# EntityFramework6\_GettingStarted

The following is a list of sections in this document. Using Microsoft Word, you can use these as hyperlinks to navigate to any particular section. But using Apache Open Office, these hyperlinks do not work; instead, they merely serve as a table of contents. You can navigate to the start of any section via bookmarks; type F5 to bring up the Navigator; then double-click Bookmark1 for 1st section header, Bookmark 2, for 2nd section header, etc.

|  |  |
| --- | --- |
| [Overview - Introduction](#_Overview_-_Introduction) | [What’s in This Course?](#_What’s_in_This) |
| [What’s in This Module?](#_What’s_in_This_1) | [Entity Framework Goals](#_Entity_Framework_Goals) |
| [Where to Use Entity Framework 6](#_Where_to_Use) | [High Level Look at How EF Works](#_High_Level_Look) |
| [Visual or Code Based Models from Scratch or Existing Database](#_Visual_or_Code) | [Where EF Fits in Your Software Architecture](#_Where_EF_Fits) |
| [From Inception to EF6: A Short History](#_From_Inception_to) | [EF7 is Coming, But EF6 is Staying, Too](#_EF7_is_Coming,) |
| [Resources](#_Resources) | [Preparing the Sample Solution](#_Preparing_the_Sample) |
| [Creating an Entity Framework Model](#_Creating_an_Entity_1) | [Valildating Your EF Model](#_Validating_Your_EF) |
| [Fixing How EF Interprets Your Model](#_Fixing_How_EF) | [Using Code First Migrations to Create a Database](#_Using_Code_First) |
| [Migrating a Database When Your Model Changes](#_Migrating_a_Database) | r |
| s | t |

## Overview - Introduction

The Pluralsight instructor recommends EF Core 3.1, instead of EF 6, and she does present a Pluralsight course “EF Core: Getting Started” (published Jan 20, 2020 on Pluralsight). I decided to persist with EF 6, because EF Core 3.1 appears to be a companion to ASP.NET Core, and my education (at this point in time) is directed to .NET Framework, not .NET Core.

In contrast to the instructor of this course, the Pluralsight instructor for the course “ASP.NET MVC 5 Fundamentals” makes the point that ASP.NET Version 4.x (in the .NET Framework camp) is not obsolescent. Refer to [Choose between ASP.NET 4.x and ASP.NET Core](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/choose-aspnet-framework?view=aspnetcore-3.1). By contrast, he says, ASP.NET Core is well suited to developers who need to develop on macOS or Linux, whereas ASP.NET version 4.x provides the full .NET framework which includes Windows Communication Foundation (WCF), WebAPI, Web Pages, and others.

## What’s in This Course?

One starts with a model, which is expressed in code. Then combine this model with Entity Framework’s Db Context, and this combination becomes a tool to interact with a database. Finally we’ll explore how to fit this combination into various types of applications – client-side and server-side.

In this chapter of this course, we start by addressing the questions “What is Entity Framework?” and “Why use Entity Framework?”. This chapter will include a brief history of Entity Framework. This chapter will include a comparison of Entity Framework 6 and Entity Framework Core.

## What’s in This Module?

Entity Framework is an Object Relational Mapper (ORM). ORMs simplify the effort to express the interface between objects in the software with relational data in a database. ORMs create connections, create commands, execute commands on the database, create/execute queries, process results. More specifically “process results” includes

* read data from queries
* spin-up instances of “domain” classes
* push the data into these instances

(“domain” classes are C# classes that model data in a database.)

Entity Framework 6 is the ORM created by Microsoft that is part of the .NET development set.

## Entity Framework Goals

Entity Framework can enhance programmer productivity, because it eliminates a very large number of redundant data-interaction tasks. It also promotes consistency within a programmer team, because it reduces the need for individuals to design their own means of data access.

There are a very large number of Entity Framework users, and Microsoft is promoting its use. Therefore, we can expect Entity Framework to be around for a long time.

Rather than writing in one of several flavors of SQL, Entity Framework embodies a single language modeled after the LINQ syntax.

## Where to Use Entity Framework 6

Entity Framework is connected with the .NET Framework, starting with .NET 4. That means you can use Entity Framework 6 with

Client-side applications -- WPF, Windows Forms, Console Applications

and

Server-side applications -- ASP.NET MVC, ASP.NET Web API, ASP.NET Web Forms, WCF Services, WCF Data Services, Windows Services.

## High Level Look at How EF Works

The 1st step is to code the Domain classes. (“Domain” classes are C# classes that model data in a database.) The next step is to apply the Domain classes to the Entity Framework API, and then to instruct Entity Framework API how to map these classes to the database schema. This sets one up with the ability to write queries - in LINQ and Entity - into SQL that is understood by the database. Finally Entity Framework translates results (for example – of a SELECT query) into instances (or a LIST of instances) based on the Domain classes. Entity Framework has the capability to work with database VIEW’s and Stored Procedures, when the need arises. When the application (client or server) deletes or modifies these instances, Entity Framework keeps track of such changes, and it provides a facility (SaveChanges) to update the database accordingly.

## Visual or Code Based Models from Scratch or Existing Database

Entity Framework provides two approaches to define the database model (or schema). (1) You can describe the model with C# classes and additional code. (2) You can develop the visual model supported by the designer in Visual Studio (or with some 3rd-party tool). If one uses the designer, it creates an XML file referred to as an EDMX (Entity Data Model XML). Then the Visual Studio designer creates classes based on that EDMX. The alternative pathway – via C# classes - is more straightforward.

At run time . . .

via approach (2): if Entity Framework finds an EDMX, it translates the EDMX into an in-memory model, which specifies the interface between the visual model and the database.

via approach (1): if Entity Framework does not find an EDMX at runtime, but discovers a code-based model, then - using its Code-first API - it translates that model into the same type of in-memory model that was mentioned in the preceding paragraph.

After the in-memory model has been built Entity Framework behaves the same regardless of how the in-memory model was created.

How does the programmer create the model?

If the database already exists, Entity Framework provides a facility to reverse-engineer the database into either (1) a Visual Model (EDMX) or (2) C# classes. But if you want to make changes to the database, you have only one option – to update the EDMX; the option to update the C# code is not available. (The Pluralsight instructor did not elaborate on what she meant by the limitation of option (2). Stephen Cole’s guess is that that you could use option (2) to build an empty database – no data. But if you wanted to change the database and keep the existing data, you would have to uses option (1).)

If you start by using the Visual Studio designer, you can translate this into a database. But (if I understand the Pluralsight instructor correctly), you cannot use this approach to make database changes via Visual Studio designer.

The most popular approach is to start with C# code. This approach permits database migration. It would employ tools to build Transaction SQL, which can be used by the Database Administrator to update the database while preserving the existing data.

## Where EF Fits in Your Software Architecture

This clip describes the modules that comprise a typical application. The non-entity-framework modules are user interfaces, application logic, business logic, etc. The entity-framework modules are

* Domain Objects (described above)
* DbContext (one or more), each of which wraps multiple Domain Objects (see below)
* Data Logic - Entity Framework code that takes the place of T-SQL queries

DbContext is the base class defined by Entity Framework. In application code, the programmer creates one or more classes derived from DbContext. Each of these classes contains code that describe the database tables.

## From Inception to EF6: A Short History

The history narrative from EF1 in 2008 through EF6 in 2016 is self-explanatory. Very few changes have been made by Pluralsight to this course since 2016.

## EF7 is Coming, But EF6 is Staying, Too

The subtitle for this Clip is “Should I Be Using EF6 or EF Core?” The development of EF Core started in 2016, and its original name was Entity Framework 7. The instructor relayed Microsoft’s advice that for new development, one should be using EF Core instead of EF6. (But as I said earlier, I am concerned that working with EF Core would entail working with .NET Core.) If the situation arises that requires migrating an application from EF6 to EF Core, it might be useful to read an article by Mikael Eliasson (June 1, 2019), “Migrating from EF6 to EF Core” (see [Resources](#_Resources) below)

## Resources

|  |  |
| --- | --- |
| EF6 and EF Core Official Docs | docs.Microsoft.com/ef |
| EF6 Development Site | github.com/dotnet/ef6 |
| EF6 Ninja Edition: What’s New in EF6 (Pluralsight) | bit.ly/PS-EF6 |
| My Blog | thedatafarm.com/blog |
| Announcing Entity Framework Core 3.1 and Entity Framework 6.4 | bit.ly/336yGGI |
| Mikael Eliasson EF6 to EF Core post | mikee.se/posts/migrating\_from\_ef6\_to\_ef\_core |

## Preparing the Sample Solution

Start by creating an ordinary class library.

* Prepare a folder in which to work. I created a subfolder VsAndOtherPlatformProjects\ENTITY\_FRAMEWORK\_6.
* Start Visual Studio. Choose the template “Class Library (.NET Framework)”; select **C#** if it has not been selected automatically; then click the **Next** command button. Visual Studio delivers a pop-up dialog **Configure your new project**.
* Type the name “NinjaDomain.Classes” into the **Project name** text box.
* Use the **…** to the right of the **Location** combo box, and select the folder that you prepared (above).
* Type the name “Ninja.Module2Demo” into the **Solution name** text box.
* Leave the “Place solution and project in the same directory” check box **unchecked**.
* Click the **Create** command button. Visual Studio creates the solution and project.

You can use a file browser to verify that ENTITY\_FRAMEWORK\_6 contains a subfolder Ninja.Module2Demo, which – in turn – contains the subfolder NinjaDomain.Classes and the file Ninja.Module2Demo.sln. Also, NinjaDomain.Classes should contain

(1) 3 subfolders – bin, obj, and Properties

(2) 2 files – Class1.cs and NinjaDomain.Classes.csproj

Examine the code for Class1.cs - created by default. It contains the code for a single class named “Class1”. What we want, instead, is 3 classes

* Ninja
* Clan
* EquipmentType

and (for this demo) we are satisfied to code all 3 of these in a single .cs file. In the **Solution Explorer** rename Class1.cs to Classes.cs.

Remove the automatically generated code

public class Class1

{

}

or

public class Classes

{

}

and replace it with the code for the three classes.

public class Ninja

{

public int Id { get; set; }

public string Name { get; set; }

public bool ServedInOniwaban { get; set; }

public Clan Clan { get; set; }

public int ClanId { get; set; }

public List<NinjaEquipment> EquipmentOwned { get; set; }

}

public class Clan

{

public int Id { get; set; }

public string ClanName { get; set; }

public List<Ninja> Ninjas { get; set; }

}

public class NinjaEquipment

{

public int Id { get; set; }

public string Name { get; set; }

public EquipmentType Type { get; set; }

public Ninja Ninja { get; set; }

}

Also, remove the unused **using** statements.

Following the practice of the Pluralsight instructor, we will keep all Enum’s in a single folder.

* Under NinjaDomain.Classes add a new folder via -- right-click | **Add** | **New Folder**. Name it “Enums”.
* Under Enums, add a new class via -- right-click | **Add** | **New Item...** |Visual C# **Class** | Name it “EquipmentType.cs” | **Add** command button. Visual Studio creates the EquipmentType class in EquipmentType.cs with n**amespace** NinjaDomain.Classes.Enums.
* We don’t really want the automatically generated code

public class EquipmentType

{

}

Replace it with the code for our **enum**

public enum EquipmentType

{

Tool = 1,

Weapon = 2,

Outwear = 3

}

* Again, remove the unused **using** statements.
* Repeat the steps above to add a 2nd **enum** - NinjaType.cs. Use the following as the body of this **enum**.

public enum NinjaType

{

Shinobi=1,

Kunoichi=2

}

Take another look at Classes.cs. Notice the red squiggly line under EquipmentType. The reason for this is that the **namespace** statements in our **enum** modules are NinjaDomain.Classes.Enums. To resolve this remove “.Enums” from the **namespace** statements, so that they have the same **namespace** declaration as that of Classes.cs. (Before now it never occurred to me that one might have multiple .cs files with the same **namespace** declaration. And when this is the case, evidently one doesn’t need **using** statements to refer to a **class** or **enum** in the other .cs file.)

(The Pluralsight instructor confessed that “enum NinjaType” is not being used. She needed it originally, and it subsequently became obsolete. She is leaving it there – not causing any harm.)

## Creating an Entity Framework Model

In the preceding clip we created domain classes. They describe tables that we plan to create via Entity Framework. It should be noted that each Ninja is associated with exactly one clan (many/1 relationship). (We don’t yet know why the Ninja class contains both a Clan and a ClanId property. This question is addressed in [Fixing How EF Interprets Your Model](#_Fixing_How_EF).) Secondly each Ninja is associated with a list of NinjaEquipment instances - Ninja.EquipmentOwned. The NinjaEquipment.Ninja property prevents equipment instances from being shared among multiple Ninjas.

Up until now Entity Framework has not been used. But we will start doing so in a new project.

* In the Solution Explorer under Solution ‘Ninja.Module2Demo’ -- right-click | **Add** | **New Project...** | C# Class Library (.NET Framework) | **Next** command button | “NinjaDomain.DataModel” for **Project name** | **Create** command button. Visual Studio builds the new source file Class1.cs.
* Now we bring in the Entity Framework API. Right-click NinjaDomain.DataModel in the Solution Explorer. Then click **Manage NuGet Packages...** . Visual Studio pops up a dialog titled **NuGet Package Manager: NinjaDomain.DataModel**.
* Click **Browse** in the upper-left corner of the dialog. Type “Entity Framework” in the **Search** box (below **Browse**). Visual Studio displays a list of packages pertaining to Entity Framework.
* Click the package titled “Entity Framework” with “Entity Framework 6” in the description. Visual Studio displays a description of the package in the right-hand panel of the dialog. It shows that version 6.4.4 is the latest stable version (as of 1/25/21).
* Click the **Install** command button. Visual Studio displays a **Preview Changes** pop-up. Click the **OK** command button. Visual Studio displays a **License Acceptance** pop-up. Click the **I Accept** command button.
* Visual Studio performs the installation, and you should see notes in Visual Studio’s output window.
* You can confirm that Entity Framework is installed by expanding the node NinjaDomain.DataModel | References. You should see “EntityFramework” and “EntityFramework.SqlServer”.

As another consequence of installing Entity Framework, the package manager created App.config. In addition to other configuration settings, App.config contains an <EntityFramework> section, and it makes a reference to **LocalDB**.

***The code shown in this Pluralsight course contains <defaultConnectionFactory> as a subsection of <EntityFramework>. <defaultConnectionFactory> is not present in App.config on my computer, and there is no mention of LocalDB. My guess is that the reason for this is that the Entity Framework NuGet package that I installed is at a later version (6.4.4) than the package installed by the Pluralsight instructor (6.1.3). Since I have successfully used version 6.4.4 in other Pluralsight courses, and since Entity Framework worked successfully with LocalDB when I shadowed these courses, I doubt that the discrepancy will cause any problems.***

Refer to NinjaDomain.DataModel/Class1.cs which we created earlier.

* Change the class name “Class1” to “NinjaContext:DbContext”. Use Visual Studio’s automated facility to add the using statement

using System.Data.Entity;

* Also, remove the unused **using** statements.
* To specify the interface between domain classes and Entity Framework, we use the Generic DbSet<*DomainClass*>. As the name “Set” suggests, DbSet<*DomainClass*> specifies a set of instances of type *DomainClass*. We will need one DbSet<> for each of the 3 domain classes that we defined. The body of class NinjaContext should, therefore, contain

public DbSet<Ninja> Ninjas {get; set;}

public DbSet<Clan> Clans { get; set; }

public DbSet<NinjaEquipment> Equipment { get; set; }

(Hover the mouse over the red-squiggly underscores, and follow Visual Studio suggestions to provide the appropriate **using** statement.)

## Validating Your EF Model

***If you watch the Visual Studio Solution Explorer during this Pluralsight clip, you might notice that*** NinjaDomain.DataModel.Class1.cs ***changes to*** NinjaDomain.DataModel.NinjaContext.cs ***(which is appropriate in light of the name of the class that it contains). The Pluralsight instructor probably assumed that we would know to rename it. To shadow this course carefully, right-click*** Class1.cs ***and use*** Rename ***in the pull-down menu to rename the file.***

The Pluralsight instructor recommends a tool that one can use to test whether the Domain Classes will produce the Entity Data Model that we hoped they would specify. I’m not sure what the instructor means by the “Entity Data Model”, but to me the model appears very much like a database schema.

* Open the pull-down menu **Extensions** in the top row of Visual Studio. Click **Manage Extensions**. Visual Studio displays a pop-up dialog titled **Manage Extensions.**
* Click **Installed** in the leftmost panel of the **Manage Extensions** dialog, and look for **Entity Framework 6 Power Tools Community Edition**. If you find it, you can skip the next 3 steps.
* Click **Online | Visual Studio Marketplace** in the leftmost panel of the dialog.
* Type “Entity” in the Search box in the upper right corner of the dialog. You should be able to find an entry in the center pane of the dialog titled **Entity Framework 6 Power Tools Community Edition**.
* Click **Entity Framework 6 Power Tools Community Edition**, and click the **Download** button that becomes visible.

To use this tool to display the Entity Data Model, you must first set NinjaDomain.DataModel as the start-up project. (Right-click NinjaDomain.DataModel in the **Solution Explorer**, and click **Set as Startup Project** in the pop-up menu.)

Right click the module that contains the **DbContext** class, NinjaContext.cs. Then – in the pop-up menu – click **Entity Framework 6 | View Entity Data Model (Read-only)**. There will be a delay (perhaps 30 seconds), and then Visual Studio will display the **Entity Data Model**.

The **Entity Data Model** which is displayed shows a “0..1 -> \*” from the Ninja class to the Ninja Equipment class, i.e. 0 or 1 Ninja related to many pieces of Ninja Equipment. This allows the possibility of Ninja Equipment instances that do not belong to any Ninja. This is not what we intended. Instead, we want it to show “1 ->\*”.

## Fixing How EF Interprets Your Model

Examine – in the domain classes (Classes.cs) - the properties that link Clan with Ninja and Ninja with Ninja Equipment. If you look closely you will note that they are not done in the same way; the Ninja class has both a Clan property and a ClanId property. (ClanId translates into a **Foreign Key** in the Entity Data Model, and my guess is that Entity Framework does this based on the name “ClanId”.) On the other hand the NinjaEquipment class has only the Ninja property; there is no NinjaId property. There is the possibility that the Ninja property could take the value **null**; then any instance of NinjaEquipment with Ninja == **null** would have 0 related Ninjas. But if we introduce the **Foreign Key** NinjaId, the possibility of 0 Ninjas disappears, because the **integer** NinjaId is non-nullable.

The Pluralsight instructor said that the redundancy between Ninja.Clan and Ninja.ClanId was introduced to make coding easier. But, on the other hand, the instructor did not need NinjaEquipment.NinjaId, because the code that dealt with this relationship required only the list Ninja.EquipmentOwned. Nevertheless, the instructor strongly encourages including the **Foreign Key** NinjaEquipment.NinjaId. Admittedly its absence will not cause any problem in a client-side application (e.g. a Console application or a Windows Communication Foundation application). But in Web applications – where things are disconnected – the problem will become obvious. For now the instructor suggests leaving the class without this **Foreign Key**, because later in the course we will see the consequences of not having this **Foreign Key**.

I tried an experiment by adding the **Foreign Key**

public int NinjaId { get; set; }

to the properties of class NinjaEquipment; then I repeated right-clicking NinjaContext.cs (in the Solution Explorer), followed by clicking **Entity Framework 6 | View Entity Data Model (Read-only)** in the pop-up menu. The experiment induced the linkage in **Entity Data Model** to change from “0..1 -> \*” to “1 -> \*”, which confirmed what the Pluralsight instructor said. The experiment is reversable; I simply deleted the NinjaId property from NinjaEquipment.

There is an easy way to ensure that every instance of NinjaEquipment belongs to a Ninja without the need to introduce the **Foreign Key**; introduce the data annotation “[Required]” as a line of code immediately before the property

public Ninja Ninja { get; set; }

(We also need “using System.ComponentModel.DataAnnotations;”.) Entity Framework interprets this to mean “non-nullable”. Note that the linkage in **Entity Data Model** changed from “0..1 -> \*” to “1 -> \*”.

There is another way, which is useful when the concern is with the relationship to the database, and when you prefer not to change the business rules in the domain classes. One can configure mappings with Entity Framework’s “Fluent API”, and this code is introduced in the class derived from **DbContext**. But using the data annotation is simpler, especially for getting started with Entity Framework.

## Using Code First Migrations to Create a Database

Entity Framework provides a number of ways to create (or migrate) a database from the model. Some of them are automated and can happen at run-time. The Pluralsight instructor prefers to avoid automated migration except when performing unit-testing. The alternative is a design-time feature that maximizes consistency and that provides the most control.

The terminology “Database Migration” means bringing the database up-to-date to keep it consistent with changes to the model. (These model changes would have been done, for example, to meet evolving requirements.) The Entity Framework API does this by comparing the current model with its previous incarnation; it then uses the difference to make the appropriate changes to the database. One can confirm that the DbMigration API is available by viewing it in the **Object Browser**. But having used **Entity Framework 6 Power Tools Community Edition** in the previous 2 sections, you can be confident that the DbMigration API is available, and the following steps to view this API in the **Object Browser** are optional.

* In Visual Studio click the **View** menu (near the upper-left corner); then click **Object Browser** in the pull-down menu. Visual Studio displays the **Object Browser** in the center pane.
* Type “DbMigration” into the **Search** combo box – near the top of the center pane. Then type the **Enter** key. Visual Studio displays the list of methods for the database-migrations API in the right half of the center pane.

In response to commands (entered into the **Package Manager Console**), the Migrations API can create the appropriate SQL – as a script file - to bring the database up-to-date, or to create a new database.

This Pluralsight course will demonstrate (1) enable Migrations, (2) use Migrations to create a new database, (3) make changes to the model, (4) use Migrations to make comparable changes to the database.

* In Visual Studio introduce the **Package Manager Console** by clicking (starting at the menu bar) **Tools | NuGet Package Manager | Package Manager Console**. In response Visual Studio displays **Package Manager Console**.
* In the top row of **Package Manager Console** Select nuget.org in **Package source:** and NinjaDomain.DataModel in **Default project:**.
* NinjaDomain.DataModel was set as the start-up project in [Valildating Your EF Model](#_Validating_Your_EF), and I have not changed this setting. I do not know whether this is a prerequisite for using Migrations, but it works with this setting.

At the **PM>** command prompt type

“enable-migrations” followed by the **Enter** key.

When Migrations does not find an existing database, it responds by creating a new folder titled “Migrations” and a new source file Configuration.csin the Migrations folder. The only configuration in this class is specified in the constructor Configuration(). This specifies that – by default – automatic migrations are suppressed. (The Pluralsight instructor stated an emphatic warning that automatic migration should be avoided.) The Seed() method in this class is a placeholder for inserting instructions that will push start-up data into the database – immediately after migration.

At the **PM>** command prompt type

“add-migration Initial” followed by the **Enter** key.

The parameter “Initial” is the name of the migration, and this parameter is required. Of course, it is good practice for the name to be descriptive. When Migrations finds neither an existing database nor a previous migration, it builds a clean-slate migration. The result is a new file named <some 15-digit integer>\_Initial.cs. The code in this file is almost readable if one is familiar with SQL databases. Starting from the beginning of the Up() method . . .

* Create a “dbo.Clans” table with two columns, Id and ClanName. Id is an “identity” integer. ClanName is a character string. Id also serves as the primary key for this table.
* Similarly create a “dbo.Ninjas” table. ClanId is a foreign key with cascading-delete constraint. ClanId also serves as a 2nd index.
* Similarly create a “dbo.NinjaEquipments” table.

To produce T-SQL code from <some 15-digit integer>\_Initial.cs, at the **PM>** command prompt type

“update-database -script” followed by the **Enter** key.

Migrations responds by building a text file containing the T-SQL code, which could be used to create the database. But we are not going to use the generated T-SQL code at this stage. Therefore, close (and do not save) the file of T-SQL code.

Instead, at the **PM>** command prompt type

“update-database -verbose” followed by the **Enter** key.

This tells Migrations to build the database from <some 15-digit integer>\_Initial.cs. “-verbose” is optional; it tells Migrations to display the list of steps while performing the build. Refer to the statement in the verbose text starting with “Target database is:”. Migrations built a **localdb** database by default; the name of this database is NinjaDomain.DataMode.NinjaContext. The Pluralsight instructor admitted that she normally supplies an explicit connection string instead of letting Migrations choose a default name. (Perhaps she will demonstrate doing that later in this course.) Notice the T-SQL code in the verbose text. Finally notice – toward the end of the verbose text – code to create an extra table [dbo].[\_\_MigrationHistory]. This table holds code that describes the most recent migration. Migrations uses this code in the subsequent migration – after the model has evolved – to calculate how to bring the database up-to-date.

The structure of the database that was just created can be seen via the **SQL Server Object Explorer**. Drill down to a **localdb** database named NinjaDomain.DataMode.NinjaContext. You can review the tables with their columns, keys, and indexes.

## Migrating a Database When Your Model Changes

Our next step is to make a change to the Model, and demonstrate how to bring the database up-to-date to make it comparable to the Model.

* Navigate to Classes.cs (containing our Domain Classes).
* Add - to the Ninja class – the date-of-birth

public System.DateTime DateOfBirth { get; set; }

* To satisfy my curiosity, I tried to refresh the **Entity Data Model** described in[Valildating Your EF Model](#_Validating_Your_EF). I repeated the steps described in[Valildating Your EF Model](#_Validating_Your_EF), and the date-of-birth column showed up in “Ninja”, just as I expected. This exercise is, of course, not necessary. But when the change is more extensive, it is probably appropriate.
* Navigate to the **Package Manager Console** - using the instructions in [Using Code First Migrations to Create a Database](#_Using_Code_First).
* In the top row of **Package Manager Console** Select nuget.org in **Package source:** and NinjaDomain.DataModel in **Default project:**.

At the **PM>** command prompt type

“add-migration AddBirthdayToNinja” followed by the **Enter** key.

Migrations finds an existing database, and it builds a 2nd DbMigration class based on the difference between the existing schema (in table dbo.\_\_MigrationHistory) and the model inferred from the Domain Classes. The result is a new file named <some 15-digit integer>\_AddBirthdayToNinja.cs. The code in the Up() method of class AddBirthdayToNinja specifies adding a new column of type DateTime to dbo.Ninjas - named “DateOfBirth”.

The DbMigration classes have both an Up() method and a Down() method. The Down() method is available (if needed) to reverse the action in building or updating a database.

Let us apply the 2nd migration. Navigate back to the **Package Manager Console.** At the **PM>** command prompt type

“update-database -verbose” followed by the **Enter** key.

Look at the text from the “verbose” option – specifically at the point where Migrations is rebuilding [dbo].[\_\_MigrationHistory]. Shortly after this point Migrations refers to the 2nd DbMigration class named AddBirthdayToNinja. You can also refer to the database in **SQL Server Object Explorer** to observe the new column in dbo.Ninjas.