot.json is copied into the container (it's in the project root by default).
- JWT validation fails if **Authority** isn't reachable or metadata isn't exposed — check your IdentityService is running and exposing **.well-known/openid-configuration**.
- If you change ocelot.json, rebuild the container or mount it as a volume.

Below is a full, standalone **BasketService** implementation in .NET 9, covering:

1. Project creation
2. NuGet dependencies
3. Folder & file structure
4. Configuration (Redis + RabbitMQ + health checks + observability)
5. Models
6. Controller
7. Dockerfile

8. Docker Compose snippet

9. Testing

1. Create the Project

```
cd Services
```

```
dotnet new webapi -n BasketService
```

```
cd BasketService
```

2. Add NuGet Packages

```
dotnet add package Microsoft.Extensions.Caching.StackExchangeRedis
```

```
dotnet add package RabbitMQ.Client
```

```
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore
```

```
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
```

```
dotnet add package OpenTelemetry.Instrumentation.Http
```

```
dotnet add package Polly
```

```
dotnet add package Microsoft.Extensions.Http.Polly
```

- **StackExchangeRedis** for Redis-backed IDistributedCache
 - **RabbitMQ.Client** for publishing checkout events
 - **OpenTelemetry** + Prometheus for metrics
 - **Polly** for resilience (if you call other services later)
-

3. Folder & File Structure

BasketService/

├─ Controllers/

| └─ BasketController.cs

├─ Models/

| └─ Basket.cs

| └─ BasketItem.cs

├─ appsettings.json

├─ BasketService.csproj

├─ Program.cs

└─ Dockerfile

4. Configuration: **appsettings.json**

```
{  
  
  "Redis": {  
  
    "ConnectionString": "redis:6379"  
  
  },  
  
  "RabbitMQ": {  
  
    "HostName": "rabbitmq",  
  
    "UserName": "guest",  
  
    "Password": "guest",  
  
    "QueueName": "orderQueue"  
  
  },  
  
  "Logging": {  
  
    "LogLevel": {  
  
      "Default": "Information",  
  
      "Microsoft.Hosting.Lifetime": "Information"  
  
    }  
  
  }  
}
```

5. Program.cs

```
using BasketService.Models;

using Microsoft.Extensions.Caching.StackExchangeRedis;

using Microsoft.Extensions.Diagnostics.HealthChecks;

using OpenTelemetry.Metrics;

using OpenTelemetry.Trace;

using Polly;

using RabbitMQ.Client;

using System.Text;

using System.Text.Json;


var builder = WebApplication.CreateBuilder(args);


// 1) Configuration sections

var redisConfig = builder.Configuration.GetSection("Redis")["ConnectionString"];

var rmqConfig = builder.Configuration.GetSection("RabbitMQ");


// 2) Add Redis-backed cache
```

```
builder.Services.AddStackExchangeRedisCache(opts =>
```

```
{
```

```
    opts.Configuration = redisConfig;
```

```
    opts.InstanceName = "Basket_";
```

```
});
```

```
// 3) Add RabbitMQ connection factory
```

```
builder.Services.AddSingleton(sp => new ConnectionFactory
```

```
{
```

```
    HostName = rmqConfig["HostName"],
```

```
    UserName = rmqConfig["UserName"],
```

```
    Password = rmqConfig["Password"]
```

```
});
```

```
// 4) Controllers
```

```
builder.Services.AddControllers();
```

```
// 5) Health checks (Redis & self)
```

```
builder.Services.AddHealthChecks()
```

```
.AddRedis(redisConfig, name: "redis", failureStatus: HealthStatus.Unhealthy);
```

```
// 6) OpenTelemetry: tracing + metrics + Prometheus
```

```
builder.Services.AddOpenTelemetryTracing(tp => tp
```

```
.AddAspNetCoreInstrumentation()
```

```
.AddHttpClientInstrumentation()
```

```
.AddConsoleExporter());
```

```
builder.Services.AddOpenTelemetryMetrics(mp => mp
```

```
.AddAspNetCoreInstrumentation()
```

```
.AddHttpClientInstrumentation()
```

```
.AddPrometheusExporter());
```

```
var app = builder.Build();
```

```
// 7) Map health & metrics endpoints
```

```
app.MapHealthChecks("/health");
```

```
app.MapPrometheusScrapingEndpoint(); // exposes /metrics
```

```
// 8) Map controllers
```

```
app.MapControllers();
```

```
app.Run();
```

6. Models

Models/BasketItem.cs

```
namespace BasketService.Models
```

```
{
```

```
    public class BasketItem
```

```
    {
```

```
        public string ProductId { get; set; } = default!;
```

```
        public string ProductName { get; set; } = default!;
```

```
        public decimal Price { get; set; }
```

```
        public int Quantity { get; set; }
```

```
    }
```

```
}
```


Models/Basket.cs

```
namespace BasketService.Models

{

    public class Basket

    {

        public string UserId { get; set; } = default!;

        public List<BasketItem> Items { get; set; } = new();

    }

}
```

7. Controller: Controllers/BasketController.cs

```
using BasketService.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.Extensions.Caching.Distributed;

using RabbitMQ.Client;

using System.Text;

using System.Text.Json;
```

```
namespace BasketService.Controllers
```

```
{
```

```
    [ApiController]
```

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

```
    public class BasketController : ControllerBase
```

```
    {
```

```
        private readonly IDistributedCache _cache;
```

```
        private readonly ConnectionFactory _factory;
```

```
        private readonly string _queueName;
```

```
        public BasketController(IDistributedCache cache, IConfiguration config,  
                                ConnectionFactory factory)
```

```
        {
```

```
            _cache = cache;
```

```
            _factory = factory;
```

```
            _queueName = config.GetSection("RabbitMQ")["QueueName"]!;
```

```
        }
```

```
        // GET api/basket/{userId}
```

```
        [HttpGet("{userId}")]
```

```
public async Task<ActionResult<Basket>> Get(string userId)

{

    var data = await _cache.GetStringAsync(userId);

    if (string.IsNullOrEmpty(data))

        return Ok(new Basket { UserId = userId });

    return Ok(JsonSerializer.Deserialize<Basket>(data));

}
```

// POST api/basket

[HttpPost]

```
public async Task<ActionResult> Update([FromBody] Basket basket)

{

    var options = new DistributedCacheEntryOptions

    {

        SlidingExpiration = TimeSpan.FromHours(1)

    };

    var json = JsonSerializer.Serialize(basket);

    await _cache.SetStringAsync(basket.UserId, json, options);

    return NoContent();

}
```

```
}
```

```
// POST api/basket/{userId}/checkout
```

```
[HttpPost("{userId}/checkout")]
```

```
public async Task<ActionResult> Checkout(string userId)
```

```
{
```

```
    // 1) Retrieve basket
```

```
    var data = await _cache.GetStringAsync(userId);
```

```
    if (string.IsNullOrEmpty(data))
```

```
        return BadRequest("Basket is empty.");
```

```
    var basket = JsonSerializer.Deserialize<Basket>(data);
```

```
    // 2) Publish checkout event
```

```
    using var conn = _factory.CreateConnection();
```

```
    using var channel = conn.CreateModel();
```

```
    channel.QueueDeclare(_queueName, durable: true, exclusive: false,  
autoDelete: false);
```

```
    var body = Encoding.UTF8.GetBytes(JsonSerializer.Serialize(basket));
```

```
channel.BasicPublish(exchange: "",
                    routingKey: _queueName,
                    basicProperties: null,
                    body: body);

// 3) Optionally clear the basket

await _cache.RemoveAsync(userId);

return Accepted($"Checkout event for {userId} published.");
}
}
}
```

8. Dockerfile

Place this file at [Services/BasketService/Dockerfile](#):

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

```
WORKDIR /app
```

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

WORKDIR /src

COPY *.csproj ./

RUN dotnet restore

COPY . .

RUN dotnet publish -c Release -o /app/publish

FROM base AS final

WORKDIR /app

COPY --from=build /app/publish .

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

9. Docker Compose Snippet

Add to your root `docker-compose.yml`:

basketservice:

build: Services/BasketService

container_name: basketservice

ports:

- "5102:80"

depends_on:

- redis
- rabbitmq

environment:

- Redis__ConnectionString=redis:6379
- RabbitMQ__HostName=rabbitmq
- RabbitMQ__UserName=guest
- RabbitMQ__Password=guest
- RabbitMQ__QueueName=orderQueue

Also ensure you have `redis` and `rabbitmq` services defined:

redis:

image: redis:7-alpine

container_name: redis

ports:

- "6379:6379"

rabbitmq:

image: rabbitmq:3-management

container_name: rabbitmq

ports:

- "5672:5672"

- "15672:15672"

10. Testing

Start services

docker-compose up --build basketservice redis rabbitmq

1.

Create/Update basket

curl -X POST http://localhost:5102/api/basket \

-H "Content-Type: application/json" \

-d

'{"userId":"alice","items":[{"productId":"1","productName":"Widget","price":9.99,"quantity":2}]}'

2.

Retrieve basket

curl http://localhost:5102/api/basket/alice

3.

Checkout

```
curl -X POST http://localhost:5102/api/basket/alice/checkout
```

4. You should see in the **rabbitmq** logs (or management UI) a message in **orderQueue** containing the serialized basket.

5. Health & Metrics

- Health: `curl http://localhost:5102/health`
- Metrics: `curl http://localhost:5102/metrics`

Your **BasketService** is now fully functional: it stores carts in Redis, publishes checkout events to RabbitMQ, exposes health and Prometheus metrics, and can be container-run via Docker Compose.

Below is a complete **CatalogService** implementation in .NET 9 with both gRPC and HTTP endpoints, MongoDB persistence, health checks, observability, and Docker support.

1. Create the Project

```
cd Services
dotnet new grpc -n CatalogService
cd CatalogService
```

This scaffolds a gRPC-enabled ASP.NET Core project.

2. Add NuGet Packages

```
dotnet add package Grpc.AspNetCore
dotnet add package Grpc.Tools
dotnet add package MongoDB.Driver
dotnet add package AspNetCore.HealthChecks.MongoDb
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.Http
```

- **Grpc.Tools** (build-time) + **Grpc.AspNetCore** for gRPC
 - **MongoDB.Driver** for data storage
 - **AspNetCore.HealthChecks.MongoDb** for built-in Mongo health check
 - **OpenTelemetry** + Prometheus exporter
-

3. Folder & File Structure

```
CatalogService/
├── Controllers/
│   └── ProductsController.cs
├── Data/
│   ├── CatalogContext.cs
│   ├── IProductRepository.cs
│   └── ProductRepository.cs
├── Models/
│   └── Product.cs
├── Protos/
│   └── catalog.proto
├── Services/
│   └── CatalogServiceImpl.cs
├── appsettings.json
├── CatalogService.csproj
├── Program.cs
└── Dockerfile
```

4. **appsettings.json**

```
{
  "MongoSettings": {
    "ConnectionString": "mongodb://catalog-mongo:27017",
    "DatabaseName": "CatalogDb"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

5. CatalogService.csproj

Make sure the proto is compiled and the gRPC server stub is generated:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net9.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Grpc.AspNetCore" Version="2.58.0" />
    <PackageReference Include="Grpc.Tools" Version="2.58.0">
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="MongoDB.Driver" Version="2.21.0" />
    <PackageReference Include="AspNetCore.HealthChecks.MongoDb" Version="6.0.1" />
    <PackageReference Include="OpenTelemetry.Exporter.Prometheus.AspNetCore"
Version="1.5.0" />
    <PackageReference Include="OpenTelemetry.Instrumentation.AspNetCore" Version="1.5.0"
/>
    <PackageReference Include="OpenTelemetry.Instrumentation.Http" Version="1.5.0" />
  </ItemGroup>

  <ItemGroup>
    <Protobuf Include="Protos\catalog.proto" GrpcServices="Server" />
  </ItemGroup>
</Project>
```

6. Protobuf Contract: **Protos/catalog.proto**

```
syntax = "proto3";
option csharp_namespace = "CatalogGrpc";

package catalog;

service Catalog {
  rpc GetAll (Empty) returns (ProductList);
}

message Empty {}

message Product {
  string id      = 1;
  string name    = 2;
  string description = 3;
  double price   = 4;
}

message ProductList {
  repeated Product items = 1;
}
```

7. MongoDB Context & Repository

Data/CatalogContext.cs

```
using MongoDB.Driver;
using CatalogService.Models;

namespace CatalogService.Data
{
    public class CatalogContext
    {
        private readonly IMongoDatabase _db;
        public CatalogContext(IConfiguration config)
        {
            var settings = config.GetSection("MongoSettings");
            var client = new MongoClient(settings["ConnectionString"]);
            _db = client.GetDatabase(settings["DatabaseName"]);
        }
    }
}
```

```
        public IMongoCollection<Product> Products => _db.GetCollection<Product>("Products");
    }
}
```

Data/IProductRepository.cs

```
using CatalogService.Models;

namespace CatalogService.Data
{
    public interface IProductRepository
    {
        Task<IEnumerable<Product>> GetAllAsync();
    }
}
```

Data/ProductRepository.cs

```
using CatalogService.Models;
using MongoDB.Driver;

namespace CatalogService.Data
{
    public class ProductRepository : IProductRepository
    {
        private readonly CatalogContext _context;
        public ProductRepository(CatalogContext context) => _context = context;

        public async Task<IEnumerable<Product>> GetAllAsync()
        {
            return await _context.Products.Find(_ => true).ToListAsync();
        }
    }
}
```

8. Domain Model: Models/Product.cs

```
namespace CatalogService.Models
{
    public class Product
```

```

{
    public string Id      { get; set; } = default!;
    public string Name    { get; set; } = default!;
    public string Description { get; set; } = default!;
    public double Price   { get; set; }
}
}

```

9. gRPC Service Implementation:

Services/CatalogServiceImpl.cs

```

using CatalogGrpc;
using Grpc.Core;
using CatalogService.Data;
using CatalogService.Models;

namespace CatalogService.Services
{
    public class CatalogServiceImpl : Catalog.CatalogBase
    {
        private readonly IProductRepository _repo;
        public CatalogServiceImpl(IProductRepository repo) => _repo = repo;

        public override async Task<ProductList> GetAll(Empty request, ServerCallContext context)
        {
            var reply = new ProductList();
            var products = await _repo.GetAllAsync();
            foreach (var p in products)
            {
                reply.Items.Add(new CatalogGrpc.Product {
                    Id      = p.Id,
                    Name    = p.Name,
                    Description = p.Description,
                    Price   = p.Price
                });
            }
            return reply;
        }
    }
}

```

10. HTTP Controller:

Controllers/ProductsController.cs

```
using CatalogService.Data;
using CatalogService.Models;
using Microsoft.AspNetCore.Mvc;

namespace CatalogService.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase
    {
        private readonly IProductRepository _repo;
        public ProductsController(IProductRepository repo) => _repo = repo;

        [HttpGet]
        public async Task<IEnumerable<Product>> Get() => await _repo.GetAllAsync();

        [HttpGet("{id}")]
        public async Task<ActionResult<Product>> Get(string id)
        {
            var all = await _repo.GetAllAsync();
            var p = all.FirstOrDefault(x => x.Id == id);
            return p == null ? NotFound() : p;
        }
    }
}
```

11. Startup & Observability: Program.cs

```
using CatalogService.Data;
using CatalogService.Services;
using Microsoft.Extensions.Diagnostics.HealthChecks;
using OpenTelemetry.Metrics;
using OpenTelemetry.Trace;

var builder = WebApplication.CreateBuilder(args);
```

```

// 1) MongoDB settings & DI
builder.Services.Configure<MongoSettings>(
    builder.Configuration.GetSection("MongoSettings"));
builder.Services.AddSingleton<CatalogContext>();
builder.Services.AddScoped<IProductRepository, ProductRepository>();

// 2) gRPC & Controllers
builder.Services.AddGrpc();
builder.Services.AddGrpcReflection();
builder.Services.AddControllers();

// 3) Health checks (Mongo)
var mongoConn = builder.Configuration["MongoSettings:ConnectionString"];
builder.Services.AddHealthChecks()
    .AddMongoDb(mongoConn, name: "mongodb", failureStatus: HealthStatus.Unhealthy);

// 4) OpenTelemetry Tracing & Metrics
builder.Services.AddOpenTelemetryTracing(tp => tp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddConsoleExporter());

builder.Services.AddOpenTelemetryMetrics(mp => mp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddPrometheusExporter());

var app = builder.Build();

// 5) Endpoints
app.MapHealthChecks("/health");
app.MapPrometheusScrapingEndpoint(); // /metrics
app.MapGrpcService<CatalogServiceImpl>();
app.MapGrpcReflectionService();
app.MapControllers();

app.Run();

```

12. Dockerfile

Save as **CatalogService/Dockerfile**:


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

```
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src
COPY *.csproj ./
RUN dotnet restore
```

```
COPY . .
RUN dotnet publish -c Release -o /app/publish
```

```
FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "CatalogService.dll"]
```

13. Docker Compose Snippet

Add to your root **docker-compose.yml**:

```
catalogservice:
  build: Services/CatalogService
  container_name: catalogservice
  ports:
    - "5101:80"
  depends_on:
    - catalog-mongo
  environment:
    - MongoSettings__ConnectionString=mongodb://catalog-mongo:27017
    - MongoSettings__DatabaseName=CatalogDb
```

```
catalog-mongo:
  image: mongo:6
  container_name: catalog-mongo
  ports:
    - "27017:27017"
  volumes:
    - catalog-data:/data/db
```

```
volumes:
  catalog-data:
```

14. Seed Data & Testing

Start Mongo + Service

```
docker-compose up --build catalog-mongo catalogservice
```

1.

Seed some data (optional):

```
# In another shell, insert a product via mongo CLI
docker exec -it catalog-mongo mongo --eval \
'db.CatalogDb.Products.insertOne({ Name:"Widget", Description:"Demo", Price:9.99 })'
```

2.

Test gRPC:

```
grpcurl -plaintext localhost:5101 catalog.Catalog/GetAll
```

3.

Test HTTP:

```
curl http://localhost:5101/api/products
```

4.

Health & Metrics:

```
curl http://localhost:5101/health
curl http://localhost:5101/metrics
```

5.

Your **CatalogService** is now up and running with:

- **gRPC** endpoint (`GetAll`)
- **HTTP** API (`/api/products`)

- **MongoDB** persistence
- **Health check** at [/health](#)
- **Prometheus metrics** at [/metrics](#)
- **Dockerized** for easy orchestration.

Below is a complete **IdentityService** implementation in .NET 9 using Duende IdentityServer, with in-memory configuration and test users, health checks, and Docker support.

1. Create the Project

```
cd Services
dotnet new webapi -n IdentityService
cd IdentityService
```

2. Add NuGet Packages

```
dotnet add package Duende.IdentityServer
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
dotnet add package AspNetCore.HealthChecks.UI.Client
dotnet add package AspNetCore.HealthChecks.UI.InMemory.Storage
```

- **Duende.IdentityServer**: OAuth2/OpenID Connect server
 - **JwtBearer**: for downstream token validation (if you add any protected endpoints)
 - **HealthChecks.UI** packages: simple UI/dashboard for health checks
-

3. Folder & File Structure

```
IdentityService/
├─ Config/
```

```
|   L Config.cs
|--- Controllers/
|   L HealthController.cs
|--- appsettings.json
|--- IdentityService.csproj
|--- Program.cs
|--- Dockerfile
```

4. In-Memory Configuration: **Config/Config.cs**

```
using Duende.IdentityServer.Models;
using Duende.IdentityServer.Test;
using System.Collections.Generic;

namespace IdentityService.Config
{
    public static class Config
    {
        // API scopes represent what resources clients can request
        public static IEnumerable<ApiScope> ApiScopes =>
            new List<ApiScope>
            {
                new ApiScope("api.read", "Read access to protected APIs"),
                new ApiScope("api.write", "Write access to protected APIs")
            };

        // Identity resources (e.g. openid, profile)
        public static IEnumerable<IdentityResource> IdentityResources =>
            new List<IdentityResource>
            {
                new IdentityResources.OpenId(),
                new IdentityResources.Profile()
            };

        // Clients that can request tokens from our IdentityServer
        public static IEnumerable<Client> Clients =>
            new List<Client>
            {
                new Client
                {
                    ClientId = "client",
                    ClientName = "E-Shop Client",
                }
            }
        }
    }
```

```

        AllowedGrantTypes = GrantTypes.ResourceOwnerPassword,
        ClientSecrets = { new Secret("secret".Sha256()) },
        AllowedScopes = { "openid", "profile", "api.read", "api.write" }
    }
};

// Test users — DO NOT use in production
public static List<TestUser> TestUsers =>
    new List<TestUser>
    {
        new TestUser
        {
            SubjectId = "1",
            Username = "alice",
            Password = "password",
            Claims =
            {
                new("name", "Alice"),
                new("email", "alice@example.com")
            }
        },
        new TestUser
        {
            SubjectId = "2",
            Username = "bob",
            Password = "password",
            Claims =
            {
                new("name", "Bob"),
                new("email", "bob@example.com")
            }
        }
    };
}
}

```

5. Health Endpoint Controller:

Controllers/HealthController.cs

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Diagnostics.HealthChecks;

```

```

namespace IdentityService.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class HealthController : ControllerBase
    {
        private readonly HealthCheckService _hcService;
        public HealthController(HealthCheckService hcService) => _hcService = hcService;

        [HttpGet]
        public async Task<IActionResult> Get()
        {
            var report = await _hcService.CheckHealthAsync();
            var status = report.Status == HealthStatus.Healthy ? 200 : 503;
            return StatusCode(status, report);
        }
    }
}

```

6. App Settings: **appsettings.json**

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "HealthChecksUI": {
    "HealthChecks": [
      { "Name": "self", "Uri": "/health" }
    ],
    "EvaluationTimeInSeconds": 10
  }
}

```

7. Startup & IdentityServer Configuration: **Program.cs**

```

using Duende.IdentityServer;

```

```

using Duende.IdentityServer.Services;
using IdentityService.Config;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.Extensions.Diagnostics.HealthChecks;

var builder = WebApplication.CreateBuilder(args);

// 1) Add IdentityServer with in-memory config
builder.Services.AddIdentityServer(options =>
{
    options.EmitStaticAudienceClaim = true;
})
.AddInMemoryApiScopes(Config.ApiScopes)
.AddInMemoryIdentityResources(Config.IdentityResources)
.AddInMemoryClients(Config.Clients)
.AddTestUsers(Config.TestUsers);

// 2) (Optional) If you add any protected APIs in this service
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
.AddJwtBearer("Bearer", options =>
{
    options.Authority = "http://localhost:5001";
    options.RequireHttpsMetadata = false;
    options.TokenValidationParameters.ValidateAudience = false;
});

// 3) Add health checks for self
builder.Services.AddHealthChecks()
.AddCheck("self", () => HealthCheckResult.Healthy());

// 4) Add HealthChecks UI (in-memory)
builder.Services.AddHealthChecksUI()
.AddInMemoryStorage();

builder.Services.AddControllers();

var app = builder.Build();

// 5) Middleware pipeline
app.UseRouting();

// IdentityServer endpoints
app.UseIdentityServer();

```

```
// Health checks and UI
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapHealthChecks("/health", new HealthCheckOptions()
    {
        ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
    });
    endpoints.MapHealthChecksUI();
});

app.Run();
```

8. Dockerfile

Create **Services/IdentityService/Dockerfile**:

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

```
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src
COPY *.csproj ./
RUN dotnet restore
```

```
COPY . .
RUN dotnet publish -c Release -o /app/publish
```

```
FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "IdentityService.dll"]
```

9. Docker Compose Snippet

Add to your root **docker-compose.yml** under **services::**

```
identityservice:
  build: Services/IdentityService
```



```
container_name: identityservice
ports:
  - "5001:80"
```

10. Test It End-to-End

Start IdentityService

`docker-compose up --build identityservice`

- 1.
2. **Check health/UI**
 - Health JSON: <http://localhost:5001/health>
 - Health UI: <http://localhost:5001/healthchecks-ui>

Request a token

```
curl -X POST http://localhost:5001/connect/token \
-d
"grant_type=password&client_id=client&client_secret=secret&username=alice&password=password&scope=api.read"
```

3. You should receive a JSON response with an **access_token**.
4. **Use the token**
 - Any downstream service (via API Gateway) can now validate this JWT using [Authority=http://identityservice:80](http://identityservice:80).

Your **IdentityService** is now running as an OAuth2/OpenID Connect server with:

- In-memory clients, scopes, and test users
- [/connect/*](#) endpoints for token issuance

- Health and HealthChecks UI
- Docker and Docker Compose support

Next, point your API Gateway at <http://identityservice:80> as the **Authority** for JWT validation.

Below is a complete **OrderService** implementation in .NET 9, which:

- Exposes HTTP CRUD endpoints for orders
- Listens to RabbitMQ **orderQueue** events and persists them
- Uses PostgreSQL via EF Core
- Includes health checks, OpenTelemetry metrics/tracing
- Is containerized with Docker and runnable via Docker Compose

1. Create the Project

```
cd Services
dotnet new webapi -n OrderService
cd OrderService
```

2. Add NuGet Packages

```
dotnet add package RabbitMQ.Client
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
dotnet add package AspNetCore.HealthChecks.UI.Client
dotnet add package AspNetCore.HealthChecks.UI.InMemory.Storage
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.Http
```

- **RabbitMQ.Client** for consuming checkout events
 - **EF Core + Npgsql** for PostgreSQL ORM
 - **HealthChecks.UI** for self health dashboard
 - **OpenTelemetry** + Prometheus exporter
-

3. Folder & File Structure

```
OrderService/
├─ Config/
│   └─ RabbitMqSettings.cs
├─ Controllers/
│   └─ OrdersController.cs
├─ Data/
│   ├── OrderContext.cs
│   ├── IOrderRepository.cs
│   └─ OrderRepository.cs
├─ HostedServices/
│   └─ OrderConsumer.cs
├─ Models/
│   ├── Order.cs
│   └─ OrderItem.cs
├─ appsettings.json
├─ OrderService.csproj
├─ Program.cs
└─ Dockerfile
```

4. Configuration: **appsettings.json**

```
{
  "ConnectionStrings": {
    "OrderDatabase":
      "Host=orders-db;Database=orders;Username=postgres;Password=Pass123;"
  },
  "RabbitMq": {
    "HostName": "rabbitmq",
    "UserName": "guest",
```

```

    "Password": "guest",
    "QueueName": "orderQueue"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "HealthChecksUI": {
    "HealthChecks": [
      { "Name": "self", "Uri": "/health" },
      { "Name": "postgres", "Uri": "/health/postgres" }
    ],
    "EvaluationTimeInSeconds": 10
  }
}

```

5. Domain Models

Models/OrderItem.cs

```

namespace OrderService.Models
{
    public class OrderItem
    {
        public int Id { get; set; }
        public string ProductId { get; set; } = default!;
        public string ProductName { get; set; } = default!;
        public decimal Price { get; set; }
        public int Quantity { get; set; }

        // FK back to Order
        public int OrderId { get; set; }
        public Order Order { get; set; } = default!;
    }
}

```

Models/Order.cs

```

namespace OrderService.Models

```

```

{
    public class Order
    {
        public int      Id      { get; set; }
        public string    UserId  { get; set; } = default!;
        public DateTime  CreatedAt { get; set; }
        public List<OrderItem> Items { get; set; } = new();
    }
}

```

6. EF Core Context & Repository

Data/OrderContext.cs

```

using Microsoft.EntityFrameworkCore;
using OrderService.Models;

namespace OrderService.Data
{
    public class OrderContext : DbContext
    {
        public OrderContext(DbContextOptions<OrderContext> opts) : base(opts) { }

        public DbSet<Order> Orders { get; set; }
        public DbSet<OrderItem> OrderItems { get; set; }
    }
}

```

Data/IOrderRepository.cs

```

using OrderService.Models;

namespace OrderService.Data
{
    public interface IOrderRepository
    {
        Task<IEnumerable<Order>> GetAllAsync();
        Task<Order?>           GetByIdAsync(int id);
        Task<Order>             CreateAsync(Order order);
    }
}

```

Data/OrderRepository.cs

```
using Microsoft.EntityFrameworkCore;
using OrderService.Models;

namespace OrderService.Data
{
    public class OrderRepository : IOrderRepository
    {
        private readonly OrderContext _ctx;
        public OrderRepository(OrderContext ctx) => _ctx = ctx;

        public async Task<IEnumerable<Order>> GetAllAsync() =>
            await _ctx.Orders.Include(o => o.Items).ToListAsync();

        public async Task<Order?> GetByIdAsync(int id) =>
            await _ctx.Orders.Include(o => o.Items)
                .FirstOrDefaultAsync(o => o.Id == id);

        public async Task<Order> CreateAsync(Order order)
        {
            order.CreatedAt = DateTime.UtcNow;
            _ctx.Orders.Add(order);
            await _ctx.SaveChangesAsync();
            return order;
        }
    }
}
```

7. RabbitMQ Consumer Hosted Service

Config/RabbitMqSettings.cs

```
namespace OrderService.Config
{
    public class RabbitMqSettings
    {
        public string HostName { get; set; } = default!;
        public string UserName { get; set; } = default!;
        public string Password { get; set; } = default!;
    }
}
```

```

        public string QueueName { get; set; } = default!;
    }
}

```

HostedServices/OrderConsumer.cs

```

using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using OrderService.Config;
using OrderService.Data;
using OrderService.Models;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System.Text;
using System.Text.Json;

namespace OrderService.HostedServices
{
    public class OrderConsumer : BackgroundService
    {
        private readonly IOrderRepository _repo;
        private readonly RabbitMqSettings _settings;

        public OrderConsumer(IOrderRepository repo, IOptions<RabbitMqSettings> opts)
        {
            _repo = repo;
            _settings = opts.Value;
        }

        protected override Task ExecuteAsync(CancellationToken token)
        {
            {
                var factory = new ConnectionFactory
                {
                    HostName = _settings.HostName,
                    UserName = _settings.UserName,
                    Password = _settings.Password
                };
                var conn = factory.CreateConnection();
                var channel = conn.CreateModel();

                channel.QueueDeclare(_settings.QueueName, durable: true, exclusive: false,
autoDelete: false);

                var consumer = new EventingBasicConsumer(channel);

```

```

        consumer.Received += async (s, ea) =>
        {
            var json = Encoding.UTF8.GetString(ea.Body.ToArray());
            var basket = JsonSerializer.Deserialize<Order>(json)!;

            // Persist as new Order
            await _repo.CreateAsync(basket);
        };

        channel.BasicConsume(_settings.QueueName, autoAck: true, consumer: consumer);
        return Task.CompletedTask;
    }
}
}

```

We're reusing the `Order` model to deserialize the basket event; in real scenarios you might map a separate contract.

8. HTTP API Controller

Controllers/OrdersController.cs

```

using Microsoft.AspNetCore.Mvc;
using OrderService.Data;
using OrderService.Models;

namespace OrderService.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class OrdersController : ControllerBase
    {
        private readonly IOrderRepository _repo;
        public OrdersController(IOrderRepository repo) => _repo = repo;

        // GET api/orders
        [HttpGet]
        public async Task<IEnumerable<Order>> GetAll() =>
            await _repo.GetAllAsync();

        // GET api/orders/5
    }
}

```



```

[HttpGet("{id}")]
public async Task<ActionResult<Order>> Get(int id)
{
    var order = await _repo.GetByIdAsync(id);
    if (order is null) return NotFound();
    return order;
}

// POST api/orders
[HttpPost]
public async Task<ActionResult<Order>> Create([FromBody] Order order)
{
    var created = await _repo.CreateAsync(order);
    return CreatedAtAction(nameof(Get), new { id = created.Id }, created);
}
}
}

```

9. Program.cs (Wiring Everything)

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Diagnostics.HealthChecks;
using Microsoft.Extensions.Options;
using OrderService.Config;
using OrderService.Data;
using OrderService.HostedServices;
using OpenTelemetry.Metrics;
using OpenTelemetry.Trace;

var builder = WebApplication.CreateBuilder(args);

// 1) Bind settings
builder.Services.Configure<RabbitMqSettings>(builder.Configuration.GetSection("RabbitMq"));

// 2) EF Core + PostgreSQL
builder.Services.AddDbContext<OrderContext>(opts =>
    opts.UseNpgsql(builder.Configuration.GetConnectionString("OrderDatabase")));
builder.Services.AddScoped<IOrderRepository, OrderRepository>();

// 3) RabbitMQ consumer
builder.Services.AddHostedService<OrderConsumer>();

```

```

// 4) Controllers
builder.Services.AddControllers();

// 5) HealthChecks (self + PostgreSQL)
builder.Services.AddHealthChecks()
    .AddCheck("self", () => HealthCheckResult.Healthy())
    .AddNpgSql(builder.Configuration.GetConnectionString("OrderDatabase"), name: "postgres");

// 6) OpenTelemetry
builder.Services.AddOpenTelemetryTracing(tp => tp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddConsoleExporter());

builder.Services.AddOpenTelemetryMetrics(mp => mp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddPrometheusExporter());

var app = builder.Build();

// 7) Migrate DB at startup (optional)
using (var scope = app.Services.CreateScope())
{
    var db = scope.ServiceProvider.GetRequiredService<OrderContext>();
    db.Database.Migrate();
}

// 8) Map endpoints
app.MapHealthChecks("/health");
app.MapPrometheusScrapingEndpoint(); // /metrics
app.MapControllers();

app.Run();

```

10. Dockerfile

```

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

```

```

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

```

```
COPY *.csproj ./
RUN dotnet restore

COPY . .
RUN dotnet publish -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "OrderService.dll"]
```

11. Docker Compose Snippet

Add under **services**: in your root **docker-compose.yml**:

```
orders-db:
  image: postgres:14
  container_name: orders-db
  environment:
    - POSTGRES_DB=orders
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=Pass123
  ports:
    - "5432:5432"
  volumes:
    - orders-data:/var/lib/postgresql/data
```

```
orderservice:
  build: Services/OrderService
  container_name: orderservice
  ports:
    - "5103:80"
  depends_on:
    - orders-db
    - rabbitmq
  environment:
    -
```

```
ConnectionString__OrderDatabase=Host=orders-db;Database=orders;Username=postgres;Password=Pass123;
  - RabbitMq__HostName=rabbitmq
  - RabbitMq__UserName=guest
  - RabbitMq__Password=guest
```

- RabbitMq__QueueName=orderQueue

volumes:

orders-data:

12. Testing

Start dependencies

docker-compose up --build orders-db rabbitmq

1.

Start OrderService

docker-compose up -d orderservice

2.

Send a basket event (simulate checkout)

Using rabbitmqadmin or code. Example using .NET CLI:
dotnet tool install --global RabbitMQ.Client.CLI
rabbitmqadmin publish routing_key=orderQueue
payload='{\"userId\":\"alice\",\"items\": [{\"productId\":\"1\",\"productName\":\"Widget\",\"price\":9.99,\"quantity\":2}]}'

3.

Query Orders

curl http://localhost:5103/api/orders

4.

Check health & metrics

curl http://localhost:5103/health
curl http://localhost:5103/metrics

5.

Your **OrderService** is now fully functional: it consumes checkout events from RabbitMQ, persists them into PostgreSQL with EF Core, exposes REST endpoints for order CRUD, and supports health checks and Prometheus metrics.

Below is a full **PaymentService** implementation in .NET 9, mirroring the patterns of the other services:

1. Create the Project

```
cd Services
dotnet new webapi -n PaymentService
cd PaymentService
```

2. Add NuGet Packages

```
dotnet add package RabbitMQ.Client
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
dotnet add package ASP.NET Core Health Checks UI Client
dotnet add package ASP.NET Core Health Checks UI InMemory Storage
dotnet add package OpenTelemetry.Exporter.Prometheus.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.AspNetCore
dotnet add package OpenTelemetry.Instrumentation.Http
```

- **RabbitMQ.Client** to consume “PaymentRequested” events
 - **EF Core + Npgsql** to persist payments into PostgreSQL
 - **HealthChecks.UI** for a health dashboard
 - **OpenTelemetry** + Prometheus exporter for metrics/tracing
-

3. Folder & File Structure

PaymentService/

```
|— Config/
|   └─ RabbitMqSettings.cs
|— Controllers/
|   └─ PaymentsController.cs
|— Data/
|   ├── PaymentContext.cs
|   ├── IPaymentRepository.cs
|   └─ PaymentRepository.cs
|— HostedServices/
|   └─ PaymentConsumer.cs
|— Models/
|   └─ Payment.cs
|— appsettings.json
|— PaymentService.csproj
|— Program.cs
└─ Dockerfile
```

4. Configuration: **appsettings.json**

```
{
  "ConnectionStrings": {
    "PaymentDatabase":
      "Host=payment-db;Database=payments;Username=postgres;Password=Pass123;"
  },
  "RabbitMq": {
    "HostName": "rabbitmq",
    "UserName": "guest",
    "Password": "guest",
    "QueueName": "paymentQueue"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "HealthChecksUI": {
    "HealthChecks": [
      { "Name": "self", "Uri": "/health" },
      { "Name": "postgres", "Uri": "/health/db" }
    ],
    "EvaluationTimeInSeconds": 10
  }
}
```

```
}  
}
```

5. Domain Model: **Models/Payment.cs**

```
namespace PaymentService.Models  
{  
    public class Payment  
    {  
        public int    Id          { get; set; }  
        public int    OrderId     { get; set; }  
        public string  UserId      { get; set; } = default!;  
        public decimal Amount     { get; set; }  
        public DateTime ProcessedAt { get; set; }  
        public string  Status      { get; set; } = default!;  
    }  
}
```

6. EF Core Context & Repository

Data/PaymentContext.cs

```
using Microsoft.EntityFrameworkCore;  
using PaymentService.Models;  
  
namespace PaymentService.Data  
{  
    public class PaymentContext : DbContext  
    {  
        public PaymentContext(DbContextOptions<PaymentContext> opts) : base(opts) { }  
  
        public DbSet<Payment> Payments { get; set; }  
    }  
}
```

Data/IPaymentRepository.cs

```
using PaymentService.Models;
```

```

namespace PaymentService.Data
{
    public interface IPaymentRepository
    {
        Task<Payment> CreateAsync(Payment payment);
        Task<IEnumerable<Payment>> GetAllAsync();
    }
}

```

Data/PaymentRepository.cs

```

using PaymentService.Models;
using Microsoft.EntityFrameworkCore;

namespace PaymentService.Data
{
    public class PaymentRepository : IPaymentRepository
    {
        private readonly PaymentContext _ctx;
        public PaymentRepository(PaymentContext ctx) => _ctx = ctx;

        public async Task<Payment> CreateAsync(Payment payment)
        {
            payment.ProcessedAt = DateTime.UtcNow;
            _ctx.Payments.Add(payment);
            await _ctx.SaveChangesAsync();
            return payment;
        }

        public async Task<IEnumerable<Payment>> GetAllAsync() =>
            await _ctx.Payments.AsNoTracking().ToListAsync();
    }
}

```

7. RabbitMQ Consumer Hosted Service

Config/RabbitMqSettings.cs

```

namespace PaymentService.Config
{
    public class RabbitMqSettings

```



```

{
    public string HostName { get; set; } = default!;
    public string UserName { get; set; } = default!;
    public string Password { get; set; } = default!;
    public string QueueName { get; set; } = default!;
}
}

```

HostedServices/PaymentConsumer.cs

```

using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using PaymentService.Config;
using PaymentService.Data;
using PaymentService.Models;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System.Text;
using System.Text.Json;

namespace PaymentService.HostedServices
{
    public class PaymentConsumer : BackgroundService
    {
        private readonly IPaymentRepository _repo;
        private readonly RabbitMqSettings _settings;

        public PaymentConsumer(IPaymentRepository repo, IOptions<RabbitMqSettings> opts)
        {
            _repo = repo;
            _settings = opts.Value;
        }

        protected override Task ExecuteAsync(CancellationToken token)
        {
            var factory = new ConnectionFactory
            {
                HostName = _settings.HostName,
                UserName = _settings.UserName,
                Password = _settings.Password
            };
            var conn = factory.CreateConnection();
            var channel = conn.CreateModel();

```

```

        channel.QueueDeclare(_settings.QueueName, durable: true, exclusive: false,
autoDelete: false);

        var consumer = new EventingBasicConsumer(channel);
        consumer.Received += async (s, ea) =>
        {
            var json = Encoding.UTF8.GetString(ea.Body.ToArray());
            // Expect event payload: { orderId, userId, amount }
            var evt = JsonSerializer.Deserialize<Payment>(json)!;

            // Process payment (stub) and persist
            evt.Status = "Processed";
            await _repo.CreateAsync(evt);
        };

        channel.BasicConsume(_settings.QueueName, autoAck: true, consumer: consumer);
        return Task.CompletedTask;
    }
}

```

Here we reuse the `Payment` model as our event contract. In a real system you might define a separate DTO.

8. HTTP API Controller

Controllers/PaymentsController.cs

```

using Microsoft.AspNetCore.Mvc;
using PaymentService.Data;
using PaymentService.Models;

namespace PaymentService.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class PaymentsController : ControllerBase
    {
        private readonly IPaymentRepository _repo;
        public PaymentsController(IPaymentRepository repo) => _repo = repo;
    }
}

```

```

// GET api/payments
[HttpGet]
public async Task<IEnumerable<Payment>> GetAll() =>
    await _repo.GetAllAsync();

// POST api/payments/process
// Accepts a payment request (if you want manual triggers)
[HttpPost("process")]
public async Task<ActionResult<Payment>> Process([FromBody] Payment req)
{
    req.Status = "Processed";
    var payment = await _repo.CreateAsync(req);
    return CreatedAtAction(nameof(GetAll), new { id = payment.Id }, payment);
}
}
}

```

9. Startup & Observability: Program.cs

```

using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Diagnostics.HealthChecks;
using PaymentService.Config;
using PaymentService.Data;
using PaymentService.HostedServices;
using OpenTelemetry.Metrics;
using OpenTelemetry.Trace;

var builder = WebApplication.CreateBuilder(args);

// 1) Bind RabbitMQ settings
builder.Services.Configure<RabbitMqSettings>(builder.Configuration.GetSection("RabbitMq"));

// 2) EF Core + PostgreSQL
builder.Services.AddDbContext<PaymentContext>(opts =>
    opts.UseNpgsql(builder.Configuration.GetConnectionString("PaymentDatabase")));
builder.Services.AddScoped<IPaymentRepository, PaymentRepository>();

// 3) RabbitMQ consumer hosted service
builder.Services.AddHostedService<PaymentConsumer>();

// 4) Controllers
builder.Services.AddControllers();

```

```

// 5) Health Checks (self + PostgreSQL)
builder.Services.AddHealthChecks()
    .AddCheck("self", () => HealthCheckResult.Healthy())
    .AddNpgsql(builder.Configuration.GetConnectionString("PaymentDatabase"), name: "db");

// 6) OpenTelemetry
builder.Services.AddOpenTelemetryTracing(tp => tp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddConsoleExporter());

builder.Services.AddOpenTelemetryMetrics(mp => mp
    .AddAspNetCoreInstrumentation()
    .AddHttpClientInstrumentation()
    .AddPrometheusExporter());

var app = builder.Build();

// 7) Migrate DB on startup (optional)
using(var scope = app.Services.CreateScope())
{
    scope.ServiceProvider.GetRequiredService<PaymentContext>().Database.Migrate();
}

// 8) Map endpoints
app.MapHealthChecks("/health");
app.MapHealthChecks("/health/db", new HealthCheckOptions { Predicate = r => r.Name == "db"
});
app.MapPrometheusScrapingEndpoint(); // /metrics
app.MapControllers();

app.Run();

```

10. Dockerfile

Save as **PaymentService/Dockerfile**:

```

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

```

```

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

```

```
WORKDIR /src
COPY *.csproj ./
RUN dotnet restore

COPY . .
RUN dotnet publish -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "PaymentService.dll"]
```

11. Docker Compose Snippet

Add under **services**: in your root **docker-compose.yml**:

```
payment-db:
  image: postgres:14
  container_name: payment-db
  environment:
    - POSTGRES_DB=payments
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=Pass123
  ports:
    - "5433:5432"
  volumes:
    - payment-data:/var/lib/postgresql/data
```

```
paymentservice:
  build: Services/PaymentService
  container_name: paymentservice
  ports:
    - "5104:80"
  depends_on:
    - payment-db
    - rabbitmq
  environment:
    -
```

```
ConnectionString__PaymentDatabase=Host=payment-db;Database=payments;Username=postgres;Password=Pass123;
  - RabbitMq__HostName=rabbitmq
  - RabbitMq__UserName=guest
```

- RabbitMq__Password=guest
- RabbitMq__QueueName=paymentQueue

volumes:

payment-data:

12. Testing

Start dependencies

`docker-compose up --build payment-db rabbitmq`

1.

Start PaymentService

`docker-compose up -d paymentservice`

2.

Simulate a payment request

```
curl -X POST http://localhost:5104/api/payments/process \  
-H "Content-Type: application/json" \  
-d '{"orderId":1,"userId":"alice","amount":19.95}'
```

3.

Or send an event to **paymentQueue**

```
rabbitmqadmin publish routing_key=paymentQueue \  
payload='{"orderId":1,"userId":"alice","amount":19.95}'
```

4.

Query records

`curl http://localhost:5104/api/payments`

5.

Check health & metrics

`curl http://localhost:5104/health`

```
curl http://localhost:5104/health/db
curl http://localhost:5104/metrics
```

6.

Your **PaymentService** is now fully operational, consuming events from RabbitMQ, persisting payments into PostgreSQL, exposing REST endpoints, and supporting health checks and Prometheus metrics.

Here's the updated **root** `docker-compose.yml` that brings up **all** services (plus their dependencies), followed by step-by-step instructions to build, run, and test via the API Gateway.

version: '3.8'

services:

— Identity

—
identityservice:

build: Services/IdentityService
container_name: identityservice
ports:
- "5001:80"

— Catalog + MongoDB

catalog-mongo:

image: mongo:6
container_name: catalog-mongo
ports:
- "27017:27017"
volumes:
- catalog-data:/data/db

catalogservice:

build: Services/CatalogService
container_name: catalogservice
ports:
- "5101:80"
depends_on:
- catalog-mongo

environment:

- MongoSettings__ConnectionString=mongodb://catalog-mongo:27017
- MongoSettings__DatabaseName=CatalogDb

——— Basket + Redis + RabbitMQ

redis:

- image: redis:7-alpine
- container_name: redis
- ports:
 - "6379:6379"

rabbitmq:

- image: rabbitmq:3-management
- container_name: rabbitmq
- ports:
 - "5672:5672"
 - "15672:15672"

basketservice:

- build: Services/BasketService
- container_name: basketservice
- ports:
 - "5102:80"
- depends_on:
 - redis
 - rabbitmq
- environment:
 - Redis__ConnectionString=redis:6379
 - RabbitMQ__HostName=rabbitmq
 - RabbitMQ__UserName=guest
 - RabbitMQ__Password=guest
 - RabbitMQ__QueueName=orderQueue

——— Order + Postgres

orders-db:

- image: postgres:14
- container_name: orders-db
- environment:
 - POSTGRES_DB=orders
 - POSTGRES_USER=postgres
 - POSTGRES_PASSWORD=Pass123
- ports:

- "5432:5432"

volumes:

- orders-data:/var/lib/postgresql/data

orderservice:

build: Services/OrderService

container_name: orderservice

ports:

- "5103:80"

depends_on:

- orders-db

- rabbitmq

environment:

-

ConnectionString__OrderDatabase=Host=orders-db;Database=orders;Username=postgres;Password=Pass123;

- RabbitMq__HostName=rabbitmq

- RabbitMq__UserName=guest

- RabbitMq__Password=guest

- RabbitMq__QueueName=orderQueue

——— Payment + Postgres

payment-db:

image: postgres:14

container_name: payment-db

environment:

- POSTGRES_DB=payments

- POSTGRES_USER=postgres

- POSTGRES_PASSWORD=Pass123

ports:

- "5433:5432"

volumes:

- payment-data:/var/lib/postgresql/data

paymentservice:

build: Services/PaymentService

container_name: paymentservice

ports:

- "5104:80"

depends_on:

- payment-db

- rabbitmq

environment:

-
ConnectionStrings__PaymentDatabase=Host=payment-db;Database=payments;Username=postgres;Password=Pass123;

- RabbitMq__HostName=rabbitmq
- RabbitMq__UserName=guest
- RabbitMq__Password=guest
- RabbitMq__QueueName=paymentQueue

— API Gateway

api-gateway:

build: Services/ApiGateway

container_name: api-gateway

ports:

- "5000:80"

depends_on:

- identityservice
- catalogservice
- basketservice
- orderservice
- paymentservice

environment:

- IdentityServer__Authority=http://identityservice:80

volumes:

catalog-data:

orders-data:

payment-data:

How to Run

Build & start all services

From the **project root** (where this `docker-compose.yml` lives):

`docker-compose up --build -d`

1. This will pull/build images, create volumes, and start your entire stack in detached mode.

Verify containers are healthy

`docker-compose ps`

2. You should see all services (Up) and ports mapped.

How to Test via the API Gateway

All calls go through the **API Gateway** on **port 5000**.

1. Get an Access Token

```
export TOKEN=$(
  curl -s -X POST http://localhost:5000/connect/token \
    -d
    "grant_type=password&client_id=client&client_secret=secret&username=alice&password=password&scope=api.read api.write" \
    | jq -r .access_token
)
```

2. Catalog: List Products

```
curl -H "Authorization: Bearer $TOKEN" \
  http://localhost:5000/catalog/api/products
```

3. Basket: Add Item, View, Checkout

```
# Add product "1" to Alice's basket
curl -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"1"}' \
  http://localhost:5000/basket/alice/items
```

```
# View basket
curl -H "Authorization: Bearer $TOKEN" \
  http://localhost:5000/basket/alice
```

```
# Checkout (publishes to RabbitMQ → OrderService)
curl -X POST \
  -H "Authorization: Bearer $TOKEN" \
```

`http://localhost:5000/basket/alice/checkout`

4. Orders: Verify Order Was Created

```
curl -H "Authorization: Bearer $TOKEN" \
  http://localhost:5000/order/api/orders
```

5. Payments: Process a Payment (Manual)

You can trigger a manual payment via HTTP, or have your OrderService publish to `paymentQueue`. For manual:

```
curl -X POST \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"orderId": 1, "userId": "alice", "amount": 19.95 }' \
  http://localhost:5000/payment/api/payments/process
```

Then view all payments:

```
curl -H "Authorization: Bearer $TOKEN" \
  http://localhost:5000/payment/api/payments
```

Direct Health & Metrics Endpoints

If you need health or Prometheus metrics for individual services, hit them directly on their ports:

Service	Health URL	Metrics URL
Identity	http://localhost:5001/health	—
Catalog	http://localhost:5101/health	http://localhost:5101/metrics
Basket	http://localhost:5102/health	http://localhost:5102/metrics
Order	http://localhost:5103/health	http://localhost:5103/metrics
Payment	http://localhost:5104/health	http://localhost:5104/metrics

Services/IdentityService/Dockerfile

```
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src

# Copy csproj and restore
COPY *.csproj ./
RUN dotnet restore

# Copy everything and publish
COPY . .
RUN dotnet publish -c Release -o /app/publish

# Runtime stage
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "IdentityService.dll"]
```

Services/CatalogService/Dockerfile

```
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src

COPY *.csproj ./
RUN dotnet restore

COPY . .
RUN dotnet publish -c Release -o /app/publish

# Runtime stage
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "CatalogService.dll"]
```

Services/BasketService/Dockerfile

```
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src

COPY *.csproj ./
RUN dotnet restore

COPY . .
RUN dotnet publish -c Release -o /app/publish

# Runtime stage
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "BasketService.dll"]
```

Services/OrderService/Dockerfile

```
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
WORKDIR /src

COPY *.csproj ./
RUN dotnet restore

COPY . .
RUN dotnet publish -c Release -o /app/publish

# Runtime stage
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime
WORKDIR /app
COPY --from=build /app/publish .
ENTRYPOINT ["dotnet", "OrderService.dll"]
```

Services/PaymentService/Dockerfile

```
# Build stage
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build
```

```
WORKDIR /src
```

```
COPY *.csproj ./  
RUN dotnet restore
```

```
COPY . .  
RUN dotnet publish -c Release -o /app/publish
```

```
# Runtime stage  
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime  
WORKDIR /app  
COPY --from=build /app/publish .  
ENTRYPOINT ["dotnet", "PaymentService.dll"]
```

Services/ApiGateway/Dockerfile

```
# Build stage  
FROM mcr.microsoft.com/dotnet/sdk:9.0 AS build  
WORKDIR /src
```

```
COPY *.csproj ./  
RUN dotnet restore
```

```
COPY . .  
RUN dotnet publish -c Release -o /app/publish
```

```
# Runtime stage  
FROM mcr.microsoft.com/dotnet/aspnet:9.0 AS runtime  
WORKDIR /app  
COPY --from=build /app/publish .  
ENTRYPOINT ["dotnet", "ApiGateway.dll"]
```

How to Build & Run

From the **solution root** (where your `docker-compose.yml` sits):

Rebuild images

```
docker-compose build
```

1.

Start all services

```
docker-compose up -d
```

2.

Verify

```
docker-compose ps
```

3.

Each service will now run its own container built from its Dockerfile. You can test via the API Gateway on port 5000 as described earlier.