



# Desarrollo de Software Empresarial

CICLO 02 -2024

# Guía 2: Introducción a ASP.NET MVC – Parte #2

## Competencias

- Que el estudiante a configurar una cadena la conexión a base de datos para un proyecto ASP .NET MVC.
- Que el estudiante aprenda a acceder a la base de datos.
- Que el estudiante aprenda a implementar migraciones incrementales.

## Introducción Teórica

### SQL Server Express LocalDB

SQL Server Express LocalDB es una versión ligera y de bajo consumo de recursos de Microsoft SQL Server Express, diseñada principalmente para el desarrollo y las pruebas de aplicaciones. LocalDB se ejecuta en modo de usuario y no requiere instalación de servicio ni configuración complicada.

### Características Clave

- *Facilidad de Uso:*
  - No requiere una instalación complicada ni configuración de servicio.
  - Se ejecuta en modo de usuario, lo que facilita su uso para desarrolladores y entornos de prueba.
- *Ligero y Portátil:*
  - LocalDB es una versión ligera y portátil de SQL Server, ideal para desarrolladores que necesitan una base de datos SQL Server para pruebas y desarrollo sin la sobrecarga de una instalación completa de SQL Server.
  - Puede ser fácilmente distribuido junto con las aplicaciones.

- *Compatibilidad:*
  - Ofrece compatibilidad con la mayoría de las características de SQL Server, como T-SQL y funciones de administración de bases de datos.
  - Permite trabajar con bases de datos en formato .mdf, lo que facilita la migración a versiones más completas de SQL Server.
- *Almacenamiento:*
  - Soporta bases de datos de hasta 10 GB, lo que es adecuado para aplicaciones pequeñas y medianas.
  - Utiliza un archivo de datos (.mdf) que se puede mover entre diferentes instancias de SQL Server.
- *Escalabilidad Limitada:*
  - No está diseñado para entornos de producción a gran escala, pero puede escalarse fácilmente a versiones más robustas de SQL Server si es necesario.

En resumen, SQL Server Express LocalDB es una herramienta ideal para desarrolladores que buscan una solución de base de datos ligera y fácil de usar para el desarrollo y las pruebas, pero tiene limitaciones que la hacen inadecuada para aplicaciones de producción o entornos que requieran mayores capacidades y funcionalidades avanzadas.

## Migraciones con Entity Framework

¿Qué es Entity Framework?

Entity Framework (EF) es un Object-Relational Mapper (ORM) que permite a los desarrolladores de .NET trabajar con bases de datos utilizando objetos .NET. Simplifica el proceso de acceder y manipular datos al mapear clases del modelo de dominio a tablas de la base de datos y viceversa.

¿Qué son las Migraciones?

Las migraciones en Entity Framework son un sistema para gestionar los cambios en el esquema de la base de datos a lo largo del tiempo en un modelo **Code First**, a la vez que se mantiene sincronizado con el modelo de datos de la aplicación. Esto permite a los desarrolladores evolucionar su base de datos a medida que cambian sus requisitos.

# Materiales y Equipos

1. Guía No 2.
2. Computadora con programa Microsoft Visual Studio 2022.

# Procedimiento

## Configuración de conexión a la base de datos

Para configurar la conexión a la base de datos en aplicaciones ASP.NET MVC utilizaremos el archivo `appsettings.json`. Este archivo es fundamental y puede ser utilizado para almacenar diversos parámetros de configuración de una manera que es fácil de mantener y actualizar sin necesidad de recompilar la aplicación.

Para configurar la conexión a la base de datos en un proyecto nuevo o existente se detalla los pasos a continuación:

1. Abra el proyecto de su elección.
2. Localice el archivo **appsetting.json**, inicialmente debe contener genérica para el log de eventos de la aplicación.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

3. Agregar el siguiente contenido al archivo para configurar la conexión a la base de datos

```
{
  "ConnectionStrings": {
    "PelículasCN": "Server=(localdb)\\MSSQLLocalDB;Database=Películas;Trusted_Connection=True;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

4. Registrar el contexto en contenedor de dependencias en el archivo **Program.cs**.

```
builder.Services.AddDbContext<PelículasDbContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("PelículasCN")));

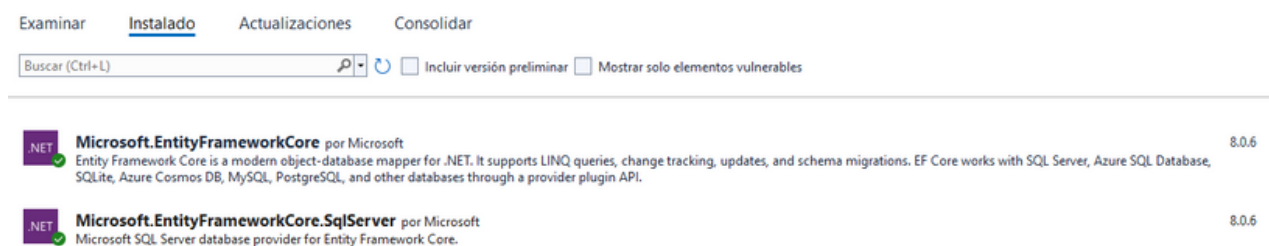
using MVCPelícula.Models;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

//Add DbContext
builder.Services.AddDbContext<PelículasDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("PelículasCN")));
```

**Nota: si al utilizar la configuración anterior genera un error, verifique que tenga correctamente instalados los siguientes paquetes en su proyecto.**



Examinar Instalado Actualizaciones Consolidar

Buscar (Ctrl+L)   ☐ Incluir versión preliminar ☐ Mostrar solo elementos vulnerables

**Microsoft.EntityFrameworkCore** por Microsoft 8.0.6  
Entity Framework Core is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, and schema migrations. EF Core works with SQL Server, Azure SQL Database, SQLite, Azure Cosmos DB, MySQL, PostgreSQL, and other databases through a provider plugin API.

**Microsoft.EntityFrameworkCore.SqlServer** por Microsoft 8.0.6  
Microsoft SQL Server database provider for Entity Framework Core.

## Proceso de Migraciones

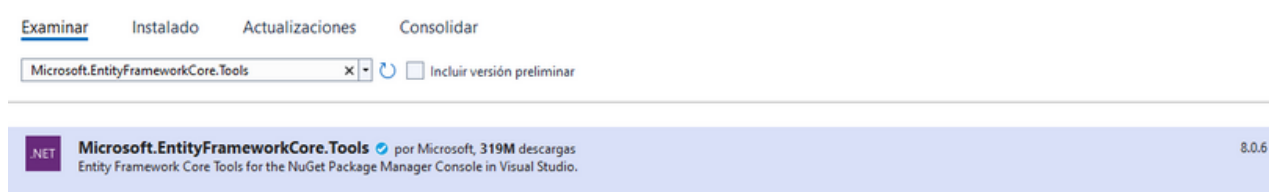
Las migraciones con Entity Framework proporcionan una manera poderosa y eficiente de gestionar los cambios en el esquema de la base de datos, manteniéndolo sincronizado con el modelo de datos de la aplicación. Esto facilita la evolución de la aplicación y la base de datos de manera coherente y controlada.

El proceso para utilizar migraciones con Entity Framework es el siguiente:

1. Utilizaremos el modelo Película (creado en la guía no. 1), además, se utilizará el contexto de datos, que actúa como un puente entre las entidades y la base de datos.

```
namespace MVCPelícula.Models
{
    1 referencia
    public class Película
    {
        0 referencias
        public int ID { get; set; }
        0 referencias
        public string Título { get; set; }
        0 referencias
        public DateTime FechaLanzamiento { get; set; }
        0 referencias
        public string Genero { get; set; }
        0 referencias
        public decimal Precio { get; set; }
    }
}
```

2. Instalar el paquete **Microsoft.EntityFrameworkCore.Tools** desde el Administrador de Paquetes NuGet, de lo contrario no podremos utilizar los comandos para agregar, aplicar y revertir las migraciones.

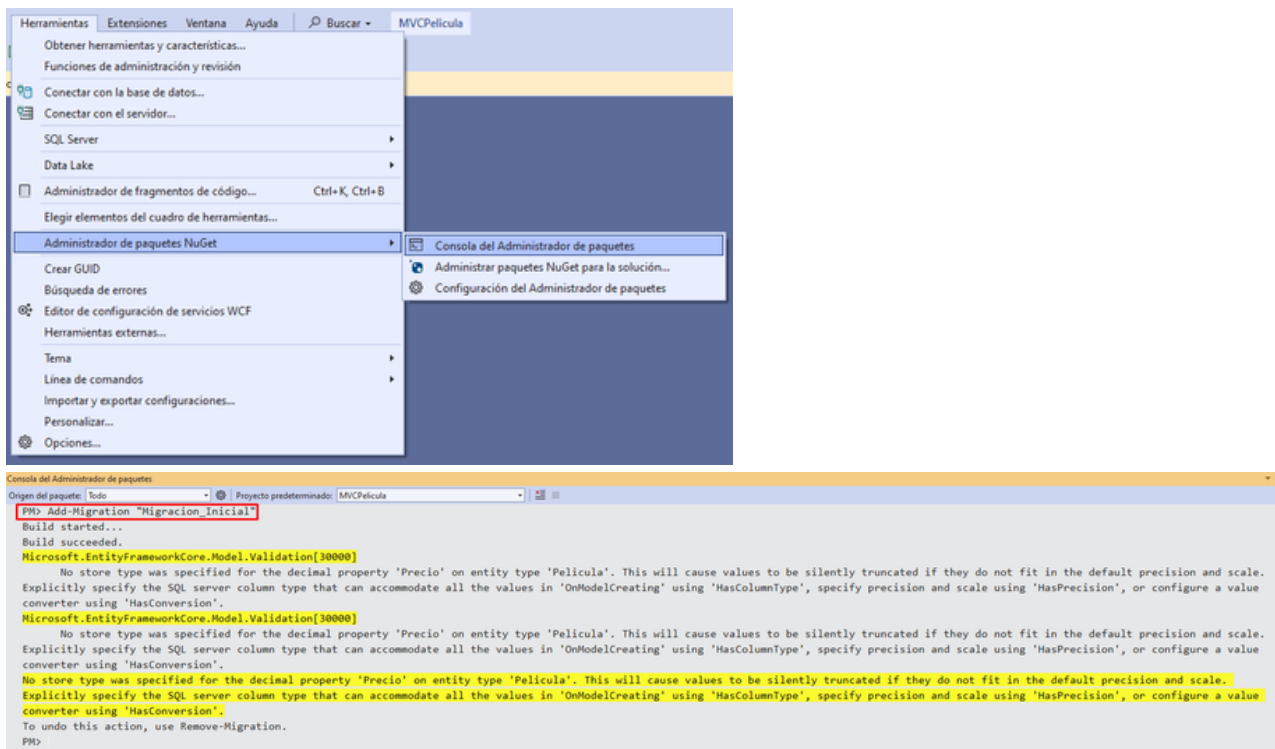


Examinar Instalado Actualizaciones Consolidar

Microsoft.EntityFrameworkCore.Tools   ☐ Incluir versión preliminar

**Microsoft.EntityFrameworkCore.Tools** por Microsoft, 319M descargas 8.0.6  
Entity Framework Core Tools for the NuGet Package Manager Console in Visual Studio.

3. Después de cualquier cambio en nuestro modelo es necesario que generemos la migración correspondiente, esto con el fin que nuestro modelo de datos corresponda con el esquema de base de datos definido. Para ellos es necesario abrir la consola del administrador de paquete NuGet y escribir el siguiente comando **Add-Migration "Migracion\_Inicial"**.



4. Una vez generada la migración, podremos ver que se generó un nuevo archivo en el directorio **Migrations** con el nombre de la migración y un timestamp. Dentro de éste archivo se distinguen dos métodos **Up()** y **Down()**, donde, el método **Up()** se encarga de aplicar los cambios y el método **Down()** contiene las instrucciones inversas para revertir los cambios aplicados.

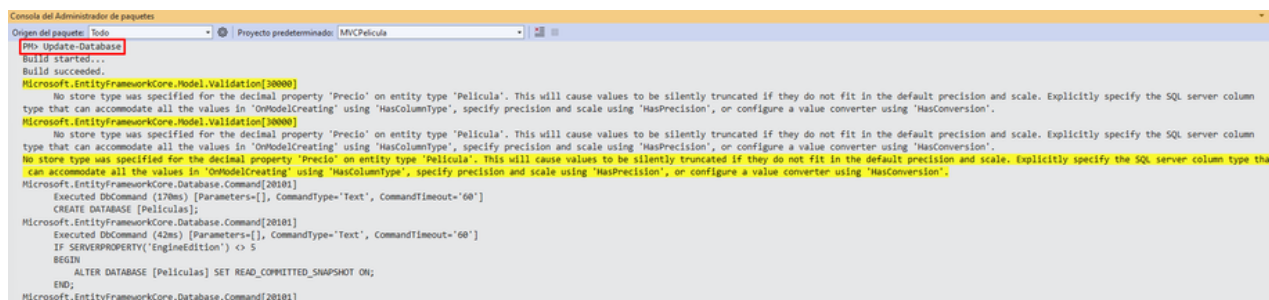
```
using System;
using Microsoft.EntityFrameworkCore.Migrations;

#nullable disable

namespace MVCPelicula.Migrations
{
    /// <inheritdoc />
    1 referencia
    public partial class Migracion_Inicial : Migration
    {
        /// <inheritdoc />
        0 referencias
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Peliculas",
                columns: table => new
                {
                    ID = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Titulo = table.Column<string>(type: "nvarchar(max)", nullable: false),
                    FechaLanzamiento = table.Column<DateTime>(type: "datetime2", nullable: false),
                    Genero = table.Column<string>(type: "nvarchar(max)", nullable: false),
                    Precio = table.Column<decimal>(type: "decimal(18,2)", nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Peliculas", x => x.ID);
                });
        }

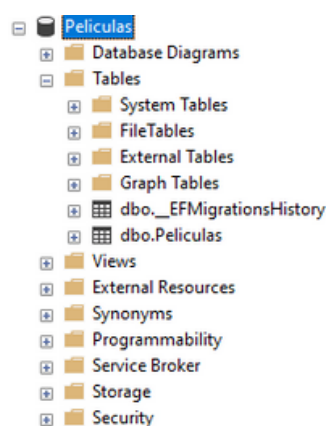
        /// <inheritdoc />
        0 referencias
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Peliculas");
        }
    }
}
```

5. Hasta este punto ningún cambio se ve reflejado en nuestra base de datos, por lo que necesitamos aplicar todas las migraciones pendientes. Para ello, utilizaremos el comando **Update-Database**.



```
Consola del Administrador de paquetes
Origen del paquete: Todo
Proyecto predeterminado: MVCPelcula
PM> Update-Database
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
No store type was specified for the decimal property 'Precio' on entity type 'Pelcula'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
No store type was specified for the decimal property 'Precio' on entity type 'Pelcula'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (170ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
CREATE DATABASE [Pelculas];
Microsoft.EntityFrameworkCore.Database.Command[20101]
Executed DbCommand (42ms) [Parameters=[], CommandType='Text', CommandTimeout='60']
IF SERVERPROPERTY('EngineEdition') <> 5
BEGIN
ALTER DATABASE [Pelculas] SET READ_COMMITTED_SNAPSHOT ON;
END;
Microsoft.EntityFrameworkCore.Database.Command[20101]
```

Se puede comprobar que las migraciones fueron aplicadas abriendo nuestro gestor de base de datos.



## Revertir o remover migraciones

Las migraciones permiten realizar cambios graduales en la base de datos, aplicando solo los cambios necesarios desde la última migración, por lo que puede surgir la necesidad de revertir un o varias migraciones, todo dependiendo a que versión del esquema a la que necesitamos llegar.

Existen dos escenarios:

1. La migración fue generada y no ha sido aplicada a la base de datos.

- En este caso solo tendremos que utilizar el comando **Remove-Migration**.
- El archivo generado de la migración será eliminado, el modelo puede ser modificado y posteriormente se generará una nueva migración.

2. La migración fue generada y ya fue aplicada a la base de datos.

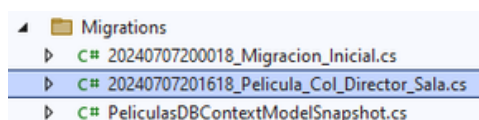
- En este caso tendremos que conocer a qué migración queremos restaurar el esquema, para ello se utiliza el comando **Get-Migration**.
- Teniendo identificada la migración a la cual queremos restaurar la base de datos utilizaremos el comando **Update-Database NombreMigracionAnterior**.
- Es importante aclarar que la lista de migraciones no se verá afectada, únicamente el estado de la base de datos.

Procederemos a cambiar el modelo agregando un campo para comprobar el funcionamiento:

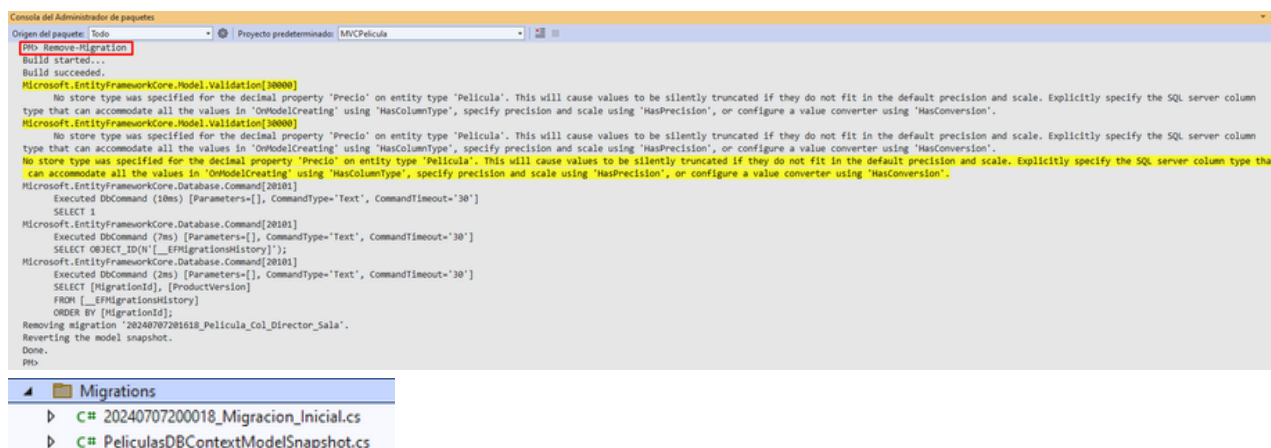
### 1. Agregaremos un nuevo campo al modelo **Pelicula**

```
namespace MVCPelicula.Models
{
    1 referencia
    public class Pelicula
    {
        0 referencias
        public int ID { get; set; }
        0 referencias
        public string Titulo { get; set; }
        0 referencias
        public DateTime FechaLanzamiento { get; set; }
        0 referencias
        public string Genero { get; set; }
        0 referencias
        public decimal Precio { get; set; }
        0 referencias
        public string Director { get; set; }
        0 referencias
        public string Sala { get; set; }
    }
}
```

### 2. Generamos la nueva migración con el comando **Add-Migration Pelicula\_Col\_Director\_Sala**.



### 3. Antes de aplicar la migración se decide que la propiedad **Sala** no corresponde al modelo **Película** por lo que se toma la decisión de remover esa migración para quitar la propiedad del modelo. Par ello, se utiliza el comando **Remove-Migration**.



### 4. Se borra la propiedad **Sala** del modelo Película y generamos nuevamente la migración con el comando **Add-Migration Pelicula\_Col\_Director**. Es así, como nuestra nueva migración solo contiene las instrucciones necesario para agregar la nueva columna Director.



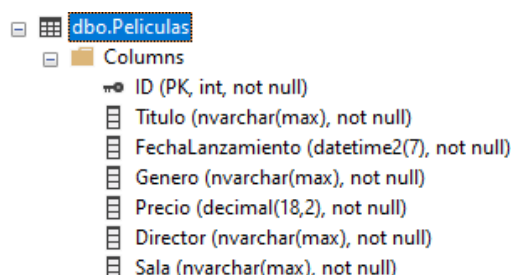
```

0 referencias
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "Director",
        table: "Peliculas",
        type: "nvarchar(max)",
        nullable: false,
        defaultValue: "");

    migrationBuilder.AddColumn<string>(
        name: "Sala",
        table: "Peliculas",
        type: "nvarchar(max)",
        nullable: false,
        defaultValue: "");
}

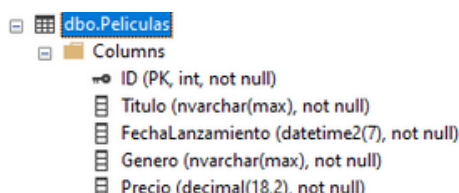
```

5. Aplicamos todas las migraciones pendientes con el comando **Update-Database**. En nuestro gestor de base de datos comprobaremos que los cambios fueron aplicados



6. Revertiremos la migración **Pelicula\_Col\_Director** con la ejecución del comando **Update-Database Migracion\_Inicial**. Esto nos permitirá devolver la base de datos al estado correspondiente a la migración inicial.

**NOTA:** es importante tomar en cuenta que al revertir migraciones aplicadas se pueden perder datos dependiente la lógica de la base de datos. Estas acciones deben realizar con un análisis previo.



## Configuración del Modelo con Atributos

Las anotaciones de datos son atributos que se aplican a las clases y propiedades de un modelo de datos para definir reglas de validación, comportamiento de la base de datos, y configuraciones de relaciones. Estas anotaciones proporcionan una forma declarativa de establecer estas configuraciones directamente en el código.

Algunas de ellas son:

- **[Key]:** Indica que la propiedad es la clave principal de la entidad, por conversión Entity Framework reconoce siempre como llave principal las propiedades con nombre Id o ID, por lo que esta anotación es requerida cuando la llave principal tenga un nombre diferente.
- **[Required]:** Especifica que una propiedad es obligatoria y no puede ser nula.

- **[StringLength]:** Define la longitud máxima y, opcionalmente, la mínima de una cadena.
- **[Column]:** Personaliza la asignación de una propiedad a una columna en la base de datos.
- **[ForeignKey]:** Define una propiedad como clave externa y establece la relación entre entidades.
- **[NotMapped]:** Indica que una propiedad no debe mapearse a ninguna columna en la base de datos. Es decir, las propiedades que estén decoradas con este atributo no serán mapeadas en migraciones y no se verán reflejadas en la base de datos.
- **[Table]:** Se aplica a la clase de la entidad para definir el nombre de la tabla y, opcionalmente, el esquema.

Para configurar un modelo se debe realizar el siguiente proceso:

1. Abrir el modelo que requiera configurar.
2. Decorar las propiedades requeridas según la lógica del negocio.

```
using System.ComponentModel.DataAnnotations;

namespace MVCPelicula.Models
{
    1 referencia
    public class Pelicula
    {
        0 referencias
        public int ID { get; set; }

        [StringLength(250)]
        [Required]
        0 referencias
        public string Titulo { get; set; }

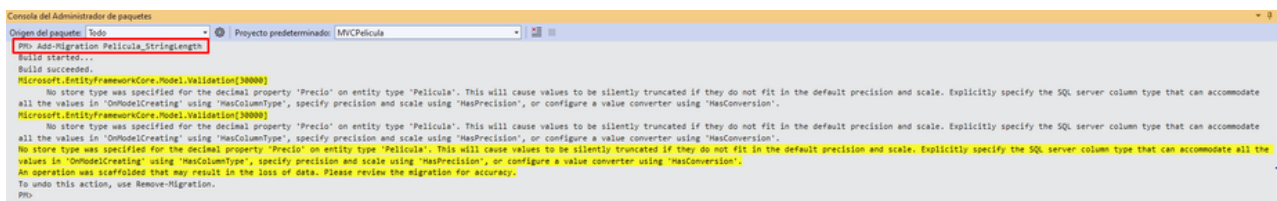
        0 referencias
        public DateTime FechaLanzamiento { get; set; }

        [StringLength(15)]
        [Required]
        0 referencias
        public string Genero { get; set; }

        [Required]
        0 referencias
        public decimal Precio { get; set; }

        [StringLength(250)]
        [Required]
        0 referencias
        public string Director { get; set; }
    }
}
```

3. Al decorar las propiedades de nuestro modelo es necesario generar nuevas migraciones para que corresponda con la base de datos.



4. Si inspeccionamos la migración generada podremos ver que las acciones que se realizarán tienen que ver con modificación de columna.

0 referencias

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropColumn(
        name: "Sala",
        table: "Películas");

    migrationBuilder.AlterColumn<string>(
        name: "Titulo",
        table: "Películas",
        type: "nvarchar(250)",
        maxLength: 250,
        nullable: false,
        oldClrType: typeof(string),
        oldType: "nvarchar(max)");

    migrationBuilder.AlterColumn<string>(
        name: "Genero",
        table: "Películas",
        type: "nvarchar(15)",
        maxLength: 15,
        nullable: false,
        oldClrType: typeof(string),
        oldType: "nvarchar(max)");

    migrationBuilder.AlterColumn<string>(
        name: "Director",
        table: "Películas",
        type: "nvarchar(250)",
        maxLength: 250,
        nullable: false,
        oldClrType: typeof(string),
        oldType: "nvarchar(max)");
}
```

0 referencias

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AlterColumn<string>(
        name: "Titulo",
        table: "Películas",
        type: "nvarchar(max)",
        nullable: false,
        oldClrType: typeof(string),
        oldType: "nvarchar(250)",
        oldMaxLength: 250);

    migrationBuilder.AlterColumn<string>(
        name: "Genero",
        table: "Películas",
        type: "nvarchar(max)",
        nullable: false,
        oldClrType: typeof(string),
        oldType: "nvarchar(15)",
        oldMaxLength: 15);

    migrationBuilder.AlterColumn<string>(
        name: "Director",
        table: "Películas",
        type: "nvarchar(max)",
        nullable: false,
        oldClrType: typeof(string),
        oldType: "nvarchar(250)",
        oldMaxLength: 250);

    migrationBuilder.AddColumn<string>(
        name: "Sala",
        table: "Películas",
        type: "nvarchar(max)",
        nullable: false,
        defaultValue: "");
}
```

5. Al aplicar la migración podremos ver los cambios reflejados en la base de datos.

dbo.Peliculas	
Columns	
ID (PK, int, not null)	
Titulo (nvarchar(250), not null)	
FechaLanzamiento (datetime2(7), not null)	
Genero (nvarchar(15), not null)	
Precio (decimal(18,2), not null)	
Director (nvarchar(250), not null)	

## Llaves foráneas en Entity Framework

Entity Framework usa convenciones para inferir las relaciones basadas en los nombres de las propiedades. Además, si los nombres de las propiedades por alguna razón necesitan ser diferentes a las convenciones establecidas se puede utilizar **Data Annotations** o la **Fluent API** para establecer las relaciones entre modelos.

### Usando convenciones

Para que Entity Framework infiera que en nuestro modelos existe relaciones es necesario especificar la propiedad que servirá como llave foránea de la siguiente manera:

**public <Tipo de dato de la clave principal> <nombre del modelo> Id { get; set; }**

Además es necesario agregar una propiedad adicional que servirá para obtener los registros relacionadas. Esta propiedad se le conoce como **propiedad de navegación**.

1. Proceda a crear una nueva clase dentro de la carpeta **Models**, llamada **Genero** y agregue lo siguiente:

```
namespace MVCPelicula.Models
{
    0 referencias
    public class Genero
    {
        0 referencias
        public int Id { get; set; }
        0 referencias
        public string Nombre { get; set; }

        //Propiedad de navegación
        0 referencias
        public ICollection<Pelicula> Peliculas { get; set; }
    }
}
```

2. Modifique el archivo **PeliculasDbContext** y agregue la referencia al modelo de **Genero** que acaba de crear.

```
using Microsoft.EntityFrameworkCore;

namespace MVCPelicula.Models
{
    6 referencias
    public class PeliculasDbContext: DbContext
    {
        0 referencias
        public PeliculasDbContext(DbContextOptions options) : base(options)
        {
        }

        0 referencias
        public DbSet<Pelicula> Peliculas { get; set; }

        0 referencias
        public DbSet<Genero> Generos { get; set; }
    }
}
```

## Usando Data Annotations

Se utiliza el atributo **[ForeignKey]** para especificar la llave foránea explícitamente.

### 1. Realice las siguientes modificaciones al modelo **Pelicula**

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace MVCPelicula.Models
{
    2 referencias
    public class Pelicula
    {
        0 referencias
        public int ID { get; set; }

        [StringLength(250)]
        [Required]
        0 referencias
        public string Titulo { get; set; }

        0 referencias
        public DateTime FechaLanzamiento { get; set; }

        //Propiedad para la llave foránea
        [Required]
        [ForeignKey("Genero")]
        0 referencias
        public int? GeneroId { get; set; }

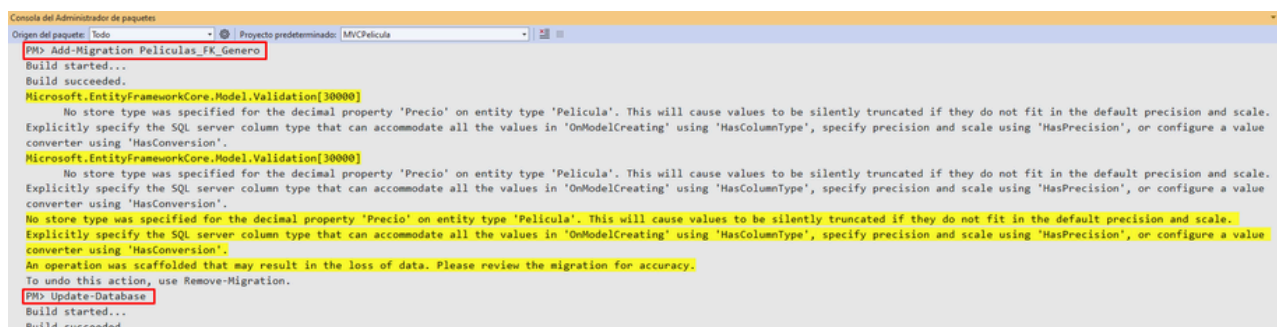
        //Propiedad de navegación
        0 referencias
        public Genero Genero { get; set; }

        [Required]
        0 referencias
        public decimal Precio { get; set; }

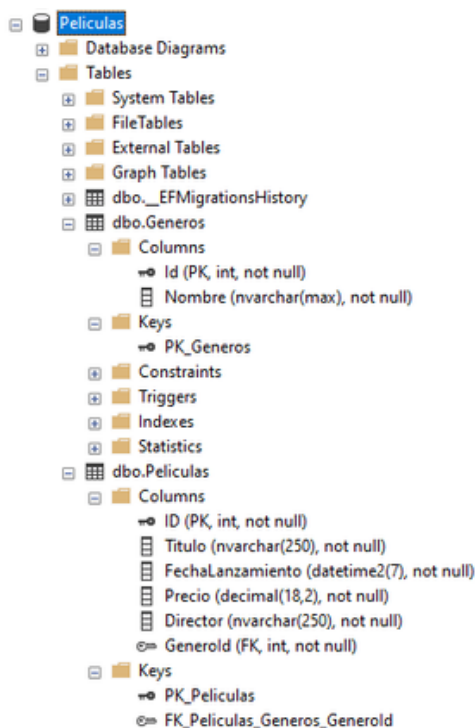
        [StringLength(250)]
        [Required]
        0 referencias
        public string Director { get; set; }
    }
}
```

**NOTA:** en el modelo que esta relacionado, en este caso **Genero**, se puede agregar otra propiedad de navegación inversa. La cual nos permite obtener, para este ejemplo, todas las películas relacionadas a cada genero.

### 2. Agregamos y aplicamos la respectiva migración para que las relaciones se vean reflejadas en la base de datos.



```
Consola del Administrador de paquetes
Origen del paquete: Todo | Proyecto predeterminado: MVCPelicula
PM> Add-Migration Peliculas_FK_Genero
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
No store type was specified for the decimal property 'Precio' on entity type 'Pelicula'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
Microsoft.EntityFrameworkCore.Model.Validation[30000]
No store type was specified for the decimal property 'Precio' on entity type 'Pelicula'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
```

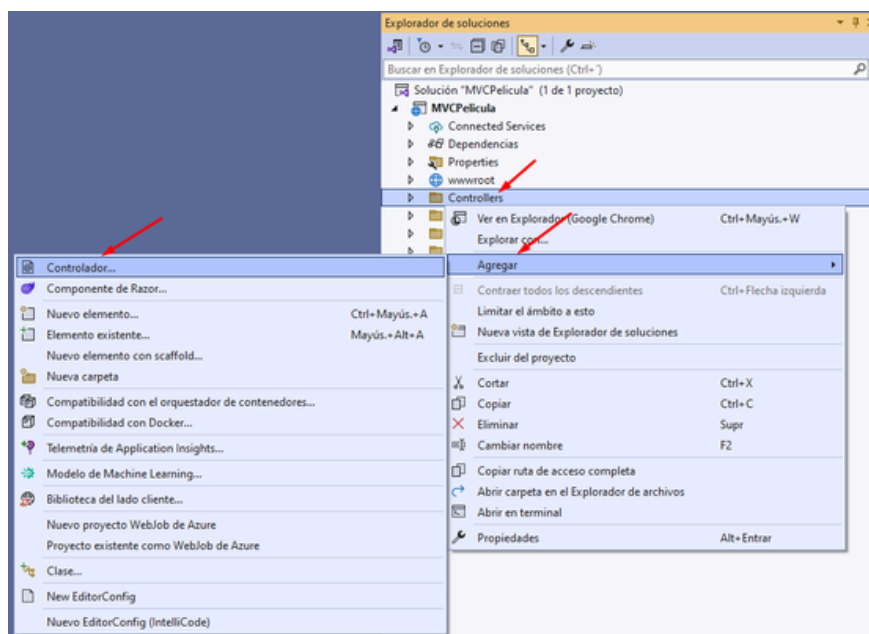


## Generando un CRUD

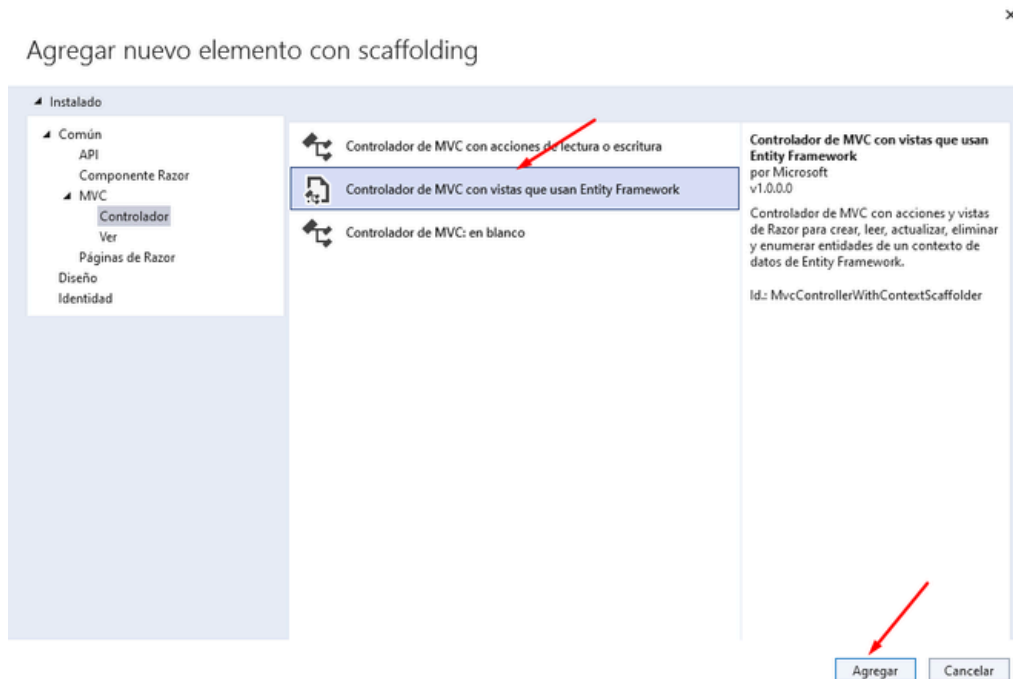
Usar scaffolding en combinación con Entity Framework facilita la creación de una estructura básica para las operaciones CRUD (Create, Read, Update, Delete) en una aplicación ASP.NET Core. El scaffolding genera automáticamente código para las operaciones CRUD y proporciona una base sobre la cual puedes construir funcionalidades adicionales.

El proceso para generar un CRUD es el siguiente:

1. En el explorador de soluciones tenemos que dar click derecho sobre el directorio **Controllers**, luego en **Agregar -> Controlador**.



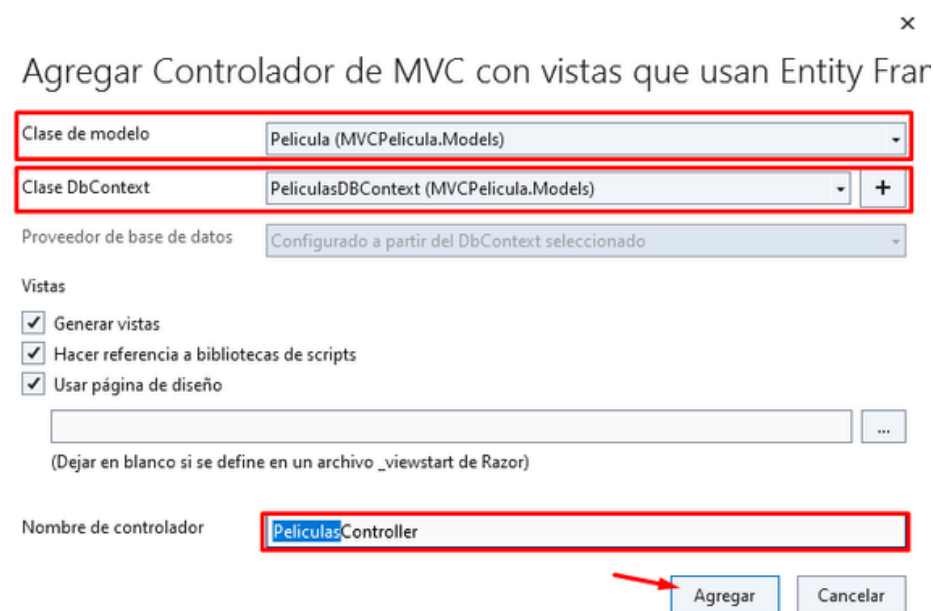
2. Del cuadro de dialogo seleccionamos **Controlador MVC con vistas que usan Entity Framework**.



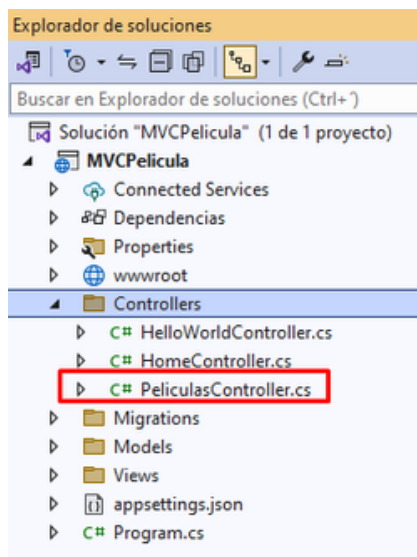
4. Luego se seleccionar el tipo de controlador que queremos generar, se muestra un cuadro de dialogo donde tenemos que especificar:

- **Clase de modelo:** Pelicula (MVCPelicula.Models)
- **Clase Db Context:** PeliculasDbContext (MVCPeliculas.Models)
- **Si el controlador se van a generar las vistas o no:** para este caso seleccionaremos que **sí**
- **Si se desea utilizar una clase Layout distinta a la predeterminada.**
- **El nombre del controlador**, por defecto se genera el controlador con el nombre del modelo en plural (esta opción puede ser desactivada por medio de codigo).

Con esto crearemos el CRUD del modelo **Pelicula**.



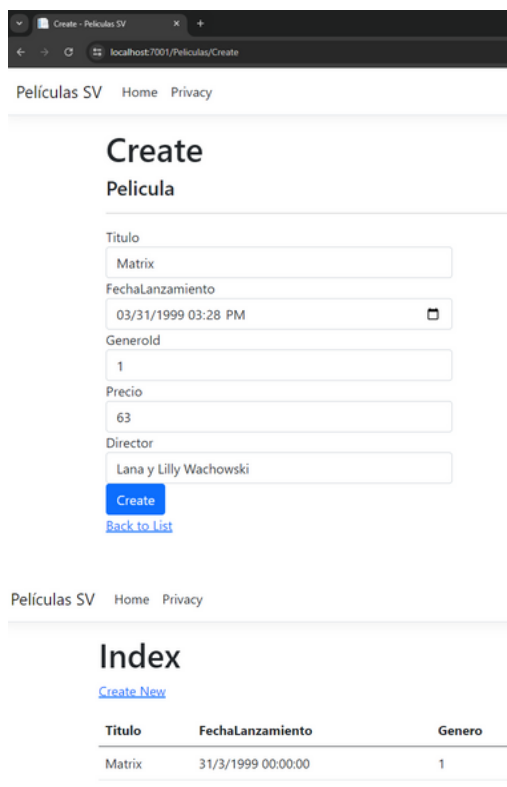
5. Dar click en agregar y se creará automáticamente todo el código del controlador y las vistas. Cuando éste proceso finalice en el explorador de soluciones podremos ver los siguientes archivos.



6. El código del controlador y el diseño de vistas puede ser modificados a voluntad. Esto con el fin de adecuar según las necesidades y lógica especial que se desea aplicar.

7. Para comprobar el funcionamiento del controlador podemos acceder directamente a la ruta del mismo localhost:XXXX/Peliculas o modificando la vista **\_Layout.cshtml** para incluir el enlace.

**Nota:** antes de agregar una nueva película, agregue un registro de género en la tabla, ya que de lo contrario no podrá crear registros de películas.





Nota: si al probar al CRUD de películas, le genera un error o no funciona correctamente. Esto se debe a que se está validando la propiedad de navegación **Genero**. Esto tiene muchas formas de solucionarse, pero por el momento simplemente eliminaremos dicha validación. Para hacer esto, agregue la instrucción "ModelState.Remove("Genero");" al inicio de los métodos HttpPost Create y HttpPost Edit.

```
// POST: Peliculas/Create
// To protect from overposting attacks, enable the specific properties you want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
0 referencias
public async Task<IActionResult> Create([Bind("ID,Titulo,FechaLanzamiento,GeneroId,Precio,Director")] Pelicula pelicula)
{
    //Removemos la validación de la propiedad de navegación Genero
    ModelState.Remove("Genero");

    // POST: Peliculas/Edit/5
    // To protect from overposting attacks, enable the specific properties you want to bind to.
    // For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
    [HttpPost]
    [ValidateAntiForgeryToken]
0 referencias
public async Task<IActionResult> Edit(int id, [Bind("ID,Titulo,FechaLanzamiento,GeneroId,Precio,Director")] Pelicula pelicula)
{
    //Removemos la validación de la propiedad de navegación Genero
    ModelState.Remove("Genero");
```

## Desarrollo de Habilidades

### Ejercicio 1

- Complete la aplicación generando el código necesario para tener el CRUD de los géneros de películas.
- Realice las modificaciones necesarias en la vistas de Películas, para que al momento de agregar y/o modificar un registro, aparezca el nombre del género en vez de su Id.

### Ejercicio 2

Se le solicita que utilizando Code First diseñar una base de datos utilizando Entity Framework que permita gestionar empleados, proyectos y las asignaciones de proyectos a empleados. El diseño debe incluir tres tablas relacionadas:

- Tabla Empleados: Contendrá información de los empleados, como el nombre, el apellido, la fecha de contratación y el puesto.
- Tabla Proyectos: Contendrá información de los proyectos, incluyendo el nombre del proyecto, la descripción y la fecha de inicio.
- Tabla Asignaciones: Registrará la asignación de empleados a proyectos, especificando la fecha de asignación y un rol específico que desempeña el empleado en el proyecto.

# Bibliografía

BRADFORD, M.; VALVERDE, R. & TALLA, M. (2010). Modern ERP: Select, Implement & Use Today's Advanced Business Systems. North Carolina, State University: Universidad de Carolina del Norte.

VALVERDE, R. (2012). Information Systems Reengineering for Modern Business Systems. IGI Global, USA.

SHIELDS, M. (2001). E-Business and ERP Rapid Implementation and Project.