Ultimate Guide to Building a Hexagonal Architecture Project in .NET

This guide walks you through creating a hexagonal architecture project in .NET, organizing the application into layers with clear boundaries and dependencies. We'll use an e-commerce example focused on managing products.

Step 1: Create the Solution and Projects

Hexagonal architecture separates concerns into distinct layers. We'll create four projects:

- **Domain**: Core business logic and entities.
- **Application**: Application services and ports (interfaces).
- **Infrastructure**: Adapters (e.g., database implementations).
- **Api**: Entry point for external interactions.

Commands

```
dotnet new sln -n ECommerce
dotnet new classlib -n ECommerce.Domain
dotnet sln add ECommerce.Domain/ECommerce.Domain.csproj
dotnet new classlib -n ECommerce.Application
dotnet sln add ECommerce.Application/ECommerce.Application.csproj
dotnet new classlib -n ECommerce.Infrastructure
dotnet sln add ECommerce.Infrastructure/ECommerce.Infrastructure.csproj
dotnet new webapi -n ECommerce.Api
dotnet sln add ECommerce.Api/ECommerce.Api.csproj
```

Step 2: Set Up Project References

Define dependencies to enforce the hexagonal structure:

- `ECommerce.Application` depends on `ECommerce.Domain`.
- `ECommerce.Infrastructure` depends on `ECommerce.Application` and `ECommerce.Domain`.

- `ECommerce.Api` depends on `ECommerce.Application` and `ECommerce.Infrastructure`.

Commands

```
dotnet add ECommerce.Application/ECommerce.Application.csproj reference
ECommerce.Domain/ECommerce.Domain.csproj
dotnet add ECommerce.Infrastructure/ECommerce.Infrastructure.csproj reference
ECommerce.Application/ECommerce.Application.csproj
dotnet add ECommerce.Infrastructure/ECommerce.Infrastructure.csproj reference
ECommerce.Domain/ECommerce.Domain.csproj
dotnet add ECommerce.Api/ECommerce.Api.csproj reference
ECommerce.Application/ECommerce.Application.csproj
dotnet add ECommerce.Api/ECommerce.Api.csproj reference
ECommerce.Infrastructure/ECommerce.Infrastructure.csproj
```

Step 3: Define the Domain Model

The domain layer contains entities and business rules. Create a `Product` entity in `ECommerce.Domain`.

Product Entity

```
// ECommerce.Domain/Entities/Product.cs
namespace ECommerce.Domain.Entities
{
    public class Product
    {
        public Guid Id { get; private set; }
        public string Name { get; private set; }
        public decimal Price { get; private set; }
        public int Stock { get; private set; }

        private Product() { }

        public Product(Guid id, string name, decimal price, int stock)
        {
            if (string.IsNullOrWhiteSpace(name)) throw new ArgumentException("Name cannot be
empty.");
            if (price < 0) throw new ArgumentException("Price cannot be negative.");
            if (stock < 0) throw new ArgumentException("Stock cannot be negative.");
            Id = id;
            Name = name;
            Price = price;
            Stock = stock;
        }

        public void UpdateName(string newName)
        {
            if (string.IsNullOrWhiteSpace(newName)) throw new ArgumentException("Name cannot
be empty.");
            Name = newName;
        }

        public void UpdatePrice(decimal newPrice)
        {
            if (newPrice < 0) throw new ArgumentException("Price cannot be negative.");
            Price = newPrice;
        }

        public void UpdateStock(int newStock)
        {
            if (newStock < 0) throw new ArgumentException("Stock cannot be negative.");
            Stock = newStock;
        }
    }
}
```

Step 4: Define Ports and Application Services

The application layer defines ports (interfaces) and services that orchestrate business logic.

4.1 Define Out-Ports (Repository Interface)

```
// ECommerce.Application/Ports/IProductRepository.cs
using ECommerce.Domain.Entities;

namespace ECommerce.Application.Ports
{
    public interface IProductRepository
    {
        Task<Product> GetByIdAsync(Guid id);
        Task<IEnumerable<Product>> GetAllAsync();
        Task AddAsync(Product product);
        Task UpdateAsync(Product product);
        Task DeleteAsync(Guid id);
    }
}
```

4.2 Define In-Ports (Service Interface)

```
// ECommerce.Application/Ports/IProductService.cs
using ECommerce.Application.Dtos;

namespace ECommerce.Application.Ports
{
    public interface IProductService
    {
        Task<ProductDto> GetProductAsync(Guid id);
        Task<IEnumerable<ProductDto>> GetAllProductsAsync();
        Task<ProductDto> CreateProductAsync(string name, decimal price, int stock);
        Task UpdateProductAsync(Guid id, string name, decimal price, int stock);
        Task DeleteProductAsync(Guid id);
    }
}
```

4.3 Define DTOs

```
// ECommerce.Application/Dtos/ProductDto.cs
namespace ECommerce.Application.Dtos
{
    public record ProductDto(Guid Id, string Name, decimal Price, int Stock);
}
```

4.4 Implement Application Service

```
// ECommerce.Application/Services/ProductService.cs
using ECommerce.Application.Dtos;
using ECommerce.Application.Ports;
using ECommerce.Domain.Entities;

namespace ECommerce.Application.Services
{
    public class ProductService : IProductService
    {
        private readonly IProductRepository _productRepository;
        public ProductService(IProductRepository productRepository)
        {
            _productRepository = productRepository;
        }

        public async Task<ProductDto> GetProductAsync(Guid id)
        {
            var product = await _productRepository.GetByIdAsync(id);
            if (product == null) throw new Exception("Product not found.");
            return new ProductDto(product.Id, product.Name, product.Price, product.Stock);
        }

        public async Task<IEnumerable<ProductDto>> GetAllProductsAsync()
        {
            var products = await _productRepository.GetAllAsync();
            return products.Select(p => new ProductDto(p.Id, p.Name, p.Price, p.Stock));
        }

        public async Task<ProductDto> CreateProductAsync(string name, decimal price, int stock)
        {
            var product = new Product(Guid.NewGuid(), name, price, stock);
            await _productRepository.AddAsync(product);
            return new ProductDto(product.Id, product.Name, product.Price, product.Stock);
        }

        public async Task UpdateProductAsync(Guid id, string name, decimal price, int stock)
        {
            var product = await _productRepository.GetByIdAsync(id);
            if (product == null) throw new Exception("Product not found.");
            product.UpdateName(name);
            product.UpdatePrice(price);
            product.UpdateStock(stock);
            await _productRepository.UpdateAsync(product);
        }

        public async Task DeleteProductAsync(Guid id)
        {
            await _productRepository.DeleteAsync(id);
        }
    }
}
```

Step 5: Implement Infrastructure Layer

The infrastructure layer provides concrete implementations (adapters) like database access using Entity Framework Core.

5.1 Install NuGet Packages

For `ECommerce.Infrastructure`:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

For `ECommerce.Api`:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.SqlServer
dotnet add package Microsoft.EntityFrameworkCore.Design
```

5.2 Define DbContext

```
// ECommerce.Infrastructure/Data/ECommerceDbContext.cs
using ECommerce.Domain.Entities;
using Microsoft.EntityFrameworkCore;

namespace ECommerce.Infrastructure.Data
{
    public class ECommerceDbContext : DbContext
    {
        public DbSet<Product> Products { get; set; }

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

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Product>().HasKey(p => p.Id);
        }
    }
}
```

5.3 Implement Repository

```
// ECommerce.Infrastructure/Repositories/ProductRepository.cs
using ECommerce.Application.Ports;
using ECommerce.Domain.Entities;
using ECommerce.Infrastructure.Data;
using Microsoft.EntityFrameworkCore;

namespace ECommerce.Infrastructure.Repositories
{
    public class ProductRepository : IProductRepository
    {
        private readonly ECommerceDbContext _context;

        public ProductRepository(ECommerceDbContext context)
        {
            _context = context;
        }

        public async Task<Product> GetByIdAsync(Guid id)
        {
            return await _context.Products.FindAsync(id);
        }

        public async Task<IEnumerable<Product>> GetAllAsync()
        {
            return await _context.Products.ToListAsync();
        }

        public async Task AddAsync(Product product)
        {
            await _context.Products.AddAsync(product);
            await _context.SaveChangesAsync();
        }

        public async Task UpdateAsync(Product product)
        {
            _context.Products.Update(product);
            await _context.SaveChangesAsync();
        }

        public async Task DeleteAsync(Guid id)
        {
            var product = await GetByIdAsync(id);
            if (product != null)
            {
                _context.Products.Remove(product);
                await _context.SaveChangesAsync();
            }
        }
    }
}
```

Step 6: Set Up the API Layer

The API layer serves as the entry point for external interactions, configured with dependency injection and Swagger for API documentation and testing.

6.1 Install Swashbuckle.AspNetCore

Before configuring Swagger, you need to install the `Swashbuckle.AspNetCore` NuGet package in the `ECommerce.Api` project. This package provides the tools to generate Swagger documentation and set up the Swagger UI.

Run the following command from the solution directory (or navigate to the `ECommerce.Api` directory):

```
dotnet add ECommerce.Api/ECommerce.Api.csproj package Swashbuckle.AspNetCore
--version 6.2.3
```

- **Why**: This package is essential for enabling Swagger in your ASP.NET Core API. It includes the necessary components to generate API documentation and provide an interactive UI for testing endpoints.
- **Version**: Specifying `--version 6.2.3` ensures compatibility with .NET 6. Adjust the version based on your .NET version if needed.

After adding the package, restore the solution to ensure all dependencies are up to date:

```
dotnet restore
```

- **Note**: While `dotnet add package` typically triggers a restore, running `dotnet restore` explicitly ensures all packages are properly installed.

6.2 Configure Program.cs

With the package installed, configure Swagger in the `Program.cs` file of the `ECommerce.Api` project. This involves registering Swagger services and enabling the Swagger middleware.

Update `Program.cs` as follows:

```
// ECommerce.Api/Program.cs
using ECommerce.Application.Ports;
using ECommerce.Application.Services;
using ECommerce.Infrastructure.Data;
using ECommerce.Infrastructure.Repositories;
using Microsoft.EntityFrameworkCore;
using Microsoft.OpenApi.Models;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
builder.Services.AddDbContext<ECommerceDbContext>(options =>
   options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));
builder.Services.AddScoped<IProductRepository, ProductRepository>();
builder.Services.AddScoped<IProductService, ProductService>();

// Add Swagger services
builder.Services.AddSwaggerGen(c =>
{
   c.SwaggerDoc("v1", new OpenApiInfo { Title = "ECommerce API", Version = "v1" });
});

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
   app.UseSwagger(); // Enable Swagger JSON endpoint
   app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "ECommerce API
v1")); // Enable Swagger UI
}

app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

- **Key Configurations**:

    - `builder.Services.AddSwaggerGen()`: Registers the Swagger generator with a document named "v1" and basic API information.
    - `app.UseSwagger()`: Enables the Swagger middleware to generate the Swagger JSON document.
    - `app.UseSwaggerUI()`: Sets up the Swagger UI for interactive API testing, available at `/swagger`.

6.3 Update appsettings.json

Ensure your `appsettings.json` file in `ECommerce.Api` contains the correct connection string for your database (e.g., SQL Server in Docker):

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost,1433;Database=ECommerceDb;User
Id=sa;Password=StrongP@ssw0rd!;TrustServerCertificate=True"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

- **Note**: Adjust the connection string if your database setup differs.

6.4 Implement Controller

The `ProductsController` exposes API endpoints using the application services. Below is an example implementation:

```
// ECommerce.Api/Controllers/ProductsController.cs
using ECommerce.Application.Dtos;
using ECommerce.Application.Ports;
using Microsoft.AspNetCore.Mvc;

namespace ECommerce.Api.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase
    {
        private readonly IProductService _productService;

        public ProductsController(IProductService productService)
        {
            _productService = productService;
        }

        [HttpGet]
        public async Task<IActionResult> GetAllProducts()
        {
            var products = await _productService.GetAllProductsAsync();
            return Ok(products);
        }

        [HttpPost]
        public async Task<IActionResult> CreateProduct([FromBody] CreateProductRequest
request)
        {
            var product = await _productService.CreateProductAsync(request.Name, request.Price,
request.Stock);
            return CreatedAtAction(nameof(GetProduct), new { id = product.Id }, product);
        }

        // Additional endpoints (GET by ID, PUT, DELETE) can be added here
    }

    public record CreateProductRequest(string Name, decimal Price, int Stock);
}
```

- **Endpoints**:
  - `GET /api/products`: Retrieves all products.
  - `POST /api/products`: Creates a new product.
- **Swagger Integration**: With Swagger enabled, these endpoints will automatically appear in the Swagger UI for testing.

Step 7: Set Up Docker for SQL Server

Use Docker to run SQL Server for a consistent database setup.

Command

docker run -e "ACCEPT_EULA=Y" -e "SA_PASSWORD=StrongP@ssw0rd!" -p 1433:1433
--name sqlserverhex -d mcr.microsoft.com/mssql/server:2022-latest

---

Step 8: Run Migrations and Test

Create the database and test the API using Swagger.

Commands

dotnet ef migrations add InitialCreate --project ECommerce.Infrastructure --startup-project ECommerce.Api
dotnet ef database update --project ECommerce.Infrastructure --startup-project ECommerce.Api

Run the API:

dotnet run --project ECommerce.Api

Open Swagger at `https://localhost:<port>/swagger` (port varies, e.g., 5001 or 7180) to test endpoints like `GET /api/products`, `POST /api/products`, etc.

Summary

You now have a fully functional hexagonal architecture project in .NET:

- **Domain**: Defines `Product` with business rules.
- **Application**: Orchestrates logic via `IProductService` and `IProductRepository`.
- **Infrastructure**: Implements persistence with EF Core.
- **Api**: Exposes endpoints with Swagger integration.

This setup ensures loose coupling, testability, and adaptability to future changes.