# **Atributos .NET para Mapeo y Validación**

Esta es una lista exhaustiva y organizada de los atributos más relevantes en el ecosistema de .NET para el mapeo de datos, la validación del modelo y la serialización JSON.

## 1. Atributos de Mapeo de Datos (Entity Framework Core)

Estos atributos (System.ComponentModel.DataAnnotations.Schema y Microsoft.EntityFrameworkCore) son utilizados por EF Core para definir cómo tus clases y propiedades se relacionan con el esquema de la base de datos.

* **[Table("NombreDeTabla")]**: Especifica el nombre de la tabla en la base de datos si es diferente al de la clase.
  + Ejemplo: [Table("Productos")]
* **[Column("NombreColumna", TypeName = "sql\_type")]**: Especifica el nombre y el tipo de la columna en la base de datos.
  + Ejemplo: [Column("Precio", TypeName = "decimal(18, 2)")]
* **[Key]**: Define la clave primaria de la entidad. No es necesario si la propiedad se llama Id o <NombreDeClase>Id.
  + Ejemplo: [Key] public int ProductoId { get; set; }
* **[ForeignKey("NombrePropiedad")]**: Define la clave foránea para la relación. Se usa en la propiedad de navegación o en la propiedad de la clave foránea misma.
  + Ejemplo: [ForeignKey("CategoriaId")] public Categoria Categoria { get; set; }
* **[InverseProperty("NombrePropiedadDeNavegacion")]**: Se usa para especificar la propiedad de navegación opuesta en una relación, ayudando a EF Core a comprender la conexión.
  + Ejemplo: [InverseProperty("Producto")] public ICollection<DetalleOrden> Detalles { get; set; }
* **[NotMapped]**: Excluye una propiedad de ser mapeada a una columna en la base de datos, lo que es útil para propiedades de solo lógica.
  + Ejemplo: [NotMapped] public string NombreCompleto { get; set; }
* **[DatabaseGenerated(DatabaseGeneratedOption.Identity)]**: Especifica cómo la base de datos genera el valor de la columna (ej. Identity para auto-incremento, Computed para un valor calculado).
  + Ejemplo: [DatabaseGenerated(DatabaseGeneratedOption.Identity)] public int Id { get; set; }
* **[StringLength(maxLength)]**: Limita la longitud máxima de una propiedad de tipo string, lo que se traduce en un VARCHAR(maxLength) en SQL.
  + Ejemplo: [StringLength(255)] public string Descripcion { get; set; }
* **[MaxLength(maxLength)]**: Similar a StringLength, pero solo establece la longitud máxima.
  + Ejemplo: [MaxLength(200)] public string NombreProducto { get; set; }
* **[Precision(precision, scale)]**: Define la precisión y escala para un tipo numérico o decimal.
  + Ejemplo: [Precision(18, 2)] public decimal Precio { get; set; }
* **[Required]**: Indica que una columna no puede ser nula. EF la mapea a un campo NOT NULL.
  + Ejemplo: [Required] public string Nombre { get; set; }
* **[Index(unique: true)]**: Crea un índice sobre la columna o columnas especificadas para mejorar el rendimiento de las consultas.
  + Ejemplo: [Index(unique: true)] public string CodigoSKU { get; set; }
* **[Comment("Texto del comentario")]**: Agrega un comentario a la tabla o columna en la base de datos para documentación.
  + Ejemplo: [Comment("Nombre del producto como aparece en el catálogo")] public string NombreProducto { get; set; }
* **[Timestamp]**: Se usa para una propiedad byte[]. EF Core la mapea a una columna rowversion para el control de concurrencia.
  + Ejemplo: [Timestamp] public byte[] VersionFila { get; set; }
* **[Owned]**: Se utiliza en una clase que no necesita su propia tabla. Sus propiedades se mapearán como columnas en la tabla de la entidad que la contiene.

## 2. Atributos de Validación (Data Annotations)

Estos atributos (System.ComponentModel.DataAnnotations) se usan para validar los datos del modelo en el frontend y el backend.

* **[Required]**: Indica que un campo es obligatorio.
  + Ejemplo: [Required] public string NombreUsuario { get; set; }
* **[StringLength(maxLength, MinimumLength = minLength)]**: Define un rango de longitud para una cadena.
  + Ejemplo: [StringLength(50, MinimumLength = 5)] public string Titulo { get; set; }
* **[MinLength(minLength)]**: Define la longitud mínima de una cadena.
  + Ejemplo: [MinLength(5)] public string Comentario { get; set; }
* **[MaxLength(maxLength)]**: Define la longitud máxima de una cadena.
  + Ejemplo: [MaxLength(50)] public string Etiqueta { get; set; }
* **[Range(min, max)]**: Valida que el valor numérico esté dentro de un rango.
  + Ejemplo: [Range(18, 99)] public int Edad { get; set; }
* **[RegularExpression("regex")]**: Valida la propiedad contra una expresión regular.
  + Ejemplo: [RegularExpression(@"^[0-9]{5}$")] public string CodigoPostal { get; set; }
* **[EmailAddress]**: Valida que el formato del string sea el de un correo electrónico.
  + Ejemplo: [EmailAddress] public string CorreoElectronico { get; set; }
* **[Phone]**: Valida que el formato del string sea el de un número de teléfono.
  + Ejemplo: [Phone] public string Telefono { get; set; }
* **[Url]**: Valida que el formato del string sea el de una URL.
  + Ejemplo: [Url] public string SitioWeb { get; set; }
* **[Compare("OtraPropiedad")]**: Compara dos propiedades, como al confirmar una contraseña.
  + Ejemplo: [Compare("Password")] public string ConfirmarPassword { get; set; }
* **[CreditCard]**: Valida el formato de un número de tarjeta de crédito.
  + Ejemplo: [CreditCard] public string NumeroTarjeta { get; set; }
* **[FileExtensions(Extensions = "ext1,ext2")]**: Valida que la extensión de un archivo sea una de las permitidas.
  + Ejemplo: [FileExtensions(Extensions = "jpg,png,gif")] public string FotoPerfil { get; set; }
* **[DataType(DataType.Password)]**: Ayuda a los frameworks (como ASP.NET Core) a renderizar la UI adecuada (ej. un campo de contraseña).
  + Ejemplo: [DataType(DataType.Password)] public string Password { get; set; }
* **[Display(Name = "Nombre")]**: Proporciona un nombre amigable para la propiedad, usado en la UI.
  + Ejemplo: [Display(Name = "Nombre de Usuario")] public string UserName { get; set; }

## 3. Atributos para la Serialización de JSON

Estos atributos (System.Text.Json.Serialization) controlan cómo las propiedades de tus objetos se serializan y deserializan a JSON.

* **[JsonPropertyName("nombrePropiedadJson")]**: Especifica el nombre de la propiedad en el JSON si es diferente al de la clase de C#.
  + Ejemplo: [JsonPropertyName("producto\_id")] public int Id { get; set; }
* **[JsonIgnore]**: Ignora una propiedad durante la serialización.
  + Ejemplo: [JsonIgnore] public string ClaveApi { get; set; }
* **[JsonInclude]**: Incluye una propiedad durante la serialización que de otro modo sería ignorada (por ejemplo, si es un campo privado o una propiedad de solo lectura).
  + Ejemplo: [JsonInclude] private readonly int \_nivelSeguridad;
* **[JsonExtensionData]**: Permite manejar propiedades JSON adicionales que no están definidas en la clase.
  + Ejemplo: [JsonExtensionData] public Dictionary<string, object> PropiedadesAdicionales { get; set; }
* **[JsonNumberHandling(options)]**: Controla cómo se manejan los números durante la serialización.
  + Ejemplo: [JsonNumberHandling(JsonNumberHandling.AllowReadingFromString)] public int Valor { get; set; }
* **[JsonConverter(typeof(MiClaseConverter))]**: Permite especificar un convertidor personalizado para una propiedad.
  + Ejemplo: [JsonConverter(typeof(JsonStringEnumConverter))] public EstadoPedido Estado { get; set; }
* **[JsonConstructor]**: Indica el constructor que se debe usar al deserializar un objeto desde JSON.
  + Ejemplo: [JsonConstructor] public Persona(string nombre, int edad) { ... }
* **[JsonIgnoreCondition(JsonIgnoreCondition.WhenWritingDefault)]**: Ignora la propiedad si su valor es el valor predeterminado del tipo al escribir el JSON (ej. 0 para enteros o null para objetos).
  + Ejemplo: [JsonIgnoreCondition(JsonIgnoreCondition.WhenWritingDefault)] public int Cantidad { get; set; }

**ORGANIZACION DE CARPETAS**

### La Carpeta Data

Se llama Data porque su propósito principal es manejar **el acceso a los datos**. Es el lugar donde resides con todo lo relacionado con la interacción directa con la base de datos. El DbContext vive aquí porque su única responsabilidad es ser ese puente entre tus clases y las tablas de la base de datos.

Piensa en ello como el "cuarto de máquinas" de tu aplicación, donde se realizan todas las operaciones de lectura y escritura.

### El Patrón Repository, Interface y Services

Estos términos se refieren a un patrón de diseño que te ayuda a separar las responsabilidades de tu código.

* **Repository**: Un **repositorio** es una clase que actúa como un "guardián" de una entidad específica. Su trabajo es centralizar todas las operaciones de la base de datos para una entidad. Por ejemplo, tendrías un ProductoRepository que contiene todos los métodos para crear, leer, actualizar y eliminar productos. Esto evita que tengas código de base de datos disperso por toda tu aplicación.
* **Interface**: Una **interfaz** es un contrato. Define qué métodos debe tener una clase sin decir cómo se implementan. Por ejemplo, una interfaz llamada IProductoRepository diría "esta clase debe tener un método GetById(), un método Add(), etc.". El uso de interfaces es crucial porque permite que tu capa de servicios dependa del **contrato**, no de la implementación. Esto hace que tu código sea flexible y fácil de probar, ya que puedes "conectar" diferentes implementaciones del repositorio si es necesario (por ejemplo, una para una base de datos real y otra para pruebas).
* **Services**: La carpeta de **servicios** (o capa de lógica de negocio) es donde está el verdadero "cerebro" de tu aplicación. Aquí es donde resides la lógica que no tiene nada que ver con la base de datos. Por ejemplo, si un producto tiene que tener un precio mínimo, esa validación viviría en un ProductoService, el cual usaría al ProductoRepository para obtener el producto de la base de datos.

En resumen, la idea es que la capa de servicios **no sepa** que estás usando EF Core o una base de datos SQL. Solo sabe que necesita un repositorio para obtener y guardar datos. Esto crea una arquitectura desacoplada y muy bien organizada.

**CODIGO DE CARPETAS**

**DATA/**

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Data.Common;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Data

{

public class T\_ElectronicDbContext : DbContext

{

public T\_ElectronicDbContext(DbContextOptions options) : base(options)

{

}

public DbSet<Producto> Productos { get; set; }

public DbSet<Categoria> Categorias { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Producto>()

.HasOne(p => p.Categoria)

.WithMany(c => c.Productos)

.HasForeignKey(p => p.Categoria\_Id);

//Agregar datos por ahora no es necesario

/\*

modelBuilder.Entity<Marca>().HasData(

new Marca

{

//Se debe poner Id para los Seeders

Id = 1,

Nombre = "Samsung",

}

);

\*/

}

}

}

**INTERFACES/**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Interfaces

{

public interface IProductoRepository

{

Task<IEnumerable<Producto>> GetAllAsync();

Task<Producto> GetByIdAsync(int id);

Task AddAsync(Producto producto);

Task UpdateAsync(Producto producto);

Task DeleteAsync(int id);

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Interfaces

{

public interface IProductoService

{

Task<IEnumerable<Producto>> GetAllProductosAsync();

Task<Producto> GetProductoByIdAsync(int id);

Task AddProductoAsync(Producto producto);

Task UpdateProductoAsync(Producto producto);

Task DeleteProductoAsync(int id);

}

}

**REPOSITORY/**

using Microsoft.EntityFrameworkCore;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using T\_Electronic.Logic.Data;

using T\_Electronic.Logic.Interfaces;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Repository

{

public class ProductoRepository : IProductoRepository

{

private readonly T\_ElectronicDbContext \_context;

public ProductoRepository(T\_ElectronicDbContext context)

{

\_context = context;

}

public async Task<IEnumerable<Producto>> GetAllAsync()

{

// Incluye la categoría para evitar errores de referencia nula

return await \_context.Productos.Include(p => p.Categoria).ToListAsync();

}

public async Task<Producto> GetByIdAsync(int id)

{

// Busca un producto por ID y también incluye la categoría

return await \_context.Productos.Include(p => p.Categoria).FirstOrDefaultAsync(p => p.Id == id);

}

public async Task AddAsync(Producto producto)

{

await \_context.Productos.AddAsync(producto);

await \_context.SaveChangesAsync();

}

public async Task UpdateAsync(Producto producto)

{

\_context.Productos.Update(producto);

await \_context.SaveChangesAsync();

}

public async Task DeleteAsync(int id)

{

var producto = await \_context.Productos.FindAsync(id);

if (producto != null)

{

\_context.Productos.Remove(producto);

await \_context.SaveChangesAsync();

}

}

}

}

**SERVICE/**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using T\_Electronic.Logic.Interfaces;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Services

{

public class ProductoService : IProductoService

{

private readonly IProductoRepository \_productoRepository;

// Inyecta el repositorio a través del constructor

public ProductoService(IProductoRepository productoRepository)

{

\_productoRepository = productoRepository;

}

// Implementa los métodos usando el repositorio

public async Task<IEnumerable<Producto>> GetAllProductosAsync()

{

return await \_productoRepository.GetAllAsync();

}

public async Task<Producto> GetProductoByIdAsync(int id)

{

return await \_productoRepository.GetByIdAsync(id);

}

public async Task AddProductoAsync(Producto producto)

{

await \_productoRepository.AddAsync(producto);

}

public async Task UpdateProductoAsync(Producto producto)

{

await \_productoRepository.UpdateAsync(producto);

}

public async Task DeleteProductoAsync(int id)

{

await \_productoRepository.DeleteAsync(id);

}

}

}

**CONTROLLERS/**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using T\_Electronic.Logic.Interfaces;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Web.Controllers

{

public class ProductoController : Controller

{

// Campos privados para la inyección de dependencias de los servicios.

private readonly IProductoService \_productoService;

private readonly ICategoriaService \_categoriaService;

public ProductoController(IProductoService productoService, ICategoriaService categoriaService)

{

\_productoService = productoService;

\_categoriaService = categoriaService;

}

// GET: Producto/Index

public async Task<IActionResult> Index()

{

var productos = await \_productoService.GetAllProductosAsync();

return View(productos);

}

// GET: Producto/Details/5

public async Task<IActionResult> Details(int id)

{

// Busca un producto por ID de forma asíncrona.

var producto = await \_productoService.GetProductoByIdAsync(id);

if (producto == null)

{

return NotFound();

}

return View(producto);

}

// GET: Producto/Create

public async Task<IActionResult> Create()

{

ViewBag.Categorias = await \_categoriaService.GetAllCategoriasAsync();

return View();

}

// POST: Producto/Create

public async Task<IActionResult> Create(Producto producto)

{

if (ModelState.IsValid)

{

await \_productoService.AddProductoAsync(producto);

return RedirectToAction(nameof(Index));

}

ViewBag.Categorias = await \_categoriaService.GetAllCategoriasAsync();

return View(producto);

}

// GET: Producto/Edit/5

public async Task<IActionResult> Edit(int id)

{

// Busca el producto a editar.

var producto = await \_productoService.GetProductoByIdAsync(id);

if (producto == null)

{

return NotFound();

}

ViewBag.Categorias = await \_categoriaService.GetAllCategoriasAsync();

return View(producto);

}

// POST: Producto/Edit/5

public async Task<IActionResult> Edit(int id, Producto producto)

{

if (id != producto.Id)

{

return NotFound();

}

if (ModelState.IsValid)

{

await \_productoService.UpdateProductoAsync(producto);

return RedirectToAction(nameof(Index));

}

ViewBag.Categorias = await \_categoriaService.GetAllCategoriasAsync();

return View(producto);

}

// GET: Producto/Delete/5

public async Task<IActionResult> Delete(int id)

{

var producto = await \_productoService.GetProductoByIdAsync(id);

if (producto == null)

{

return NotFound();

}

return View(producto);

}

// POST: Producto/Delete/5

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public async Task<IActionResult> DeleteConfirmed(int id)

{

await \_productoService.DeleteProductoAsync(id);

return RedirectToAction(nameof(Index));

}

}

}

**EXPLICACION DEL CODIGO**

### Análisis del Código

#### 1. T\_ElectronicDbContext.cs

Este archivo es tu **capa de acceso a datos**. Lo que hace es:

* **Conecta tu aplicación con la base de datos:** El DbContext es el puente.
* **Define tus tablas:** DbSet<Producto> y DbSet<Categoria> son las colecciones que representan las tablas en tu base de datos.
* **Configura la relación uno a muchos:** El método OnModelCreating es crucial. Aquí estás usando la **API Fluente** para decirle a EF Core que un Producto tiene una Categoria y que una Categoria puede tener muchos Productos, con la clave foránea Categoria\_Id. Esto es mucho mejor que usar atributos en la clase, ya que centraliza la configuración de tu modelo.

#### 2. IProductoRepository.cs

Este es el **contrato del repositorio**. Su propósito es:

* **Definir la funcionalidad:** Aquí solo declaras lo que el repositorio puede hacer (GetAllAsync, GetByIdAsync, etc.) sin dar detalles de cómo lo hace.
* **Desacoplar las capas:** Esta interfaz permite que tu capa de servicio dependa de un contrato, no de una implementación específica. Si más tarde quisieras usar otra tecnología de base de datos (como Dapper), solo tendrías que crear una nueva clase de repositorio que implemente esta misma interfaz.

#### 3. IProductoService.cs

Este es el **contrato del servicio de negocio**. Aunque en este caso la interfaz del servicio es casi idéntica a la del repositorio, su propósito es diferente:

* **Definir la lógica de negocio:** En aplicaciones más grandes, aquí agregarías métodos que incluyan lógica de negocio compleja que combine múltiples llamadas al repositorio. Por ejemplo, un método que agregue un producto y también registre un evento en un log.

#### 4. ProductoRepository.cs

Esta es la **implementación del repositorio**. Su trabajo es:

* **Manejar la base de datos:** Esta clase es la única que sabe que estás usando EF Core.
* **Implementar la interfaz:** Contiene el código real para interactuar con el DbContext y obtener los datos.
* **Manejo de relaciones:** Usas .Include(p => p.Categoria) en los métodos GetAllAsync y GetByIdAsync. Esto es muy importante, ya que asegura que cuando traigas un producto, también se cargue la información de su categoría, evitando el error de referencia nula.

#### 5. ProductoService.cs

Esta es la **capa de servicio**, donde reside tu lógica de negocio. Lo que hace es:

* **Delegar el trabajo:** Llama a los métodos del repositorio para obtener, agregar, actualizar o eliminar datos.
* **Aplicar reglas de negocio:** Aunque en tu código los métodos son simples, aquí es donde agregarías validaciones y lógica. Por ejemplo, antes de llamar a AddAsync, podrías verificar que el nombre del producto no esté vacío.

#### 6. ProductoController.cs

Este es el **controlador web**. Su responsabilidad es:

* **Manejar las peticiones HTTP:** Cada método (Index, Details, Create, etc.) responde a una petición específica del usuario.
* **Usar la inyección de dependencias:** En el constructor, se inyectan IProductoService y ICategoriaService. Esto permite que el controlador use los servicios sin tener que crearlos.
* **Delegar el trabajo a los servicios:** El controlador no tiene ninguna lógica de base de datos. Simplemente llama a los métodos del servicio (\_productoService.GetAllProductosAsync()) y delega la tarea.
* **Devolver el resultado:** Retorna una View con los datos o redirige a otra acción.
* **Uso de ModelState.IsValid:** Esto es clave para las validaciones en ASP.NET Core. Verifica si los datos que llegaron en la petición son válidos según los atributos que has definido en tu modelo (como [Required]).

**USANDO DAPPER**

using System.Data;

using Dapper;

using Microsoft.EntityFrameworkCore;

using T\_Electronic.Logic.Data;

using T\_Electronic.Logic.Interfaces;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Repository

{

public class ProductoRepository : IProductoRepository

{

private readonly T\_ElectronicDbContext \_context;

private readonly IDbConnection \_dbConnection;

public ProductoRepository(T\_ElectronicDbContext context)

{

\_context = context;

// Obtiene la conexión de la base de datos desde el DbContext.

\_dbConnection = \_context.Database.GetDbConnection();

}

// Usa Dapper para obtener todos los productos.

public async Task<IEnumerable<Producto>> GetAllAsync()

{

// La consulta SQL debe incluir la tabla Categoria para cargar los datos relacionados.

var sql = "SELECT p.\*, c.\* FROM Productos AS p INNER JOIN Categorias AS c ON p.Categoria\_Id = c.Id";

// Dapper puede mapear el resultado de la consulta a varios objetos.

// La función lambda (p, c) => ... indica cómo se deben mapear los objetos.

var productos = await \_dbConnection.QueryAsync<Producto, Categoria, Producto>(

sql,

(producto, categoria) =>

{

// Asigna la categoria al producto

producto.Categoria = categoria;

return producto;

},

splitOn: "Id"); // "splitOn" indica a Dapper dónde dividir el resultado de la consulta.

return productos;

}

// Usa Dapper para obtener un producto por su ID.

public async Task<Producto> GetByIdAsync(int id)

{

var sql = "SELECT p.\*, c.\* FROM Productos AS p INNER JOIN Categorias AS c ON p.Categoria\_Id = c.Id WHERE p.Id = @Id";

var producto = await \_dbConnection.QueryAsync<Producto, Categoria, Producto>(

sql,

(p, c) =>

{

p.Categoria = c;

return p;

},

new { Id = id }, // Pasa el parámetro de forma segura para evitar SQL Injection.

splitOn: "Id");

return producto.FirstOrDefault();

}

// Los métodos de escritura seguirán usando EF Core.

public async Task AddAsync(Producto producto)

{

await \_context.Productos.AddAsync(producto);

await \_context.SaveChangesAsync();

}

public async Task UpdateAsync(Producto producto)

{

\_context.Productos.Update(producto);

await \_context.SaveChangesAsync();

}

public async Task DeleteAsync(int id)

{

var producto = await \_context.Productos.FindAsync(id);

if (producto != null)

{

\_context.Productos.Remove(producto);

await \_context.SaveChangesAsync();

}

}

}

}

**AGREGANDO UNA NUEVA FUNCION PARA EXTRAER POR NOMBRE**

**INTERFACES/**

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Interfaces

{

public interface IProductoRepository

{

Task<IEnumerable<Producto>> GetAllAsync();

Task<Producto> GetByIdAsync(int id);

Task<Producto> GetByNameAsync(string nombre); // ¡Nuevo método para el contrato!

Task AddAsync(Producto producto);

Task UpdateAsync(Producto producto);

Task DeleteAsync(int id);

}

}

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Interfaces

{

public interface IProductoService

{

Task<IEnumerable<Producto>> GetAllProductosAsync();

Task<Producto> GetProductoByIdAsync(int id);

Task<Producto> GetProductoByNameAsync(string nombre); // ¡Nuevo método expuesto!

Task AddProductoAsync(Producto producto);

Task UpdateProductoAsync(Producto producto);

Task DeleteProductoAsync(int id);

}

}

**REPOSITORY/**

using Dapper;

using Microsoft.EntityFrameworkCore;

using System.Data;

using T\_Electronic.Logic.Data;

using T\_Electronic.Logic.Interfaces;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Logic.Repository

{

public class ProductoRepository : IProductoRepository

{

private readonly T\_ElectronicDbContext \_context;

private readonly IDbConnection \_dbConnection;

public ProductoRepository(T\_ElectronicDbContext context)

{

\_context = context;

\_dbConnection = \_context.Database.GetDbConnection();

}

public async Task<IEnumerable<Producto>> GetAllAsync()

{

var sql = "SELECT p.\*, c.\* FROM Productos AS p INNER JOIN Categorias AS c ON p.Categoria\_Id = c.Id";

var productos = await \_dbConnection.QueryAsync<Producto, Categoria, Producto>(

sql,

(producto, categoria) =>

{

producto.Categoria = categoria;

return producto;

},

splitOn: "Id");

return productos;

}

public async Task<Producto> GetByIdAsync(int id)

{

var sql = "SELECT p.\*, c.\* FROM Productos AS p INNER JOIN Categorias AS c ON p.Categoria\_Id = c.Id WHERE p.Id = @Id";

var producto = await \_dbConnection.QueryAsync<Producto, Categoria, Producto>(

sql,

(p, c) =>

{

p.Categoria = c;

return p;

},

new { Id = id },

splitOn: "Id");

return producto.FirstOrDefault();

}

public async Task<Producto> GetByNameAsync(string nombre) // ¡Implementación del nuevo método!

{

var sql = "SELECT p.\*, c.\* FROM Productos AS p INNER JOIN Categorias AS c ON p.Categoria\_Id = c.Id WHERE p.Nombre = @Nombre";

var producto = await \_dbConnection.QueryAsync<Producto, Categoria, Producto>(

sql,

(p, c) =>

{

p.Categoria = c;

return p;

},

new { Nombre = nombre },

splitOn: "Id");

return producto.FirstOrDefault();

}

public async Task AddAsync(Producto producto)

{

await \_context.Productos.AddAsync(producto);

await \_context.SaveChangesAsync();

}

public async Task UpdateAsync(Producto producto)

{

\_context.Productos.Update(producto);

await \_context.SaveChangesAsync();

}

public async Task DeleteAsync(int id)

{

var producto = await \_context.Productos.FindAsync(id);

if (producto != null)

{

\_context.Productos.Remove(producto);

await \_context.SaveChangesAsync();

}

}

}

}

**SERVICES/**

using T\_Electronic.Logic.Interfaces;

using T\_Electronic.Logic.Models;

using T\_Electronic.Logic.Repository;

namespace T\_Electronic.Logic.Services

{

public class ProductoService : IProductoService

{

private readonly IProductoRepository \_productoRepository;

public ProductoService(IProductoRepository productoRepository)

{

\_productoRepository = productoRepository;

}

public async Task<IEnumerable<Producto>> GetAllProductosAsync()

{

return await \_productoRepository.GetAllAsync();

}

public async Task<Producto> GetProductoByIdAsync(int id)

{

return await \_productoRepository.GetByIdAsync(id);

}

public async Task<Producto> GetProductoByNameAsync(string nombre) // ¡Implementación en el servicio!

{

return await \_productoRepository.GetByNameAsync(nombre);

}

public async Task AddProductoAsync(Producto producto)

{

await \_productoRepository.AddAsync(producto);

}

public async Task UpdateProductoAsync(Producto producto)

{

await \_productoRepository.UpdateAsync(producto);

}

public async Task DeleteProductoAsync(int id)

{

await \_productoRepository.DeleteAsync(id);

}

}

}

**CONTROLLERS/**

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using T\_Electronic.Logic.Interfaces;

using T\_Electronic.Logic.Models;

namespace T\_Electronic.Web.Controllers

{

public class ProductoController : Controller

{

private readonly IProductoService \_productoService;

private readonly ICategoriaService \_categoriaService;

public ProductoController(IProductoService productoService, ICategoriaService categoriaService)

{

\_productoService = productoService;

\_categoriaService = categoriaService;

}

// ... Métodos GET/POST existentes ...

// GET: Producto/BuscarPorNombre?nombre=ejemplo

[HttpGet("BuscarPorNombre")]

public async Task<IActionResult> BuscarPorNombre(string nombre)

{

if (string.IsNullOrWhiteSpace(nombre))

{

return BadRequest("El nombre no puede estar vacío.");

}

var producto = await \_productoService.GetProductoByNameAsync(nombre);

if (producto == null)

{

return NotFound($"No se encontró ningún producto con el nombre '{nombre}'.");

}

return View("Details", producto);

}

}

}

**PORQUE ENTITY FRAMEWORK Y DAPPER**

Esa es una pregunta excelente y va al corazón del diseño de arquitecturas de software eficientes. La combinación de Dapper y Entity Framework Core es una práctica común para optimizar el rendimiento, y se conoce como el patrón del **"mejor de ambos mundos"**.

### ¿Por qué se usa una combinación de Dapper y EF Core?

La razón principal es que **Entity Framework Core y Dapper están diseñados para propósitos diferentes**. No son competidores directos, sino que se complementan entre sí, cada uno destacando en su tarea.

#### EF Core: La Conveniencia y el Poder del ORM

Entity Framework Core es un **ORM (Object-Relational Mapper)** completo. Su objetivo es hacer tu vida más fácil al abstraer la base de datos por completo.

* **Para la escritura (Add, Update, Delete)**: EF Core es insuperable. Cuando llamas a \_context.SaveChanges(), EF Core se encarga de todo el **"trabajo pesado"**:
  + Realiza un seguimiento de todos los cambios que has hecho a tus objetos (ediciones, eliminaciones, etc.).
  + Genera automáticamente los comandos SQL de inserción, actualización o eliminación.
  + Maneja las transacciones de manera segura para garantizar que todas las operaciones se completen o se reviertan si algo falla.
* **En resumen**: Para las operaciones de escritura, el poder de **abstracción** y **gestión de transacciones** de EF Core es una gran ventaja, ya que te permite concentrarte en la lógica de negocio en lugar de en el SQL.

### Dapper: La Velocidad y el Control

Dapper es un **micro-ORM**. Su única responsabilidad es ejecutar consultas SQL y mapear los resultados a tus objetos C# de la manera más rápida posible. Dapper no tiene seguimiento de cambios, ni un modelo de datos complejo, ni genera consultas SQL. Tú mismo escribes el SQL.

* **Para la lectura (Get)**: Dapper brilla con luz propia. Al no tener el "equipaje" del seguimiento de cambios y de la generación de consultas, Dapper es **mucho más rápido** que EF Core en las operaciones de lectura.
  + **Rendimiento**: Ejecuta consultas de forma casi tan rápida como el código ADO.NET nativo.
  + **Control**: Tienes **control total** sobre la consulta SQL, lo que te permite optimizarla manualmente para casos de uso específicos, como consultas complejas con JOIN o la carga de datos masivos.

### Conclusión: La Estrategia Híbrida

La estrategia es utilizar la mejor herramienta para cada tarea:

* **Usar EF Core para la escritura** es ideal porque su sistema de seguimiento de cambios y su conveniencia te ahorran mucho tiempo y te ayudan a evitar errores.
* **Usar Dapper para la lectura** es perfecto para las partes de tu aplicación donde el rendimiento es crítico. La mayoría de las aplicaciones tienen muchas más lecturas que escrituras, por lo que esta optimización puede tener un gran impacto en la velocidad.

Tu código es un ejemplo perfecto de esta estrategia. El ProductoRepository es el único que sabe si está usando EF Core o Dapper, mientras que tu ProductoService y tu ProductoController solo ven el contrato de la interfaz IProductoRepository. Esto demuestra que tu arquitectura está muy bien diseñada y es fácil de mantener.

**CONEXION A SQL Y CORS**

**Appseting.json**

"ConnectionStrings": {

"DefaultConnection": "Server = LUKYBOOK\\SQLEXPRESS01; Database = TiendaElectronica; Trusted\_Connection = True; MultipleActiveResultSets=true;Encrypt=False"

},

"OrigenesPermitidos": "http://localhost:4200"

**Program.css**

using Electronic.Logic.Data;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Configuration;

// Define el nombre de la política de CORS.

var MyAllowSpecificOrigins = "\_myAllowSpecificOrigins";

// Configura el constructor de la aplicación web.

var builder = WebApplication.CreateBuilder(args);

// Configura el servicio de base de datos (DbContext).

// La aplicación usará la cadena de conexión "DefaultConnection" de appsettings.json

// para conectarse a SQL Server.

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

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

// Agrega los servicios necesarios para que los controladores API funcionen.

builder.Services.AddControllers();

// Agrega los servicios de OpenAPI para la documentación automática de la API (ej. Swagger UI).

builder.Services.AddOpenApi();

// Lee la configuración de orígenes permitidos desde appsettings.json.

var origenesPermitidos = builder.Configuration.GetValue<String>("OrigenesPermitidos")!.Split(",");

// Agrega los servicios de CORS para permitir peticiones desde otros dominios.

builder.Services.AddCors(options =>

{

// Define una política de seguridad que permite solicitudes desde los orígenes especificados.

options.AddPolicy(name: MyAllowSpecificOrigins, policy =>

{

policy.WithOrigins(origenesPermitidos).AllowAnyHeader().AllowAnyMethod();

});

});

// Construye la aplicación web a partir de la configuración.

var app = builder.Build();

// Configura el pipeline de solicitudes HTTP.

// Habilita la interfaz de usuario de la documentación de la API en el entorno de desarrollo.

if (app.Environment.IsDevelopment())

{

app.MapOpenApi();

}

// Middleware que redirige automáticamente las solicitudes HTTP a HTTPS.

app.UseHttpsRedirection();

// Middleware que enruta las solicitudes a los controladores correctos.

// DEBE ir antes de UseCors y UseAuthorization.

app.UseRouting();

// Middleware que habilita la política de CORS definida.

app.UseCors(MyAllowSpecificOrigins);

// Middleware que gestiona la autenticación y autorización del usuario.

// DEBE ir después de UseCors.

app.UseAuthorization();

// Mapea las rutas de los controladores a los puntos de acceso de la aplicación.

app.MapControllers();

// Inicia la aplicación y el servidor web para escuchar las peticiones.

app.Run();

**MIGRACIONES DESDE LOGIC**

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Design;

using Microsoft.Extensions.Configuration;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace T\_Electronic.Logic.Data

{

public class T\_ElectronicDbContextFactory : IDesignTimeDbContextFactory<T\_ElectronicDbContext>

{

public T\_ElectronicDbContext CreateDbContext(string[] args)

{

// La ruta correcta al directorio raíz de la aplicación web

var basePath = Path.Combine(

Directory.GetCurrentDirectory(),

"..",

"T-Electronic.Web");

// Construir la configuración desde appsettings.json

IConfigurationRoot configuration = new ConfigurationBuilder()

.SetBasePath(basePath)

.AddJsonFile("appsettings.json")

.Build();

// Obtener la cadena de conexión de la configuración

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

var optionsBuilder = new DbContextOptionsBuilder<T\_ElectronicDbContext>();

optionsBuilder.UseSqlServer(connectionString);

return new T\_ElectronicDbContext(optionsBuilder.Options);

}

}

}

### ¿Por Qué Necesitas un DbContextFactory?

Esa es una pregunta excelente, porque el archivo T\_ElectronicDbContextFactory no es parte de la lógica de tu aplicación en sí, sino una herramienta auxiliar que resuelve un problema específico del desarrollo. La clave está en la interfaz que implementa: **IDesignTimeDbContextFactory<T\_ElectronicDbContext>**.

Para explicarlo de forma simple, imagina que tu aplicación web es como una orquesta, y tu DbContext es el violín principal. Cuando la orquesta está tocando (es decir, cuando tu aplicación está en ejecución), un director (Program.cs) sabe exactamente dónde encontrar y cómo configurar ese violín, porque tiene toda la información a mano.

El problema surge cuando quieres hacer una **migración**. El comando dotnet ef migrations add no ejecuta toda tu orquesta; es más como un luthier que necesita el violín para hacer un ajuste. Este luthier (las herramientas de diseño de EF) no tiene al director de la orquesta (Program.cs) para que le diga cómo crear una instancia del DbContext. Simplemente está en un entorno de desarrollo, aislado del flujo normal de la aplicación.

Aquí es donde entra el DbContextFactory.

#### 1. El Propósito del IDesignTimeDbContextFactory

La interfaz IDesignTimeDbContextFactory es un **contrato** que le dice a las herramientas de Entity Framework: "Si no puedes encontrar cómo crear una instancia del DbContext por tu cuenta, usa este método para hacerlo". Proporciona la receta que las herramientas necesitan para:

1. Encontrar el archivo appsettings.json (que contiene la cadena de conexión).
2. Leer esa cadena de conexión.
3. Configurar el DbContext con esa información.
4. Crear y devolver una nueva instancia del DbContext.

#### 2. Desglose de tu Código

Tu código hace exactamente esto de forma manual, lo que es necesario porque las herramientas de EF no saben dónde está tu archivo de configuración por defecto.

* var basePath = Path.Combine(...): Este código es fundamental. Le dice al DbContextFactory que busque el archivo appsettings.json en un directorio específico. El .. se usa para navegar hasta el directorio padre y luego encontrar la carpeta de tu proyecto web (T-Electronic.Web).
* IConfigurationRoot configuration = new ConfigurationBuilder()...: Esto construye una instancia de configuración. Es la forma manual de leer los valores del archivo JSON, tal como lo hace tu aplicación automáticamente cuando se inicia.
* var connectionString = configuration.GetConnectionString("DefaultConnection"): Con la configuración ya cargada, simplemente lee la cadena de conexión con el nombre que definiste.
* optionsBuilder.UseSqlServer(connectionString): Esta línea le dice a la base de datos que tu DbContext va a usar SQL Server y le proporciona la cadena de conexión.
* return new T\_ElectronicDbContext(optionsBuilder.Options): Finalmente, crea una nueva instancia de tu DbContext usando las opciones de configuración que acabas de construir.

En resumen, este archivo es la **solución de emergencia** que creaste para que las herramientas de Entity Framework puedan trabajar. Es la "hoja de instrucciones" que les permite conectarse a la base de datos para ejecutar comandos como Add-Migration, Update-Database o Scaffold-DbContext.

AUTENTICACION

**1. Instalar los Paquetes NuGet**

Abre tu terminal o la consola del Administrador de paquetes en Visual Studio y ejecuta los siguientes comandos en el directorio de tu proyecto:

1. **Microsoft.AspNetCore.Authentication.JwtBearer**: Este paquete contiene la funcionalidad que tu API necesita para validar los tokens JWT que llegan en las solicitudes.

dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer

1. **System.IdentityModel.Tokens.Jwt**: Es la librería principal para generar y validar los JWT. Aunque la primera puede tener algunas dependencias, es bueno asegurarte de tenerla explícitamente.

dotnet add package System.IdentityModel.Tokens.Jwt

1. **BCrypt.Net-Next**: Como mencioné antes, este paquete es crucial para hashear y verificar las contraseñas de manera segura, lo que evita guardarlas en texto plano en la base de datos.

dotnet add package BCrypt.Net-Next

**2. Configurar el JWT en Program.cs**

Una vez que tengas los paquetes, el siguiente paso es **configurar el middleware de autenticación** en tu archivo Program.cs (si estás usando ASP.NET Core 6 o superior).

Debes agregar el servicio de autenticación y especificar que usarás JwtBearer para validar los tokens. Aquí te muestro cómo se vería la configuración:

1. **Define una clave secreta para firmar los tokens**. Esta clave es fundamental para la seguridad, ya que se usa para verificar la autenticidad del token. Es una buena práctica guardarla en el archivo appsettings.json y no en el código.

**En appsettings.json:**

JSON

"AppSettings": {

"Token": "esta-es-una-clave-super-secreta-y-larga-para-tu-api-con-jwt"

},

**Importante**: La clave debe ser larga y compleja. ¡No uses el ejemplo en producción!

1. **Agrega los servicios de autenticación en Program.cs**:

// Primero, agrega el servicio de autenticación

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)

.AddJwtBearer(options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

// Verifica que la clave de la firma sea correcta

ValidateIssuerSigningKey = true,

// Define la clave secreta

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8

.GetBytes(builder.Configuration.GetSection("AppSettings:Token").Value)),

// Deshabilita la validación del emisor (issuer) y la audiencia (audience) por simplicidad inicial

ValidateIssuer = false,

ValidateAudience = false

};

});

1. **Habilita el middleware de autenticación y autorización**:

var app = builder.Build();

// ... (otras configuraciones)

app.UseAuthentication();

app.UseAuthorization();

// ...

Con estos pasos, tu proyecto estará preparado para generar y validar tokens JWT. El siguiente paso sería la lógica para el registro y el login, lo cual iría en tu lógica de clases.