# 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_1) | [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) | [Validating 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) | [Creating Visual and Code Models from Existing Databases](#_Creating_Visual_and) |
| [Creating a Code-based Model and Database - Resources](#_Creating_a_Code-based) | [Inserting Objects](#_Inserting_Objects) |
| [Querying Simple Objects](#_Querying_Simple_Objects) | [Updating Modified Objects](#_Updating_Modified_Objects) |
| [Retrieving Data with the Find and SqlQuery Methods](#_Retrieving_Data_with) | [Deleting Simple Objects](#_Deleting_Simple_Objects) |
|  |  |

[Appendix I: Program.cs – Contains EF Methods for Interacting with Data](#_Appendix_I:_Program.cs_2)

## 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) an **Entity Data Model** (visual - EDMX) or (2) C# classes. But if you want to make changes to the database, you have only one option – to update the **Entity Data Model**; 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 use 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 intended to specify in the Domain Classes. We will see an example of an **Entity Data Model** later in this section; it resembles a visual 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 reversible; 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 [Validating 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[Validating Your EF Model](#_Validating_Your_EF). I repeated the steps described in[Validating 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 for a minor change. 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.

## Creating Visual and Code Models from Existing Databases

When you are working with an existing database, which was not built with Entity Framework, the EF Designer provides facilities to reverse engineer - to create the model or to create domain classes from the database. The notes in this clip give a brief description of this facility, and more advanced Pluralsight courses go into greater detail.

To shadow this clip I needed a small sample database. The database that I obtained for this purpose is provided at <https://www.w3resource.com/sql/sql-table.php>. This example delivers the table in the form of 6 sections of text. Each section contains T-SQL code; 3 of these create 3 tables; the other 3 populate data into the tables. On my computer I have installed SQL Server Developer. I used the following steps to set up this small-sample database under SQL Server Developer.

1. In the Visual Studio **SQL Server Object Explorer,** SQL Server Developer is denoted by a node labeled

*ComputerName* (SQL Server *VersionNumber* . . . )

1. Expand this node. Visual Studio displays a node titled **Databases**.
2. Right-click the **Databases** node. Visual Studio displays a pop-up menu.
3. Click **Add New Database** in the menu. Visual Studio displays a new node under **Databases** titled “New Database”.
4. Rename “New Database” to an appropriate name. I typed “SmallExample”.
5. Right-click the node representing the newly created database. Visual Studio displays a pop-up menu.
6. Click **New Query...** In the menu. Visual Studio opens an SQL-code window in the center panel.
7. For each of the 6 sections of text . . .
   1. Select the text from the web site, and type <ctrl>C to copy it into the clipboard.
   2. Paste the text (via <ctrl>V) into the SQL-code window.
   3. Replace all instances of “NUMBER” with “DECIMAL”. Replace all instances of “VARCHAR2” with “VARCHAR”.
   4. Click the right-arrow icon in the upper-left corner of the SQL-code window. Visual Studio executes the SQL code – either creating a table or populating a table.

To create an **Entity Data Model** (EDMX) from the database . . .

* Follow the steps at the beginning of [Creating an Entity Framework Model](#_Creating_an_Entity_1) to create a new class-library project titled EDMXFromDB.
* Click **Tools** in the Visual Studio menu bar. Visual Studio displays a pull-down menu. Click **Connect to Database...** In the menu. Visual Studio displays a pop-up dialog titled **Add Connection**.
* Make certain that “Microsoft SQL Server (SqlClient)” is displayed in the **Data Source:** text box. Type “.” into the **Server name:** text box. Visual Studio responds by enabling the **Select or enter the database name:** radio button.
* Select the database name (e.g. SmallExample) in the combo box below the radio button. Click the **OK** command button. Visual Studio confirms the database connection by displaying the database name in the left panel, **Server Explorer**, under the **Data Connections** node.
* Right-click EDMXFromDB in the **Solution Explorer**, and select **Add | New Item...** In the pop-up menu. Visual Studio displays the **Add New Item** dialog.
* Ensure that **Visual C# Items** is selected in the leftmost panel. Find and click **ADO.NET Entity Data Model** in the center panel. Type an appropriate name into the **Name** textbox at the bottom of the dialog. The Pluralsight instructor was working with an Adventure Works database, and she selected “AWModel”. My database is named “SmallExample”; therefore, I have selected “SEModel”. Then click the **Add** command button. Visual Studio displays the **Entity Data Model Wizard** in place of the **Add New Item** dialog.
* From the 1st page of the wizard, select the option titled **EF Designer from Database**. Then click the **Next** command button. The wizard goes to the next step titled **Choose Your Data Connection**.
* If you have a connection with more than one database, choose the database from the pull-down list box near the top of the dialog. Otherwise, the database will already be displayed. Click the **Next** command button. Visual Studio displays the next step of the wizard, **Choose Your Version**.
* I recommend the default, Entity Framework 6.x. Click the **Next** command button. Visual Studio displays the last step of the wizard, **Choose Your Database Objects and Settings**.
* This last step of the wizard permits some simplification if you don’t want to import everything. You can select which tables, views, or stored procedures you want to import.
* Click the **Finish** command button to exit the wizard. Visual Studio displays the **Entity Data Model** (EDMX) for my selected database (SmallExample). (Earlier in this course we saw an **Entity Data Model** for NinjaDomain.DataModel.NinjaContext in [Validating Your EF Model](#_Validating_Your_EF).)

There are other actions that resulted from completing the wizard. Expand the EDMXFromDB.**References** node, and observe that **EntityFramework** and **EntityFramework.SqlServer** have been added. There are two “.tt” files and a “.Designer.cs” file under EDMXFromDB; the “.tt” files are template files. The designer triggers a code generator that reads the template files and XML behind the model; the result of this process is the generation of classes. The Pluralsight instructor gives a brief survey of these automatically generated files, but I was unable to follow the discussion. I could not find DbContext classes nor XML. It appears that Entity Framework 6 has evolved, and that some of these automatically generated files are no longer used.

To create a code-first(domain classes) model from the database . . .

* Follow the steps at the beginning of [Creating an Entity Framework Model](#_Creating_an_Entity_1) to create a new class-library project titled CodeModelFromDB.
* Under project CodeModelFromDB in the **Solution Explorer**, rename Class1.cs to SEModel.cs.
* Click **Tools** in the Visual Studio menu bar. Visual Studio displays a pull-down menu. Click **Connect to Database...** In the menu. Visual Studio displays a pop-up dialog titled **Add Connection**.
* Make certain that “Microsoft SQL Server (SqlClient)” is displayed in the **Data Source:** text box. Type “.” into the **Server name:** text box. Visual Studio responds by enabling the **Select or enter the database name:** radio button.
* Select the database name (e.g. SmallExample) in the combo box below the radio button. Click the **OK** command button. Visual Studio confirms the database connection by displaying the database name in the left panel, **Server Explorer**, under the **Data Connections** node.
* Right-click CodeModelFromDB in the **Solution Explorer**, and select **Add | New Item...** In the pop-up menu. Visual Studio displays the **Add New Item** dialog.
* Ensure that **Visual C# Items** is selected in the leftmost panel. Find and click **ADO.NET Entity Data Model** in the center panel. Type an appropriate name into the **Name** textbox at the bottom of the dialog. The Pluralsight instructor was working with an Adventure Works database, and she selected “AWModel”. My database is named “SmallExample”; therefore, I have selected “SE\_Model”. Then click the **Add** command button. Visual Studio displays the **Entity Data Model Wizard** in place of the **Add New Item** dialog.
* From the 1st page of the wizard, select the option titled **Code First from Database**. Then click the **Next** command button. The wizard goes to the next step titled **Choose Your Data Connection**.
* If you have a connection with more than one database, choose the database from the pull-down list box near the top of the dialog. Otherwise, the database will already be displayed. Click the **Next** command button. Visual Studio displays the next step of the wizard, **Choose Your Database Objects and Settings**.
* This last step of the wizard permits some simplification if you don’t want to import everything. You can select which tables or views you want to import.
* Click the **Finish** command button to exit the wizard. The Wizard creates code-first classes. SE\_Model.cs contains the class SE\_Model, which is derived from DbContext. SE\_Model.cs also contains some additional configuration in the OnModelCreating() method. Note the other classes, whose names were copied from the table names in the database.

***I used the instructions in*** [Validating Your EF Model](#_Validating_Your_EF) ***to build an*** **Entity Data Model** ***from the classes (2nd part of this clip), and I was gratified to find that the diagram was the same as the .EDMX diagram produced in the 1st part of this clip.***

## Creating a Code-based Model and Database - Resources

The Pluralsight instructor displayed the following list of Resources.

**EF6 Development Site:** entityframework.codeplex.com

**EF6 Ninja Edition:** What’s New in EF6 (Pluralsight): bit.ly/PS-EF6

**Looking Ahead to EF7 (Pluralsight):** bit.ly/PS-EF7Alpha

**EF7 Development Site:** github.com/aspnet/entityframework

**My Blog:** thedatafarm.com/blog

**EF Team Blog:** blogs.msdn.com/adonet

## Inserting Objects

The steps to add a console application to our solution (the solution that has the Entity Framework classes) are as follows.

* Right click the solution (NinjaModule2Demo). Visual Studio displays a pop-up menu. Click **Add | New Project...**. Visual Studio displays a pop-up dialog titled **Add a new project.**
* In the upper-right corner select **C#** as the language, **Windows** as the platform, and **Console** as the project type. Windows responds by significantly reducing the number of options in the right-hand pane of the dialog.
* Click **Console App (.NET Framework)**. Then click the **Next** command button. Windows displays the dialog titled **Configure your new project**.
* Change the **Project name** to ConsoleApplication. **Location** should refer to the same folder that contains the Entity Framework classes. Click the **Create** command button.

Induce project ConsoleApplication to have a **Reference** to the Entity Framework classes: right-click ConsoleApplication.References; Visual Studio displays a pop-up menu; select **Add Reference...**; Visual Studio displays a pop-up dialog titled **Reference Manager – *ProjectName.*** Check the boxes next to NinjaDomain.Classes and NinjaDomain.DataModel; finally click the **OK** command button.

We also need ConsoleApplication to have a **Reference** to the Entity Framework API. In [Creating an Entity Framework Model](#_Creating_an_Entity_1) we used the **NuGet Package Manager** dialog. I am confident that we could repeat those steps. Instead, the Pluralsight instructor shows an alternative using the **Package Manager Console**, which was introduced in [Using Code First Migrations to Create a Database](#_Using_Code_First). Start the **Package Manager Console**. In the **Package Manager Console** dialog, set **Package source:** to nuget.org, and set **Default project:** to ConsoleApplication.

At the **PM>** command prompt type

“install-package EntityFramework” followed by the **Enter** key.

The response from the **Package Manager** is similar to what we saw in [Creating an Entity Framework Model](#_Creating_an_Entity_1); **Package Manager** added (1) references “EntityFramework” and “EntityFramework.SqlServer” and (2) ConsoleManager/App.config. We’ll keep this instance of App.config and delete NinjaDomain.DataModel/Migrations/App.Config.

Our database, NinjaDomain.DataMode.NinjaContext, contains no data. This is how projects normally start, and the data are added by application programs that we build. In this instance, however, we deviate, and manually add one piece of data to the dbo.Clans table; this will simplify the process of learning how to develop code-first software to work with data -- extract, modify, write, remove etc. We use Visual Studio’s **SQL Server Object Explorer** to add one record to dbo.Clans, which contains “Vermont Ninjas” in the ClanName column.

* In **SQL Server Object Explorer** navigate the tree to find the node labeled ”dbo.Clans”.
* Right-click this node, and click **View Data** in the pop-up menu. Visual   
  Studio displays the table in the center pane.
* Select the box under “ClanName” on the 1st row. Type “Vermont Ninjas” followed by the ENTER key.

We now turn our focus to ConsoleApplication/Program.cs. Program.cs contains code that invokes Entity Framework Code First instructions as an interface to our database. It, therefore, contains a substantial amount of code. To minimize the amount of typing I decided to copy the body of Program.cs from the Pluralsight results for this course into (1) the appendix of this document, [Appendix I: Program.cs – Contains EF Methods for Interacting with Data](#_Appendix_I:_Program.cs), and (2) the body of Program.cs in Visual Studio.

Program.cs contains an executive method - Main() - and several worker methods – InsertNinja(), InsertMultipleNinjas(), SimpleNinjaQueries(), etc.

***Observe that all of the worker methods contain the following pattern of C# code. Being unfamiliar with “***using()***” in this context, I investigated. What I found was that*** using ***declares and instantiates an*** IDisposable ***object and specifies its scope - the*** using ***block. The*** using ***statement ensures that if an exception occurs within this block, then*** Dispose ***or*** DisposeAsync ***is called. It is recommended practice to employ*** using ***in this fashion for*** DbContext ***objects.***

using (var context = new NinjaContext())

{

// code for interacting with the database

}

We start with InsertNinja(), whose purpose is to add one row to the table dbo.Ninjas. The first part of InsertNinja() builds a single Ninja object. The last part contains Entity Framework statements that interact with the database via a NinjaContext object.

*context.Ninjas.Add(ninja);* says add the Ninja object to the Ninjas DbSet. But execution of this addition is deferred until we invoke the SaveChanges() DbContext method.

*context.SaveChanges();* says execute the action that was prescribed earlier.

*context.Database.Log = Console.WriteLine;* is issued at the beginning of these Entity Framework statements – to induce EF to write text to the database’s log that describes the actions that will be performed between now and the *SaveChanges()* statement.

Click the green triangle (labeled “Start”) in Visual Studio’s toolbar to build and run the program. The response is a console window that shows what the program has done. Text in this window describes the activity performed on the database. By default the start of this text contains SQL that describes the state of the database and migration history. The more relevant part of the history is near the end. The history shows that after Entity Framework added the Ninja row, it issued a SELECT command to return the “identity” ID that was automatically generated. Examine the database via the **Server Object Explorer**, and observe that the Ninja row was added correctly.

Remove the comment from the 1st statement in Main() (*Database.SetInitializer(new NullDatabaseInitializer<NinjaContext>()*). This statement inhibits the database from going through initialization when working with NinjaContext. It will probably be appropriate to uncomment this statement when the program is deployed for production.

Edit InsertNinja() by changing the name and birthday of this 2nd Ninja. Run the program again. We notice that the text in console window is reduced. When you examine the database again - via the **Server Object Explorer -** click the circular arrow (refresh) in the toolbar of the window where the dbo.Ninjas is displayed. You now see the 2nd Ninja row.

With Entity Framework 6 you are permitted to send only one command to the database at a time. Using *context.Ninjas.Add()* you would be adding only one row; however, one can use the *context.Ninjas.AddRange()* method that takes an IEnumerable argument. Examine method InsertMultipleNinjas() in Program.cs. You can test this method by commenting out *InsertNinja()* in Main(), uncommenting *InsertMultipleNinjas()* in Main(), and clicking the **Start** icon in Visual Studio’s toolbar.

As the console read-out shows, Entity Framework (1) opened a connection, (2) started a transaction, (3) added Ninja Leonardo, (4) added Ninja Raphael, (5) committed the transaction, and (6) closed the connection. If anything within the transaction fails, everything gets rolled back, and Entity Framework will print a specific error message. There are more advanced techniques available for connections and transactions, but this is the default behavior. As before, examine the new rows in dbo.Ninjas.

## Querying Simple Objects

We now turn our attention to SimpleNinjaQueries() in Program.cs. This method contains code that is the counterpart to SELECT in T-SQL. Again, we instantiate context.Ninjas (a DbSet<Ninjas> object). But this time instead of .Add() or .AddRange(), we invoke .ToList(). The result is

var ninjas = context.Ninjas.ToList();

.ToList() is an executing method; the query does not read from the database until an executing method is invoked. This statement returns the set of all ninjas in dbo.Ninjas into a **List** object. An alternative – possibly more readable – is to split the above into 2 separate statements.

var query = context.Ninjas;

var ninjas = query.ToList();

The 1st statement is analogous to specifying an SQL query. The 2nd statement invokes the query.

There is another alternative. Instead of looping through the **List** object, one might loop through the query, itself.

foreach ( var ninja in context.Ninjas )

{

Console.WriteLine(ninja.Name);

}

Although this code is satisfactory in this case (with a small number of database rows to process, and a brief amount of processing for each row), it is usually considered a dangerous practice, because the database connection remains open until the loop completes.

To continue shadowing the Pluralsight course

* Comment-out everything in the **using** block except “var ninjas = context.Ninjas.ToList();”.
* Set a breakpoint at the end of the **using** block.
* Comment-out the statement in Main() that invoked the previous worker method.
* Uncomment the statement that invokes this worker method (SimpleNinjaQueries()).
* Build and run the program.
* When execution pauses for the break, examine the results – the **List** of ninjas – in the Visual Studio **Locals** window.

A slightly more elaborate syntax is the introduction of a simple WHERE clause.

* Comment-out everything in the **using** block except “var ninjas = context.Ninjas.Where(n => n.Name == "Raphael");” and the **foreach** loop at the end.
* Build and run the program. The only selected row is the one whose name is “Raphael”.

Selecting based on the date-of-birth is similar.

var ninjas = context.Ninjas

.Where ( n => n.DateOfBirth >= new DateTime(1984, 1, 1) );

We test that variation in similar fashion; the names of 3 ninjas are printed in console output.

When you want the result to be a single row from the database, use .FirstOrDefault(). FirstOrDefault() is an “executing” method; if the set is not empty, it returns the 1st element of the set; if the set is empty, it returns **null**.

var ninja = context.Ninjas // notice that “ninja” has replaced “ninjas” (no longer a **List**)

.Where ( n => n.DateOfBirth >= new DateTime(1984, 1, 1) )

.FirstOrDefault();

Console.WriteLine ( ninja.Name );

The previous text can be expressed more concisely (if you like) by moving the Linq expression from the .Where clause to the .FirstOrDefault claue. Specifically

var ninja = context.Ninjas

.FirstOrDefault ( n => n.DateOfBirth >= new DateTime(1984, 1, 1) );

Console.WriteLine ( ninja.Name );

When you include the “context.Database.Log” statement (described earlier in the [Inserting Objects](#_Inserting_Objects) section), the generated SQL displays in the console output.

Performance is usually improved when filtering is done in the database (instead of filtering a large result set in memory). The last example illustrates the use of .Skip and .Take to reduce the number of rows in the result from the query. (.Skip and .Take are convenient to use for paging in web application. To obtain a brief reminder of what they do, hover the mouse over them in Visual Studio.)

## Updating Modified Objects

The examples in this section modify objects in the database.

Change Main() in Program.cs so that QueryAndUpdateNinja() executes. Run the program, and examine the text in the console output. The first set of SQL indicates reading a Ninja object from the database (in this case the first Ninja object encountered). The “context” variable keeps track of objects that are modified. Therefore, in response to SaveChanges() the information is available to update the appropriate Ninja object with a toggled value for its ServedInOniwaban column.

***The pattern of code in*** QueryAndUpdateNinja() ***is not limited to a single change in a single object. Multiple fields in an object can be changed. Multiple rows in a table can be changed. Objects in multiple tables can be changed. I found it instructive to experiment with a small step toward generalization. I added the statements***

var clan = context.Clans.FirstOrDefault();

clan.ClanName += "X";

***After I ran the modified program, I found that – in addition to toggling ServedInOniwaban - the program had changed the name in dbo.Clans to “***Vermont ClansX”.

Change Main() in Program.cs so that QueryAndUpdateNinjaDisconnected() executes. The second example deals with “disconnected applications” - described as follows.

One or more objects are retrieved from the database. These are displayed to a human, who interacts with a **user interface**. The “disconnected application” interprets some of these interactions as instructions to modify one or more of the database objects. Entity Framework code performs the modifications.

Two separate “using (var context = new NinjaContext())” blocks are required, one to retrieve the database objects and the second to update one or more of these objects. Refer to QueryAndUpdateNinjaDisconnected(), which contains the two blocks. The statement between the two blocks

ninja.ServedInOniwaban = (!ninja.ServedInOniwaban);

is a surrogate for the **user interface**, where the human indicates what changes are to be made.

Take a look at the block of code labeled

// Update Changes 1. This will not work.

The statement “context.Ninjas.Attach(ninja);“ tells Entity Framework to be aware of the object “ninja” when it subsequently saves changes to the database. It doesn’t work, because Entity Framework does not know that this object is different from the value in the database. To instruct Entity Framework that this object needs to be updated, we need one more statement, “context.Entry(ninja).State = EntityState.Modified;”

The block of code labeled

// Update Changes 2. This will work.

contains the additional statement. Try running the program with this 2nd version. Notice – in the console output – that Entity Framework changes all of the columns when updating "ninja”; Entity Framework does not know which column has changed.

***Similar to the experiment described above for*** QueryAndUpdateNinja(), ***I tried adding a second object to change – the name in dbo.Clans. It worked as expected. When multiple objects are changed in memory, similar changes occur in the database as long as the pair of statements are executed for each changed object:***

context . *DbSet\_object .* Attach ( *object\_in\_memory* );

context . Entry ( *object\_in\_memory* ) . State = EntityState . Modified;

***Also, it is not necessary to retrieve the object from database before updating it; however, make certain that the*** Id ***column contains the correct value.***

## Retrieving Data with the Find and SqlQuery Methods

In addition to Linq queries, the DbSet class provides other useful methods. Two of these - .Find() and .SqlQuery() are introduced in this section.

Change Main() in Program.cs so that RetrieveDataWithFind() executes. The DbSet.Find method takes a single integer argument, which it matches against the primary key in the table that identifies the DbSet. The return value is the object that contains that key value in the primary-key column. In other words, in the context of the method RetrieveDataWithFind(), the statement

var ninja = context.Ninjas.Find(keyval);

returns the ninja object whose primary-key value is 4 (keyval had been set to 4 earlier).

Of course, it is not necessary to assign the “found” value to a Ninja object. Instead, the “var ninja . . .” statement and the next statement, “Console.WriteLine( . . .”, could be compressed into a single statement

Console.WriteLine ( "After Find#1:" + context . Ninjas . Find ( keyval ) . Name );

Next suppose that you want to introduce another statement that includes

“context . Ninjas . Find ( keyval ) . ServedInOniwaban )”

But wouldn’t this be wasteful – making a 2nd retrieval of the same Ninja object? It turns out that the answer is “no”. Find() will first check to see if the object already exists in memory and is being tracked by the context. It will query the database only if the object is not found in memory. Try running the program. Notice in the console output that the SQL is listed only once even though “context.Ninjas.Find(keyval)” is executed twice.

Observe, also, in the console output that the first SQL statement is “SELECT TOP (2)”. To explain why the query is attempting to get two Ninja objects, consider a small change to the code in RetrieveDataWithFind(). Substitute

var ninjas = context.Ninjas . Where ( n => n.Name == "Leonardo" );

var ninja = ninjas . SingleOrDefault();

In place of

var ninja = context.Ninjas.Find(keyval);

SingleOrDefault() is a method to fetch a single object from the database set. To achieve this, SingleOrDefault() attempts to retrieve a 2-object set. If the set is empty, the return value is **null**. If the set contains exactly one object, the return value is the object. If the set contains two objects, SingleOrDefault() throws an exception. Run the modified program. Notice - in the console output – that the SQL produced for SingleOrDefault() is very similar to the SQL produced for Find(). (I do not know how Find() can encounter two objects with the same primary key value; one would have to create – somehow – a corrupt database.)

The .SqlQuery() method is used to interact with the database via a stored procedure. The Pluralsight course does not describe how to insert a stored procedure into the database. At the time I was writing this document I had to do some research and experimenting to learn how to do this. The instructions are as follows.

* In Visual Studio navigate to **SQL Server Object Explorer**. Visual Studio shows the top (root) node of the tree of databases.
* Drill down as follows: SQL Server > (localdb) MSSQLLocalDB > Databases > NinjaDomain.DataModel.NinjaContext > Programmability
* Right-click **Stored Procedures** node. Visual Studio displays a pop-up menu.
* Click **Add New Stored Procedure...** In the center pane Visual Studio displays a template for a new stored procedure.
* On line 1 change “Procedure” to the name of the new stored procedure, e.g. “GetOldNinjas”.
* Lines 2 and 3 show samples of how to declare parameters. Since we are dealing with a stored procedure that has no parameters, delete lines 2 and 3. Since we are dealing with a stored procedure that does not return a value, delete the last line of the template “RETURN 0”.
* The line after AS shows how the body of the procedure might begin. Replace this with the body recommended by the Pluralsight instructor -

SELECT \* FROM Ninjas WHERE DateOfBirth <= '1/1/1980'

* Click the **Update** icon at the left edge of the toolbar. Visual Studio displays a pop-up dialog titled **Preview Database Updates**.
* Click the **Update Database**  command button.
* Expand the **Stored Procedures** node in the **SQL Server Object Explorer**. Visual Studio displays a list of the database’s stored procedures. You should see the name of the stored procedure that you just added.

Change Main() in Program.cs so that RetrieveDataWithStoredProc() executes. The DbSet.SqlQuery method takes a string argument, which contains T-SQL code. It is recommended that you limit this text to

exec *name\_of\_stored\_procedure*

In order to induce the query to execute, we need either (1) to loop through the returned DbSet ( foreach ( var ninja in ninjas ) { . . . } ), or (2) to invoke an executing method ( .FirstOrDefault(), .SingleOrDefault(), .ToList(), etc. ); this was discussed earlier in [Querying Simple Objects](#_Querying_Simple_Objects).

The Pluralsight instructor said that Entity Framework provides a way to pass parameters to a stored procedure, and that this will be demonstrated by example later in the course.

## Deleting Simple Objects

***This section illustrates removing a row from a database table. Not wanting to disturb the 1st 4 rows in the dbo.Ninjas table, I took the liberty of making some modifications to the Pluralsight examples. I added a 5th Ninja (named Donatello) using Visual Studio’s SQL Server; and I revised the Pluralsight examples so that the 5th Ninja is deleted. In the 1st example, method*** DeleteNinja()***, this 5th Ninja is retrieved via***

.OrderByDescending ( n => n.Id ) . FirstOrDefault()

Add a 5th Ninja as I described above. Change Main() in Program.cs so that DeleteNinja() executes. Notice the .Remove() method followed by .SaveChanges(). Run the program; then reexamine the Ninjas table via Visual Studio’s SQL Server; notice that Donatello has been deleted.

It is more likely that the application would be disconnected (see the discussion in [Updating Modified Objects](#_Updating_Modified_Objects)). To illustrate this make the following changes to DeleteNinja().

* Comment out “context . Ninjas . Remove ( ninja );” and “context . SaveChanges();”.
* Remove “/\*” and “\*/” in the bottom part of DeleteNinja().
* Add the 5th Ninja, Donatello, back into dbo.Ninjas via Visual Studio’s SQL Server.

The code in the bottom part of DeleteNinja() is similar to QueryAndUpdateNinjaDisconnected(), and it solves the same problem – where the Ninja object is no longer being tracked, because it was retrieved in an earlier NinjaContext() block. The Pluralsight instructor illustrates two options -- either

“context . Ninjas . Attach ( ninja );” followed by “context . Ninjas . Remove ( ninja );”

or

“context . Entry ( ninja ) . State = EntityState . Deleted;”

Don’t forget to add Donatello back in dbo.Ninjas before trying the 2nd option.

DeleteNinjaWithKeyValue() consists of features that have already been discussed, and no farther comments are needed. However, the Pluralsight instructor points out that using it entails two trips to the database – (1) to put the ninja object into a state where it is being tracked by the NinjaContext(), and (2) to delete the ninja object. This seems wasteful, and the Pluralsight instructor recommended invoking a stored procedure, instead. I tried a different alternative – see DeleteNinjaWithKeyValue2() in [Appendix I: Program.cs – Contains EF Methods for Interacting with Data](#_Appendix_I:_Program.cs_2). I tested it after adding a 5th ninja and copying its Id value to the initialized value of “keyval” in DeleteNinjaWithKeyValue2(). It seems to work.

Using a stored procedure to perform deletions is similar to what we did in [Retrieving Data with the Find and SqlQuery Methods](#_Retrieving_Data_with) - in method RetrieveDataWithStoredProc(). The example for deleting with a stored procedure is shown in DeleteNinjaViaStoredProcedure(). RetrieveDataWithStoredProc() uses the DbContext method Ninjas.SqlQuery(), whereas DeleteNinjaViaStoredProcedure() uses the DbContext method Database.ExecuteSqlCommand(). The major difference is that Ninjas.SqlQuery() returns a DbSet for a specific table, whereas ExecuteSqlCommand() is not associated with any particular table. To see ExecuteSqlCommand() working . . .

* Change Main() in Program.cs so that DeleteNinjaViaStoredProcedure() executes.
* Add a 5th Ninja as I described above. Note the Id value assigned to this 5th Ninja.
* Change DeleteNinjaViaStoredProcedure() in Program.cs, so that the initialiazed value of keyval is the Id value assigned to the 5th Ninja.
* Add the stored procedure DeleteNinjaViaId to our database, NinjaDomain.DataMode.NinjaContext. Use the following T-SQL code.

CREATE PROCEDURE [dbo].[DeleteNinjaViaId]

@pId int

AS

DELETE FROM dbo.Ninjas WHERE Id = @pId;

([Retrieving Data with the Find and SqlQuery Methods](#_Retrieving_Data_with) contains instructions for creating a stored procedure.)

* Run the program.
* Refresh and then examine the data in dbo.Ninjas. Note that the 5th Ninja has been deleted.

## Appendix I: Program.cs – Contains EF Methods for Interacting with Data

The following is a copy of the code in Program.cs, which was downloaded from the Exercise files of “Getting Started with Entity Framework 6**”.**  Program.cs contains code that invokes Entity Framework Code First instructions as an interface to a database.

I have made some changes to the text below after copying it from the Exercise files. During the Pluralsight instruction, the code was modified, and I used commenting to show the old code and the modified code.

using NinjaDomain.Classes;

using NinjaDomain.DataModel;

using System;

using System.Data.Entity;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace ConsoleApplication

{

class Program

{

static void Main(string[] args)

{

// Database.SetInitializer(new NullDatabaseInitializer<NinjaContext>());

InsertNinja();

// InsertMultipleNinjas();

//SimpleNinjaQueries();

//QueryAndUpdateNinja();

//DeleteNinja();

//RetrieveDataWithFind();

//RetrieveDataWithStoredProc();

//DeleteNinjaWithKeyValue();

//DeleteNinjaWithKeyValue2();

//DeleteNinjaViaStoredProcedure();

//QueryAndUpdateNinjaDisconnected();

//InsertNinjaWithEquipment();

//SimpleNinjaGraphQuery();

//ProjectionQuery();

//QueryAndUpdateNinjaDisconnected();

//ReseedDatabase();

Console.ReadKey();

}

private static void InsertNinja()

{

var ninja = new Ninja

{

// Name = "JulieSan",

Name = "SampsonSan",

ServedInOniwaban = false,

// DateOfBirth = new DateTime(1980, 1, 1),

DateOfBirth = new DateTime(2008, 1, 28),

ClanId = 1

};

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

context.Ninjas.Add(ninja);

context.SaveChanges();

}

}

private static void InsertMultipleNinjas()

{

var ninja1 = new Ninja

{

Name = "Leonardo",

ServedInOniwaban = false,

DateOfBirth = new DateTime(1984, 1, 1),

ClanId = 1

};

var ninja2 = new Ninja

{

Name = "Raphael",

ServedInOniwaban = false,

DateOfBirth = new DateTime(1985, 1, 1),

ClanId = 1

};

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

context.Ninjas.AddRange(new List<Ninja> { ninja1, ninja2 });

context.SaveChanges();

}

}

private static void SimpleNinjaQueries()

{

using (var context = new NinjaContext())

{

// var ninjas = context.Ninjas.ToList();

//var ninjas = context.Ninjas.Where(n => n.Name == "Raphael");

//var ninjas = context.Ninjas

// .Where(n => n.DateOfBirth >= new DateTime(1984, 1, 1));

//var ninja = context.Ninjas

// .Where(n => n.DateOfBirth >= new DateTime(1984, 1, 1))

// .FirstOrDefault();

//Console.WriteLine(ninja.Name);

context.Database.Log = Console.WriteLine;

var ninjas = context.Ninjas

.Where(n => n.DateOfBirth >= new DateTime(1984, 1, 1))

.OrderBy(n => n.Name)

.Skip(1).Take(1);

// If the argument to .Take were greater than 1, it might be good practice to end the

// previous statement with .ToList(). This would close the database connection while looping // over the result set in the subsequent foreach() block.

//var query = context.Ninjas;

// var someninjas = query.ToList();

foreach (var ninja in ninjas)

{

Console.WriteLine(ninja.Name);

}

}

}

private static void QueryAndUpdateNinja()

{

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

var ninja = context.Ninjas.FirstOrDefault();

// var clan = context.Clans.FirstOrDefault();

ninja.ServedInOniwaban = (!ninja.ServedInOniwaban);

// clan.ClanName += "X";

context.SaveChanges();

}

}

private static void QueryAndUpdateNinjaDisconnected()

{

Ninja ninja;

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

ninja = context.Ninjas.FirstOrDefault();

}

ninja.ServedInOniwaban = (!ninja.ServedInOniwaban);

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

context.Ninjas.Attach(ninja);

context.Entry(ninja).State = EntityState.Modified;

context.SaveChanges();

}

}

private static void RetrieveDataWithFind()

{

var keyval = 4;

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

//var ninjas = context.Ninjas.Where(n => n.Name == "Leonardo");

//var ninja = ninjas.SingleOrDefault();

var ninja = context.Ninjas.Find(keyval);

Console.WriteLine("After Find#1:" + ninja.Name);

var someNinja = context.Ninjas.Find(keyval);

Console.WriteLine("After Find#2:" + someNinja.Name);

ninja = null;

}

}

private static void RetrieveDataWithStoredProc()

{

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

var ninjas = context.Ninjas.SqlQuery("exec GetOldNinjas")/\*.ToList()\*/;

foreach (var ninja in ninjas)

{

Console.WriteLine(ninja.Name);

}

}

}

private static void DeleteNinja()

{

Ninja ninja;

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

ninja = context.Ninjas.OrderByDescending(n => n.Id).FirstOrDefault();

context.Ninjas.Remove(ninja);

context.SaveChanges();

}

/\*

// The following is a surrogate for the UI in a disconnected application

Console.WriteLine("ID = {0}, Name = {1}. Delete this?", ninja.Id, ninja.Name);

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

context.Ninjas.Attach(ninja);

context.Ninjas.Remove(ninja);

//context.Entry(ninja).State = EntityState.Deleted;

context.SaveChanges();

}

\*/

}

private static void DeleteNinjaWithKeyValue()

{

var keyval = 1;

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

var ninja = context.Ninjas.Find(keyval);

context.Ninjas.Remove(ninja);

context.SaveChanges();

}

}

private static void DeleteNinjaWithKeyValue2()

{

var keyval = 12;

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

var ninja = new Ninja

{

Id = keyval

};

context.Entry(ninja).State = EntityState.Deleted;

context.SaveChanges();

}

}

private static void DeleteNinjaViaStoredProcedure()

{

var keyval = 14;

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

context.Database.ExecuteSqlCommand(

"exec DeleteNinjaViaId {0}", keyval);

}

}

private static void InsertNinjaWithEquipment()

{

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

var ninja = new Ninja

{

Name = "Kacy Catanzaro",

ServedInOniwaban = false,

DateOfBirth = new DateTime(1990, 1, 14),

ClanId = 1

};

var muscles = new NinjaEquipment

{

Name = "Muscles",

Type = EquipmentType.Tool,

};

var spunk = new NinjaEquipment

{

Name = "Spunk",

Type = EquipmentType.Weapon

};

ninja.EquipmentOwned.Add(muscles);

ninja.EquipmentOwned.Add(spunk);

context.Ninjas.Add(ninja);

context.SaveChanges();

}

}

private static void SimpleNinjaGraphQuery()

{

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

//var ninjas = context.Ninjas.Include(n => n.EquipmentOwned)

// .FirstOrDefault(n => n.Name.StartsWith("Kacy"));

var ninja = context.Ninjas

.FirstOrDefault(n => n.Name.StartsWith("Kacy"));

Console.WriteLine("Ninja Retrieved:" + ninja.Name);

context.Entry(ninja).Collection(n => n.EquipmentOwned).Load();

}

}

private static void ProjectionQuery()

{

using (var context = new NinjaContext())

{

context.Database.Log = Console.WriteLine;

var ninjas = context.Ninjas

.Select(n => new { n.Name, n.DateOfBirth, n.EquipmentOwned })

.ToList();

}

}

private static void ReseedDatabase()

{

Database.SetInitializer(new DropCreateDatabaseAlways<NinjaContext>());

using (var context = new NinjaContext())

{

context.Clans.Add(new Clan { ClanName = "Vermont Clan" });

var j = new Ninja

{

Name = "JulieSan",

ServedInOniwaban = false,

DateOfBirth = new DateTime(1980, 1, 1),

ClanId = 1

};

var s = new Ninja

{

Name = "SampsonSan",

ServedInOniwaban = false,

DateOfBirth = new DateTime(2008, 1, 28),

ClanId = 1

};

var l = new Ninja

{

Name = "Leonardo",

ServedInOniwaban = false,

DateOfBirth = new DateTime(1984, 1, 1),

ClanId = 1

};

var r = new Ninja

{

Name = "Raphael",

ServedInOniwaban = false,

DateOfBirth = new DateTime(1985, 1, 1),

ClanId = 1

};

context.Ninjas.AddRange(new List<Ninja> { j, s, l, r });

context.SaveChanges();

context.Database.ExecuteSqlCommand(

@"CREATE PROCEDURE GetOldNinjas

AS SELECT \* FROM Ninjas WHERE DateOfBirth<='1/1/1980'");

context.Database.ExecuteSqlCommand(

@"CREATE PROCEDURE DeleteNinjaViaId

@Id int

AS

DELETE from Ninjas Where Id = @id

RETURN @@rowcount");

}

}

}

}