Nice — this is a complete project spec. Below I’ll give you a compact, ready-to-copy implementation and all the commands / tests / troubleshooting notes you’ll need so the SalesService (port **8080**) and ApiGateway (Ocelot, port **8081**) run exactly as required.

I’ll include:

* Models and DbContext
* SalesService (ASP.NET Core Web API) — controller, Program.cs, appsettings, EF setup
* ApiGatewayService (Ocelot) — Program.cs and ocelot.json
* EF migration / run commands
* Minimal NUnit tests for the SalesController (GET & POST)
* Quick troubleshooting checklist

All code examples assume **.NET 6** (since dotnet-ef 6.0.6 is referenced). Adjust to .NET 7+ if needed.

**1 — SalesService (port 8080)**

**Project structure (dotnetapp/SalesService)**

/SalesService

/Controllers

SalesController.cs

/Data

AppDbContext.cs

/Models

Sales.cs

appsettings.json

Program.cs

SalesService.csproj

**Models/Sales.cs**

namespace SalesService.Models

{

public class Sales

{

public int SalesId { get; set; } // identity

public string Name { get; set; } = string.Empty;

public string Description { get; set; } = string.Empty;

public decimal Price { get; set; }

}

}

**Data/AppDbContext.cs**

using Microsoft.EntityFrameworkCore;

using SalesService.Models;

namespace SalesService.Data

{

public class AppDbContext : DbContext

{

public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }

public DbSet<Sales> Sales { get; set; } = null!;

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Sales>()

.HasKey(s => s.SalesId);

modelBuilder.Entity<Sales>()

.Property(s => s.SalesId)

.ValueGeneratedOnAdd();

}

}

}

**Controllers/SalesController.cs**

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using SalesService.Data;

using SalesService.Models;

namespace SalesService.Controllers

{

[ApiController]

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

public class SalesController : ControllerBase

{

private readonly AppDbContext \_db;

private readonly ILogger<SalesController> \_logger;

public SalesController(AppDbContext db, ILogger<SalesController> logger)

{

\_db = db;

\_logger = logger;

}

[HttpGet]

public async Task<ActionResult<IEnumerable<Sales>>> GetSales()

{

var list = await \_db.Sales.AsNoTracking().ToListAsync();

return Ok(list);

}

[HttpPost]

public async Task<ActionResult> CreateSale([FromBody] Sales sale)

{

if (sale == null) return BadRequest(new { message = "Sale data is required" });

\_db.Sales.Add(sale);

await \_db.SaveChangesAsync();

var response = new {

message = "Sale created successfully",

data = sale

};

// return created with route and body

return CreatedAtAction(nameof(GetSales), new { id = sale.SalesId }, response);

}

}

}

**appsettings.json (sample)**

{

"ConnectionStrings": {

"DefaultConnection": "User ID=sa;password=examlyMssql@123;server=localhost;Database=appdb;Trusted\_Connection=False;Persist Security Info=False;Encrypt=False"

},

"Kestrel": {

"Endpoints": {

"Http": {

"Url": "http://localhost:8080"

}

}

},

"Logging": { "LogLevel": { "Default": "Information", "Microsoft": "Warning" } }

}

**Program.cs**

using Microsoft.EntityFrameworkCore;

using SalesService.Data;

var builder = WebApplication.CreateBuilder(args);

// configuration & DB

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

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

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

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Ensure URL/Port binding: Kestrel configured via appsettings.json

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseRouting();

app.UseAuthorization();

app.MapControllers();

app.Run();

Note: the Kestrel URL in appsettings.json ensures the app listens on port 8080 when started with dotnet run. You can also set --urls http://localhost:8080 at runtime.

**2 — ApiGatewayService (Ocelot) (port 8081)**

**Project structure (dotnetapp/ApiGatewayService)**

/ApiGatewayService

Program.cs

ocelot.json

ApiGatewayService.csproj

appsettings.json

**ocelot.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/api/sales",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{ "Host": "localhost", "Port": 8080 }

],

"UpstreamPathTemplate": "/sales-api/sales",

"UpstreamHttpMethod": [ "GET", "POST" ]

},

{

"DownstreamPathTemplate": "/api/sales/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{ "Host": "localhost", "Port": 8080 }

],

"UpstreamPathTemplate": "/sales-api/sales/{everything}",

"UpstreamHttpMethod": [ "GET", "POST", "PUT", "DELETE" ]

}

],

"GlobalConfiguration": {

"RequestIdKey": "OcRequestId"

}

}

This routes /sales-api/sales to http://localhost:8080/api/sales. Adjust if you need different path handling.

**Program.cs (ApiGatewayService)**

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.Hosting;

using Ocelot.DependencyInjection;

using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

// Add Ocelot config file

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

// required for Ocelot

builder.Services.AddOcelot(builder.Configuration);

var app = builder.Build();

app.MapGet("/", () => "API Gateway running.");

await app.UseOcelot();

**appsettings.json (optional) — port binding**

{

"Kestrel": {

"Endpoints": {

"Http": {

"Url": "http://localhost:8081"

}

}

}

}

Start ApiGateway with dotnet run inside ApiGatewayService folder. Ensure SalesService is running first.

**3 — EF / DB setup commands**

From project root:

1. Navigate to SalesService:

cd dotnetapp/SalesService

1. Create local tool manifest & install dotnet-ef (per your note)

dotnet new tool-manifest

dotnet tool install --local dotnet-ef --version 6.0.6

1. Restore

dotnet restore

1. Add Migration & Update DB (example)

dotnet dotnet-ef migrations add InitialSetup

dotnet dotnet-ef database update

If using SQL Server CLI to create DB first:

sqlcmd -U sa -P "examlyMssql@123" -Q "CREATE DATABASE appdb"

Then run dotnet dotnet-ef database update.

**4 — Run both services (order matters)**

1. In one terminal (SalesService):

cd dotnetapp/SalesService

dotnet run

# or

dotnet run --urls http://localhost:8080

1. In another terminal (ApiGatewayService):

cd dotnetapp/ApiGatewayService

dotnet run

# or

dotnet run --urls http://localhost:8081

Check:

* Sales swagger: http://localhost:8080/swagger/index.html (if in dev and swagger enabled).
* Gateway base: http://localhost:8081/sales-api/sales should forward to http://localhost:8080/api/sales.

You mentioned a public swagger link — make sure SalesService is running and that the examly runner exposes the port as documented.

**5 — NUnit test examples (SalesService)**

Place tests in a separate test project (e.g., SalesService.Tests). Below are simple unit tests using EF InMemory to validate controller behavior.

**SalesControllerTests.cs**

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Logging.Abstractions;

using NUnit.Framework;

using SalesService.Controllers;

using SalesService.Data;

using SalesService.Models;

namespace SalesService.Tests

{

public class SalesControllerTests

{

private AppDbContext CreateInMemoryContext()

{

var options = new DbContextOptionsBuilder<AppDbContext>()

.UseInMemoryDatabase(databaseName: Guid.NewGuid().ToString())

.Options;

return new AppDbContext(options);

}

[Test]

public async Task GetSales\_ReturnsEmptyList\_WhenNoSales()

{

using var ctx = CreateInMemoryContext();

var controller = new SalesController(ctx, NullLogger<SalesController>.Instance);

var result = await controller.GetSales();

Assert.IsNotNull(result);

var ok = result.Result as Microsoft.AspNetCore.Mvc.OkObjectResult;

Assert.IsNotNull(ok);

var list = ok.Value as IEnumerable<Sales>;

Assert.IsNotNull(list);

Assert.IsEmpty(list);

}

[Test]

public async Task CreateSale\_ReturnsCreated\_WithSale()

{

using var ctx = CreateInMemoryContext();

var controller = new SalesController(ctx, NullLogger<SalesController>.Instance);

var sale = new Sales { Name = "Test", Description = "d", Price = 12.55M };

var actionResult = await controller.CreateSale(sale);

var created = actionResult as Microsoft.AspNetCore.Mvc.CreatedAtActionResult;

Assert.IsNotNull(created);

// Verify DB has one entry

var all = await ctx.Sales.ToListAsync();

Assert.AreEqual(1, all.Count);

Assert.AreEqual("Test", all[0].Name);

}

}

}

Run tests:

cd dotnetapp/SalesService.Tests

dotnet test

If test harness expects NUnit, ensure the test project's csproj references NUnit and Microsoft.EntityFrameworkCore.InMemory.

**6 — Common troubleshooting & checklist**

* Ports: Confirm SalesService listens on **8080** and ApiGateway on **8081**. Use --urls flag if appsettings binding doesn't take effect.
* DB: If migrations fail, ensure SQL Server is up, DB name matches connection string, and SA credentials are correct.
* CORS: If calling Gateway from browser, add CORS rules on SalesService or configure Ocelot CORS.
* Ocelot: If you change ocelot.json, restart ApiGateway or enable reload on change (we added reloadOnChange).
* Swagger: Confirm UseSwagger() is enabled in Development environment or call app.Environment.IsDevelopment() appropriately.
* Logging: Check logs for binding errors (Address already in use).
* Run order: Start SalesService first so Ocelot can forward requests.

**7 — Small notes to satisfy automated graders (from spec)**

* Ensure GET /api/sales returns **200 OK** with collection.
* Ensure POST /api/sales returns **201 Created** and a **custom JSON response** that indicates success (we return { message, data }).
* Ensure the ApiGateway routes /sales-api/sales to SalesService /api/sales for both GET & POST.
* Make sure both apps are running at ports **8080** and **8081** before running tests or clicking "Submit Project".

If you want, I can:

* paste ready-to-create csproj files for both projects and for the test project,
* provide a docker-compose.yml to run SQL Server + SalesService + ApiGateway,
* or generate Ocelot policies for load balancing or rate-limiting.