Entity Framework
Code First Migrations

Alex Casquete

Email: acasquete@outlook.com
LinkedIn: https://www.linkedin.com/in/alexcasquete/

Abril 2012

RESUMEN

Una de las características más solicitadas desde la introducción de *Code First* (CF) en *Entity Framework* (EF), ha sido poder actualizar de forma incremental el esquema de la base de datos a medida que modificábamos el modelo de entidades. Durante los últimos meses, y gracias a las múltiples versiones preliminares que se han ido liberando, hemos tenido la oportunidad de ver y probar la evolución de cada funcionalidad dirigida a facilitar este proceso de migración. Ahora, con la reciente liberación de la versión 4.3 EF, que ha venido seguida de una actualización menor, ya tenemos disponible la primera versión con licencia *Go-Live*, es decir desplegable en entornos de producción, de *Code First Migrations*. En este artículo daremos un recorrido por todas las mejoras incorporadas, comenzando por la nueva forma de configurar CF y los comandos básicos para realizar la primera y sucesivas migraciones. Continuaremos viendo cómo actualizar las bases de datos de proyectos existentes para que trabajen con la nueva versión de EF, y los diferentes métodos que podemos utilizar para aplicar las migraciones en entornos de pruebas o producción.

Nota: Este artículo se publicó en el número 91 de la revista dotNetManía en abril de 2012.

ABSTRACT

Since the introduction of Code First (CF) in Entity Framework (EF), one of the most requested features has been to be able to incrementally update the database schema as we modify the entity model. During the last months, and thanks to the multiple preliminary versions that have been released, we have had the opportunity to see and test the evolution of each functionality aimed at facilitating this migration process. Now, with the recent release of version 4.3 EF, which has been followed by a minor update, we already have available the first version with a Go-Live license, that is, deployable in production environments, from Code First Migrations. In this article, we will tour all improvements included in EF, starting with the new way of configuring CF and the basic commands to perform the first and subsequent migrations. We will continue to see how to update existing project databases to work with the new version of EF and the different methods to apply migrations in test or production environments.

Note: This article was published in the issue 91 of April 2012 of the dotNetMania magazine.

1 Comenzando

Al igual que pasó con EF 4.2, y como probablemente suceda con las próximas entregas, la última versión no dispone de instaladores MSI (*Windows Installer*) y sólo está disponible para descargar a través de *NuGet*. Por lo tanto, para obtener la versión más reciente (actualmente la 4.3.1) es necesario tener instalada la extensión «*NuGet Package Manager*» en *Visual Studio*. Además, debemos tener en cuenta que la versión 4.3 es compatible con *Visual Studio 2010* y *Visual Studio 11 Beta*, sin embargo, no es recomendable utilizar la versión 4.5 del *Framework .NET* ya que existen diversos problemas de compatibilidad, relacionados con cambios en los algunos espacios de nombres. En el caso de encontrarnos en este escenario, deberíamos utilizar la versión 5 de EF que desde hace unas semanas se encuentra en versión *Beta*.

Para instalar el paquete y añadir las dependencias de EF a nuestro proyecto podemos utilizar la consola de *NuGet* (accesible desde el menú *Herramientas* > *Library Package Manager* > *Package Manager Console*) para ejecutar el comando Install-package EntityFramework (Figura 1).

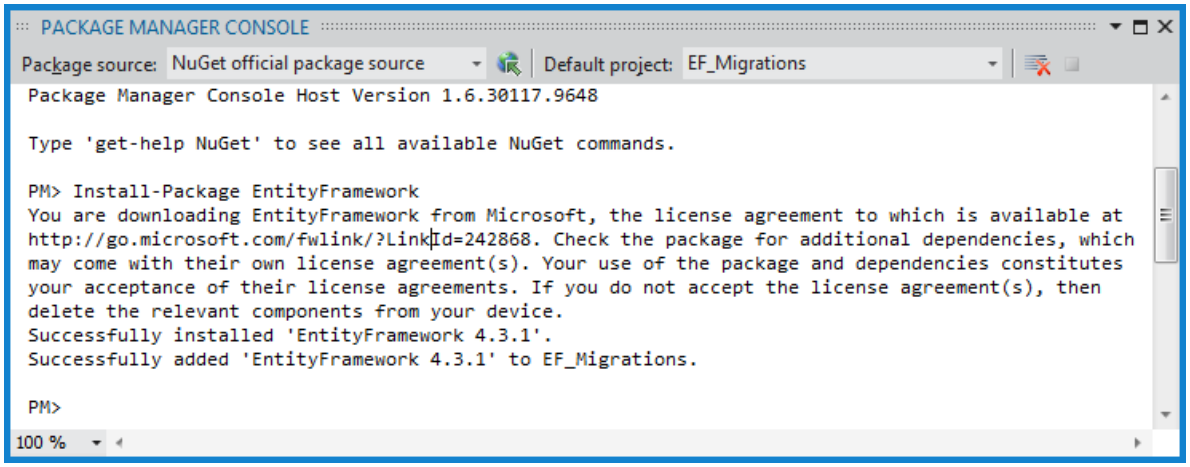

Figura 1 - Consola del administrador de paquetes de *Nuget* (*Package Manager Console*)

Al instalar el paquete, vemos los primeros cambios introducidos en EF 4.3. El fichero de configuración de nuestro proyecto (app.config o web.config) contiene una nueva sección **entityFramework** (Figura 2), que nos va a permitir especificar la factoría de conexión que por defecto CF utilizará para conectarse con la base de datos. Es importante comentar que CF sigue un principio de «convención sobre configuración", así que podemos prescindir totalmente de esta configuración teniendo en cuenta que la tendremos que modificar si los valores por defecto no son válidos para nuestro entorno.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.Config-
File.EntityFrameworkSection, EntityFramework, Version=4.3.1.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089" />
  </configSections>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnec-
tionFactory, EntityFramework">
      <parameters>
        <parameter value="Data Source=(localdb)\v11.0; Integrated Security=True;
MultipleActiveResultSets=True" />
      </parameters>
    </defaultConnectionFactory>
  </entityFramework>
</configuration>
```

Figura 2 - Fichero de configuración con la nueva sección de *entity framework*

2 Factorías de conexión

EF contiene dos factorías de conexión: **SqlCeConnectionFactory** y **SqlConnectionFactory.** Una para crear un objeto **DbConnection** para SQL Server Compact Edition y la otra para SQL Server. Si no estableciésemos una factoría de conexión, se utilizaría de forma predeterminada **SqlConnectionFactory** apuntando a la instancia SQLEXPRESS local. Visual Studio 11 Beta utiliza LocalDB en lugar de SQLEXPRESS, pero al instalar el paquete comprueba qué base de datos está instalada y utiliza el fichero de configuración para establecer la base de datos por defecto. La configuración que aparece en la Figura 2 indica que se utilizará una instancia en LocalDB para los contextos que no tengan establecida una cadena de conexión.

Se establece especificando el nombre de tipo calificado en el elemento **defaultConnectionFactory**. Esta factoría tiene un constructor que nos permite sobrescribir la cadena de conexión.

3 Cambiando el modelo

Para mostrar la funcionalidad de migración vamos a partir del modelo y contexto que aparecen en el código de la Figura 3. El perspicaz lector advertirá la poca complejidad del modelo, pero para nuestro particular «Hola Mundo» de migraciones, será suficiente.

```
namespace EF_Migrations
{
    using System.Data.Entity;

    public class Book
    {
        public int BookId { get; set; }
        public string Title { get; set; }
    }

    public class LibrarySample : DbContext
    {
        public IDbSet<Book> Books { get; set; }
    }

    class Program
    {
        static void Main()
        {
            using (var db = new LibrarySample())
            {
                db.Books.Add(new Book { Title = "El primer libro" });
                db.SaveChanges();
            }
        }
    }
}
```

Figura 3 - Modelo y contexto

Al ejecutar la aplicación CFnos crea una base de datos en nuestra instancia de SQL local con el nombre **EF_Migration.LibrarySample** y toda la estructura que podemos ver en la Figura 4.

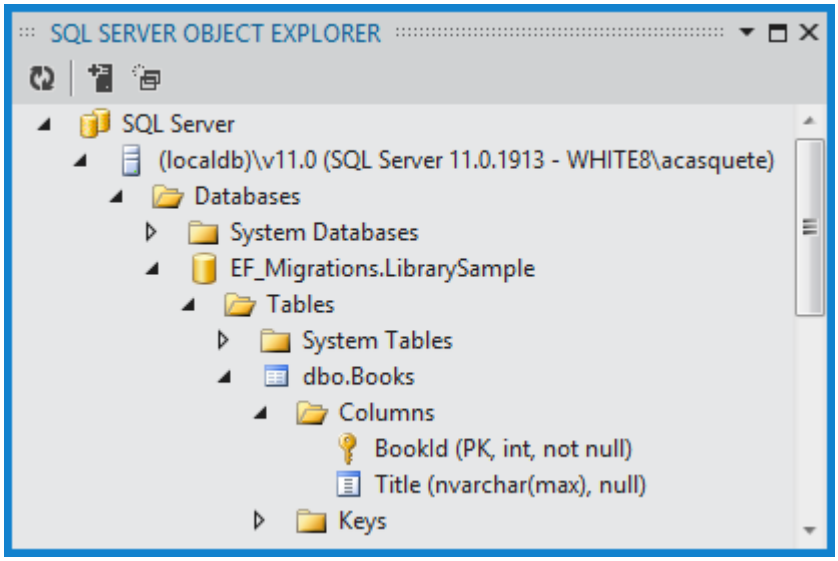

Figura 4 - explorador de *SQL server* con la base de datos creada

Si modificamos nuestro modelo de entidades, por ejemplo, añadiendo una nueva propiedad en la clase **Book** y ejecutamos de nuevo la aplicación, obtendremos la excepción

InvalidOperationException porque el modelo no está sincronizado con la base de datos, es decir, la estructura de la clase **Book** no coincide con la de la tabla que EF ha generado de forma automática. Hasta aquí todo normal, pero el mensaje de la excepción ya nos da una pista de lo que vamos a tener que hacer a partir de ahora.

"The model backing the 'BlogContext' context has changed since the database was created. Consider using Code First Migrations to update the database"

Con las versiones anteriores contábamos con varias alternativas para volver a sincronizar el modelo con la base de datos: podíamos actualizar la estructura de forma manual, podíamos eliminar la base de datos y ejecutar de nuevo la aplicación para que EF la volviese a crear, o podíamos por último, utilizar algún inicializador de base de datos para que, cuando hubiese algún cambio en el modelo, EF eliminase y recrease la base de datos. Todo esto lo podemos seguir haciendo de la misma forma, pero vamos a ver que utilizando la nueva característica de migraciones, mantener sincronizado el modelo se convierte en una tarea muy sencilla y nos evita el tener que inicializar la base de datos.

4 Migraciones automáticas

Para poder usar las migraciones tenemos que habilitarlas ejecutando en la consola del administrador de paquetes el comando **Enable-Migrations**, pero como además en este primer ejemplo vamos utilizar las migraciones automáticas debemos especificar el parámetro **EnabledAutomaticMigrations**.

Si observamos el explorador de la solución, veremos que al ejecutar el último comando se ha añadido el directorio **Migrations** a la estructura de nuestro proyecto. Dentro de este directorio encontraremos la clase **Configuration** que nos permite configurar el comportamiento de las migraciones en un contexto determinado. A su vez, el método **Seed** lo podemos utilizar para añadir información a las tablas. Una vez tenemos las migraciones habilitadas, tenemos que ejecutar el comando **Update-Database** para aplicar todos los cambios que hayamos hecho en el modelo a la base de datos. El comando por defecto no muestra ninguna información, si queremos ver el código SQL que será ejecutado podemos utilizar el parámetro **Verbose**.

Code First Migrations detecta cambios automáticamente cuando añadimos, renombramos o eliminamos una propiedad o una clase. Muchas veces mediante las migraciones automáticas nos puede, pero hay ocasiones en las que vamos a querer modificar el comportamiento, por ejemplo si queremos establecer en un campo un valor por defecto distinto al predeterminado. Para esto vamos a tener que hacer uso de las migraciones por código.

5 Migraciones por código

El siguiente comando que podemos utilizar en la consola es **Add-migration**. Este comando requiere que pasemos un valor como parámetro que servirá para asignar un nombre a la migración y para así poderla identificar más adelante para cuando queramos, por ejemplo, revertir una migración.

Para probar las migraciones por código, realizamos otro cambio en nuestro modelo, añadiendo una nueva propiedad y ejecutamos el comando **Add-migration** pasando como parámetro «SegundaMigración».

Este comando nos genera una nueva clase en el directorio **Migrations** con el nombre de la migración precedido de un *timestamp* para facilitar la ordenación. Esta clase tiene los métodos sobrescritos **Up** y **Down** de la clase **DBMigrations**. Estos métodos contienen las operaciones que se deben ejecutar durante los procesos de actualización en ambos sentidos, a una versión superior (*Up*) o a una inferior (*Down*).

Además en el diseñador tenemos la información adicional para poder calcular los cambios del modelo y poder replicar las migraciones automáticas cuando despleguemos en producción.

En este momento podemos realizar más cambios en nuestro modelo y añadirlo a la misma migración ejecutando el mismo comando, podemos crear una nueva migración o podemos modificar añadiendo operaciones. Los métodos disponibles para agregar distintas operaciones son los que aparecen en la Figura 5. De entre todos, puede que el método más interesante sea **Sql** ya que nos permite ejecutar cualquier comando SQL. Esto nos puede ser muy útil cuando queramos realizar una actualización de datos.

AddColumn, DropColumn	Agrega una operación para agregar o eliminar una columna a una tabla existente.
AddForeignKey, DropForeignKey	Agrega una operación para crear o eliminar una restricción de clave foránea.
AddPrimaryKey, DropPrimaryKey	Agrega una operación para crear o eliminar una clave principal.
AlterColumn	Agrega una operación para modificar la definición de una columna existente.
CreateIndex, DropIndex	Agrega una operación para crear o eliminar un índice en una columna.
CreateTable, DropTable	Agrega una operación para crear o eliminar una tabla.
MoveTable	Agrega una operación para mover una tabla a un Nuevo esquema.
RenameColumn, RenameTable	Agrega una operación para renombrar una columna o una tabla.
Sql	Agrega una operación para ejecutar un comando SQL.

Figura 5 - Métodos disponibles para agregar operaciones en las migraciones por código

Es muy importante que si definimos operaciones para actualizar la base de datos, proporcionemos el equivalente para devolver la base de datos a su estado anterior. En el código de la Figura 6 podemos ver como tenemos el código para añadir la columna

PublishYear en la tabla **Books,** y además estamos ejecutando una consulta para establecer un valor de a todos los registros que no lo tengan.

```
public partial class SegundaMigracion : DbMigration
{
    public override void Up()
    {
        AddColumn("Books", "PublishYear", c => c.Int(nullable: false, default-
Value: DateTime.Now.Year));

        Sql("UPDATE dbo.Books SET PublishYear = 2012 WHERE PublishYear IS NULL");
    }

    public override void Down()
    {
        DropColumn("Books", "PublishYear");
    }
}
```

Figura 6 - Migración por código que añade una columna y ejecuta un comando SQL

Una vez tenemos la migración definida solamente ejecutamos el comando **Update-database** para aplicar los cambios a la base de datos.

6 Revertir una migración

En muchas ocasiones necesitaremos volver a una versión específica de la base de datos. Para eso podemos utilizar el parámetro **TargetMigration** para revertir a es a migración. Por ejemplo podemos utilizar:

Update-Database –TargetMigration.

Si intentamos revertir una migración que comporte perdida de información tendremos que utilizar el parámetro **Force** en caso contrario obtendremos un error. También es posible volver a la versión inicial utilizando:

Update-Database–TargetMigration:$InitialDatabase –Force

7 Desplegando en producción

En los ejemplos anteriores se ha utilizado el comando **Update-Database** para actualizar la base de datos. Esto es muy útil para entornos de desarrollo, pero lo habitual para desplegar en entornos de producción es ejecutar únicamente el script SQL. El comando **Update-Database** dispone de otro parámetro con el que podemos indicar que no actualice la base de datos sino que nos genere un fichero SQL.

También podemos indicar la migración inicial y la migración final mediante los parámetros **SourceMigration** y **TargetMigration**.

Update-Database -Script -SourceMigration:$InitialDatabase

Si no especificamos una migración destino CF utilizará la última, incluyendo también las migraciones automáticas que se hayan ejecutado.

8 Actualizando desde EF 4.x

La diferencia principal con versiones anteriores de EF, es que la versión 4.3 genera la tabla **__MigrationHistory** en las tablas del sistema, que se crea cuando llevamos a cabo la primera migración o cuando la base de datos se inicializa, y contiene información de las migraciones que se han aplicado. La misma se utiliza cada vez que ejecutamos los comandos **Add-Migration** o **Update-Migration** para ver que es lo que ha cambiado del modelo y que es lo que se necesita actualizar:

Enable-Migrations –EnableAutomaticMigrations

Antes de la primera migración tenemos tener nuestro modelo sincronizado con el esquema de la base de datos. Una vez estamos seguros utilizamos el siguiente comando para añadir la migración inicial:

Add-Migration Inicial -IgnoreChanges

El parámetro **IgnoreChanges** se ha añadido en la actualización 4.3.1. Antes del mismo, si no utilizásemos este parámetro, CF nos generaría una migración con todas las entidades de nuestro modelo y nosotros tendríamos que encargarnos de eliminar. Es importante recalcar que si no tenemos nuestro dominio sincronizado con la base de datos, los cambios que tengamos no se verán reflejados en la base de datos, ya que tenemos una migración vacía, es decir, para CF no habráningún cambio.

Si estamos actualizando desde EF 4.1 o 4.2, también podemos eliminar la tabla **EdmMetadata**, ya que no se hace uso. Para quitarla podemos añadir código para a la migración inicial mediante el método **Sql** que ya conocemos.

```
public partial class InitialMigration : DbMigration
{
    public override void Up()
    {
        Sql("DROP TABLE EdmMetadata");
    }
}
```

Ahora ya tenemos todo lo necesario para poder ejecutar la primera migración. A partir de este momento podremos utilizar las migraciones automáticas o por código de la misma forma que hemos visto anteriormente.

9 Migrar desde código

He comentado antes que normalmente para entornos de producción tenemos que ejecutar un *script*, y hemos visto también que lo podíamos hacer mediante el parámetro **Script**, pero también podemos tener diferentes entornos de producción, cada uno con versiones distintas de base de datos. En estos casos deberíamos generar todos los *scripts* modificando la migración origen. Esto como nos podemos imaginar ni es práctico ni elegante.

Lo que podemos hacer en estas situaciones es utilizar la clase **DbMigrator**. De hecho los comandos que hemos estado utilizando desde la consola están llamando a esta misma clase.

Para poder ejecutar las migraciones desde código, tenemos que instanciar la clase **Configuration** que se añadió al proyecto cuando habilitábamos las migraciones. Esta instancia la pasamos en el constructor del objeto **DBMigrator**, y por último llamamos al método **Update** que actualizará la base de datos a la última migración.

```
var configuration = new Configuration();
var migrator = new DbMigrator(configuration);
migrator.Update("201203120011483_SegundaMigracion");
```

Figura 7 - Ejecutar una migración desde código

El método **Update** tiene una sobrecarga que acepta un parámetro para indicar la migración destino. En la Figura 7 podemos ver la sencillez con que podemos ejecutar una migración desde código, en este caso estamos actualizando a la migración. En este caso estamos pasando el identificador completo de la migración. También podemos especificar la base de datos donde queremos especificar las migraciones utilizando la propiedad **TargetDatabase** de la configuración.

10 Conclusiones

En este artículo hemos visto como EF 4.3 nos proporciona un mecanismo muy sencillo y versátil para actualizar la base de datos según los cambios hechos en nuestro modelo de entidades. Podemos dejar que EF realice los cambios automáticamente o mediante las migraciones basadas en código podemos personalizar la forma en que se migrarán. También hemos visto como podemos actualizar a una versión superior o inferior y como poder desplegar los cambios en nuestros entornos de pruebas o producción.