مقدمه ای بر

Microsoft Entity Fraemework (Code First)

ترجمه و تالیف : مهندس مهدی کیانی

ویرایش اول
پاییز ۹۴

Introduction to Microsoft Entity Framework Code-First

By

Mehdi, Kiani

www.mkiani.ir

تقدیم به

همه آنانی که عاشق آموختن هستند

و با سپاس فراوان از همسر بزرگوارم که همیشه مشوق من بودند.

فهرست مطالب

درباره نویسنده

مهدی کیانی متولد اصفهان می باشد.او بیش از یک دهه است که در حوزه کامپیوتر و به صورت تخصصی در زمینه برنامه نویسی پلت فرم دات نت فعالیت می کند. در این مدت دوره های آموزشی متعددی را در آموزشگاه های خصوصی،دولتی و نیمه دولتی برگزار کرده است. از یادگیری در همه امور لذت می برد و علاوه بر برنامه نویسی در زمینه های هنری نیز فعالیت می کند.در لینک زیر می توانید بیوگرافی او را مشاهده نمائید.

http://mkiani.ir/pages/biography

درباره کتاب

این کتاب دارای چهار فصل بوده که در مجموع بیش از ۱۲۰ صفحه مطلب را شامل می شود.

در فصل اول مقدمه ای کوتاه در رابطه با تاریخچه Entity Framework و مدل های آن صحبت خواهد شد.

در فصل دوم اولین برنامه ای که از تکنولوژی Entity Framework استفاده می کند را ایجاد و در آن چهار عمل اصلی کار با داده ها یعنی CRUD[1] پیاده سازی خواهند شد.

در فصل سوم به مفاهیم عمیق تری از Entity Framework در رابطه با پایگاه های داده ای، نظیر نگاشت ستون ها و خواص آن ها، ارتباط بین موجودیت ها و ... بحث خواهد شد.

و نهایتا در فصل چهار م سایر موارد در خصوص Entity Framework به همراه آخرین امکانات آن نظیر Database Migration (Entity Framework Migration) مورد بررسی قرار خواهند گرفت.

مطالب این کتاب برگرفته از کتاب Code-First Development With Entity Frmaework نوشته Sergey Barskiy می باشد. البته در نوشتن مطالب به ترجمه صرف پرداخته نشده است بلکه برداشت نویسنده از کتاب مذکور همراه با منابع و کتب دیگر در این زمینه و نیز تجربیات شخصی ترکیب شده است و نگارش این کتاب بر این اساس شکل گرفته است.

افرادی که تجربیاتی در زمینه نوشتن مقاله یا مطالب علمی دارند نیک می دانند که چه کار بسیار دشواری است. با تمامی کمبود وقتی که داشتم سعی کردم تا حد امکان مطالب کتاب بدون ایراد در اختیار شما قرار گیرد. لذا صمیمانه تقاضامندم ایرادات مطالب را به همراه نقطه نظرات خود از طریق پست الکترونیک ذیل با من در میان گذاشته تا در نسخه های بعدی رفع گردند.

mkiani۳۰۰۰@gmail.com

[1] Create(Insert)-Retrive(Read-Select)-Update-Delete

این کتاب برای چه کسانی می باشد/نمی باشد

اگر شما از دسته برنامه نویسانی که دارای تجربه کار با یکی از زبان های برنامه نویسی دات نت نظیر سی شارپ یا ویژوال بیسیک و دستورات ADO.Net برای کار با پایگاه داده های نظیر Sql Server می باشید و می خواهید استفاده از Entity Framework در برنامه های خود را فراگیرد این کتاب مناسب شما خواهد بود. برای درک مفاهیم این کتاب نیاز به دانش اولیه در زمینه برنامه نویسی سی شارپ و همچنین مفاهیم ADO.Net و دستورات ابتدایی T-Sql می باشد.لذا چنانچه شما هیچ اطلاعاتی در هریک از زمینه های مذکور اطلاعات ندارید بهتر است قبل از مطالعه این کتاب پیش نیاز های لازم را فراگیرید. علاوه بر این همانطور که از نام کتاب مشخص است این کتاب صرفا مقدمه ای در خصوص EF به شما ارائه می دهد. هدف از این کتاب صرفا ارائه یک نقشه راه در جهت حرکت به سمت این تکنولوژی است. لذا خواننده گرامی می بایستی با مطالعه سایر منابع و مراجع اصلی این تکنولوژی توانمندی های خود را در این زمینه بهبود بخشند تا بتوانند از آن در پروژه های واقعی خود بهره گیرند.

مقدمه

یکی از تغییرات اساسی که از زمان ظهور دات نت تا به امروز در این تکنولوژی شاهد بودیم به روز رسانی های متعدد در زمینه کار با داده ها در برنامه های تجاری توسط ماکروسافت به عنوان مالک تکنولوژی دات نت بوده است. از به وجود آمدن کتابخانه ADO.Net گرفته تا کتابخانه Linq که بعد ها Linq to Sql بر اساس آن به وجود آمد و سپس پیاده سازی مفهوم ORM[1] توسط Entity Framework.

کتابخانه Entity Framework که در زمان نوشتن این کتاب نسخه ۶,۱,۳ آن در دسترس برنامه نویسان قرار دارد یک ابزار ORM می باشد. به صورت بسیار خلاصه ORM را می توان نگاشت بین اشیای درون برنامه و جداول رابطه ای درون پایگاه داده هایی نظیر Sql Server دانست. کتابخانه Entity Framework دارای سه مدل کلی Database First ، Model First و Code First می باشد که توضیحات تکمیلی این سه مدل را در ادامه کتاب خواهید دید. این کتاب به آموزش مدل سوم یعنی Code First از این کتابخانه می پردازد.

[1] Object Relational Mapping

فصل اول : مقدمه ای بر Entity Framework

مقدمه

در این فصل شما با مقدمات Entity Framework ، تاریخچه مختصری از زمان پیدایش آن تا زمان حال و نیز علل لزوم استفاده یا عدم استفاده از آن را فرا خواهید گرفت.

در این فصل شما با مفاهیم :

- چرا Entity Framework
- انواع(مدل های) Entity Framework
- معماری EF

در زمینه توسعه برنامه هایی با کتابخانه Entity Framework در مدل Code-First آشنا خواهید شد.

چرا Entity Framework

اخیرا کمتر برنامه ای وجود دارد که نیاز به نگهداری داده ها نداشته باشد. اغلب برنامه های تجاری که امروز ه نوشته می شوند نیاز به مکانیزمی برای نگهداری داده بوده تا بتوان در مواقع لزوم از آن داده ها استفاده کرد. واژه RDBMS که مخفف Relational Database Management System می باشد ابزاری است که توسعه دهندگان نرم افزار از آن برای مدیریت داده ها مورد استفاده قرار می دهند. برخی از این RDBMS های مهم عبارتند از SQL

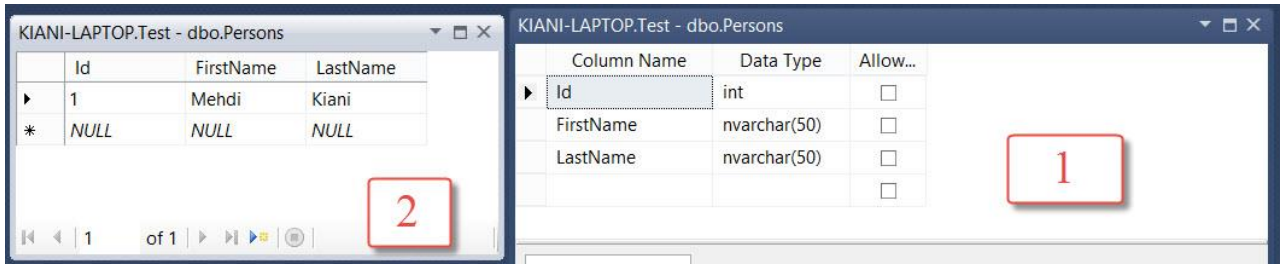
Server ، MySQL ، Oracle و ... که برنامه نویسان مختلف با توجه به تخصص و تکنولوژی مورد استفاده خود از یکی از این ابزار ها بهره می برند.

زبان کد نویسی RDBMS ها دستورات SQL (مخفف Structured Query Langue) می باشد که توسط موسسه ANSI توسعه داده شده است.

اگر شما جزء دسته برنامه نویسان دات نت باشید یقینا تا بحال از SQL Server به عنوان RDBMS و دستورات SQL برای مدیریت داده های خود بهره برده اید. همانطور که می دانید چهار عمل ایجاد،دسترسی(خواندن)، ویرایش و حذف از اعمال اصلی کار با هر پایگاه داده ای است. به این چهارعمل اصطلاحا CRUD می گویند. حرف C مخفف Create ، حرف R مخفف Retrieve یا Read ، حرف U مخفف Update و حرف D مخفف Delete.

RDBMS ها داری مجموعه ای از جداول برای نگهداری اطلاعات هستند. هر جدول دارای تعدادی سطر و ستون می باشد. ستون های یک جدول معرف ساختار جدول و سطر های آن معرف داده ها ی جدول میباشند.

به عنوان مثال در شکل ۱-۱ یک جدول به نام Person دارای سه فیلد Id ، FirstName و LastName تعریف شده است :



شکل ۱ - ۱

در شکل ۱-۱ بخش شماره ۱ ، شمای جدول و بخش شماره ۲ داده های جدول نشان داده شده اند.

اگرقبلا با دستورات ADO.Net کار کرده باشید، برای خواندن اطلاعات از یک جدول (مثلا جدول Person فوق) از یک پایگاه داده(مثلا Sql Server) به صورت خلاصه باید روال زیر را طی می کردید:

۱- ایجاد یک ارتباط با پایگاه داده(DbConnection)

۲- ایجاد یک شی دستوری(DbCommand) همراه با دستور Sql (مثلا به صورت زیر)

```
SELECT * FROM Persons
```

۳- اجرای دستور و واکشی اطلاعات(Execute Reader)

۴- قرار گیری اطلاعات واکشی شده در یک شی دات نتی نظیر DataReader

۵- خواندن اطلاعات از شی DataReader مثلا به صورت زیر(با فرض اینکه dr نام شی DataReader باشد)

```
If(dr.Reade()){
String FirstName =dr[“LastName”].ToString();
String LastName =dr[“FirstName”].ToString();
}
```

همانطور که مشاهده می کنید از نام ستون های جدول برای دریافت اطلاعات استفاده شده است. اگر در دستورات فوق به اشتباه به جای FirstName از First_Name استفاده کنید چه اتفاق می افتد؟ ظاهرا هیچ اتفاقی!برنامه بدون خطا کمپایل می شود اما در زمان اجرای برنامه با خطا مواجه خواهید شد! چرا؟ چون نام ستونی از اطلاعات که شما به دنبال آن هستید(First_Name) در جدول Person تعریف نشده است.

همچنین اگر تغییری در ساختار پایگاه داده اعمال شود شما نیازمند تغییرات زیادی در سورس برنامه خواهید بود تا بتوانید تغییرات اعمال شده در پایگاه داده را به سورس برنامه منتقل کنید.

برای حل مشکلاتی از این دست شرکت های مختلف اقدام به ایجاد ابزاری به نام ORM نمودند. واژه ORM مخفف Object Relational Mapping می باشد. در یک تعریف خلاصه ORM ها ابزارهایی برای ساده تر شدن مدیریت کار با پایگاه داده ها می باشند. کلمه مدیریت در اینجا می تواند معانی زیادی داشته باشد. از ایجاد پایگاه داده گرفته تا عملیات CRUD تا مدیریت تغییرات و ....

درواقع ORM ها یک واسط بین برنامه نویس و پایگاه داده هستند. با استفاده از ORM ها شما نگران روش خواندن اطلاعات، دستورات SQL و ... نمی باشید. تنها چیزی که شما می بایستی در ORM ها بدانید نحوه استفاده از این ابزار هاست.

در دنیای دات نت ORM ها ساختار پایگاه داده شما و جداول شما را به صورت کلاس های دات نت برای شما مهیا می کنند. شما کافیست با نحوه استفاده از کلاس ها، شناخت و استفاده از کلاس های مربوط به مجموعه ها، استفاده از دستورات LINQ به جای استفاده از دستورات SQL آشنایی داشته باشید.

اولین ابزار ORM که شرکت ماکروسافت اقدام به ایجاد و معرفی آن نمود که صرفا برای کار با پایگاه داده های Sql Server و Sql Server Compact به کار می رفت Linq to Sql نام داشت که همراه با دات نت فریم ورک ۳,۵ به برنامه نویسان معرفی شد.

به علت محدودیت هایی که این ORM داشت ماکروسافت نسخه بعدی ORM را با نام Entity Framework در سال ۲۰۰۸ معرفی کرد. Entity Framework بسیار کامل تر از Linq to Sql بود و چون دارای یک معماری قوی بود به سرعت جایگزین Linq to Sql شد. برخلاف Linq to Sql ابزار Entity Framework این قابلیت را دارد که نه تنها با پایگاه داده SQL Server کار کند بلکه با ایجاد Provider ها مختلف امکان برقراری و مدیریت ارتباط با پایگاه داده های مختلف را نیز خواهد داشت.

نکته: از این پس از کلمه EF به صورت اختصار به جای Entity Framework استفاده خواهیم کرد.

انواع(مدل های) Entity Framework

اولین نوعی که از EF معرفی شد مدل Database First بود. در این حالت برای پایگاه داده ای که از قبل وجود داشت یک فایل Entity Data Model (فایلی با پسوند edmx) ساخته می شد که دارای سه بخش Logical ، Storage و Mapping بود. بخش Logical که گاها آن را بخش مفهومی (Conceptual) نیز می نامیدند مربوط به سمت کد نویسی(کد های زبان سی شارپ یا وی بی)، بخش Storage اطلاعات مربوط به نحوه ذخیره سازی داده ها در پایگاه داده و نهایتا بخش Mapping نگاشت بین دو بخش قبلی را بر عهده داشت. هر زمانی که تغییری در پایگاه داده رخ می داد می بایستی مدل داده ای نیز مجددا ایجاد شود تا تغییرات درون پایگاه داده را در خود منعکس نماید.

در ورژن بعدی Entity Framework از حالت دیگری به نام Model First پشتیبانی شد. در این حالت شما می توانستید ابتدا با توجه به ابزار های موجود در Toolbox مدل داده ای خود را طراحی کنید و سپس بر اساس مدل خود و اسکریپتی که مدل داده ای ایجاد می کرد اقدام به ایجاد پایگاه داده نمائید.

در هر دو روش قبلی چون کلاس های مربوط به کار با Entity Framework توسط ابزار ORM به صورت مستقیم ایجاد می شد نگهداری کد را کمی سخت می کرد و برنامه نویسان کنترل کاملی بر روی کد های تولید شده توسط این ابزار نداشتند.

حالت سوم EF که امروزه در برنامه های مبتنی بر EF از این حالت استفاده می شود حالت Code First می باشد. در این حالت خبری از فایل edmx نیست. در این حالت شما با استفاده از کلاس های معمول دات نت و استفاده از کلاس های EF می توانید مدل داده ای خود را طراحی کرده و با استفاده ازآن ها به راحتی عملیات مختلف را بر روی پایگاه داده انجام دهید.

در این آموزش نیز ما از حالت سوم یعنی Code First استفاده خواهیم کرد.

معماری EF

در ابتدا باید یادآوری کنم همانطور که اشاره کردم ORM ها که EF نیز یکی از آن ها ست صرفا یک واسط بین شما به عنوان برنامه نویس و پایگاه داده به عنوان محل ذخیره سازی داده ها می باشد که عملیات مابین این دو طرف را بسیار راحت تر و ساده تر می کند و باعث می شود تا برنامه نویس بیش از آن که به جزئیات دستورات SQL بیندیشد به مدل داده ای خود و معماری برنامه خود بیندیشد. شما این را باید بدانید که هر کاری که EF انجام می دهد نهایتا توسط کلاس های ADO.Net انجام خواهد شد. اما مراحلی که به صورت خلاصه در بالا بیان شد برای ایجاد Connection ، ایجاد Command و... از دید شما مخفی شده و این فعالیت ها توسط EF انجام می پذیرد.

کلاس DbContext را می توان قلب EF در حالت Code First دانست. کلاس DbContext را می توانید به عنوان پایگاه داده در نظر بگیرید. پایگاه داده مجموعه از جداول را مشخص می کند. DbContext نیز می تواند توسط کلاس ژنریکی به نام DbSet و نوع کلاسی که به آن نسبت می دهید معرف جداول شما باشد.

هر جدول در زبان سی شارپ با یک کلاس معمولی (POCO [1]) معرفی می شود. ستون های جدول در کلاس هایی که برای EF تعریف می کنید توسط خواص تعریف شده در آن کلاس مشخص می شوند. هر سطر از جدول نیز در EF بیانگر یک نمونه از کلاس متناظر با آن جدول می باشد. بنابر این ایجاد یک نمونه جدید از کلاس و ذخیره آن توسط DbContext مانند این است که شما یک دستور Insert در پایگاه داده انجام داده و یک رکورد جدید به جدول مورد نظر اضافه کرده باشید.

نگران نباشید در بخش های بعدی همراه با مثال مفاهیم بالا را بهتر درک خواهید کرد. در اینجا کافی است تا مفاهیم زیر را به صورت کلی بدانید:

۱- کلاس DbContext در یک نگاه سطح بالا همان معرف پایگاه داده شما می باشد(یک کلاس می بایستی از DbContext مشتق شود)

۲- کلاس DbSet<TRowType> نگهدارنده یک جدول می باشد. TRowType یک کلاس معمول در دات نت می باشد که متناظر با یک جدول در پایگاه داده است.

۳- خواص هر کلاس همانند ستون های جدول می باشند.

۴- هر نمونه از کلاس مانند یک سطر داده در جدول مربوطه در پایگاه داده می باشد.

۵- توسط دستورات LINQ بر روی اشیایی که در DbContext تعریف می کنید می توانید به داده ها دسترسی داشته و آن ها را تغییر دهید.(این تغییرات درون حافظه انجام می شود)

۶- با استفاده از متد های موجود در DbContext می توانید تغییرات را در پایگاه داده منعکس کنید(متد SaveChanges)

۷- کاری که EF برای شما انجام می دهد این است که دستورات LINQ شما را به دستورات SQL تبدیل کرده و با استفاده از کلاس های ADO.Net آن ها را در پایگاه داده منعکس می کند.

[1] Plain Old CLR Object

۸- می توان با نوشتن Provider های شخصی یک EF ایجاد کرد که با پایگاه داده ای غیر از SQL کار کند. کافیست Provider مذکور بتواند SQL معادل با دستورات زبان برنامه نویسی شما را ایجاد کند و نیز بتواند داده های خوانده شده از پایگاه داده را به اشیاء(کلاس) شما در زبان برنامه نویسی تبدیل کند.

نسخه های مختلف Entity Framework

همانطور که گفته شد اولین نسخه Entity Framwork همراه با دات نت ۳,۵ (۳,۵ sp۱ Net Framework.) با عنوان نسخه ۳,۵ ارائه شد و تا امروز نسخه کلی ۶ آن در دسترس می باشد. جدول ۱-۱ به صورت خلاصه نسخه های مختلف EF را همراه تغییرات آن نشان می دهد:

ردیف	نسخه	تغییرات
۱	۳,۵	اولین نسخه توزیع شده همراه با دانت فریم ورک ۳,۵ sp۱ و ویژوال استودیو ۲۰۰۸ SP۱ پشتیبانی از مدل Database-First Design ...
۲	۴,۰	همزمان با دات نت فریم ورک ۴,۰ و ویژوال استودیو ۲۰۱۰ پشتیبانی از مدل Model-First Design Lazy Loading T۴ Code Generation ...
۳	۴,۱	دات نت فریم ورک ۴,۰ و ویژوال استودیو ۲۰۱۰ پشتیبانی از مدل Code-First Design DbContext Data Annotations Fluent API ...
٤	۴,۳	Code-First Migration Automatic Migration ...
٥	٥,۰	پشتیبانی از انواع شمارشی (Enum) انواع پیچیده ...

٦	۶,۰	پشتیبانی از کوئری های غیر همزمان (Async Query) بهبود در مدیریت ارتباطات پشتیبانی از پیکربندی ها بر اساس کد نویسی (Code-Based Configuration) ...

جدول ۱ - ۱

جدول ۱-۱ صرفا نسخه هایی را نشان می دهد که تغییرات در آن ها به نسبت نسخه قبل چشمگیر بوده است. در بین نسخه های فوق، زیر نسخه های دیگری نیز منتشر شدند که عموما یا مربوط به رفع باگ های نسخه های ماقبل خود بوده یا تغییرات آن به نسبت نسخه های فوق بسیار چشمگیر نبوده است.

نکته : در این کتاب از آخرین نسخه توزیع شده آن تا این لحظه یعنی نسخه ۶,۱,۳ استفاده شده است.

خلاصه

در این فصل شما با کلیات ابتدایی Entity-Framework و مدل های آن آشنا شدید. معماری EF واینکه در پس زمینه این تکنولوژی چه اتفاقی رخ می دهد بیان شد. در فصل دوم اولین برنامه با این کتابخانه را خواهید نوشت.پس منتظر نمانید و سریعا به فصل دوم بروید!

فصل دوم : ایجاد اولین پروژه

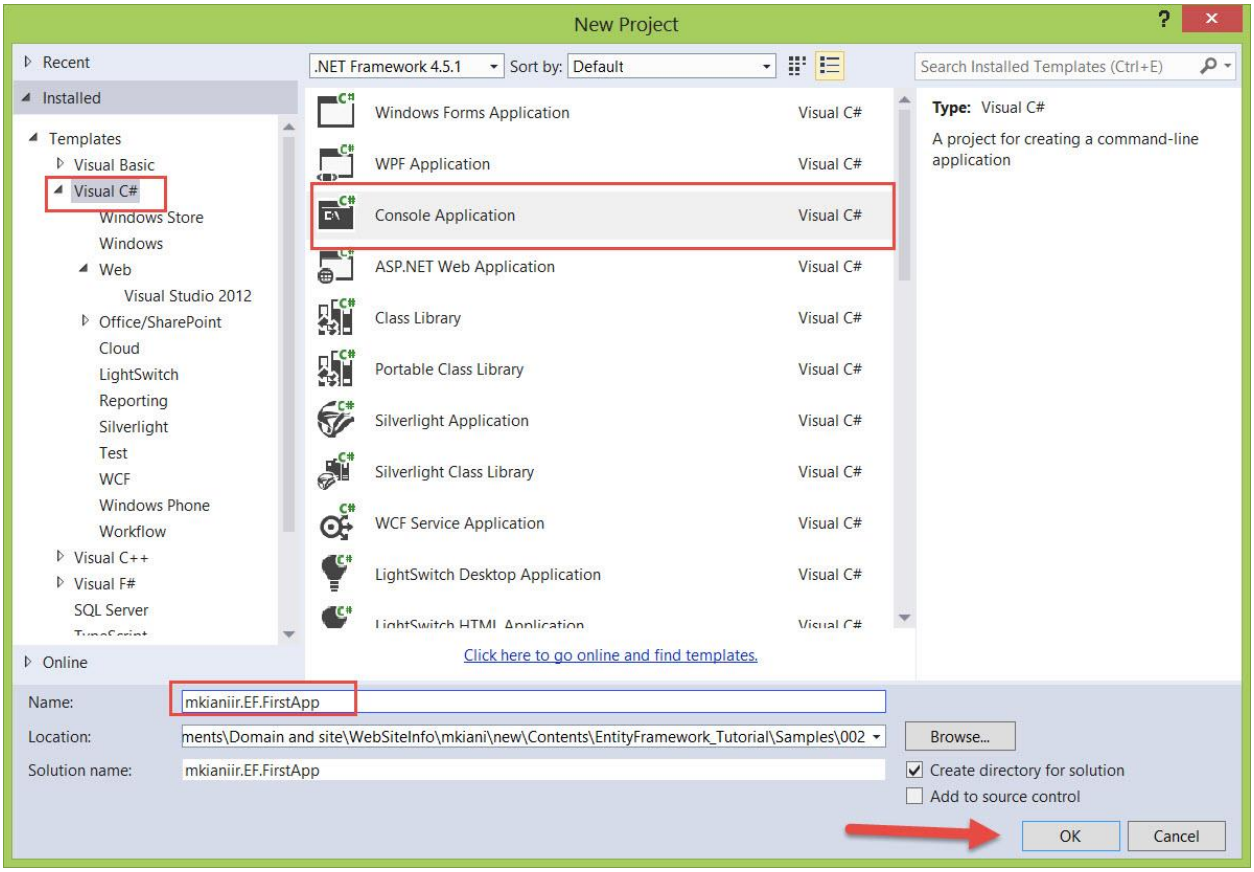
مقدمه

در فصل اول کلیات معماری Entity Framework بیان شد. در این فصل اولین برنامه EF را ایجاد خواهیم کرد تا با مفاهیم گفته شده در فصل اول آشنایی بیشتری پیدا کنید و با پیاده سازی آن به صورت عملی آشنا شوید. در این فصل از یک قالب Console Application برای برنامه استفاده خواهیم کرد. همچنین از Sql Server برای ذخیره پایگاه داده بهره خواهیم برد. بنابر این قبل از ادامه مطالب لازم است تا ابزارهای مورد نیاز را بر روی سیستم خود نصب نمائید.

من از ویژوال استودیو ۲۰۱۳ و Sql Server ۲۰۱۴ استفاده می کنم.در زمان نوشتن این کتاب نسخه ۲۰۱۵ ویژوال استودیو هنوز به صورت رسمی منتشر نشده است.

ساخت پروژه

ویژوال استودیو را باز کنید. از منوی New گزینه Project را کلیک کنید. در پنجره New Project زبان سی شارپ و قالب Console Application را انتخاب نمائید. نام و مسیر ذخیره سازی پروژه را تعیین و روی دکمه Ok کلیک کنید.



شکل ۲ - ۱

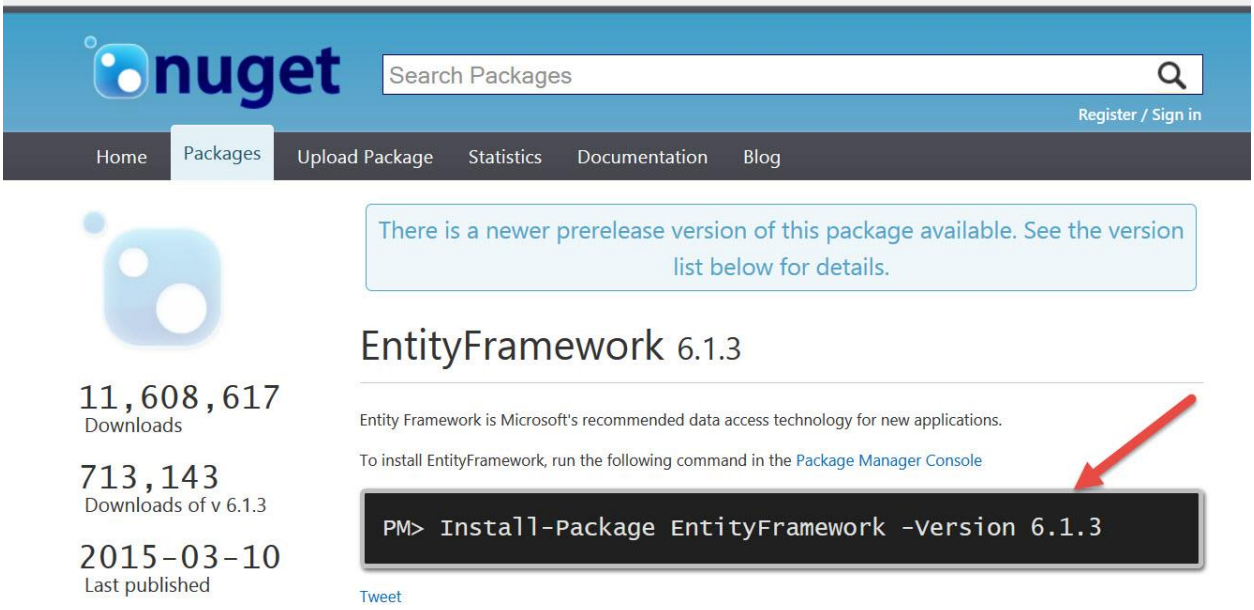
نکته: EF قابل استفاده برای همه مدل های برنامه نویسی دات نت نظیر Windows Forms، WPF ، Web Form و MVC می باشد. به دلیل سادگی و تمرکز بیشتر بر روی ساختار EF من از مدل Console Application استفاده کرده ام.

با استفاده از پنجره Package Manager Console (از منوی Tools و سپس منوی Library Package Manager) کتابخانه EF را با استفاده از دستور زیر نصب کنید:

```
Install-Package EntityFramework -Version ٦,١,٣
```

در زمان نوشتن این مطلب نسخه ۶,۱,۳ از کتابخانه EF آماده بهره برداری شده است. چنانچه در زمانی که شما از این دستور استفاده می کنید با خطا مواجه شدید به ادرس https://www.nuget.org/packages/EntityFramework

بروید و طبق مستندات آن عمل کنید.

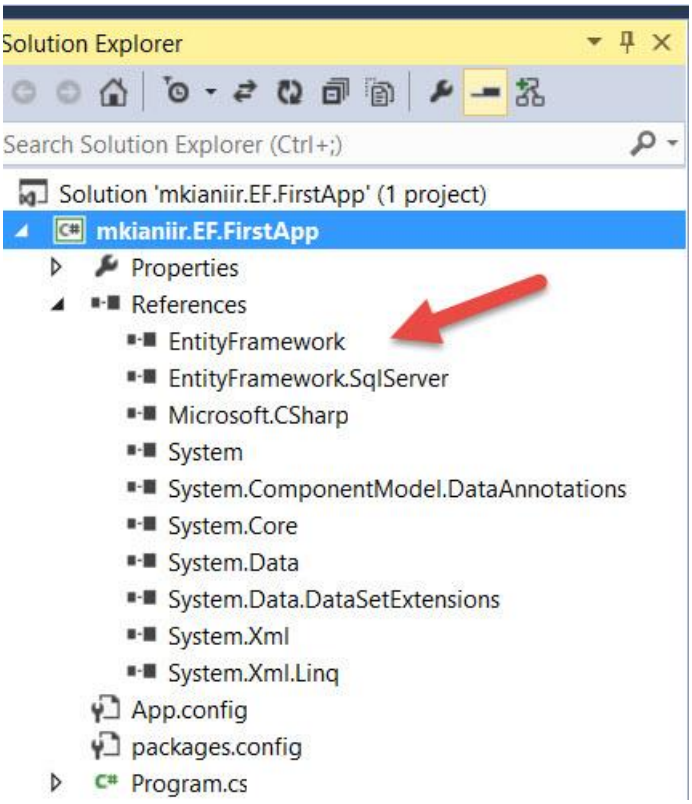


اگر دستور گفته شده را در پنجره Package Manager تایپ کنید پس از مدتی کتابخانه EF به پروژه شما اضافه خواهد شد.

شکل ۲ - ۲

نکته برای نصب Package ها از طریق پکیج منیجر نیوگت[۱] نیاز دارید تا اینترنت متصل باشید.

[۱] نیوگت ابزاری است برای مدیریت کتابخانه های مورد نیاز برنامه نویسان پلت فرم ها ماکروسافتی نظیر دات نت. توسط این ابزار همواره می توانید از اخرین نسخه های کتابخا نه ها مطلع و تغییرات آن ها را به صورت آنلاین بررسی نمائید. توصیه می کنم با این ابزار آشنا و از آن استفاده نمائید. برای توضیحات بیشتر به آدرس https://www.nuget.org/ مراجعه نمائید.

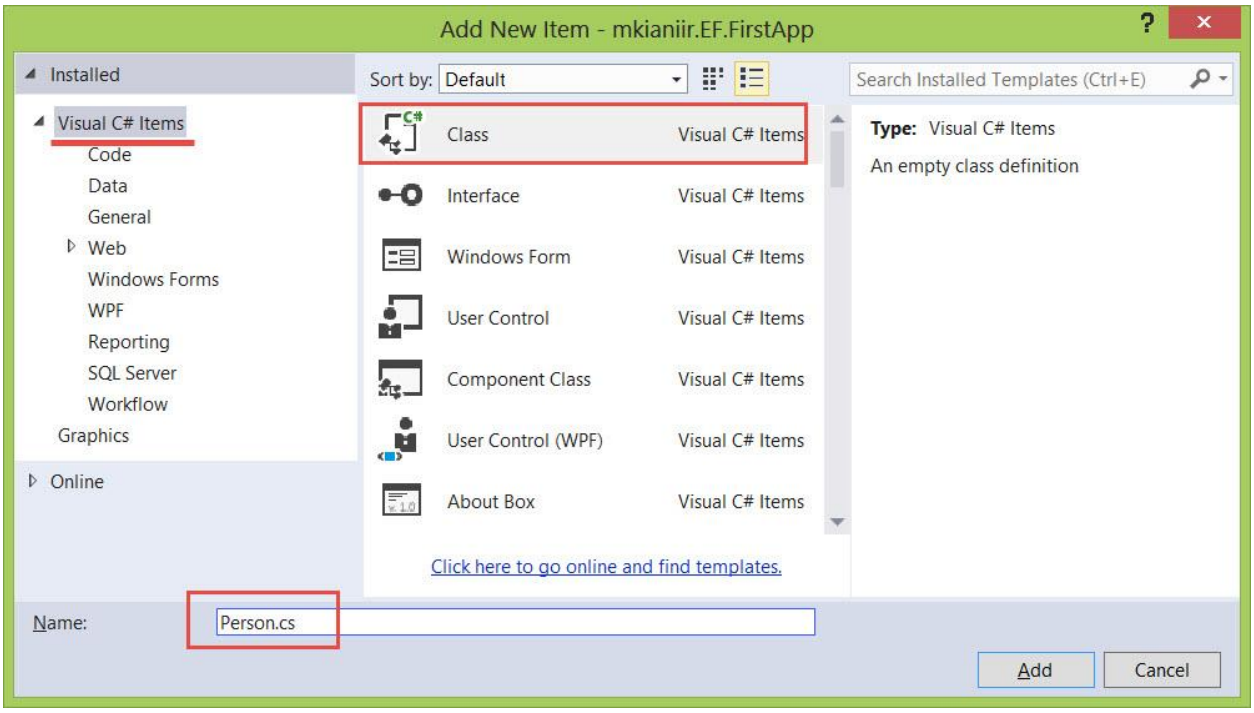


شکل ۲- ۳

همانطور که مشاهده می کنید پس از نصب کتابخانه Entity Framework ارجاعات مورد نیاز به پروژه به صورت خودکار اضافه شده است.

ایجاد کلاس Person

بر روی پروژه کلیک راست کنید و از گزینه Add گزینه Class را انتخاب نمائید. در پنجره Add New Item نام کلاس را Person قرار دهید و دکمه Add را کلیک کنید.(شکل ۲-۴)



شکل ۲- ٤

دستورات کلاس Person را به صورت زیر تغیر دهید:

```
public class Person
  {
      public Int32 Id
      {
          get;
          set;
      }
      public String FirstName
      {
          get;
          set;
      }
      public String LastName
      {
          get;
          set;
      }
  }
```

حال کلاس دیگری به نام PersonContext ایجاد کنید و کد های آن را طبق زیر تغییر دهید:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
```

```
using System.Text;
using System.Threading.Tasks;

namespace mkianiir.EF.FirstApp
{
    public class PersonContext : DbContext
    {
       public PersonContext():base("mkianiir.EF")
        {

        }
        public DbSet<Person> Persons
        {
            get;
            set;
        }
    }
}
```

همانطور که مشاهده می کنید کلاس PersonContext از کلاس DbContext مشتق شده است. کلاس DbContext در فضای نام System.Data.Entity قرار دارد. بنابر این لازم است تا این فضای نام را به مجموعه فضای نام ها کلاس PersonContext اضافه کنید.

توسط دستور base در Constructor کلاس PersonContext مقدار mkianiir.EF را به Constructor کلاس پدر یعنی DbContext ارسال شده است. این بدان معناست که EF از رشته اتصالی به نام mkianiir.EF در فایل App.config برای اتصال به پایگاه داده استفاده خواهد کرد.

خاصیت Persons معرف مجموعه ای از Person ها می باشد. Persons را مانند یک جدول به نام Persons در پایگاه داده در نظر بگیرید. هر Person یک سطر از این جدول خواهد بود.

فایل App.config را باز کرده و رشته اتصال را به صورتی که در زیر مشاهده می کنید در ان تعریف کنید.

```
<connectionStrings>
    <clear/>
    <add name="mkianiir.EF"
         connectionString="Data Source=.;Initial Catalog=Mkianiir_EF_FirstApp;Integrated
Security=true"
         providerName="System.Data.SqlClient"
         />
  </connectionStrings>
```

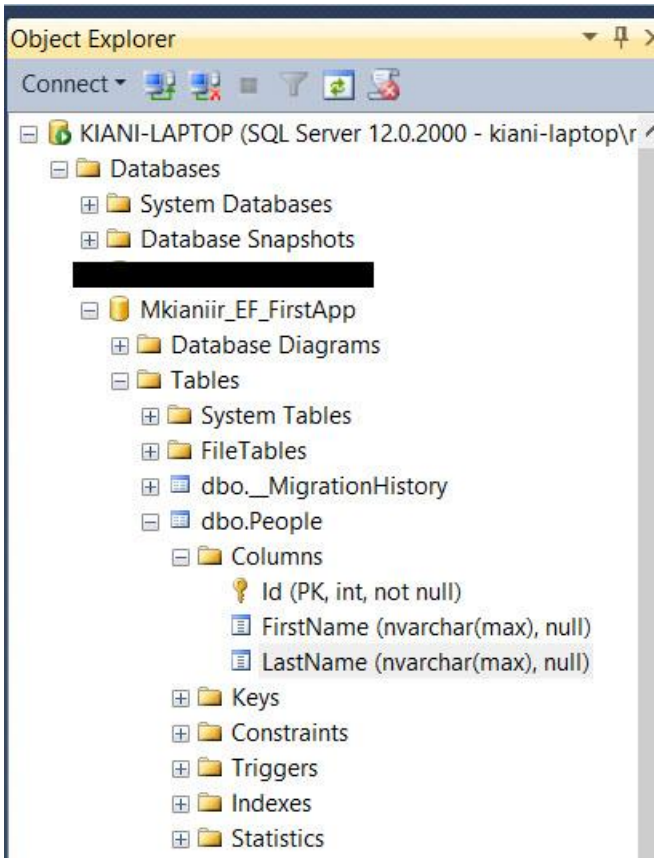
ایجاد پایگاه داده توسط کلاس PersonContext

کلاس Program را باز کنید و دستورات زیررا در متد Main بنویسید.

```
using (var context =new PersonContext())
          {
              context.Database.CreateIfNotExists();
          }
```

همانطور که مشاهده می کنید یک نمونه از کلاس PersonContext ایجاد شده است. خاصیت Database در کلاس DbContext اشاره به پایگاه داده خواهد داشت و دارای متد هایی برای هندل کردن آن می باشد. یکی از این متد ها CreateIfNotExists است. این متد چنانچه پایگاه داده ایجاد نشده باشد بر اساس اشیای تعریف شده درون آن اقدام به ایجاد پایگاه داده خواهد کرد.

حال برنامه را اجرا کنید. صفحه Console باز شده و بشته خواهد شد. اگر همه موارد را به درستی انجام داده باشید پایگاه داده ای به نام Mkianiir_EF_FirstApp در Sql Server Management Studio ایجاد خواهد شد.



شکل ۲- ۵

همانور که مشاهده می کنید پایگاه داده ای به نام Mkianiir_EF_FirstApp تشکیل شده است. درون پوشه Tables در این پایگاه داده جدولی به نام People ایجاد شده است. همانظور که مشخص است برای هر خاصیت در کلاس Person یک ستون هم نام آن ایجاد شده است.(ستون های Id ، FirstName و LastName)

همانطور که می دانید نوع داده nvarchar می توانید به صورت متغیر(با تعیین حداکثر طول) یا به صورت max باشد. اینکه چرا EF نوع ستون های FirstName و LastName را nvarchar(max) و یا ستون Id را به عنوان کلید اصلی شناخته در بخش های بعدی توضیح داده خواهند شد. همچنین اگر دقت کنید نام People به عنوان نام جدول برای کلاس Person در نظر گرفته شده است! این بدان معنی است که خود EF نام Person را به حالت جمع تبدیل کرده و نام People را برای آن در نظر گرفته است. این موضوع علاوه بر اینکه می تواند جالب باشد می تواند مورد انتظار شما نیز نباشد.ممکن است شما مایل باشید که نام Persons (نام خاصیت در کلاس PersonContext) به عنوان نام جدول در نظر گرفته شود.

هماطور که اشاره شد نیازی به نگرانی در این خصوص نمی باشد. در بخش های بعدی این موارد بیشتر توضیح داده می شوند. همچنین EF تمامی امکانات را در اختیار شما قرار خواهد داد تا بتوانید مدیریت کاملی بر آنچه که اتفاق می افتد داشته باشید.

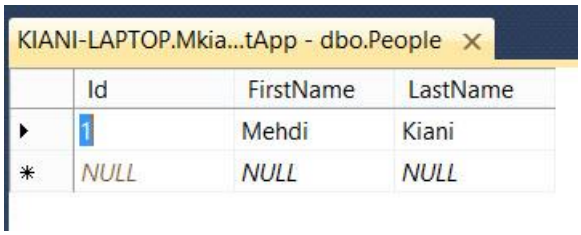
درج رکورد در جدول

فایل Program.cs را باز کنید و دستورات زیر را به آن اضافه کنید:

```
static void Main(string[] args)
{
    using (var context =new PersonContext())
    {
        context.Database.CreateIfNotExists();

        Person p1 = new Person
        {
            FirstName = "Mehdi",
            LastName = "Kiani"
        };
```

```
            context.Persons.Add(p1);
            context.SaveChanges();
        }


    }
```

حال برنامه را مجددا اجرا نمائید.پنجره Console باز و بسته خواهد شد. حال اگر به SSMS مراجعه کنید مشاهده خواهدی کرد که یک رکورد به جدول People اضافه شده است.


KIANI-LAPTOP.Mkia...tApp - dbo.People

	Id	FirstName	LastName
▶	1	Mehdi	Kiani
*	NULL	NULL	NULL

شکل ۲- ۶

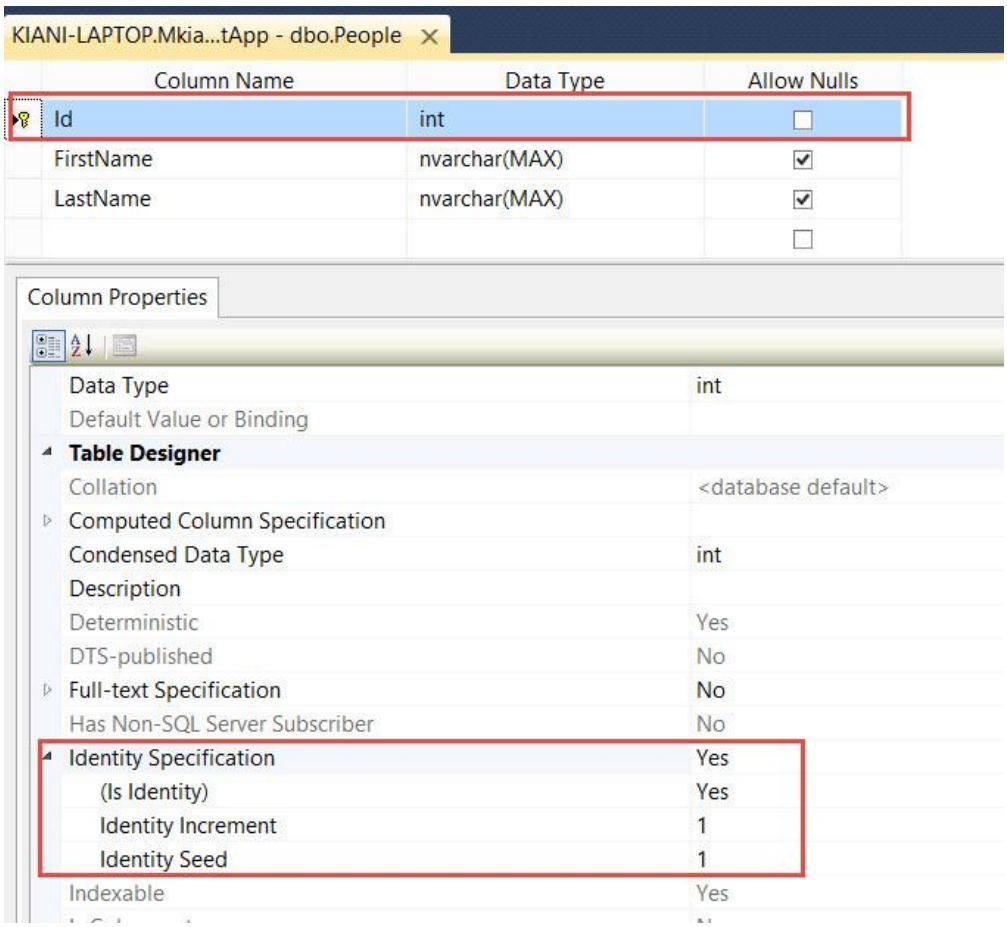
چه اتفاقی افتاد؟ شما یک نمونه از کلاس Person به نام p۱ ایجاد کردید و مقادیر FirstName و LastName را مقدار دهی کردید. سپس توسط شی context به خاصیت Persons که مجموعه ای از Person ها را نگهداری می کند دسترسی پیدا کرده و p۱ را به آن اضافه کرده ایم. سپس با استفاده از متد SaveChanges مربوط به DbContext تغییرات را در پایگاه داده منتقل کرده ایم!

همانطور که مشاهده می کنید بدون اینکه شما مستقیما از کلاس های ADO.Net نظیر SqlConnection ، SqlCommand و ... استفاده کنید به راحتی توانستید یک رکورد در جدولی درون پایگاه داده درج کنید. این موارد همگی به مدد EF قابل انجام شده است.

چرا خاصیت Id مقدار دهی نشده است؟

اگر به دستورات فوق مجددا دقت کنید متوجه خواهید شد که خاصیت Id مربوط به کلاس Person در شی p۱ مقدار دهی نشده است. حال اینکه در شکل فوق مشخص است که مقدار ۱ برای آن در نظر گرفته شده است.علت چیست؟

پاسخ به این سوال در این نگته است که ستون Id در جدول People به صورت Identity تعرف شده است و بنابر این مسئولیت ایجاد مقادیر این رکورد بر عهده پایگاه داده خواهد بود.



شکل ۲ - ۷

تمامی تنظیمات فوق توسط EF و بدون دخالت ما انجام شده است. البته این امکان وجود دارد که شما تمامی این تنظیمات را شخصا بر عهده بگیرید که در ادامه مباحث تا حد ممکن و نیاز این تنظیمات بیان خواهد شد.

دسترسی به اطلاعات

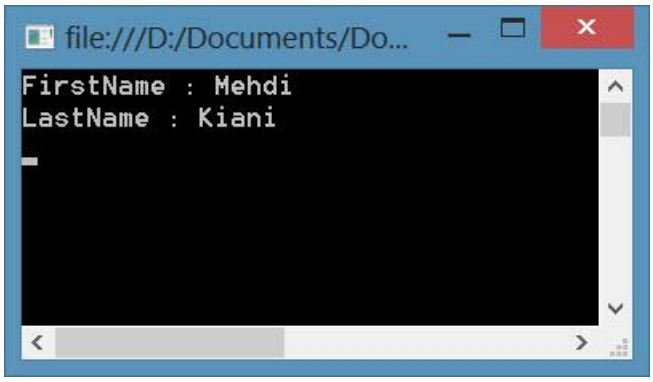
در بخش قبلی یک رکورد به جدول People اضافه کردیم. در این بخش به نحوه بازیابی اطلاعات جدول خواهیم پرداخت.

کلاس Program را باز کرده و دستورات متد Main را به صورت زیر تغییر دهید:

```
static void Main(string[] args)
{
    using (var context = new PersonContext())
    {
        context.Database.CreateIfNotExists();

        var persosns = context.Persons;
        foreach (var item in persosns)
        {
            Console.WriteLine("FirstName : " + item.FirstName);
            Console.WriteLine("LastName : " + item.LastName);
        }


    }
    Console.ReadKey();

}
```

همانطور که بیشتر گفته شد خاصیت Persons از کلاس PersonContext مجموعه ای از Person ها را در اختیار خود دارد. هر Person معادل یک رکورد از جدول People خواهد بود.بنابر این با یک دستور تکرار نظیر foreach می توانیم به تک تک رکورد های درون جدول People دسترسی پیدا کنیم.

برنامه را اجرا و خروجی را مشاهده کنید:



شکل ۲- ۸

ویرایش اطلاعات

برای ویرایش اطلاعات ابتدا می بایستی رکوردی را که می خواهیم ویرایش کنیم را واکشی کرده سپس تغییرات مورد نظر را اعمال کنیم و در نهایت تغییرات را ذخیره کرده تا در پایگاه داده اعمال شود.

دستورات متد Main کلاس Program را به صورت زیر تغییر دهید:

```
        static void Main(string[] args)
        {
            using (var context = new PersonContext())
            {
                context.Database.CreateIfNotExists();
                if (context.Persons.Any())
                {
                    var person = context.Persons.Where(w =>
w.FirstName.Contains("mehdi")).First();
                    DateTime updateDate = DateTime.Now;
                    person.FirstName = "mehdi_was updated at "
+updateDate.ToShortDateString();
                    person.LastName = "kiani_was updated at " +
updateDate.ToShortDateString();
                    context.SaveChanges();
                    Console.WriteLine("person updates successfully, press any key to
exit");
                }

            }
            Console.ReadKey();

        }
```

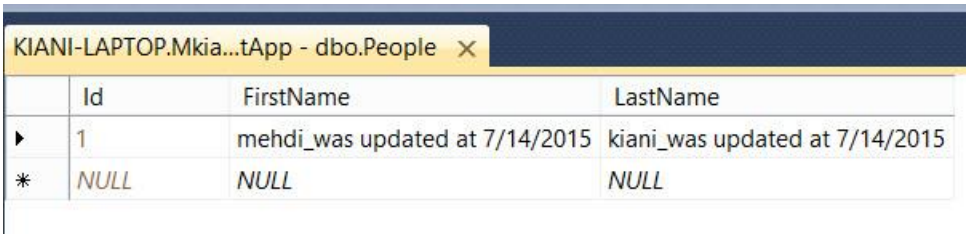
در دستور if با استفاده از متد الحاقی Any از دستورات LINQ بررسی کردیم که آیا مجموعه Persons دارای هیچ رکوردی هست یا خیر. چنانچه جواب مثبت است آنگاه اولین رکورد از بین رکورد هایی که ستون FirstName آن ها دارای عبارت mehdi می باشد در متغیری به نام person قرار خواهد گرفت.

سپس مقادیر FirstName و LastName را به عبارت دلخوا تغییر داده و نهایتا تغییرات را ذخیره کرده ایم:

خروجی حاصل از دستورات فوق به صورت زیر خواهد بود:

شکل ۲- ۹

حال اگر جدول People را باز کنیم خواهیم دید که مقادیر FirstName و LastName تغییر کرده است:



شکل ۲- ۱۰

در هنگام واکشی اطلاعات از دستورات LINQ برای فیلتر کردن اطلاعات استفاده شده است.درون متد Where که یک متد الحاقی می باشد از عبارات لامبدا برای فیلتر کردن رکورد ها استفاده شده است. همچنین متد First اولین رکورد درون مجموعه اطلاعات یافت شده را بر می گردند.

حذف اطلاعات

حذف اطلاعات نیز به سادگی سایر دستورات درج و ویرایش است. برای این منظور کافی است رکورد مورد نظر را از مجموعه رکورد ها یافته و سپس آن را از مجموعه رکورد ها حذف و نهایتا تغییرات را ذخیره کرد.

دستورات متد Main را به صورت زیر تغییر دهید:

```
static void Main(string[] args)
{
    using (var context = new PersonContext())
```

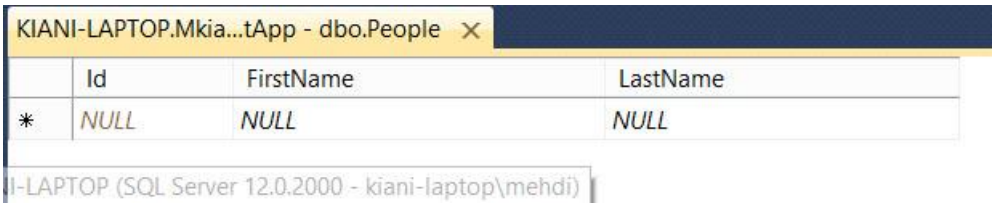
```
            {
                context.Database.CreateIfNotExists();
                if (context.Persons.Any())
                {
                    var person = context.Persons.Where(w =>
w.FirstName.Contains("mehdi")).First();
                    context.Persons.Remove(person);
                    context.SaveChanges();
                    Console.WriteLine("person was removed!, press any key to exit");
                }

            }
            Console.ReadKey();

        }
```

در دستورات فوق همانند دستورات ویرایش یک رکورد را واکشی کرده ایم. سپس توسط متد Remove آن را از مجموعه رکورد ها حذف و در نهایت تغییرات را ذخیره کرده ایم.

نتیجه اجرای دستورات فوق در شکل زیر نشان داده شده است:

شکل ۲- ۱۱

حال اگر به SSMS رجوع کنید مشاهده می کنید که رکورد مورد نظر از جدول People حذف شده است.



شکل ۲- ۱۲

تغییر در شمای پایگاه داده

اگر تا بع حال یک پروژه واقعی نوشته باشید می دانید که در طول حیات پروژه خود بارها و بارها نیاز است تا تغییراتی را در آن اعمال کنید. این تغییرات ممکن است مربوط به بخش کد نویسی پروژه یا مربوط به تغییرات پایگاه داده یا سایر تغییرات دیگر باشد. یکی از این تغییرات معمولا در پایگاه داده ها رخ می دهند. فیلد های جدیدی به یک جدول اضافه می شوند.فیلد هایی از یک جدول ممکن است حذف شوند.جدولی یا پروسیجری به پایگاه داده اضافه یا حذف شوند و ....

در هر تغییری که در پایگاه داده شما رخ می دهد شما می بایستی کد های پروژه خود را به نحوی تغییر دهید تا با ساختار جدید پایگاه داده همخوانی داشته باشد.

به عنوا مثال فرض کنید بخواهید موجودیت دیگری به نام Content به پروژه اضافه کنیم. برای این منظور یک کلاس به نام Content به شکل زیر به پروژه اضافه خواهیم کرد:

```
public class Content
 {
     public Int32 Id
     {
         get;
         set;
     }
     public String Title
     {
         get;
         set;
     }
     public String Body
     {
         get;
         set;
     }
     public String Author
     {
         get;
         set;
     }
 }
```

کلاس Content دارای چهار خاصیت Id ، Title ، Body و Author می باشد. بنابر این انتظار داریم در جدول متناظر با کلاس مذکور که در پایگاه داده تشکیل می شود برای هر یک از این خواص یک ستون متناظر تشکیل شود.

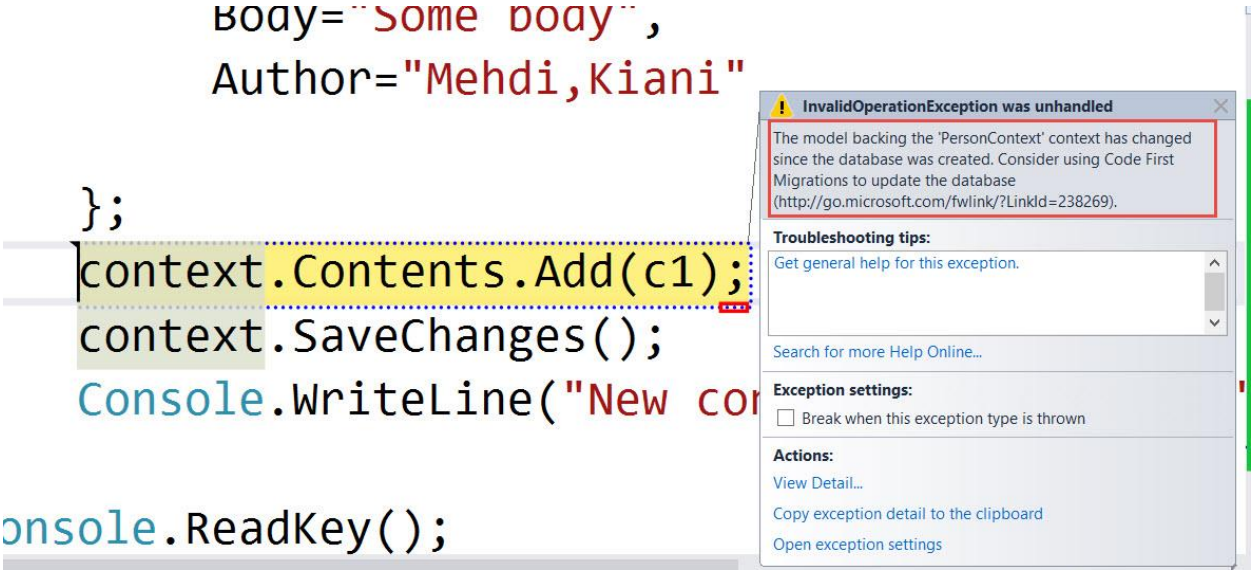
حال کلاس PersonContext را باز کرده و کدهای آن را مطابق زیر تغییر دهید:

```
public class PersonContext : DbContext
{
    public PersonContext()
        : base("mkianiir.EF")
    {

    }

    public DbSet<Person> Persons
    {
        get;
        set;
    }
    public DbSet<Content> Contents
    {
        get;
        set;
    }
}
```

حال دستورات متد Main از کلاس Program را به صورت زیر تغییر داده و برنامه را اجرا کنید:

```
    static void Main(string[] args)
    {
        using (var context = new PersonContext())
        {
            context.Database.CreateIfNotExists();

            Content c1 = new Content
            {
                 Title="Sample title",
                 Body="Some body",
                 Author="Mehdi,Kiani"

            };
            context.Contents.Add(c1);
            context.SaveChanges();
            Console.WriteLine("New content was created!");
        }
        Console.ReadKey();

    }
```

اگر برنامه را اجرا کنید با خطای زیر مواجه خواهید شد:



شکل ۲- ۱۳

اگر به متنی که با کادر قرمز رنگ در شکل فوق نشان داده شده است دقت کنید متوجه خواهید شد که مفهوم استثنای ایجاد شده بدین صورت است:

"مدل مربوط به کانتکست PersonContext نسبت به زمانی که پایگاه داده ایجاد شده است تغییر کرده است.برای بروز رسانی پایگاه داده از مفهوم Migration استفاده نمائید."

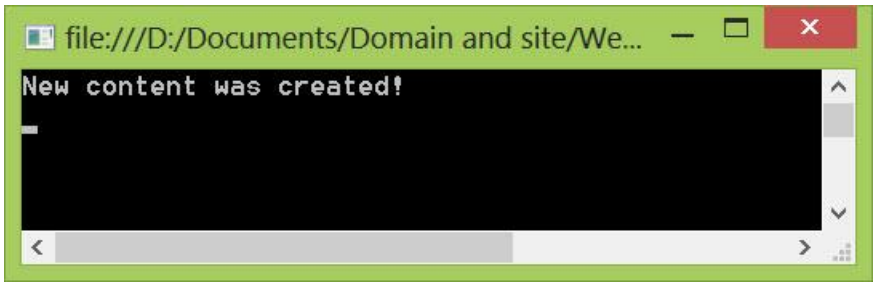
چه اتفاقی افتاده است؟ زمانی که برای اولین بار مدل داده ای را ساختیم و سپس پایگاه داده از روی آن مدل ساخته شد تنها موجودیت Person در آن تعریف شده بود که معادل با آن جدولی به نام People در پایگاه داده ایجاد شد. حال در این بخش موجودیت دیگری به نام Content را به پروژه اضافه کردیم. این اتفاق باعث شده است تا بین مدل داده ای و پایگاه داده نا همخوانی به وجود بیاید. در واقع درون کد می خواهیم به داده ای دسترسی پیدا کنیم که در پایگاه داده تعریف نشده است.

در این حالت می بایستی با استفاده از مفهومی به نام Migration که در بخش های بعدی به آن خواهیم پرداخت پایگاه داده را بروز رسانی کنیم.

فعلا تا آن زمان و برای رفع مشکل دستور زیر را به ابتدای متد Main اضافه کنید:

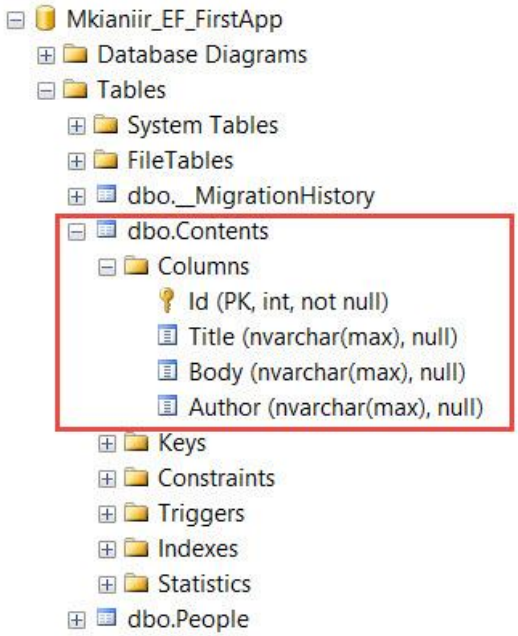
```
Database.SetInitializer(new DropCreateDatabaseIfModelChanges<PersonContext>());
```

حال اگر مجددا برنامه را اجرا کنید برنامه بدون خطا اجرا و دستورات مربوط به درج Content بدون مشکل اجرا خواهند شد.



شکل ۲- ۱۴

حال اگر سری به پایگاه داده بزنید مشکاهده خواهید کرد که علاوه بر جدول People یک جدول به نام Contents نیز به پایگاه داده اضافه شده است و یک رکورد نیز در ان درج شده است.



شکل ۲- ۱۵

KIANI-LAPTOP.Mkia...pp - dbo.Contents

	Id	Title	Body	Author
▶	1	Sample title	Some body	Mehdi,Kiani
*	NULL	NULL	NULL	NULL

شکل ۲- ۱۶

نکته: دستوری که برای حل مشکل فوق استفاده کردیم بدین معنی است که چنانچه مدل داده ای تغییر کرد، پایگاه داده فعلی حذف و مجددا ایجاد شود. به واژه حذف دقت کنید. زمانی که یک پایگاه داده حذف می شود چه اتفاقی می افتد؟ بله. تمامی اطلاعات قبلی درون جداول آن از بین خواهند رفت و این یقینا راهکاری نیست که شما بتوانید از آن در یک پروژه واقعی استفاده نمائید.همانطور که گفته شد راهکار آن بهره گیری از Migration بوده که در بخش های بعدی به توضیح آن خواهیم پرداخت.

خلاصه

تبریک می گویم(!) در این فصل شما توانستید اولین پروژه خود را با Entity Framework Code-First به انجام برسانید. در این فصل شما آموختید که چگونه یک شی DbContext ایجاد کنید. همچنین آموختید که برای شبیه سازی دنیای پیرامون خود اشیایی را توسط کلاس هایی ایجاد کنید و سپس توسط کلاس های تعریف شده در برنامه توانستید پایگاه داده را ایجاد کرده و نهایتا عملیات معمول پایگاه داده یعنی درج رکورد، دستیابی به رکورد ها، ویرایش و حذف رکورد ها از پایگاه داده را به انجام برسانید. در فصل بعدی با جزئیات بیشتری در رابطه با ساختار پایگاه داده، جداول و ستون های جداول و به طور کلی شمای پایگاه داده مطالبی را خواهید آموخت.

فصل سوم : طراحی ساختار پایگاه داده

در فصل قبل یک برنامه ساده بر اساس معماری EF و با پیاده سازی اعمال CRUD به طور خلاصه توضیح داده شد. در این بخش به امکانات بیشتری از EF در مورد طراحی مدل ها و انعکاس آن ها در پایگاه داده برای طراحی ساختار پایگاه داده خواهیم پرداخت.

نگاشت انواع داده ای بین .Net و Sql Server

اولین موردی که در این بخش نیاز است تا با آن آشنا شوید این است که بدانید معادل هر نو داده ای در .Net چه نوعی در Sql Server خواهد شد و این نگاشت توسط EF به چه شکلی انجام خواهد گرفت. در زیر جدول برخی از پرکاربردترین انواع داده ای دات نت و معادل های آن در Sql Server نشان داده شده اند.

SQL Server Database type	.NET Framework type
Bigint	Int64
binary, varbinary	Byte[]
Bit	Boolean
date, datetime, datetime۲, smalldatetime	DateTime
Datetimeoffset	DateTimeOffset
decimal, money, smallmoney, numeric	Decimal
float	Double
int	Int32
nchar, nvarchar,, char, varchar	String
real	Single
rowversion, timestamp	Byte[]
smallint	Int16

time	TimeSpan
tinyint	Byte
uniqueidentifier	Guid

جدول ۳ - ۱

اگر چه به خاطر سپاری جدول فوق بسیار ساده است اما این کار لزومی ندارد. چرا که EF عملیات نگاشت را برای شما انجام خواهد داد. همانطور که گفته شد لیست فوق شامل عمومی ترین انواعی است که معمولا در هر برنامه دات نتی مورد استفاده قرار می گیرد. جهت مشاهده لیست کامل این انواع و نگاشت های آن به آدرس

https://msdn.microsoft.com/en-us/library/cc۲۸٪۷۱۶۷۲۹v=vs.۲۹٪۱۱۰.aspx

رجوع کنید.

پیکر بندی و نگاشت بین انواع داده های .Net و Sql Server در EF

در بخش قبلی یک کلاس به نام Person تعریف کردیم که دارای سه خاصیت Id ، FirstNameو LastName بود. خاصیت Id از نوع int و خاصیت های FirstName و LastName از نوع String تعریف شدند.

همانطور که می دانید برای ذخیره مقادیر رشته ای در Sql Server انواع داده مختلفی وجود دارد. انواع Varchar ، Nvarchar ، Char و Nchar از این دسته هستند.

نکته: حرف N در انواع فوق معرف Unicode می باشد. این انواع برای کاراکتر های رشته ای بکار می روند که خارج از کاراکتر های زبان انگلیسی هستند. مانند حروف الفای زبان فارسی یا ژاپنی و ..... این انواع برای هر کاراکتر دو بایت فضا اشغال خواهند کرد.

و نیز همانطور که می دانید انواع فوق می توانند دارای طول های متغیر یا ثابت باشند. به عنوان مثال نوع داده char(۵۰) برای یک ستون به این معناست که داده های آن ستون می بایستی ۵۰ کاراکتر باشند. یا نوع داده nvarchar(۵۰) برای یک ستون به این معناست که داده های آن ستون می توانند حداکثر شامل ۵۰ کاراکتر یونیکدی باشند.

بنابر این نیاز به مکانیزمی است که EF بتواند این خصوصیات جداول و ستون های یک پایگاه داده را مشخص کند. در EF این کار به سه روش انجام می پذیرد:

۱- استفاده از صفت های موجود در فضای نام System.ComponentModel.DataAnnotations

۲- استفاده از فایل پیکربندی

۳- استفاده از کتابخانه DbModelBuilder که به EF Fluent API نیز معروف است.

استفاده از صفت ها برای پیکر بندی ساختار جداول و ستون ها

در فضای نام System.ComponentModel.DataAnnotations صفت هایی قرار دارند که به چند منظور از جمله اعتبار سنجی داده ها مورد استفاده قرار می گیرند. EF از همین صفات برای معرفی ساختار پایگاه داده تحت کنترل خود بهره می گیرد.

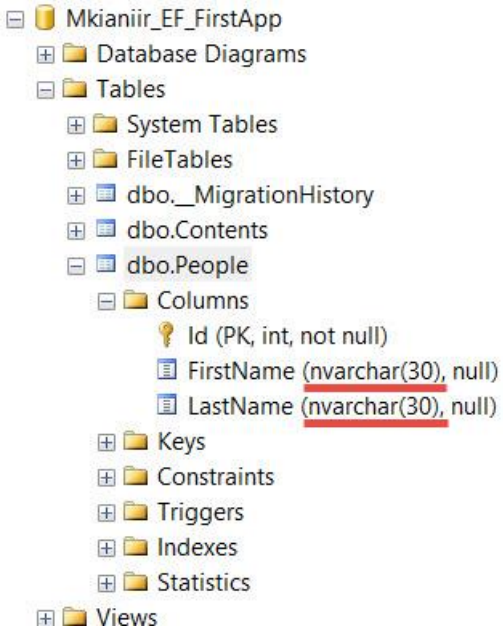
پروژه ای که در بخش قبلی ایجاد کردید را باز کنید و کلاس Person را به صورت زیر تغییر دهید:

```
public class Person
 {
     public Int32 Id
     {
         get;
         set;
     }
    [MaxLength(30)]
     public String FirstName
     {
         get;
         set;
     }
    [MaxLength(30)]
     public String LastName
     {
         get;
         set;
     }
 }
```

همانطور که مشاهده می کنید از صفت MaxLength برای خواص FirstName و LastName تعریف شده است. خوشبختانه نام کلاس هایی که برای این منظور در DataAnnotions تعریف شده است به قدر کافی گویای

عملکرد خود هستند. به عنوان مثال صفتی که در این مثال استفاده شده است بیانگر این است که ستون های FirstName و LastName حداکثر می توانند شامل ۳۰ کاراکتر باشند.

تغییرات را ذخیره و برنامه را محددا اجرا نمائید.



شکل ۳- ۱

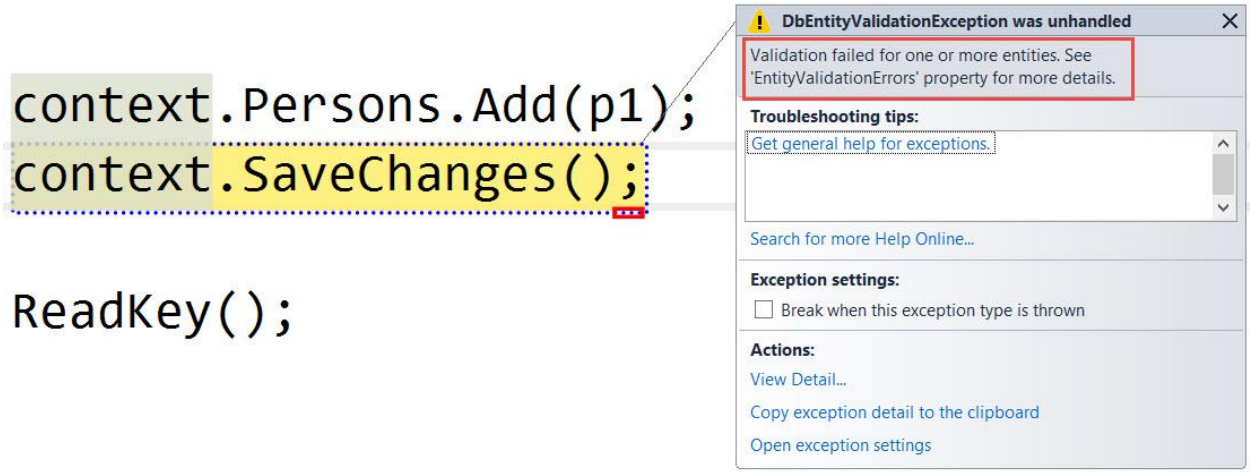
همانطور که مشاهده می کنید ستون های FirstName و LastName از نوع nvarchar(max) به نوع nvarchar(۳۰) تغییر پیدا کرده اند.

حال دستورات متد Main را به صورت زیر تغییر دهید:

```
        static void Main(string[] args)
        {
            using (var context = new PersonContext())
            {
                Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<PersonContext>());
                context.Database.CreateIfNotExists();

                    Person p1 = new Person
                           {
                               FirstName = "Mehdi_aaaaaaaaaaaaaaaaaaaaaaaaaaa",
                               LastName = "Kiani"
                           };

                    context.Persons.Add(p1);
```

```
                context.SaveChanges();
            }
            Console.ReadKey();

        }
```

همانطور که مشاهده می کنید طول مقدار داده ای که برای FirstName مورد استفاده قرار گرفته است بیش از ۳۰ کاراکتر است. حال برنامه را اجرا کنید.



شکل ۳- ۲

همانطور که مشاهده می کنید برنامه با استثنای DbEntityValidationException مواجه شده است. این استثنا در فضای نام System.Data.Entity.Validation تعریف شده است.

کلاس MaxLengthAttribute از کلاس ValidationAttribute مشتق می شود.کلاس ValidationAttribute دارای خاصیتی به نام ErrorMessage می باشد.خاصیت ErrorMessage برای نگهداری خطای اعتبار سنجی به کار می رود.

کلاس Person را به صورت زیر تغییر دهید:

```
public class Person
 {
```

```
        public Int32 Id
        {
            get;
            set;
        }
       [MaxLength(30,ErrorMessage="First name cannot be longer than 30")]
        public String FirstName
        {
            get;
            set;
        }
       [MaxLength(30, ErrorMessage = "Last name cannot be longer than 30")]
        public String LastName
        {
            get;
            set;
        }
    }
```

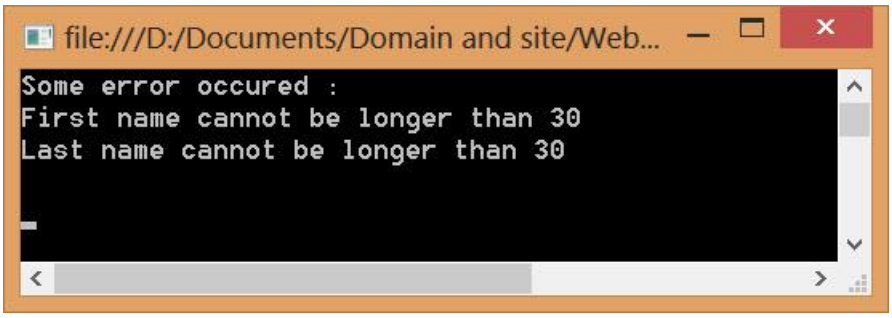
حال چنانچه خطای اعتبار سنجی در خصوص FirstName و LastName رخ دهد مقدار ErrorMessage به عنوان متن خطا درنظر گرفته می شود. استثنای DbEntityValidationException دارای خاصیتی به نام EntityValidationErrors می باشد که در شکل فوق نیز به آن اشاره شده است. خاصیت EntityValidationErrors مجموعه ای از کلاس DbEntityValidationResult می باشد. هرگاه در هر موجودیتی خطای اعتبار سنجی رخ دهد یک نمونه از کلاس DbEntityValidationResult برای ان موجودیت ایجاد شده، و به مجموعه EntityValidationErrors اضافه می شود. کلاس DbEntityValidationResult دارای خاصیتی به نام ValidationErrors می باشد. کلاس ValidationErrors مجموعه ای از کلاس DbValidationError است. برای هر خطای اعتبار سنجی درون یک موجودیت یک نمونه از کلاس DbValidationError ایجاد و در مجموعه ValidationErrors قرار خواهد گرفت. کلاس DbValitionError دارای خاصیتی به نام ErrorMessage می باشد. مقدار ErrorMessage درون صفت MaxLength درون مقدار ErrorMessage کلاس DbValidationError قرار خواهد گرفت.

برای مدیریت استثنای فوق و نمایش خطای رخ داده دستورات فوق را دورن بلاک trye-catche به صورت زیر قرار دهید:

```
        static void Main(string[] args)
        {
            using (var context = new PersonContext())
            {
                Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<PersonContext>());
```

```
            context.Database.CreateIfNotExists();

            try
            {
                Person p1 = new Person
                            {
                                FirstName = "Mehdi_aaaaaaaaaaaaaaaaaaaaaaaaaa",
                              LastName = "Kiani_aaaaaaaaaaaaaaaaaaaaaaaaaa"
                            };

                context.Persons.Add(p1);
                context.SaveChanges();
            }
            catch (DbEntityValidationException ex)
            {

                String errors = "Some error occured :";
                errors += Environment.NewLine;
                foreach (var eve in ex.EntityValidationErrors)
                {
                    foreach (var ve in eve.ValidationErrors)
                    {
                        errors += ve.ErrorMessage;
                        errors += Environment.NewLine;
                    }
                }
                Console.WriteLine(errors);
            }
        }
        Console.ReadKey();

}
```

حال برنامه را اجرا کنید و خروجی حاصل از اجرا را مشاهده کنید.



شکل ۳- ۳

همانطور که در شکل مشاهده می کنید دو خطای اعتبار سنجی در خروجی نمایش داده شده است.

استفاده از Fluent API

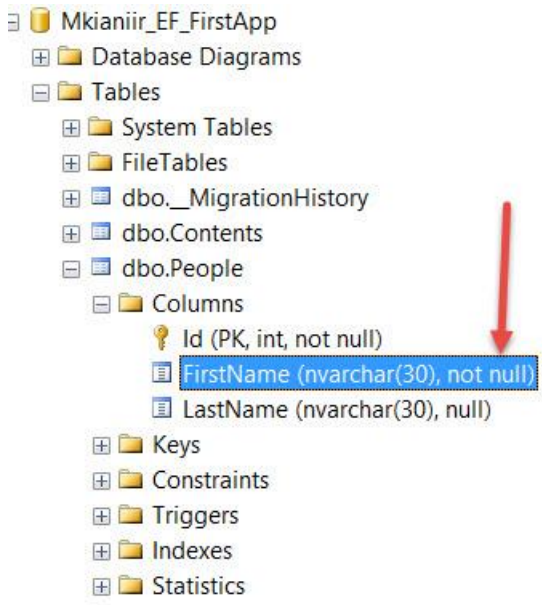
اگر چه استفاده از صفات موجود در فضای نام System.ComponentModel.DataAnnotations برای اعمال قواعد ساختاری پایگاه داده و اشیای درون آن بسیار مفید خواهد بود اما دارای محدودیت هایی است که امکان ایجاد تمامی قواعد را به شما نخواهد داد. در این حالت EF از کتابخانه DbModelBuilder که به Fluent API نیز معروف است بهره می گیرد.

برای استفاده از این کتابخانه می بایستی متد OnModelCreating را در کلاس PersonContext دوباره نویسی کنیم. همانطور که از نام این متد مشخص است قبل از اینکه مدل ایجاد شود و پایگاه داده ساخته شود این متد فراخوانی می گردد.

فایل PersonContext را باز کنید و دستورات زیر را به آن اضافه کنید:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<Person>().Property(p => p.FirstName).IsRequired();
        }
```

دستورات فوق بیانگر این است که خاصیت FirstName از کلاس Person نمی تواند مقدار Null داشته باشد.بنابر این زمانی که پایگاه داده ایجاد می شود ستون FirstName از جدول Person به صورت not null تعریف خواهد شد. این موضوع در شکل زیر نشان داده شده است.

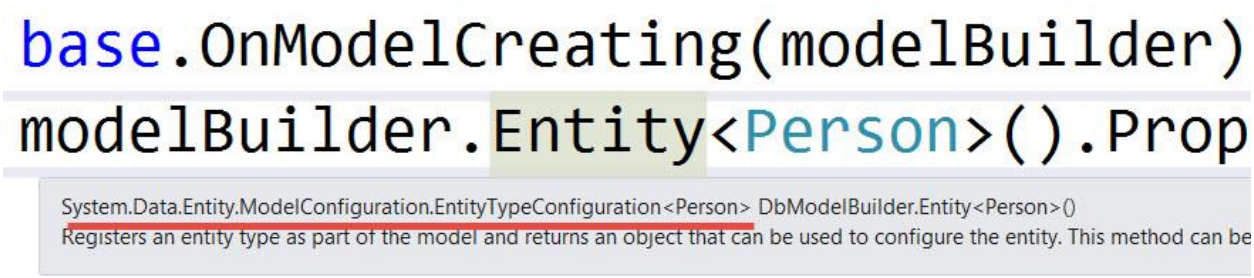


شکل ۳- ٤

استفاده از روش فوق اگرچه بسیار راحت و دستورات آن قابل فهم می باشند اما در یک پروژه واقعی که تعداد موجودیت (Entity) ها زیاد و هر موجودیت نیز دارای خواص مربوط به خود می باشد می تواند مدیریت کدهای نوشته شده را سخت کند.

EF برای این مواقع نیز راه حل خوبی به شما ارائه خواهد داد. البته این راه حل به نوعی همان راه حل قبلی است با این تفاوت که امکان مجزا کرن پیکربندی های هر موجودیت را به شکل جداگانه در اختیار شما قرار می دهد.

متد Entity از کلاس ModelBuiler که در دستورات فوق استفاده شده است یک نمونه از کلاس EntityTypeConfiguration بر می گرداند. و چون این متد به صورت ژنریک و با نوع Person به کار رفته است نوع بازگشتی این متد EntityTypeConfiguration<Person> می باشد.



شکل ۳- ۵

EF به شما اجازه می دهد که با ارث بری از کلاس EntityTypeConfiguration برای هر موجودیت دلخواهی، پیکربندی اجزای آن موجودیت را در یک کلاس جداگانه قرار داده و سپس این کلاس های پیکربندی را به مجموعه پیکربندی های کتابخانه Fluent اضافه کنید.

کلاسی به نام PersonConfig به پروژه اضافه کنید و دستورات آن را مطابق زیر تغییر دهید:

```
public class PersonConfig : EntityTypeConfiguration<Person>
  {
      public PersonConfig()
      {
          Property(p => p.FirstName).HasMaxLength(30).IsRequired();
      }
  }
```

کلاس PersonConfig از کلاس EntityTypeConfiguration همراه با نوع موجودیت Person مشتق شده است. در Cunstructor این کلاس از متد Property با عبارات لامبدا شبیه به آن چیزی که در دستورات قبلی مشاهده کردید استفاده کرده است.

حال می توانید متد OnModelCreating از کلاس PersonConfig را به مجموعه پیکربندی های EF اضافه نمائید. برای این منظور کد های کلاس PersonContext را به صورت زیر تغییر دهید:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Configurations.Add(new PersonConfig());
}
```

با استفاده از این روش می توانید پیکربندی های هر موجودیت را به صورت جداگانه ایجاد کنید. به این ترتیب هم از کتابخانه Fluent بهره برده اید و هم کدهای نوشته شده قابل مدیریت می باشند.

برقراری ارتباط بین موجودیت ها

همانطور که می دانید ارتباط بین موجودیت ها در پایگاه داده های امروزی جزء لاینفک طراحی ها می باشد. حرف R در عبارت RDBMS نیز معرف همین ارتباط می باشد. در طراحی پایگاه داده ها به طور کلی سه نوع ارتباط بین دو موجودیت می تواند وجود داشته باشد:

۱- ارتباط یک به یک

۲- ارتباط یک به چند

۳- ارتباط چند به چند

در ارتباطات یک به یک، یک رکورد از یک موجودیت با صفر یا یک رکورد از یک موجودیت دیگر در ارتباط خواهد بود. ارتباط بین موجودیت Person و PersonDetails که در آن هر رکورد از PersonDetails جزئیات مربوط به یک رکورد از Person را نگهداری می کند نمونه ای از این ارتباط می تواند باشد. در این حالت جدول Person جدول اصلی و جدول PersonDetails جدول وابسته می باشد. تا زمانی که رکوردی در جدول Person موجود نباشد امکان درج رکوردی در جدول PersonDetails نخواهد بود.

در ارتباط یک به چند، یک رکورد از یک موجودیت با صفر، یک یا چند رکورد از موجودیت دیگر در ارتباط خواهد بود. ارتباط بین Person و Book می تواند از این دسته باشد. یعنی یک رکورد از موجودیت Person می تواند با صفر ، یک یا با چند رکورد از موجودیت Book ارتباط داشته باشد. معنای مفهومی آن هم این است که هر شخص می تواند دارای صفر، یک یا چند کتاب باشد.

در ارتباط چند به چند، یک یا چند رکورد از موجودیت اول با صفر یک یا چند رکورد از موجودیت دوم و بلعکس ارتباط خواهند داشت. ارتباط استاد و درس می تواند از این نوع باشد. هر استاد می تواند صفر،یک و یا چند درس را تدریس کند و هر درس نیز می تواند توسط صفر،یک یا چند استاد ارائه شود. همچنین ارتباط بین Personو Book نیز می تواند چند به چند باشد. بدین معنی که هر شخص می تواند نویسنده چند کتاب باشد و هر کتاب می تواند دارای بیش از یک نویسنده باشد. جدول کاربران(Users) و نقش ها(Roles) نیز یکی دیگر از نمونه های ارتباط

چند به چند می باشد که در بسیاری از برنامه ها مورد استفاده قرار می گیرد.هر کاربر می تواند دارای صفر،یک یا چند نقش باشد و همچنین هر نقش می تواند متعلق به صفر،یک یا چند کاربر باشد. پس مشاهده می کنید که ارتباطات بین موجودیت ها می تواند از یک برنامه به برنامه دیگر متفاوت باشد.

از سه نوع ارتباط فوق، ارتباط سوم هنگام طراحی و در فرایند نرمال سازی به ارتباط هایی از نوع دوم تبدیل خواهد شد. همچنین ارتباط از نوع اول یعنی ارتباطات یک به یک عموما به ندرت در طراحی ها استفاده می گردد. اما ارتباط نوع دوم یعنی ارتباط یک به چند بیشترین نوع ارتباطی است که امروزه در پایگاه داده های رایج نظیر Sql Server مورد استفاده قرار می گیرد.

EF با ایجاد خواص در کلاس هایی که معرف جداول هستند می تواند این ارتباطات را ایجاد کنند.

ارتباط یک به یک

همانطور که گفته شد این نوع ارتباط خیلی در طراحی پایگاه داده های امورزی موسوم نیست. چراکه غالبا ترکیب دو جدولی که دارای ارتباط یک به یک هستند می توانند به یک جدول تبدیل شوند. EF نیر این قابلیت را برای ایجاد این نوع ارتباطات برای شما مهیا می سازد. ارتباط یک یه بک بیتشر در مواقعی کاربرد دارد که بخواهیم جزئیاتی را برای یک رکورد در یک جدول مجزا در نظر بگیریم.معمولا زمانی که بخواهیم اطلاعاتی که دارای حجم زیادی هستند مثلا عکس کاربران را نگهداری کنیم. عموما در این حالت ستون های مربوط به این نوع داده ها را در یک جدول دیگر نگهداری خواهیم کرد.

برای مثال یک کلاس به نام PersonInfo ایجاد کنید و دستورات آن را طبق زیر تغییر دهید:

```
public class PersonInfo
 {

     public Int32 Id
     {
         get;
         set;
     }
     public String FatherName
     {
         get;
         set;
```

```
        }

        public String IdentityNo
        {
            get;
            set;
        }
        public Int32 Age
        {
            get;
            set;
        }
        Public virtual Person Person
        {
            get;
            set;
        }
    }
```

همانطور که می بینید یک خاصیت از نوع کلاس Person علاوه بر خواص دیگر به کلاس PersonInfo اضافه شده است.

همچنین خاصیت زیر را به کلاس Person اضافه کنید:

```
public virtual PersonInfo Info
       {
           get;
           set;
       }
```

همانطور که می دانید در یک ارتباط یک به یک بین دو موجودیت، یکی از آن ها موجودیت اصلی و دیگری موجودیت وابسته می باشد. این یعنی اینکه تا زمانی که رکوردی درون موجودیت اصلی وجود نداشته باشد امکان ایجاد رکورد برای موجودیت وابسته نیست. در این مثال موجودیت Person موجودیت اصلی و PersonInfo موجودیت وابسته می باشد. این تعاریف اصلی و وابستگی را نیز می بایستی در پیکربندی EF مشخص کنیم تا EF بتواند آن را برای پایگاه داده اعمال کند.

برای این منظور کلاس PersonConfig را بازکنید و دستورات آن را طبق زیر تغییر دهید:

```
    public class PersonConfig : EntityTypeConfiguration<Person>
    {
        public PersonConfig()
        {
            Property(p => p.FirstName).HasMaxLength(30).IsRequired();
            HasOptional(o => o.Info).WithRequired(r => r.Person);
```

```
        }
    }
```

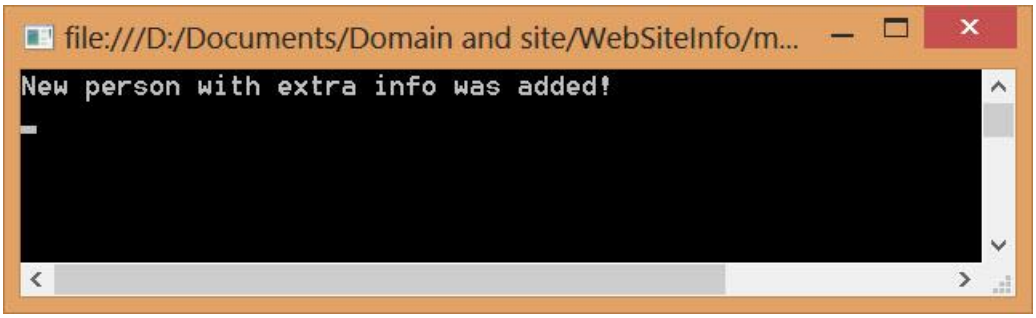
در کلاس پیکربندی PersonConfig به خطی که با رنگ زرد مشخص شده است دقت کنید. توسط متد HasOptional مشخص کرده ایم که مقدار فیلد Info از کلاس Person می تواند اختیاری باشد. اما فیلد Person از کلاس PersonInfo می بایستی الزامی باشد. این دستور توسط متد WithRequired انجام شده است.

برای تست تغییرات جدید متد Main مربوط به کلاس Program را به صورت زیر تغییر دهید:

```
        static void Main(string[] args)
        {
            using (var context = new PersonContext())
            {
                Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<PersonContext>());
                context.Database.CreateIfNotExists();

                try
                {
                    Person p1 = new Person
                                {
                                    FirstName = "p1_first_name",
                                    LastName = "p1_last_name"
                                };
                    p1.Info = new PersonInfo
                    {
                         FatherName="p1_father_name",
                          IdentityNo="125",
                           Age=55

                    };
                    context.Persons.Add(p1);
                    context.SaveChanges();
  Console.WriteLine("New person with extra info was added!");
                }
                catch (DbEntityValidationException ex)
                {

                    String errors = "Some error occured :";
                    errors += Environment.NewLine;
                    foreach (var eve in ex.EntityValidationErrors)
                    {

                        foreach (var ve in eve.ValidationErrors)
                        {

                            errors += ve.ErrorMessage;
                            errors += Environment.NewLine;
                        }
                    }
                    Console.WriteLine(errors);
```

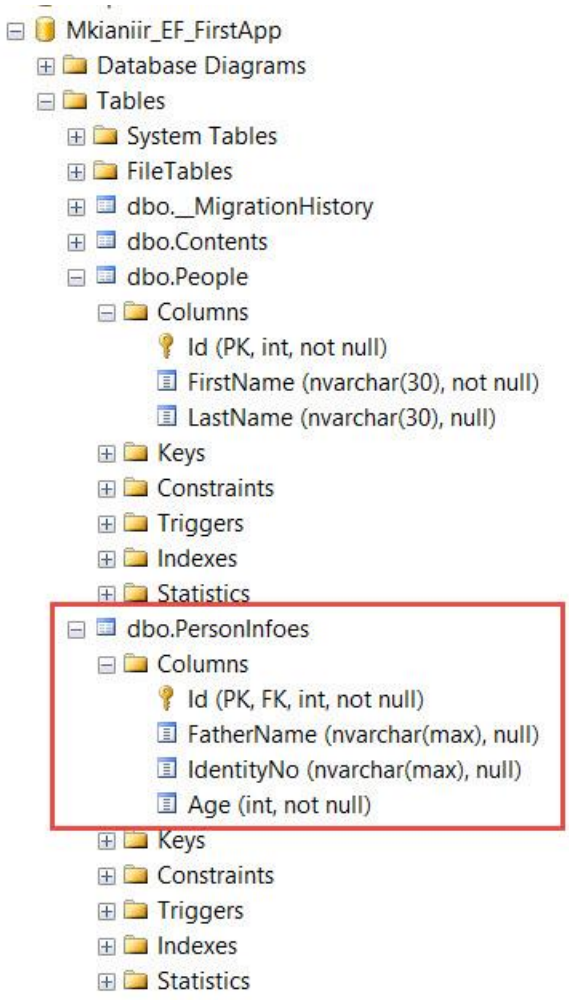
```
            }
        }
        Console.ReadKey();

    }
```

در دستورات فوق در قسمت هایلات زرد رنگ همانطور که مشخص است یک نمونه از کلاس PersonInfo ایجاد شده و به خاصیت Info شی p۱ نسبت داده شده است. اگر دستورات را مطابق آنچه که بیان کردم انجام داده باشید پس از اجرای برنامه با خروجی شبیه به شکل زیر مواجه خواهید شد.



شکل ۳- ۶

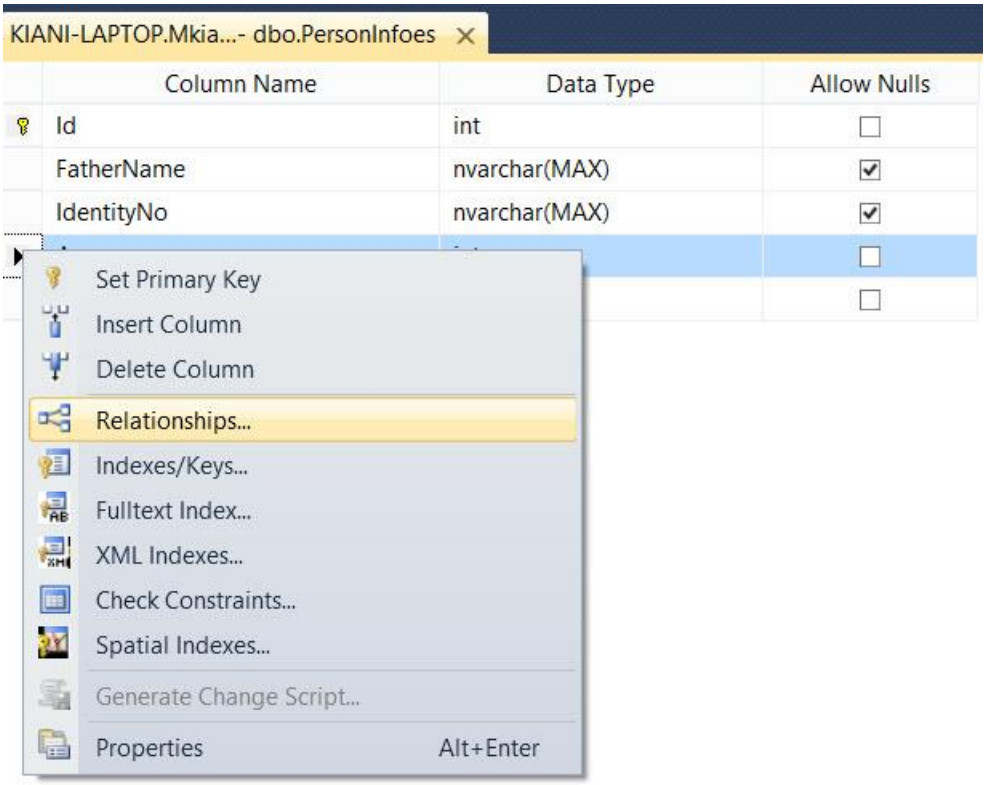
حال به پایگاه داده رفته و تغییرات را مشاهده کنید.



شکل ۳- ۷

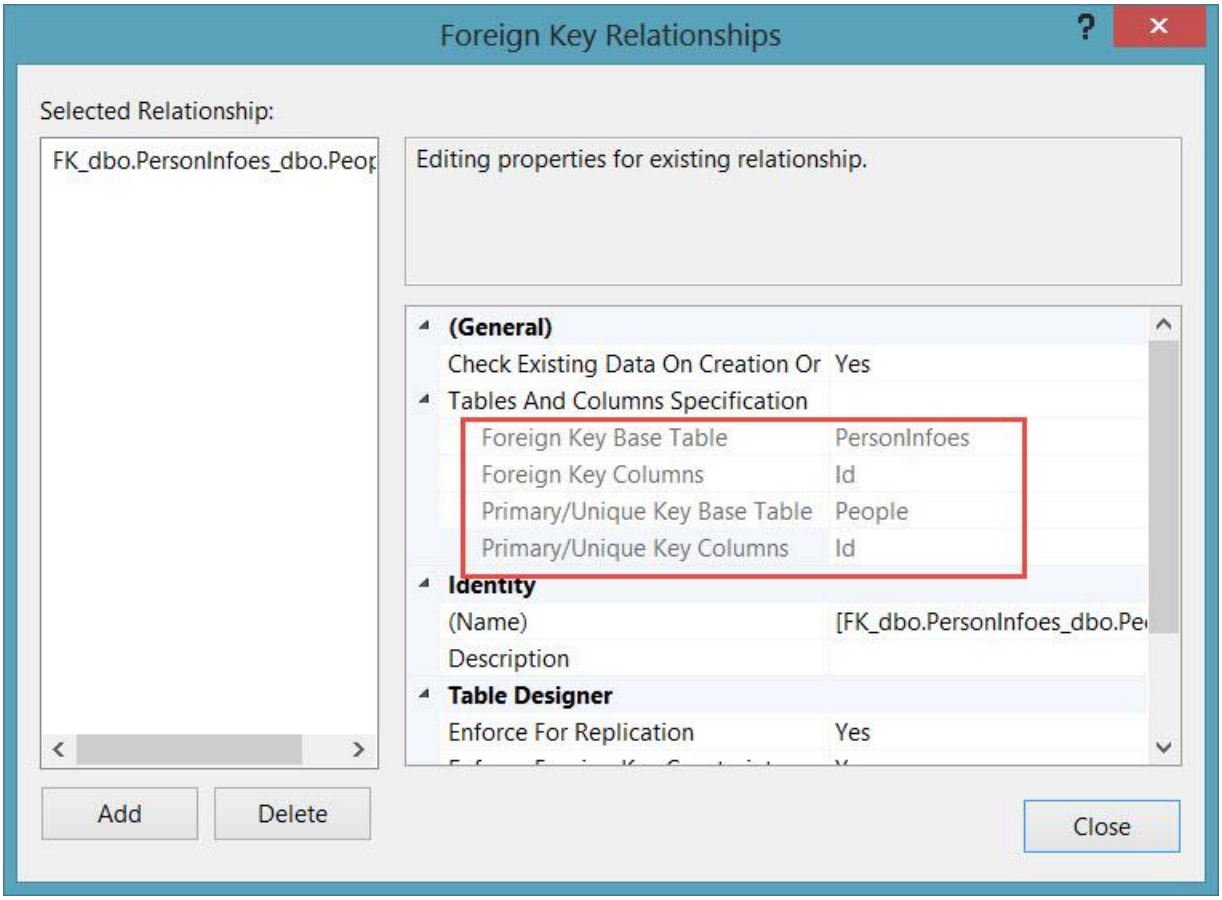
همانطور که مشاهده می کنید جدولی به نام PersonInfoes ایجاد شده است.دقت داشته باشید که خواص Person و Info به ترتیب در کلاس های PersonInfo و Person تعریف شدند در پایگاه داده اثری از آن ها نیست.خوب نیازی هم به آن ها نیست. EF از آن ها استفاده می کند تا بتواند شمای پایگاه داده را ایجاد کرده و در هنگام برنامه نویسی داده های واکشی شده و ارتباط بین موجودیت ها را توسط این خواص در اختیار شما قرار دهد.

بر روی جدول PersonInfos کلیک راست کنید و گزینه Relationships را کلیک کنید:



شکل ۳- ۸

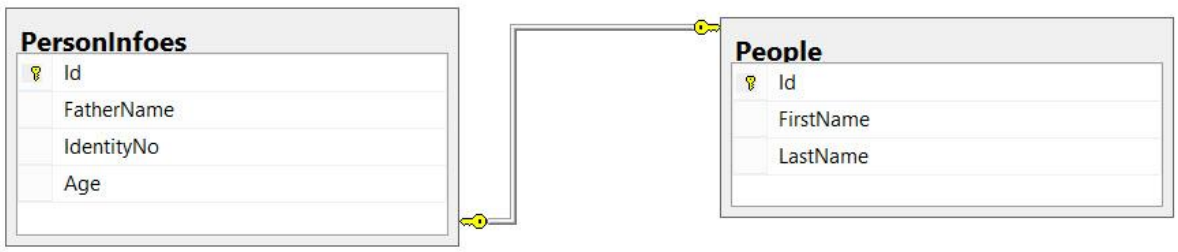
اگر به پنجره Foreign Key Relationships دقت کنید یک ارتباط بین جداول People و PersonInfos ایجاد شده است.



شکل ۳- ۹

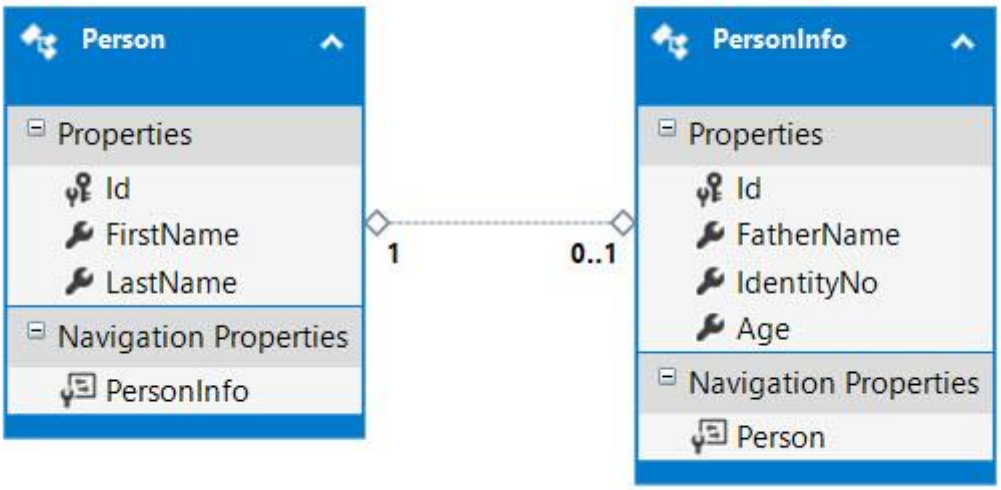
هانطور که مشاهده می کنید فیلد Id از جدول People با فیلد Id از جدول PersonInfos در یک ارتباط یک به یک قرار گرفته اند.

این ارتباط در دیاگرامی که با این دو جدول ایجاد شده نیز نشان داده شده است:



شکل ۳- ۱۰

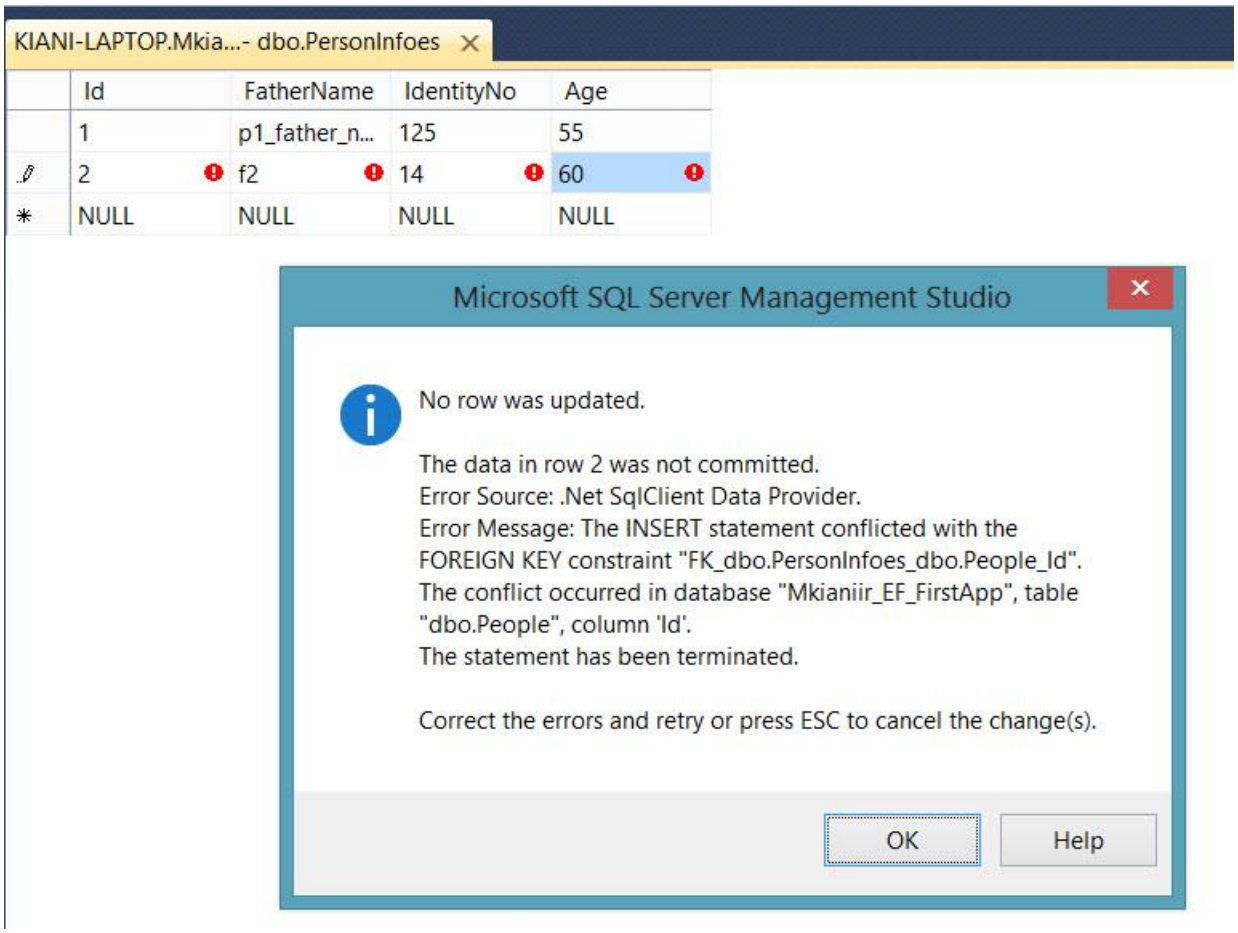
اگر یک فایل Entity Data Model برای دو جدول مذکور ایجاد کنید ارتباطی شبیه به شکل زیر را به شما نشان خواهد داد.



شکل ۳- ۱۱

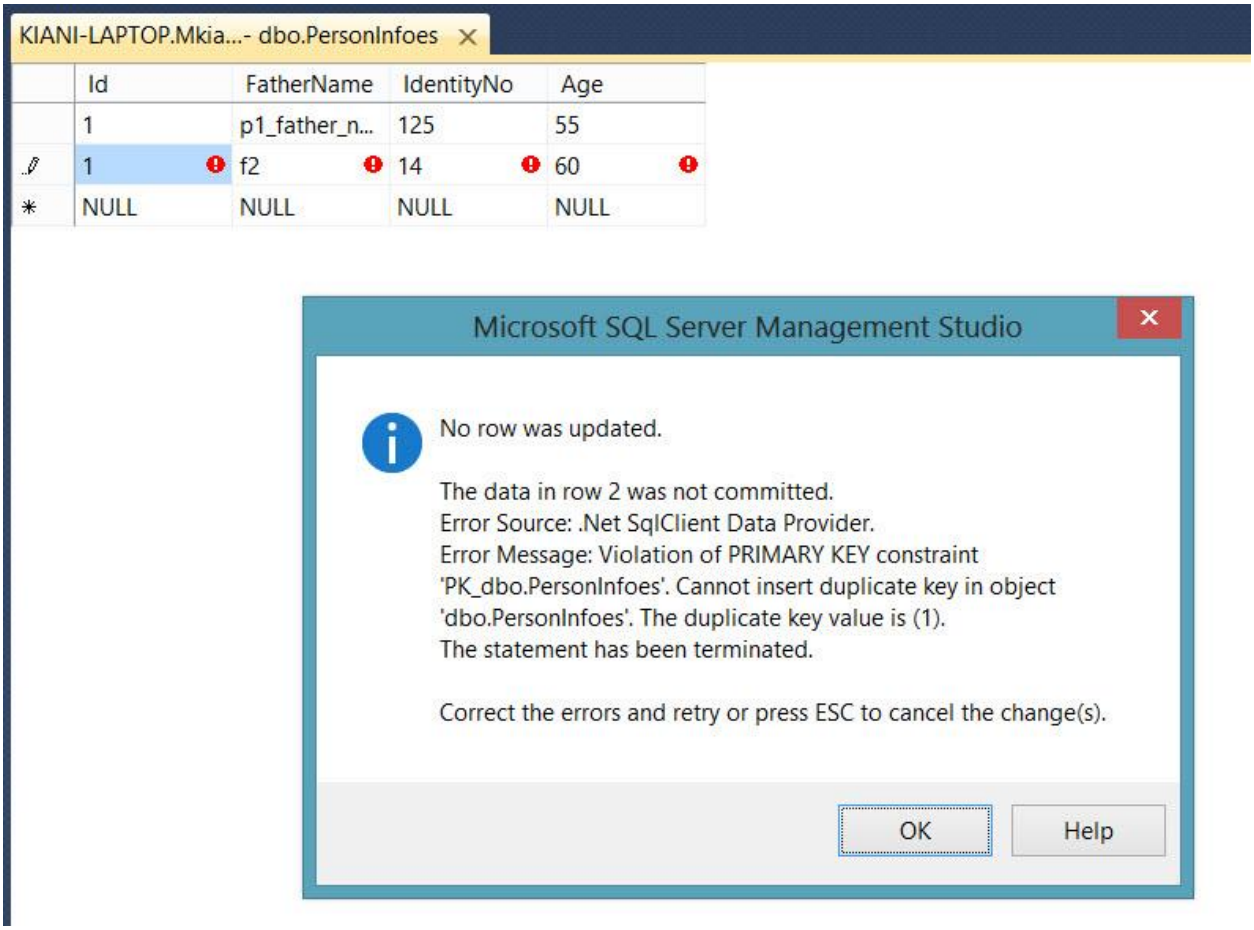
همانطور که در شکل می بینید یک رکورد از جدول People می تواند با صفر یا یک رکورد از جدول PersonInfoes ارتباط داشته باشد. مقدار صفر به این معناست که جدول PersonInfos موجودیتی وابسته می باشد. درواقع جدول People می تواند دارای رکوردی باشد که متناظر با آن هیچ رکوردی در جدول PersonInfos نباشد اما عکس این مطلب امکان پذیر نمی باشد.

در پایگاه داده نیز این مهم به دقت بررسی می شود. سعی کنید که یک رکورد دستی در جدول PersonInfos با Id که مقدار آن ۲ است وارد نمائید. بله Sql Server مجوز این کار را به شما نمی دهد. چرا که هیچ رکوردی با Id که مقدار ان ۲ باشد در جدول People درج نشده است.



شکل ۳- ۱۲

حال سعی کنید یک رکورد با مقدار ۱ برای Id در جدول PersonInfos وارد کنید. بله این کار هم به شما اجازه داده نخواهد شد. چرا؟ چون ارتباط بین People و PersonInfos یک ارتباط یک به یک می باشد و چون یک روکورد با مقدار ۱ برای Id در جدول PersonInfos درج شده است بنابر این دیگر امکان درج رکورد با Id تکراری وجود نخواهد داشت.



شکل ۳- ۱۳

ارتباط یک به چند:

این نوع ارتباط پرکاربردترین نوع ارتباط در پایگاه داده های امروزی می باشد. در این نوع ارتباط یک رکورد از یک موجودیت می تواند با صفر، یک و یا چند رکورد از موجودیت دوم ارتباط داشته باشد. در این ارتباط موجودیت اول موجودیت اصلی و دومی وابسته می باشد.

یک کلاس به نام Post به صورت زیر ایجاد کنید:

```
public class Post
{

    public Int32 Id
    {
        get;
        set;
    }
```

```
        public String Title
        {
            get;
            set;
        }
        public String Body
        {
            get;
            set;
        }

        public Int32 AuthorId
        {
            get;
            set;
        }
        Public virtual Person Author
        {
            get;
            set;
        }

    }
```

کلاس Post علاوه بر خواص ویژه خود مانند Titleو Body و Id دارای دو خاصیت دیگر به نام های AuthorId و Author می باشد. خاصیت AuthorId به عنوان کلید خارجی عمل خواهد کرد. همچنین خاصیت Author به موجودیت Person اشاره خواهد داشت.

حال کلاس Person را باز کنید و دستورات زیر را به آن اضافه کنید:

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

کلاس Person دارای خاصیتی به نام Posts از جنس ICollection<Post> می باشد. این خاصیتی لیستی از Post های موجودیت Person را در بر می گیرد.

حال باید نحوه ارتباط را در کلاس پیکربندی Person اعمال کنیم.

کلاس PersonConfig را باز کنید و دستورات آن را طبق زیر تغییر دهید:

```
    public class PersonConfig : EntityTypeConfiguration<Person>
    {
        public PersonConfig()
        {
            Property(p => p.FirstName).HasMaxLength(30).IsRequired();
```

```
            HasOptional(o => o.Info).WithRequired(r => r.Person);
            HasMany(m => m.Posts).WithRequired(r => r.Author).HasForeignKey(f =>
f.AuthorId);
        }
    }
```

در دستورات فوق به خط زرد رنگ دقت کنید. توسط متد HasMany مشخص کرده ایم که موجودیت Person می تواند مجموعه ای از Post ها را داشته باشد. همچنین توسط متد WithRequired اعلام شده است که در موجودیت Post الزاما می بایست Author که نمونه ای از موجودیت Person می باشد تعریف شده باشد. این بدان معنی است که هر رکورد از موجودیت Person می تواند با صفر،یک و یا مجموعه ای از موجودیت Post در ارتباط باشد.همچنین موجودیت Person یک موجودیت اصلی و موجودیت Post یک موجودیت وابسته است. چرا که موجودیت Person می تواند دارای هیچ Post ای نباشد اما عکس این قضیه صحیح نمی باشد. این از قواعد رابطه یک به چند می باشد. این اصلی بودن و وابستگی توسط خاصیت AuthorId به عنوان کلید خارجی مشخص شده است.

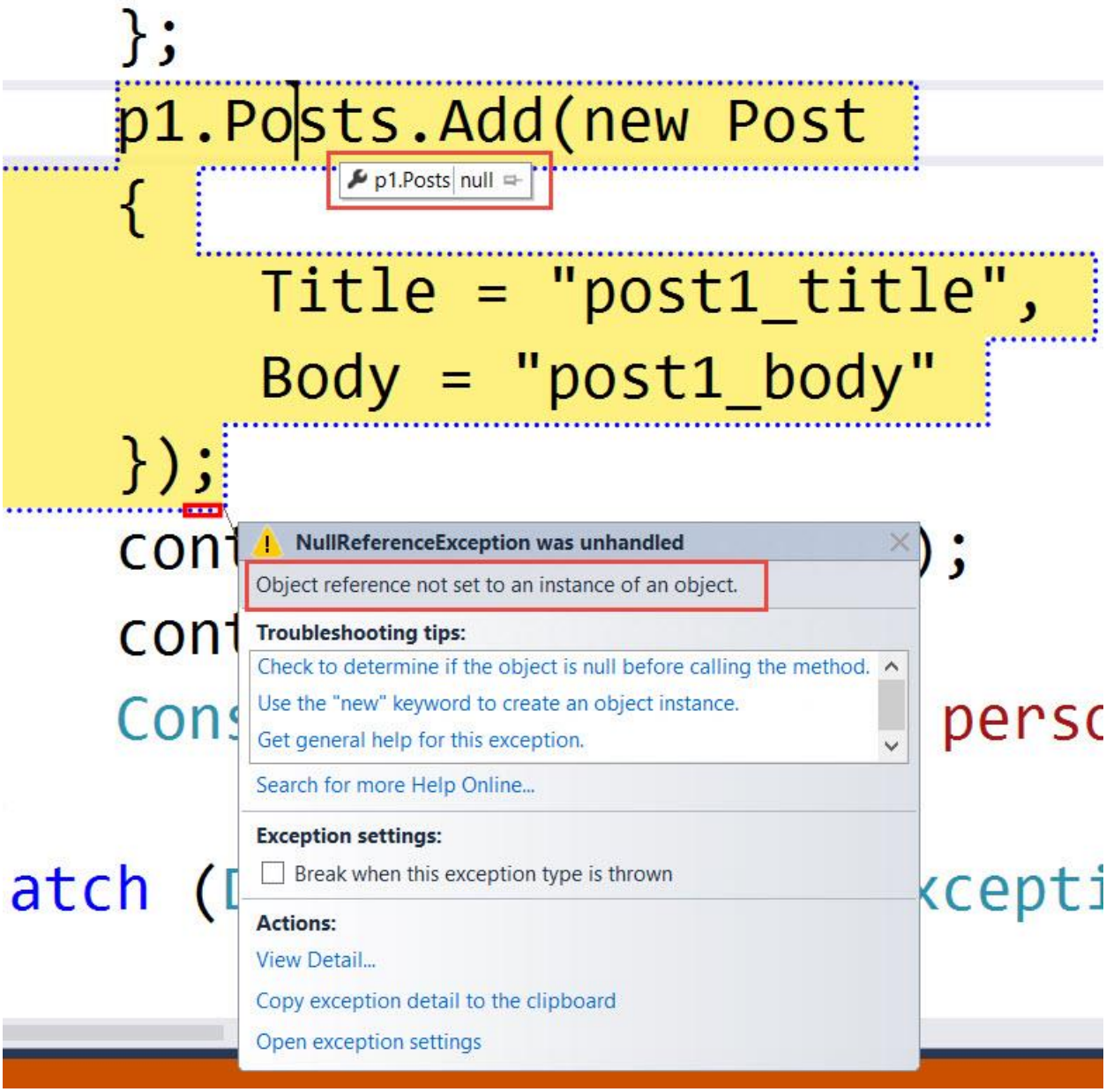
دستورات متد Main کلاس Program را به صورت زیر تغییر دهید :

```
        static void Main(string[] args)
        {
            using (var context = new PersonContext())
            {
                Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<PersonContext>());
                context.Database.CreateIfNotExists();

                try
                {
                    Person p1 = new Person
                                {
                                    FirstName = "p1_first_name",
                                    LastName = "p1_last_name"
                                };
                    p1.Info = new PersonInfo
                    {
                        FatherName = "p1_father_name",
                        IdentityNo = "125",
                        Age = 55

                    };
                    p1.Posts.Add(new Post
                    {
                        Title = "post1_title",
                        Body = "post1_body"
                    });
                    context.Persons.Add(p1);
                    context.SaveChanges();
                    Console.WriteLine("New person with extra info was added!");
                }
```

```
            catch (DbEntityValidationException ex)
            {

                String errors = "Some error occured :";
                errors += Environment.NewLine;
                foreach (var eve in ex.EntityValidationErrors)
                {

                    foreach (var ve in eve.ValidationErrors)
                    {

                        errors += ve.ErrorMessage;
                        errors += Environment.NewLine;
                    }
                }
                Console.WriteLine(errors);
            }
        }
        Console.ReadKey();

    }
```

به دستوراتی که با رنگ زرد مشخص شده اند دقت کنید. این دستورات یک نمونه از موجودیت Post ایجاد کرده و به مجموعه Post های Person اضافه کرده است.

اگر برنامه را اجرا کنید برنامه با استثنایی که در شکل زیر نشان داده شده است برخورد خواهد کرد:



شکل ۳- ۱٤

همانطور که از عکس فوق کاملا مشخص است استثنای NullReference رخ داده است. می دانید که این استثنا زمانی رخ خواهد داد که سعی در دسترسی به اعضای شی که null است داشته باشیم. در اینجا نیز مقدار خاصیت Posts برابر با null است که در شکل نیز نشان داده شده است. برای این منظور کافی است تا در Cunstructor

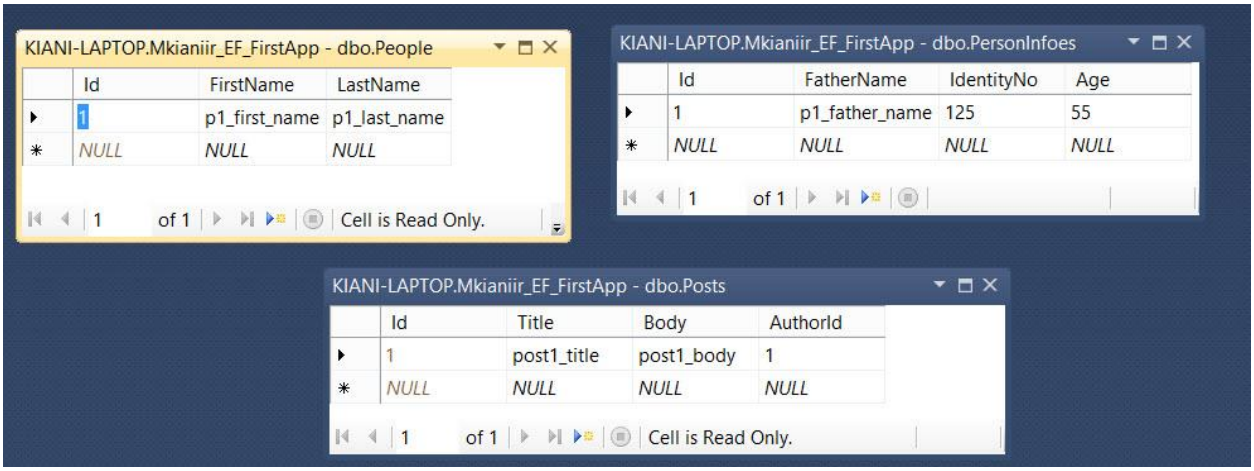
کلاس Person خاصیت Posts را مقدار دهی اولیه کنیم. برای این منظور کلاس Person را به صورت زیر تغییر دهید:

```
public partial class Person
{
    public Person()
    {
        Posts = new List<Post>();
    }
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public virtual PersonInfo Info { get; set; }

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

حال مجددا برنامه را اجرا کنید. اگر دستورات را طبق آنچه که بیان شد انجام داده باشید باید با شکلی شبیه به شکل زیر برخورد کنید:

شکل ۳- ۱۵

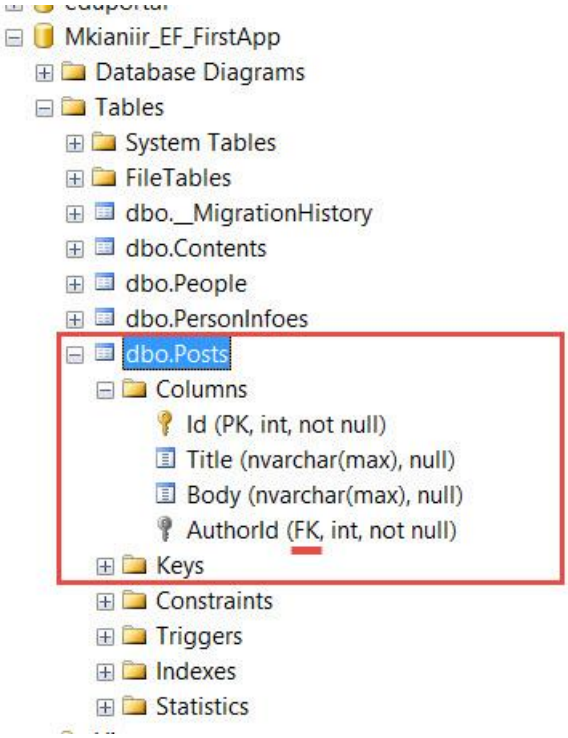
همانطور که می بینید برنامه اجرا شده و یک نمونه از کلاس Person به همراه موجودیت های وابسته به آن نظیر PersonInfo و Post در پایگاه داده درج شده است. این موارد در شکل زیر نشان داده شده است:



شکل ۳- ۱۶

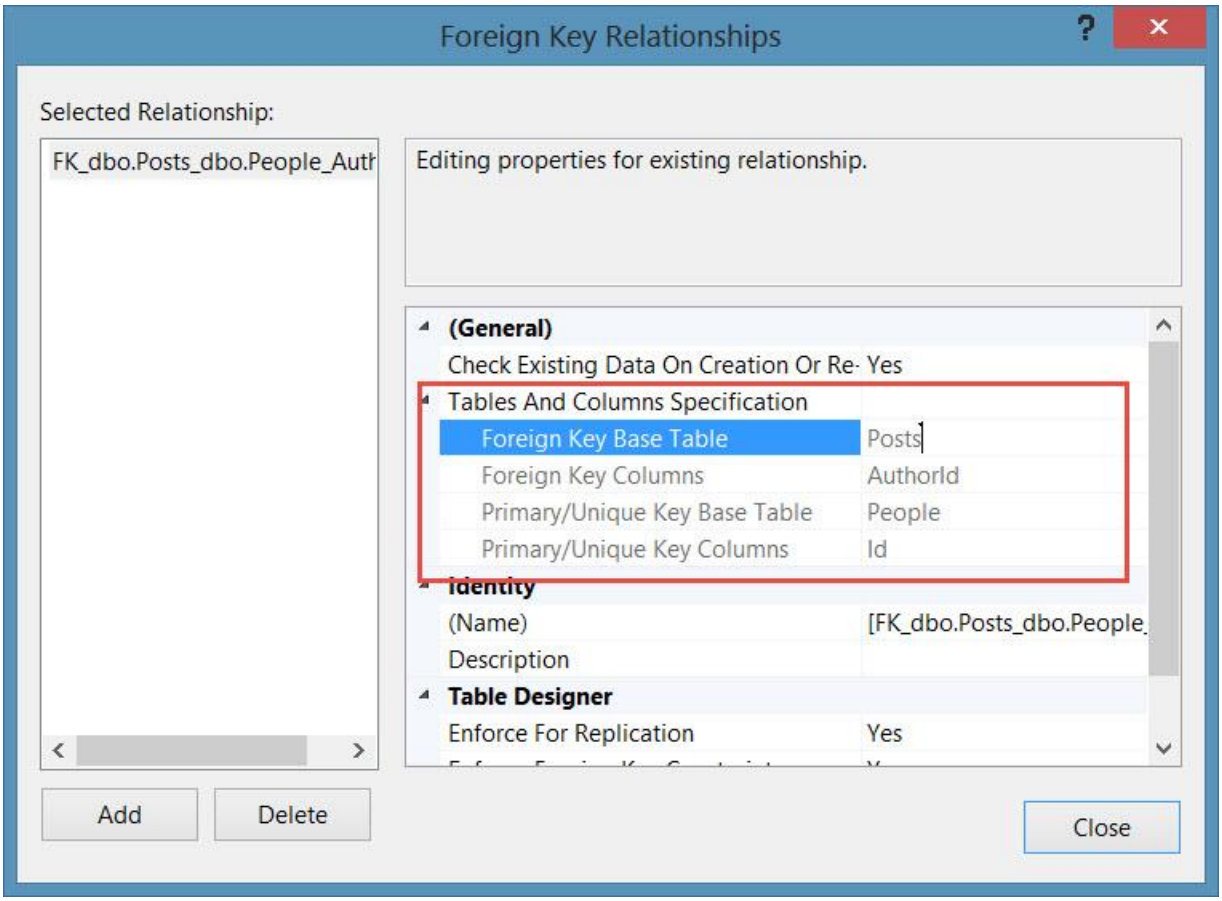
حال ببینیم چه اتفاقی در ساختار پایگاه داده رخ داده است.

قبل از هر چیز یک جدول دیگر به نام Posts در پایگاه داده ایجاد شده است:



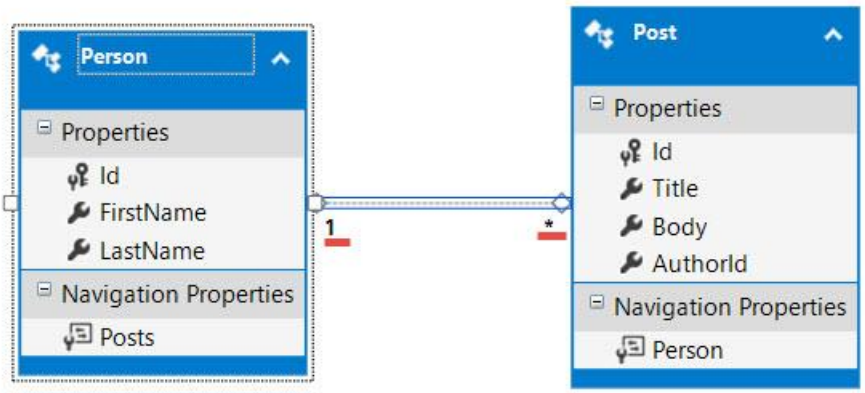
شکل ۳- ۱۷

همانطور که مشاهده می کنید جدول Posts با چهار ستون Id، Title،Body و AuthorId ایجاد شده است. به ستون AuthorId دقت کنید. همانظور که مشخص شده است این ستون به عنوان کلید خارجی که به جدول People اشاره خواهد کرد تعریف شده است که در شکل زیر نشان داده شده است:



شکل ۳- ۱۸

همانطور که مشاهده می کنید ستون Id از جدول People با ستون AuthorId از جدول Posts دارای ارتباط می باشند. اگر یک Entity Data Model برای دو جدول People و Posts ایجاد کنید شکلی شبیه به زیر را مشاهده خواهید کرد:



شکل ۳- ۱۹

همانطور که مشاهده می کنید ارتباط یک به چند به خوبی در شکل فوق مشخص شده است.

ارتباط چند به چند

سومین نوع ارتباطی که بین دو موجودیت در پایگاه داده می تواند وجود داشته باشد ارتباط چند به چند است. در این نوع ارتباط صفر،یک یا چند رکورد از موجودیت اول با صفر،یک یا چند رکورد از موجودیت دوم می تواند در ارتباط باشد.به عنوان مثال ارتباط بین موجودیت استاد و درس را در نظر بگیرید. هر درس می تواند توسط چند استاد ارائه شود و هر استاد می تواند چندین درس را ارائه دهد.یعنی صفر،یک یا چند رکورد از موجودیت درس می تواند با صفر،یک یا چند رکورد از موجودیت استاد در ارتباط باشد. به علت افزونگی داده ای که در ارتباطات چند به چند رخ خواهد داد این نوع ارتباطات در طراحی پایگاه داده به دو ارتباط یک به چند تبدیل می شود. در این حالت نیاز به جدول واسطی پیدا خواهیم کرد.

برای درک این موضوع دو کلاس به نام های Course و Master به صورت زیر ایجاد کنید:

```
public class Course
{
    public Int32 Id
    {
        get;
        set;
    }
    public String Title
```

```
        {
            get;
            set;
        }
        public virtual ICollection<Master> Masters
        {
            get;
            set;
        }
    }



    public class Master
    {
        public Int32 Id
        {
            get;
            set;
        }
        public String Name
        {
            get;
            set;
        }
        public virtual ICollection<Course> Courses
        {
            get;
            set;
        }
    }
```

کلاس Course دارای خاصیت Id به عنوان کلید اصلی و Title به عنوان نام درس در نظر گرفته شده است. برای کلاس Master نیز دارای خاصیت Id به عنوان کلید اصلی و Name به عنوان نام استاد در نظر گرفته شده است.

در هردو کلاس خاصیتی مجموعه ای به نام های Courses و Masters تعریف شده است. این بدان معنی است که هر نمونه از کلاس Course می تواند دارای چند نمونه از کلاس Master باشد و بلعکس.

حال کلاس PersonContext را باز کنید و دستورات زیر را بدان اضافه کنید:

```
    public DbSet<Course> Courses
     {
         get;
         set;
     }

     public DbSet<Master> Masters
```

```
    {
        get;
        set;
    }
```

حال برای پیکر بندی ارتباط بین دو موجودیت Course و Master یک کلاس به نام CourseConfig به صورت زیر ایجاد کنید:

```
public class CourseConfig : EntityTypeConfiguration<Course>
{

    public CourseConfig()
    {
        HasMany(l => l.Masters).WithMany(r => r.Courses).Map(m =>
        {
            m.MapLeftKey("CourseId");
            m.MapRightKey("MasterId");
        });
    }
}
```

کلاس CourseConfig با ارث برای از کلاس EntityTypeConfiguration با نوع Course قرار است تا ارتباط بین موجودیت های Course و Master را پیکر بندی کند.عملکرد متدهای HasMany و WithMany از نامشان کاملا مشخص است. متد HasMany بیانگر این است که هر رکورد از Course می تواند با تعدای رکورد از Master در ارتباط باشد. متد WithMany بیانگر این است که هر رکورد از Master نیز می تواند با تعدای رکورد از Course در ارتباط باشد.

همانطور که گفته شد برای پیاده سازی یک ارتباط چند به چند نیاز به یک جدول واسط داریم. این جدول را که Junction Table می نامند عموما دارای دو ستون بیشتر نمی باشد(البته این بدان معنی نیست که این جدول نمی تواند ستون های بیشتری داشته باشد بلکه در برخی از مواقع نیاز است تا ستون های بیشتری در این جدول واسط به کار رود) که عبارتند از دو کلید خارجی که یکی به جدول Course و دیگری به جدول Master اشاره می کند. عملکرد متد Map تعریف همین ساختار می باشد. کلید CourseId به موجودیت Course و کلید MasterId به موجودیت Master اشاره خواهد کرد.

کلاس PersonContext را باز کنید و دستور زیر را به متد OnModelCreating اضافه کنید:

```
modelBuilder.Configurations.Add(new CourseConfig());
```

این دستور تنظیمات پیکربندی مربوط به Course و Master را به مجموعه پیکربندی های EF اضافه می کند.

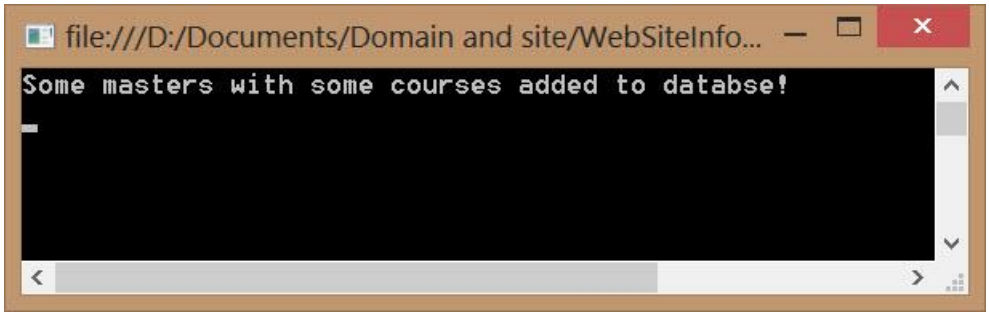
حال متد Main از کلاس Program را به صورت زیر تغییر دهید:

```
        static void Main(string[] args)
        {
            using (var context = new PersonContext())
            {
                Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<PersonContext>());
                context.Database.CreateIfNotExists();


                try
                {

                    Course c1 = new Course
                    {
                        Title = "course1"
                    };
                    Course c2 = new Course
                    {
                        Title = "course2"
                    };
                    Course c3 = new Course
                    {
                        Title = "course3"
                    };

                    Master m1 = new Master
                    {
                        Name = "master1"
                    };
                    Master m2 = new Master
                    {
                        Name = "master2"
                    };

                    m1.Courses.Add(c1);
                    m1.Courses.Add(c2);
                    m2.Courses.Add(c1);
                    m2.Courses.Add(c2);
                    m2.Courses.Add(c3);
                    context.Masters.Add(m1);
                    context.Masters.Add(m2);
                    context.SaveChanges();
                    Console.WriteLine("Some masters with some courses added to
databse!");
                }
                catch (DbEntityValidationException ex)
                {

                    String errors = "Some error occured :";
                    errors += Environment.NewLine;
                    foreach (var eve in ex.EntityValidationErrors)
                    {
```

```
                    foreach (var ve in eve.ValidationErrors)
                    {

                        errors += ve.ErrorMessage;
                        errors += Environment.NewLine;
                    }
                }
                Console.WriteLine(errors);
            }
        }
        Console.ReadKey();

    }
```

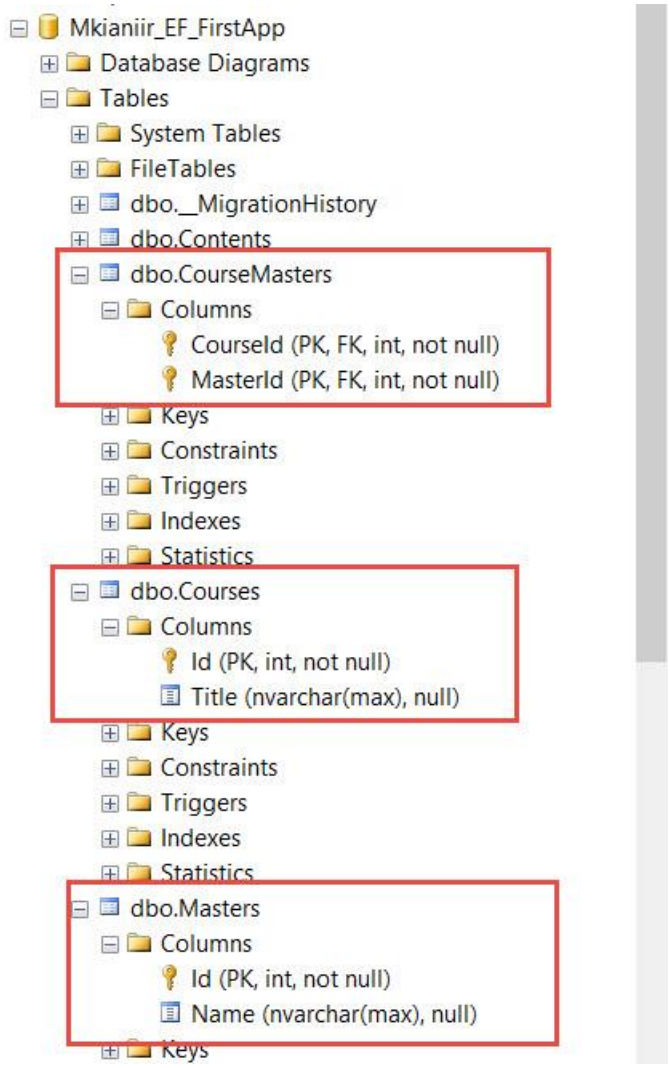
همانطور که مشاهده می کنید سه نمونه از کلاس Course با عناوین course۱، course۲ و course۳ ایجاد شده است. همچنین دو نمونه از کلاس Master با نام های master۱ و master۲ ایجاد شده است. دروس course۱ و course۲ به master۱ و دروس course۱، course۲ و course۳ به master۲ نسبت داده شده اند.

حال برنامه را اجرا کنید. اگر طبق موارد فوق پیش رفته باشید می بایست برنامه بدون خطا اجرا شده و شکل زیر در خروجی نشان داده شود:



شکل ۳- ۲۰

حال به پایگاه داده رفته و تغییرات پایگاه داده را مشاهده خواهیم کرد.



شکل ۳- ۲۱

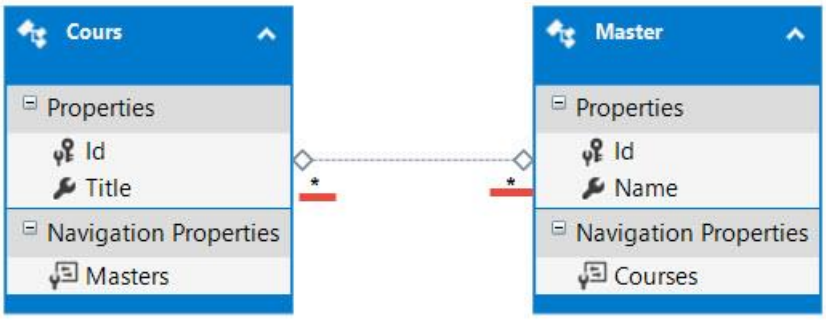
همانطور که مشاهده می کنید سه جدول Courses، Masters و CourseMasters در پایگاه داده اضافه شده است. جدول Courses دارای ستون های Id به عنوان کلید اصلی و Title به عنوان نام درس و جدول Masters دارای ستون های Id به عنوان کلید اصلی و Name به عنوان نام استاد ایجاد شده اند.

اگر به جدول CourseMasters دقت کنید خواهید دید که صرفا دارای دو ستون CourseId و MasterId می باشد. ستون CourseId به عنوان کلید اصلی همین جدول و نیز کلید خارجی به جدول Courses می باشد. همچنین ستون MastrId به عنوان کلید اصلی همین جدول و نیز کلید خارجی به جدول Masters می باشد.

دقت کنید که هر دو ستون CourseId و MasterId هردو با هم کلید های اصلی جدول CourseMasters را تشکیل می دهند. این بدان معناست که امکان درج رکورد هایی که هم CourseId آن و هم MasterId آن

تکراری باشند امکان پذیر نیست. اما می تواند چند رکورد داشت که تنها CourseId آن تکراری و یا فقط MasterId آن ها تکراری باشد یا اینکه هیچ کدام تکراری نباشند.

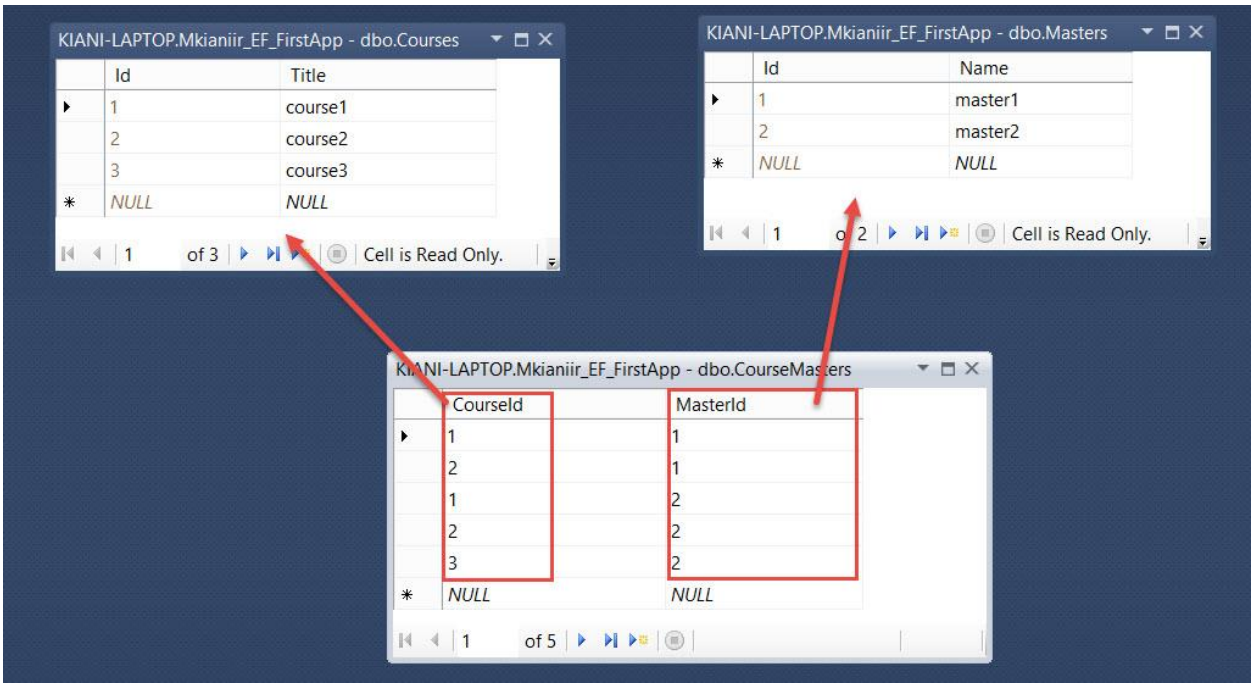
اگر با استفاده از Entity Data Model دو جدول Courses و Master را را نمایش دهیم به شکلی شبیه به شکل زیر خواهیم رسید.



شکل ۳- ۲۲

همانطور که مشاهده می کنید این دو جدول در ارتباط چند به چند قرار دارند.

در شکل زیر داده های درج شده درون این جداول نشان داده شده است:



شکل ۳- ۲۳

همانطور که مشاهده می کنید جدول CourseMasters به عنوان یک واسط بین دو جدول Coursesو Masters عمل می کند. به عنوان مثال اولین رکورد از این جدول بیانگر این است که درسی به آی دی ۱ با استادی ای دی ۱ در ارتباط است.

خلاصه

در این فصل به مفاهیم عمیق تری در رابطه با Entity Framework پرداختیم. مشاهده کردید که کتابخانه Entity Framework قابلیت های متعددی را برای کنترل کامل برروی ایجاد پایگاه داده و موجودیت های آن در اختیار شما قرار می دهد. توانستید پیکربندی های مختلفی را هم توسط صفات و هم توسط کتابخانه Fluent API برای موجودیت ها و اجزای آن ها انجام دهید. علاوه بر این در این فصل انواع ارتباطات بین جداول پایگاه داده و نحوه پیاده سازی آن ها در Entity Framework شرح داده شدند. در فصل چهارم که فصل پایانی کتاب نیز می باشد سایر مفاهیم مرتبط با Entity Frmaework Code-First و امکانات جدید آن بویژه Database Migration (Entity Framework Migration) بحث خواهد شد.

فصل چهارم : سایر مفاهیم Entity Framework Code-First

مقدمه

در این فصل که فصل پایانی کتاب نیز می باشد سایر مفاهیم مرتبط به Entity Framework Code-First مورد بررسی قرار خواهند گرفت. در این فصل بارگزاری به روش های Lazy و Eager ، پیاده سازی انواع داده ای پیچیده، پیاده سازی انواع شمارشی،کار با Vewi ها و پروسیجر ها و نهایتا استفاده از Database Migration برای اعمال تغییرات و بروزرسانی پایگاه داده بدون از بین رفتن داده های موجود مورد بحث و بررسی قرار خواهند گرفت.

بارگزاری به روش Lazy و Eager

در بخش قبلی انواع ارتباط بین موجودیت ها و نحوه پیاده سازی آن ها بیان شد. به عنوان مثال بین موجودیت Person و Post یک ارتباط یک به چند تعریف شد. در آن جا یک خاصیت به صورت زیر در کلاس Person تعریف شد:

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

خاصیت فوق مجموعه ای از Post های ایجاد شده توسط Person را نشان می دهد. حال سوال این است که زمانی که به شی Persons از کلاس PersonContext دسترسی پیدا می کنیم آیا EF می بایستی Post های مرتبط با آن Person را نیز بارگزاری کند یا خیر؟

در اینجاست که مبحثی به نام Lazy Loading یا Eager Loading در EF مطرح می شود. توجه داشته باشید که Lazyو Eager صرفا ویژه ارتباط های یک به چند نیستند. در واقع در هرگونه ارتباطی که بین موجودیت ها وجود داشته باشد بحث Lazy Loading و Eager Loading نیز وجود خواهد داشت.

چه زمانی از Lazy و چه زمانی از Eager استفاده کنیم؟

زمانی که بخواهیم رکورد های مرتبط با یک رکورد به صورت همزمان از پایگاه داده خوانده شوند از Eager استفاده خواهیم کرد. به عنوان مثال فرض کنید زمانی که یک رکورد از Person از پایگاه داده خوانده می شود بخواهیم رکورد های مرتبط با آن، یعنی رکورد های جدول Posts نیز از پایگا داده واکشی شوند می بایستی از حالت Eager Loading استفاده کنیم. در غیر این صورت از حالت Lazy Loading استفاده خواهیم کرد. در حالت Lazy Loading در زمان واکشی یک رکورد از پایگاه داده(مثلا رکوردی از موجو دیت Person) رکورد های وابسته آن(مثلا رکورد های جدول Posts که مربوط به رکورد Person واکشی شده است) واکشی نخواهند شد. EF در حالت Lazy Loading واکشی رکورد های وابسته را به زمانی که کاربر واقعا بخواهد از آن ها استفاده کند موکول خواهد کرد. در حالت پیش فرض EF از حالت Lazy Loading استفاده می کند که شما می تواند این حالت پیش فرض را تغییر دهید. برای غیر فعال کردن حالت Lazy Loading از دستور زیر استفاده کنید:

```
context.Configuration.LazyLoadingEnabled = false;
```

شی context در مثال ما نمونه ای است از کلاس PersonContext.

دستور فوق در متد Main پس از ایجاد context نوشته شده است.

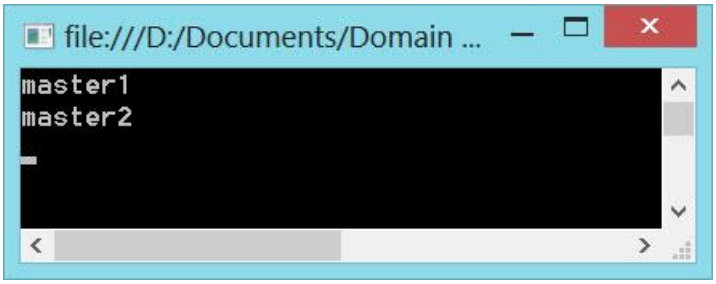
اینکه از کدام روش (Lazy یا Eager) استفاده کنید کاملا بستگی به داده ها و سناریوی برنامه شما دارد. اما اگر مطمئن نیستید که از داده های مرتبط با یک داده در آن واحد نیاز دارید از Lazy استفاده نمائید. استفاده از حالت Eager در زمانی که ارتباطات تو درتو و زیاد باشند می توانند منجر به ایجاد کوئری هایی شوند که ممکن است تاثیر منفی در کارایی برنامه داشته باشد. اما این بدان معنا نیست که Lazy Loading همواره کارایی بهتری نسبت به Eager Loading خواهد داشت.

نمونه ای از حالت بارگزاری Lazy

دستور زیر را در متد Main از کلاس Program نوشته و آن را اجرا کنید:

```
var masters = context.Masters;
foreach (var master in masters.ToList())
        {

                Console.WriteLine(master.Name);
        }
```

دستور فوق لیست اساتید را از پایگاه داده خوانده و در کنسول نمایش می دهد:



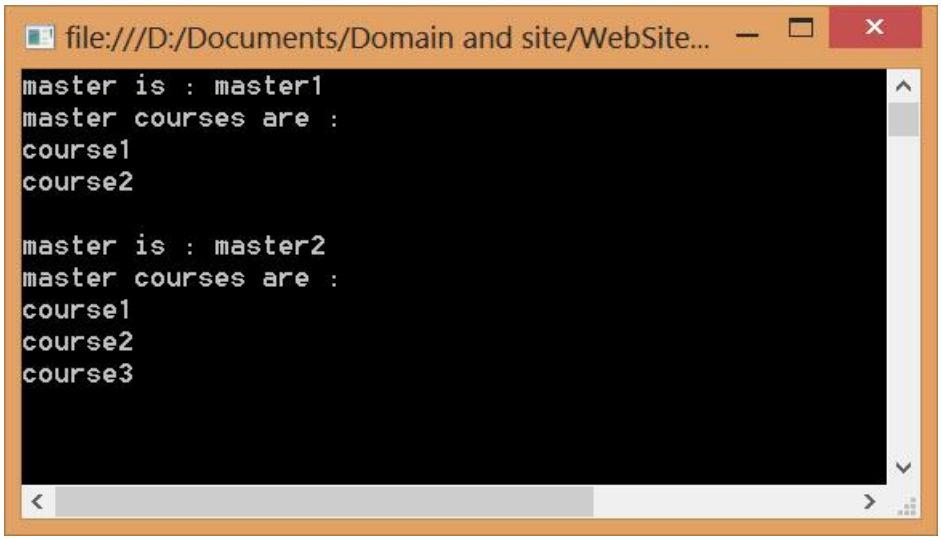
شکل ٤ - ١

حال دستورات متد Main را به صورت زیر تغییر دهید:

```
var masters = context.Masters;
                    foreach (var master in masters.ToList())
                    {
                        String result = "master is : " + master.Name+
Environment.NewLine;
                        result += "master courses are : " + Environment.NewLine;
                        foreach (var course in master.Courses)
                        {
                            result += course.Title + Environment.NewLine;
                        }
                        Console.WriteLine(result);
                    }
```

همانطور که مشاهده می کنید دو دستور foreach به صورت تو در تو قرار گرفته است. دستور foreach بیرونی اطلاعات مربوط به Master را واکشی کرده و دستور foreach داخلی برای هر رکورد Master اطلاعات مربوط به Course را واکشی می کند.

خروجی دستورات فوق به صورت زیر خواهد بود:



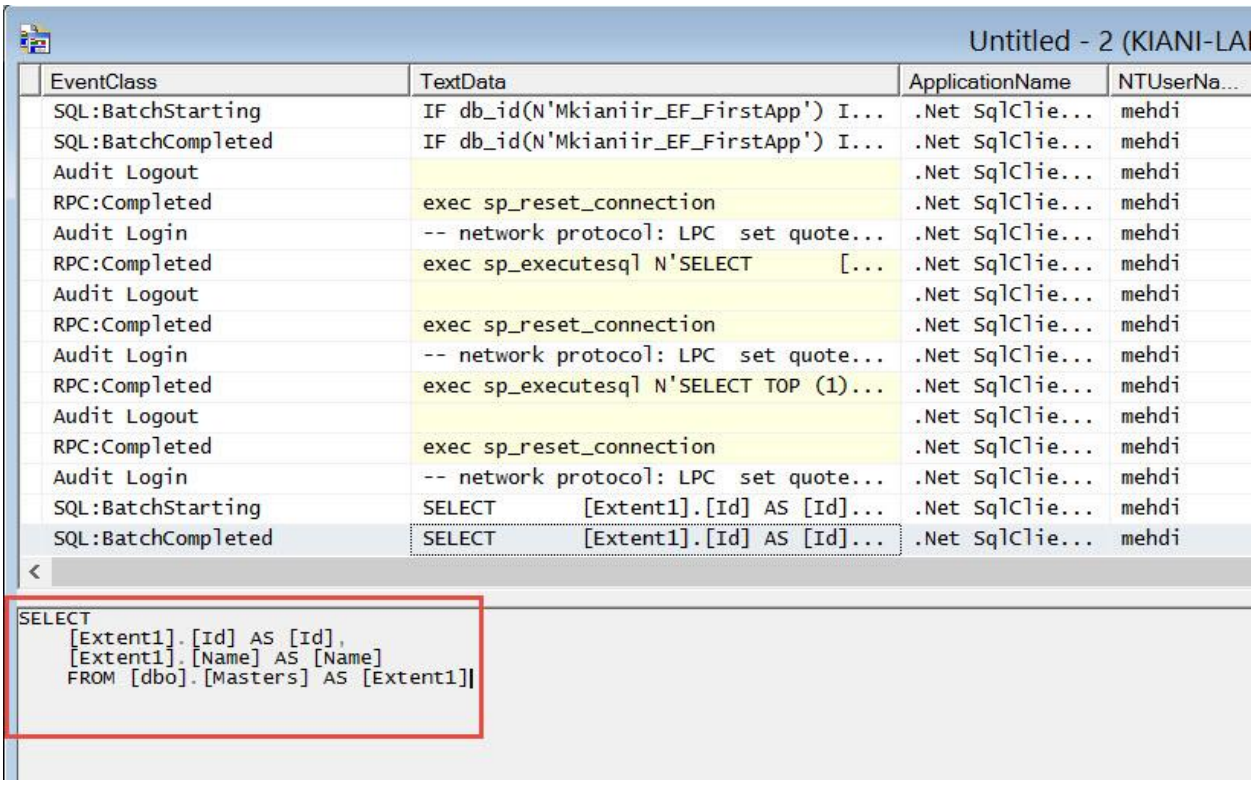
شکل ٤ - ٢

همانطور که مشاهده می کنید Course های مربوط به هر Master نیز پس از دسترسی به خاصیت Courses از هر Master بارگزاری و درخروجی نشان داده شده است.

اگر کوئری های ایجاد شده در دستورات فوق را با Sql Profiler بررسی کنید متوجه خواهید شد که در گام اول یعنی foreach بیرونی کوئری زیر به پایگاه داده فرستاده خواهد شد:

```
SELECT
    [Extent1].[Id] AS [Id],
    [Extent1].[Name] AS [Name]
    FROM [dbo].[Masters] AS [Extent1]
```

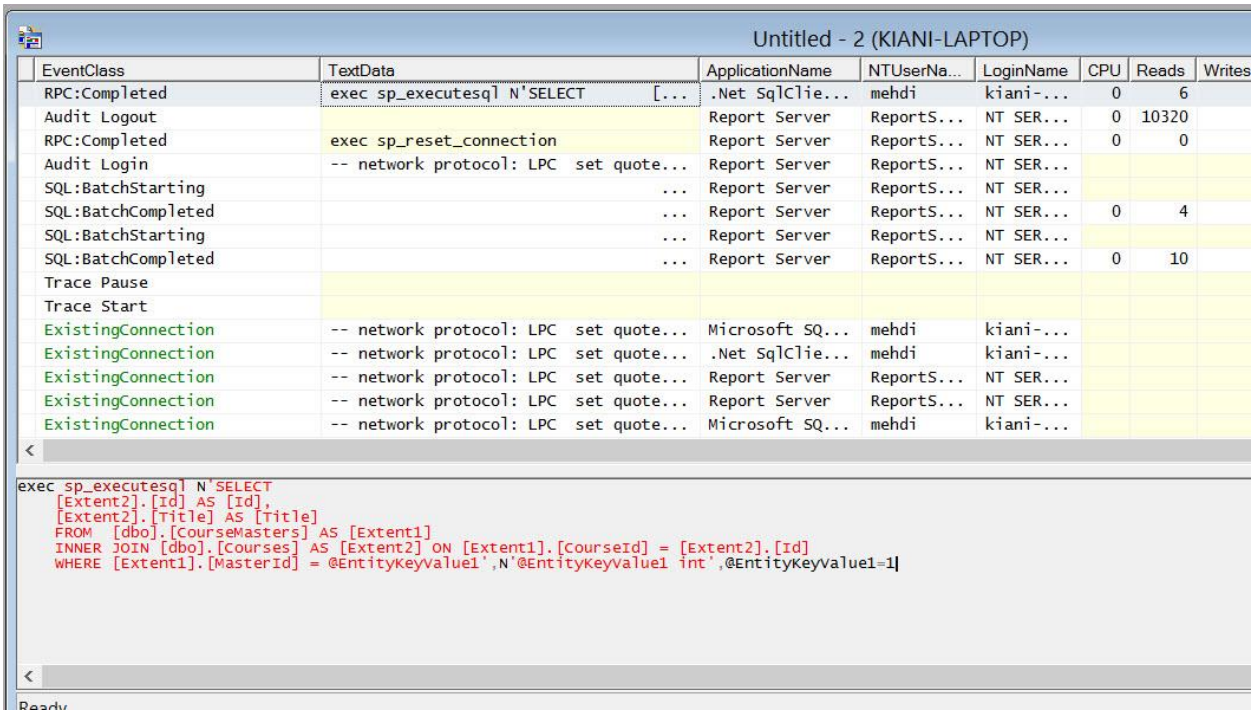
این موضوع را در شکل زیر مشاهده می کنید:



شکل ٤ - ٣

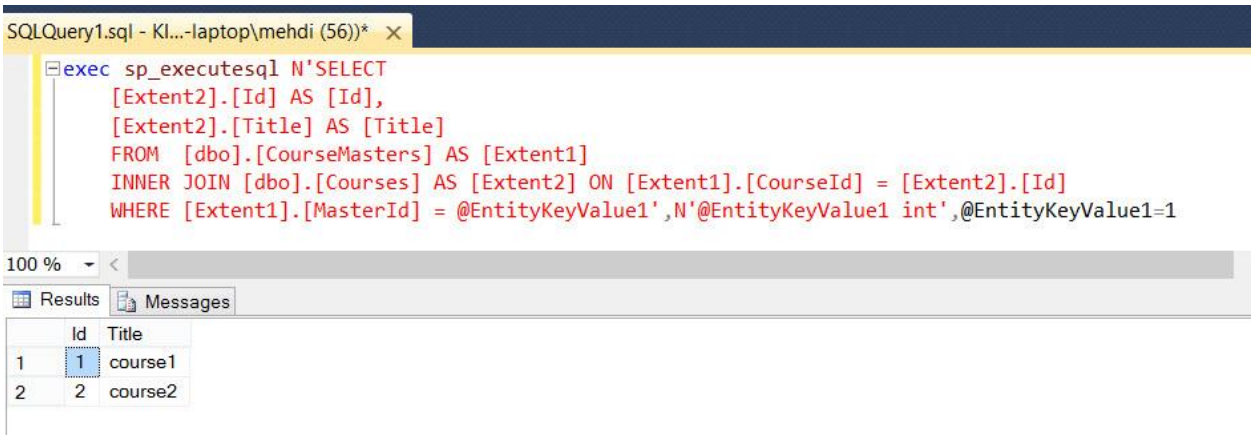
پس از گذر از مرحله اول و رسید به مرحله دوم یعنی foreach داخلی و جایی که به خاصیت Courses دسترسی پیدا کرده ایم کوئری ارسال شده به پایگاه داده به صورت زیر خواهد بود:

```
exec sp_executesql N'SELECT
    [Extent2].[Id] AS [Id],
    [Extent2].[Title] AS [Title]
    FROM  [dbo].[CourseMasters] AS [Extent1]
    INNER JOIN [dbo].[Courses] AS [Extent2] ON [Extent1].[CourseId] = [Extent2].[Id]
    WHERE [Extent1].[MasterId] = @EntityKeyValue1',N'@EntityKeyValue1
int',@EntityKeyValue1=1
```



شکل ٤ - ٤

اگر دستور فوق را در SSMS اجرا کنید خروجی زیر را مشاهده خواهید کرد:



شکل ٤ - ٥

مقدار EntityKeyValue همانظور که مشاهده می کنید برابر با ۱ می باشد و این همان آی دی مربوط به استاد می باشد و دروس مربوط به استاد با شناسه ۱ واکشی شده است.

بنابر این در Lazy Loading واکشی اطلاعات تا زمانی که نیاز به آن ها نباشد به تاخیر خواهد افتاد.

نمونه ای از حالت بارگزاری Eager

در بخش قبلی با نحوه بارگزاری اطلاعات به روش Lazy اشنا شدیم. در این بخش روش Eager را بررسی خواهیم کرد. با استفاده از متد Include در دستورت قبلی می توانیم روش بارگزاری Lazy را به Eager تبدیل کنیم.

دستور

```
var masters = context.Masters;
```

در متد Main را به صورت زیر تغییر دهید:

```
var masters = context.Masters.Include(i=>i.Courses);
```

متد Include دارای یک پارامتر از جنس Property Expression که توسط عبارات لامبدا مقدار دهی شده است. در دستور فوق به EF گفته شده که همزمان با واکشی Master ها ، Course های مربوط به آن ها را نیز بارگزاری کن.

حال اگر برنامه را اجرا کنید خروجی شبیه به خروجی بخش قبل خواهید گرفت اما با این تفاوت که کوئری که در این حالت به Sql Server ارسال می شود با حالت قبل متفاوت است.

در این حالت کوئری زیر به Sql Server ارسال می شود:

```
SELECT
    [Project1].[Id] AS [Id],
    [Project1].[Name] AS [Name],
    [Project1].[C1] AS [C1],
    [Project1].[Id1] AS [Id1],
    [Project1].[Title] AS [Title]
    FROM ( SELECT
        [Extent1].[Id] AS [Id],
        [Extent1].[Name] AS [Name],
        [Join1].[Id] AS [Id1],
        [Join1].[Title] AS [Title],
        CASE WHEN ([Join1].[CourseId] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C1]
        FROM  [dbo].[Masters] AS [Extent1]
        LEFT OUTER JOIN  (SELECT [Extent2].[CourseId] AS [CourseId], [Extent2].[MasterId]
AS [MasterId], [Extent3].[Id] AS [Id], [Extent3].[Title] AS [Title]
            FROM  [dbo].[CourseMasters] AS [Extent2]
            INNER JOIN [dbo].[Courses] AS [Extent3] ON [Extent3].[Id] =
[Extent2].[CourseId] ) AS [Join1] ON [Extent1].[Id] = [Join1].[MasterId]
    )  AS [Project1]
    ORDER BY [Project1].[Id] ASC, [Project1].[C1] ASC
```

در شکل زیر و با استفاده از Sql Profiler این مورد نشان داده شده است:

Untitled - 1 (KIANI-LAPTOP)

EventClass	TextData	ApplicationName	NTUserNa...	LoginName	CPU	Reads	Writes	Duration	ClientProcessID
Audit Login	-- network protocol: LPC set quote...	.Net SqlClie...	mehdi	kiani-...					14812
RPC:Completed	exec sp_executesql N'SELECT TOP (1)...	.Net SqlClie...	mehdi	kiani-...	0	2	0	0	14812
Audit Logout		.Net SqlClie...	mehdi	kiani-...	0	40	0	220	14812
RPC:Completed	exec sp_reset_connection	.Net SqlClie...	mehdi	kiani-...	0	0	0	0	14812
Audit Login	-- network protocol: LPC set quote...	.Net SqlClie...	mehdi	kiani-...					14812
SQL:BatchStarting	SELECT [Project1].[Id] AS [Id...	.Net SqlClie...	mehdi	kiani-...					14812
SQL:BatchCompleted	SELECT [Project1].[Id] AS [Id...	.Net SqlClie...	mehdi	kiani-...	0	17	0	0	14812
Trace Pause									

```
SELECT
    [Project1].[Id] AS [Id],
    [Project1].[Name] AS [Name],
    [Project1].[C1] AS [C1],
    [Project1].[Id1] AS [Id1],
    [Project1].[Title] AS [Title]
    FROM ( SELECT
        [Extent1].[Id] AS [Id],
        [Extent1].[Name] AS [Name],
        [Join1].[Id] AS [Id1],
        [Join1].[Title] AS [Title],
        CASE WHEN ([Join1].[CourseId] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C1]
        FROM  [dbo].[Masters] AS [Extent1]
        LEFT OUTER JOIN  (SELECT [Extent2].[CourseId] AS [CourseId], [Extent2].[MasterId] AS [MasterId], [Extent3].[Id] AS [Id], [Extent3].[Title] AS [Title]
            FROM  [dbo].[CourseMasters] AS [Extent2]
            INNER JOIN [dbo].[Courses] AS [Extent3] ON [Extent3].[Id] = [Extent2].[CourseId] ) AS [Join1] ON [Extent1].[Id] = [Join1].[MasterId]
    )  AS [Project1]
    ORDER BY [Project1].[Id] ASC, [Project1].[C1] ASC
```

شکل ٤ - ٦

اگر کوئری فوی را در SSMS اجرا کنید خروجی شبیه به خروجی زیر به شما خواهد داد:

SQLQuery1.sql - KI...-laptop\mehdi (58))*

```
SELECT
    [Project1].[Id] AS [Id],
    [Project1].[Name] AS [Name],
    [Project1].[C1] AS [C1],
    [Project1].[Id1] AS [Id1],
    [Project1].[Title] AS [Title]
    FROM ( SELECT
        [Extent1].[Id] AS [Id],
        [Extent1].[Name] AS [Name],
        [Join1].[Id] AS [Id1],
        [Join1].[Title] AS [Title],
        CASE WHEN ([Join1].[CourseId] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C1]
        FROM  [dbo].[Masters] AS [Extent1]
        LEFT OUTER JOIN  (SELECT [Extent2].[CourseId] AS [CourseId], [Extent2].[MasterId] AS [MasterId],
            FROM  [dbo].[CourseMasters] AS [Extent2]
            INNER JOIN [dbo].[Courses] AS [Extent3] ON [Extent3].[Id] = [Extent2].[CourseId] ) AS [Join1
    )  AS [Project1]
    ORDER BY [Project1].[Id] ASC, [Project1].[C1] ASC
```

100 %

Results | Messages

	Id	Name	C1	Id1	Title
1	1	master1	1	1	course1
2	1	master1	1	2	course2
3	2	master2	1	1	course1
4	2	master2	1	2	course2
5	2	master2	1	3	course3

شکل ٤ - ٧

همانطور که مشاهده می کنید در یک کوئری رکورد های مربوط به master و course به صورت همزمان واکشی شده اند.

انواع پیچیده

یک نوع داده پیچیده که در کتب مرجع به آن Complex Type می گویند در EF به نوعی گفته می شود که به زیر مجموعه ای از ستون های یک جدول نگاشت می شود. نوع های پیچیده همان کلاس های دات نت و بسیار شبیه به Entity ها می باشند با این تفاوت که نوع های پیچیده چون در سایر Entity ها مورد استفاده می گیرند دارای فیلد کلید اصلی نمی باشند.

انواع پیچیده زمانی که بخواهید فیلد های مشترکی بین چند Entity را شبیه سازی کنید مفید هستند. به عنوان مثال نوع Address می تواند یک نوع داده پیچیده باشد. نوع Address می تواند برای Entity های مختلفی نظیر Person ، Company و ... به کار رود.

کلاس Address را به صورت زیر تعریف کنید:

```
public class Address
 {

     public String Street
     {
         get;
         set;
     }
     public String Blvd
     {
         get;
         set;
     }
     public String City
     {
         get;
         set;
     }
     public String State
     {
         get;
         set;
     }
     public String PostalCode
     {
         get;
         set;
     }
 }
```

حال از کلاس فوق می توانید به عنوان یک زیر نوع در موجودیت هایی نظیر Company و Person و یا هر موجودیت دیگری که فکر می کنید نیاز به زیر نوع آدرس داشته باشد استفاده نمائید.کلاس Person را باز کرده و دستورات آن را به صورت زیر تغییر دهید:

```
public partial class Person
{
    public Person()
    {
        Posts = new List<Post>();
        Address = new Address();
    }
    public int Id
    {
        get;
        set;
    }
    public string FirstName
    {
        get;
        set;
    }
    public string LastName
    {
        get;
        set;
    }

    public virtual PersonInfo Info
    {
        get;
        set;
    }

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

همانطور که مشاهده می کنید یک خاصیت از نوع کلاس Address تعریف شده است. دقت کنید که در Constructor کلاس Person خاصیت Address مقدار دهی اولیه شده است. این مقدار دهی اولیه باعث می

شود تا از استثنای NullReference در برنامه جلوگیری شود. اگر این مقدار دهی اولیه صورت نپذیرید آنگاه دستوراتی شبیه به دستورات زیر باعث ایجاد استثنای مذکور خواهند شد.

شکل ٤ - ٨

پیکربندی انواع پیچیده

پیکربندی انواع پیچیده دقیقا همانند پیکربندی Entity های می باشد با این تفاوت که به جای ارث بری از کلاس EntityTypeConfiguration می بایستی از کلاس ComplexTypeConfugration ارث بری کنید.

یک کلاس به نام AddressConfig ایجاد کنید و دستورات آن را مانند زیر تغییر دهید:

```
public class AddressConfig : ComplexTypeConfiguration<Address>
{
    public AddressConfig()
    {
        Property(p => p.Street).HasMaxLength(40).IsRequired();
    }
}
```

همانطور که مشاهده می کنید دستورات پیکربندی برای انواع پیچیده تفاوتی با موجودیت های اصلی ندارند. حال می بایستی فایل پیکربندی فوق را به مجموعه پیکربندی های پروژه اضافه کنید. برای این منظور متد OnModelCreating مربوط به کلاس PersonContext را به صورت زیر تغییر دهید:

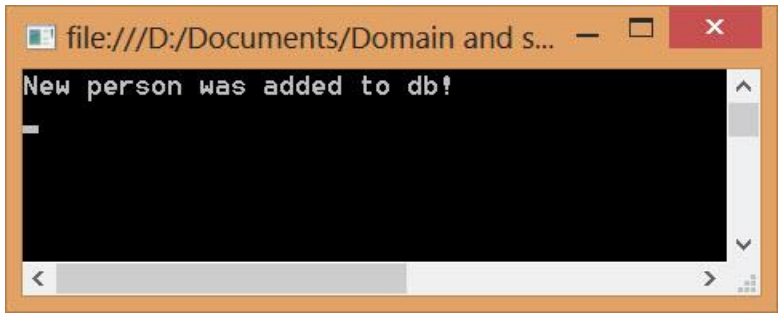
```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);
    modelBuilder.Configurations.Add(new PersonConfig());
    modelBuilder.Configurations.Add(new CourseConfig());
    modelBuilder.Configurations.Add(new AddressConfig());

}
```

حال دستورات متد Main کلاس Program را به صورت زیر تغییر دهید تا در هنگام اجرای برنامه یک رکورد برای Person در جدول People درج شود:

```
static void Main(string[] args)
{
    using (var context = new PersonContext())
    {
        Database.SetInitializer(new
DropCreateDatabaseIfModelChanges<PersonContext>());
        context.Database.CreateIfNotExists();
        //context.Configuration.LazyLoadingEnabled = false;

        try
        {
            Person p1 = new Person
            {
                FirstName = "Mehdi",
                LastName = "Kiani"
            };
            p1.Address.State = "Isfahan";
            p1.Address.City = "Isfahan";
            p1.Address.Street = "Khajoo";
            context.Persons.Add(p1);
            context.SaveChanges();
            Console.WriteLine("New person was added to db!");
        }
        catch (DbEntityValidationException ex)
        {

            String errors = "Some error occured :";
            errors += Environment.NewLine;
            foreach (var eve in ex.EntityValidationErrors)
            {

                foreach (var ve in eve.ValidationErrors)
```

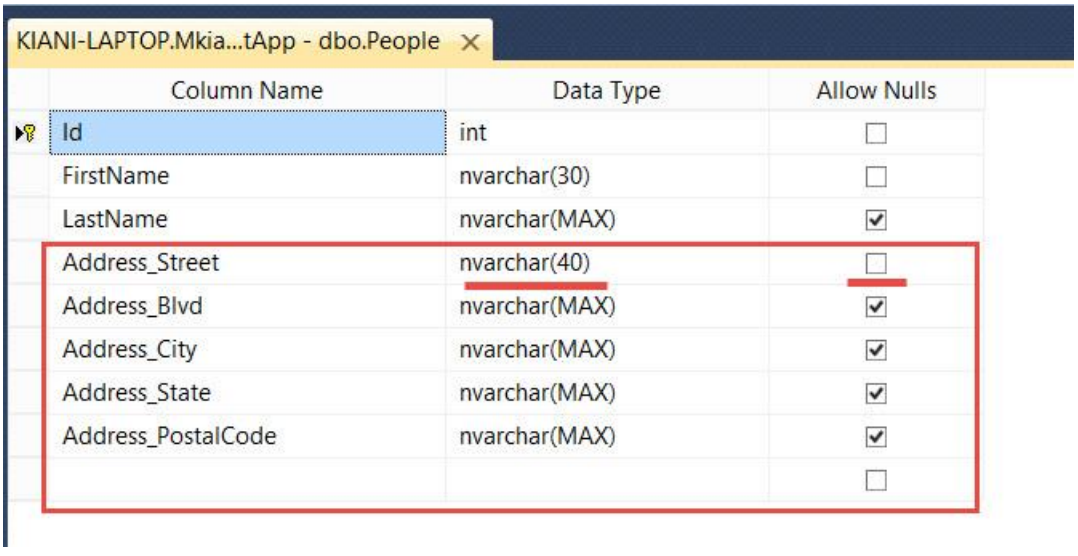
```
                {

                    errors += ve.ErrorMessage;
                    errors += Environment.NewLine;
                }
            }
            Console.WriteLine(errors);
        }
    }
    Console.ReadKey();

}
```

حال اگر برنامه را اجرا کنید وخطایی در دستورات نداشته باشید می بایستی خروجی شبیه به زیر بدست آید:



شکل ٤ - ٩

حال جدول People را باز کنید و تغییرات درون آن را مشاهده نمائید:



شکل ٤ - ١٠

همانطور که مشاهده می کنید فیلد های مربوط به نوع Address با پیشوند Address نامگذاری شده اند.اگرچه من به شخصه این نامگذاری را برای انواع پیچیده به خاطر تفکیکی که با سایر ستون های موجودیت ها ایجاد می کند ترجیح می دهم اما شما می تواند آن را تغییرداده و نام ستون ها را خودتان تعیین نمائید که در ادامه نحوه تغییر آن ایجاد می شود.

نکته دیگری که در شکل فوق قابل توجه است نوع ستون Address_Street می باشد که طبق فایل پیکربندی از نوع nvarchar(۴۰) و Not Null می باشد.

نامگذاری صریح جداول و ستون ها

همانطور که تاکنون دیده اید نام موجودیت Person توسط EF به نام People در پایگاه داده تغییر کرده است. این جمع بندی نام موجودیت ها برای جداول توسط EF تعیین می شود. البته شما الزام به پیروی از آن ها نمی باشد. به عنوان مثال ممکن است بخواهید برای موجودیت Person جدولی به نام Persons (به جای People) در پایگاه داده ایجاد شود. این عمل هم توسط صفات موجود در فضای نام System.ComponentModel.DataAnnotation امکان پذیر است هم می توانید با استفاده از دستورات مربوط به کتابخانه Fluent این کار را انجام دهید. این تعیین نام علاوه بر جداول برای ستون های جداول نیز امکان پذیر است. در این بخش با استفاده از دستورات Fluent این تغییرات را اعمال می کنیم.

کلاس PersonConfig را باز کنید و دستورات ان را طبق زیر تغییر دهید:

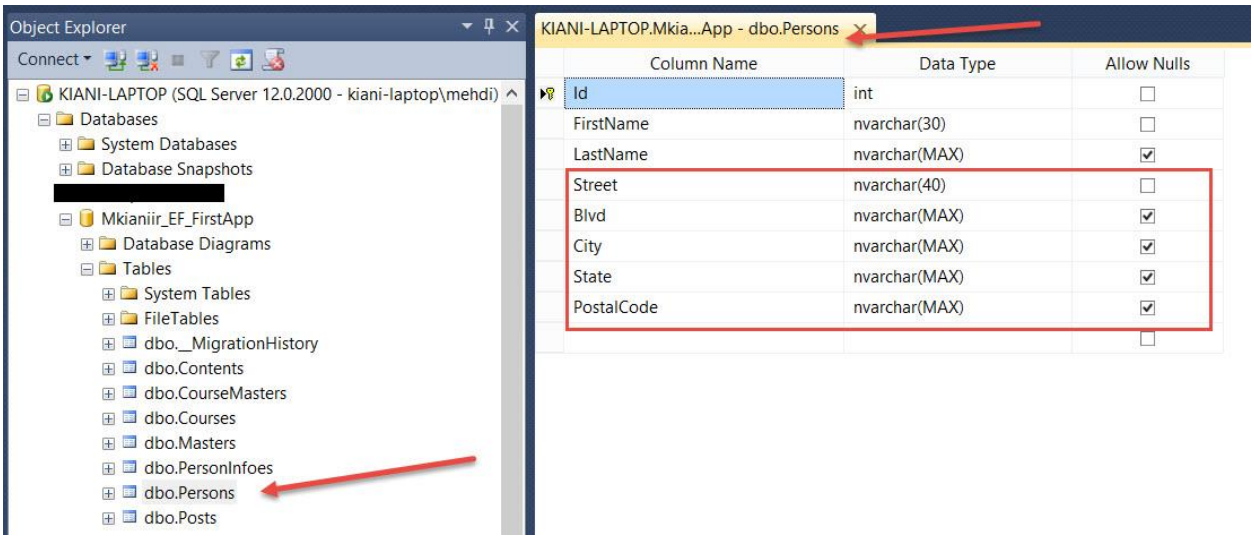
```
    public class PersonConfig : EntityTypeConfiguration<Person>
    {
        public PersonConfig()
        {
            ToTable("Persons");
            Property(p => p.FirstName).HasMaxLength(30).IsRequired();
            HasOptional(o => o.Info).WithRequired(r => r.Person);
            HasMany(m => m.Posts).WithRequired(r => r.Author).HasForeignKey(f =>
f.AuthorId);
        }
    }
```

با استفاده از متد ToTable می توانید نام جدول را تعیین کنید.

حال کلاس AddressConfig را باز کنید و طبق زیر تغییرات را اعمال کنید:

```
public class AddressConfig : ComplexTypeConfiguration<Address>
{
    public AddressConfig()
    {
        Property(p => p.Street).
            HasMaxLength(40).
            IsRequired().HasColumnName("Street");
        Property(p => p.Blvd).HasColumnName("Blvd");
        Property(p => p.State).HasColumnName("State");
        Property(p => p.City).HasColumnName("City");
        Property(p => p.PostalCode).HasColumnName("PostalCode");
    }
}
```

توسط متد HasColumnName می توانید نام ستون دلخواهی را برای یک خاصیت در نظر بگیرید. حال اگر مجددا برنامه را اجرا کنید تغییرات را در پایگاه داده مشاهده خواهید کرد:



شکل ٤ - ١١

همانطور که در شکل مشخص است نام جدول People به Persons تغییر و پیشوند Address برای فیلد های مربوط به آن برداشته شده است.

چشم پوشی از خاصیت

گاهی مواقع نیاز داریم که برخی از خواص موجودیت ها صرفا در سمت کد نویسی قابل استفاده باشند و نخواهیم ستونی متناظر با آن خاصیت در پایگاه داده داشته باشیم.اینگونه خواص معمولا خواصی هستند یک سری عملیات منطقی در موجو دیت انجام می دهند و مقادیر آن ها بر اسا مقادیر سایر خواص تامین می شوند. در این حالت با استفاده از متد Ignore می توانیم آن خاصیت را از مجموعه ستون های پایگاه داده حذف کنیم.

به عنوان مثال خاصیت زیر را به کلاس Person اضافه کنید:

```
public String CodeOnlyProperty
        {
            get;
            set;
        }
```

حال کلاس PersonConfig را بازکنید و دستور Ignore را مطابق زیر به آن اضافه کنید:

```
    public class PersonConfig : EntityTypeConfiguration<Person>
    {
        public PersonConfig()
        {
            ToTable("Persons");
            Property(p => p.FirstName).HasMaxLength(30).IsRequired();
            HasOptional(o => o.Info).WithRequired(r => r.Person);
            HasMany(m => m.Posts).WithRequired(r => r.Author).HasForeignKey(f =>
f.AuthorId);
            Ignore(p => p.CodeOnlyProperty);
        }
    }
```

حال اگر برنامه را اجرا کنید فیلدی به نام CodeOnlyProperty در جدول Persons ایجاد نخواهد شد.

انواع داده شمارشی

همانطور که میدانید انواع داده شمارشی برای استفاده از نام به جای اعداد به جهت خوانایی بالاتر کد ها مورد استفاده قرار می گیرند. به عنوان مثال فرض کنید موجودیت Person دارای وضعیت های فعال، غیر فعال و نامشخص باشد.خوب می توانید برای این حالت یک فیلد عددی در نظر بگیرید. مثلا مقدار ۱ برای فعال، مقدار ۰ برای غیر فعال

و مقدار -۱ برای وضعیت نامشخص. با این روش برای ایجاد رکوردی با وضعیت فعال باید دستوری شبیه به دستور زیر بنویسیم:

```
Person p1 = new Person();
P1.PersonState=1;
```

همانطور که مشاهده می کنید دستورات فوق اگرچه قابلیت اجرایی دارند اما قابلیت خوانایی پایینی دارند. هر شخصی متوجه نخواهد شد که مقدار ۱ به چه معناست. چون معنای فعال بودن برای مقدار ۱ صرفا در ذهن شما به عنوان توسعه دهنده می باشد.

حال اگر یک نوع داده شمارشی به نام PersonState به صورت زیر داشته باشیم:

```
public enum PersonState
    {
        Active,
        Deactive,
        Unknown
    }
```

و سپس خاصیت PersonState را برای موجودیت Person به صورت زیر تعریف کنیم:

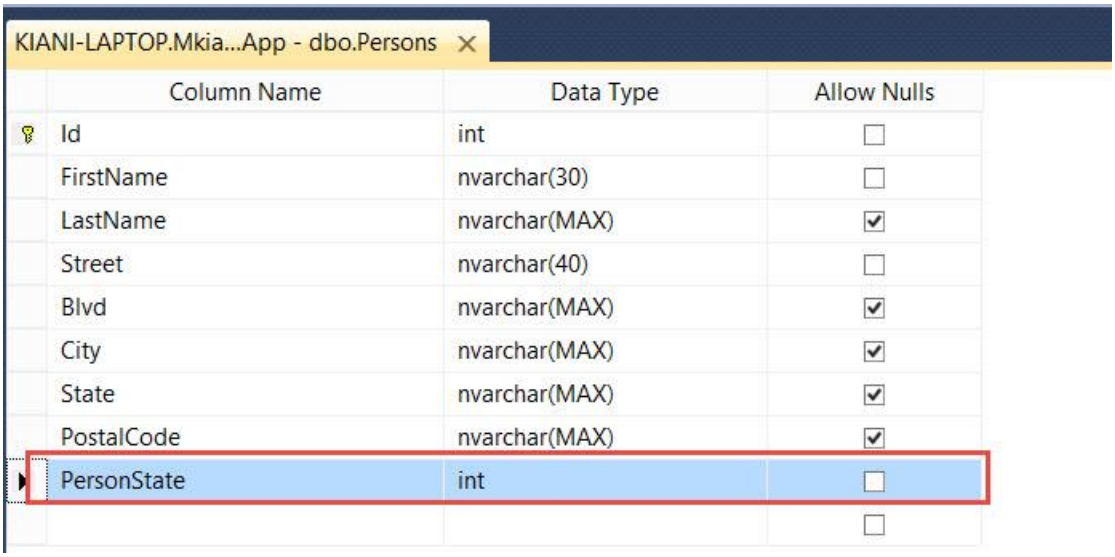
```
public PersonState PersonState
        {
            get;
            set;
        }
```

آنگاه برای مشخص کردن شخصی با وضعیت فعال باید دستوری شبیه به دستور زیر بنویسم:

```
Person p1 = new Person
       {
              FirstName = "Mehdi",
              LastName = "Kiani"
       };
       p1.PersonState = PersonState.Active;
```

پرواضح است که این روش به مراتب قابلیت درک و خوانایی بالاتری دارد.

حال اگر با توجه به تغییرات فوق برنامه را اجرا کنید و سپس نگاهی به جدول Persons در پایگاه داده بیندازید ستون PersonState را مشاهده خواهید کرد:



شکل ٤ - ١٢

همانطور که در شکل فوق نیز قابل مشاهده است ستون PersonState از نوع int تعریف شده است. EF به صورت خودکار مقادیر عددی را به نوع داده شمارشی و بلعکس تبدیل خواهد کرد. بنابر این در سمت کد نویسی شما با نوع داده شمارشی کار خواهید کرد و در سمت پایگاه داده مقادیر عددی متناظر با آن درج خواهد شد.همانطور که در فصل اول اشاره شد پشتیبانی از انواع شمارشی از نسخه ۵ آغاز شد.

یک موجودیت با چند جدول

فرض کنید بخواهیم عکس شخص را به همراه سایر اطلاعات اودر پایگاه داده ذخیره کنیم. در پایگاه داده معمولا برای این دسته از ستون ها نوع داده varbinary(max) و در سمت دات نت آرایه ای از بایت ها در نظر گرفته می شود.به این نوع داده ها عموما (BLOB) می گویند. واژه BLOB مخفف Binary Large Objects می باشد. این دسته از خواص معمولا حجم زیادی از فضا را اشغال می کنند. در این حالت بهتر است که این ستون ها در جدولی مجزا قرار گیرند. این حالت یکی از حالت هایی است که ارتباطات یک به یک در آن کاربرد دارد. خوشبختانه EF این امکان را به راحتی برای شما فراهم کرده است:

کلاس Person را باز کرده و دستورات آن را طبق زیر تغییر دهید:

```
public partial class Person
{
    public Person()
    {
```

```
            Posts = new List<Post>();
            Address = new Address();
        }
        public int Id
        {
            get;
            set;
        }
        public string FirstName
        {
            get;
            set;
        }
        public string LastName
        {
            get;
            set;
        }

        public virtual PersonInfo Info
        {
            get;
            set;
        }

        public virtual ICollection<Post> Posts
        {
            get;
            set;
        }
        public Address Address
        {
            get;
            set;
        }
        public String CodeOnlyProperty
        {
            get;
            set;
        }
        public PersonState PersonState
        {
            get;
            set;
        }
        public Byte[] Photo
        {
            get;
            set;
        }
    }
```

همانطور که مشاهده می کنید خاصیتی از جنس آرایه ای از بایت ها برای موجودیت Person ایجاد شده است.

حال اگر برنامه را اجرا کنید تغییرات درون جدول Person به صورت زیر خواهد بود:

KIANI-LAPTOP.Mkia...App - dbo.Persons

Column Name	Data Type	Allow Nulls
Id	int	☐
FirstName	nvarchar(30)	☐
LastName	nvarchar(MAX)	☑
Street	nvarchar(40)	☐
Blvd	nvarchar(MAX)	☑
City	nvarchar(MAX)	☑
State	nvarchar(MAX)	☑
PostalCode	nvarchar(MAX)	☑
PersonState	int	☐
Photo	varbinary(MAX)	☑
		☐

همامنطور که مشاهده می کنید ستونی به نام Photo از جنس varbinary(max) برای جدول Person ایجاد شده است. حال می خواهیم این فیلد را به جدولی دیگر مثلا PersonBlobs منتقل کنیم. برای این منظور کلاس PersonConfig را مطابق زیر تغییر دهید:

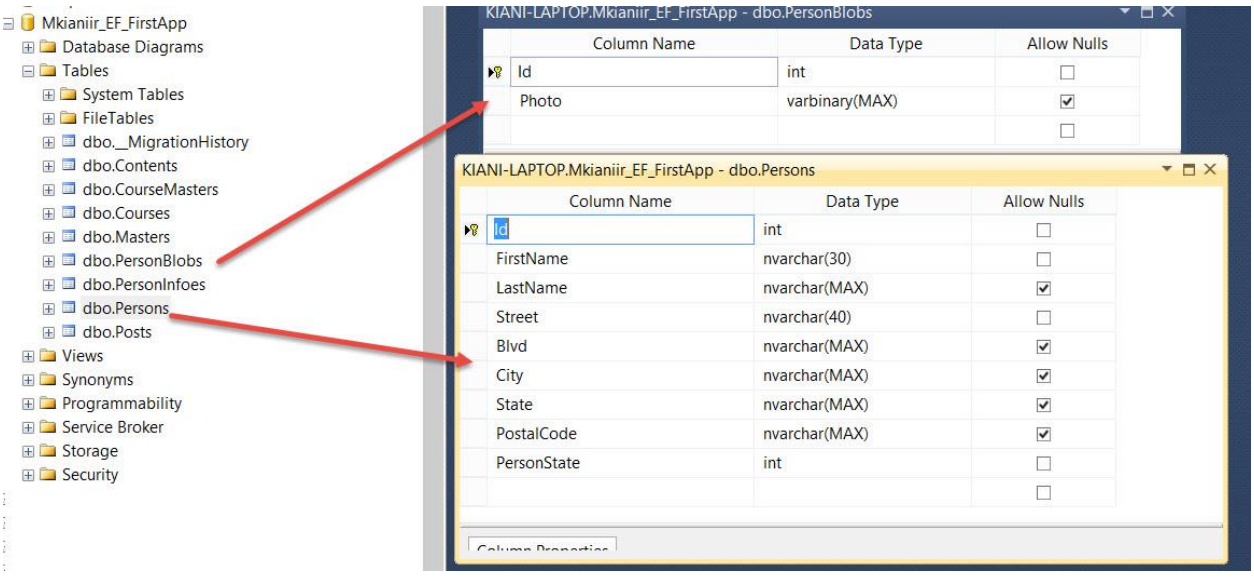
```
public class PersonConfig : EntityTypeConfiguration<Person>
{
    public PersonConfig()
    {
          //ToTable("Persons");

            Map(p =>
            {
                p.Properties(ph =>
                    new
                    {
                        ph.Address,
                        ph.FirstName,
                        ph.LastName,
                        ph.PersonState
                    });
                p.ToTable("Persons");
            });

            Map(p =>
```

```
            {
                p.Properties(ph =>
                    new
                    {
                        ph.Photo
                    });
                p.ToTable("PersonBlobs");
            });

            Property(p => p.FirstName).HasMaxLength(30).IsRequired();
            HasOptional(o => o.Info).WithRequired(r => r.Person);
            HasMany(m => m.Posts).WithRequired(r => r.Author).HasForeignKey(f =>
f.AuthorId);
            Ignore(p => p.CodeOnlyProperty);
        }
    }
```

همانطور که مشاهده می کنید دو نگاشت متفاوت برای موجودیت Person تعریف شده است. یکی شامل خاصیت Photo که به جدول PersonBlobs نگاشت شده است و دیگری شامل خصوصیت های Address ، FirstName ، LastName و PersonState که به جدول Persons نگاشت شده اند.

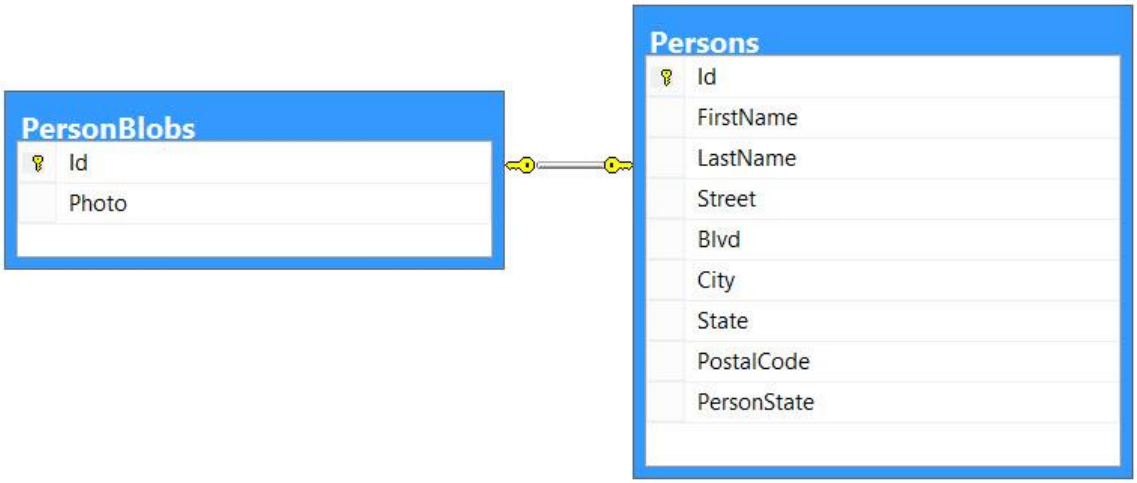
دقت داشته باشید که در این حالت دیگر نیازی به دستور ToTable(“Persons”) که در بخش قبلی برای پیکربندی نوشتیم نداریم. همانطور که مشاهده می کنید این دستور به صورت توضیح درامده است.

برنامه را اجرا کنید تا تغییرات در پایگاه داده اعمال شود:



شکل ٤ - ١٣

همانطور که مشاهده می کنید دو جدول به نام های Persons برای نگهداری فیلد های اصلی و PersonBlobs برای نگهداری فیلد Photo در پایگاه داده ایجاد شده است. اگر یک فایل دیاگرام ایجاد کنید متوجه خواهید شد که یک ارتباط یک به یک بین دو جدول فوق ایجاد شده است:



شکل ٤ - ١٤

همانطور که مشاهده می کنید ستون Id به عنوان کلید اصلی برای هر دو جدول مورد استفاده قرار گرفته است.

به این عملیاتی که در این بخش انجام شد Entity Splitting می گویند. در واقع یک موجودیت(Entity) را در دو جدول تقسیم کرده ایم. EF قابلیت عکس این عمل را نیز دارد. یعنی یک جدول را به دو موجودیت تقسیم کنیم. به این عمل Table Splitting می گویند.

یک جدول با چند موجودیت

این امکان زمانی که بخواهیم فیلد های اصلی یک موجودیت را از فیلد های فرعی آن جدا کنیم مفید خواهد بود. این کار باعث می شود که زمانی که نیازی به فیلد های فرعی نداریم آن ها را از پایگاه داده واکشی نکرده و بنابر این تاثیر مثبتی بر کارایی برنامه خواهیم داشت.

به عنوان مثال فرض کنید موجودیتی به نام کارمند می خواهیم در پروژه تعریف کنیم. هر کارمند دارای یک سری خصوصیات اصلی مانند نام و نام خانوادگی و ... دارد. همچنین یک سری خصوصیات فرعی نیز می توان برای کارمند در نظر گرفت.مثلا نام پدر،نام مادر و ...

در این حالت می توانیم دو کلاس مجزا تعریف کنیم. یکی برای اشاره به خصوصیات اصلی و دیگری برای نگهداری خصوصیات فرعی. شاید بگویید این روش را قبلا در یک ارتباط یک به یک نیز مشاهده کردیم.بله درست است اما در ارتباط یک به یک دو جدول جداگانه برای فیلد ها تعریف می شود و در هر جدول دسته ای از خواص در ون آن قرار می گیرند. اما در Table Splitting یک جدول برای دو موجودیت به کار می روند.علاوه بر این در حالت Table Splitting میتوان یک جدول را به بیش از دو موجودیت تقسیم کرد.

برای مثال دو کلاس به نام های Emploee و EmploeeDetails به صورت زیر به پروژه اضافه کنید:

```
    [Table("Emploees")]
    public class Emploee
    {
        [Key]
        public Int32 EmploeeId
        {
            get;
            set;
        }
        public String FirstName
        {
            get;
            set;
        }
        public String LastName
        {
            get;
            set;
        }
        public String Email
        {
            get;
            set;
        }
```

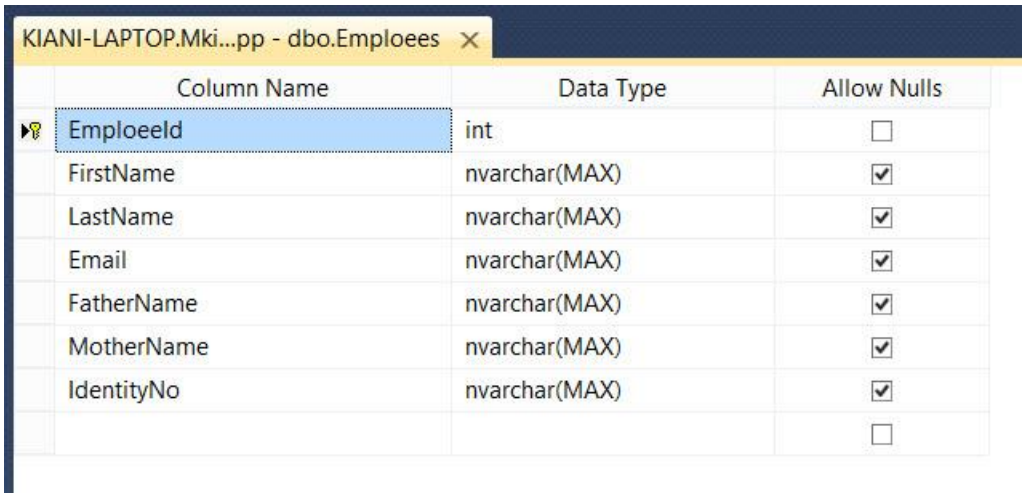
```
        public EmploeeDetails Details
        {
            get;
            set;
        }
    }


    [Table("Emploees")]
    public class EmploeeDetails
    {
        [Key]
        public Int32 EmploeeId
        {
            get;
            set;
        }
        public String FatherName
        {
            get;
            set;
        }
        public String MotherName
        {
            get;
            set;
        }
        public String IdentityNo
        {
            get;
            set;
        }
        public Emploee Emploee
        {
            get;
            set;
        }
    }
```

همانطور که مشاهده می کنید در هر کلاس یک سری خصوصیت تعریف شده است. همچنین در هر کلاس یک خاصیت از نوع کلاس دیگر تعریف شده است.علاوه بر این فیلد EmploeeId به عنوان کلید اصلی در هر دو جدول تعریف شده است. همچنین هر دو کلاس به یک جدول به نام Emploees نگاشته شده اند.(در این حالت از صفات برای پیکربندی موجودیت ها استفاده شده است که البته می توانید توسط دستورات Fluent نیز این کار را انجام دهید)

حال دستورات زیر را درون متد OnModelCreating در کلاس PersonContext اضافه کنید تا موقع ایجاد پایگاه داده جدول Emploees نیز ایجاد گردد:

```
modelBuilder.Entity<Emploee>().HasRequired(r => r.Details).WithRequiredPrincipal(r => r.Emploee);
```

دستور فوق یک ارتباط دو طرفه بین موجودیت Emploee و EmploeeDetails ایجاد می کند. حال برنامه را اجرا کنید و نتیجه را در پایگاه داده مشاهده نمائید:



شکل ٤ - ١٥

همانطور که مشاهده می کنید یک جدول به نام Emploees که شامل خصوصیات موجودیت Emploee و نیز EmploeeDetails می باشد تشکیل شده است. توجه کنید که ستون EmploeeId به عنوان کلید اصلی تعریف شده است.

اصلاح پایگاه داده با استفاده از Entity Framework Migration

تاکنون هر تغییری که در پایگاه داده اعمال کردیم باعث شد تا پایگاه داده حذف و مجددا ایجاد شود. این عمل را با استفاده از Initializer ها در EF انجام دادیم:

```
Database.SetInitializer(new DropCreateDatabaseIfModelChanges<PersonContext>());
```

همانطور که پیشتر نیز گفته شد با استفاده از دستور فوق به EF اعلام کردیم که هر زمان که تغییری در مدل داده ای در برنامه به وجود آمد پایگاه داده را حذف و مجددا ایجاد کن. همانطور که می دانید حذف پایگاه داده به معنای از بین رفتن تمامی اطلاعات آن می باشد و پرواضح است که این روش به هیچ وجه نمی تواند در یک پروژه واقعی که تغییر در آن در طی زمان وجود پروژه اجتناب ناپذیر است به کار گرفته شود.

EF قابلیتی را به نام Migration در اختیار شما قرار می دهد تا بتوانید بدون حذف کردن پایگاه داده موجود و از بین رفتن اطلاعات، تغییرات مورد نظر را در پایگاه داده اعمال نمائید.

اگر به پایگاه داده هایی که تاکنون در طول آموزش ایجاد کرده ایم دقت کرده باشید جدولی به نام MigrationHistory_ به صورت خودکار در پایگاه داده همواره ایجاد می شود. این جدول ارتباطی با منطق برنامه ندارد.عموما شما در طول پروژه از این جدول استفاده ای نخواهید برد.اما EF از این جدول برای بررسی تغییرات مدل داده ای برنامه استفاده می کند. اجازه دهید نگاهی با ساختار این جدول بیندازیم:

	MigrationId	ContextKey	Model	ProductVersi...
1	201507180810549_InitialCreate	mkianiir.EF.FirstApp.PersonContext	0x1F8B0800000000000400ED5DDB6EE3BA157D2FD07F10F4...	6.1.3-40302

شکل ٤ - ١٦

ستون MigrationId نام فایل اولیه Migration می باشد(در ادامه با این فایل ها آشنا خواهید شد).مقدار عددی ابتدای نام، زمان ایجاد این فایل را نشان می دهد.ستون ContextKey نام کلاس Context پروژه می باشد(هر پروژه می تواند بیش از یک Context برای یک پایگاه داده داشته باشد) وستون Model نیز معادل Hash شده کلاس Context برای بررسی تغییرات می باشد.

EF به ازای هر تغییری که در پایگاه داده به روش Migration ایجاد شود یک رکورد در این جدول درج می نماید و بر اساس رکوردهای درج شده تشخیص می دهد که آیا مدل داده ای تغییری داشته یا خیر.

کتابخانه Migration به دو صورت قابل استفاده می باشد.

۱- به صورت خودکار (AutomaticMigrationEnabled)

۲- به صورت دستی یا صریح و با استفاده از دستورات کتابخانه Migration

یک پروژه جدید ایجاد نمائید. من نام آن را mkianiir.EF.Migration قرار دادم.

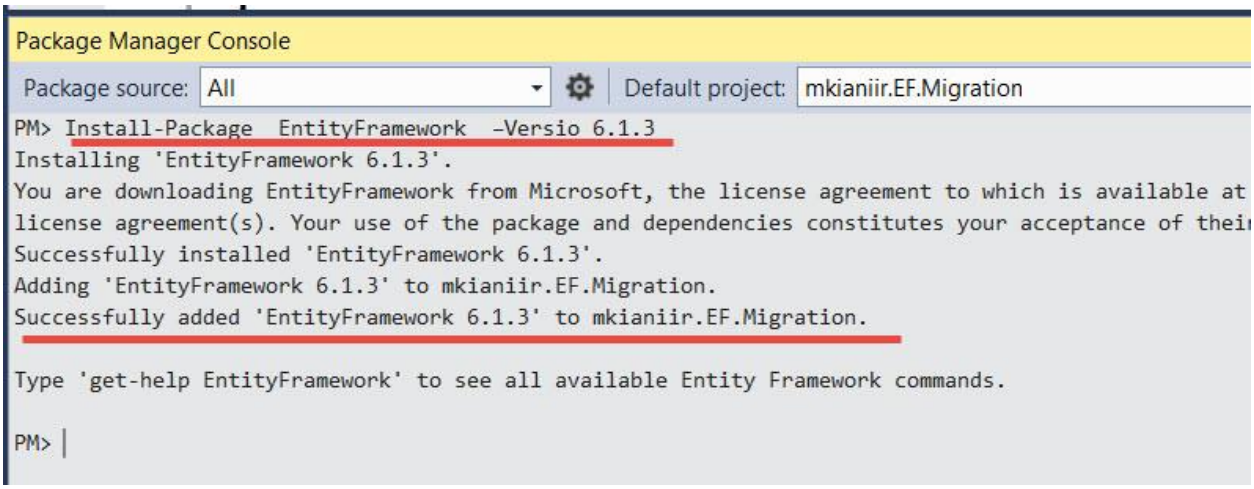
یک کلاس به نام Person به صوت زیر ایجاد نمائید:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace mkianiir.EF.Migration
{
    [Table("Persons")]
    public class Person
    {

        [Key]
        public Int32 PersonId
        {
            get;
            set;
        }
        public String FirstName
        {
            get;
            set;
        }
        public String LastName
        {
            get;
            set;
        }
    }
}
```

به فضای نام های مربوط به DataAnnotation ها دقت کنید.

حال با استفاده از کنسول نیوگت دستور زیر را اجرا کنید تا کتابخانه EF به پروژه اضافه شود:

Install-Package EntityFramework –Version ۶,۱,۳



شکل ٤ - ١٧

حال یک کلاس دیگر به نام MigrationContext به صورت زیر به پروژه اضافه کنید:

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace mkianiir.EF.Migration
{
    public class MigrationContext : DbContext
    {
       public MigrationContext()
            : base("mkianiir.EF")
        {

        }

        public DbSet<Person> Persons
        {
            get;
            set;
        }
    }
}
```

تنظیمات مربوط به رشته اتصال پایگاه داده را در فایل App.config اعمال کنید:

```
  <connectionStrings>
    <clear/>
    <add name="mkianiir.EF"
         connectionString="Data Source=.;Initial Catalog=Mkianiir_EF_Migration;Integrated
Security=SSPI"
         providerName="System.Data.SqlClient"
         />
```

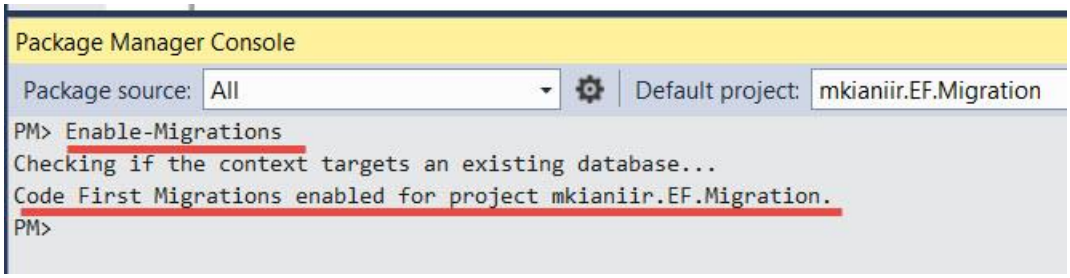
```
</connectionStrings>
```

به فضای نام System.Data.Entity دقت کنید.

حال به کنسول نیوگت رفت و دستور زیر را اجرا کنید:

Enable-Migrations

این دستور باعث می شود که Migration برای پروژه شما فعال شود.



شکل ٤ - ١٨

با این عمل یک پوشه به نام Migrations در Solution Explorer ایجاد می گردد.در این پوشه کلاسی به نام Configuration.cs به صورت زیر تعریف شده است:

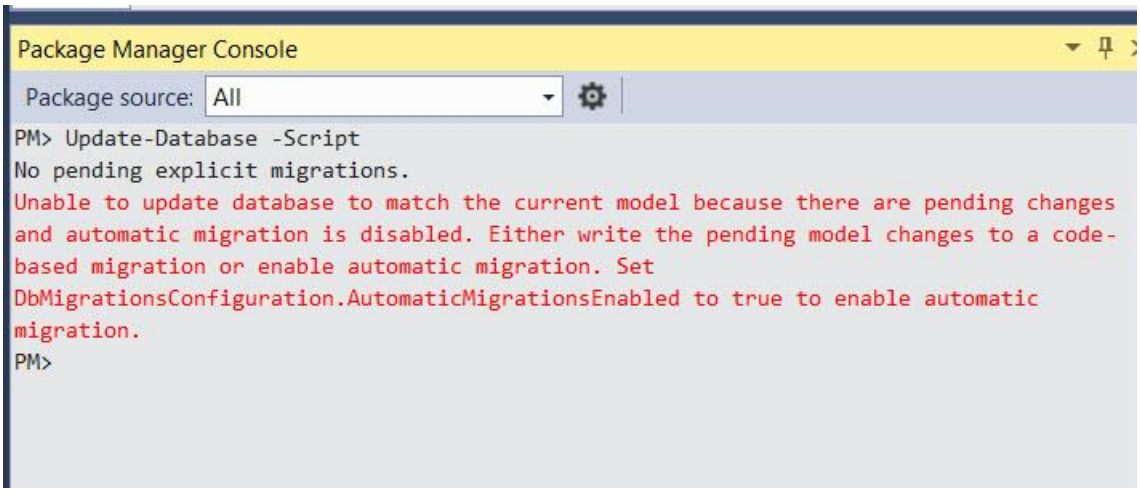
```
using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration :
DbMigrationsConfiguration<mkianiir.EF.Migration.MigrationContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(mkianiir.EF.Migration.MigrationContext context)
        {
            context.Persons.Add(new Person
            {
                FirstName = "Mehdi",
                LastName = "Kiani"
            });        }
    }
```

همانطور که مشاهده می کنید کلاس Configuration از کلاس DbMigrationConfiguration با نوع کلاس MigrationContext (Context مربوط به پروژه) مشتق می شود.

در کلاس Configuration متد Seed به صورت Override نوشته شده است. از این متد گاها برای درج یا ویرایش رکورد ها پس از عملیات Migration استفاده می شود هر بار که عمل Migration انجام می شد دستورات این متد نیز اجرا می شوند. پس بنابر این چون تعداد دفعاتی که این متد اجرا می شود بیش از یک بار می باشد می بایستی دستورات آن به دقت نوشته شوند.

به پنجره کنسول نیوگت رفته و دستور زیر را در آن اجرا کنید:

```
Update-Database  –Script
```

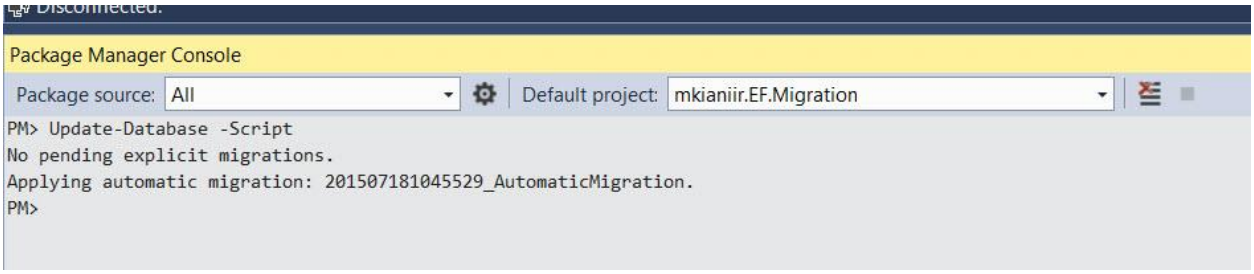


شکل ٤ - ١٩

همانطور که مشاهده می کنید جهت اجرای دستور می بایستی خاصیت AutomaticMigrationsEnabled را از مقدار false به مقدار true تغییر دهیم. برای این منظور کلاس Configuration را باز کرده و در Constructor این کلاس مقدار AutomaticMigrationsEnabled را برابر با true قرار دهید و سپس مجددا دستور فوق را در کنسول نیوگت اجرا کنید:

```
public Configuration()
        {
            AutomaticMigrationsEnabled = true;
        }
```

با اجرای دستور فوق پنجره کنسول شبیه به عکس زیر خواهد بود:



شکل ٤ - ٢٠

اگر به ویژوال استودیو دقت کنید یک فایل جدید به نام SqlQuery_١.sql ایجاد و دستورات زیر در آن نوشته شده است.

```
CREATE TABLE [dbo].[Persons] (
    [PersonId] [int] NOT NULL IDENTITY,
    [FirstName] [nvarchar](max),
    [LastName] [nvarchar](max),
    CONSTRAINT [PK_dbo.Persons] PRIMARY KEY ([PersonId])
)
CREATE TABLE [dbo].[__MigrationHistory] (
    [MigrationId] [nvarchar](150) NOT NULL,
    [ContextKey] [nvarchar](300) NOT NULL,
    [Model] [varbinary](max) NOT NULL,
    [ProductVersion] [nvarchar](32) NOT NULL,
    CONSTRAINT [PK_dbo.__MigrationHistory] PRIMARY KEY ([MigrationId], [ContextKey])
)
INSERT [dbo].[__MigrationHistory]([MigrationId], [ContextKey], [Model], [ProductVersion])
VALUES (N'201507181045529_AutomaticMigration',
N'mkianiir.EF.Migration.Migrations.Configuration',
0x1F8B0800000000000400CD57CB6EE33614DD17987F20B8EA0019318F4D1B4833489D78604C9C04A3CCEC69E
9DA21860F55A402FBDBBAE827F5177AA9B7A5D8B183162DBC91C87B0FCF3DBC0FF9AF3FFE0C3FAD9524CF905B
617444CF82534A402726157A15D1C22D3FFC423F7D7CF7537893AA35F9DED85D783BF4D436A24FCE65978CD9E
40914B78112496EAC59BA20318AF1D4B0F3D3D35FD9D9190384A0884548F8B5D04E28285FF075627402992BB8
9C9B14A4ADD771272E51C91D5760339E4044D50FC1B5107970330DE6629573878C28B99282239918E49212AEB
571E5C6E5370BB1CB8D5EC5192E70F9B8C900ED965C5AA843B8ECCC0F8DE6F4DC47C33AC7062A29AC33EA48C0
B38B5A1E36747F93C8B4950F05BC41A1DDC6475D8A18D107BC430F3C3CEA7222736FB643E1A0F23B212FEE9EB
4A98119E47F2764524857E41069285CCEE50979281652245F60F3687E808E7421659F2972C5BDAD055C7AC84D
06B9DB7C85E516FF594A09DBF66643F7D679E459C539D3EEE29C923B24C21712DAA4E869123B93C367D080614
2FAC09D831CEF74964229EB88C3E0C4A9C8ADF38FCD91988A585894CCF9FA16F4CA3D45141F29998A35A4CD4A
4DE39B165887E8E4F2025E3BE996FF2B0785ACCB9F715661D93A2E509C9A449B0E7E03D6EE8524C372ACF3CCD
6E76D875501C7E0B6EECC52D2F1A88A3C68F2F825BA2DB1AE91B0AA93341D87ED6839E19C67192AD76B41F50A
89ABFE33F9101F5F95AAC260897DA1385BB6ED4998757C05835D3C1A99962975CD1D5F707F7793548DCC46D7B
043E2E6B86DA58715D809DFD8FBE7CA675FA71802754A4E3138850554C6092D9DB6358D1CCB39C025CF7716F4
C4C842E9FD0D621F52AF50FB50BDE5C3B1BA52EC4375AB63A4900D9419DE011B5DC2A0F50DEF745F4D0C4DDAD
3DBDA18D44058E7E3EBB37994A095092528D1B3487D72C61BEB4005DE20887F97132930DECE608EF9B404EBAA
194171369E0F66FBFF67CE326B5379D0B0FD4FC79CF0FABE3AC84653F0C8C9A69F799E3CF1FC67C5D7EFFB706
```

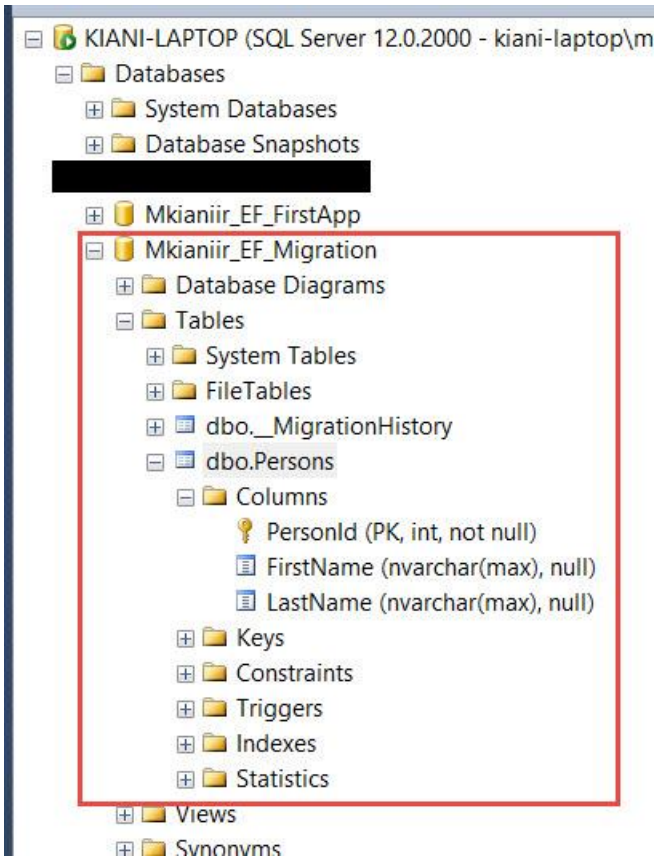
```
F995E47801D37A1C61DF490F1B367FA54091ED17461907C45B29D596F1C4DE3720B59FF8339BC062B561D84FF
7CD690F83CEE401B9B995E9A466D8CABCFA831195CC61C1C4F519EABDC89254F1C6E27606DF939F19DCB024D6
ED402D299BE2F5C56B82B6B412DE4D6C750C8F69F5FCEDF6DCEE17DE6DFEC3F1102D2141802DCEBDF0A21D396
F7749CDABB207CA6D4F582ACF0730AE1569B16E9CEE803816AF9AE2103EDABED11542611CCDEEB983FC35BB8E
147D42DAC78B269BAE66E90D72F625BF6F05A709CE3CAD6189DBFFF13C8FCBFC08F7F038F71DB8A370E0000 ,
N'6.1.3-40302')
```

زمانی که دستور Update-Database -Script را اجرا می کنید چنانچه تغییراتی در مدل رخ داده باشد یک فایل sql ایجاد و دستورات مورد نیاز برای اصلاح پایگاه داده در ان درج می گردد. کافیست این دستورات را در محیط sql در یک New Query اجرا کنید تا پایگاه داده شما بروز شود. در حالت لوکال می توانید با استفاده از دستور Update-Database پایگاه داده را ایجاد و تغییرات را اعمال نمائید.

مجددا به پنجره کنسول نیوگت رفته و دستو زیر را ایجاد کنید:

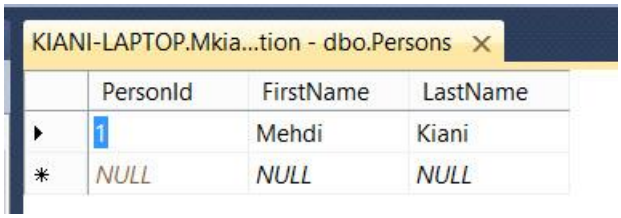
```
Update-Database
```

پس ا ز پایان اجرای دستور فوق به محیط Sql Server بروید و تغغیرات را مشاهده کنید:



شکل ٤ - ٢١

همانطور که مشاهده می کنید پایگاه داده Mkianiir_EF_Migration ایجاد و درون ان دو جدول MigrationHistory_ و Persosns ایجاد شده است. اگر به داده های جدول Persons نگاه بیندازید متوجه خواهید شده که یک رکورد در این جدول به ثبت رسیده است:



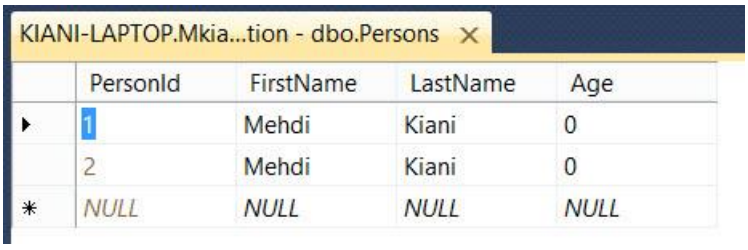
شکل ٤ - ٢٢

این همان رکوردی است که در در متد Seed دستورات ان را نوشته بودیم.

حال کلاس Person را باز کرده و خاصیت Age را به صورت زیر به آن اضافه کنید:

```
public Int32 Age
        {
            get;
            set;
        }
```

حال مجددا دستور Update-Database را در کنسول نیوگت اجرا کنید. چنانچه به پایگاه داده مراجعه کنید خواهید دید که ستون Age از نوع integer و با مقدار پیش فرض ۰ در جدول Persons ایجاد شده است.همانطور که مشاهده می کندی با بروز رسانی پایگاه داده، داده های قبلی که در پایگاه داده بود از بین نرفت:



شکل ٤ - ٢٣

همانطور که مشاهده می کنید اولین رکورد جدول Person که در بروز رسانی اول پایگاه داده ایجاد شده بود به قوت خود باقی مانده است. همچنین رکور دوم در بروز رسانی دوم ایجاد شده است. همانطور که گفته شد در هر بروز رسانی پایگاه داده توسط Migration دستورات متد Seed اجرا خواهند شد و چون تاکنون دوبار پایگاه داده بروز رسانی شده است بنابر این دو رکورد در جدول Person درج شده است.

نکته: همانطور که گفته شد متد Seed در هر بروزرسانی اجرا می شود پس می بایستی در نوشتن دستورات آن دقت لازم را بعمل آورده تا جامعیت داده ها حفظ شود.

مجددا خاصیت دیگری به نام NickName را به کلاس Person به صورت زیر اضافه کنید و سپس مجددا توسط کنسول نیوگت و با دستور Update-Database پایگاه داده را بروز رسانی کنید:

```
[MaxLength(50)]
 public String NickName
 {
     get;
     set;
 }
```

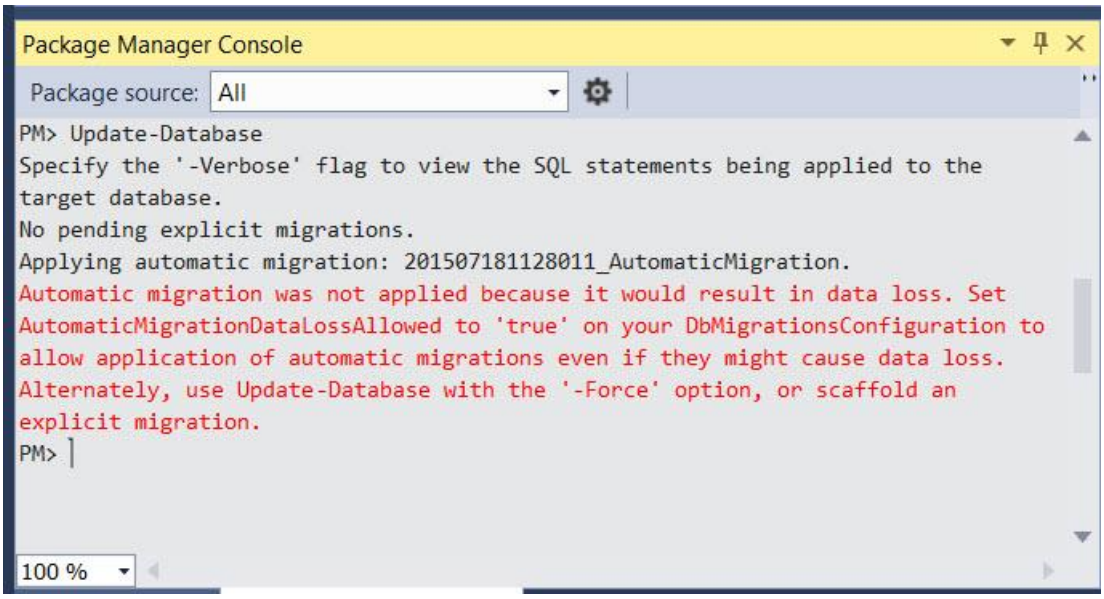
همانطور که مشاهده می کنید حداکثر ۵۰ کاراکتر برای فیلد NickName در نظر گرفته شده است.

پس از بروز رسانی پایگاه داده مشاهده خواهید کرد که ستون NickName با حداکثر طول ۵۰ در جدول Persons ایجاد شده است. همچنین مقدار Null برای این فیلد در رکورد های این جدول در نظر گفته شده است. حال فرض کنید بخواهیم حداکثر طول فیلد NickName را از ۵۰ به ۴۰ تغییر دهیم. با این کار ممکن است داده های فیلد NickName مربوط به رکورد هایی که مقدار فیلد NickName آن ها از ۴۰ کاراکتر بیشتر باشند از بین بروند. ببینیم در این حالت عکس العمل Migration چه خواهد بود؟

خاصیت NickName در کلاس Person را به صورت زیر تغییر دهید:

```
[MaxLength(40)]
 public String NickName
 {
     get;
     set;
 }
```

مجددا دستور Update-Database را در پنجره کنسول نیوگت اجرا کنید:



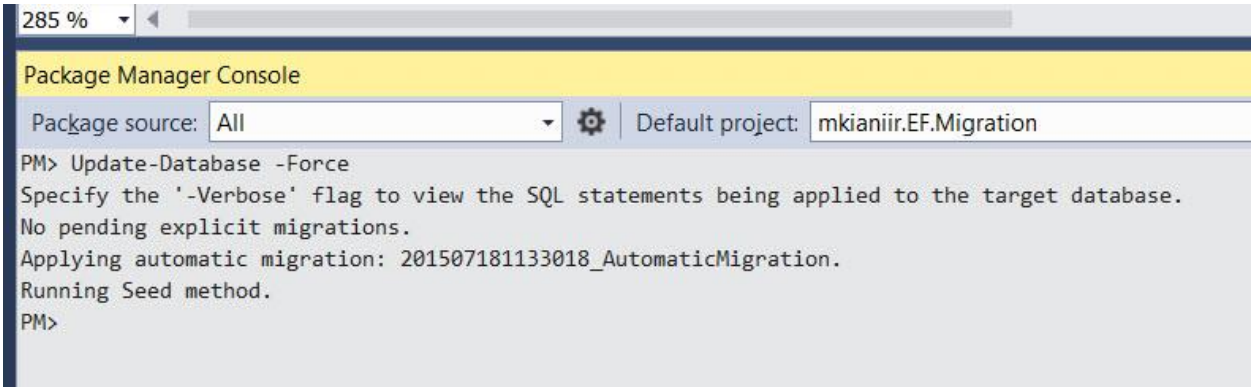
شکل ٤ - ٢٤

همانطور که مشاهده می کنید تغییرات در پایگاه داده اعمال نشده است چراکه ممکن است داده های موجود در پایگاه داده از بین برود. به همین جهت از شما خواسته شده چنانچه مطمئنید که می خواهید این تغغیرات را اعمال کنید می بایستی یکی از این دو کار زیر را انجام دهید:

۱- یا در کلاس Configuration مقدار خاصیت AutomaticMigrationDataLossAllowed را برابر با true قرار دهید.

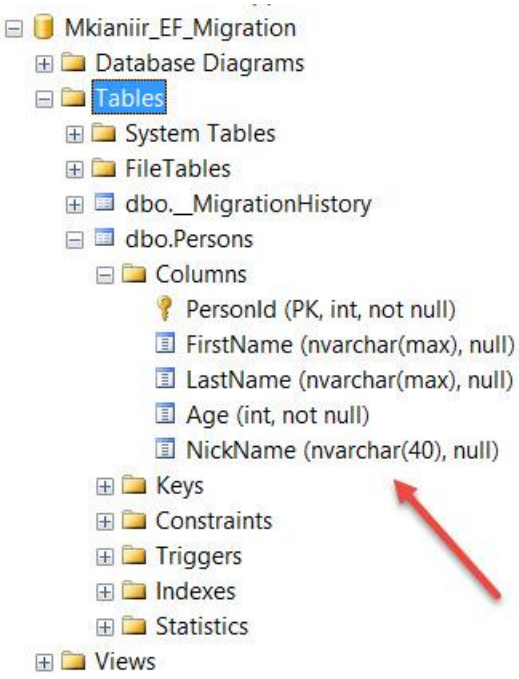
۲- دستور Update-Database را همراه با آرگومان Force– به کار ببرید.

در اینجا ازروش دوم استفاده می کنیم. بنابر این در پنجره کنسول نیوگت دستور زیر را اجرا کنید:



شکل ٤ - ٢٥

همانطور که مشاهده می کنید دستور بدون خطا اجرا خواهد شد و تغغیرات در پایگاه داده اعمال خواهند شد.



شکل ٤ - ٢٦

استفاده از کتابخانه Migration

همانطور که در ابتدای این بخش گفته شد علاوه بر حالت خودکار Migration می توان از حالت دستی یا Manual نیز برای عملیات Migration استفاده کرد. در این حالت اغلب اگرچه نیاز به عملیات بیشری نسبت به حالت خودکار میباشد اما در عوض امکان کنترل بیشتری بر روی عملیات خواهید داشت.

برای شروع، پایگاه داده ای که در بخش قبل ایجاد شده بود را از Sql Server حذف نمائید. همچنین مقدار خاصیت AutomaticMigrationsEnabled در کلاس Configuration را به مقدار پیش فرض آن یعنی false تغییر دهید.

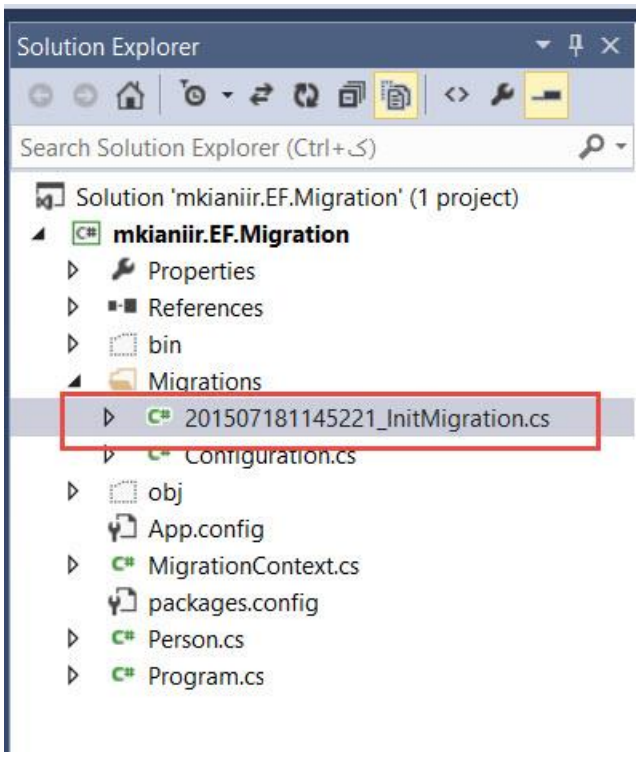
حال کنسول نیوگت را باز کنید و دستور زیر را در ان اجرا کنید:

```
Add-Migration       InitMigration
```

با اجرای دستور فوق یک کلاس به نام InitMigration در پوشه Migrations در Solution Explorer به صورت زیر اضافه می شود:

```
public partial class InitMigration : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Persons",
            c => new
                {
                    PersonId = c.Int(nullable: false, identity: true),
                    FirstName = c.String(),
                    LastName = c.String(),
                    Age = c.Int(nullable: false),
                    NickName = c.String(maxLength: 40),
                })
            .PrimaryKey(t => t.PersonId);

    }

    public override void Down()
    {
        DropTable("dbo.Persons");
    }
}
```

همانطور که مشاهده می کنید این کلاس از کلاس DbMigration مشتق شده است.

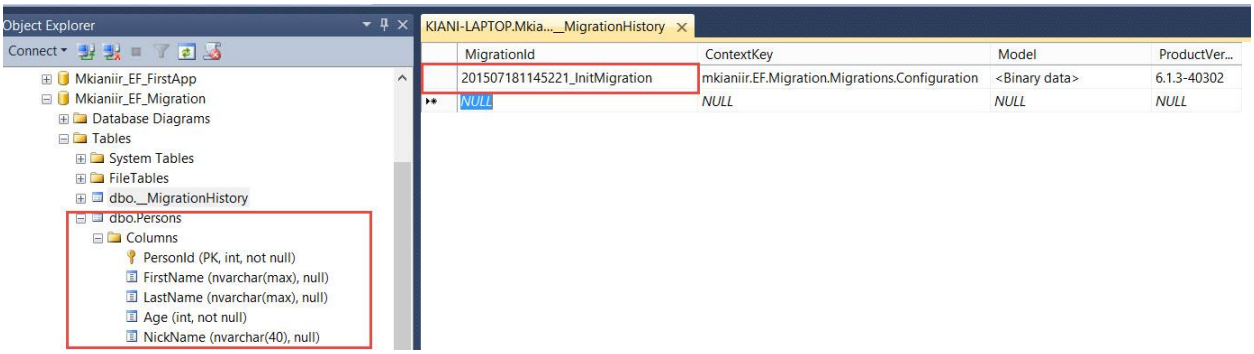


شکل ٤ - ٢٧

اعدادی که قبل از نام فایل مربوط به کلاس InitMigration قرار داده شده است زمان ایجاد این فایل را نشان می دهد. این مقدار بعدا در جدول MigrationHisotry_ درج خواهد شد.

اگر به دستورات کلاس InitMigration دقت کنید دو متد Up و Down به صورت Override شده وجود دارد.دستورات درون Up تغییرات جدید را برای پایگاه داده اعمال می کند و دستورات Down برای Rollback کردن این دستورات به کار می رود . این دستورات زمانی به کار می روند که ممکن است در زمانی بخواهیم تغییرات پایگاه داده را به نسخه قبلی آن باز گردانیم.

حال اگر دستور Update-Database را مجددا اجرا کنید پایگاه داده جدید با دو جدول Persons و MigrationHistory_ ایجاد خواهد شد:



شکل ٤ - ٢٨

اگر به جدول migrationhistory_ دقت کنید مشاهده خواهید کرد یک رکورد در ان درج شه است که مقدار MigrationId آن برابر با نام فایل InitMigration می باشد که کمی قبل تر آن را ایجاد کردیم.

حال کلاس Person را باز کنید و خاصیت زیر را به ان اضافه کنید:

```
public String IdentityNo
        {
            get;
            set;
        }
```

حال دستور زیر را در پنجره کنسول نیوگت اجرا کنید:

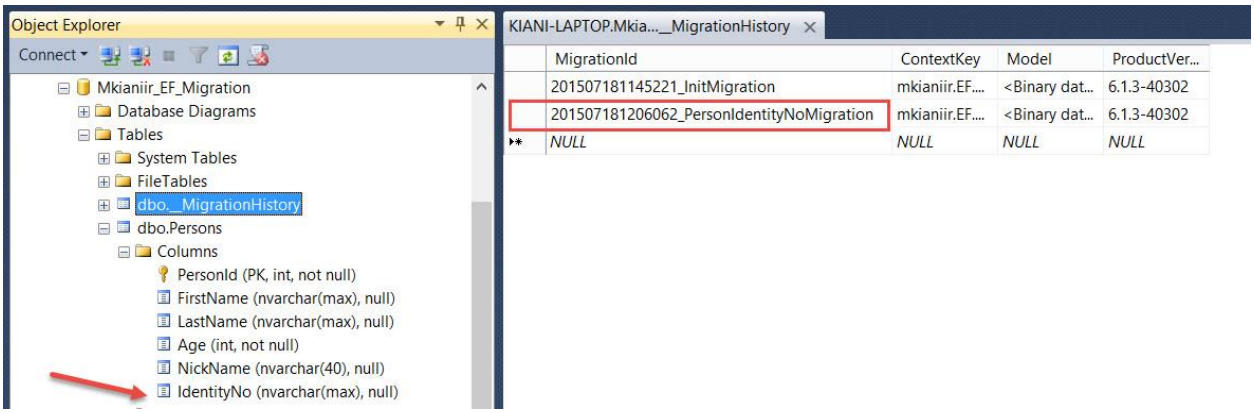
```
Add-Migration PersonIdentityNoMigration
```

با اجرای دستور فوق یک کلاس به نام PersonIdentityNoMigration که مشتق شده از کلاس DbMigration است با دستورات زیر به پوشه Migrations اضافه می شود:

```
public partial class PersonIdentityNoMigration : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Persons", "IdentityNo", c => c.String());
    }

    public override void Down()
    {
        DropColumn("dbo.Persons", "IdentityNo");
    }
}
```

همانطور که مشاهده می کنید در متد Up دستور ایجاد ستون جدیدی به نام IdentityNo و در متد Down دستور حذف ستون به صورت خودکار ایجاد شده است.

حال با دستور Update-Database می توانید تغغیرات ایجاد شده در مدل را به پایگاه داده منتقل کنید:



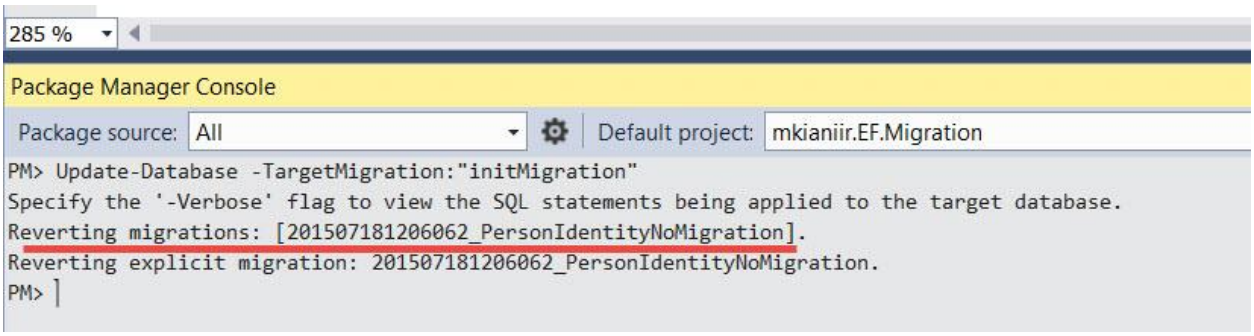
شکل ٤ - ٢٩

همانطور که در شکل فوق مشاهده می کنید علاوه بر ایجاد ستون IdentityNo در جدول Person یک رکورد دیگری در جدول MigrationHistory_ ایجاد شده است که مقدار ستون MigrationId آن نام فایل کلاس PersonIdentityNoMigration می باشد.

بازگشت به عقب:

همانطور که گفته شد یکی از مزایای Migration های دستی یا صریح این است که می توانیم در هر زمانی که بخواهیم به نسخه های قبلی روجوع کنیم. به عنوان مثال دستور زیر در کنسول نیوگت اجرا کنید:

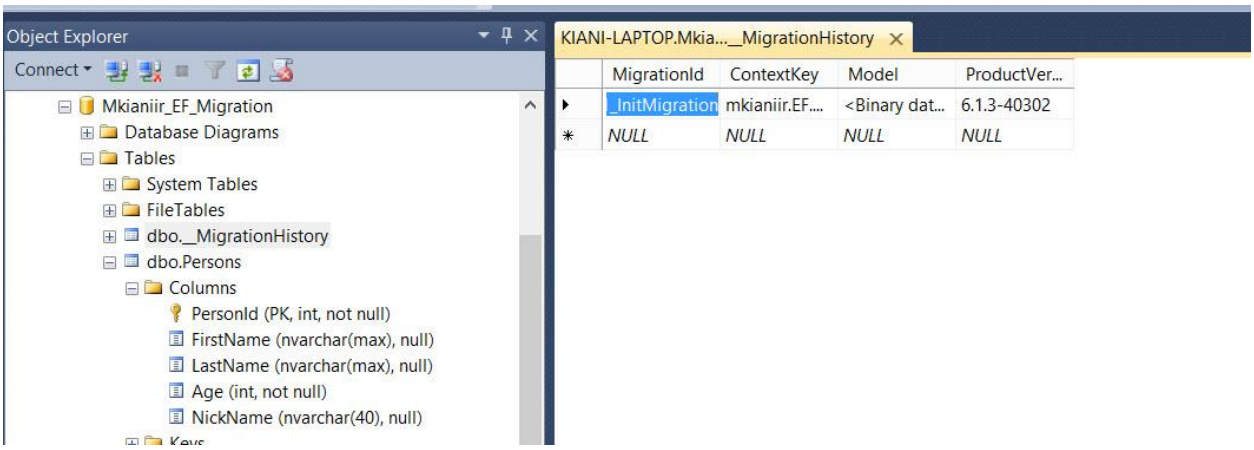
```
Update-Database -TargetMigration:"initMigration"
```



شکل ٤ - ٣٠

همانطور مشاهده می کنید تغییرات پایگاه داده به حالت InitMigration برگشته و Migration هایی که بعد از آن ایجاد شده اند نظیر PersonIdentityNoMigration همگی Rollback شده اند.

حال اگر به پایگاه داده رجوع کنید مشاهده خواهید کرد ستون IdentityNo که در نتیجه آخرین Migration به پایگاه داده ایجاد شده بود از جدول Persons حذف شده است. همچنین رکورد های مربط به Migration هایی که بعد از InitMigration ایجاد شده اند نیز حذف شده اند.



برای Rollback کردن تمامی Migration ها می توانید از دستور زیر استفاده نمائید:

```
Update-Database      -TargetMigration:٠
```

یا از دستور زیر استفاده نمائید:

```
Update-Database -TargetMigration:$InitialDatabase
```

کار با View ها در EF

یقینا با View ها در Sql Server آشنایی دارید. View ها ساختار هایی مانند جداول هستند که می توانند دو یا چند جدول را ادغام کرده و یک نمایش سطح بالایی از آن ها ارائه دهد. در حال حاضر EF Code First پشتیبانی کاملی از View ندارد. اما می توانید شبیه آن چه که با جداول انجام می دادید با View ها نیز انجام دهید.اجازه دهید با یک مثال وارد مباحث این بخش شویم:

یک پروژه جدید ایجاد کنید(من نام آن را mkianiir_EF_OtherTopics گذاشتم).سپس توسط کنسول نیوگت کتابخانه EF را نصب کنید. همچنین Migration را برای این پروژه فعالی کنید.

پس از انجام عملیات فوق دو کلاس به نام های Person و PersonType به صورت زیر تعریف کنید:

```
 [Table("PersonTypes")]
  public class PersonType
   {
        public Int32 PersonTypeId
        {
            get;
            set;
        }
        public String Name
        {
            get;
            set;
        }
        public virtual ICollection<Person> Persons
        {
            get;
            set;
        }
    }
```

```
[Table("Persons")]
    public class Person
    {
        [Key]
        public Int32 PersonId
        {
            get;
            set;
        }
        public String FirstName
        {
            get;
            set;
        }
        public String LastName
```

```
        {
            get;
            set;
        }

        public virtual PersonType PersonType
        {
            get;
            set;
        }

    }
```

حال یک کلاس به نام PersonContext به صورت زیر ایجاد نمائید:

```
    public class PersonContext : DbContext
    {
        public PersonContext()
            : base("mkianiir.EF")
        {

        }
        public DbSet<Person> Persons
        {
            get;
            set;
        }
        public DbSet<PersonType> PersonTypes
        {
            get;
            set;
        }
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            modelBuilder.Entity<PersonType>().HasMany(m => m.Persons).WithRequired();
        }
    }
```

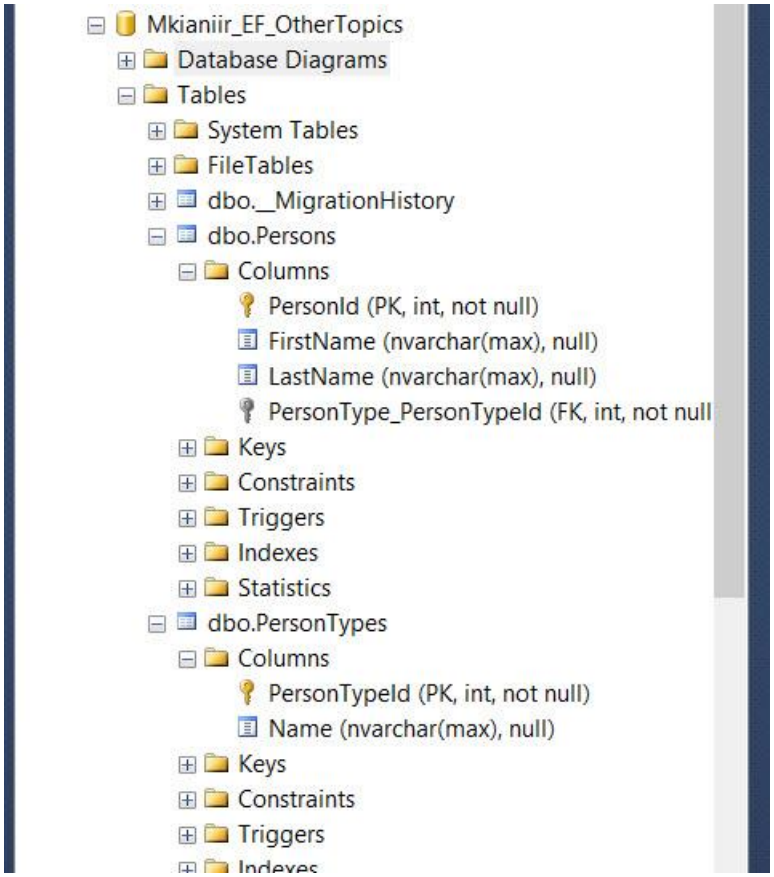
در کلاس Context فوق مجموعه های Persons و PersonTypes و همچنین ارتباط بین آن ها(ارتباط یک به چند از سوی PersonType به سوی Person) تعریف شده است.

دستورات زیر که مربوط به تعریف رشته اتصال به پایگاه داده می باشد را در فایل App.config تعریف کنید:

```
<connectionStrings>
    <clear/>
    <add name="mkianiir.EF"
         connectionString="Data Source=.;Initial
Catalog=Mkianiir_EF_OtherTopics;Integrated Security=SSPI"
         providerName="System.Data.SqlClient"
```

```
        />
    </connectionStrings>
```

حال توسط دستورات Migration اقدام به ایجاد پایگاه داده نمائید. پس انجام دستورات می بایستی پایگاه داده ای شبیه به عکس زیر تولید شده باشد.



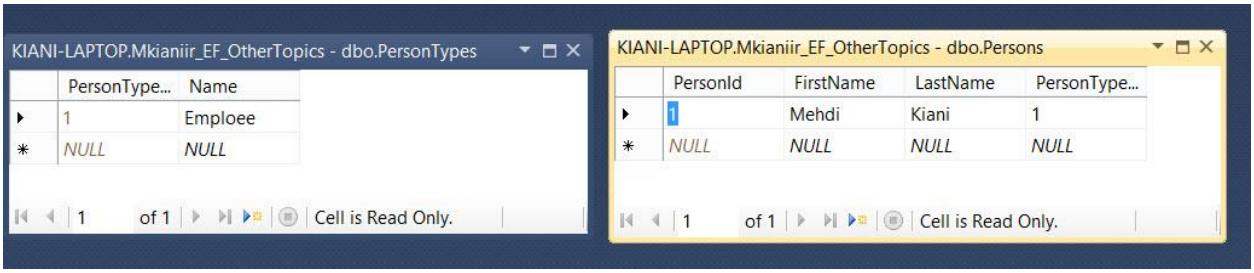
شکل ٤ - ٣١

حال دستورات زیر را در متد Main کلاس Program بنویسید و برنامه را اجرا کنید:

```
using (var context = new PersonContext())
            {
                PersonType pte = new PersonType
                {
                    Name = "Emploee"
                };

                Person p1 = new Person
                {
                    FirstName = "Mehdi",
                    LastName = "Kiani",
                    PersonType = pte
```

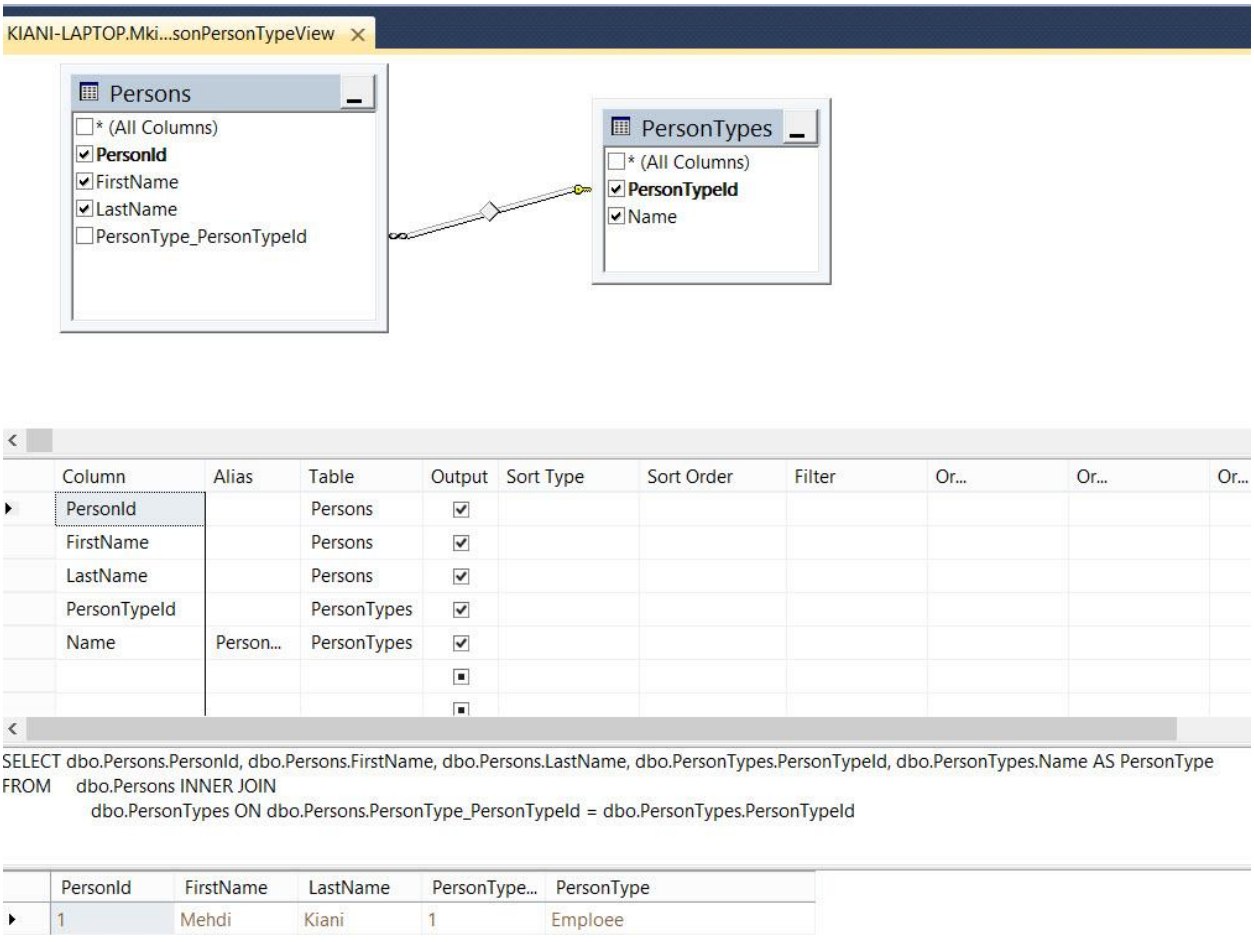
```
            };
            context.Persons.Add(p1);
            context.SaveChanges();
        }
```

پس اجرای برنامه یک رکورد به جدول Persons و یک رکورد نیز به جدول PersonTypes اضافه خواهد شد.



شکل ٤ - ٣٢

حال به پایگاه داده رفته و یک View با نام PersonPersonTypeView به صورت زیر ایجاد کنید:



شکل ٤ - ٣٣

همانطور که مشاهده می کنید این ویو رکورد های Person را به همراه نوع آن واکشی می کند.

حال یک کلاس به نام PersonPersonTypeView ایجاد کنید و دستورات ان را مطابق زیر تغییر دهید:

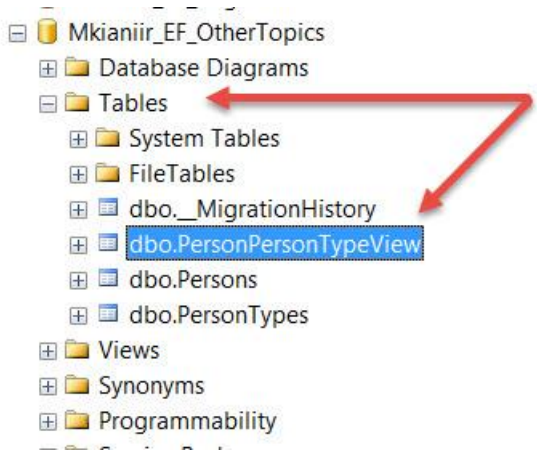
```
   [Table("PersonPersonTypeView")]
   public class PersonPersonTypeView
    {
        public Int32 PersonId
        {
            get;
            set;
        }
        public String FirstName
        {
            get;
            set;
        }
        public String LastName
        {
            get;
            set;
        }
        public Int32 PersonTypeId
        {
            get;
            set;
        }
        public String PersonType
        {
            get;
            set;
        }
    }
```

حال دستورات زیر را به کلاس PersonContext اضافه کنید:

```
public DbSet<PersonPersonTypeView> PersonPersonTypes
        {
            get;
            set;
        }
```

حال اگر با دستورات Update-Database از کتابخانه Migration اقدام به بروز رسانی پایگاه داده نمائید خواهید دید که جدول جدیدی در پایگاه داده تشکیل نخواهد شد بلکه ویوی PersonPersonType برای موجودیت PersonPersonType مورد استفاده قرار خواهد گرفت. در صورتی که اگر پایگاه داده را حذف و

مجددا بروز رسانی نمائید خواهید دید که به جای ویو، یک جدول به نام PersonPersonType تشکیل خواهد شد:



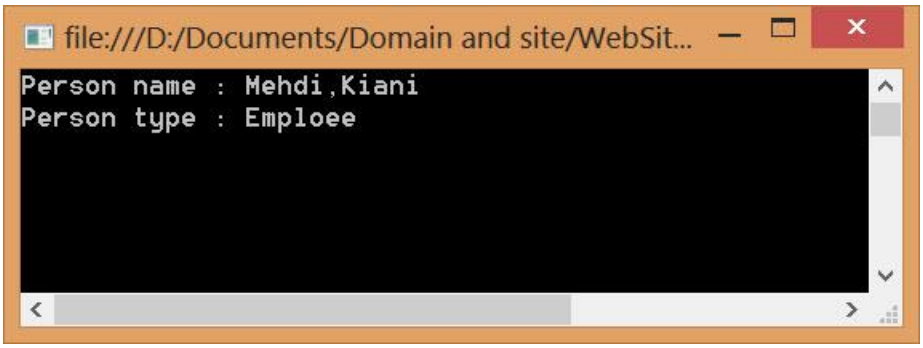
شکل ٤ - ٣٤

که در این حالت نیاز دارید جدول را حذف و به جای آن یک ویو با همین نام ایجاد نمائید.(این عمل را می توانید درون متد Seed کلاس Configuration بنویسد تا به صورت خودکار در هر بروز رسانی انجام شود و نیاز به انجام دستی آن نداشته باشید)

حال دستورات زیر را در متد Main نوشته و برنامه را اجرا کنید:

```
using (var context = new PersonContext())
            {

                var ppt = context.PersonPersonTypes.ToList();
                foreach (var item in ppt)
                {
                    Console.WriteLine("Person name : {0},{1} \r\nPerson type :
{2}",item.FirstName ,item.LastName,item.PersonType);
                }
                Console.ReadKey();
            }
```

خروجی حاصل از اجرای دستورات فوق به صورت زیر خواهد بود:



شکل ٤ - ٣٥

نکته: اگرچه ممکن است نیاز چندانی به وجود View ها نباشد و بتوان با دستورات EF درخواست های مورد نیاز را مرتفع کرد اما دانستن روش کار با ویو ها ضروری به نظر می رسد چرا که ممکن است بخواهید از پایگاه داده ای که از قبل طراحی شده است و دارای ویو های مختلفی می باشد استفاده نمائید.

استفاده از دستورات Sql در EF

کتابخانه EF این اجازه را به شما می دهد که علاوه بر دسترسی به داده ها توسط موجودیت ها که موجب ایجاد کوئری توسط EF می شود، خودتان نیز بتوانید دستورات Sql را به صورت مستقیم اجرا کنید. برای این منظور از متد SqlQuery خاصیت Database شی Context استفاده خواهیم کرد.

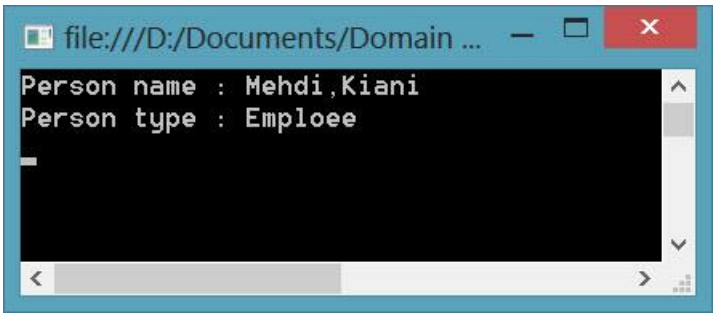
به دستورات زیر توجه کنید:

```
String sql = @"select * from PersonPersonTypeView where FirstName ={0}";
                var persons = context.Database.SqlQuery<PersonPersonTypeView>(sql,
"Mehdi");
                foreach (var item in persons)
                {
                    Console.WriteLine("Person name : {0},{1} \r\nPerson type : {2}",
item.FirstName, item.LastName, item.PersonType);
                }
```

دستورات فوق یک دستور sql را با استفاده از متد ژنریک SqlQuery اجرا می کند. متد SqlQuery دو پاراتر دریافت کرده است. پارامتر اول رشته sql و پارامتر دوم مقادیری است که درهنگام اجرای دستور درون این رشته

قرار خواهند گرفت. همانور که مشاهده می کنید از روش پارامتری در دستور sql استفاده شده است. این روش برای جلوگیری از Sql Injection مورد استفاده قرار می گیرد.

خروجی این دستور لیستی PersonPersonTypeView خواهد بود که خروجی آن در شکل زیر نشان داده شده است:



شکل ٤ - ٣٦

استفاده از متد Sql Query برای Stored Procedure ها

همانند دستورات Sql که توسط SqlQuery قابل اجرا بودند می توان Stored Procedure را نیز توسط این متد اجرا کرد. برای این منظور یک Stored Procedure به صورت زیر در پایگاه داده ایجاد کنید:

```
CREATE PROCEDURE SelectPersonFromView

AS
BEGIN

	SET NOCOUNT ON;
SELECT * FROM PersonPersonTypeView AS pptv
END
GO
```

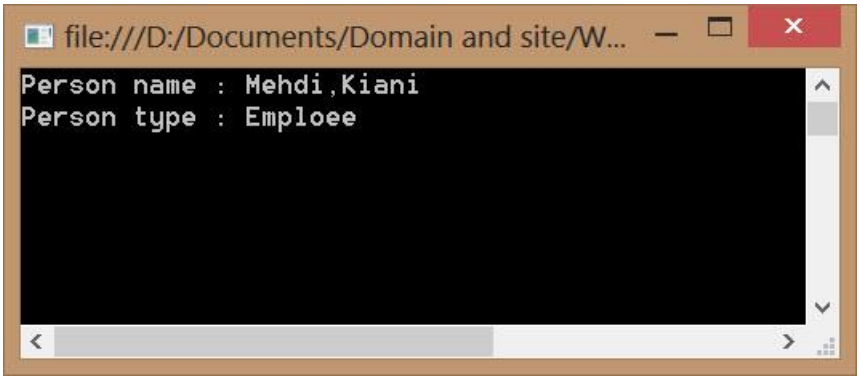
دستورات فوق یک Stored Procedure به نام SelectPersonFromView ایجاد می کند. همانظور که مشخص است این پروسیجر رکورد های ویوی PersonPersonTypeView را واکشی می کند.

حال دستورات زیر را درمتد Main نوشته و برنامه را اجرا کنید:

```
String sql = @"SelectPersonFromView";
```

```
                var persons = context.Database.SqlQuery<PersonPersonTypeView>(sql);
                foreach (var item in persons)
                {
                    Console.WriteLine("Person name : {0},{1} \r\nPerson type : {2}",
item.FirstName, item.LastName, item.PersonType);
                }
```

خروجی برنامه به صورت زیر خوهد بود:



شکل ٤ - ٣٧

همانطور که مشاهده می کنید از نام پروسیجر به جای دستورات sql استفاده شده است و مابقی دستورات شبیه به دستورات قبلی می باشد. در حالت استفاده از پروسیجر نیز همانند دستورات sql می توانید پارامتر های مختلفی را به پروسیجر ارسال نمائید که روش آن مشابه ارسال پارامتر به دستورات sql می باشد که در بخش قبلی نمونه ان را دیدید.

اگر بخواهیم پروسیجری که عملیات درج،ویرایش و یا حذف انجام می دهد را توسط EF اجرا کنیم می بایستی به جای متد SqlQuery از متد ExecuteSqlCommand استفاده نمائیم.

یک پروسیجر به نام InsertPerson به صورت زیر در پایگاه داده ایجاد کنید:

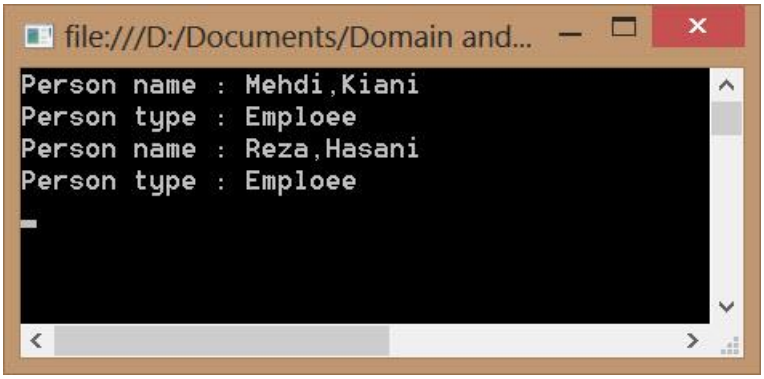
```
CREATE PROCEDURE InsertPerson
       @FirstName NVARCHAR(50) ,
       @LastName NVARCHAR(50),
       @PersonTypeId INT
AS
BEGIN
       SET NOCOUNT ON;
       INSERT INTO Persons
       VALUES
```

```
        (
          @FirstName,
          @lastName,
          @PersonTypeId
        )
END
GO
```

حال دستورات زیر را در متد Main نشوته و برنامه را اجرا کنید:

```
String sql = @"InsertPerson {0},{1},{2}";
                context.Database.ExecuteSqlCommand(sql, "Reza", "Hasani", 1);

                sql = @"SelectPersonFromView";
                var persons = context.Database.SqlQuery<PersonPersonTypeView>(sql);
                foreach (var item in persons)
                {
                    Console.WriteLine("Person name : {0},{1} \r\nPerson type : {2}",
item.FirstName, item.LastName, item.PersonType);
                }
```

دو خط ابتدایی دستورات فوق پروسیجر InsertPerson را با سه پارامتر FirstName، LastName و PersonTypeId اجرا می کند. این اجرا توسط متد ExecuteSqlCommand اتفاق می افتد. دستورات بعدی نیز توسط پروسیجر SelectPersonFromView رکورد های موجود در ویو را پس از درج رکورد جدید نمایش می دهد:



شکل ٤ - ٣٨

نگاشت پروسیجر ها

همانطور که در مثال های پیشین در طول آموزش اشاره شد زمانی که یک نمونه از یک موجودیت ایجاد می شود و توسط متد SaveChanges مربوط به شی Context در پایگاه داده ذخیره می شود مکانیزمی که EF در این مسیر طی می کند این است که دستور SQL متناظر با عملیات درخواستی را ایجاد و به همراه داده های کاربر که در نمونه ای از موجودیت قرار دارند به سمت پایگاه داده ارسال کرده و اجرا می کند.

ممکن است شما به هر دلیلی نخواهید از این مکانیزم استفاده کنید و بخواهید کنترل دستورات SQL را نیز خودتان بر عهده بگیرید. معماری EF به شما این امکان را می دهد تا موجودیت شما را به پروسیجر هایی در پایگاه داده نگاشت کند. برای درک بهتر این موضوع یک کلاس به نام Company به صورت زیر به پروژه اضافه کنید:

```
   [Table("Companies")]
   public class Company
    {
        [Key]
        public Int32 CompanyId
        {
            get;
            set;
        }
        public String Name
        {
            get;
            set;
        }
    }
```

به جهت سادگی دو فیلد CompanyId به عنوان کلید اصلی و فیلد Name به جهت نام کمپانی در نظر گرفته شده است.

حال یک کلاس به نام CompanyConfig به صورت زیر به پروژه اضافه کنید:

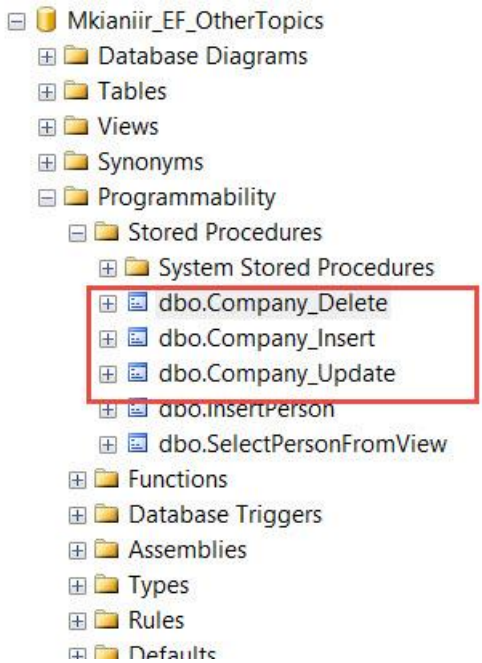
```
public class CompanyConfig :EntityTypeConfiguration<Company>
    {
       public CompanyConfig()
       {
           MapToStoredProcedures();
       }
    }
```

همانطور که مشاهده می کنید متد MapToStoredProcedure تنها متدی است که از کتابخانه Fluent برای پیکربندی موجودیت Company استفاده شده است.

حال توسط دستور زیر کلاس CompanyConfig را به مجموعه پیکربندی های EF اضافه کنید.کلاس PersonContext را باز کنید و دستور زیر را در متد OnModelCreating بنویسد:

```
modelBuilder.Configurations.Add(new CompanyConfig());
```

حال توسط دستور Update-Database از کتابخانه Migration پایگاه داده را بروز رسانی کنید:

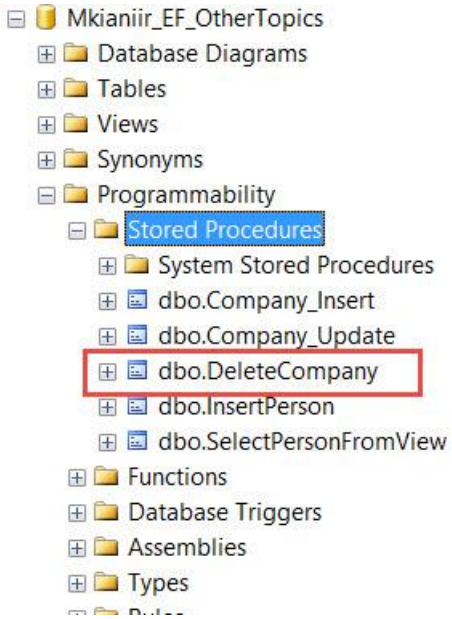


شکل ٤ - ٣٩

همانطور که مشاهده می کنید سه پروسیجر به نام های Company_Delete، Company_Insert و Company_Update در پایگاه داده ایجاد شده است.

متد MapToStoredProcedure دارای حالت دیگری نیز می باشد که به شما اجازه پیکربندی دقیق تری را نیز می دهد. به عنوان مثال پیکر بندی زیر نام پروسیجر مربوط به حذف را از Company_Delete به DeleteCompany اغییر می دهد:

```
public class CompanyConfig :EntityTypeConfiguration<Company>
    {
        public CompanyConfig()
        {
            MapToStoredProcedures(sp => sp.Delete(spd=>spd.HasName("DeleteCompany")));
        }
    }
```



شکل ٤ - ٤٠

خلاصه

در فصل پایانی کتاب سایر مفاهیم عمومی مرتبط به کتابخانه Entity Framework و پیاده سازی آن مورد بررسی قرار گرفت. در این بخش شما توانستید انواع پیچیده و نیز انواع داده شمارشی را پیاده سازی نمائید.همچنین یکی از مفاهیم مهمی که همیشه در مورد Entity Framework محل بحث بوده که توسط Entity Framework Migration مرتفع شد تغییر در شمای پایگاه داده می باشد که در این فصل آموختید چگونه توسط Migration و دستورات و کتابخانه آن می توانید بدون نگرانی در خصوص از بین رفتن داده های موجود در پایگاه داده اقدام به تغییر و بروز رسانی پایگاه داده نمائید.

سخن آخر

در پایان امیدوارم تمامی مباحثی که در این کتاب بیان شد بتواند راهگشای شما در استفاده از کتابخانه Entity Framework در پروژه های شما باشد. نکته مهمی که باید به ان توجه کنید و قبلا نیز به ان اشاره کردم این است که این کتاب صرفا به عنوان یک راهنمای اولیه جهت حرکت در این مسیر می باشد. به بسیاری از جزئیات در این کتاب پرداخته نشده است چرا که هدف از این کتاب صرفا ایجاد جرقه ای در بین برنامه نویسانی است که دوست دارند از روش های قدیمی تر کار با پایگاه های داده به روش های جدید تری تغییر دیدگاه دهند. اینکه از این تکنولوژی استفاده نمائید یا خیز کاملا بستگی به شما، توانایی شما، شرایط پروژه و بسیاری ملزومات دیگری است که نمی توان نسخه ای واحد برای همه موقعیت ها نوشت. اما یادگیری هر چیزی دارای لذتی است که امیدوارم از خواندن مطالب این کتاب لذت کافی و وافی را برده باشد.

منابع

۱- Code-First development with entity framework,Sergey Barskiy,Packt Publishing
۲- Entity Framework ۶ Recipes,Brain Driscoll,Nitin Gupta,Robert Vettor,Zeeshan Hirani,Larry Tenny,Apress Publishing
۳- Pro Entity Framework,Scott Keilen,Apress Publishing
۴- Entity Framework Documentation