
Persistencia de Datos

— Desarrollo Web en Entorno
Servidor —

Contenidos

- Introducción
- Tipos de SGBD a usar
- Entity Framework Core
 - ¿Qué es EF Core?
 - ¿Qué es un ORM?
 - Code First Vs Database First
- Code First
 - Crear un modelo
 - Scaffold de un proyecto (Elementos creados)
 - Migraciones
 - Seeders
- DataBase First
 - Conectar con la DB
 - Generación automática de modelo + scaffold de los modelos
- Modelo híbrido
- Trabajando con tablas relacionadas
 - One to One
 - One to Many
 - Many to Many

Introducción

Hasta ahora hemos trabajado con datos que tan solo han existido en la memoria volátil del ordenador.

Esto provoca que los datos se pierdan cuando se reinicia la aplicación.

Para evitar esto, toda aplicación del lado servidor utiliza BBDD para hacer que la información con la que trabaja sea persistente.

Gracias a esto los datos perdurarán en el tiempo más allá del tiempo que la aplicación esté en ejecución.

Las aplicaciones, una vez conectadas al SGBD, podrán realizar operaciones CRUD de los modelos con los que trabajan.

Entity Framework Core

Entity Framework Core (en adelante EF Core) es un moderno **ORM** (Object-Relation Mapper) para .NET

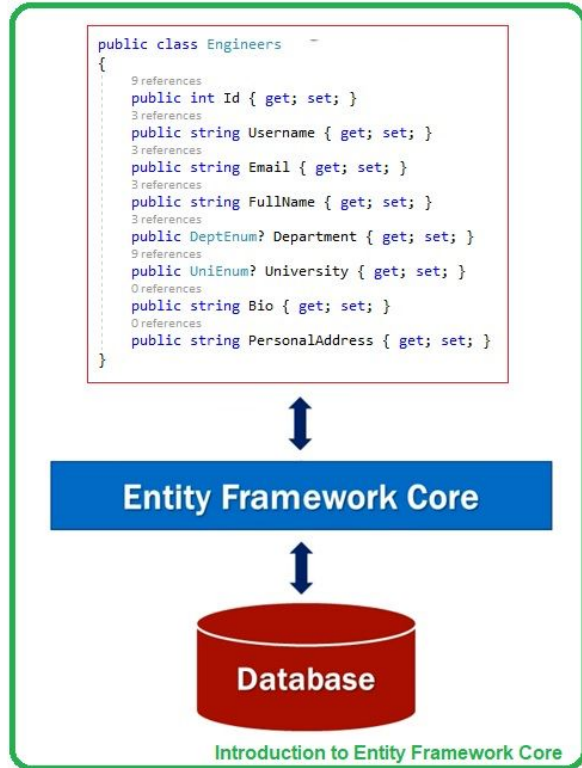
Esta herramienta permite mapear las clases de los modelos con las tablas de la base de datos:

- Permite a los desarrolladores trabajar con una BD usando objetos .NET
- Permite prescindir de la mayor parte del código de acceso a datos que normalmente es necesario escribir.

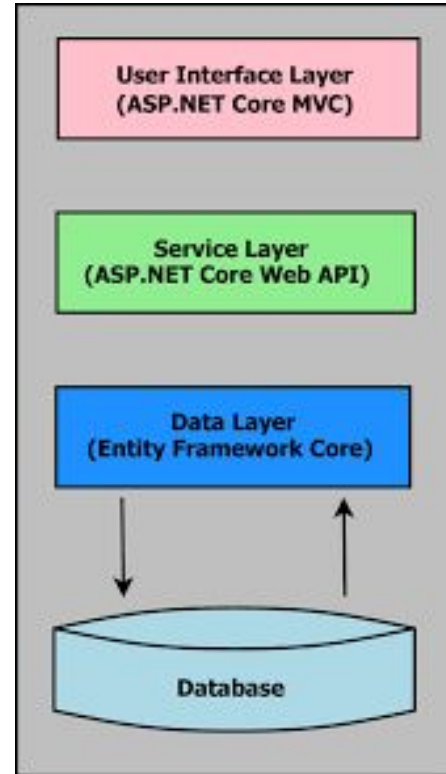
Se compone de un conjunto de librerías que tendrán que ser importadas en el proyecto que necesite utilizarlo.

Es el software que hace de puente entre la aplicación y la BD, facilitando la recuperación y almacenamiento de la información.

Entity Framework Core



[Ref1](#)



[Ref2](#)

Entity Framework Core

EF Core puede tener acceso a muchas bases de datos diferentes a través de bibliotecas de complementos denominadas proveedores de bases de datos (database providers).

Como veremos más adelante, estas bibliotecas se importan en los proyectos .NET mediante **paquetes NuGet**.

Algunas de las bases de datos que se pueden usar con EF Core son:

- SQL Server
- Mysql
- PostgreSQL
- Oracle
- SQLite, [etc.](#)



Es el que usaremos

Entity Framework Core

SQL Server

Es un sistema de gestión de base de datos (SGBD) relacional, desarrollado por Microsoft.

Existen varias ediciones. Algunas son:

- SQL Server en Azure
- SQL Server Enterprise
- SQL Server Developer
- SQL Server Express

Usaremos SQL Server Express LocalDB

Entity Framework Core

SQL Server Express LocalDB

Esta versión tiene las siguientes características:

- Es una versión ligera del motor de base de datos SQL Server Express, instalada de forma predeterminada con Visual Studio
- Se inicia a petición mediante una cadena de conexión
- Está destinado al desarrollo de programas. Se ejecuta en modo usuario, sin necesidad de una configuración compleja
- De forma predeterminada, crea archivos *.mdf* en el directorio *C:/Usuarios/{user}*

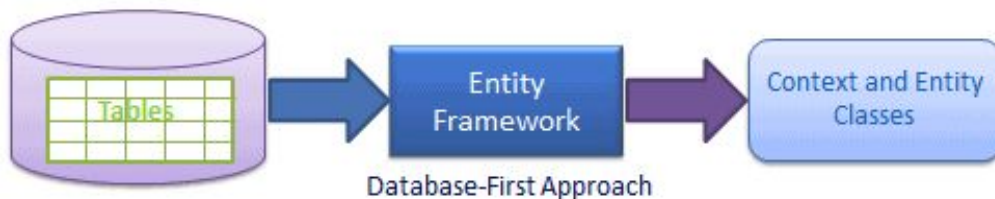
Entity Framework Core

EF Core permite trabajar de dos formas con la BD:

- **Code First:** Se definen las clases de los modelos, con sus relaciones, y se generan automáticamente las tablas en la base de datos



- **Database First:** Partiendo de una base de datos relacional, se utiliza el ORM para generar las clases de los modelos de la aplicación.



Entity Framework Core

¿Code First ó Database First?

Hay que elegir una de las dos en función de algunos aspectos:

- Code First
 - El equipo de desarrollo maneja mejor el código que las bases de datos
 - El proyecto no trabaja sobre una BD previa, sino que la crea desde el inicio
- Database First
 - El equipo de desarrollo se defiende bien con las bases de datos
 - El proyecto se ha de adaptar a una base de datos existente

EF Core - Code First

Partiremos de un nuevo proyecto ASP.NET Core MVC

Varios de los conceptos tratados en este apartado también los usaremos cuando trabajemos de la forma Database First

Para trabajar de esta manera veremos los siguientes aspectos:

- Creación del modelo
- Paquetes NuGet
- Scaffolding de vistas y controlador
- Migraciones
- Seeder

EF Core - Code First

Creación del Modelo

En este tipo de aproximación partiremos de un determinado modelo.

La creación de un modelo es algo que ya se ha tratado anteriormente.

Todos los atributos de validación que puedan tener los campos del modelo se utilizarán para la creación de las columnas de la tabla correspondiente, según lo considere EF Core.

Un ejemplo de Modelo, que usaremos para este ejemplo, podría ser el siguiente:

EF Core - Code First

```
public class Car
{
    0 referencias
    public int Id { get; set; }
    [Required(ErrorMessage = "Campo Obligatorio")]
    [MaxLength(15, ErrorMessage = "El campo no puede tener más de 15 caracteres")]
    [MinLength(3, ErrorMessage = "El campo ha de tener mínimo 3 caracteres")]
    [DisplayName("Modelo")]
    0 referencias
    public string Model { get; set; }
    [Required(ErrorMessage = "Campo Obligatorio")]
    [MaxLength(15, ErrorMessage = "El campo no puede tener más de 15 caracteres")]
    [MinLength(3, ErrorMessage = "El campo ha de tener mínimo 3 caracteres")]
    [DisplayName("Marca")]
    0 referencias
    public string Brand { get; set; }
    [StringLength(7, ErrorMessage = "La matrícula ha de tener 7 caracteres", MinimumLength = 7)]
    [DisplayName("Matrícula")]
    0 referencias
    public string CarCode { get; set; }
    [DataType(DataType.Date)]
    0 referencias
    public DateTime PurchaseDate { get; set; }
    [Required(ErrorMessage = "Campo Obligatorio")]
    [Range(1, 9, ErrorMessage = "El número de asientos ha de estar entre 1 y 9")]
    [DisplayName("Número de Asientos")]
    0 referencias
    public int SeatNum { get; set; }
}
```

EF Core - Code First

Paquetes NuGet

Los paquetes NuGet son librerías formadas por código compilado (archivos DLL) que podemos incluir en un proyecto .NET para ampliar su funcionalidad.

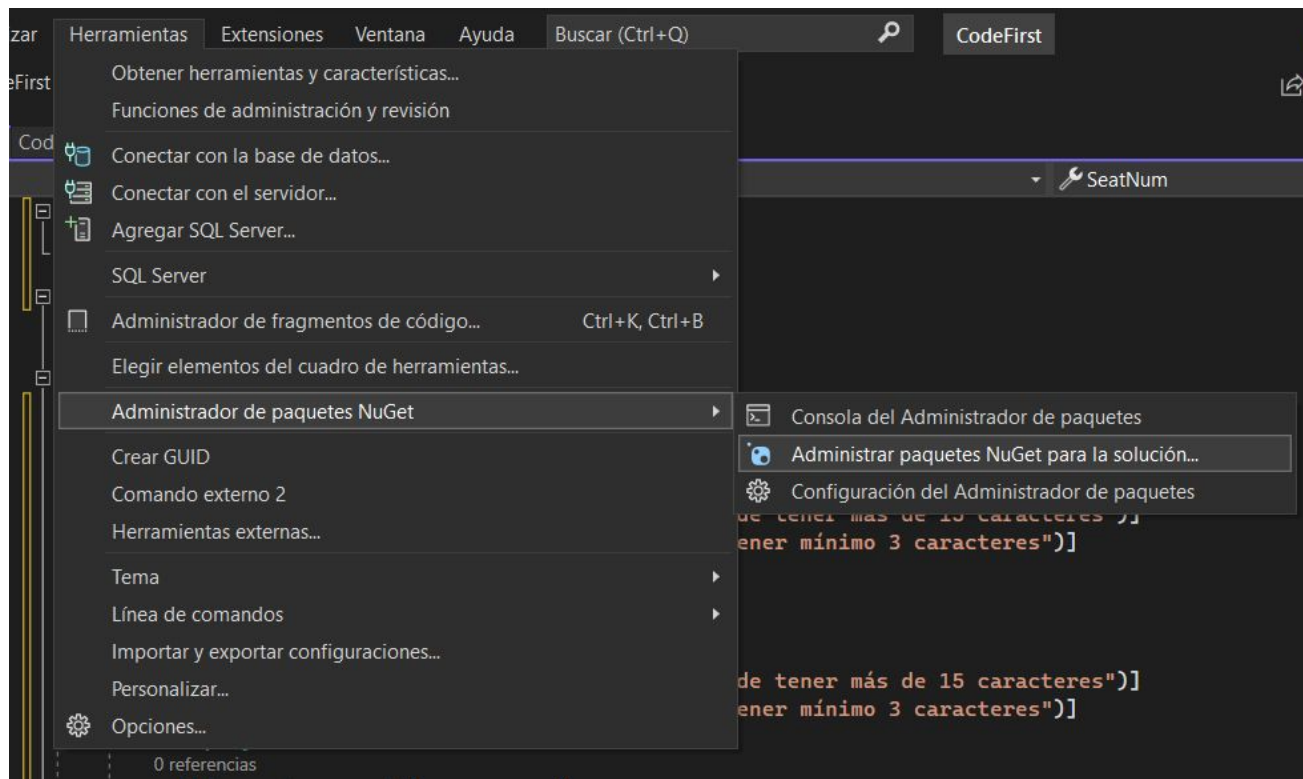
Son unidades reutilizables que los desarrolladores ponen a disposición de la comunidad para que se usen en los proyectos.

Se pueden instalar de dos formas:

- Usando la Consola de Administrador de paquetes
- Mediante el panel de Administrar paquetes NuGet para la solución

Algunas acciones son necesarias realizarlas por consola, como veremos.




EF Core - Code First



EF Core - Code First

Paquetes NuGet

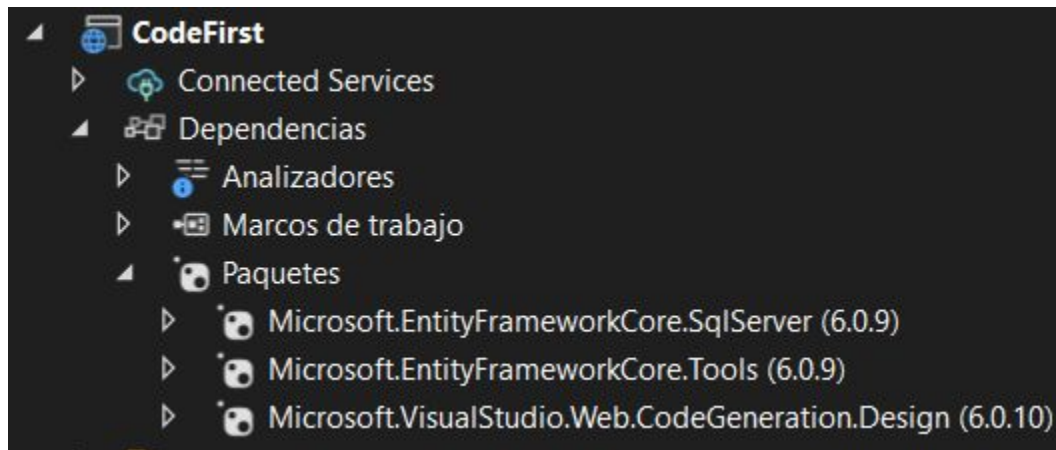
En la pestaña “Examinar” buscaremos e instalaremos los siguientes 3 paquetes:

	Microsoft.EntityFrameworkCore.SqlServer por Microsoft Microsoft SQL Server database provider for Entity Framework Core.	6.0.9
	Microsoft.EntityFrameworkCore.Tools por Microsoft Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.	6.0.9
	Microsoft.VisualStudio.Web.CodeGeneration.Design por Microsoft Code Generation tool for ASP.NET Core. Contains the dotnet-aspnet-codegenerator command used for generating controllers and views.	6.0.10

EF Core - Code First

Paquetes NuGet

Vemos que tras la instalación aparecen estos 3 paquetes como dependencias de nuestro proyecto:



EF Core - Code First

MVC Scaffolding

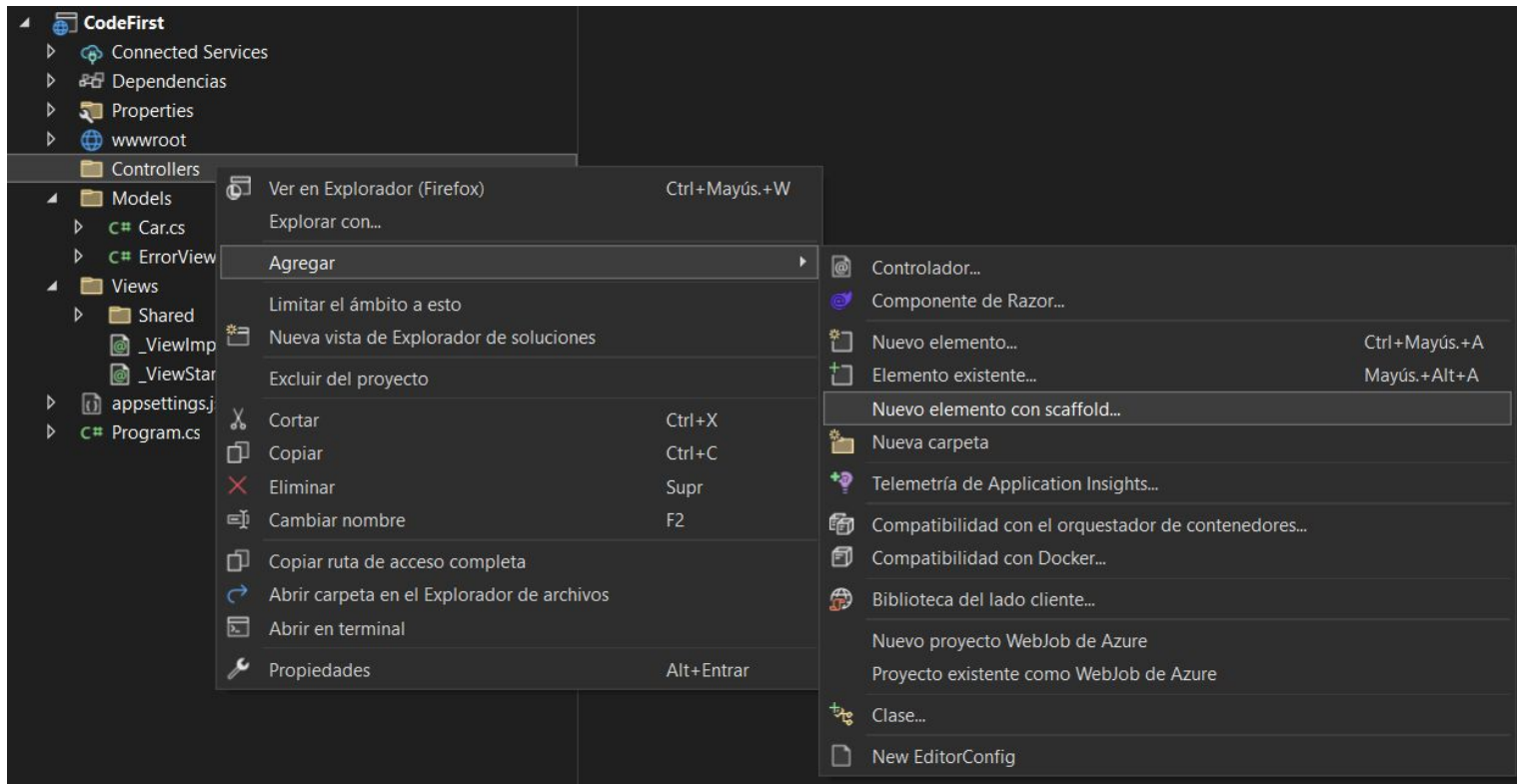
El *scaffolding* (andamiaje en inglés) es una técnica usada en muchos frameworks MVC para generar código básico para operaciones CRUD.

Esto genera el controlador, con sus acciones correspondientes, y las vistas necesarias para poder realizar este tipo de operaciones de persistencia con la base de datos.

Esta acción se realiza en base a un modelo definido y genera una gran cantidad de archivos y código que veremos con detenimiento.

Todo el código generado puede no ser útil, y será el desarrollador quien tenga que realizar las adaptaciones necesarias.

EF Core - Code First



EF Core - Code First



EF Core - Code First

Agregar Controlador de MVC con vistas que usan Entity Framework

Clase de modelo: Car (CodeFirst.Models)

Clase de contexto de datos: CodeFirst.Data.CodeFirstContext

Vistas

- ☒ Generar vistas
- ☒ Hacer referencia a bibliotecas de scripts
- ☒ Usar página de diseño

(Dejar en blanco si se define en un archivo _viewstart de Razor)

Nombre de controlador: CarsController

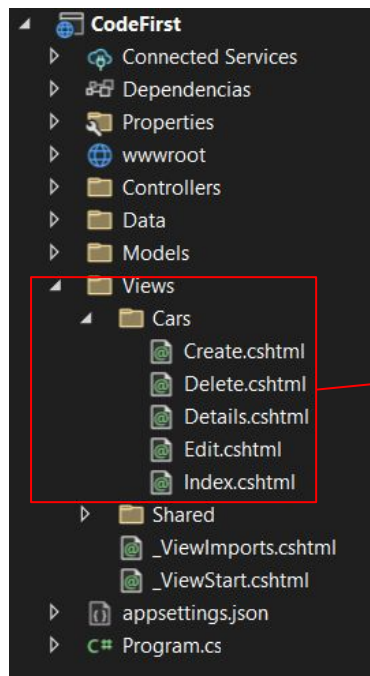
Agregar Cancelar

Modelo sobre el que se va a realizar el Scaffold

Pulsando + nos muestra la clase de contexto de datos por defecto de la aplicación, que es la que habrá que elegir

EF Core - Code First

MVC Scaffolding - Elementos Creados/Actualizados

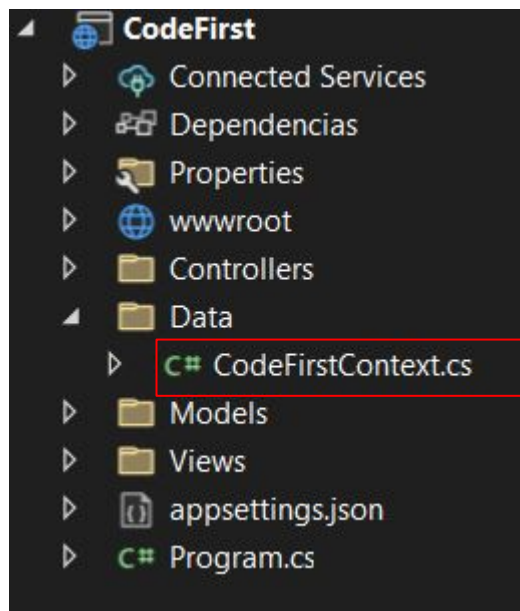


Todas las vistas relacionadas con el modelo que nos van a permitir:

- Listar
- Crear
- Ver detalles
- Editar
- Eliminar

EF Core - Code First

MVC Scaffolding - Elementos Creados/Actualizados



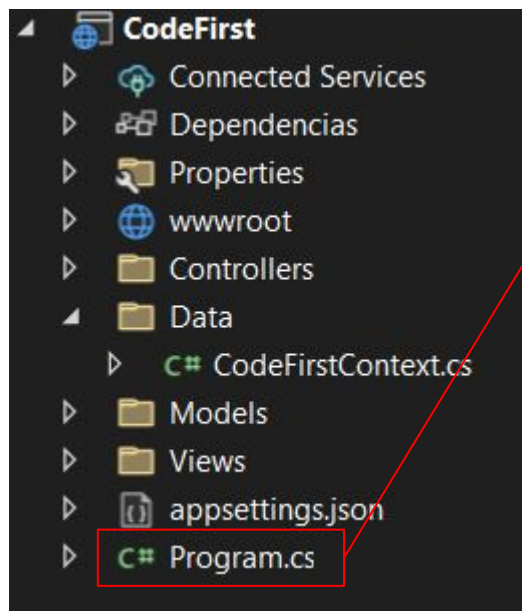
Es la clase de tipo DbContext de la aplicación.
Es la clase primaria responsable de interactuar con la base de datos.

Se encarga de las siguientes acciones sobre la base de datos:

- Consultas
- Seguimiento de cambios
- Operaciones de persistencia CUD
- Almacenamiento en caché
- Gestionar relaciones (CF o DBF)
- Materialización de objetos

EF Core - Code First

MVC Scaffolding - Elementos Creados/Actualizados

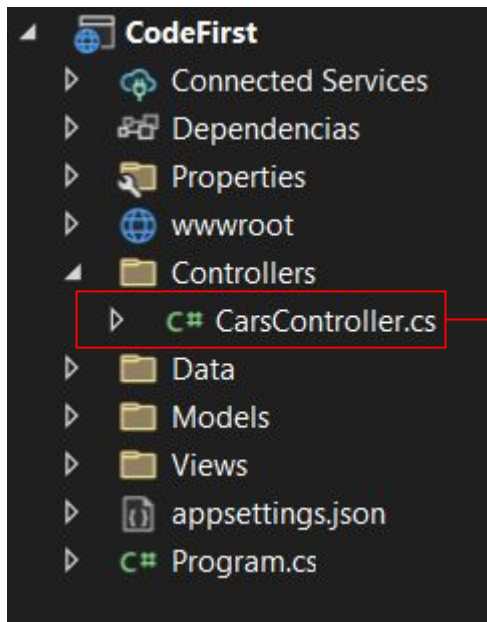


Se actualiza este archivo para registrar el contexto de la base de datos

```
builder.Services.AddDbContext<CodeFirstContext>(options =>  
    options.UseSqlServer(builder.Configuration.GetConnectionString("CodeFirstContext"))
```


EF Core - Code First

MVC Scaffolding - Elementos Creados/Actualizados



Controlador del modelo del que hemos hecho el *scaffold*.

Contiene las siguientes acciones para realizar sobre el modelo

- Listar
- Crear
- Ver detalles
- Editar
- Eliminar

EF Core - Code First

MVC Scaffolding - Elementos Creados/Actualizados

```
public class CarsController : Controller
{
    private readonly CodeFirstContext _context;

    0 referencias
    public CarsController(CodeFirstContext context)
    {
        _context = context;
    }

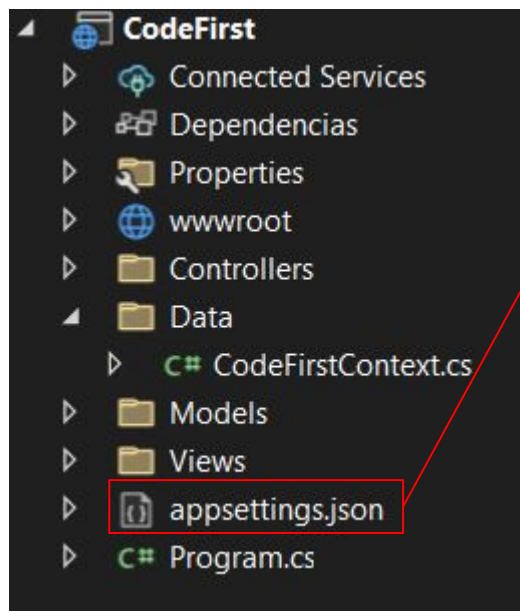
    // GET: Cars
    3 referencias
    public async Task<IActionResult> Index()
    {
        return View(await _context.Car.ToListAsync());
    }
}
```

El controlador declara una referencia al DbContext de la aplicación y la inicializa en el constructor. Se utilizará para realizar los accesos a la Base de Datos

Cuando trabajamos con bases de datos, como las peticiones son asíncronas, todas las acciones del controlador que accedan a esta tendrán que trabajar de forma asíncrona. [Más información.](#) [Y más.](#)

EF Core - Code First

MVC Scaffolding - Elementos Creados/Actualizados



Se agrega la cadena de conexión de la base de datos

```
"ConnectionStrings": {  
  "CodeFirstContext": "Server=(localdb)\\mssqllocaldb;Database=CodeFirst.Data;  
}
```

EF Core - Code First

SQL Express LocalDB

Para consultar el contenido de la base de datos tenemos dos formas:

- Explorador de objetos de SQL Server (Herramienta de Visual Studio)
- Sql Server Management Studio

Ambas son válidas para consultar el contenido de nuestra base de datos local de SQL Express

EF Core - Code First

SQL Express LocalDB - Consultar estado del servicio

```
PS C:\Users\asanhid> sqllocaldb info
MSSQLLocalDB
PS C:\Users\asanhid> sqllocaldb info MSSQLLocalDB
Name:                MSSQLLocalDB
Version:             15.0.4153.1
Shared name:
Owner:               AzureAD\asanhid
Auto-create:         Yes
State:               Running
Last start time:     03/10/2022 17:18:06
Instance pipe name:  np:\\.\pipe\LOCALDB#42D22625\tsql\query
```

Ejercicio

Realizar las siguientes acciones:

- Crear un nuevo proyecto ASP.NET MVC al que añadirás una clase de un modelo con al menos 4 campos, variados en su tipo, más el campo Id.
- Añadir atributos de validación de los campos del modelo.
- Utiliza las técnicas tratadas en las diapositivas anteriores para generar el *scaffold* en base a este modelo creado. Se tendrá que generar el controlador, con sus acciones correspondientes, y las vistas para implementar todas las operaciones CRUD de ese modelo.

TIP: Si creas un repositorio propio en el que vayas haciendo subidas para cada paso de este ejercicio, y los que vengan, tendrás un buen ejemplo dividido en partes de el código que se genera en cada uno de estos pasos y los futuros.

EF Core - Code First

Migraciones

Si consultamos la base de datos en este punto, incluso tras haber compilado y ejecutado la aplicación, veremos que no hay ninguna base de datos ni tablas creadas.

Para ello hemos de crear la migración inicial, que se encargará de crear la base de datos con el nombre del proyecto y una primera tabla en base al modelo creado.

Las migraciones son un conjunto de herramientas que crean y actualizan una base de datos para que coincida con el modelo de datos.

EF Core - Code First

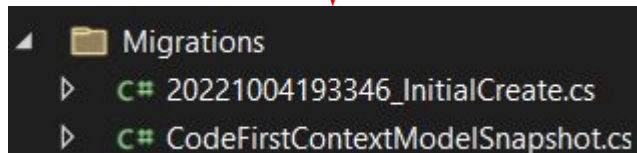
Migraciones

Para realizar la primera migración hay que introducir los siguientes comandos en la Consola de Administrador de paquetes

```
Add-Migration InitialCreate  
Update-Database
```

Este comando actualiza la base de datos con lo que se indique en los archivos de migración

Este comando crea el archivo de la primera migración con el nombre: `<timestamp>_InitialCreate.cs` en la carpeta Migrations. Esta clase tiene la información para crear las tablas que se definan.

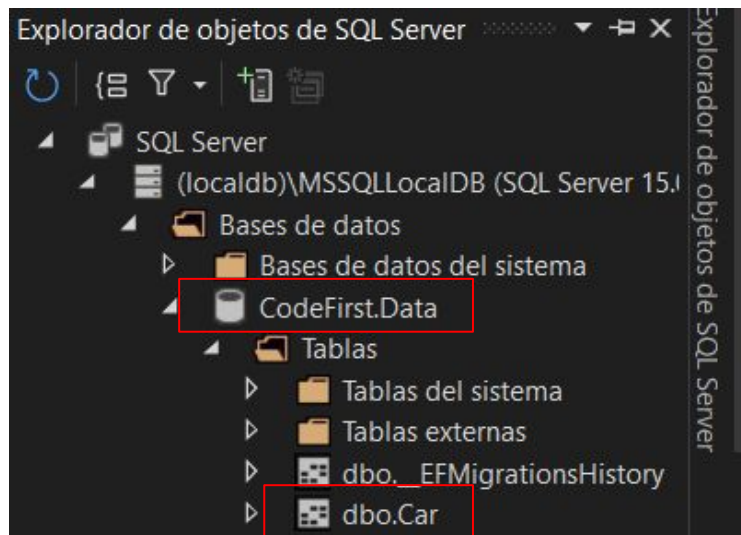


```
Migrations  
└─ 20221004193346_InitialCreate.cs  
└─ CodeFirstContextModelSnapshot.cs
```


EF Core - Code First

Migraciones

Si ahora exploramos SQL Server LocalDB veremos que se ha creado la base de datos y la tabla del modelo.



EF Core - Code First

Migraciones - Excluir Modelos de una Migración

En diversas situaciones, como en el caso de haber generado los modelos mediante técnicas *Database First*, tendremos la necesidad de que si se realizan migraciones de forma posterior, no se trate de generar tablas de estos modelos mencionados, ya que ya tienen una tabla correspondiente en la Base de Datos.

Para que estos modelos no se traten de incluir en migraciones usaremos la siguiente instrucción, dentro del método `OnModelCreating` de la `DbContext`.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Album>().ToTable("Album", t => t.ExcludeFromMigrations());
}
```

EF Core - Code First

Migraciones - Excluir Campos de un Modelo de una Migración

Al igual que en el caso anterior, se presentarán situaciones en las que lo que queramos excluir en una migración sea algún campo en concreto de un Modelo.

Esto nos permitirá utilizar algún campo de un modelo que no tenga su correspondiente campo en una tabla, pero que nos servirá para la gestión interna de la aplicación.

Esto se indicará en “onModelCreating” de la clase de contexto de esta forma:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Album>().Ignore(a => a.Artist);
}
```

EF Core - Code First

Migraciones

En este punto, si modificamos la ruta por defecto de la aplicación y la ejecutamos veremos que se carga un listado vacío de elementos del modelo sobre el que hemos hecho *scaffolding*.

Es posible insertar datos por defecto de un modelo en su tabla correspondiente mediante un nuevo elemento denominado *seeder*.

EF Core - Code First

Seeders

Los *seeders* nos permiten insertar registros en determinadas tablas que sean necesarios para el funcionamiento de la aplicación.

EF Core proporciona una API para insertar registros en la base de datos (*seeding*) como parte de una migración.

Para ello usaremos el método “HasData” mediante el método “modelBuilder.Entity<T>”.

Esto lo realizaremos en el método “OnModelCreating” de nuestra clase de contexto (DbContext)

Veamos cómo se hace con un ejemplo:

EF Core - Code First

Seeders

Veamos cómo se hace con un ejemplo que añade un elemento a la clase *Author*:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>().HasData(
        new Author
        {
            AuthorId = 1,
            FirstName = "William",
            LastName = "Shakespeare"
        }
    );
}
```

EF Core - Code First

Seeders

Para añadir datos en la BD de dos entidades que estén relacionadas:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Author>().HasData(
        new Author
        {
            AuthorId = 1,
            FirstName = "William",
            LastName = "Shakespeare"
        }
    );
    modelBuilder.Entity<Book>().HasData(
        new Book { BookId = 1, AuthorId = 1, Title = "Hamlet" },
        new Book { BookId = 2, AuthorId = 1, Title = "King Lear" },
        new Book { BookId = 3, AuthorId = 1, Title = "Othello" }
    );
}
```

EF Core - Code First

Seeders

Tras definir los registros que queremos añadir a cada entidad, de la forma vista, tendremos que generar una migración en la consola:

- Add-Migration <nombreDeLaMigración>

Una vez realizada la migración, que incluirá los datos a incorporar a la BD, actualizaremos al BD:

- Update-Database

Ejercicio

Partiendo del ejercicio anterior de estas diapositivas, realizar las siguientes acciones:

- Crear una migración que cree una base de datos con el nombre del proyecto, en nuestro SQL LocalDB, y una tabla vacía con los campos de nuestro modelo.
- Crear un *seeder* que incluya 3 registros en la tabla de nuestro modelo al iniciar la aplicación, en caso de que esta estuviera vacía.

EF Core - DataBase First

En este tipo de enfoque partiremos de una base de datos ya existente.

A partir de esta generamos los modelos, gracias al ORM EF Core, y mediante *scaffolding* se crearán los controladores y vistas asociadas al modelo que elijamos.

Trataremos los siguientes aspectos:

- Importar una BD en SQL Server Express LocalDB
- Generar Modelos de las tablas de la BD
- Generar controlador y vistas del modelo
- Añadir Clase de Contexto al Builder de la Aplicación

EF Core - DataBase First

Importar una BD

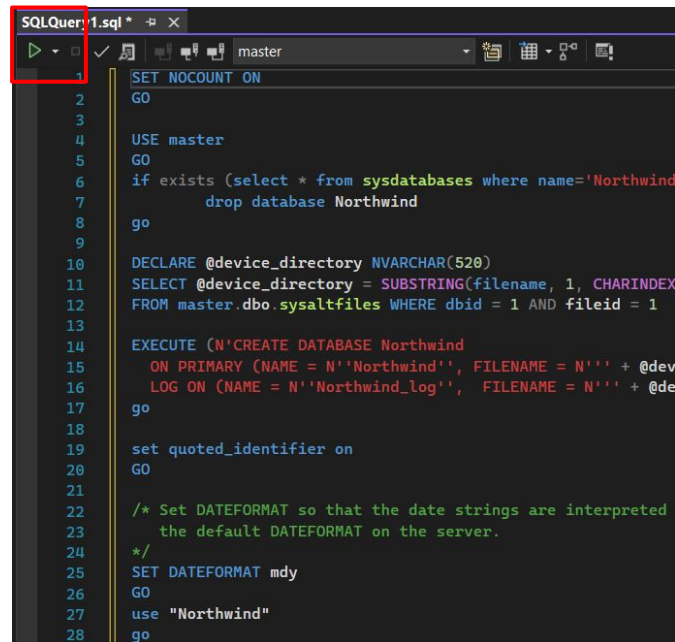
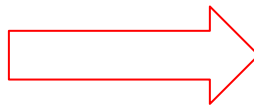
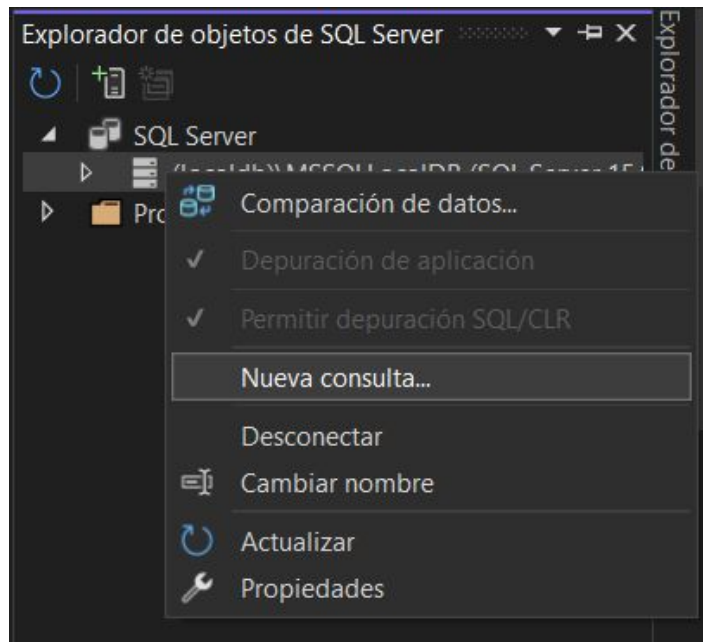
Dado que este enfoque parte de una base de datos existente, importamos una base de datos, con su conjunto de tablas, en nuestra base de datos local SQL Server Express LocalDB.

Utilizaremos la base de datos que podemos encontrar [aquí](#), y realizaremos:

- Descarga del archivo .zip y descomprimos.
- Abrimos el archivo de extensión .sql con un editor de texto (Sublime, notepad++, etc) y copiamos todas las consultas
- Realizamos una nueva consulta desde el explorador de objetos SQL Server de Visual Studio

EF Core - DataBase First

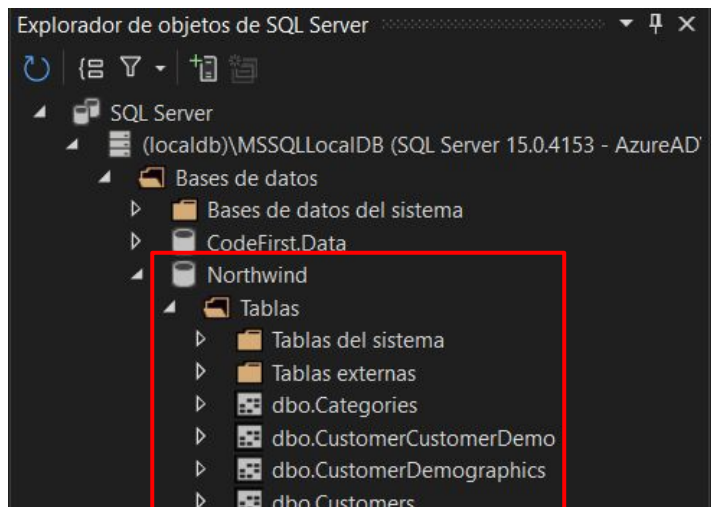
Importar una BD



EF Core - DataBase First

Importar una BD

Si inspeccionamos en el “Explorador de objetos de SQL Server” veremos que se ha creado la base de datos (Northwind en este ejemplo) con sus tablas



EF Core - DataBase First

Generar Modelos de la BD

Para realizar esta acción tendremos que hacer “ingeniería inversa” usando el comando Scaffold-DbContext

Esta acción generará las clases de los modelos y la clase de contexto (derivada de DbContext) basándose en el esquema de la base de datos existente.

NOTA: Es necesario tener instalados los paquetes NuGet:

- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

EF Core - DataBase First

Generar Modelos de la BD

El comando a usar tiene la siguiente Sintaxis:

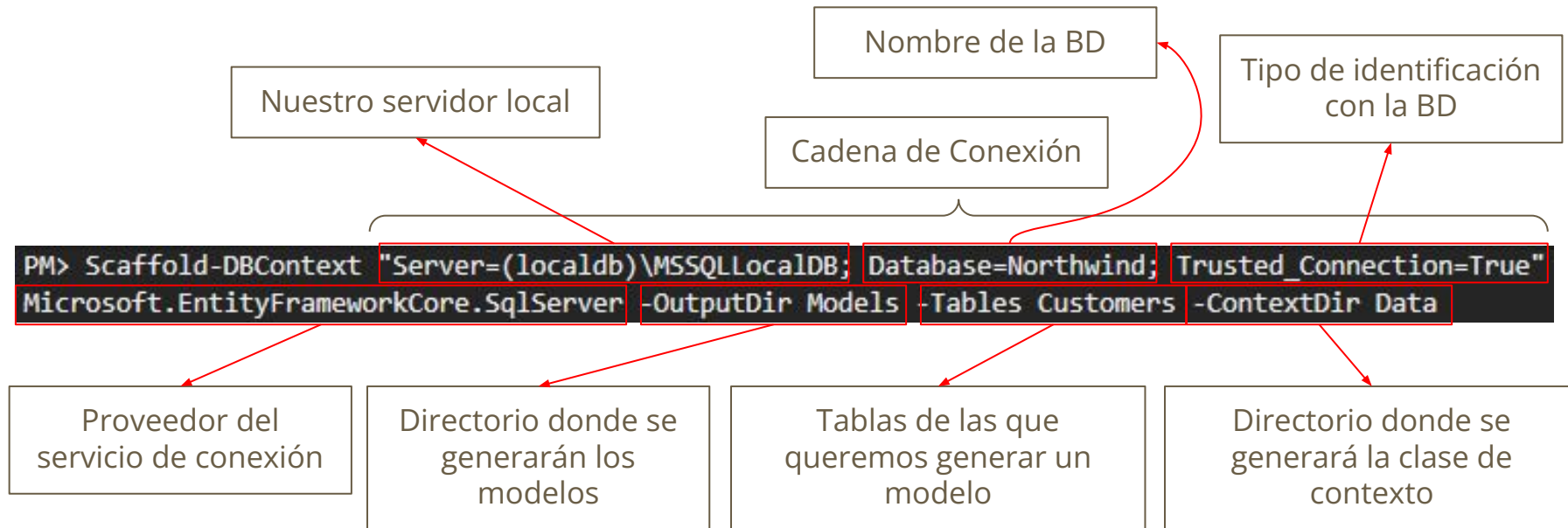
```
Scaffold-DbContext [-Connection] [-Provider] [-OutputDir] [-Context] [-Schemas>] [-Tables>]  
                  [-DataAnnotations] [-Force] [-Project] [-StartupProject] [<CommonParameters>]
```

Usaremos la “Consola de paquetes NuGet” para escribir el comando, que tendrá el siguiente aspecto:

EF Core - DataBase First

Generar Modelos de la BD

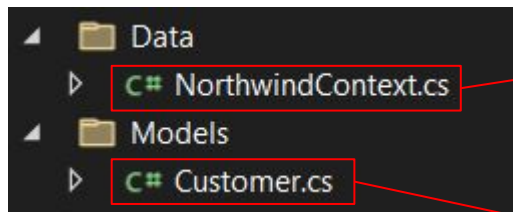
El comando a usar tiene la siguiente Sintaxis:



EF Core - DataBase First

Generar Modelos de la BD

Esta acción genera lo siguiente:



Clase del contexto de
conexión con la BD

Clase del modelo de la entidad de
la BD elegida

EF Core - DataBase First

Generar Controlador y Vistas del Modelo

En este punto ya podemos realizar el *scaffolding* en base al modelo, de la misma forma que lo hicimos en el enfoque *Code First* pero esta vez con el modelo generado por ingeniería inversa.

NOTA: Esta vez no tendremos que generar una nueva Clase de contexto de datos, ya que la hemos generado con el comando *Scaffold-DbContext*.

Por ello, tendremos que seleccionar la clase generada, y guardada en el directorio Data, en el desplegable correspondiente al hacer el *Scaffold*

EF Core - DataBase First

Añadir Clase de Contexto al Builder de la Aplicación

Finalmente, para que nuestro proyecto pueda realizar consultas con la base de datos importada, tenemos que añadir nuestra clase de contexto (DbContext) como servicio en el builder de nuestra WebApplication

Esto lo realizaremos en el archivo "Program.cs"

```
builder.Services.AddDbContext<NorthwindContext>();
```

Si miramos en "NorthwindContext.cs" vemos que la cadena de conexión está definida dentro, junto a un mensaje *Warning*.

Lo recomendable es incluir esa cadena en el archivo appsettings.json de la misma forma que se incluye al hacer *CodeFirst*.

Ejercicio

Realizar las siguientes acciones:

- Generar un nuevo proyecto ASP.NET Core MVC
- Importar la base de datos del ejemplo
- Generar, mediante el ORM, un modelo de la entidad *Suppliers*
- Realizar el *scaffold* del controlador y vistas
- Editar lo necesario para conseguir ejecutar el proyecto y que se listen por defecto los elementos de la entidad *Suppliers*

Enfoque Híbrido

En ocasiones será necesario aplicar los dos enfoques utilizados en un mismo proyecto.

Esto sucede cuando iniciamos un proyecto que parte de una base de datos existente (utilizaremos Database First para generar los modelos) y queramos aplicarla con los nuevos modelos generados en el código (utilizando Code First)

Partiendo de la base de datos Northwind, realizaremos lo siguiente:

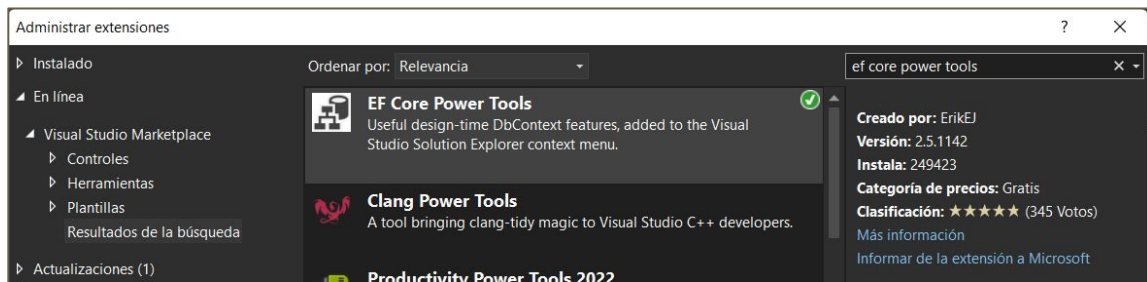
- Generar el modelo de una tabla
- *Scaffold* de vistas y controlador
- Creación de un nuevo modelo
- Migración y Seeder

Enfoque Híbrido

Generar el modelo de una tabla

Esta vez realizaremos este proceso de ingeniería inversa mediante una herramienta, en forma de extensión de VS, que ofrece interfaz gráfica para el proceso

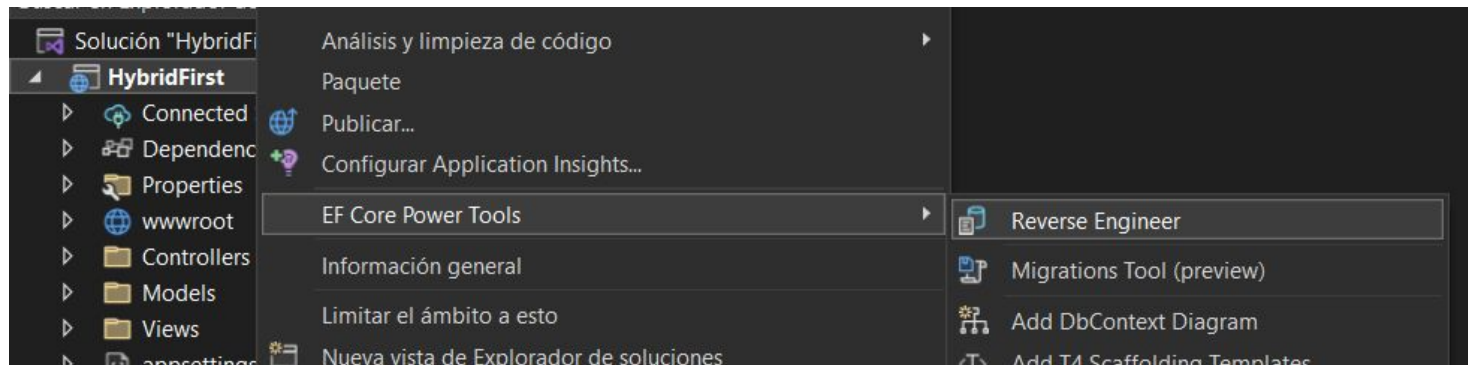
Buscamos e instalamos la extensión “EF Core Power Tools” desde el administrador de extensiones en Extensiones → Administrar Extensiones



Enfoque Híbrido

Generar el modelo de una tabla

Seleccionamos la opción *Reverse Engineer* en el menú contextual del proyecto.

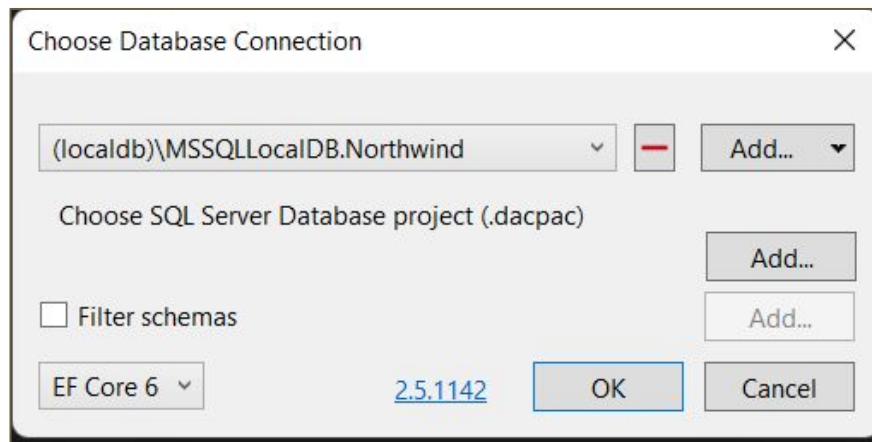


Enfoque Híbrido

Generar el modelo de una tabla

Indicamos la base de datos SQL Server creada en nuestro LocalDB

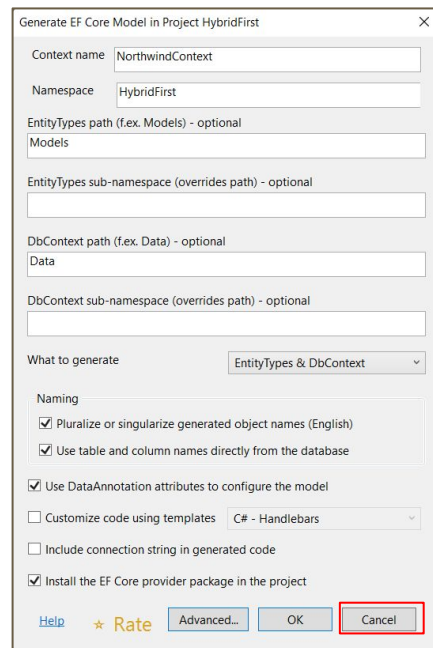
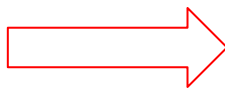
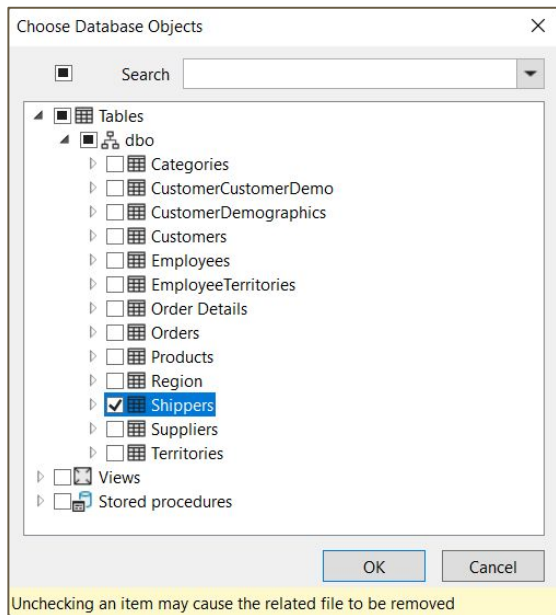
Podemos obtener este dato de la cadena de conexión de la BD.



Enfoque Híbrido

Generar el modelo de una tabla

Indicamos las tablas y seleccionamos las opciones que correspondan.



Enfoque Híbrido

Generar el modelo de una tabla

Esto genera los modelos de las entidades elegidas, la clase de contexto y un archivo `efpt.config.json` con la información del proceso realizado.

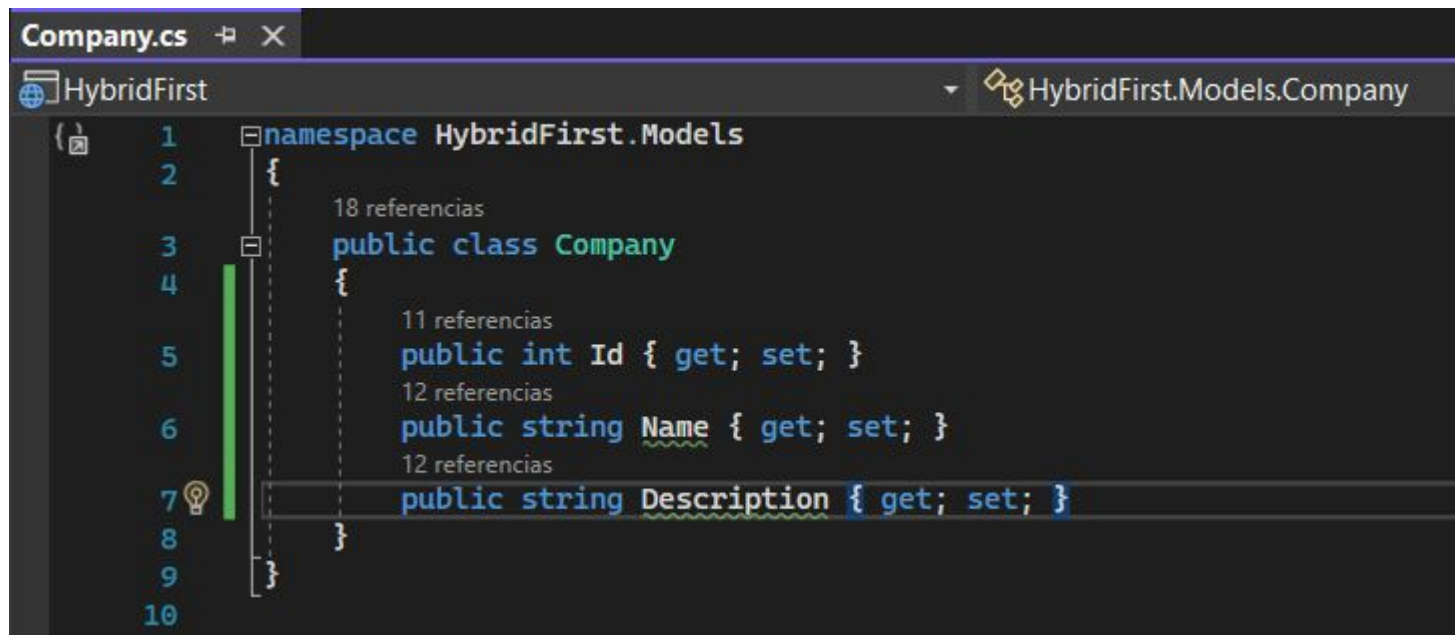
En este punto, y para resolver ciertos errores actuales o futuros, se tendrán que realizar algunas acciones:

- Instalar los paquetes NuGet utilizados en los ejemplos anteriores
- Añadir la clase de contexto al *services provider* del *builder* de la aplicación
- Incluir la cadena de conexión en la clase de contexto o en el archivo “`appsettings.json`” (mejor esta segunda opción).

Enfoque Híbrido

Creación de un nuevo modelo

Creamos un modelo sencillo *Company* y hacemos el *scaffolding* de este.



```
Company.cs  + - X
HybridFirst  HybridFirst.Models.Company
1 namespace HybridFirst.Models
2 {
3     18 referencias
4     public class Company
5     {
6         11 referencias
7         public int Id { get; set; }
8         12 referencias
9         public string Name { get; set; }
10        12 referencias
11        public string Description { get; set; }
12    }
13 }
```

Enfoque Híbrido

Scaffold de vistas y controlador

Este proceso es idéntico al realizado anteriormente y no tendría que surgir ningún problema si se han realizado correctamente las acciones indicadas en la diapositiva anterior.

En este punto solo nos queda usar el ORM de EF Core para crear la tabla de este nuevo modelo en la base de datos.

Lo hacemos mediante una migración y la rellenamos mediante un *seeder*.

Enfoque Híbrido

Migración y *seeder*

Procedemos de la misma forma que hicimos en el ejemplo de Code First, pero solo para este nuevo modelo *Company* que hemos creado.

Creamos la migración: esta vez tendremos que editar el archivo de migración generado para que solamente actúe sobre el modelo del que queremos generar tabla en la BD, ya que el otro ya existe en la BD.

Verificamos que se haya creado la tabla correctamente.

Creamos la clase *seeder* para alimentar la tabla generada con 3 registros y la invocamos en Program.cs como anteriormente.

Relaciones Entre Entidades

La forma en que las entidades se relacionan en una base de datos (de ahí que son bases de datos relacionales) ha de ser reflejado de alguna manera en las clases de los modelos del proyecto.

Las formas en que se pueden definir estas relaciones en los modelos son múltiples, pero todas hacen referencia a 3 formas básicas de relación entre entidades:

- One to One
- One to Many
- Many to Many

Relaciones Entre Entidades

One to One ([Ref.](#))

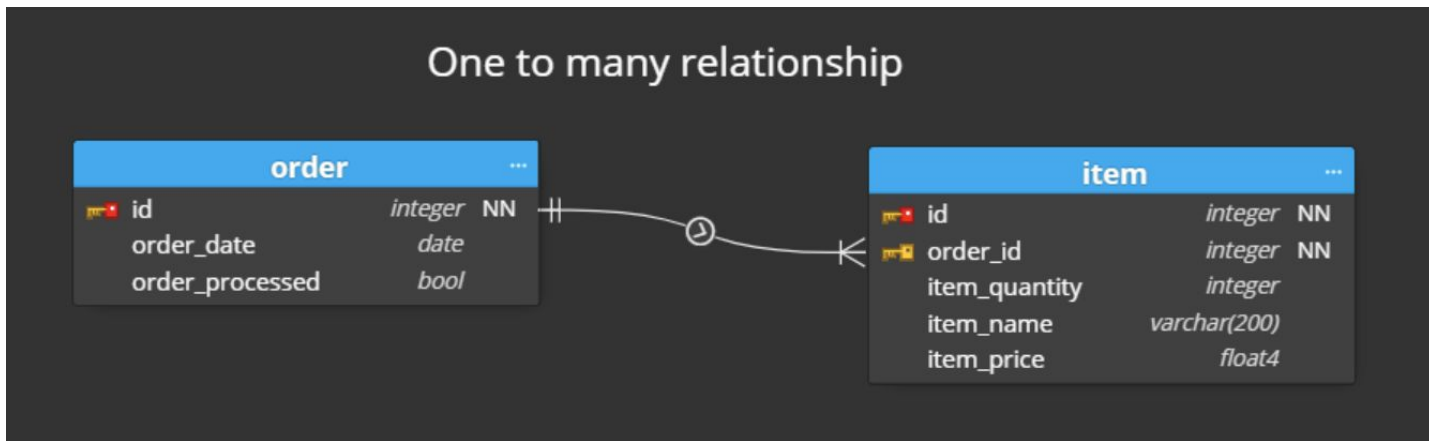
Son aquellas relaciones entre dos tablas donde un registro de una entidad hace referencia a otro registro de otra entidad, y viceversa.



Relaciones Entre Entidades

One to Many ([Ref.](#))

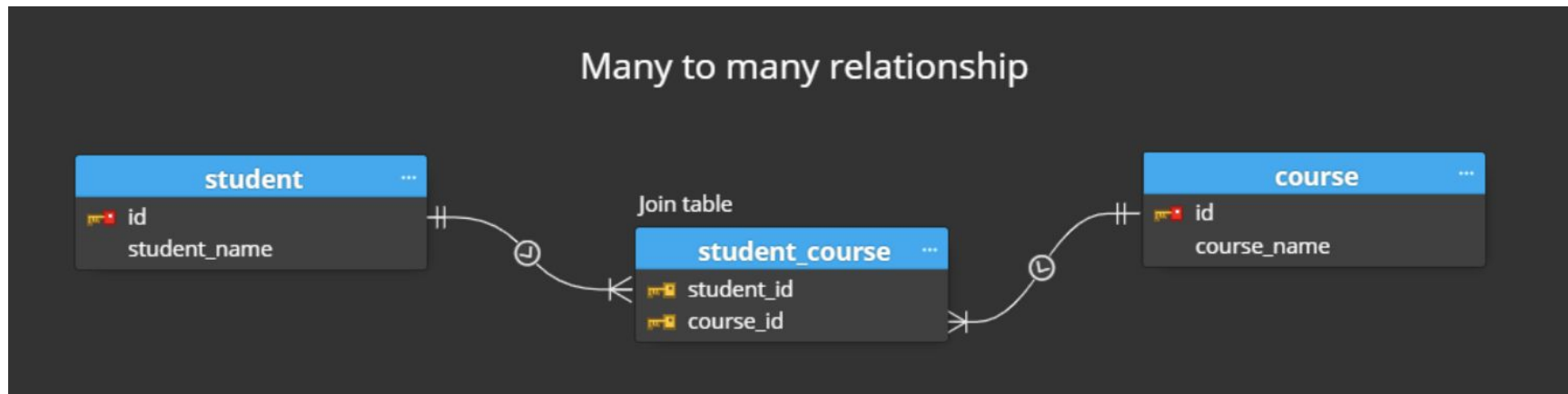
Son aquellas relaciones entre dos tablas donde un registro de una entidad hace referencia a otro registro de otra entidad, y el contrario puede ser referenciado por varios registros.



Relaciones Entre Entidades

Many to Many ([Ref.](#))

Son aquellas relaciones entre dos tablas donde un registro de una entidad puede ser referenciado por varios de otra entidad, y viceversa.



Relaciones Entre Entidades

Ejemplo de relación One to Many

Para entender cómo se lleva esto a la práctica vamos a incluir un campo en nuestro modelo *Company* que haga referencia a un elemento del modelo *Shipper*. Se han de modificar las clases implicadas de la siguiente forma:

```
public class Company
{
    11 referencias
    public int Id { get; set; }
    15 referencias
    public string? Name { get; set; }
    15 referencias
    public string? Description { get; set; }

    11 referencias
    public int ShipperID { get; set; }
    9 referencias
    public Shipper? Shipper { get; set; }
}
```

```
public partial class Shipper
{
    [Key]
    11 referencias
    public int ShipperID { get; set; }
    [Required]
    [StringLength(40)]
    15 referencias
    public string CompanyName { get; set; }
    [StringLength(24)]
    12 referencias
    public string Phone { get; set; }

    0 referencias
    public List<Company> Company { get; set; }
}
```

Relaciones Entre Entidades

Ejemplo de relación One to Many

Una vez modificados los modelos, se tendrán que realizar las siguientes acciones:

- Deshacer los cambios de la BD realizados por la migración inicial
- Borrar la tabla creada
- Generar una nueva migración que incluya los nuevos cambios del modelo
- Actualizar la base de datos con esta migración
- Actualizar el *seeder* si lo hubiera
- Regenerar el *scaffold* de vistas y controladores

Ejercicio

Realizar las siguientes acciones:

- Partiendo de la base de datos de Northwind, mediante técnicas de Database First, generar el modelo de la tabla *Shippers*, así como el *scaffold* de las vistas y controlador asociadas a este modelo.
- Generar manualmente una clase *Company* que tenga los campos (Id, Name y Description). Como en el caso anterior, generar el scaffold de vistas y controlador de este modelo. Mediante técnicas Code First, generar la tabla correspondiente en la BD.
- Verificar que la aplicación se ejecuta sin errores y se cargan las acciones Index de cada modelo.
- Generar una relación One-To-Many de *Company* a *Shippers* de forma que cada empresa trabaje únicamente con otra empresa de transporte.
- Realizar nuevas migraciones del modelo *Company* para incluir estos cambios en la BD y crear una clase SeedData para insertar tres registros en la tabla *Company* que tengan una clave foránea a *Shippers*