Entity framework

Entity Framework Core Code First

Paquetes Necesarios

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

DB Context

Esta clase es una implementacion del context de entity framework y los dbset serian las tablas

La clase ContextDb O es la puerta de acceso a la base de datos, aca tengo definidas en forma de DbSet cada una de las tablas de las entidades que voy a persistir

Tambien instalamos el paquete EntityFrameworkCore en el paquete WebApi o Factory para poder hacer services.AddDbContext<…>()

Context Factory

Sirve para crear contextos con distintas configuraciones, lo vamos a usar para testing, e indirectamente se usa cuando creamos las migraciones ya que EF Core busca alguna clase que herede de IDesignTimeDbContextfactory y si la encuentra, ejecuta nuestro metodo GetNewContext.

```
using Microsoft.Data.Sqlite;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.Extensions.Configuration;

namespace Vidly.DataAccess.Contexts;

// Para testing usar SQLite, por defecto usar SQLServer
public enum ContextType { SQLite, SQLServer }

public class ContextFactory : IDesignTimeDbContextFactory<VidlyContext>
{
    public VidlyContext CreateDbContext(string[] args)
    {
        return GetNewContext();
    }

    public static VidlyContext GetNewContext(ContextType type = ContextType.SQLServer)
    {
        var builder = new DbContextOptionsBuilder<VidlyContext>();
        DbContextOptions options = null;

        if (type == ContextType.SQLite)
        {
            options = GetSqliteConfig(builder);
        }
        else
        {
            options = GetSqlServerConfig(builder);
        }

        return new VidlyContext(options);
    }

    private static DbContextOptions GetSqliteConfig(DbContextOptionsBuilder builder)
    {
        var connection = new SqliteConnection("Filename=:memory:");
        builder.UseSqlite(connection);
        return builder.Options;
    }

    private static DbContextOptions GetSqlServerConfig(DbContextOptionsBuilder builder)
    {
        //Gets directory from startup project being used, NOT this class's path
        var directory = Directory.GetCurrentDirectory();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(directory)
            .AddJsonFile("appsettings.json")
            .Build();

        var connectionString = configuration.GetConnectionString("VidlyDB");
```

```
        builder.UseSqlServer(connectionString);
        return builder.Options;
    }
}
```

Luego para hacer las migraciones y para correrlas tenemos los siguientes comandos

```
dotnet ef migrations add NombreMigracion -p "Path al proyecto de DataAccess"
dotnet ef database update -p "Path al proyecto de DataAccess"
```

Migraciones

Las migraciones son la manera de mantener el schema de la BD sincronizado con el Dominio, por esto cada vez que se modifica el dominio se deberá crear una migración.

Commando	Descripción
dotnet ef migrations add NOMBRE_DE_LA_MIGRATION	Este comando creará la migración. Crea 3 archivos .cs 1) *: Contiene las operaciones Up() y Down() que se aplicaran a la BD para remover o añadir objetos. 2)* .Designer: Contiene la metadata que va a ser usada por EF Core. 3) ModelSnapshot: Contiene un snapshot del modelo actual. Que será usada para determinar qué cambio cuando se realice la siguiente migración.
dotnet ef database update	Este comando crea la BD en base al context, las clases del dominio y el snapshot de la migración.
dotnet ef migrations remove	Este comando remueve la ultima migración y revierte el snapshot a la migración anterior. Esto solo puede ocurrir si la migración no fue aplicada todavía.
dotnet ef database update NOMBRE_DE_LA_MIGRATION	Este commando lleva la BD al migración del nombre NOMBRE_DE_LA_MIGRATION.

Repositorios

Luego creamos los repositorios para cada caso en particular y los registramos para la inyección de dependencias.

Testing

Utilizamos el ContextFactory para crear un DbContext con SQLite, luego cargamos los datos al context a mano y testeamos los repositorios. → Mirar DataAccess.Test en el repo.