UTN
Facultad Regional Córdoba

TUP

Tecnicatura Universitaria en Programación

PROGRAMACIÓN II

Unidad Temática N°3:
Mapeo Objeto-Relacional

Material Teórico
1° Año – 2° Cuatrimestre

MAPEO OBJETO-RELACIONAL

Introducción

Un ORM (Object Relational Mapping o Mapeo Objeto-Relacional en castellano) es una herramienta que permite mapear, o lo que es lo mismo, convertir los objetos de una aplicación a un formato adecuado para ser almacenados en cualquier base de datos, creando para ello una base de datos virtual donde los datos disponibles en nuestra aplicación quedan vinculados con la base de datos final.

Lo que se obtiene de la definición anterior, es que además de convertir, los ORM ayudan a eliminar todo el lenguaje tedioso de sentencias SQL necesario para realizar las acciones CRUD (Create, Read, Update, Delete) en el código, ya que es el propio ORM quien se encarga de ello.

En el desarrollo de aplicaciones web modernas, la integración de un ORM como es Entity Framework Core (EF) en las soluciones .NET es esencial para una gestión eficiente y estructurada de datos. Esta unidad detalla cómo utilizar EF para construir una API web en C#, desde la configuración inicial hasta la implementación de operaciones CRUD avanzadas y consultas complejas.

ENTITY FRAMEWORK (EF) CORE

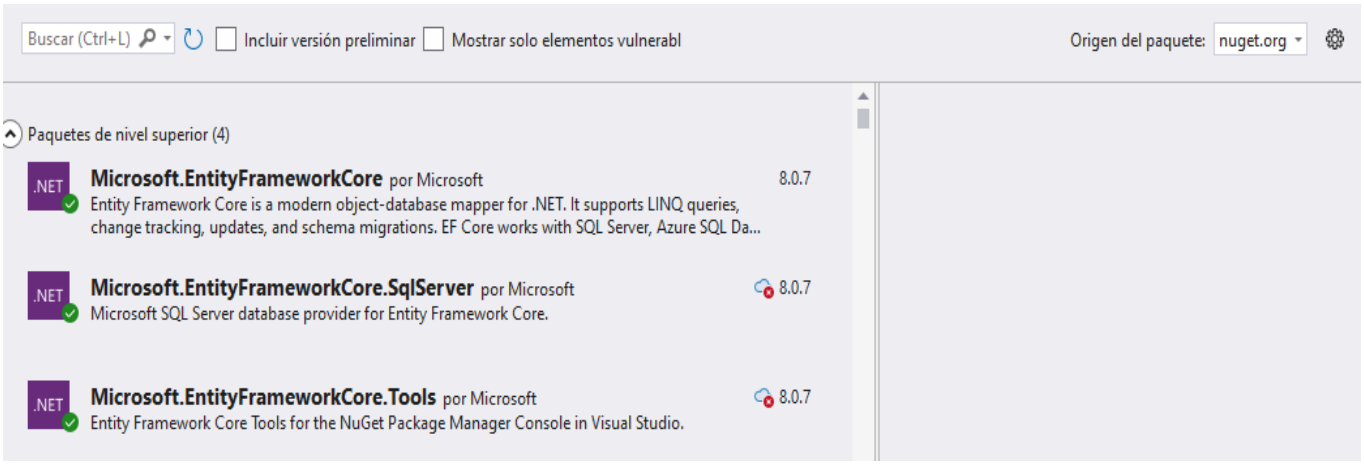
Entity Framework Core es un mapeador de bases de datos de objetos moderno para .NET. Admite consultas LINQ, seguimiento de cambios, actualizaciones y migraciones de esquemas. EF Core funciona con SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL y otras bases de datos a través de una API de complemento de proveedor. Es una versión ligera, extensible, de código abierto y multiplataforma de la popular tecnología de acceso a datos.

EF Core puede actuar como asignador relacional de objetos, que se encarga de lo siguiente:

- Permite a los desarrolladores de .NET trabajar con una base de datos usando objetos .NET.
- Permite prescindir de la mayor parte del código de acceso a datos que normalmente es necesario escribir.

Configuración del Proyecto

Antes de comenzar, es fundamental configurar un proyecto de API web y asegurarse de tener instalado Entity Framework Core para la interacción con la base de datos. Para instalar EF Core, instale el paquete de los proveedores de bases de datos de EF Core que quiera establecer como destino. En ese caso se utiliza SQLServer, tal como se muestra en la siguiente imagen:



Modelado de Datos

El modelado de datos es crucial en cualquier aplicación que utilice Entity Framework para interactuar con una base de datos relacional. Aquí se define cómo se estructuran y se relacionan las entidades dentro del contexto de la aplicación.

Definición de la Clase Producto

En este ejemplo la clase Producto se define con las propiedades básicas como Id, Nombre, Precio, FechaBaja, y MotivoBaja. Estas propiedades representan los campos de la tabla Productos en la base de datos.

```
public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public decimal Precio { get; set; }
    public DateTime? FechaBaja { get; set; }
    public string MotivoBaja { get; set; }
}
```

Donde:

- **Id**: Es el identificador único de cada producto, generalmente generado automáticamente por la base de datos.
- **Nombre**: El nombre del producto.
- **Precio**: El precio del producto.
- **FechaBaja** y **MotivoBaja**: Estos campos indican la fecha y el motivo por los cuales un producto puede ser dado de baja.
- Por defecto tanto el nombre de la clase como el nombre de cada propiedad se corresponderán unívocamente con el nombre de la tabla y columnas respectivamente. En caso de tener diferencias entre ambos identificadores es posible indicar dicha situación en el propio modelo, tal como se muestran en el siguiente código:

```
[Table("T_Productos")]

public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public decimal Precio { get; set; }

    [Column("Fecha_De_Baja")] // Cambia el nombre del campo en la
                              //base de datos
    public DateTime? FechaBaja { get; set; }
```

```
    public string MotivoBaja { get; set; }
}
```

- En este ejemplo, el atributo `[Column("Fecha_De_Baja")]` indica que en la base de datos el campo se llamará `Fecha_De_Baja` en lugar de `FechaBaja`. Por su parte en la base de datos existirá una tabla llamada `T_Productos` en lugar de `Producto`. Este tipo de configuraciones también es posible definirlo en el método **OnModelCreating** de la clase DbContext, tal como se indica a continuación:

```
public class ApplicationDbContext : DbContext
{
    public
ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
: base(options)
    {
    }

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

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
        // Cambiar el nombre de la tabla para la entidad Producto
        modelBuilder.Entity<Producto>().ToTable("T_Productos");

        // Cambiar el nombre del campo FechaBaja en la entidad
        //Producto
        modelBuilder.Entity<Producto>()
                    .Property(p => p.FechaBaja)
                    .HasColumnName("Fecha_De_Baja");

        // Configuración de relaciones u otras configuraciones
        //aquí...
    }
}
```

Especificación de Campos Obligatorios y Primary Key

Para indicar explícitamente qué campos son obligatorios y cuál es la clave primaria, se puede utilizar data annotations o configuraciones en el modelo de datos, como se muestra a continuación:

```
public class Producto
{
    [Key] // Especifica que Id es la clave primaria
    public int Id { get; set; }

    [Required] // Nombre es requerido
    public string Nombre { get; set; }

    [Required] // Precio es requerido
    public decimal Precio { get; set; }
```

```
    public DateTime? FechaBaja { get; set; }
    public string MotivoBaja { get; set; }
}
```

Contexto de Base de Datos

El contexto de base de datos en Entity Framework representa **una sesión con la base de datos**, y es responsable de configurar las relaciones entre las entidades y de mapear estas entidades a las tablas correspondientes en la base de datos.

```
public class ApplicationDbContext:DbContext
{
    public
ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options): base(options)
    {
    }

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

En este ejemplo:

- **ApplicationDbContext**: Es la clase que hereda de DbContext y representa el contexto de la base de datos para la aplicación (el identificador lo elige el desarrollador y es una buena práctica sufijar el nombre con **DbContext**).
  El parámetro DbContextOptions<ApplicationDbContext> options en el constructor del DbContext proporciona las opciones de configuración necesarias para inicializar y configurar el contexto de base de datos ApplicationDbContext. Este parámetro es fundamental para establecer cómo Entity Framework interactuará con la base de datos subyacente y cómo se configurarán las conexiones y otros aspectos de la persistencia de datos.
- **DbSet**<Producto>: Es una propiedad DbSet que representa la colección de entidades de tipo Producto. Entity Framework utiliza DbSet para realizar operaciones de consulta, inserción, actualización y eliminación sobre la tabla correspondiente en la base de datos. Notar que se declara como `public` porque esta clase es utilizada por el propio EF.

Configuración de DbContextOptions

La configuración de DbContextOptions<ApplicationDbContext> se realiza típicamente en la clase Program en los proyectos Asp.Net Core Web API (que para proyectos anteriores se realizaba mediante el método ConfigureServices de la clase Startup). Aquí se establecen las opciones de conexión a la base de datos, el proveedor de base de datos utilizado (SQL Server, PostgreSQL, MySQL, etc.), y otras configuraciones específicas de Entity Framework Core. El siguiente código muestra cómo en la clase Program es posible configurar directamente las opciones del DBContext:

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

      // Add services to the container.
      builder.Services.AddDbContext<ApplicationDbContext>(options =>
      options.UseSqlServer(builder.Configuration.GetConnectionString
      ("DefaultConnection")));
  }
```

Explicación del Código

- WebApplication.CreateBuilder(args): Este método inicializa un constructor de aplicaciones web que proporciona un WebApplicationBuilder (builder) que se utiliza para configurar la aplicación.
- builder.Services.AddDbContext<ApplicationDbContext>(...): Este método agrega el contexto de la base de datos ApplicationDbContext al contenedor de servicios (Services). Se configura el proveedor de base de datos SQL Server y se obtiene la cadena de conexión desde la configuración de la aplicación (builder.Configuration.GetConnectionString("DefaultConnection")). Para la configuración de la cadena de conexión es necesario editar el archivo appsettings.json del proyecto como se muestra a continuación:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
```

```
    }
}
```

Expandiendo el ejemplo

La idea es expandir el ejemplo con una relación entre productos y categorías para entender el concepto de modelo. Por análisis del dominio *"todo producto pertenece a una categoría específica"*. Aquí se muestra cómo definir estas entidades y su relación en Entity Framework:

```
public class Producto
{
    [Key]
    public int Id { get; set; }

    [Required]
    public string Nombre { get; set; }

    [Required]
    public decimal Precio { get; set; }

    public DateTime? FechaBaja { get; set; }
    public string MotivoBaja { get; set; }

    // Relación con Categoría
    public int CategoriaId { get; set; }
    public Categoria Categoria { get; set; }
}


public class Categoria
{
    [Key] // Especifica que Id es la clave primaria
    public int Id { get; set; }

    [Required] // Nombre es requerido
    public string Nombre { get; set; }

    // Relación con Productos
    public List<Producto> Productos { get; set; }
}
```

Notar que:

- **Producto**: tiene una propiedad CategoriaId que representa la clave externa a la tabla Categorias. La propiedad de navegación Categoria permite acceder a la categoría relacionada.
- **Categoria**: contiene una lista de productos (Productos) que pertenecen a esta categoría.
- Es necesario actualizar la **configuración en el DBContext**

```
public class ApplicationDbContext : DbContext
{
    public
ApplicationDbContext(DbContextOptions<ApplicationDbContext>
options) : base(options)
    {
    }

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

    protected override void OnModelCreating(ModelBuilder
modelBuilder)
    {
       // Configuración de la relación uno a muchos entre
      // Producto y Categoria
        modelBuilder.Entity<Producto>()
                    .HasOne(p => p.Categoria)
                    .WithMany(c => c.Productos)
                    .HasForeignKey(p => p.CategoriaId);
    }
}
```

En el método **OnModelCreating**, se configura la relación uno a muchos entre Producto y Categoria. Esto establece que un producto pertenece a una sola categoría, pero una categoría puede tener varios productos.

Relaciones entre entidades

Entity Framework Core soporta varios tipos de relaciones entre entidades, cada una con su propósito y uso específico. A continuación, se enumeran los tipos de relaciones más comunes, junto con ejemplos básicos de cómo se pueden implementar.

1. Relación Uno a Uno (One-to-One)

En una relación uno a uno, una entidad A está relacionada con exactamente una entidad B y viceversa.

Ejemplo**:**

```
public class Estudiante
{
    public int EstudianteId { get; set; }
    public string Nombre { get; set; }
    public Direccion Direccion { get; set; }
}

public class Direccion
{
    public int DireccionId { get; set; }
    public string Ciudad { get; set; }
```

```
    public string Calle { get; set; }

    // Propiedad de navegación inversa
    public Estudiante Estudiante { get; set; }
}
```

En este caso, cada estudiante tiene exactamente una dirección asociada.

2. Relación Uno a Muchos (One-to-Many)

En una relación uno a muchos, una entidad A está relacionada con una colección de entidades B, pero cada entidad B solo puede estar relacionada con una entidad A.

Ejemplo:

```
public class Departamento
{
    public int DepartamentoId { get; set; }
    public string Nombre { get; set; }
    public ICollection<Empleado> Empleados { get; set; }
}

public class Empleado
{
    public int EmpleadoId { get; set; }
    public string Nombre { get; set; }

    // Clave foránea
    public int DepartamentoId { get; set; }

    // Propiedad de navegación inversa
    public Departamento Departamento { get; set; }
}
```

En este caso, un departamento puede tener muchos empleados, pero cada empleado solo puede pertenecer a un departamento.

3. Relación Muchos a Muchos (Many-to-Many)

En una relación muchos a muchos, una entidad A puede estar relacionada con muchas entidades B, y viceversa.

Ejemplo:

```
public class Estudiante
{
    public int EstudianteId { get; set; }
    public string Nombre { get; set; }
    public ICollection<CursoEstudiante> CursosEstudiantes { get;
set; }
}

public class Curso
{
    public int CursoId { get; set; }
```

```
    public string Nombre { get; set; }
    public ICollection<CursoEstudiante> CursosEstudiantes { get;
set; }
}

public class CursoEstudiante
{
    public int EstudianteId { get; set; }
    public Estudiante Estudiante { get; set; }

    public int CursoId { get; set; }
    public Curso Curso { get; set; }
}
```

En este ejemplo, un estudiante puede estar inscrito en muchos cursos y un curso puede tener muchos estudiantes inscritos.

Configuración en Entity Framework Core

La configuración de estas relaciones se realiza utilizando Fluent API en el método OnModelCreating del contexto de base de datos (DbContext), donde se especifica cómo se mapean las claves y las relaciones entre las entidades.

- Ejemplo de Configuración Uno a Uno:

```
modelBuilder.Entity<Estudiante>()
    .HasOne(e => e.Direccion)
    .WithOne(d => d.Estudiante)
    .HasForeignKey<Direccion>(d => d.EstudianteId);
```

- Ejemplo de Configuración Uno a Muchos:

```
modelBuilder.Entity<Departamento>()
    .HasMany(d => d.Empleados)
    .WithOne(e => e.Departamento)
    .HasForeignKey(e => e.DepartamentoId);
```

- Ejemplo de Configuración Muchos a Muchos:

```
modelBuilder.Entity<CursoEstudiante>()
    .HasKey(ce => new { ce.EstudianteId, ce.CursoId });

modelBuilder.Entity<CursoEstudiante>()
    .HasOne(ce => ce.Estudiante)
    .WithMany(e => e.CursosEstudiantes)
    .HasForeignKey(ce => ce.EstudianteId);

modelBuilder.Entity<CursoEstudiante>()
    .HasOne(ce => ce.Curso)
    .WithMany(c => c.CursosEstudiantes)
    .HasForeignKey(ce => ce.CursoId);
```

Conclusiones

Al utilizar Entity Framework Core para modelar y trabajar con relaciones entre entidades, es esencial comprender los diferentes tipos de relaciones y cómo configurarlas adecuadamente en el contexto de tu aplicación. Esto no solo facilita la gestión de datos complejos, sino que también optimiza el rendimiento de las consultas y asegura la integridad referencial en la base de datos.

Carga diferida vs carga anticipada

En Entity Framework Core, además de definir las relaciones entre entidades, es importante considerar cómo se cargarán los datos relacionados cuando se accede a través de la propiedad de navegación. Entity Framework Core ofrece dos enfoques principales para cargar datos relacionados: Lazy Loading (carga diferida) y Eager Loading (carga anticipada).

Lazy Loading (Carga Diferida)

Lazy Loading es una técnica donde los datos relacionados no se cargan automáticamente cuando se recupera la entidad principal. En su lugar, se cargan desde la base de datos la primera vez que se accede a la propiedad de navegación relacionada. Para habilitar Lazy Loading en Entity Framework Core, sigue estos pasos:

1. Instalar Paquetes NuGet

   Instalar el paquete NuGet Microsoft.EntityFrameworkCore.Proxies. Esto proporciona soporte para la generación de proxies dinámicos necesarios para Lazy Loading.

2. Configuración en el DbContext

   En el método `OnConfiguring` de tu `DbContext`, habilita Lazy Loading con el método UseLazyLoadingProxies:

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
    optionsBuilder.UseLazyLoadingProxies()
                  .UseSqlServer("cadena_de_conexion");
}
```

3. Uso en Entidades

Definir las entidades con propiedades de navegación virtuales para permitir que Entity Framework Core genere proxies para ellas:

```
public class Estudiante
{
    public int EstudianteId { get; set; }
    public string Nombre { get; set; }
    public virtual Direccion Direccion { get; set; } // Propiedad
virtual
}

public class Direccion
{
    public int DireccionId { get; set; }
    public string Ciudad { get; set; }
    public string Calle { get; set; }

    public virtual Estudiante Estudiante { get; set; } //
Propiedad virtual
}
```

Eager Loading (Carga Anticipada)

Eager Loading es el proceso de cargar datos relacionados junto con la entidad principal de manera anticipada, en una sola consulta a la base de datos. Esto ayuda a minimizar la cantidad de consultas realizadas a la base de datos y optimiza el rendimiento de la aplicación en ciertos escenarios.

Por ejemplo, suponiendo que se necesita cargar anticipadamente la dirección cuando se recupera un estudiante, se puede configurar en el DbContext la siguiente propiedad:

```
protected override void OnModelCreating(ModelBuilder modelBuilder){
        modelBuilder.Entity<Estudiante>().Navigation(e => e.
                Direccion).AutoInclude();
}
```

Consideraciones

- Proxies Virtuales: Las propiedades de navegación deben ser declaradas como virtual para permitir que Entity Framework Core genere proxies que habiliten Lazy Loading.
- Rendimiento: Aunque Lazy Loading puede simplificar el código y optimizar la carga de datos, también puede conducir a problemas de N+1 si no se maneja correctamente.

Proceso de Migraciones en Entity Framework Core

Una vez definidos los modelos y configuraciones, el siguiente paso es impactar sobre la base de datos destino. Para ellos es necesario abrir una terminal o consola en el proyecto y ejecutar el siguiente comando de dotnet CLI para crear una nueva migración:

```
dotnet ef migrations add NombreDeLaMigracion
```

Donde NombreDeLaMigracion se remplaza con un nombre descriptivo que refleje los cambios realizados en el modelo. Este comando analiza las diferencias entre el modelo actual y la base de datos existente y genera archivos de migración del proyecto. Es necesario que el proyecto compile correctamente para que el comando se ejecute exitosamente.

Después de crear la migración, se aplican los cambios a la base de datos ejecutando el siguiente comando:

```
dotnet ef database update
```

Este comando aplica la última migración pendiente a la base de datos configurada en el DbContext. EF Core ejecutará automáticamente los scripts SQL necesarios para actualizar la estructura de la base de datos de acuerdo con los cambios definidos en la migración.

Consideraciones Importantes

- *Seguimiento de Migraciones*: Es importante mantener un registro de las migraciones aplicadas en tu proyecto para poder revertir cambios si es necesario o para migrar bases de datos en diferentes entornos (desarrollo, pruebas, producción).
- *Control de Versiones*: Las migraciones son parte del control de versiones de tu aplicación y deben ser compartidas entre los miembros del equipo y aplicadas en todos los entornos de manera consistente.
- *Actualización de Modelos*: Cada vez que realices cambios significativos en tus modelos de entidad o en las configuraciones de DbContext, deberías crear una nueva migración y aplicarla para reflejar esos cambios en la base de datos.

En resumen, las migraciones en Entity Framework Core son esenciales para mantener la consistencia entre el modelo de datos y la estructura de la base de datos subyacente, facilitando así la evolución de tu aplicación a medida que cambian los requisitos y el diseño del sistema.

PATRÓN REPOSITORY CON EF

El patrón Repository con Entity Framework Core, permite crear una capa de abstracción entre la lógica de negocio y el acceso a datos, lo cual facilita la separación de responsabilidades y mejora la mantenibilidad de la aplicación.

Implementación paso a paso

A continuación se describe paso a paso para crear un repositorio para el modelo Producto (analizado en los ejemplos anteriores) utilizando Entity Framework Core.

Paso 1: Definición de la Entidad Producto y Categoria

Primero, definir las entidades Producto y Categoria. Tomando el modelo anterior:

```
public class Producto
{
    public int ProductoId { get; set; }
    public string Nombre { get; set; }
    public decimal Precio { get; set; }
    public DateTime FechaBaja { get; set; }
    public string MotivoBaja { get; set; }

    // Relación con Categoria
    public int CategoriaId { get; set; }
    public Categoria Categoria { get; set; }
}

public class Categoria
{
    public int CategoriaId { get; set; }
    public string Nombre { get; set; }

    // Propiedad de navegación inversa
    public ICollection<Producto> Productos { get; set; }
}
```

Paso 2: Creación de Interfaces de Repositorio

Crear una interfaz IProductoRepository que defina las operaciones CRUD para gestionar productos. Esto permite una fácil sustitución de implementaciones y facilita las pruebas unitarias.

```
public interface IProductoRepository
{
    List<Producto> GetAll();
    Producto GetById(int id);
    void Save(Producto producto);
```

```
    void Delete(int id);
}
```

Paso 3: Implementar el repositorio

A continuación, se implementa esta interfaz en una clase concreta llamada ProductoRepository, utilizando EF Core para realizar las operaciones de acceso a datos:

```
public class ProductoRepository : IProductoRepository
{
    private readonly ApplicationDbContext _context;

    public ProductoRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public List<Producto> GetAll()
    {
        return _context.Producto.ToList();
    }

    public Producto GetById(int id)
    {
        return _context.Productos.FirstOrDefault(p => p.ProductoId ==
                id);
    }

    public void Save(Producto producto)
    {
        if (producto.ProductoId == 0)
        {
            _context.Productos.Add(producto);
        }
        else
        {
            _context.Productos.Update(producto);
        }
        _context.SaveChanges();
    }

    public void Delete(int id)
    {
        var producto = _context.Productos.Find(id);
        if (producto != null)
        {
            _context.Productos.Remove(producto);
            _context.SaveChanges();
        }
```

```
    }
}
```

Paso 4: Uso del Repositorio en Servicios

Finalmente, se puede utilizar el repositorio IProductoRepository tanto en servicios (capa de negocio) o controladores para interactuar con la base de datos de manera desacoplada:

```
public class ProductoService
{
    private readonly IProductoRepository _productoRepository;

    public ProductoService(IProductoRepository productoRepository)
    {
        _productoRepository = productoRepository;
    }

    public Producto ObtenerProductoPorId(int id)
    {
        return _productoRepository.GetById(id);
    }

    public List<Producto> ObtenerTodos ()
    {
        return _productoRepository.GetAll();
    }

    public void AgregarProducto(Producto producto)
    {
        _productoRepository.Save(producto);
    }

    public void ActualizarProducto(Producto producto)
    {
        _productoRepository.Save(producto);
    }

    public void EliminarProducto(int id)
    {
        _productoRepository.Delete(id);
    }
}
```

Consideraciones

- **Bloqueo del Hilo Principal:** Al utilizar métodos síncronos, cada operación de acceso a datos bloquea el hilo principal hasta que se completa. Esto

puede afectar la escalabilidad y la capacidad de respuesta de la aplicación, especialmente en operaciones intensivas de E/S.

- **Manejo de Excepciones:** manejar adecuadamente las excepciones que puedan surgir al interactuar con la base de datos de manera síncrona, para evitar problemas de rendimiento y estabilidad.

Introducción del asincronismo

Para mejorar la capacidad de respuesta de la aplicación y evitar bloqueos del hilo principal, es posible migrar las operaciones de acceso a datos a métodos asincrónicos utilizando `async` y `await`.

Palabras clave async y await

- **async:** La palabra clave `async` se utiliza para declarar que un método puede realizar operaciones asincrónicas. Esto permite que el método se ejecute de forma asincrónica, liberando el hilo principal mientras espera que se completen las operaciones.
- **await:** La palabra clave `await` se utiliza dentro de un método `async` para pausar la ejecución de ese método hasta que se complete una operación asincrónica. Cuando se encuentra con `await`, el control vuelve al método que llamó al método `async`, permitiendo que el hilo principal continúe ejecutándose si es necesario.

Flujo de Ejecución en Métodos Asincrónicos

Cuando se invoca un método marcado como **async**, comienza a ejecutarse de manera síncrona hasta que encuentra una operación **await**.

Al encontrar **await**, el método **async** devuelve el control al método que lo invocó, liberando el hilo principal para otras tareas.

Una vez que la operación asincrónica finaliza, el método **async** reanuda su ejecución desde el punto en que se detuvo, utilizando el hilo que está disponible en ese momento.

Ejemplo Básico

El siguiente ejemplo ilustra de manera sencilla cómo se utilizan async y await:

```
public async Task<int> ObtenerResultadoAsincrono()
{
    // Simulación de una operación asincrónica que tarda tiempo
    await Task.Delay(2000); // Espera asincrónica de 2 segundos

    return 42;
}

public async Task EjemploLlamadaAsincrona()
{
    Console.WriteLine("Inicio de Ejemplo");

    int resultado = await ObtenerResultadoAsincrono();

    Console.WriteLine($"Resultado obtenido: {resultado}");

    Console.WriteLine("Fin de Ejemplo");
}
```

En C#, **Task** es una representación de una unidad de trabajo asincrónica que se ejecuta en segundo plano. Permite ejecutar operaciones de manera asíncrona sin bloquear el hilo principal de ejecución. Su uso adecuado junto con async y await permite mejorar la capacidad de respuesta y la eficiencia de las aplicaciones

Implementación del Repositorio con Async

Por último, en la implementación concreta del repositorio (ProductoRepository), se integra el uso de métodos asíncronos de la siguiente manera:

```
public class ProductoRepository : IProductoRepository
{
    private readonly ApplicationDbContext _context;

    public ProductoRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<List<Producto>> GetAllAsync()
    {
        return await _context.Productos
       .ToListAsync();
```

```
        }

        public async Task<Producto> GetByIdAsync(int id)
        {
            return await _context.Productos
                .FirstOrDefaultAsync(p => p.ProductoId == id);
        }

        public async Task SaveAsync(Producto producto)
        {
            if (producto.ProductoId == 0)
            {
                _context.Productos.Add(producto);
            }
            else
            {
                _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();
            }
        }
    }
```

Consultas Avanzadas con LINQ y EF

Entity Framework permite escribir consultas avanzadas utilizando Language Integrated Query (LINQ), que facilita la manipulación de datos a través de consultas integradas en el código C#. Esto no solo mejora la eficiencia, sino que también asegura la seguridad al trabajar con la base de datos de manera estructurada y tipada.

El siguiente es un ejemplo básico para obtener una lista de productos ordenados por precio descendente:

```
using System.Linq;

public class ProductoService
{
```

```
    public List<Producto>
ObtenerProductosOrdenadosPorPrecioDescendente()
    {
        var productosOrdenados = _repository.GetAll()
            .OrderByDescending(p => p.Precio)
            .ToList();

        return productosOrdenados;
    }
}
```

En este ejemplo:

- OrderByDescending() es un método de LINQ que ordena los productos por el campo Precio en orden descendente.
- .ToList() convierte el resultado en una lista de objetos Producto.

Filtrado con Condiciones Compuestas

```
public List<Producto> FiltrarProductosPorCategoriaYPrecio(string
categoria, decimal precioMinimo)
{
    var productosFiltrados = _repository.GetAll()
        .Where(p => p.Categoria.Nombre == categoria && p.Precio >=
precioMinimo)
        .ToList();

    return productosFiltrados;
}
```

En este ejemplo, Where() se utiliza para filtrar productos basados en condiciones compuestas.

Beneficios de LINQ y Entity Framework

- **Legibilidad y Mantenibilidad:** LINQ ofrece una sintaxis legible que se asemeja al lenguaje natural, lo que facilita la comprensión del código y su mantenimiento.
- **Seguridad:** Al ser tipado y estar integrado con Entity Framework, LINQ garantiza que las consultas se traduzcan en consultas SQL seguras y optimizadas.
- **Optimización de Consultas:** Entity Framework se encarga de traducir las consultas LINQ en consultas SQL eficientes, aprovechando las capacidades del motor de base de datos subyacente.

WEB API CON ENTITY FRAMEWORK

Por último, solo resta integrar los conceptos abordados en las secciones anteriores con lo estudiado en la unidad anterior. El siguiente paso a paso pretende crear un ejemplo básico que demuestre cómo configurar y utilizar un repositorio para operaciones CRUD sobre una entidad de dominio en una API RESTful.

Paso a paso: Integración Web API con EF

Una vez creado el proyecto, definido correctamente el modelo mediante EF Core (continuando con el caso de Productos y Categorias) y ejecutada la migración sobre la base de datos, los pasos para completar el desarrollo de la Web API se enumeran a continuación:

1. **Configuración de Inyección de Dependencias**

En la clase Program.cs de. proyecto, se configura la inyección de dependencias para el repositorio y el contexto de Entity Framework.

```
var builder = WebApplication.CreateBuilder(args); // Configuración del

DbContext builder.Services.AddDbContext<ApplicationDbContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultCon
nection")));

// Configuración del Repositorio
builder.Services.AddScoped<IProductoRepository, ProductoRepository>();
// Otros servicios y configuraciones builder.Services.AddControllers();
```

2. **Crear un controlador y utilizar el repositorio.**

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

public class ProductosController : ControllerBase
{
    private readonly IProductoRepository _productoRepository;

    public ProductosController(IProductoRepository productoRepository)
    {
        _productoRepository = productoRepository;
    }

    // GET: api/Productos
    [HttpGet]
    public async Task<IActionResult> GetProductos()
    {
```

```
        var productos = await _productoRepository.GetAllAsync();
        return Ok(productos);
    }

    // GET: api/Productos/5
    [HttpGet("{id}")]
    public async Task<IActionResult> GetProducto(int id)
    {
        var producto = await _productoRepository.GetByIdAsync(id);
        if (producto == null)
        {
            return NotFound();
        }
        return Ok(producto);
    }

    // POST: api/Productos
    [HttpPost]
    public async Task<IActionResult> PostProducto(Producto producto)
    {
        await _productoRepository.SaveAsync(producto);
        return CreatedAtAction(nameof(GetProducto), new { id =
producto.ProductoId }, producto);
    }

    // PUT: api/Productos/5
    [HttpPut("{id}")]
    public async Task<IActionResult> PutProducto(int id, Producto
producto)
    {
        if (id != producto.ProductoId)
        {
            return BadRequest();
        }
        await _productoRepository.SaveAsync(producto);
        return NoContent();
    }

    // DELETE: api/Productos/5
    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteProducto(int id)
    {
        await _productoRepository.DeleteAsync(id);
        return NoContent();
    }
}
```

3. Consumir API

Probar la API utilizando herramientas como Postman o Swagger para realizar solicitudes HTTP a los endpoints definidos en el controlador ProductosController.

Consideraciones Finales

- **Seguridad y Validación:** Implementar validaciones y consideraciones de seguridad adecuadas en tu API, como la validación de entrada y el manejo de excepciones.
- **Pruebas Unitarias:** Es recomendable escribir pruebas unitarias para validar el comportamiento del repositorio y los controladores API.
- **Optimización de Consultas:** Utiliza técnicas como el uso de Include() para cargar entidades relacionadas y optimizar consultas.
- **Capa de servicios:** es altamente recomendable definir una capa de servicios e inyectarlos en los controladores, en vez de utilizar los repositorios directamente. Esto queda como ejercicio para el estudiante al momento de realizar las actividades prácticas propuestas en cada guía.

Integrar un repositorio con Entity Framework dentro de una Web API en C# proporciona una estructura organizada y escalable para manejar operaciones CRUD en una aplicación, aprovechando las capacidades de EF y LINQ para una manipulación eficiente de datos.

BIBLIOGRAFÍA

Julia Lerman. (2018). Programming Entity Framework: Building Data Centric Apps with the ADO.NET Entity Framework. O'Reilly Media.

Jon P. Smith. (2018). Entity Framework Core in Action. Manning Publications.

Documentación Oficial de Microsoft. Entity Framework Core. Recuperado de: https://learn.microsoft.com/en-us/ef/core/