Tema 4

Mapeo Objeto Relacional (ORM)

Acceso a Datos

salesianos
LOS BOSCOS · LOGROÑO

Jesús Allona del Río @ Salesianos Los Boscos - Logroño

Acceso Básico a BD

En el acceso básico a BD desde un lenguaje orientado a objetos (POO), es necesario mezclar código y conceptos muy diferentes.

Por un lado tenemos la aplicación en sí. Dentro de éste programa hacemos uso de ciertos objetos especializados en conectarse con bases de datos (SqlConnection) y lanzar consultas contra ellas (SqlCommand). Estos objetos, en realidad son conceptos de la base de datos llevados a un programa orientado a objetos, lo cual supone una mezcla de distintos paradigmas. Finalmente, para lanzar consultas (tanto de obtención de datos, como de modificación o de gestión) se introducen **instrucciones en lenguaje SQL**, en forma de cadenas de texto, a través de estos objetos.

Además, está el problema de **los tipos de datos**. Generalmente el modo de representarlos y sus nombres pueden variar entre la plataforma de desarrollo y la base de datos (string / varchar(9), etc.). Y tampoco debemos olvidarnos de **los valores nulos** en la base de datos, que pueden causar todo tipo de problemas según el soporte que tengan en el lenguaje de programación con el que nos conectamos a la base de datos.

POO vs BDR

Otros posibles problemas y diferencias surgen del modo de pensar que tenemos en un lenguaje orientado a objetos y una base de datos. **En POO**, por ejemplo, para **representar un pedido y sus líneas de pedido** podrías definir un **objeto Pedido con una propiedad Lineas**, y si quieres consultar las líneas de un pedido solo debes escribir **pedido.Lineas** y listo, sin pensar en cómo se relacionan o de dónde sale esa información. **En** una **base de datos**, sin embargo, esto se modela con **dos tablas diferentes**, una para los **pedidos** y otra para las **líneas**, además de ciertos **índices y claves externas** entre ellas que relacionan la información. Si además se diese el caso de que una misma línea de pedido pudiese pertenecer a más de un pedido, necesitas una tercera tabla intermedia que se relaciona con las dos anteriores y que hace posible localizar la información. Como vemos, **formas completamente diferentes de pensar sobre lo mismo**.

ORM

Como acabamos de ver, lo ideal en una aplicación escrita en un lenguaje orientado a objetos sería definir una serie de clases, propiedades de éstas que hagan referencia a las otras, y trabajar de modo natural con ellas.

¿Qué quieres un pedido? Simplemente instancias un objeto Pedido pasándole su número de pedido al constructor. ¿Necesitas sus líneas de pedido? LLamas a la propiedad Lineas del objeto. ¿Quieres ver los datos de un producto que está en una de esas líneas? Solo lee la propiedad correspondiente y tendrás la información a tu alcance. Nada de consultas, nada de relaciones, de claves foráneas...

En definitiva, nada de conceptos de bases de datos en tu código orientado a objetos. **En la práctica, para ti la base de datos es como si no existiera.**

Un ORM es una biblioteca especializada en acceso a datos que genera por ti todo lo necesario para conseguir esa abstracción de la que hemos hablado. Gracias a un ORM ya no necesitas utilizar SQL nunca más, ni pensar en tablas ni en claves foráneas. Sigues pensando en objetos como hasta ahora, y te olvidas de lo que hay que hacer "por debajo" para obtenerlos.

El ORM puede generar clases a partir de las tablas de una base de datos y sus relaciones, o hacer justo lo contrario: partiendo de una jerarquía de clases crear de manera transparente las entidades necesarias en una base de datos, ocupándose de todo (ni tendrás que tocar el gestor de bases de datos para nada).

Ventajas de un ORM

- No tienes que escribir código SQL
- Te dejan sacar partido a las bondades de la programación orientada a objetos, incluyendo por supuesto la herencia.
- Nos permiten aumentar la reutilización del código y mejorar el mantenimiento del mismo, ya que tenemos un único modelo en un único lugar.
- Mayor seguridad, ya que se ocupan automáticamente de higienizar los datos que llegan, evitando posibles ataques de inyección SQL y similares.
- Hacen muchas cosas por nosotros: desde el acceso a los datos (lo obvio), hasta la conversión de tipos.

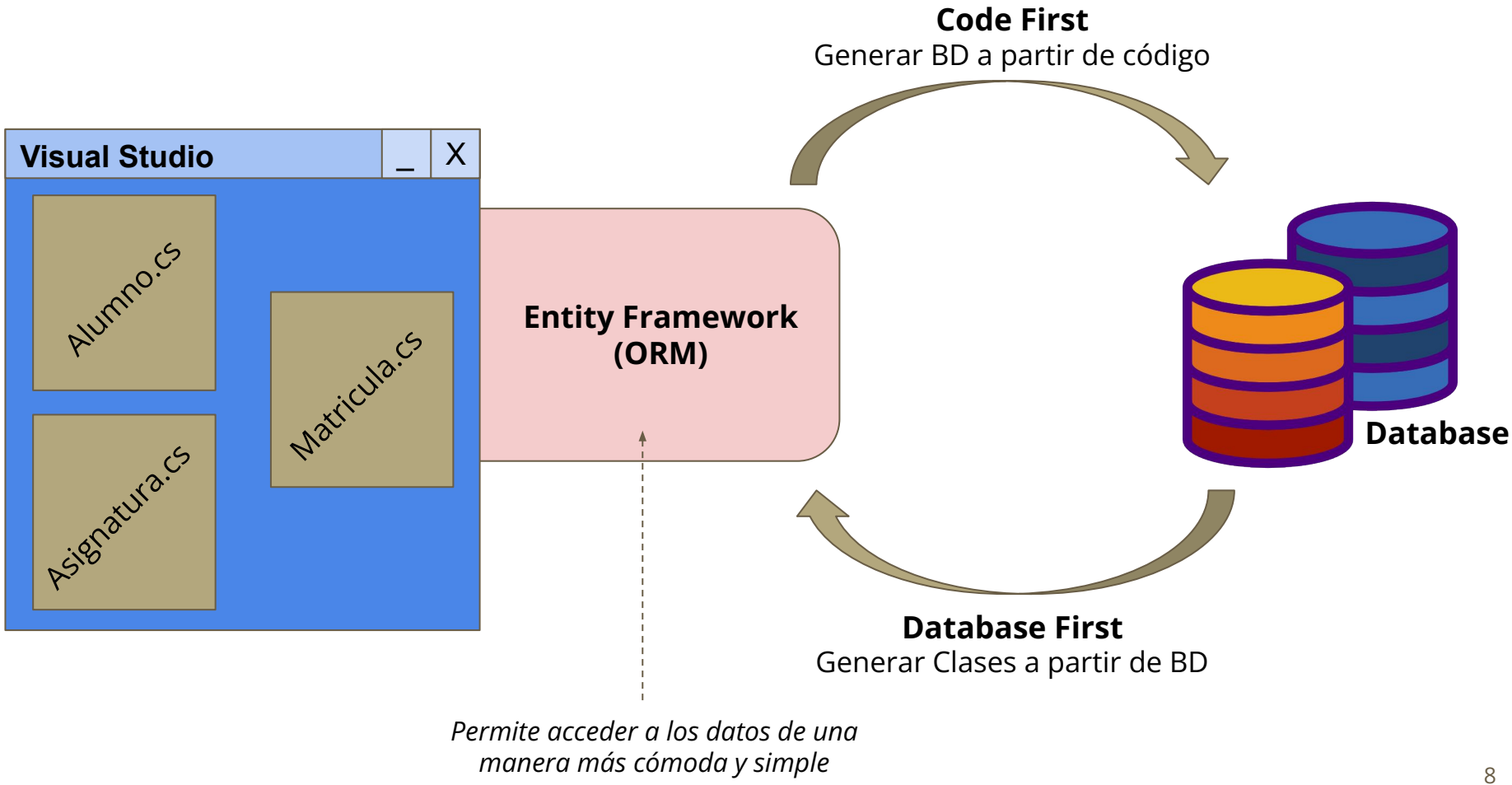
Desventajas de un ORM

- Pueden llegar a ser muy complejos, teniendo una curva de aprendizaje acentuada y requiriendo practicar con ellos hasta tener seguridad en su manejo diario.
- No son ligeros por regla general: añaden una capa de complejidad a la aplicación que puede hacer que empeore su rendimiento, especialmente si no los dominas a fondo.
- La configuración inicial que requieren se puede complicar dependiendo de la cantidad de entidades que se manejen y su complejidad, del gestor de datos subyacente, etc...
- El hecho de que te aísle de la base de datos y no tengas casi ni que pensar en ella es un arma de doble filo. Si no sabes bien lo que estás haciendo y las implicaciones que tiene en el modelo relacional puedes construir modelos que generen consultas monstruosas y muy poco óptimas contra la base de datos, agravando el problema del rendimiento y la eficiencia.

Entity Framework

Entity Framework es un mapeador relacional de objetos (**ORM** - Object Relational Mapper) para .NET con muchos años de desarrollo de características y estabilización por parte de Microsoft.

Como ORM, **EF** actúa como un puente entre los mundos de bases de datos relacionales y orientados a objetos, lo que permite a los desarrolladores escribir aplicaciones que interactúan con datos almacenados en bases de datos relacionales, con objetos .NET fuertemente tipados, que representan el dominio de la aplicación, y eliminar la necesidad de una gran parte del código de "mecánica" de acceso a datos que normalmente se debe escribir.


Code First
Generar BD a partir de código
Visual Studio
_
X
Alumno.cs
Matricula.cs
Asignatura.cs
Entity Framework
(ORM)
Database
Database First
Generar Clases a partir de BD
Permite acceder a los datos de una
manera más cómoda y simple

Database First

Database First es el método que nos permite primero crear la base de datos con sus tablas (y otras estructuras) y luego incorporarlas a la aplicación. Esto es necesario cuando la aplicación que realizamos necesita utilizar una base de datos existente. Otro uso que podemos darle es cuando necesitamos crear la base de datos, pero nos es más cómodo realizar la definición de las estructuras de la base de datos directamente con sentencias SQL, y luego importar los resultados en la aplicación (lo cual en muchos casos es muy útil y necesario, ya que nos permite definir las estructuras SQL tal cual las queremos o las necesitamos).

Code First

Code First se considera la alternativa más adecuada para aquellos casos en los que hay que crear la base de datos desde cero en conjunto con la aplicación, y es el equipo de desarrollo quien debe hacerlo. Usando Code First literalmente no deberemos escribir una sola sentencia SQL (lo cual no implica que no debamos conocer aspectos propios de su arquitectura).

Uso de Entity Framework mediante Code First

1. Debemos disponer de las clases que conforman el Modelo de nuestra aplicación. Las propiedades de tipo complejo (las que no son un tipo estándar cómo int, string…) las estableceremos cómo “virtual”.
2. Importación de la librería Entity Framework desde NuGet.
3. Crear cadena de conexión a la BD - App.Config → connectionStrings (en el proyecto de inicio)
4. Configurar el DataContext
5. Programar la población de datos (Configuration.cs → Seed) *[Opcional]*
6. Creación de la Base de Datos desde la consola de Visual Studio (Enable-migrations, Add-migration, Update-database)

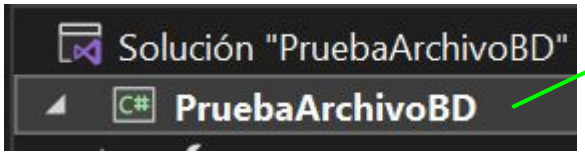
Se tiene que poner el proyecto que tiene el App.Config con la cadena de conexión cómo proyecto de inicio. Sino al lanzar el Update-Database lo buscará en el proyecto de inicio. Mejor usar: Add-Migration NombreMigracion -Force cuando hayamos modificado cosas antes de lanzar los cambios contra la BD. Mejor usar Update-Database -Verbose para revisar correctamente donde está buscando la cadena de conexión, donde está creando la BD, etc.

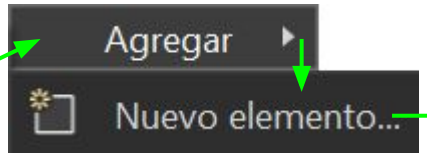
Crear LocalDB

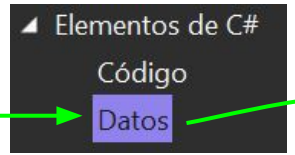
ESTABLECER EN "NO COPIAR" EL ARCHIVO .MDF, para que no se copie a bin\Debug, ya que se usará el de Data. Es más adecuado usar el de Data para evitar que no se borre al Limpiar o Recompilar la Solución.

Lo malo de SQL Server Express es que es un servidor de datos completo, y como tal, se instala en tu equipo y está todo el tiempo operativo, desde que inicias sesión, lo estés utilizando o no. Es decir, **ocupa memoria, carga las bases de datos y consume ciclos de CPU, lo estés utilizando o no**.

Sin embargo, **existe una edición menos conocida y especialmente pensada para usar cuando estás desarrollando**. Se trata de SQL Server LocalDB. Es una edición mínima de SQL Server que instala tan solo lo estrictamente necesario para poner a andar SQL Server en el momento cuando se necesite, quitándose del medio cuando no sea así. Gracias a SQL Server LocalDB tienes toda la potencia del servidor de datos, pero sin necesidad de tener un servicio consumiendo recursos de tu equipo constantemente, por lo que aprovechas mejor el hardware.



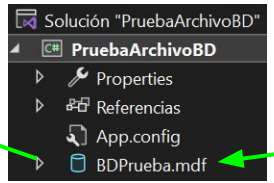


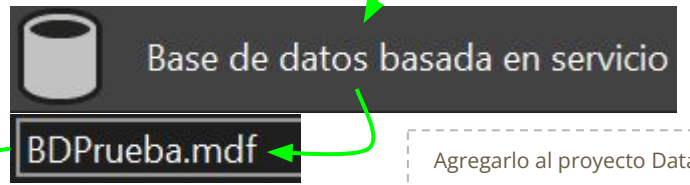


Para conectar desde Management:

(LocalDB)\.

CREATE TABLE
INSERT INTO
...



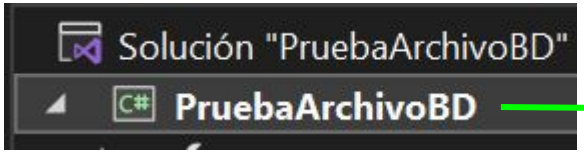


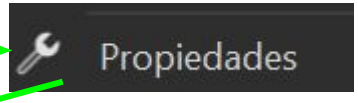
Agregarlo al proyecto Data

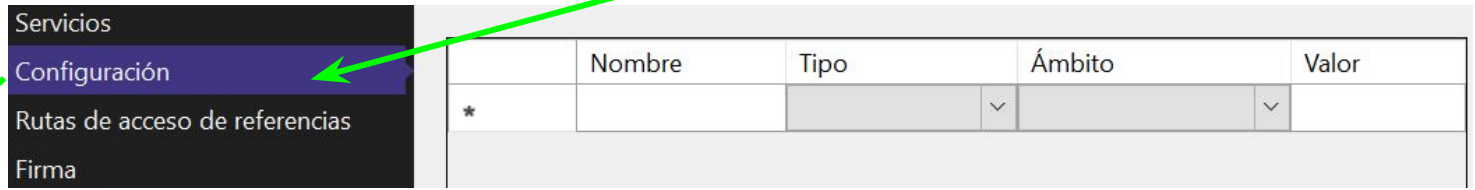
Agregar Conexión a LocalDB

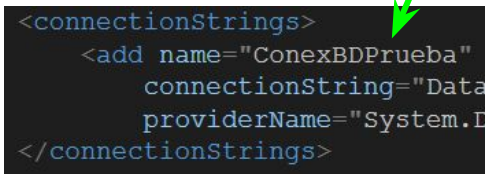
Esto hacerlo sobre las propiedades del proyecto de inicio

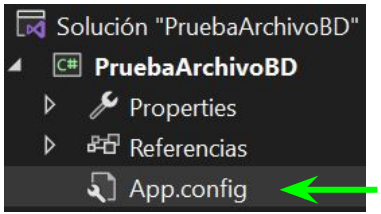
Cuando se quiera modificar una propiedad ya existente, modificar el "Valor" directamente, en lugar de volver a darle a los 3 puntitos, ya que sino, por algún motivo, no se guardan los cambios.

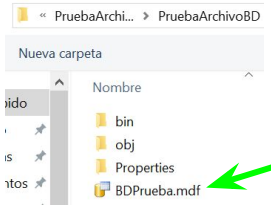


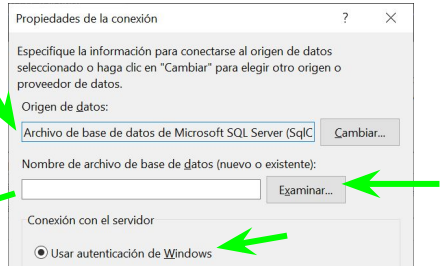












Cadena de Conexión

Es la cadena o string que contiene los datos necesarios para llevar a cabo la conexión con la base de datos deseada. Deberá introducirse en el fichero App.Config del proyecto de inicio de la solución (configuration/connectionStrings/add).
Cómo recomendación, usar el generador de cadena de conexión (Propiedades proyecto de inicio → Configuración → Agregar Cadena de conexión).

La apariencia que tiene es la siguiente:

```
<configuration>
  </connectionStrings>
      <!-- Para usar una bd de localdb (sin SQL Server): -->
      <!-- <add name="PeliculasBD_1" connectionString="Data Source=(localdb)\mssqllocaldb;Initial Catalog=PeliculasBD_1;Integrated Security=True" providerName="System.Data.SqlClient"/> -->
      <!-- Para usar con Database First: -->
      <!-- <add name="PeliculasBD_1" connectionString="metadata=res://*/PeliculasBD_1.csdl|res://*/PeliculasBD_1.ssdl|res://*/PeliculasBD_1.msl;provider=System.Data.SqlClient;provider connection
      string=&quot;data source=.;initial catalog=PeliculasBD_1;integrated security=True;multipleactiveresultsets=True;App=EntityFramework&quot;" providerName="System.Data.EntityClient" />-->
      <!-- Para usar con Code First: -->
      <add name="PeliculasBD_1" connectionString="Data Source=.\SQLEXPRESS; Initial Catalog=PeliculasBD_1; Integrated Security=True" providerName="System.Data.SqlClient"/>
  </connectionStrings>
</configuration>
```

- **name** → Nombre de la base de datos que se creará (Code First)
- **Data Source** → Es la dirección y/o puerto de la máquina donde está alojado el SGBD.
- **Initial Catalog** → Es la base de datos a la que se conectará la aplicación (si es Code First. Nombre que le dará a la BD)
- **Integrated Security** → Indica si queremos logearnos en el SGBD con las credenciales del SO donde estamos ejecutando el programa.
- **User Id** → Si no vamos a usar “Integrated Security”, sería el nombre de usuario de login.
- **Password** → Contraseña de login.

DataContext

```
using System.Data.Entity;

namespace Repository
{
    6 referencias
    public class PeliculasDbContext : DbContext
    {
        3 referencias
        public virtual DbSet<Pelicula> Peliculas { get; set; }
        0 referencias
        public virtual DbSet<Actor> Actores { get; set; }

        1 referencia
        public PeliculasDbContext()
            : base("PeliculasBD_1")
        {
            var instance = System.Data.Entity.SqlServer.SqlProviderServices.Instance;
            //Database.SetInitializer(new PeliculasInitializer());
        }

        0 referencias
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
```

Siempre se hará uso de "System.Data.Entity"

Siempre hereda de "DbContext"

DbSets con cada colección de objetos

Nombre de la cadena de conexión almacenada en App.config (connectionStrings) del proyecto de inicio. Todo el name

Truquillo que sería necesario para usar EF desde un proyecto diferente al de inicio (capas).

Método OnModelCreating, que nos va a permitir configurar el modelo mediante Fluent Api.

FluentApi

```
modelBuilder.Entity<Blog>()
    .Property(b => b.Url)
    .IsRequired();
```

VS

DataAnnotations

```
[Required]
public string Url { get; set; }
```

OnModelCreating

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Forzamos los nombres de las tablas (sino pondrá los nombres de las clases en plural Peliculas, Actors)
    modelBuilder.Entity<Pelicula>().ToTable("Pelicula");
    modelBuilder.Entity<Actor>().ToTable("Actor");

    // Configurar la relación muchos a muchos (N:N), para darle los nombres deseados a las FK y a la linking table
    modelBuilder.Entity<Pelicula>()
        .HasMany<Actor>(p => p.Actores)
        .WithMany(a => a.Peliculas)
        .Map(pa =>
        {
            pa.MapLeftKey("PeliculaId");
            pa.MapRightKey("ActorId");
            pa.ToTable("Pelicula_Actor");
        });

    // Indicamos si algún campo no es opcional (lo opuesto sería IsOptional())
    modelBuilder.Entity<Pelicula>()
        .Property(p => p.Titulo)
        .IsRequired();
}
```

Configuration.cs

La clase Configuration.cs, que podremos encontrar en la carpeta reservada Migrations de Entity Framework, nos permitirá programar la población de datos de nuestra BD. De tal manera, que cuando se cree por primera vez la BD, realice dichas inserciones de manera automática.

```
internal sealed class Configuration : DbMigrationsConfiguration<Repository.PeliculasDbContext>
{
    0 referencias
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }

    0 referencias
    protected override void Seed(Repository.PeliculasDbContext context)
    {
        IList<Pelicula> pelisIniciales = new List<Pelicula>();

        pelisIniciales.Add(new Pelicula(1, "El Señor de los Anillos I", 190));
        pelisIniciales.Add(new Pelicula(2, "Piratas del Caribe I", 150));
        pelisIniciales.Add(new Pelicula(3, "Snatch. Cerdos y Diamantes", 145));

        context.Peliculas.AddRange(pelisIniciales);

        base.Seed(context);
    }
}
```

Posibles Ajustes del Modelo a Tener en Cuenta

Antes de lanzar las migraciones deberemos tener en cuenta que Entity Framework no soporta el tipo de dato char, por lo que si tuviéramos alguna propiedad de nuestro modelo que fuese de dicho tipo, deberíamos hacer lo siguiente para dar solución a dicho problema sin que se vea trastocado el código ya existente.

```
[NotMapped]
public char Genero { get; set; }

[Column("Genero", TypeName = "char")]
[MaxLength(1)]
public string GeneroAux
{
   get => Genero.ToString();
   set => Genero = string.IsNullOrEmpty(value) ? '\0' :
value[0];
}
```

VS

```
modelBuilder.Entity<Pelicula>()
        .Ignore(p => p.Genero);

modelBuilder.Entity<Pelicula>()
        .Property(p => p.GeneroAux)
        .HasColumnType("char")
        .IsFixedLength()
        .HasMaxLength(1);
```

Migraciones Base de Datos (Crear / Modificar BD)

Haciendo uso de la consola de Visual Studio, podremos lanzar las siguientes operaciones *(Herramientas → Administrador de paquetes NuGet → Consola del administrador de paquetes)*:

- Enable-migrations
- Add-migration NombreMigracion
- Update-database

[Asegurarse de que el proyecto de inicio tiene el App.Config con la cadena de conexión]

Add-Migration NombreMigracion ***-Force*** cuando hayamos modificado cosas antes de lanzar los cambios contra la BD. Generará un fichero de migración en la carpeta Migrations, en el cual se podrán ver los cambios que se van a lanzar contra la base de datos.
Mejor usar *Update-Database* ***-Verbose*** para revisar correctamente donde está buscando la cadena de conexión, donde está creando la BD, etc.

Personalizar Migraciones

- Activar migraciones con un directorio personalizado e indicando el dbcontext con el que se desea trabajar (útil si hubiera varios dbcontext):
  Enable-Migrations -MigrationsDirectory "EntityFramework\Migrations\Pedidos" -ContextTypeName "Repository.EntityFramework.PedidosDbContext"

- Añadir migración usando una configuración concreta (la cual se habrá generado con la activación anterior):
  Add-Migration -ConfigurationTypeName "Repository.EntityFramework.Migrations.Pedidos.Configuration" Migracion_Pedidos_Inicial_1

- Si se ha cambiado la carpeta donde se almacenaban las migraciones después de haber hecho migraciones:
  Se deberá cambiar el ContextKey en la tabla MigrationHistory de la base de datos. Ejemplo:

  ```
  UPDATE __MigrationHistory
  SET ContextKey = 'Repository.EntityFramework.Migrations.Peliculas.Configuration'
  ```

- Para obtener las migraciones disponibles:
  Get-Migrations -ConfigurationTypeName Repository.EntityFramework.Migrations.Peliculas.Configuration

- Volver al estado de una migración anterior:
  Update-Database -TargetMigration "Migracion_inicial" -ConfigurationTypeName "Repository.EntityFramework.Migrations.Peliculas.Configuration"

- Si en algún momento se quiere borrar una migración, se debería borrar tanto la clase de la migración, cómo sus referencias de la tabla __MigrationHistory. Debería ser una migración “no efectiva”, la cual ha sido revertida en algún momento.

```
public class Pedido
{
    2 referencias
    public int Id { get; set; }
    1 referencia
    public string Descripcion { get; set; }
    1 referencia
    public DateTime FechaHoraPedido { get; set; }
    1 referencia
    public virtual List<LineaPedido> LineasPedido { get; set; }
}
```

```
public class LineaPedido
{
    2 referencias
    public int Id { get; set; }
    1 referencia
    public string Producto { get; set; }
    1 referencia
    public int Cantidad { get; set; }
    1 referencia
    public int PedidoId { get; set; }
    1 referencia
    public virtual Pedido Pedido { get; set; }
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Forzamos los nombres de las tablas (sino pondrá los nombres de
    modelBuilder.Entity<Pedido>().ToTable("Pedido");
    modelBuilder.Entity<LineaPedido>().ToTable("LineaPedido");
```

```
public override void Up()
{
    CreateTable(
        "dbo.Pedido",
        c => new
        {
            Id = c.Int(nullable: false, identity: true),
            Descripcion = c.String(),
            FechaHoraPedido = c.DateTime(nullable: false),
        })
        .PrimaryKey(t => t.Id);

    CreateTable(
        "dbo.LineaPedido",
        c => new
            {
                Id = c.Int(nullable: false, identity: true),
                Producto = c.String(),
                Cantidad = c.Int(nullable: false),
                PedidoId = c.Int(nullable: false),
            })
        .PrimaryKey(t => t.Id)
        .ForeignKey("dbo.Pedido", t => t.PedidoId, cascadeDelete: true)
        .Index(t => t.PedidoId);
}
```

```
public override void Down()
{
    DropForeignKey("dbo.LineaPedido", "PedidoId", "dbo.Pedido");
    DropIndex("dbo.LineaPedido", new[] { "PedidoId" });
    DropTable("dbo.Pedido");
    DropTable("dbo.LineaPedido");
}
```

Relación 1:N

```cs
public class Pelicula
{
    1 referencia
    public int Id { get; set; }
    2 referencias
    public string Titulo { get; set; }
    1 referencia
    public double Duracion { get; set; }
    2 referencias
    public virtual List<Actor> Actores { get; set; }
}
```

```cs
public class Actor
{
    1 referencia
    public int Id { get; set; }
    1 referencia
    public string Nombre { get; set; }
    1 referencia
    public DateTime FechaNacimiento { get; set; }
    1 referencia
    public char Sexo { get; set; }
}
```

```cs
public partial class Migracion_inicial : DbMigration
{
    0 referencias
    public override void Up()
    {
        CreateTable(
            "dbo.Peliculas",
            c => new
                {
                    Id = c.Int(nullable: false, identity: true),
                    Titulo = c.String(nullable: false),
                    Duracion = c.Double(nullable: false),
                })
            .PrimaryKey(t => t.Id);

        CreateTable(
            "dbo.Actors",
            c => new
                {
                    Id = c.Int(nullable: false, identity: true),
                    Nombre = c.String(),
                    FechaNacimiento = c.DateTime(nullable: false),
                    Pelicula_Id = c.Int(),
                })
            .PrimaryKey(t => t.Id)
            .ForeignKey("dbo.Peliculas", t => t.Pelicula_Id)
            .Index(t => t.Pelicula_Id);

    }
    0 referencias
    public override void Down()
    {
        DropForeignKey("dbo.Actors", "Pelicula_Id", "dbo.Peliculas");
        DropIndex("dbo.Actors", new[] { "Pelicula_Id" });
        DropTable("dbo.Actors");
        DropTable("dbo.Peliculas");
    }
}
```

Relación 1:N

*(Si no se especifica el atributo que será la FK, le añadirá un atributo con el nombre de la tabla a la que está unida, seguido de **_Id**)*

```cs
public class Pelicula
{
    3 referencias
    public int Id { get; set; }
    4 referencias
    public string Titulo { get; set; }
    3 referencias
    public double Duracion { get; set; }
    5 referencias
    public virtual List<Actor> Actores { get; set; }
}
```

```cs
public class Actor
{
    2 referencias
    public int Id { get; set; }
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public DateTime FechaNacimiento { get; set; }
    2 referencias
    public char Sexo { get; set; }
    4 referencias
    public virtual List<Pelicula> Peliculas { get; set; }
}
```

```cs
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    // Forzamos los nombres de las tablas (sino pondrá los nombres de las clases en plural Peliculas, Actors)
    modelBuilder.Entity<Pelicula>().ToTable("Pelicula");
    modelBuilder.Entity<Actor>().ToTable("Actor");

    // Configurar la relación muchos a muchos (N:N), para darle los nombres deseados a las FK y a la linking table
    modelBuilder.Entity<Pelicula>()
        .HasMany<Actor>(p => p.Actores)
        .WithMany(a => a.Peliculas)
        .Map(pa =>
        {
            pa.MapLeftKey("PeliculaId");
            pa.MapRightKey("ActorId");
            pa.ToTable("Pelicula_Actor");
        });
```

```cs
public partial class Migracion_inicial : DbMigration
{
    0 referencias
    public override void Up()
    {
        CreateTable(
            "dbo.Pelicula",
            c => new
            {
                Id = c.Int(nullable: false, identity: true),
                Titulo = c.String(nullable: false),
                Duracion = c.Double(nullable: false),
            })
            .PrimaryKey(t => t.Id);

        CreateTable(
            "dbo.Actor",
            c => new
            {
                Id = c.Int(nullable: false, identity: true),
                Nombre = c.String(),
                FechaNacimiento = c.DateTime(nullable: false),
            })
            .PrimaryKey(t => t.Id);
```

```cs
        CreateTable(
            "dbo.Pelicula_Actor",
            c => new
            {
                PeliculaId = c.Int(nullable: false),
                ActorId = c.Int(nullable: false),
            })
            .PrimaryKey(t => new { t.PeliculaId, t.ActorId })
            .ForeignKey("dbo.Pelicula", t => t.PeliculaId, cascadeDelete: true)
            .ForeignKey("dbo.Actor", t => t.ActorId, cascadeDelete: true)
            .Index(t => t.PeliculaId)
            .Index(t => t.ActorId);

    }

    0 referencias
    public override void Down()
    {
        DropForeignKey("dbo.Pelicula_Actor", "ActorId", "dbo.Actor");
        DropForeignKey("dbo.Pelicula_Actor", "PeliculaId", "dbo.Pelicula");
        DropIndex("dbo.Pelicula_Actor", new[] { "ActorId" });
        DropIndex("dbo.Pelicula_Actor", new[] { "PeliculaId" });
        DropTable("dbo.Pelicula_Actor");
        DropTable("dbo.Actor");
        DropTable("dbo.Pelicula");
    }
}
```

Relación N:N

Algunos recursos útiles para trabajar con EF

- Siempre será necesario hacer uso del **DataContext** para operar con la base de datos.

```
using (PeliculasDbContext context = new PeliculasDbContext())
{
        ...
        context.Peliculas
        ...
}
```

- Será muy útil el uso de Linq combinada con Entity Framework.
  .ToList(), .Any(), .FirstOrDefault(...), .Where(...), etc.

Carga de Datos con EF

- En EF, por defecto no se cargan los datos relacionados, por temas de eficiencia, y deberemos indicar expresamente que se produzca dicha carga. Existen diferentes maneras:
  - **Carga diligente**: al realizar la consulta de una entidad se cargan las entidades indicadas.
    context.Peliculas.Include("Actores").ToList(); → Obtiene el listado de películas con los Actores incluidos
    context.Peliculas.Include("Actores.Elementos").ToList(); → Lo mismo pero con varios niveles
  - **Carga diferida**: una entidad o colección de entidades se carga automáticamente desde la base de datos la primera vez que se tiene acceso a su propiedad. Estará activa cuando la propiedad en el objeto que la contiene esté declarada cómo "virtual".
    pelicula = context.Peliculas.Find(numeroPeli);
    pelicula.Actores.ToList(); → A partir de aquí, pelicula ya tendrá cargados los actores
  - **Carga explícita**:
    - *pelicula = context.Peliculas.Find(numeroPeli);*
      context.Entry(pelicula).Collection(p => p.Actores).Load(); → Le carga la colección de actores a la película indicada
    - *Actor* actor = context.Actores.Find(1, 14);
      ***context*.Entry(actor).Reference(a => a.Pelicula).Load();** → Le carga la película al actor indicado *(rel 1:N)*

Carga de Datos Personalizada con EF

- Aplicar filtros a la carga explícita (se deberá desactivar la carga diferida):

```
pedido = context.Pedidos.Find(numeroPedido);
context.Entry(pedido)
        .Collection(p => p.LineasPedido)
        .Query()
        .Where(l => l.PrecioUnidad > 0)
        .Load();
// Cargará en el pedido las líneas cuyo precio de unidad sea mayor de 0.
```

- Contar entidades relacionadas sin cargarlas:

```
pedido = context.Pedidos.Find(numeroPedido);
int lineasSinPrecio = context.Entry(pedido)
                             .Collection(p => p.LineasPedido)
                             .Query()
                             .Where(l => l.PrecioUnidad == 0)
                             .Count();
// Contará cuantas líneas del pedido dado no tienen precio
```

Edición de Datos con EF

- Editar datos:
  - Editar una propiedad de un objeto:

```
pedido = context.Pedidos.Find(numeroPedido);
pedido.Pagado = true;
bool result = context.SaveChanges();
```

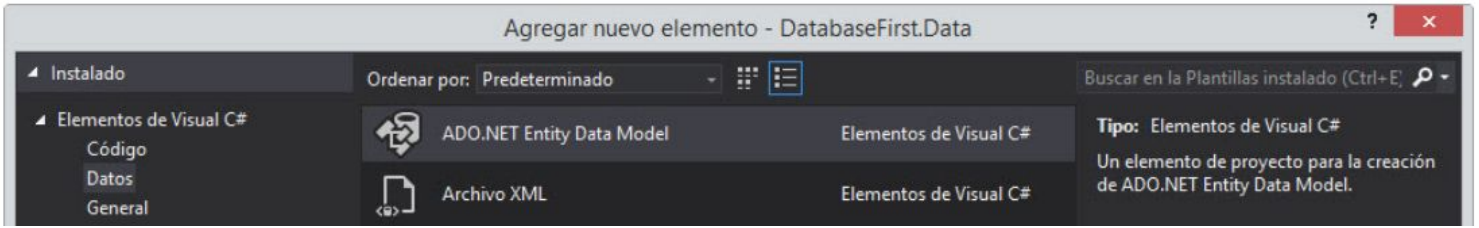
  - Editar un objeto COMPLETO a partir de otro:

```
context.Entry(pedido).CurrentValues.SetValues(pedidoEditado);
bool result = context.SaveChanges();
// SetValues nunca actualiza las propiedades de navegación. Cuando ejecuta su código, solo conoce los cambios en las propiedades simples / complejas de la entidad pasada a su método Update. Es decir, para modificar la lista de líneas de un pedido, no bastará con hacerlo directamente sobre el pedido (pedido.LineasPedido...), sino que deberemos usar el DbSet de LineasPedido (context.LineasPedido...).
```

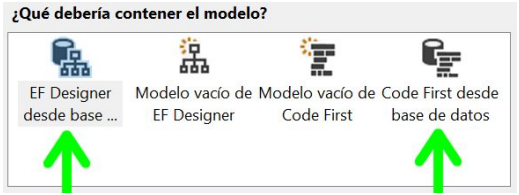
- Añadir datos:
  - context.Pedidos.Add(new Pedido(...));
  - bool result = context.SaveChanges();

Uso de Entity Framework mediante Database First

1. Debemos disponer de una base de datos creada
2. Importación de la librería Entity Framework desde NuGet.
3. Agregar al proyecto un nuevo elemento “ADO.NET Entity Data Model”

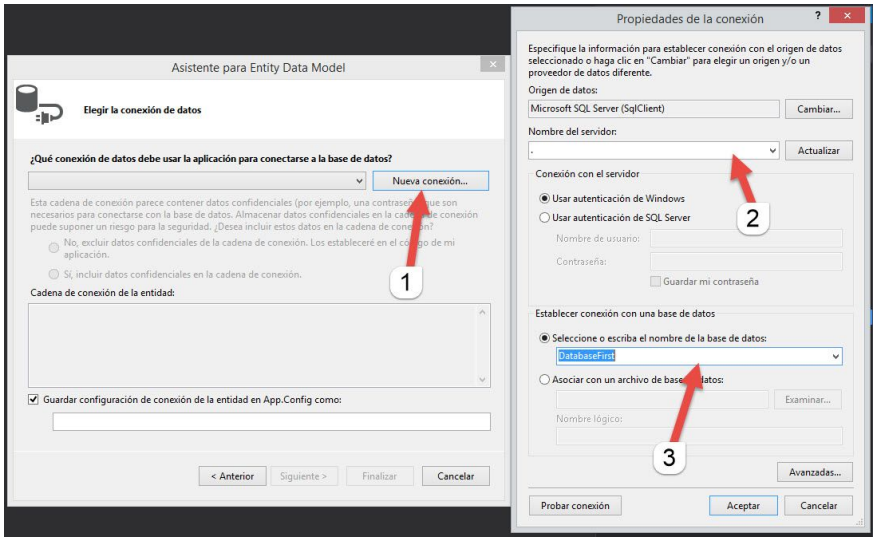


4. Seleccionar el método de generación, “Database First” o “Code First”.



Uso de Entity Framework mediante Database First

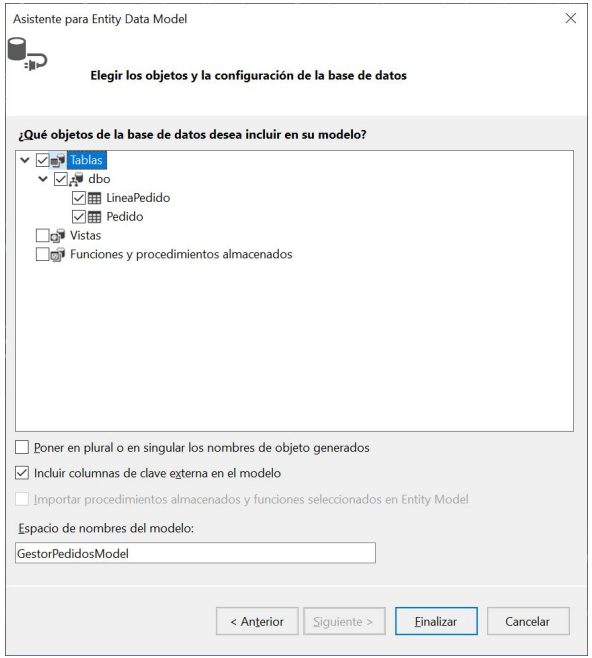
5. Nos saldrá el generador de la cadena de conexión, el cual nos guiará para crearla.



Deberemos ingresar el nombre del servidor, datos de autenticación en caso de que sean necesarios y la base de datos que queremos utilizar. Una vez que presionamos "Aceptar" tendremos la confirmación de la cadena de conexión a utilizar.

Uso de Entity Framework mediante Database First

6. Seleccionar los objetos y configuración de la base de datos.



Una vez finalizado el proceso, se hará el mapeo de las estructuras en el DataContext, y se generarán las clases que correspondan a las tablas de la base de datos.

Fin