Material teórico complementario sobre Entity Framework
Diseño de Aplicaciones I - 2016

IMPORTANTE: antes de continuar con este material se deben tener claro los siguientes conceptos, ya que se darán por sabidos:

- **Mapeo Objeto Relacional. Entity Framework.**
- **Mecanismos para trabajar con EF (en particular Code First).**
- **Crear un contexto. ¿Qué es? ¿Cómo guardo/elimino/modifico entidades en él? Utilización del contexto.**
- **Lambda Expressions. Funciones que utilizan predicados.**

Si todavía no tienes muy claro alguno de estos conceptos, es recomendable que los repases para que puedas entender correctamente este material.

1. Object/Entity State

1.1. Concepto de Estado de las entidades

Cada vez que nosotros hacemos un ".Add" al contexto desde código, estamos indicando que tenemos un objeto en nuestro modelo, que queremos que se mapee y se agregue su correspondiente objeto en la base de datos. Entonces, el contexto funciona simplemente como un intermediario entre nuestros objetos (el código de C#), y la base de datos de SQL Server. Los cambios que se producen no se confirman hasta realizar un SaveChanges(). Al ejecutarse este método, se realiza no solo la inserción del objeto en cuestión (lo que sería el volcado hacia la BD) , si no que también ocurre una sincronización en el otro sentido, es decir, desde la base de datos a nuestras entidades en memoria.

Justamente esto lo veremos al ver cómo trabaja internamente el método SaveChanges(). Al realizarse la llamada a tal método, se cambia el estado de las entidades que hayan sido afectadas desde la última transacción confirmada (desde el SaveChanges() anterior). Y es aquí donde mencionamos el concepto de estado: cada entidad tiene un estado llamado EntityState (https://msdn.microsoft.com/en-us/library/system.data.objects.dataclasses.entityobject.entitystate(v=vs.110).aspx). Tal estado, por definición, es una característica temporal de las entidades que estamos persistiendo (y que incluso nosotros a mano podemos modificar mediante la property EntityState), y cuyos posibles valores son:

- **Added: La entidad está siendo rastreada por el contexto pero todavía no existe en la base de datos. Es decir, cuando hacemos un .Add la entidad existe en nuestro DBSet del contexto pero no en la base de datos real.**

- **Unchanged: La entidad está siendo rastreada por el contexto y existe en la base de datos. La diferencia radica en que las propiedades del objeto en el contexto y en la base de datos no han cambiado, no difieren.**

- **Modified: la entidad está siendo rastreada por el contexto y existe en la base de datos, pero alguna de sus propiedades han sido cambiadas en el contexto.**

- **Deleted: la entidad está siendo rastreada por el contexto y existe en la base de datos, pero ha sido marcada para ser borrada de la base de datos. Esto ocurre cuando se llame al método SaveChanges().**

- **Detached: la entidad no es rastreada por el contexto.**

1.2. Comportamiento del SaveChanges() dependiendo del EntityState

La forma en que actúa SaveChanges() sobre cada entidad depende de los estados que estas tienen:

- **Si la entidad está en el estado *Unchanged*, no se manda ninguna actualización a la base de datos para esa entidad.**
- **Si la entidad está en el estado *Added*, el SaveChanges() inserta esas entidades en la base de datos, y las marca como *Unchanged* al finalizar.**

Ante esto, nos hacemos una pregunta: ¿Es lo mismo agregar una entidad con un .Add() al DBSet, que con un .State = EntityState.Added? Es decir, ¿las siguientes porciones de código producen el mismo resultado?

```
using (var miContexto = new ContextoDePrueba())

{

        var  unDepartamento = new Departmento { NombreDepto = "Finanzas" };

        var unEmpleado = new Empleado { Nombre = "Daniel", Apellido = "Lopez",

        Departmento = unDepartamento };

        miContexto.Entry(unEmpleado).State = EntityState.Added;

        miContexto.SaveChanges();

}

using (var miContexto = new ContextoDePrueba())

{

        var unDepartamento = new Departmento { NombreDepto = "Tecnologia" };

        var unEmpleado = new Empleado { Nombre = "Juancito", Apellido = "Gomez",

        Departmento = unDepartamento };

        miContexto.Empleados.Add(unEmpleado);

        miContexto.SaveChanges();

}
```

Y la respuesta es la siguiente: llamar a DbSet.Add y asignar el State como Added hacen lo mismo. Lo cual es: si la entidad a agregar no está siendo rastreada por el contexto, comenzará a ser rastreada al pasar al estado Added (lo cual ambos métodos hacen). Lo importante a tener en cuenta es que ambas formas son "operaciones de grafo", lo cual significa que también se afectan las entidades/colecciones "hijas". Es decir, si alguna entidad que puede ser "alcanzable" desde la entidad raíz, no está siendo rastreada, se la marcará también como Added. En nuestro ejemplo la entidad raíz sería el empleado y la entidad hija que hay es el departamento, ya que un empleado tiene un departamento.

- **Si la entidad está en el estado *Modified*, actualiza las entidades en la base de datos, y las marca como *Unchanged* al Finalizar.**

- **Si la entidad está en el estado *Deleted*, la borra de la base de datos y la marca como *Detached* en el contexto.**

Y aquí nos preguntamos lo mismo que nos preguntamos para la inserción pero para la eliminación: ¿es lo mismo eliminar una entidad con un .Remove() al DBSet, que con un .State = EntityState.Deleted? Sí, con la diferencia de que ninguna de ellas es una operación de grafo. Solo afecta el estado de la entidad a borrar, no las entidades que se encuentran en su grafo de objetos. La única excepción a esta regla se produce si habilitamos el "borrado en cascada".

1.3. Attach: Añadiendo entidades al contexto que ya existen en la base de datos

¿Qué sucede cuando desde afuera del código hemos insertado entidades a mi base de datos? Por ejemplo: supongamos que corremos nuestra aplicación e insertamos ciertos objetos en la base de datos que se encontraba vacía. Esto no debería dar problema, ya que son inserciones simples. En consecuencia nuestra base de datos ahora posee ciertos objetos. Luego, si cerrara la aplicación, los objetos deberían seguir existiendo en ella ya que justamente esa es la idea de una base de datos: lograr la persistencia. Ahora, si quisiera correr la aplicación nuevamente e insertar alguno de los mismos elementos que ya existen en la base de datos, naturalmente tendría un error. En código se lanza una excepción de Entity Framework que indica que se están intentando insertar duplicados.

Del mismo modo, qué sucede si luego de insertar objetos en el contexto utilizando la

estructura de control "using", ¿queremos volver a realizar una inserción de un objeto auxiliar asociado al anterior? Por ejemplo:

En una cierta parte del código hago:

```
var unDepartamento;

using (var miContexto1 = new ContextoDePrueba())

{

	unDepartamento = new Departmento { NombreDepto = "Tecnologia" };

	miContexto.Departamentos.Add(unDepartamento);

	miContexto.SaveChanges();

}
```

Y luego en más adelante queremos trabajar con ese "unDepartamento" en nuestro código:

```
using (var miContexto2 = new ContextoDePrueba())

{

	var unEmpleado = new Empleado { Nombre = "Luis", Apellido = "Barragué",

	Departmento = unDepartamento };

	miContexto2.Empleados.Add(unEmpleado);

	miContexto2.SaveChanges();

}
```

En este segundo caso, miContexto2 acaba de ser instanciado, por lo que tiene los DbSets "vacíos", y no conoce al departamento que ya se insertó en la base de datos. Entonces, lo que sucede es que como el contexto no está "rastreando" ni "unDeparamento", ni "unEmpleado", al insertar "unEmpleado" en el DbSet, también comenzará a trackear al "unDepartamento" asociado a tal empleado, y por ende como esta instancia actual del contexto no sabe que en la base de datos ya existe ese unDepartamento, al hacer SaveChanges, se van a insertar ambas entidades.

Pudiendo ocurrir:

1) **Si NombreDepto es clave del objeto, entonces ocurrirá una excepción diciendo que se quiere insertar un objeto en la base de datos, pero ya existe una tupla (un objeto), con tal identificador (en este caso "Tecnologia").**

2) **Si NombreDepto no es clave, y por ende la base de datos está realizando un autoenumerado propio de cada objeto insertado, tendremos en ella una inserción duplicada, es decir, tendremos dos Departamentos de nombre "Tecnologia".**

¿Por qué se está dando esto? Nosotros ya sabemos que nuestras entidades ya existen en la base de datos porque las insertamos en un contexto que vivió previamente, pero tales entidades no están siendo rastreadas en la ejecución actual por el contexto actual. Es decir, el contexto no conoce a esos objetos, ya que el contexto, ese "enlace" entre mis objetos del código y la base de datos, se llena en tiempo de ejecución, cada vez que se instancia. Si nosotros observamos el contexto en esta nueva vida del contexto, veremos que naturalmente los DbSet estarán vacíos.

Esto se da porque Entity Framework se ocupa de rastrear o "trackear" el estado de las entidades mientras estas estén conectadas a un contexto, sin embargo, cuando no lo están, es responsabilidad del programador "avisarle" a Entity Framework que hay ciertas entidades que el contexto debería conocer debido a que ya existen en la base de datos.

¿Cómo se soluciona este problema?

Usando el método Attach sobre los DbSet. Este método avisa que una cierto objeto en código ya existe en la base de datos, poniéndole estado Unchanged. Por ejemplo:

```
var perroQueYaExiste = new Perro { IdPerro = 1, Nombre = "Doug the Pug" };

using (var miContexto = new DirectorioDePerrosContexto())
{
    miContexto.Perros.Attach(perroQueYaExiste);

    // Hago lo que quiera hacer con el perro

    miContextO.SaveChanges();
}
```

Es importante destacar que no se harán cambios a la base de datos si llamamos al SaveChanges sin haber modificado nuestra entidad a la que le hicimos attach. Esto es debido a como dijimos más arriba, el estado de la entidad al hacerle Attach queda en Unchanged.

Justamente, otra forma de llevar a cabo la funcionalidad del Attach, es cambiar el estado de una entidad a Unchanged.

```
var perroQueYaExiste = new Perro { IdPerro = 1, Nombre = "Bobby };

using (var miContexto = new DirectorioDePerrosContexto())
{
    miContexto.Entry(perroQueYaExiste).State = EntityState.Unchanged;

    // Hago lo que quiera hacer con el perro

    miContextO.SaveChanges();
}
```

Algo muy importante a notar es que en ambos ejemplos si cualquiera de las entidades que "attacheamos" (sobre las que aplicamos el Attach) tiene referencias a otras entidades que aún no han sido rastreadas por el contexto, entonces esas nuevas entidades se añadirán al contexto para ser rastreadas, con estado Unchanged.

Se puede leer más sobre este método en:

**http://www.entityframeworktutorial.net/EntityFramework5/attach-disconnected-enti
y-graph.aspx**

2. Fluent API

2.1. Introducción: recordando las DataAnnotations

La ventaja principal que ofrece Code First, es el hecho de que simplemente, a partir de nuestro modelo de dominio orientado a objetos simples (también denominado POCO, "Plain Old CLR Objects", es decir objetos simples que puedan ser ejecutados por el CLR), se puede generar el modelo de la base de datos. Obviamente esta tarea no es gratis: para que el mapeo sea llevado a cabo correctamente, se requiere algo de trabajo extra que permita ajustar o "ayudar" a que Entity Framework realice ese mapeo que nosotros queremos. Esto lo hacemos por ejemplo cuando indicamos cuál queremos que sea la Primary Key de una entidad, cuando cambiamos el nombre de una tabla para evitar el que EntityFramework usa por defecto, o cuando hacemos que una cierta property de un objeto no pueda ser null en la base de datos, entre otros.

Hasta ahora, la forma en que veníamos logrando esto podía ser alguna de las siguientes:

1. **Haciendo que nuestras clases sigan ciertas convenciones para que Entity Framework mediante Code First pueda trabajar correctamente y resuelva los detalles necesarios para mapearlas. Esta forma de trabajo de Code First se basa en un principio que se llama "***Convención sobre Configuración***". En pocas palabras implica desligar al programador de ciertas responsabilidades haciendo que use convenciones establecidas para el entorno, pero sin reducir el poder de "modificar las cosas" (configuración) que tenemos.**

2. **Si nuestras clases no siguen esas convenciones, también podemos añadir ciertas "configuraciones" sobre esas clases, para darle a Entity Framework la información que necesita para realizar el mapeo. Esta forma que hemos visto es a partir del uso de ciertos atributos llamados "DataAnnotations", (pertenecientes al namespace** *System.ComponentModel.DataAnnotations***). Simplemente son marcas (como tags) que vamos colocando sobre el código de nuestras clases y que cambian el comportamiento del mapeo que Entity Framework Code First va a realizar. Esto es muy bueno ya que muchas veces este asume cosas que no queremos que se asuman, pudiendo entonces corregirlas.**

Por ejemplo:

La convención para claves en EntityFramework es que la property que queremos que sea clave primaria se llame "Id" o "*NombreDeLaClase*Id". Pero si queremos que nuestra clave se llame de otra forma, podemos usar la annotation [Key].

```
public class Perro
{
    [Key]
    public int NumeritoQueIdentificaAMiPerro { get; set; }
    public string Nombre { get; set; }
    public string Raza { get; set;}
}
```

(Para ver más ejemplos de DataAnnotations que podemos usar, visitar: http://www.entityframeworktutorial.net/code-first/dataannotation-in-code-first.aspx

Esto funciona perfecto y se aplica a todas las annotations que queramos usar. Ahora, echémosle un vistazo a la siguiente porción de código:

```
[Table("Duenios", Schema = "dbo")]
public class Duenio
{
    [Key]
    public string Cedula { get; set; }
    [Required]
    public string Nombre { get; set; }
    [MinLength(2)]
    [MaxLength(100)]
    [Index("IX_Apellido_Duenios", IsClustered = false, Order = 2)]
    public string Apellido { get; set; }
    public ICollection<Perro> PerrosQuePosee { get; set; }
}

[Table("Perros", Schema = "dbo")]
public class Perro
{
    [Column("ID", Order = 1)]
    [Key]
    public int IdDelPerro { get; set; }
    [Column("Nombre", Order = 2, TypeName = "Varchar(100)")]
    [Required]
    public string Nombre { get; set; }
    public string Raza { get; set; }
    [InverseProperty("PerrosQuePosee")]
```

```
    public Duenio Duenio { get; set; }
}
```

¿Qué vemos en el código anterior?

Se definen dos clases simples, estableciendo una relación entre ellas. Sin embargo, el código es relativamente complejo de leer, debido a que abundan las DataAnnotations para facilitar que EntityFramework Code First realice el mapeo adecuado. Se pierde entonces la simplicidad que tienen que tener los objetos POCO de nuestro dominio. Y no solo eso, también se están violando principios de diseño (por ejemplo SRP), ya que nuestras clases encapsulan más de una única responsabilidad: se encargan de definir las características y el comportamiento tanto de *Perros* como de *Duenios*, pero también se encargan de definir características de las entidades en la base de datos.

En consecuencia entonces, nuestras clases de nuestra lógica del negocio están siendo impactadas por la persistencia, mientras que lo ideal es que nuestras clases resuelvan el problema por el que fueron diseñadas, sin tener que preocuparse por ella.

2.2. Fluent API: separando la persistencia de nuestro dominio

Como vimos entonces, las DataAnnotations son interesantes de usar cuando tenemos aplicaciones simples. A medida que crece la complejidad en ellas, es aconsejable el uso de Fluent API.

Fluent API hace lo mismo que las DataAnnotations: permite customizar o configurar nuestro modelo de dominio ayudando a que el mapeo sea correcto. Sin embargo, para evitar que nuestras clases se vean impactadas por la persistencia, Fluent API provee una forma de describir esas configuraciones imperativamente, a través del código. En consecuencia, tendremos aislados nuestro modelo de dominio de la forma en que implementemos la infraestructura de persistencia.

Antes de entrar en los detalles de Fluent API, es importante aclarar que todo lo que se puede configurar con las DataAnnotations es posible con Fluent

API, pero al revés no. Fluent API provee mayores opciones de configuración, de manera que es importante que al menos sepamos cómo funciona y cómo usarlo.

Partiremos del siguiente modelo:

```
public class Autor
{
	public int IdAutor { get; set; }
	public string Nombre { get; set; }
	public string Descripcion { get; set; }
	public byte[] Foto { get; set; }
	public virtual ICollection<Libro> Libros { get; set; }
}

public class Libro
{
	public int IdLibro { get; set; }
	public string Nombre { get; set; }
	public DateTime FechaPublicacion { get; set; }
	public Autor Autor { get; set; }
}

public class BibliotecaBD : DbContext
{
	public DbSet<Autor> Autores { get; set; }
	public DbSet<Libro> Libros { get; set; }
}
```

Ahora supongamos que queremos asegurarnos que al realizar el pasaje a nuestro modelo en la base de datos, se cumplan las siguientes condiciones:

- **El Autor tiene que tener un Nombre sí o sí.** *DataAnnotation: [Required].*
- **La Descripción del Autor debe ser de máximo 500 caracteres.** *DataAnnotation: [MaxLenght(500)]*
- **Al alojarse la Foto del Autor en SQL Server, el tipo que debe tener la columna debe ser 'image'.** *DataAnnotation: [Column(TypeName='image')]*
- **El Libro tiene que tener un Nombre sí o sí.** *DataAnnotation: [Required].*

¿Cómo logramos esto con Fluent API?

Las configuraciones de Fluent API se aplican mientras que Code First está construyendo el modelo a partir de las clases. Podemos inyectar esas configuraciones sobrescribiendo el Método OnModelCreating del DbContext. Entonces, estamos integrando tales configuraciones en el contexto, dejándolas fuera de nuestras clases del dominio.

¿Por qué es esto posible?

A la hora de construir el modelo de la base de datos, el DbContext primero mira a las clases que queremos persistir y "aprende" lo que puede de ellas. Una vez realiza esto, el contexto está pronto para construir el modelo, pero existe la oportunidad de que el desarrollador interrumpa al contexto y le indique configuraciones adicionales. Entonces, el método DBContext.OnModelCreating es llamado por el contexto justo antes de que el modelo se construya. El método obviamente es virtual, permitiendo que sobreescribamos el original y agreguemos nuestra propia lógica. En esa lógica, es donde colocamos las llamadas de Fluent API.

En otras palabras, al sobrescribir el método OnModelCreating del DbContext, podemos ir construyendo las diferentes características de nuestro modelo programáticamente, encadenando llamadas de métodos que son sencillos de leer. Lo que retorna cada llamada define en consecuencia el conjunto de métodos válidos para la llamada siguiente.

Entonces, usando FluentAPI, nuestro contexto quedaría así:

```
public class BibliotecaBD : DbContext
{
	public DbSet<Autor> Autores { get; set; }
	public DbSet<Libro> Libros { get; set; }
	protected override void OnModelCreating(DbModelBuilder modelBuilder)
	{
		modelBuilder.Entity<Autor>()
		.Property(a => a.Nombre).IsRequired();

		modelBuilder.Entity<Autor>()
		.Property(a => a.Descripcion).HasMaxLength(500);

		modelBuilder.Entity<Autor>()
		.Property(a => a.Foto).HasColumnType("image");

		modelBuilder.Entity<Libro>()
```

```
            .Property(l => l.Nombre).IsRequired();
        }
}
```

Lo que estamos haciendo es, para ciertas entidades setear características a sus propiedades. Como se puede observar, el código es entendible y no necesita que se expliquen los detalles, ya que el DbModelBuilder nos da una mano. Por ejemplo, en la primer línea del método, lo que hacemos es:

1. **Le decimos al model builder que queremos afectar una de sus entidades, y usamos generics para especificar cuál entidad. En este caso, Autor.**

2. **Una vez que tenemos acceso a ella, podemos obtener la propiedad Nombre del Autor. Para ellos usamos una lambda expression.**

3. **Finalmente, encadenamos la última llamada con el método IsRequired(), que trabajando con una cierta Property de una entidad, hace que esta sea not null en la base de datos.**

2.2.1. Guía: Probando Fluent API

Este material obviamente no tiene como fin ilustrar todos los usos de Fluent API en nuestra base de datos, si no simplemente introducir y explicar cómo es que funciona de forma teórica. Para ver diferentes posibilidades de configuraciones usando Fluent API, visitar:

https://msdn.microsoft.com/en-us/data/jj591617.aspx

Y para relaciones:

https://msdn.microsoft.com/en-us/data/hh134698.aspx

https://msdn.microsoft.com/en-us/data/jj591620.aspx

1) **Comenzaremos en primer lugar creando un proyecto (Aplicación de Consola)**

2) **Instalar Entity Framework en él.**

3) **Crear clase Persona con las properties:**
   - **PersonaId (int)**
   - **Nombre (string)**
   - **Apellido (string)**

4) Crear clase ContextoDB con el DbSet Persona

5) Agregar el connection string:

```
<connectionStrings>
    <add name="ContextoDB"
connectionString="Server=NOMBREDELAINSTANCIA\SQLEXPRESS;Database=EjemploCF
;Integrated Security=True" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

6) Abrir la consola de administración de paquetes (Tools -> NuGet Package Manager):

Habilitar las migraciones:

```
Enable-Migrations
```

Actualizar la base de datos para impactar los cambios:

```
Add-Migration "persona"
```

```
Update-Database
```

Si vemos en el SQL Server Management Studio, hasta ahora como Code First pudo mapear nuestra clase persona, sin embargo, como hemos visto anteriormente, se ha basado en convenciones que nosotros hemos respetado. ¿Qué sucede si nosotros queremos hacerlo diferente, o necesitamos hacerlo de otra forma que no sea utilizando convenciones?

Ya hemos visto que tenemos dos formas de configurar y ayudar a que EF Code First realice el mapeo necesario. En este caso, usaremos Fluent API.

7) Primer cambio: Lo que haremos será cambiar la property "PersonaId" por "Cedula" y veamos que hace Entity Framework CF. Luego haremos la migración.

Lo primero que vemos es que obtenemos un error porque no se puede encontrar una Property que sea clave para la clase Persona.

Para ello, en ContextoDB, sobreescribimos el método OnModelCreating:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
}
```

Como hemos explicado anteriormente, este es el mecanismo que se usa en Fluent API para configurar una base de datos. Para ello, tenemos que indicarle entonces que nuestra property "Cedula" queremos que sea la clave de la clase "Persona".

Agregar el código para indicar que la pk sea Cedula:

```
modelBuilder.Entity<Persona>().HasKey(p => p.Cedula);
```

Lo que recibe HasKey es una Lambda Expression. Esto es simplemente una función que recibe una Persona (y que nosotros denominamos p), y que simplemente espera que devolvamos un string. En este caso p.Cedula, que se convierte en string con ToString.

Ahora queremos realizar la migración de vuelta, para poder impactar en la Base de Datos:

Add-Migration "cedula"

Update-Database

¿Qué vemos ahora en la base de datos?

8) Campos "requeridos" (que no pueden ser null)

Si vamos a la base datos en el SQL Server Management Studio, vemos como tanto el nombre como el apellido de una persona pueden ser NULL. Como en la realidad que queremos modelar deseamos que tales campos en las tablas no lo sean, podemos utilizar Fluent API para evitarlo.

```
modelBuilder.Entity<Persona>().Property(p => p.Nombre).IsRequired();
```

Realizamos las migrations para impactar en la base de datos y ver el cambio.

9) Relaciones: One-To-One y One-To-Zero-Or-one

Para analizar estas relaciones crearemos una clase nueva que estará relacionada con la clase Persona: la clase Pasaporte. Para ello entonces Creamos la clase Pasaporte.

Pasaporte:

- **Id (int)**
- **Nacionalidad (string)**
- **Persona (Persona)**

Luego, relacionamos ambas clases diciendo que la Persona tiene una property que es un Pasaporte.

Además, agregamos el DbSet de Pasaportes en el contexto.

Intentamos agregar la migración: lo que puede pasar es que EF no pueda definir la relación dominante y cuál la dependiente. Es decir, EF Code First no sabe quién tiene la Foreign Key sobre cuál. O si no, tal vez coloque la Foreign Key de una forma que nosotros no queramos.

Para ello, vamos al contexto y en el OnModelCreating agregamos:

```
modelBuilder.Entity<Pasaporte>().HasRequired<Persona>(p =>
p.Persona).WithOptional(p => p.Pasaporte);
```

Le estamos diciendo que todo pasaporte debe estar asociado a una persona, pero que una persona puede o no tener un pasaporte.

Si ahora impactamos en la base de datos y vemos los cambios, veremos que el nombre de la Foreign Key en Pasaporte es "Persona_Cedula" (el nombre de la clase sobre la cual refiere, y el nombre de la primary key de esa clase). Para indicarle que queremos que se llame solamente "Persona", agregamos a la línea anterior:

```
modelBuilder.Entity<Pasaporte>().HasRequired<Persona>(p =>
p.Persona).WithOptional(p => p.Pasaporte).Map(mc => mc.MapKey("Persona"));
```

Impactamos en la base de datos y vemos los cambios.

10) Relaciones: One-To-Many

Primero, crearemos la clase Empresa

- RUT (int)
- Nombre (string)
- Empleados (List<Persona>)

Luego, agregamos el DbSet Empresa en el contexto.

Impactar en la base de datos y ver que falla porque la clase Empresa no sigue la convención de pk. Agregamos la siguiente línea:

```
modelBuilder.Entity<Empresa>().HasKey(e => e.RUT);
```

Impactar en la base de datos y vemos el resultado. Notar que la Foreign Key aparece en persona (como tiene que ser) y que aparece nula. Esto quiere decir que una persona puede o no pertenecer a una Empresa.

Si en nuestro sistema queremos que toda persona pertenezca a una empresa, entonces lo tenemos que configurar:

```
modelBuilder.Entity<Empresa>().HasMany<Persona>(e =>
            e.Empleados).WithRequired();
```

Si ahora impactamos Impactar en la base de datos, vemos que ahora la Foreign Key es NOT NULL.

Ahora agregamos la clase Proyecto:

- **ProyectoId (int)**
- **Nombre (string)**
- **FechaComienzo (DateTime)**

Agregar el DbSet Proyecto en el contexto.

Impactamos en la base de datos. Si vemos el nombre de la tabla al realizar el mapeo, vemos que se le ha agregado "es" a final de "Proyecto", ya que intenta mapear el nombre de la clase a la tabla poniéndolo en Plural (en inglés). Para solucionarlo, agregamos:

```
modelBuilder.Entity<Proyecto>().ToTable("Proyectos");
```

Impactamos en la base de datos y vemos ahora el cambio.

Ahora, agregamos a Proyecto lo siguiente:

- **ProyectoPadre (Proyecto)**
- **SubProyectos (List<Proyecto>)**

Vemos que Entity Framework identificó que las dos relaciones forman parte de una misma relación y solo generó una FK.

Pero si agregamos a Proyecto lo siguiente:

- **SiguientesProyectos (List<Proyecto>)**

Vemos que en lugar de tener una sola FK ahora tenemos tres, porque EF Code First no puede identificar si ProyectoPadre se asocia con SubProyectos o con SiguientesProyectos.

Entonces, lo que haremos es volver a asociar es **ProyectoPadre** con **SubProyectos**. Lo primero que debemos hacer es comentar las Properties **ProyectoPadre, SubProyectos y SiguientesProyectos**. **Esto es importante porque cuando EF Code First crea una FK también crea índicas, que son usados para acceder a la información más rápido. Dado eso, no nos va a dejar agrupar las relaciones de ProyectoPadre con SubProyectos.**

Dicho y hecho eso, impactamos en la BD utilizando las migraciones.

Ahora vamos a descomentar todo y agregar la siguiente línea en el OnModelCreating del contexto:

```
modelBuilder.Entity<Proyecto>().HasOptional<Proyecto>(p =>
 p.ProyectoPadre).WithMany(p => p.SubProyectos).Map(mc =>
                    mc.MapKey("Padre"));
```

Impactamos y vemos que ahora tenemos solo dos relaciones, y que la relación ProyectoPadre – SubProyectos quedó agrupada en la fk "Padre".

Si queremos cambiar el nombre de la relación de SiguientesProyectos, mapeamos:

```
        modelBuilder.Entity<Proyecto>().HasMany<Proyecto>(p =>
p.SiguientesProyectos).WithOptional().Map(mc => mc.MapKey("Anterior"));
```

2.3. Usando Fluent API respetando Clean Code

La desventaja que tiene la forma presentada anteriormente para el uso de Fluent API, es que el nivel de cumplimiento con estándares o buenas prácticas de código limpio, es muy pobre. Si nuestra aplicación es relativamente robusta y requiere muchas configuraciones y muy avanzadas, nuestro método OnModelCreating puede quedar muy largo, con muchas responsabilidades y difícil de leer.
Lo que se puede hacer entonces es "agrupar" las configuraciones por entidad. Esto se logra a partir de extender las clases de EntityTypeConfiguration, y luego instanciarlas y agregarselas a las configuraciones del modelo.

Por ejemplo, dadas nuestras clases "Autor" y "Libro", crearemos dos clases separadas, "LibroConfiguracion" y "AutorConfiguracion", que deberían verse algo así:

```
public class LibroConfiguracion : EntityTypeConfiguration<Libro>
{
	public LibroConfiguracion()

	{

		Property(l => l.Nombre).IsRequired();

	}
}

public class AutorConfiguracion : EntityTypeConfiguration<Autor>
{
	public AutorConfiguracion()

	{
		Property(a => a.Nombre).IsRequired();
		Property(a => a.Descripcion).HasMaxLength(500);
		Property(a => a.Foto).HasColumnType("image");

	}
}
```

Finalmente, hacemos que nuestro DbContext utilice esas configuraciones:

```
public class BibliotecaBD : DbContext
{
```

```
        public DbSet<Autor> Autores { get; set; }
        public DbSet<Libro> Libros { get; set; }


        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
               modelBuilder.Configurations.Add(new LibroConfiguracion());

               modelBuilder.Configurations.Add(new AutorConfiguracion());
         }

}
```

Bibliografía usada:

Entity State:

https://msdn.microsoft.com/en-us/data/jj592676.aspx

Attaching and Dettaching objects:
https://msdn.microsoft.com/en-us/library/bb896271(v=vs.100).aspx

http://stackoverflow.com/questions/31599508/dbset-add-dbset-remove-versus-usin
-entitystate-added-entitystate-deleted

Fluent Api:

https://monjurulhabib.wordpress.com/2012/11/12/fluent-api-vs-data-annotations-w
rking-with-configuration-part-2/

http://deviq.com/persistence-ignorance/

http://programmers.stackexchange.com/questions/159007/are-fluent-interfaces-mo
e-flexible-than-attributes-and-why

http://www.codeproject.com/Articles/476966/FluentplusAPIplusvsplusDataplusAnno
ations-plusWor