Llaves fóraneas, patrón de repositorio e inyección de dependencias

Requerimientos

- Estar en un sistema *Linux* (en una distribución basada en *Debian*, *Ubuntu* o *Fedora*).
- Tener *.NetCore* instalado en el sistema, en su última version LTS (al momento la elaboración de esta guía es la versión 3.1).
- Tener *Git* instalado en el sistema.
- Un editor de texto o IDE.
- *Postman* o cualquier herramienta para probar APIs.
- Gestor de Base de Datos.
- Tener instalado EntityFrameworkCore en nuestro proyecto.

Configurando

Primero tenemos que instalar el siguiente paquete:

```
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson --version 3.1.2
```

Después en nuestro *Startup.cs*, en el método *ConfigureServices* lo modificamos y debería quedar de la siguiente manera:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContextPool<GestionArticulosContext>(options =>
        options.UseLazyLoadingProxies()
            .UseMySql(Configuration.
                GetConnectionString("DefaultConnection")));
    services.AddControllers().AddNewtonsoftJson(options =>
        options.SerializerSettings.ReferenceLoopHandling =
            Newtonsoft.Json.ReferenceLoopHandling.Ignore);
}
```

Agregando llaves foraneas

Creando una relación

Para definir una relación, en nuestro caso de Articulo a Proveedor. En la clase Articulo agregaremos los siguientes atributos:

```
// Este define la columma donde estara la llave foranea
public int ProveedorID { set; get; }
```

```
// Es objeto sera creado al obtener el articulo
public virtual Proveedor Proveedor { set; get; }
```

Una vez hecha la relación podemos hacer nuestra migración:

```
dotnet ef migrations add RelacionArticuloProveedor
```

Si se desea podemos verificar que la migración fue hecha correctamente, una vez hecho eso sincronizamos la base de datos

```
dotnet ef database update
```

Esto generara una columna en Articulos con el nombre ProveedorID.

Probando

Ejecutamos nuestro proyecto:

```
dotnet run
```

En *Postman* insertamos el siguiente cuerpo de solicitud para registrar un Articulo

```
{
    "Nombre": "Laptop",
    "Descripcion": "Laptop HP",
    "Precio": 15000.00,
    "ProveedorID": 1
}
```

Esto debería crearnos un Articulo que hace referencia a Proveedor con id = 1

Relación 1 a 1

Todos los pasos hecho en el tema pasado fueron para crear una relación de 1 a 1.

Si se hace la consulta desde Postman al Articulo recien registrado, debería devolvermos algo similar a esto:

```
/* Este es el objeto relaciónado */
{
"proveedor": {
    "id": 1,
    "nombre": "CompuSolucion",
    "telefono": "232131"
},
"lazyLoader": {},
"id": 3,
"nombre": "Laptop",
"descripcion": "Laptop HP",
"precio": 15000,
```

```
    "fechaRegistro": "2020-03-27T21:34:10.883827",
    "proveedorID": 1
}
```

Relación de 1 a muchos

Para declarar una relación de 1 a muchos, solo se debe de hacer lo siguiente a la clase de Proveedor:

```
public virtual ICollection<Articulo> Articulos { set; get; }
```

Despues de eso, hacemos la migración y sincronizamos con la base de datos

```
dotnet ef migrations add relaciónArticuloProveedorUnoAMuchos
dotnet ef database update
```

Ahora si queremos probar, en Postman podemos consultar al Proveedor que le hayamos registrado los Articulos y nos debería dar algo similiar a lo siguiente:

```
[
    {
        "articulos": [
            {
                "id": 3,
                "nombre": "Laptop",
                "descripcion": "Laptop HP",
                "precio": 15000.0,
                "fechaRegistro": "2020-03-27T21:34:10.883827",
                "proveedorID": 1
            },
            {
                "id": 4,
                "nombre": "Laptop",
                "descripcion": "Laptop HP 2",
                "precio": 15500.0,
                "fechaRegistro": "2020-03-29T13:50:30.712347",
                "proveedorID": 1
            }
        ],
        "id": 1,
        "nombre": "CompuSolucion",
        "telefono": "232131"
    }
]
```

Como se puede notar, al obtener el Proveedor, también nos da una lista de articulos registrados

Relación de muchos a muchos

> Nota: Este fragmento no es necesario seguirlo, esto solo tiene como propósito la demostración de como se implementa una relación de muchos a muchos.

Suponiendo que entre Articulos y Proveedores hubiera una relación de muchos a muchos.

Primero tenemos que hacer una clase que va representar a nuestra tabla intermedia, esta ser;ia de la siguiente manera:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Threading.Tasks;

namespace MiPrimeraApi.Models
{
    public class ArticuloTieneProveedor
    {
        public int ArticuloID { set; get; }
        public virtual Articulo Articulo { set; get; }
        public int ProveedorID{ set; get; }
        public virtual Proveedor Proveedor { set; get; }
    }
}
```

Después de esto, en nuestra clase de Proveedor y Articulo agregamos el siguiente atributo:

```
public virtual ICollection<ArticuloTieneProveedor> ArticulosTienenProveedores { set; get
```

> Nota: No es necesario que tengan el mismo nombre en las dos clases.

Una vez hecho eso, solo debemos agregar este atributo a nuestra clase *GestionArticulosContext*:

```
public DbSet<ArticuloTieneProveedor> ArticulosTienenProveedores { set; get; }
```

Y por último, debemos hacer la migración y sincronizar la base de datos

```
dotnet ef migrations add RelacionArticuloProveedorMuchosAMuchos
dotnet ef database update
```

Patrón repositorio

El patrón de repositorio tiene por objetivo crear una capa de abstracción entre la capa de acceso a los datos y la capa de lógica de negocio de una aplicación. La adición, eliminación, actualización y selección de elementos de esta colección se realiza a través de una serie de métodos sencillos, sin necesidad de tratar con asuntos de la base de datos como conexiones, comandos, cursores o lectores. Ademas su implementación puede ayudar a aislar nuestra aplicación de los cambios en el gestor de datos y a su vez facilita las pruebas de unidad automatizadas o el desarrollo basado en pruebas (TDD).

Para implementar el patrón repositorio solo hace falta hacer lo siguiente:

Crear una carpeta donde se alojaran todas las *Interfaces* y sus implementaciones.

En mi caso lo llamare a esta carpeta *Repositories*

En esta carpeta vamos a crear nuestra primera interfaz, la cuál voy a llamar *IArticuloRepository* y en esta interfaz agregaremos los métodos que necesitaremos, en este ejemplo sólo agregaré el método de obtener todos y obtener por id:

```
using MiPrimeraApi.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MiPrimeraApi.Repositories
{
    public interface IArticuloRepository
    {
        List<Articulo> ObtenerTodos();
        Articulo ObtenerPorId(int id);
    }
}
```

Despues de eso, creamos la clase *ArticuloRepository* y esta va a implementar nuestra interfaz y debería contener la siguiente manera:

```
using MiPrimeraApi.Models;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace MiPrimeraApi.Repositories
{
    public class ArticuloRepository : IArticuloRepository
    {
```

```
        private readonly GestionArticulosContext _contexto;

        public ArticuloRepository(GestionArticulosContext contexto) {
            _contexto = contexto;
        }

        public List<Articulo> ObtenerTodos()
        {
            var articulos = _contexto.Articulos.ToList();
            return articulos;
        }

        public Articulo ObtenerPorId(int id)
        {
            var articulo = _contexto.Articulos.FirstOrDefault(x => x.Id == id);
            return articulo;
        }
    }
}
```

Una vez implementados los repositorios, sólo hace falta aplicar inyección de dependencias en nuestros controladores para que se puedan ocupar los repositorios.

inyección de dependencias

La inyección de dependencias es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase la que cree dichos objetos. Esos objetos cumplen contratos que necesitan nuestras clases para poder funcionar (de ahí el concepto de dependencia). Nuestras clases no crean los objetos que necesitan, sino que se los suministra otra clase *contenedora* que inyectará la implementación deseada a nuestro contrato.

Al momento, ya hemos creado una inyección de dependencias, la cual es nuestro *GestionArticulosContext*. Como se puede notar siempre que hemos ocupado *GestionArticulosContext* jamás se ha creado una instancia de esta, sólo llega como parámetro, esto es porque en nuestro método *ConfigureServices* en *Startup.cs* hemos declarado que aplique la inyección de dependencias por medio de las siguientes líneas:

```
services.AddDbContextPool<GestionArticulosContext>(options =>
            options.UseLazyLoadingProxies()
                .UseMySql(Configuration.
                    GetConnectionString("DefaultConnection")));
```

Ahora, sólo falta ocupar nuestras interfaces de repositorio como contratos para que sean inyectadas en nuestros controladores, para hacer esto se debe hacer lo siguiente:

En *Startup.cs*, agregamos el namespace de nuestros repositorios:

```
using MiPrimeraApi.Repositories;
```

Después, en el método *ConfigureServices*, antes de la línea de *services.AddControllers()...* agregamos lo siguiente:

```
services.AddScoped<IArticuloRepository, ArticuloRepository>();
// Solo es agregar servies.AddScoped<Interfaz, Implementacion>();
```

Ahora, para ocupar el repositorio en nuestro controlador se tiene que hacer lo siguiente:

Agregar el namespace de nuestros repositorios:

```
using MiPrimeraApi.Repositories;
```

Agregar como atributo la interfaz del repositorio:

```
private readonly IArticuloRepository _repoArticulo;
```

Y ahora el constructor del controlador lo dejamos de la siguiente manera:

```
public ArticuloController(IArticuloRepository repoArticulo)
{
    _repoArticulo = repoArticulo;
}
```

Por último, para acceder a estos solo se tiene que ocupar de la siguiente manera:

```
[HttpGet]
[Route("")]
public IActionResult Obtener()
{
    var articulos = _repoArticulo.ObtenerTodos();
    return Ok(articulos);
}

[HttpGet]
[Route("{id}")]
public IActionResult ObtenerPorId(int id)
{
    var articulo = _repoArticulo.ObtenerPorId(id);
    return Ok(articulo);
}
```

Si se desea, se puede probar en Postman y al hacer la consulta, debería entregar los mismos resultados que cuando no ocupaba la inyección de dependencias.