**Microservices with API Gateway — ready-to-run template (ProductService + Ocelot ApiGateway)**

Nice — below is a compact, copy-pasteable implementation and run instructions for the two projects you described: **ProductService** (port **8080**) and **ApiGatewayService** (port **8081** using Ocelot). It uses ASP.NET Core + EF Core (SQL Server by default, with instructions for InMemory if you prefer). I included all key files (models, DbContext, controller, Program.cs, appsettings, ocelot.json) plus commands you need to run migrations and start both services.

I’ve kept the code .NET 6/7-compatible (minimal host with controllers). Replace target framework in project files as needed.

**1) ProductService (port 8080)**

**Project structure (dotnetapp/ProductService)**

ProductService/

├─ Controllers/

│ └─ ProductController.cs

├─ Data/

│ └─ ProductDbContext.cs

├─ Models/

│ └─ Product.cs

├─ appsettings.json

└─ Program.cs

**Models/Product.cs**

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace ProductService.Models

{

public class Product

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int ProductId { get; set; }

[Required]

public string Name { get; set; }

public string Description { get; set; }

[Column(TypeName = "decimal(18,2)")]

public decimal Price { get; set; }

}

}

**Data/ProductDbContext.cs**

using Microsoft.EntityFrameworkCore;

using ProductService.Models;

namespace ProductService.Data

{

public class ProductDbContext : DbContext

{

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

public DbSet<Product> Products { get; set; }

}

}

**Controllers/ProductController.cs**

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using ProductService.Data;

using ProductService.Models;

namespace ProductService.Controllers

{

[ApiController]

[Route("api/product")]

public class ProductController : ControllerBase

{

private readonly ProductDbContext \_context;

private readonly ILogger<ProductController> \_logger;

public ProductController(ProductDbContext context, ILogger<ProductController> logger)

{

\_context = context;

\_logger = logger;

}

// GET: /api/product

[HttpGet]

public async Task<ActionResult<IEnumerable<Product>>> GetProducts()

{

var products = await \_context.Products.AsNoTracking().ToListAsync();

return Ok(products); // 200 OK

}

// POST: /api/product

[HttpPost]

public async Task<ActionResult<Product>> CreateProduct([FromBody] Product product)

{

if (product == null)

return BadRequest(new { message = "Product payload is required." });

\_context.Products.Add(product);

await \_context.SaveChangesAsync();

var customResponse = new

{

status = "success",

message = "Product created successfully",

data = product

};

// Return 201 Created and the created product

return CreatedAtAction(nameof(GetProducts), new { id = product.ProductId }, customResponse);

}

}

}

Note: CreatedAtAction used with GetProducts for location — tests usually expect 201 Created and a custom JSON response as specified.

**Program.cs**

using Microsoft.EntityFrameworkCore;

using ProductService.Data;

var builder = WebApplication.CreateBuilder(args);

// Read connection string from appsettings

var configuration = builder.Configuration;

var connectionString = configuration.GetConnectionString("DefaultConnection");

// Register DbContext (SQL Server). If connectionString is empty, fall back to InMemory for quick testing.

if (!string.IsNullOrWhiteSpace(connectionString) && !connectionString.Contains("INMEMORY"))

{

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

options.UseSqlServer(connectionString));

}

else

{

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

options.UseInMemoryDatabase("ProductDb"));

}

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

// Ensure database created (in production use migrations)

using (var scope = app.Services.CreateScope())

{

var db = scope.ServiceProvider.GetRequiredService<ProductDbContext>();

db.Database.EnsureCreated();

}

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseRouting();

app.UseAuthorization();

app.MapControllers();

// Force the app to bind to port 8080 when running via 'dotnet run' (and when launchSettings not set)

app.Urls.Add("http://localhost:8080");

app.Run();

**appsettings.json**

{

"ConnectionStrings": {

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

},

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

}

}

If you want to force InMemory DB for quick local testing, set "DefaultConnection": "INMEMORY".

**EF Core / Migrations notes**

If using SQL Server and EF Core 6 CLI as described in your instructions:

1. Create tool manifest (only once per repository):

dotnet new tool-manifest

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

1. Add migration and update DB:

dotnet dotnet-ef migrations add InitialSetup

dotnet dotnet-ef database update

If dotnet dotnet-ef complains, run from the ProductService project folder and ensure TargetFramework in .csproj aligns to EF tooling version.

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

**Project structure (dotnetapp/ApiGatewayService)**

ApiGatewayService/

├─ ocelot.json

├─ Program.cs

└─ appsettings.json

**ocelot.json**

{

"Routes": [

{

"DownstreamPathTemplate": "/api/product",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 8080

}

],

"UpstreamPathTemplate": "/product-api/product",

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

}

],

"GlobalConfiguration": {

"RequestIdKey": "OcRequestId"

}

}

This routes:

* GET /product-api/product -> GET http://localhost:8080/api/product
* POST /product-api/product -> POST http://localhost:8080/api/product

**Program.cs (ApiGatewayService)**

using Ocelot.DependencyInjection;

using Ocelot.Middleware;

var builder = WebApplication.CreateBuilder(args);

// Add ocelot configuration

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

// Register Ocelot

builder.Services.AddOcelot(builder.Configuration);

var app = builder.Build();

app.Urls.Add("http://localhost:8081");

await app.UseOcelot();

**appsettings.json (ApiGatewayService)**

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

}

}

**3) Commands to run (from repository root)**

Open two terminal windows.

**Terminal A — ProductService**

cd dotnetapp/ProductService

dotnet restore

dotnet build

# Optionally run migrations if using SQL Server:

# dotnet dotnet-ef migrations add InitialSetup

# dotnet dotnet-ef database update

dotnet run

# Service will listen on http://localhost:8080

**Terminal B — ApiGatewayService**

cd dotnetapp/ApiGatewayService

dotnet restore

dotnet build

dotnet run

# Api Gateway will listen on http://localhost:8081

**4) Test the endpoints**

**Direct ProductService**

* GET products: GET http://localhost:8080/api/product → 200 OK
* Create product: POST http://localhost:8080/api/product
  + Body (JSON):

{

"name": "Sample",

"description": "Test product",

"price": 99.99

}

* Response: HTTP 201 Created and the custom JSON response:

{

"status": "success",

"message": "Product created successfully",

"data": {

"productId": 1,

"name": "Sample",

"description": "Test product",

"price": 99.99

}

}

**Via ApiGateway (Ocelot)**

* GET products through gateway: GET http://localhost:8081/product-api/product → routed to ProductService
* POST product through gateway: POST http://localhost:8081/product-api/product → routed to ProductService

**5) Swagger / UI**

With the ProductService Program.cs using Swagger in Development, you can view:  
http://localhost:8080/swagger/index.html

Your project description included a platform URL of the pattern:  
https://8080-\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*.premiumproject.examly.io/swagger/index.html  
— that is the hosted preview link (ensure service is running and port mapping on platform is configured).

**6) Troubleshooting & tips**

* **Ports**: Make sure no other process occupies 8080/8081. If using Windows, netstat -ano | findstr :8080 to find conflicts.
* **CORS**: If browser calls to gateway are blocked, enable CORS in ProductService or Ocelot as required.
* **Connection string**: If using SQL Server container or local instance, ensure SQL Server is reachable and sa password matches examlyMssql@123.
* **EF Migrations**: If dotnet dotnet-ef not found, ensure you run dotnet tool install --local dotnet-ef --version 6.0.6 inside project root (as provided in your note).
* **Custom response**: Tests often assert 201 Created and a JSON response shape; the CreateProduct returns a custom JSON with status, message, and data containing the created product.

**7) Optional: Docker-compose (quick local run)**

If helpful, I can also provide a docker-compose.yml to run the ProductService, ApiGatewayService and SQL Server container with proper port mappings and environment variables. Tell me if you want that and I’ll include it.

If you want, I can now:

* Paste full .csproj files and launchSettings for both projects,
* Provide the docker-compose.yml and Dockerfiles,
* Or produce a ready-to-run ZIP/archived layout.

Tell me which of those you'd like and I’ll produce it right away.