



Entity Framework Core

Algunas de las cosas que vamos a ver en esta seccion:




- Que es EF Core
- Creacion del contexto
- Configuracion de fluent API
- Migraciones
- Loading related data
- LINQ

Que es Entity Framework Core?

Entity Framework Core es una version mas ligera, extensible, open-source y cross-platform de la tecnología acceso a datos: **Entity Framework**. **EF Core** es un **ORM** (Object-Relational Mapper), lo que le permite a los desarrolladores de **.NET** trabajar con base de datos utilizando objetos **.NET** y eliminando la necesidad de crear código de acceso a datos.

Algunas de las base de datos que soporta son:

Base de datos

 Paquete Nuget	 Motor de base de datos	 Descripción
<u>Microsoft.EntityFrameworkCore.SqlServer</u>	SQL Server 2012 en adelante	Este proveedor permite usar EF Core con una base de datos en Microsoft SQL Server
<u>Microsoft.EntityFrameworkCore.InMemory</u>	EF Core in-memory database	Este proveedor permite usar EF Core con una base de datos en memoria. Esto puede ser muy útil para testing.
<u>Pomelo.EntityFrameworkCore.MySql</u>	MySQL, MariaDB	Este proveedor permite usar EF Core con una base de datos MySQL



Si quieren saber que otros proveedores de base de datos soporta EF Core, lean el siguiente artículo: [Database Providers](#)

Conexión a la base de datos

Para conectarnos a la base de datos, necesitaremos una clase contexto y entidades del negocio. La clase contexto es la representación de la sesión con la base de datos, lo que nos permitirá realizar queries y guardar data.

La convención de trabajo que se utiliza es **code-first**, lo que nos indica que primero tenemos que diseñar el negocio con clases para que luego **EF Core** lo pueda diseñar en un modelo de **tablas** para crear la base de datos y actualizarla. Se puede especificar configuraciones adicionales para complementar o sobrescribir lo que la tecnología descubrió a partir de la diagramación de las entidades.

Creación del contexto

Veamos que configuraciones podemos realizar y como realizarlas para tener un buen diseño de tablas en la base de datos.

Para crear la clase **contexto**, esta debería de estar en un proyecto de **acceso a datos**, de esta forma creamos una capa específica para esto y podemos cumplir con **SRP**.

Una clase contexto se vería así:

```
using Microsoft.EntityFrameworkCore;

namespace DataAccess
{
    public class MyContext : DbContext
    {
    }
}
```

Para que esto funcione hay que instalar el paquete **Microsoft.EntityFrameworkCore**, para esto se debería de ejecutar el siguiente comando:

```
dotnet add package Microsoft.EntityFrameworkCore
```

Configuración Fluent API

Crear tablas en BD

Nuestro sistema debería de tener unas entidades del negocio las cuales son las que queremos que se persistan en la base de datos mediante tablas. Para realizar esto hay que cambiar el contexto de la siguiente manera:

```
using Microsoft.EntityFrameworkCore;

namespace DataAccess.Context
{
    public class MyContext : DbContext
    {
        public DbSet<MyEntityType> Entities { get; set; }
    }
}
```



Donde **MyEntityType** sería el tipo de la **entidad** que queremos que se persista y **Entities** sería el nombre de la **tabla** en la base de datos.

Por convención, los tipos que son expuestos con DbSet son los que se incluyen en el modelo de tablas. Pero hay otras formas de declarar tipos de entidades, veamos cuales son con el siguiente código concreto de **Blogs** y **Posts**:

```
public class MyContext : DbContext
{
    public DbSet<Blog> Blogs{ get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Log>();
    }
}
```

```
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Post> Posts { get; set; }
}
```

```
public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }

    public Blog Blog { get; set; }
}
```

```
public class Log
{
    public int Id { get; set; }
    public string Message { get; set; }
}
```

Podemos ver que se van a percistir esas tres clases, de la siguiente manera:

- **Blog** se perciste porque utiliza la property DbSet como se habia dicho anteriormente
- **Post** se perciste por la navegacion **Blog.Posts**. Es un efecto en cadena
- **Log** se perciste porque se especifica en el metodo **MyContext.OnModelCreating(ModelBuilder)**

Estas son las tres formas de percistir una entidad en una tabla. El metodo **MyContext.OnModelCreating(ModelBuilder modelBuilder)** es un metodo

poderoso en sentido que es el que tiene mayor preferencia en cuestiones de configuracion, el contexto se puede configurar por fuera pero si esa misma configuracion esta en este metodo se va a tomar la del metodo y no la de afuera. Se le hace un override a la configuracion externa.

Lo recomendable es seguir una misma configuracion, y no estar mezclando las diferentes formas de hacer lo mismo. Como se dijo anteriormente, por convencion para crear las tablas es con la property DbSet.

Ignorar entidades

En el caso de que se quiera ignorar una entidad de que sea persistida como una tabla, hay dos convenciones.

- Con **Data Annotations**:

```
[NotMapped]
public class BlogMetadata
{
    public DateTime LoadedFromDataBase { get; set; }
}
```

Esto permite evitar que exista una tabla para **BlogMetadata**.



Las data annotations es un paquete que provee diferentes attributes que son usados para definir metadata en ASP.NET. Eventualmente podemos usar las de este paquete o crear las nuestras propias para realizar algún comportamiento 🙌. Si quieren saber mas sobre **DataAnnotations** pueden encontrar mas información en: [Model validation in ASP.NET Core MVC Razor Pages](#)

- Con Fluent API (con el metodo **MyContext.OnModelCreating(ModelBuilder modelBuilder)**)

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Ignore<BlogMetadata>();
}
```

Asignar el nombre a la tabla

Por convención se toma el nombre de la property de DbSet. Aca podemos encontrar 3 formas diferentes de asignar el nombre:

- Por el nombre de la property de DbSet
- Utilizando la data annotation **Table** en la clase

```
[Table("blogs")]
public class Blog
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

- Usando fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Blog>()
        .ToTable("blogs");
}
```

Y podemos hacer un **montón** de cosas mas en la **configuración** del contexto. Algunas de las cosas que podemos hacer tanto con **Data Annotation** como con **Fluent API** son:

- Darle diferentes comportamientos a las properties como establecer largo máximo de string, si es campo requerido, cambiar el nombre, establecer la key, etc.
- Configurar valores por defecto
- Configurar las relaciones entre las tablas
- Conversion de valores
- Comparador de valores
- Muchas cosas mas



Si quieren ver información mas detallada sobre la configuración del contexto, la pueden encontrar en: [Creating and configuring a model](#) o [Fluent API Configuration](#)

Migraciones

Para crear la base vamos a tener que crear migraciones y ejecutarlas para que impacten en la base de datos. En aplicaciones del mundo real, la estructura de las entidades cambia a medida que nuevas funcionalidades se van implementando, aparecen nuevas entidades o nuevas propiedades como también pueden ser sacadas y el schema de la base de datos necesita ser cambiado para que se mantenga sincronizado con la aplicación.

Las migraciones en EF Core provee un mecanismo incremental para actualizar el esquema de la base de datos para mantenerla en sincronización con la aplicación mientras que se mantiene data en la base de datos.

A muy alto nivel, las migraciones funcionan de la siguiente manera:

- Cuando un cambio en las entidades del negocio es introducido, el desarrollador utiliza la herramienta EF Core para agregar la migración correspondiente que describe la actualización para mantener el schema de la base de datos en sincronía. EF Core compara el modelo actual contra una captura (snapshot) del modelo viejo para determinar las diferencias, y poder generar los archivos de la migración
- Una vez que está la nueva migración, se puede aplicar contra la base de datos de varias formas diferentes. EF Core guarda todas las migraciones aplicadas en un historico como tabla, permitiendo saber que migración fue aplicada y cual no.

Lecturas de referencia y recomendadas

- [Entity Framework Core](#)
- [Creating and configuring a model](#)
- [Migrations Overview](#)
- [Language Integrated Query \(LINQ\)](#)
- [Fluent API Configuration](#)