Code First Migrations

This walkthrough will provide an overview Code First Migrations in Entity Framework. You can either complete the entire walkthrough or skip to the topic you are interested in. The following topics are covered:

Building an Initial Model & Database

Before we start using migrations we need a project and a Code First model to work with. For this walkthrough we are going to use the canonical **Blog** and **Post** model.

Create a new **MigrationsDemo** Console application

Add the latest version of the **EntityFramework** NuGet package to the project

Tools –> Library Package Manager –> Package Manager Console

Run the **Install-Package EntityFramework** command

Add a **Model.cs** file with the code shown below. This code defines a single **Blog** class that makes up our domain model and a **BlogContext** class that is our EF Code First context

```
using System.Data.Entity;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity.Infrastructure;

namespace MigrationsDemo
{
    public class BlogContext : DbContext
    {
        public DbSet<Blog> Blogs { get; set; }
    }

    public class Blog
    {
        public int BlogId { get; set; }
        public string Name { get; set; }
    }
}
```

Now that we have a model it's time to use it to perform data access. Update the **Program.cs** file with the code shown below.

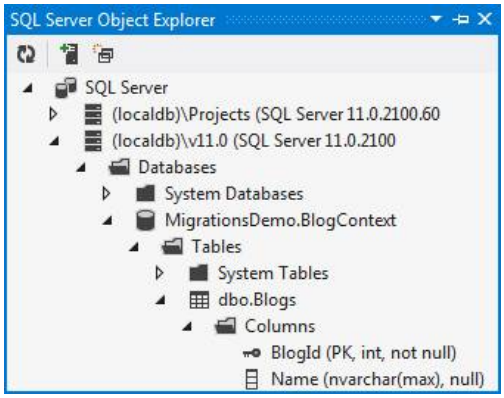
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MigrationsDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var db = new BlogContext())
            {
                db.Blogs.Add(new Blog { Name = "Another Blog " });
```

```
            db.SaveChanges();

            foreach (var blog in db.Blogs)
            {
                Console.WriteLine(blog.Name);
            }
        }

        Console.WriteLine("Press any key to exit...");
        Console.ReadKey();
    }
  }
}
```

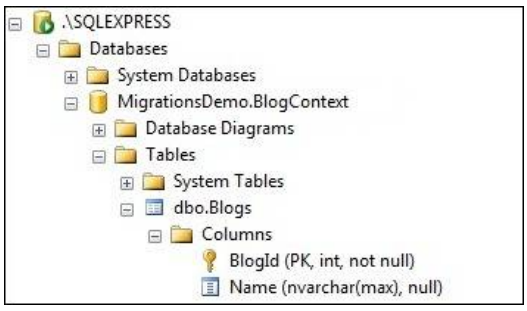
Run your application and you will see that a **MigrationsCodeDemo.BlogContext** database is created for you.

*If SQL Express is installed (included in Visual Studio 2010) then the database is created on your local SQL Express instance (**.\SQLEXPRESS**). If SQL Express is not installed then Code First will try and use LocalDb (**(localdb)\v11.0**) - LocalDb is included with Visual Studio 2012.*

***Note:** SQL Express will always get precedence if it is installed, even if you are using Visual Studio 2012*



(LocaDb Database)



(SQL Express Database)

Enabling Migrations

It's time to make some more changes to our model.

Let's introduce a Url property to the Blog class.

```
public string Url { get; set; }
```

If you were to run the application again you would get an InvalidOperationException stating *The model backing the 'BlogContext' context has changed since the database was created. Consider using Code First Migrations to update the database (http://go.microsoft.com/fwlink/?LinkId=238269).*

As the exception suggests, it's time to start using Code First Migrations. The first step is to enable migrations for our context.

Run the **Enable-Migrations** command in Package Manager Console

This command has added a **Migrations** folder to our project, this new folder contains two files:

The Configuration class. This class allows you to configure how Migrations behaves for your context. For this walkthrough we will just use the default configuration.
Because there is just a single Code First context in your project, Enable-Migrations has automatically filled in the context type this configuration applies to.

An InitialCreate migration. This migration was generated because we already had Code First create a database for us, before we enabled migrations. The code in this scaffolded migration represents the objects that have already been created in the database. In our case that is the **Blog** table with a **BlogId** and **Name** columns. The filename includes a timestamp to help with ordering.
If the database had not already been created this InitialCreate migration would not have been added to the project. Instead, the first time we call Add-Migration the code to create these tables would be scaffolded to a new migration.

Multiple Models Targeting the Same Database

When using versions prior to EF6, only one Code First model could be used to generate/manage the schema of a database. This is the result of a single **__MigrationsHistory** table per database with no way to identify which entries belong to which model.

Starting with EF6, the **Configuration** class includes a **ContextKey** property. This acts as a unique identifier for each Code First model. A corresponding column in the **__MigrationsHistory** table allows entries from multiple models to share the table. By default, this property is set to the fully qualified name of your context.

Generating & Running Migrations

Code First Migrations has two primary commands that you are going to become familiar with.

Add-Migration will scaffold the next migration based on changes you have made to your model since the last migration was created

Update-Database will apply any pending migrations to the database

We need to scaffold a migration to take care of the new Url property we have added. The **Add-Migration** command allows us to give these migrations a name, let's just call ours **AddBlogUrl**.

Run the **Add-Migration AddBlogUrl** command in Package Manager Console

In the **Migrations** folder we now have a new **AddBlogUrl** migration. The migration filename is pre-fixed with a timestamp to help with ordering

```
namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddBlogUrl : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Blogs", "Url", c => c.String());
        }

        public override void Down()
        {
            DropColumn("dbo.Blogs", "Url");
        }
    }
}
```

We could now edit or add to this migration but everything looks pretty good. Let's use **Update-Database** to apply this migration to the database.

Run the **Update-Database** command in Package Manager Console

Code First Migrations will compare the migrations in our **Migrations** folder with the ones that have been applied to the database. It will see that the **AddBlogUrl** migration needs to be applied, and run it.

The **MigrationsDemo.BlogContext** database is now updated to include the **Url** column in the **Blogs** table.

Customizing Migrations

So far we've generated and run a migration without making any changes. Now let's look at editing the code that gets generated by default.

It's time to make some more changes to our model, let's add a new **Rating** property to the **Blog** class

```
public int Rating { get; set; }
```

Let's also add a new **Post** class

```
public class Post
{
    public int PostId { get; set; }
    [MaxLength(200)]
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

We'll also add a **Posts** collection to the **Blog** class to form the other end of the relationship between **Blog** and **Post**

```
public virtual List<Post> Posts { get; set; }
```

We'll use the **Add-Migration** command to let Code First Migrations scaffold its best guess at the migration for us. We're going to call this migration **AddPostClass**.

Run the **Add-Migration AddPostClass** command in Package Manager Console.

Code First Migrations did a pretty good job of scaffolding these changes, but there are some things we might want to change:

1. First up, let's add a unique index to **Posts.Title** column
   (Adding in line 22 & 29 in the code below).
2. We're also adding a non-nullable **Blogs.Rating** column. If there is any existing data in the table it will get assigned the CLR default of the data type for new column (Rating is integer, so that would be **0**). But we want to specify a default value of **3** so that existing rows in the **Blogs** table will start with a decent rating.
   (You can see the default value specified on line 24 of the code below)

```
namespace MigrationsDemo.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddPostClass : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "dbo.Posts",
                c => new
                    {
                        PostId = c.Int(nullable: false, identity: true),
                        Title = c.String(maxLength: 200),
                        Content = c.String(),
                        BlogId = c.Int(nullable: false),
                    })
                .PrimaryKey(t => t.PostId)
                .ForeignKey("dbo.Blogs", t => t.BlogId, cascadeDelete: true)
                .Index(t => t.BlogId)
                .Index(p => p.Title, unique: true);

            AddColumn("dbo.Blogs", "Rating", c => c.Int(nullable: false, defaultValue: 3));
        }

        public override void Down()
        {
            DropIndex("dbo.Posts", new[] { "Title" });
            DropIndex("dbo.Posts", new[] { "BlogId" });
            DropForeignKey("dbo.Posts", "BlogId", "dbo.Blogs");
            DropColumn("dbo.Blogs", "Rating");
            DropTable("dbo.Posts");
        }
    }
}
```

Our edited migration is ready to go, so let's use **Update-Database** to bring the database up-to-date. This time let's specify the **–Verbose** flag so that you can see the SQL that Code First Migrations is running.

Run the **Update-Database –Verbose** command in Package Manager Console.

Data Motion / Custom SQL

So far we have looked at migration operations that don't change or move any data, now let's look at something that needs to move some data around. There is no native support for data motion yet, but we can run some arbitrary SQL commands at any point in our script.

Let's add a **Post.Abstract** property to our model. Later, we're going to pre-populate the **Abstract** for existing posts using some text from the start of the **Content** column.

```
public string Abstract { get; set; }
```

We'll use the **Add-Migration** command to let Code First Migrations scaffold its best guess at the migration for us.

Run the **Add-Migration AddPostAbstract** command in Package Manager Console.

The generated migration takes care of the schema changes but we also want to pre-populate the **Abstract** column using the first 100 characters of content for each post. We can do this by dropping down to SQL and running an **UPDATE** statement after the column is added.
(Adding in line 12 in the code below)

```
namespace MigrationsDemo.Migrations
{
    using System;
```

```
    using System.Data.Entity.Migrations;

    public partial class AddPostAbstract : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Posts", "Abstract", c => c.String());

            Sql("UPDATE dbo.Posts SET Abstract = LEFT(Content, 100) WHERE Abstract IS NULL");
        }

        public override void Down()
        {
            DropColumn("dbo.Posts", "Abstract");
        }
    }
}
```

Our edited migration looks good, so let's use **Update-Database** to bring the database up-to-date. We'll specify the **–Verbose** flag so that we can see the SQL being run against the database.

Run the **Update-Database –Verbose** command in Package Manager Console.

Migrate to a Specific Version (Including Downgrade)

So far we have always upgraded to the latest migration, but there may be times when you want upgrade/downgrade to a specific migration.

Let's say we want to migrate our database to the state it was in after running our **AddBlogUrl** migration. We can use the **–TargetMigration** switch to downgrade to this migration.

Run the **Update-Database –TargetMigration: AddBlogUrl** command in Package Manager Console.

This command will run the Down script for our **AddBlogAbstract** and **AddPostClass** migrations.

If you want to roll all the way back to an empty database then you can use the **Update-Database –TargetMigration: $InitialDatabase** command.

Getting a SQL Script

If another developer wants these changes on their machine they can just sync once we check our changes into source control. Once they have our new migrations they can just run the Update-Database command to have the changes applied locally. However if we want to push these changes out to a test server, and eventually production, we probably want a SQL script we can hand off to our DBA.

Run the **Update-Database** command but this time specify the **–Script** flag so that changes are written to a script rather than applied. We'll also specify a source and target migration to generate the script for. We want a script to go from an empty database (**$InitialDatabase**) to the latest version (migration **AddPostAbstract**).
If you don't specify a target migration, Migrations will use the latest migration as the target. If you don't specify a source migrations, Migrations will use the current state of the database.

Run the **Update-Database -Script -SourceMigration: $InitialDatabase -TargetMigration: AddPostAbstract** command in Package Manager Console

Code First Migrations will run the migration pipeline but instead of actually applying the changes it will write them out to a .sql file for you. Once the script is generated, it is opened for you in Visual Studio, ready for you to view or save.

Generating Idempotent Scripts (EF6 onwards)

Starting with EF6, if you specify **–SourceMigration $InitialDatabase** then the generated script will be 'idempotent'. Idempotent scripts can upgrade a database currently at any version to the latest version (or the specified version if you use **–TargetMigration**). The generated script includes logic to check the **__MigrationsHistory** table and only apply changes that haven't been previously applied.

Automatically Upgrading on Application Startup (MigrateDatabaseToLatestVersion Initializer)

If you are deploying your application you may want it to automatically upgrade the database (by applying any pending migrations) when the application launches. You can do this by registering the **MigrateDatabaseToLatestVersion** database initializer. A database initializer simply contains some logic that is used to make sure the database is setup correctly. This logic is run the first time the context is used within the application process (**AppDomain**).

We can update the **Program.cs** file, as shown below, to set the **MigrateDatabaseToLatestVersion** initializer for BlogContext before we use the context (Line 14). Note that you also need to add a using statement for the **System.Data.Entity** namespace (Line 5).

*When we create an instance of this initializer we need to specify the context type (**BlogContext**) and the migrations configuration (**Configuration**) - the migrations configuration is the class that got added to our **Migrations** folder when we enabled Migrations.*

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Entity;
using MigrationsDemo.Migrations;

namespace MigrationsDemo
{
```

```
        class Program
        {
            static void Main(string[] args)
            {
                Database.SetInitializer(new MigrateDatabaseToLatestVersion<BlogContext, Configuration>());

                using (var db = new BlogContext())
                {
                    db.Blogs.Add(new Blog { Name = "Another Blog " });
                    db.SaveChanges();

                    foreach (var blog in db.Blogs)
                    {
                        Console.WriteLine(blog.Name);
                    }
                }

                Console.WriteLine("Press any key to exit...");
                Console.ReadKey();
            }
        }
    }
```

Now whenever our application runs it will first check if the database it is targeting is up-to-date, and apply any pending migrations if it is not.