¿Qué es Entity Framework Core?

Entity Framework Core (EF Core) es un *ORM* moderno y de código abierto para .NET, desarrollado por Microsoft. Su propósito principal es facilitar el trabajo con bases de datos al permitir que los desarrolladores interactúen con ellas utilizando objetos y clases de C#, en lugar de escribir directamente consultas SQL. Su utilización requiere el uso de LINQ.
Ya veremos que al utilizar EF Core, aparece el concepto de "migraciones", que es algo que nos permite mantener sincronizado el modelo de datos con la base de datos.

Entity Framework 6 vs EF Core.

Algo importante a destacar y a tener en cuenta es que, así como existe la distinción entre .NET Framework (versión antigua) y .NET Core (versión moderna), también existen EF 6 (o "clásico") y EF Core (moderno) respectivamente.

EF 6 es la versión clásica y estable de Entity Framework, pensada principalmente para aplicaciones .NET Framework. Aunque aún tiene soporte, está en modo de mantenimiento, mientras que EF Core es la versión moderna y recomendada para nuevos desarrollos en la plataforma .NET moderna (.NET Core).

Podemos exponer y comparar algunas características importantes de cada uno:

EF 6 o "clásico":
****
Nombre**:** Entity Framework (a secas, o EF 6.x).
Compatible con**:** .NET Framework (principalmente).
Lanzado originalmente: en 2008.
Última versión estable (hasta 2025)**:** EF 6.4.x o 6.5.x.
Características**:**
- No multiplataforma: funciona en Windows bajo .NET Framework

EF Core:
****
Nombre: Entity Framework Core.
Compatible con: .NET Core, .NET 5**+** (incluyendo .NET 6, .NET 7, .NET 8...).
Rediseñado desde cero: no es un port directo de EF clásico.
Multiplataforma: Windows, Linux, macOS.
Última versión estable (hasta 2025): EF Core 8.
Características:
- Mejor rendimiento.
- Soporte para bases de datos modernas (SQLite, Cosmos DB, etc.).

- Nuevas características como shadow properties, global query filters, etc.
- Sigue evolucionando activamente.

Instalación.

Una vez que ya tengamos la estructura completa de nuestra aplicación creada, con su separación en capas según dispone el patrón de Clean Architecture, deberemos instalar algunos paquetes Nuget en la capa de Infraestructura para poder utilizar EF Core. Recordemos que esta arquitectura especifica que los paquetes de terceros deben instalarse en esta capa.

Cabe mencionar que EF Core es compatible con muchos motores de bases de datos relacionales populares, pero requiere que se instale el paquete específico del proveedor de base de datos que se vaya a utilizar.

Así que como nosotros vamos a utilizar el motor de DB de MySQL, instalaremos los siguientes paquetes:

- Pomelo.EntityFrameworkCore.MySql
- Microsoft.EntityFrameworkCore.Design

(NOTA: el paquete Pomelo.EntityFrameworkCore.MySql no es el único que permite instalar EF Core con compatibilidad para MySQL, también existen otros, pero este es uno de los más populares y mantenido activamente. (Otra alternativa podría ser MySql.EntityFrameworkCore, pero Pomelo es uno de los preferidos por la comunidad).

Para instalar dichos paquetes podemos optar por dos vías:
- hacerlo mediante el instalador de paquetes que ofrece la interfaz gráfica del IDE Visual Studio.
- hacerlo mediante la consola utilizando la CLI de .NET.

Recomiendo que para comenzar opten por utilizar el instalador de paquetes que ofrece la interfaz gráfica de VS que será un poco más amigable. Además es más intuitivo para seleccionar el directorio de destino y la versión del paquete que deseamos instalar. Es muy importante que si estamos utilizando .NET 8 instalemos paquetes nuguet que sean compatibles con esta versión. Nuestro proyecto no compilará si instalamos paquetes nuget con número de versión 9 o superior.

Instalación mediante la CLI de .NET.
****

Abre una consola del sistema en la capa de Infraestructura de tu aplicación o ubícate allí mediante el comando “cd [ruta]/Infrastructure”. (en esta carpeta se encuentra el archivo Infrastructure.csproj).

Una vez allí ejecuta los siguientes comandos:

dotnet add package Pomelo.EntityFrameworkCore.MySql
dotnet add package Microsoft.EntityFrameworkCore.Design

(NOTA: estas instrucciones instalarán la última versión disponible de cada paquete, que actualmente es la 9. Y como nosotros estamos usando .NET 8 podemos tener problemas de compatibilidad. Si bien se puede especificar la versión a instalar para que sea compatible con nuestro framework, recomiendo realizar la instalación por la otra vía anteriormente mencionada.)

Estos paquetes que instalaste se pueden visualizar desde Visual Studio en el desplegable de dependencias del proyecto de Infrastructure en el Explorador de Soluciones.

Instalación mediante El Administrador de Paquetes de VS
****

Desde la interfaz de VS debes acceder a:
Herramientas => Administrador de Paquetes Nuget => Administrar Paquetes Nuget para la solución…

Una vez allí, podrás ver las opciones de Examinar / Instalado / Actualizaciones

Desde Examinar puedes buscar nuevos paquetes para instalar desde la web.

En Instalado podrás consultar los paquetes actualmente instalados.

Y desde Actualizaciones puedes saber si hay alguna actualización disponible para los paquetes que ya tengas instalados en tu aplicación.

Bien, selecciona Exminar y buscar los paquetes anteriormente mencionados:

Pomelo.EntityFrameworkCore.MySql
Microsoft.EntityFrameworkCore.Design

Instala uno y luego sigue con el otro.

Es muy importante que luego de seleccionar un paquete, antes de instalarlo selecciones el proyecto de destino donde lo vas instalar (Infraestructura), y asegúrate de seleccionar alguna versión compatible con .NET 8 (cualquier que comience con número de versión 8). Luego de hacer esto dale clic a instalar y sigue los pasos.

Una vez que hayas instalado ambos paquetes verifica en el archivo de Infrastructure.csproj o en su desplegable de Dependencias que ambos paquetes aparezcan en su listado de dependencias.

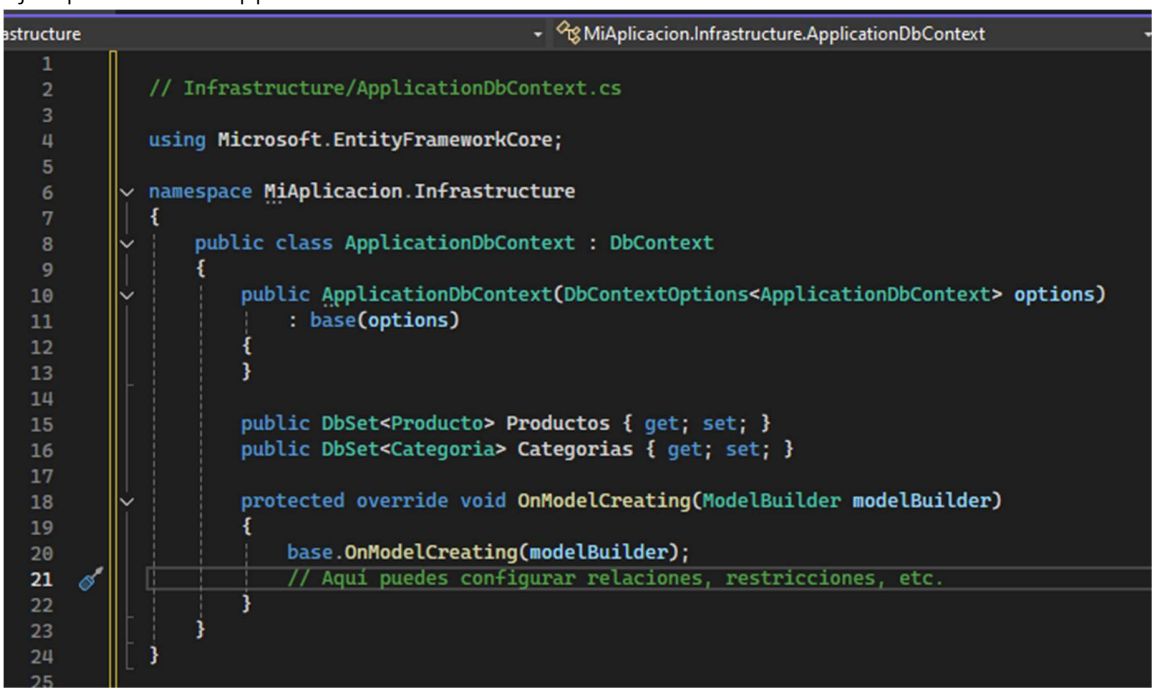
Configuración de EF Core.

Una vez instalados todos los paquetes necesarios, el siguiente paso es configurar EF Core en nuestra aplicación.

Para la configuración básica debemos hacer las siguientes cosas:
- Crear la clase ApplicationDbContext en la capa de Infraestructura;
- Especificar las propiedades DbSet<T> dentro de la clase ApplicationDbContext;
- Armar y guardar el ConnectionString a la base de datos en el archivo appSettings.Development.json;
- Registrar la clase ApplicationDbContext en el contenedor de servicios de la clase Program.cs;

Clase ApplicationDbContext.
****
Es una clase que hereda de DbContext, y normalmente la define el desarrollador para especificar en ella qué entidades quiere mapear a la base de datos y cómo.
Podría decirse que es la clase principal que se utiliza para interactuar con la base de datos.
Representa una sesión con la base de datos, lo que permite consultar y guardar datos.
Debe agregarse al contenedor de servicios de la clase Program.cs y luego podremos inyectarla por constructor en los repositorios para poder interactuar con la DB.

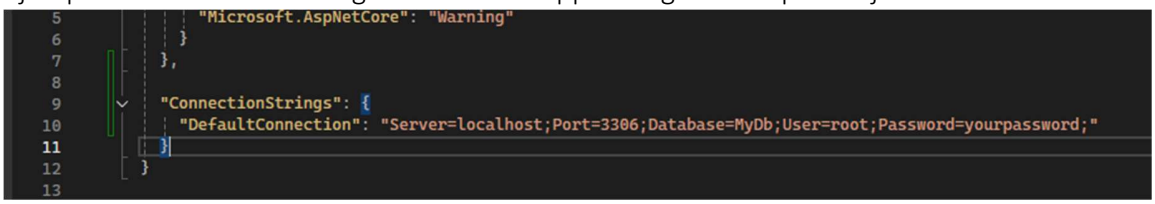
Ejemplo de clase ApplicationDbContext:



Propiedades DbSet<T>
****
En una propiedad que se declara dentro de la clase que herede de DbContext, y representa una tabla de una entidad de nuestro dominio en la base de datos. (Por ejemplo DbSet<User> representará la tabla de usuarios, y le indica al DbContext que mapee una entidad User a dicha tabla)

ConnectionString
****
Es una cadena que contiene todos los datos necesarios para definir la conexión a la base de datos. Dichos datos van uno detrás del otro en formato clave/valor separados por “;”. El formato de dicha cadena puede variar según el proveedor de base de datos que se utilice. Luego, en paso siguiente veremos cómo accedemos a esta cadena desde la clase Program.cs para poder agregar la clase ApplicationDbContext al contenedor de servicios y poder configurar la conexión a la base de datos.

Ejemplo de ConnectionString en el archivo appSettings.Development.json:

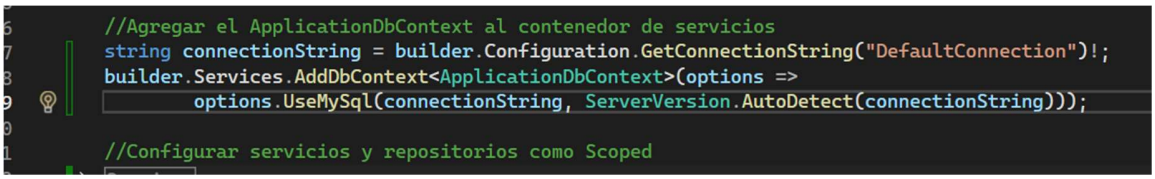


Registrar ApplicationDbContext en Program.cs
****

Registrar el ApplicationDbContext en el contenedor de servicios dentro de Program.cs es necesario porque ASP.NET Core usa la inyección de dependencias (DI) como su mecanismo principal para crear y administrar objetos durante la ejecución de la aplicación, incluyendo el DbContext.
Entonces, cuando una clase (normalmente un repositorio) necesite interactuar con la base de datos, podremos inyectarle por constructor el ApplicationDbContext, y si hicimos previamente dicha configuración en Program.cs, ASP.NET Core sabrá cómo crear una instancia de ApplicationDbContext para poder inyectarla donde sea necesario.

De esta manera le estaremos indicando al framework:
- el tipo de DbContext que deseamos usar (ApplicationDbContext en nuestro caso);
- qué cadena de conexión debe usar para conectarse a la base de datos;
- con qué ciclo de vida debe instanciar dicha clase (por defecto scoped, una instancia por solicitud HTTP)

Ejemplo:



Migraciones.

Una migración es una manera de registrar y aplicar los cambios en el modelo de datos (clases C#) a la base de datos. O sea que nos permite sincronizar la base de datos con nuestro modelo de clases, manteniendo un historial de los cambios estructurales (como creación de tablas, modificación de columnas, claves foráneas, etc.).

¿Cómo podemos realizar nuestra primera migración? Bueno, primero tenemos que crearla y luego aplicarla. (NOTA: antes de intentar realizar una migración, asegúrate de tener iniciado y corriendo el servicio de MySQL en tu ordenador)

Para crear una migración, puedes optar entre hacerlo desde la CLI de .NET, desde la (PMC) Consola del Administrador de Paquetes de VS, o también desde el código.
Nosotros veremos las opciones de la CLI y la de la PMC de VS.

Versión desde la CLI de .NET
****

Para crearla podemos hacerlo con el siguiente comando de CLI de .NET:

```
dotnet ef migrations add NombreDeLaMigracion
```

Y luego, para aplicarla lo hacemos con el siguiente comando:

```
dotnet ef database update
```

(Nota: para poder ejecutar estos comandos es necesario tener instalado el paquete de herramientas dotnet-ef. Puedes verificar si lo tienes instalado ejecutando el siguiente comando en la consola del sistema:

```
dotnet ef –version
```

Si lo tienes instalado esto lo detectará y mostrará la versión que tienes instalada. Si no lo tienes instalado puedes instalarlo con el siguiente comando:

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

Y si por casualidad tienes instalada una versión antigua de esta herramienta, lo que podría ocasionarte problemas al intentar ejecutar una migración, puedes actualizar dicha herramienta con el comando:

```
dotnet tool update --global dotnet-ef
```

Con esto actualizarás dotnet-ef a la versión más reciente compatible con el SDK de .NET que tienes instalado.)

Si bien estos comandos de migraciones para la CLI que mencioné anteriormente son válidos, a continuación, veremos unas variantes de los mismos pero más avanzadas (se les especifican algunos parámetros). Recomiendo utilizarlas:

Este para crear la migración:

```
dotnet ef migrations add InitialMigration --context ApplicationDbContext --
startup-project src/Presentation --project src/Infrastructure -o Data/Migrations
```

Y este para aplicarla:

```
dotnet ef database update --context ApplicationDbContext --startup-project
src/Presentation --project src/Infrastructure
```

Versión desde la PMC de VS

****

Puedes acceder a la PMC de VS si desde la interfaz principal vas:

- Herramientas -> Administrador de Paquetes Nuget -> Consola del administrador de paquetes

Y una vez que abras dicha consola puedes ejecutar el siguiente comando para crear una migración:

Add-Migration NombreDeTuMigración

Y con el siguiente comando aplicas una migración que creaste:

Update-Database

Como verás ambas versiones de creación de migraciones (CLI o PMC) son bastante similares.

Inyectar y usar el DbContext (ApplicationDbContext)

Una vez que hayas realizado todos los pasos anteriores y hayas conseguido crear y aplicar tu migración inicial para crear la DB… vas a necesitar inyectar la clase ApplicationDbContext el algún repositorio o donde lo requiera tu aplicación para poder conectarte a la base de datos y consultar, agregar, modificar o eliminar registros.

Y la manera de hacer esto es simplemente inyectar dicha clase por constructor como hemos hecho hasta ahora con algunas otras clases como servicios o repositorios.

A continuación muestro un ejemplo:

```
namespace Infrastructure;

public class ProductRepository : IProductRepository
{
    private readonly ApplicationDbContext _context;

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

    public Product Create(Product newProduct)
    {
        _context.Products.Add(newProduct);
        _context.SaveChanges();

        return newProduct;
    }
```

Observa en el ejemplo, que luego de inyectar dicha clase podemos acceder por notación de punto a sus propiedades, que justamente serán las tablas de nuestra base de datos. Y con esta metodología podemos utilizar LINQ como hemos aprendido, para realizar consultar a la base de datos sin la necesidad de escribir consultas en lenguaje SQL.

Observa también que, en el ejemplo luego de utilizar el método Add() para que EF Core agregue dicha entidad y la mapee en un nuevo registro de su tabla en la DB, luego se llama al método SaveChanges().
Este método es muy importante para confirmar los cambios en la base de datos luego de realizar alguna operación que inserte, modifique o elimine algún registro. Cuando realizas una simple operación de consulta como en un GetById() o en un GetAll() obviamente no es necesario llamar a SaveChanges().

Por último, veremos algunos ejemplos de cómo interactuar con la clase ApplicationDbContext y realizar las operaciones del CRUD básico en una tabla de la DB:

- Consultar todos:
    _context.TableName.ToList()

- Consultar un registro por id:
    _context.TableName.FirstOrDefault(x => x.Id == x);

- Crear nuevo registro:

```
_context.TableName.Add(newEntity);
```

- Modificar un registro:

```
_context.TableName.Update(updatedEntity);
```

- Eliminar un registro:

```
_context.TableName.Remove(entityToRemove);
```

Y eso sería todo. Recuerda agregar _context.SaveChanges() en los casos que corresponda.