



P r o f e s s i o n a l E x p e r t i s e D i s t i l l e d

Mastering Entity Framework

Effortlessly produce data-driven applications for .NET to address the competing demands of data storage and data modeling with Entity Framework

Rahul Rajat Singh

[PACKT] enterprise 
PUBLISHING professional expertise distilled

www.it-ebooks.info

Mastering Entity Framework

Effortlessly produce data-driven applications for .NET to address the competing demands of data storage and data modeling with Entity Framework

Rahul Rajat Singh



BIRMINGHAM - MUMBAI

Mastering Entity Framework

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2015

Production reference: 1190215

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-100-3

www.packtpub.com

Credits

Author

Rahul Rajat Singh

Reviewers

Abhishek Luv

Jason De Oliveira

Zsolt Soczo

Commissioning Editor

Amarabha Banerjee

Acquisition Editors

Richard Gall

Gregy Wild

Content Development Editor

Shubhangi Dhamgaye

Technical Editors

Mrunal M. Chavan

Shiny Poojary

Ryan Kochery

Copy Editors

Rashmi Sawant

Stuti Srivastava

Ashwati Thampi

Project Coordinator

Harshal Ved

Proofreaders

Paul Hindle

Stephen Silk

Indexer

Tejal Soni

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Author

Rahul Rajat Singh has over 9 years of experience in developing software applications and websites. After completing his bachelor's degree in computer engineering in 2005, he started working with Tata Consultancy Services (TCS), where he worked on developing audio/video playback engines using C++ for a multimedia company. During this period, he also developed a few small applications using C# and that was the beginning of his love for C#.

After working with TCS for more than 4 years, he took up an independent software assignment and worked on developing a Windows Forms application using C#. During this period, he also started lecturing on programming languages for engineering students. This teaching assignment encouraged him to share his knowledge through more channels, and he started sharing articles on www.codeproject.com.

He started working on web applications when he joined MPOnline Limited. MPOnline Limited, being the best e-governance portal of the country, gave him immense opportunities to learn and implement a lot of web technologies, such as ASP.NET, ADO.NET, JavaScript, SQL Server, Web services, and WCF services.

Currently, he is working with a Danish company called Headfitted Solutions as a technical architect, where he develops applications using ASP.NET MVC, Entity Framework, scalable REST-based services using WCF and ASP.NET Web API, and Single-Page Applications (SPAs) using Backbone.js. He is currently learning more about software architecture and enterprise architecture.

Rahul loves software development, and his interests involve system programming, website development, and learning/teaching subjects related to computer science / information systems. He was also awarded CodeProject MVP in 2013 and DZone MVB in 2014. He blogs regularly about technology on his blog (www.rahulrajatsingh.com).

I would like to thank my wife and daughter for their love and support and being patient with the long working hours that enabled me to write this book. I would like to thank my parents for everything they have done for me. Everything I am today is because of them. Finally, I would like to thank a couple of my mentors Mr. Himanshu Agnihotri and Mr. Amalraj P for supporting me during my struggle period and showing me the right path.

About the Reviewers

Abhishek Luv has been developing and designing websites and web applications for the last 2 years. During these years, Abhishek has been involved in technologies such as C#, ASP.NET, ASP.NET MVC, Visual Studio, and Entity Framework, and now he is mostly working on Microsoft stack.

He is currently working to become an Orchard CMS expert; that aside, he is presently a software engineer at Develop2Deploy (<http://www.develop2deploy.com/>).

He is a founder of the Orchard CMS India Community website (<http://www.orchardproject.net.in/>) and a contributor to the official Orchard CMS documentation website and has created numerous online courses on Orchard CMS.

Outside of the normal day-to-day activities, he is also a cofounder of a video training start-up (<http://www.thevideotrainer.in/>). You can reach him at abhishek@abhishekluv.in.

Jason De Oliveira works as the CTO for Cellenza (<http://www.cellenza.com>), an IT consulting company specializing in Microsoft technologies and Agile methodology in Paris, France. He is an experienced manager and senior solutions architect, with advanced skills in software architecture and enterprise architecture.

Jason works for big companies and helps them to realize complex and challenging software projects. He frequently collaborates with Microsoft, and you can find him quite often at the Microsoft Technology Center (MTC) in Paris.

He loves sharing his knowledge and experience via his blog, by speaking at conferences, writing technical books, writing articles in the technical press, giving software courses as an MTC, and coaching coworkers in his company.

Microsoft awarded him with the Microsoft Most Valuable Professional (MVP C#) Award in 2011 for his numerous contributions to the Microsoft community. Microsoft seeks to recognize the best and brightest from technology communities around the world with the MVP Award. These exceptional and highly respected individuals come from more than 90 countries, serving their local online and offline communities and have an impact worldwide. Jason is very proud to be one of them.

Please feel free to contact him via his blog if you need any technical assistance or want to exchange information on technical subjects (<http://www.jasondeoliveira.com>).

Jason has worked on the following books:

- *.NET Framework 4.5 Expert Programming Cookbook* (English), Packt Publishing
- *WCF 4.5 Multi-Layer Services Development with LINQ to Entities* (English), Packt Publishing
- *.NET 4.5 Parallel Extensions Cookbook* (English), Packt Publishing
- *WCF Multi-layer Services Development with Entity Framework* (English), Packt Publishing
- *Visual Studio 2013: Concevoir, développer et gérer des projets Web, les gérer avec TFS 2013* (French), ENI

I would like to thank my lovely wife, Orianne, and my beautiful daughters, Julia and Léonie, for supporting me in my work and accepting long days and short nights during the weeks and sometimes even during the weekends. My life would not be the same without them!

Zsolt Soczo received his MSc degree in electrical engineering and currently works as an independent software developer, a mentor, and a consultant for enterprise companies. Since 2000, he has been involved in .NET- and SQL-Server-based development. Zsolt has a passion for solving complex problems by applying his extensive .NET and SQL Server skills. He enjoys optimizing slow databases and architecting large enterprise systems. In his spare time, he loves to develop and execute algorithmic trading strategies on financial markets.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Instant updates on new Packt books

Get notified! Find out when new books are published by following @PacktEnterprise on Twitter or the *Packt Enterprise* Facebook page.

Table of Contents

Preface	1
Chapter 1: Introduction to Entity Framework	7
Entity Framework modeling and persistence	8
Understanding the Entity Data Model	9
Understanding theObjectContext class	9
Development styles and different Entity Framework approaches	10
Comparing the development styles	11
The Database First approach	11
The Model First approach	12
The Code First approach	12
Entity Framework Database First approach	12
Entity Framework Model First approach	22
Entity Framework Code First approach	30
Performing CRUD operations using Entity Framework	32
Reading a list of items	32
Reading a specific item	33
Creating a new item	34
Updating an existing item	35
Deleting an item	37
Choosing persistence approaches	38
Summary	38
Chapter 2: Entity Framework DB First – Managing Entity Relationships	39
Understanding database relationships	39
One-to-many relationship	40
One-to-one relationship	40
Many-to-many relationship	41
Creating the Entity model	43

Modeling a one-to-many relationship	44
Modeling a one-to-one relationship using Entity Framework	47
Modeling a many-to-many relationship using Entity Framework	49
Using navigation properties for data access	51
Retrieving a specific item	51
Retrieving a list of items	52
Adding an item	53
Updating an item	54
Deleting an item	55
Entity Framework – behind the scenes	56
Summary	57
Chapter 3: Entity Framework DB First – Performing Model Validations	59
Model validations using Entity Framework	59
Model validations using partial class methods	60
Understanding partial methods	60
Using partial methods to perform model validations	62
Model validations using data annotations	66
Specifying validation rules using data annotations	67
Validating the required fields	67
Validating the length of fields	68
Regular expression-based validations	69
Triggering validations using data annotations	70
Trigger validations in data binding environments	70
Trigger validations in non-data binding environments	70
Implementing custom validations using data annotations	72
Summary	74
Chapter 4: Entity Framework DB First – Inheritance Relationships between Entities	75
Domain modeling using inheritance in Entity Framework	75
The Table per Type inheritance	76
Generating the default Entity Data Model	77
Deleting default relationships	78
Adding inheritance relationships between entities	79
Using the entities via the DbContext object	81
The Table per Class Hierarchy inheritance	82
Generating the default Entity Data Model	84
Adding concrete classes to the Entity Data Model	84
Mapping the concrete class properties to the respective tables and columns	87
Making the base class entity abstract	90

Using the entities via the DbContext object	90
The Table per Concrete Class inheritance	92
Generating the default Entity Data Model	93
Creating the abstract class	94
Modifying the CDSL to cater to the change	95
Specifying the mapping to implement the TPT inheritance	96
Using the entities via the DbContext object	98
Choosing the inheritance strategy	99
Summary	100
Chapter 5: Entity Framework DB First – Using Views, Stored Procedures, and Functions	101
Using views, procedures, and functions	101
Using Entity Framework with views	102
Using Entity Framework with stored procedures	106
Defining stored procedures	106
Using Entity Framework with functions	112
Using scalar functions	112
Using table valued functions	114
Summary	117
Chapter 6: Entity Framework Code First – Domain Modeling and Managing Entity Relationships	119
Understanding the Entity Framework Code First approach	120
Understanding the Code First conventions and configurations	121
Implementing Entity Framework Code First	121
More on domain class configurations	124
Managing Entity relationships using the Code First approach	126
Implementing one-to-many relationships	126
Implementing one-to-one relationships	131
Implementing many-to-many relationships	135
Inheritance with the Entity Framework Code First approach	140
Implementing the TPT inheritance	140
Implementing the TPH inheritance	145
Implementing the TPC inheritance	150
Summary	154
Chapter 7: Entity Framework Code First – Managing Database Creation and Seeding Data	155
Managing database connections	155
Managing connections using a configuration file	156
Using the existing ConnectionString	158
Using an existing connection	158

Managing database initialization	159
Setting the initialization strategy	160
Seeding data	161
Summary	165
Chapter 8: Querying the Entity Data Model – LINQ to Entities	167
Understanding LINQ to Entities	167
Querying data using LINQ to Entities	169
Using LINQ to Entities – an example-based approach	172
Executing simple queries	173
Using the navigation properties with LINQ to Entities	174
Filtering data using LINQ to Entities	175
Using LINQ projections with LINQ to Entities	177
Grouping using LINQ to Entities	178
Ordering using LINQ to Entities	179
Aggregate operators with LINQ to Entities	181
Count	181
Sum	183
Min	183
Max	184
Average	185
Partitioning/paging data using LINQ to Entities	185
Skip	186
Take	187
Implementing paging	187
Implementing join using LINQ to Entities	188
Lazy loading and eager loading	189
Lazy loading	189
Eager loading	192
Summary	192
Chapter 9: Querying the Object Model – Entity SQL	193
Understanding Entity SQL	193
Understanding EntityConnection	194
Entity SQL with ObjectQuery	196
Querying data using Entity SQL with ObjectQuery	198
Executing parameterized Entity SQL with ObjectQuery	199
Navigation properties using Entity SQL with ObjectQuery	201
Aggregate functions with Entity SQL using ObjectQuery	202
Count	203
Sum	203
Min	204
Max	205
Average	205

Ordering data with Entity SQL using ObjectQuery	206
Grouping data using Entity SQL with ObjectQuery	207
Partitioning/paging data using Entity SQL ObjectQuery	208
Skip	208
Limit	209
Implementing paging	209
Using Entity SQL with EntityCommand	210
Querying data using Entity SQL with EntityCommand	210
Parameterized Entity SQL with EntityCommand	211
Summary	213
Chapter 10: Managing Concurrency Using Entity Framework	215
Understanding concurrency	215
Understanding optimistic concurrency	217
Ignore the conflict/forcing updates	217
Partial updates	217
Warn/ask the user	217
Reject the changes	217
Understanding pessimistic concurrency	218
Implementing optimistic concurrency using Entity Framework	219
Entity Framework's default concurrency	219
Designing applications to handle field level concurrency	222
Implementing field level concurrency	226
Implementing RowVersion for concurrency	229
Entity Framework and pessimistic concurrency	231
Summary	232
Chapter 11: Managing Transactions Using Entity Framework	233
Understanding transactions	233
Setting up the test environment	234
Entity Framework's default transaction handling	235
Using TransactionScope to handle transactions	237
Managing transactions using Entity Framework 6	238
Using an existing transaction	239
Choosing the appropriate transaction management	241
Summary	242
Chapter 12: Implementing a Small Blogging Platform Using Entity Framework	243
Understanding the application requirements	243
Visualizing our database design	244
Creating the Entity Data Model	245
Creating the entity classes	246
The User entity	246

Table of Contents

The Role entity	246
The Category entity	247
The Blog entity	247
The Comment entity	248
Creating relationships and navigation properties	248
The User entity	249
The Role entity	250
The Category entity	250
The Blog entity	251
The Comment entity	252
Implementing the DbContext class	253
Performing data access	255
Understanding the Repository pattern	255
Understanding Unit of Work	259
Managing categories	261
Listing categories	262
Adding a category	262
Updating a category	264
Deleting a category	265
Managing blogs	266
Adding a new blog	266
Updating a blog	268
Deleting a blog	269
Listing blogs on the home page	270
Showing a single blog	273
Managing comments	275
Listing categories	275
Adding a comment	277
Deleting a comment	278
Using other Entity Framework approaches	279
Summary	280
Index	281

Preface

Software developers using .NET technologies have been using ADO.NET for data access for more than a decade. It was with the release of .NET Framework 3.5 that Microsoft provided one more data access technology – Entity Framework. Entity Framework is an Object Relational Mapper (ORM) that sits on top of ADO.NET. Entity Framework lets the developer write the data access code in terms of models rather than SQL queries, which makes creating the data access layer much simpler and easier.

This book is for the .NET developers who develop data-driven applications using ADO.NET or other data access technologies. This book will give them everything that is required to effectively develop and manage data-driven applications using Entity Framework. We will learn the various approaches that Entity Framework provides us and which approach should be used for which scenario. We will learn how to perform domain modeling, validations, transaction and concurrency handling using Entity Framework.

We will also take a look at the various ways of querying the data using Entity Framework. We will see how we can reuse database procedures using Entity Framework. We will also take a look at passing dynamic queries from the application using Entity Framework. This book will help developers build all the skills required to effectively develop and manage data-driven applications using Entity Framework.

What this book covers

Chapter 1, Introduction to Entity Framework, introduces the user to Entity Framework. We will show how a data-centric application can benefit from the Database First approach and a domain-centric application can benefit from a Model First or Code First approach.

Chapter 2, Entity Framework DB First – Managing Entity Relationships, discusses how relationships can be managed using Entity Framework (one-to-one, one-to-many, and many-to-many).

Chapter 3, Entity Framework DB First – Performing Model Validations, discusses how model validations can be performed using the Entity Framework provided partial methods.

Chapter 4, Entity Framework DB First – Inheritance Relationships between Entities, discusses the Table per Type, Table per Hierarchy, and Table per Concrete Class inheritance relationships in Entity Framework.

Chapter 5, Entity Framework DB First – Using Views, Stored Procedures, and Functions, discusses how we can use Entity Framework with stored procedures and functions.

Chapter 6, Entity Framework Code First – Domain Modeling and Managing Entity Relationships, discusses how we can map the domain entities to the database tables. We will learn how to manage the relationships between models by using the Code First conventions and configurations.

Chapter 7, Entity Framework Code First – Managing Database Creation and Seeding Data, discusses how we can manage the database creation process and insert some dummy data into the database.

Chapter 8, Querying the Entity Data Model – LINQ to Entities, discusses how we can query the object model using LINQ to Entities.

Chapter 9, Querying the Object Model – Entity SQL, discusses how we can query the object model using Entity SQL.

Chapter 10, Managing Concurrency Using Entity Framework, covers various concurrency management techniques using Entity Framework.

Chapter 11, Managing Transactions Using Entity Framework, discusses how to manage transactions using Entity Framework.

Chapter 12, Implementing a Small Blogging Platform Using Entity Framework, introduces a complete sample application using the Entity Framework Code First approach to demonstrate all the concepts in action.

What you need for this book

The following software is required to complete the practice exercises:

- Windows 7 or higher
- SQL Server Developer Edition or Express edition
- Visual Studio 2010 or higher
- .NET Framework 4.0 or higher
- Entity Framework 4.0 or higher

Who this book is for

This book is for .NET developers who develop data-driven applications using ADO.NET or other data access technologies. This book will give you everything you need to effectively develop and manage data-driven applications using Entity Framework.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

"Using `IQueryable`, gives you a chance to create a complex LINQ query using multiple statements without executing the query at the database level."

A block of code is set as follows:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    IEnumerable<Employee> employees = db.Employees
        .Where(employee => employee.LastName == "Singh");
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
public partial class Role
{
    public Role()
    {
```


```
        Users = new HashSet<User>();
    }


    public int Id { get; set; }

    [Required]
    [StringLength(256)]
    public string Name { get; set; }

    public virtual ICollection<User> Users { get; set; }
}
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Using this approach will check for concurrency issues for only those fields that are marked with **Concurrency Mode** as **Fixed**."

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Introduction to Entity Framework

Entity Framework is an **Object Relational Mapper (ORM)** from Microsoft that lets the application's developers work with relational data as business models. It eliminates the need for most of the plumbing code that developers write (while using ADO.NET) for data access. Entity Framework provides a comprehensive, model-based system that makes the creation of a data access layer very easy for the developers by freeing them from writing similar data access code for all the domain models. The initial release of Entity Framework was Entity Framework 3.5. It was released with .NET Framework 3.5 SP1 and Visual Studio 2008 SP1. Entity Framework has evolved a lot since then, and the current version is 6.0.

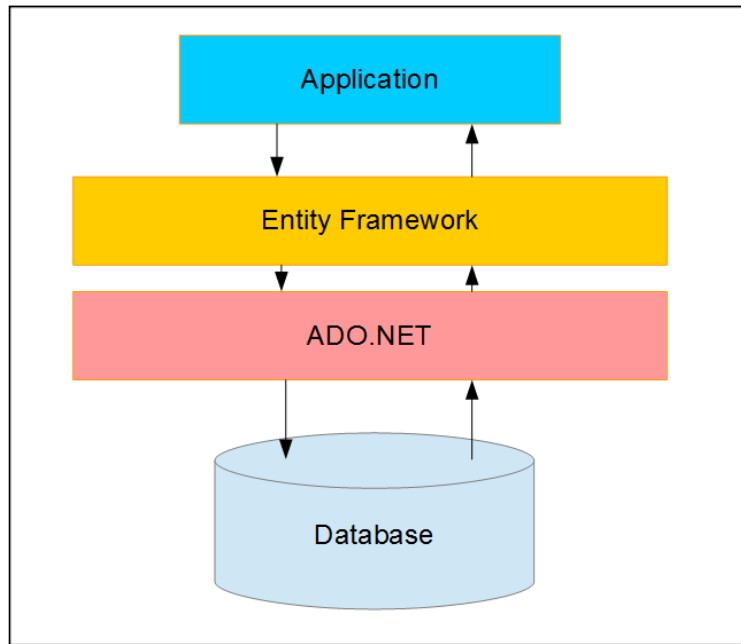
Entity Framework eases the task of creating a data access layer by enabling the access of data, by representing the data as a conceptual model, that is, a set of entities and relationships. The application can perform the basic CRUD (create, read, update, and delete) operations and easily manage one-to-one, one-to-many, and many-to-many relationships between the entities.

Here are a few benefits of using Entity Framework:

- The development time is reduced since the developers don't have to write all the ADO.NET plumbing code needed for data access
- We can have all the data access logic written in a higher-level language such as C# rather than writing SQL queries and stored procedures
- Since the database tables cannot have advanced relationships (inheritance) as the domain entities can, the business model, that is, the conceptual model can be used to suit the application domain using relationships among the entities.

- The underlying data store can be replaced relatively easily if we use an ORM since all the data access logic is present in our application instead of the data layer. If an ORM is not being used, it would be comparatively difficult to do so.

Let's try to visualize the Entity Framework architecture:



The Entity Framework architecture

From the preceding diagram, we can see that Entity Framework is written on top of the ADO.NET framework, and underneath, it is still using the ADO.NET methods and classes to perform the data operations.

Entity Framework modeling and persistence

Entity Framework relies on the conceptual data model for all its working. Let's try to understand what the **Entity Data Model (EDM)** is and how Entity Framework uses it to manage the database operations.

Understanding the Entity Data Model

The conceptual data model is the heart of Entity Framework. To use Entity Framework, we have to create the conceptual data model, that is, the EDM. The EDM defines our conceptual model classes, the relationships between those classes, and the mapping of those models to the database schema.

Once our EDM is created, we can perform all the CRUD operations (create, retrieve, update, and delete) on our conceptual model, and Entity Framework will translate all these object queries to database queries (SQL). Once these queries are executed, the results will again be converted to conceptual model object instances by Entity Framework. Entity Framework will use the mapping information stored in the EDM to perform this translation of object queries to SQL queries, and the relational data to conceptual models.

Once our EDM is ready, we can perform the CRUD operations using the model objects. To be able to perform the CRUD operations, we have to use the `ObjectContext` class. Let's try to understand what the `ObjectContext` class is and how it can be used to perform the CRUD operations.

Understanding the ObjectContext class

Once I have my EDM created, I will have all the entities that I can use in my application. However, I still need a central arbitrator that will let me perform various operations on these entities. This arbitrator in Entity Framework is the `ObjectContext` class.

The `ObjectContext` class is the main object in Entity Framework. It is the class that is responsible for:

- Managing database connection
- Providing support to perform CRUD operations
- Keeping track of changes in the models so that the models can be updated in the database

The `ObjectContext` class can be thought of as an orchestrator that will manage all the entities in the EDM, and lets us perform all of the database operations for these entities.



There is another class, `DbContext`, that is very similar to the `ObjectContext` class. In fact, the `DbContext` class is just a wrapper on top of the `ObjectContext` class. It is a rather newer API, and it provides a better API to manage database connections and perform CRUD operations.

Since `DbContext` is a better API, we will be using `DbContext` to perform all the database operations.

The `ObjectContext` class has a `SaveChanges` method that we have to call when we want to save the new or changed objects to the database.

Development styles and different Entity Framework approaches

Before we start looking at the various options to create the EDM, let's talk about the development styles followed by the developers. Some organizations have separate teams working on the application and the database. In such cases, the database design is done first and then the application development starts. Another reason for doing the database design first is when the application being developed is a data centric application itself.

Instead, it might be possible that the application demands the creation of the conceptual domain models first. Then, based on these conceptual domain models, the database tables will be created, and the application will implement the business logic in terms of these conceptual business models.

Another possibility is that we are creating an application that is highly domain-centric, and the application contains the domain models implemented as classes. The database is needed only to persist these models with all the relations.

There are a range of different scenarios you might find yourself in – different approaches become appropriate according to the demands of your situation.

Entity Framework provides support for all these development styles and scenarios. We know that Entity Framework operates on the EDM, and lets us create this EDM in three ways in order to cater to the different development styles:

- **Database First:** This is the approach that will be used with an existing database schema. In this approach, the EDM will be created from the database schema. This approach is best suited for applications that use an already existing database.

- **Code First:** This is the approach where all the domain models are written in the form of classes. These classes will constitute our EDM. The database schema will be created from these models. This approach is best suited for applications that are highly domain-centric and will have the domain model classes created first. The database here is needed only as a persistence mechanism for these domain models.
- **Model First:** This approach is very similar to the Code First approach, but in this case, we use a visual EDM designer to design our models. The database schema and the classes will be generated by this conceptual model. The model will give us the SQL statements needed to create the database, and we can use it to create our database and connect up with our application. For all practical purposes, it's the same as the Code First approach, but in this approach, we have the visual EDM designer available.

Comparing the development styles

Before we start looking at these development styles in detail, let's try to do a comparative analysis of these styles. It will help us in understanding these styles in detail.

The Database First approach

In a Database First approach, the main benefit that the developers have is that if the database is created, they will spend very little time writing the data access layer. The EDM can be created from the database and then we can change it as per our application needs; our data access layer is ready to use. Here are a few scenarios where the Database First approach is very useful:

- When we are working with a legacy database.
- When we are working in a scenario where the database design is being done by another team of DBAs, and once the database is ready, the application development starts.
- When we are working on a data centric application, that is, the application domain model is the database itself, and it is being changed frequently to cater to new requirements. For instance, when the tables are being updated regularly and new columns are being added to it frequently then we can simply use this approach, and the application code will not break. We simply have to write the code to cater to the newly added columns.

The Model First approach

Similar to the Database First approach, in the Model First approach, we ultimately end up with the EDM. Using this EDM, we can create our conceptual model and the database. The only reason to choose the Model First approach is that we really want to use the Visual Entity Designer. There is no other strong reason to choose this approach over the other two.

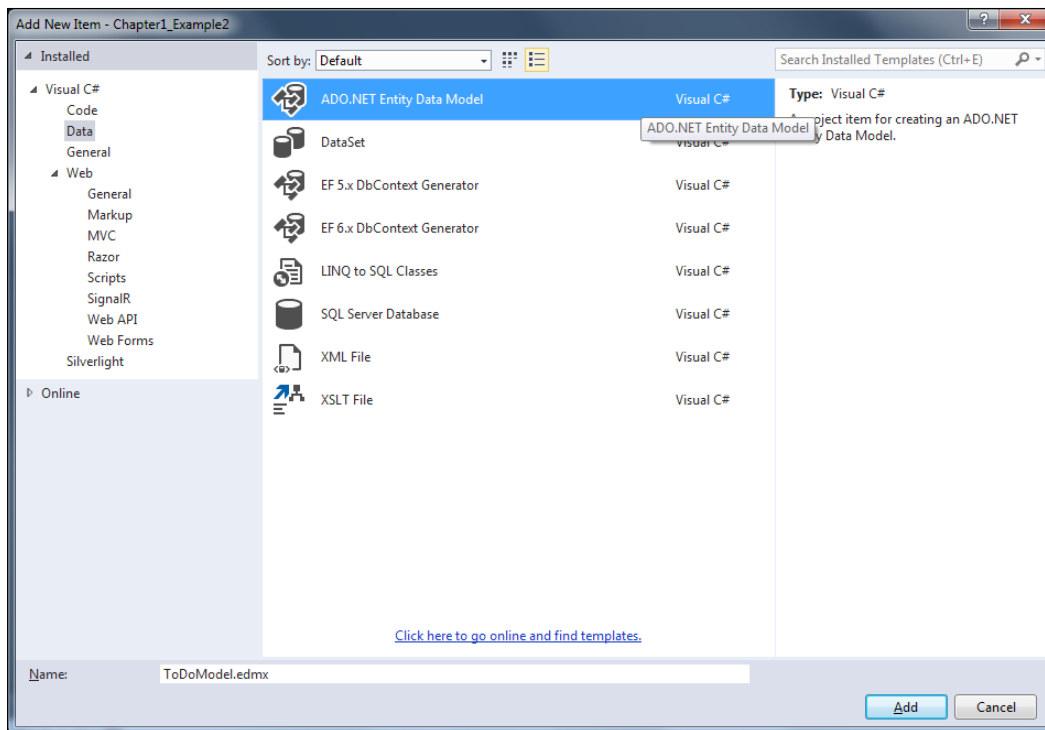
The Code First approach

The Code First approach is usually helpful where all the business logic is implemented in terms of classes, and the database is simply being used as a persistence mechanism for these models. Here are a few reasons why one might want to choose the Code First approach:

- The database is simply a persistence mechanism for the models, that is, no logic is in the database.
- Full control over the code, that is, there is no auto-generated model and context code.
- The database will not be changed manually. The model classes will always change and based on this, the database should change itself.


Entity Framework Database First approach

This approach has been available since the first version of Entity Framework. In this approach, we start with an already existing database and start creating our EDM from the existing database. Let's try to see how this works using the Entity Framework Database First approach with our ToDo sample. To use the Entity Framework Database First approach, we have to add a new ADO.NET EDM to our project:



Add a new ADO.NET EDM

Let's start by defining a database for the ToDo application. Let's create a simple table that will keep a track of all the ToDo items:

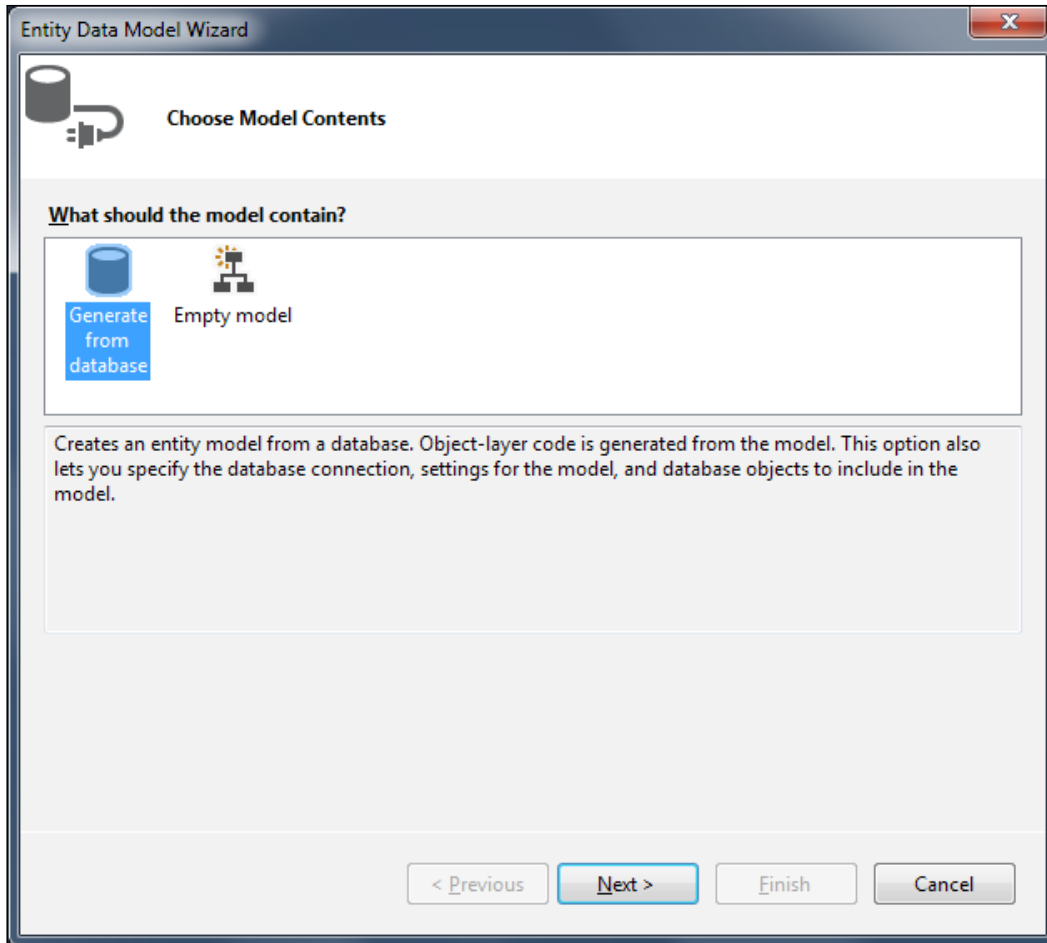
	Name	Data Type	Allow Nulls	Default
	Id	int	<input type="checkbox"/>	
	Todo	nvarchar(MAX)	<input type="checkbox"/>	
	IsDone	bit	<input type="checkbox"/>	0

A table schema for the sample ToDo application

The application requirements state that it should be possible to perform the following activities on the Todos:

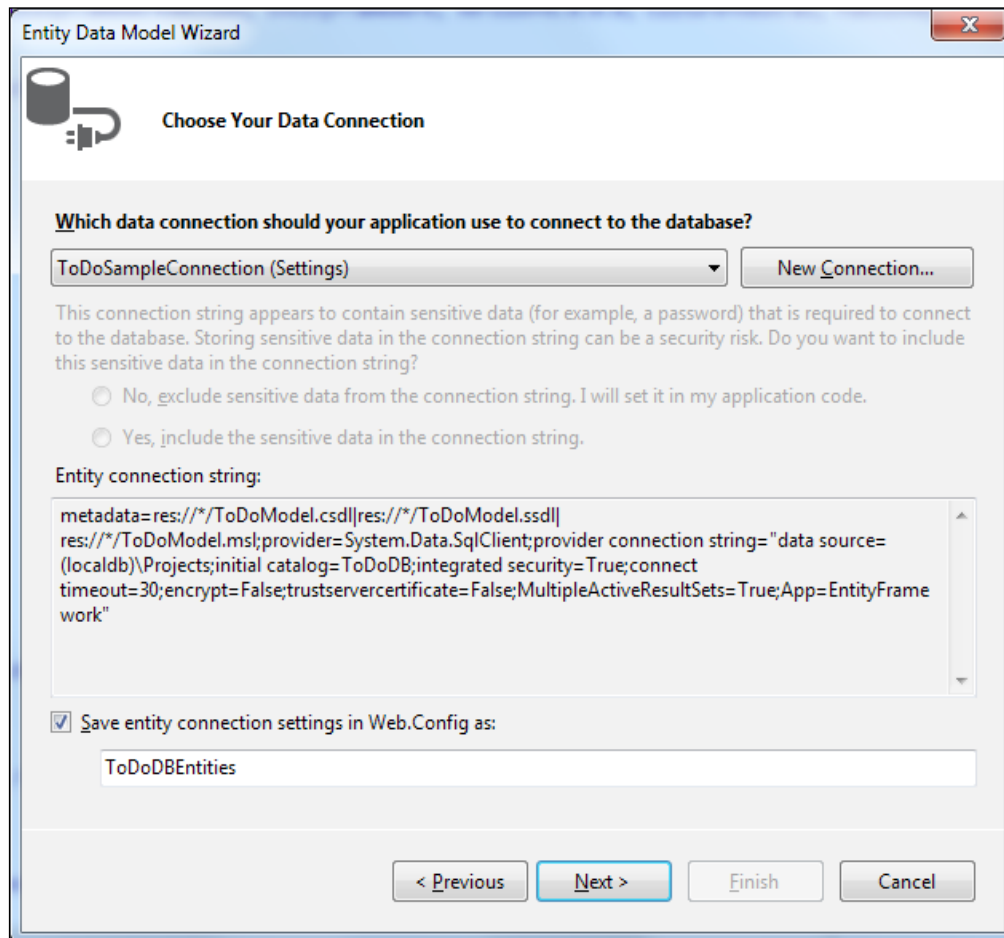
- Read the list of Todos
- Create a ToDo item
- Read a specific ToDo item
- Update the status of a ToDo item
- Delete a ToDo item

When we add the EDM, it asks us whether we want to create the conceptual model from the existing database, or if we want to create an empty model. For the Database First approach, we need to select an existing database. The following screenshot shows the wizard step that asks you to select the database:



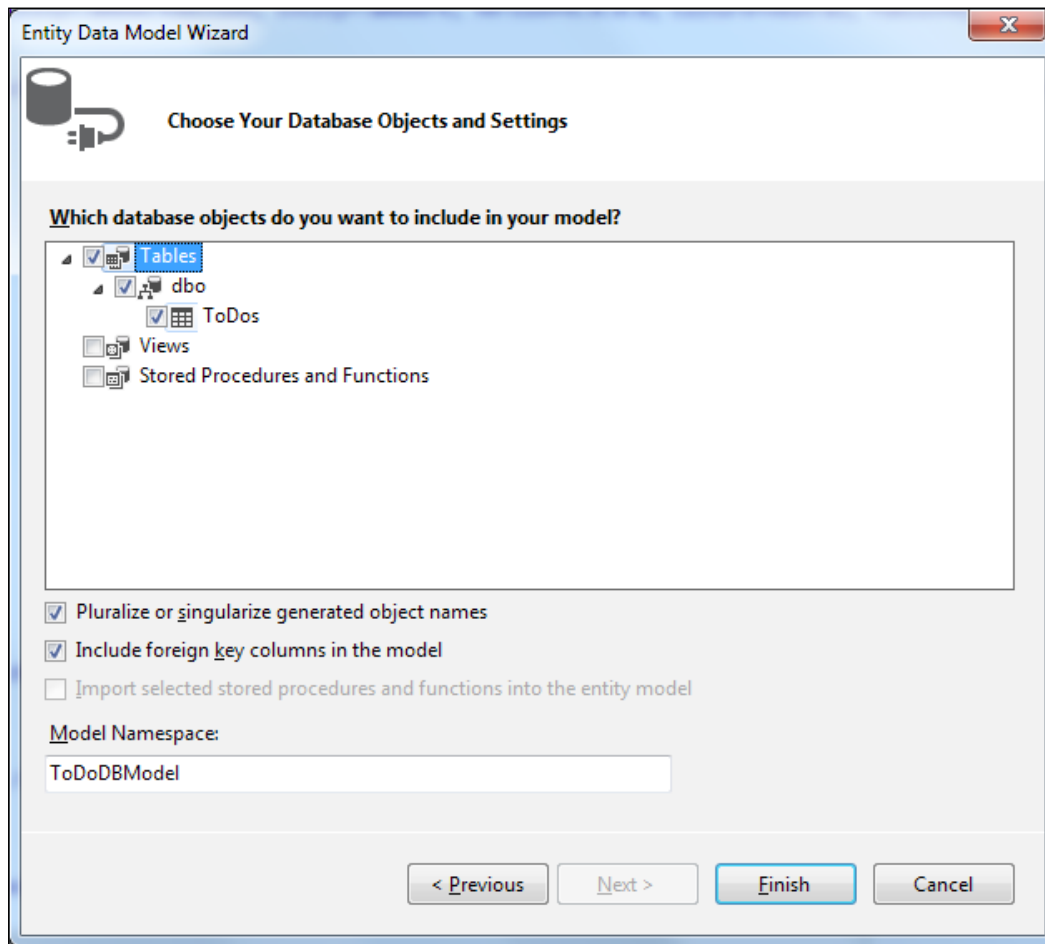
ADO.NET EDM wizard step 1—generate from the database

Once we choose to add the EDM, we need to specify the database that should be used to create it. So, the next step in the wizard will ask you for the connection to the database. Let's see what this wizard step looks like:



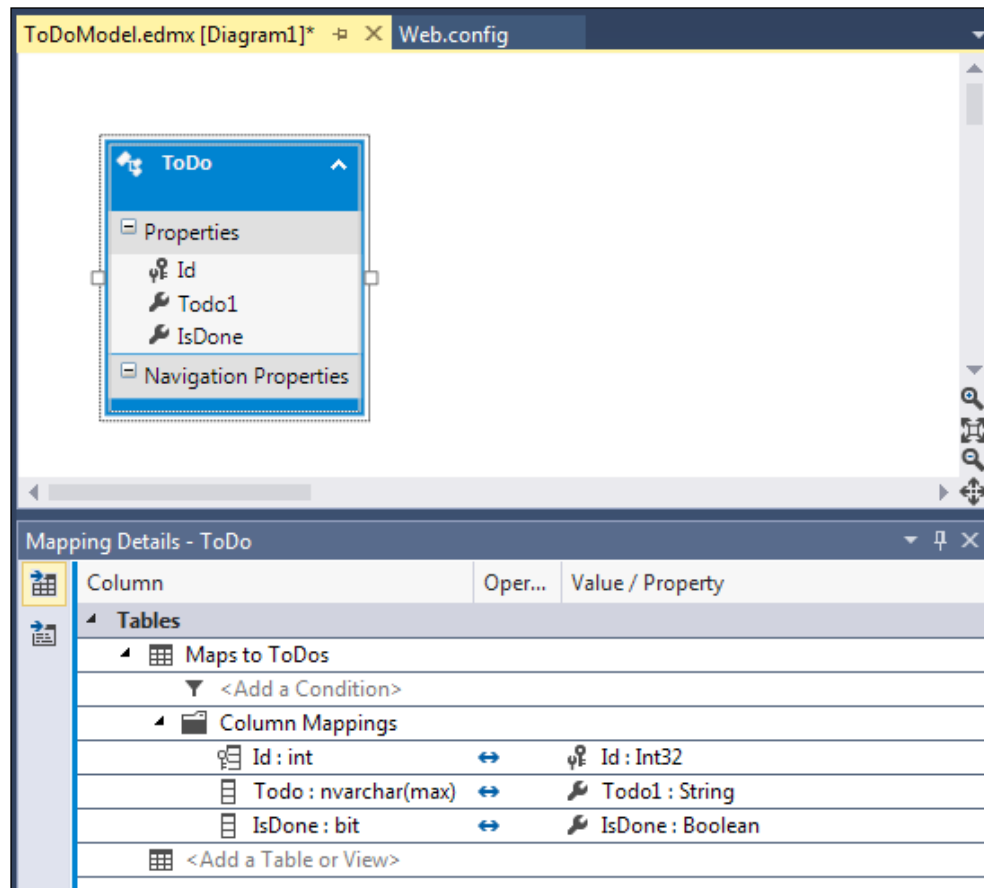
ADO.NET EDM wizard step 2 – select the database

Once the connection is successfully established, the wizard shows us all the tables, views, procedures, and functions that we will want to access from our application. Using this wizard, we can also choose to include the foreign key relations in our conceptual model and singularization of the model name, in case the table name is plural in our conceptual model:



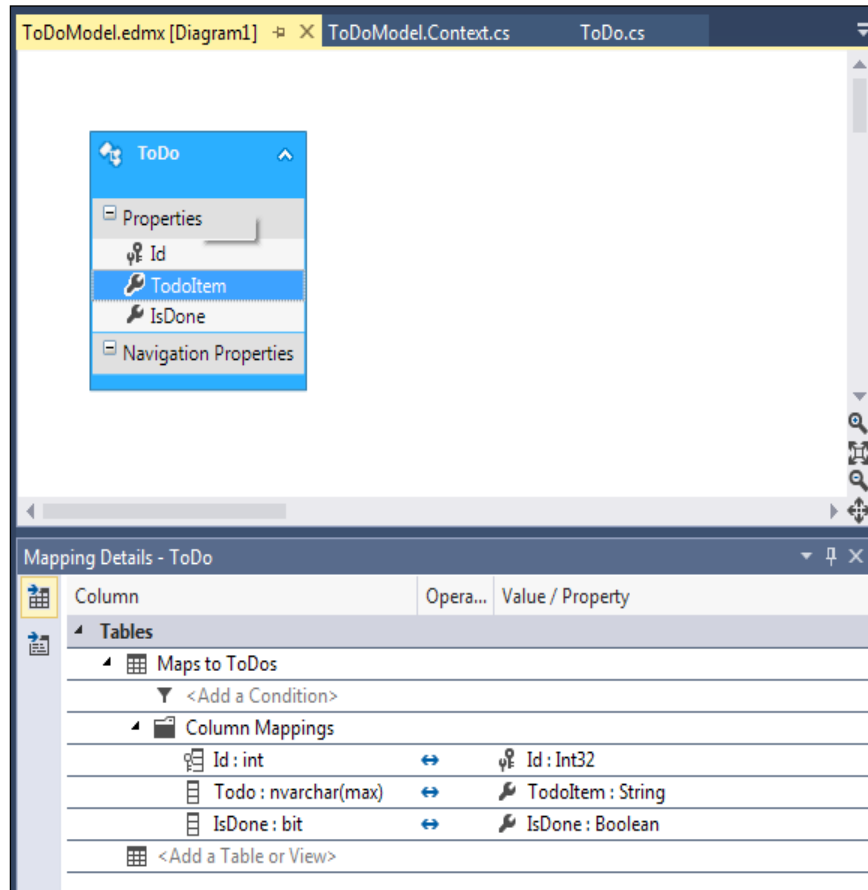
ADO.NET EDM wizard step 3—select the schema elements to be included in the EDM

Once the conceptual model is created, the EDM will be created for use in our application. The application will create a .EDMX file that contains all the information about our conceptual entity model and its mapping to the database tables:



Visual Entity Designer showing the default model

We can modify the conceptual model and change the mapping using entity designer. Let's try to change the property `ToDo1` to `ToDoItem` in our conceptual model. Note that only the class property has been changed, and not the column name:



Visual Entity Designer showing the customized model

If we open the EDMX file, we can find the following three sections:

- Conceptual schema definition: This specifies how a strongly typed model for our conceptual model will be created:

```
<edmx:ConceptualModels>
  <Schema Namespace="ToDoDBModel" Alias="Self" annotation:UseStrongSpatialTypes="false" xmlns:annotation="http://schemas.microsoft.com/ado/2009/02/edm/annotation" xmlns="http://schemas.microsoft.com/ado/2009/11/edm">
    <EntityType Name="ToDo">
      <Key>
```

```

        <PropertyRef Name="Id" />
    </Key>
    <Property Name="Id" Type="Int32" Nullable="false" />
    <Property Name="TodoItem" Type="String" MaxLength="Max"
FixedLength="false" Unicode="true" Nullable="false" />
    <Property Name="IsDone" Type="Boolean" Nullable="false" />
</EntityType>
<EntityContainer Name="ToDoDBEntities" annotation:LazyLoadingEna
bled="true">
    <EntitySet Name="Todos" EntityType="Self.ToDo" />
</EntityContainer>
</Schema>
</edmx:ConceptualModels>

```

- **Storage schema definition:** This specifies how the storage model is created, that is, how the values are stored in the database:

```

<edmx:StorageModels>
    <Schema Namespace="ToDoDBModel.Store" Provider="System.
Data.SqlClient" ProviderManifestToken="2012" Alias="Self"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/
EntityStoreSchemaGenerator" xmlns="http://schemas.microsoft.com/
ado/2009/11/edm/ssdl">
        <EntityType Name="Todos">
            <Key>
                <PropertyRef Name="Id" />
            </Key>
            <Property Name="Id" Type="int" Nullable="false" />
            <Property Name="Todo" Type="nvarchar(max)" Nullable="false" />
            <Property Name="IsDone" Type="bit" Nullable="false" />
        </EntityType>
        <EntityContainer Name="ToDoDBModelStoreContainer">
            <EntitySet Name="Todos" EntityType="Self.Todos" Schema="dbo"
store:Type="Tables" />
        </EntityContainer>
    </Schema>
</edmx:StorageModels>

```

- **Mapping:** This specifies the mapping between the conceptual model and the storage model:

```

<edmx:Mappings>
    <Mapping Space="C-S" xmlns="http://schemas.microsoft.com/
ado/2009/11/mapping/cs">
        <EntityContainerMapping StorageEntityContainer="ToDoDBModelStore
Container" CdmEntityContainer="ToDoDBEntities">
            <EntitySetMapping Name="Todos">

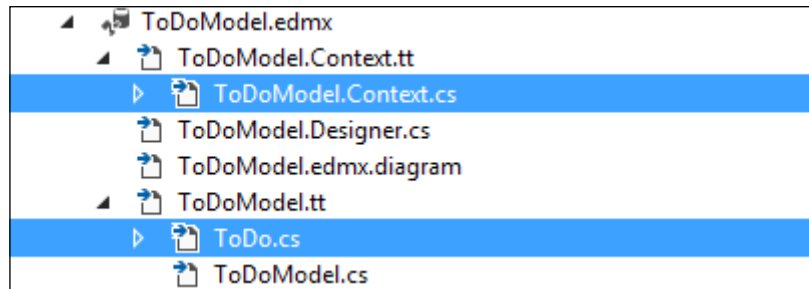
```

```
<EntityTypeMapping TypeName="ToDoDBModel.ToDo">
  <MappingFragment StoreEntitySet="Todos">
    <ScalarProperty Name="Id" ColumnName="Id" />
    <ScalarProperty Name="ToDoItem" ColumnName="ToDo" />
    <ScalarProperty Name="IsDone" ColumnName="IsDone" />
  </MappingFragment>
</EntityTypeMapping>
</EntitySetMapping>
</EntityContainerMapping>
</Mapping>
</edmx:Mappings>
```



We should always prefer to change these values from the Visual Entity Designer. We can modify this XML manually to change it as per our needs, but Visual Entity Designer allows us to perform most of the modifications, and unless we need to make a change that is not supported by entity designer, we should try not to make them manually.

Once the EDM is created, we will get a class generated for each model object present in our conceptual model. We will also get the `DbContext` class that will let's use these strongly typed models:



Solution Explorer showing the generated DbContext and Model class

The generated `DbContext` class will look like this:

```
public partial class ToDoDBEntities : DbContext
{
    public ToDoDBEntities()
        : base("name=ToDoDBEntities")
    {
    }

    public virtual DbSet<ToDo> Todos { get; set; }
}
```

The generated model class will look like this:

```
public partial class ToDo
{
    public int Id { get; set; }
    public string TodoItem { get; set; }
    public bool IsDone { get; set; }
}
```

Now we are ready to write typed queries against the generated conceptual model, and Entity Framework will execute our queries on the database, and return the results in the form of these strongly typed models. Creating new data or updating the existing data is also very easy, because Entity Framework automatically tracks the changes we make in our models, and lets us save the changes to the database. This can simply be done by calling the `SaveChanges` method on the `DbContext` class. One important thing to note here is that the read operation can utilize **Language-Integrated Query (LINQ)** to query the object data model, and Entity Framework will take care of converting it to the appropriate SQL query and retrieve the results.

If our application is a data-centric application, and all we need is to be able to perform CRUD operations using strongly typed objects, this is all that we are going to need. However, one of the great benefits of Entity Framework is being able to modify the conceptual models to better adapt to our domain needs, and then use the mappings to map the domain model to the database tables. We can have relationships between our domain models and still let them persist the data in a given set of tables.



In the Database First approach, the Entity Model is able to incrementally update the conceptual model if the database schema is updated. We just need to right-click on the visual designer and chose **Update Model from Database**.

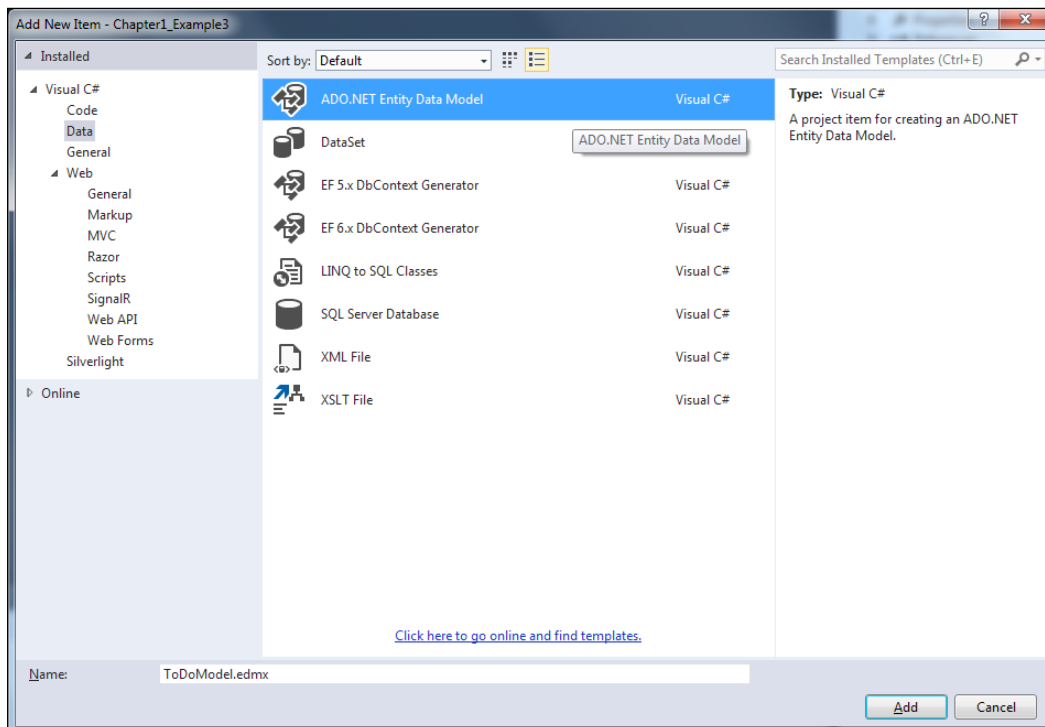
In the later chapters, we will see how to manage multiple models, how we can define relationships between the entities, introduce inheritance hierarchies, customize entities, and much more. This will give you a good foundation to get started with the Entity Framework Database First approach if you haven't used it before, and will refresh your memory if you have.

Entity Framework Model First approach

The Model First approach is useful when we don't have a database to start with. Using this approach, we can create our models directly in the Visual Entity Designer, and then create the database schema based on our model. In this approach too, we can create entities and their properties, define relationships and constraints between the entities, and create inheritance relationships.

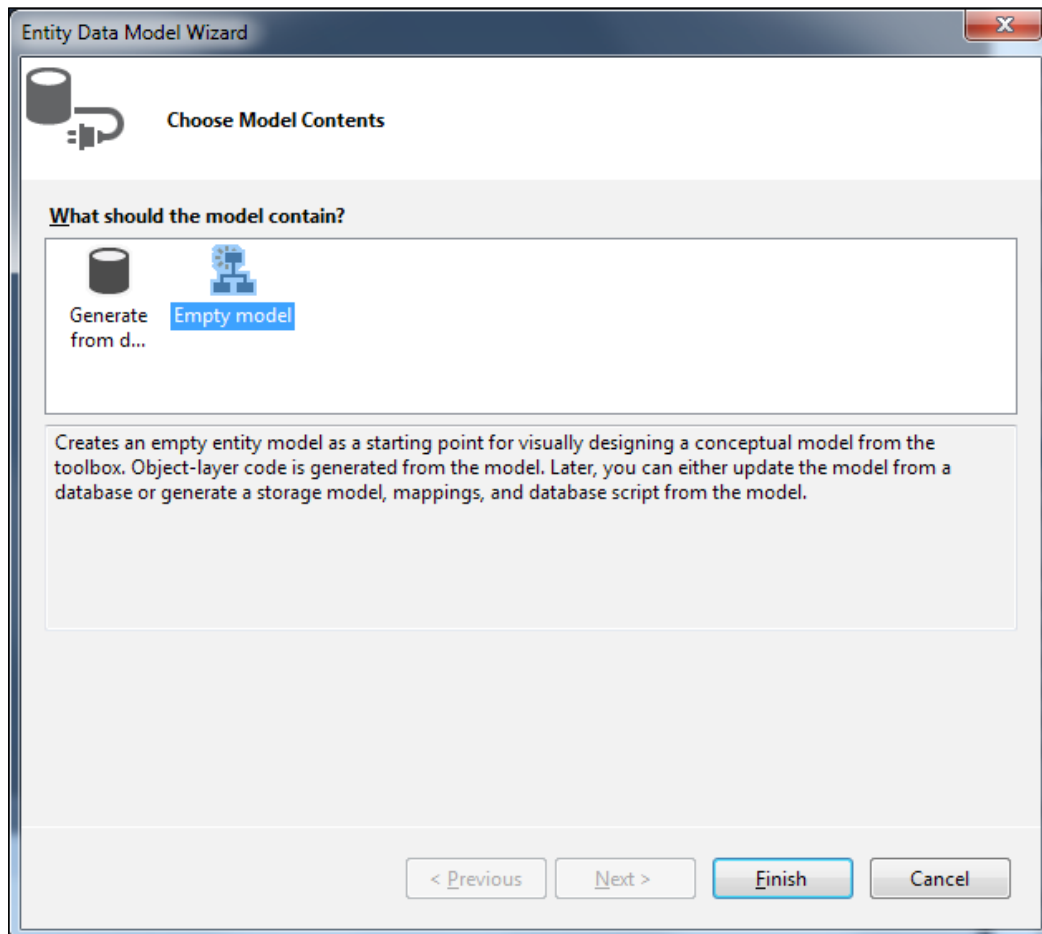
In this approach, we will first create the conceptual model, and based on this the database and the strongly typed objects will be created. To start with the Model First approach, we first need to add an ADO.NET EDM.

Let's try to add an ADO.NET EDM in our application:



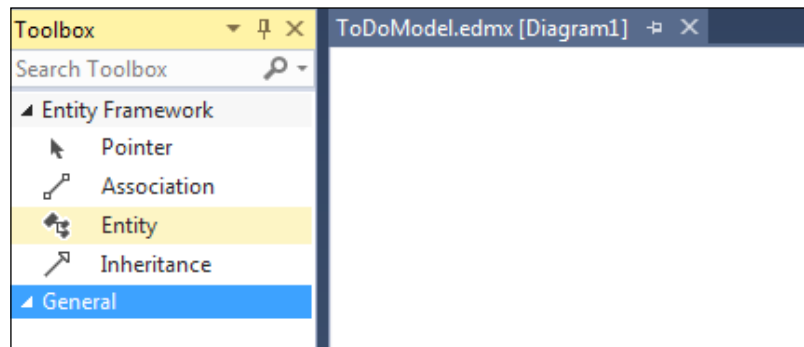
Add a new ADO.NET EDM

Since we are planning to start with an empty model, we have to select an empty model in the wizard. The following screenshot shows the wizard step that asks you to select the option. We have to select **Empty model** in this step:



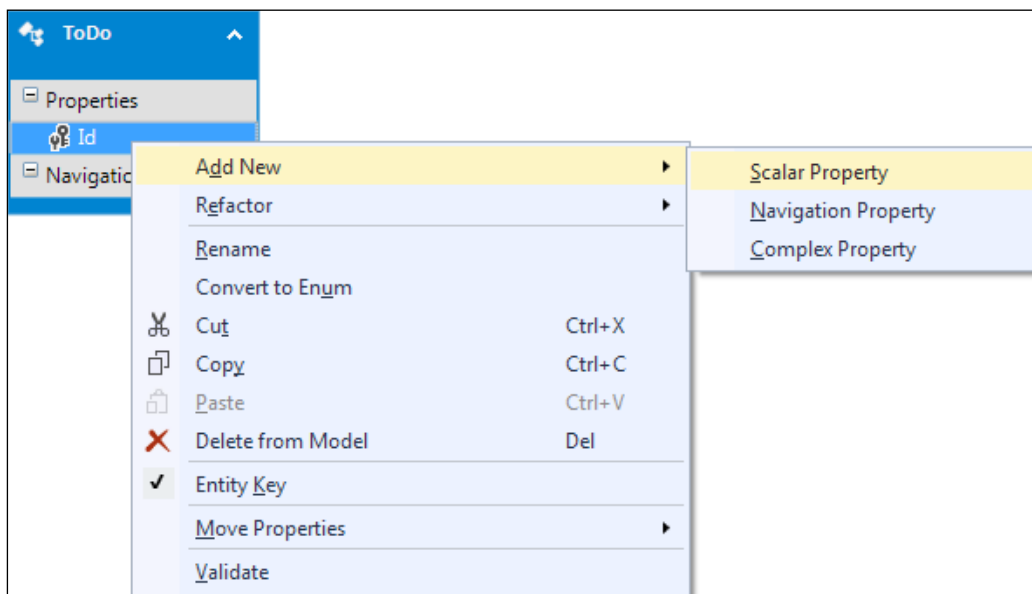
ADO.NET EDM wizard step 1 –select the Model First approach

Once we choose the empty model, Entity Framework will show us the Visual Entity Designer. Now we can add the entities and relations to the designer area from the toolbar. The Entity Designer and the toolbar will look something like this:



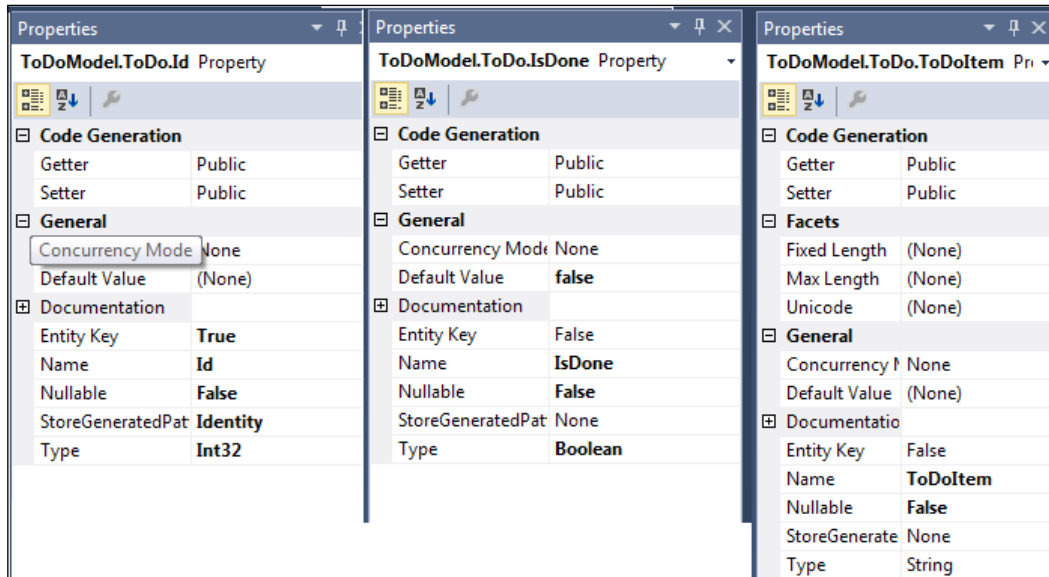
An empty Visual Entity Designer after selecting the Model First approach

Let's try to add a new entity to our `ToDo` application and add properties to that model. Let's add a few properties to our `ToDo` model:



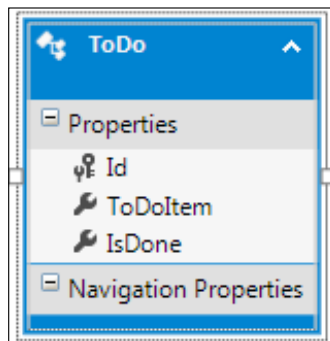
Adding a new property from Visual Entity Designer

Let's add the scalar properties needed for our `ToDo` model. Let's make the ID field a primary key field, a string property for `ToDoItem`, and a Boolean property for `IsDone`:



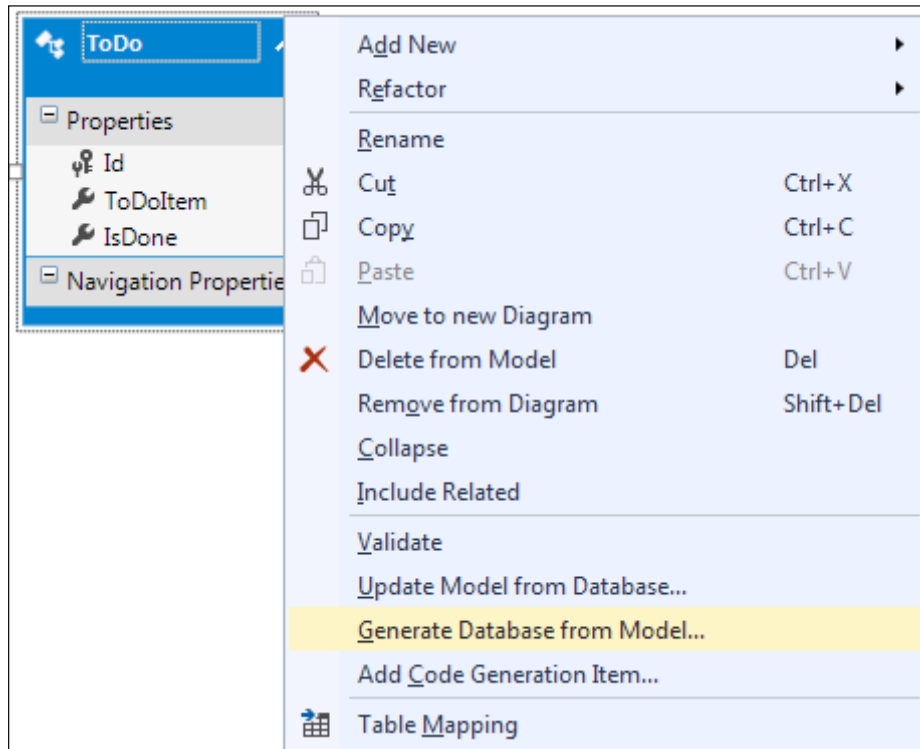
Visual Studio's Property panes showing the properties of a conceptual model

The created entity will look something like this:



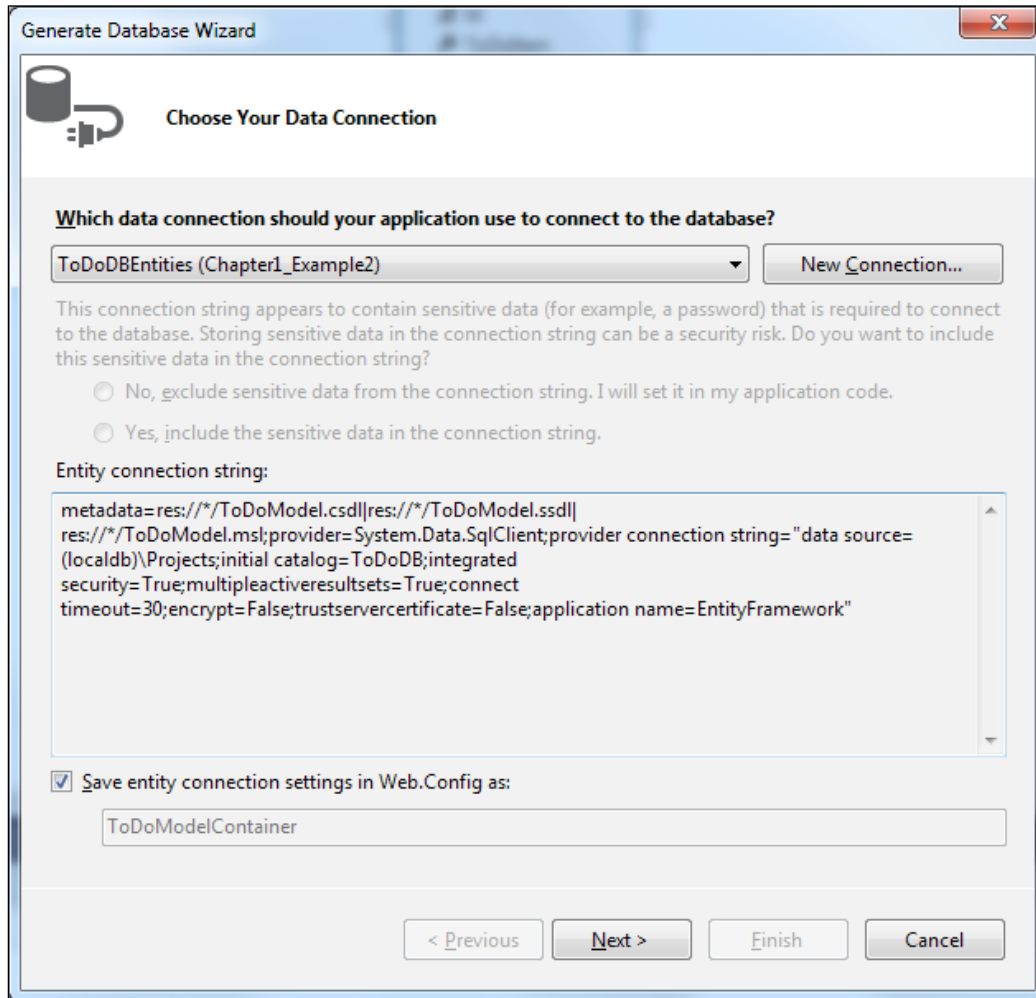
The final model created from the Visual Entity Designer

Once we have the conceptual model created for our application, we have to create the database from this conceptual model. This can be done by right-clicking on the entity and choosing **Generate Database from Model**:



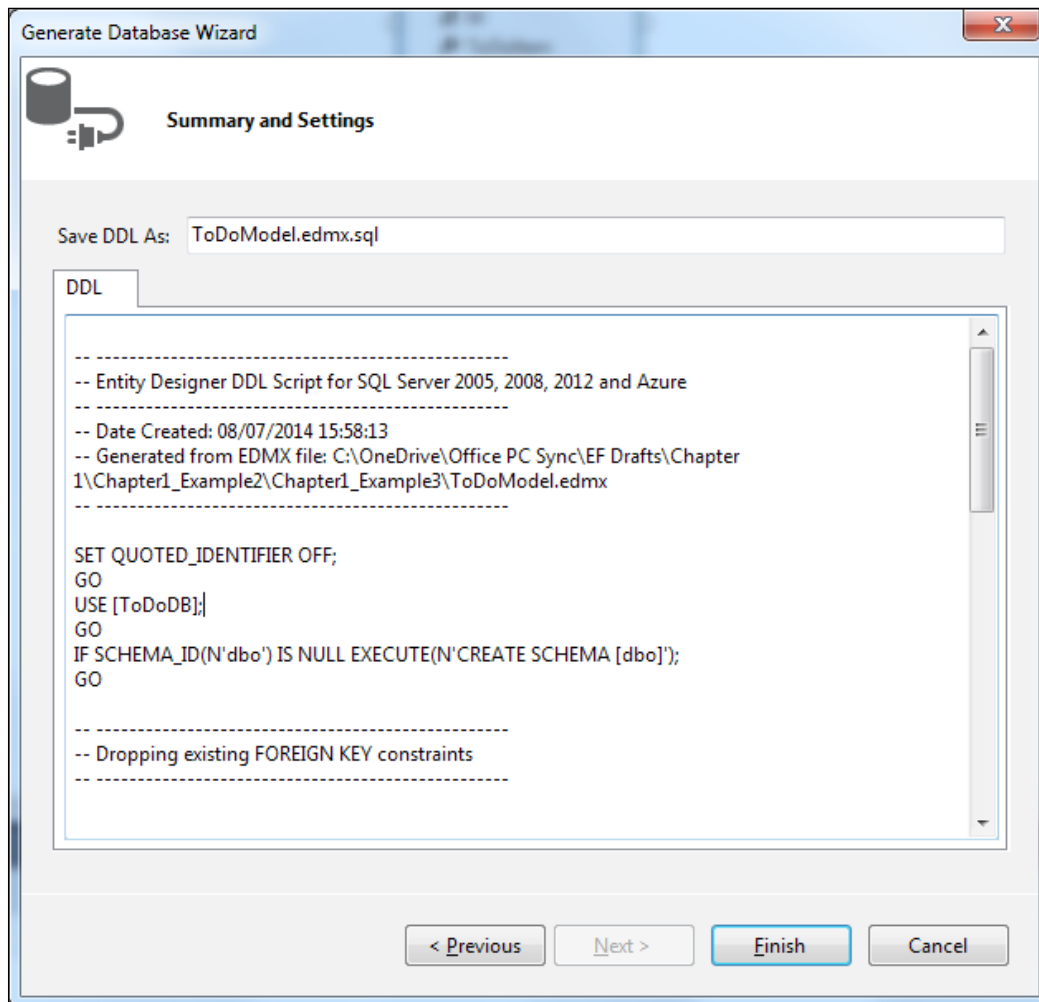
Generating database scripts from the Visual Entity Designer

This will prompt you to select the database connection. This wizard will not create the actual database, but it will generate the DDL for us, and we will have to use that DDL to create the database ourselves:



Wizard step to select the connection string to be used for a generated database script

After we select the database connection, the wizard will give us the SQL to create the database on the specified connection:



Wizard step showing the create database script

We can now copy this SQL and create the database. The Entity Framework has already created `connectionString` to use this connection, and wired up the `DbContext` class to perform the operations on this database. Let's take a look at the generated `DbContext` class:

```
public partial class ToDoModelContainer : DbContext
{
    public ToDoModelContainer()
        : base("name=ToDoModelContainer")
    {
    }

    public virtual DbSet<ToDo> ToDoes { get; set; }
}
```


The strongly typed objects will also be generated from the conceptual model. Let's take a look at our generated `ToDo` class:

```
public partial class ToDo
{
    public ToDo()
    {
        this.IsDone = false;
    }

    public int Id { get; set; }
    public string ToDoItem { get; set; }
    public bool IsDone { get; set; }
}
```

If we are using the Model First approach, one important thing to consider is that the incremental changes in our model will not perform the incremental updates to the database. Rather, it will give us the SQL to create the updated database from the ground up.

[



If we want to perform incremental updates to the database, we can use the *Database Generation Power Pack* from Microsoft (<https://visualstudiogallery.msdn.microsoft.com/df3541c3-d833-4b65-b942-989e7ec74c87/>). Using this, we can perform incremental updates to the database.

]

Entity Framework Code First approach

Now, let's take a look at the Entity Framework Code First approach. In this approach, we will start by creating the model classes ourselves. We will then implement our business logic against these classes, and then if we need to persist these models, we can use the Entity Framework Code First framework to save these models to the database. Thus, the Entity Framework Code First approach enables us to write simple C# classes, that is, **Plain Old CLR Objects (POCOs)** for our models, and then lets us persist them in a data store by defining a `DbContext` class for our model classes.

So, if we want to implement our `ToDo` application using the Code First approach, we will first have to create the `ToDo` model ourselves. Let's create a simple `ToDo` class that will keep track of our `ToDo` items:

```
public class ToDo
{
    public int Id { get; set; }
    public string ToDoItem { get; set; }
    public bool IsDone { get; set; }
}
```

Once we have the `ToDo` class ready, we need to start thinking about how the database will be created from our classes. We need to tell Entity Framework how to use our classes and generate the database accordingly. We want to tell the database generation module about all the persistence information, such as table names, key columns, and so on. We can do this in our model classes. So, if we try to put these attributes in our model class, our resultant `ToDo` model will look like this:

```
[Table("ToDo")] // Table name
public class ToDo
{
    [Key] // Primary key
    public int Id { get; set; }

    [Column("ToDoItem", TypeName="ntext")]
    public string ToDoItem { get; set; }

    [Column("IsDone", TypeName="bit")]
    public bool IsDone { get; set; }
}
```

In the preceding code, we updated our class with a table attribute passing in the table name. This will tell the Entity Framework that a table should be created for this class. Each property of the class can be updated with the `Column` attribute, and we can specify the name of the column this property should be mapped to and the data type. The `Key` attribute will be used to specify the property that should be used as a primary key.

Entity Framework uses the `DbContext` class to save the models to the database. In the Code First approach, we need to create the `DbContext` class ourselves too. Let's see how we can implement our `DbContext` class:

```
public partial class ToDoDbContext : DbContext
{
    public ToDoDbContext()
        : base("name=ToDoConnectionString")
    {
    }

    public DbSet<ToDo> ToDoes { get; set; }
}
```

The context class that we are creating should be derived from the `DbContext` class. The context class will also keep `DbSet` of `ToDo`s that will represent the list of `ToDo` items being retrieved from the table. The constructor of the context class will pass the name of `connectionString` to the base `DbContext` class. This `connectionString` will be used to connect to the database.

Another possibility is to keep the name of the `connectionString` the same as the `DbContext` class name. The Entity Framework will be able to use the corresponding `DbContext` class with this `connectionString` to persist the data. This is an example of the *Convention over configurations* principle. However, this is also flexible, so that we have the possibility of giving custom names to the `connectionStrings`.

Now to be able to use Entity Framework to persist the data, we just need to create this `connectionString` in the XML configuration, and Entity Framework will take care of creating the database and performing the CRUD operations on their respective models.


This has shown you how to use the Code First approach using a single model. In later chapters, you will see how to use multiple models and have relationships between them, how we can define complex types, and customize models as per our application requirements.

We have looked at how Entity Framework supports various development approaches and lets us create the conceptual model and data access logic very easily. We have seen that Entity Framework depends on the EDM to perform the database CRUD operations, and we have also looked at the various ways to create the conceptual model and map it to the database using the Database First, Model First, and Code First approaches.

To fully understand the benefits of Entity Framework over ADO.NET, we must also look at how easy and efficient it is to use Entity Framework to perform the actual operations. This will also show how performing CRUD operations is the same, irrespective of the approach we take. Let's now see how we can perform CRUD operations using Entity Framework to fully understand the benefits of Entity Framework.

Performing CRUD operations using Entity Framework

We will perform CRUD operations on our `ToDo` model using the Entity Framework `DbContext` class.

 It doesn't matter which approach we select to create the database and conceptual model (Database First, Code First, or Model First); from the code perspective, it's the same code that the developers need to write in order to use Entity Framework.

Now let's say we have a model for our `ToDo` item, that is, `ToDo`. We also have the `DbContext` class created to perform CRUD operations on the `ToDo` model. Let's take a look at how we can use Entity Framework to perform CRUD operations on our `ToDo` model. The important thing to note in these examples would be the use of LINQ. We will write all our queries in LINQ, and Entity Framework will take care of converting it to the appropriate SQL and return the results.

Reading a list of items

The way Entity Framework works is that using the `DbContext` class, we can retrieve all the data by simply accessing the model collection associated with the view. So, to fetch a list of items, we just need to get the model collection from the context class:

```
using(ToDoDBEntities db = new ToDoDBEntities())
{
    IEnumerable<ToDo> todoItems = db.Todos;
}
```

Let's try to understand the preceding code:

1. We created an object of the `DbContext` class:

```
ToDoDBEntities db = new ToDoDBEntities();
```

2. We then used the `DbContext` object to fetch the `ToDo` entities:

```
IEnumerable<ToDo> todoItems = db.Todos;
```

After this call, the list of `ToDo` items will be available in the `ToDo` items collection if we enumerate `ToDo` items. Entity Framework will internally perform the following activities for us to fetch the result:

3. Parsed our request for the data.
4. Generated the needed SQL query to perform this action.
5. Used the context class to wire up the SQL to the database.
6. Fetched the results from the database.
7. Created the strongly typed models from the retrieved results and return them to the user.



If we need to perform eager loading, we can do this by calling `ToList()` on the collection as: `IEnumerable<ToDo> todoItems = db.Todos.ToList();`

Reading a specific item

If we want to retrieve a single item from the model collection, we have many ways of doing it. Typically, we can use any LINQ function to fetch the single record from the model collection. Also, if we want to fetch the result using the primary key, we can use the `Find` method to do it. Let's look at a few ways in which we can retrieve a single item using Entity Framework:

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
    ToDo todo1 = db.Todos.Find(id);

    ToDo todo3 = db.Todos.FirstOrDefault(item => item.TODOItem == "Test
item");
    ToDo todo2 = db.Todos.SingleOrDefault(item => item.TODOItem == "Test
item");
}
```


Let's try to understand the preceding code:

1. We created an object of the `DbContext` class:

```
ToDoDBEntities db = new ToDoDBEntities();
```
2. Retrieved an item by passing the key values, that is, using the `Find` function:

```
ToDo todo1 = db.Todos.Find(id);
```
3. Retrieved an item by passing a non-key value using the `FirstOrDefault` function:

```
ToDo todo2 = db.Todos.FirstOrDefault(item => item.TODOItem == "Test item");
```
4. Retrieved an item by passing a non-key value using the `SingleOrDefault` function:

```
ToDo todo2 = db.Todos.SingleOrDefault(item => item.TODOItem == "Test item");
```



The `FirstOrDefault` and `SingleOrDefault` functions will return the first occurrence of the item if it is found, else it will return null. We should always check for the null values before using the item retrieved using these functions. `SingleOrDefault` will throw an exception if it finds multiple items matching the request criteria.

Creating a new item

To create an item, we simply need to create the model, assign the values in the model properties, and then call the `SaveChanges` method on Entity Framework. Now let's see how we can create a `ToDo` item using Entity Framework:

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
    ToDo todoItem = new ToDo();
    todoItem.TODOItem = "This is a test item.";
    todoItem.IsDone = false;
    db.Todos.Add(todoItem);

    db.SaveChanges();
}
```

Let's try to understand the preceding code:

1. We created an object of the `DbContext` class:

```
ToDoDBEntities db = new ToDoDBEntities();
```
2. Created a new `ToDo` item:

```
ToDo todoItem = new ToDo();
```
3. Populated the `ToDo` item properties with the desired values:

```
todoItem.TODOItem = "This is a test item.";
todoItem.IsDone = false;
```
4. Added the item to the context class:

```
db.Todos.Add(todoItem);
```
5. Called Entity Framework's `SaveChanges` method to persist these changes in the database:

```
db.SaveChanges();
```



Entity Framework's `DbContext` class is able to identify a new model and generate the respective insert query for it to create a new record on the database.

Updating an existing item

To update an existing item, there are two possibilities. The first possibility is that we will fetch the model, update some of the properties of the model, and then save the updated model in the database. To do this, we just need to fetch the required model using the `DbContext` class, update its properties, and then call the `SaveChanges` method on the `DbContext` object. Let's try to update a model with the ID 3:

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
    int id = 3;
    ToDo todo = db.Todos.Find(id);

    todo.TODOItem = "This has been updated";
    todo.IsDone = false;
    db.SaveChanges();
}
```

Let's try to understand the preceding code:

1. We created an object of the `DbContext` class:

```
ToDoDBEntities db = new ToDoDBEntities();
```
2. Fetched the item with `id = 3`:

```
int id = 3;  
ToDo todo = db.Todos.Find(id);
```
3. Updated the model properties:

```
todo.TODOItem = "This has been updated";  
todo.IsDone = false;
```
4. Called Entity Framework's `SaveChanges` method to persist these changes in the database:

```
db.SaveChanges();
```

What is happening in the preceding code is that the `DbContext` class was still in scope when the entity was being updated and saved. The `DbContext` class is able to track the changes in the entity if it is in scope. However, this approach is rarely useful, because in real applications in a typical n-tier application, the entities will be passed through multiple layers, and any layer can update this entity, and at the time of updating the entity, objects are usually disconnected from the context.

So to perform the updates in such a disconnected environment, we need to attach the updated entity back to let the `DbContext` class identify the modifications made in the entity, and then generate the appropriate update SQL accordingly and persist the updates in the database. This can be done by attaching the entity back to the `DbContext` class. Let's see how a `ToDo` model can be attached to the `DbContext` class if it is being passed from a different tier:

```
using (ToDoDBEntities db = new ToDoDBEntities())  
{  
    db.Entry(todo).State = EntityState.Modified;  
    db.SaveChanges();  
}
```

Let's try to understand the preceding code:

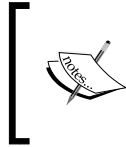
1. We created an object of the `DbContext` class:

```
ToDoDBEntities db = new ToDoDBEntities();
```
2. Attached the model to the `DbContext` class and marked this model as modified:

```
db.Entry(todo).State = EntityState.Modified;
```

3. Called Entity Framework's `SaveChanges` method to persist these changes in the database:

```
db.SaveChanges();
```



In the preceding code, the `ToDo` model is being modified and passed from some other layer. This is a fairly advanced topic and if it is not clear at this stage, please move on; this will be covered in detail in further chapters.

Deleting an item

To delete an item, we just need to fetch the item that needs to be deleted, call the `Remove` method on the model collection, and call the `SaveChanges` method on the `DbContext` class. Let's try to delete a model with the ID 3:

```
using (ToDoDBEntities db = new ToDoDBEntities())
{
    int id = 3;
    ToDo todo = db.ToDos.Find(id);
    db.ToDos.Remove(todo);
    db.SaveChanges();
}
```

Let's try to understand the preceding code:

1. We created an object of the `DbContext` class:

```
ToDoDBEntities db = new ToDoDBEntities();
```

2. Fetched the item with `id = 3`:

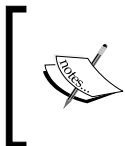
```
int id = 3;
ToDo todo = db.ToDos.Find(id);
```

3. Removed the item from the collection:

```
db.ToDos.Remove(todo);
```

4. Called Entity Framework's `SaveChanges` method to persist these changes in the database:

```
db.SaveChanges();
```



It is also possible that the model entities are getting created and deleted while being disconnected from the `DbContext` class. In these cases, we need to attach the models to the `DbContext` class, and mark them using the `EntityState` property.

Choosing persistence approaches

Now, we saw the three options to create the EDM and the `DbContext` class. Which approach should be used in which scenario, is often a question that many developers ask. There is no definite reason for choosing any particular approach, but here is a small checklist that the developers might find very useful while deciding on which approach to choose. If at any point, the answer to any question is yes, then that approach should be used without even looking at the next question in the checklist:

Question	Approach
Is there a legacy database or does the database already exist?	Database First
Will we be getting a database created by the DBAs before starting the development?	Database First
Will there be frequent database changes, based on which our application should change?	Database First
Do we want to use the Visual Entity Designer to generate the database and model classes?	Model First
Do we have the model classes already and we need the database to save the data only?	Code First
Do we want to write all the model classes, implement them, and then think about the database storage later?	Code First
We don't want to deal with the auto-generated classes and would prefer to write them ourselves	Code First
Is the answer to all the preceding questions "no"?	Database First

Summary

In this chapter, we talked about the ORMs and how ORMs can ease the task of writing data access code. We saw what Entity Framework is and how it works. We then saw the various approaches for using Entity Framework. We performed CRUD operations on a simple model to see Entity Framework in action. Finally, we looked at which approach would better fit our needs based on the given scenario.

In the next chapter, we will see how we can use the Entity Framework Database First approach to model complex relationships between database tables as relationships between the domain models. We will also see how to work with these related models using Entity Framework.

2

Entity Framework DB First – Managing Entity Relationships

In this chapter, we will see how to model relationships between tables in our Entity Data Model and use Entity Framework to work with these related entities. We will look at the types of relationship between the tables, and then we will see how this can be modeled in our Entity Data Model. We will then see how we can perform CRUD operations by utilizing these relationships.

Understanding database relationships

In a very simple application, it is possible to have a single table. However, as the complexity of the application grows and the need to store data increases, the number of tables also increases. If we are trying to store a complex related set of data, then these tables will also relate to each other using foreign keys. To model our database correctly and have a properly normalized database, we will end up with a set of related tables. Let's look at the various types of relationship that can exist between tables in a database.

There are three types of relationship that can exist between tables:

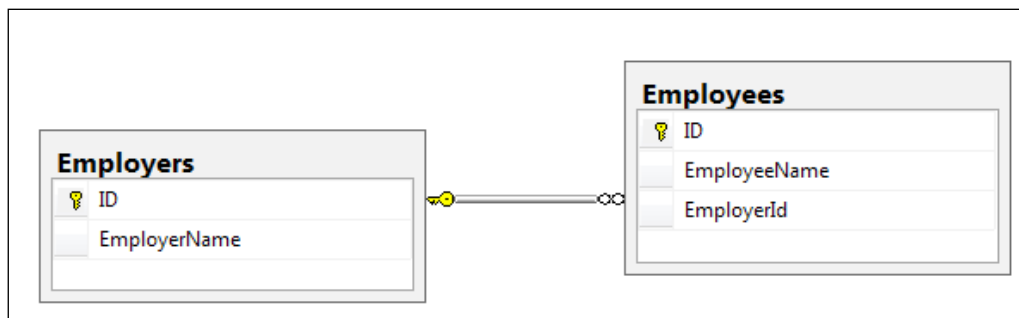
- One-to-many relationship
- One-to-one relationship
- Many-to-many relationship

Let's try to understand these relationships from a real-world perspective. We will then see how these relationships are realized in the database.


One-to-many relationship

This type of relationship is needed when a row from one table is related to many rows in another table. One typical example of this is the employer-employee relationship. One organization has many employees, that is, for each row in the **Employers** table, there will be many rows in the **Employees** table. Alternatively, every employee in the Employees table will these belong to only one employer in the Employers table. It means that an Employer can have 0...* Employees but that an Employee has to be attached to an Employer.

To implement a one-to-many relationship, we have to create a foreign key in the table that is on the *many* end, Employees in our case. This will refer to the Employers table's primary key. The primary key of the table at the *one* end of relationship, that is, the Employers table will have a unique constraint. Let's try to visualize this relationship by looking at the database schema created for it.



A one-to-many relationship between tables

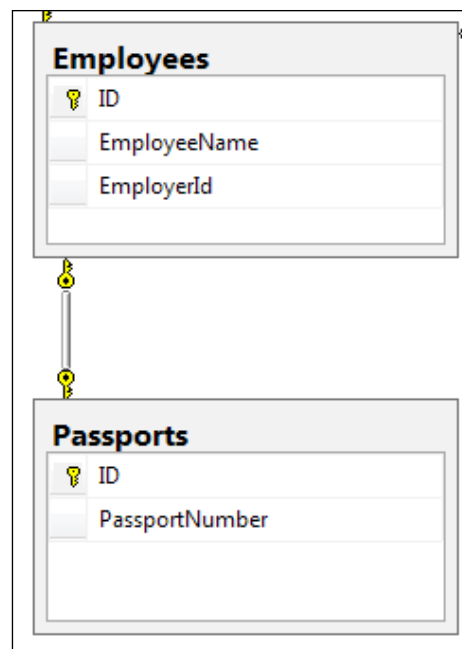
 In the preceding screenshot, the key symbol shows the *one* side of the relationship and the infinity symbol represents the *many* side of the relationship. This means that every record in the Employees table should have a related record in the Employers table, and for every record in the Employers table, there could be many related records in the Employees table.

One-to-one relationship

This is not a very common scenario, but it is possible. In this scenario, a row from one table will be related to only one row in another table. **Employees** and **Passports** are examples of this. One employee can have only one passport. So, every row in the Employees table will be related to a single row in the Passports table.

One way to implement a one-to-one relationship in the database is to have a foreign key in the related table, that is, the Passports table in our case. These foreign key columns will have a unique constraint to ensure that for every passport, there is only one relation to the Employees table.

Unfortunately, the Entity Framework does not support this way of creating a one-to-one relationship. There is another way to create a one-to-one relationship and that is by making both the tables involved in the relationship share a primary key. So, if we create a one-to-one relationship by creating both the Passports table and the Employees table share a primary key, then our schema will look like this:



A one-to-one relationship between tables

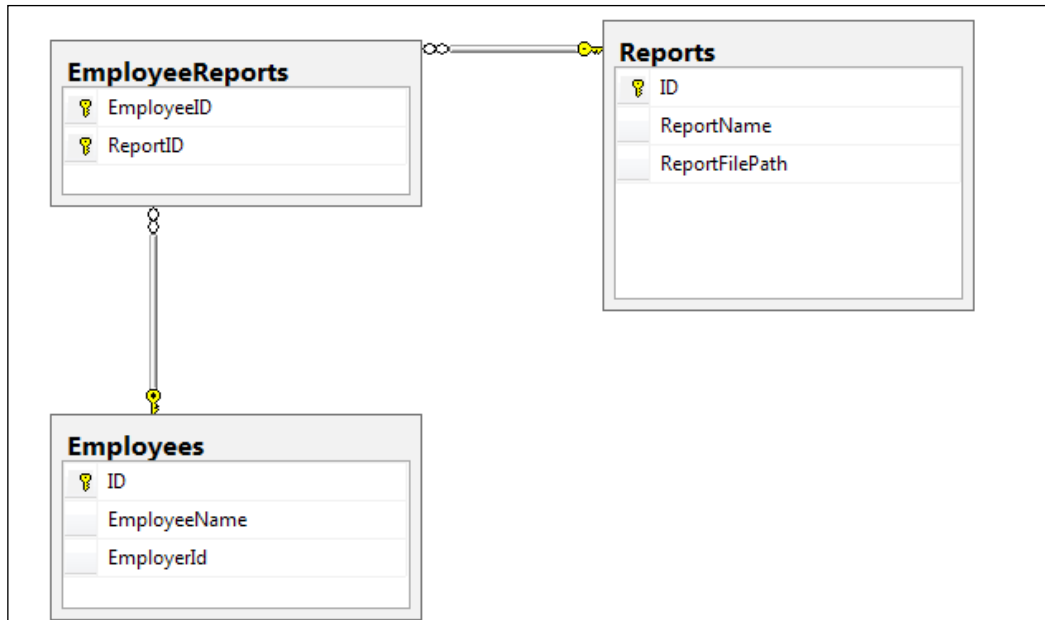


In the preceding screenshot, both the tables will have the key symbol, as it is a one-to-one relationship.


Many-to-many relationship


This is a very common scenario where one row from a table could be related to many rows of another table and vice versa. An example of this type of relationship could be the **EmployeeReports** relationship. One report can be written by many employees and alternatively, an employee may have written many reports.

To implement a many-to-many relationship, we need to create a third table. This table is often called a join table or a junction table. The primary keys of this table consist of the foreign keys from both the tables involved in the relationship. Let's try to visualize this relationship by looking at the database schema created for it.



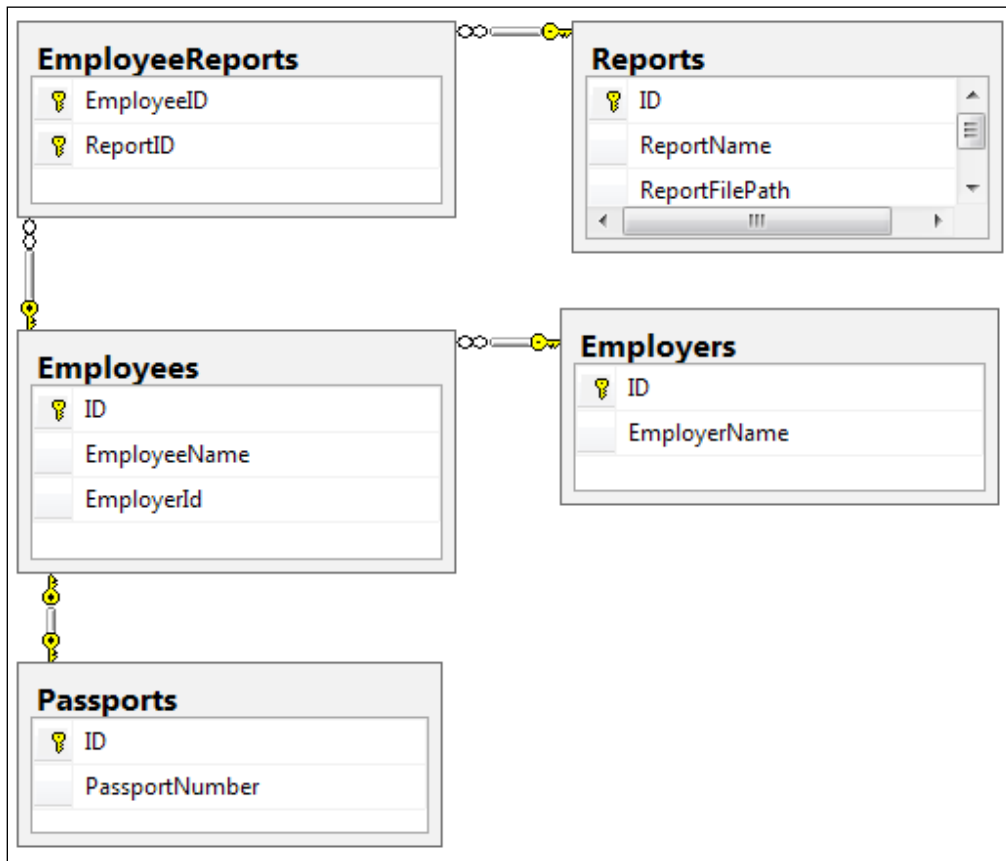
A many-to-many relationship between tables

 In the preceding screenshot, both the tables involved in the relationship are mapped as one-to-many with our join table, which would effectively give us a many-to-many relationship between these two tables.

 **Downloading the example code**
You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Creating the Entity model

Once we have all the relationships created in the database, the complete database will look like this:

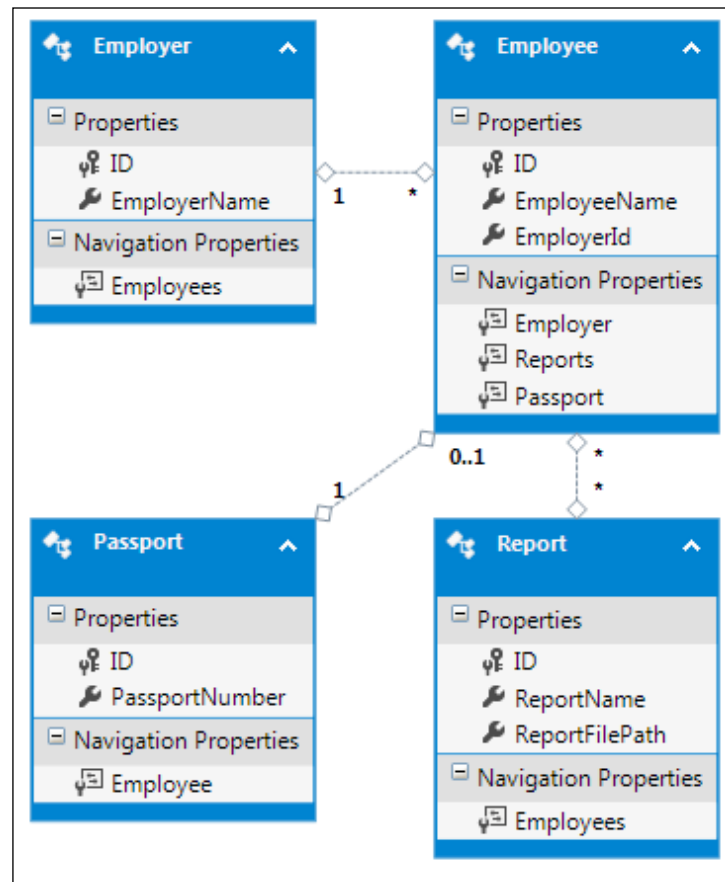


The database schema for the sample application



We will be using this database to create a sample application to understand how to model all the relationships using Entity Framework.

Now, let's add a new ADO.NET Entity Data Model in our application, and then, using this existing database, we will create the Entity Data Model, that is, the conceptual model for our application.

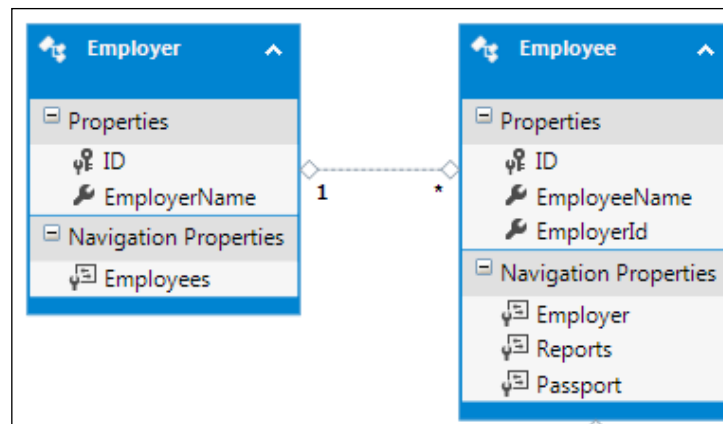


Entity Data Model generated using the Database First approach

We will now start looking at how these relationships are modeled in the conceptual model and how we can use them to perform operations on related entities.

Modeling a one-to-many relationship

Now, let's start our discussion by looking at the models that have one-to-many relationships. If we look at the Employer and Employee relation, we can see that the Entity Frameworks Entity Data Model generator was able to figure out the one-to-many relationship between the database tables and was able to create the same relationship in the conceptual Entity Data Model as well.



A one-to-many relationship between entities

Also, the relationship in the Entity Data Model is represented by **1** with the entity at the *one* end of the relationship and by ***** with the entity on the *many* end of the relationship.

If we look at our Entity Data Model, we can see that the Entity Framework created **Navigation Properties** in the entities. The entity at the *one* end of the relationship will have a navigation property of the collection type to hold multiple entities on the *many* end. Similarly, the Entity on the *many* end of the relation will have a navigation property that will let us access the entity related to it on the *one* end.

In our current example, the `Employer` model contains a navigation property, **Employees**, using which we can retrieve/update the list of employees associated with `Employer`. The `Employee` model also contains a navigation property, **Employer**, using which we can retrieve/update the **Employer** information for a given `Employee`.

If we look at the EDMX XML markup, we can see that the SSDL contains the information about the one-to-many relationship between the tables.

```
<Association Name="FK_Employees_Employers">
  <End Role="Employers" Type="Self.Employers" Multiplicity="1" />
  <End Role="Employees" Type="Self.Employees" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Employers">
      <PropertyRef Name="ID" />
    </Principal>
    <Dependent Role="Employees">
      <PropertyRef Name="EmployerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

SSDL showing a one-to-many relationship

The same relationship is also mentioned in the CSDL to indicate the one-to-many relationship between the entities.

```
<Association Name="FK_Employees_Employers">
  <End Role="Employers" Type="Self.Employer" Multiplicity="1" />
  <End Role="Employees" Type="Self.Employee" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Employers">
      <PropertyRef Name="ID" />
    </Principal>
    <Dependent Role="Employees">
      <PropertyRef Name="EmployerId" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

CSDL showing a one-to-many relationship

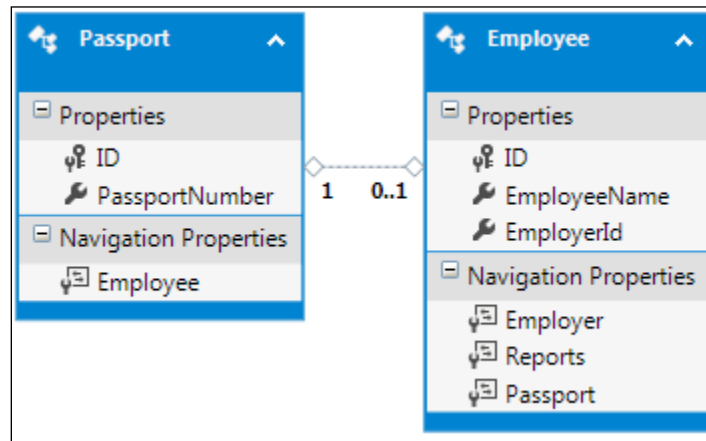
The MSL in this case will simply contain the mapping information of both the tables involved in the relationship to respective database columns.

```
<Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
  <EntityContainerMapping StorageEntityContainer="todoDbModelStoreContainer" CdmEnti
  <EntitySetMapping Name="Employees">
    <EntityTypeMapping TypeName="todoDbModel.Employee">
      <MappingFragment StoreEntitySet="Employees">
        <ScalarProperty Name="ID" ColumnName="ID" />
        <ScalarProperty Name="EmployeeName" ColumnName="EmployeeName" />
        <ScalarProperty Name="EmployerId" ColumnName="EmployerId" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
  <EntitySetMapping Name="Employers">
    <EntityTypeMapping TypeName="todoDbModel.Employer">
      <MappingFragment StoreEntitySet="Employers">
        <ScalarProperty Name="ID" ColumnName="ID" />
        <ScalarProperty Name="EmployerName" ColumnName="EmployerName" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
```

MSL showing the Employees and Employer entity mapping

Modeling a one-to-one relationship using Entity Framework

Now, let's look at the models that have one-to-one relationships. The `Employees` table and `Passports` table has a one-to-one relationship. If we look at the `Employee` and `Passport` relationship, we can see that the Entity Framework Entity Data Model generator was able to figure out the one-to-one relationship between these tables and was able to create the same relationship in the conceptual Entity Data Model as well.



A one-to-one relationship between entities

Also, one-to-one relationships in the Entity Data Model are represented by `1` with the `Employee` entity, because this is the main entity in the relation. The `Passport` entity shows `0..1`, which means that there could be either 0 or 1 entries in the `Passport` table for each employee.

Entity Framework also created **Navigation Properties** in the entities. Navigation properties are properties that link the entities to related entities. Using navigation properties, we can access the entities related to a given entity. The `Employee` model contains a navigation property, **`Passport`**, using which we can access the related `Passport` entity and similarly, the `Passport` model contains a navigation property, **`Employee`**, using which we can access the `Employee` object related to `Passport`.

If we look at the EDMX XML markup, we can see that the SSDL contains the information about this one-to-one relationship between the tables.

```
<Association Name="FK_Employees_Passports1">
  <End Role="Passports" Type="Self.Passports" Multiplicity="1" />
  <End Role="Employees" Type="Self.Employees" Multiplicity="0..1" />
  <ReferentialConstraint>
    <Principal Role="Passports">
      <PropertyRef Name="ID" />
    </Principal>
    <Dependent Role="Employees">
      <PropertyRef Name="ID" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

SSDL showing a one-to-one relationship

The same relationship is also mentioned in the CSDL to indicate the one-to-one relationship between these entities.

```
<Association Name="FK_Employees_Passports1">
  <End Type="todoDbModel.Passport" Role="Passport" Multiplicity="1" />
  <End Type="todoDbModel.Employee" Role="Employee" Multiplicity="0..1" />
  <ReferentialConstraint>
    <Principal Role="Passport">
      <PropertyRef Name="ID" />
    </Principal>
    <Dependent Role="Employee">
      <PropertyRef Name="ID" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

CSDL showing a one-to-one relationship

MSL in this case will simply contain the mapping information of both the tables involved in the relationship to the respective database columns.

```
<EntitySetMapping Name="Reports">
  <EntityTypeMapping TypeName="todoDbModel.Report">
    <MappingFragment StoreEntitySet="Reports">
      <ScalarProperty Name="ID" ColumnName="ID" />
      <ScalarProperty Name="ReportName" ColumnName="ReportName" />
      <ScalarProperty Name="ReportFilePath" ColumnName="ReportFilePath" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

MSL showing Reports entity mapping

Modeling a many-to-many relationship using Entity Framework

Now, let's look at the models that have many-to-many relationships. The `Employees` table and the `Reports` table have a many-to-many relationship. If we look at the `Employer` and `Report` relationship, we can see that the Entity Framework Entity Data Model generator was able to figure out this many-to-many relationship between these tables and was able to create the same relationship in the conceptual Entity Data Model as well.

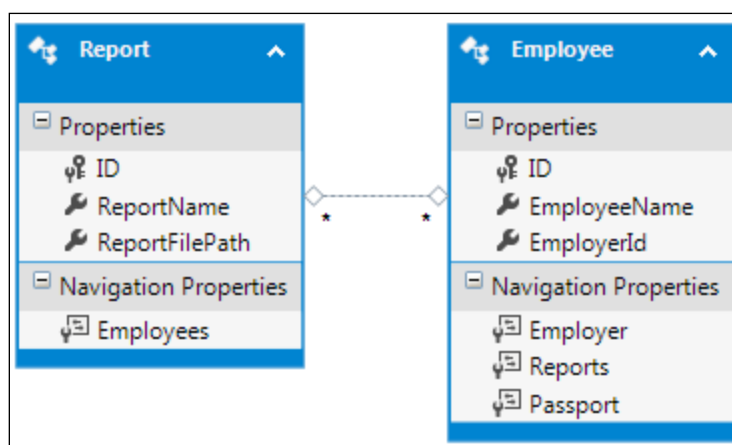
The important thing to notice here is that the Entity Framework was also able to understand that the `EmployeeReports` table was created solely to manage a many-to-many relationship, and is actually not needed in the conceptual model.



Entity Framework is able to figure out this many-to-many relationship and omit the joining table when the joining table has only PK columns for both tables.

Not having the joining table as a separate entity is actually beneficial as the entities will then have direct navigation properties to the related entities, thus eliminating the need for us to navigate to the related entity ourselves.

If we have other columns in the joining table, then Entity Framework will not be able to figure out this many-to-many relationship, and it will create all the entities as normal entities. In such a scenario, we have to manually join the entities in our object query to achieve the desired results and fetch the appropriate data.



A many-to-many relationship between entities

Also, the relationship in the Entity Data Model is represented by * with both the entities to denote a many-to-many relationship. Entity Framework also created **Navigation Properties** in the entities. The `Employee` model contains a Navigation property, **Reports**, using which we can access the related records from the `Reports` table and similarly, the `Report` model contains a navigation property, **Employees**, using which we can access `Employee` records related to `Reports`.

If we look at the EDMX XML markup in the following diagram, we can see that the SSDL contains information about this many-to-many relationship between the tables.

```
<Association Name="FK_EmployeeReports_Employees">
  <End Role="Employees" Type="Self.Employees" Multiplicity="1" />
  <End Role="EmployeeReports" Type="Self.EmployeeReports" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Employees">
      <PropertyRef Name="ID" />
    </Principal>
    <Dependent Role="EmployeeReports">
      <PropertyRef Name="EmployeeID" />
    </Dependent>
  </ReferentialConstraint>
</Association>
<Association Name="FK_EmployeeReports_Reports">
  <End Role="Reports" Type="Self.Reports" Multiplicity="1" />
  <End Role="EmployeeReports" Type="Self.EmployeeReports" Multiplicity="*" />
  <ReferentialConstraint>
    <Principal Role="Reports">
      <PropertyRef Name="ID" />
    </Principal>
    <Dependent Role="EmployeeReports">
      <PropertyRef Name="ReportID" />
    </Dependent>
  </ReferentialConstraint>
</Association>
```

SSDL showing a many-to-many relationship

The same relationship is also mentioned in the CSDL to indicate the many-to-many relationship between these entities.

```
<Association Name="EmployeeReports">
  <End Role="Employees" Type="Self.Employee" Multiplicity="*" />
  <End Role="Reports" Type="Self.Report" Multiplicity="*" />
</Association>
```

CSDL showing a many-to-many relationship

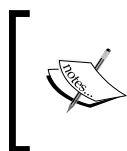
Now, as the Entity Framework is not showing the joining table and is able to create navigation properties directly in the entities participating in the many-to-many relationship, the mapping part of the EDMX contains information about the joining table that is responsible for facilitating this many-to-many relationship. This is done by defining `AssociationSet` in the mapping as follows:

```
<AssociationSetMapping Name="EmployeeReports" TypeName="todoDbModel.EmployeeReports" StoreEntitySet="EmployeeReports">
  <EndProperty Name="Employees">
    <ScalarProperty Name="ID" ColumnName="EmployeeID" />
  </EndProperty>
  <EndProperty Name="Reports">
    <ScalarProperty Name="ID" ColumnName="ReportID" />
  </EndProperty>
</AssociationSetMapping>
```

Mapping showing how the joining table has been mapped to facilitate a many-to-many relationship

Using navigation properties for data access

We have just discussed that the Entity Framework creates navigation properties to facilitate the data access operations on related entities. Let's now look at how we can use these navigation properties to perform CRUD operations on related entities.



We will only be working with entities involved in one-to-many relationships, that is, the Employer and Employee entity. However, using navigation properties to perform CRUD operation is more or less similar for all relationships.

Retrieving a specific item

If we have the entity at the *many* end of the application and we want to retrieve the entity at the *one* end, we can use the navigation property to do this. From our sample application's perspective, we will see how we can retrieve the employer for a given employee. This can be achieved by using the **Employer** navigation property, available in the Employer object. Let's see how this can be done in code:

```
public Employer GetEmployerByEmployee(int employeeId)
{
    Employer employer = null;
    using (SampleDbEntities db = new SampleDbEntities())
    {
        Employee employee = db.Employees.Find(employeeId);
        if (employee != null)
        {
```

```
        employer = employee.Employer;
    }
}

return employer;
}
```

In the preceding code, we create a `DbContext` object, and using this context object, fetch the `Employee` by the given `employeeId`. If we find an `Employee` object for this, we retrieve the `Employer` object associated with it. We do this by using the navigation properties. Internally, what Entity Framework does is that it creates a SQL query to fetch the employer associated with the employee with a given `employeeId`.

Retrieving a list of items

Let's start by looking at how we can use the entity at the *one* end of the relationship and retrieve the list of all the related entities at the *many* end. From our sample application's perspective, we will see how we can retrieve a list of employees if we have the employer object. This can be achieved by using the navigation property, `Employees`, available in the `Employer` object. Let's see how this can be done in code:

```
public List<Employee> GetEmployeesByEmployer(int employerId)
{
    List<Employee> employeeList = null;
    using (SampleDbEntities db = new SampleDbEntities())
    {
        Employer employer = db.Employers.Find(employerId);
        if (employer != null)
        {
            employeeList = employer.Employees.ToList();
        }
    }

    return employeeList;
}
```

In the preceding code, we create the `DbContext` object and using this object, we fetch the `Employer` by the given `employerId` function. If we find an `Employer` object with this ID, we retrieve the `Employees` associated with it. Internally, Entity Framework is generating a SQL to fetch all the employees associated with this `employerid` and return all the records as a collection of `Employee` entities.



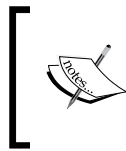
If the requirement was to find a particular Employee instead of the list, we can further query the Employee list retrieved.

Adding an item

If we want to add an entity in a relationship, we can simply assign it or add it to the Navigation property. For example, if we want to associate Employee with Employer, we just need to add the employee object to the employer's Employee collection. Let's see how this can be done in code:

```
public void AddEmployee(Employee employee, int employerId)
{
    using (SampleDbEntities db = new SampleDbEntities())
    {
        Employer employer = db.Employers.Find(employerId);
        employer.Employees.Add(employee);
        db.SaveChanges();
    }
}
```

In the preceding code, we create the DbContext object and using this object, we fetch Employer by the given employerId. If we find an Employer object with this ID, we add the Employee object to the employer's Employee collection and finally save the changes to database with `db.SaveChanges()`. What happens here is that if this Employee entity points to an existing Employee record in the table, the EmployerId value for that employee record will be updated to the ID of the new Employer. If this Employee record does not exist in the database, then a new record for this Employee will be created, and it will have the EmployerId of the given Employer object.



If this employee object that we are adding to the employer already exists in the database, only the EmployerId value will be updated in that record. If this employee record did not exist, a new record will be created for it.

Updating an item


Now, let's say that we want to update a related entity using the main entity, that is, the entity that is associated via the navigation property with the current entity we are using. For example, we want to update an Employee record using an Employer object that is related to this employee. Let's see how this can be done:

```
public void UpdateEmployee(Employee modifiedEmployee, int employerId)
{
    using (SampleDbEntities db = new SampleDbEntities())
    {
        Employer employer = db.Employers.Find(employerId);
        Employee employee = employer.Employees.FirstOrDefault
            (emp => emp.ID == newEmployee.ID);

        if (employee != null)
        {
            employee.EmployeeName = modifiedEmployee.EmployeeName;
        }
        else
        {
            employer.Employees.Add(employee);
        }

        db.SaveChanges();
    }
}
```

In the preceding code, we create the DbContext, and using this object, we fetch the Employer by the given employerId. If we find an Employer object with this ID, we find the existing Employee by using the ID in the new Employee object. Once the Employee is found, we update the object with new values and save the changes to the database with db.SaveChanges();.



The preceding code shows a very contrived example of updating the related entity. If we have an Employee object, then we can directly update it by associating it with the DbContext class and marking its state as updated. However, the intention of the preceding code was to explain how the related entities can be updated.

Deleting an item

Now, let's say that we want to delete an associated entity from the *many* end provided that we have the entity at the *one* end and the ID of one of the many elements to be deleted. From our sample application's perspective, let's see how we can delete an `Employee` with a given ID by using the `Employer` entity:

```
public void DeleteEmployee(int employeeId, int employerId)
{
    using (SampleDbEntities db = new SampleDbEntities())
    {
        Employer employer = db.Employers.Find(employerId);
        Employee employee = employer.Employees.FirstOrDefault(emp =>
emp.ID == employeeId);

        if (employee != null)
        {
            employer.Employees.Remove(employee);
            db.SaveChanges();
        }
    }
}
```

Let's try to understand the preceding code:

1. Create `DbContext`:
`SampleDbEntities db = new SampleDbEntities();`
2. Fetch the `Employer` by the given `employerId`:
`Employer employer = db.Employers.Find(employerId);`
3. Find the existing `Employee` by using the ID in the new `Employee` object:
`Employee employee = employer.Employees.FirstOrDefault(emp => emp.ID == employeeId);`
4. If the `Employee` is found, delete `Employee`:
`employer.Employees.Remove(employee);`
5. Save the changes. Please note that it is very important to call save changes since otherwise there will be no modifications to the data.
`db.SaveChanges();`



The preceding code will remove the records from the table. If we don't want to remove the records from the table and just want to remove the relationship between the entities, then we can assign null to the navigation properties and save the changes. This will remove the relation between the entities:

```
employer.Employees = null;  
db.SaveChanges();
```

Entity Framework – behind the scenes

Let's try to understand how Entity Framework used these navigation properties to retrieve the data. Let's look at the same example that we saw while retrieving Employer by using the Employee entity. In this scenario, our Employee entity contains a navigation property, Employer, using which we can access Employer related to Employee. The Employer entity also contains a navigation property, Employees, which will let us access the employees related to the given employer. So, if we need to access Employer related to Employee, it can be done as follows:

```
using (SampleDbEntities db = new SampleDbEntities())  
{  
    Employee employee = db.Employees.SingleOrDefault(  
        employeeId);  
    if (employee != null)  
    {  
        employer = employee.Employer;  
    }  
}
```

What happens behind the scenes is that Entity Framework generates SQL based on our queries and retrieves the data from the tables.

Let's see what Entity Framework is doing behind the scenes to make this work:

1. Analyze our object query. In our example, we are simply querying all the results, but this could very well be a complicated LINQ query as well.
2. Check the database mapping of the Employee entity.
3. Use the navigation property and find the database mappings of the Employer entity.
4. Use the one-to-many relationship to create an Inner Join statement.
5. Generate the appropriate SQL and retrieve the data.
6. Populate the retrieved data into the Employer and Employee models.

If we look at the generated SQL, it will look like this:

```
SELECT
  ..[Extent2].[ID] AS [ID],
  ..[Extent2].[EmployerName] AS [EmployerName]
FROM
  ..[dbo].[Employees] AS [Extent1]
INNER JOIN
  ..[dbo].[Employers] AS [Extent2] ON [Extent1].[EmployerId] =
    [Extent2].[ID]
WHERE
  ..([Extent1].[ID] = @p__linq__0) AND (@p__linq__0 IS NOT NULL)
```



To see the SQL query generated by Entity Framework, we can use a `ToString()` query on a `DbQuery` object. Unfortunately, this can only be done if we use the query syntax to retrieve the data. To look at the generated SQL for the preceding code, we first need to write the equivalent query syntax for this and then perform `ToString()` on that:

```
var _employer = from c in db.Employees
                 where c.ID == employeeId
                 select c.Employer;

var sql = _employer.ToString();
```

We can also use SQL Profiler to look at the Entity-Framework-generated queries.

Summary

In this chapter, we looked at how we can manage database table's relationships in our conceptual model using the Entity Framework Database First approach. We also looked at how Entity Framework keeps track of the relationships between entities and lets us use the relationships effectively by providing navigation properties in entities. We also saw how we can perform database operations in related entities using the navigation properties. In the next chapter, we will look at how we can perform model validations using Entity Framework to ensure that only valid values get persisted in the database.

3

Entity Framework DB First – Performing Model Validations

In this chapter, we will see how we can perform model validations using Entity Framework. We will take a look at the various approaches available for model validations, and implement a small sample application to validate the models using each approach. Finally, we will take a look at which approach should be used in which scenario. Model validations discussed in this chapter are not specific to the Database First approach. It is a functionality related to entities so it is applicable to all three approaches of Entity Framework.

Model validations using Entity Framework

Every application will have certain business rules associated to maintain the consistency of the data that is being saved in the database. It is our application's responsibility to perform the business rule validations while inserting or updating the data in a database.

Classic ADO.NET applications contain these validations in either the presentation layer or business logic layer. Using Entity Framework's validation features, we can perform model validations in the presentation and business logic layers. The best approach from the application perspective would be to use a combination of both presentation layer validations and business logic layer validations. Validations in the presentation layer will provide quick feedback to the user, and complex business rules can be validated in the business logic layer.

There are two ways in which we can perform model validations using Entity Framework:

- Model validations using partial class methods
- Model validations using data annotations

Each of these approaches has its pros and cons in usage and both can be used simultaneously based on the validation requirements. Let's take a look at both these approaches one by one and try to implement a sample application for each approach to understand the concepts in detail.

Model validations using partial class methods

When we use Entity Framework's Database First approach, Entity Framework is responsible for generating entities. To facilitate validations for these generated entities, which Entity Framework generates, it also creates partial methods that will be invoked when the entity's property value is changed. The methods are named `OnPropertyNameChanging`, and we can use these methods as hooks to perform custom validations whenever the property values are updated.

Before we start looking at this approach for validation, let's try to understand the concept of partial methods.

Understanding partial methods

In C#, it is possible to split class definitions across multiple files by creating partial classes. We can simply split the class definition across multiple places using the `partial` keyword. This feature is particularly useful when we are dealing with generated code, as we don't have to modify the generated code. However, we can still add custom code to the class, and it will work in unison with the generated code. The following code (`1003EN_03_1_code.txt`) shows how this can be implemented:

```
partial class Contact
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

partial class Contact
{
    public DateTime DateOfBirth { get; set; }
}
```

The concept of partial methods takes this approach one step further. It is possible to create partial methods in such a way that the method declaration can reside in one partial class and its definition can be put in another partial class.

The benefit of this feature is that one partial class can declare a method and specify where it will be called and another partial class can specify the definition such as the behavior when this function is called. So, if we try to modify the preceding `Contact` class to have a partial method that will be called every time the `LastName` value is changed, the code will look as follows:

```
partial class Contact
{
    public string FirstName { get; set; }

    private string m_LastName;
    public string LastName
    {
        get
        {
            return m_LastName;
        }
        set
        {
            OnLastNameChanging(value);
            m_LastName = value;
        }
    }

    partial void OnLastNameChanging(string newValue);
}
```

If we don't provide the definition for this partial method in any of the partial classes, the function call will be ignored. However, if we specify the definition of the partial method, it will be invoked whenever the method is called. Let us write the definition of the partial function `OnLastNameChanging` in our second partial class, and try to avoid setting values that are more than 50 characters in length:

```
partial class Contact
{
    public DateTime DateOfBirth { get; set; }

    partial void OnLastNameChanging(string newValue)
    {
        if (newValue.Length > 50)
```

```
        {  
            throw new Exception("LastName - length exceeded");  
        }  
    }  
}
```

These partial methods are very useful to perform custom validations using Entity Framework's Database First approach. The generated code will have the `OnPropertyChanging` partial method declarations, and the partial methods will be called whenever the property value changes. Let's see how we can use these partial methods to perform custom validations.



One important thing to note here is that the partial methods and the method calls will only be generated when we are using `EntityType` based generators. Entity Framework's default generator uses `POCO` classes, and this generator does not generate these partial functions. We can also modify the T4 generator files to achieve this.

To use partial functions, we have to use the `EntityType` generator, which can be done by removing the existing `tt` classes under the EDMX file, and then selecting **Add code generation item** from the context menu and adding `EF 6.x EntityType Generator`.

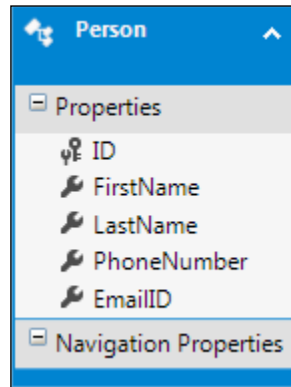
Using partial methods to perform model validations

Let's now take a look at how we can use these partial methods to perform model validations. Let's create a simple **People** table in the database. We will try to insert data in this table, and see how we can perform model validations before inserting the data in this table.

People	
<input type="checkbox"/>	ID
<input type="checkbox"/>	FirstName
<input type="checkbox"/>	LastName
<input type="checkbox"/>	PhoneNumber
<input type="checkbox"/>	EmailID
<input type="checkbox"/>	

Schema for the People table

Once the table is created, let's use Entity Framework's Database First approach with EF 6.x EntityObject Generator, and generate the Entity Data Model for this table. The generated entity for this table will look like the following screenshot.



Generated Entity model, Person, for the People table

If we take a look at the generated entity's code, we can see that the entity class contains partial methods such as `OnPropertyChanging` and `OnPropertyChanged`. For example, for the `LastName` property, it contains partial methods such as `OnLastNameChanging` and `OnLastNameChanged`.

The `OnPropertyChanging` partial method will be called before the value of the property is changed. This gives us the possibility of validating the proposed value before it is assigned. The `OnPropertyChanged` method will be called after the property value is changed. This is helpful when we need to take some action once the property of an entity is changed.

Let's implement a simple validation where we will try to validate the e-mail ID format to ensure that this field will only contain valid e-mail IDs. For this, we will implement the partial class for our model, **Person**, and inside this partial class, we will implement the `OnEmailIDChanging` partial method.

In this method, we will use a simplified regular expression to validate the format of the e-mail. If the e-mail is not in a proper format, we will raise an exception, which the UI layer can handle, and let the user know about the problem. Let's see how we can implement this partial method:

```
partial class Person
{
    partial void OnEmailIDChanging(string value)
    {
        bool isEmail = Regex.IsMatch(value, @"^\w+[a-zA-Z_]+?\.[a-zA-Z]{2,3}$", RegexOptions.IgnoreCase);
    }
}
```

```
        if (isEmail == false)
        {
            throw new ArgumentException
                ("Invalid email address");
        }
    }
}
```

In the preceding code, first, we implemented a partial class for our model `Person`. Inside this partial class, we implemented the `OnEmailIDChanging` partial method. Entity Framework will call this function whenever a new value is set for the `EmailID` property. We will use the regular expression based method, `Regex.IsMatch`, to check whether the new value is a valid e-mail or not. If the new value is not a valid e-mail, then we throw an exception stating that the e-mail supplied is in an invalid format.

If we don't want to throw an exception, in cases of invalid data, then we use another method to let the application know about the error by implementing `IValidatableObject` in our partial class. Property-change-based validation is like a trigger-based validation. It reacts immediately, and generally it validates a single property only. `IValidatableObject` is mainly used for multiple property validations, when there are business rules that are based on relationships between the properties:

```
partial class Person : IValidatableObject
{
    List<ValidationResult> validationErrors = new
        List<ValidationResult>();

    public IEnumerable<ValidationResult> Validate(ValidationContext
        validationContext)
    {
        return validationErrors;
    }
}
```

When we implement the `IValidatableObject` interface, we need to implement the `Validate` method. This `Validate` method will be called from the UI layer, and it will return any validation errors to the presentation layer. This method will also be called when we call the `SaveChanges` method on the `DbContext` class.

Now we need to hook this `Validate` method with our partial method validation. To do this, we need to add the error message in this `ValidationResult` collection from our partial method. So, if we implement our partial method to add the validation errors in this collection, our class will look like this:

```

partial class Person : IValidatableObject
{
    List<ValidationResult> validationErrors = new
    List<ValidationResult>();

    public IEnumerable<ValidationResult> Validate(ValidationContext
validationContext)
    {
        return validationErrors;
    }

    partial void OnEmailIDChanging(string value)
    {
        validationErrors.Clear();

        bool isEmail = Regex.IsMatch(value, @"^\w+[a-zA-Z_]+?\.[
a-zA-Z]{2,3}$", RegexOptions.IgnoreCase);

        if (isEmail == false)
        {
            ValidationResult vResult = new ValidationResult("Invalid
email address", new string[] { "EmailID" });
            validationErrors.Add(vResult);
        }
    }
}

```

In this new implementation, whenever we get invalid data in our partial method, we create an object of `ValidationResult`, and add it to the `ValidationResult` collection that is available at entity level. Whenever the application calls `validate`, the validation results will be passed to the application.



The first step in the `OnEmailIDChanging` method is to clear the collection that holds the errors. This is to remove the previous validation errors from the collection before returning.

From a best practices perspective, we should not even store the errors in the collection. We can directly yield them too.

One might ask why we need to put the validation logic in the entities, as this can easily be done in the UI layer too. Putting the validation logic there is useful because it decouples the entity validation logic from the UI layer. If our entity class is being used by many applications, then putting the validation logic inside the entities will ensure the consistency of validation rules across all the applications.



To decide what validations should be put in the entities and what validations should be put in UI, we should try to analyze whether the validation rule applies to individual properties of an entity or whether it checks values across multiple entities. If the validation rule applies to an individual property of an entity, it should be put inside the entity, otherwise it should be put on the layer above the entity, for example, the business logic layer or perhaps the UI layer.

Model validations using data annotations

We can also perform validations using data annotations on your entity class. This is even possible when Entity Framework is generating the entities. The major benefit of using data annotation-based validations is that it makes it very easy for us to perform validations by simply adding attributes to model properties. It is also helpful in scenarios where we want to use POCO generators with our Database First approach, where we will not have the partial methods made available to perform the validations.



Validations using data annotations can also be used in Entity Framework's Code First approach to perform model validations.

To perform model validations using data annotations, we need to add the validation attributes to our entity class. The first question that comes to mind now is how can we add the attributes to the entity classes since our entity classes are autogenerated by Entity Framework.

To add the validation attributes to our generated entity classes, we need to create a partial class for our generated entity. In this partial class, we have to add the `MetadataType` attribute and point this attribute at a class that is responsible for applying Data Annotations to our entity's properties.

Let's create a `MetadataType` class for our `Person` entity:

```
[MetadataType(typeof(PersonMetaData))]  
partial class Person  
{  
}  
class PersonMetaData  
{  
}
```

In the preceding code, we updated the `Person` class with the `MetaDataType` attribute and specified that all the data annotation attributes will be defined in the `PersonMetaData` class.



In the case of Entity Framework's Code First approach, we don't need to create this `MetaData Type` (`PersonMetaData`) class, as we can put the data annotations in the entity class itself.

Now that we have created a `MetaData Type` class, let's try to add some validation logic using the data annotation attributes.

Specifying validation rules using data annotations

Let's now see how we can use data annotation attributes to implement validation logic for our entity class. Let's try to implement the following validation rules in our `Person` entity:

- All the properties are required
- The length of each property should not exceed a given length
- `PhoneNumber` should contain only numbers
- `Email` should be in the proper e-mail format

Let's see how we can implement these rules in our `PersonMetaData` class.

Validating the required fields

Let's start by looking at how we can use data annotations to validate a required field. To validate a required field, we have to decorate the property with a `Required` attribute. Let's see how we do this with the properties of the `PersonMetaData` entity:

```
class PersonMetaData
{
    [Required(ErrorMessage = "First Name is required")]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Last Name is required")]
    public string LastName { get; set; }

    [Required(ErrorMessage = "Phone number is required")]
    public string PhoneNumber { get; set; }
}
```

```
        [Required(ErrorMessage = "Email ID is required")]
        public string Email { get; set; }
    }
```

The Required attribute also has a property called `ErrorMessage`, which can be used to specify the error message that should be passed to the application when the validation rule for the associated entity property fails.

Validating the length of fields

We can also validate the entity properties for maximum allowed length using the `StringLength` attribute. Let's try to put these `StringLength` validations in our `PersonMetaData` class:

```
class PersonMetaData
{
    [Required(ErrorMessage = "First Name is required")]
    [StringLength(100, ErrorMessage = "First Name length should be
less than 100")]

    public string FirstName { get; set; }

    [Required(ErrorMessage = "Last Name is required")]
    [StringLength(100, ErrorMessage = "Last Name length should be less
than 100")]

    public string LastName { get; set; }
    [Required(ErrorMessage = "Phone number is required")]
    [StringLength(15, ErrorMessage = "Phone number length should be
less than 15")]

    public string PhoneNumber { get; set; }

    [Required(ErrorMessage = "Email ID is required")]
    [StringLength(35, ErrorMessage = "eMail Length should be less than
35")]

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

The first argument in the `StringLength` attribute specifies the maximum length allowed for the property, and the `ErrorMessage` argument specifies the error that will be passed to the application if the validation rule for the associated property fails.

Regular expression-based validations

If we want to validate our entity properties with some regular expressions, we can use the `RegularExpression` attribute. Let's try to validate the `PhoneNumber` and `EmailID` properties using `RegularExpression` attribute:

```
class PersonMetaData
{
    [Required(ErrorMessage = "First Name is required")]
    [StringLength(100, ErrorMessage = "First Name length should be
less than 100")]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Last Name is required")]
    [StringLength(100, ErrorMessage = "Last Name length should be less
than 100")]
    public string LastName { get; set; }

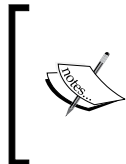
    [Required(ErrorMessage = "Phone number is required")]
    [StringLength(15, ErrorMessage = "Phone number length should be
less than 15")]
    [RegularExpression(@"^[0-9]{0,15}$", ErrorMessage = "Phone Number
should contain only numbers")]

    public string PhoneNumber { get; set; }

    [Required(ErrorMessage = "Email ID is required")]
    [StringLength(35, ErrorMessage = "eMail Length should be less than
35")]
    [RegularExpression(@"^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$",
ErrorMessage = "E Mail is not in proper format")]

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

Now the `PhoneNumber` and `Email` properties should also conform to the given regular expression. If they don't conform to the specified regular expression, the `ErrorMessage` will be propagated to the application.



Since the properties of our `PersonMetaData` class are specified with multiple attributes, all the validation rules will apply to the properties, that is, all the properties are now required and have certain maximum string length to conform to. Also, the `PhoneNumber` and `Email` fields should conform to the given regular expression too.

Triggering validations using data annotations

Once we have specified the data annotation attributes, we need to trigger them to validate the actual model. The trigger method will actually depend on the type of application that is using Entity Framework. There are two types of applications:

- Applications that support data binding environments and data annotations such as ASP.NET MVC and Silverlight applications
- Applications/frameworks that don't support data annotations explicitly such as Windows Forms, Web services, or any other .NET application

Let's see how we can trigger the validations in these types of application.

Trigger validations in data binding environments

In the case of applications that support data binding environments with data annotations support, we don't have to do anything explicitly to trigger the validations. The validations will get triggered automatically. These applications are configured to pull the validation rules automatically either from the entity class itself (Code First approach) or from another class that is associated with the entity class via the `MetadataType` attribute (Database First or Model First approach). In such applications, whenever the entity is being created or updated, the metadata will be pulled and in the case of validation failure, the error message will be passed to the UI layer automatically.

Trigger validations in non-data binding environments

If we are working on an application that does not support data binding environments, we need to explicitly trigger the validations. There are two scenarios based on where the data annotations are specified:

- The data annotation attributes are in the entity class itself (Code First approach)
- The data annotation attributes are in another class associated with the entity class via the `MetadataType` attribute

In the first case, where the entity class itself contains the data annotation attributes, we just need to do the following to explicitly trigger the validations for our `Person` entity:

```
Person person = new Person();  
  
ValidationContext context = new ValidationContext(person);
```

```
List<ValidationResult> results = new List<ValidationResult>();

bool isValid = Validator.TryValidateObject(person, context, results);
```

In the preceding code, we created a new object for `ValidationContext` that will be used to perform the validations. We then created a list of `ValidationResults` to hold the validation errors. Finally, we called the `Validator.TryValidateObject` function to trigger the validations. This function will return true if the validation is successful, otherwise it will return false. In case the validation fails, the validation errors will be found in the result, that is, the collection of `ValidationResult`.



In Entity Framework's Code First approach, the data annotation attributes will be specified in the entity class itself. So the preceding code will suffice to trigger the validations.

The challenge lies in triggering the validation rules in the second scenario where the data annotation attributes are defined in the `MetaData` class and not in the actual model.

In this scenario, we need to somehow let `ValidationContext` know that the data annotations are specified in another class. This can be done by letting `TypeDescriptor` know about the class that contains the data annotation attributes, as follows:

```
var descProvider = new AssociatedMetadataTypeTypeDescriptionProvider(
    typeof(Person), typeof(PersonMetaData));
TypeDescriptor.AddProviderTransparent(descProvider, typeof(Person));
```

In the preceding code, we created the `AssociatedMetadataTypeTypeDescriptionProvider` object to create a mapping that tells the `PersonMetaData` class contains the data annotation attributes for the `Person` class. This provider is then passed to `TypeDescriptor` to let the data annotations `ValidationContext` know that the validation attributes for the `Person` entity should be used from the `PersonMetaData` class.



This needs to be done only once when the application is starting.

Once this is done we can use `ValidationContext` to explicitly trigger the validations, as shown in the previous section.

Implementing custom validations using data annotations

We can use data annotations for validating a `Required` field, having a string conform to a maximum length using the `StringLength` attribute, and regular expression-based validations using the `RegularExpression` attribute. There are a few other data attributes such as `EnumDataTypeAttribute` that is used to map an enum value to a property and ultimately to the database column, `DataTypeAttribute`, to map a data type to the database column, and `RangeAttribute` that can be used to specify the range constraints for the value of a data field. From the usage perspective, they are quite similar to the ones we saw earlier and thus we will not discuss them in detail here.

What if we need to perform some custom validation along with these three validations?

Partial methods provide you with greater flexibility to perform custom validations, but the problem with the partial method validation technique is that it will not be available if we are using POCO generators with our Entity Framework's Database First or Model First approach, or if we are using Code First approach. In this case, the only option available to perform custom validations is writing custom data annotation attributes.

Let's see how we can perform custom validations for entity properties using data annotations, that is, implementing custom data annotation validation attributes. Let's say we want to implement a custom validation where we will check whether the e-mail provided by the user is already present in the system or not. If the e-mail is already present, we will raise a validation error.

To implement a custom validation attribute, we need to create a class that is derived from the `ValidationAttribute` abstract class. We need to implement the `IsValid` method in this class. This `IsValid` method will be called whenever the validations are triggered. Let's write a class `UniqueEmailAttribute` that will contain our custom validation logic:

```
public class UniqueEmailAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext validationContext)
    {
        string email = value as string;

        if (email != null)
        {
            using (SampleDbEntities db = new SampleDbEntities())
            {
```

```

        var personWithEmailFound = db.People.Any(item => item.
Email == email);
        if (personWithEmailFound == true)
        {
            return new ValidationResult("This email already
exists in the system.");
        }
    }
}

return ValidationResult.Success;
}
}

```

In the preceding code, we retrieved the new value of email from the value parameter. Then we checked whether there are any records in the database with this email. If we are able to find any record in the database with this email, we raise a validation error that this email is already present in the system.

This attribute will not work properly because when we check whether email already exists in the system, the result of the entity being updated will also be returned. Since the e-mail ID is found in the entity itself, which we are trying to update, we will get a validation error. To circumvent this, we need to check which entity is being updated:

```

Person person = validationContext.ObjectInstance as
Person;

```

However, while checking whether the e-mail exists in the system or not, we need to exclude this entity. This can be done as follows:



```

var personWithEmailFound =
    db.People.Any(item => item.Email == email && item.
ID != person.ID);
if (personWithEmailFound == true)
{
    return new ValidationResult("This email already
exists in the system.");
}

```

This way we can ensure that our custom validation will work in an update scenario also.

Now, to use this custom validation, we just need to update our entity's property with this attribute. So the Email property in our PersonMetaData class will look like the following after implementing this custom validation attribute:

```
[Required(ErrorMessage = "Email ID is required")]
[StringLength(35, ErrorMessage = "eMail Length should be less than 35")]
[RegularExpression(@"^\w+@[a-zA-Z_]+?\.[a-zA-Z]{2,3}$", ErrorMessage = "E Mail is not in proper format")]
[UniqueEmail]
public string Email { get; set; }
```

Now when the model validations are triggered our custom validation logic will also be executed.



We can also specify the error message while updating the EmailID property of the PersonMetaData class, because the base class already defines it:

```
[UniqueEmail(ErrorMessage = "Email ID already exists in the system.")]
public string EmailID { get; set; }
```

To use this error message, we just need to use this property while creating the ValidationResult object in our attribute class:

```
return new ValidationResult(this.ErrorMessage);
```

Summary

In this chapter, we looked at a few ways of performing validations for generated entities using partial methods and data annotations. We have also seen how we can trigger the validations in non-data binding environments, and how we can write custom data annotation attributes to perform custom validations. The data annotation-based validations are very useful in the usual validation scenarios, and we should try to use them for those. In case we need custom validations, we can do this using the partial method approach.

In the next chapter, we will take a look at advanced domain modeling techniques using Entity Framework's Database First approach using an inheritance relationship between entities.

4

Entity Framework DB First – Inheritance Relationships between Entities

So far, we have seen how we can use various approaches of Entity Framework, how we can manage database table relationships, and how to perform model validations using Entity Framework. In this chapter, we will see how we can implement the inheritance relationship between the entities. We will see how we can change the generated conceptual model to implement the inheritance relationship, and how it will benefit us in using the entities in an object-oriented manner and the database tables in a relational manner.

Domain modeling using inheritance in Entity Framework

One of the major challenges while using a relational database is to manage the domain logic in an object-oriented manner when the database itself is implemented in a relational manner. ORMs like Entity Framework provide the strongly typed objects, that is, entities for the relational tables. However, it might be possible that the entities generated for the database tables are logically related to each other, and they can be better modeled using inheritance relationships rather than having independent entities.

Entity Framework lets us create inheritance relationships between the entities, so that we can work with the entities in an object-oriented manner, and internally, the data will get persisted in the respective tables. Entity Framework provides us three ways of object relational domain modeling using the inheritance relationship:

- The **Table per Type (TPT)** inheritance
- The **Table per Class Hierarchy (TPH)** inheritance
- The **Table per Concrete Class (TPC)** inheritance

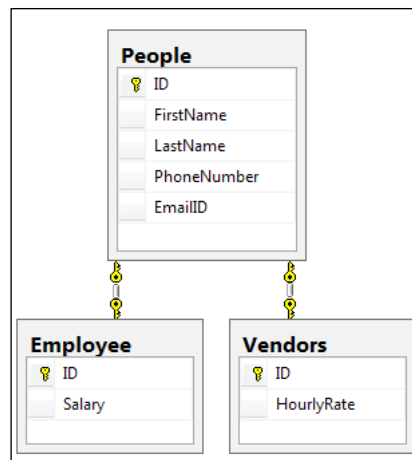
Let's now take a look at the scenarios where the generated entities are not logically related, and how we can use these inheritance relationships to create a better domain model by implementing inheritance relationships between entities using the Entity Framework Database First approach.

The Table per Type inheritance


The **Table per Type (TPT)** inheritance is useful when our database has tables that are related to each other using a one-to-one relationship. This relation is being maintained in the database by a shared primary key. To illustrate this, let's take a look at an example scenario.

Let's assume a scenario where an organization maintains a database of all the people who work in a department. Some of them are employees getting a fixed salary, and some of them are vendors who are hired at an hourly rate. This is modeled in the database by having all the common data in a table called `Person`, and there are separate tables for the data that is specific to the employees and vendors.

Let's visualize this scenario by looking at the database schema:



The database schema showing the TPT inheritance database schema

 The ID column for the `People` table can be an auto-increment identity column, but it should not be an auto-increment identity column for the `Employee` and `Vendors` tables.

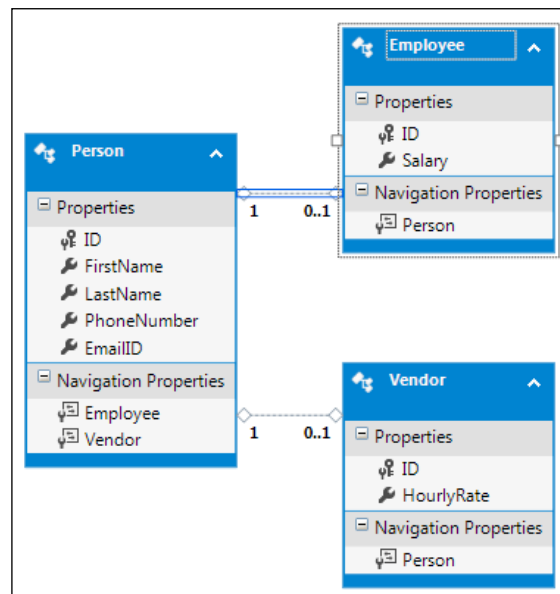
In the preceding figure, the `People` table contains all the data common to both type of worker. The `Employee` table contains the data specific to the employees and the `Vendors` table contains the data specific to the vendors. These tables have a shared primary key and thus, there is a one-to-one relationship between the tables.

To implement the TPT inheritance, we need to perform the following steps in our application:

1. Generate the default Entity Data Model.
2. Delete the default relationships.
3. Add the inheritance relationship between the entities.
4. Use the entities via the `DbContext` object.

Generating the default Entity Data Model

Let's add a new ADO.NET Entity Data Model to our application, and generate the conceptual Entity Model for these tables. The default generated Entity Model will look like this:



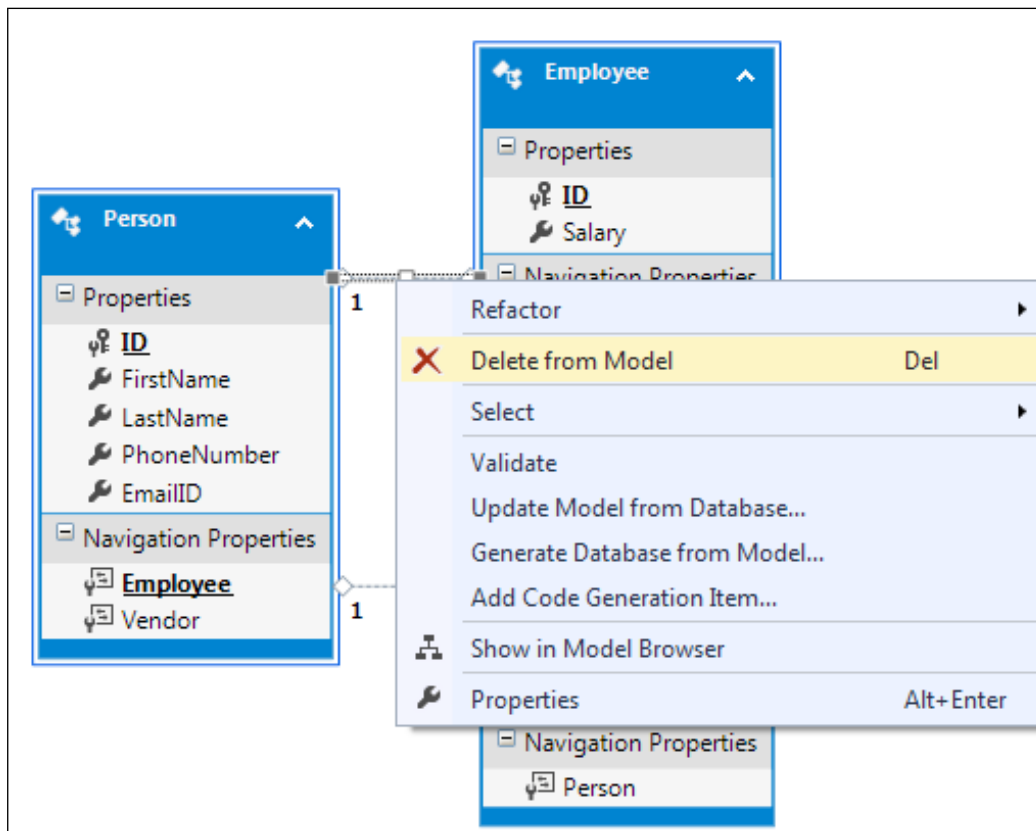
The generated Entity Data Model where the TPT inheritance could be used

Looking at the preceding conceptual model, we can see that Entity Framework is able to figure out the one-to-one relationship between the tables and creates the entities with the same relationship.

However, if we take a look at the generated entities from our application domain perspective, it is fairly evident that these entities can be better managed if they have an inheritance relationship between them. So, let's see how we can modify the generated conceptual model to implement the inheritance relationship, and Entity Framework will take care of updating the data in the respective tables.

Deleting default relationships

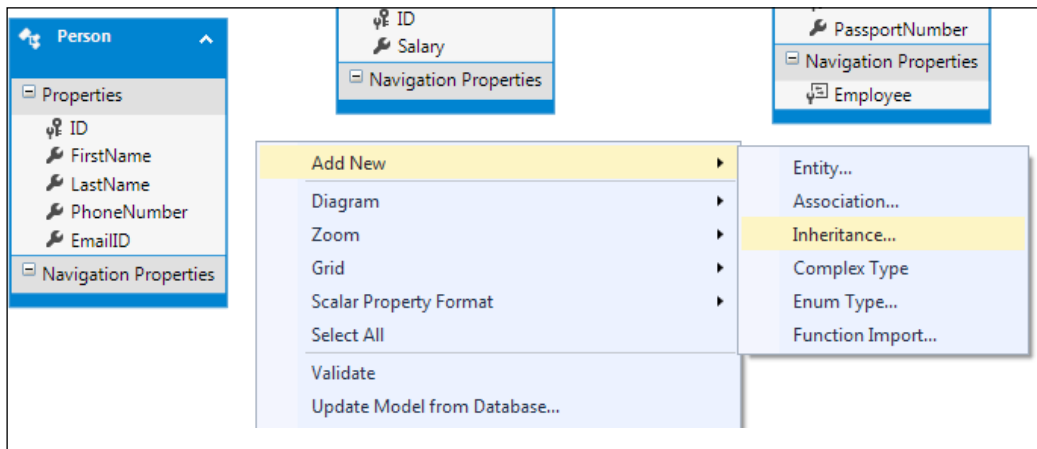
The first thing we need to do to create the inheritance relationship is to delete the existing relationship from the Entity Model. This can be done by right-clicking on the relationship and selecting **Delete from Model** as follows:



Deleting an existing relationship from the Entity Model

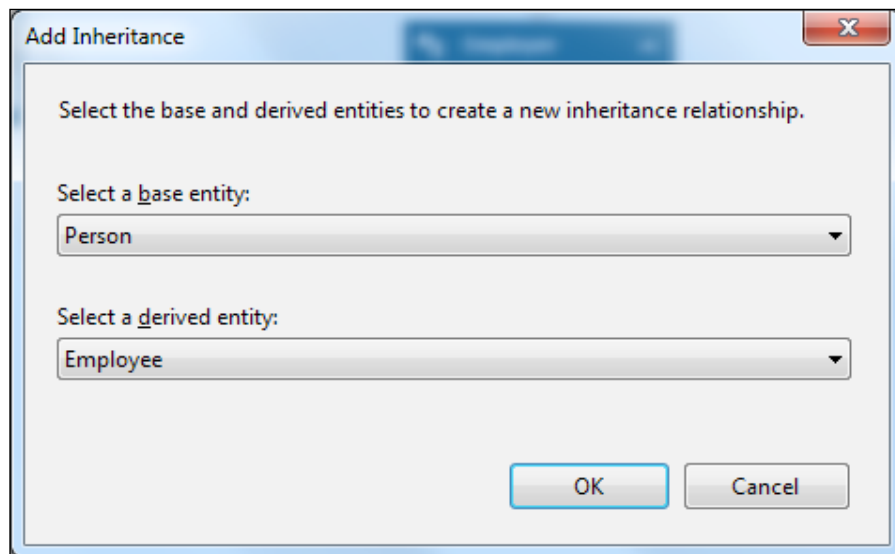
Adding inheritance relationships between entities

Once the relationships are deleted, we can add the new inheritance relationships in our Entity Model as follows:



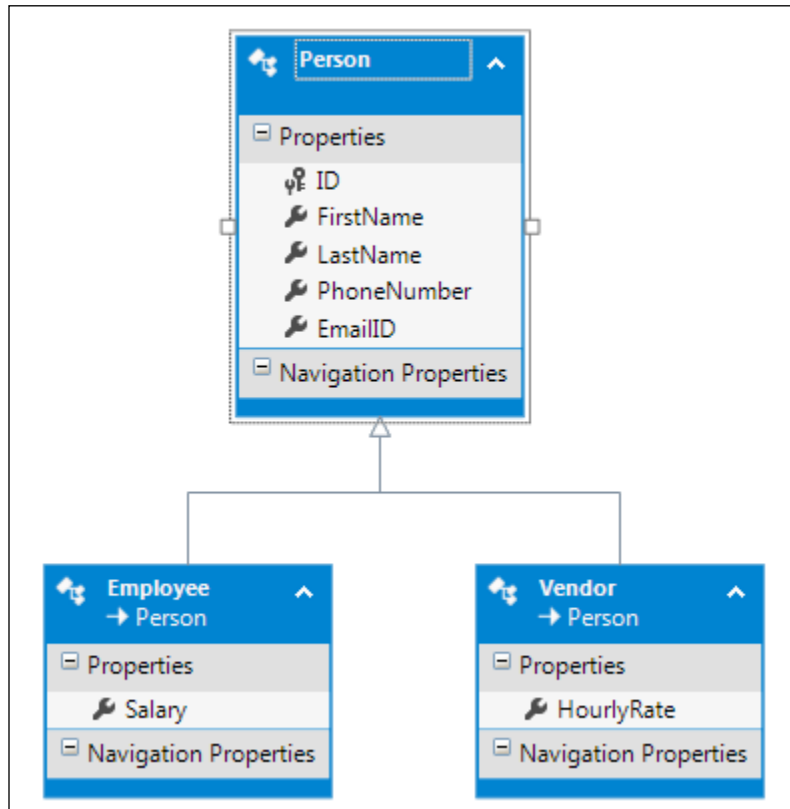
Adding inheritance relationships in the Entity Model

When we add an inheritance relationship, the Visual Entity Designer will ask for the base class and derived class as follows:



Selecting the base class and derived class participating in the inheritance relationship

Once the inheritance relationship is created, the Entity Model will look like this:



Inheritance relationship in the Entity Model



After creating the inheritance relationship, we will get a compile error that the ID property is defined in all the entities. To resolve this problem, we need to delete the ID column from the derived classes. This will still keep the ID column that maps the derived classes as it is.

So, from the application perspective, the ID column is defined in the base class but from the mapping perspective, it is mapped in both the base class and derived class, so that the data will get inserted into tables mapped in both the base and derived entities.

With this inheritance relationship in place, the entities can be used in an object-oriented manner, and Entity Framework will take care of updating the respective tables for each entity.

Using the entities via the DbContext object

As we know, DbContext is the primary class that should be used to perform various operations on entities. Let's try to use our SampleDbContext class to create an Employee and a Vendor using this Entity Model and see how the data gets updated in the database:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Employee employee = new Employee();
    employee.FirstName = "Employee 1";
    employee.LastName = "Employee 1";
    employee.PhoneNumber = "1234567";
    employee.Salary = 50000;
    employee.EmailID = "employee1@test.com";

    Vendor vendor = new Vendor();
    vendor.FirstName = "vendor 1";
    vendor.LastName = "vendor 1";
    vendor.PhoneNumber = "1234567";
    vendor.HourlyRate = 100;
    vendor.EmailID = "vendor1@test.com";

    db.People.Add(employee);
    db.People.Add(vendor);
    db.SaveChanges();
}
```


In the preceding code, what we are doing is creating an object of the `Employee` and `Vendor` type, and then adding them to `People` using the `DbContext` object. What Entity Framework will do internally is that it will look at the mappings of the base entity and the derived entities, and then push the respective data into the respective tables. So, if we take a look at the data inserted in the database, it will look like the following:

People: Query(hf...ENTS\TODODB.MDF) X					
	ID	FirstName	LastName	PhoneNumber	EmailID
	2	Employee 1	Employee 1	1234567	employee1@test.com
	3	vendor 1	vendor 1	1234567	vendor1@test.com

Employee: Query...ENTS\TODODB.MDF) X		
	ID	Salary
	2	50000

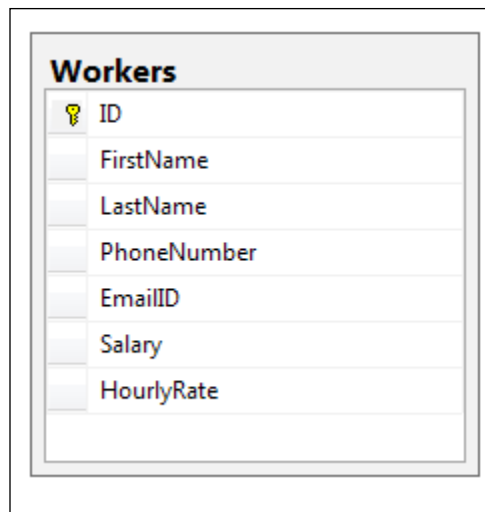
Vendors: Query(h...ENTS\TODODB.MDF) X		
	ID	HourlyRate
	3	100

A database snapshot of the inserted data

It is clearly visible from the preceding database snapshot that Entity Framework looks at our inheritance relationship and pushes the data into the `Person`, `Employee`, and `Vendor` tables.

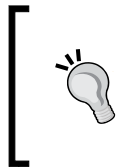
The Table per Class Hierarchy inheritance

The **Table per Class Hierarchy (TPH)** inheritance is modeled by having a single database table for all the entity classes in the inheritance hierarchy. The TPH inheritance is useful in cases where all the information about the related entities is stored in a single table. For example, using the earlier scenario, let's try to model the database in such a way that it will only contain a single table called `Workers` to store the `Employee` and `Vendor` details. Let's try to visualize this table:



A database schema showing the TPH inheritance database schema

Now what will happen in this case is that the common fields will be populated whenever we create a type of worker. `Salary` will only contain a value if the worker is of type `Employee`. The `HourlyRate` field will be null in this case. If the worker is of type `Vendor`, then the `HourlyRate` field will have a value, and `Salary` will be null.



This pattern is not very elegant from a database perspective. Since we are trying to keep unrelated data in a single table, our table is not normalized. There will always be some redundant columns that contain null values if we use this approach. We should try not to use this pattern unless it is absolutely needed.

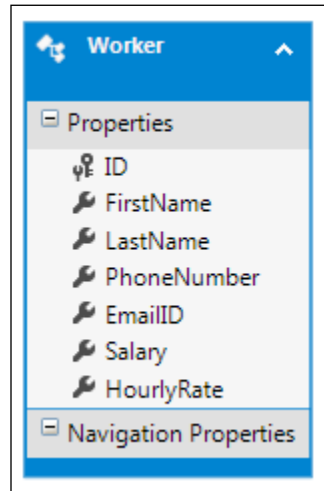
To implement the TPH inheritance relationship using the preceding table structure, we need to perform the following activities:

1. Generate the default Entity Data Model.
2. Add concrete classes to the Entity Data Model.
3. Map the concrete class properties to their respective tables and columns.
4. Make the base class entity abstract.
5. Use the entities via the `DbContext` object.

Let's discuss this in detail.

Generating the default Entity Data Model

Let's now generate the Entity Data Model for this table. The Entity Framework will create a single entity, `Worker`, for this table:



The generated model for the table created for implementing the TPH inheritance

Adding concrete classes to the Entity Data Model

From the application perspective, it would be a much better solution if we have classes such as `Employee` and `Vendor`, which are derived from the `Worker` entity. The `Worker` class will contain all the common properties, and `Employee` and `Vendor` will contain their respective properties. So, let's add new entities for `Employee` and `Vendor`. While creating the entity, we can specify the base class entity as `Worker`, which is as follows:

Add Entity

Properties

Entity name:
Employee

Base type:
Worker

Entity Set:
Workers

Key Property

☐ Create key property

Property name:
Id

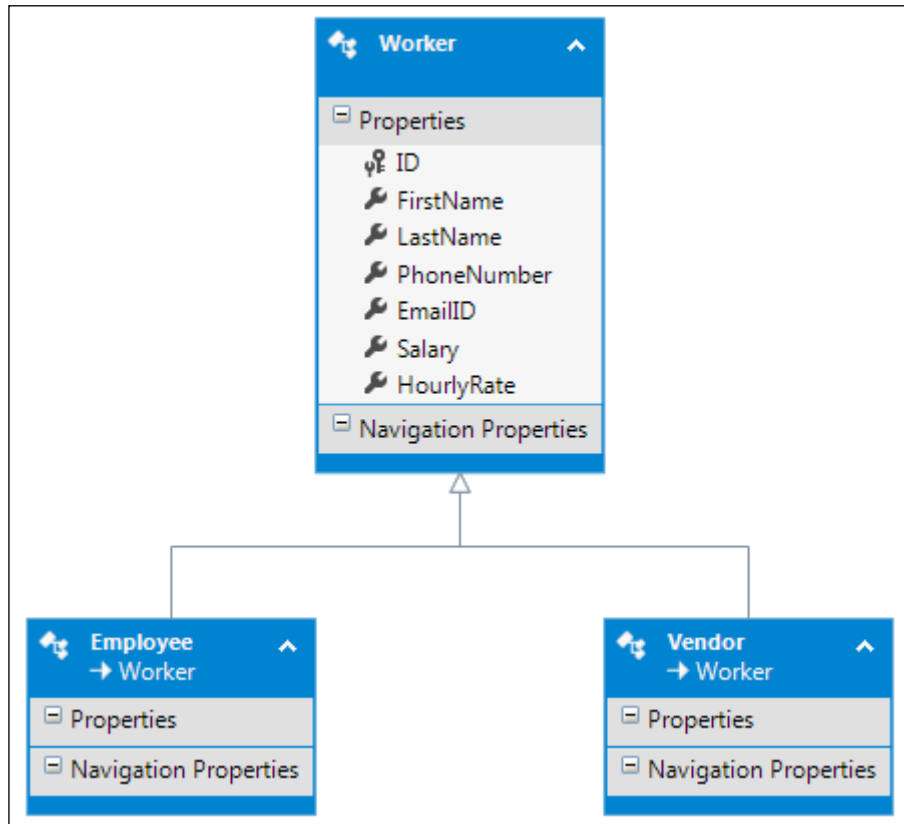
Property type:
Int32

OK Cancel

Adding a new entity in the Entity Data Model using a base class type

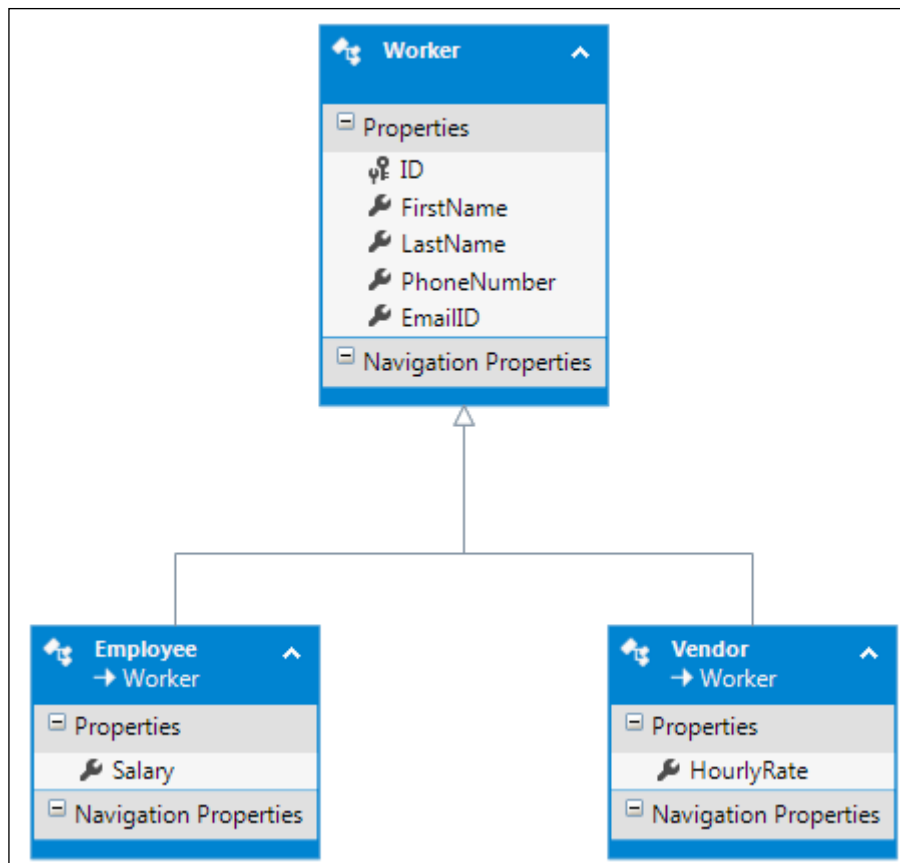
Similarly, we will add the `Vendor` entity to our Entity Data Model, and specify the `Worker` entity as its base class entity.

Once the entities are generated, our conceptual model will look like this:



The Entity Data Model after adding the derived entities

Next, we have to remove the **Salary** and **HourlyRate** properties from the **Worker** entity, and put them in the **Employee** and the **Vendor** entities respectively. So, once the properties are put into the respective entities, our final Entity Data model will look like this:



The Entity Data Model after moving the respective properties into the derived entities

Mapping the concrete class properties to the respective tables and columns

After this, we have to define the column mappings in the derived classes to let the derived classes know which table and column should be used to put the data. We also need to specify the mapping condition.

The Employee entity should save the Salary property's value in the Salary column of the Workers table when the Salary property is Not Null and HourlyRate is Null:

Table mapping and conditions to map the Employee entity to the respective tables

Once this mapping is done, we have to mark the Salary property as `Nullable=false` in the entity property window. This will let Entity Framework know that if someone is creating an object of the Employee type, then the Salary field is mandatory:

Setting the Employee entity properties as Nullable

Similarly, the Vendor entity should save the HourlyRate property's value in the HourlyRate column of the Workers table when Salary is Null and HourlyRate is Not Null:










Mapping Details - Vendor			
	Column	Operator	Value / Property
	└─ Tables		
	└─  Maps to Workers		
	└─ When Salary	Is	Null
	└─ And HourlyRate	Is	Not Null
	└─ <Add a Condition>		
	└─  Column Mappings		
	└─  Salary : decimal	↔	
	└─  HourlyRate : decimal	↔	 HourlyRate : Decimal
└─  <Add a Table or View>			

Table mapping and conditions to map the Vendor entity to the respective tables

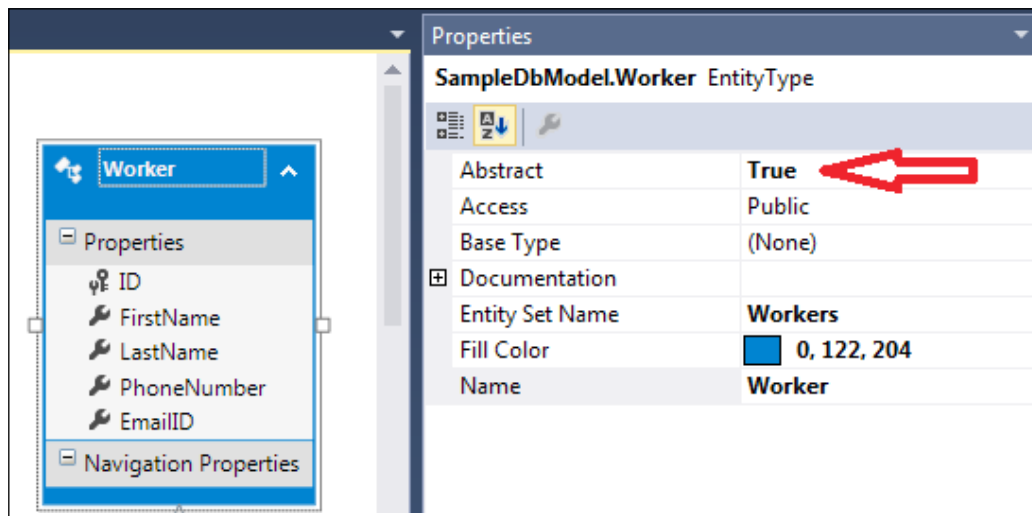
And similar to the Employee class, we also have to mark the HourlyRate property as Nullable=false in the Entity Property window. This will help Entity Framework know that if someone is creating an object of the Vendor type, then the HourlyRate field is mandatory:

Properties	
SampleDbModel.Employee.Salary Property	
Concurrency Mode	None
Default Value	(None)
Documentation	
Entity Key	False
Getter	Public
Name	Salary
Nullable	False ←
Precision	18
Scale	0
Setter	Public
StoreGeneratedPattern	None
Type	Decimal

Setting the Vendor entity properties to Nullable

Making the base class entity abstract

There is one last change needed to be able to use these models. To be able to use these models, we need to mark the base class as abstract, so that Entity Framework is able to resolve the object of `Employee` and `Vendors` to the `Workers` table.



Making the base class Workers as abstract

This will also be a better model from the application perspective because the `Worker` entity itself has no meaning from the application domain perspective.

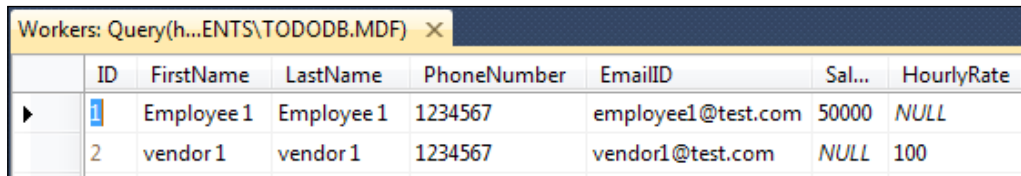
Using the entities via the DbContext object

Now we have our Entity Data Model configured to use the TPH inheritance. Let's try to create an `Employee` object and a `Vendor` object, and add them to the database using the TPH inheritance hierarchy:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Employee employee = new Employee();
    employee.FirstName = "Employee 1";
    employee.LastName = "Employee 1";
    employee.PhoneNumber = "1234567";
    employee.Salary = 50000;
    employee.EmailID = "employee1@test.com";
```

```
Vendor vendor = new Vendor();  
vendor.FirstName = "vendor 1";  
vendor.LastName = "vendor 1";  
vendor.PhoneNumber = "1234567";  
vendor.HourlyRate = 100;  
vendor.EmailID = "vendor1@test.com";  
  
db.Workers.Add(employee);  
db.Workers.Add(vendor);  
db.SaveChanges();  
}
```

In the preceding code, we created objects of the `Employee` and `Vendor` types, and then added them to the `Workers` collection using the `DbContext` object. Entity Framework will look at the mappings of the base entity and the derived entities, will check the mapping conditions and the actual values of the properties, and then push the data to the respective tables. So, let's take a look at the data inserted in the `Workers` table:



The screenshot shows a SQL Server query window titled "Workers: Query(h...ENTS\TODODB.MDF) ×". The query results are displayed in a table with 8 columns: ID, FirstName, LastName, PhoneNumber, EmailID, Sal..., and HourlyRate. There are two rows of data. The first row has ID 1, FirstName Employee 1, LastName Employee 1, PhoneNumber 1234567, EmailID employee1@test.com, Sal... 50000, and HourlyRate NULL. The second row has ID 2, FirstName vendor 1, LastName vendor 1, PhoneNumber 1234567, EmailID vendor1@test.com, Sal... NULL, and HourlyRate 100.

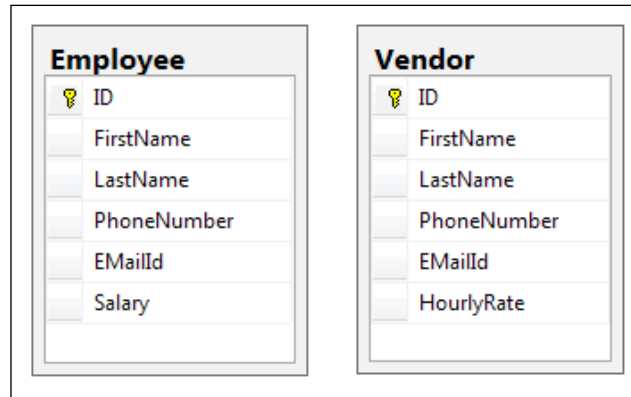
	ID	FirstName	LastName	PhoneNumber	EmailID	Sal...	HourlyRate
▶	1	Employee 1	Employee 1	1234567	employee1@test.com	50000	NULL
	2	vendor 1	vendor 1	1234567	vendor1@test.com	NULL	100

A database snapshot after inserting the data using the `Employee` and `Vendor` entities

So, we can see that for our `Employee` and `Vendor` models, the actual data is being kept in the same table using Entity Framework's TPH inheritance.

The Table per Concrete Class inheritance

The **Table per Concrete Class (TPC)** inheritance can be used when the database contains separate tables for all the logical entities, and these tables have some common fields. In our existing example, if there are two separate tables of `Employee` and `Vendor`, then the database schema would look like the following:



The database schema showing the TPC inheritance database schema

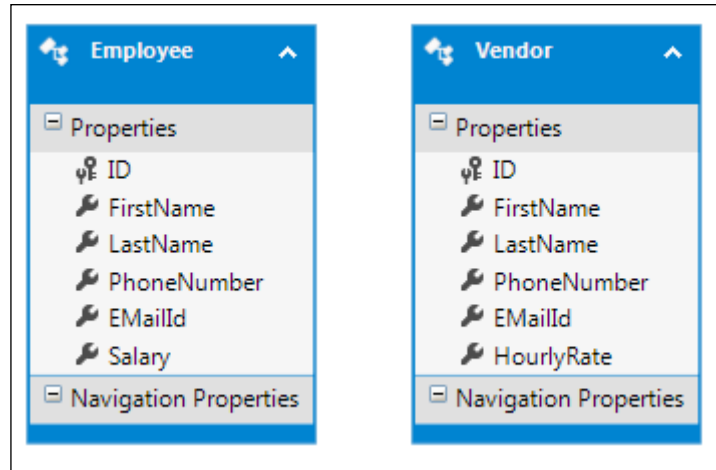
One of the major problems in such a database design is the duplication of columns in the tables, which is not recommended from the database normalization perspective.

To implement the TPC inheritance, we need to perform the following tasks:

1. Generate the default Entity Data Model.
2. Create the abstract class.
3. Modify the CDSL to cater to the change.
4. Specify the mapping to implement the TPT inheritance.
5. Use the entities via the `DBContext` object.

Generating the default Entity Data Model

Let's now take a look at the generated entities for this database schema:

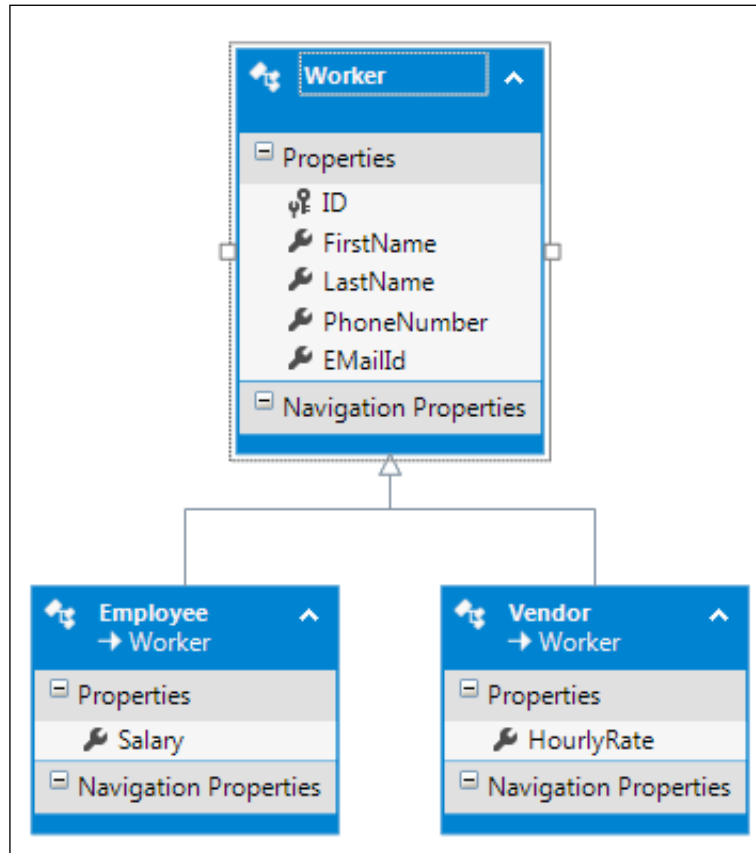


The default generated entities for the TPC inheritance database schema

Entity Framework has given us separate entities for these two tables. From our application domain perspective, we can use these entities in a better way if all the common properties are moved to a common abstract class. The `Employee` and `Vendor` entities will contain the properties specific to them and inherit from this abstract class to use all the common properties.

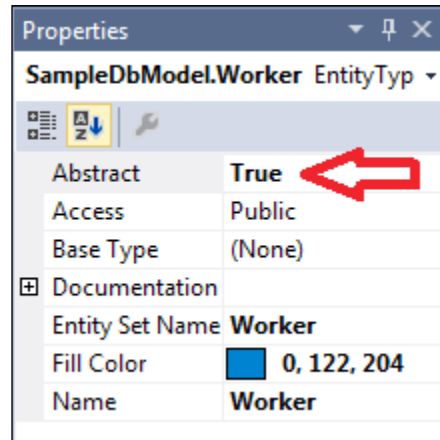
Creating the abstract class

Let's add a new entity called `Worker` to our conceptual model and move the common properties into this entity:



Adding a base class for all the common properties

Next, we have to mark this class as abstract from the properties window:



Marking the base class as abstract class

Modifying the CDSL to cater to the change

Next, we have to specify the mapping for these tables. Unfortunately, the Visual Entity Designer has no support for this type of mapping, so we need to perform this mapping ourselves in the EDMX XML file.

The **conceptual schema definition language (CSDL)** part of the EDMX file is all set since we have already moved the common properties into the abstract class. So, now we should be able to use these properties with an abstract class handle. The problem will come in the **storage schema definition language (SSDL)** and **mapping specification language (MSL)**.

The first thing that we need to do is to change the SSDL to let Entity Framework know that the abstract class *Worker* is capable of saving the data in two tables. This can be done by setting the *EntitySet* name in the *EntityContainer* tags as follows:

```
<EntityContainer Name="todoDbModelStoreContainer">
  <EntitySet Name="Employee" EntityType="Self.Employee" Schema="dbo"
store:Type="Tables" />
  <EntitySet Name="Vendor" EntityType="Self.Vendor" Schema="dbo"
store:Type="Tables" />
</EntityContainer>
```

Specifying the mapping to implement the TPT inheritance

Next, we need to change the MSL to properly map the properties to the respective tables based on the actual type of object. For this, we have to specify `EntitySetMapping`. The `EntitySetMapping` should look like the following:

```
<EntityContainerMapping StorageEntityContainer="todoDbModelStoreContainer" CdmEntityContainer="SampleDbEntities">
  <EntitySetMapping Name="Workers">
    <EntityTypeMapping TypeName="IsTypeOf(SampleDbModel.Vendor)">
      <MappingFragment StoreEntitySet="Vendor">
        <ScalarProperty Name="HourlyRate" ColumnName="HourlyRate" />
        <ScalarProperty Name="EMailId" ColumnName="EMailId" />
        <ScalarProperty Name="PhoneNumber" ColumnName="PhoneNumber" />
        <ScalarProperty Name="LastName" ColumnName="LastName" />
        <ScalarProperty Name="FirstName" ColumnName="FirstName" />
        <ScalarProperty Name="ID" ColumnName="ID" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
  <EntitySetMapping Name="Employees">
    <EntityTypeMapping TypeName="IsTypeOf(SampleDbModel.Employee)">
      <MappingFragment StoreEntitySet="Employee">
        <ScalarProperty Name="ID" ColumnName="ID" />
        <ScalarProperty Name="Salary" ColumnName="Salary" />
        <ScalarProperty Name="EMailId" ColumnName="EMailId" />
        <ScalarProperty Name="PhoneNumber" ColumnName="PhoneNumber" />
        <ScalarProperty Name="LastName" ColumnName="LastName" />
        <ScalarProperty Name="FirstName" ColumnName="FirstName" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
</EntityContainerMapping>
```

In the preceding code, we specified that if the actual type of object is `Vendor`, then the properties should map to the columns in the `Vendor` table, and if the actual type of entity is `Employee`, the properties should map to the `Employee` table, as shown in the following screenshot:

Mapping Details - Employee		
Column	Operator	Value / Property
<div> <div>Tables</div> <div> <div>Maps to Employee</div> <div><Add a Condition></div> <div>Column Mappings</div> <div> <div>ID : int ↔ ID : Int32</div> <div>FirstName : nvarchar ↔ FirstName : String</div> <div>LastName : nvarchar ↔ LastName : String</div> <div>PhoneNumber : nvarchar ↔ PhoneNumber : String</div> <div>EMailId : nvarchar ↔ EMailId : String</div> <div>Salary : decimal ↔ Salary : Decimal</div> <div><Add a Table or View></div> </div> </div> </div>		

After EDMX modifications, the mapping are visible in Visual Entity Designer



If we now open the EDMX file again, we can see the properties being mapped to the respective tables in the respective entities. Doing this mapping from Visual Entity Designer is not possible, unfortunately.

Using the entities via the DbContext object

Let's use these entities from our code:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Employee employee = new Employee();
    employee.FirstName = "Employee 1";
    employee.LastName = "Employee 1";
    employee.PhoneNumber = "1234567";
    employee.Salary = 50000;
    employee.EMailId = "employee1@test.com";

    Vendor vendor = new Vendor();
    vendor.FirstName = "vendor 1";
    vendor.LastName = "vendor 1";
    vendor.PhoneNumber = "1234567";
    vendor.HourlyRate = 100;
    vendor.EMailId = "vendor1@test.com";

    db.Workers.Add(employee);
    db.Workers.Add(vendor);
    db.SaveChanges();
}
```

In the preceding code, we created objects of the `Employee` and `Vendor` types and saved them using the `Workers` entity set, which is actually an abstract class. If we take a look at the inserted database, we will see the following:

	ID	FirstName	LastName	PhoneNumber	EMailId	HourlyRate
►	1	vendor 1	vendor 1	1234567	vendor1@test.c...	100
*	NULL	NULL	NULL	NULL	NULL	NULL
	ID	FirstName	LastName	PhoneNumber	EMailId	Salary
►	1	Employee 1	Employee 1	1234567	employee1@t...	50000
*	NULL	NULL	NULL	NULL	NULL	NULL

Database snapshot of the inserted data using TPC inheritance

From the preceding screenshot, it is clear that the data is being pushed to the respective tables.



The insert operation we saw in the previous code is successful but there will be an exception in the application. This exception is because when Entity Framework tries to access the values that are in the abstract class, it finds two records with same ID, and since the ID column is specified as a primary key, two records with the same value is a problem in this scenario. This exception clearly shows that the store/database generated identity columns will not work with the TPC inheritance.

If we want to use the TPC inheritance, then we either need to use GUID based IDs, or pass the ID from the application, or perhaps use some database mechanism that can maintain the uniqueness of auto-generated columns across multiple tables.

Choosing the inheritance strategy

Now that we know about all the inheritance strategies supported by Entity Framework, let's try to analyze these approaches. The most important thing is that there is no single strategy that will work for all the scenarios. Especially if we have a legacy database. The best option would be to analyze the application requirements and then look at the existing table structure to see which approach is best suited.

The Table per Class Hierarchy inheritance tends to give us denormalized tables and have redundant columns. We should only use it when the number of properties in the derived classes is very less, so that the number of redundant columns is also less, and this denormalized structure will not create problems over a period of time.

Contrary to TPH, if we have a lot of properties specific to derived classes and only a few common properties, we can use the Table per Concrete Class inheritance. However, in this approach, we will end up with some properties being repeated in all the tables. Also, this approach imposes some limitations such as we cannot use auto-increment identity columns in the database.

If we have a lot of common properties that could go into a base class and a lot of properties specific to derived classes, then perhaps Table per Type is the best option to go with.



In any case, complex inheritance relationships that become unmanageable in the long run should be avoided. One alternative could be to have separate domain models to implement the application logic in an object-oriented manner, and then use mappers to map these domain models to Entity Framework's generated entity models.

Summary

In this chapter, we looked at the various types of inheritance relationship using Entity Framework. We saw how these inheritance relationships can be implemented, and some guidelines on which should be used in which scenario. In the next chapter, we will see how we can use stored procedures and functions with Entity Framework.

5

Entity Framework DB First – Using Views, Stored Procedures, and Functions

So far, we have seen how we can use Entity Framework with an existing database, perform domain modeling, validate the data that is being pushed into the database, and use Entity Framework inheritance to manage the entities in an object-oriented manner. One question that is still unanswered is how we can reuse the code/logic (views, stored procedures, and functions) that exists in the data layer (database). In this chapter, we will take a look at how we can use Entity Framework to access the view, stored procedures, and functions defined in the database.

Using views, procedures, and functions

In some organizations, the policies dictate whether all data access should be done using the procedures, functions, and views defined in the database. Another possibility is that our database is being used by multiple applications, and to ensure that every task is performed in the same manner by all the applications, the database-related tasks are defined in the data layer in the following forms:

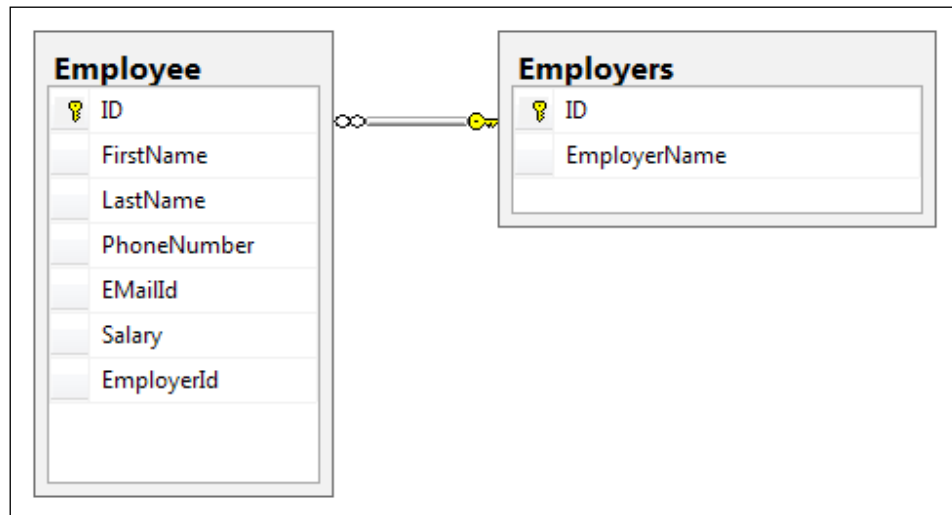
- Views
- Stored procedures
- Scalar functions
- Table valued functions

Entity Framework makes it very easy for the application to use views, stored procedures, and functions defined in the database. Let's see how we can use Entity Framework to perform these activities.

Using Entity Framework with views

Views are typically created to access the data in a safe manner. The views don't allow you to change the data. They can only be used to access the data but you are not able to modify it. In every application, there is an area that only needs to read the data from one or multiple tables, but modifying this data should not be allowed. Views are particularly useful in such scenarios. Another very important reason is to select and prepare the data that is already in the database to retrieve a view that is directly usable within your applications without the need to transform the data outside the database.

Let's see how we can use views with Entity Framework. Let's create a simple database schema with tables `Employee` and `Employers`. These tables have a one-to-many relationship between them, which is achieved by entering `EmployerId` in the `Employee` table:



A database schema showing Employee and Employer relationship

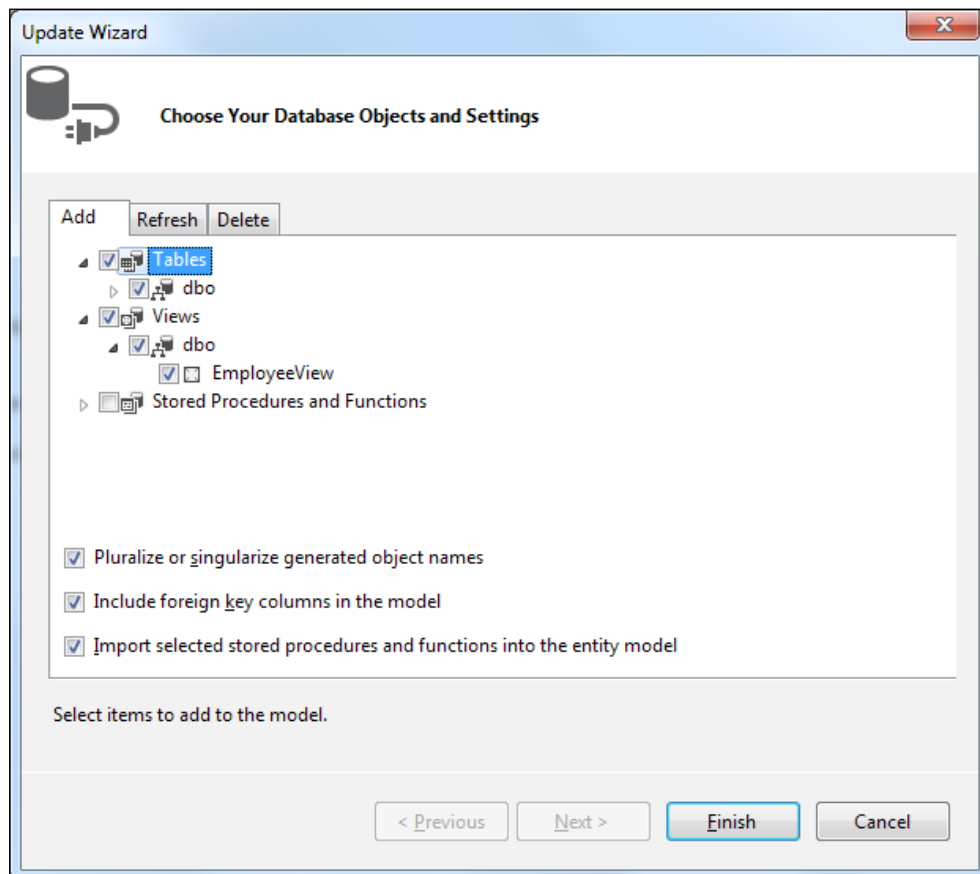
Now, let's create a view that will display all the `Employee` information along with `EmployerName` as follows (1003EN_05_1_code.txt):

```
CREATE VIEW EmployeeView
AS

SELECT
    dbo.Employee.FirstName,
    dbo.Employee.ID,
    dbo.Employee.LastName,
    dbo.Employee.PhoneNumber,
```

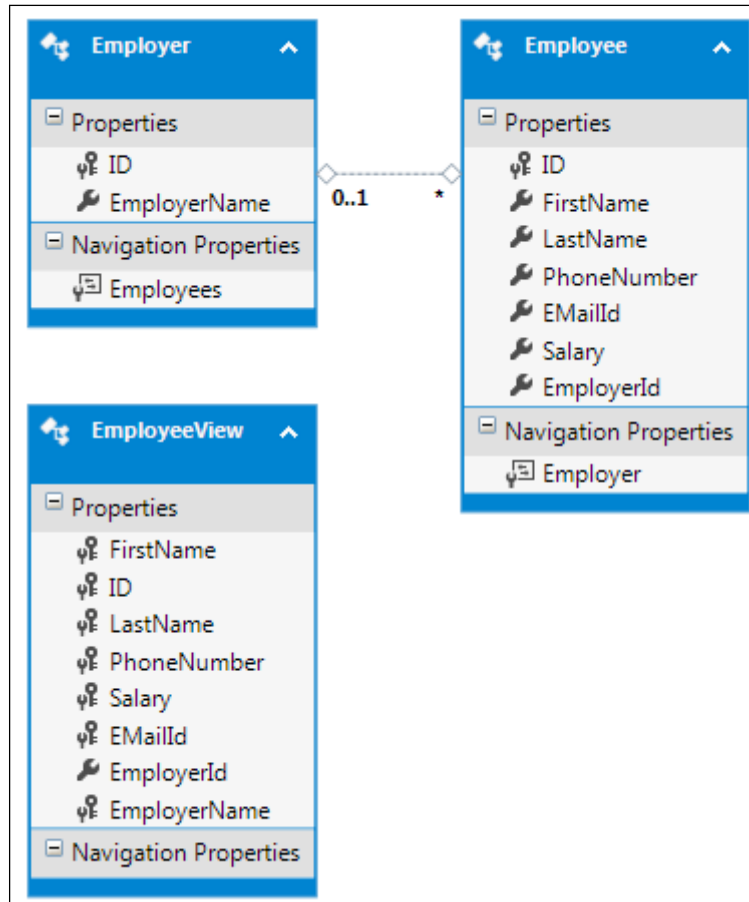
```
    dbo.Employee.Salary,  
    dbo.Employee.EMailId,  
    dbo.Employee.EmployerId,  
    dbo.Employers.EmployerName  
FROM  
    dbo.Employee  
LEFT OUTER JOIN  
    dbo.Employers  
ON  
    dbo.Employee.EmployerId = dbo.Employers.ID  
  
GO
```

Now, let's create the EDM for this database schema. The important thing to remember while doing this is that we need to include the database views too when we are creating our EDM:



Importing views into the EDM

Once the EDM is created, we can see that the data model contains entities corresponding to the tables as well as one entity for our view:



The EDM showing the database view as an entity

If we take a look at the generated model for the view entity, then we can see that it contains the properties for all the columns that the view is returning:

```
public partial class EmployeeView
{
    public string FirstName { get; set; }
    public int ID { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public decimal Salary { get; set; }
    public string EMailId { get; set; }
```

```

public Nullable<int> EmployerId { get; set; }
public string EmployerName { get; set; }
}

```

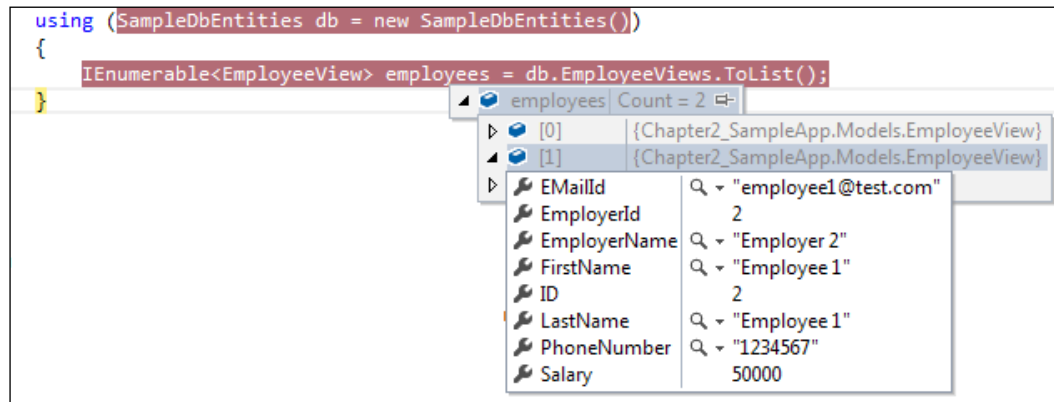
Now, if we want to fetch the data using this view, all we need to do is to use the `DbContext` object to retrieve this conceptual model:

```

using (SampleDbEntities db = new SampleDbEntities())
{
    IEnumerable<EmployeeView> employees = db.EmployeeViews.ToList();
}

```

What happens when we run the preceding code is that Entity Framework will execute the view in the database, and then map the result values with the properties of the `EmployeeView` entity, and return the collection of the `EmployeeView` entity. Let's try to visualize the data that is being retrieved by executing the preceding code:



Fetching the Employee data using a database view

In the preceding screenshot, we can see that `EmployeeView` contains the information from `Employee` as well as the `Employer` table, which is all being populated by executing the view on the database.



Views are particularly useful when we have to fetch data from multiple tables and show it to the user. If we have scenarios where we have conditional queries where the application will pass some parameters, and the data should be fetched based on these parameters, then it is better to use stored procedures rather than using views.

Using Entity Framework with stored procedures

Stored procedures are a way to define the data access functionality in the database itself. The major benefit of using stored procedures is that if we have multiple applications using the same database, then using stored procedures ensures that the data access functionality is consistent across all the applications since it is defined in the data layer itself. Also, stored procedures allow better performance and better encapsulation. Stored procedures also shield us from the SQL injection security concerns, so they allow better security.



Stored procedures are far more powerful than views. Views should only be used to read data from the database. Views are more like virtual tables from the database perspective. Stored procedures, in contrast, can be used to create, read, update, and delete data.

Entity Framework lets us use stored procedures to perform all the CRUD operations. Let's take a look at an example to understand how we can use Entity Framework to use stored procedures to operate on the data.

Let's work on the Employee table that we created in the previous section. First, let's define all the stored procedures to perform all the CRUD operations on this Employee table.

Defining stored procedures

The Create operation can simply be performed by a stored procedure that accepts all the values as parameters to the stored procedure and then inserts the data into the table:

```
Create PROCEDURE dbo.AddEmployee
    @firstname nvarchar(200),
    @lastname nvarchar(200),
    @phonenum nvarchar(20),
    @email nvarchar(50),
    @salary decimal(18, 0),
    @employerId int
AS
    insert into Employee(FirstName, LastName, PhoneNumber, EmailId,
    Salary, EmployerId)
    values (@firstname, @lastname, @phonenum, @email, @salary, @
    employerId)
RETURN
```

To retrieve the list of all Employees, let's create one stored procedure:

```
Create PROCEDURE dbo.GetEmployees
AS
    Select * from Employee
RETURN
```

To retrieve a specific Employee record, we will have to pass the ID of that Employee to the stored procedure, and the procedure will return the employee details associated with the Employee with that ID:

```
Create PROCEDURE dbo.GetEmployeeById
    @id int
AS
    Select * from Employee
    where ID = @id
RETURN
```

To update the record, we will have to pass all the updated values to the stored procedure along with the ID of the employee. The procedure will use the ID, and update the record with this ID with the new values:

```
Create PROCEDURE dbo.UpdateEmployee
    @id int,
    @firstname nvarchar(200),
    @lastname nvarchar(200),
    @phonenummer nvarchar(20),
    @email nvarchar(50),
    @salary decimal(18, 0),
    @employerId int
AS
    update Employee
    set
        FirstName = @firstname,
        LastName = @lastname,
        PhoneNumber = @phonenummer,
        EMailId = @email,
        Salary = @salary,
        EmployerId = @employerId
    where
        ID = @id
RETURN
```

Finally, to delete an employee, we just need to pass the ID of the record to be deleted and the procedure will delete the record:

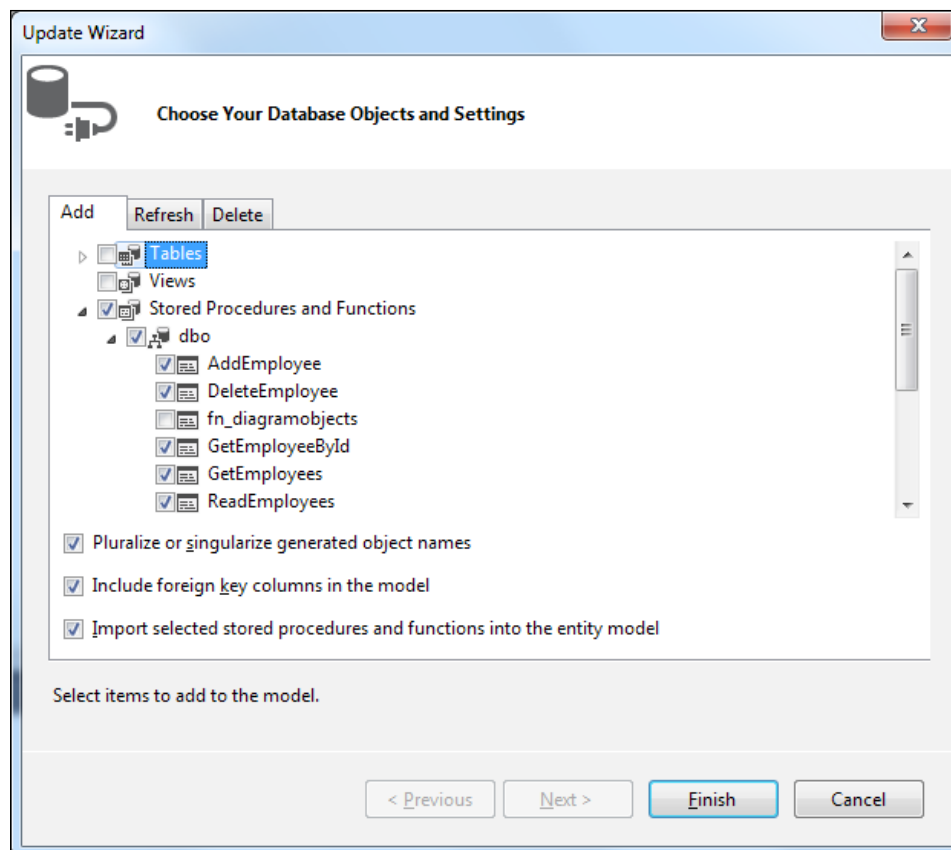
```
Create PROCEDURE dbo.DeleteEmployee
    @id int
AS
    delete from Employee where ID=id
RETURN
```



A detailed explanation of these stored procedures is outside the scope of this book.

Now we have some basic stored procedures defined and ready to be used.

Using stored procedures, let's now update our Entity Data Model to include the stored procedures too. We will then see how we can use these stored procedures:



Adding stored procedures into our EDM

Once we update the EDM, the stored procedures are also available in the entity data model. There are two ways of using the stored procedures from our application. First, every entity can be mapped to use a stored procedure to perform the insert, update, and delete operations.

In this mapping, we have to specify the procedures that should be used to perform the respective operations and the mapping of the entity properties with the stored procedure's parameters:

Mapping Details - Employee			
	Parameter / Column	Operator	Property
Functions			
Insert Using AddEmployee			
Parameters			
@ firstname : nvarchar	←	✎	FirstName : String
@ lastname : nvarchar	←	✎	FirstName : String
@ phonenumber : nvarchar	←	✎	PhoneNumber : String
@ email : nvarchar	←	✎	EMailId : String
@ salary : decimal	←	✎	Salary : Decimal
@ employerId : int	←	✎	EmployerId : Int32
Result Column Bindings			
<Add Result Binding>			
Update Using UpdateEmployee			
Parameters			
@ id : int	←	✎	ID : Int32
@ firstname : nvarchar	←	✎	FirstName : String
@ lastname : nvarchar	←	✎	LastName : String
@ phonenumber : nvarchar	←	✎	PhoneNumber : String
@ email : nvarchar	←	✎	EMailId : String
@ salary : decimal	←	✎	Salary : Decimal
@ employerId : int	←	✎	EmployerId : Int32
Result Column Bindings			
<Add Result Binding>			
Delete Using DeleteEmployee			
Parameters			
@ id : int	←	✎	ID : Int32

Once this mapping is specified, we can use the entities with the `DbContext` class, and the insert, update, and delete operations will work using the stored procedures:

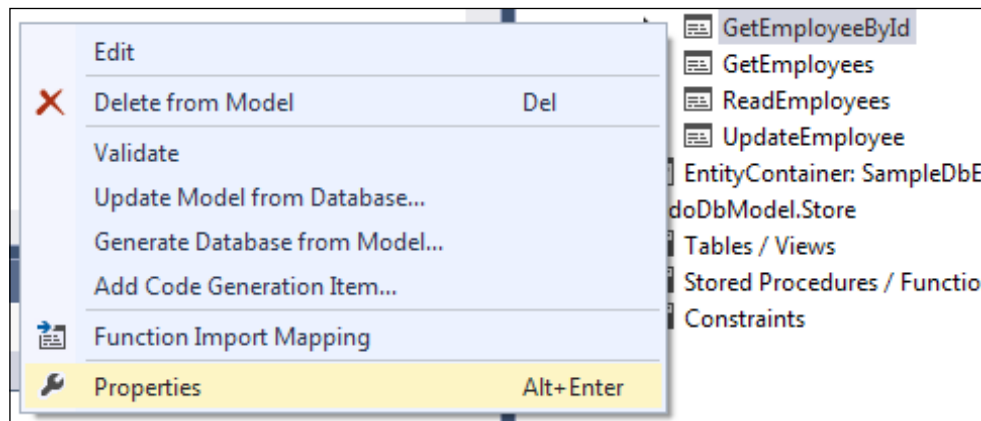
```
using (SampleDbContext db = new SampleDbContext())
{
    Employee employee = new Employee();
    employee.FirstName = "test Employee";
    employee.LastName = "test lastname";
    employee.PhoneNumber = "1234567890";
    employee.EmailId = "test@test.com";
    employee.Salary = 50000;

    db.Employees.Add(employee);
    db.SaveChanges();
}
```

In the preceding code, the new `Employee` entity will be created. When this `Employee` object is added using `DbContext`, Entity Framework will use the mappings to use the stored procedure that should be used to insert the data and the mappings of the object properties to the parameters specified in the procedure. Saving the changes will execute that stored procedure with the given parameter values.

Another possibility is that we have a stored procedure that is not responsible for inserting, updating, or deleting an entity, but it provides some custom logic. For this, we can execute the stored procedure directly using the `DbContext` object. For instance, if we need to execute the `GetEmployeeById` procedure, we can do this in the following manner.

First, we need to open the model browser for our EDMX and then right-click on and select properties on the procedure that we want to use:



Checking the properties of imported stored procedures

In the properties window, we will select the return type of the stored procedure. Our current procedure returns an `Employee` entity, so let's select the return type as an `Employee` entity:

Add Function Import

Function Import Name:
GetEmployeesById

☐ Function Import is composable

Stored Procedure / Function Name:
GetEmployeeById

Returns a Collection Of

☐ None

☐ Scalars:

☐ Complex:

☒ Entities:

Stored Procedure / Function Column Information


Click on "Get Column Information" above to retrieve the stored procedure's or function's schema. Once the schema is available, click on "Create New Complex Type" below to create a compatible complex type. You can also always update an existing complex type to match the returned schema. The changes will be applied to the model once you ...

Adding a Function Import for the stored procedure

Once this is done, the stored procedure is available as a function on our `DbContext` object that returns `ObjectResult<Employee>`. To execute this stored procedure, we just need to invoke the respective function with the needed parameters as follows:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    int id = 1;
    ObjectResult<Employee> employeeResult = db.GetEmployeeById(id);
}
```

The preceding code will invoke the `GetEmployeeById` procedure, pass the `id` value, and retrieve the result.

 If our stored procedure returns data, which is not an entity in our conceptual data model, then we need to create a complex type to hold the return value from the procedure.

Using Entity Framework with functions

Functions provide an excellent way of writing reusable logic in the data layer itself. Functions are computed values and cannot perform permanent environmental changes to the SQL Server. There are two types of functions that can be defined in the database:

- Scalar functions
- Table values functions

Let's see how we can use these functions using Entity Framework.

Using scalar functions

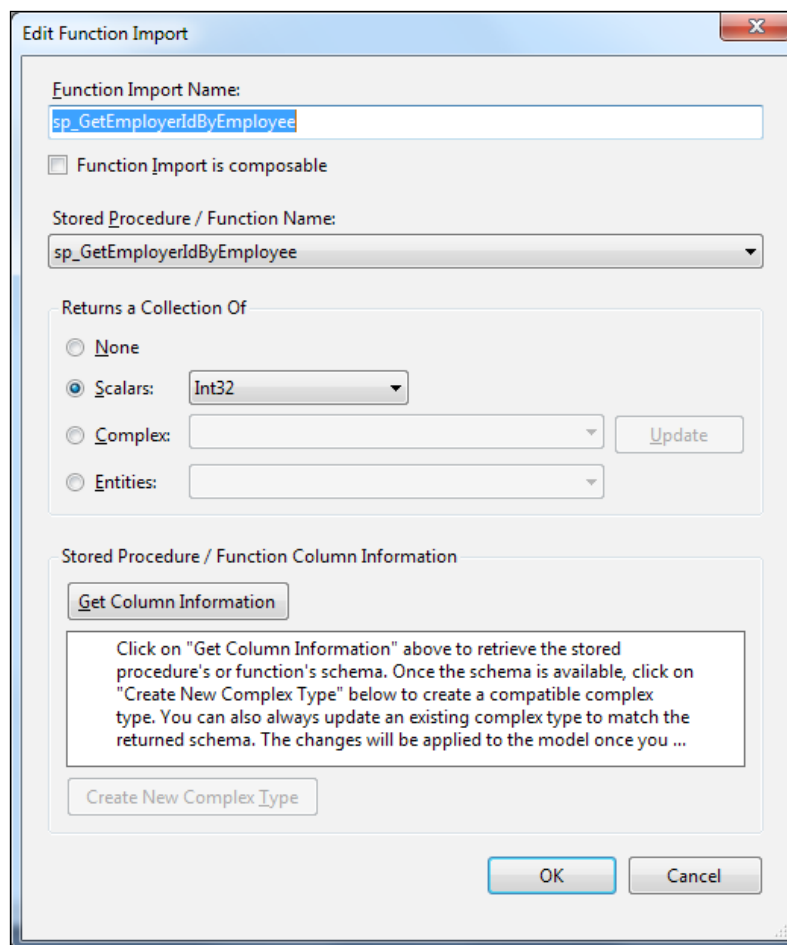
A scalar function is a function that is defined in the database and returns a scalar value. For example, if we need to find the `EmployerId` for an `Employee` by passing the ID of the `Employee`, we can do this by defining a scalar function as follows:

```
Create FUNCTION dbo.GetEmployerIdForEmployee (@id int) RETURNS int
AS
BEGIN
    DECLARE @result int;
    select @result = EmployerId from Employee where ID = @id;
    RETURN @result;
END
```

Unfortunately, Entity Framework does not support using a scalar function directly. If we want to use a scalar function in our application, we have to wrap it inside a stored procedure as follows:

```
Create PROCEDURE dbo.sp_GetEmployerIdByEmployee
    @id int
AS
    select dbo.GetEmployerIdForEmployee(@id);
RETURN
```

Now this stored procedure is added to the Entity Data model in the same way as shown in the previous section. The only thing that we need to keep in mind is that this stored procedure's return type should be mapped to a scalar type from the **Function Import** properties.



Adding the Function Import to the stored procedure by wrapping the scalar function

Now this stored procedure is available as a function with our `DbContext` object as follows:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    int id = 1;
    ObjectResult<int?> employerIdResult = db.sp_
GetEmployerIdByEmployee(id);
}
```

The preceding code will execute our stored procedure that wraps our scalar function and thus, effectively, our scalar function will be called by executing the preceding code.

Using table valued functions

Table valued functions (TVF) are simply the user-defined functions that return tabular data. A complete explanation of table valued functions is outside the scope of this book, but let's try to see how a table valued function can be defined, and how it can be used from the application.

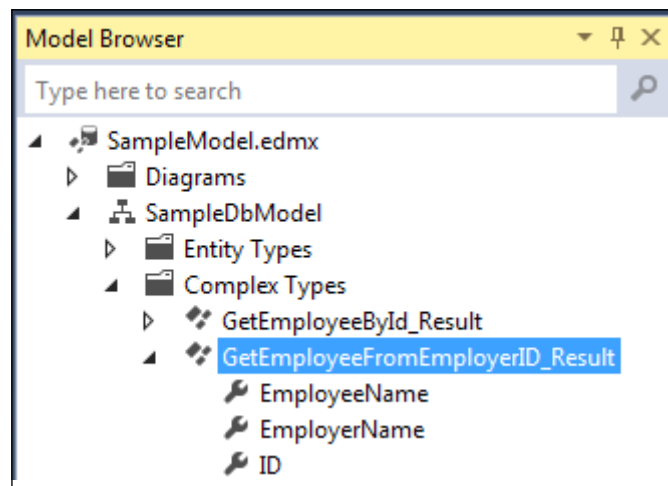
Let's define a simple table valued function that will return the `Employee` details with the `EmployerName` details for a given `EmployerID`:

```
Create FUNCTION dbo.GetEmployeeFromEmployerID(@id int)
RETURNS @table_variable TABLE (ID int, EmployeeName nvarchar(200),
EmployerName nvarchar(200))
AS
BEGIN
    INSERT INTO @table_variable
    SELECT
        employee.ID,
        employee.FirstName,      Employer.EmployerName
    FROM
        Employee employee,
        Employer employer
    where
        employee.EmployerID = employer.ID
        and employer.ID = @id;
RETURN;
END
```

What happens in this function is that the function creates a `TABLE` variable, and then, using the details from the `Employer` and `Employee` tables for the given `employerId`, returns the table for the result found. Let's see how we can use this table valued function from the application using Entity Framework.

From Entity Framework's perspective, using a table valued function is exactly the same as using a stored procedure. We first need to add the stored procedure on the conceptual Entity Data Model. Then, from the **Function Import** properties, we can specify the return type.

In our current example, we return a table that cannot be mapped to any existing entity. When we use this TVF, Entity Framework will add a complex type to store the result retrieved from the function:



Creating/modifying the complex type from the model browser

If we take a look at the **Function Import** properties, we can see that the result is already mapped to this auto-created complex type:

Edit Function Import

Function Import Name:
GetEmployeeFromEmployerID

☒ Function Import is composable

Stored Procedure / Function Name:
GetEmployeeFromEmployerID

Returns a Collection Of

☐ None

☐ Scalars:

☒ Complex: GetEmployeeFromEmployerID_Result

☐ Entities:

Stored Procedure / Function Column Information

Click on "Get Column Information" above to retrieve the stored procedure's or function's schema. Once the schema is available, click on "Create New Complex Type" below to create a compatible complex type. You can also always update an existing complex type to match the returned schema. The changes will be applied to the model once you ...

Creating/modifying Function Import for a table valued function returning a complex type

Using a table valued function from the code is also very similar to the way we use stored procedures. This table valued function will also be available as a function on the `DbContext` object and can be used as follows:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    int id = 1;
    IQueryable<GetEmployeeFromEmployerID_Result> employerIdResult = db
        .GetEmployeeFromEmployerID(id);
}
```

The preceding code will invoke the table valued function and get back the result in the complex type that Entity Framework created.

Summary

In this chapter, we saw how we can use Entity Framework with views, stored procedures, and functions defined in the database. At this juncture, we have all of the information needed to work with Entity Framework using the Database First approach. In the next chapter, we will start looking at the other end of the spectrum where we have the application models created, and we want to use Entity Framework to persist the model data in the database.

6

Entity Framework Code First – Domain Modeling and Managing Entity Relationships

So far, we have seen how we can use Entity Framework with an existing database using the Entity Framework Code First approach. The other end of the spectrum is to see how we can design our application domain models in such a way that we can use Entity Framework to persist the domain entity's data in the database. This can be done using the Entity Framework Code First approach.



The same can also be achieved using the Entity Framework Model First approach. The only difference in the Code First approach is that we will be using Visual Entity Designer instead of writing our entity **Plain Old CLR Objects (POCOs)** classes ourselves. Conceptually, both these approaches will be similar, and thus in this chapter, we will discuss the Code First approach.

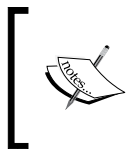
Understanding the Entity Framework Code First approach

Before we start discussing the Code First approach, it would be a good idea to talk about the application architecture approach being followed. Traditionally, we have designed an application using a bottom-up approach, that is, we used to think about the database first and then used this data-centric approach to build our application on top of it. This approach is still very useful for the applications that are data intensive, or perhaps the database also contains some business logic that is being used by multiple applications. For such applications, if we want to use Entity Framework, we have to use the Database First approach.

Another way (or rather the modern way) of designing applications is using the domain-centric approach (Domain Driven Design). Domain Driven Design is more of a top-down approach, where we start designing our application by thinking in terms of the domain models and entities that are needed to realize our application. The database is merely used for persistence of this model data. Using Domain Driven Design would mean that we design our models and entities as per our application needs, and they are persistence ignorant, that is, they can be saved using any database technology. In such scenarios, we should use the Entity Framework Code First approach, as this lets us create POCOs for our domain models that are persistence ignorant. Entity Framework will take care of storing the entities data in any database.

Here are a few benefits of using the Entity Framework Code First approach:

- It supports Domain Driven Design
- We can start developing our application early, as we don't have to wait for the database to be created
- The persistence layer, that is, the underlying database can be changed without having any impact on the models



Discussing the details of DDD is not in the scope of this book. To know more about DDD, please refer to this article from MSDN magazine available at <http://msdn.microsoft.com/en-us/magazine/dd419654.aspx>.

Understanding the Code First conventions and configurations

The first thing that we need to understand is the concept of convention over configuration. The Code First approach expects the model classes to follow some conventions, so that the database persistence logic can be extracted from the model. For example, if we define a property named `ID` in the model class, this property value will be treated as the primary key for the table that is being mapped to this class. The benefit of this convention-based approach is that if we are following the conventions, then we don't have to write any extra code to manage the database persistence logic. The downside of this approach is that if a convention is not followed, and Entity Framework is not able to extract the needed information from the model, an exception will be thrown at runtime.

In scenarios where we are not able to follow the conventions but still need Entity Framework to persist the data, we need to provide some additional information with our model using the Code First configuration options, so that Entity Framework is able to extract all the needed information from our model. For example, if we cannot have a property named `ID` in our model as the primary key, we need to update the desired property with the `[Key]` attribute and that will be taken as the primary key.



Entity Framework uses these conventions to create table names by pluralizing the class' name, and it creates the columns with the same names as the class' properties.

Implementing Entity Framework Code First

Let's try to understand how we can use the Entity Framework Code First approach by working on a small application. Let's try to create a POCO for an employee, and see how we can use Entity Framework to perform CRUD operations on this model:

1. Let's start by creating the Employee model (1003OS_06_1_code.txt):

```
public class Employee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public string EMailId { get; set; }
}
```


If we use this model, the generated table name will be `Employees`, the column names will be the name of the properties, and the `ID` field will be treated as the primary key. All this will be because of the Code First conventions.

2. If we want to override these default convention-based values and provide our values, then we can do this by updating the class and properties with the needed attributes:

```
[Table("Employees")]
public class Employee
{
    [Key]
    public int ID { get; set; }

    [Column("FirstName")]
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public string EMailId { get; set; }
}
```

3. In the preceding code, we used Entity Framework configuration to define the table name, column name for the `FirstName` property, and the primary key column specification.
4. Now that our domain model is ready, let's see how we can create the `DbContext` class to perform CRUD operations on this model. We need to define our `DbContext` class by inheriting from the `DbContext` class. This context class should contain `DbSet<Employee>`, so that it is able to perform CRUD operations on the `Employee` model:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Employee> Employees { get; set; }
}
```

What this context class will do is that when we use it to perform the operations on our model, it will first look for `connectionString` with the same name as the `DbContext` class. Using this `connectionString`, it will perform the operations on the `Employees` table. The `Employees` table-related information will be extracted by looking at the `Employee` class and using the Entity Framework conventions and configuration options.

We can use a custom `connectionString` with the context class:

```
<connectionStrings>
<add name="SampleDbEntities" connectionString="Data
Source=(LocalDb)\v11.0;Initial Catalog=codeFirstSample;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\codeFirstSample.
mdf" providerName="System.Data.SqlClient" />
</connectionStrings>
```

5. However, we will have to pass the `connectionString` to the `DbContext` class in the class' constructor:

```
public class SampleDbEntities : DbContext
{
    public SampleDbEntities()
        : base("name=SampleDbEntities")
    {
    }

    public virtual DbSet<Employee> Employees { get; set; }
}
```

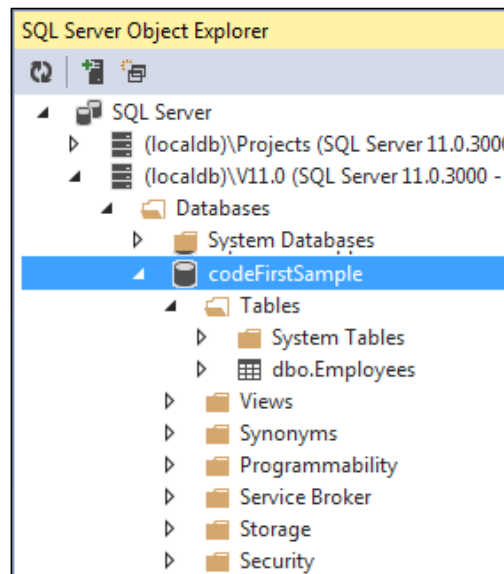
6. Now we can use this context class to perform CRUD operations on our `Employee` model. Let's take a look at a simple create operation to understand this better:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Employee employee = new Employee();
    employee.FirstName = "test Employee";
    employee.LastName = "test lastname";
    employee.PhoneNumber = "1234567890";
    employee.EmailId = "test@test.com";

    db.Employees.Add(employee);
    db.SaveChanges();
}
```

What the preceding code will do is use our `Employee` class to extract the schema information, and use our context class to connect to the database specified in the `connectionString` that is linked to this `DbContext` class. Using `db.Employees.Add(employee)`, will associate our `Employee` model object with the context class and `db.SaveChanges()` will persist the data in the database.

If we look for the database at the given connection string, we can see that Entity Framework has created a database for us:



A created database using a simple Entity Framework Code First approach

Also, if we look at the table data, we can see that the data has been saved in the Employees table:

ID	FirstName	LastName	PhoneNumber	EMailId
1	test Employee	test lastname	1234567890	test@test.com

The data inserted into the table using a simple Entity Framework Code First approach

So, when we are using the Code First approach, Entity Framework will take care of creating the database for us on the first run. Then onwards, it will use the same database to perform further database operations.

More on domain class configurations

Earlier we discussed that if we want to override the default conventions of Entity Framework, we have to provide our own custom configurations. There are two ways in which we can provide these configurations:

- Data annotations
- Fluent API

The data annotations approach is the one we used to specify the table our class should map to, the key property and the column that the `FirstName` property should map to. Let's take a look at a few more attributes that can be specified to override the default conventions and use the configurations to define a database schema:

- **Table:** This specifies the name of the table this class should map to
- **Column:** This specifies the name of the column that a class property should map to
- **Key:** This specifies the property that should be treated as a primary key
- **Timestamp:** This marks a property as a timestamp column in the database
- **ForeignKey:** This specifies the foreign key property for a navigation property
- **NotMapped:** This specifies that the property should not be mapped to any column
- **DatabaseGenerated:** This specifies that the property should be mapped to the computed column of the database table



The `DatabaseGenerated` attribute can also be used to map to the auto-increment identity column.

Another way of providing the configurations is to use the fluent API. The `DbContext` class has the `OnModelCreating` method, which can be used to configure the class to the database schema mapping fluently. So, if we have to define the same mappings in a fluent manner, our `DbContext` class will look like the following:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        //Configure domain classes using Fluent API here
        modelBuilder.Entity<Employee>().ToTable("Employees");
        modelBuilder.Entity<Employee>().HasKey(e => e.ID);
        modelBuilder.Entity<Employee>().Property(e => e.FirstName).
HasColumnName("FirstName");

        base.OnModelCreating(modelBuilder);
    }
}
```

What the preceding code is doing is specifying that the `Employee` class should be mapped to the `Employees` table in the database, the `ID` property should be treated as a primary key, and the `FirstName` property should be mapped to a column named `FirstName`.



One important deciding factor for using the fluent API is that if we are using external POCO classes, that is, the classes being used are from a class library, and we cannot change the definitions, as it is not possible to use data annotations to provide the mapping details. In such cases, we must use the fluent API, as it is better suited for such scenarios.

Managing Entity relationships using the Code First approach

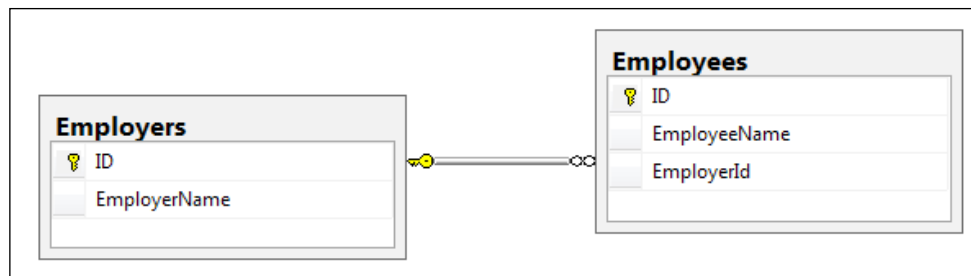
Now that we know how to define simple domain classes using the Code First approach, and we can use the `DbContext` class to perform the database operations using the domain classes, let's see how we can implement database multiplicity relationships using these domain classes. We will see how to implement the following types of relationships, using Entity Framework Code First:

- One-to-many relationship
- One-to-one relationship
- Many-to-many relationship

We will also take a look at the various ways in which these relationships can be implemented, such as using Entity Framework conventions, data annotations, and fluent API for these configurations.

Implementing one-to-many relationships

To configure a one-to-many relationship in the database, we can either rely on Entity Framework conventions, or we can use the data annotations/fluent API-based configurations to explicitly create the relationship. Let's say we have to create a one-to-many relationship between the `Employers` and `Employees` table in the database as follows:



Desired database schema to implement a one-to-many relationship

For this, let's start by creating an `Employee` class that will be mapped to the `Employees` table:

```
[Table("Employees")]
public class Employee
{
    [Key]
    public int ID { get; set; }

    [Column("EmployeeName")]
    public string EmployeeName { get; set; }
}
```

Similarly, we need to have a class that will be mapped to the `Employers` table:

```
[Table("Employers")]
public class Employer
{
    [Key]
    public int ID { get; set; }

    [Column("EmployerName")]
    public string EmployerName { get; set; }
}
```

Now, if we want to create a one-to-many relationship between these two tables, we simply need the navigation properties created in the classes.

So, Employee should contain a navigation property of Employer since Employee is on the many side of the relationship, and every Employee record can have only one Employer associated with it:

```
[Table("Employees")]
public class Employee
{
    [Key]
    public int ID { get; set; }

    [Column("EmployeeName")]
    public string EmployeeName { get; set; }

    public virtual Employer Employer { get; set; }
}
```

Similarly, the Employer class should also contain the navigation property. Since the Employer class is on the one side of the relationship, the navigation property will be a collection of Employees. So every Employer class can be associated with many Employees:

```
[Table("Employers")]
public class Employer
{
    [Key]
    public int ID { get; set; }

    [Column("EmployerName")]
    public string EmployerName { get; set; }

    public virtual ICollection<Employee> Employees { get; set; }
}
```

Now, with this code, Entity Framework will be able to extract the required information and will create a one-to-many relationship between the Employer and Employee tables in the database too.

If we are using the fluent API to configure the mapping, then we need to specify this relationship's details in the fluent mappings too. The following code snippet shows how we can specify the one-to-many relationship details using the fluent API:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Employee> Employees { get; set; }
    public virtual DbSet<Employer> Employers { get; set; }
}
```

```

        protected override void OnModelCreating(DbModelBuilder
modelBuilder)
        {
            modelBuilder.Entity<Employee>().HasRequired<Employer>(employee
=> employee.Employer)
                .WithMany(employer => employer.Employees).HasForeignKey(employer
=> employer.ID);

            base.OnModelCreating(modelBuilder);
        }
    }

```

In the preceding code, we specified that the `Employer` property is required for the `Employee` object, and it has a one-to-many relationship with the `Employers` table, where the `ID` column in the `Employers` table should be used as the foreign key in the `Employees` table.



The fluent API is only needed if the entity classes are not following Entity Framework conventions, and the mapping information cannot be extracted from the entity classes by Entity Framework.



Now let's try to create an `Employee` and `Employer` record, and try to save them in the database:

```

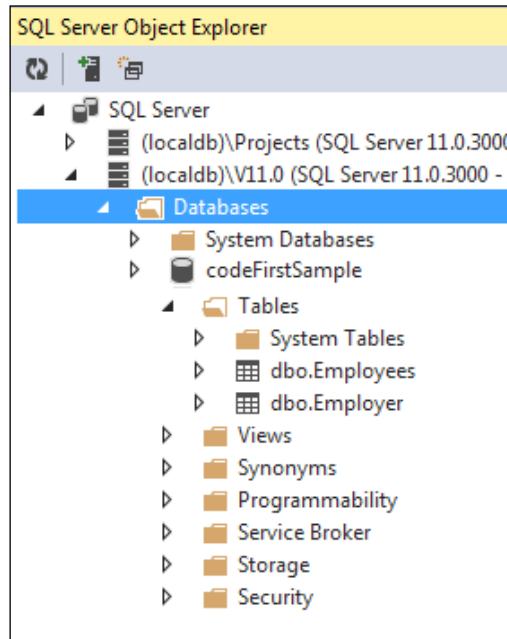
using (SampleDbEntities db = new SampleDbEntities())
{
    Employer employer = new Employer();
    employer.EmployerName = "Test Employer";

    Employee employee = new Employee();
    employee.EmployeeName = "test Employee";

    db.Employees.Add(employee);
    db.SaveChanges();
}

```


In the preceding code, we created an `Employer` object and then created an `Employee` object, and associated `Employer` with the `Employee` object. Let's take a look at the created tables in the database for these classes:



The created database using the Code First approach for a one-to-many relationship

Let's also take a look at the data in these tables:

dbo.Employer [Data]		dbo.Employees [Data]	
Max Rows: 1000		Max Rows: 1000	
ID	EmployerName	ID	EmployeeName
1	Test Employer		

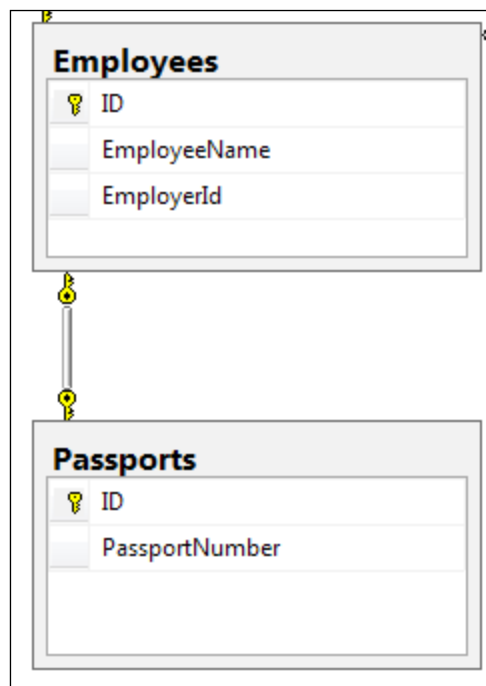
dbo.Employer [Data]		dbo.Employees [Data]	
Max Rows: 1000		Max Rows: 1000	
ID	EmployeeName	Employer_ID	
2	test Employee	1	

The data inserted into the table using the Code First approach for a one-to-many relationship

In the preceding screenshot, we can see that the Entity Framework has created a one-to-many relationship between `Employer` and `Employee` and has put the data into these tables accordingly.

Implementing one-to-one relationships

As with one-to-many relationships, a one-to-one relationship in the database can be implemented using either Entity Framework conventions, by explicitly using data annotations based configurations, or via the fluent API-based configurations. Let's say we have to create a one-to-one relationship between the `Employers` and `Passports` tables in the database as follows:



A desired database schema to implement a one-to-one relationship

Let's create the `Employee` class that will be mapped to the `Employees` table:

```
[Table("Employees")]
public class Employee
{
    [Key]
    public int ID { get; set; }
```

```
        [Column("EmployeeName")]
        public string EmployeeName { get; set; }
    }
```

Now, we also need to create a class that will be mapped to the `Passports` table:

```
[Table("Passports")]
public class Passport
{
    [Key]
    public int ID { get; set; }

    [Column("PassportNumber")]
    public string PassportNumber { get; set; }
}
```

Our objective here is to create a one-to-one relationship between the `Employee` and `Passport` classes. The Entity Framework will create the same relationship between the `Employees` and `Passports` tables. To do this, we need to create a navigation property for `Passport` in the `Employee` class:

```
[Table("Employees")]
public class Employee
{
    [Key]
    public int ID { get; set; }

    [Column("EmployeeName")]
    public string EmployeeName { get; set; }

    public virtual Passport Passport { get; set; }
}
```

Similarly, the `Passport` class should also contain the navigation property to `Employee`. Since for every `Passport` record, there should be only one `Employee` record, and this navigation property should also be just a single object (unlike the collection in the previous example). One important thing to keep in mind is that the `ID` property for this class should not be database generated, since in a one-to-one relationship, both the participating tables should have the same primary key. To accomplish this, we also need to mark the `ID` property of the `Passports` table as a foreign key for the `Employees` table:

```
[Table("Passports")]
public class Passport
{
    [Key, ForeignKey("Employee")]
```

```

    public int ID { get; set; }

    [Column("PassportNumber")]
    public string PassportNumber { get; set; }

    public virtual Employee Employee { get; set; }
}

```

Now, with this code, the Entity Framework will be able to extract the required information and will create a one-to-one relationship between the `Employees` and `Passports` tables in the database.

If we are using the fluent API to configure the mapping, then we need to specify the relationship details in the fluent mappings too. The following code snippet shows how we can specify the one-to-one relationship details using the fluent API:

```

public class SampleDbEntities : DbContext
{
    public virtual DbSet<Employee> Employees { get; set; }
    public virtual DbSet<Passport> Passports { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        // Line 1
        modelBuilder.Entity<Passport>().HasKey(e => e.ID);

        // Line 2
        modelBuilder.Entity<Passport>()
            .Property(e => e.ID)
            .HasDatabaseGeneratedOption(DatabaseGeneratedOpti
on.None);

        // Line 3
        modelBuilder.Entity<Passport>()
            .HasRequired(e => e.Employee)
            .WithRequiredDependent(s => s.Passport);

        base.OnModelCreating(modelBuilder);
    }
}

```

Now let's see what is being done in the preceding code. Line 1 lets the Entity Framework know that the `ID` property of the `Passport` class should be used as a key. Line 2 specifies that the key corresponding to the `ID` property should not be database generated, and Line 3 specifies that there is a one-to-one relationship between the `Employees` and `Passports` tables.

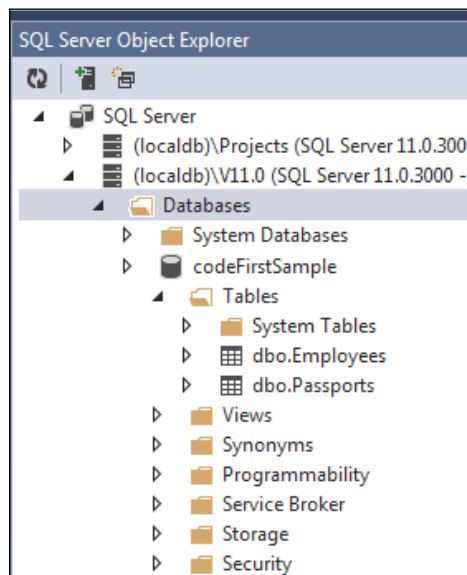
Now let's try to create an `Employee` and a `Passport` record, and try to save them in the database:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Passport passport = new Passport();
    passport.PassportNumber = "Test123";

    Employee employee = new Employee();
    employee.EmployeeName = "test Employee";
    employee.Passport = passport;

    db.Employees.Add(employee);
    db.SaveChanges();
}
```

In the preceding code, we created a `Passport` object and then created an `Employee` object, and associated the `Passport` with the `Employee`. Let's take a look at the created tables in the database for these classes:



The created database using the Code First approach for a one-to-one relationship

Now let's also take a look at the data in these tables:

dbo.Passports [Data]		dbo.Employees	
	ID		EmployeeName
	1		test Employee

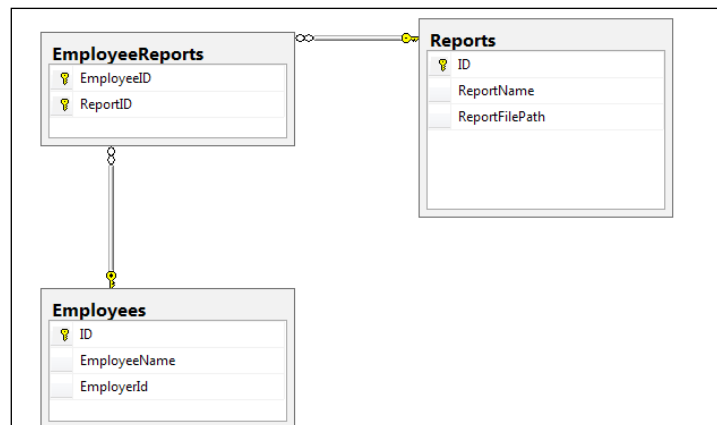
dbo.Passports [Data]		dbo.Employees	
	ID		PassportNum...
	1		Test123

The data inserted into the table using the Code First approach for a one-to-one relationship

In the preceding screenshot, we can see that Entity Framework has created a one-to-one relationship between `Passport` and `Employee` and has put the data into these tables accordingly.

Implementing many-to-many relationships

To model a many-to-many relationship in the database, we can use Entity Framework conventions, data annotations based configurations, or the fluent API-based configurations. Let's say we have to create a many-to-many relationship between the `Employers` and `Reports` tables in the database. For this, we need a joining table to implement the many-to-many relationship. The desired database schema looks like the following:



The desired database schema to implement a many-to-many relationship

Let's create the Employee class that will be mapped to the Employees table:

```
[Table("Employees")]
public class Employee
{
    [Key]
    public int ID { get; set; }

    [Column("EmployeeName")]
    public string EmployeeName { get; set; }
}
```

We also need to create a class that will be mapped to the Reports table:

```
[Table("Reports")]
public class Report
{
    [Key]
    public int ID { get; set; }

    [Column("ReportName")]
    public string ReportName { get; set; }

    public virtual Employee ReportFilePath { get; set; }
}
```

To create a many-to-many relationship between the Employee and Report class, we need to create the navigation property as a collection of Report in the Employee class. This will associate every Employee class with multiple Reports:

```
[Table("Employees")]
public class Employee
{
    public Employee()
    {
        Reports = new HashSet<Report>();
    }

    [Key]
    public int ID { get; set; }

    [Column("EmployeeName")]
    public string EmployeeName { get; set; }

    public virtual ICollection<Report> Reports { get; set; }
}
```

Similarly, the `Reports` class should also contain the navigation property to a collection of `Employee`. This will let every `Report` be associated with many `Employees`:

```
[Table("Reports")]
public class Report
{
    public Report()
    {
        Employees = new HashSet<Employee>();
    }

    [Key]
    public int ID { get; set; }

    [Column("ReportName")]
    public string ReportName { get; set; }

    public virtual Employee ReportFilePath { get; set; }

    public virtual ICollection<Employee> Employees { get; set; }
}
```

Now, with this code, Entity Framework will be able to extract the required information and will create a many-to-many relationship between the `Employees` and `Reports` tables in the database.

If we are using the fluent API to configure the mapping, then we need to specify the relationship details in the fluent mappings too. The following code snippet shows how we specify the many-to-many relationship details using fluent API:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Employee> Employees { get; set; }
    public virtual DbSet<Report> Reports { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Employee>().HasMany<Report>(employee =>
employee.Reports).WithMany(report => report.Employees).Map(c =>
        {
            c.MapLeftKey("EmployeeID");
            c.MapRightKey("ReportID");
            c.ToTable("EmployeeReports");
        }
    }
}
```



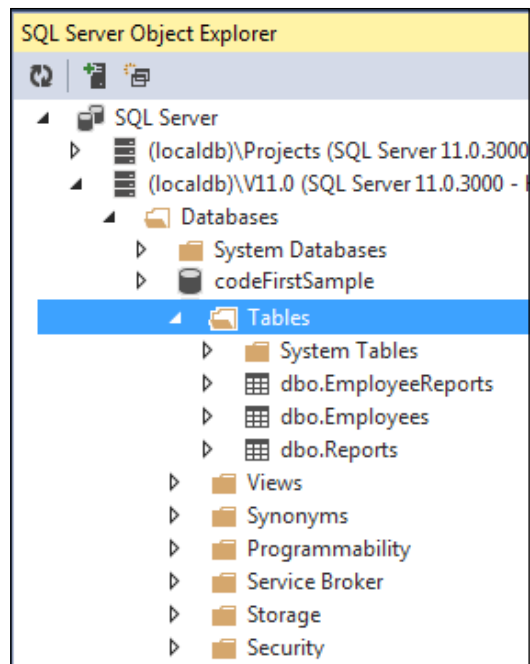
```
        });  
  
        base.OnModelCreating(modelBuilder);  
    }  
}
```

Now let's see what is being done in the preceding code. Line 1 lets Entity Framework know that there is a many-to-many relationship between the `Employee` and `Report` entities. Line 2 shows that the `EmployeeID` property of the `Employee` class should be used as a key. Line 3 specifies that the `ReportID` property should be used as a key for the `Report` entity. The next line specifies that the `EmployeeReport` is the table that will be used as a joining table to implement this many-to-many relationship in the database.

Now let's try to create an `Employee` and a `Report` record, and save them in the database:

```
using (SampleDbEntities db = new SampleDbEntities())  
{  
    Report report1 = new Report();  
    report1.ReportName = "Test Report 1";  
    report1.ReportName = @"C:\Temp\Report1.rpt";  
  
    Report report2 = new Report();  
    report2.ReportName = "Test Report 2";  
    report2.ReportName = @"C:\Temp\Report2.rpt";  
  
    db.Reports.Add(report1);  
    db.Reports.Add(report2);  
  
    Employee employee = new Employee();  
    employee.EmployeeName = "Test Employee";  
    employee.Reports.Add(report2);  
  
    db.Employees.Add(employee);  
    db.SaveChanges();  
}
```

In the preceding code, we created a couple of `Report` objects and then created an `Employee` object and associated the `Report` with the `Employee`. If we take a look at the created tables in the database for these classes, we can see that the respective tables for these classes have been created, and a joining table has also been created to facilitate a many-to-many relationship between these two tables:



A created database using the Code First approach for a many-to-many relationship

Let's also take a look at the data in these tables:

dbo.Reports [Data]	dbo.Reports [Data]	Web.config
Max Rows: 1000		
ID	ReportName	ReportID
1	C:\Temp\Report2.rpt	NULL
2	C:\Temp\Report1.rpt	NULL

dbo.Employees [Data]	dbo.Reports [Data]	Web.config
Max Rows: 1000		
ID	EmployeeName	
1	Test Employee	

dbo.EmployeeReports [Data]	dbo.Employees [Data]
Max Rows: 1000	
EmployeeID	ReportID
1	1

The data inserted into the table using the Code First approach for a many-to-many relationship

From the preceding screenshot, it's quite clear how Entity Framework has created a many-to-many relationship between the `Report` and `Employee` and has put the data into these tables accordingly.

Inheritance with the Entity Framework Code First approach

So far, we have seen how to use the Entity Framework Code First approach to map domain entities to database tables. We also looked at how we can have entities with multiplicity relationships, and how to use Entity Framework to map the relationships in such a way that these relationships exist between the database tables too.

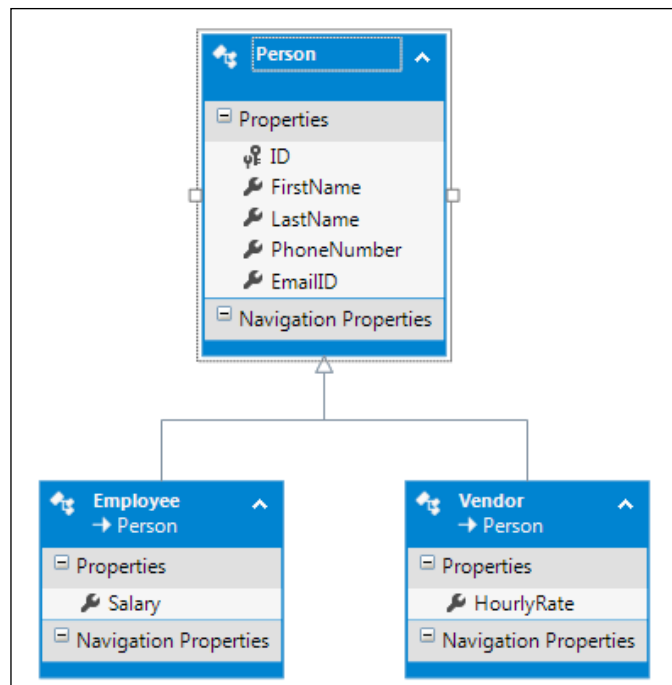
Now let's see how we can have inheritance relationships between domain entities, and use Entity Framework to map the data to the respective tables. We will see how to use the Entity Framework Code First approach to manage the following inheritance types:

- The **Table per Type (TPT)** inheritance
- The **Table per Class Hierarchy (TPH)** inheritance
- The **Table per Concrete Class (TPC)** inheritance

Implementing the TPT inheritance

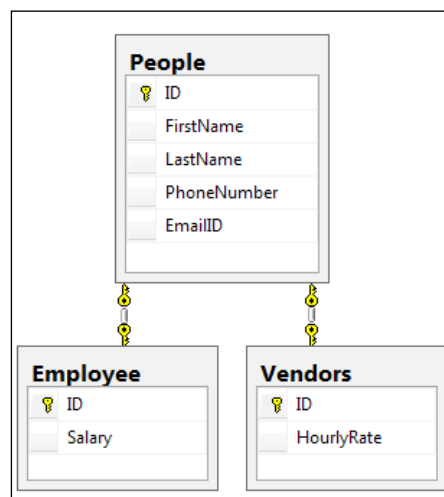
The TPT inheritance is useful when we have domain entities that have an inheritance relationship, and we want them to be modeled in our database in such a way that each domain entity will be mapped to a separate table. These tables will be related to each other using a one-to-one relationship, and this relation will be maintained in the database by a shared primary key. To illustrate this, let's take a look at an example scenario.

Let's assume a scenario where an organization maintains a database of all the people who work in a department. Some of them are their employees who get a fixed salary and some of them are vendors who are hired on an hourly rate. To model this, we will create three domain entities, `Person`, `Employee`, and `Vendor`. The `Person` class will be the base class, and the `Employee` and the `Vendor` classes will be derived from the `Person` class. Let's try to illustrate this entity inheritance relationship to put things in perspective:



The inheritance relationship between domain entities

In the TPT inheritance, we want a separate table for each of the domain classes, where these tables will have a shared primary key. So the generated database should look like the following:



The desired database schema to implement the TPT inheritance

Now let's start by creating the `Person` base class:

```
public class Person
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public string EmailId { get; set; }
}
```

For the `Person` entity, we will be using the Entity Framework conventions to map the class to the database table. Now, let's create the `Employee` class, and use the data annotations to map this table data to the `Employee` table:

```
[Table("Employee")]
public class Employee : Person
{
    public decimal Salary { get; set; }
}
```

Similarly, for the `Vendor` entity we will specify the data annotations to map the `Vendor` class to the `Vendors` table:

```
[Table("Vendors")]
public class Vendor : Person
{
    public decimal HourlyRate { get; set; }
}
```

So the `Vendor` class will now be mapped to the `Vendors` table in the database.

Now, let's define the `DbContext` class that will be used to perform the operations on these entities:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Person> People { get; set; }
}
```

In the preceding code, the `DbContext` class only contains the `DbSet` for our `Person` model. Since the other two domain models are derived from this model, we can add them to this `DbSet` collection, and Entity Framework will use polymorphism to use the actual domain model. Also, since we are using data annotations to specify the entity to table mapping, we don't have to specify anything extra in this class.

If we want to do this configuration using the fluent API, then we have to specify the mapping in the DbContext class as follows:



```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Person> People { get; set; }

    protected override void
    OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>().
        ToTable("Employee");
        modelBuilder.Entity<Vendor>().
        ToTable("Vendors");
    }
}
```

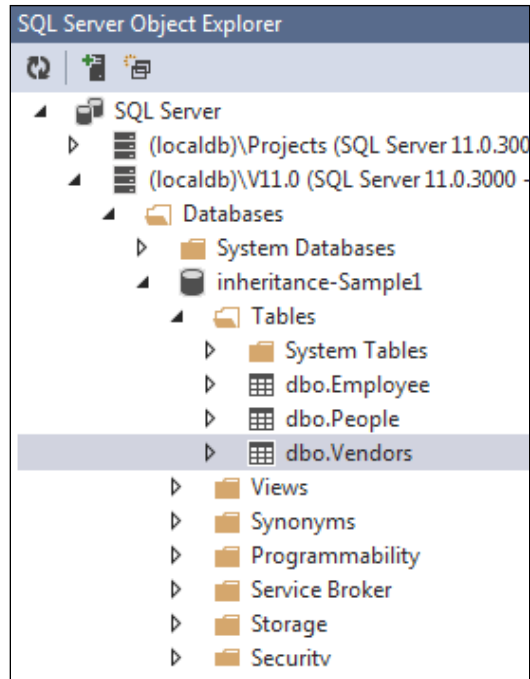
Now let's try to use these domain entities to create an Employee and a Vendor type:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Employee employee = new Employee();
    employee.FirstName = "Employee 1";
    employee.LastName = "Employee 1";
    employee.PhoneNumber = "1234567";
    employee.Salary = 50000;
    employee.EMailId = "employee1@test.com";

    Vendor vendor = new Vendor();
    vendor.FirstName = "vendor 1";
    vendor.LastName = "vendor 1";
    vendor.PhoneNumber = "1234567";
    vendor.HourlyRate = 100;
    vendor.EMailId = "vendor1@test.com";

    db.People.Add(employee);
    db.People.Add(vendor);
    db.SaveChanges();
}
```

In the preceding code, we created an object of type `Employee` and an object of type `Vendor`. We then added this to the `People` collection of the `DbContext` class. When we run the code, the database will be created for this schema. Let's see what tables are created in the database when we execute the preceding code:



The created database using the Code First approach
for the TPT inheritance

In the preceding screenshot, we can see that a separate table has been created for each domain model. Let's see what the actual data in these tables looks like:

People: Query(hf...ENTS\TODODB.MDF) ✕					
	ID	FirstName	LastName	PhoneNumber	EmailID
	2	Employee 1	Employee 1	1234567	employee1@test.com
	3	vendor 1	vendor 1	1234567	vendor1@test.com

Employee: Query...ENTS\TODODB.MDF) ✕		
	ID	Salary
	2	50000

Vendors: Query(h...ENTS\TODODB.MDF) ✕		
	ID	HourlyRate
	3	100

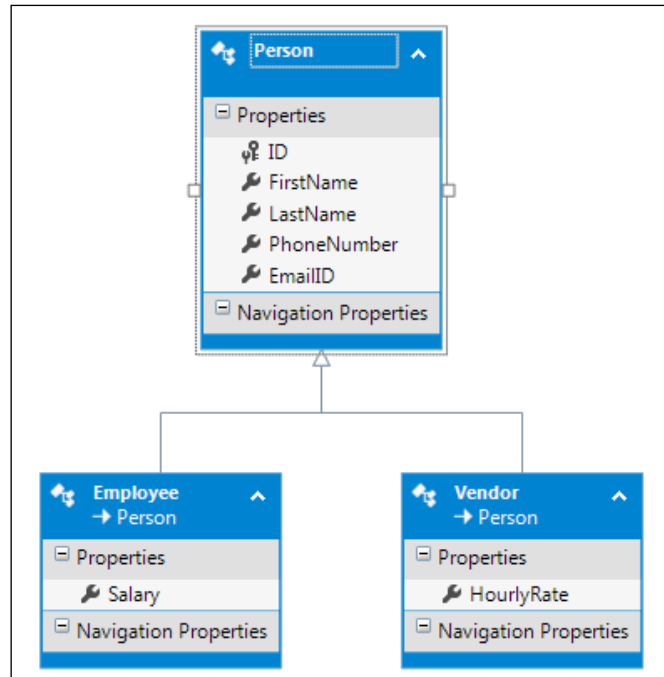
The data inserted into the table using the Code First approach for the TPT inheritance

We can see that the respective tables contain the respective data, and there is a shared primary key between these tables to facilitate a one-to-one relationship between the tables.

Implementing the TPH inheritance

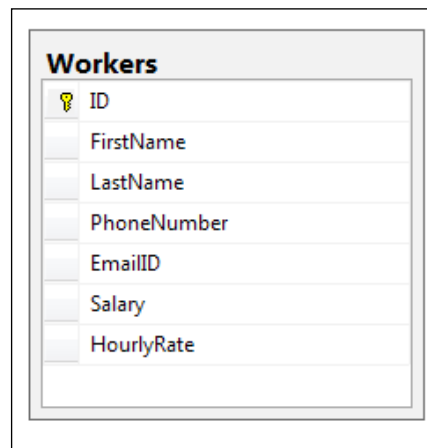
The TPH inheritance is useful when we have domain entities that have inheritance relationships, but we want them to be modeled in our database in such a way that the data from all the entity classes will be saved in a single table.

From the domain entity perspective, we want our models to still maintain the inheritance hierarchy. So, our entity model will look like this:



The inheritance relationship between domain entities

However, from the database perspective, there should only be one table to store this data. So the generated database should look like the following:



The desired database schema to implement the TPH inheritance

Now what will happen in this case is that the common fields will be populated whenever we create a type of worker. `Salary` will only contain a value if the worker is of type `Employee`. The `HourlyRate` field will be `null` in this case. If the worker is of type `Vendor`, then the `HourlyRate` field will have some value and `Salary` will be `null`.



This pattern is not very elegant from a database perspective; since we are trying to keep unrelated data in a single table, our table is not normalized. There will always be some redundant columns that contain null values if we use this approach.

Let's see how we can configure our domain entities to implement this inheritance. Let's start by creating the `Person` base class:

```
public class Person
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public string EMailId { get; set; }
}
```

For the `Person` class, we will be using the Entity Framework conventions to map the class to the database table. Now, let's create the `Employee` and `Vendor` classes. If we don't specify any configurations for mapping, Entity Framework will by default use the parent class name as the table name and map the derived class properties to that table:

```
public class Employee : Person
{
    public decimal Salary { get; set; }
}

public class Vendor : Person
{
    public decimal HourlyRate { get; set; }
}
```

Now, let's define the `DbContext` class that will be used to perform the operations on these entities:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Person> People { get; set; }
}
```

In the preceding code, the `DbContext` class only contains `DbSet` for our `Person` model. Since the other two domain models are derived from this model, we can add them to this `DbSet` collection, and Entity Framework will use polymorphism to use the actual domain model. Also, since we are using data annotations to specify the entity to table mapping, we don't have to specify anything extra in this class.

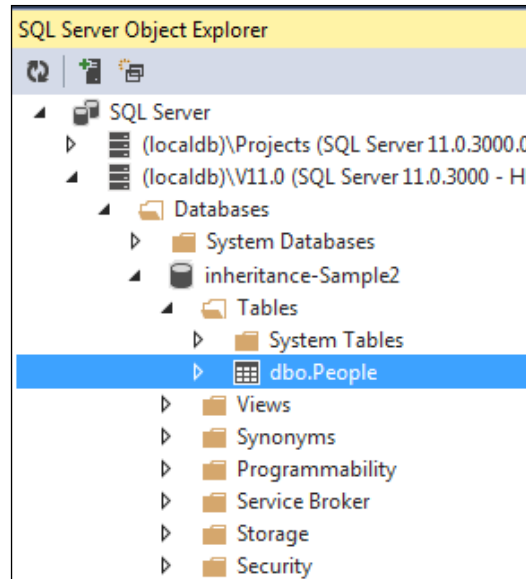
Now, let's try to use these domain entities to create an `Employee` and a `Vendor` type:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Employee employee = new Employee();
    employee.FirstName = "Employee 1";
    employee.LastName = "Employee 1";
    employee.PhoneNumber = "1234567";
    employee.Salary = 50000;
    employee.EmailId = "employee1@test.com";

    Vendor vendor = new Vendor();
    vendor.FirstName = "vendor 1";
    vendor.LastName = "vendor 1";
    vendor.PhoneNumber = "1234567";
    vendor.HourlyRate = 100;
    vendor.EmailId = "vendor1@test.com";

    db.People.Add(employee);
    db.People.Add(vendor);
    db.SaveChanges();
}
```

In the preceding code, we created an object of type `Employee` and an object of type `Vendor`. We then added this to the `People` collection of the `DbContext` class. When we run the code, the database will be created for this schema. Let's see how the tables are created in the database:



The created database using the Code First approach for the TPH inheritance

In the preceding screenshot, we can see that only the single table `People` exists in the database. Let's see what the actual data looks like in this table:

ID	FirstName	LastName	PhoneNum...	EMailId	Salary	HourlyRate	Discriminator
1	Employee 1	Employee 1	1234567	employee1@te...	50000.00	NULL	Employee
2	vendor 1	vendor 1	1234567	vendor1@test.c...	NULL	100.00	Vendor

The data inserted into the table using the Code First approach for the TPH inheritance

From the preceding screenshot, it can be seen that the `People` table contains the data from both our models. One interesting thing to note here is that Entity Framework added a column called `Discriminator` that can be used to find the actual type of record, that is, `Employee` or `Vendor` from the `Person` table.

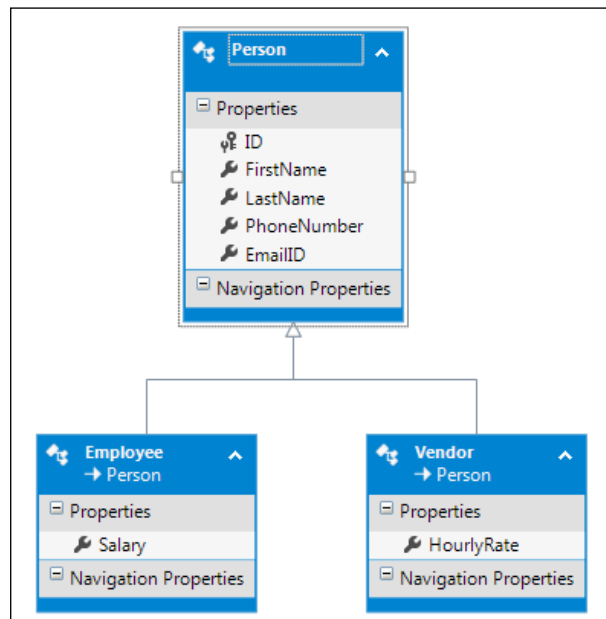


So we can say that if we don't provide any specific configuration and have inheritance between the domain entities, by default Entity Framework will treat it as the TPH inheritance and put the data in a single table.

Implementing the TPC inheritance

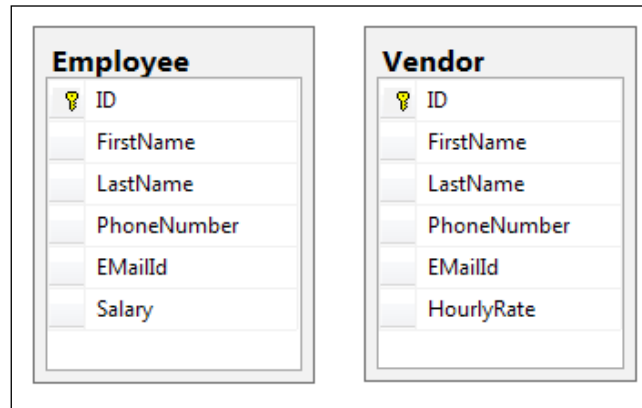
The TPC inheritance is useful when we have multiple domain entities derived from a base entity, and we want to model them in our database in such a way that the data from all the concrete classes should be kept in separate tables, and there should be no table for the abstract class entity.

From the domain entity perspective, we want our models to still maintain the inheritance hierarchy. So our entity model will look like this:

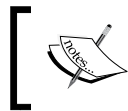


The inheritance relationship between domain entities

However, from the database perspective, there should be tables for all the concrete classes and no table for the abstract class should be created. The generated database should look like the following:



The desired database schema to implement the TPC inheritance



One of the major problems in such a database design is the duplication of columns in the tables, which is not recommended from a database normalization perspective.

Let's start by creating the Person base class. Since we are saying that this class should be abstract, let's mark this class as abstract too:

```
public abstract class Person
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public string EMailId { get; set; }
}
```

Now let's create the Employee and Vendor classes. We will derive these classes from the Person class:

```
public class Employee : Person
{
    public decimal Salary { get; set; }
}
```

```
public class Vendor : Person
{
    public decimal HourlyRate { get; set; }
}
```

Now, let's define the DbContext class that will be used to perform the operations on these entities:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Person> People { get; set; }
}
```

This is the tricky part. If we leave the entities and context classes as they are right now, it will be treated as a TPH inheritance. If we need to implement the TPC inheritance, then we need to map the classes in the context class using the fluent API:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Person> People { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Employee>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("Employee");
        });

        modelBuilder.Entity<Vendor>().Map(m =>
        {
            m.MapInheritedProperties();
            m.ToTable("Vendors");
        });
    }
}
```

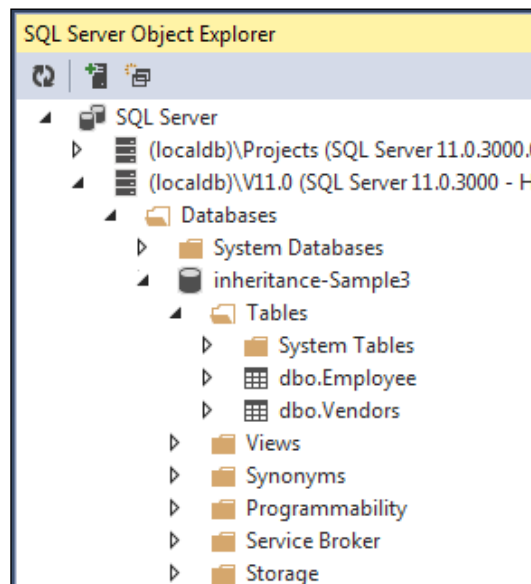
In the preceding code, we checked the actual type of the object in the model creating function and then mapped it to the respective table based on the type of object.

Now let's try to use these domain entities to create an Employee and a Vendor type:

```
using (SampleDbEntities db = new SampleDbEntities())
{
    Employee employee = new Employee();
}
```

```
employee.FirstName = "Employee 1";  
employee.LastName = "Employee 1";  
employee.PhoneNumber = "1234567";  
employee.Salary = 50000;  
employee.EMailId = "employee1@test.com";  
  
Vendor vendor = new Vendor();  
vendor.FirstName = "vendor 1";  
vendor.LastName = "vendor 1";  
vendor.PhoneNumber = "1234567";  
vendor.HourlyRate = 100;  
vendor.EMailId = "vendor1@test.com";  
  
db.People.Add(employee);  
db.People.Add(vendor);  
db.SaveChanges();  
}
```

In the preceding code, we created an object of type `Employee` and an object of type `Vendor`. We then added these to the `People` collection of the `DbContext` class. When we run the code, the database will be created for this schema. Let's see how the tables are created in the database:



The created database using the Code First approach for the TPC inheritance

In the preceding screenshot, we can see that there is a table for each of the concrete class types, that is, `Employee` and `Vendor`. Let's see what the actual data looks like in these tables:

	ID	FirstName	LastName	PhoneNumber	EMailId	HourlyRate
▶	1	vendor 1	vendor 1	1234567	vendor1@test.c...	100
*	NULL	NULL	NULL	NULL	NULL	NULL

	ID	FirstName	LastName	PhoneNumber	EMailId	Salary
▶	1	Employee 1	Employee 1	1234567	employee1@t...	50000
*	NULL	NULL	NULL	NULL	NULL	NULL

The data inserted into the table using the Code First approach for the TPC inheritance

From the preceding screenshot, it can be seen that the `Employee` and `Vendor` tables contain all the information that has been pushed to the database using the `People` class, which was an abstract class.



The insert is successful but there will be an exception in the application. This exception is because when Entity Framework tries to access the values that are in the abstract class, it finds two records with the same ID, whereas the ID column is specified as the primary key and that has two records with the same values is a problem in this scenario. This exception clearly shows that the store/database generated identity columns will not work with the TPC inheritance.

If we want to use the TPC inheritance, then either we need to either use GUID based IDs, pass the ID from the application, or perhaps use some database mechanism that can maintain the uniqueness of the autogenerated columns across multiple tables.

Summary

In this chapter, we saw how to use the Entity Framework Code First approach to use domain entities in our application and persist data in the database. We also looked at how to manage multiplicity relationships between entities, and use Entity Framework to map these relationships to the database. We also looked at how to use Entity Framework to manage entities involved in inheritance relationships and create the desired schema for these entities. In the next chapter, we will see how we can manage the database creation process, and insert some dummy data into the database.

7

Entity Framework Code First – Managing Database Creation and Seeding Data

So far, we have seen how we can use various Entity Framework approaches. We have seen the details of the Database First approach, and how we can use Entity Framework's Code First approach to create the database from our domain entities. Now, let's see how we can use Entity Framework to manage database creation. We will take a look at how we can specify the location and name of the database. We will also see how we can initialize the database using Entity Framework's provided mechanism. Finally, we will take a look at how we can seed the database with some initial data using Entity Framework.

Managing database connections

The first thing that comes to mind when we use Entity Framework code first is how we will manage the various connection parameters. In this section, we will see how we can manage the various connection parameters such as database location and database name.

Managing connections using a configuration file

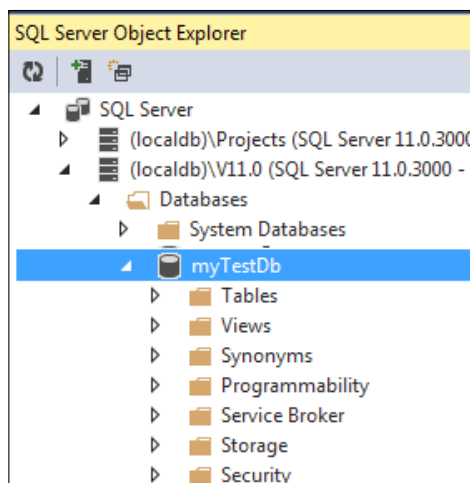
In the previous chapters, we saw that if we created `ConnectionString` with the same name as the `DbContext` class in our configuration file, then Entity Framework will use this connection string and figure out the location of the database and the database name. For example, our `DbContext` class looks like the following:

```
public class SampleDbEntities : DbContext
{
    // Code here
}
```

If we define a connection string with the same name as the `DbContext` class, Entity Framework will use that connection string to perform the database operations:

```
<connectionStrings>
  <add name="SampleDbEntities" connectionString="Data
    Source=(LocalDb)\v11.0;Initial Catalog=myTestDb;Integrated
    Security=SSPI;AttachDBFilename=|DataDirectory|\myTestDb.mdf"
    providerName="System.Data.SqlClient" />
</connectionStrings>
```

What happens when we put this `connectionString` in the configuration file is that when we run the application, Entity Framework will look for the name of our context class, that is, `SampleDbEntities`, and looks for `connectionString` with the same name in the configuration file. It will then use this connection string to figure out which database provider should be used. In this case, we are using `SqlClient`, so SQL Server will be used. It will then check for the database location, which is the current data directory in this case. It will then create a database file named `myTestDb.mdf` in the specified location, and create a database named `myTestDb` by looking at the `Initial Catalog` attribute of `connectionString`.



A database screenshot showing the created database

From the preceding screenshot, we can see that when we run the application, the database has been created with the name and location as specified in our `connectionString`.



Using the configuration files to specify the database location and name is perhaps the easiest and most efficient way to control the connection parameters for our context class. Another benefit of this approach is that if we want to have a separate `connectionString` for the development, production, and staging environments, then it's just a matter of changing the `connectionString` in the configuration file and pointing it to a required database at the time of deployment.

An important thing to note here is that the `connectionString` defined in the configuration file has the highest precedence, and it will override all the other connection parameters specified otherwise (which we will see in the coming sections).



From a best practices perspective, using the configuration file is perhaps not recommended. Injecting the connection string is a more preferred way as it gives more control and oversight to the developer.

Using the existing ConnectionString


Let's consider a scenario where we already have a `connectionString` defined in the database, and we want to use the same `connectionString` with our context class. Let's say we have the following `connectionString` defined in our configuration class:

```
<connectionStrings>
  <add name="AppConnection" connectionString="Data Source=(LocalDb)\
v11.0;Initial Catalog=testDb;Integrated Security=SSPI;AttachDBFilename
=|DataDirectory|\testDb.mdf" providerName="System.Data.SqlClient" />
</connectionStrings>
```

If we want to use this `connectionString` with our context class, then we will have to pass this `connectionString` name to the `DbContext` constructor. Let's see how we can change our context class to use this `connectionString`:

```
public class SampleDbEntities : DbContext
{
    public SampleDbEntities()
        :base("name=AppConnection")
    {
    }
}
```

What we are doing here is that we are passing the `connectionString` name to the base `DbContext` class' constructor, and by doing this, our context class will start using this `connectionString`.

 If we still have another `connectionString` defined in the configuration file that has the same name as the context class, then that `connectionString` will be used. Whatever changes we made in our context class to pass the `connectionString` name will simply be ignored.

Using an existing connection

Another scenario is that only a part of our application uses the Entity Framework Code First approach (usually in the case of old applications), and we want to use an existing database connection with our context class. If we want to achieve this, then we need to pass the connection object to the base `DbContext` constructor. Let's see how our context class will change if we need to achieve this:

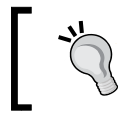
```
public class SampleDbEntities : DbContext
{
    public SampleDbEntities(DbConnection con)
```

```

        :base(con, contextOwnsConnection: false)
    {
    }
}

```

In the preceding code, we defined a constructor that takes the connection object and then passes it to the base `DbContext` object. Another parameter that we pass to the `DbContext` constructor is `contextOwnsConnection`. We pass this parameter as `false` because this connection is being passed to this context class from the outside, and someone would want to reuse this connection after the context goes out of scope.



If we pass `contextOwnsConnection` as `true`, then the context class will close the connection as soon as it goes out of scope.

Managing database initialization

When we run our Entity Framework Code First application for the first time, Entity Framework will do the following:

1. Check the `DbContext` class that is being used.
2. Find the `connectionString` that should be used with this context class.
3. Find the domain entities and extract the schema-related information.
4. The database will be created.
5. Data will be inserted into the system.

What will happen if the database already exists? What will happen if the application is run a second time? Should it create the database again or simply use the existing database? These are a few questions that can be answered simply by understanding the database initialization process of Entity Framework.

Once the schema information is extracted, Entity Framework will use database initializers to push the schema information to the database. There are various possible strategies for database initializers, and Entity Framework's default strategy is to create the database if it doesn't exist, and use the existing one if it exists. However, it is also possible to override this default behavior. The possible database initialization strategies are as follows:

- **CreateDatabaseIfNotExists:** This will create the database if it does not exist, and if it exists, then it will use the existing database. If there is a mismatch in the schema information extracted from the domain models and the actual database schema, an exception will be thrown.

- `DropCreateDatabaseAlways`: If this strategy is used, the database is dropped every time the application is run. This is mostly useful in the early stages of the development cycle when we usually design our domain entities. It is also useful from a unit tests perspective.
- `DropCreateDatabaseIfModelChanges`: If this strategy is used, it will create the database if it does not exist, and if it exists, then it will use the existing database. However, the important thing to note here is that if there is a mismatch in the schema information extracted from the domain models and the actual database schema, the database will be dropped and created again.
- `MigrateDatabaseToLatestVersion`: If we use this initializer, then Entity Framework will automatically update the database schema whenever our entity model is updated. The important part to note here is that it will update the schema without losing the data or update the already existing database objects in the existing database.

 The `MigrateDatabaseToLatestVersion` initializer is available only from Entity Framework version 4.3 onwards.

Setting the initialization strategy

By default, Entity Framework uses `CreateDatabaseIfNotExists` as the default initializer. To override the default database initialization strategy, we need to use the `Database.SetInitializer` method in the constructor of our `DbContext` class. Let's see how this can be done to use the `DropCreateDatabaseIfModelChanges` strategy rather than the default strategy:

```
public class SampleDbEntities : DbContext
{
    public SampleDbEntities()
        :base("name=AppConnection")
    {
        Database.SetInitializer<SampleDbEntities>(new DropCreateDatabaseIfModelChanges<SampleDbEntities>());
    }
}
```

In the preceding code, whenever the object of our context class is created, the `Database.SetInitializer` method will be called, and it will set the database initialization strategy to `DropCreateDatabaseIfModelChanges`.

If we are in the production environment, then we don't want to lose the existing data by mistake. It is also possible to turn off the initializers. We just need to pass null to the `Database.SetInitializer` method. The following code shows how this can be done:



```
public class SampleDbEntities : DbContext
{
    public SampleDbEntities()
        :base("name=AppConnection")
    {
        Database.SetInitializer<SampleDbEntities>
            (null);
    }
}
```

Seeding data

So far, we have seen how we can take control over the location of the database and name of the database and how we can set the database initialization strategy. One thing to note here is that the database that will be created by Entity Framework will always be an empty database irrespective of the strategy we chose to initialize the database.

There are many scenarios where we might want to have some values inserted in the database once it is created but before it is used. Master tables and lookup tables are such examples. Also, in some cases, the application deployment wants the data for the administrators to be populated with their default credentials. Or perhaps we want some dummy data in our tables to test how our application behaves in a specific scenario.

Seeding data is very important when we are using the `DropCreateDatabaseAlways` or `DropCreateDatabaseIfModelChanges` initialization strategies, since every time we run the application, the database will be recreated (in the latter case, every time the model is changed), and manually inserting the data in the tables after each database creation will become very tedious. Let's see how we can use Entity Framework to insert/seed dummy data after the database has been created.

To seed some initial data in the database, we need to create our own database initializer class that will:

- Derive data from the existing database initializer class
- Seed the database during creation

Let's say we want to use the `DropCreateDatabaseAlways` strategy in our application. We have a model, `Employer`, defined in our code as follows:

```
public class Employer
{
    public int ID { get; set; }
    public string EmployerName { get; set; }
}
```

We also have our `DbContext` class created to use this model with the Entity Framework Code First approach:

```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Employer> Employers { get; set; }
}
```

Now we want to seed some data each time a database is created. The first thing we need to do is to create our own database initialization class, which should be derived from the `DropCreateDatabaseAlways` class:

```
public class MyCustomDropCreateDatabaseAlways
    : DropCreateDatabaseAlways<SampleDbEntities>
{
}
}
```

In the preceding code, we created a custom database initializer class and derived it from the `DropCreateDatabaseAlways` class and used `SampleDbEntities` for the initialization. Now what we need to do to seed the data using this initializer class is to override the `Seed` method of the `DropCreateDatabaseAlways` class:

```
public class MyCustomDropCreateDatabaseAlways
    : DropCreateDatabaseAlways<SampleDbEntities>
{
    protected override void Seed(SampleDbEntities context)
    {
    }
}
}
```

In the preceding code, we are overriding the `Seed` method of the `DropCreateDatabaseAlways` class. This method has access to the `DbContext` class, and it can use this `DbContext` class to insert some data in the database:

```
public class MyCustomDropCreateDatabaseAlways
    : DropCreateDatabaseAlways<SampleDbEntities>
```

```
{
    protected override void Seed(SampleDbEntities context)
    {
        Employer employer1 = new Employer();
        employer1.EmployerName = "Employer1";

        Employer employer2 = new Employer();
        employer2.EmployerName = "Employer2";

        Employer employer3 = new Employer();
        employer3.EmployerName = "Employer3";

        context.Employers.Add(employer1);
        context.Employers.Add(employer2);
        context.Employers.Add(employer3);

        base.Seed(context);
    }
}
```

What the preceding code is doing is creating the `Employer` objects and adding them to the `Employers` collection of the `DbContext` class. One important thing to note in this code is that we are not calling `DbContext.SaveChanges()` because it will automatically be called in the base class.

Now let's use this initializer with our context class:

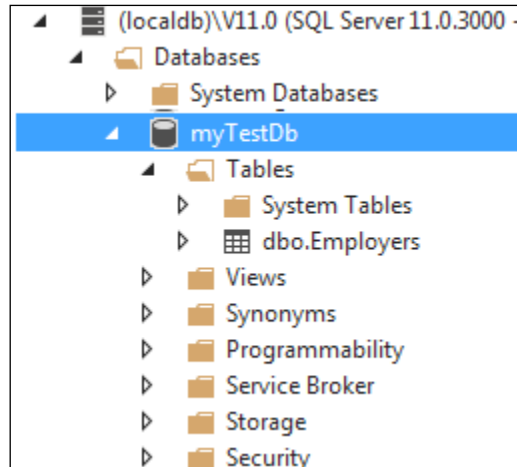
```
public class SampleDbEntities : DbContext
{
    public virtual DbSet<Employer> Employers { get; set; }

    public SampleDbEntities()
    {
        Database.SetInitializer<SampleDbEntities>(new
        MyCustomDropCreateDatabaseAlways());
    }
}
```

Here, we are setting our database initializer `MyCustomDropCreateDatabaseAlways`, as the initializer should be used whenever the object of the context class is created. Let's try to run the application with the following code and see what happens:

```
using(SampleDbEntities db = new SampleDbEntities())
{
}
}
```

When we run the application, and simply create the `DbContext` object, the database will be created for us.



A database screenshot showing the created database


In the preceding screenshot, we can see that the database contains a table corresponding to our `Employer` entity class. Also, let's take a look at the data in the `Employers` table:

A screenshot of the SQL Server Enterprise Manager interface showing the data in the `dbo.Employers` table. The table has two columns: `ID` and `EmployerName`. The data is as follows:

ID	EmployerName
1	Employer1
2	Employer2
3	Employer3

A database screenshot showing the seeded data in the table

From the preceding screenshot, we can see that our custom initializer was successful in seeding the dummy data in the `Employers` table.

 We can also execute raw SQL queries using our context class. So, it is also possible to affect the database schema from the database initialization's `Seed` method using the context class and pass raw SQL to the database.

Summary

In this chapter, we saw how we can control the various database connection parameters such as database location, database name, schema, and so on. We also saw how we can use database initializers to use the database initialization strategy that fits our application needs, and finally, we looked at how we can seed some data using database initializers while using the Code First approach.

At this juncture, we know all about setting up the database using the Database First and Code First approaches. The Model First approach is the same as the Code First approach, but we use Visual Entity Designer to create our entities rather than writing POCO classes. We know how we can manage Entity relationships and map them as database relationships; we also know how we can use entity inheritance to manage our domain entities in a better manner. In the next chapter, we will take a look at how we can use LINQ to Entities to query the Entity Data Model.

8

Querying the Entity Data Model – LINQ to Entities

So far, we have seen how we can create the Entity Data Model using the Database First approach and Code First approach of Entity Framework. We also looked at domain modeling, performing model validations, and taking control of the database connection parameters using Entity Framework. Once we have all the domain modeling done, we would want to query the object model. There are two ways in which we can query the Entity Data Model:

- LINQ to Entities
- Entity SQL

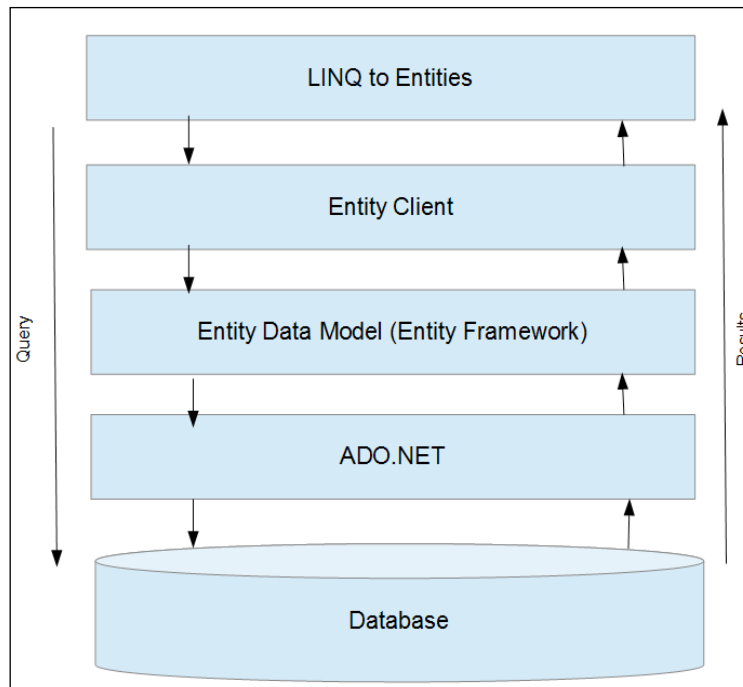
In this chapter, we will discuss how we can query the Entity Data Model using LINQ to Entities.

Understanding LINQ to Entities

Language-Integrated Query (LINQ) is a technique for querying data from .NET languages. LINQ to Entities is the mechanism that facilitates the use of LINQ to write queries against our conceptual model such as the Entity Data Model.

Since LINQ is a declarative language, let's focus on what data we need rather than how it should be retrieved. LINQ to Entities provides a nice abstraction over the Entity Data Model so that we can use LINQ to specify what data should be retrieved, and the LINQ to Entities provider will take care of accessing the database and fetching the required data for us.

When we use LINQ to Entities to execute the LINQ queries against the Entity Data Model, these LINQ queries are first compiled to determine what data we want to fetch. It will then be executed and from the application's perspective, the results will be returned as CLR objects, that is, something that .NET understands.



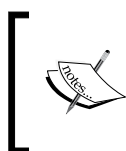
Architecture diagram showing how LINQ to Entities works

However, if we want to understand the complete process, it's crucial to understand how these queries are being executed and retrieving the results back as .NET objects. In the preceding diagram, let's visualize how LINQ to Entities fits in the overall architecture, and how our LINQ queries get executed using Entity Framework.

The preceding diagram shows that LINQ to Entities relies on `EntityClient` to be able to work with the Entity Framework conceptual data model. Let's see how LINQ to SQL executes the query and returns the result to the application:

1. The application creates a LINQ query.
2. LINQ to Entities will transform our LINQ queries to the `EntityClient` commands.
3. The `EntityClient` command will then use Entity Framework and the Entity Data Model to translate these commands into the SQL query.

4. The SQL query will then be passed to the database using the underlying ADO.NET provider.
5. The query will be executed on the database.
6. The result will be returned to the Entity Framework.
7. Entity Framework will convert the result back to CLR types such as domain entities.
8. `EntityClient` will use projects and return the required results to the application.



The `EntityClient` object resides in the `System.Data.EntityClient` namespace. We don't have to create this object explicitly. We just need to use this namespace and LINQ to Entities will take care of the rest.

If we want to use LINQ to Entities with multiple types of database, we just need to use the proper ADO.NET provider for the database, and `EntityClient` will be able to use this making our LINQ queries work seamlessly against any database.

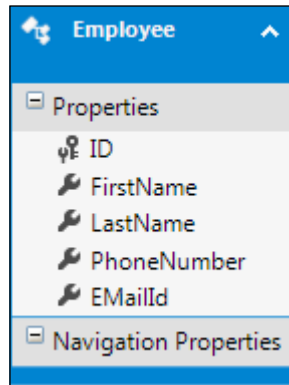
Querying data using LINQ to Entities

Now that we know how LINQ to Entities works internally, let's take a look at how we can query the Entity Data Model using LINQ. There are two ways in which we can write LINQ queries:

- The query syntax
- The method syntax

Choosing between these two approaches is purely a developer's choice. Performance wise both are the same. The query syntax is comparatively easy to understand but offers less flexibility. The method syntax, on the contrary, is a little difficult to understand but offers great flexibility. It is possible to chain multiple queries using the method syntax and thus achieve maximum results in a single statement.

To understand this better, let's write a sample LINQ query using both these approaches. Let's take an example database with only one `Employee` table. The generated Entity Data Model for this database will look like the following:



A generated entity for the `Employee` table

If we use the Code First approach, the POCO entity for this model will look like the following code: (1003OS_08_1_code.txt)

```
public partial class Employee
{
    public int ID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string PhoneNumber { get; set; }
    public string EMailId { get; set; }
}
```

The `DbContext` class for the preceding Entity Data Model will look like the following:

```
public partial class SampleDbEntities : DbContext
{
    public DbSet<Employee> Employees { get; set; }
}
```

Now, if we want to execute the LINQ queries to fetch the `Employee` data using this Entity Data Model, we can do this on the `Employees` collection of our context class. Let's say we want to retrieve all the `Employees` collection with the `LastName` value Singh. Let's see how this can be done using LINQ to Entities with the `DbContext` class:

```
using(SampleDbEntities db = new SampleDbEntities())
{
    var employees = from employee in db.Employees
                     where employee.LastName == "Singh"
                     select employee;
}
```

In the preceding code, we first created the `DbContext` object and then used LINQ to query `Employees`. Let's see how we can achieve the same using the method syntax:

```
using(SampleDbEntities db = new SampleDbEntities())
{
    var employees = db.Employees
        .Where(employee => employee.LastName == "Singh");
}
```

The preceding code also retrieves the list of `Employees` with `LastName` as `Singh` using the method syntax.

Now let's take a look at the output type being received from the LINQ query. In the preceding code snippets, we stored the query results in an implicitly typed variable called `employees`. Using LINQ to Entities, we can use implicitly typed variables as output types that will let the compiler derive the output type based on the query. Typically, this output type will be of the type `IQueryable<T>`. In our case, this will be `IQueryable<Employee>`.

It's also possible to specify the precise output type of our query by specifying the type of the return variable such as `IQueryable<Employee>` or `IEnumerable<Employee>`. So, the preceding query can also be written as follows:

```
using(SampleDbEntities db = new SampleDbEntities())
{
    IEnumerable<Employee> employees = db.Employees
        .Where(employee => employee.LastName == "Singh");
}
```

In this case, we are explicitly specifying the type of the variable that will contain the result of our LINQ query.

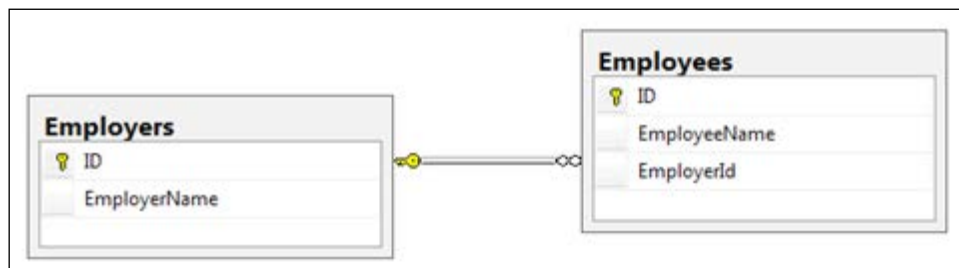


While using LINQ to Entities, it is important to understand when to use `IEnumerable` and `IQueryable`. If we use `IEnumerable`, the query will be executed immediately. If we use `IQueryable`, the query execution will be deferred until the application requests the enumeration.

Now let's see what should be considered while deciding whether to use `IQueryable` or `IEnumerable`. Using `IQueryable` gives you a chance to create a complex LINQ query using multiple statements without executing the query at the database level. The query gets executed only when the final LINQ query gets enumerated.

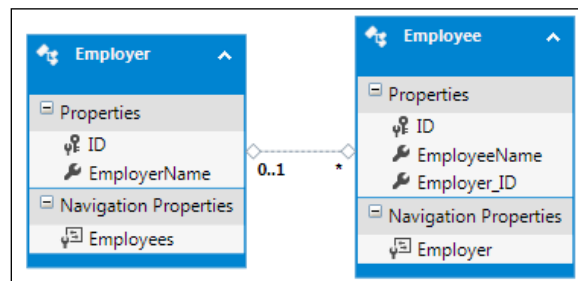
Using LINQ to Entities – an example-based approach

In this section, we will implement a few LINQ queries that we might have to write quite often while working with a data-centric application. We will be using the following database in our application:



A database schema for the sample application

Now let's generate the Entity Data Model from this database. The generated Entity Data Model will look like the following:



A generated Entity Data Model for the sample application

Let's now take a look at how to perform some common operations on this Entity Data Model using LINQ to Entities.

Executing simple queries

Let's start by looking at executing simple queries using LINQ to Entities. The first scenario could be that we want to retrieve a list of all the data from the table. Let's try to retrieve a list of all the employees using the LINQ query syntax:

```
using(SampleDbEntities db = new SampleDbEntities())
{
    var employees = from employee in db.Employees
                    select employee;
}
```

If we want to write the same query in the LINQ method syntax, then the code will look like the following:

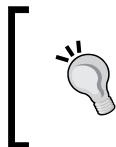
```
using(SampleDbEntities db = new SampleDbEntities())
{
    var employees = db.Employees;
}
```

What is happening in the preceding code is that we are using the `DbContext` object and accessing the `Employees` property. We can execute the LINQ queries on the `DbSet<T>` property exposed by the context class, which in our case is `Employees`. The preceding LINQ query will return a list of employees.

Let's take a look at the SQL generated by this LINQ query:

```
SELECT
    [Extent1].[ID] AS [ID],
    [Extent1].[EmployeeName] AS [EmployeeName],
    [Extent1].[Employer_ID] AS [Employer_ID]
FROM [dbo].[Employees] AS [Extent1]
```

From the preceding query, it can be seen that Entity Framework translated our LINQ query and fetched all the data from the `Employees` table.



LINQPad is an excellent tool to practice LINQ to Entities. It has the option of using Entity Framework's `DbContext` or `ObjectContext` object, and lets us use LINQ to Entities to query the database. We can also watch the generated SQL queries using LINQPad too.

Using the navigation properties with LINQ to Entities

If there is a relationship between the entities (one-to-one, one-to-many, or many-to-many), this relationship is exposed in the form of a navigation property in their respective entities. In our current example, the `Employee` entity has an `Employer` property to keep track of the employer this employee belongs to, and the `Employer` entity contains an `Employees` property that returns all the employees of that employer. Navigation properties make it very easy to navigate from one entity to its related entities. In this section, we will see how we can use LINQ to Entities to use the navigation properties to fetch data from the related entities. Let's say we want to fetch all the `Employees` property that belong to the `Employer` entity with ID as 4. Let's try to write a LINQ query to fetch the list of employees using the `Employer` entity:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = from employer in db.Employers
                    where employer.ID == 4
                    select employer.Employees;
}
```

If we want to write the same query using the method syntax, use the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = db.Employers
                    .Where(employer => employer.ID == 4)
                    .Select(employer => employer.Employees);
}
```

The preceding code will first fetch the `Employer` entity with an ID as 4, and then use the navigation property to fetch the associated employees.

Let's try another example where we want to retrieve the `Employer` entity for an `Employee` with ID as 5:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employer = from employee in db.Employees
                   where employee.ID == 5
                   select employee.Employer;
}
```

If we want to write the same query using the method syntax, use the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employer = db.Employees
```

```

        .Where(employee => employee.ID == 5)
        .Select(employee => employee.Employer);
    }

```

The preceding code will first fetch the `Employee` entity where the `ID` is 5, and then it will fetch the employer associated with this `Employee` entity using the navigation property. Let's take a look at the generated SQL for this LINQ query:

```

SELECT
    [Extent2].[ID] AS [ID],
    [Extent2].[EmployerName] AS [EmployerName]
FROM    [dbo].[Employees] AS [Extent1]
LEFT OUTER JOIN [dbo].[Employer] AS [Extent2]
    ON [Extent1].[Employer_ID] = [Extent2].[ID]
WHERE 5 = [Extent1].[ID]

```

From the preceding SQL, we can see that Entity Framework was able to use the navigation properties in our LINQ query and created the appropriate SQL to fetch the records matching our criteria.

Filtering data using LINQ to Entities

We have seen how we can retrieve all the records using LINQ to Entities. If we want to filter the records based on some criteria, we can do this using `Where` in the LINQ query. Let's say we want to retrieve the `Employees` value who have `ID` greater than 100 but less than 500. Let's see how we can achieve this using LINQ query syntax:

```

using(SampleDbEntities db = new SampleDbEntities())
{
    var employees = from employee in db.Employees
                    where employee.ID > 100 && employee.ID < 500
                    select employee;
}

```

If we want to write the same query in the LINQ method syntax, then the code will look like the following:

```

using(SampleDbEntities db = new SampleDbEntities())
{
    var employees = db.Employees.Where(employee => employee.ID > 100
    && employee.ID < 500);
}

```

The preceding code will fetch all the `Employee` records that match the criteria provided in the `Where` clause. The `EntityClient` property will be able to take the `Where` clause, and let Entity Framework know that the data requested should be filtered based on the criteria provided in the `Where` clause.

Let's take a look at the SQL generated by this LINQ query:

```
SELECT
    [Extent1].[ID] AS [ID],
    [Extent1].[EmployeeName] AS [EmployeeName],
    [Extent1].[Employer_ID] AS [Employer_ID]
FROM [dbo].[Employees] AS [Extent1]
WHERE ([Extent1].[ID] > 100) AND ([Extent1].[ID] < 500)
```

From the preceding query, it can be seen that Entity Framework translated our LINQ query and has put a `Where` clause to filter the data that is being requested from the `Employees` table. This is very important from the performance perspective.

We can create complex filtering expressions using C# conditional operators and `EntityClient`, and Entity Framework will take care of generating the appropriate SQL and returning the results. For example, we will use the following LINQ query:

```
using(SampleDbEntities db = new SampleDbEntities())
{
    from employee in db.Employees
    where employee.ID > 100 && employee.ID < 500
    || employee.ID == 1
    || employee.Employer_ID == 5
    select employee;
}
```



Then, Entity Framework will generate the following SQL query for this:

```
SELECT
    [Extent1].[ID] AS [ID],
    [Extent1].[EmployeeName] AS [EmployeeName],
    [Extent1].[Employer_ID] AS [Employer_ID]
FROM [dbo].[Employees] AS [Extent1]
WHERE (([Extent1].[ID] > 100) AND ([Extent1].[ID] < 500)) OR (1 = [Extent1].[ID]) OR (5 = [Extent1].[Employer_ID])
```

Thus, it can be seen that using LINQ to Entities we can formulate pretty complex queries very easily.

Using LINQ projections with LINQ to Entities

In the previous section, we discussed how to filter the data and return the complete entities. LINQ projections essentially mean that instead of returning a complete entity we want to return either a subset of properties from that entity in the form of an object or perhaps, we want to return an object that contains properties from multiple entities.

Projections are particularly useful when we are dealing with **ViewModel** in our application. We can return a ViewModel type object directly from the LINQ query. Let's take a scenario where we want to retrieve a list of all the employers where the result should contain the `EmployerName` value and a list of all the employees that are associated with this employer. Let's see how we can do this using projections in LINQ to Entities:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employers = from employer in db.Employers
                    select new
                    {
                        EmployerName = employer.EmployerName,
                        EmployeeList = employer.Employees
                    };
}
```

In the preceding code, we returned a new collection of objects where each object will contain two properties. The first property `EmployerName` is the name of the employer and the `EmployeeList` property contains a list of employees that are associated with the employer. In this case, the compiler will create an anonymous object and assign the return value to it. If we have a type that defines the return type (perhaps a `ViewModel`), then we can return the result of that type too. Let's see how this can be done:

```
class EmployeeListWithEmployerName
{
    public string EmployerName { get; set; }
    public ICollection<Employee> EmployeeList { get; set; }
}

using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    IQueryable<EmployeeListWithEmployerName> employers = from
employer in db.Employers
                    select new EmployeeListWithEmployerName
                    {

```



```
        EmployerName = employer.EmployerName,  
        EmployeeList = employer.Employees  
    };  
}
```

The preceding code will now return a specific type of object, `EmployeeListWithEmployerName`, instead of relying on the compiler to create the anonymous type to handle the result. If we want to write the same query using the method syntax, the query will look like the following:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())  
{  
    IQueryable<EmployeeListWithEmployerName> employeeList =  
        db.Employers.Select  
        (  
            employer =>  
                new EmployeeListWithEmployerName  
                {  
                    EmployerName = employer.EmployerName,  
                    EmployeeList = employer.Employees  
                }  
        );  
}
```

The preceding code is essentially doing the same thing as in the previous query but uses the method syntax instead. Using projections with LINQ to Entities is very useful because it enables us to return complex types that contain data from multiple entities. Entity Framework will take care of creating the appropriate SQL query accordingly.



Handling results in `IQueryable<T>` also improves the performance, as the generated SQL will not be executed until the items from the result are enumerated.

Grouping using LINQ to Entities

Grouping is also very essential when it comes to working with data-centric applications. In this section, we will see how we can use LINQ to Entities to perform grouping operations. Let's say we want to fetch the list of all the employees grouped by Employer IDs. Let's see how we can do this using the LINQ to Entities query syntax:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())  
{  
    var result = from employee in db.Employees  
                  group employee by employee.Employer_ID
```

```

        into employeeGroup
        select new
        {
            EmployerID = employeeGroup.Key,
            EmployeeEntity = employeeGroup
        };
    }

```

If we write this query in the form of the method syntax, it will look like the following:

```

using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var result = db.Employees.GroupBy
        (employee => employee.Employer_ID)
        .Select
        (
            employeeGroup =>
            new
            {
                EmployerID = employeeGroup.Key,
                EmployeeEntity = employeeGroup
            }
        );
}

```

The preceding code will group the `Employee` records by `Employer_ID`. The results will be returned as a projection that will create an anonymous object. The `Employer_ID` property of the result will contain the ID of the employer with which we have performed grouping. The `EmployeeEntity` property will contain a collection of `Employees` found for this `EmployerID`.

Ordering using LINQ to Entities

Ordering the records in ascending or descending order of any particular column property is also a very frequently used operation. We can use LINQ to Entities to pass the order by clause, and Entity Framework will take care of retrieving the results for us. Let's try to retrieve the list of `Employees` in ascending order of the IDs:

```

using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = from employee in db.Employees
                    orderby employee.ID
                    select employee;
}

```

The method syntax equivalent of this query will be as follows:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = db.Employees
        .OrderBy(employee => employee.ID);
}
```

What this code does is that it passes the LINQ query to the `EntityClient`, and the `EntityClient` will see that the order by clause has been requested for a property. Let's take a look at the generated SQL:

```
SELECT
    [Extent1].[ID] AS [ID],
    [Extent1].[EmployeeName] AS [EmployeeName],
    [Extent1].[Employer_ID] AS [Employer_ID]
FROM [dbo].[Employees] AS [Extent1]
ORDER BY [Extent1].[ID] ASC
```

From the preceding code, we can see that the `EntityClient` has generated the SQL appropriately using the order by clause with the corresponding column.

If we want to order in a descending manner, we have to put the keyword `descending` after the `orderby` property:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = from employee in db.Employees
        orderby employee.ID descending
        select employee;
}
```

If the descending order is needed in the method syntax, we have to use the `OrderByDescending` function instead of `OrderBy`:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = db.Employees
        .OrderByDescending(employee => employee.ID);
}
```

Now the preceding code will return the results ordered in descending order of their IDs. We can also check the generated SQL as follows:

```
SELECT
    [Extent1].[ID] AS [ID],
    [Extent1].[EmployeeName] AS [EmployeeName],
    [Extent1].[Employer_ID] AS [Employer_ID]
FROM [dbo].[Employees] AS [Extent1]
ORDER BY [Extent1].[ID] DESC
```

We can see that the `OrderByDescending` function has taken effect in the generated SQL too.

Aggregate operators with LINQ to Entities

Let's now take a look at how we can use aggregate operators using LINQ to Entities. We will see how we can perform the following aggregate operations using LINQ to Entities:

- Count
- Sum
- Min
- Max
- Average

Count

Let's start with getting the count of the number of `Employees` that belong to `Employer` with ID as 4 using the LINQ query syntax:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    int numberEmployees = (from employee in db.Employees
                           where employee.Employer_ID == 4
                           select employee).Count();
}
```

One important thing to note here is that `Count` has no query syntax, so if we want to retrieve the count, we need to mix the query syntax, and use `Count` as the method syntax. If we want to convert this whole query into the method syntax, it can be written as follows:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    int numberEmployees = db.Employees
        .Where(employee => employee.Employer_ID == 4)
        .Count();
}
```

In the preceding code, we first retrieved a list of `Employees` whose `Employer_ID` is 4. We executed the `Count` function on the query that will effectively return the number of `Employees` with `Employer_ID` as 4. Let's take a look at the generated SQL for this query:

```
SELECT
    [GroupBy1].[A1] AS [C1]
FROM ( SELECT
        COUNT(1) AS [A1]
      FROM [dbo].[Employees] AS [Extent1]
      WHERE 4 = [Extent1].[Employer_ID]
    ) AS [GroupBy1]
```

From the preceding query, we can see that the `Count` LINQ query has actually been translated to the SQL query by Entity Framework.



It is important to understand that all the aggregate functions can be used only in the method syntax and not in the query syntax. If we want to use the aggregate operators in the query syntax, then we will have to use the mixed syntax, as shown in the preceding code. In further examples, we will only be using the method syntax to query the Entity Data Model.

Sum

Now, let's take a look at how we can use the `Sum` function with LINQ to Entities. Let's say the `Employee` table also contains a column for `Salary`. The `Employee` entity will also contain a `Salary` property corresponding to the `Salary` column. Let's see how we can retrieve the sum of all the `Salary` values of `Employees` that belong to Employer with ID as 4:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    int sumOfSalaries = db.Employees
        .Where(employee => employee.Employer_ID == 4)
        .Select(employee => employee.Salary)
        .Sum();
}
```

In the preceding code, we first retrieved the salary of `Employees` whose `Employer_ID` is 4. We executed the `Sum` function on the query that will return the total salary of all the employees that belong to Employer with ID as 4. Let's take a look at the generated SQL for this query:

```
SELECT
    [GroupBy1].[A1] AS [C1]
FROM ( SELECT
        SUM([Extent1].[Salary]) AS [A1]
      FROM [dbo].[Employees] AS [Extent1]
      WHERE 4 = [Extent1].[Employer_ID]
    ) AS [GroupBy1]
```

From the preceding query, we can see that the `Sum` function in our LINQ query has actually been translated into the appropriate SQL by Entity Framework and has thus fetched only the relevant data from the database.

Min

Now, let's take a look at how we can use the `Min` function with LINQ to Entities. If we want to retrieve the minimum salary from the employees that belong to Employer with ID as 4, run the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    int minimumSalary = db.Employees
        .Where(employee => employee.Employer_ID == 4)
        .Select(employee => employee.Salary)
        .Min();
}
```

In the preceding code, we first retrieved the salary of Employees whose Employer_ID is 4. We executed the Min function on the query which will then return the minimum salary of the employees that belong to Employer with ID as 4. Let's take a look at the generated SQL for this query:

```
SELECT
    [GroupBy1].[A1] AS [C1]
FROM ( SELECT
        MIN([Extent1].[Salary]) AS [A1]
      FROM [dbo].[Employees] AS [Extent1]
      WHERE 4 = [Extent1].[Employer_ID]
    ) AS [GroupBy1]
```

From the preceding query, we can see that the Min function in our LINQ query has actually been translated into the appropriate SQL by Entity Framework and has thus fetched only the relevant data from the database.

Max

Now, let's take a look at how we can use the Max function with LINQ to Entities. If we want to retrieve the maximum salary from the employees that belong to Employer with ID as 4, use the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    int maximumSalary = db.Employees
        .Where(employee => employee.Employer_ID == 4)
        .Select(employee => employee.Salary)
        .Max();
}
```

In the preceding code, we first retrieved the salaries of the employees whose Employer_ID is 4. We executed the Max function on the query which will then return the maximum salary of the employees that belong to Employer with ID as 4. Let's take a look at the generated SQL for this query:

```
SELECT
    [GroupBy1].[A1] AS [C1]
FROM ( SELECT
        MAX([Extent1].[Salary]) AS [A1]
      FROM [dbo].[Employees] AS [Extent1]
      WHERE 4 = [Extent1].[Employer_ID]
    ) AS [GroupBy1]
```

From the preceding query, we can see that the Max function in our LINQ query has actually been translated to the appropriate SQL by Entity Framework and has thus fetched only the relevant data from the database.

Average

Let's now take a look at how we can calculate the average value using the `Average` function with LINQ to Entities. If we want to retrieve the average salary of the employees that belong to `Employer` with ID as 4, run the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    double averageSalary = db.Employees
        .Where(employee => employee.Employer_ID == 4)
        .Select(employee => employee.Salary)
        .Average();
}
```

In the preceding code, we first retrieved the salaries of the employees whose `Employer_ID` is 4. We executed the `Average` function on the query which will then return the average salary of `Employees` who belong to `Employer` with ID as 4. We can take a look at the generated SQL for this query:

```
SELECT
    [GroupBy1].[A1] AS [C1]
FROM ( SELECT
        AVG( CAST( [Extent1].[Salary] AS float)) AS [A1]
        FROM [dbo].[Employees] AS [Extent1]
        WHERE 4 = [Extent1].[Employer_ID]
    ) AS [GroupBy1]
```

From the preceding query, we can see that the `Average` function in our LINQ query has actually been translated to the appropriate SQL by Entity Framework and has thus fetched only the relevant data from the database.

Partitioning/paging data using LINQ to Entities

Partitioning data means to take a select set of data from the complete data.

Partitioning is particularly useful in implementing paging in our applications. There are two main methods when it comes to implementing partitioning:

- Skip
- Take

Let's see how we can implement partitioning using these two methods.

Skip

Skip is used when we want to skip the first few records from the result. Let's say we want to write a query where we want to sort all the employees on their ID values and then return the result of all except the first 10 employees:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = db.Employees
        .OrderBy(employee => employee.ID)
        .Skip(10);
}
```

The preceding code will first order the Employee data based on the ID value and then return all the records except the first 10. If we take a look at the generated SQL for the Skip query, it will look like the following:

```
SELECT
    [Extent1].[ID] AS [ID],
    [Extent1].[EmployeeName] AS [EmployeeName],
    [Extent1].[Employer_ID] AS [Employer_ID]
FROM
    (
        SELECT
            [Extent1].[ID] AS [ID],
            [Extent1].[EmployeeName] AS [EmployeeName],
            [Extent1].[Employer_ID] AS [Employer_ID],
            row_number() OVER (ORDER BY [Extent1].[ID] ASC) AS
[ row_number]
        FROM [dbo].[Employees] AS [Extent1]
    ) AS [Extent1]
WHERE [Extent1].[row_number] > 10
ORDER BY [Extent1].[ID] ASC
```

We can see from the preceding SQL that the Skip query has been translated to the SQL that uses row_number to skip the starting records.

Take

Take is used when we want to limit the number of items to be retrieved from the result. Let's say we want to sort all the employees on their ID values and then return the first 10 records only:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = db.Employees
        .OrderBy(employee => employee.ID)
        .Take(10);
}
```

The preceding code will return only the first 10 Employees after sorting them in ascending order of their IDs. Let's see how the generated SQL looks for this LINQ query:

```
SELECT TOP (10)
    [Extent1].[ID] AS [ID],
    [Extent1].[EmployeeName] AS [EmployeeName],
    [Extent1].[Employer_ID] AS [Employer_ID]
FROM [dbo].[Employees] AS [Extent1]
ORDER BY [Extent1].[ID] ASC
```

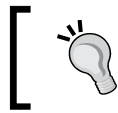
From the query, it can be seen that the Take query has been appropriately translated to the TOP keyword in SQL.

Implementing paging

If we want to implement paging, we have to use Skip and Take in the same query. For example, if we want to retrieve the second page of the data, and each page shows 10 records, we have to skip the first 10 records and then fetch the next 10 records for the user:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = db.Employees
        .OrderBy(employee => employee.ID)
        .Skip(10)
        .Take(10);
}
```

The preceding code will skip the first 10 records and then return the next 10 records to the application. This is very useful while implementing paging in our applications.



As with aggregate functions, partitioning operators only have the method syntax.

Implementing join using LINQ to Entities

If two entities are related to each other, Entity Framework will create a navigation property in the entity to access the related entity. There might also be a possibility that the two entities have a common property and yet the relationship is not defined in the database. If we still want to utilize this implicit relationship, we have to join the related entities.

To see how join works, let's now use the navigation property `Employer` that exists in the `Employee` entity. If we want to retrieve a list of employers with a list of all the employees that belong to that employer, we have to perform a join to accomplish it. Let's see how this can be done using the LINQ query syntax:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employersList = from employer in db.Employers
                        join
                        employee in db.Employees
                        on
                        employer.ID equals employee.Employer_ID into employeeGroup
                        select new
                        {
                            EmployerName = employer.EmployerName,
                            EmployeesList = employeeGroup
                        };
}
```

The preceding code will join the `Employee` and `Employer` entities, and then push the list of all the employees belonging to an employee into `employeeGroup`. We then use projections to return a list of `Employers`, where every `Employer` contains a list of `Employees` that belong to that `Employer`.

If we want to write the same query using the LINQ method syntax, we have to use the `GroupJoin` method as follows:

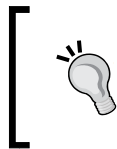
```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employersList = db.Employers.GroupJoin
    (
        db.Employees,
```

```

        employer => (Int32?)(employer.ID),
        employee => employee.Employer_ID,
        (employer, employeeGroup) => new
        {
            EmployerName = employer.EmployerName,
            EmployeesList = employeeGroup
        }
    };
}

```

In the preceding code, we used `GroupJoin` with the `Employers` collection of the `DbContext` class. The first parameter is the `Employees` collection that indicates which entity collection to join to, the second and third parameter represents the respective properties that should be used to join the entities. The last parameter is a **lambda** expression that returns the results using LINQ projections.



The join or `GroupJoin` is the LINQ equivalent of SQL LEFT OUTER JOIN. This will always return all the elements of the entity collection that are to the left irrespective of whether or not the entity collection on the right-hand side of join contains any entities.

Lazy loading and eager loading

When we are working with LINQ to Entities, it is important to understand the concepts of lazy loading and eager loading, as understanding these will greatly help you in writing efficient LINQ queries. Let's try to understand lazy loading and eager loading from the LINQ to Entities perspective.

Lazy loading

Lazy loading is the process where none of the entities involved in the LINQ query will be loaded from the database until the result of the query is enumerated. If the loaded entity contains the navigation properties of the other entities, then these related entities will not be loaded until the navigation property is being accessed from the user code.

In our application, the `Employee` model looks like the following:

```

public partial class Employee
{
    public int ID { get; set; }
    public string EmployeeName { get; set; }
    public Nullable<int> Employer_ID { get; set; }
}

```

```
public virtual Employer Employer { get; set; }

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

If we try to fetch a list of `Employees` using LINQ to Entities, the code will look like the following:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = from employee in db.Employees
                    select employee;
}
```

The important thing to note here is that the data is still not loaded from the database. To load the data, we need to enumerate the employees:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = from employee in db.Employees
                    select employee;

    List<Employee> employeeList = employees.ToList();
}
```

Once this code gets executed, the `employeeList` variable will contain all the employees. Now due to lazy loading only the `Employee` data has been fetched from the database. The `Employer` data will be fetched only when we try to access the `Employer` navigation property of the `Employee` entity:

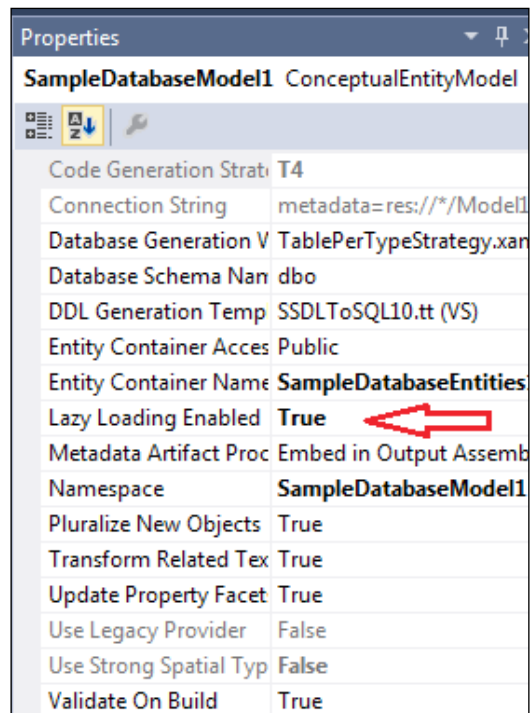
```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = from employee in db.Employees
                    select employee;

    List<Employee> employeeList = employees.ToList();

    // Here the Employer data will be fetched from the database
    Employer employer = employeeList.ElementAt(0).Employer;
}
```

The preceding code will find the first employee from the list and access its `Employer` property. At this call, the `Employer` data will be fetched from the database.

When using Database First and Model First approach, lazy loading can be turned on and off from the **Properties** window of Visual Entity Designer.



The Properties window showing how to change the lazy loading option

When using the Code First approach, lazy loading depends on the nature of the navigation property. If the navigation property is virtual, then lazy loading is enabled. If we want to disable lazy loading, we have to make the navigation property non virtual as follows:

```
public partial class Employee
{
    public int ID { get; set; }
    public string EmployeeName { get; set; }
    public Nullable<int> Employer_ID { get; set; }

    public Employer Employer { get; set; }

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

If we want to disable lazy loading for all the entities, then it can be done from the context class as follows:

```
public partial class SampleDatabaseEntities : DbContext
{
    public SampleDatabaseEntities()
    {
        this.Configuration.LazyLoadingEnabled = false;
    }

    public DbSet<Employee> Employees { get; set; }
    public DbSet<Employer> Employers { get; set; }
}
```

The preceding code will disable lazy loading for all the entities in this context.

Eager loading

Eager loading is the process where we can load the related entity along with the main entity in the query itself. To implement eager loading, we need to use the `Include` keyword. Let's see how we can eagerly load all the `Employer` data along with the `Employee` data from the query itself:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    List<Employee> employeeList = db.Employees
        .Include("Employer").ToList();
}
```

Now, when the list of `Employees` is fetched from the database, the data of the related employer will also be loaded from the database.

Summary

In this chapter, we saw how we can use LINQ to Entities to query the data using Entity Framework. We saw how we can perform the various data retrieval tasks seamlessly using LINQ to Entities with Entity Framework. In the next chapter, we will take a look at how we can use Entity SQL to query the Entity Data Model and retrieve data using Entity Framework.

9

Querying the Object Model – Entity SQL

In the previous chapters, we looked at the various approaches of Entity Framework, how to implement relationships between the domain models. We also looked at how we can query the Entity Data Model using LINQ to Entities. Another way to query the Entity Data Model is by using Entity SQL. In this chapter, we will discuss the details of Entity SQL, and how we can query the object model using it.

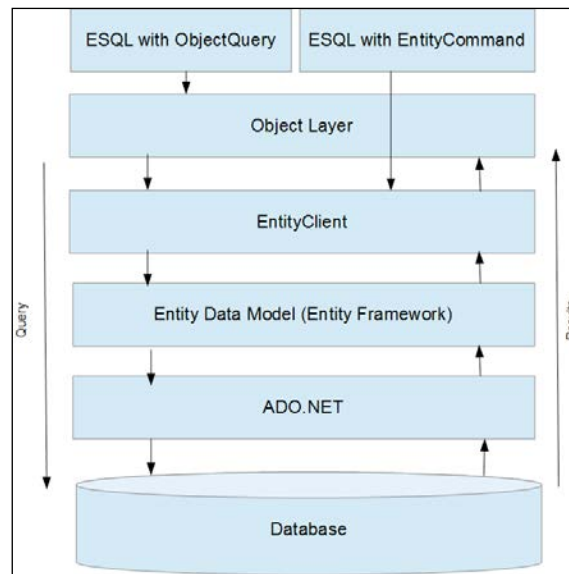
Understanding Entity SQL

Entity SQL or ESQL provides an SQL-like syntax to write queries in terms of our model classes rather than in terms of tables. ESQL is a lesser used technology when it comes to querying the Entity Data Model but it is very useful when it comes to executing dynamic queries against our data model. Another major benefit of using ESQL is that it eliminates the need to know the database schema, and we can target the schema of our Entity Data Model. Many of us are already familiar with SQL and thus working with ESQL is very easy. We just need to know Entity Framework and ESQL related details, and we will find ourselves ready to use ESQL.

There are two ways of executing queries using ESQL:

- Using ESQL with `ObjectQuery`
- Using ESQL with `EntityCommand`

Before we get started with these two approaches, let's take a look at how they fit into the overall architecture.



An architecture diagram showing how ObjectQuery and EntityCommand interact with Entity Framework

In the preceding diagram, we can see that if we write ESQ using `ObjectQuery`, our queries will be executed at the object layer. What this means is that we can use ESQ and still retrieve our entity models as results. Contrary to this, if we use ESQ with `EntityCommand`, we will get the results as read-only row sets. In this chapter, we will take a look at both these techniques, and see how we can use ESQ to query our Entity Data Model.

Before we start discussing how to use `ObjectQuery` and `EntityCommand` to query the Entity Data model, we should first understand the `EntityConnection` object, which is like a pipeline, that will be used to pass our Entity SQL queries to the Entity Data Model.

Understanding EntityConnection

To be able to use Entity SQL, we first need to create an `EntityConnection` object to be able to connect to the Entity Data Model and pass queries to it. The `EntityConnection` object lies in the `System.Data.EntityClient` namespace. To be able to use Entity SQL, we just need to create the `EntityConnection` object, and pass the connection string. If we are using the Database First approach, then the connection string should be the entity connection that contains the information about the `.csdl`, `.msl`, and `.ssdl` files.

Let's say we have the connection string stored in our application configuration file:

```
<add name="SampleDatabaseEntities" connectionString="metadata=res://*/
Models.SampleModel.csdl|res://*/Models.SampleModel.ssdl|res://*/
Models.SampleModel.msl;provider=System.Data.SqlClient;provider
connection string="data source=(localdb)\V11.0;initial catalog=Sa
mpleDatabase;integrated security=True;MultipleActiveResultSets=True;Ap
p=EntityFramework";" providerName="System.Data.EntityClient" /></
connectionStrings>
```

If we want to create the `EntityConnection` object, we just need to pass the name of the connection string in the `EntityConnection` constructor as follows:

```
EntityConnection eConnection = new EntityConnection("name=SampleDatab
aseEntities");
```



If the connection string is not stored in the configuration file, we can use the `EntityConnectionStringBuilder` method to create the connection string dynamically.

The preceding code will create an object of `EntityConnection` using the `SampleDatabaseEntities` connection string.

Developers who have worked with ADO.NET will find the `EntityConnection` object very similar to the `DbConnection` object from the usage perspective. In fact, all the best practices for `DbConnection` should also be followed with `EntityConnection`. So, if we were to create an `EntityConnection` object without leaving any dangling open connections, we should either remember to close it after use or perhaps put it inside a `using` statement such as the following code:

```
using (EntityConnection eConnection = new EntityConnection("name=Sample
DatabaseEntities"))
{
    eConnection.Open();

    if (eConnection.State == System.Data.ConnectionState.Open)
    {
        // Connection is open
    }
}
```

The preceding code will create an `EntityConnection` object, open the `EntityConnection` object, and check for the state of the connection. We can use this connection object inside the `using` block and once the object goes out of scope, the connection will automatically be closed.

Now that we know how to create and use the `EntityConnection` object, we know how to create the pipeline that will be used to pass our Entity SQL queries to the Entity Data Model. Let's now see how we can use `ObjectQuery` and `EntityCommand` to query the Entity Data Model using Entity SQL.

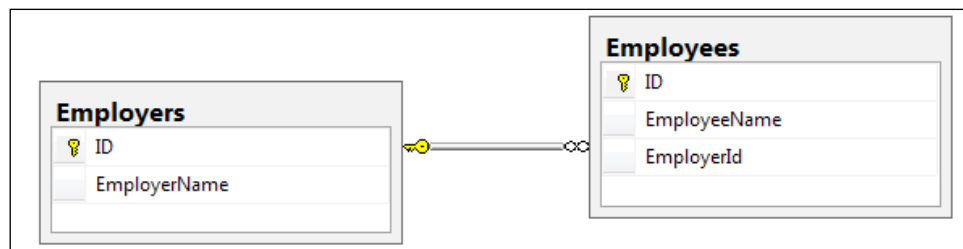
Entity SQL with ObjectQuery

If we use the `ObjectQuery` class to execute the Entity SQL queries, our queries will target the object layer of the Entity Framework. This enables us to write queries and retrieve the results in terms of our model entities. Similar to LINQ to Entities, if we use the `ObjectQuery` class, the `EntityClient` object will facilitate passing the queries to the conceptual model and then return the results in terms of model entities.

Using `ObjectQuery` is comparatively easier than using `EntityCommand` because of two reasons:

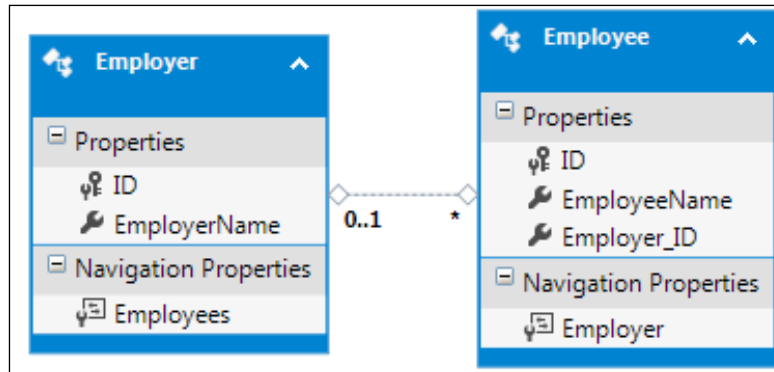
- Firstly, while using `ObjectQuery`, we can use the `ObjectContext` class that will eliminate the need for creating the `EntityConnection` object explicitly since `ObjectContext` has a connection object already.
- Secondly, this approach returns the model entities, and this eliminates the need for parsing the raw results. This will be more evident when we take a look at the `EntityCommand` approach further on in this chapter.

To understand how to use Entity SQL with `ObjectQuery`, let's use a sample database with the following schema:



Database schema for the sample database

If we use the Database First or Model First approach, the Entity Data Model for this schema would look like the following screenshot:



The generated Entity Data Model created for the sample application

If we are using the Code First approach, the POCO entities will look like the following:

```

public partial class Employee
{
    public int ID { get; set; }
    public string EmployeeName { get; set; }
    public Nullable<int> Employer_ID { get; set; }

    public virtual Employer Employer { get; set; }

    public int Salary { get; set; }
}

public partial class Employer
{
    public Employer()
    {
        this.Employees = new HashSet<Employee>();
    }

    public int ID { get; set; }
    public string EmployeeName { get; set; }

    public virtual ICollection<Employee> Employees { get; set; }
}
  
```

The context class will look as follows:

```
public partial class SampleDatabaseEntities : DbContext
{
    public SampleDatabaseEntities()
        : base("name=SampleDatabaseEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public DbSet<Employee> Employees { get; set; }
    public DbSet<Employer> Employers { get; set; }
}
```

The preceding code snippet shows our entity models and context classes. We will now use an example-based approach to understand Entity SQL with `ObjectQuery`.

Querying data using Entity SQL with `ObjectQuery`

Let's see how we can fetch a list of entities using Entity SQL. To put things in perspective, let's see what the LINQ to Entities query would look like if we want to fetch a list of `Employees`:

```
using(SampleDbEntities db = new SampleDbEntities())
{
    var employees = from employee in db.Employees
                    select employee;
}
```

If we try to write the Entity SQL equivalent of this query using `ObjectQuery`, the code will look like the following:

```
using(SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee";

    var adapter = (IObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<Employee> objQuery = new ObjectQuery<Employee>(query,
```

```
objContext);

    List<Employee> employees = objQuery.ToList();
}
```

In the preceding code, we first created an object of the `DbContext` class. We then type cast it to `ObjectContext` and opened `EntityConnection` using the `Connection` property of the `ObjectContext` class. We then created an `ObjectQuery` class that will return the results of type `Employee` and passes the ESQL query and `ObjectContext` instances to it. Then, we enumerated the results using the `ToList()` method to retrieve a list of employees. It is at this enumeration that the actual query will be executed, and the data will be fetched from the database.



Looking at the preceding example, it is quite hard to understand the benefits of ESQL over LINQ to Entities, as both of them return the same results but the code for using ESQL is comparatively complex. It is important to understand that the real benefits of ESQL will only be seen if we have to create queries dynamically.

Executing parameterized Entity SQL with ObjectQuery

In the previous section, we executed a very simple query to find a list of `Employees`. In this section, we will see how we can pass the parameterized queries. The real power of ESQL lies in the ability to create dynamic queries. Let's say we have multiple areas of the application defining a condition on which the data should be filtered. For example, we want to find a list of `Employees` who have a specific range of IDs. The minimum and maximum value for the ID is being supplied from different parts of the application. For such scenarios, we can use the parameterized ESQL queries. We can have `ObjectParameters` created whenever we get a parameter value, and then we will dynamically create an ESQL query and retrieve the result.

Let's say we want to retrieve a list of employees with an ID value greater than 100 and less than 500. The LINQ to Entities query for this will look like the following:

```
using(SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = from employee in db.Employees
                    where employee.ID > 100 && employee.ID < 500
                    select employee;
}
```

If we want to execute the same query using parameterized ESQL, the code will look like the following:

```
// This value is being created in one part of application
ObjectParameter minValue = new ObjectParameter("minValue", 100);
// This value is being created in another part of application
ObjectParameter maxValue = new ObjectParameter("maxValue", 500);

using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee WHERE Employee.ID > @
minValue AND Employee.ID < @maxValue";

    var adapter = (IObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<Employee> objQuery = new ObjectQuery<Employee>(query,
objContext);

    objQuery.Parameters.Add(minValue);
    objQuery.Parameters.Add(maxValue);

    List<Employee> employees = objQuery.ToList();
}
```

In the preceding code, we first created two `ObjectParameter` objects that will hold the minimum and maximum ID values. This can be thought of as these parameters are being created in different parts of the application. We then defined our SQL query where we specified that the query expects parameters in place of `@minValue` and `@maxValue`. Next, we created the `DbContext` class and type cast it to the `ObjectContext` object and created the `ObjectQuery` object. We then added the `ObjectParameters` object to the `ObjectQuery` class.

The query will be executed when we enumerate the results. The value of `@minValue` and `@maxValue` in our query will be replaced by the `ObjectParameter` values, and the data will be fetched from the database.

Navigation properties using Entity SQL with ObjectQuery

If we use ESQL with `ObjectQuery`, our queries will be targeted to the object layer of Entity Framework. The Entity Framework generated entities already contain the navigation properties to access the related entities. This means that while using ESQL with `ObjectQuery`, we can also use the navigation properties in our SQL query to retrieve the desired results. Let's say we want to retrieve a list of `Employees` where every item should contain the `EmployeeName` and `EmployerName` values from the associated `Employer` entity. The result will be stored in the following `ViewModel`:

```
class Result
{
    public string EmployeeName { get; set; }
    public string EmployerName { get; set; }
}
```

If we have to write a LINQ to Entities query for this, this would look like the following:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    var employees = db.Employees.Select
    (
        employee => new Result
        {
            EmployeeName = employee.EmployeeName,
            EmployerName = employee.Employer.EmployerName
        }
    );
}
```

The preceding LINQ to Entities code will use LINQ selection and projection to retrieve the desired results. Let's see how the same can be achieved using ESQL with `ObjectQuery`:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select Employee.EmployeeName as EmployeeName,
Employee.Employer as Employer from SampleDatabaseEntities.Employees as
Employee";

    var adapter = (IObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();
}
```



```
ObjectQuery<DbDataRecord> objQuery = new ObjectQuery<DbDataRecord>(
    query, objContext);

List<Result> result = new List<Result>();

foreach (DbDataRecord item in objQuery)
{
    string employeeName = item[0] as string;
    string employerName = string.Empty;

    Employer employer = item[1] as Employer;

    if (employer != null)
    {
        employerName = employer.EmployerName;
    }

    result.Add(new Result
    {
        EmployeeName = employeeName,
        EmployerName = employerName
    });
}
```

In the preceding code, the first important thing to note is the use of the navigation property in the ESQL query. We used this query to retrieve the `Employee` properties and also to retrieve the associated `Employer` entity with it. The second important thing to note is the type being used with `ObjectQuery`. In this case, we used `DbDataRecord` and then retrieved the data. The desired result is then being created by iterating over `DbDataRecord`.

Aggregate functions with Entity SQL using ObjectQuery

Let's now take a look at how we can use aggregate operators using Entity SQL. We will see how we can perform the following aggregate operations using LINQ to Entities:

- Count
- Sum
- Min
- Max
- Average

Count

Let's start with getting the count of the number of Employees that belong to Employer with ID as 4 using Entity SQL:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select Count(Employee.ID) from
SampleDatabaseEntities.Employees as Employee WHERE Employee.Employer_
ID = 4";

    var adapter = (ObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<DbDataRecord> objQuery = new ObjectQuery<DbDataRecord>
(query, objContext);

    foreach (DbDataRecord item in objQuery)
    {
        int? count = item[0] as int?;
    }
}
```

In the preceding code, we created our Entity SQL to retrieve the count of all the employees that have `Employer_ID` as 4. We then executed the query using `ObjectQuery` and retrieved the results in the form of `DbDataRecord`. The final count is then extracted from `DbDataRecord`.

Sum

Now, let's take a look at how we can use the `Sum` function with Entity SQL. Let's say the `Employee` table also contains a column for `Salary`. The `Employee` entity will also contain a `Salary` property that corresponds to the `Salary` column. Let's see how we can retrieve the sum of all the `Salary` properties of `Employees` that belong to Employer with ID as 4:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select SUM(Employee.ID) from
SampleDatabaseEntities.Employees as Employee WHERE Employee.Employer_
ID = 4";

    var adapter = (ObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();
}
```

```
ObjectQuery<DbDataRecord> objQuery = new ObjectQuery<DbDataRecord>(
    query, objContext);

    foreach (DbDataRecord item in objQuery)
    {
        int? sum = item[0] as int?;
    }
}
```

In the preceding code, we created our Entity SQL to retrieve the sum of the salary of all the employees with `Employer_ID` as 4. We then executed the query using `ObjectQuery` and retrieved the results in the form of `DbDataRecord`. The final sum is then extracted from `DbDataRecord`.

Min

Now let's take a look at how we can use the `Min` function with Entity SQL. If we want to retrieve the minimum salary of the employees that belong to `Employer` with `ID` as 4, run the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select MIN(Employee.ID) from
SampleDatabaseEntities.Employees as Employee WHERE Employee.Employer_
ID = 4";

    var adapter = (IObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<DbDataRecord> objQuery = new ObjectQuery<DbDataRecord>(
        query, objContext);

    foreach (DbDataRecord item in objQuery)
    {
        int? minimum = item[0] as int?;
    }
}
```

In the preceding code, we created our Entity SQL to retrieve the minimum salary from the salaries of all the employees with `Employer_ID` as 4. We then executed the query using `ObjectQuery` and retrieved the results in the form of `DbDataRecord`. The minimum salary value is then extracted from `DbDataRecord`.

Max

Now let's take a look at how we can use the `Max` function with Entity SQL. If we want to retrieve the maximum salary of the employees that belong to `Employer` with `ID` as 4, use the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select MAX(Employee.ID) from
SampleDatabaseEntities.Employees as Employee WHERE Employee.Employer_
ID = 4";

    var adapter = (ObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<DbDataRecord> objQuery = new ObjectQuery<DbDataRecord>
(query, objContext);

    foreach (DbDataRecord item in objQuery)
    {
        int? maximum = item[0] as int?;
    }
}
```

In the preceding code, we created our Entity SQL to retrieve the maximum salary from the salaries of all the employees with `Employer_ID` as 4. We then executed the query using `ObjectQuery` and retrieved the results in the form of `DbDataRecord`. The maximum salary value is then extracted from `DbDataRecord`.

Average

Now let's take a look at how we can use the `Average` function with Entity SQL. If we want to retrieve the average salary of the employees that belong to `Employer` with `ID` as 4, use the following code:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select AVG(Employee.ID) from
SampleDatabaseEntities.Employees as Employee WHERE Employee.Employer_
ID = 4";

    var adapter = (ObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();
```

```
ObjectQuery<DbDataRecord> objQuery = new ObjectQuery<DbDataRecord>(
    query, objContext);

foreach (DbDataRecord item in objQuery)
{
    int? average = item[0] as int?;
}
}
```

In the preceding code, we created our Entity SQL to retrieve the average salary of all the employees with `Employer_ID` as 4. We then executed the query using `ObjectQuery` and retrieved the results in the form of `DbDataRecord`. The average salary value is then extracted from `DbDataRecord`.

Ordering data with Entity SQL using ObjectQuery

Ordering/sorting the records in ascending or descending order of any particular column property is also a very frequently used operation. We can use Entity SQL to pass the `order by` clause, and Entity Framework will take care of retrieving the results for us. The ESQl `order by` syntax is the same as the SQL `order by` syntax. Let's try to retrieve the list of employees in ascending order of the ID:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee ORDER BY Employee.ID";

    var adapter = (ObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<Employee> objQuery = new ObjectQuery<Employee>(query,
objContext);
    List<Employee> employees = objQuery.ToList();
}
```

This preceding code contains the `ORDER BY` clause in the query. When the query gets executed, the `Employee` list returned will be ordered in ascending order of the ID.



If we want to order in descending order, we simply need to pass the DESC keyword, which is the same as in SQL. So the query to retrieve the results in descending order of ID will look like the following:

```
string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee ORDER BY
Employee.ID DESC";
```

Grouping data using Entity SQL with ObjectQuery

Grouping is also very essential when it comes to working with data-centric applications. In this section, we will see how we can use Entity SQL to perform grouping operations. Let's say we want to fetch the list of all the employees grouped by their Employer IDs. Let's see how we can do this using Entity SQL:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select
Employer.ID,
(
    Select
    VALUE Employee
    from SampleDatabaseEntities.Employees as Employee
    where Employee.Employer_ID = Employer.ID
) as Employee
from SampleDatabaseEntities.Employers as Employer
GROUP BY
Employer.ID";

    var adapter = (ObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<DbDataRecord> objQuery = new ObjectQuery<DbDataRecord>
(query, objContext);

    foreach (DbDataRecord item in objQuery)
    {
        int? employerId = item[0] as int?;
        IEnumerable<Employee> employeesList = item[1] as
IEnumerable<Employee>;
    }
}
```

In the preceding code, we fetched a list of `Employees` grouped by `Employer_ID`. This is being achieved by fetching the list of `Employers` and the associated employees. We then grouped the results on `Employer.ID`. The result will contain a collection where every item will contain the `ID` value of `Employer` and a collection of `Employees` associated with it.

Partitioning/paging data using Entity SQL ObjectQuery

Partitioning data means to take a select set of data from the complete data. Partitioning is particularly useful in implementing paging in our applications. There are two main ESQL keywords when it comes to implementing partitioning:

- Skip
- Limit

Let's see how we can implement partitioning using these two methods.

Skip

`Skip` is used when we want to skip the first few records from the result. Let's say we want to write a query where we want to sort all the employees by their `ID` values and then return the result of all except the first 10 `Employees`:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee ORDER BY Employee.ID SKIP
10";

    var adapter = (IObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<Employee> objQuery = new ObjectQuery<Employee>(query,
objContext);
    List<Employee> employees = objQuery.ToList();
}
```

The preceding code will first order the `Employee` data based on the `ID` values and then return all the records except for the first 10.



It should be noted that the skip method for paging only works if the data does not change much, or it is read only. Otherwise, we might get unexpected results.

Limit

Limit is used when we want to limit the number of items to be retrieved from the result. Let's say we want to sort all the employees by their ID values and then return the first 10 records only:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee ORDER BY Employee.ID
LIMIT 10";

    var adapter = (IObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();

    ObjectQuery<Employee> objQuery = new ObjectQuery<Employee>(query,
objContext);
    List<Employee> employees = objQuery.ToList();
}
```

The preceding code will return only the first 10 Employees after sorting them in ascending order of the IDs.

Implementing paging

If we want to implement paging, we have to use Skip and Limit in the same query. For example, if we want to retrieve the second page of the data and each page shows 10 records, we have to skip the first 10 records, and then fetch the next 10 records for the user:

```
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee ORDER BY Employee.ID SKIP
10 LIMIT 10";

    var adapter = (IObjectContextAdapter)db;
    var objContext = adapter.ObjectContext;
    objContext.Connection.Open();
}
```



```
ObjectQuery<Employee> objQuery = new ObjectQuery<Employee>(query,
objContext);
List<Employee> employees = objQuery.ToList();
}
```

The preceding code will skip the first 10 records and then return the next 10 to the application. This is very useful when implementing paging in our applications.

Using Entity SQL with EntityCommand

In the previous section, we saw how to execute Entity SQL using `ObjectQuery`. In this section, we will see how we can use Entity SQL using `EntityCommand`. The `EntityCommand` object is derived from the `DbCommand` object, which would mean that using the `EntityCommand` object with `EntityConnection` will be very similar to using `DbCommand` with `DbConnection`.

One important thing to understand is that if we use `EntityCommand`, the command will be executed directly at the `EntityClient` layer and not the object layer. This means that the results of Entity SQL with `EntityCommand` will not be returned in terms of entity objects, but will just give us read-only access to data using the `EntityDataReader` object. The `EntityDataReader` object will give us a stream of raw data, and it is up to us to convert this data into meaningful domain entities.

Querying data using Entity SQL with EntityCommand

Let's see how we can use Entity SQL with `EntityCommand` to query the data. The following code will retrieve a list of `Employees` using Entity SQL with `EntityCommand`:

```
using (EntityConnection con = new EntityConnection("name=SampleDatabaseEntities"))
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee";

    EntityCommand cmd = con.CreateCommand();
    cmd.CommandText = query;
    cmd.CommandType = System.Data.CommandType.Text;

    List<Employee> employees = new List<Employee>();

    con.Open();
}
```

```

        using (EntityDataReader reader = cmd.
            ExecuteReader(CommandBehavior.SequentialAccess))
        {
            while(reader.Read())
            {
                Employee employee = new Employee();
                employee.ID = (int)reader["ID"];
                employee.EmployeeName = reader["EmployeeName"] as string;
                employee.Employer_ID = reader["Employer_ID"] as int?;

                employees.Add(employee);
            }
        }
    }
}

```

In the preceding code, we first created a `EntityConnection` object and passed the entity connection string to it. Then we defined our ESQl query to fetch a list of `Employees`. We then created a `EntityCommand` object and specified the query type and text. We then opened `EntityConnection` and executed the command to fill the data in an `EntityDataReader` in a sequential manner. We then extracted the employee information from the `EntityDataReader` and created the list of employees.



The similarity between ADO.NET data access code and Entity SQL with `EntityCommand` is sometimes very useful when we are migrating an ADO.NET application to use Entity Framework, and there is a complex query that cannot be implemented using LINQ to Entities.

Parameterized Entity SQL with `EntityCommand`

In the previous section, we executed a very simple query to find a list of `Employees`. Now let's see how we can create parameterized queries and execute them using Entity SQL with `EntityCommand`. Let's say we want to retrieve a list of employees with ID values greater than 100 and less than 500. The LINQ to Entities query for this will look like the following:

```

using (EntityConnection con = new EntityConnection("name=SampleDatabaseEntities"))
{
    string query = @"Select VALUE Employee from
SampleDatabaseEntities.Employees as Employee WHERE Employee.ID > @
minValue AND Employee.ID < @maxValue";
}

```

```
EntityCommand cmd = con.CreateCommand();
cmd.CommandText = query;
cmd.CommandType = System.Data.CommandType.Text;

EntityParameter minValue = new EntityParameter {ParameterName =
"minValue", Value = 100 };
EntityParameter maxValue = new EntityParameter {ParameterName =
"maxValue", Value = 500 };

cmd.Parameters.Add(minValue);
cmd.Parameters.Add(maxValue);

List<Employee> employees = new List<Employee>();

con.Open();
using (EntityDataReader reader = cmd.
ExecuteReader(CommandBehavior.SequentialAccess))
{
    while (reader.Read())
    {
        Employee employee = new Employee();
        employee.ID = (int)reader["ID"];
        employee.EmployeeName = reader["EmployeeName"] as string;
        employee.Employer_ID = reader["Employer_ID"] as int?;

        employees.Add(employee);
    }
}
```

In the preceding code, we first created an `EntityConnection` object and passed the entity connection string to it. Then we defined our ESQL query to fetch a list of `Employees`. The important thing to note here is that the query contains two parameters to filter the results based on `@minValue` and `@maxValue`. We then created an `EntityCommand` object and specified the query type and text. We then created two `EntityParameter` objects that will contain the values of the `@minValue` and `@maxValue` parameter. We then opened `EntityConnection` and executed the command to fill the data in an `EntityDataReader` in a sequential manner. The `EntityDataReader` will now only retrieve the `Employees` values that have an `ID` value between `@minValue` and `@maxValue`.



Now that we have seen how to use Entity SQL with `EntityCommand`, all the query operations that we have seen in the `ObjectQuery` section can be executed using `EntityCommand`. The only difference would be that we need to manually read the data one by one from the `EntityDataReader` and tailor the results to our needs.

Summary

In this chapter, we looked at how we can use Entity SQL to query the Entity Data Model. We saw how we can use Entity SQL with `ObjectQuery` and `EntityCommand` to retrieve the data. In the previous chapter, we looked at how the Entity Data Model can be queried using LINQ to Entities. Now that we know how to use Entity Framework, how to perform domain modeling, and how to query the Entity Data Model, let's take a look at some advanced topics. In the next chapter, we will see how to manage concurrency-related issues using Entity Framework.

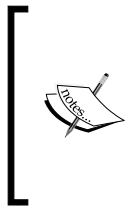
10

Managing Concurrency Using Entity Framework

So far, we have seen how we can use the various approaches of Entity Framework, domain modeling, performing model validations, managing Entity relationships, querying data using LINQ to Entities, and Entity SQL. When we talk about data-driven applications, concurrency and transaction management also plays crucial role. In this chapter, we will take a look at the various concurrency management techniques using Entity Framework. In the next chapter, we will see how to manage transactions with Entity Framework.

Understanding concurrency

Concurrency management deals with allowing multiple entities to be updated simultaneously. This means effectively allowing multiple database operations on the same data simultaneously. Concurrency is a way of managing multiple operations on a database and at the same time allowing **atomicity, consistency, isolation, and durability** (ACID) properties of the database operations.



In this chapter, we will not be discussing the details of ACID properties but we will instead focus on how to manage concurrency using Entity Framework. You are advised to read about database ACID properties if you don't know about it as this will be very helpful in understanding this and the next chapter.

To understand the problem, let's take a look at a few concurrency scenarios. Let's say we have two users accessing the same database. Here are a few scenarios where concurrent access can lead to some potential problems:

- User 1 and user 2 are both trying to update the same entity
- User 1 and user 2 are both trying to delete the same entity
- User 1 is trying to update the same entity where user 2 has deleted it
- User 1 has requested to read an entity, and user 2 has updated it after he has read it.

These are just a few (but not all) scenarios that can potentially lead us to wrong data. Imagine a scenario when hundreds of users are trying to operate on the same data. This problem of concurrency will have a much greater effect on the system.

There are two ways in which we can handle concurrency-related issues:

- **Optimistic concurrency:** In this approach, whenever data is requested from the database, the data is read and kept in the application memory. No explicit locks are put at the database level. The data operations will be applied in the order they are received by the data layer.
- **Pessimistic concurrency:** In this approach, whenever data is requested from the database, the data is read, and an explicit lock will be acquired on this data so that no other user can request this data. This will decrease the chances of having concurrency-related problems. The downside of this approach is that locking is an expensive operation, and it decreases the overall performance of the application. Entity Framework supports optimistic concurrency control as we are working at the entity layer. Thus, all the data will be present in memory once it is read. We can configure Entity Framework to use pessimistic concurrency too but pessimistic concurrency is not directly supported by Entity Framework.

Before we look at how to implement concurrency with Entity Framework, let's try to understand the optimistic and pessimistic concurrency details. This will help us understand how to implement concurrency with Entity Framework.

Understanding optimistic concurrency

In optimistic concurrency, whenever data is requested from the database, the data is read and kept in the application memory. No explicit locks are put on the database. Since in this approach no explicit locks are put, this approach is more scalable and flexible when compared to pessimistic concurrency. The important thing to note when using optimistic concurrency is that if there are any conflicts, these conflicts will need to be handled by the application itself. The most important thing when using optimistic concurrency control is to have a conflict handling strategy in our application, and to let the application user know if their changes are not persisted due to conflicts. Optimistic concurrency essentially means allowing the conflict to happen and then dealing with it in an appropriate manner.

Here are a few examples of strategies for dealing with conflicts.

Ignore the conflict/forcing updates

In this strategy, the application lets all the users change the same set of data, and lets all the changes pass through to the database. This would mean that the database will show values from the very last update. This can lead to potential data loss, as the changes from many of the users will be lost and only changes from the last user will be visible.

Partial updates

In this case, we will also allow all the changes, but we will not update the complete row, only the columns that the specific user has updated. This would mean that if two users are trying to update the same record but different columns, then both the updates will be successful and changes from both users will be visible.

Warn/ask the user

Whenever a user tries to update a record that has been updated after he read it, warn the user that the data has been changed by someone else, and ask for confirmation that he would still like to overwrite the data or first check the already updated data.

Reject the changes

Whenever a user tries to update a record that has been updated after he read it, let the user know that he is not allowed to update the data at this time since it has already been updated by someone else.

Understanding pessimistic concurrency

Contrary to optimistic concurrency, the goal of pessimistic concurrency is to never let any conflicts happen. This is achieved by putting an explicit lock on the records before using them. There are two types of locks that can be acquired on the database records:


- A read-only lock
- An update lock

When a read-only lock is put on a record, the application can only read the record. If the application wants to update the records, it will have to acquire an update lock on the record. If a read-only lock is put on a record, this record can still be used by the requests that want a read-only lock. However, if an update lock is required, the request will have to wait till all the read-only locks are released. Along the same lines, if an update lock is put on the record, no other request can put a lock on this record, and the request will have to wait until the existing update lock is released.

From the preceding description of pessimistic concurrency, it might appear as if using pessimistic concurrency will solve all the concurrency-related issues, as we won't have to deal with these issues in our application. However, this is actually not true. There are a few problems and overheads with using pessimistic concurrency that we need to keep in mind before we decide to choose it for concurrency management. The following are a few of the problems that we face while using pessimistic concurrency:

- Our application will have to manage all the locks that every operation is acquiring
- The application performance will decrease due to the memory requirements of the locking mechanism

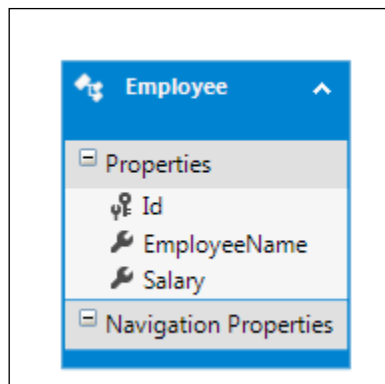
The possibility of deadlock conditions increases as multiple requests are waiting for locks acquired by each other. Due to these reasons, Entity Framework does not support pessimistic concurrency directly. If we want to use pessimistic concurrency, we have to write custom code for database access. Also, when using pessimistic concurrency, LINQ to Entities will not work properly.

 We should try not to use pessimistic concurrency as much as possible. The concurrency-related conflicts can be handled using a `TIMESTAMP` field or the `ROWVERSION` type. We will talk about these in later sections of this chapter.

Implementing optimistic concurrency using Entity Framework

There are a number of ways in which we can implement optimistic concurrency using Entity Framework. In this section, we will take a look at a few of them.

Let's use a simple database to test the concurrency issues. Let's have a single `Employee` table for our testing purpose. The `Employee` table will contain only three fields: `ID`, `EmployeeName`, and `Salary`. The generated Entity Data Model for this database will look like the following:



The Entity Data Model for the sample application

Let's now try to see how we can implement optimistic concurrency using Entity Framework.

Entity Framework's default concurrency

Before we do anything explicitly, let's see how Entity Framework handles concurrency by default. Let's say we have our application designed to update the `Salary` value of an employee. Let's implement a function to retrieve the values for the employee from the database and another to update the `Employee` data in the database:

```
// Lets fetch the employee details for a given ID value
public Employee GetEmployee(int id)
{
    Employee employee = null;

    using (SampleDatabaseEntities db = new SampleDatabaseEntities())
    {
```

```
        employee = db.Employees.Find(id);
    }

    return employee;
}

// Update the Employee data in the database
public void UpdateEmployee(Employee employee)
{
    using (SampleDatabaseEntities db = new SampleDatabaseEntities())
    {
        ((IObjectContextAdapter)db).ObjectContext.
        CreateObjectSet<Employee>().Attach(employee);
        ((IObjectContextAdapter)db).ObjectContext.ObjectStateManager.
        ChangeObjectState(employee, EntityState.Modified);

        db.SaveChanges();
    }
}
```

In the preceding code, the `GetEmployee()` function will retrieve the `Employee` data for a given ID. The `UpdateEmployee()` function will take the updated `Employee` model and commit the changes to the database.

Now to visualize a possible concurrency issue, let's try to simulate a scenario with our code. We will implement a scenario where two users will read the `Employee` data, and then each user will try to update the separate fields of the `Employee` entity:

```
// 1. User1 retrieves the Employee data
Employee employee1 = GetEmployee(1);

// 2. User2 retrieves the Employee data
Employee employee2 = GetEmployee(1);

// 3. User1 tries to update the salary only
employee1.Salary = 100;
UpdateEmployee(employee1);

// 4. User2 tries to update the Name
employee2.EmployeeName = "New EmployeeName";
UpdateEmployee(employee2);
```

In the preceding code, we tried to simulate the concurrency problem. Let's see what this code does in these four sections (section numbers are marked in the code comments):

- 1: Here user 1 reads the `Employee` data
- 2: Here user 2 is reads the same `Employee` data
- 3: Here user 1 updates the `Salary` field of the `Employee` model
- 4: Here user 2 updates the `EmployeeName` field of the same `Employee` model

Now both users have the same copy of data, and both of them are trying to update the same record. Before executing this code, let's take a look at the existing data in the following table.

▶	1	Employee 1	500
	2	Employee 2	200

A database snapshot before running our code

Now for testing, let's put a break point in our code before executing section 4:

```
// User 1 retrieves the Employee data
Employee employee1 = GetEmployee(1);

// User 2 retrieves the Employee data
Employee employee2 = GetEmployee(1);

// User 1 tries to update the salary only
employee1.Salary = 100;
UpdateEmployee(employee1);

// User 2 tries to update the Name
employee2.EmployeeName = "New EmployeeName";
UpdateEmployee(employee2);
```

The screenshot showing where to put the breakpoint for testing

If we take a look at the data at this instance, we can see that the changes from user 1 are being shown in the database.

▶	1	Employee 1	100
	2	Employee 2	200


Database snapshot after section 3 code executes

Now let's continue the execution and see how the data is getting affected.

▶	1	New EmployeeName	500
	2	Employee 2	200

Database snapshot after section 4 code executes

From the preceding screenshot, it can be seen that user 2's request has been successful but user 1's changes were lost. So from the preceding code, it is clear that if we use Entity Framework to update the entire record, the last request will always win, and it will overwrite all the changes made by the previous requests.

[ The default behavior is always the last request wins. It doesn't matter whether multiple requests modify the same data or not.]

Designing applications to handle field level concurrency

In this section, we will see how we can write our application code in such a way that the field level concurrency issues can be handled. The idea here is to design our application in such a way that only the updated fields will get changed in the database. This will ensure that if multiple users are updating the separate fields, all the changes are persisted in the database.

The key to this is to let the application identify all the columns that the user is requesting to update and then selectively updating those fields for this user. To accomplish this, we need two things:

- A method that will give us a clone of the original model with only the user requested properties updated with the new values
- The update method will check for which properties the original requested model values have changed and then update only those values in the database

So, let's first create a simple method that will take the model properties' values, and return the new model that will contain the same values for all the properties except for the properties that the user is trying to update.

```
public Employee GetUpdatedEmployee(int id, string employeeName,
decimal? salaryValue)
{
    Employee employee = new Employee();
    employee.Id = id;
```

```

        employee.EmployeeName = employeeName;
        employee.Salary = salaryValue;

        return employee;
    }

```


So, if a user only wants to update the Salary field, the call to this function will look like the following:

```

Employee newEmployeeData1 = GetUpdatedEmployee(employee2.Id,
employee1.EmployeeName, 100);

```

From the preceding code, `employee1` is the original object that is retrieved when the user requests the model and 100 is the new Salary value.

 This function is very simplistic, and it only shows how to get a clone object with a few properties along with the updated values. In the real world, we will rarely see such code. In such cases, to facilitate this, we can use the mapping module that does domain model to data model mapping.

Next, we need to change our update function. The update function will now implement the following algorithm to update the data:

1. Retrieve the latest model values from the database.
2. Check the original model and updated model to find the list of the changed properties.
3. Only update the changed properties in the model retrieved in step 1.
4. Save the changes.

The code for this algorithm will look like the following:

```

// Update the Employee data in the database field level checks
public void UpdateEmployeeEnhanced(Employee originalEmployeeData,
Employee newEmployeeData)
{
    using (SampleDatabaseEntities db = new SampleDatabaseEntities())
    {
        // Lets get the latest snapshot of data from the database
        Employee employee = db.Employees.Find(originalEmployeeData.
Id);

        // Now one by check for the properties updated by the user
        if (originalEmployeeData.EmployeeName != newEmployeeData.
EmployeeName)
        {

```

```
        // Update the new value in the current database
        employee.EmployeeName = newEmployeeData.EmployeeName;
    }
    if (originalEmployeeData.Salary != newEmployeeData.Salary)
    {
        // Update the value in the current database snapshot
        employee.Salary = newEmployeeData.Salary;
    }

    db.SaveChanges();
}
}
```

In the preceding code, we first retrieved the latest `Employee` data from the database. We then checked the originally requested `Employee` model with the `Employee` model that contains the updated values. We then updated the value of only the properties where the value of the originally requested `Employee` model is different from the updated model. Let's update our application code to use these two functions and examine the results:

```
// 1. User1 retrieves the Employee data
Employee employee1 = GetEmployee(1);

// 2. User2 retrieves the Employee data
Employee employee2 = GetEmployee(1);

// 3. User1 tries to update the salary only by creating a new Employee
Model
Employee newEmployeeData1 = GetUpdatedEmployee(employee2.Id,
employee1.EmployeeName, 100);
UpdateEmployeeEnhanced(employee1, newEmployeeData1);

// 4. User2 tries to update the Name
Employee newEmployeeData2 = GetUpdatedEmployee(employee2.Id, "New
EmployeeName", employee2.Salary);
UpdateEmployeeEnhanced(employee1, newEmployeeData2);
```

Now the application code is divided into four sections (section numbers are marked in the comments):

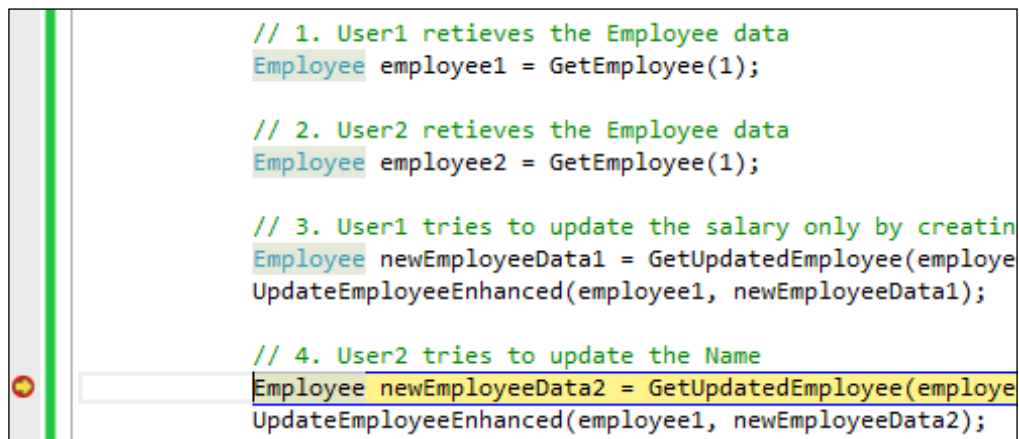
- 1: Here user 1 reads the `Employee` data
- 2: Here user 2 reads the same `Employee` data
- 3: Here user 1 updates the `Salary` field of the `Employee` model using our new approach
- 4: Here user 2 updates the `EmployeeName` value of the same `Employee` model using our new approach

Let's see what happens when we run this code. The database snapshot of the table before we run the code looks like this.

▶	1	Employee 1	500
	2	Employee 2	200

The database snapshot before executing our code

Let's now put a break point before section 4 gets executed:



The screenshot showing where to put the breakpoint for testing

At this point, the request from user 1 has been executed, and the database now contains the updated values:

▶	1	Employee 1	100
	2	Employee 2	200


A database snapshot after the code in section 3 is executed

Let's now resume the application execution to see the effect of the code in section 4:

▶	1	New EmployeeName	100
	2	Employee 2	200

A database snapshot after the code in section 4 is executed

Now we can see that the database contains the updated values from both requests. Thus, our application is now able to handle the concurrent updates effectively if the users update separate fields.

 If multiple users are updating the same field, this approach will still show the updated values from only the last request.

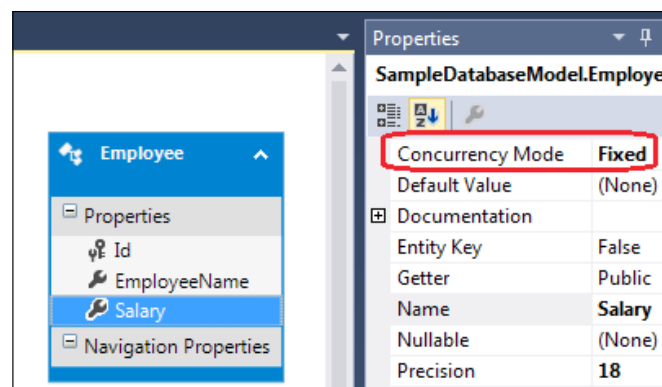
This approach reduces a few of the concurrency-related problems but using this approach means that we have to write a lot of extra code to handle concurrency issues. In the next section, we will take a look at how we can use Entity Framework's provided mechanisms to handle concurrency issues.

Implementing field level concurrency

In the previous sections, we saw how Entity Framework handles concurrency by default (the last update wins), and how we can design our applications to deal with concurrency issues if multiple users are trying to update separate fields. We have still not talked about how to handle concurrency issues if multiple users are trying to update the same field. In this section, we will take a look at how we can have field level concurrency using Entity Framework and deal with concurrency issues involving multiple users updating the same fields.

Entity Framework lets us specify the field level concurrency so that if a user is trying to update a field and the same field has been updated by someone else, we will get a concurrency-related exception. Using this approach, we can handle concurrency-related issues more effectively when multiple users try to update the same field.

To implement this, we need to set the `Concurrency Mode` property of the entity field to `Fixed`. Let's say we want to configure our sample application to deal with field level concurrency on the `Salary` property of the `Employee` field. To accomplish this, we need to go to Visual Entity Designer, and set the `Concurrency Mode` property for the `Salary` property as `Fixed` (as shown in the following screenshot):



The Properties window showing how to change the Concurrency Mode value

Once this is done, we have our Entity Framework configured to check for concurrency issues for the Salary property. To test this, let's use the following code:

```
// 1. User1 retrieves the Employee data
Employee employee1 = GetEmployee(1);

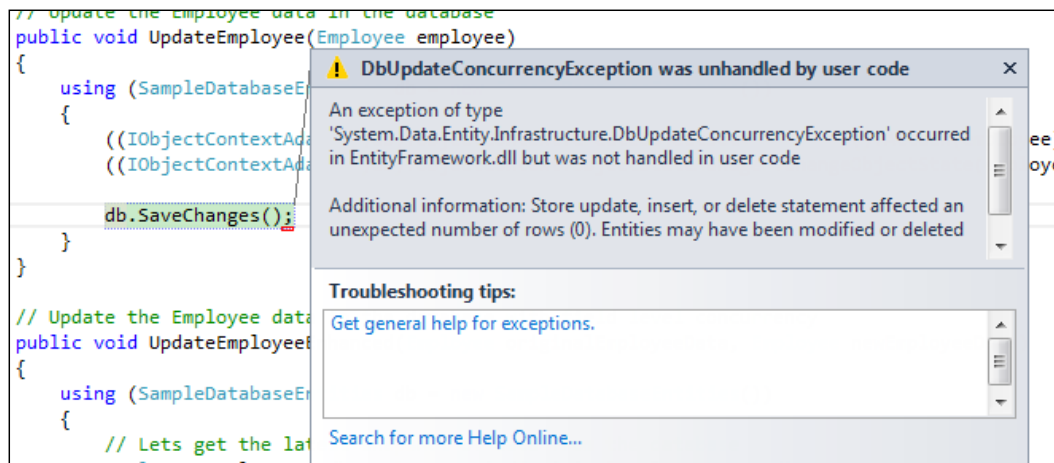
// 2. User2 retrieves the Employee data
Employee employee2 = GetEmployee(1);

// 3. User1 tries to update the salary only
employee1.Salary = 100;
UpdateEmployee(employee1);

// 4. User2 tries to update the Name
employee2.Salary = 200;
UpdateEmployee(employee2);
```

In the preceding code, first, user 1 retrieves the Employee data and then user 2 retrieves the same Employee data. User 1 then updates the Salary field and then in section 4, user 2 updates the Salary field.

With the **Concurrency Mode** value of the Salary property set to Fixed, Entity Framework will throw an exception in section 4 (where user 2 is trying to update the Salary field of Employee when user 1 has changed it already). The exception will look like the following.



Concurrency exception caught in an application

So with field level concurrency set to the **Fixed** mode, our application is able to identify potential concurrency problems on the same field. Now the question is how do we deal with these issues? To deal with these issues, we need to catch the `DbUpdateConcurrencyException` exception thrown by Entity Framework, and let the user know about the potential concurrency problem. The user can then choose to request the latest data and try to perform the operation again. So our calling code will be something similar to the following:

```
try
{
    // 1. User1 retrieves the Employee data
    Employee employee1 = GetEmployee(1);

    // 2. User2 retrieves the Employee data
    Employee employee2 = GetEmployee(1);

    // 3. User1 tries to update the salary only
    employee1.Salary = 100;
    UpdateEmployee(employee1);

    // 4. User2 tries to update the Name
    employee2.Salary = 200;
    UpdateEmployee(employee2);
}
catch (DbUpdateConcurrencyException ex)
{
    // Notify the user with the possible concurrency issue
    // and let him decide how to proceed further
}
```


Using this approach, we can handle field level concurrency issues more effectively. The only downside of this approach is that it makes the application a little slower if we have **Concurrency Mode** set for more than one field of an entity (as the generated SQL query will be much larger). The alternative to this approach is using `RowVersion` for concurrency, which is comparatively more efficient. We will discuss `RowVersion` in the next section.



Using this approach will check for concurrency issues for only those fields marked with **Concurrency Mode** as **Fixed**. Other fields are still vulnerable to concurrency issues.

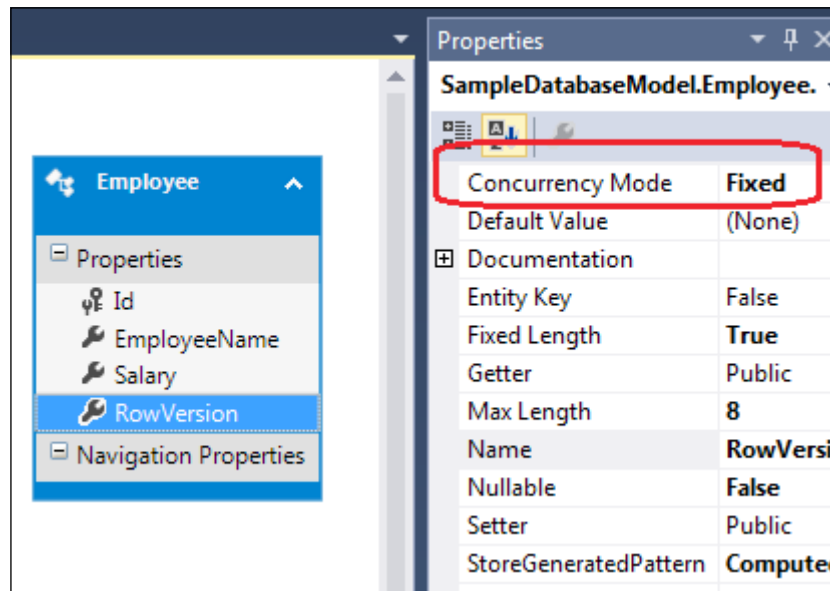
Implementing RowVersion for concurrency

If we use the field-specific concurrency mode for more than one field, then it will slow the application because, the generated SQL will be much larger. The more efficient alternative to this is to use the RowVersion mechanism. The RowVersion mechanism utilizes the database feature where a new value of row version will be created each time the row is updated. How this can be done in the database is shown in the following screenshot:

	Name	Data Type	Allow Nulls	Default
	 Id	int	<input type="checkbox"/>	
	EmployeeName	nvarchar(200)	<input checked="" type="checkbox"/>	
	Salary	decimal(18,0)	<input checked="" type="checkbox"/>	
	RowVersion	rowversion	<input type="checkbox"/>	

Adding a rowversion type column in the table

Now with this change in the database, let's update our Entity Data Model, and set the **Concurrency Mode** value for the **RowVersion** column to **Fixed**.



The Properties window showing how to change the Concurrency Mode

Now what will happen is that Entity Framework will track this column value for concurrency control. Let's try to update the separate columns now and see the effect:

```
// 1. User1 retrieves the Employee data
Employee employee1 = GetEmployee(1);

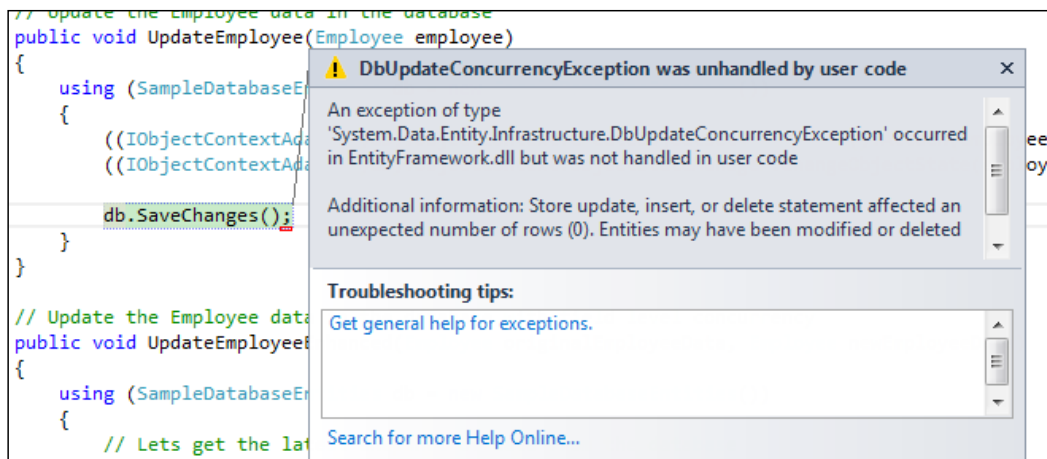
// 2. User2 retrieves the Employee data
Employee employee2 = GetEmployee(1);

// 3. User1 tries to update the salary only
employee1.Salary = 100;
UpdateEmployee(employee1);

// 4. User2 tries to update the Name
employee2.EmployeeName = "New EmployeeName";
UpdateEmployee(employee2);
```

In the preceding code, first, user 1 retrieved the Employee data and then user 2 retrieved the same Employee data. User 1 then updated the Salary field and then in section 4, user 2 updated the EmployeeName field of the same Employee.

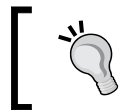
Entity Framework will throw an exception in section 4 (where user 2 is trying to update the EmployeeName field of the employee when user 1 has already changed the salary field of the same employee). The exception will look like the following.



Concurrency exception caught in an application

Any time a user tries to update a record, which has been updated by someone else, he will get the `DbUpdateConcurrencyException` exception.

This approach is essentially the same as using field-specific concurrency but since we are using concurrency control on a field that will get affected if any of the properties of the object changes, we have effectively implemented the concurrency control for the complete row. This mechanism is comparatively more efficient because we are not setting the concurrency mode for multiple fields but rather for only one field and getting the concurrency control for the complete row. This will generate better SQL than setting concurrency for each field separately would have generated.



In older databases, the `rowversion` data type is not present. For such databases, the `TIMESTAMP` data type can be used to achieve the same results.



Entity Framework and pessimistic concurrency

Pessimistic concurrency deals with acquiring database locks on the data that is being used by a user. Whenever a user reads a row from the database, a read-only lock is acquired on this record. Other users can also request a read-only lock during this period. If a user requests an update, an update lock is acquired on the row and until this lock is released, no other users are allowed to read or update the data of this row.

Acquiring and releasing locks is a resource intensive process. Also, the application needs to take care of these locks which would mean added complexity in the application. Here are a few disadvantages of using pessimistic concurrency control:

- Applications tend to consume more memory, as holding locks will need more memory
- The overall performance of the application will be slower because the chances of requests waiting for locks to be released will increase with more users
- If we are using Entity Framework, LINQ to Entities will not work properly even if only a few records are locked
- Chances of deadlock also increase with more users, as multiple users wait for locks to be released that are acquired by each other

Due to these reasons, Entity Framework does not support pessimistic concurrency. It is recommended that you use the `rowversion` and `TIMESTAMP` based approaches to manage concurrency issues. There are some workarounds to implement pessimistic concurrency with Entity Framework that would further add to the complexity of the application.

Summary

In this chapter, we looked at managing concurrency-related issues using Entity Framework. We discussed how to implement optimistic concurrency using Entity Framework. We looked at the basics of pessimistic concurrency and the reasons why Entity Framework does not support and recommend its use. In the next chapter, we will take a look at how to manage transactions using Entity Framework.

11

Managing Transactions Using Entity Framework

In the previous chapter, we talked about managing concurrency-related issues using Entity Framework. Another important topic when dealing with data centric applications is transaction management. ADO.NET provides a very clean and efficient API for transaction management. Since Entity Framework runs on top of ADO.NET, it is able to utilize the transaction management features of ADO.NET.


In this chapter, we will see how we can manage transactions using Entity Framework. We will first try to understand the default behavior of Entity Framework and then we will see how we can take complete control of transaction management using an Entity Framework provided API.

Understanding transactions

When we talk about transactions from the database perspective, it means a set of operations treated as one operation that is indivisible. Either all the operations should succeed or none of them. The concept of a transaction is to have a unit of work that is reliable. All the database operations involved in the transaction should be treated as a unit of work.

From the application perspective, if we want to have multiple database operations treated as a single unit of work, we should wrap them in a transaction. In order to be able to use transactions, an application needs to perform the following steps:

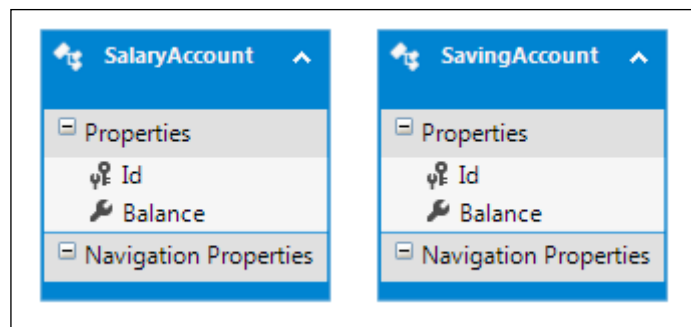
1. Begin transaction.
2. Execute all the queries, and perform all the database operations that need to be treated as single unit of work.
3. If all the operations are successful, commit the transaction.
4. If any of the operations fail, roll back the transaction.

 The preceding steps show the typical steps involved in any action that needs transaction support. These steps are technology/framework agnostic. Typically, a user will have to implement these steps to implement a transaction.

ADO.NET provides a very clean and efficient API to work with transactions. Since Entity Framework used ADO.NET internally, it is also able to utilize it to manage transactions. Let's now take a look at how Entity Framework manages transactions by default.

Setting up the test environment

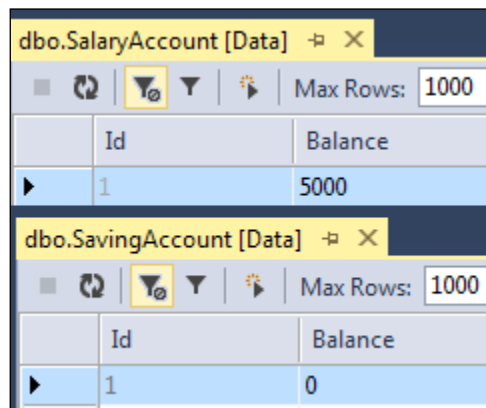
To understand transactions, let's create a sample application that we can use to understand the related concepts. Let's create a simple application to simulate bank account scenarios. Just for the sake of simplicity, let's say a hypothetical bank uses separate tables for its various account types. Let's create two tables in the database: one for `SalaryAccount` and another for `SavingAccount`. The generated Entity Data Model for this sample database will look like the following:



The generated Entity Data Model for the test application

From the application perspective, whenever a user chooses to transfer some money from `SalaryAccount` to `SavingAccount`, this operation should be treated as a single unit of work. It should never happen that the amount gets debited from `SalaryAccount` and never gets credited in the `SavingAccount`. Let's use this scenario to see how we can manage transactions using Entity Framework.

The initial amount of 5,000 will be present in `SalaryAccount`, and we will try to move 1,000 from `SalaryAccount` to `SavingAccount`. So, the initial snapshot of the data in the database looks like the following:

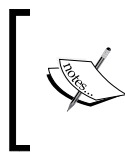


dbo.SalaryAccount [Data]	
Id	Balance
1	5000

dbo.SavingAccount [Data]	
Id	Balance
1	0

A database snapshot showing initial data in both the tables

Now let's try to use Entity Framework to transfer 1,000 from `SalaryAccount` to `SavingAccount` using transactions.



This is a very contrived example created just to illustrate the concepts involved in a simple manner. The real-world database for the same scenario would be far more complex and optimized.

Entity Framework's default transaction handling

The default behavior of Entity Framework is that whenever we perform any operation that involves a CREATE, UPDATE, or DELETE query on the database, it will create a transaction by default. This transaction will be committed whenever the `SaveChanges()` method on the `DbContext` class is called.

So, to implement our scenario, we will have to write code like the following:

```
decimal amountToTransfer = 1000;
int accountId = 1;

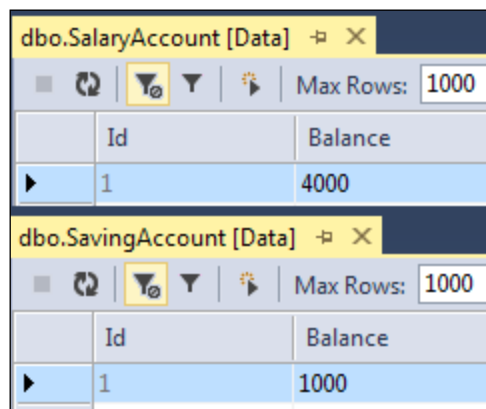
using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    // let us retrieve the accounts involved in transaction
    SalaryAccount salaryAccount = db.SalaryAccounts.Find(accountId);
    SavingAccount savingAccount = db.SavingAccounts.Find(accountId);

    // Lets debit from the SalaryAccount
    salaryAccount.Balance -= amountToTransfer;

    // Now lets credit the SavingAccount
    savingAccount.Balance += amountToTransfer;

    // Call SaveChanges to commit the transaction
    db.SaveChanges();
}
```

In the preceding code, we first retrieved the `SalaryAccount` and `SavingAccount` details associated with the given `accountId`. We then debited the requested amount from `SalaryAccount` and then credited the same amount to `SavingAccount`. A call to the `SaveChanges()` method will commit the transaction in the database. So, if we take a look at the database snapshot after executing the preceding code, it will look like the following:




The image shows two table snapshots from SQL Server Enterprise Manager. The first snapshot is for the `dbo.SalaryAccount` table, showing a single row with `Id` 1 and `Balance` 4000. The second snapshot is for the `dbo.SavingAccount` table, showing a single row with `Id` 1 and `Balance` 1000. Both snapshots have a 'Max Rows' limit of 1000.

Id	Balance
1	4000

Id	Balance
1	1000

A database snapshot showing the data after running our code

So, effectively both the operations are wrapped into one transaction and will be carried out as a single unit of work. If any of the operations fail, there will be no change in the data.

 Since putting READ operations within transactions provides no benefits but decreases the overall performance of the application, Entity Framework never uses transactions on the operations involving SELECT query on the database.

Using TransactionScope to handle transactions

If we have a scenario where we have multiple `DbContext` objects, and we want to associate operations involving multiple `DbContext` objects as a single unit of work, we need to wrap the call to `SaveChanges()` within a `TransactionScope` object. To illustrate this, let's use two separate instances of the `DbContext` class to perform the debit and credit operations in our sample application:

```
decimal amountToTransfer = 1000;
int accountId = 1;

using (TransactionScope ts = new TransactionScope(TransactionScopeOption.Required))
{
    SampleDatabaseEntities db1 = new SampleDatabaseEntities();
    SampleDatabaseEntities db2 = new SampleDatabaseEntities();

    // let us retrieve the accounts involved in transaction
    SalaryAccount salaryAccount = db1.SalaryAccounts.Find(accountId);
    SavingAccount savingAccount = db2.SavingAccounts.Find(accountId);

    // Lets debit from the SalaryAccount
    salaryAccount.Balance -= amountToTransfer;

    // Now lets credit the SavingAccount
    savingAccount.Balance += amountToTransfer;

    db1.SaveChanges();
    db2.SaveChanges();

    ts.Complete();
}
```

In the preceding code, we used two separate instances of `DbContext` to perform the debit and credit operations. So the default behavior of Entity Framework will not work. The respective transactions associated with the context object will not be committed on the call to the respective `SaveChanges()` methods. Instead, since they are within the scope of a `TransactionScope` object, the transaction will only be committed when the `Complete()` method of `TransactionScope` gets called. If any of the operations fail, an exception will occur and `TransactionScope` will end without calling `Complete()`, which will effectively roll back the changes.

Managing transactions using Entity Framework 6

With Entity Framework 6 onwards, Entity Framework provides the `Database.BeginTransaction()` method on the `DbContext` object. This is particularly useful when we want to execute raw SQL commands using our context class and associate them with transactions. This lets us override the default transaction that gets associated with `DbContext`.

Let's see how we can use this new method to manage transactions. Let's use raw SQL to debit `SalaryAccount` and our model class to credit `SavingAccount`:

```
decimal amountToTransfer = 1000;
int accountId = 1;

using (SampleDatabaseEntities db = new SampleDatabaseEntities())
{
    using (DbContextTransaction tx = db.Database.BeginTransaction())
    {
        try
        {
            db.Database.ExecuteSqlCommand("Update SalaryAccount set
            Balance = Balance - @amountToDebit where id = @accountId",
                new SqlParameter("@amountToDebit", amountToTransfer),
                new SqlParameter("@accountId", accountId));

            SavingAccount savingAccount = db.SavingAccounts.
            Find(accountId);
            savingAccount.Balance += amountToTransfer;
            db.SaveChanges();

            tx.Commit();
        }
        catch
    }
}
```

```

        {
            tx.Rollback();
        }
    }
}

```

In the preceding code, we first created an instance of our `DbContext` class. We then used this instance to start a transaction by calling the `Database.BeginTransaction()` method. This method gives us a handle to the `DbContextTransaction` object, which we can later use to either commit or roll back the transaction. We then debited `SalaryAccount` using raw SQL and credited the `SavingAccount` using the model class. Calling `SaveChanges()` will only affect the second operation but not commit the transaction. If both the operations are successful, we can call `Commit` on the `DbContextTransaction` object. If not, we can handle the exception and call `Rollback`.



This approach is only available in Entity Framework Version 6. If we use the previous version of Entity Framework, we have to rely on `TransactionScope` to manage the transactions.

Using an existing transaction

There are times when we want to use an existing transaction with our Entity Framework's `DbContext` class. The reason for this could be that some of the operations are being done in separate parts of the application. Another possible reason could be that we are using Entity Framework with a legacy application, and this legacy application uses a class library that gives us the handle for the transaction and/or database connection.

For such scenarios, Entity Framework also lets us use an existing connection that is associated with the transaction with the `DbContext` class. To illustrate this, let's first write a simple function that will perform the debit operation using plain ADO.NET:

```

public bool DebitSalaryAccount(SqlConnection connection,
    SqlTransaction tx, int accountId, decimal amountToDebit)
{
    SqlCommand cmd = connection.CreateCommand();
    cmd.Transaction = tx;
    int result = 0;

    cmd.CommandType = CommandType.Text;
    cmd.CommandText = "Update SalaryAccount set Balance = Balance - @
amountToDebit where id = @accountId";
    cmd.Parameters.Add(new SqlParameter("@amountToDebit",
amountToDebit));
}

```

```
cmd.Parameters.Add(new SqlParameter("@accountId", accountId));

try
{
    result = cmd.ExecuteNonQuery();
}
catch
{
    throw;
}

return (result == 1);
}
```

What this function does is take the connection and transaction objects from the caller and then execute the update query to debit the account.

Now, let's assume that this function is being provided to us in a library that we cannot change. In this scenario, we will not be able to use the `Database.BeginTransaction` method since we want the `SqlConnection` and `SqlTransaction` objects to pass to this function and still be able to put it into our transaction. To facilitate this, we need to first create an `SqlConnection` and then begin an `SqlTransaction`. We can then use these objects to first call this function with our `DbContext` class, as shown in the following code. Insert the following code: 1003OS_11_5_code.txt:

```
decimal amountToTransfer = 1000;
int accountId = 1;

using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    using (SqlTransaction tx = connection.BeginTransaction())
    {
        try
        {
            bool result = DebitSalaryAccount(connection, tx,
            accountId, amountToTransfer);

            if (result == false)
            {
                throw new Exception("Unable to debit from account");
            }
            using (SampleDatabaseEntities db = new SampleDatabaseEntities(connection, contextOwnsConnection: false))
            {
```

```
        db.Database.UseTransaction(tx);

        SavingAccount savingAccount = db.SavingAccounts.
Find(accountId);
        savingAccount.Balance += amountToTransfer;
        db.SaveChanges();
    }

    tx.Commit();
}
catch
{
    tx.Rollback();
}
}
```

In the preceding code, we first created an `SqlConnection` and then associated an `SqlTransaction` with this connection. Once the transaction is started, we used this connection and transaction object to call our legacy method. We then checked whether the call to our legacy method succeeded or not. If it failed, we threw an exception, which rolled back the transaction. If it was successful, we just used the `DbContext` class and our `SavingAccount` entity to credit the amount and then committed the transaction.

Choosing the appropriate transaction management

Now that we know the various ways to handle transactions using Entity Framework, let's try to see which technique should be used in which scenarios:

- If we have only one `DbContext` class, then we should try to use the default transaction management of Entity Framework. We should always try to implement our application in such a way that all the operations constituting a single unit of work get executed within the scope of the same `DbContext` object, and the `SaveChanges()` method will take care of committing the transaction.
- If we use multiple `DbContext` objects, then perhaps putting the calls within the scope of a `TransactionScope` object is the best way to manage the transactions.

- If we want to execute raw SQL queries and associate these operations with the transactions, then we should use the Entity Framework provided `Database.BeginTransaction()` method that is available on the `DbContext` object. This approach, however, will only work with Entity Framework 6. It will not work with previous versions.
- If we want to use Entity Framework with any legacy application that demands `SqlTransaction`, then we should use the `Database.UseTransaction()` method, which is also available in Entity Framework 6.

Summary

In this chapter, we looked at how to manage transactions using Entity Framework. We looked at the default behavior of Entity Framework when it comes to transaction management. We also looked at how we can take control of transaction management using Entity Framework. At this juncture, we are equipped with most of the information needed to implement any application using Entity Framework. In the next chapter, we will try to create a sample application that will use Entity Framework for data access. This sample application will give you an idea of how to use Entity Framework in real-world applications.

12

Implementing a Small Blogging Platform Using Entity Framework

Now that we have seen how to use Entity Framework in our applications, let's try to create the data access layer for a complete application using Entity Framework to see all of the concepts in action. We will try to build a simple blogging platform and will use Entity Framework for data access. We will use the Entity Framework Code First approach for this application. First, we will take a look at the requirements for our application. We will then try to visualize the data access needs of the application. We will then create POCO entities for our models, and see how we can perform various data access operations using the Entity Framework's Code First approach.

Understanding the application requirements

Let's get started with our sample application by first understanding the requirements for our application. We will develop a simple application that can be used as a blogging framework. It will be like WordPress Engine but on a very tiny scale, with support for only a few features. Let's first list all of the features we want in our blogging framework:

- Authors can create blog categories
- Authors can manage (add and update) categories
- Authors can create new blogs
- Authors can manage (update/delete) their blogs

- The home page will show a list of all blogs with a blog title, date of posting, and the first 200 characters from the blog with a **Read More** link
- On clicking on the title of the blog on the home page, a page showing the complete blog will be presented to the user
- The users should be able to post comments on the blog
- The users should be able to delete their comments

We will be developing a simple data access layer in the form of a class library that will facilitate the implementation of these features. Since we are creating a class library for data access, it can be used by a client application.



From a blogging framework's perspective, these are not a complete set of features, but these features are enough to demonstrate the Entity Framework-related concepts. I will be creating a simple ASP.NET MVC application on top of this class library to show the application in action. Discussing ASP.NET MVC application code is out of scope from this book's perspective, but I will be showing screenshots of the running application to show the working of our data access layer.

Visualizing our database design

Before we start creating our POCO entities, let's try to visualize the database that we will need to implement the previously mentioned features. First, let's have tables that will cater to user registration, authentication and authorization mechanism. We can create the following three tables to accomplish this task:

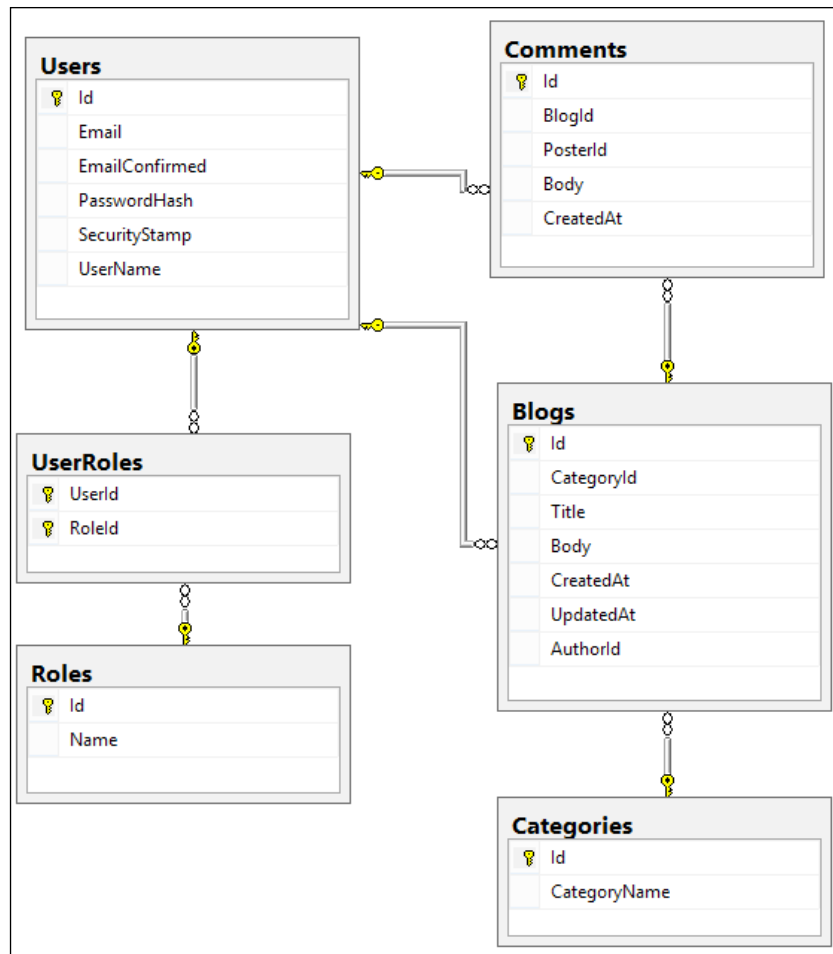
- **Users:** This table will keep all the users' information that is needed for the users to log in and manage their account.
- **Roles:** This table will keep track of the roles within our application. In our application, we will have two roles: `Authors` and `Users`. `Authors` will be able to post blogs and manage them. `Users` will only be able to post comments on blogs.
- **UserRoles:** This is a simple joining table that will be created to facilitate many-to-many relationships between users and roles.

Now that we have the basic authentication and authorization tables in place, let's see what tables are needed to implement our application-specific features. The following tables will be needed to implement our application-specific features:

- **Categories:** This table will store all the blog categories created by the authors
- **Blogs:** This table will store all the blogs created by the authors

- **Comments:** This table will store the comments posted on the blogs by the users

Let's try to visualize this database by drawing the schema diagram for the proposed database:



Visualization of the database schema for the sample application

Creating the Entity Data Model

Let's now start by creating our POCO entities one by one. We will first create all the entity classes with all the properties that we want in their respective entities. Once the entity classes are ready, we will take a look at how to implement relationships between the entities, and start adding the navigation properties in the entity classes.

Creating the entity classes

Let's take a look at all the entities needed one by one.

The User entity

First, let's create the entity class to hold the `User` details. The `User` entity is responsible for holding the data that is needed to authenticate the user.

The following code shows what the `User` entity would look like:

```
public partial class User
{
    public int Id { get; set; }
    public string Email { get; set; }
    public bool EmailConfirmed { get; set; }
    public string PasswordHash { get; set; }
    public string SecurityStamp { get; set; }
    public string UserName { get; set; }
}
```

The preceding code shows the `User` entity. The `Id` property is an auto-increment identity property of the class. The other properties are the properties that are needed to authenticate the `User` entity.



The `User` model has been created with an assumption that we will be using hashing and salting techniques to store the user passwords. `PasswordHash` will contain the hash for the user provided password and `SecurityStamp` will contain the salt value. The `User` model can be changed accordingly if any other mechanism for password storage is to be used.

The Role entity

Let's now take a look at the `Role` entity. The primary purpose of the `Role` class is authorization, that is, to identify whether the logged in user is an author or a user.

The following code shows what the `Role` entity will look like:

```
public partial class Role
{
    public int Id { get; set; }

    [Required]
    [StringLength(256)]
    public string Name { get; set; }
}
```

The preceding code shows the `Role` entity. The `Id` property is an auto-increment identity property. The `Name` property will store the name of the role. The `Name` property has been marked as `Required` to make it a mandatory value that should be provided by the user. We also specify the maximum length that should be allowed for the `Name` property to help the Code First module generate the database columns appropriately.

The Category entity

Let's now create the `Category` entity. The following code shows what the `Category` entity will look like:

```
public partial class Category
{
    public int Id { get; set; }

    [Required]
    [StringLength(200)]
    public string CategoryName { get; set; }
}
```

The preceding code shows the `Category` entity. The `Id` property is an auto-increment identity property, and the `CategoryName` property will hold the name of the category.

The Blog entity

Now that we have the `Category` entity ready, let's create the `Blog` entity that will actually hold the blog data for the user. The following code shows the `Blog` entity for our application:

```
public partial class Blog
{
    public int Id { get; set; }

    [Required]
    [StringLength(500)]
    public string Title { get; set; }

    [Required]
    public string Body { get; set; }

    public DateTime CreatedAt { get; set; }
    public DateTime UpdatedAt { get; set; }
}
```

The preceding code shows the `Blog` entity. The `Id` property is an auto-increment identity property. The `Title` property will store the title of the blog. `Body` will store the actual contents of the blog. `CreatedAt` will store the date when this blog was created, and `UpdatedAt` will store the date when this blog was updated (if it's been updated).

The Comment entity

The users should also be able to post comments on the blogs. So let's create an entity to store the comments. The following code shows what the `Comment` entity will look like:

```
public partial class Comment
{
    public int Id { get; set; }
    public string Body { get; set; }
    public DateTime? CreatedAt { get; set; }
}
```

The preceding code shows the `Comment` entity. The `Id` property is an auto-increment identity property. The `Body` property will contain the actual content of the comment, and the `CreatedAt` property will indicate the date when this comment was posted.

Creating relationships and navigation properties

So far, we created only standalone entities. Our entity classes do not reflect the relationships between the entities. Let's now look at how we can implement relationships between these entities. Before we start looking at the code, let's take a look at all the relationships needed:

- **User-Role:** The `User` and `Role` entities have a many-to-many relationship as any user can be in multiple roles, and every role can have many users
- **Category-Blog:** The `Category` and `Blog` entities have a one-to-many relationship as every blog will belong to one category, and each category can have multiple blogs
- **User-Blog:** The `User` and `Blog` entities have a one-to-many relationship as every blog belongs to a user, and each user can have multiple blogs
- **Blog-Comment:** The `Blog` and `Comment` entities have one-to-many relationship as a comment will always belong to a blog, and each blog can have multiple comments

- **User-Comment:** The `User` and `Comment` entities have a one-to-many relationship as every comment will belong to a user, and each user can have multiple comments

Let's now take a look at how these relationships can be implemented in the entity classes. Let's see how our entity classes will change to reflect these relationships.

The User entity

Let's start with the `User` entity. Let's see what we need to add to the `User` entity to implement relationships:

- Since every user can have multiple roles, we need a collection of roles as the navigation property
- Since every user can have multiple blogs, we need a collection of blogs as the navigation property
- Since every user can have multiple comments, we need a collection of comments as the navigation property

So, our final `User` entity will look like the following:

```
public partial class User
{
    public User()
    {
        Blogs = new HashSet<Blog>();
        Comments = new HashSet<Comment>();
        Roles = new HashSet<Role>();
    }

    public int Id { get; set; }
    public string Email { get; set; }
    public bool EmailConfirmed { get; set; }
    public string PasswordHash { get; set; }
    public string SecurityStamp { get; set; }
    public string UserName { get; set; }

    public virtual ICollection<Blog> Blogs { get; set; }
    public virtual ICollection<Comment> Comments { get; set; }
    public virtual ICollection<Role> Roles { get; set; }
}
```

In the preceding code, we added three properties; one each for the `Blogs`, `Comments`, and `Roles` relationships.

The Role entity

Let's now see how we can implement relationships with the `Role` entity.

Since every role can contain multiple users, let's add a collection of users as a navigation property:

```
public partial class Role
{
    public Role()
    {
        Users = new HashSet<User>();
    }

    public int Id { get; set; }

    [Required]
    [StringLength(256)]
    public string Name { get; set; }

    public virtual ICollection<User> Users { get; set; }
}
```

In the preceding code, we added a navigation property `Users` to implement the many-to-many relationship.

The Category entity

Let's now see how we can implement relationships with the `Category` entity.

Since every `Category` entity can contain multiple blogs, let's add a collection of blogs as a navigation property in the `Category` entity:

```
public partial class Category
{
    public Category()
    {
        Blogs = new HashSet<Blog>();
    }

    public int Id { get; set; }

    [Required]
    [StringLength(200)]
    public string CategoryName { get; set; }
}
```

```
        public virtual ICollection<Blog> Blogs { get; set; }  
    }
```

In the preceding code, we added a collection of Blogs as a navigation property in order to implement the relationship with the Blog entity.

The Blog entity

Let's now see how we can implement relationships in the Blog entity:

- Since every Blog entity belongs to a Category entity, we need to add a property to implement the foreign key relationship with the Category entity and a navigation property to the Category entity
- Since every Blog entity belongs to a User entity, we need to add a property to implement the foreign key relationship with the User entity and a navigation property to the User entity
- Since every Blog entity can contain multiple comments, let's add a collection of comments as a navigation property

The code for adding these properties is as follows:

```
public partial class Blog  
{  
    public Blog()  
    {  
        Comments = new HashSet<Comment>();  
    }  
  
    public int Id { get; set; }  
  
    [Required]  
    [StringLength(500)]  
    public string Title { get; set; }  
  
    [Required]  
    public string Body { get; set; }  
    public DateTime CreatedAt { get; set; }  
    public DateTime UpdatedAt { get; set; }  
  
    public int CategoryId { get; set; }  
    public int AuthorId { get; set; }  
}
```

```
        public virtual Category Category { get; set; }
        public virtual User User { get; set; }
        public virtual ICollection<Comment> Comments { get; set; }
    }
}
```

In the preceding code, the `CategoryId` property will be used to create the foreign key relationship with `Category`, and `AuthorId` will be used to create the foreign key relationship with `User`. We also added the navigation properties for `Category`, `User`, and `Comments` to implement their respective relationships with the entities.

The Comment entity

Let's now see how we can implement relationships in the `Comment` entity:

- Since every `Comment` entity belongs to a `Blog` entity, we need to add a property to implement the foreign key relationship with the `Blog` entity and a navigation property to the `Blog` entity
- Since every `Comment` entity belongs to a `User` entity, we need to add a property to implement the foreign key relationship with the `User` entity and a navigation property to the `User` entity

The code to implement the relationships is as follows:

```
public partial class Comment
{
    public int Id { get; set; }
    public string Body { get; set; }
    public DateTime? CreatedAt { get; set; }

    public int? BlogId { get; set; }
    public int? PosterId { get; set; }

    public virtual Blog Blog { get; set; }
    public virtual User User { get; set; }
}
```

In the preceding code, we added the `BlogId` property that will be used to create the foreign key relationship with `Blog`, and `PosterId` will be used to create the foreign key relationship with `User`. We also added the navigation properties for `Blog` and `User` to implement their respective relationships with the entities.

Implementing the DbContext class

Now that we have all the entities ready, let's create our DbContext class. The DbContext class contains the DbSet object for all the entities we created so far. We also need to override the OnModelCreating method to indicate the following relationships between the entities:

- One-to-many relationship between Category and Blog
- One-to-many relationship between Blog and Comment
- One-to-many relationship between User and Blog
- One-to-many relationship between User and Comment
- Many-to-many relationship between User and Role

Let's see what our DbContext class will look like with the preceding implementations:

```
public partial class BlogAppContext : DbContext
{
    public BlogAppContext()
        : base("name=BlogAppConnectionString")
    {
    }

    public virtual DbSet<Blog> Blogs { get; set; }
    public virtual DbSet<Category> Categories { get; set; }
    public virtual DbSet<Comment> Comments { get; set; }
    public virtual DbSet<Role> Roles { get; set; }
    public virtual DbSet<User> Users { get; set; }

    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        // One to many relationship between Category and Blog.
        modelBuilder.Entity<Category>()
            .HasMany(e => e.Blogs)
            .WithRequired(e => e.Category)
            .HasForeignKey(e => e.CategoryId)
            .WillCascadeOnDelete(false);
    }
}
```

```
// One to many relationship between Blog and Comment
modelBuilder.Entity<Blog>()
    .HasMany(e => e.Comments)
    .WithRequired(e => e.Blog)
    .HasForeignKey(e => e.BlogId)
    .WillCascadeOnDelete(false);

// One to many relationship between User and Blog.
modelBuilder.Entity<User>()
    .HasMany(e => e.Blogs)
    .WithRequired(e => e.User)
    .HasForeignKey(e => e.AuthorId)
    .WillCascadeOnDelete(false);

// One to many relationship between User and Comment.
modelBuilder.Entity<User>()
    .HasMany(e => e.Comments)
    .WithOptional(e => e.User)
    .HasForeignKey(e => e.PosterId);

// Many to many relationship between User and Role.
modelBuilder.Entity<User>()
    .HasMany(e => e.Roles)
    .WithMany(e => e.Users)
    .Map(m => m.ToTable("UserRoles")
        .MapLeftKey("UserId")
        .MapRightKey("RoleId"));
}
```

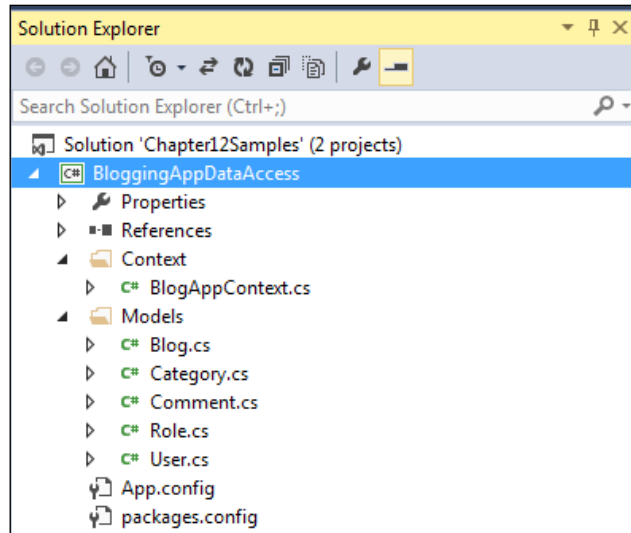
In the preceding code, we created our `DbContext` class, `BlogAppContext`. The constructor takes the name of the connection string that should be used. We are also overriding the `OnModelCreating` method to implement the various relationships between the entities.



The comments in the code indicate which code block is for which relationship. Also, note the last code block of the `OnModelCreating` method. This indicates that Entity Framework will generate a joining table to implement the many-to-many relationship with between User and Role.

Performing data access

Let's now start implementing the actual data access layer for our application. The first thing we need to do is to create a class library and copy all the model classes and context classes into the class library project. The following screenshot shows how this can be done:



Solution Explorer showing the class library that contains the POCO classes

To implement an extensible data access layer, we will use the Repository and Unit of Work patterns.

Understanding the Repository pattern

The Repository pattern is particularly useful when we have separate data models and domain models. Repository can act as a mediator between the data model and the domain model. Internally, it can talk to the database in terms of data models and will return the domain model to the application layers above it.

Since we are using data models as domain models in our application, we will be returning the same model. In case we want to use separate data models and domain models, we just need to map the data model values to the domain model or perhaps use any mapping library to perform the mapping.

So, let's start by defining our repository interface, `IRepository`:

```
public interface IRepository<T> where T : class
{
    IEnumerable<T> GetAll(Expression<Func<T, bool>> predicate = null);
    T Get(Expression<Func<T, bool>> predicate);
    void Add(T entity);
    void Update(T entity);
    void Delete(T entity);
    long Count();
}
```

In the preceding code, we created an interface that defines all the basic functions that are needed on an entity. The `GetAll` function returns multiple values, that is, a collection of entities that match the predicate passed to the function. The `Get` function returns the single value that matches the predicate. The `Add` function can be used to create a new entity, `Update` is to update an existing entity, `delete` is to delete the entity, and `Count` will return the total number of entities present in the database.

Now that we have the `IRepository` interface created, let's see how we can create concrete repository classes for our entities. Let's start by creating the `CategoryRepository` class:

```
class CategoryRepository : IRepository<Category>
{
    BlogAppContext m_Context = null;

    public CategoryRepository(BlogAppContext context)
    {
        m_Context = context;
    }

    public IEnumerable<Category> GetAll(System.Linq.Expressions.
Expression<Func<Category, bool>> predicate = null)
    {
        return m_Context.Categories.Where(predicate);
    }

    public Category Get(System.Linq.Expressions.
Expression<Func<Category, bool>> predicate)
    {
        return m_Context.Categories.SingleOrDefault(predicate);
    }

    public void Add(Category entity)
```

```

    {
        m_Context.Categories.Add(entity);
    }

    public void Update(Category entity)
    {
        m_Context.Categories.Attach(entity);
        ((IOBJECTContextAdapter)m_Context).ObjectContext.
        ObjectStateManager.ChangeObjectState(entity, EntityState.Modified);
    }

    public void Delete(Category entity)
    {
        m_Context.Categories.Remove(entity);
    }

    public int Count()
    {
        return m_Context.Categories.Count();
    }
}

```

In the preceding code, we used the `BlogAppContext` class to perform all the CRUD operations on the `Category` entity and implemented the `IRepository` interface. Along the same lines, we can create the concrete repository classes `BlogRepository` and `CommentRepository`. However, since all the code shown in the preceding class will remain more or less the same with the exception that each concrete class will use its respective property of the context class, for example, `CategoryRepository` will use `Categories`, `BlogRepository` will use `Blogs`. So, instead of having a separate repository class for each entity, why don't we create a generic repository class that will work for all the entity classes.

The generic repository class will look like the following:

```

class Repository<T> : IRepository<T> where T : class
{
    private BlogAppContext m_Context = null;

    DbSet<T> m_DbSet;

    public Repository(BlogAppContext context)
    {
        m_Context = context;
        m_DbSet = m_Context.Set<T>();
    }
}

```



```
        public IEnumerable<T> GetAll(Expression<Func<T, bool>> predicate =
null)
        {
            if (predicate != null)
            {
                return m_DbSet.Where(predicate);
            }

            return m_DbSet.AsEnumerable();
        }

        public T Get(Expression<Func<T, bool>> predicate)
        {
            return m_DbSet.FirstOrDefault(predicate);
        }

        public void Add(T entity)
        {
            m_DbSet.Add(entity);
        }

        public void Update(T entity)
        {
            m_DbSet.Attach(entity);
            ((IObjectContextAdapter)m_Context).ObjectContext.
ObjectStateManager.ChangeObjectState(entity, EntityState.Modified);
        }

        public void Delete(T entity)
        {
            m_DbSet.Remove(entity);
        }

        public long Count()
        {
            return m_DbSet.Count();
        }
    }
```

The preceding code shows a generic repository class. This is useful because now we have a single class that can be used to perform CRUD operations on all the entities.



Implementing a generic repository is a good idea since an application can contain hundreds of entities and writing these many repositories can be a very tedious process.

Understanding Unit of Work

In the previous chapters, we saw that the `DbContext` class by default supports a transaction. When we instantiate a new `DbContext` object, a new transaction will be created, and it will be committed when the `SaveChanges` method is called. Now the question is how can we perform operations across multiple code modules, and still use the same `DbContext` object to keep all these operations in a single transaction. The answer to this question is to use Unit of Work.

Unit of Work is a class that keeps track of all the operations in a transaction, and then performs all the operations as a single atomic unit. If we take a look at our repository classes, we can see that the `DbContext` object is being passed to them from the outside. Also, the repository classes are missing the call to the `SaveChanges` method. The reason for this is that we will now create a Unit of Work class that will pass the `DbContext` object to their respective repositories. Whenever we want to save the changes, we will call the `SaveChanges` method on our `UnitOfWork`, and it will call the `SaveChanges` method on our `DbContext` class. This will enable all the operations involving multiple repositories to be part of a single transaction.

Let's see what our `UnitOfWork` class will look like:

```
public class UnitOfWork : IDisposable
{
    private BlogAppContext m_Context = null;
    private Repository<Category> categoryRepository = null;
    private Repository<Blog> blogRepository = null;
    private Repository<Comment> commentRepository = null;
    private Repository<User> userRepository = null;
    private Repository<Role> roleRepository = null;

    public UnitOfWork()
    {
        m_Context = new BlogAppContext();
    }
}
```

```
public void SaveChanges()
{
    m_Context.SaveChanges();
}

public Repository<Category> CategoryRepository
{
    get
    {
        if (categoryRepository == null)
        {
            categoryRepository = new Repository<Category>(m_
Context);
        }
        return categoryRepository;
    }
}

public Repository<Blog> BlogRepository
{
    get
    {
        if (blogRepository == null)
        {
            blogRepository = new Repository<Blog>(m_Context);
        }
        return blogRepository;
    }
}

public Repository<Comment> CommentRepository
{
    get
    {
        if (commentRepository == null)
        {
            commentRepository = new Repository<Comment>(m_
Context);
        }
        return commentRepository;
    }
}

public Repository<User> UserRepository
```

```

    {
        get
        {
            if (userRepository == null)
            {
                userRepository = new Repository<User>(m_Context);
            }
            return userRepository;
        }
    }

    public Repository<Role> RoleRepository
    {
        get
        {
            if(roleRepository == null)
            {
                roleRepository = new Repository<Role>(m_Context);
            }
            return roleRepository;
        }
    }
}

```

The preceding code shows our `UnitOfWork` class. This class will take care of creating the actual repository class and passing the same instance of the `DbContext` object to these repositories. When we are done with all the operations, we can call the `SaveChanges` method on this class, and it will call the `SaveChanges` method on the `DbContext` object and this effectively saves all the data.

The transaction is now associated with the scope of our `UnitOfWork` class. Every `UnitOfWork` object will have its own transactions that can be used to work with multiple entities using their respective repository classes and will be committed when the `SaveChanges` method on `UnitOfWork` will get called.



The `UnitOfWork` class can also be implemented in a generic manner so that we don't have to add multiple properties, one for each repository class.

Managing categories

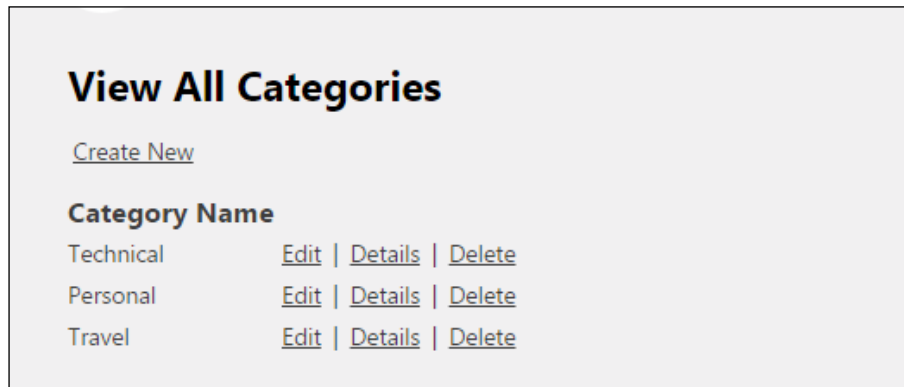
Now that we have our data access library in place, let's see how we can perform various operations on the `Category` entity.

Listing categories

To list the categories, we simply need to create our `UnitOfWork` object, and use `CategoryRepository` to retrieve the list of all the categories in the system:

```
using (UnitOfWork uow = new UnitOfWork())
{
    List<Category> categories = uow.CategoryRepository
        .GetAll()
        .ToList();
    return View(categories);
}
```

In the preceding code, we retrieved a list of all the categories and passed them to the MVC view. The view will then display all the categories, as shown in the following screenshot:



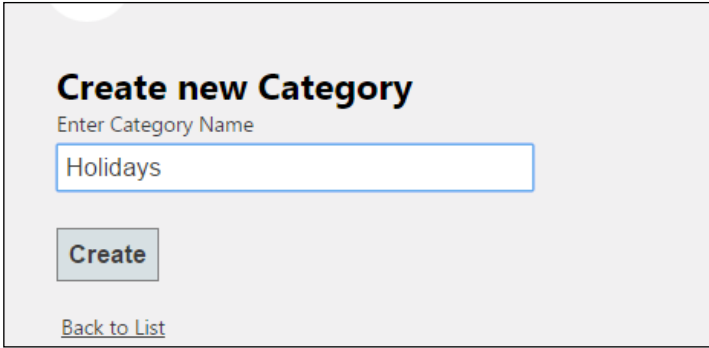
A screenshot showing a list of categories



Since we are using ASP.NET MVC, the code blocks will contain some MVC-specific code. You are free to ignore this code as this just facilitates the viewing/accessing of the retrieved data.

Adding a category

Let's now try to add a new Category. To add a new category, we need to create a Category model, and call the `Add` function of `CategoryRepository` and pass the model to it. First, let's look at the view that we will use to create a new category:



Create new Category

Enter Category Name

Holidays

Create

[Back to List](#)

A screenshot of how to create a new category

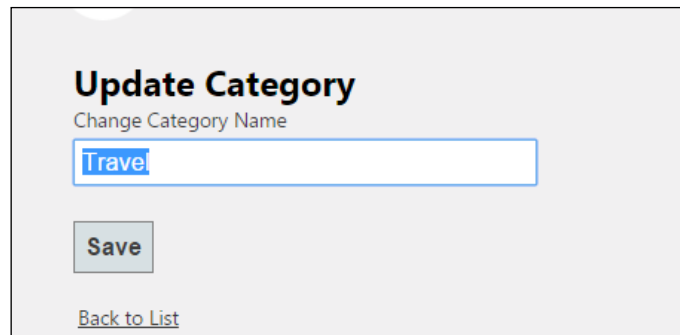
Now let's look at the code that actually creates a category in the system using our `UnitOfWork` and repository class:

```
public ActionResult Create(Category model)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
            uow.CategoryRepository.Add(model);
            uow.SaveChanges();
        }
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The preceding code shows the `Action` method of the MVC controller responsible for creating a new category. The `Category` model is being created in the view and being passed to this method. We use our `UnitOfWork` class and `CategoryRepository` to add the new category.

Updating a category

Let's now try to update an existing category. To update a category, we need to retrieve the `Category` model from the database, update the required properties, call the `Update` function of `CategoryRepository`, and pass the model to it. First, let's take a look at the view that we will use to update a `Category` model:



A screenshot of how to update a category

Now let's look at the `Action` method that retrieves the `Category` model and passes it to the preceding view:

```
public ActionResult Edit(int id)
{
    using (UnitOfWork uow = new UnitOfWork())
    {
        Category category = uow.CategoryRepository.Get(item => item.
            Id == id);

        return View(category);
    }
}
```

In the preceding code, we use the `id` value to retrieve a category and then pass it to the view so that it can be updated. The user will then update the `Category` model on the view and pass this back to the controller. Let's take a look at the `Action` method that will take this updated `Category` model, and use Entity Framework to update the data in the database:

```
[HttpPost]
public ActionResult Edit(Category model)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
```

```
        {
            uow.CategoryRepository.Update(model);
            uow.SaveChanges();
        }

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The preceding code shows the Action method of the MVC controller that is responsible for updating a category in the database. The updated Category model is being passed to this method. We use our UnitOfWork class and CategoryRepository to attach this updated model with the DbContext instance, and update the data in the database.

Deleting a category

To delete a category, we need to retrieve the Category model from the database, call the Delete function of CategoryRepository and pass the model to it. Now let's take a look at the Action method that retrieves the Category model using the id and deletes it from the system:

```
[HttpPost]
public ActionResult Delete(int id, FormCollection collection)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
            Category category = uow.CategoryRepository.Get(item =>
item.Id == id);
            uow.CategoryRepository.Delete(category);
            uow.SaveChanges();
        }

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```



The preceding code shows the `Action` method of the MVC controller that is responsible for deleting a category from the database. The view will pass the `id` value of the category that is to be deleted. We will use this `id`, and use our `UnitOfWork` class and `CategoryRepository` to first retrieve the `Category` model and then delete it from the system.

Managing blogs

Now that we have implemented the CRUD operations for `Category`, let's see how we can perform various operations on the `Blog` entity.

Adding a new blog

Let's now try to add a new blog. To add a new blog, we need to create a `Blog` model, call the `Add` function of `BlogRepository` and pass the model to it. First, let's look at the view that we will use to create a new blog:



Add a new Blog

Title

Select Category

Body

Founded in 2004 in Birmingham, UK, Packt's mission is to help the world put software to work in new ways, through the delivery of effective learning and information services to IT professionals.

Working towards that vision, we have published over 2000 books and videos so far, providing IT professionals with the actionable knowledge they need to get the job done—whether that's specific

A screenshot of how to create a new blog

In the preceding screenshot, we asked the author to provide a blog title, select the category this blog should belong to, and the actual contents of the blog. From these UI elements, a `Blog` model will be created and then passed on to the `Action` method of the controller. Let's take a look at the controller method that will add this blog to the system using our `UnitOfWork` and `Repository` class:

```

[HttpPost]
public ActionResult Create(Blog model)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
            model.CategoryId = Convert.ToInt32(Request.
Form["CategoryId"]);
            model.CreatedAt = DateTime.Now;
            model.UpdatedAt = DateTime.Now;
            model.AuthorId = uow.UserRepository.Get(user => user.
UserName == User.Identity.Name).Id;

            uow.BlogRepository.Add(model);
            uow.SaveChanges();
        }
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

```

The preceding code shows the Action method of the MVC controller that is responsible for creating a new blog. The Blog model is created in the view and is passed to this method. We then retrieve the selected CategoryId from the view and populate the CategoryId of the Blog model. Properties such as CreatedAt, UpdatedAt, and AuthorId are also populated with the appropriate values. We use our UnitOfWork and BlogRepository classes to add this new blog.



For the sake of simplicity, we will use an HTML **TextArea** to add the blog content. In the real world, it would be better to use a rich web text editor such as **CKEditor** for this purpose.

Updating a blog

Let's now try to update an existing blog. To update a blog, we just need to retrieve the Blog model from the database, update the required properties, call the Update function of BlogRepository, and pass the model to it. First, let's take a look at the view that we will use to update a blog:



Update Blog

Title
About Packt Publishing

Select Category
Technical ▾

Body

Founded in 2004 in Birmingham, UK, Packt's mission is to help the world put software to work in new ways, through the delivery of effective learning and information services to IT professionals.

Working towards that vision, we have published over 2000 books and videos so far, providing IT professionals with the actionable

Save

A screenshot of how to update a blog

Now let's look at the Action method that retrieves the Blog model and passes it to the preceding view:

```
public ActionResult Edit(int id)
{
    using (UnitOfWork uow = new UnitOfWork())
    {
        Blog blog = uow.BlogRepository.Get(item => item.Id == id);
        return View(blog);
    }
}
```

In the preceding code, we use the `id` value to retrieve a blog and then pass it to the view so that it can be updated. The user can then update the Title, Category, or the Body of the Blog model in the view and then pass this back to the controller. Let's take a look at the Action method that will take this updated Blog model, and use Entity Framework to update the data in the database:

```
[HttpPost]
public ActionResult Edit(Blog model)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
            model.CategoryId = Convert.ToInt32(Request.
Form["CategoryId"]);
            model.UpdatedAt = DateTime.Now;

            uow.BlogRepository.Update(model);
            uow.SaveChanges();
        }
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The preceding code shows the Action method of the MVC controller that is responsible for updating a blog in the database. The updated Blog model is passed to this method. We then update the CategoryId and UpdatedAt properties of this Blog model, and then use our UnitOfWork class and BlogRepository to attach this updated model with the DbContext instance, and update the data in the database.

Deleting a blog

To delete a blog, we need to retrieve the Blog model from the database, and call the Delete function of BlogRepository and pass the model to it. Now let's look at the Action method that retrieves the Blog model using the id and delete it from the system:

```
[HttpPost]
public ActionResult Delete(int id, FormCollection collection)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
```

```
        Blog blog = uow.BlogRepository.Get(item => item.Id == id);
        uow.BlogRepository.Delete(blog);
        uow.SaveChanges();
    }

    return RedirectToAction("Index");
}
catch
{
    return View();
}
```

The preceding code shows the `Action` method of the MVC controller that is responsible for deleting a blog from the database. The view will pass the `id` value of the blog that is to be deleted. We will use this `id`, and use our `UnitOfWork` class and `BlogRepository` to first retrieve the `Blog` model and then delete it from the system.



The update and delete operations on the blog should only be available to the author of the blog. Thus, the links for these operations are only shown when the logged in user is the author of the blog. The following code shows how this is being done in an ASP.NET MVC view page:

```
if (User.Identity.Name == Model.User.UserName)
{
    // Links to update and delete the blog
}
```

In the preceding code, the `User.Identity.Name` item will return the name of the currently logged in user and the `Model.User.UserName` item will return the name of the author of the blog.

Listing blogs on the home page

Let's now take a look at how we can list the blogs on the home page. This is going to be a little tricky because we would want to implement the home page with the following features:

- Only the top 10 blogs should be visible on the home page, with the possibility to navigate to the previous blogs
- The blog title and the first 200 characters should be visible for each blog entry
- Every blog should also show the number of comments posted on it
- Every blog should also show the date it was posted and last updated
- Every blog should show the author's name

To implement these features, the first thing we need to do is to create a ViewModel to show this information. We will then use LINQ to Entities projections to retrieve all this information from the database. Let's see what this ViewModel class will look like:

```
public class BlogSummaryViewModel
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Summary { get; set; }
    public string AuthorName { get; set; }
    public int CommentsCount { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime UpdatedAt { get; set; }
}
```

The preceding code shows the BlogSummaryViewModel class that we will be using to display the list of blogs on the home page. Let's now take a look at the Action method that will retrieve the list of blogs and pass them to the view to be displayed:

```
public ActionResult Index(int page = 0)
{
    using (UnitOfWork uow = new UnitOfWork())
    {
        IEnumerable<Blog> blogs = uow.BlogRepository.GetAll();

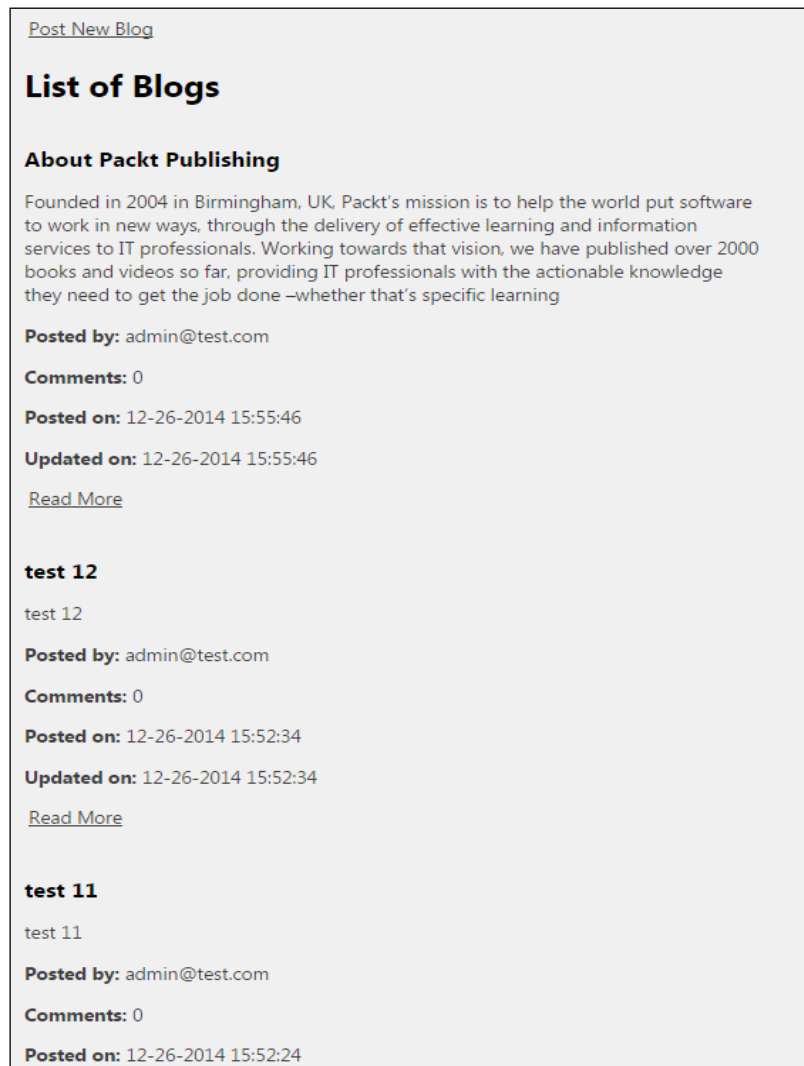
        blogs = blogs.OrderByDescending(blog => blog.UpdatedAt);

        blogs = blogs.Skip(page * 10).Take(10);

        IEnumerable<BlogSummaryViewModel> list = from blog in blogs
        select new BlogSummaryViewModel
        {
            Id = blog.Id,
            Title = blog.Title,
            Summary = blog.Body.Length > 400 ? blog.Body.Substring(0,
400) : blog.Body,
            AuthorName = blog.User.UserName,
            CommentsCount = blog.Comments.Count(),
            CreatedAt = blog.CreatedAt,
            UpdatedAt = blog.UpdatedAt
        };

        return View(list.ToList());
    }
}
```

In the preceding code, we fetched the list of blogs, ordered them in descending order of `UpdatedAt`, and then retrieved only the first 10 results based on the selected page the user is on. We then used LINQ to Entities projections to create a list of our `ViewModel` and passed it to the view. If we look at the view, we can see the list of blogs being rendered as per our requirements:



A screenshot for the list of blogs



The objective of this project is to show the functionality; thus, the HTML for the views is pretty basic and does not contain flashy styles and formatting.

Showing a single blog

Let's now take a look at the page where the user will read the blog. This is the page that will open when the user clicks on the **Read More** link on the blog list page.

On this page, we would like to show the following:

- Title of the blog
- Body of the blog
- Posted by
- Date on which this blog was first posted
- Date on which this blog was last updated
- A list of all of the comments on this blog
- Link to post a new comment on the blog

Let's now look at the Action method that will retrieve the blogs using the given Id value:

```
public ActionResult Details(int id)
{
    using (UnitOfWork uow = new UnitOfWork())
    {
        Blog blog = uow.BlogRepository.Get(item => item.Id == id);

        return View(blog);
    }
}
```


In the preceding code, we used the `id` value to retrieve the blog and sent it to the view. The view will display the blog, as shown in the following screenshot:



A screenshot for reading the blog

From the preceding screenshot, we can see that we extracted all the required information from the `Blog` model, and showed it to the user. After the blog contents, we showed a list of comments posted for this blog. There is also a link to add new comments on the page.



The new comment can also be added by providing an `Add Comment` section on this page itself. It's not been done this way, just to keep things simple from the sample application's perspective.

Managing comments

Now let's see how we can perform various operations on the `Comments` entity.

Listing categories

Since we already have the navigation property, `Comments`, in our `Blog` entity, to display the list of comments on the `Blog` details page, we just need to iterate through this collection and show all the comments:

```
@foreach (Comment item in Model.Comments)
{
    <hr />
    <p>
        @item.Body
        <br/>
        <b>Posted by: </b> @item.User.UserName

    </p>
}
```

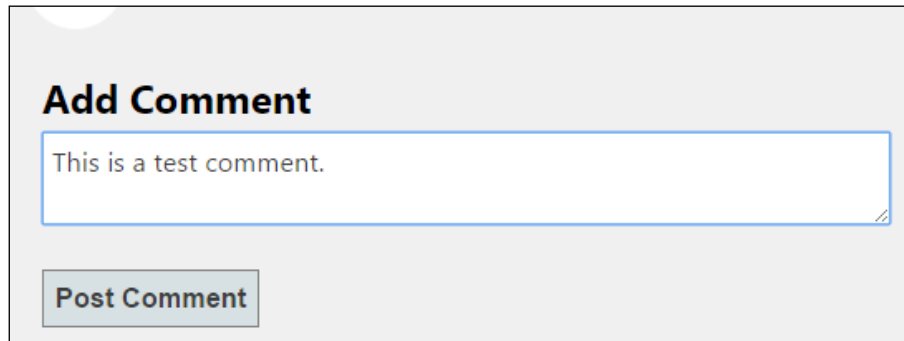
In the preceding code, we retrieved a list of all the comments posted for the blog and displayed the `body` item and poster's name. The result can be seen on the details page, as shown in the following screenshot:



A screenshot showing the list of all comments

Adding a comment

Let's now see how to add a new comment on a blog. To add a new comment, we need to create a `Comment` model, and call the `Add` function of `CommentRepository` and pass the model to it. The important thing to note here is that the `BlogId` value of this model should be the `Id` value of the blog to which this comment is being posted. First, let's take a look at the view that we will use to create a new comment:



A screenshot of how to create a new comment

Now let's look at the code that is actually creating a category in the system using our `UnitOfWork` and repository class:

```
[HttpPost]
public ActionResult CreateComment(Comment model)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
            model.CreatedAt = DateTime.Now;
            model.PosterId = uow.UserRepository.Get(user => user.
UserName == User.Identity.Name).Id;

            uow.CommentRepository.Add(model);
            uow.SaveChanges();
        }
        return RedirectToAction("Details", new { id = model.BlogId });
    }
    catch
    {
        return View();
    }
}
```

The preceding code shows the Action method of the MVC controller that is responsible for creating a new `Comment` model. The `Comment` model is being created in the view and is being passed to this method. The view itself populates the `BlogId` with the `Id` value of the blog this `Comment` is being posted to. We then use our `UnitOfWork` class and `CommentsRepository` to add this new `Comment` to the database using Entity Framework.

Deleting a comment

To delete a comment, we will use the `Id` value of the comment. We will retrieve the `Comment` model from the database using the given `Id`, and call the `Delete` function of `CommentsRepository` and pass the model to it. Now let's look at the Action method that retrieves the `Comment` model using `id` and deletes it from the system:

```
[HttpPost]
public ActionResult DeleteComment(int id, FormCollection collection)
{
    try
    {
        using (UnitOfWork uow = new UnitOfWork())
        {
            Comment comment = uow.CommentsRepository.Get(item => item.
Id == id);
            uow.CommentsRepository.Delete(comment);
            uow.SaveChanges();
        }

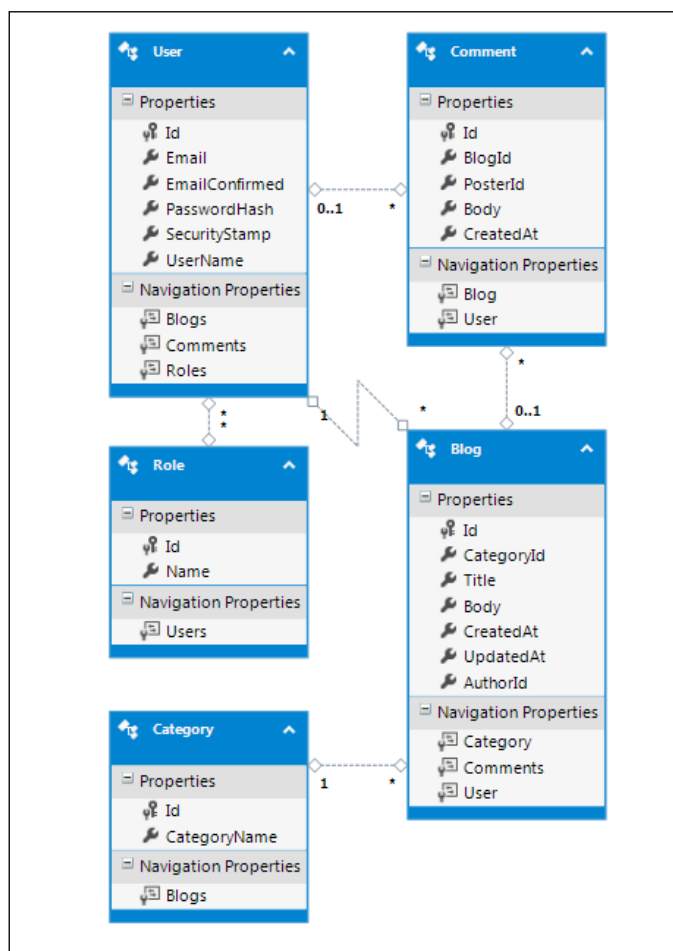
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

The preceding code shows the Action method of the MVC controller that is responsible for deleting a comment from the database. The view will pass the `id` value of the `Comment` model that is to be deleted. We will use this `id`, and use our `UnitOfWork` class and `CommentsRepository` to first retrieve the `Comment` model and then delete it from the system.

Using other Entity Framework approaches

In this chapter, we used the Entity Framework Code First approach to demonstrate how we can create a data-centric application using Entity Framework as the ORM. The reason for choosing the Code First approach is that this approach is the most verbose approach, and we get to see the code for all the entities.

If we were to use the Database First approach for this application, we just need to create the database that we saw in the beginning of the chapter, and add an Entity Data Model using the Database First approach. If we had used the Database First approach, our Entity Data Model would have looked like the following:



A generated Entity Data Model for the test application

If we want to use the Model First approach, then we need to use the Visual Entity Designer and create this model ourselves. The rest of the application code that we have seen would have remained the same even if we had used the database first or model first approach.



In larger applications using Entity Framework, typically, there are two set of models: domain model and data model. We have not done this to keep our examples simple.

Summary

In this chapter, we saw how to use Entity Framework to create data-centric applications. We saw how to use the Code First approach to develop a small blogging application. This application, when compared to real-world applications, is not even the tip of the iceberg. Real-world applications will be far larger in scale and far more complex. The idea behind this chapter was to get the user acquainted with how to start application development using Entity Framework, and how to develop end-to-end features.

With this, we reached the end of the book. Entity Framework is a vast topic and when it comes to using an ORM for data access, the devil is in the details. We tried to cover most of the topics related to Entity Framework and hopefully after reading this book you will find yourself fully geared up to use Entity Framework in any application. I hope this book has been informative.

Index

A

ACID properties 215

ADO.NET 234

aggregate operators, Entity SQL

about 202

Average 205, 206

Count 203

Max 205

Min 204

Sum 203, 204

aggregate operators, LINQ to Entities

about 181

Average 185

Count 181

Max 184

Min 183, 184

Sum 183

applications

designing, for handling field level

concurrency 222-226

requisites 243, 244

atomicity, consistency, isolation, and durability. *See* **ACID properties**

Average function 185, 205, 206

B

Blog entity

about 247, 248

relationships, implementing in 251, 252

blogs

adding 266, 267

deleting 269, 270

listing, on home page 270-272

managing 266

updating 268, 269

Blogs table 244

C

categories

adding 262, 263

deleting 265, 266

listing 262

managing 261

updating 264, 265

Categories table 244

Category entity

about 247

relationships, implementing in 250

CDSL 95

CKEditor 267

Code First approach 11, 12

Comment entity

about 248

relationships, implementing in 252

comments

adding 277, 278

deleting 278

listing 275

managing 275

Comments table 245

conceptual schema definition

language. *See* **CSDL**

concurrency

about 215

handling, in Entity Framework 219-222

RowVersion, implementing for 229-231

concurrency management 215

concurrency-related issues, handling

optimistic concurrency 216

pessimistic concurrency 216

configuration file

used, for managing database
connections 156, 157

Count function 181, 203

Create operation 106

CRUD operations, performing with

Entity Framework

about 32

existing item, updating 35, 36

item, deleting 37

list of items, reading 32, 33

new item, creating 34

specific item, reading 33, 34

custom validations

implementing, data annotations used 72-74

D

data

filtering, LINQ to Entities used 175, 176

grouping, with Entity SQL 207

ordering, with Entity SQL 206

querying, Entity SQL used 210, 211

querying, LINQ to Entities used 169-172

seeding 161-164

data access

performing 255

data annotations

used, for implementing custom
validations 72-74

used, for performing model
validations 66, 67

used, for specifying validation rules 67

used, for triggering validations 70

database connections, managing

about 155

configuration file used 156, 157

database design

visualizing 244, 245

Database First approach 10, 11

Database Generation Power Pack, Microsoft

URL 29

database initialization

managing 159

database initialization, strategies

CreateDatabaseIfNotExists 159

DropCreateDatabaseAlways 160

DropCreateDatabaseIfModelChanges 160

MigrateDatabaseToLatestVersion 160

setting 160

database relationships 39

data binding environments

validations, triggering in 70

data, querying

Entity SQL with ObjectQuery,
used 198, 199

DbContext class

implementing 253, 254

development styles

Code First 11

comparing 11, 12

Database First 10

Model First 11

domain class configurations, Entity

Framework Code First approach

Column 125

DatabaseGenerated 125

ForeignKey 125

Key 125

NotMapped 125

Table 125

Timestamp 125

domain modeling, with inheritance 75

E

eager loading 189, 192

EDMX file

conceptual schema definition 18

mapping 19

storage schema definition 19

EntityClient object 169

EntityCommand

Entity SQL, used for querying data 210, 211

Entity SQL, using with 210

using, with parameterized Entity

SQL 211, 212

EntityConnection object

creating 194, 195

Entity Data Model (EDM)

about 8, 9

- creating 245
- entity classes, creating 246
- Entity Framework**
 - about 7
 - and pessimistic concurrency 231
 - architecture, visualizing 8
 - benefits 7
 - concurrency, handling 219-222
 - domain modeling, with inheritance 75
 - modeling 8
 - persistence 8
 - used, for implementing optimistic concurrency 219
 - used, for modeling many-to-many relationship 49-51
 - used, for modeling one-to-one relationship 47, 48
 - used, for performing model validations 59
 - using, with functions 112
 - using, with stored procedures 106
 - using, with views 102-105
- Entity Framework 6**
 - used, for managing transactions 238, 239
- Entity Framework Code First approach**
 - about 120
 - benefits 120
 - configurations 121
 - conventions 121
 - domain class configurations 124-126
 - implementing 121-124
 - using 30, 31
- Entity Framework Database First approach 12-21**
- Entity Framework, default transaction handling 235, 236**
- Entity Framework Model First approach 22-29**
- Entity model**
 - creating 43, 44
- Entity relationships, Code First approach**
 - managing 126
 - many-to-many relationships, implementing 135-140
 - one-to-many relationships, implementing 126-131
 - one-to-one relationships, implementing 131-135
- Entity SQL**
 - about 193, 194
 - data, grouping with 207
 - data, ordering with 206
 - navigation properties, using with 201, 202
 - using, with EntityCommand 210
 - using, with ObjectQuery 196-198
- Entity SQL, with ObjectQuery**
 - used, for querying data 198, 199
- existing SqlConnection**
 - using 158

F

- field level concurrency**
 - implementing 226-228
- fields**
 - validating 67, 68
- functions**
 - Entity Framework, using with 112
 - scalar function 112-114
 - table valued functions (TVF) 114-117
 - using 101

G

- grouping**
 - performing, LINQ to Entities used 178, 179

I

- inheritance strategy**
 - selecting 99
- inheritance, with Entity Framework Code First approach**
 - about 140
 - Table per Class Hierarchy (TPH)
 - inheritance, implementing 145-150
 - Table per Concrete Class (TPC) inheritance, implementing 150-154
 - Table per Type (TPT) inheritance, implementing 140-145

J

join

- implementing, LINQ to Entities
 - used 188, 189

L

lambda expression 189

Language-Integrated Query. *See* LINQ

lazy loading 189-191

length, of fields

- validating 68

Limit query 209

LINQ 21, 167

LINQ projections

- using, with LINQ to Entities 177, 178

LINQ to Entities

- about 167-169
- aggregate operators 181-185
- LINQ projections, using with 177, 178
- navigation properties, using with 174, 175
- used, for executing simple queries 173
- used, for filtering data 175, 176
- used, for grouping operations 178, 179
- used, for implementing join 188, 189
- used, for ordering 179-181
- used, for querying data 169-172
- using 172, 173

M

many-to-many relationship

- about 41
- implementing 42
- modeling, Entity Framework used 49-51

mapping specification language. *See* MSL

Max function 184, 205

Min function 183, 184, 204

Model First approach 11, 12

model validations, performing

- data annotations used 66, 67
- Entity Framework used 59
- partial class methods used 60
- partial methods used 62-66

MSL 95

N

navigation properties

- creating 248
- using, with Entity SQL 201, 202
- using, with LINQ to Entities 174, 175

navigation properties, data access

- about 51
- item, adding 53
- item, deleting 55
- item, updating 54
- list of items, retrieving 52
- specific item, retrieving 51, 52

non-data binding environments

- validations, triggering in 70, 71

O

ObjectContext class 9, 10

ObjectQuery

- Entity SQL, using with 196-198
- parameterized Entity SQL, executing
 - with 199, 200
- used, for grouping data with
 - Entity SQL 207
- used, for ordering data with
 - Entity SQL 206

object relational domain modeling,

inheritance relationship

- Table per Class Hierarchy (TPH) 76, 82, 83
- Table per Concrete Class (TPC) 76, 92
- Table per Type (TPT) 76, 77

Object Relational Mapper (ORM) 7

one-to-many relationships

- about 40
- implementing 40
- modeling 44-46

one-to-one relationship

- about 40
- implementing 41
- modeling, Entity Framework used 47, 48

optimistic concurrency

- about 216, 217
- changes, rejecting 217
- conflict/forcing updates, ignoring 217
- implementing, Entity Framework used 219
- partial updates 217
- user, warning 217

ordering
with LINQ to Entities 179-181

P

paging, Entity SQL ObjectQuery
implementing 209, 210
paging, LINQ to Entities
implementing 187
parameterized Entity SQL
EntityCommand, using with 211, 212
executing, with ObjectQuery 199, 200
partial class methods
used, for performing model validations 60
partial methods
about 60-62
used, for performing model
validations 62-66
**partitioning methods, Entity SQL
ObjectQuery**
about 208
Limit 209
Skip 208
partitioning methods, LINQ to Entities
about 185
Skip 186
Take 187
persistence approaches
selecting 38
pessimistic concurrency
about 216-218
and Entity Framework 231
issues 218
Plain Old CLR Objects (POCO) 30, 119
procedures
using 101

R

regular expression-based validations 69
relationships
Blog-Comment 248
Category-Blog 248
implementing, in Blog entity 251, 252
implementing, in Category entity 250
implementing, in Comment entity 252
implementing, in Role entity 250
implementing, in User entity 249

many-to-many relationship 41, 42
one-to-many relationships 40
one-to-one relationship 40, 41
properties, creating 248
User-Blog 248
User-Comment 249
User-Role 248

Repository pattern 255-258

Role entity
about 246
relationships, implementing in 250

Roles table 244

RowVersion
implementing, for concurrency 229-231

S

scalar function
using 112-114
simple queries
executing, LINQ to Entities used 173
single blog
showing 273-275
Skip query 186, 208
SSDL 95
stored procedures
defining 106-112
Entity Framework, using with 106
Sum function 183, 203, 204

T

Table per Class Hierarchy (TPH) inheritance
about 82, 83
concrete classes, adding to Entity Data
Model 84-86
concrete class properties, mapping to
respective tables and columns 87-89
default Entity Data Model, generating 84
entities, using via DbContext object 90, 91
Table per Concrete Class (TPC) inheritance
about 92
abstract class, creating 94, 95
CDSL, modifying 95
default Entity Data Model, generating 93
entities, using via DbContext object 98
mapping, specifying for TPT inheritance
implementation 96, 97

Table per Type (TPT) inheritance

- about 76, 77
- default Entity Data Model,
 - generating 77, 78
- default relationships, deleting 78
- entities, using via DbContext object 81, 82
- inheritance relationships, adding between
 - entities 79, 80

table valued functions (TVF)

- using 114-117

Take query 187**test environment**

- setting up 234, 235

TextArea 267**transaction management**

- selecting 241

transactions

- about 233
- existing transaction, using with Entity Framework's DbContext class 239-241
- handling, TransactionScope used 237
- managing, with Entity Framework 6 238, 239

TransactionScope

- used, for handling transactions 237

U**Unit of Work class 259, 261****User entity**

- about 246
- relationships, implementing in 249

UserRoles table 244**Users table 244****V****validation rules**

- specifying, data annotations used 67

validations, triggering

- data annotations used 70
- in data binding environments 70
- in non-data binding environments 70, 71

ViewModel 177**views**

- Entity Framework, using with 102-105
- using 101



Thank you for buying **Mastering Entity Framework**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

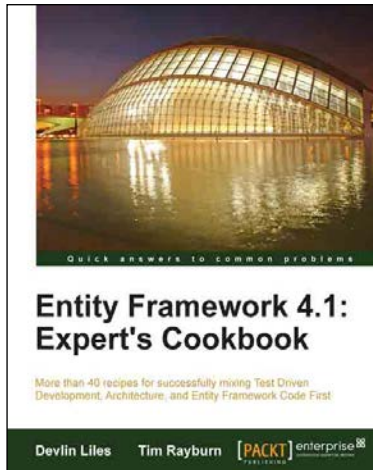
About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft, and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



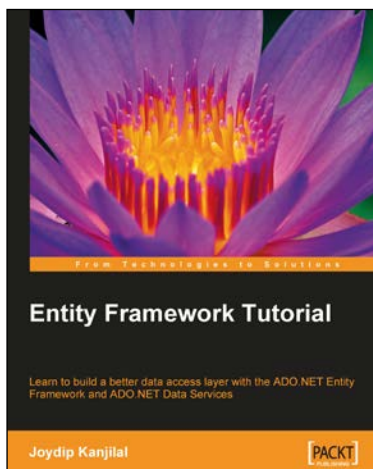
Entity Framework 4.1: Expert's Cookbook

ISBN: 978-1-84968-446-0

Paperback: 352 pages

More than 40 recipes for successfully mixing Test Driven Development, Architecture, and Entity Framework Code First

1. Hands-on solutions with reusable code examples.
2. Strategies for enterprise ready usage.
3. Examples based on real world experience.
4. Detailed and advanced examples of query management.



Entity Framework Tutorial

ISBN: 978-1-84719-522-7

Paperback: 228 pages

Learn to build a better data access layer with the ADO.NET Entity Framework and ADO.NET Data Services

1. Clear and concise guide to the ADO.NET Entity Framework with plentiful code examples.
2. Create Entity Data Models from your database and use them in your applications.
3. Learn about the Entity Client data provider and create statements in Entity SQL.
4. Learn about ADO.NET Data Services and how they work with the Entity Framework.

Please check www.PacktPub.com for information on our titles



WCF 4.5 Multi-Layer Services Development with Entity Framework

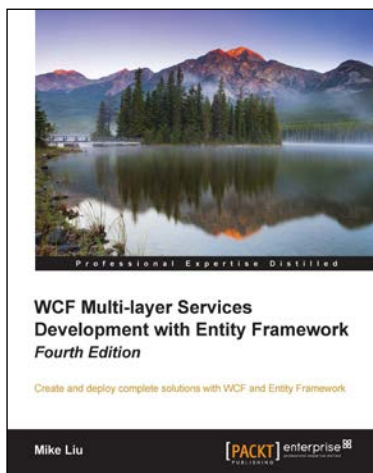
Third Edition

ISBN: 978-1-84968-766-9

Paperback: 394 pages

Build SOA applications on Microsoft platforms with this hands-on guide

1. This book will teach you WCF, Entity Framework, LINQ, and LINQ to Entities quickly and easily.
2. Apply best practices to your WCF services and utilize Entity Framework in your WCF services.
3. Practical, with step-by-step instructions and precise screenshots, this is a truly hands-on book for all C++, C#, and VB.NET developers.



WCF Multi-layer Services Development with Entity Framework

Fourth Edition

ISBN: 978-1-78439-104-1

Paperback: 378 pages

Create and deploy complete solutions with WCF and Entity Framework

1. Build SOA applications on Microsoft platforms.
2. Apply best practices to your WCF services and utilize Entity Framework to access underlying data storage.
3. A step-by-step, practical guide with nifty screenshots to create six WCF and Entity Framework solutions from scratch.

Please check www.PacktPub.com for information on our titles