## 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) | n |
| o | p |
| q | 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 to hold this project (and possibly other projects introduced in this Pluralsight course. 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.Model2Demo, which – in turn – contains the subfolder NinjaDomain.Classes and the file Ninja.Model2Demo.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** statement in our **enum** modules is 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 did not occurr to me that one might have multiple .cs files with the same **namespace** declaration in different folders. And when this is the case, evidently one doesn’t need **using** statements to refer to a **class** or **enum** in another .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