



# Guia Entity Framework CORE

Dejemosnos de teoría y empecemos a desarrollar un buen contexto respetando buenas practicas y siendo uniformes.

## Requerimientos previos de tecnologia

- SQL Server - Version: 2017 o superior
  - **Windows:** Es mas simple, seguir los pasos de [aquí](#)
  - **MacOS:** Requiere un par de pasos mas. Es necesario instalar Docker: Seguir los pasos de [aquí](#), [aquí](#), o [aquí](#)
- SQL GUI
  - **Windows::** La mejor opción es SQL Server Management Studio. Es la GUI mas completa para interactuar con estas bases de datos. Solo disponible para windows. [Link](#)
  - **MacOS/Linux:** Existen otras opciones. La mejor es [Azure Data Studio](#), herramienta open source de microsoft disponible para todas las plataformas (Windows, MacOS, Linux. Tambien se encuentra [DBeaver](#)

## Paquetes necesarios para trabajar

- **Microsoft.EntityFrameworkCore.**
  - Podemos crear un **modelado de las entidades de negocio** y crear una **clase contexto** que representa la **sesión** con la base de datos permitiéndonos realizar **query** y **guardar data**.
  - Lo vamos a instalar en el proyecto donde se encuentre el **contexto**. En nuestro caso el proyecto lo vamos a llamar **DataAccess**.
- **Microsoft.EntityFrameworkCore.Design.**
  - Este paquete sirve para que **EF Core** logre interpretar nuestras **entidades de negocio** y pueda hacer un **modelado de tablas** y aplicarlo a la base de datos.
  - Este paquete lo vamos a tener en el proyecto con el framework **netcoreapp3.1** que es desde donde se van a crear y ejecutar las migraciones.
- **Microsoft.EntityFrameworkCore.SqlServer.**
  - Es el proveedor de base de datos para **SQL**
  - Este paquete lo vamos a tener en el proyecto donde se encuentre la **configuracion del proveedor de base de datos**.
- **Microsoft.EntityFrameworkCore.Tools.**
  - Este paquete nos permite **crear y aplicar las migraciones** y generar código a partir de una base de datos existentes. Los comandos son una extension del comando multi plataforma [dotnet](#).
  - Este paquete lo vamos a instalar de forma **global**

## Let's get started

## Preparacion del ambiente

Lo primero que tenemos que hacer antes que nada es instalar el paquete **Microsoft.EntityFrameworkCore.Tools** de manera global. Para hacer ello tenemos que abrir la consola y escribir el siguiente comando:

```
dotnet tool install --global dotnet-ef
```

Para chequear que se haya instalado correctamente, correr el siguiente comando:

```
dotnet restore
dotnet ef
```

Como una pequeña introduccion, los comandos se refieren a un **proyecto** y a un **proyecto startup**.

- El **proyecto** es también conocido como **target project**, porque es donde los comandos van agregar o eliminar archivos. Por defecto, el proyecto en el directorio en el que se este corriendo el comando, es el **target project**. Se puede especificar otro proyecto como **target project** usando la opción **--project o --p**.
- El **proyecto startup** es en donde se compila y corren los comandos. Los comandos tienen que hacer un **build** de la aplicación para obtener información del proyecto, como por ejemplo el **connection string** y la configuración de las entidades de negocio como la del contexto. Por defecto, el proyecto en donde se este corriendo los comandos es el **startup**. Se puede especificar uno diferente con la opción **--startup-project**.

## Creando la capa DataAccess


Como se menciono anteriormente vamos a necesitar una clase en particular la cual es el **contexto**. Esta clase tiene que heredar de [DbContext](#).

Esta clase va tener las responsabilidades de representar la **sesión** con la base de datos y la **configuración** necesaria para las tablas.

Podemos ubicarlo en varios lugares, algunos de ellos puede ser:

- **DataAccess**
  - Tiene sentido ubicarlo aca ya que se van a encontrar todos los repositorios que van a ejecutar queries y guardar data utilizando [LINQ](#). Tal vez lo mas correcto seria ubicarlo en un paquete diferente al de los repositorios. Podriamos tener un paquete **DataAccess.Context** donde estaria el contexto y otro **DataAccess.Repository** donde estarian los repositorios.
- **Domain**
  - Se pondria aca ya que nuestro contexto es una clase concreta altamente acoplada a nuestras entidades del negocio lo cual tiene sentido que convivan bajo el mismo proyecto. De igual manera como se menciono anteriormente, lo mas correcto seria que estuviese en un inner package como **Domain.Context** y las entidades de negocio **Domain.Entity**. De todas formas en este proyecto ya estamos rompiendo con **SRP** porque maneja la responsabilidad de manejar la **sesión** con la bd y el **modelado de las entidades de negocio**.
- **Context**
  - Seria otro proyecto con este nombre el cual su **única** responsabilidad es la de mantener la **sesión** con la base de datos, por lo tanto estamos respetando **SRP**. Tiene las otras ventajas de que queda aislado de otras responsabilidades lo cual va a permitir que sea un componente reusable y reemplazable si se diseña de forma genérica. Esto significa que el contexto lo podemos reutilizar en otro proyecto lo cual limitaría los tiempos de desarrollo en un nuevo sistema y también si se quiere mantener de forma separada, se estaría cumpliendo al estar aislado en su propia capa.

Como nosotros vamos a estar viendo contextos concretos lo vamos a desarrollar en **DataAccess** para tambien limitar la complejidad.

Si siguieron la guía anterior (  Inyección de dependencias) ya deberían de tener una clase contexto la cual es **VidlyContext** ubicada en **DataAccess**.

## Configuración del proveedor de base de datos

Para realizar las migraciones se tiene que tener configurado el proveedor de base de datos correspondiente. En este caso vamos usar el proveedor de base de datos **Microsoft.EntityFrameworkCore.SqlServer**. Las migraciones como se corren en un ambiente de desarrollo sería correcto tener hardcoded el connection string de la bd, de igual manera hay que ser conscientes de que el connection string cuando esta en modo producción se tendría que leer desde un archivo de configuración, lo veremos también.

Ahora, donde realizamos la configuración del proveedor de base de datos? Donde debería de estar encapsulada esta data? Depende de lo que queramos lograr es como se configuraría. Si queremos lograr un contexto bastante extensible en el uso de diferentes proveedores de base de datos lo mejor sería que se le inyecte al contexto desde afuera y que este lo utilice mediante una abstracción. Si queremos algo mas concreto y mas rígido, se ubicaría de forma hardcodeda en el contexto mismo. Si siguen en dudas de como realizar esta responsabilidad, el patron **experto** de los **GRASP** los ayudara a darse cuenta de como realizarlo.

## Contexto concreto

Para realizarlo de forma **concreta**, esto quiere decir que lo haremos **dentro** de la clase contexto, fue porque identificamos que aca es el mejor lugar donde debería de estar la configuración. Veamos como quedaría **VidlyContext** y hagamos un pequeño análisis.

```
namespace DataAccess.Context
{
    public class VidlyContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }

        public VidlyContext() { }
        public VidlyContext(DbContextOptions options) : base(options) { }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer("Server=ENVY15\\DANIELACEVEDO; Database=VidlyDB; Integrated Security=True; Trusted_Connec
        }
    }
}
```

Esta es nuestra clase ya conocida **VidlyContext**, el único código nuevo y que analizaremos es el que esta con **fondo amarillo**.

Se puede ver el método **DbContext.OnConfiguring(DbContextOptionsBuilder)** el cual es un método que se encuentra en la clase padre pero que no tiene cuerpo, se dejo la implementacion a las clases hijas. Este método se llama en cada instancia cuando se crea. De esta forma solamente se utiliza esta configuración, en caso de que se provea una configuración externa, no se tomara en cuenta porque se sobrescribirá por esta.

Veamos algunos **pros**:

- Sabemos el lugar exacto en donde esta la configuración lo cual nos sera de mucha ayuda en **minimizar el tiempo** en la **detección de errores** si existe en la conexión con la bd.
- Estamos cumpliendo con **experto** de **GRASP**

Veamos algunos **cons**:

- Es **rígido**
- El **connection string esta hardcoded**, lo cual no permite que se **cambie** en **tiempo de ejecución**, si no que tiene que ser **antes de compilar**.
- El **connection string** hardcoded implica **conflictos** entre el **equipo** ya que en cada pull tienen que cambiarlo para que se adecue a la maquina de cada uno.
- Si se quiere usar otro **proveedor** de base de datos no se puede porque se esta indicando uno concreto.

Veamos de mejorar un poco esos cons:

```
namespace DataAccess.Context
{
    public class VidlyContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }

        public VidlyContext() { }
        public VidlyContext(DbContextOptions options) : base(options) { }
```

```

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if(!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer("Server=ENVY15\\DANIELACEVEDO;
                                    Database=VidlyDB; Integrated Security=True;
                                    Trusted_Connection=True;
                                    MultipleActiveResultSets=True");
    }
}
}
}

```

Ahora lo que logramos con ese **if** es si se configura otro proveedor de base de datos no se sobrescribiera por el que se este indicando. Entonces utilizaríamos el **hardcode** para las migraciones (si estas están siendo creadas desde **DataAccess** o desde otra capa que no configure el proveedor de base de datos).

En pocas palabras se va a utilizar el proveedor de base de datos **hardcode** siempre y cuando no se este configurando por afuera.

Mejoremoslo un poco mas de la siguiente manera:

```

namespace DataAccess.Context
{
    public class VidlyContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }

        public VidlyContext() { }
        public VidlyContext(DbContextOptions options) : base(options) { }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            if(!optionsBuilder.IsConfigured)
            {
                string directory = Directory.GetCurrentDirectory();

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

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

                optionsBuilder.UseSqlServer(connectionString);
            }
        }
    }
}

```

Aca mejoramos la parte de no **hardcode** el **connection string** para evitar la problemática de tener que cambiar el connection string cada vez que se hace un pull.

Podemos ver el código resaltado en amarillo la forma de leer un archivo **json** con nombre **appsettings.json**, donde se encuentra una sección llamada **VidlyDB**, el **connection string**.

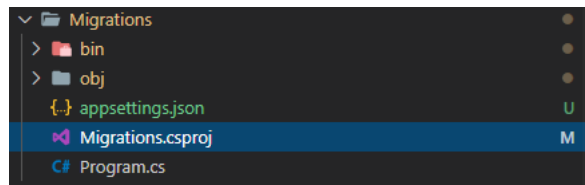
De igual manera se esta **hardcodeando** el proveedor de base de datos en caso de que no se provea uno.

Podemos encontrar un archivo **appsettings.json** en la capa de **WebApi** pero esto no quiere decir que se va a utilizar este, se tiene que **copiar** dicho archivo y **pegar** en **Migrations** y agregar esta **nueva sección** para que **funcione**. De lo contrario se lanzara una excepción. En definitiva se va a utilizar el archivo **appsettings.json** del **startup project**, cuando ustedes esten probando el sistema su **startup project** es **WebApi** por lo tanto este debería de tener un **connection string** tambien para que pueda impactar a la bd. Ahora como nuestro startup project es **Migrations** necesitamos que **appsettings.json** este situado ahi.

Para que la lectura de dicho archivo funcione se tiene que agregar el paquete:

**Microsoft.Extensions.Configuration.Json.**

Entonces nuestra capa **Migrations** debería de estar asi:



Y nuestro **appsettings.json** así:

```
{
  "ConnectionStrings": {
    "VidlyDB": "Server=ENVY15\\DANIELACEVEDO; Database=VidlyDB; Integrated Security=True; Trusted_Connection=True; MultipleActiveResultSets=True"
  }
}
```

Como solo nos interesa una sección, podemos eliminar el resto de secciones si se copio el **appsettings.json** de **WebApi**.

Como se puede apreciar se esta usando el método **DbContextOptionsBuilder.UseSqlServer(string)**, para poder utilizarlo se tiene que instalar el paquete: **Microsoft.EntityFrameworkCore.SqlServer**.

En pocas palabras, donde este la configuración del proveedor de base de datos es donde se tiene que agregar el paquete correspondiente a esa capa. En este caso se esta configurando el proveedor de base de datos **SQL** en **DataAccess** entonces se instala **Microsoft.EntityFrameworkCore.SqlServer**.

## Creando la migración

Ahora que tenemos el **contexto** configurado, pasemos a **crear** la **primera** migración. Las migraciones se pueden correr desde **DataAccess**, **WebApi** u otra **capa**.

La capa **WebApi** es la encargada de procesar las **request**, esa es su responsabilidad, correr las migraciones desde aca es asignarle otra responsabilidad la cual tiene que atarse a la tecnología **EF Core** y estaríamos **rompiendo** con **SRP**. La **WebApi** no tiene porque conocer la **forma** ni la **tecnología** que se esta usando para persistir los datos. Dicho esto podríamos **descartar WebApi**.

**DataAccess** parece el lugar correcto para realizarlo, pero como **DataAccess** es del framework **netstandard2.0**, la cual es de **clases** y no un proyecto **startup**, no se podrían realizar las migraciones en un principio ahí. Para hacerlo se tendría que **cambiar** el **framework** el cual se encuentra en el archivo **DataAccess.csproj**, de **netstandard2.0** a **netcoreapp3.1** y ahí se podrían ejecutar las migraciones situados en ese directorio.

Realicemos una **capa** específica para las **migraciones**. Para esto vamos a crear un proyecto **startup** de consola llamado **Migrations**. Para realizar esto se debe de correr el siguiente comando:

```
dotnet new console -n Migrations
```

Una vez que se creo el proyecto se van a tener que agregar los siguientes paquetes:

- **Microsoft.EntityFrameworkCore.Design**

Estos paquetes tienen que **situarse** desde donde se van a **correr** las **migraciones**. En el **startup project**.

Ahora hay que referenciar **DataAccess** en **Migrations**, de la siguiente manera parados en la raíz del proyecto:

```
dotnet add Migrations reference DataAccess
```

Ahora que tenemos todo listo para crear las migraciones pasemos a crear la primera. Para esto hay que **ubicarnos** en la carpeta **Migrations**, el cual es el directorio de nuestro **proyecto startup** y correr el siguiente comando:

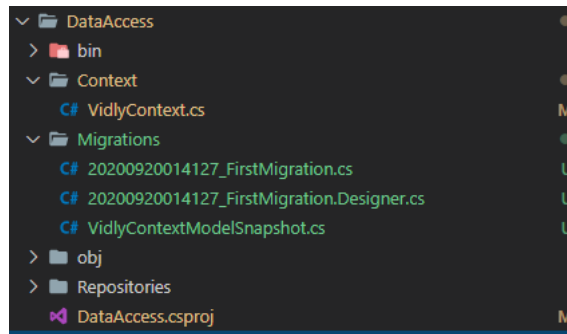
```
dotnet ef migrations add FirstMigration -p ../DataAccess
```

Este comando esta usando las herramientas del paquete que instalamos globalmente (**Microsoft.EntityFrameworkCore.Tools**) porque se esta usando el prefijo **ef**, también podemos ver que se esta especificando el **target project** el cual es **DataAccess** que es en donde se encuentra la **configuración** del **context** y no se especifica el **startup project** porque por defecto es en donde se esta ejecutando la **migración**.

Una vez ejecutado el comando deberíamos de ver una **salida** como la siguiente:

```
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
```

Una vez que tenemos esta salida, automáticamente se debería de crear una carpeta con el nombre **Migrations** con **tres** archivos en el **target project**, en nuestro caso es **DataAccess**. Veamos estas salidas:



Veamos estas tres clases:

### FirstMigration.cs

Este es el archivo principal de los 3. Contiene las operaciones necesarias para **aplicar** la migración (método **Up**) o para **revertir** la migración (método **Down**).

### FirstMigration.Designer

Es la **metadata** de la **migración**. Contiene información utilizada por **EF Core**.

### VidlyContextModelSnapshot

Es una **captura** del modelo actual. Se utiliza para **determinar** que se **cambio** cuando se agrega una nueva **migración**.

Para que la **bd** se pueda ir actualizando de forma **incremental** es de suma **importancia** que se mantenga el **histórico** de las migraciones para que se pueda identificar los cambios y realizar las operaciones correspondientes. De esta forma podemos lograr que no se pierdan los datos ya cargados en la base de datos. Gracias a las migraciones podemos evitar la eliminación completa de una base de datos en el momento de un cambio, la cual se optaba como una solución ante situaciones de cambios de modelos.

El **contexto** nos provee una **configuración** particular para **definir valores** por **defectos**. Esto nos es útil para cuando **existen** datos en la **bd** y se **agrega** una nueva **columna** que **no** acepta **null**, con esta configuración **evitaríamos problemas** en la **aplicación** de la **migración**.

## Creando/modificando la base de datos a partir de la migración

Ahora veamos como aplicar la migración que se creo, para ello se tiene que correr el siguiente comando:

```
dotnet ef database update -p ../DataAccess
```

Una vez aplicado el comando deberíamos de ver la siguiente salida:

```
Build started...
Build succeeded.
Applying migration '20200920014127_FirstMigration'.
Done.
```

Una vez que se aplico la migración podemos ir al **Management Studio**, actualizar las bases de datos y podríamos ver nuestra base de datos.

Como pueden ver la salida de la **creación** de la migración y la **aplicación** de la misma dice **Build succeeded**, esto verifica lo que se mencionaba anteriormente que se **diseña** en tiempo de **compilación**. En caso de que existe un **error** en tiempo de **compilación** se tira un error **Build Failed**. Haciendo una **capa específica** para las **migraciones** es na forma de **evitar** este **error** porque esta aislado de lo que es el sistema.

## Mejora en Factory

Como ahora la configuración del proveedor de base de datos se maneja dentro del contexto. Podríamos eliminar la configuración que realiza Factory lo que llevaría a eliminar la dependencia al paquete **Microsoft.EntityFrameworkCore.SqlServer**. Nos quedaría de la siguiente manera:

```
namespace Factory
{
    public class ServiceFactory
    {
        private readonly IServiceCollection services;

        public ServiceFactory(IServiceCollection services)
        {
            this.services = services;
        }

        public void AddCustomServices()
        {
            services.AddScoped<IMovieRepository, MovieRepository>();
            services.AddScoped<IMovieLogic, MovieLogic>();
        }

        public void AddDbContextService()
        {
            services.AddDbContext<DbContext, VidlyContext>();
        }
    }
}
```

Los cambios se pueden visualizar en el código con fondo amarillo.

Estos cambios impactarían en **WebApi.Startup(IConfiguration)**. Veamos como queda **WebApi.Startup(IConfiguration)**:

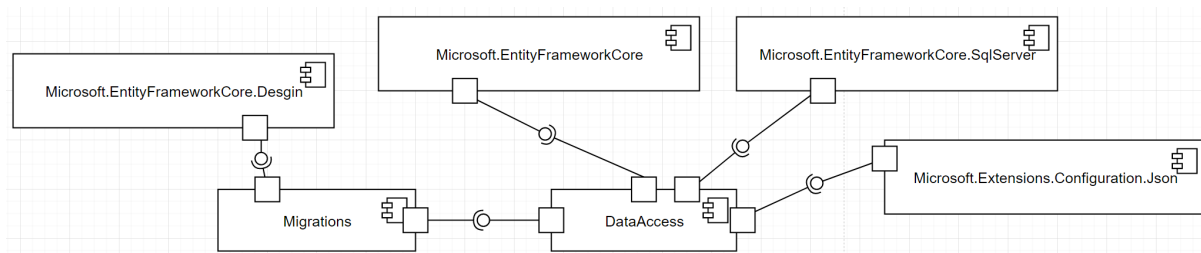
```
namespace WebApi
{
    public class Startup
    {
        //...STARTUP CODE
        public void ConfigureServices(IServiceCollection services)
        {
            //...CONFIGURE_SERVICES CODE

            ServiceFactory factory = new ServiceFactory(services);
            factory.AddCustomServices();
            factory.AddDbContextService();
        }
    }
}
```

## En resumen

Al final de la guía nos debería de quedar los siguientes componentes con sus respectivas dependencias:

### Vista de componentes y conectores



## DataAccess.csproj

A continuación se muestra los paquetes y referencias de **DataAccess** para que puedan comparar con el suyo:

```

<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.8" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="3.1.8" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="3.1.8" />
  </ItemGroup>

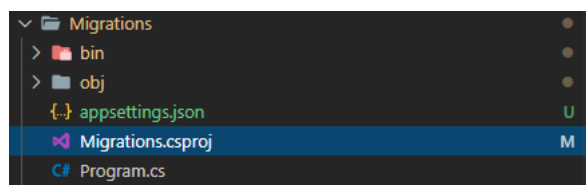
  <ItemGroup>
    <ProjectReference Include="..\Domain\Domain.csproj" />
    <ProjectReference Include="..\DataAccessInterface\DataAccessInterface.csproj" />
  </ItemGroup>

  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
</Project>

```

## Migrations

La imagen a continuación es la del explorador de soluciones que muestra **Migrations**. Bastante simple:



El archivo **appsettings.json** lo pueden llamar como quieran, acuérdense de que tiene que ser el mismo nombre que ponen en **VidlyContext**. Este archivo debería de estar incluido en el **git ignore** para que en cada **pull** no cambie mi connection string.

## Migrations.csproj

La imagen a continuación se muestra los paquetes y referencias de **Migrations** para que puedan comparar con el suyo:



```

<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.8">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\DataAccess\DataAccess.csproj" />
  </ItemGroup>

</Project>

```

## Appsettings.json

De modo de ilustración se muestra como sería el **appsettings.json**:

```

{
  "ConnectionStrings": {
    "VidlyDB": "Server=ENVY15\\DANIELACEVEDO; Database=VidlyDB; Integrated Security=True; Trusted_Connection=True; MultipleActiveResultSets=True"
  }
}

```