**40. ¿Qué es Entity Framework?**

**1. Concepto General**

* **Entity Framework (EF)** es un **ORM (Object Relational Mapping)**.
* Es el **ORM por defecto de .NET** y el más recomendado para trabajar con bases de datos dentro de este ecosistema.
* Permite mapear (relacionar) una base de datos relacional en objetos del lenguaje de programación (C#).
* **Idea principal**: una **tabla** en la base de datos se representa como una **clase (modelo)** en el código.
  + Cada fila de la tabla se transforma en un **objeto**.
  + Cada columna de la tabla se transforma en una **propiedad** de la clase.

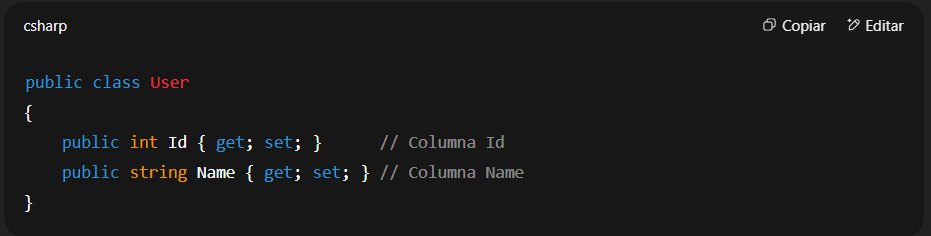
**2. ¿Qué es un ORM?**

* **Object Relational Mapping (ORM)** significa "Mapeo Objeto-Relacional".
* Su función es traducir las estructuras de una base de datos relacional (tablas, columnas, relaciones) en objetos y clases del lenguaje de programación.
* Ejemplo:
  + Tabla Users → Clase User
  + Tabla Roles → Clase Role
  + Relación Users.RoleId → Roles.Id → Se mapea en una **propiedad de navegación** (User.Role).

Con esto, en vez de escribir **consultas SQL manuales** todo el tiempo, el desarrollador trabaja directamente con objetos y colecciones.

**3. Modelos en Entity Framework**

* Un **modelo** es una clase que representa una tabla.
* Contiene:
  + **Propiedades** que equivalen a las columnas.
  + **Tipos de datos** equivalentes (ejemplo: string ↔ VARCHAR, int ↔ INT, DateTime ↔ DATETIME).
* Ejemplo simple:



**4. Enfoques de trabajo con Entity Framework**

Existen **tres formas principales** de usar EF:

**🔹 1. Code First**

* Se escriben primero las **clases (modelos)** en C#.
* Luego, mediante comandos de EF, se generan automáticamente las **tablas en la base de datos**.
* Ideal cuando:
  + No existe una base de datos previa.
  + Se quiere diseñar la BD desde el código.
* Proceso:
  + Crear clases modelo.
  + Ejecutar comandos de migración (Add-Migration, Update-Database).
  + EF crea las tablas en la BD.

**🔹 2. Database First**

* Se parte de una base de datos ya existente.
* EF genera automáticamente las **clases modelo** a partir de las tablas.
* Ideal cuando:
  + La base de datos ya existe.
  + Hay un equipo dedicado exclusivamente a la BD.
* Proceso:
  + Conectar EF a la BD.
  + Ejecutar un "scaffolding" para generar las clases a partir de las tablas.

**🔹 3. Manual (Clases + BD a mano)**

* El programador crea **las tablas en SQL** manualmente y **las clases en C#** manualmente.
* Solo debe asegurarse de que ambas estén sincronizadas.
* Se usa cuando:
  + Se quiere máximo control.
  + Se desconfía de los comandos automáticos (aunque es raro que fallen).
  + Se trabaja en equipos grandes con roles separados.

**5. Ventajas de Entity Framework**

* Permite **abstraerse del SQL** (no es necesario escribir consultas SQL en la mayoría de los casos).
* Reduce código repetitivo de acceso a datos.
* Facilita el mantenimiento y la evolución del sistema.
* Soporta múltiples motores de base de datos (SQL Server, MySQL, MariaDB, PostgreSQL, etc.).

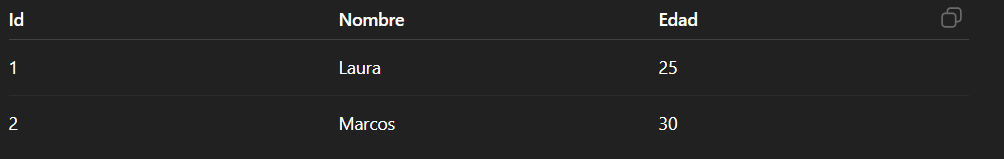
**6. Resumen de Opciones**

1. **Code First** → Creas primero las clases, EF genera la BD.
2. **Database First** → Creas primero la BD, EF genera las clases.
3. **Manual** → Creas ambas a mano, asegurando que coincidan.

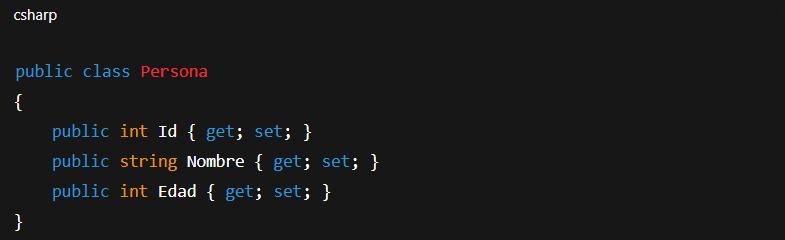
En el curso se trabajará con **Code First**.

**7. Ejemplo Visual**

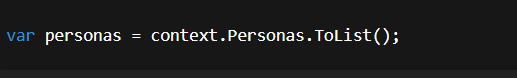
Imagina que tienes la tabla:



Esto se representa en EF como:



Y al consultar en C#:



Obtendrás una lista de objetos Persona, en lugar de trabajar con filas de SQL.

📌 **Conclusión:**  
Entity Framework es una herramienta clave en .NET que simplifica el trabajo con bases de datos gracias al ORM. Permite elegir entre **Code First**, **Database First** o un enfoque manual, dependiendo del proyecto y el equipo. En este curso, se usará principalmente **Code First** para aprender desde la creación de clases hasta su reflejo en la base de datos.

**41. Instalación de Entity Framework**

**1. Características Generales**

* **Entity Framework** es un paquete **gratuito y open source**.
* No viene incluido directamente en los proyectos de .NET, pero se puede instalar fácilmente a través de **NuGet**.
* Para trabajar en el curso, se usará **Entity Framework Core (EF Core)**, que es la versión moderna y optimizada del framework.

**2. ¿Qué es NuGet?**

* **NuGet** es el **administrador de paquetes** de .NET.
* Permite agregar librerías externas (como Entity Framework, JWT, AutoMapper, etc.) de forma rápida y sencilla.
* Los paquetes instalados se listan en la carpeta **Dependencias → Paquetes** dentro del proyecto.

**3. Instalación de Entity Framework en .NET**

**🔹 Paso 1: Instalar el proveedor de base de datos**

Como se usará **SQL Server**, se debe instalar el paquete correspondiente:

* Nombre del paquete:  
  Microsoft.EntityFrameworkCore.SqlServer
* Este paquete permite que EF Core pueda comunicarse con SQL Server.

**Procedimiento:**

1. Clic derecho sobre **Dependencias** en el proyecto.
2. Seleccionar **Administrar paquetes NuGet**.
3. Ir a la pestaña **Examinar**.
4. Buscar Entity Framework Core SqlServer.
5. Seleccionar el paquete Microsoft.EntityFrameworkCore.SqlServer.
6. Hacer clic en **Instalar** y aceptar las condiciones.

**🔹 Paso 2: Instalar las herramientas para Code First**

Como se trabajará con el enfoque **Code First** (crear clases y luego generar tablas en la BD), se necesitan las **herramientas de migración**:

* Nombre del paquete:  
  Microsoft.EntityFrameworkCore.Tools
* Este paquete permite ejecutar comandos como:
  + Add-Migration → Crear migraciones.
  + Update-Database → Actualizar la base de datos a partir de las migraciones.
  + Remove-Migration → Eliminar la última migración creada.

**Procedimiento:**

1. En **NuGet**, buscar Entity Framework Core Tools.
2. Seleccionar el paquete Microsoft.EntityFrameworkCore.Tools.
3. Instalarlo y aceptar las condiciones.

**4. Verificación de instalación**

* Una vez instalados, en el explorador de soluciones → **Dependencias → Paquetes** deberían aparecer:
  + Microsoft.EntityFrameworkCore.SqlServer
  + Microsoft.EntityFrameworkCore.Tools

Esto confirma que el proyecto ya está listo para usar EF Core con SQL Server.

**5. Alternativa: Instalación por Consola**

También se puede instalar con la **Consola del Administrador de Paquetes (Package Manager Console)**:

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Install-Package Microsoft.EntityFrameworkCore.Tools

O mediante **.NET CLI** (si no se quiere usar Visual Studio):

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

dotnet add package Microsoft.EntityFrameworkCore.Tools

**6. Resumen**

* **EF Core** no viene por defecto, hay que instalarlo.
* Para trabajar con **SQL Server + Code First** se requieren **dos paquetes básicos**:
  1. Microsoft.EntityFrameworkCore.SqlServer → conexión a SQL Server.
  2. Microsoft.EntityFrameworkCore.Tools → comandos de migración.
* Se pueden instalar:
  1. Visualmente desde NuGet.
  2. O mediante comandos (Install-Package o dotnet add package).

📌 **Conclusión:**  
La instalación de EF Core es sencilla y flexible. Basta con instalar los dos paquetes principales: **SqlServer** (proveedor de BD) y **Tools** (herramientas de migración). Con esto el proyecto ya está preparado para trabajar bajo el enfoque **Code First**.

**42. Creación de Modelos**

Excelente, sigamos con el apunte para este tercer video ✅.

**Apunte: Creación de Modelos en Entity Framework**

**1. ¿Qué es un modelo?**

* Un **modelo** es una **clase en C#** que representa una **tabla en la base de datos**.
* Dependiendo del autor, se le puede llamar:
  + **Modelo**
  + **Entidad (Entity)**
  + **Clase de dominio**
* La idea es la misma: un modelo refleja la estructura de la tabla para trabajar con ella desde código.

**2. Objetivo de los modelos**

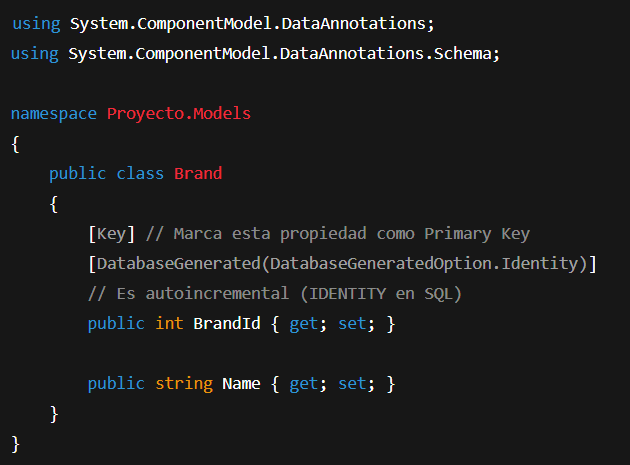
* Permiten manipular los datos de la base de datos desde C# **como si fueran objetos**.
* Con EF se pueden hacer operaciones como:
  + **Insertar**
  + **Modificar**
  + **Eliminar**
  + **Consultar**
* Todo esto con **LINQ**, lo cual brinda mucha flexibilidad sin necesidad de escribir SQL manual.

**3. Organización del proyecto**

* Se recomienda crear una carpeta llamada **Models** (plural por convención, pero puede ser singular).
* Dentro de esta carpeta se agregan las clases que representan cada tabla.

**4. Ejemplo práctico: Modelo Brand**

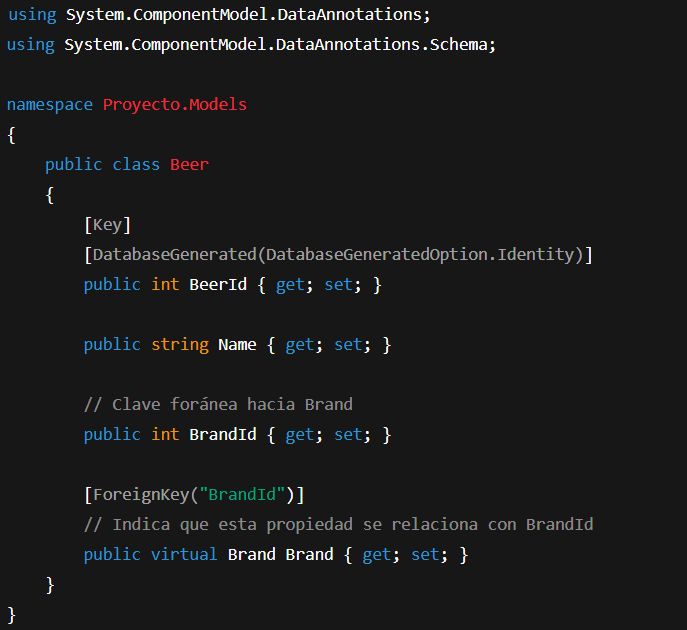
Creamos una clase Brand para representar las marcas de cerveza.

**Explicación:**

* [Key]: Define la propiedad como **Primary Key**.
* [DatabaseGenerated(DatabaseGeneratedOption.Identity)]: Indica que es **autoincrementable**.
* BrandId: clave primaria de la tabla.
* Name: nombre de la marca.

**5. Ejemplo práctico: Modelo Beer**

Creamos una clase Beer para representar las cervezas.

**Explicación:**

* BeerId: identificador único para cada cerveza (**Primary Key**).
* Name: nombre de la cerveza.
* BrandId: **Foreign Key** que conecta la cerveza con una marca.
* [ForeignKey("BrandId")]: establece explícitamente que Beer.Brand se relaciona con Beer.BrandId.
* public virtual Brand Brand: propiedad de **navegación** que permite acceder al objeto relacionado.

**6. Relaciones en Entity Framework**

* **Relación uno a muchos (1:N)**:
  + Una marca puede tener muchas cervezas.
  + Una cerveza pertenece a una sola marca.
* En términos de EF:
  + Beer tiene un campo BrandId como **FK**.
  + Beer tiene una propiedad de navegación Brand.
  + (Opcionalmente, en Brand se puede agregar una lista de Beers para la relación inversa).

Ejemplo en Brand:



**7. Resumen**

1. Un **modelo** representa una tabla de la BD como clase.
2. Se usan **atributos** ([Key], [DatabaseGenerated], [ForeignKey]) para definir:
   * Primary Keys
   * Autoincrementales
   * Relaciones entre tablas
3. Ejemplo:
   * Brand ↔ tabla **Brand**
   * Beer ↔ tabla **Beer**, con FK hacia Brand.

📌 **Conclusión:**  
Con los modelos definidos (Brand y Beer), ya tenemos la representación en C# de nuestras tablas de base de datos. Esto será la base para trabajar con EF bajo el enfoque **Code First**, donde a partir de estas clases se generarán las tablas y sus relaciones.

**43. Creación de Contexto**

**1. ¿Qué es el Context?**

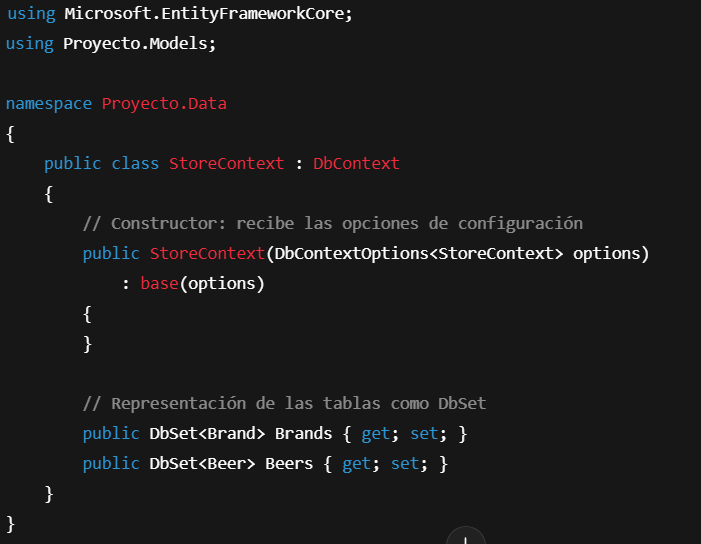
* El **DbContext** es el **núcleo de Entity Framework**.
* Funciona como un **intermediario** entre:
  + Tus **clases/modelos** (Brand, Beer, etc.)
  + Y la **base de datos real** (SQL Server, MySQL, etc.).

**Funciones principales:**

1. Mantiene la **cadena de conexión** (cómo se conecta a la BD).
2. Indica **qué entidades (tablas)** forman parte de la base de datos.
3. Administra operaciones como:
   * Crear tablas.
   * Ejecutar consultas.
   * Guardar cambios (inserts, updates, deletes).

**2. Creación del Contexto**

Se crea una clase que **hereda** de DbContext.  
Ejemplo del curso: StoreContext.



**3. Explicación paso a paso**

**🔹 Herencia de DbContext**



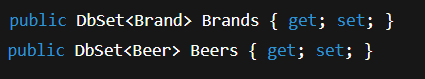
* Indica que StoreContext será un **contexto de EF**.

**🔹 Constructor con DbContextOptions**



* Permite pasar la configuración del contexto (cadena de conexión, proveedor de BD, etc.).
* El base(options) se encarga de entregárselo a la clase padre (DbContext).

**🔹 DbSet**

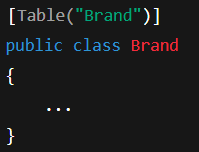


* Cada DbSet<T> representa una **tabla en la base de datos**:
  + DbSet<Brand> → tabla **Brands**
  + DbSet<Beer> → tabla **Beers**
* Los DbSet permiten:
  + Consultar datos (context.Brands.ToList()).
  + Agregar registros (context.Brands.Add(new Brand {...})).
  + Modificar y eliminar registros.

**4. Nombres de tablas y pluralización**

* Por defecto, EF genera las tablas en **plural**:
  + Clase Brand → Tabla Brands.
  + Clase Beer → Tabla Beers.
* Esto se debe a la convención de EF.
* Si se quiere forzar un nombre específico (ej. singular), se puede configurar con **Data Annotations** o **Fluent API**.

Ejemplo con Data Annotations:



**5. Resumen**

* El **Context** es el corazón de EF: conecta tus modelos con la base de datos.
* Se crea una clase que hereda de DbContext.
* Se definen los **DbSet** para cada modelo (cada tabla).
* El **constructor** recibe DbContextOptions y se los pasa al padre.
* Por defecto, EF pluraliza los nombres de las tablas, aunque puede configurarse.

📌 **Conclusión:**  
El **StoreContext** será el punto central para interactuar con la base de datos en Entity Framework. A través de él se definen las tablas (DbSet), se ejecutan consultas y se aplican cambios. En el curso, este contexto será la base sobre la cual se construirá el acceso a datos mediante el enfoque **Code First**.

**44. Inyección de Contexto**

**Apunte: Inyección de Contexto en Entity Framework**

**1. ¿Qué significa inyectar el contexto?**

* **El DbContext (ej. StoreContext) debe ser accesible en controladores y servicios para interactuar con la base de datos.**
* **Esto se hace mediante Inyección de Dependencias (DI).**
* **En ASP.NET Core, se configura en Program.cs usando el contenedor de servicios (builder.Services).**

**2. Configuración de la cadena de conexión**

**La cadena de conexión indica cómo conectarse a la base de datos.  
Se agrega en el archivo appsettings.json.**

**Ejemplo:**

**{**

**"ConnectionStrings": {**

**"StoreConnection": "Server=HECTORLAB\\SQLEXPRESS;Database=Store;Trusted\_Connection=True;TrustServerCertificate=True;"**

**},**

**"Logging": {**

**"LogLevel": {**

**"Default": "Information"**

**}**

**},**

**"AllowedHosts": "\*"**

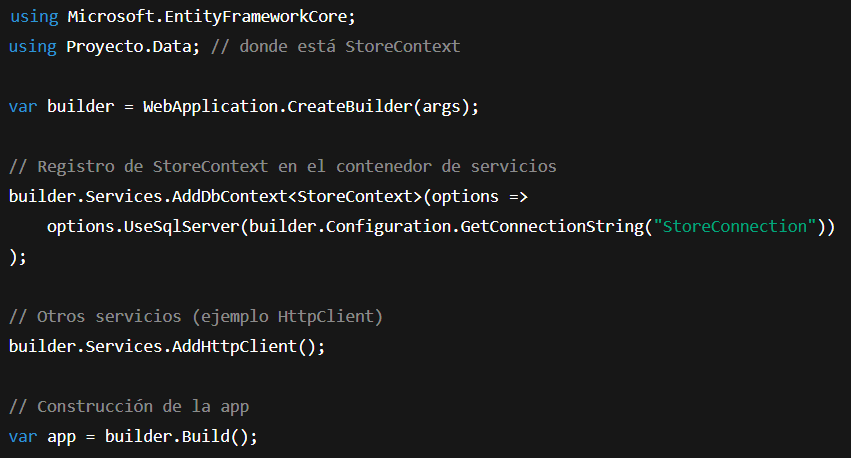
**}**

**Componentes:**

* **Server → Nombre del servidor de SQL Server (HECTORLAB\SQLEXPRESS en este caso).**
* **Database → Nombre de la base (Store).**
* **Trusted\_Connection=True → Usa autenticación de Windows (si no, se colocan User Id=...;Password=...;).**
* **TrustServerCertificate=True → Evita errores de certificado en entornos de desarrollo.**

**3. Registro del Contexto en Program.cs**

**Para que el contexto esté disponible en controladores/servicios, se debe inyectar con AddDbContext.**

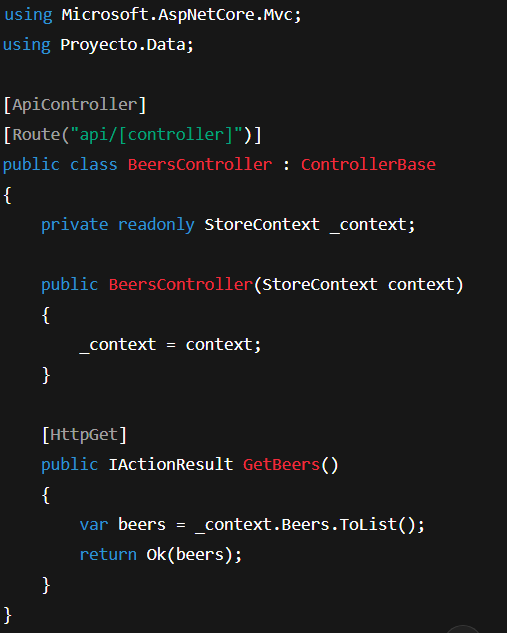
****

**4. Explicación paso a paso**

1. **builder.Services.AddDbContext<StoreContext>(...)**
   * **Registra el StoreContext como servicio disponible para inyección.**
   * **Cada vez que un controlador lo necesite, el sistema lo entrega automáticamente.**
2. **options.UseSqlServer(...)**
   * **Indica que se usará SQL Server como base de datos.**
   * **Usa la cadena de conexión definida en appsettings.json.**
3. **builder.Configuration.GetConnectionString("StoreConnection")**
   * **Obtiene la cadena de conexión definida en la sección "ConnectionStrings" con la clave "StoreConnection".**

**5. Uso en controladores/servicios**

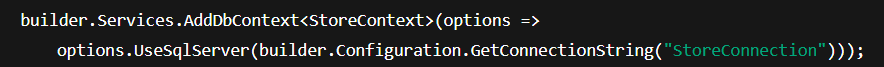
**Una vez registrado, se puede inyectar el StoreContext directamente en constructores:**

**Explicación:**

* **El controlador pide un StoreContext.**
* **ASP.NET Core lo entrega automáticamente gracias a la inyección configurada en Program.cs.**

**6. Resumen**

1. **Cadena de conexión se define en appsettings.json bajo ConnectionStrings.**
2. **En Program.cs, se inyecta el contexto con:**

****

1. **Los controladores/servicios pueden recibir el StoreContext por Inyección de Dependencias.**
2. **Esto permite trabajar con la base de datos desde cualquier parte del proyecto de manera centralizada y segura.**

**📌 Conclusión:  
La inyección del contexto es el paso clave que conecta la aplicación con la base de datos. Gracias a AddDbContext, Entity Framework puede usar la cadena de conexión definida en appsettings.json y entregar el StoreContext en controladores y servicios sin necesidad de instanciarlo manualmente.**

**45. Migración Inicial**

**1. ¿Qué son las migraciones?**

* **Una migración es un cambio controlado en la base de datos, generado a partir de tus modelos de C#.**
* **Sirve para versionar la base de datos, llevando un historial de:**
  + **Nuevas tablas creadas.**
  + **Campos agregados o eliminados.**
  + **Relaciones nuevas o modificadas.**
* **También permite revertir cambios (rollback).**

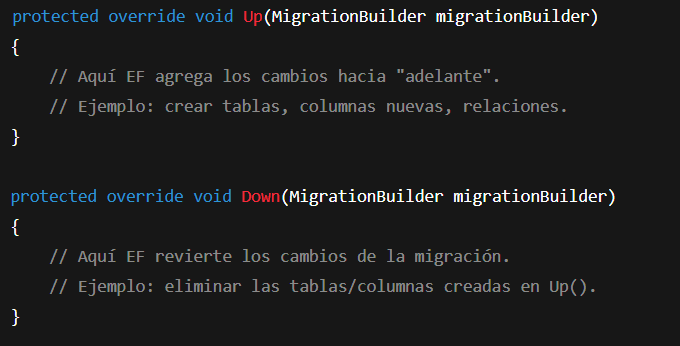
**👉 En resumen: Las migraciones son el “Git de tu base de datos”.**

**2. ¿Por qué usarlas?**

* **Ventajas:**
  + **Permiten que todos en un equipo trabajen con la misma versión de la BD.**
  + **Guardan el historial de cambios.**
  + **Se pueden revertir cambios si algo salió mal.**
* **Alternativa sin migraciones:**
  + **Podrías crear la base directamente con SQL y mantenerla sincronizada manualmente con tus clases.**
  + **Esto funciona, pero pierdes el versionado y el control automático de cambios.**

**3. ¿Cómo funcionan?**

**Cuando creas una migración, EF genera una clase con dos métodos principales:**

****

**¿se ejecutan siempre?**

* **Up → Sí, siempre que aplicás la migración con Update-Database.**
* **Down → No, solo si vos decidís revertir una migración.**

**Ejemplo inicial (initdb)**

**Si tienes los modelos Brand y Beer, EF genera algo parecido a esto:**

**protected override void Up(MigrationBuilder migrationBuilder)**

**{**

**migrationBuilder.CreateTable(**

**name: "Brands",**

**columns: table => new**

**{**

**BrandId = table.Column<int>(nullable: false)**

**.Annotation("SqlServer:Identity", "1, 1"),**

**Name = table.Column<string>(nullable: true)**

**},**

**constraints: table =>**

**{**

**table.PrimaryKey("PK\_Brands", x => x.BrandId);**

**});**

**migrationBuilder.CreateTable(**

**name: "Beers",**

**columns: table => new**

**{**

**BeerId = table.Column<int>(nullable: false)**

**.Annotation("SqlServer:Identity", "1, 1"),**

**Name = table.Column<string>(nullable: true),**

**BrandId = table.Column<int>(nullable: false)**

**},**

**constraints: table =>**

**{**

**table.PrimaryKey("PK\_Beers", x => x.BeerId);**

**table.ForeignKey(**

**name: "FK\_Beers\_Brands\_BrandId",**

**column: x => x.BrandId,**

**principalTable: "Brands",**

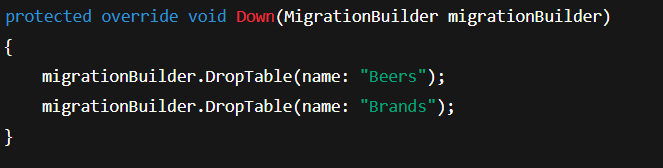
**principalColumn: "BrandId",**

**onDelete: ReferentialAction.Cascade);**

**});**

**}**

**Y el Down sería el inverso:**

****

**4. Flujo de trabajo con migraciones**

**Paso 1: Crear una migración**

**Desde la Consola del Administrador de Paquetes (Visual Studio):**

****

* **initdb es el nombre de la migración.**
* **EF genera una nueva clase dentro de la carpeta Migrations.**

**Paso 2: Aplicar la migración a la base**

**Ejecuta:**

****

* **EF toma la migración y ejecuta el SQL equivalente en la base de datos.**
* **Resultado: se crean las tablas (Brands, Beers, etc.).**

**Paso 3: Hacer cambios en los modelos**

**Ejemplo: agregamos un nuevo campo en Beer:**

****

**Paso 4: Nueva migración**

****

* **EF genera otra clase con un migrationBuilder.AddColumn(...).**

**Paso 5: Aplicar la nueva migración**

****

* **La base se actualiza y ahora la tabla Beers incluye la columna AlcoholPercentage.**

**5. Resumen de comandos importantes**

* **Crear migración inicial:**

**Add-Migration initdb**

* **Aplicar migración a la base:**

**Update-Database**

* **Revertir la última migración aplicada:**

**Remove-Migration**

* **Revertir la BD a una migración específica:**

**Update-Database NombreMigracion**

**6. Advertencia**

* **En desarrollo, las migraciones son muy útiles porque podés probar, revertir, volver a aplicar.**
* **En producción, hay que ser muy cuidadoso, porque un Down puede borrar datos reales.**
* **Lo común en producción es:**
  + **Generar los scripts SQL de las migraciones.**
  + **Revisarlos antes de aplicarlos en la base real.**

**7. Resumen final**

* **Las Migraciones permiten a EF llevar un control de cambios versionados de la base de datos.**
* **Funcionan mediante clases con métodos Up (aplicar cambios) y Down (revertir cambios).**
* **Se usan con comandos (Add-Migration, Update-Database) y se guardan en la carpeta Migrations.**
* **Son ideales en entornos de desarrollo y trabajo en equipo.**

**📌 Conclusión:  
Las migraciones son el mecanismo que hace posible que Code First funcione en la práctica. Con ellas, tu base de datos evoluciona junto con tu código, y se mantiene un historial de todos los cambios para control y reversión.**

**46. Creación de Base de Datos**

**1. Contexto**

* Hasta ahora solo teníamos el **código** (modelos, contexto, migraciones creadas).
* El paso siguiente es que **Entity Framework genere físicamente las tablas** en SQL Server.
* Para esto:
  + Primero se crea la **base de datos vacía** en SQL Server.
  + Luego se ejecuta Update-Database en Visual Studio para aplicar las migraciones y construir las tablas.

**2. Crear la base de datos manualmente**

1. Abrir **SQL Server Management Studio (SSMS)**.
2. En el árbol de objetos, hacer clic derecho en **Databases → New Database**.
3. Asignar un nombre (debe coincidir con el de tu connection string, en este caso Store).
4. Aceptar → se crea la base de datos (sin tablas todavía).

**3. Aplicar las migraciones desde Visual Studio**

1. Abrir la **Consola del Administrador de Paquetes NuGet**:
   * Menú → **Herramientas** → **Administrador de paquetes NuGet** → **Consola de administrador de paquetes**.
2. Ejecutar:

Update-Database

* + EF aplica todas las migraciones pendientes sobre la base.
  + Se crean las tablas que corresponden a los modelos (Brand, Beer, etc.).
  + También se crea una tabla interna llamada \_\_EFMigrationsHistory (donde EF lleva registro de qué migraciones ya fueron aplicadas).

**4. Posibles errores comunes**

**🔹 Error 1: Globalization Culture Not Found Exception**

* Se da por un bug en .NET 8 al trabajar con globalización.
* Solución:
  1. Abrir el archivo **.csproj** del proyecto.
  2. Buscar la propiedad InvariantGlobalization.
  3. Cambiarla a **false**:

<InvariantGlobalization>false</InvariantGlobalization>

**🔹 Error 2: Certificado del servidor**

* Si no agregás TrustServerCertificate=True en tu connection string, podés recibir un error de SSL.
* Asegurate de tenerlo:

"ConnectionStrings": {

"StoreConnection": "Server=BOOK-655ITP02RO\\SQLEXPRESS;Database=Store;User Id=usuario;Password=clave;TrustServerCertificate=True;"

}

**5. Verificar que las tablas se crearon**

1. En **SSMS**, expandir la base Store.
2. Hacer clic derecho en **Tables → Refresh** (no se actualiza automáticamente).
3. Deberían aparecer al menos **tres tablas**:
   * Brands
   * Beers
   * \_\_EFMigrationsHistory (tabla interna de EF).

**6. Visualizar con Database Diagram**

* En SSMS:
  1. Ir a **Database Diagrams → New Database Diagram**.
  2. Seleccionar las tablas Brands y Beers.
  3. EF muestra gráficamente las tablas y la relación.
* Aquí se puede verificar que:
  1. Beer.BrandId está como clave foránea hacia Brand.BrandId.
  2. Las tablas están correctamente relacionadas.

**7. Resumen**

* **Crear la base vacía en SQL Server** manualmente.
* Ejecutar Update-Database → EF aplica las migraciones y construye las tablas.
* Corregir errores comunes (InvariantGlobalization, certificado SSL).
* Verificar en SQL Server Management Studio con un **refresh**.
* Usar el **Database Diagram** para confirmar relaciones y llaves foráneas.

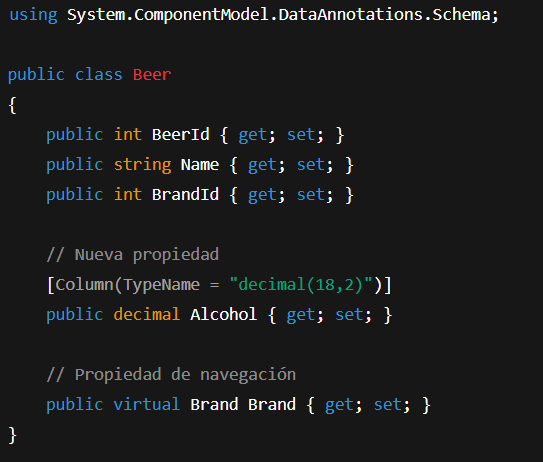
**47. Modificación en Modelos de Entity Framework**

**1. Escenario real**

* En un sistema, la **base de datos nunca es estática**: siempre habrá que **agregar, modificar o eliminar columnas/tablas**.
* Con **Code First + Migraciones**, estos cambios se hacen directamente en el código (modelos) y luego se reflejan en la BD con una nueva migración.

**2. Ejemplo: Agregar un nuevo campo**

Queremos agregar un campo Alcohol (decimal) en la entidad Beer.



**Explicación**

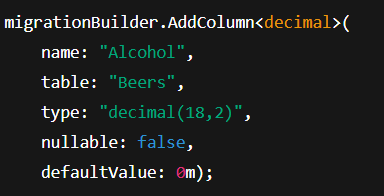
* [Column(TypeName = "decimal(18,2)")]
  + Indica el tipo exacto de columna en SQL Server.
  + 18,2 significa:
    - Hasta **18 dígitos en total**.
    - **2 dígitos decimales** (ejemplo: 99.99).
* Si no se especifica, EF puede fallar al generar la columna decimal.

**3. Crear una nueva migración**

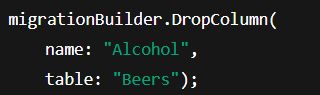
Desde la consola de NuGet:

Add-Migration AddAlcoholToBeer

* Esto genera un archivo de migración en la carpeta **Migrations**.
* En el método Up, EF escribe el SQL para agregar la columna:



* En el método Down, escribe cómo eliminarla:



**4. Revertir la migración (opcional)**

Si te equivocaste en algo y la migración aún no se aplicó:

Remove-Migration

➡ El archivo de migración se elimina.

Si ya la aplicaste con Update-Database y querés volver atrás:

Update-Database NombreMigracionAnterior

**5. Aplicar los cambios a la BD**

Ejecutar:

Update-Database

* EF aplica la nueva migración.
* La tabla Beers ahora tiene una nueva columna Alcohol.

**6. Verificar en SQL Server**

1. En **SQL Server Management Studio** (SSMS), refrescar las tablas.
2. Revisar la tabla Beers → aparece la columna Alcohol con tipo decimal(18,2).
3. En el **Database Diagram**, se verá también reflejado el cambio.

**7. Resumen de pasos para modificar la BD**

| **Paso** | **Acción** | **Comando** |
| --- | --- | --- |
| 1 | Modificar la clase modelo (ej: agregar Alcohol) | — |
| 2 | Crear migración | Add-Migration Nombre |
| 3 | (Opcional) revertir antes de aplicar | Remove-Migration |
| 4 | Aplicar cambios en la BD | Update-Database |
| 5 | Verificar en SQL Server | Refresh en SSMS |

📌 **Conclusión:**  
Las **modificaciones en la BD** en EF Core siguen el mismo patrón que la creación inicial: cambiar el modelo → generar migración → aplicar a la BD. Con esto, la BD evoluciona junto con el código, manteniendo un control de versiones claro.

**Comandos de EF Core en la Consola de Paquetes**

| **Comando** | **¿Qué hace?** | **Ejemplo de uso** |
| --- | --- | --- |
| **Add-Migration <Nombre>** | Crea una nueva migración a partir de los cambios en tus modelos. | Add-Migration InitDb (crea la migración inicial con las tablas de tus modelos). |
| **Update-Database** | Aplica todas las migraciones pendientes a la base de datos (ejecuta los métodos Up). | Update-Database (crea las tablas en la BD según la última migración). |
| **Update-Database <Nombre>** | Revierte o avanza la base de datos hasta la migración indicada. | Update-Database InitDb (revierte la base de datos al estado de la migración InitDb). |
| **Remove-Migration** | Elimina la última migración creada (si todavía no se aplicó a la BD). | Remove-Migration |
| **Script-Migration** | Genera un script SQL con todos los cambios desde la primera hasta la última migración. | Script-Migration |
| **Script-Migration <Desde> <Hasta>** | Genera un script SQL solo entre dos migraciones específicas. | Script-Migration InitDb AddAlcoholPercentage |
| **Get-Migration** | Lista todas las migraciones que tiene el proyecto. | Get-Migration |

**Si querés ver todos los comandos disponibles en NuGet console:**

Get-Help EntityFramework