



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Code-First Development with Entity Framework

Take your data access skills to the next level with Entity Framework

Sergey Barskiy

[PACKT]
PUBLISHING

www.allitebooks.com

Code-First Development with Entity Framework

Take your data access skills to the next level
with Entity Framework

Sergey Barskiy

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Code-First Development with 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: March 2015

Production reference: 1110315

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

ISBN 978-1-78439-627-5

www.packtpub.com

Credits

Author

Sergey Barskiy

Project Coordinator

Rashi Khivansara

Reviewers

Erik Ejlskov Jensen

Andriy Svyryd

Proofreaders

Maria Gould

Elinor Perry-Smith

Commissioning Editor

Sarah Crofton

Indexer

Mariammal Chettiyar

Acquisition Editor

Usha Iyer

Production Coordinator

Manu Joseph

Content Development Editor

Natasha D'Souza

Cover Work

Manu Joseph

Technical Editor

Narsimha Pai

Copy Editor

Deepa Nambiar

About the Author

Sergey Barskiy is an architect with Tyler Technologies. He lives in Atlanta, GA. He has been developing software for almost 20 years. Sergey is a Microsoft MVP. He holds these Microsoft certifications: MCPD, MCTS, MCSD for .NET, MCAD for .NET, MCDBA, and MCP. He has been working with Microsoft Technologies for over 15 years. He is a frequent speaker at various regional and national conferences, such as VS Live, DevLink, CodeStock, and Atlanta Code Camp, as well as local user groups. He is one of the organizers of Atlanta Code Camp. He authored articles for *Code Magazine*.

Sergey Barskiy has been using Entity Framework since it was first released to the public. He has deployed a number of projects to production that used Entity Framework over the years. He has used the Code-First approach on a few different projects as well. Sergey has produced an online video training course for this technology. He has spoken on Entity Framework Code-First at a number of national and regional conferences and events.

You can tweet to him at @SergeyBarskiy or e-mail him at sergey@barskiy.com.

I would like to thank my family for putting up with my busy schedule during the time I was working on this book. I want to also thank Packt Publishing for giving me the courage and opportunity to work on this project.

About the Reviewers

Erik Ejlskov Jensen is a Danish .NET developer who specializes in .NET data development. He is a Microsoft MVP for SQL Server and shares tips and code via his blog at <http://erikej.blogspot.com> and Twitter at @ErikEJ. He is a project manager for a number of SQL Server Compact and SQLite tools on the Codeplex site, and he is the creator of the popular free Visual Studio add-in SQL Server Compact/SQLite Toolbox. He also contributes to a number of open source projects, including Entity Framework.

Andriy Svyryd was born in Ukraine, and then he moved to Mexico, where he graduated from Universidad Nacional Autónoma de México (UNAM). His first job was at Microsoft, where he worked on several projects related to data modeling. He was a developer on the Entity Framework team for 4 years.

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.

Table of Contents

Preface	v
Chapter 1: Introducing Entity Framework	1
What is ORM?	1
A brief history of Entity Framework	3
The capabilities of Entity Framework	4
The Entity Framework architecture	5
Self-test questions	6
Summary	6
Chapter 2: Your First Entity Framework Application	7
Creating a new project that uses Entity Framework	8
Creating a new database based on .NET classes	9
Saving a new record to the database	12
Querying data in a database	15
Updating a record	16
Deleting a row from the database	17
Introduction to schema changes	18
Self-test questions	22
Summary	23
Chapter 3: Defining the Database Structure	25
Creating table structures	26
Mapping .NET types to SQL types	26
Configuring primitive properties	27
Handling nullable properties	33
Defining relationships	35
The One-to-Many relationship	35
The Many-to-Many relationship	41
The One-to-One relationship	42

Self-test questions	44
Summary	45
Chapter 4: Querying, Inserting, Updating, and Deleting Data	47
The basics of LINQ	47
Filtering data in queries	49
Sorting data in queries	51
Exploring LINQ functions	52
Element operations	52
Quantifiers	53
Working with related entities	54
Filtering based on related data	54
Lazy and eager loading	55
Inserting data into the database	57
Updating data in the database	60
Deleting data from the database	65
Working with in-memory data	67
Self-test questions	69
Summary	70
Chapter 5: Advanced Modeling and Querying Techniques	73
Advanced modeling techniques	74
Complex types	74
Using an explicit table and column mappings	77
Adding supporting columns	78
Enumerations	79
Using multiple tables for a single entity	80
Advanced querying techniques	83
Projections	83
Aggregations and grouping	88
Advanced query construction	89
Paging data with windowing functions	92
Using joins	93
Groupings and left outer joins	95
Set operations	101
Self-test questions	102
Summary	104

Chapter 6: Working with Views, Stored Procedures, the Asynchronous API, and Concurrency	105
Working with views	106
Working with stored procedures	110
Create, update, and delete entities with stored procedures	112
The asynchronous API	115
Handling concurrency	119
Self-test questions	123
Summary	124
Chapter 7: Database Migrations and Additional Features	125
Enabling and running migrations	126
Using the migrations API	130
Applying migrations	135
Applying migrations via a script	136
Applying migrations via migrate.exe	136
Applying migrations via an initializer	137
Adding migrations to an existing database	138
Additional Entity Framework features	139
Custom conventions	139
Geospatial data	140
Dependency injection and logging	140
Startup performance	141
Multiple contexts per database	141
Self-test questions	142
Summary	143
Appendix: Answers to Self-test Questions	145
Index	151

Preface

I have been writing applications on the Microsoft platform for almost 2 decades. Many, if not all of them, use databases to persist user data. I have used many technologies to access data, starting with ADO.NET. Object Relational Mapping (ORM) tools, have many advantages over ADO.NET. They allow developers to write data access code faster and safer. ORM tools have been designed to solve impedance mismatch problems between object-oriented programming and relational databases. Microsoft's Entity Framework is the company's answer to the demand for an ORM from .NET developers. This book is the guide that will help you acquire the necessary skills to program your applications using Entity Framework.

This book centers on the Code-First approach with Entity Framework, which has become the most common way of using the technology. Code-First allows developers to control the entire data access layer of their applications from the .NET code. This approach simplifies and streamlines the entire application development life cycle, keeping developers coding inside Visual Studio, the only tool they need to use Entity Framework.

The book starts with the basic concepts of defining the database structure via C# and VB.NET code, then progresses to full data access. Chapters cover create, read, update, and delete operations (CRUD) with Entity Framework. It also shows how to update the Relational Database Management System, (RDBMS) structure, via the migrations API. It explores aspects of data access in both .NET languages using the Languages INtegration Query (LINQ), API. Because of Microsoft's continuous commitment to both C# and VB.NET, the book contains examples in both languages in every chapter.

I have been using Entity Framework since 2008, and I felt that I had the necessary experience to write a book on the subject. I spoke on the topic on many conferences and events and saw tremendous interest in creating a concise guide to Entity Framework. This was one of my primary motivations in creating a shorter textbook. I read many technical books while working in the industry, and I myself, at times, had trouble maintaining the focus while reading 800-page technical books. They definitely have a place in the industry and are very useful. However, I feel they are intimidating for the developers who are just getting started with a particular technology. My hope is that this book will get you going quickly on the new topic and have you writing data access code in a few hours. You should be able to master the foundation behind Entity Framework with this book quickly and easily.

What this book covers

Chapter 1, Introducing Entity Framework, gives us an understanding of what the, Object Relational Mapping (ORM) technology brings to developers. You learn the history of Entity Framework as an example of an ORM. We study the architecture behind the Entity Framework technology.

Chapter 2, Your First Entity Framework Application, teaches us how to create our first project that uses Entity Framework. We create classes that map to database tables. We observe how our target database is created when the project is run. Finally, we save and retrieve our first data from the created database.

Chapter 3, Defining the Database Structure, dives deep into details of mappings between classes and tables. We create maps between properties to columns as well as rules that govern such mappings. We define relationships between classes that translate into relationships between tables. We exercise multiple approaches that can be used to define the mappings.

Chapter 4, Querying, Inserting, Updating, and Deleting Data, discusses how to use the LINQ API, that allows developers to retrieve the data from the database. We sort, filter, and perform element operations and use quantifiers. We query related entities. You learn the advantages and pitfalls of eager and lazy loading. We insert, delete, and update the data.

Chapter 5, Advanced Modeling and Querying Techniques, dives deeper into modeling and querying techniques. We use complex types to have more consistency in the database structures. We create an explicit table and column names. We define structures that use table and entity splitting. We use projections in queries to make them more efficient and summarize our data. We page the data for retrieval, breaking it up for presentation to the users. We use joins to create queries that use related entities.

Chapter 6, Working with Views, Stored Procedures, the Asynchronous API, and Concurrency, shows how to access with database views from Entity Framework. We query data via stored procedures using the Entity Framework API. We perform create, update, and delete operations with stored procedures. We exercise Entity Frameworks and the asynchronous API and learn the advantages and pitfalls of asynchronicity. We implement concurrency handling, learning to handle the situation when multiple users attempt to update the same data.

Chapter 7, Database Migrations and Additional Features, shows how to enable migrations on our Entity Framework project, creating and updating the database schema without data loss. We use implicit migrations first, then create explicit migrations, customizing our migration code. We use common aspects of the migrations API, adding columns and specifying default values. We apply migrations using multiple approaches. We create migrations from an existing database. We dive briefly into useful Entity Framework features, not covered previously.

Appendix, Answers to Self-test Questions, contains answers to questions you will find throughout the book.

What you need for this book

In order to run the sample code, you will need access to Visual Studio 2013. You can use free Community Edition. You also need an instance of SQL Server 2008 R2 or higher on your machine. The free Express edition of SQL Server can be used.

Who this book is for

This book is intended for software developers with some prior experience in the Microsoft .NET framework who want to learn how to use Entity Framework. Maybe you have used SQL for years, but want to write data access code more easily and safely, using C# or VB.NET instead. This book is for you if you want to learn how to use this Microsoft ORM to create strongly typed data access logic, or want to get your database changes deployed with minimal effort. This book will get you up and going quickly, providing many examples for C# and VB.NET programmers that illustrate all the key concepts of Entity Framework.

Conventions

In this book, you will find a number of styles of text 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, user input are shown as follows:
"You also need at least one class that represents the database itself, which will inherit from DbContext."


A block of code is set as follows:

```
Public Class Person
    Property PersonId() As Integer
    Property FirstName() As String
    Property LastName() As String
End Class
```

Any command-line input or output is written as follows. "If we need to get detailed help for the PowerShell commandlet `Enable-Migrations`, we just need to type `Get-Help Enable-Migrations`."

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

 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 <http://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

Introducing Entity Framework

In this chapter, you will be introduced to Entity Framework. You will gain an understanding of **Object-Relational Mapping (ORM)** tools and the problems they solve. A brief history of Entity Framework will also be covered in this chapter. We will examine the capabilities of Entity Framework and its architecture.

In this chapter, we will cover the following topics:

- ORM tools and the problems they solve
- A brief history of Entity Framework
- The capabilities of Entity Framework
- The overall architecture of Entity Framework

What is ORM?

When it comes to business software, almost all of it needs to store data that pertains to its functions. For many decades, **Relational Database Management System (RDBMS)** has been a go-to data storage for developers. ORM is a set of technologies that allows developers to access RDBMS data from an object-oriented programming language. There are other RDBMSes available, such as SQL Server, Oracle, DB2, MySQL, and many more. These database systems share some common characteristics. Each system supports one or more databases. Databases consist of many tables. Each table stores data in a tabular format, divided into columns and rows. Data rows in multiple tables may relate to each other. For example, a person's details stored in the `Person` table can have phone numbers stored in a separate `Phones` table.

In the following screenshot, you can see a table that allows you to store a person's information, specifically their first and last names, along with a unique identifier for each person. This type of storage, where similar data items are grouped together into tabular structures is typical:

PersonId	FirstName	LastName
1	John	Doe
2	Jane	Williams
3	John	Smith
4	James	Johnson

Each column can also be constrained in some ways. For example, **PersonId** is an integer column. **LastName** is `nvarchar(50)` column, which means you can store Unicode data of variable size in it, up to 50 characters. You will see in subsequent chapters how we describe this information using Entity Framework.

The data stored in each column and row combination is scalar data, such as number or string. When software needs to persist or retrieve data, it must describe its intent, such as insert or select query, using the database-specific language called **Structured Query Language (SQL)**. SQL is a common standard for all relational database systems, as issued by the **American National Standards Institute (ANSI)**. However, some database systems have their own dialect on top of the common standard. In this book, we are not going to dive into the depths of SQL, but some concepts are important to understand. There are some basic commands that we need to look at. These are typically described as CRUD. **CRUD** stands for **Create, Retrieve, Update, and Delete**. For example, if you want to retrieve or query the data from the preceding example, you would type the following:

```
SELECT PersonId, FirstName, LastName
FROM Person
```

Historically, before tools such as Entity Framework, developers embedded SQL language statements inside the software code using .NET languages, such as C# or VB.NET or other programming languages, such as C++ or Java. The reason for this is that these languages do not natively speak or understand SQL. For example, to retrieve the data from the database and manipulate it as objects, you would write a fair amount of code using *ADO.NET*, .NET Framework's data access built-in framework. You would need to define a class to hold a person's data. Then, you would need to open a connection to the database, create a command that uses the preceding query as its text, execute the command's reader, and iterate through the reader results, populating an instance of our `Person` class with the data from the reader. As you can see, there would be a lot of steps involved. More importantly, the code we'd write would be quite fragile.

For example, if we change the column name in our database from `FirstName` to `First_Name`, our code would still compile just fine, but would throw an exception when we try to run it. Moreover, the data in the database was stored as scalar values organized in columns and rows in a table, but our destination was an object or object graph. As you can see, this way of accessing the data has a number of issues.

First of all, there is a type mismatch between RDBMS column types and .NET types. Second, there is a mismatch between storage, which is a collection of scalar values, and destination, which is an object with properties. To further complicate the situation, our person object could also have a complex property that contains a list of phone numbers, which would be represented by a completely different table. These problems are collectively referred to as **impedance mismatch** between object-oriented programming and relational databases.

The set of tools called *ORM* came about to solve this mismatch problem. An ORM tool represents data stored in database tables as objects, native to a programming language, such as .NET languages, C#, and VB.NET. ORM tools have many advantages over the traditional code, such as ADO.NET code that we mentioned. They expose the data using native .NET types. They expose related data using simple .NET properties. They provide compile time checking. They solve the problem with typos. Developers do not have to use SQL, a different language. Instead in the .NET world, developers use **Language INtegrated Query (LINQ)** to query the data. LINQ is simply part of C# and VB.NET languages. We will cover the basics of LINQ in subsequent chapters. By the same token, programmers use an ORM tool's API to persist data to the database. Finally, as we will see later, you will write less code. Less code means fewer bugs, right?

A brief history of Entity Framework

Over the years, there have been many ORM tools entering the market; some commercial, others open source. Microsoft developed its own tools. First one was *LINQ to SQL*, which was built on .NET 3.5. This ORM only worked with SQL Server and SQL Server Compact. **Entity Framework**, which first shipped in 2008, was the second attempt. It had a number of advantages over LINQ to SQL. First of all, it had provider architecture, thus was open to working with all relational database engines, not just SQL Server, given that a provider was written for the engine in question. All major RDBMSes have Entity Framework providers at this point in time.

Entity Framework went through a few revisions. In the first version, only *Database First* approach was supported. What this meant was that you would point the designer to an existing database. As a result, code was generated that would contain a database and table abstractions. In addition to the code, an *EDMX* file was also created. This XML file contained Entity Data Model. It consisted of three models: logical, storage, and mapping. The logical, sometimes called conceptual, model is the one you will code against in C# or VB.NET. Storage model describes how data is stored in a database. The mapping model, as the name implies, provides the mapping between logical and storage models. If you were to change anything in the database, you would need to refresh the generated model. The C# or VB.NET code is also generated again. The mapping model has a class based on `ObjectContext` that has collection properties for each table in the database. Each collection is a generic collection, where collection item type is inherited from a base class in Entity Framework. Each class has properties that correspond to columns in the matching table.

In the second revision, version 4, the *Model-First* approach was supported as well. With this approach, you can use design surface to create entities, and then the designer would produce the SQL script to generate the database. With this approach, the *EDMX* file was still created, and the final result was the same as with the *Database First* approach. Developers had access to the same set of classes to give them the ability to persist and query data.

Finally, the Entity Framework *Code-First* approach was shipped in version 4.1. This approach eliminated the need for the *EDMX* file. It also eliminated the dependency on Entity Framework base classes that each entity in the model inherited from. As a result, the code became more testable. This approach also eliminated the need for the designer. You could just type your classes, and they would automatically be mapped to tables in the database. There have been subsequent Entity Framework *Code-First* releases after the the initial 4.1 version.

The capabilities of Entity Framework

Entity Framework can do a lot for us as Microsoft developers. First of all, it is capable of exposing the database as a set of objects. It does so by utilizing a couple of key classes. First and foremost, you need to be aware of `DbContext`. This class is at the heart of Entity Framework *Code-First*. At a high level, it is a database abstraction. Databases consist of tables, each consisting of rows and columns. `DbContext` in turn has generic collection properties; each of which can be typed as `DbSet<TRowType>`, corresponding to each table. Each object within the collection, referred to as an entity, represents a row in the corresponding table. Columns are defined by properties of the `TRowType` class that is specified as a generic argument of each collection.

Once this structure is laid out, you are capable of querying the underlying database by using LINQ queries. If you add a brand new instance of the `TRowType` class to its parent collection and then save the changes using the `DbContext` API, this new object will *become* a row in the corresponding table, where each property value of that object will become a column value in the target row. On top of this, Entity Framework has capabilities to represent other database artifacts, such as procedures and functions. You will be able to query the data using functions, just like tables using LINQ again. The question of evolving the database structure is an important one. In most cases, you will need to add columns and tables, as your application changes. Entity Framework addresses this need via the Migrations feature. This ability will allow you to alter the database structure through C# code. In addition to adding and deleting tables and columns, you will be able to add *indexes*. **Migrations** allow developers to evolve a schema without data loss. As you can see, Entity Framework exposes everything you need to access the data in your C# or VB.NET code without wiring SQL and treats your database as another part of your overall application code. You can check migrations code into source control, since it is also C# code!

The Entity Framework architecture

Entity Framework is built on the provider architecture. When a developer creates a LINQ query using C# or VB.NET, the framework engine in conjunction with a provider converts it into an actual SQL statement that is sent to the database. Any given provider is the link between Entity Framework and a specific RDBMS that this provider is written for. In this book, we will concentrate on the Code-First approach, but this architecture is used in the Database First approach as well. Once the provider executes the final SQL command, its results are materialized into .NET objects by Entity Framework. Data reader is used for this purpose. It is important to understand that Entity Framework is still built on top of ADO.NET, thus it uses concepts such as connection, command, and data reader. When it comes to data persistence, in other words; insert, update, and delete functionalities, the flow is as follows: In the case of inserts, a developer adds an instance of an entity class to the context. Similarly, an entity previously added to the context can be flagged as changed or deleted, causing the update or delete SQL statement to be executed against the database, respectively. Entity Framework examines the state of each object in its context, using the provider again to create an RDBMS-specific insert, update, or delete command.

Self-test questions

Q1. Which of these problems does an ORM tool solve?

1. Types in RDBMS and .NET framework are the same
2. Impedance mismatch between RDBMS and object-orientated programming
3. Learning SQL is hard

Q2. Developers must write SQL queries to work with Entity Framework. True or false?

Q3. What is the name of the technology that Entity framework uses to apply structural changes to the target database?

1. Updates
2. Conversions
3. Migrations

Q4. Which is the key class that represents database abstraction with the Entity Framework Code-First approach?

1. DbContext
- 2.ObjectContext
3. DataContext

Q5. Entity Framework can only work with Microsoft databases, such as SQL Server. True or false?

Summary

In this chapter, we took a look at how data is stored in RDBMS systems. We saw the shortcomings of using embedded SQL to access the data. We understood what ORM tools are all about and what problems they solve. We examined the history behind Entity Framework. We saw the capabilities of Entity Framework. Finally, we had a brief excursion into the Entity Framework architecture.

In the next chapter, we will actually build our first application based on Entity Framework Code-First.

2

Your First Entity Framework Application

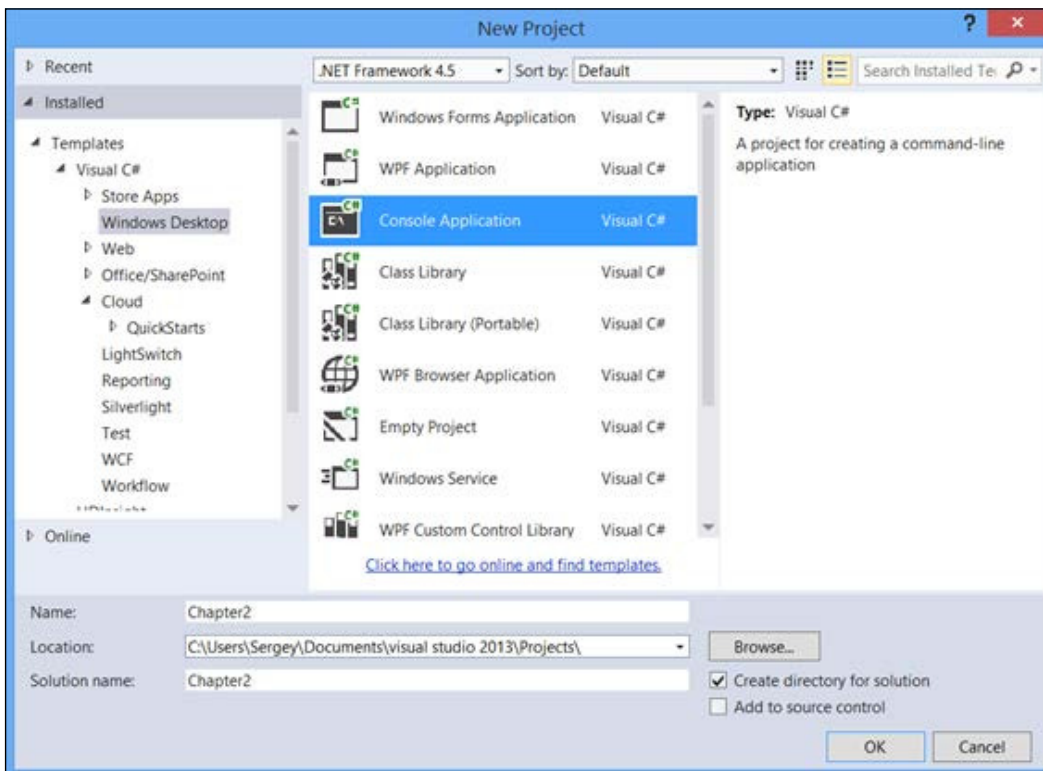
In this chapter, we will work through the creation of a brand new project that uses Entity Framework. We will create classes that map to tables in the target database. We will then insert a row into that table using the Entity Framework API. We will also query this using LINQ. Next, we will update and delete our test data. Finally, we will take a look at how to handle schema changes.

In this chapter, we will cover the following topics:

- Creating a new project using Entity Framework
- Adding the necessary references to be able to write Entity Framework code
- Creating a new database based on written classes
- Saving a new record
- Querying the saved data
- Deleting and updating the data in the database
- An introduction to schema changes


Creating a new project that uses Entity Framework

First of all, it is important to understand how Entity Framework is distributed. Even though it is an open source project, Microsoft employees curate the project as well as write the lion's share of all the code. You can actually download the source code from CodePlex at <https://entityframework.codeplex.com/>. However, the easiest way to add this technology to your project is to use NuGet. The NuGet technology allows anyone to create useful libraries and publish them on the web to let other developers take advantage of it. Microsoft is in charge of publishing Entity Framework on the NuGet website. The package is simply called Entity Framework. In addition to the core Entity Framework, it also contains the Entity Framework provider for SQL Server. We will work with the latest version of it. You can add it to any .NET project. Let's just create a **Console Application** for our project first, and then add the Entity Framework package to it. Create your project and solution by going through **File | New | Project**, then picking either C# or VB.NET, and then finally selecting **Console Application** under the **Windows Desktop** node, as shown in the following screenshot:



Now, you can use either the **Package Manager Console** window or **Manage NuGet Packages** for the **Solution** window, to add the `EntityFramework` package to your solution. Both windows are available by navigating to **Tools | NuGet Package Manager** for the **Solution** menu in Visual Studio. If you are using the **Package Manager Console** window, just type `Install-Package EntityFramework`.


In this window, hit `Enter` key. If you are using **Manage Packages** for the **Solution** window, type `EntityFramework` in the search box, click on **Search**, and then add the package with the ID of Entity Framework, which should be the first package in the result set. Once the package has been added, the project will contain all the necessary references. You are now ready to start writing code.

 You must be connected to the Internet to use the NuGet online package repository.

Creating a new database based on .NET classes

When it comes to working with data, we will need to create at least two types of classes. We need to create one or more classes to map the tables in the database, where each class represents a row of data in the corresponding table. You also need at least one class that represents the database itself, which will inherit from `DbContext`. To start with, let's create a class with the same structure as the `Person` table from the *Chapter 1, Introducing Entity Framework*, with properties for `id` and the first and last names. Here is how the class looks in C#:

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

 **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.

As you can see, we need to define properties with types that match our desired database types. In this case, the `String` type in .NET will map to all possible character types in an RDBMS. Numeric types are easier. In our case, the `Integer` type in .NET will map to the `int` type in SQL Server. Most of the time, you do not need to be concerned with this level of detail. Just model your classes to represent the data you need, using the standard .NET types for its properties, and let Entity Framework figure out what native RDBMS types are needed to persist this data. The same class in VB.NET looks as follows:

```
Public Class Person
    Property PersonId() As Integer
    Property FirstName() As String
    Property LastName() As String
End Class
```

Now, let's write out a context, which will be our database abstraction with a single table: `Person`. By this logic, we only need a single property in our context to represent this table. Since a table consists of rows, it is logical to assume that this property must be a collection of persons. Entity Framework has a specific class for this exact purpose: `DbSet`. Here is what our context class looks like:

```
public class Context : DbContext
{
    public Context()
        : base("name=chapter2")
    {

    }

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

It was mentioned before that `DbContext` needs to be the base class for all our Entity-Framework-based context classes, and you can see this in the preceding example. You also see the constructor that calls the base constructor. We pass the connection string configuration, and in this case, we specify that we will use a connecting string with the key of `chapter2` defined in the application configuration file, which can be `app.config` or `web.config` depending on your application type. Here is how this configuration looks in `app.config` file in our console application:

```
<connectionStrings>
<add name="chapter2"
connectionString="Data Source=.;Initial Catalog=Chapter2;Integrated
Security=SSPI"
providerName="System.Data.SqlClient"/>
</connectionStrings>
```

It is important to notice is that we are using a standard connection string that does not contain anything specific to Entity Framework. This allows you to use the same connection string for other purposes, such as reports. Here is how the same `Context` class looks in VB.NET:

```
Public Class Context
    Inherits DbContext
    Public Sub New()
        MyBase.New("name=chapter2")
    End Sub
    Property People As DbSet(Of Person)
End Class
```

Apply Code[packt] and CodeEnd[packt]style

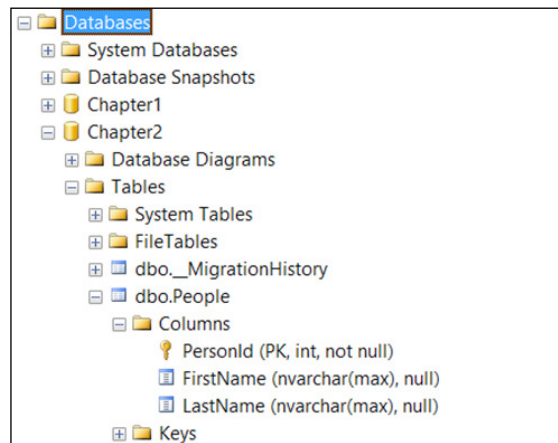
We are actually ready to run our application and see what happens. We need to make sure that the `Chapter2` database specified in the connection string as the initial catalog (database) does not exist in the default instance of SQL Server, specified by "." inside the connection string. You will learn more about database creation in *Chapter 7, Database Migrations and Additional Features*. In this chapter, you will learn one other important thing: The database is created upon the first access to the results of a query or an update/insert operation. We can also create it using the Entity Framework database API. Here is how we do it inside our console application:

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

What you see in the preceding code is the use of the database object that forces the creation of the database if it does not already exist. Alternatively, we can access any data via the Entity Framework API to cause database creation to occur. Here is the `DbContext` definition code in VB.NET:

```
Sub Main()
    Using context = New Context()
        context.Database.CreateIfNotExists()
    End Using
End Sub
```

Finally, we just need to make sure that the connection strings match your computer setup. Now we can run your console application. After we run it, we can open **SQL Server Management Studio (SSMS)** and verify the results of our application. Alternatively, we can use the **Server Explorer** window in Visual Studio. You should see the **Chapter2** database with one table in it, as seen in the following screenshot:



You probably noticed that the first and last name columns are of the type `nvarchar(max)`. This is due to defaults that Entity Framework uses. It makes a lot of assumptions on the developers' behalf when it comes to defining database types and structures. In the preceding example, Entity Framework guessed that the `PersonId` property should map to the primary key. Because it was of the type integer, the column was also set up as the identity column in SQL Server. On top of that, our class name `Person` was pluralized when the table was defined in SQL Server. What this means is that a lot of decisions are made based on conventions in Entity Framework. We will learn a lot more about conventions in subsequent chapters.

Saving a new record to the database

It is time to add some data to our table in the created database. We will insert a row, which is sometimes called a record. A record in the `People` table consists of values in three columns: **PersonId**, **FirstName**, and **LastName**. These values are going to be based on property values of an instance of the `Person` class, which is mapped to the `People` table. This is an important concept to remember. When we create entity classes, their purpose is to map to tables. There are some exceptions to this rule, which we will see in later chapters. This table is represented by a collection-based property in our `Context` class.

This property's type is `DbSet` of `Person` and its name is `People`. Conceptually, you can think of adding objects to that collection to be equivalent to inserting rows into the database's corresponding table. You need to use the `Add` method of `DbSet` to implement the addition of new data. The `DbContext` class has the `SaveChanges` method, which is responsible for committing all the pending changes to the database. It does so by examining the state of all the objects in the context. All such objects are housed within each of the collection properties based on `DbSet` in the context class. In our case, there is only one such collection in the `Context` class: the `People` property. `Context` tracks the state of each one of the objects in all its `DbSet` properties. This state can be "Deleted", "Added", "Modified", or "Unchanged". We can easily decipher how each of the states, with the exception of "Unchanged", will result in a corresponding query sent to the RDMBS. If you want to create multiple rows in a table, you just need to add multiple instances of .NET object based on the class that corresponds to the table in question. The next step is to commit your changes to the database using the `SaveChanges` method. This method runs as a single transaction. As a result, all pending database changes are persisted as a single unit of work, participating in this transaction. If one of the commits fails, the entire batch of changes will be rolled back upon exception. The `DbContext.SaveChanges` method is transactional. This enables you to commit a batch of logically related changes as a single operation, thus ensuring transactional consistency and data integrity.


Let's take a look at the code that adds a row to the `People` table:

```
static void Main(string[] args)
{
    using (var context = new Context())
    {
        context.Database.CreateIfNotExists();
        var person = new Person
        {
            FirstName = "John",
            LastName = "Doe"
        };
        context.People.Add(person);
        context.SaveChanges();
    }
}
```

In this sample code, we are creating a new instance of the `Person` class, populating the first and last names. You will notice that we did not set a value for the `PersonId` property. The reason for this is that this property corresponds to the identity column in SQL Server, which means its value is generated by the database. This value will be automatically populated in the `person` variable's object immediately after the save. You can verify this by setting a breakpoint on the line after the `SaveChanges` call, and checking the value of the `PersonId` property of the `person` variable. Another thing to notice is that the instance of the `Context` class is wrapped inside the `Using` statement. It is important to always follow this coding pattern. `DbContext` implements an `IDisposable` interface. It does so because it contains an instance of `DbConnection` that points to the database specified in the connection string. It is very important to properly dispose of the database connection in Entity Framework, just like it was important in ADO.NET. Here is the same code in VB.NET:

```
Sub Main()  
    Using context = New Context()  
        context.Database.CreateIfNotExists()  
        Dim person = New Person With {  
            .FirstName = "John",  
            .LastName = "Doe"  
        }  
        context.People.Add(person)  
        context.SaveChanges()  
    End Using  
End Sub
```

If you would like to add one more row, just create another instance of the `Person` class and add it to the same `People` collection. To verify that the data was successfully inserted, you can just open SQL Server Management Studio or SQL Server Object Explorer inside Visual Studio, and look at the data in the `People` table. Now, let's see how we can retrieve the data in the database using Entity Framework.

 SQL Server Object Explorer can be found under the View menu in Visual Studio. If you cannot find this window, you may need to install SSDT or SQL Server Data Tools from <https://msdn.microsoft.com/en-us/data/tools.aspx>.

Querying data in a database

In this section, we are going to look at our data using the query capabilities of Entity Framework. Typically, we will use LINQ to do this. We are going to start with a simple example though, accessing the data directly through `DbSet`. We will take a deeper look at LINQ in subsequent chapters. The code is quite simple and is as follows:

```
using (var context = new Context())
{
    var savedPeople = context.People;
}
```

If you set a breakpoint on the line with the last curly brace and look at the `savedPeople` variable in the **Watch** window, you will see one peculiar thing, something called **Results View**, shown in the following screenshot:

Name	Value
savedPeople	(SELECT [Extent1].[PersonId] AS [PersonId], [Extent1].[Firs
base	(SELECT [Extent1].[PersonId] AS [PersonId], [Extent1].[Firs
Local	Count = 0
Non-Public members	
Results View	Expanding the Results View will enumerate the IEnumerable

This illustrates an important concept. Entity Framework is using delayed query execution. In other words, the actual query command is sent to the database when the results of that LINQ query are accessed or enumerated. Entity Framework is doing so based on the `IQueryable` interface that `DbSet` implements. We can enumerate the results of our query using a simple loop, thus causing the SQL execution, for example:

```
using (var context = new Context())
{
    var savedPeople = context.People;
    foreach (var person in savedPeople)
    {
        Console.WriteLine("Last name:{0},first name:{1},id {2}",
            person.LastName, person.FirstName, person.PersonId);
    }
}
Console.ReadKey();
```

If you run the preceding code, you will see the list of `people` in the console window. It will show the people you just added to the database in the previous step. What we do is simply access the entire `DbSet` or table data by pointing our variable to the property of the context that contains the people collection. This is roughly equivalent to the SQL query `SELECT * FROM PEOPLE`. Entity Framework then reads in the results, creating actual instances of the `Person` class, then organizing them into a collection. This process is called *materialization*, that is, the creation of .NET objects from `DbDataReader` that is reading the data from the database. If you only see a single row in the output and would like to see more than a single record, just run your insert code a few more times. The same code in VB.NET looks as follows:

```
Using context = New Context()  
    Dim savedPeople = context.People  
    For Each person In savedPeople  
        Console.WriteLine("Last name:{0},first name:{1},id {2}",  
            person.LastName, person.FirstName, person.PersonId)  
    Next  
End Using
```

To summarize what we have seen in the preceding code, we insert rows into the database by simply adding objects to a collection that corresponds to the table we are targeting.

Updating a record

Let's take a look at how we can change the data after inserting it. This is done in the SQL world by issuing an `UPDATE` command. In the Entity Framework world, you do not need to perform this step. Instead, we just need to find an instance of an object in the collection, change its properties, and then call the familiar `SaveChanges` method. Now, we just need to get an object from the database to update. You just saw how to do this in the *Querying data in a database* section. Here is what the update code looks like:

```
using (var context = new Context())  
{  
    var savedPeople = context.People;  
    if (savedPeople.Any())  
    {  
        var person = savedPeople.First();  
        person.FirstName = "Johnny";  
        person.LastName = "Benson";  
        context.SaveChanges();  
    }  
}
```

As you can see, we simply point to the `People` property of the context. Then, we check to make sure that there is at least one entity in the collection using the `Any()` method, which is part of LINQ. Then, we get the first object in the collection using the `First()` method. We could have just as easily pointed to any other object in the collection. After this, we set two properties of the found `Person` object to some new values. Finally, we issue `SaveChange()` just like in the example of the insert operation. If you run this code while SQL Server Profiler is running, you will see the SQL queries that Entity Framework creates and issues in conjunction with the SQL Server Entity Framework provider. Entity Framework maintains the state of changed objects and is responsible for generating appropriate update queries. Here is how the same code looks in VB.NET:

```
Using context = New Context()
    Dim savedPeople = context.People
    If savedPeople.Any() Then
        With savedPeople.First()
            .FirstName = "Johnny"
            .LastName = "Benson"
        End With
        context.SaveChanges()
    End If
End Using
```

Deleting a row from the database

Now let's try to delete a record from the database. First of all, we need to find a row to delete. If you look at the update example, you will see exactly how you can do this. In this example, we will employ a slightly different technique, finding a row by its primary key. In our example, it is the `PersonId` property's value. Just find a value to delete by running the code from a query example and writing down the appropriate value of the `PersonId` property. Once we have this value, we can use the `Find` method of `DbSet` to locate the correct object. Finally, we will mark the object as deleted using the `DbContext` API's `Remove` method, as shown in the following code:

```
using (var context = new Context())
{
    var personId = 2;
    var person = context.People.Find(personId);
    if (person != null)
    {
        context.People.Remove(person);
        context.SaveChanges();
    }
}
```

You will notice that we also have a check to make sure that the `Find` operation was successful in checking its results, making sure that the object retrieved is not null. The `delete` operation is performed by calling the `Remove` method on `DbSet`. As we saw in the preceding example, we always need to call `SaveChanges` when we want to persist our modifications to the database. These modifications can include update or insert, or delete in this case. In subsequent chapters, we will see other API calls that perform the same tasks. Here is how this code looks in VB.NET:

```
Using context = New Context()  
    Dim personId As Integer = 4  
    Dim person = context.People.Find(personId)  
    If person IsNot Nothing Then  
        context.People.Remove(person)  
        context.SaveChanges()  
    End If  
End Using
```

The value 4 in the preceding example contains the primary key value for a row in the `Person` table and does not carry any specific meaning beyond that.

Introduction to schema changes

It is good for everyone to experiment with his or her first application. However, it is very likely that we will encounter an exception if we make changes to the `Person` class or add another collection to the `Context` class. Let's take a look at what happens when we add more classes and properties to the context. In this example, we are going to create a `Company` class and add it the context as a collection. Here is another simple class that represents a second table in our database:

```
public class Company  
{  
    public int CompanyId { get; set; }  
    public string Name { get; set; }  
}
```

Here is how our context class definition looks after the addition of the new collection:

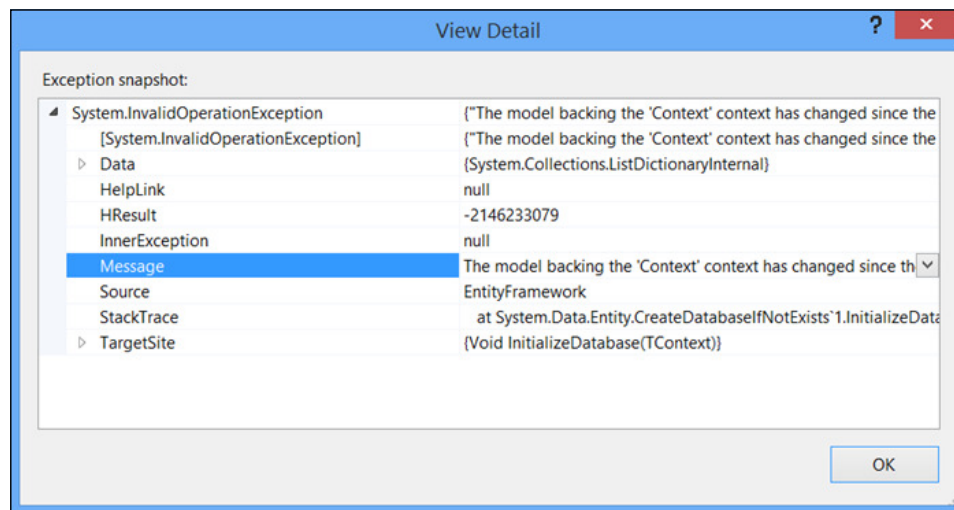
```
public class Context : DbContext  
{  
    public Context()  
        : base("name=chapter2")  
    {  
    }  
}
```

```
public DbSet<Person> People { get; set; }  
public DbSet<Company> Companies { get; set; }  
}
```

This code illustrates an important concept. We can now see that our context represents the entire database, consisting of multiple tables. Each one becomes a property on our context class. Here is how the same code looks in VB.NET:

```
Public Class Company  
    Property CompanyId() As Integer  
    Property Name() As String  
End Class  
  
Public Class Context  
    Inherits DbContext  
    Public Sub New()  
        MyBase.New("name=chapter2")  
    End Sub  
    Property People As DbSet(Of Person)  
    Property Companies() As DbSet(of Company)  
End Class
```

Now if we try to run the project and enumerate the `Companies` collection by accessing `context.Companies` inside a `for each` loop, we will get an exception, as seen in the following screenshot:



The entire error text is **The model backing the 'Context' context has changed since the database was created. Consider using Code First Migrations to update the database (<http://go.microsoft.com/fwlink/?LinkId=238269>)**. Sometimes you will get a different error message that states that a specific table does not exist in the database. The point, however, is the same—you cannot change the context, that is the database schema without adding code to handle this situation. You will learn a lot more about this circumstance in *Chapter 7, Migrating Databases and Existing Databases* on migrations, but let's address the immediate need to handle schema changes in our Entity Framework experiments. This is where the concept of initializers come in. Initializers are run when Entity Framework accesses the database for the first time during the instantiation process or during the first access of data. There are three initializers in Entity Framework that we need to be concerned with right now:

- `CreateDatabaseIfNotExists<TContext>`
- `DropCreateDatabaseIfModelChanges<TContext>`
- `DropCreateDatabaseAlways<TContext>`

When it comes to VB.NET, we use slightly different syntax, for example `CreateDatabaseIfNotExists(Of TContext)`. If we look at the names of these classes, it quite clear what these initializers do. `CreateDatabaseIfNotExistsinitializer` is the default that is run when you do not specify another one. It will check whether the database exists, and if not, create it along with the structure specified by the context and classes that it refers to in its properties. The second initializer in the list recreates the database when the model specified in the context class changes. This could be caused by changes to any class that maps to a table, as well as the addition or removal of collections from the context. Finally, the last initializer always recreates the database; in other words, every run of the software that uses it will result in a new database. Let's create and use the second one, as shown: `DropCreateDatabaseIfModelChanges`:

```
public class Initializer : DropCreateDatabaseIfModelChanges<Context>
{
}
```

In order to use this new initializer, we must let Entity Framework know before we create an instance of our context for the first time in our application. We can do so by accessing the static method `SetInitializer` on the `Database` class, which is also part of the Entity Framework API. In our console application, we need to do it in the first line of code that is executed in our application, for example:

```
static void Main(string[] args)
{
    Database.SetInitializer(new Initializer());
    // more code follows
}
```

All we need to do is to create a new instance of the initializer and set it on the database object. Now, if we run our application again, we will not see the exception. Instead, our database will be recreated with the new structure.



If you have the database in question open inside another application, such as SQL Server Management Studio, you will get a different exception, informing you that Entity Framework cannot obtain an exclusive lock on the database in order to drop it. Just close all other applications and you will be able to proceed.

Here is how the same code looks in VB.NET:

```
Public Class Initializer
    Inherits DropCreateDatabaseIfModelChanges(Of Context)
End Class

Sub Main()
    Database.SetInitializer(New Initializer)
    ' more code follows
```


There is one more important note to make here. Because the initializers we mentioned drop databases, you will lose all of the data you accumulated in the database. Obviously, this makes the initializer we just used unsuitable for production purposes. However, these initializers come in quite handy when you are learning Entity Framework and early in your projects' lifetimes, that is, during the rapid prototyping phase. We can also call `Database.SetInitializer` and pass in null (*nothing* in VB.NET) instead of the actual instance. This will override the default behavior and always throw an exception if your class-based model/context does not match the database any longer. This will include when the database does not exist. Initializers have one more interesting feature we want to look at. They allow you to run the code after the target database is created. You can do so by overwriting the `Seed` method. This method takes one parameter, which is an instance of your `Context` class, for example in the following code:

```
public class Initializer : DropCreateDatabaseIfModelChanges<Context>
{
    protected override void Seed(Context context)
    {
        context.Companies.Add(new Company
        {
            Name = "My company"
        });
    }
}
```

As you can see, I am using familiar code to add a company object to the `Companies` collection of my `context`. Unlike the standard addition code, I do not need to call `SaveChanges`, although there is no harm in doing so. Here is how the same code looks in VB.NET:

```
Public Class Initializer
    Inherits DropCreateDatabaseIfModelChanges(Of Context)

    Protected Overrides Sub Seed(ByVal context As Context)
        context.Companies.Add(New Company() With {
            .Name = "My company"
        })
    End Sub
End Class
```

 We can use Entity Framework migrations in order to update the production database.

I hope you will now take some time to experiment with what you have learned in this chapter.

Self-test questions

Q1. What base class can be used to represent a table in a database inside the `DbContext` collection's property?

1. `List<T>/List (of T)`
2. `DbSet<T>/DbSet (of T)`
3. `ICollection<T>/ICollection(of T)`

Q2. You do not have to call `Dispose` on `DbContext` after use, true or false?

Q3. Which method can be used to locate a row in the database using the primary key in Entity Framework?

1. Find
2. Locate
3. Define

Q4. Which method of `DbSet` can you use after finding a record to delete it?

1. Delete
2. Remove
3. Erase

Q5. You want to easily update the last name of a person in a record stored in the database. You can do so in Entity Framework by:

1. Issuing a SQL command
2. Getting the corresponding object, setting the `LastName` property, and calling `SaveChanges`
3. Creating an instance with the same id and different values for `LastName`, then adding it to `DbSet` using the `Add` method, and then calling `SaveChanges`

Q6. You have changed a class that is mapped to a table by adding another property to it. What happens if you set the database initializer to null and run the program?

1. All other columns' data is shown
2. An exception is thrown
3. The database is changed to match the new schema, but the data is lost

Summary

In this chapter, we created our first Entity Framework-based application. We started by creating a new console application .NET project. We then added the Entity Framework reference using NuGet. Then, we decided what data we wanted to store in the database and created a class that maps to a table in the database, that is, the `Person` class. Then, we created our database abstraction, the `Context` class, inheriting from `DbContext`. We specified the desired connection string in its constructor and added this connection string to the application configuration file. Then, we added a single property to our context, `People`, which was a collection of `Person` objects, of the type `DbSet of Person`. At this point, we ran our application. We observed that a database was created with a single table, based on the this property. The database creation process used many conventions, including the table name and making the `PersonId` column unique (by `identity`) and primary key.

We then worked on adding a row to the preceding table. We created an instance of the `Person` class, setting some properties on it at the same time. We created an instance of `Context`, following the `IDisposable` pattern. We then added the instance of `Person` to the collection specified by the `People` property of `Context` using the `Add` method. Finally, we called `SaveChanges` to commit our in-memory objects to the database, thus making them rows in the `Person` table. Updating the inserted data was easy, as well. We queried the rows from the database by enumerating the objects inside the `People` property of `Context`. We picked a row, changed its properties, and called `SaveChanges` to update the data. Deletion was done using the `Find` method instead of enumerating a query, and then marking the object for deletion by calling `Remove` method of `DbSet` and calling `SaveChanges` yet again, thus deleting the data from the database. We finished the chapter by mentioning that Entity Framework has many ways to perform some actions, and this includes CRUD operations. We will see other ways to archive these tasks in subsequent chapters. Congratulations on getting your first app working!

3

Defining the Database Structure

In this chapter, you will learn how to specify the details of our database structure by using the Entity Framework API. We will build on what you have learned in *Chapter 2, Your First Entity Framework Application*, and write entity classes that define types of columns in destination tables. We will discover how to specify a relationship between tables in your database, through properties in entity classes and the configuration API. We will look at various ways to configure table structures. We will also see how .NET types map to SQL Server column types.

In this chapter, we will cover how to:

- Create classes that define a table structure using the simple and primitive types
- Handle nullable and required properties
- Define attributes and configuration classes, as well as use the model builder API to specify column types
- Specify One-to-One, One-to-Many, and Many-to-Many relationships between classes

Creating table structures

Let's consider all the structures that can be created.

Mapping .NET types to SQL types

Before we start, it would be helpful to take a look at the mappings between .NET types and SQL Server column types. You remember that there is a distinct mismatch between the two, which is one of the problems that Entity Framework strives to fix. You can also find similar mappings between .NET and other RDBMSes, such as Oracle. In this book, we will concentrate on SQL Server. It is not always important to keep these mappings in mind. For example, if you define a property in .NET as `integer`, you can safely assume that Entity Framework will handle the column's definition and use the appropriate type; for example, `int` in SQL Server. Here are the mappings for the most commonly used .NET types:

SQL Server Database type	.NET Framework type
<code>bigint</code>	<code>Int64</code>
<code>binary</code> , <code>varbinary</code>	<code>Byte[]</code>
<code>bit</code>	<code>Boolean</code>
<code>date</code> , <code>datetime</code> , <code>datetime2</code> , <code>smalldatetime</code>	<code>DateTime</code>
<code>datetimeoffset</code>	<code>DateTimeOffset</code>
<code>decimal</code> , <code>money</code> , <code>smallmoney</code> , <code>numeric</code>	<code>Decimal</code>
<code>float</code>	<code>Double</code>
<code>int</code>	<code>Int32</code>
<code>nchar</code> , <code>nvarchar</code> , <code>char</code> , <code>varchar</code>	<code>String</code>
<code>real</code>	<code>Single</code>
<code>rowversion</code> , <code>timestamp</code>	<code>Byte[]</code>
<code>smallint</code>	<code>Int16</code>
<code>time</code>	<code>TimeSpan</code>
<code>tinyint</code>	<code>Byte</code>
<code>uniqueidentifier</code>	<code>Guid</code>

You can view the complete list on the MSDN website at [http://msdn.microsoft.com/en-us/library/cc716729\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/cc716729(v=vs.110).aspx). If you want to see the mapping for other database engines, you can easily find them on the Internet. For example, you can find Oracle mappings at [http://msdn.microsoft.com/en-us/library/cc716726\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/cc716726(v=vs.110).aspx).

Configuring primitive properties

Let's start the discussion by looking at string properties. You noticed that SQL Server has many types that map to the string type in .NET. The same is the case with other major RDBMSes. Unlike numeric types, it is actually important to decide how you want to store string-based information in the database. The reason is that most relational database management engines have multiple character storage types. They usually have character types that start with the letter N. This letter signifies that the data to be stored in such columns is **Unicode** data, based on character sets used to store each character in the double-byte format. These types are necessary to store data using the most non-Latin-based languages, such as Chinese. So, if you are writing an application for US-based users who only use English, you can define your string data as `varchar` or `char` instead of `nvarchar` or `nchar`, to save physical storage and possibly speed up queries. You can also use fixed length or variable length character columns, signified by the presence or absence of `var` in the type name. Hence, you need to make some decisions before writing the code. Now, let's look at examples. Let's start with an example of the `Person` class again. Let's presume that the first and last name properties need to be of variable length and accommodate, at most, 30 characters each. The middle name is of a fixed length of one character.

There are a few ways to configure database structures in Entity Framework. You can use the following:

- Attributes
- Configuration Classes
- The `DbModelBuilder` API

Let's start with attributes. The property-defining attributes are part of .NET and live in the `System.ComponentModel.DataAnnotations` namespace. Here is how we will configure the `Person` class's string properties using attributes:

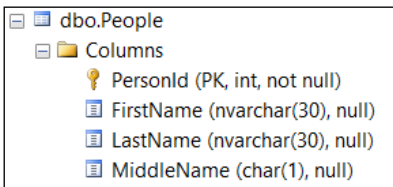
```
public class Person
{
    public int PersonId { get; set; }

    [MaxLength(30)]
    public string FirstName { get; set; }
```

```
[MaxLength(30)]
public string LastName { get; set; }

[StringLength(1, MinimumLength = 1)]
[Column(TypeName = "char")]
public string MiddleName { get; set; }
}
```

We are using a few different attributes to fulfill our goal. We are using the `MaxLength` attributes to configure two properties, specifying the maximum length. We are using `StringLength` to specify both minimum and maximum length for the middle name property. We are also using the `Column` attribute to force the `char` type for the middle name column. Entity Framework will use the Unicode storage by convention, unless explicitly specified otherwise. If we run the code using the preceding configuration, we will end up with exactly the structure we were aiming for, as you can see in the following screenshot:



Here is how this code looks in VB.NET:

```
Public Class Person
    Property PersonId() As Integer

    <MaxLength(30)>
        Property FirstName() As String

    <MaxLength(30)>
        Property LastName() As String

    <StringLength(1, MinimumLength := 1)>
    <Column(TypeName := "char")>
        Property MiddleName() As String
End Class
```

There is one important thing to know about data annotation attributes. In addition to specifying the data structure, they will also be used as validation attributes. If you, for example, attempt to add a person object with a middle name longer than one character and save this data, you will get an exception before the actual SQL query is constructed and sent to the database. The message details will contain information related to the violated rule(s).

You probably spotted a small problem with the column attribute. We are essentially hardcoding the column type. What would happen if we were to run our project against another RDBMS that uses type names that differ from SQL Server? Entity classes that map to tables should ideally be agnostic to persistence, but in our cases they carry some RDBMS-specific information. In addition to this, we currently do not control the message text for the errors that would occur if we violate the validation rules we specified. So, to deal with this issue, we need to add the error message, as shown in the following lines of code:


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

The same attribute in VB.NET looks as follows:

```
<MaxLength(30, ErrorMessage:="First name cannot be longer than 30")>
Property FirstName() As String
```

Data annotations support localization, so we do not need to hardcode the error message in English. Finally, some situations cannot be handled with annotations at all, such as delete rules for relationships. Luckily, Microsoft provides multiple ways to configure table structures and column types. One way is to use the `DbModelBuilder` API. To get to this API, we need to override the `OnModelCreating` method of our `DbContext` object. Then, we can configure properties for an entity class, `Person` in our case, as shown in the following code:

```
public class Context : DbContext
{
    public Context()
        : base("name=chapter2")
    {
    }
    public DbSet<Person> People { get; set; }
    protected override void OnModelCreating(DbModelBuilder
modelBuilder)
    {
        modelBuilder.Entity<Person>().Property(p =>p.FirstName)
            .HasMaxLength(30);
        modelBuilder.Entity<Person>().Property(p =>p.LastName)
            .HasMaxLength(30);
        modelBuilder.Entity<Person>().Property(p =>p.MiddleName)
            .HasMaxLength(1)
            .IsFixedLength()
            .IsUnicode(false);
    }
}
```

 The Entity class refers to a class that maps to a table in the database.

In the preceding code, we get an instance of the `EntityTypeConfiguration` class from the context by providing the type of entity we want to configure, `Person` in our case. Then, we configure one property of the `Person` class at a time, using the `Property` method. If we follow this coding practice, we do not need to use data annotation attributes any longer. You will notice the parity between attributes we used in the prior example and methods exposed on the `StringPropertyConfiguration` class. In the case of the last name, we simply configure the maximum length. Since the default for string properties is `Unicode`, we did not need to configure the `Unicode` setting explicitly. In the case of the middle name, we did specify that it is a non-Unicode column. Additionally, we specified that the middle name is a fixed-width column. As a result, we did not have to hardcode the char type for the `MiddleName` property. So, you now see that the configuration API has an advantage over data annotations in this case.

There is one problem with the preceding code that you undoubtedly noticed. For one table with three columns, the `OnModelCreating` method is pretty short. What if we have 1,000 tables? The preceding approach will result in unmanageable code as the number of entities grows. Entity Framework provides a way to break this code apart using a separate configuration class for each entity class. We will call our configuration *buddy* class for `Person`, class `PersonMap`, as shown in the following code:

```
public class PersonMap : EntityTypeConfiguration<Person>
{
    public PersonMap()
    {
        Property(p => p.FirstName)
            .HasMaxLength(30);
        Property(p => p.LastName)
            .HasMaxLength(30);
        Property(p => p.MiddleName)
            .HasMaxLength(1)
            .IsFixedLength()
            .IsUnicode(false);
    }
}
```

The code is virtually identical to the one we wrote using model builder, but it is now much better organized to handle multiple entities. You also saw that we chain multiple methods for a single property together. This chaining of methods is called the Fluent API. The same code in VB.NET looks as follows:


```

Public Class PersonMap
    Inherits EntityTypeConfiguration(Of Person)

    Public Sub New()
        Me.Property(Function(p) p.FirstName) _
            .HasMaxLength(30)
        Me.Property(Function(p) p.LastName) _
            .HasMaxLength(30)
        Me.Property(Function(p) p.MiddleName) _
            .HasMaxLength(1).IsFixedLength().IsUnicode(False)
    End Sub
End Class

```

Finally, we need to tell our `Context` class that we have a configuration class for it to use during the database structure generation. You can do so in the familiar `OnModelCreating` method. We need to add an instance of our `Configuration` class to `modelBuilder`'s `Configuration` collection, as shown in the following code:

```

protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Configurations.Add(new PersonMap());
}

```

We now successfully configured string properties for the `Person` class, thereby configuring the `People` table in our database. We used the entity configuration class tied to the `Person` class via the generic type parameter. The same code in VB.NET looks as follows:

```

Protected Overrides Sub OnModelCreating(ByValmodelBuilder As
    DbModelBuilder)
    modelBuilder.Configurations.Add(New PersonMap)
End Sub

```

In the preceding example we used `StringPropertyConfiguration` class. It was used to configure string properties. This class inherits from `LengthPropertyConfiguration`, which in turn inherits from `PrimitivePropertyConfiguration`. These classes support definition for columns that correspond to primitive properties. What primitive properties does Entity Framework support in this fashion? Here is the short list, as follows:

- `DateTimePropertyConfiguration`
- `DecimalPropertyConfiguration`
- `BinaryPropertyConfiguration`
- `StringPropertyConfiguration`

Let's add more properties to the `Person` class to exercise various configurations, and become more familiar with the available API, in addition to using other property types, such as integer. Here are the updated `Person` and `PersonMap` classes:

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string MiddleName { get; set; }
    public DateTime BirthDate { get; set; }
    public decimal HeightInFeet { get; set; }
    public byte[] Photo { get; set; }
    public bool IsActive { get; set; }
    public int NumberOfCars { get; set; }
}

public class PersonMap()
{
    Property(p =>p.FirstName)
        .HasMaxLength(30);
    Property(p =>p.LastName)
        .HasMaxLength(30);
    Property(p =>p.MiddleName)
        .HasMaxLength(1)
        .IsFixedLength()
        .IsUnicode(false);
    Property(p =>p.HeightInFeet)
        .HasPrecision(4, 2);
    Property(p =>p.Photo)
        .IsVariableLength()
        .HasMaxLength(4000);
}
```

Different methods are available on various property types configuration classes, based on what is applicable at the database level. For example, decimal columns support precision and scale and the number of digits before and after decimal points, so we can configure these for decimal properties. Only the length makes sense for binary properties, so we specify that we can store an image up to 4,000 bytes in the `Photo` column. For other properties, such as integer and Boolean, there is really nothing to configure, hence just specifying property type is sufficient. Here is what the updated database structure looks like:



Handling nullable properties

As you noticed, some columns are nullable and some are not. Entity Framework decides if a column is nullable based on the property type by convention. For example, the string type allows null values, hence matching character-based columns are nullable. On the other hand, `datetime` and `int` variables cannot be set to null in .NET, hence these columns are non-nullable. What should we do if we want to make these columns nullable, or make string storing columns to be filled compulsory? There are two ways to accomplish this as well. You can just use nullable types to make columns nullable, and this is by far the easiest way. For example, to allow a blank `BirthDate` value, we can change the property declaration to the following:

```
public DateTime? BirthDate { get; set; }
```

On the other hand, if we want to make sure that the first name is required, we can add code to the configuration class, as shown:

```
Property(p =>p.FirstName)
    .HasMaxLength(30)
    .IsRequired();
```

There is an opposite of `IsRequired` on property configuration classes; the `IsOptional` method. This method allows you do make non-nullable types to be nullable columns in the database. It is better to just use nullable types, as it makes it easier to leave property values as null in .NET instead of trying to, for example, convert specified values to null for storage in the database.

Here is how the same code looks in VB.NET:

```
Public Class Person
    Property PersonId() As Integer
    Property FirstName() As String
    Property LastName() As String
    Property MiddleName() As String
```

```
Property BirthDate() As DateTime?
Property HeightInFeet() As Decimal
Property Photo() As Byte()
Property IsActive() As Boolean
Property NumberOfCars() As Integer
End Class
Public Class PersonMap
    Inherits EntityTypeConfiguration(Of Person)

    Public Sub New()
        Me.Property(Function(p) p.FirstName) _
            .HasMaxLength(30) _
            .IsRequired()
        Me.Property(Function(p) p.LastName) _
            .HasMaxLength(30) _
            .IsRequired()
        Me.Property(Function(p) p.MiddleName) _
            .HasMaxLength(1) .IsFixedLength() .IsUnicode(False)
        Me.Property(Function(p) p.BirthDate) _
            .HasPrecision(1)
        Me.Property(Function(p) p.HeightInFeet) _
            .HasPrecision(4, 2)
        Me.Property(Function(p) p.Photo) _
            .IsVariableLength() .HasMaxLength(4000)
    End Sub
End Class
```

It is useful at this point to experiment with simple properties to get a feel of Entity Framework's configuration approach, and study the types not covered in the preceding example. It is important to notice that some of the SQL Server types shown in .NET to SQL Server mapping in the preceding table are specific to SQL Server and may not be available in other relational database systems. As a result, you will not find the configuration API to support such columns. The answer to this problem is to use the `HasColumnType` method on property-configuring classes, and specify the type you want to use explicitly by name. If you want to generically support multiple database engines, just write a helper class for this purpose that will return the appropriate database type as string, based on the currently configured database engine. You cannot use this approach for attributes because parameters must be constants. All primitive property configuration classes share two more methods, `HasColumnName` and `HasColumnOrder`. The latter allows you to precisely control the ordinal position of a column in the table. The former allows you to have a column name different from the property name. If they are the same, you do not need to use explicit column names.

However, if you are dealing with a legacy database with short column names for example, the ability to rename properties that map to such columns comes in quite handy, as it increases the code readability. On the other hand, you do not have to have Entity Framework control your table structure. You can use Entity Framework Code-First to talk to an existing database just as well. Still, you have to have correct mapping between properties in classes and columns in tables regardless of whether you update the database structure yourself, or you let Entity Framework do it.

Defining relationships

Now let's take a look at relationships between classes, and consequently between tables. There are three main types of relationships in database theory:

- One-to-Many
- One-to-One
- Many-to-Many

First of all, let's define what a relationship is. It is defined based on how two or more objects relate to each other. It is identified by the multiplicity value on both ends of the relationship. For example, One-to-Many means that on one end of a relationship, sometimes called the parent, we only have one entity. On the other end of the relationship, we can have multiple entities, sometimes called children. The Entity Framework API refers to those ends as principal and dependent, respectively. The One-to-Many relationship has a slight variation, called One-or-Zero-to-Many, which means that a child may or may not have a parent. The One-to-One relationship has variants as well, where either end of a relationship is optional. Let's take a look at them separately.

The One-to-Many relationship

Let's model an example to get a clear understanding of the One-to-Many relationship. Let's say that in our sample project, each person from earlier in this chapter can have multiple phone numbers. Hence, our next object would be a phone. We are going to create a very simple object with an identifier and a phone number, as follows:

```
public class Phone
{
    public int PhoneId { get; set; }
    public string PhoneNumber { get; set; }
}
```

We also need to add a property onto the context using the type `DbSet` of `Phone` so that we can have access to phone numbers. You have seen this code many times already. Each person can have many phone numbers. This just screams collection of phone numbers on the `person` object, right? So, we will just add a new property to the `Person` class. We will use `ICollection` of `Phone` instead of a specific type of collection. This will work just fine, and in fact, you can let Entity Framework define the collections for you. Here is what the new property definition looks like:

```
public virtual ICollection<Phone> Phones { get; set; }
```

In order to avoid potential null reference exception possibility, we want to create a new instance of collection when a `Person` object is created. We will use `HashSet` of the `T` collection type, as shown in the following code:

```
public Person()  
{  
    Phones = new HashSet<Phone>();  
}
```

You will notice that we use the **virtual** keyword (**Overridable** in VB.NET) when defining the property. This keyword will enable us to use lazy loading when looking for phone numbers for a person. What this means is that Entity Framework will actually dynamically load phone objects into the collection from the database on demand, at the time you attempt to access the phone's property. This is a new concept called **lazy loading**, which we will cover in more detail in subsequent chapters. It is so called because Entity Framework will initially not issue a query to populate phone numbers, but instead will load the data when code asks for it. There is an alternate approach to loading related data called **eager loading**. With this approach, the phone numbers would be proactively loaded prior to access to the phone's property. For now, let's assume that we want to take advantage of the lazy loading functionality, hence the **virtual** keyword. Interestingly enough, there is no person identifier contained in the `Phone` class. This is something that you must do as a database developer. However, in the Entity Framework world, you have the flexibility to omit this property. The reason is that we may not have a legitimate business reason to know person ID when looking at phones. In our case, we want to know about the phone number only in the context of a person and never as a standalone object. Let's convince ourselves that this will actually work, by running the app after adding a new person with some phone numbers to our app and saving them as a batch. Once we do this, we will take a look at the structure that is created.

Here is what the same code in VB.NET looks like:

```
Public Class Phone  
    Property PhoneId() As Integer
```

```
        Property PhoneNumber() As String
    End Class

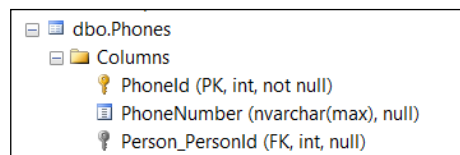
    Overridable Property Phones() As ICollection(Of Phone)

    Public Sub New
        Phones = new HashSet(Of Phone)
    End Sub
```

Now, let's write a few more lines of code to actually add a person with phone numbers to our database. We already saw how to add and save a new person; now we just need to add some phone numbers, as shown in the following code:

```
using (var context = new Context())
{
    var person = new Person
    {
        LastName = "Doe",
        FirstName = "John",
        IsActive = true
    };
    person.Phones.Add(new Phone { PhoneNumber = "123-446-7890" });
    person.Phones.Add(new Phone { PhoneNumber = "123-446-7891" });
    context.People.Add(person);
    context.SaveChanges();
}
```

The preceding code is not really specific to Entity Framework, with the exception of context method calls. We simply write the basic object-oriented code. We create an instance of a class and then add a few more instances of another class to a collection. Entity Framework contains all the magic required to create the appropriate database structure, as well as convert object operations to database queries. To convince yourself that this worked, simply open SSMS and look at the data and table structure, as shown in the following screenshot:



Entity Framework uses a naming convention to define **PersonId** in the `Phone` table. If you do not like this naming convention, you can easily add some code to our mapping class of the person object to correct the issue. Just add a property called `PersonId` to the `Phone` class, which is what we want to name the column. Then, we need to configure the person object and teach it that it has many phones, each containing a link back to the person in the `PersonId` property. Here is what this code looks like in our `PersonMap` class:

```
HasMany(p =>p.Phones)
    .IsRequired()
    .HasForeignKey(ph =>ph.PersonId);
```

This code is quite typical for relationship definitions. The `HasMany` method tells Entity Framework that there is a One-to-Many relationship between the `Person` and `Phone` classes. The `IsRequired` method specifies that the `Person` link on `Phones` is required. In other words, the phone is not a standalone object and must be linked to a person. Finally, the `HasForeignKey` method identifies which property serves as this link. Consequently, Entity Framework will use `PersonId` as the column name. We do not need to populate this new property at all; our code will still work fine, even with these new changes.

We should also take look at an additional use case for the One-to-Many configuration. This use case arises when we have a lookup property on a main entity that points to another entity. Lookup properties point to a full parent of a child entity. Such properties are useful when you need access to parent information while manipulating or examining a child record. For example, let's add a person type to the person entity in the previous example. Essentially, this example is still One-to-Many, but the approach is slightly different. In this case, you would typically use a dropdown control on the main entity edit screen that contains values from a lookup parent table. The lookup table in our example is very simple; it just contains ID and name columns. We are also going to make this relationship optional to illustrate how to add nullable foreign keys. Hence, the new property `PersonTypeId` in the person class must be nullable. Since we are going to make all primary keys be identity columns, we will use the nullable integer type to define this property. We will also need to add another property, using actual `PersonType` in order to configure the relationship, as shown in the following code:

```
public int? PersonTypeId { get; set; }
public virtual PersonTypePersonType { get; set; }
```

Conversely, we will need to add a collection of people to the `PersonType` class to signify that there could be many people for each type, as shown in the following code:

```
public class PersonType
{
```



```

        public int PersonTypeId { get; set; }
        public string TypeName { get; set; }
        public virtual ICollection<Person> Persons { get; set; }
    }

```

The same code in VB.NET looks as follows:

```

Property PersonTypeId() As Integer?
Overridable Property PersonType() As PersonType

Public Class PersonType
    Property PersonTypeId() As Integer
    Property TypeName() As String
    Property Persons() As ICollection(Of Person)
End Class

```

When it comes to relationships, you can configure them from either side of the relationship, principal or dependent. So, let's create a new `PersonTypeMap` class that will serve this purpose, as shown in the following code:

```

public class PersonTypeMap : EntityTypeConfiguration<PersonType>
{
    public PersonTypeMap()
    {
        HasMany(pt =>pt.Persons)
        .WithOptional(p =>p.PersonType)
        .HasForeignKey(p =>p.PersonTypeId)
        .WillCascadeOnDelete(false);
    }
}

```

We use the `WithOptional` method to signify that the foreign key constraint will be nullable. We also specify the delete rule for the constraint, using the `WillCascadeOnDelete` method. Most database engines support multiple actions for delete rules for foreign key relationship constraints. These rules specify what happens when a parent is deleted. You can set the foreign key column to null, that is, do nothing, thus creating an error if child rows exist or delete all related dependents. Entity Framework allows developers to either delete all child rows or do nothing. There are database administrators who object to the use of cascade delete rules because some engines do not provide adequate logging when this happens. Here is how the code looks in VB.NET:

```

Public Class PersonTypeMap
    Inherits EntityTypeConfiguration(Of PersonType)
    Public Sub New()

```

```
        HasMany(Function(pt) pt.Persons) _  
            .WithOptional(Function(p) p.PersonType) _  
            .HasForeignKey(Function(pt) pt.PersonTypeId) _  
            .WillCascadeOnDelete(False)  
    End Sub  
End Class
```


We must remember to update `DbContext` (the `Context` class) and add a new `DbSet` to it, this time of the type `PersonType`, as well as adding our new `PersonType` configuration class to the configurations collection on the model builder. As an alternative to calling `WillCascadeOnDelete`, you can globally remove the convention from the model builder and turn off the cascade delete rule for the entire database model in the `OnModelCreating` method of the context. Conventions in Entity Framework allow developers to perform certain configuration actions globally, in all entity classes within a context. We will discuss conventions in more detail in a later chapter. The following specific convention deals with relationships only:

```
protected override void OnModelCreating(  
    DbModelBuilder modelBuilder)  
{  
    modelBuilder.Conventions  
        .Remove<OneToManyCascadeDeleteConvention>();  
    modelBuilder.Conventions  
        .Remove<ManyToManyCascadeDeleteConvention>();  
}
```

The code in VB.NET is identical, as shown here:

```
Protected Overrides Sub OnModelCreating(ByValmodelBuilder As  
    DbModelBuilder)  
    modelBuilder.Conventions.Remove(Of  
        OneToManyCascadeDeleteConvention)()  
    modelBuilder.Conventions.Remove(Of  
        ManyToManyCascadeDeleteConvention)()  
End Sub
```

We actually remove two conventions that control delete behavior – one for One-to-Many, the other for Many-to-Many relationships.

 Entity Framework contains many conventions you can remove if necessary.

The Many-to-Many relationship

The Many-to-Many relationship is used when you have multiple entities on both ends of the relationship. For example, one person can work for multiple companies, and each company can employ multiple people. At the database layer, this relationship will be defined in the so-called **junction** table, sometimes also called a cross reference table. This table will contain primary key columns from tables on both ends of the relationship. There are two use cases for this type of relationship that matter to us. A junction table can have no additional values or columns, or it can have additional data. If a junction table has no other data, we technically do not need to have this table represented in the object model at all. Let's code this situation. We will add a new class to model the company and add code to the person map to specify the relationship. Just like in the One-to-Many relationship, we will have a property that is a collection of related entities. We will add new collections to both `Person` and `Company` classes, using the related entity as the type of the collection. Here is how the `Company` class looks:

```
public class Company
{
    public int CompanyId { get; set; }
    public string CompanyName { get; set; }
    public virtual ICollection<Person> Persons { get; set; }
}
```

Now, we just need to add code to the entity type configuration class that defines the person table to specify the other end of the relationship. The code will exist in the `PersonMap` class, as shown in this example:

```
HasMany(p =>p.Companies)
    .WithMany(c =>c.Persons)
    .Map(m =>
    {
        m.MapLeftKey("PersonId");
        m.MapRightKey("CompanyId");
    });
```

Technically speaking, this configuration is optional if you are okay with Entity Framework generating names for columns. What we mean is that Entity Framework will actually create a junction table solely based on classes and properties that define both ends of the relationship, because they have collection properties for related entities. Since we want something different from the default, we specify column names explicitly in the junction table. The same code in VB.NET looks like the following:

```
HasMany(Function(p) p.Companies) _
    .WithMany(Function(c) c.Persons) _
```

```
.Map (Sub (m)
  m.MapLeftKey ("PersonId")
  m.MapRightKey ("CompanyId")
End Sub)
```

If you look at the database structure, you will find our junction table, called `PersonCompanies` in the database. Feel free to take out this configuration code and regenerate the database to see what the default naming convention is.

What if our junction table needs to hold more data? For example, we want to add hire date for each person, as the date they start working for a company. In this case, we will actually need to add a class to model the junction table. We can call it `PersonCompany`, for example. It will still have the same two primary key properties; company and person identifiers. It will also have properties for one person and one company and a property for the hire date. Also, both `Person` and `Company` classes will have a collection of `PersonCompanies` instead of `companies` and `persons`, respectively. Finally, you will configure the Many-to-Many relationship, just like in the previous example between this new class and the `Person` class, and between the new class and the `Company` class. Take a few minutes and model this use case for practice if you would like.

The One-to-One relationship

The One-to-One relationship is not very common, but does occasionally come up. You may choose to pursue this design if you have a lot of optional data for an entity that is grouped somehow. For example, a person can be a student with a college name and enrolment date. These fields will be nullable for anyone who is not a student. So, we will group these fields into another entity called student, as shown in the following code:

```
public class Student
{
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
    public string CollegeName { get; set; }
    public DateTime EnrollmentDate { get; set; }
}
```

The same code in VB.NET looks as follows:

```
Public Class Student
    Property PersonId() As Integer
    Overridable Property Person() As Person
    Property CollegeName() As String
    Property EnrollmentDate() As DateTime
End Class
```

You will notice that we continue to use the virtual keyword (`Overridable` in VB) in order to enable lazy loading. Our configuration class looks very familiar to you, as shown in the following code:

```
public class StudentMap : EntityTypeConfiguration<Student>
{
    public StudentMap()
    {
        HasRequired(s =>s.Person)
            .WithOptional(p =>p.Student);
        HasKey(s =>s.PersonId);
        Property(s =>s.CollegeName)
            .HasMaxLength(50).IsRequired();
    }
}
```

There is one new method we use—`HasKey`. This specifies the primary key for a table; in other words, a unique value that will allow us to find an entity. We did not have to use it before because we followed a naming convention. Entity Framework will figure out the key if the property name consists of the class name plus the "Id" suffix or just "Id". Since we use `PersonId` as the primary key value now, we need to provide an additional hint to the runtime, and that is where the `HasKey` method comes in. The primary key in the child table will also become the foreign key to the parent table. Here is how this class looks in VB.NET:

```
Public Class StudentMap
    Inherits EntityTypeConfiguration(Of Student)
    Public Sub New()
        HasRequired(Function(s) s.Person) _
            .WithOptional(Function(p) p.Student)
        HasKey(Function(s) s.PersonId)
        Me.Property(Function(s) s.CollegeName) _
            .HasMaxLength(50).IsRequired()
    End Sub
End Class
```

Because this relationship is optional, it is called *One-or-Zero-to-One*. You may have another use case where both ends of the relationship are required. This type is called *One-to-One*. For example, each person must have a single login, which is mandatory. The code for this use case will be almost identical, with one exception—you will use the `WithRequiredDependent` or `WithRequiredPrincipal` method instead of the `WithOptional` method.



We can always configure relationships from either the dependent or principal side of the relationship. We need to always configure both ends of a One-to-One relationship, using the `Has` and `With` methods to ensure that the One-to-One relationship is created.

Self-test questions

Q1. You would like to define a column to store a number without a fractional value, but you want to make the value optional. What .NET type should you use for such property?

1. Decimal
2. Decimal?
3. Int
4. Int?

Q2. If you want to make the first name column to be non-nullable in the database, you can rely on default conventions in Entity Framework and avoid all configuration, true or false?

Q3. You cannot override conventions that are preloaded with Entity Framework, such as the one that makes all foreign key constraints be setup to cascade on delete, true or false?

Q4. Which of the following is not a type of relationship?

1. One-to-Many
2. Many-to-Many
3. One-or-Zero-to-Many
4. Many-to-Default

Q5. The best way to configure all properties in all classes is to list them one by one in `OnModelCreating` method of the context, true or false?

Q6. If you do not configure any additional information for string properties, what type will be used in the SQL Server database?

1. NVARCHAR(4000)
2. NVARCHAR(MAX)
3. VARBINARY(MAX)
4. VARCHAR(MAX)

Q7. Which is not an appropriate name for the first end in a relationship definition?

1. Principal
2. Parent
3. Domain

Q8. If you want to use a "buddy" class to configure an entity, what class do you need to inherit from?

1. `EntityTypeConfiguration<T>` (of `T`)
2. `PrimitivePropertyConfiguration<T>` (of `T`)
3. `ComplexTypeConfiguration<T>` (of `T`)
4. `EntityConfiguration<T>` (of `T`)

Summary

In this chapter, you learned how we can configure persistence layer details for entities and classes that map to database structures, specifically tables. We learned that we can use attributes, entity type configuration classes or model builder APIs to perform this task. We discovered that we can make columns nullable by using appropriate nullable types in .NET. We looked at mappings between .NET types and SQL Server types as an example of an RDMBS. We discovered that primitive types, such as numbers and strings, have corresponding property configuration classes that expose methods, allowing us to make those properties required, or configure the maximum allowable length. We learned that using the `EntityTypeConfiguration` class allows us to neatly organize our configuration code. We saw that this class exposes the API that affords developers an opportunity to configure all the properties in a fluent manner.

We also learned that classes can have relationships between each other. We saw there are three distinct types of relationships: One-to-Many (or One-to-Zero-to-Many), Many-to-many, and One-to-One (or One-to-Zero-to-One). The type names refer to a multiplicity value at each end of the relationship, with the principal or parent being the first end and dependent or child being the second one. We saw that configuration objects, such as entity type configuration, expose the API that allows us to configure the relationship using methods such as `HasMany` or `HasRequired`. We discovered that in the case of the Many-to-Many relationship, you may have additional data in the junction table or not. If no additional data is required, the entire table only exists in the database and not in the object model.

In the next chapter, we will start using our structure to query and manipulate the data.

4

Querying, Inserting, Updating, and Deleting Data

In this chapter, we will learn how to query data in your database using Entity Framework and LINQ. We will understand the details of query life cycle. We will see how to use eager and lazy loading. We will also study how to sort and filter data. You will learn the use of relationships in our queries. We will add, update, and delete data in the database using multiple approaches for each operation.

In this chapter, we will cover the following topics:

- How to use a method and the query syntax with LINQ
- How to filter and sort data in your queries
- Learn the pitfalls and advantages of lazy and eager loading
- Cover multiple approaches for data manipulation

The basics of LINQ

Language INtegrated Query (LINQ) is the language that we use with Entity Framework to construct and execute queries against a database. A **query** is a statement that retrieves data from one or more tables. LINQ has many query implementations. At the most basic level, .NET includes LINQ in an object's functionality that allows you to query in-memory collections. LINQ to entities is typically the name that is used when talking about LINQ in relation to Entity Framework. This technology uses Entity Framework in conjunction with a provider for a specific RDBMS to convert LINQ statements to SQL queries. Entity Framework takes care of materialization; the process of converting the results of SQL queries into collections of .NET objects or individual objects.

When you use LINQ to entities queries, you will find out that the SQL is executed against the database when you enumerate the query results. Entity Framework converts LINQ queries to expression trees and then command trees and then they are passed to the provider, which finally executes a SQL query against the database engine it supports. For example, if you step through the following code while you have SQL Profiler running, you will see that the query will be run against the database when you step through the second line, but not the first line of code, as shown in the following code snippet:

```
var query = context.People;  
var data = query.ToList();
```

The same code in VB.NET looks as follows:

```
Dim query = context.People  
Dim data = query.ToList()
```

It may not be obvious from the first reading of this code, but the `ToList` function is the function that causes the enumeration of the query results to occur. The Entity Framework provider for SQL Server is used in the implementation of the `IQueryable` interface in our user case. Hence, when the `GetEnumerator` function is called on `IQueryable`, which is one of the interfaces that `DbSet<T>` implements, a SQL query is constructed and run by Entity Framework and the SQL Server provider for Entity Framework.

LINQ supports two types of query syntax:

- The method syntax
- The query syntax

It is entirely up to you which syntax you want to use, as there is parity between both syntaxes. As the name implies, the **query syntax** looks similar to SQL queries from the language perspective. The **method syntax**, on the other hand, looks like typical function calls in C# or VB.NET.

All of the following examples do not include the code that creates and disposes of `DbContext` for brevity. The context is stored in the `context` variable. We are going to use a database with a structure very similar to the one in *Chapter 3, Defining the Database Structure*, for all of the examples in this chapter.

Filtering data in queries

Filtering refers to the process of narrowing down the results of your query base on a condition. Let's take a look at both syntaxes of LINQ using filtering as an example:

```
var query = from person in context.People
            where person.HeightInFeet >= 6
            select person;
var methodQuery = context.People.Where(p => p.HeightInFeet >= 6);
```

The purpose behind the query is to retrieve the people who are at least 6 feet tall. This means that we have a specific condition that all the data in our results must match. Both lines of code answer this question. The first line uses query syntax. There are distinct parallels to the SQL query syntax. We have the `from`, `where`, and `select` blocks of code. Their order is different from SQL, but they certainly serve the same purpose. When it comes to actual filtering, we are using the `where` keyword to specify the filter that is applied to a query variable called `person`, which we used in the `from` block of our query. The actual filter is using the ordinary C# or VB.NET syntax to compare two expressions; the property of a `person` object, and a constant for the minimum height. The second line of code uses the method syntax. As the name implies, we see method calls via extension methods on any list that implements `IEnumerable`. We do not see the `select` method call, as it is optional in queries based on the method syntax. We will see examples of the `select` method in the next chapter. We do see the `where` method call that is applied to the collection, which is the source of the data. The same source was mentioned in the `from` block inside the query syntax solution to this filter problem. The same method syntax code in VB.NET looks only slightly differently because of the language differences between VB and C# for *Lambda* expressions. The query syntax code is virtually identical in both languages. The following is the VB.NET syntax:

```
Dim query = From person In context.People
            Where person.HeightInFeet >= 6
            Select person
Dim methodQuery = context.People.Where(Function(p) p.HeightInFeet
>= 6)
```

We will continue to see both syntaxes in the following examples to provide a thorough explanation of the differences between the two. Typically, you would want to stick to the same syntax in order to have more consistent code, but you are free to pick the syntax you like better. If you are familiar with SQL and use it on a regular basis, you may find the query syntax a bit more intuitive. Ultimately, it does not matter which one you pick.

You can also combine multiple filter expressions in a single query. As you might have guessed, it is just a matter of **ANDing** the two filter criteria. We will demonstrate this by providing an example that is using a string-based filter as well as a Boolean filter, as shown in the following code snippet:

```
var query = from person in context.People
            where
                person.HeightInFeet >= 6 &&
                person.FirstName.Contains("J") &&
                person.IsActive
            select person;
var methodQuery = context.People
    .Where(p =>
        p.HeightInFeet >= 6 &&
        p.FirstName.Contains("J") &&
        p.IsActive);
```

We are using the standard `Contains` method of the `String` class to perform filtering similar to the `LIKE` query in SQL. This signifies that the standard .NET code is converted to SQL by Entity Framework and the provider together. Boolean or date comparisons work exactly the same way. You can use any Boolean expressions for filtering. The code in VB.NET is very similar with the exception of a few keywords, as shown in the following code snippet:

```
Dim query = From person In context.People
            Where
                person.HeightInFeet >= 6 And
                person.FirstName.Contains("J") And
                person.IsActive
            Select person
Dim methodQuery = context.People _
    .Where(Function(p) p.HeightInFeet >= 6 And
        p.FirstName.Contains("J") And
        p.IsActive)
```

All your standard comparison operators work just fine in LINQ queries, as well as Entity Framework queries.



Remember that the case sensitivity of string comparison depends on the database engine and collation. SQL Server is case insensitive by default, whereas Oracle is case sensitive by default for Latin-based collation. This applies to the filtering and sorting of data. The `ToUpper` method of the `String` class is your friend for solving case sensitivity.

Sorting data in queries

Most of the time, as you retrieve your data from the database, you need to present it in a certain order. This order can include one or more fields and can be ascending, going from the smallest to the largest value, or descending, going from the largest to the smallest. You will find the sorting query syntax quite familiar, if you are accustomed to writing SQL queries. The following example will sort the person data on the last and first names of a person in ascending order. We are also going to combine filtering with sorting to illustrate how these two concepts work together in a single query, as shown in the following code snippet:

```
var query = from person in context.People
            where person.IsActive
            orderby person.LastName, person.FirstName
            select person;

var methodQuery = context.People
    .Where(p => p.IsActive)
    .OrderBy(p => p.LastName)
    .ThenBy(p => p.FirstName);
```

The first query uses the query syntax and the second one uses the method syntax. If you want to sort by multiple fields, then instead of using the `Order By` clause for both fields, you will need to use the `ThenBy` operator for the method syntax. Here is how the queries look in VB.NET:

```
Dim query = From person In context.People
            Where person.IsActive
            Order By person.LastName, person.FirstName
            Select person

Dim methodQuery = context.People _
    .Where(Function(p) p.IsActive) _
    .OrderBy(Function(p) p.LastName) _
    .ThenBy(Function(p) p.FirstName)
```

When the descending sort order is required, you will need to use `OrderByDescending` for the first field and `ThenByDescending` for subsequent fields in queries with the method syntax. If you are using the query syntax, you will need to follow the property name with the **descending** keyword. You are free to combine the descending and ascending orders in a single query, as shown in the following code snippet:

```
var query = from person in context.People
            where person.IsActive
            orderby person.LastName descending, person.FirstName
            descending
            select person;
```

Exploring LINQ functions

There are many LINQ functions that you will need to know in order to be proficient with queries in Entity Framework.

Element operations

Element operations allow you to select a single row. Sometimes they are enhanced to select null if a row that matches the target condition does not exist. Typically, you would combine element functions with a filter condition, though this is not necessary. As element functions work on sets of data, you will need to construct a query first and then apply an element function. If you are using the method syntax, you can combine both actions into a single statement. If you are using the query syntax, you would need to apply an element function to the entire query. For example, let's select a single record based on the last name of a person, as shown in the following code:

```
var query = from person in context.People
            where person.LastName == "Doe"
            select person;
var first = query.First();

var methodQuery = context.People.Where(p => p.LastName == "Doe");
first = methodQuery.First();
```

In the preceding example, we used the `First()` method to find a first matching row in the database. If you look at the type of the `first` variable, you will see that it is of the type `person`. Hence, we fetch an item from a set of persons. Here is the same example in VB.NET:

```
Dim query = From person In context.People
            Where person.LastName = "Doe"
            Select person
Dim first = query.First()
Dim methodQuery = context.People _
                .Where(Function(p) p.LastName = "Doe")
first = methodQuery.First()
```

You can easily combine the last two lines of the example into a single statement, which is much more common. The `First` function has an overload that accepts an expression for the filter condition or the `Where` clause as shown in the following code line:

```
first = context.People.First(p => p.LastName == "Doe");
```

This can also be represented in VB.NET as follows:

```
first = context.People.First(Function(p) p.LastName = "Doe")
```

You will notice that the LINQ query syntax is a bit more verbose in the case of element operations. Hence, most people use the method syntax, as shown in the last example, for the purposes of finding a single row in the database. This typically results in a single line of code that you have to write.

When you use the `First` function and you have multiple rows in the database that match the condition, only one row will be picked up and returned. If you have no rows that match the condition, an exception will be thrown. Hence, if you want to guard against such an exception, you can use the `FirstOrDefault` function instead of `First`. As we deal with entities, which are classes or reference types, their default is null. So, before you use the results of the `FirstOrDefault` execution, you need to test for a null value. Outside of this fact, the `FirstOrDefault` code is identical to `First`.

There are two more operations that are similar to `First`, called `Single` and `SingleOrDefault`. The only difference from `First` is that if you have more than one row that matches your condition, an exception is thrown. Feel free to write an example now, that is using the `Single` function. LINQ has other element operators, such as `Last` and `ElementAt`, however, they do not make much sense in terms of Entity Framework, hence an exception will be thrown if you use them inside LINQ to entities queries. You can use them inside LINQ to objects queries. If you want to use the `Last` function with Entity Framework, simply use `First` and reverse the sorting order.

Quantifiers

Occasionally, you need to check whether you have at least one row that matches a condition, or all rows match a filter. This is where the `Any` and `All` operations come in. Together, they are referred to as **quantifiers**. As the name implies, these operators return a Boolean value. Both are applied to a query, hence the code is quite similar to the preceding `First` example. This similarity is shown in the code:

```
var hasDoes = (from person in context.People
               where person.LastName == "Doe"
               select person).Any();
hasDoes = context.People.Any(p => p.LastName == "Doe");
var allHaveJ = context.People.All(p => p.FirstName.Contains("J"));
```

There are a few things to notice here. First of all, we combined a query with a quantifier in a single statement to illustrate how you can cut down on the number of lines of code using the query syntax. Then, we performed the same check—whether there is at least one person with the last name of `Doe` in the database using the method syntax. Finally, we checked to make sure that all people have the letter `J` in their first names. Here is the same code using VB.NET:

```
Dim hasDoes = (From person In context.People
               Where person.LastName = "Doe"
               Select person).Any()

hasDoes = context.People.Any(Function(p) p.LastName = "Doe")

Dim allHaveJ = context.People.All(Function(p)
p.FirstName.Contains("J"))
```

You can easily substitute the `Any` call with a standard query with a `Where` clause and check whether the result has any rows. However, you want to use `Any` for such a purpose instead, because it will result in a more efficient SQL statement. Typically, it will use the `EXISTS` syntax instead of the `WHERE` SQL syntax, thus short-circuiting the execution upon finding the first row.

Working with related entities

At times, we need to write queries that involve more than a single entity type. We are now going to take a look at how we can work with entities that are involved in relationships.

Filtering based on related data

Sometimes, we need to filter based on related data. For example, we want to find people who have phone numbers that start with "1". It is worth noting that this additional task is performed purely based on relationships between a person and phone entities, using the `Phones` property on the `person` class, as shown in the following code snippet:

```
var query = from person in context.People
            where person.IsActive &&
            person.Phones.Any(ph =>
ph.PhoneNumber.StartsWith("1"))
            select person;

var methodQuery = context.People
    .Where(p => p.IsActive &&
    p.Phones.Any(ph => ph.PhoneNumber.StartsWith("1")));
```


Again, we use the plain string function, `StartsWith` in the preceding case, yet this will translate into `LIKE '1%'` in the where clause in the SQL Server case. We also use the familiar `Any` function to only find people with at least one phone number that starts with the number 1. We also (in a way) add the method syntax to the query syntax, giving us additional flexibility. Here is how this code looks in VB.NET:

```
Dim query = From person In context.People
             Where person.IsActive And
                   person.Phones.Any(Function(ph)
                                     ph.PhoneNumber.StartsWith("1"))
             Select person

Dim methodQuery = context.People _
                  .Where(Function(p) p.IsActive And
                          p.Phones.Any(Function(ph)
                                        ph.PhoneNumber.StartsWith("1")))
```

Lazy and eager loading

We previously talked about the difference between eager and lazy loading. Both concepts are approaches to loading-related entities. For example, you may use either approach to load phone numbers for a person. If you are trying to decide which approach you need to use in a particular situation, the following rule should answer this question for you. If you are not sure that you are going to need related entries, you can use lazy loading. If you know you will certainly need related data, then use eager loading. You will need to be careful when you decide to eagerly load many relationships. This can result in very complex queries and thus can have performance implications. You can encounter problems with lazy loading as well. For example, you want to retrieve 100 rows from the `People` table and then display phone numbers for each person. If you use lazy loading, as you start enumerating through a list of phone numbers, Entity Framework will issue a query to retrieve that data for each person's phones list. This processing will result in 101 queries issued against the database. One query will retrieve 100 people, then one more for each person to get phone numbers. It takes time to perform these actions, so you may be looking at an extra few hundred milliseconds when using lazy loading in this use case. You need to carefully decide which approach is right for each of your situations. By default, lazy loading is enabled in Entity Framework. You can turn it off for an instance of `DbContext` by accessing configuration options on `DbContext` after context is created, which is shown as follows:

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

It may be useful to you to know how lazy loading is implemented in Entity Framework. If you run the following code and break on the line that executes the `foreach` loop that looks on phone numbers and examine an instance of the `person` object, you will notice something interesting. The type of the `person` object is not really `Person`, but instead something like `System.Data.Entity.DynamicProxies.Person_XXXXXX`. Entity Framework dynamically created a class that inherits from `Person` in order to intercept property getter calls: in our case, the `Phones` property. Then, in the property getter, it dynamically issues a query to populate the `Phones` list. However, your code can just assume that the data will be automatically populated. This is shown in the following code snippet:

```
var query = from person in context.People
            select person;

foreach (var person in query.ToList())
{
    foreach (var phone in person.Phones)
    {
```

Now, let's take a look at eager loading. You have to use the `Include` method in order to proactively load the related data you need. There is one parameter that this method takes, and it is the property expression that points to a related entities property. Let's implement the preceding code, but use eager loading. This is shown in the following code snippet:

```
var query = from person in context.People.Include(p => p.Phones)
            select person;

foreach (var person in query)
{
```

There are two overloads of the `include` method. The one we used took a property expression. You can also use a string to describe the path of the relationship. This is shown in the following code snippet:

```
var query = from person in context.People.Include("Phones")
            select person;
```


Ordinarily, you want to use the property expression method because you can take advantage of compile-time checking. The only time you need to use string-based overload is when your path cannot be described via a property expression. For example, if you want to load multiple levels of relationships. If `Phones` were to have types, your `Include` method may look like `Include("Phones.PhoneType")`.

Here is how the code that uses eager loading looks in VB.NET:

```
Dim query = From person In context.People.Include(Function(p)
    p.Phones)
           Select person

For Each person As Person In query
    Console.WriteLine(person.LastName)
    For Each phone As Phone In person.Phones
```

You noticed that we used the `ToList` call before running through the query when lazy loading is used. This is typically not necessary and really not even recommended. However, in the case of lazy loading, you have to follow this pattern. The problem is that when a lazy loaded property is populated by Entity Framework, a new data reader is created. However, we already have an open data reader that is reading in our primary, top-level query, which is the person query in our case. ADO.NET has a limitation, where only one open reader is allowed per database connection. As a result, if you remove the function `ToList` from the code that is using lazy loading, you will encounter an exception. A call to functions such as `ToList` or `ToArray` cause what is referred to as an **immediate execution** of the query to occur. This is in contrast to simply enumerating the results of a query, as in the eager loading example, when we use deferred query execution. SQL is executed when query results are enumerated in both cases, but with a slight difference.

 You should consider the performance implications of using eager loading versus lazy loading any time you are retrieving related data.

Inserting data into the database

There are many ways to insert new data into your database. You can add new objects to the collection, as we did in previous chapters. You can also set the state to `Added` on each entity. If you are adding entities that contain child entities, the `Added` state is propagated to all the objects in the graph. In other words, Entity Framework assumes that you are attaching a new object graph if the root entity is new. The **object graph** term typically refers to a number of related entities that form a complex tree structure. For example, if we have a `person` object with a number of phone numbers contained in a `list` property on the `Person` class, we are dealing with an object graph, where the `person` entity is a root object. `Phone` entities are, in essence, children of that `person` object. Since we have seen a simple functionality, let's work through this complex addition scenario.

First, we will create a new `person` instance with phone numbers. Then, we will add this `person` instance to the context. Finally, we will call `SaveChanges` to commit the rows to the database as shown in the following code snippet:

```
var person = new Person
{
    BirthDate = new DateTime(1980, 1, 2),
    FirstName = "John",
    HeightInFeet = 6.1M,
    IsActive = true,
    LastName = "Doe",
    MiddleName = "M"
};

person.Phones.Add(new Phone { PhoneNumber = "1-222-333-4444" });
person.Phones.Add(new Phone { PhoneNumber = "1-333-4444-5555" });

using (var context = new Context())
{
    context.People.Add(person);
    context.SaveChanges();
}
```

There are a few differences from the code we saw previously. We create our objects before we initialize the context. This stresses the point that Entity Framework tracks whether entities in the context at the time are attached or added. You can, for example, have your `person` entity from the preceding example be a parameter to a method inside the Web API controller. Then, inside that function, you will just add the entity with the children to the context and save it, as shown. Here is how this code looks in VB.NET:

```
Dim person = New Person() With {
    .BirthDate = New DateTime(1980, 1, 2),
    .FirstName = "John",
    .HeightInFeet = 6.1D,
    .IsActive = True,
    .LastName = "Doe",
    .MiddleName = "M"
}

person.Phones.Add(New Phone() With {.PhoneNumber = "1-222-333-4444"})
person.Phones.Add(New Phone() With {.PhoneNumber = "1-333-4444-5555"})
Using context = New Context()
```

```

    context.People.Add(person)
    context.SaveChanges()
End Using

```

You can also add multiple entities at the same time, since `DbSet` has the `AddRange` method that will allow you to pass in a number of entities.

This is not the only way to insert new data, though it is simple, straightforward, and easy to read and understand. Another way is to directly set the entity state using the `DbContext` API, as shown in the following code snippet:

```

using (var context = new Context())
{
    context.Entry(person2).State = EntityState.Added;
    context.SaveChanges();
}

```

We did not include all the code, just the differences from the approach that is using the `Add` method on `DbSet`. Here is how the code looks in VB.NET:

```

Using context = New Context()
    context.Entry(person2).State = EntityState.Added
    context.SaveChanges()
End Using

```

The `Entry` method on `DbContext` returns an instance of the `DbEntityEntry` class. This class has a number of useful properties and methods that are needed for more advanced implementations and scenarios with Entity Framework. For example, you might use `OriginalValues` and `DatabaseValues` in order to handle conflict resolution during optimistic concurrency handling. There is also the `GetValidationResult` method that you can use to ensure that the data is valid from the perspective of the rules we specified in our `EntityTypeConfiguration` classes for our entities. For example, if you have a required string property, you cannot leave it as null, and this method will provide you with an error. You can also use `DbEntityEntry` to inquire what the state of the object is instead of setting a new state. If you want to, try to get the state right after calling the `Add` method in the first insert example and ensure that the state for the new person is indeed `Added`. These are the states supported by the `EntityState` enumeration:

State	Description
Added	A new entity is added. This state will result in an <i>insert</i> operation.
Deleted	An entity is marked for deletion. When this state is set, the entity will be removed from <code>DbSet</code> . This state will result in a <i>delete</i> operation.
Detached	The entity is not tracked by <code>DbContext</code> .

State	Description
Modified	One or more properties of the entity have been changed since DbContext started tracking the entity. This state will result in an <i>update</i> operation.
Unchanged	No properties of the entity have been changed since DbContext started tracking the entity.

Updating data in the database

What does it mean to update data in the database? We want to replace one or more column values in a table's row with new values. Entity Framework will issue an update query when it knows that an entity has changed since it was first attached to DbContext, either by viewing a LINQ query that was enumerated, or via a call to Attach method of DbSet. The simplest way to find an entity you want to update is to use a query. Then, change one or more properties to new values and call SaveChanges. From the moment we query the data, Entity Framework will start tracking changes to each property. When SaveChanges is finally called, only changed properties will be included in the update SQL operation. When you want to find an entity to update in the database, you typically look for it based on the primary key value. We already saw how to use the Where method to achieve this. Entity Framework also has the Find method exposed on DbSet. This method takes one or more values as parameters that correspond to the primary key of the table mapped to that DbSet. We use the column that has a unique ID as the primary key in our example, hence we only need a single value. If you use **composite primary keys**, consisting of more than one column, which is typical for junction tables, you will need to pass the values for each column that the primary key is comprised of in the exact order of the primary key columns. If you are following along with this exercise, open SSMS or the **SQL Server Object Explorer** window inside Visual Studio and find the ID of a person in the People table to practice on. In the following code, the value passed to the Find function is 1:

```
using (var context = new Context())
{
    var person = context.People.Find(1);
    person.FirstName = "New Name";
    context.SaveChanges();
}
```

The same code in VB.NET looks as follows:

```
Using context = New Context()
    Dim person = context.People.Find(1)
```

```

        person.FirstName = "New Name"
        context.SaveChanges()
    End Using

```

If you trap the SQL query sent to the SQL Server when `SaveChanges` is called, it will look as follows:

```

UPDATE [dbo].[People]
SET [FirstName] = @0
WHERE ([PersonId] = @1)

```

This proves that indeed only the changes that are made explicitly are sent back to the database. For example, the last name was not updated, since it was not changed. If you look in SQL Profiler for the entire code block, you will see that the `Find` method also resulted in a query that is shown in the following code snippet:

```

SELECT TOP (2)
    [Extent1].[PersonId] AS [PersonId],
    [Extent1].[PersonTypeId] AS [PersonTypeId],
    [Extent1].[FirstName] AS [FirstName],
    [Extent1].[LastName] AS [LastName],
    [Extent1].[MiddleName] AS [MiddleName],
    [Extent1].[BirthDate] AS [BirthDate],
    [Extent1].[HeightInFeet] AS [HeightInFeet],
    [Extent1].[IsActive] AS [IsActive]
FROM [dbo].[People] AS [Extent1]
WHERE [Extent1].[PersonId] = @p0

```

`Find` was translated into the `SingleOrDefault` method call. That is why we selected the `Top (2)` rows. We want to make sure that there is only one entity that matches the primary key.

If you are writing a desktop Windows application, you may choose to use the approach of keeping the context around after a query is fired to issue updates. The entity must remain connected to the context from a query to the `SaveChanges` call timeline. You can find an entity, let the user make changes, and finally call `SaveChanges`. If you want to model this approach in our code, user interactions correspond to the `person.FirstName = "New Name"` line of code. If you are working on a web application, this approach does not work. You cannot keep the original context around or between two web server calls. You do not really need to take the overhead of finding an entity twice, once to show to the user and the second time to update. Instead, let's use the second approach from the insert examples and set the state, as shown in the following code snippet:

```

var person2 = new Person
{

```

```
        PersonId = 1,
        BirthDate = new DateTime(1980, 1, 2),
        FirstName = "Jonathan",
        HeightInFeet = 6.1m,
        IsActive = true,
        LastName = "Smith",
        MiddleName = "M"
    };
    person2.Phones.Add(new Phone
    {
        PhoneNumber = "updated 1",
        PhoneId = 1,
        PersonId = 1
    });
    person2.Phones.Add(new Phone
    {
        PhoneNumber = "updated 2",
        PhoneId = 2,
        PersonId = 1
    });
    using (var context = new Context())
    {
        context.Entry(person2).State = EntityState.Modified;
        context.SaveChanges();
    }
```

You can imagine that the data we created initially, prior to instantiating the context, was submitted via a web call to our Web API controller. Then, once inside the controller, we create the context, set the state, and save changes. If you look at the results of this code inside the database, you might be surprised to find out that the person data was updated, but the phone data was not. This occurred because of a fundamental difference between the insert and update implementation inside Entity Framework. When you set the state to modified, Entity Framework does not propagate this change to the entire object graph. So, to make this code work properly, we need to add a little bit more code, as shown in the following code snippet:

```
using (var context = new Context())
{
    context.Entry(person2).State = EntityState.Modified;
    foreach (var phone in person2.Phones)
    {
        context.Entry(phone).State = EntityState.Modified;
    }
    context.SaveChanges();
}
```


All we had to do manually was set the state of each changed entity. Of course, if you have a new phone number in the collection, you can set its state to `Added` instead of `Modified`. There is one more important concept contained within the code. Whenever we use the state change approach, we must know all the columns' data, including the primary key for each entity. This is because Entity Framework assumes that when the state is changed, all the properties need to be updated. Here is how the code looks in VB.NET:

```
Dim person2 = New Person() With {
    .PersonId = 1,
    .BirthDate = New DateTime(1980, 1, 2),
    .FirstName = "Jonathan",
    .HeightInFeet = 6.1D,
    .IsActive = True,
    .LastName = "Smith",
    .MiddleName = "M"
}
person2.Phones.Add(New Phone() With {.PhoneNumber = "updated 1",
    .PhoneId = 1, .PersonId = 1})
person2.Phones.Add(New Phone() With {.PhoneNumber = "updated 2",
    .PhoneId = 2, .PersonId = 1})
Using context = New Context()
    context.Entry(person2).State = EntityState.Modified
    For Each phone In person2.Phones
        context.Entry(phone).State = EntityState.Modified
    Next
    context.SaveChanges()
End Using
```

If you capture SQL queries sent when this code is run, you will see three *update* queries – one for the person and one more for each of the two phone numbers.

It is also worth repeating that Entity Framework tracks the state of the entities once they are attached to the context. So, if you query the data, the context starts tracking your entities. If you are writing a web application, this tracking becomes an unnecessary overhead for query operations. The reason it is unnecessary is because you will dispose of the content, destroying the tracking as soon as the web request to get the data completes. Entity Framework has a way to reduce this overhead. For example:

```
using (var context = new Context())
{
    var query = context.People.Include(p =>
        p.Phones).AsNoTracking();
}
```

```
        foreach (var person in query)
        {
            foreach (var phone in person.Phones)
            {
            }
        }
    }
```

If you put a breakpoint inside the loop and check the entity state, using `context.Entry(person).State` and `context.Entry(phone).State` expressions, you will see that the state is `Detached` for both entities. This means that this entity is not tracked by the context, thus reducing your overhead by using the `AsNoTracking` method.

The same code in VB.NET looks as follows:

```
Using context = New Context()
    Dim query = From person In context.People.Include(Function(p)
p.Phones).AsNoTracking()
                Select person
    For Each person As Person In query
        For Each phone As Phone In person.Phones
            Next
        Next
    End Using
```

We also combined turning off change tracking with eager loading. What if even in web environments you only want to update just the properties that are changed by a user? One big assumption is that you will have to track what is changed in your web application on the client. Assuming that this is accomplished, you can use yet another approach to accomplish the update operation. You can use the `Attach` method on `DbSet`. This method essentially sets the state to `Unchanged` and `context` starts tracking the entity in question. After you attach an entity, you can just set one of the changed properties at a time. You must know in advance which properties have changed. For example:

```
var person3 = new Person
{
    PersonId = 1,
    BirthDate = new DateTime(1980, 1, 2),
    FirstName = "Jonathan",
    HeightInFeet = 6.1m,
    IsActive = true,
    LastName = "Smith",
    MiddleName = "M"
};
```

```
using (var context = new Context())
{
    context.People.Attach(person3);
    person3.LastName = "Updated";
    context.SaveChanges();
}
```

This code will result in a query that only updates the `LastName` column and nothing else. The same code in VB.NET looks as follows:

```
Dim person3 = New Person() With {
    .PersonId = 1,
    .BirthDate = New DateTime(1980, 1, 2),
    .FirstName = "Jonathan",
    .HeightInFeet = 6.1D,
    .IsActive = True,
    .LastName = "Smith",
    .MiddleName = "M"
}
Using context = New Context()
    context.People.Attach(person3)
    person3.LastName = "Updated"
    context.SaveChanges()
End Using
```

Alternatively, instead of calling the `Attach` method, you can simply set the state by calling `context.Entry(person3).State = EntityState.Unchanged`. Just replace one line of code that calls `Attach` with this line and you are done.

Deleting data from the database

Interestingly enough, there is a lot of similarity in the approaches we used for updates and deletions. We can use a query to find data and then mark it for deletion by using the `Remove` method of `DbSet`. This approach actually has the same drawbacks as it does with the update, resulting in a select query in addition to the delete query. Nonetheless, let's take a look at how it is done:

```
using (var context = new Context())
{
    var toDelete = context.People.Find(personId);
    toDelete.Phones.ToList().ForEach(phone =>
context.Phones.Remove(phone));
    context.People.Remove(toDelete);
    context.SaveChanges();
}
```

This code deletes each child entity, *phone* in our case, and then deletes the root entity. You would have to know the primary key value for the entity you want to delete. The preceding code assumes that you have this value in the `personId` variable. In the case of a web application, this value will be submitted to the method that handles deletion. Alternatively, we could use the `RemoveRange` method to remove multiple entities in a single statement. Here is how the code looks in VB.NET:

```
Using context = New Context()
    Dim toDelete = context.People.Find(personId)
    toDelete.Phones.ToList().ForEach(Function(phone)
context.Phones.Remove(phone))
    context.People.Remove(toDelete)
    context.SaveChanges()
End Using
```

There is one important difference between this code and the insert code. We have to manually delete each child record, by removing it from a corresponding collection. Code that is provided relies on lazy loading to work to populate the list of phones for a person. You can also rely on a cascade of *delete* operation instead, though some DBAs will frown at this practice.

Now, let's delete entities by setting a state on each entity. Again, we need to account for dependent entities. For example:

```
var toDeleteByState = new Person { PersonId = personId };
toDeleteByState.Phones.Add(new Phone
{
    PhoneId = phoneId1,
    PersonId = personId
});
toDeleteByState.Phones.Add(new Phone
{
    PhoneId = phoneId2,
    PersonId = personId
});

using (var context = new Context())
{
    context.People.Attach(toDeleteByState);
    foreach (var phone in toDeleteByState.Phones.ToList())
    {
        context.Entry(phone).State = EntityState.Deleted;
    }
}
```

```

        context.Entry(toDeleteByState).State = EntityState.Deleted;
        context.SaveChanges();
    }

```

You undoubtedly noticed something very different from any other data manipulation. In order to delete a person, we only need to set the primary key property and nothing else. For phones, we just needed to set the primary key property and parent identifier property. In the case of web applications, you need to submit all the identifiers, or you will need to resort to requerying child data to find the identifiers. Here is how this code looks in VB.NET:

```

Dim toDeleteByState = New Person With { .PersonId = personId }
toDeleteByState.Phones.Add(New Phone With { .PhoneId = phoneId1,
        .PersonId = personId })
toDeleteByState.Phones.Add(New Phone With { .PhoneId = phoneId2,
        .PersonId = personId })

Using context = New Context()
    context.People.Attach(toDeleteByState)
    For Each phone In toDeleteByState.Phones.ToList()
        context.Entry(phone).State = EntityState.Deleted
    Next
    context.Entry(toDeleteByState).State = EntityState.Deleted
    context.SaveChanges()
End Using

```

You can submit full entities for deletion as well. If you are sticking to strict guidelines for deletion for REST web services, those define that only an identifier should be submitted with a web request. So, you will need to decide for yourself which of the two approaches works better for your specific circumstances.

Working with in-memory data

Sometimes, you need the ability to find an entity in an existing context instead of the database. Entity Framework, by default, will always execute queries against the database when you create new context. What if your update involves calling many methods and you want to find what data was added by one of the previous methods? You can force a query to execute only against in-memory data attached to the context using the `Local` property of `DbSet`. For instance:

```

var localQuery = context.People.Local.Where(p =>
    p.LastName.Contains("o")).ToList();

```

The same code in VB.NET looks as follows:

```
Dim localQuery = context.People.Local.Where(Function(p)
    p.LastName.Contains("o")).ToList()
```

What we also know is that the `Find` method searches local `context` first, prior to constructing the database query. You can easily confirm this by forcing another query to load the data you are looking for and then running a `Find` against one of the found entities. For instance:

```
var query = context.People.ToList();
var findQuery = context.People.Find(1);
```

If you run SQL Profiler along with this code, you will see that one query is executed against the database, which confirms that `Find` runs on in-memory data first. The same code in VB.NET looks as follows:

```
Dim query = context.People.ToList()
Dim findQuery = context.People.Find(1)
```

In-memory data also provides access to all the entities with their respective states via the `DbChangeTracker` object. It allows you to look at the entities and their `DbEntityEntry` objects as well. For example:

```
foreach (var dbEntityEntry in
    context.ChangeTracker.Entries<Person>())
{
    Console.WriteLine(dbEntityEntry.State);
    Console.WriteLine(dbEntityEntry.Entity.LastName);
}
```

In the preceding code, we get all the person entries that `DbContext` is tracking as entity entry objects. We can then look at the state of each object as well as an actual `Person` object that the entity entry belongs to.

Here is the same code in VB.NET:

```
For Each dbEntityEntry In context.ChangeTracker.Entries(Of Person)
    Console.WriteLine(dbEntityEntry.State)
    Console.WriteLine(dbEntityEntry.Entity.LastName)
Next
```

This API provides developers with rich capabilities to examine in-memory data at any time.

Self-test questions

Q1. Which of the following is NOT a syntax supported by LINQ?

1. Method.
2. SQL.
3. Query.

Q2. If you retrieve an entity via LINQ from a database, make changes to it and call `SaveChanges`, all properties are updated in the database, not only the changed ones, true or false?

Q3. In order to sort the data by multiple properties, you simply need to call `OrderBy` multiple times in LINQ that is using the method syntax, true or false?

Q4. How to add two filter conditions to a LINQ query with the query syntax?

1. Use multiple `Where` calls.
2. Use logical AND operator.
3. Issue two queries.

Q5. You want to add multiple new entities to `DbSet`. How can you accomplish this?

1. By calling the `Add` method and passing an instance of the class specified in the context property.
2. By calling `AddRange` and pass an enumerable of the target entity type.
3. By setting the state to `Added` using the context API on each new entity.
4. All of the above.

Q6. If you want to create a new entity with some child entities, also known as object graph, you must call `Add` on the parent and each child in order to persist the object graph, true or false?

Q7. If you set a state of an entity to `modified`, all columns in the corresponding table are updated when `SaveChanges` is called, true or false?

Q8. You need to call `Add` and then `Remove` on an entity in order to trigger a delete query to be issued when `SaveChanges` is called, true or false?

Q9. Which entity state does not result in a query against the database when `SaveChanges` is called?

1. Added.
2. Detached.
3. Deleted.
4. Modified.

Q10. Which property of `DbSet` gives you access to entities already loaded into the context from a database?

1. Memory.
2. Local.
3. Loaded.

Summary

In this chapter, you learned how to issue basic queries to retrieve entities from a database. LINQ was the driving force behind getting the data from the database through Entity Framework. There are two basic approaches to using LINQ: the method and query syntaxes. The query syntax is quite similar to SQL, whereas the method syntax may be more suitable for developers with more experience in C# or VB.NET than SQL. We saw how to use both approaches to filter data. We saw that the where clause can be used to combine multiple conditions using logical operators. The ordering of data can be done as well, using either the `OrderBy` and `OrderByDescending` methods or ascending or descending keywords. In order to support multiple orders by conditions, we can use the `ThenBy` method. What we retrieved from the database were entities that were mapped to tables. We often do not need to think about this; simply assume that the Entity Framework persistence engine is taking care of these nitty-gritty details for us.

Lazy and eager loading are two basic concepts that are available to developers in order to access related entities from a single root entity. Using the `Include` method allows us to proactively load related data using a single call to the database. Foregoing this approach would result in many calls to the database, however, this is still a good thing in some cases, especially if you are not sure what related data you need. The bottom line is that it is important to recognize the drawbacks and benefits of eager versus lazy loading in order to write efficient and scalable code.

Once entities are retrieved via a query, Entity Framework starts tracking all the changes made to them. As a result, update queries only contain changed data and not all the properties from an entity. There are a few ways to update the data in the database. We could keep entities attached to the context, make changes, and call `SaveChanges` to persist the data. Alternatively, we can just set a state on an entity not attached to the context to `modified` and then save the changes. We saw that persisting an updated object graph requires us to set state on each entity in the graph. We did not have to do this in the case of new data. Marking the root object to be in a new state automatically puts all child entities in a new state as well, thus limiting the amount of work we have to do to insert new related entities. As an alternative to setting the state to `new`, we could simply add an entity to `DbSet`. Deleting the data was not much different. We could set the state to `deleted`, or attach an entity to context, then remove it from `DbSet`. Calling `SaveChanges` after that resulted in the delete query to be issued against the database. We will see many more advanced scenarios for querying data from a database in the next chapter. We will also look at several additional database modeling techniques.

5

Advanced Modeling and Querying Techniques

In this chapter, you will learn how to use advanced modeling techniques to create the database structure. We will learn how to use complex types to create data structures that are reusable in multiple entity types. We will learn how to use enumerations to create a range of distinct values for a column or property. We will understand how to split an entity across multiple tables. We will learn how to support existing databases, while using names for classes and properties that do not match tables and columns in our database. We will also look at additional querying techniques, including aggregation, paging, grouping, and projections.

In this chapter, we will cover how to:

- Create complex types, reusable in many entities
- Define an enumeration and use it in a query
- Create an entity that is stored in multiple tables
- Use explicit column and table names in entity to table mappings
- Create queries that use projections with anonymous and explicit types
- Summarize data, using aggregate functions
- Create windowed queries
- Use explicit joins in queries
- Use set operations

Advanced modeling techniques

So far, we have covered many straightforward scenarios that one can easily model with Entity Framework to create database structures. All of them are mapped to one table with scalar values to a class with a matching set of properties. There are use cases when this approach does not work quite as well, and we will walk through a functionality in Entity Framework that supports more complex modeling techniques.

Complex types

Complex types are classes that map to a subset of columns in a table in the database. They are similar to entity classes, except that they do not contain key fields and do not directly map to an entire table. Complex types are helpful when we have the same set of properties that are common to multiple entities. Another use case is when we want to group some properties, in order to provide a clear semantic meaning to such a group of properties. By introducing a complex type into our modeling workflow, we provide more consistency for database structures across multiple tables. This occurs because we define the common attributes for such tables in a single place, which will be our complex type. The prototypical example is address fields. Given the examples we have seen in prior chapters, let's add addresses to both person and company classes. Here is how the address class, referred to as complex type, looks:

```
public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string Zip { get; set; }
}
```

We are looking at a simple class here with a set of properties that define an address. Here is the same code in VB.NET:

```
Public Class Address
    Public Property Street() As String
    Public Property City() As String
    Public Property State() As String
    Public Property Zip() As String
End Class
```

The second step is to make this class part of a larger picture by introducing the `Address` type property in both the `Company` and `Person` classes. Here is, for example, how the `Company` class looks after this change:

```
public class Company
{
    public Company()
    {
        Persons = new HashSet<Person>();
        Address = new Address();
    }
    public int CompanyId { get; set; }
    public string CompanyName { get; set; }
    public Address Address { get; set; }
    public ICollection<Person> Persons { get; set; }
}
```

There is an important step we need to take now. We need to initialize an instance of the `Address` class in the `Company` class's constructor. Without this simple step, we can easily encounter a null reference exception any time we create a new instance of `Company` and try to set a street on the address. When data is retrieved from the database via a query, Entity Framework automatically initializes the instance of the `Address` class during the materialization process. An additional initialization that we just added manually exists to cover the creation of new entity scenarios. Here is how the code looks in VB.NET:

```
Public Class Company
    Public Sub New()
        Persons = New HashSet(Of Person)
        Address = new Address()
    End Sub
    Property CompanyId() As Integer
    Property CompanyName() As String
    Property Address() As Address
    Overridable Property Persons() As ICollection(Of Person)
End Class
```

The next step is to provide the configuration for our complex type. We can do so in a way that is virtually identical to entity classes, by providing a configuration class for the complex type. The only difference is that we use a different base class, `ComplexTypeConfiguration`, not `EntityTypeConfiguration`. The code inside this configuration class is identical to the code in entity configuration classes, using the exact same property configuration methods.

For example, consider this code snippet:

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

The preceding example only shows one property being configured, but in the code provided with this book, all properties are configured in the same fashion as the `Street` property. Finally, we must remember to add an instance of `AddressMap` to the collection of configurations of the context. For example, consider this code snippet:

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

Here is how the same code looks in VB.NET:

```
Public Class AddressMap
    Inherits ComplexTypeConfiguration(Of Address)
    Public Sub New()
        Me.Property(Function(p)
p.Street).HasMaxLength(40).IsRequired()
    End Sub
End Class

Protected Overrides Sub OnModelCreating(ByVal modelBuilder As
DbModelBuilder)
    modelBuilder.Configurations.Add(New CompanyMap)
    modelBuilder.Configurations.Add(New AddressMap)
End Sub
```

If we were to run this code and look at the created database structure, we would see that the `Address` columns names in the `Company` table are prefixed with the complex type's name. So, a column to store the name of the street is called `Address_Street`. This is typically not something that we want, which leads us to the next discussion about supporting explicit column and table names.

Using an explicit table and column mappings

There are many use cases that require us to explicitly specify a column or a table name. We just saw one, but there are more. For example, we can add Entity Framework on top of an existing database that uses a naming convention developers of data access layer do not like. Explicit names solve this problem.

In order to specify a column name that is different from a matching property name, we can use the `HasColumnName` method available for primitive property configurations, as shown in the following code snippet:

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

One can configure properties on entity types in the exact same way we just configured our complex type. We can see more examples in the code that accompanies this book. Here is how this looks in VB.NET:

```
Public Class AddressMap
    Inherits ComplexTypeConfiguration(Of Address)
    Public Sub New()
        Me.Property(Function(p) p.Street) _
            .HasMaxLength(40) _
            .IsRequired() _
            .HasColumnName("Street")
    End Sub
End Class
```

In order to specify the table name for an entity, we have to use the `ToTable` method of the `EntityTypeConfiguration` class. For example, here is how we can specify the table name for a person type entity:

```
public class PersonTypeMap : EntityTypeConfiguration<PersonType>
{
    public PersonTypeMap()
    {
        ToTable("TypeOfPerson");
    }
}
```

This example is a bit contrived, as we could have just as easily changed the class name. Here is how we can specify a name for a table in VB.NET:

```
Public Class PersonTypeMap
    Inherits EntityTypeConfiguration(Of PersonType)
    Public Sub New()
        ToTable("TypeOfPerson")
    End Sub
End Class
```

Adding supporting columns

In addition to changing column names, sometimes we want to add a property to an entity that we do not want to store in the database. In other words, we want to add some business logic into an entity to help us work with it outside of Entity Framework. I am by no means advocating embedding business logic inside our entity classes, but merely providing an alternative to computed columns in SQL Server. For example, let's add the `FullName` property to our `Person` class and return a concatenation of `LastName` and `FirstName`:

```
public string FullName
{
    get
    {
        return string.Format("{0} {1}", FirstName, LastName);
    }
    set
    {
        var names = value.Split(new string[] { " " },
StringSplitOptions.RemoveEmptyEntries);
        FirstName = names[0];
        LastName = names[1];
    }
}
```

Here is how the same property looks in VB.NET:

```
Public Property FullName() As String
    Get
        Return String.Format("{0} {1}", FirstName, LastName)
    End Get
    Set(value As String)
        Dim names = value.Split(New String() { " " },
StringSplitOptions.RemoveEmptyEntries)
        FirstName = names(0)
        LastName = names(1)
    End Set
End Property
```


If we run this code, we will see that a new column called `FullName` was added to the `People` table. This is not what we want; there is no reason for us to persist the full name. To fix the problem, we just need to use the `Ignore` method of the `EntityTypeConfiguration` class. This is shown in the following example:

```
public class PersonMap : EntityTypeConfiguration<Person>
{
    public PersonMap()
    {
        Ignore(p => p.FullName);
    }
}
```

This approach of ignoring certain properties in a persistence layer could prove useful when developers are dealing with legacy databases. One thing we must remember is that we cannot query based on ignored properties, since they do not exist in the backend. Here is how this code looks in VB.NET:

```
Public Class PersonMap
    Inherits EntityTypeConfiguration(Of Person)
    Public Sub New()
        Ignore(Function(p) p.FullName)
    End Sub
End Class
```

Enumerations

We are all familiar with the use of enumerations. They make our code much more readable, since we can use descriptive names instead of magic numbers. For example, let's say that each type `Person` can be in one of three states: `Active`, `Inactive`, or `Unknown`. This is a prototypical scenario that calls for the use of enumerations. Entity Framework now has full support for enumerations. First of all, we need to define enumeration itself. For example, consider this code snippet:

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

This can also be shown in VB.NET, like the following code:

```
Public Enum PersonState
    Active
    Inactive
    Unknown
End Enum
```

The next step is to simply add a property of the type `PersonState` to the `Person` class. For instance, consider this code fragment:

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

Here is how this new property is defined in VB.NET:

```
Public Class Person
    Property PersonState() As PersonState
```

Technically, there is nothing else we need to do. We can just run this code to create our database structure. Once this is done, queries like the following one would work:

```
var people = context.People
    .Where(p=>p.PersonState == PersonState.Inactive);
```

Here is the same query in VB.NET:

```
Dim people = context.People _
    .Where(Function(p) p.PersonState =
    PersonState.Inactive)
```

Writing readable, easy to understand code is very important, and native support for enumerations in Entity Framework is very useful for such situations.

Using multiple tables for a single entity

The ability to split an entity across multiple tables plays an important role in scenarios where we have to store **Binary Large Objects (BLOBs)** in the database, which is a commonly occurring situation. Some database administrators like to see BLOBs in a separate table, especially if they are not frequently accessed, in order to optimize a database's physical storage. As we recall, we represent BLOBs (the `varbinary(MAX)` column type in the SQL Server case) as byte arrays in .NET. Entity Framework aims to abstract a developer from storage details, so ideally we do not want our entities to reflect storage specific details. This is where the **entity splitting** feature comes in, which allows us to store one entity in multiple tables, with a subset of properties persisted in each table. Let's say that we want to store a person's photo, which can be a large object, in a separate table. We can use the `Map` method of the `EntityTypeConfiguration` class in order to configure such properties. We will demonstrate how to configure multiple properties, because the syntax is slightly different for two or more properties versus just a single property.

First of all, here is how our `Person` class looks with new properties:

```
public class Person
{
    public byte[] Photo { get; set; }
    public byte[] FamilyPicture { get; set; }
```

The same code in VB.NET looks as follows:

```
Public Class Person
    Property Photo() As Byte()
    Property FamilyPicture() As Byte()
```

In order to split an entity, `Person` in our case, we need to specify the table for each subset of columns, using explicit table names, similarly to what we did previously in this chapter. We will use anonymous types in order to provide property groupings. This code belongs in the configuration class for the `Person` class; `EntityTypeConfiguration` of type `Person` in our case. For example, consider this code snippet:

```
public class PersonMap : EntityTypeConfiguration<Person>
{
    public PersonMap()
    {
        Map(p =>
        {
            p.Properties(ph =>
                new
                {
                    ph.Photo,
                    ph.FamilyPicture
                });
            p.ToTable("PersonBlob");
        });
        Map(p =>
        {
            p.Properties(ph =>
                new
                {
                    ph.Address,
                    ph.BirthDate,
                    ph.FirstName,
                    ph.HeightInFeet,
                    ph.IsActive,
                    ph.LastName,
```

```
        ph.MiddleName,  
        ph.PersonState,  
        ph.PersonTypeId  
    });  
    p.ToTable("Person");  
});  
}  
}
```

We need to omit the actual primary key property, person's identifier, from both mappings because it actually belongs in both tables. We move binary columns to the `PersonBlob` table, keeping the rest of the columns in the `Person` table. We also mapped the complex type property as part of the same approach. Here is how code looks in VB.NET:

```
Public Class PersonMap  
    Inherits EntityTypeConfiguration(Of Person)  
    Public Sub New()  
        Map(Sub(p)  
            p.Properties(Function(m) _  
                New With {  
                    m.Photo,  
                    m.FamilyPicture})  
            p.ToTable("PersonBlob")  
        End Sub)  
        Map(Sub(p)  
            p.Properties(Function(m) _  
                New With {  
                    m.Address,  
                    m.BirthDate,  
                    m.FirstName,  
                    m.HeightInFeet,  
                    m.IsActive,  
                    m.LastName,  
                    m.MiddleName,  
                    m.PersonState,  
                    m.PersonTypeId})  
            p.ToTable("Person")  
        End Sub)  
    End Sub  
End Class
```

We can also do the opposite, that is, map multiple entity types to a single table. This process is called **table splitting** versus **entity splitting** in the preceding example. The use case for this scenario is exactly the same; we want to separate infrequently accessed properties into its own class, but then relate the two classes together. The data is stored in a single table, but only frequently used properties will be accessed by the main entity. The code is exactly the same as we saw previously with any two related entities, except we map both of them to the same table, using the explicit table mapping for both entities with the exact same table name. As a result, we can retrieve one entity, but omit related entities with large column data from the query. When we need to load related large object data, simply use the `Include` method covered previously.

Advanced querying techniques

We talked about querying techniques in previous chapters. The topic of data retrieval is very extensive and requires developers to be thoroughly familiar with everything LINQ has to offer, in order to address scenarios that come up on a daily basis. In this chapter, we will cover many advanced topics. We want to make sure that developers are prepared for the vast majority of tasks they do daily, but some rarely encountered scenarios may not be addressed in this book.

Projections

Projections refer to a process of retrieving a subset of columns from one or more tables in a single query instead of all the columns, as we saw earlier in this book. Projections are very important from the perspective of efficiency and performance. If we only need to present the first and last names of a person to a user of our software, we have no reason to get all the columns from the `Person` table. Now we are faced with a question. How do we represent this data from the object perspective? We can still use the `Person` class, but if we do not populate all the properties, then we will potentially mislead ourselves or other developers. Instead, we have two better options to use for a class to read the data into:

- Use anonymous types
- Use an explicitly defined class that matches our query data

If we consume a projection query's results in the same method as where the query itself is defined, we use anonymous types. **Anonymous types** in .NET are classes that are not explicitly named and their structure is derived from usage. However, if the data is passed around from one method to another, it is much more convenient to define a type for projection query. We will take a look at both approaches.

Another question we want to address is how to include data from multiple tables in a single projection, which is also possible with Entity Framework. Moreover, we need to remember that projections can be combined with sorting and filtering techniques, as well as other querying concepts. Let's demonstrate these concepts with the following problem. We want to select active persons, sorted by their last and first names, but only show the first name, last name, and a person type's name in the resulting set. The `Person` type's name will come from an entity related to a person. Properties that expose the related data are referred to as **association** properties. Let's first use the anonymous type. In order to create a projection using LINQ's method syntax, we need to use the `Select` method, as shown in the following example:

```
var people = context.People
    .Where(p => p.PersonState == PersonState.Active)
    .OrderBy(p => p.LastName)
    .ThenBy(p => p.FirstName)
    .Select(p => new
    {
        p.LastName,
        p.FirstName,
        p.PersonType.TypeName
    });
foreach (var person in people)
{
    Console.WriteLine("{0} {1} {2}",
        person.FirstName, person.LastName, person.TypeName);
}
```

A lot of the preceding code looks familiar already. The `Select` method itself takes one parameter, an expression of a function, where the function parameter is an instance of the source of our query, a `person` instance in our case. The function returns an instance of the result of the selection. In this case, we use an anonymous type with three properties. We do not specify the property names explicitly, so the source's property names are used. We can clearly see this as we look at the code that loops through the results. Here is how the code looks in VB.NET:

```
Dim people = context.People _
    .Where(Function(p) p.PersonState = PersonState.Active) _
    .OrderBy(Function(p) p.LastName) _
    .ThenBy(Function(p) p.FirstName) _
    .Select(Function(p) New With { _
        p.LastName, _
        p.FirstName, _
```

```

        p.PersonType.TypeName _
    })
    For Each person In people
        Console.WriteLine("{0} {1} {2}",
            person.FirstName,
            person.LastName,
            person.TypeName)
    Next

```

This is how the code looks using the query syntax:

```

var query = from onePerson in context.People
            where onePerson.PersonState == PersonState.Active
            orderby onePerson.LastName, onePerson.FirstName
            select new
            {
                Last = onePerson.LastName,
                First = onePerson.FirstName,
                onePerson.PersonType.TypeName
            };

```

Even though we did not define the type that the query returns, we still have the full **IntelliSense** support and strong typing, due to the fact that .NET creates an actual type for our anonymous type declaration. We can also rename the columns in the resulting anonymous type, which is what we did in the preceding query syntax example. Of course, we can rename properties in the same way in statements that use the method syntax. Here is the same code in VB.NET:

```

Dim query = From onePerson In context.People
            Where onePerson.PersonState = PersonState.Active
            Order By onePerson.LastName, onePerson.FirstName
            Select New With { _
                .Last = onePerson.LastName, _
                .First = onePerson.FirstName, _
                onePerson.PersonType.TypeName _
            }

```

Now, let's see how the code changes when an explicit type is used. The only change in both approaches is that in addition to the `New` keyword, we need to specify the actual type. For example, if we convert the query syntax example to use the explicit type, the code will look as follows:

```

var explicitQuery =
    from onePerson in context.People
    where onePerson.PersonState == PersonState.Active

```

```
orderby onePerson.LastName, onePerson.FirstName
select new PersonInfo
{
    LastName = onePerson.LastName,
    FirstName = onePerson.FirstName,
    PersonType = onePerson.PersonType.TypeName,
    PersonId = onePerson.PersonId
};
```

PersonInfo is the type we use to capture the query results into, and we can easily see that it just has a handful of properties that mostly match the `Person` class with the exception of the type name property, `PersonType`, which is a string-based property; for example, consider this code snippet:

```
public class PersonInfo
{
    public int PersonId { get; set; }
    public string PersonType { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Here is how the code looks in VB.NET:

```
Dim explicitQuery = _
    From onePerson In context.People
    Where onePerson.PersonState = PersonState.Active
    Order By onePerson.LastName, onePerson.FirstName
    Select New PersonInfo With { _
        .LastName = onePerson.LastName, _
        .FirstName = onePerson.FirstName, _
        .PersonType = onePerson.PersonType.TypeName, _
        .PersonId = onePerson.PersonId
    }
```

We did not have to do anything special to get the data from a related table, and this is a fact worth noting. We simply walked through the relationship chain to get to the correct data, then included only the data we needed from the related entity, which is the `PersonType` name. We can also select multiple entities into a property, assuming that this property is a collection type. For example, we can include a list of Phones with our data by adding `IEnumerable` of the string `Phones` property to our `PersonInfo` class and selecting the phone numbers as follows:

```
var explicitQuery =
    from onePerson in context.People
```



```

where onePerson.PersonState == PersonState.Active
orderby onePerson.LastName, onePerson.FirstName
select new PersonInfo
{
    LastName = onePerson.LastName,
    FirstName = onePerson.FirstName,
    PersonType = onePerson.PersonType.TypeName,
    PersonId = onePerson.PersonId,
    Phones = onePerson.Phones.Select(ph=>ph.PhoneNumber)
};
foreach (var person in explicitQuery)
{
    Console.WriteLine("{0} {1} {2} {3}",
        person.FirstName, person.LastName,
        person.PersonType, person.PersonId);
    foreach (var phone in person.Phones)
    {
        Console.WriteLine("    " + phone);
    }
}

```

What we see in the preceding code is two nested projection queries. The second one selects only one column from a set of related entities, `Phones`, for each type `person`. This query will result in a complex select statement sent to the backend, but it will only be a single query to execute, although it results in a set of entities, each with another set of related entities. This demonstrates the true power of Entity Framework in solving very complex problems, which would require a significant amount of code if a stored procedure were to be used. Here is how this code looks in VB.NET:

```

Dim explicitQuery = _
    From onePerson In context.People
    Where onePerson.PersonState = PersonState.Active
    Order By onePerson.LastName, onePerson.FirstName
    Select New PersonInfo With { _
        .LastName = onePerson.LastName, _
        .FirstName = onePerson.FirstName, _
        .PersonType = onePerson.PersonType.TypeName, _
        .PersonId = onePerson.PersonId, _
        .Phones = onePerson.Phones.Select( _
            Function(ph) ph.PhoneNumber)
    }
For Each person In explicitQuery
    Console.WriteLine("{0} {1} {2} {3}",
        person.FirstName,

```

```
                person.LastName,  
                person.PersonType,  
                person.PersonId)  
    For Each phone In person.Phones  
        Console.WriteLine("    " + phone)  
    Next  
Next
```

If you like, try to write the preceding query using the method syntax, as an exercise.

Aggregations and grouping

Next, we will discuss how to aggregate the data. Aggregation is a process of getting a summary view of data, and converting a set of data values into a single value. There are many functions that LINQ supports that allow us to aggregate the data. Of course, by extension, Entity Framework supports them as well. It is important to note that aggregation occurs in the backend, not in memory inside .NET runtime, which would be inefficient. It is possible to accidentally aggregate in memory. This occurs when we get the results first in a .NET collection, then aggregate against this collection. To avoid the problem, make sure that you construct a query with aggregations in it and then execute it once. Here are common aggregation methods we use:

- **Count:** This counts the number of entities, typically based on a condition
- **Sum:** This creates a sum, typically of numeric values
- **Min:** This determines a minimum value
- **Max:** This determines a maximum value
- **Average:** This computes an average value

All of these functions can, of course, be combined with other queries. For example, let's find the minimum `BirthDate` of all the people that come in our sample database:

```
var min = context.People.Min(p => p.BirthDate);
```

Let's try to determine the maximum `BirthDate` in VB.NET:

```
Dim min = context.People.Max(Function(p) p.BirthDate)
```

These examples are pretty simple, getting a scalar value aggregate using the `DateTime` type in our example.

The `Count` function optionally accepts a condition that specifies what needs to be evaluated to true in order to be counted, for example, consider this code snippet:

```
var count = context.People.Count(p =>  
    p.PersonState == PersonState.Active);
```

Here is how this count looks in VB.NET:

```
Dim count = context.People.Count( _  
    Function(p) p.PersonState = PersonState.Active)
```

This code computes the total number of people in the table that are in the active state, where the state is stored in the `PersonState` column inside the `People` table.

We can use aggregations inside other queries. For example, let's count the number of phone numbers for each person, modifying the projection query we used in the beginning of the chapter:

```
var people = context.People  
    .Select(p => new  
    {  
        p.LastName,  
        p.FirstName,  
        p.Phones.Count  
    });
```

We used an anonymous type and aggregate function inline to set an additional property on the resulting object. Here is the same code in VB.NET:

```
Dim people = context.People _  
    .Select(Function(p) New With { _  
        p.LastName, _  
        p.FirstName, _  
        p.Phones.Count  
    })
```

We use the method syntax in the preceding examples. We can combine aggregation method calls with the code created, with the query syntax as well.

Advanced query construction

We can chain LINQ methods, calling one after another, as we saw. For example, we can combine ordering and filtering in a single query, by chaining the `OrderBy` and `Where` methods together. All of them return the `IQueryable` interface. It also has a subinterface, `IOrderedQueryable`, that is used to represent results of ordered queries. If we write any query with Entity Framework that returns a set of objects, it can abstractly be thought of as an implementation of `IQueryable`. As a result, we can modify the query, creating a new query, and our changes are not executed until this final query is enumerated. Let's create a query by adding a condition and an order separately, and then applying an aggregate function to the result.

We will compute the sum of persons' heights in our example, as shown in the following code:

```
var query = from onePerson in context.People
            where onePerson.PersonState == PersonState.Active
            select new
            {
                onePerson.HeightInFeet,
                onePerson.PersonId
            };
query = query.OrderBy(p => p.HeightInFeet);
var sum = query.Sum(p => p.HeightInFeet);
Console.WriteLine(sum);
```

Let's walk through this code, as we want to make sure it is clear as to what is going in. Firstly, we create a basic query, filtering in only the active people and selecting a person identifier and height from the `People` collection, which corresponds to the `People` table in the database. What is important to point out now is that no SQL query has executed against the database so far, since we did not yet enumerate the results of our query. Next, we sort the query, using the method syntax. Finally, we access the query result by using aggregate functions. Aggregate functions will cause the final query to execute, the only query that results in SQL statements to be sent to the database. This query will sum up the height of all the active people. Here is how this code looks in VB.NET:

```
Dim query = From onePerson In context.People _
            Where onePerson.PersonState = PersonState.Active
            Select New With { _
                onePerson.HeightInFeet, _
                onePerson.PersonId
            }
query = query.OrderBy(Function(p) p.HeightInFeet)
Dim sum = query.Sum(Function(p) p.HeightInFeet)
Console.WriteLine(sum)
```

We talked about deferred query execution before, but we used to create a query in a single line of code. There is no such limitation in Entity Framework. There is a specific use case to create queries this way. For example, we get a complex criteria object, and we need to check a specific property on our criteria before deciding whether we need to add a filter condition. The ability to create a query in multiple steps may come in handy in such scenarios.

On the other hand, we can also embed such criteria data directly in our query, as shown in the following code snippet:

```
query = from onePerson in context.People
        where !criteria.FilterActive ||
              (criteria.FilterActive &&
               onePerson.PersonState == PersonState.Active)
        select new
        {
            onePerson.HeightInFeet,
            onePerson.PersonId
        };
```

In the preceding example, we are working with a criteria object with the `FilterActive` property. We directly embed checks to see whether this property is set to `true` in our query code and execute the conditional code as a result. This is a very powerful concept that allows us to write code that goes far beyond the simple queries we worked on in previous chapters. Here is how this query looks in VB.NET:

```
query = From onePerson In context.People _
        Where Not criteria.FilterActive Or _
              (criteria.FilterActive And
               onePerson.PersonState = PersonState.Active)
        Select New With { _
            onePerson.HeightInFeet, _
            onePerson.PersonId
        }
```

Entity Framework will interpret the .NET code and will convert it to appropriate SQL syntax. Not all .NET code will work for us in such a fashion, but many commonly used functions will work. For example, many string functions such as `ToUpper`, `ToLower`, and `Contains` will translate into appropriate SQL queries. Boolean and number comparison will work fine as well. Our default approach should be to use .NET code as part of our queries. .NET functions are translated properly via a backend provider as part of SQL query execution and interpretation. Entity Framework also has additional useful functions that are located in the `System.Data.Entity.DbFunctions` class. It exposes many `DateTime` and `string` operations that come in handy in a number of scenarios. For example, this is how we can add days to a particular date as part of our query:

```
query = from onePerson in context.People
        where DbFunctions.AddDays(onePerson.BirthDate, 2) >
              new DateTime(1970,1,1)
        select new
```

```
{
    onePerson.HeightInFeet,
    onePerson.PersonId
};
```

Here is what this code looks like in VB.NET:

```
query = From onePerson In context.People _
        Where DbFunctions.AddDays(onePerson.BirthDate, 2) >
            New DateTime(1970, 1, 1)
        Select New With { _
            onePerson.HeightInFeet, _
            onePerson.PersonId
        }
```

To summarize, we can often find a way to embed .NET native functions inside our queries. As a result, it looks like we are ordinarily writing C# or VB.NET code, but it results in appropriate backend queries.

Paging data with windowing functions


Paging through our data is an extremely common scenario. It allows us to write applications that are efficient and do not consume large amounts of memory. Just imagine what would happen if we try to show in our user interface all the data from a table with millions of rows in it. Entity Framework via LINQ comes with an easy way to accomplish paging. Windowing functions, also known as **paging functions**, have this name because they provide us with a small window into large amounts of data. When we page through data, we deal with just two numbers:

- The page number that we want to show
- The number of rows per page

Entity Framework has two methods, `Skip` and `Take`, to allow developers to page through the data. The `Skip` method allows us to skip some number of rows, essentially rows from previous pages. The `Take` method allows us to retrieve some number of rows. Here is how we can retrieve a page of data from our table. Our `criteria` object contains the current page number and number of rows per page or page size; for example, consider this code snippet:


```
var people = context.People
    .OrderBy(p => p.LastName)
    .Skip((criteria.PageNumber - 1) * criteria.PageSize)
    .Take(criteria.PageSize);
```

We perform simple math to figure out how many rows we need to skip, based on the page number and page size from criteria object.

 We must sort the data before using the `Skip` method, or an exception will be thrown.

Here is how the same code looks in VB.NET:

```
Dim query = context.People _
    .OrderBy(Function(p) p.LastName) _
    .Skip((criteria.PageNumber - 1) * criteria.PageSize) _
    .Take(criteria.PageSize)
```

 We can easily combine paging with other functions, such as filtering.

Using joins

Joins allow developers to combine data from multiple tables into a single query, based on a condition that relates rows from two tables together. We already performed this function in previous examples when we dealt with related tables. Relationships simply abstract out the underlying database joins from developers. Joins come into play when Entity Framework's overall logical model does not specify relationships, for some reason. This may occur when data is not fully normalized or because we are working with a legacy database that we may not be able to change. At that point, we must join unrelated tables based on a condition. The **join** keyword in LINQ translates into the `INNER JOIN` operation in SQL, thus eliminating unmatched rows from the result set. This is how `INNER JOIN` works; we must have at least one matching row in the table on the right-hand side of the join, in order to see data from the table on the left-hand side of the join. Let's examine a simple use case where we manually join a person and person type, retrieving the first and last names of a person along with the type name using joins. Just like in SQL, we will see the same components to construct joins: the left-hand table or collection, right table, join condition, and selection of data based on data available via joins. Here is how this looks using the LINQ query syntax:

```
var people = from person in context.People
             join personType in context.PersonTypes
             on person.PersonTypeId equals personType.PersonTypeId
             select new
             {
```

```
        person.LastName,  
        person.FirstName,  
        personType.TypeName  
    };
```

We need to declare row-level variables for both the joined tables, `person`, and `personType` in the preceding example. We see the new usage of the `equals` keyword that is now used to support join conditions inside LINQ. It is used to specify the relationship condition between the left-hand table, `People`, and the right table, `PersonTypes`. The `PersonTypeId` value must be the same in rows from both tables in order to be included in the result set. Finally, we use the projection to select a subset of columns from both tables using variables for each row. The same code in VB.NET looks very similar, as shown in the following code:

```
Dim people =  
    From person In context.People  
    Join personType In context.PersonTypes  
      On person.PersonTypeId Equals personType.PersonTypeId  
    Select New With  
    {  
        person.LastName,  
        person.FirstName,  
        personType.TypeName  
    }
```

You undoubtedly noticed similarities with SQL syntax. If we switch to the method syntax for LINQ, here is the code we will see:

```
people = context.People  
    .Join(  
        context.PersonTypes,  
        person => person.PersonTypeId,  
        personType => personType.PersonTypeId,  
        (person, type) => new  
        {  
            Person = person,  
            PersonType = type  
        })  
    .Select(p => new  
    {  
        p.Person.LastName,  
        p.Person.FirstName,  
        p.PersonType.TypeName  
    });
```


The `Join` method in LINQ takes the same parameters as any join would: the right-hand table and condition. The left-hand table is what `Join` is called on. The final parameter specifies what needs to be selected out of the join; in our case, we just select all the data values from both the tables. We can use an anonymous type or explicit types, and select a subset of data. Even though we appear to select every column, this is not how SQL is constructed, only data specified in the final `Select` query is retrieved from the database, that is, the person's last and first names and matching type name. Here is the same code in VB.NET:

```
people = context.People _
    .Join(context.PersonTypes, _
        Function(person) person.PersonTypeId, _
        Function(personType) personType.PersonTypeId, _
        Function(person, personType) New With {
            .Person = person,
            .PersonType = personType}) _
    .Select(Function(p) New With {
        p.Person.LastName,
        p.Person.FirstName,
        p.PersonType.TypeName
    })
```

We will see the `LEFT OUTER` join implementation in the next section. The `LEFT OUTER` join allow us to see the data from the table on the left-hand side of the join, even when we have to match rows in the table on the right.

Groupings and left outer joins

So far, we saw simple groupings that aggregate based on a single column and produced a single result. There are many other scenarios where we need to provide more than one column in a grouped or aggregated result set. For example, we want to see how many people we have, based on a month they were born in. We are not starting with a simple example, but this example will give us a thorough understanding of how to group and select data at the same time, as shown in the following code snippet:

```
var query =
    from onePerson in context.People
    group onePerson by new { onePerson.BirthDate.Value.Month }
    into monthGroup
    select new
    {
        Month = monthGroup.Key.Month,
        Count = monthGroup.Count()
    };
```

The query is based on the `Person` collection from the `People` property of our context. We loop through the data based on the person's data, specified in the `onePerson` row variable. We group based on the month of `BirthDate`. We use an anonymous type to demonstrate that we can group based on more than one field, just by adding more properties to this anonymous type, which specifies the group key(s). We also name the group so that we can access its data in final select. We select an anonymous type as well, grabbing key values from the group as well as our aggregate value, using the `Count` function. You can use other aggregate functions in a grouping. Here is how the same looks in VB.NET:

```
Dim query =
    From onePerson In context.People
    Group onePerson By personWithBirthday =
        New With { .Month = onePerson.BirthDate.Value.Month }
        Into monthGroup = Group
    Select New With
        {
            .Count = monthGroup.Count(),
            .Month = personWithBirthday.Month
        }
```

There are some differences from C#. We have to name our group variable (`personWithBirthday`) and use the `Group` keyword when defining the `monthGroup` variable. Let's also take a look at the same grouping functions, using the method syntax:

```
var methodQuery =
    context.People
    .GroupBy(
        onePerson => new { onePerson.BirthDate.Value.Month },
        monthGroup => monthGroup)
    .Select(monthGroup => new
    {
        Month = monthGroup.Key.Month,
        Count = monthGroup.Count()
    });
```

The `GroupBy` method in LINQ typically takes two parameters; that is, the group definition or key fields and the selector that will be used when retrieving group data. We are quite familiar with the `Select` method now, and we can use our group variable to select not only the group's key fields, but also additional aggregates based on the group definition. Here is how the same query looks in VB.NET:

```
Dim methodQuery =
    context.People _
```

```

        .GroupBy(Function(person) New With { .Month =
person.BirthDate.Value.Month}, _
            Function(monthGroup) monthGroup) _
        .Select(Function(monthGroup) New With
            {
                .Month = monthGroup.Key.Month,
                .Count = monthGroup.Count()
            })

```

Let's now take a look at LEFT OUTER joins with LINQ. Unfortunately, there is no keyword in LINQ that supports LEFT OUTER joins out of the box. Instead, we have to use grouping with a new keyword: `DefaultIfEmpty`. This function returns the default for the right table's selectable data if it is missing, typically just the null value. Another new keyword we will see is `GroupJoin`, which correlates two sets of data based on a condition, thus returning the results of the join. Let's take a look at an example. Our `Person` object supports null for the `person` type value. So, let's select a few columns from the `Person` table and `person` type, using the "Unknown" string when the `person` type is null, as shown in this code snippet:

```

var query =
    from person in context.People
    join personType in context.PersonTypes
        on person.PersonTypeId equals
personType.PersonTypeId into finalGroup
    from groupedData in finalGroup.DefaultIfEmpty()
    select new
    {
        person.LastName,
        person.FirstName,
        TypeName = groupedData.TypeName ?? "Unknown"
    };

```

Let's read this statement. We are starting with the `People` dataset. We join the `person` types based on the `person` type identifying, grouping the data based on `person`. In the next line, we specify that we want to see the default value for the `person` type inside the group, if the joined data from the right-hand table (the `person` type) is missing. This is what the `DefaultIfEmpty` method does. Finally, we select from the grouped data and primary table's data, coalescing the missing `person` type with the "Unknown" word. Here is how the same code looks in VB.NET:

```

Dim query =
    From person In context.People
    Group Join personType In context.PersonTypes
        On person.PersonTypeId Equals personType.PersonTypeId Into
finalGroup = Group

```

```
From groupedData In finalGroup.DefaultIfEmpty()
Select New With
{
    .LastName = person.LastName,
    .FirstName = person.FirstName,
    .TypeName = If(groupedData.TypeName Is Nothing, "Unknown",
groupedData.TypeName)
}
```

VB.NET code is slightly different. We still start with the people set. We join with the person type while grouping the data based on person. We have to use the `Group` keyword to specify that we want everything from the `Group` type. In the next line, we see the second `from` keyword, selecting our grouped data. In the next few lines, we select the person data and coalesce the missing type with the word "Unknown".

We have not seen two `from` keywords in the same query before. This is possible, and in terms of SQL, similar to the older, pre-JOIN syntax, where we can specify multiple tables in the `From` portion of our SQL statement and specify the join condition inside the `WHERE` portion.

Let's take a look at the same statement using the method syntax. In this case, both C# and VB.NET code use the `GroupJoin` method:

```
var methodQuery = context.People
    .GroupJoin(
        context.PersonTypes,
        person => person.PersonTypeId,
        personType => personType.PersonTypeId,
        (person, type) => new
        {
            Person = person,
            PersonType = type
        })
    .SelectMany(groupedData =>
        groupedData.PersonType.DefaultIfEmpty(),
        (group, personType) => new
        {
            group.Person.LastName,
            group.Person.FirstName,
            TypeName = personType.TypeName ?? "Unknown"
        });
```

We see something new in this code, so let's walk through it. We start with `person` again, and then we join with `person` types, based on the person type identifier. We then select both `person` and `personType` data in the grouped results, based on `person`. Next, we see the `SelectMany` keyword. We project a specified function, `DefaultIfEmpty`, onto the grouped results, and then select some data from `person` and `person` type, substituting null with the word "Unknown", where `personType` is missing.

Here is how this code looks in VB.NET:

```
Dim methodQuery =
    context.People _
        .GroupJoin(
            context.PersonTypes,
            Function(person) person.PersonTypeId,
            Function(personType) personType.PersonTypeId,
            Function(person, type) New With
            {
                .Person = person,
                .PersonType = type
            }) _
        .SelectMany(Function(groupedData) _
            groupedData.PersonType.DefaultIfEmpty(),
            Function(group, personType) New With
            {
                .LastName = group.Person.LastName,
                .FirstName = group.Person.FirstName,
                .TypeName = If (personType.TypeName Is Nothing,
                    "Unknown", personType.TypeName)
            })
```

`SelectMany` has another usage in LINQ. In addition to LEFT OUTER joins, it is also used in selecting many child and parent records into a single result set. Previously, we saw that if we select `person` and `Phones`, we see one `person` object with a collection of phones in the `Phones` property. We could unroll this data with `SelectMany` and select a result where `person` data is repeated with each child that belongs to the parent in question; for example, seeing the first name of a person and their phone number, where the person's first name is repeated for each phone. This is shown in the following code snippet:

```
var query =
    from onePerson in context.People
    from onePhone in onePerson.Phones
    orderby onePerson.LastName, onePhone.PhoneNumber
```

```
select new
{
    onePerson.LastName,
    onePerson.FirstName,
    onePhone.PhoneNumber
};
```

We see familiar code here. We use the `from` keyword twice to bring two sets of records together. We sort them. Then, we select some data from both sets using a projection. Here is the same query in VB.NET:

```
Dim query =
    From onePerson In context.People
    From onePhone In onePerson.Phones
    Order By onePerson.LastName, onePhone.PhoneNumber
    Select New With
    {
        onePerson.LastName,
        onePerson.FirstName,
        onePhone.PhoneNumber
    }
```

We can also write the same statement using the method syntax. This time, use the `SelectMany` function. This function takes an expression with related data, `Phones` in our case, which is applied to another set of data, the `person` set in our case. The second parameter to the function is the selector that is the output of the final result; for example, consider this code snippet:

```
var methodQuery =
    context.People
        .SelectMany(person => person.Phones, (person, phone) =>
new
    {
        person.LastName,
        person.FirstName,
        phone.PhoneNumber
    })
    .OrderBy(p => p.LastName)
    .ThenBy(p => p.PhoneNumber);
```

Here is how the code looks in VB.NET:

```
Dim methodQuery =
    context.People _
        .SelectMany( _
```

```

        Function(person) person.Phones, _
        Function(person, phone) New With
    {
        person.LastName,
        person.FirstName,
        phone.PhoneNumber
    }) _
    .OrderBy(Function(p) p.LastName) _
    .ThenBy(Function(p) p.PhoneNumber)

```

Set operations

LINQ supports the following set operators:

- Distinct
- Union
- Intersect
- Except

`Distinct`, just as the same implies, returns a set of unique values across a set of data. For example, let's determine all unique person types across all of the people in the table:

```

var uniqueQuery = context.People
    .Select(p => p.PersonType.TypeName)
    .Distinct();

```

It is important to note that the distinct selection is not limited to a single field as in the preceding example. We can apply the `Distinct` operator to results that contain many columns or properties, both anonymous and explicit types. Here is how the same code looks in VB.NET:

```

Dim uniqueQuery = context.People _
    .Select(Function(p) p.PersonType.TypeName) _
    .Distinct()

```

We can also perform unions of multiple queries. `Union` combines results of two queries into a single result set. For example, if we want to select all names of both `People` and `Companies` into a single results set, we can do something like the following:

```

var unionQuery = context.People
    .Select(p => new
    {

```

```
        Name = p.LastName + " " + p.FirstName,
        RowType = "Person"
    })
    .Union(context.Companies.Select(c => new
    {
        Name = c.CompanyName,
        RowType = "Company"
    })))
    .OrderBy(result => result.RowType)
    .ThenBy(result => result.Name);
```

We are using an anonymous type in this case with two properties: name and record type. We see a new interesting capability, that is, the ability to concatenate strings as part of a query. This code is executed by the backend, as is the case in all the examples we covered so far. Entity Framework is smart enough to handle such use cases. We also apply a sort order to the result set. Here is how the code looks in VB.NET:

```
Dim unionQuery = context.People _
    .Select(Function(p) New With
    {
        .Name = p.LastName + " " + p.FirstName,
        .RowType = "Person"
    }) _
    .Union(context.Companies.Select(Function(c) New With
    {
        .Name = c.CompanyName,
        .RowType = "Company"
    }))) _
    .OrderBy(Function(result) result.RowType) _
    .ThenBy(Function(result) result.Name)
```

Intersect looks for common data between two queries. Except looks for differences between two query input sets. Syntactically, there is no difference between Union and Intersect. So, to exercise this idea, try to replace the Union keyword with the Intersect or Except keyword and, examine the results.

Self-test questions

Q1. Which base class do you use to configure a class used to contain a number of properties that are common to multiple entities?

1. EntityTypeConfiguration
2. ComplexTypeConfiguration
3. CommonTypeConfiguration

Q2. You must have table names and class names that always match in Entity Framework, true or false?

Q3. Every property in an entity is always persisted to the database, true or false?

Q4. What is the name of the technique that involves selecting a subset of columns from a table via a query in Entity Framework?

1. Projection
2. Subquery
3. You cannot do this in Entity Framework

Q5. You must declare a result type to dynamically select columns from multiple tables via a single query, true or false?

Q6. You must use the **Join** keyword to select data from related tables in a single query, true or false?

Q7. Which function can be used in a method syntax query to repeat parent data along with unique child data in a single query result?

1. Select
2. GroupJoin
3. SelectMany

Q8. Which LINQ method can be used to find all unique values for one column in a table?

1. Distinct
2. Unique
3. Union

Q9. Which single LINQ method can be used to accomplish the selection of data from multiple tables, that is similar to LEFT OUTER JOIN in SQL?

1. Join
2. LeftJoin
3. RightJoin
4. You have to use a combination of methods, as no single method exists

Q10. Which two methods can you use to page through the data?

1. Miss and Yield
2. Skip and Take
3. Group and Take

Q11. You cannot create grouping queries based on multiple fields from a table, true or false?

Summary

We covered a lot of ground in this chapter. Since practice makes perfect, I always recommend that everyone tries out the covered concepts by writing code and experimenting. We looked at some new modeling techniques. We saw that we can create additional classes, called complex types, to group properties common to multiple entities to ensure consistency in our models. We saw that using enumerations in models can lead to more readable code, and were convinced that Entity Framework has first-class support for enumerations. We saw that we do not have to have the names of our classes and properties match database structure exactly. We could use the `HasColumnName` and `ToTable` methods to provide alternative names at the database level.

We looked at many advanced query techniques. Most concepts can be used with both the query and method syntax of LINQ. We used projections to select subsets of columns from one or more tables in a single query. We saw how we can accomplish projection queries with anonymous and explicit types. We saw how we can aggregate the data to compute maximum, minimum, and a sum of data to get single values. We were also able to group the data in more advanced ways, providing grouping by many properties, and aggregates inside the same result set, using the `GroupBy` method. We looked at how we can use the same technique in order to accomplish the LEFT OUTER join functionality, and the ability to retrieve the data from the left-hand table, even though there are no matching rows in the table on the right. We saw how `SelectMany` can be used to create a functionality similar to JOIN in SQL, where some of the data from one table is repeated in matching child rows in the result set. Finally, we took a look at set operations that allow us to find distinct data as well as combine data from multiple queries via union operations.

In the next chapter, we will look at working with stored procedures and views, and database artifacts that do not map directly to entities, as tables do. We will also see how to handle concurrency, a circumstance where multiple users try to update the same data at the same time. We will understand the advantages of using Entity Framework's asynchronous API.

6

Working with Views, Stored Procedures, the Asynchronous API, and Concurrency

In this chapter, you will learn how to integrate Entity Framework with additional database objects, specifically views and stored procedures. We will see how to take advantage of existing stored procedures and functions to retrieve and change the data. You will learn how to persist changed entities from our context using stored procedures. We will gain an understanding of the advantages of asynchronous processing and see how Entity Framework supports this concept via its built-in API. Finally, you will learn why concurrency is important for a multi-user application and what options are available in Entity Framework to implement optimistic concurrency.

In this chapter, we will cover how to:

- Get data from a view
- Get data from a stored procedure or table-valued function
- Map create, update, and delete operations on a table to a set of stored procedures
- Use the asynchronous API to get and save the data
- Implement multi-user concurrency handling

Working with views

Views in an RDBMS fulfill an important role. They allow developers to combine data from multiple tables into a structure that looks like a table, but do not provide persistence. Thus, we have an abstraction on top of raw table data. One can use this approach to provide different security rights, for example. We can also simplify queries we have to write, especially if we access the data defined by views quite frequently in multiple places in our code. Entity Framework Code-First does not fully support views as of now. As a result, we have to use a workaround. One approach would be to write code as if a view was really a table, that is, let Entity Framework define this table, then drop the table, and create a replacement view. We will still end up with strongly typed data with full query support. Let's start with the same database structure we used before, including person and person type. Our view will combine a few columns from the `Person` table and `Person` type name, as shown in the following code snippet:

```
public class PersonViewInfo
{
    public int PersonId { get; set; }
    public string TypeName { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Here is the same class in VB.NET:

```
Public Class PersonViewInfo
    Public Property PersonId() As Integer
    Public Property TypeName() As String
    Public Property FirstName() As String
    Public Property LastName() As String
End Class
```

Now, we need to create a configuration class for two reasons. We need to specify a primary key column because we do not follow the naming convention that Entity Framework assumes for primary keys. Then, we need to specify the table name, which will be our view name, as shown in the following code:

```
public class PersonViewInfoMap :
    EntityTypeConfiguration<PersonViewInfo>
{
    public PersonViewInfoMap()
    {
        HasKey(p => p.PersonId);
       .ToTable("PersonView");
    }
}
```

Here is the same class in VB.NET:

```
Public Class PersonViewInfoMap
    Inherits EntityTypeConfiguration(Of PersonViewInfo)
    Public Sub New()
        HasKey(Function(p) p.PersonId)
       .ToTable("PersonView")
    End Sub
End Class
```

Finally, we need to add a property to our context that exposes this data, as shown here:

```
public DbSet<PersonViewInfo> PersonView { get; set; }
```

The same property in VB.NET looks quite familiar to us, as shown in the following code:

```
Property PersonView() As DbSet(Of PersonViewInfo)
```

Now, we need to work with our initializer to drop the table and create a view in its place. We are using one of the initializers we created before. When we cover migrations, we will see that the same approach works there as well, with virtually identical code. Here is the code we added to the `Seed` method of our initializer, as shown in the following code:

```
public class Initializer :
    DropCreateDatabaseIfModelChanges<Context>
{
    protected override void Seed(Context context)
    {
        context.Database.ExecuteSqlCommand("DROP TABLE
PersonView");
        context.Database.ExecuteSqlCommand(
            @"CREATE VIEW [dbo].[PersonView]
AS
SELECT
    dbo.People.PersonId,
    dbo.People.FirstName,
    dbo.People.LastName,
    dbo.PersonTypes.TypeName
FROM
    dbo.People
INNER JOIN dbo.PersonTypes
    ON dbo.People.PersonTypeId =
dbo.PersonTypes.PersonTypeId
");
    }
}
```

In the preceding code, we first drop the table using the `ExecuteSqlCommand` method of the `Database` object. This method is useful because it allows the developer to execute arbitrary SQL code against the backend. We call this method twice, the first time to drop the tables and the second time to create our view.

The same initializer code in VB.NET looks as follows:

```
Public Class Initializer
    Inherits DropCreateDatabaseIfModelChanges(Of Context)
    Protected Overrides Sub Seed(ByVal context As Context)
        context.Database.ExecuteSqlCommand("DROP TABLE
PersonView")
        context.Database.ExecuteSqlCommand( <![CDATA[
CREATE VIEW [dbo].[PersonView]
AS
SELECT
    dbo.People.PersonId,
    dbo.People.FirstName,
    dbo.People.LastName,
    dbo.PersonTypes.TypeName
FROM
    dbo.People
INNER JOIN dbo.PersonTypes
ON dbo.People.PersonTypeId =
dbo.PersonTypes.PersonTypeId]] >.Value())
    End Sub
End Class
```

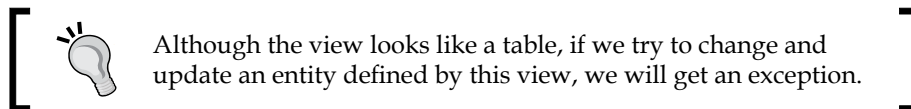
Since VB.NET does not support multiline strings such as C#, we are using XML literals instead, getting a value of a single node. This just makes SQL code more readable.

We are now ready to query our data. This is shown in the following code snippet:

```
using (var context = new Context())
{
    var people = context.PersonView
        .Where(p => p.PersonId > 0)
        .OrderBy(p => p.LastName)
        .ToList();
    foreach (var personViewInfo in people)
    {
        Console.WriteLine(personViewInfo.LastName);
    }
}
```

As we can see, there is literally no difference in accessing our view or any other table. Here is the same code in VB.NET:

```
Using context = New Context()
    Dim people = context.PersonView _
        .Where(Function(p) p.PersonId > 0) _
        .OrderBy(Function(p) p.LastName) _
        .ToList()
    For Each personViewInfo In people
        Console.WriteLine(personViewInfo.LastName)
    Next
End Using
```



If we do not want to play around with tables in such a way, we can still use the initializer to define our view, but query the data using a different method of the Database object, `SqlQuery`. This method has the same parameters as `ExecuteSqlCommand`, but is expected to return a result set, in our case, a collection of `PersonViewInfo` objects, as shown in the following code:


```
using (var context = new Context())
{
    var sql = @"SELECT * FROM PERSONVIEW WHERE PERSONID > {0} ";
    var peopleViaCommand =
context.Database.SqlQuery<PersonViewInfo>(
    sql,
    0);
    foreach (var personViewInfo in peopleViaCommand)
    {
        Console.WriteLine(personViewInfo.LastName);
    }
}
```

The `SqlQuery` method takes generic type parameters, which define what data will be materialized when a raw SQL command is executed. The text of the command itself is simply parameterized SQL. We need to use parameters to ensure that our dynamic code is not subject to SQL injection. **SQL injection** is a process in which a malicious user can execute arbitrary SQL code by providing specific input values. Entity Framework is not subject to such attacks on its own. Here is the same code in VB.NET:

```
Using context = New Context()
    Dim sql = "SELECT * FROM PERSONVIEW WHERE PERSONID > {0} "
```

```
Dim peopleViaCommand = context.Database.SqlQuery(Of
PersonViewInfo)(sql, 0)
    For Each personViewInfo In peopleViaCommand
        Console.WriteLine(personViewInfo.LastName)
    Next
End Using
```

We not only saw how to use views in Entity Framework, but saw two extremely useful methods of the `Database` object, which allows us to execute arbitrary SQL statements and optionally materialize the results of such queries. The generic type parameter does not have to be a class. You can use the native .NET type, such as a string or an integer.

 It is not always necessary to use views. Entity Framework allows us to easily combine multiple tables in a single query.

Working with stored procedures

The process of working with stored procedures in Entity Framework is similar to the process of working with views. We will use the same two methods we just saw on the `Database` object—`SqlQuery` and `ExecuteSqlCommand`. In order to read a number of rows from a stored procedure, we simply need a class that we will use to materialize all the rows of retrieved data into a collection of instances of this class. For example, to read the data from the stored procedure, consider this query:

```
CREATE PROCEDURE [dbo].[SelectCompanies]
    @dateAdded as DateTime
AS
BEGIN
    SELECT CompanyId, CompanyName
    FROM Companies
    WHERE DateAdded > @dateAdded
END
```

We just need a class that matches the results of our stored procedure, as shown in the following code:

```
public class CompanyInfo
{
    public int CompanyId { get; set; }
    public string CompanyName { get; set; }
}
```


The same class looks as follows in VB.NET:

```
Public Class CompanyInfo
    Property CompanyId() As Integer
    Property CompanyName() As String
End Class
```

We are now able to read the data using the `SqlQuery` method, as shown in the following code:

```
sql = @"SelectCompanies {0}";
var companies = context.Database.SqlQuery<CompanyInfo>(
    sql,
    DateTime.Today.AddYears(-10));
foreach (var companyInfo in companies)
{
```

We specified which class we used to read the results of the query call. We also provided a formatted placeholder when we created our SQL statement for a parameter that the stored procedure takes. We provided a value for that parameter when we called `SqlQuery`. If one has to provide multiple parameters, one just needs to provide an array of values to `SqlQuery` and provide formatted placeholders, separated by commas as part of our SQL statement. We could have used a table values function instead of a stored procedure as well. Here is how the code looks in VB.NET:

```
sql = "SelectCompanies {0}"
Dim companies = context.Database.SqlQuery(Of CompanyInfo)(
    sql,
    DateTime.Today.AddYears(-10))
For Each companyInfo As CompanyInfo In companies
```

Another use case is when our stored procedure does not return any values, but instead simply issues a command against one or more tables in the database. It does not matter as much what a procedure does, just that it does not need to return a value. For example, here is a stored procedure that updates multiple rows in a table in our database:

```
CREATE PROCEDURE dbo.UpdateCompanies
    @dateAdded as DateTime,
    @activeFlag as Bit
AS
BEGIN
    UPDATE Companies
    Set DateAdded = @dateAdded,
        IsActive = @activeFlag
END
```

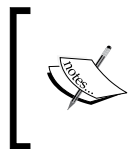
In order to call this procedure, we will use `ExecuteSqlCommand`. This method returns a single value – the number of rows affected by the stored procedure or any other SQL statement. You do not need to capture this value if you are not interested in it, as shown in this code snippet:

```
var sql = @"UpdateCompanies {0}, {1}";
var rowsAffected =
    context.Database.ExecuteSqlCommand(
        sql, DateTime.Now, true);
```

We see that we needed to provide two parameters. We needed to provide them in the exact same order the stored procedure expected them. They are passed into `ExecuteSqlCommand` as the parameter array, except we did not need to create an array explicitly. Here is how the code looks in VB.NET:

```
Dim sql = "UpdateCompanies {0}, {1}"
Dim rowsAffected =
    context.Database.ExecuteSqlCommand( _
        sql, DateTime.Now, True)
```

Entity Framework eliminates the need for stored procedures to a large extent. However, there may still be reasons to use them. Some of the reasons include security standards, legacy database, or efficiency. For example, you may need to update thousands of rows in a single operation and retrieve them through Entity Framework; updating each row at a time and then saving those instances is not efficient. You could also update data inside any stored procedure, even if you call it with the `SqlQuery` method.



Developers can also execute any arbitrary SQL statements, following the exact same technique as stored procedures. Just provide your SQL statement, instead of the stored procedure name to the `SqlQuery` or `ExecuteSqlCommand` method.

Create, update, and delete entities with stored procedures

So far, we have always used the built-in functionality that comes with Entity Framework that generates SQL statements to insert, update, or delete the entities. There are use cases when we would want to use stored procedures to achieve the same result. Developers may have requirements to use stored procedures for security reasons. You may be dealing with an existing database that has these procedures already built in.

Entity Framework Code-First now has full support for such updates. We can configure the support for stored procedures using the familiar `EntityTypeConfiguration` class. We can do so simply by calling the `MapToStoredProcedures` method. Entity Framework will create stored procedures for us automatically if we let it manage database structures. We can override a stored procedure name or parameter names, if we want to, using appropriate overloads of the `MapToStoredProcedures` method. Let's use the `Company` table in our example:

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

If we just run the code to create or update the database, we will see new procedures created for us, named `Company_Insert` for an insert operation and similar names for other operations. Here is how the same class looks in VB.NET:

```
Public Class CompanyMap
    Inherits EntityTypeConfiguration(Of Company)
    Public Sub New()
        MapToStoredProcedures()
    End Sub
End Class
```

Here is how we can customize our procedure names if necessary:

```
public class CompanyMap :
    EntityTypeConfiguration<Company>
{
    public CompanyMap()
    {
        MapToStoredProcedures(config =>
            {
                config.Delete(
                    procConfig =>
                    {
                        procConfig.HasName("CompanyDelete");
                        procConfig.Parameter(company =>
company.CompanyId, "companyId");
                    });
            });
    }
}
```

```
        config.Insert(procConfig =>
procConfig.HasName("CompanyInsert"));
        config.Update(procConfig =>
procConfig.HasName("CompanyUpdate"));
    });
}
```

In this code, we performed the following:

- Changed the stored procedure name that deletes a company to `CompanyDelete`
- Changed the parameter name that this procedure accepts to `companyId` and specified that the value comes from the `CompanyId` property
- Changed the stored procedure name that performs insert operations on `CompanyInsert`
- Changed the stored procedure name that performs updates to `CompanyUpdate`

Here is how the code looks in VB.NET:

```
Public Class CompanyMap
    Inherits EntityTypeConfiguration(Of Company)
    Public Sub New()
        MapToStoredProcedures( _
            Sub(config)
                config.Delete(
                    Sub(procConfig)
                        procConfig.HasName("CompanyDelete")
                        procConfig.Parameter(Function(company)
company.CompanyId, "companyId")
                    End Sub
                )
                config.Insert(Function(procConfig)
procConfig.HasName("CompanyInsert"))
                config.Update(Function(procConfig)
procConfig.HasName("CompanyUpdate"))
            End Sub
        )
    End Sub
End Class
```

Of course, if you do not need to customize the names, your code will be much simpler.

The asynchronous API

So far, all of our database operations with Entity Framework have been synchronous. In other words, our .NET program waited for any given database operation, such as a query or an update, to complete before moving forward. In many cases, there is nothing wrong with this approach. There are use cases, however, where an ability to perform such operations asynchronously is important. In these cases, we let .NET use its execution thread while the software waits for the database operation to complete. For example, if you are creating a web application utilizing the asynchronous approach, we can be more efficient with server resources, releasing web worker threads back to the thread pool while we are waiting for the database to finish processing a request, whether it is a save or retrieve operation. Even in a desktop application, the asynchronous API is useful because the user can potentially perform other tasks in the application, instead of waiting on a possibly time-consuming query or save operation. In other words, the .NET thread does not need to wait for a database thread to complete its work. In a number of applications, the asynchronous API does not provide benefits and could even be harmful from the performance perspective due to the thread context switching. Before using the asynchronous API, developers need to make sure it will benefit them.

Entity Framework exposes a number of asynchronous operations. By convention, all such methods end with the `Async` suffix. For save operations, we can use the `SaveChangesAsync` method on `DbContext`. There are many methods for query operations. For example, many aggregate functions have asynchronous counterparts, such as `SumAsync` or `AverageAsync`. We can also asynchronously read a result set into a list or an array using `ToListAsync` or `ToArrayAsync`, respectively. Also, we can enumerate through the results of a query using `ForEachAsync`. Let's look at a few examples.

This is how we can get the list of objects from a database asynchronously:

```
private static async Task<IEnumerable<Company>>
GetCompaniesAsync()
{
    using (var context = new Context())
    {
        return await context.Companies
            .OrderBy(c => c.CompanyName)
            .ToListAsync();
    }
}
```

It is important to notice that we follow typical **async/await** usage patterns. Our function is flagged as **async** and returns a task object, specifically a task of a list of companies. We create a `DbContext` object inside our function. Then, we create creating a query that returns all companies, ordered by their names. Then, we return the results of this query wrapped inside an asynchronous list generation. We have to await this return value, as we need to follow **async/await** patterns. Here is how this code looks in VB.NET:

```
Private Async Function GetCompaniesAsync() As Task(Of
IEnumerable(Of Company))
    Using context = New Context()
        Return Await context.Companies.OrderBy( _
            Function(c) c.CompanyName).ToListAsync()
    End Using
End Function
```



Any Entity Framework query can be converted to its asynchronous version using `ToToListAsync` or `ToAsyncArray`.

Next, let's create a new record asynchronously:

```
private static async Task<Company> AddCompanyAsync(Company
company)
{
    using (var context = new Context())
    {
        context.Companies.Add(company);
        await context.SaveChangesAsync();
        return company;
    }
}
```

Again, we are wrapping the operation inside the **async** function. We are accepting a parameter, `company` in our case. We are adding this `company` to the context. Finally, we save asynchronously and return the saved `company`. Here is how the code looks in VB.NET:

```
Private Async Function AddCompanyAsync(company As Company) As
Task(Of Company)
    Using context = New Context()
        context.Companies.Add(company)
        Await context.SaveChangesAsync()
        Return company
    End Using
End Function
```

Next, we can locate a record asynchronously. We can use any number of methods here, such as `Single` or `First`. Both of them have asynchronous versions. We will use the `Find` method in our example, as shown in the following code snippet:

```
private static async Task<Company> FindCompanyAsync(int companyId)
{
    using (var context = new Context())
    {
        return await context.Companies
            .FindAsync(companyId);
    }
}
```

We see the familiar asynchronous pattern in this code snippet as well. We just use `FindAsync`, which takes the exact same parameters as the synchronous version. In general, all asynchronous methods in Entity Framework have the same signature, as far as parameters are concerned, as their synchronous counterparts.

Here is how the same method looks in VB.NET:

```
Private Async Function FindCompanyAsync(companyId As Integer) As
Task(Of Company)
    Using context = New Context()
        Return Await context.Companies.FindAsync(companyId)
    End Using
End Function
```

As we mentioned before, aggregate functions have the `async` versions as well. For example, here is how we compute count asynchronously:

```
private static async Task<int> ComputeCountAsync()
{
    using (var context = new Context())
    {
        return await context.Companies
            .CountAsync(c => c.IsActive);
    }
}
```

We use the `CountAsync` method and pass in a condition, which is exactly what we would have done if we were to call the synchronous version, the `Count` function. Here is how the code looks in VB.NET:

```
Private Async Function ComputeCountAsync() As Task(Of Integer)
    Using context = New Context()
        Return Await context.Companies.CountAsync( _
```

```
        Function(c) c.IsActive)
    End Using
End Function
```

If you would like to loop asynchronously through query results, you can also use `ForEachAsync`, which you can attach to any query as well. For example, here is how we can loop through `Companies`:

```
private static async Task LoopAsync()
{
    using (var context = new Context())
    {
        await context.Companies.ForEachAsync(c =>
        {
            c.IsActive = true;
        });
        await context.SaveChangesAsync();
    }
}
```

In the preceding code, we run through all type `Companies`, but we could have just as easily run through a result of any query, after applying an order or a filter. Here is what this method looks like in VB.NET:

```
Private Async Function LoopAsync() As Task
    Using context = New Context()
        Await context.Companies.ForEachAsync( _
            Sub(c)
                c.IsActive = True
            End Sub)
        Await context.SaveChangesAsync()
    End Using
End Function
```

We followed the accepted naming conventions, adding the `Async` suffix to our asynchronous functions. Usually, these functions would be called from a method that is also flagged as `async` and we would have awaited a result of our asynchronous functions. If this is not possible, we can always use the `Task` API and wait for a task to complete. For example, we can access the result of a task, causing the current thread to pause and let the task finish executing, as shown in the following code snippet:

```
Console.WriteLine(
    FindCompanyAsync(companyId).Result.CompanyName);
```


In this example, we call the previously defined function and then access the `Result` property of the task to cause the asynchronous function to finish executing. This is how the code would look in VB.NET:

```
Console.WriteLine(FindCompanyAsync(companyId).Result.CompanyName)
```

When deciding whether or not to use the asynchronous API, we need to research and make sure that there is a reason to do so. We should also ensure that the entire calling chain of methods is asynchronous to gain maximum coding benefits. Finally, resort to the Task API when you need to.

Handling concurrency

Most applications have to deal with concurrency. **Concurrency** is a circumstance where two users modify the same entity at the same time. There are two types of concurrency handling: optimistic and pessimistic. There is no concurrency where the last user always wins. When this happens, there is a silent data loss, where the first user's changes are overwritten without notice, so it is not frequently used. In the case of **pessimistic** concurrency, only one user can edit a record at a time and the second user gets an error, stating that they cannot make any changes at that time. Although this approach is safe, it does not scale well and results in poor user experience. As a result, most applications use **optimistic** concurrency, allowing multiple users to make changes, but checking for a concurrency situation at the time changes are being saved. At that time if two users changed the same row of data, applications issue an error to the second user, letting them know that they need to redo the changes. Some developers at times go an extra mile and assist users in redoing their changes. Entity Framework comes with a built-in optimistic concurrency API. A developer has to pick a column that will play the role of the row version. The row version is incremented every time the row of data is updated. Any time an update query is issued against a row with concurrency columns, the current row version is put into the `where` clause; thus if data has changed since it was first retrieved, no rows are updated as the result of such SQL statement. Entity Framework checks the number of rows updated, and if this number is not 1, a concurrency exception is thrown. In the case of the SQL Server `RowVersion`, also known as `TimeStamp`, a column is used for concurrency. SQL Server automatically increments this column for all updates to each row. The matching type for the `TimeStamp` column in SQL Server is `byte array` in .NET. Let's start by updating our `Person` object to support concurrency. We are going to omit some properties for brevity, as shown in the following code snippet:

```
public class Person
{
    public int PersonId { get; set; }
    public byte[] RowVersion { get; set; }
}
```

We added a new property called `RowVersion` using the `Byte` array as the type. Here is how this change looks in VB.NET:

```
Public Class Person
    Property PersonId() As Integer
    Property RowVersion() As Byte()
End Class
```

We also need to configure this property using our `EntityTypeConfiguration` class to let Entity Framework know that we added a concurrency property, as shown in the following code snippet:

```
public class PersonMap : EntityTypeConfiguration<Person>
{
    public PersonMap()
    {
        Property(p => p.RowVersion)
            .IsFixedLength()
            .HasMaxLength(8)
        .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Computed)
            .IsRowVersion();
    }
}
```

We omitted some properties again, but configured `RowVersion` to be our concurrency column. We flagged it as such by calling the `IsRowVersion` method, as well as configuring the size for SQL Server and flagging it for Entity Framework as database generated. Technically, we only need to call the `IsRowVersion` method, but this code makes it clear as to how the property is configured. We can and should remove other method calls, as they are not needed. Here is how VB.NET code looks:

```
Public Class PersonMap
    Inherits EntityTypeConfiguration(Of Person)

    Public Sub New()
        Me.Property(Function(p) p.RowVersion) _
            .IsFixedLength() _
            .HasMaxLength(8) _
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.
Computed) _
            .IsRowVersion()
    End Sub
End Class
```

Now we are ready to write some code to ensure our concurrency configuration works. It is hard to simulate two users in a single routine, so we will play some tricks, using the knowledge we gained previously, as shown in the following code:

```
private static void ConcurrencyExample()
{
    var person = new Person
    {
        BirthDate = new DateTime(1970, 1, 2),
        FirstName = "Aaron",
        HeightInFeet = 6M,
        IsActive = true,
        LastName = "Smith"
    };
    int personId;
    using (var context = new Context())
    {
        context.People.Add(person);
        context.SaveChanges();
        personId = person.PersonId;
    }
    //simulate second user
    using (var context = new Context())
    {
        context.People.Find(personId).IsActive = false;
        context.SaveChanges();
    }
    //back to first user
    try
    {
        using (var context = new Context())
        {
            context.Entry(person).State = EntityState.Unchanged;
            person.IsActive = false;
            context.SaveChanges();
        }
        Console.WriteLine("Concurrency error should occur!");
    }
    catch (DbUpdateConcurrencyException)
    {
        Console.WriteLine("Expected concurrency error");
    }
    Console.ReadKey();
}
```

This method is a bit lengthy, so let's walk through it. In the first few lines, we created a new person instance and added it to the database by adding it to the `People` collection, and then calling `SaveChanges` on our context. We then pretend to be a second user, updating the same row by calling the `Find` method, changing one property, and then issuing the `SaveChanges` call. This action will increment the row version inside the database. Next, we are pretended to be the first user, using the original person instance that still has the original row version value. We set the state to unmodified, thus attaching it to the context. Then, we changed a single property and saved the changes again. This time we get a specific concurrency exception of the `DbUpdateConcurrencyException` type. This is how the code looks in VB.NET:

```
Private Sub ConcurrencyExample()  
    Dim person = New Person() With {  
        .BirthDate = New DateTime(1970, 1, 2),  
        .FirstName = "Aaron",  
        .HeightInFeet = 6D,  
        .IsActive = True,  
        .LastName = "Smith"  
    }  
    Dim personId As Integer  
    Using context = New Context()  
        context.People.Add(person)  
        context.SaveChanges()  
        personId = person.PersonId  
    End Using  
    'simulate second user  
    Using context = New Context()  
        context.People.Find(personId).IsActive = False  
        context.SaveChanges()  
    End Using  
    'back to first user  
    Try  
        Using context = New Context()  
            context.Entry(person).State = EntityState.Unchanged  
            person.IsActive = False  
            context.SaveChanges()  
        End Using  
        Console.WriteLine("Concurrency error should occur!")  
    Catch exception As DbUpdateConcurrencyException  
        Console.WriteLine("Expected concurrency error")  
    End Try  
    Console.ReadKey()  
End Sub
```

This exception handling code is something we always need to write when implementing concurrency. We need to show the user a nice descriptive message. At that point, they will need to refresh their data with the current database values and then redo the changes. If we, as developers, want to assist users in this task, we can use Entity Framework's `DbEntityEntry` class to get the current database values.

Self-test questions

Q1. You cannot get data from a view inside Entity Framework, true or false?

Q2. Which database method can be used to query and retrieve data using SQL statements?

1. `ExecuteSqlCommand`
2. `Execute`
3. `SqlQuery`

Q3. You have to write all SQL Server stored procedures by hand if you want to map CRUD operations for an entity type to a set of stored procedures, true or false?

Q4. There is no downside to using the asynchronous API, true or false?

Q5. Which method can be used to asynchronously save changes in `DbContext`?

1. `SaveChanges`
2. `SaveChangesAsync`
3. `SaveChangesAsynchronously`

Q6. What method needs to be called to mark a property as a concurrency property inside the entity type configuration class?

1. `RowVersion()`
2. `HasDatabaseGenerationOption()`
3. `IsRowVersion()`

Q7. What exception type indicates a concurrency error?

1. `DbUpdateConcurrencyException`
2. `ConcurrencyException`
3. `OptimisticConcurrencyException`

Summary

Entity Framework provides a lot of value to the developers, allowing them to use C# or VB.NET code to manipulate database data. However, sometimes we have to drop a level lower, accessing data a bit more directly through views, dynamic SQL statements and/or stored procedures. We can use the `ExecuteSqlCommand` method to execute any arbitrary SQL code, including raw SQL or stored procedure. We can use the `SqlQuery` method to retrieve data from a view, stored procedure, or any other SQL statement, and Entity Framework takes care of materializing the data for us, based on the result type we provide. It is important to follow best practices when providing parameters to those two methods to avoid SQL injection vulnerability.

Entity Framework also supports environments where there are requirements to perform all updates to entities via stored procedures. The framework will even write them for us, and we would only need to write one line of code per entity for this type of support, assuming we are happy with naming conventions and coding standards for such procedures.

Entity Framework now provides support for asynchronous operations, including both query and updates. Developers must take care when using such techniques to avoid potential performance implications. In some technologies, the asynchronous API fits really well, the Web API being a good example.

We must always take care of our data, avoiding data loss at all costs. This is where concurrency handling built into Entity Framework comes in. It allows us to provide users with appropriate feedback, while helping us to avoid silent data loss. We just need to mark a property as concurrency check property, and Entity Framework will throw an exception when two users make changes to the same entity at the same time. We just need to handle this exception and provide users with an application-specific error message.

In the next chapter, we will conclude our discussion of Entity Framework by learning how to update production database structure without data loss using, migrations.

7

Database Migrations and Additional Features

In this chapter, you will learn how to make structural database changes using the Entity Framework migrations API. Previously, we used an initializer to drop and recreate the database to handle such changes. Now, you will learn how to use Entity Framework migrations to achieve the same end result without data loss. We will also discuss the process of integrating Entity Framework with an existing database, instead of allowing the framework to create a database from scratch. We will also take a look at some additional features in Entity Framework that we need to be aware of that are not commonly used on a daily basis.

In this chapter, we will cover how to:

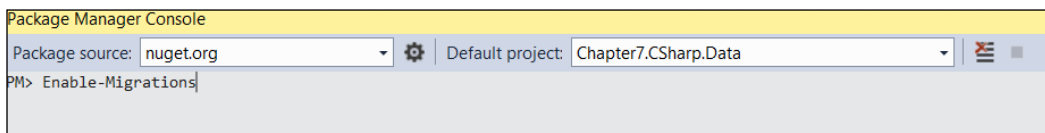
- Enable migrations on a project that uses Entity Framework
- Use automatic migrations
- Create explicit migrations
- Add database artifacts, such as indexes
- Add migrations to an existing database
- Use additional Entity Framework features (not covered in previous chapters)

Enabling and running migrations

Entity Framework is an ORM tool, thus it works with a database. We already saw that we are faced with the challenge of keeping an RDBMS structure and our Entity Framework entities synchronized. Previously, we used an initializer to drop and recreate the database to have the new structure match our context and entities. Obviously, we cannot do this in production. So, we have two choices. We can pick another tool, for example, SSDT for SQL Server, to separately maintain and upgrade database artifacts. The second choice, the one we are going to work on in this chapter, is to use Entity Framework itself to update the database at such times when the structure changes. In order to utilize this technology, we have to enable migrations on our project.

Previously, we used a single project for our application and Entity Framework's entity classes. This is not a common structure for typical non-trivial solutions. It is more likely that we would separate Entity Framework objects into their own project. This project would be of the type `class library`. We will do so in the sample project we are going to work on in this chapter. We can create this additional **Data** project following the same simple steps as we did before. We need to add the Entity Framework NuGet package reference to the new `class library` project, and write entity classes and the `context` class. Then, we can add this project as a reference to our application's main project, the console app, in the downloadable sample.

The next step is to enable migrations for our **Data** project. We will use the NuGet **Package Manager Console** window we referred to in previous chapters. We can pull up this window by navigating to **Tools | NuGet Package Manager | Package Manager Console** from the Visual Studio menu. Once this window is visible, select the **Data** project from the project drop-down menu, then type `Enable-Migrations` in the window, and press the *Enter* key, as shown in the following screenshot:



If we need to get detailed help for the PowerShell commandlet, `Enable-Migrations`, we just type `Get-Help Enable-Migrations`. We will find the parameters' information, which in part enables developers to point migrations to a specific project or connection string. In our case, we did not need to specify any parameters because we added the target connection string to the configuration file inside our **Data** project. After we run this command, we will see an additional folder created in our project called **Migrations**. There will be a class inside that folder that specifies migration configuration, tying it to our context class through the generic type parameter, as shown in the following code:

```
internal sealed class Configuration :
    DbMigrationsConfiguration<Chapter7.CSharp.Data.Context>
{
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }
    protected override void Seed(Chapter7.CSharp.Data.Context
context)
    {
    }
}
```

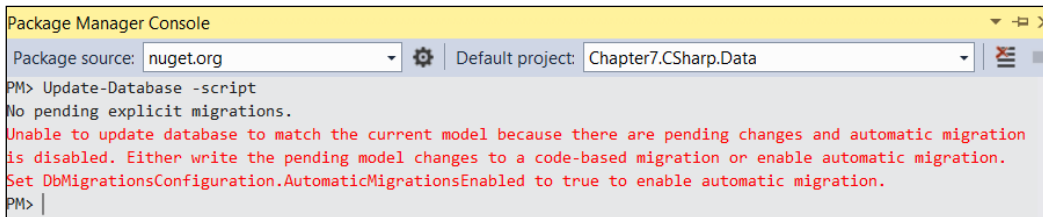
Here is the same class in VB.NET as follows:

```
Friend NotInheritable Class Configuration
    Inherits DbMigrationsConfiguration(Of Context)

    Public Sub New()
        AutomaticMigrationsEnabled = False
    End Sub
    Protected Overrides Sub Seed(context As Context)
    End Sub
End Class
```

This class also has the `Seed` method, which is invoked every time migrations are applied to a database, enabling developers to perform miscellaneous tasks, such as inserting seed data. Since this method can be run many times on a database, we need to ensure that seeded data is not duplicated. Thus, we need to check whether our data already exists in the target database before inserting it.

Now we are ready to proceed with the database creation. If we are working locally, simply creating and/or upgrading the local database, we can continue using the **Package Manager Console** window. This time we can use the `Update-Database` commandlet. Again, we can use the `Get-Help` commandlet to take a look at the parameters we can work with. At this point, we are interested in the `-script` parameter. This parameter is useful, as it will generate a migration SQL script that we can hand to our DBA or run ourselves. When the `Update-Database` commandlet is run, it will compare the structure defined by our entity classes and the `Context` class against the physical database. In our case, we can omit parameters because we copied the connection string into our **Data** project and we only have a single class that inherits from `DbContext` in the project. If we run the command now, we will get the following error, shown in the following screenshot:



This error refers to the setting that allows us to enable the *automatic migration* generation. Let's update our migration's configuration class to enable automatic migrations, just as the error informs us. Then, we can build the solution and rerun the commandlet to create the script. We will see that the SQL script will open in Visual Studio. We can then create the target database by running the script. This functionality is useful when we are working with a DBA who needs to review our upgrade scripts. Since we do not need to do this locally, let's create the local database by running the `Update-Database` commandlet without any parameters. No errors should be shown. If we now open **SQL Server Management Studio (SSMS)**, we will see our new **Chapter 7** database! Congratulations, we just used Entity Framework migrations for the first time!

Automatic migrations are really easy to use. We can just make changes and rerun `Update-Database` to propagate the changes to our SQL Server database. To verify that there is no data loss, let's manually add a row to our `People` table using SSMS, as shown in the following screenshot:

	PersonId	FirstName	LastName
▶	1	John	Doe

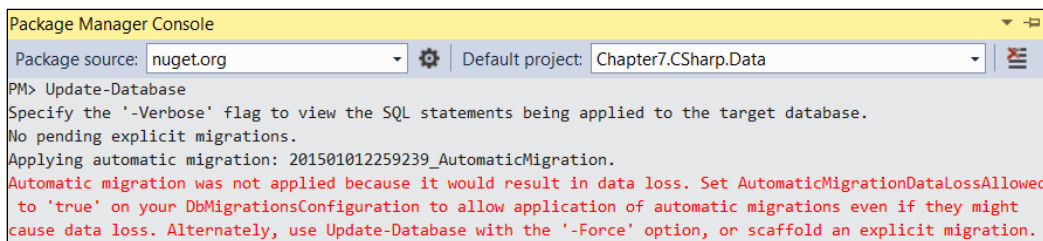
Let's try the following now. We are going to add a new property to the `Person` class, called `Age`, which is a numeric property, and rerun `Update-Database`. The class looks as follows:

```
public class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
}
```

This is how the class looks in VB.NET:

```
Public Class Person
    Property PersonId() As Integer
    Property FirstName() As String
    Property LastName() As String
    Property Age() As Integer
End Class
```

After we update the database, we will see that our data is preserved. Our existing row is still there, and the `Age` column value is 0. If we look at the table structure, we will notice our new `Age` column, which has the database default value of 0. This is what Entity Framework does for us. For non-nullable columns, it attempts to pick a default value, which in fact is the type's default. Let's add a non-nullable string property with a maximum size of 50 characters, called `NickName`. We remember that we need to update the entity configuration class to do so. Say, we accidentally made a mistake, and we want to make the `NickName` column smaller, say 40 characters. Let's make this change and attempt to update the database again. We will see an error as shown in the following screenshot:



We now need to use another parameter for `Update-Database`, called `Force`. It forces change to run, even when it results in potential data loss. We can run `Update-Database -Force` to update our database this time. Alternatively, we can just enable support for data loss as Entity Framework exposes this setting, just like the error text tells us.

As we saw, simple scenarios can be easily accommodated via automatic migrations. This approach falls apart when our migrations get more complicated, as we will see in the following content.

Using the migrations API

Let's add non-nullable date property:

```
public DateTime DateAdded { get; set; }
```

This is how the new property looks in VB.NET:

```
Property DateAdded() As DateTime
```

We want this new column to default to the current date. If we update our database again, we will see that the new column will have a default of 1/1/1900. This is not what we want, and here is when we need to switch to explicit migrations. In general, explicit migrations are more flexible than automatic ones. Although we need to write more code, we have far more control over the flow of migrations, their names, and the rollback process. If we start mixing the two approaches, we may get confused. For example, we would have to search the project to see if a column was added by automatic or manual migration. So, in order to provide consistency and for maintenance purposes, we may want to standardize on explicit migrations, and at that point, we should disable automatic migrations.

In order to get started with this approach, let's drop our sample database in SSMS, disable automatic migrations using the property we saw before on our migration configuration class, and create our first manual migration.

In order to create our initial database migration, we need to use a new commandlet, `Add-Migration`. Here is how our command looks:

```
Add-Migration InitialMigration
```

`InitialMigration` is just a name; one can provide a different name if desired. This action will add a new class to our **Migrations** folder. The physical file will be named something similar to **201501012315236_InitialMigration**. The filename is prefixed with the time when the migration was created, helping us organize migrations within the folder.

The generated class looks as follows:

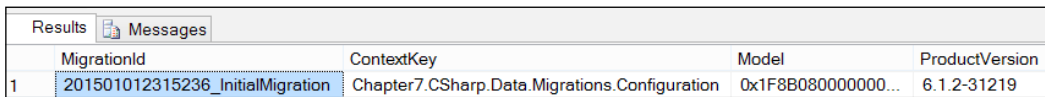
```
public partial class InitialMigration : DbMigration
{
    public override void Up()
    {
        CreateTable(
            "dbo.Companies",
            c => new
            {
                CompanyId = c.Int(nullable: false, identity:
true),
                Name = c.String(),
            })
            .PrimaryKey(t => t.CompanyId);

        CreateTable(
            "dbo.People",
            c => new
            {
                PersonId = c.Int(nullable: false, identity:
true),
                FirstName = c.String(nullable: false,
maxLength: 30),
                LastName = c.String(nullable: false,
maxLength: 30),
                NickName = c.String(nullable: false,
maxLength: 40),
                Age = c.Int(nullable: false),
            })
            .PrimaryKey(t => t.PersonId);
    }

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

Let's take a closer look at the generated code. We are using the base class called `DbMigration` to implement our migration. We override the `Up` and `Down` methods. The `Up` method moves our database structure forward, creating two new tables in our case. The `Down` method helps us revert the changes in case we discover software issues and want to rollback to the previous structure which may be required at a later date. Let's update the database one more time by running `Update-Database`. We will notice that our database was created along with two new tables.

If we take a closer look at the created database, we will see another table, `_MigrationHistory`. Here is what the data looks like in this table, as shown in the next screenshot:



MigrationId	ContextKey	Model	ProductVersion
201501012315236_InitialMigration	Chapter7.CSharp.Data.Migrations.Configuration	0x1F8B08000000...	6.1.2-31219

We see that the migration identifier (**MigrationId**) corresponds to the file name for our initial migration. The **Model** column contains the context hash value, uniquely identifying it for the Entity Framework engine. Finally, the **ContextKey** column contains the class name for our context's configuration class.

Let's go back to the previous example and add the `DateAdded` property back to our class. Then, let's create a new migration for this new property, again using the `Add-Migration` commandlet, as shown in the following code line:

Add-Migration PersonDateAdded

Here is the code that was generated by Entity Framework:

```
public partial class PersonDateAdded : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.People", "DateAdded", c =>
c.DateTime(nullable: false));
    }

    public override void Down()
    {
        DropColumn("dbo.People", "DateAdded");
    }
}
```

We already saw the `AddTable` method. Now, we also see the `AddColumn` method. It takes the table and column names as well as the column type, specified via the corresponding .NET type. We are going to add a custom default value this time. Migrations support hardcoded default values as well as the database engine's default values that are specified as strings. In order to specify the hardcoded default, we can use the `defaultValue` parameter. We will use `defaultValueSql` instead, as shown in the following code snippet:

```
public partial class PersonDateAdded : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.People", "DateAdded",
            c => c.DateTime(nullable: false, defaultValueSql:
"GetDate()"));
    }

    public override void Down()
    {
        DropColumn("dbo.People", "DateAdded");
    }
}
```

We used the SQL server `GetDate` function to populate the newly added column with the current date, as per our business requirements. The `AddColumn` method and column configuration classes support a variety of other parameters besides a default value, using the `ColumnModel` class. Essentially, we can specify many of the same values that we can in our `EntityTypeConfiguration` class. All parameters that can be discovered based on our `EntityTypeConfiguration` class will be scripted for us automatically by Entity Framework migrations. The default value is something we can add manually.

The `DbMigration` base class, which is used to write migrations, supports maintenance of many database artifacts besides columns and tables. We can perform the following:

- Create, drop, and alter stored procedures
- Add and drop foreign keys
- Move artifacts, such as tables and stored procedures, between schemas
- Rename objects, such as tables, procedures, and columns
- Maintain primary key constraints
- Create, rename, and drop indexes

Finally, when we encounter a unique circumstance where none of the stated methods work, we can use either the `Sql` or `SqlFile` method. Just as their names implies, they allow us to execute arbitrary SQL statements as part of any migration. The former method takes a string that represents SQL statement(s). The latter takes a file name, whereas the file itself contains any number of SQL statements.

All migrations, by default, run as part of an overarching transaction, ensuring that either all migration operations succeed, or none. This is certainly true for SQL Server. This may or may not be the case for other RDBMSes. For example, Oracle does not support transactions on structural operations, defined by **Data Definition Language (DDL)**. DDL is simply a term that refers to SQL statements that define data structures. There is also **Data Manipulation Language (DML)**, which refers to CRUD operation statements or other statements that manipulate the data.

We do not have to have pending changes to create a migration. For example, in order to create an index, no pending changes are needed. Alternatively, we can use the API introduced in Entity Framework 9.6.1, which enables us to create indexes via the model builder API. For the purposes of this example, we will use the migration API. We still follow the same steps as before, running the `Add-Migration` commandlet. This will add a migration to our project, but both `Up` and `Down` methods will be empty. Now, we just need to add some custom code to create the desired index, as shown in the following code snippet:

```
public partial class PersonPersonNamesIndex : DbMigration
{
    public override void Up()
    {
        CreateIndex(
            "People",
            new[] { "LastName", "FirstName" },
            name: "IX_PERSON_NAMES");
    }
    public override void Down()
    {
        DropIndex("People", "IX_PERSON_NAMES");
    }
}
```

In this migration, we create a new index with the name of `IX_PERSON_NAMES` on the `People` table that contains two columns: `LastName` and `FirstName`. In the `Down` method, we revert this change, dropping an index. Here is how the code looks in VB.NET:

```
Partial Public Class PersonPersonNamesIndex
    Inherits DbMigration
    Public Overrides Sub Up()
```

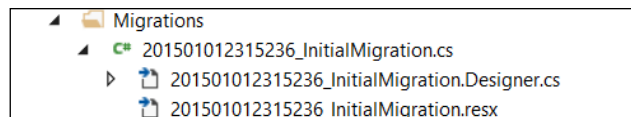


```

        CreateIndex( _
            "People", _
            New String() {"LastName", "FirstName"}, _
            name:="IX_PERSON_NAMES")
    End Sub
    Public Overrides Sub Down()
        DropIndex("People", "IX_PERSON_NAMES")
    End Sub
End Class

```

Until now, we have not seen the `Down` method being used. It turns out that Entity Framework migrations support target migration as part of its API, allowing developers to move the database structure to any version, that is, migration. Migrations are sorted by their time of creation, essentially the file name, coded into the migration designer file. You can see that each migration contains three files by expanding any migration in the **Solution Explorer**, as shown in the following screenshot. The first file is the **actual migration code**. The second file, containing the word **Designer**, specifies the migration identifier and a few other properties. The third file, **resource file**, contains values such as the schema name and migration target hash.



Let's now try to drop the index by specifying the target migration to be the one just before our index-creating migration. Its name is `PersonDateAdded` from the previous example. To do so, we just need to run the `Update-Database` commandlet with the target migration parameter, as shown here:

```
Update-Database -TargetMigration "PersonDateAdded"
```

Entity Framework will immediately inform us which migrations were reverted, in our case, just the index-creating migration. If we now look at the database structure, we will see that the index no longer exists.

Applying migrations

So far, we applied all migrations using Visual Studio. This works really well when developers are working on features inside Visual Studio. However, when it comes to updating, testing, or production environments, this approach does not really work.

In order to update such software installations, we are given more options, which are as follows:

- Generate the changes script
- Use `migrate.exe`
- Use the migrating initializer

Applying migrations via a script

We can easily generate a script by running the `Update-Database` commandlet with the `Script` parameter inside the same **Package Manager Console** window, using the following code line:

```
Update-Database -Script
```

As soon as this commandlet completes, the generated script will be opened. It will contain all the required changes to bring the structure of the target database up to date. We just need to give this script to our DBA, who will maintain our production environment.



We need to specify the correct connection string to the database that matches our target environment, since the migrations API compares the live database with the context from the data folder. We can use either another parameter to `Update-Database` and provide this connection string, or use a proper connection string in the configuration file that is used by our **Data** project.

Applying migrations via `migrate.exe`

`Migrate.exe` is a utility that is shipped with Entity Framework. It will be located in the same NuGet package folder as the Entity Framework DLL itself. We just need to distribute this utility with the `binaries` folder of our application to allow the utility to find all the assemblies it needs to work. This utility takes the same parameters as the `Update-Database` commandlet, for example:

```
migrate.exe Chapter7.VB.data
/connectionString="Data Source=.;Initial Catalog=Chapter7;Integrated
Security=SSPI"
/connectionProviderName="System.Data.SqlClient"
/startupConfigurationFile=Chapter7.VB.exe.config
```

We separated the command line into multiple lines for clarity, putting each argument on its own line for readability. The first argument is the assembly containing our context and migrations. Then, we specify the connection string, provider, and configuration file. We need the configuration file because our context's constructor is set up to take the connection string name from the configuration file.

Applying migrations via an initializer

We already saw how to use an initializer to recreate a database when structural changes are required. Entity Framework comes with an initializer base class that can be used to apply pending migrations. The base class is called `MigrateDatabaseToLatestVersion`. Here is how we define our initializer:

```
public class Initializer :  
    MigrateDatabaseToLatestVersion<Context, Configuration>  
{  
}
```

This is a very simple class; there is no code we need to write for it, unless we want to use the `InitializeDatabase` method, which allows us to run some code when migrations are applied. This method gets an instance of our `DbContext` object, thus we can add more data to the database in this method or perform other functions. Here is how this code looks in VB.NET:

```
Public Class Initializer  
    Inherits MigrateDatabaseToLatestVersion  
    (Of Context, Configuration)  
End Class
```

Alternatively, we can use our migration configuration class, which has the familiar `Seed` method to populate our database with some seeded data.

Now, we just need to plug this new initializer into Entity Framework at the application startup time and call the context to force migrations to be applied, as shown in the following code snippet:

```
Database.SetInitializer(new Initializer());  
using (var context = new Context())  
{  
    context.Database.Initialize(true);  
}
```


We already saw similar code when we worked with other initializers. Here, we also call the `Initialize` method on the database to force schema verification and migrations application on an existing database. If the database does not exist, it will be created. Here is how the code looks in VB.NET:

```
Database.SetInitializer(new Initializer)
Using context = new Context
    context.Database.Initialize(True)
End Using
```

We do not have to call the initialization method at the application start up time; it will automatically run during the first query execution. This code just makes migration application time more predictable.

Adding migrations to an existing database

Sometimes, we have a use case where we want to add Entity Framework migrations to an existing database so that we can move from one way to handle schema changes to the migrations API. Of course, since our database is already in production, we need to let migrations know that we are starting with a known state. This is quite easy to do with another parameter to the `Add-Migration` commandlet: `-IgnoreChanges`. When we issue this command, Entity Framework will create an empty migration. It will assume that your model defined by context and entities are compatible with our database. Once you update the database by running this migration, no schema changes will take place, but a new row will be added to the `_MigrationHistory` table for this initial migration. Once this is accomplished, we can safely switch to the Entity Framework migration API to maintain schema changes from that point on.

 Some database systems do not support underscore as the first character for a table name. Entity Framework allows developers to customize this name.

Another use case we want to address is when we also want to create entities for this existing database, thus adding Entity Framework not only to an existing database, but also to an existing software. This task can be accomplished with **Entity Framework Power Tools**. This Visual Studio extension is available on Visual Studio gallery at <https://visualstudiogallery.msdn.microsoft.com/72a60b14-1581-4b9b-89f2-846072eff19d/>. Once we install this extension, we will see a new option available on the right-click menu, **Reverse Engineer Code First**, at the project level.

All developers need to do is point this to the database they want to support with Entity Framework, and this tool will scaffold entities, context, and configuration classes for all entity classes for all tables in the database. We can also use **Entity Framework Tools**, which can be downloaded from the download center at <http://www.microsoft.com/en-us/download/details.aspx?id=40762>. This set of tools supports Code-First generation from a database as well. In order to use this functionality, we just need to select **ADO.NET Entity Data Model** from the **Add New Item** dialog and then follow the steps shown by the wizard.

Additional Entity Framework features

Let's take a look at a few more features that do not neatly fit into anything we have talked about thus far. They are not frequently used, but it is important that developers know that these features exist.

Custom conventions


Sometimes, we want to make global changes that are applied to many entity types or tables. For example, we want all decimal fields be of a certain size by default, unless we specify otherwise. We may also want to globally set all string properties to be mapped to non-Unicode columns because our application is intended only for English-speaking users. We can accomplish such tasks by using the global configuration API or custom conventions. Inside conventions, we also have access to the public mapping API, which allows us to inspect current mappings between entities to database tables and columns. For example, here is how we can set all string properties to be stored as non-Unicode columns in our database:

```
protected override void OnModelCreating(DbModelBuilder
modelBuilder)
{
    modelBuilder.Properties<string>()
        .Configure(config => config.IsUnicode(false));
}
```

We use our context class, which inherits from `DbContext` to accomplish our goal. Here is how the code looks in VB.NET:

```
Protected Overrides Sub OnModelCreating(ByVal modelBuilder As
DbModelBuilder)
    modelBuilder.Properties(Of String) _
        .Configure(Function(p) p.IsUnicode(False))
End Sub
```

We could also write the same code as a custom convention and add it to the conventions collection inside model builder. In order to do so, we will create a class that inherits from the `Convention` base class, override the constructor, then use the same preceding code, call the `Properties` method of the `Convention` class instead of `modelBuilder`. If you like, feel free to practice by writing such conventions.

 We must always remember to add custom conventions to the conventions collection inside `modelBuilder`.

These types of conventions are referred to as configuration conventions. Entity Framework has the model convention API to create two types of conventions: the store model and conceptual model conventions. The purpose is still the same. These conventions allow us to apply changes globally to many places in our model, instead of entity by entity and property by property. We can also write multiple conventions per .NET type, as Entity Framework allows us to control the order in which conventions are applied.

Geospatial data

Beside scalar types, such as string or decimal, Entity Framework also supports geospatial data via the `DbGeometry` and `DbGeography` types in .NET. These types have built-in support and proper translation to support geospatial queries, such as the distance between two points on a map. These specific query methods are available under the geospatial properties of our entities as part of any query. In other words, we still write .NET code when we work with spatial types.

Dependency injection and logging

Entity Framework now implements the service location pattern, thus enabling dependency injection. Dependency injection is used to support configuration methods. For example, we can create our own dependency resolver that uses our custom approach to create common Entity Framework objects, such as `IDbConnectionFactory`. We can read documentation for Entity Framework to find out what classes or interfaces we can inject into a running application, and force Entity Framework to use them instead of the default implementations. For more information, read the MSDN article, available at <http://msdn.microsoft.com/en-us/data/jj680697>.

We can also inject a custom logger into Entity Framework, so that we can log all actions executed by Entity Framework to our custom logging source. The `Database` object has the `Log` property that developers can set in order to create a custom log. This `Log` property expects a method with one string parameter. Set it to `Console.WriteLine` for your console application as an experiment. If you don't like the format of such logging, a custom formatter can be also created.

Startup performance

Startup time can be relatively long for large databases and contexts at times. **Entity Framework Power Tools** allows us to speed up this process by exposing an ability to us to pregenerate views. We are not talking about database views in this case. Instead, we are referring to statements that entity framework generates to be able to create CRUD operation statements. All we need to do to in order to generate these precompiled views is right-click on a file that contains the class derived from `DbContext` after we install power tools and select the **Generate Views** action under the **Entity Framework** menu. This action will create all the code that needs to be compiled into our assembly.

Multiple contexts per database

We do not always have to put all collections that map to tables inside a single context. There are a few advantages to using multiple `DbContext` classes. This approach will likely reduce the start up time, since this time is generally proportionate to a number of collections inside the context that is being accessed for the first time. It will also reduce the surface of data exposed by each context to developers. Also, it will help developers organize the data into data modules. Of course, if we use migrations, we still need a context that contains every collection or table, as we will use this context for migrations support. This would be the only context we actually need to configure. When we use multiple contexts and save data in a single transaction to multiple contexts, we need to take a few extra steps. Each `SaveChanges` call is transactional on its own, but we need to create a single overarching transaction across all `SaveChanges` calls. We may find it easier to use a single large `DbContext` class with all the collections in it for save operations that involve multiple modules.

Self-test questions

Q1. You have to enable migrations on a project to take advantage of schema updates built into Entity Framework, true or false?

Q2. Automatic migrations work 100 percent of the time; there is no reason to ever create explicit migrations, true or false?

Q3. You do not need access to the target database in order to generate the migrations script to the latest version, true or false?

Q4. In order to add migrations to an existing production database, you need to do which of the following?

1. Just enable automatic migrations
2. Create an initial migration, scripting the entire database
3. Create an initial empty migration

Q5. You cannot use Visual Studio in order to update a local development environment, true or false?

Q6. Entity Framework migrations have no support for stored procedures, so you must use other tools to achieve this task, true or false?

Q7. In order to set a common precision and scale for all decimal fields across all the entity classes and tables, you have to specify this size for every such field for each entity, true or false?

Q8. If you want to log the commands fired against your RDBMS by Entity Framework, you can only use database tools, such as the SQL Server profiler, true or false?

Q9. You have to use stored procedures in order to determine a distance between two geographic points, stored in our database using coordinates specified by SQL Server geography data types, true or false?

Summary

In this chapter, we saw how to use Entity Framework to maintain a database schema. You learned that we can enable migrations on a project by running the `Enable-Migrations` commandlet inside the NuGet package manager console. Once we enabled migrations, which created a configuration class, we could start moving the schema of our database forward. Developers have two options for migrations. They can rely on automatic migrations or create explicit migrations. Automatic migrations have limitations. Some tasks, such as setting a default value, are not possible. In order to ensure migrations consistency, developers may opt to only use explicit migrations. All explicit migrations inherit from the `DbMigration` class, which contains methods to allow developers to update a schema of the target database. This class exposes a method that allows us to create or drop tables, create, drop and alter columns, create and drop indexes, and so on. Finally, when an appropriate method is not found or when we need to simply make data only changes, we can use the `Sql` method to run arbitrary SQL command(s). If we need to enable migrations on an existing database, we simply need to create one empty migration, thus marking our context as up to date with our database. Once this empty initial migration is created, we can start writing migrations as usual. We can update a database we use in our development environment quite easily, using the `Update-Database` commandlet inside Visual Studio. When it comes to updating a production database, this strategy does not work. Thus, we have to use a different approach. We can use an initializer to migrate the database. We can use `migrate.exe` or we can generate a migration script inside Visual Studio. If we use script generation, we must have access to the production database, or at least an empty database with the same schema.

In this book, we did not cover all the details of Entity Framework, as the surface of its API is quite large. We did cover all the features that developers use on a daily basis. There are some other really cool features that we will encounter once in a while. So, we needed to take a quick look at such features. Entity Framework supports geospatial data now. We can use logging capabilities in order to capture the details of the commands that Entity Framework creates to be run against the database. We also can speed up the startup time of Entity Framework by using multiple context classes or pregenerating views.

This concludes our adventure into the exciting world of data access with Entity Framework. You learned how to maintain database structures and manipulate and query data by writing C# or VB.NET code. You have learned a lot of information that makes us better data access developers in the Microsoft world.

Answers to Self-test Questions

Chapter 1: Introducing Entity Framework

Q1. Impedance mismatch between RDBMS and object-orientated programming is the main problem that ORM tools solve. They enable developers to talk to databases in the same way they talk to any other object, using the same programming language, such as C# or VB.NET.

Q2. This statement is false. LINQ can be used to create queries in Entity Framework, thus enabling developers to use C# or VB.NET instead of the SQL language.

Q3. Entity Framework Migrations are used to script and apply structural changes to the database, thus moving it from one version of your software to the next.

Q4. `DbContext` is the abstraction that represents a database you are working with using Entity Framework Code-First. It has collection-based properties that represent tables in the database.

Q5. The answer is false. As Entity Framework uses the provider architecture; it can work with any database that has a provider written for it. At this point, all major database engines are supported, such as MySQL, DB2, and Oracle.

Chapter 2: Your First Entity Framework Application

Q1. `DbSet<T>` is the class you should be using to define a property in your context class that corresponds to a table in your database. The type parameter `T` represents a class that defines that table's structure in terms of .NET.

Q2. As `DbContext` holds an underlying connection to the database, you should utilize the `IDisposable` pattern and call `Dispose` on your context when you are done using it. You can also use the `Using` keyword to achieve the same.

Q3. The `Find` method can be used to locate a row in the database. It takes one or more parameters corresponding to the values of the primary key. If you have a single column that defines the primary key, only one value is needed. Multiple parameter values are reserved for tables with complex multicolumn primary keys.

Q4. You can use the `Remove` method and pass in an instance you would like to be deleted from the database when `SaveChanges` is called on your context.

Q5. You can just find the corresponding object and set its `LastName` property to new values. Then, you can call `SaveChanges` to commit the updated data to the database. You can use the `Find` method or LINQ to locate the matching row in the database. We will see other methods to issue updates in later chapters.

Q6. You will get an exception because no initializer is used. We will see in later chapters how migrations solve this problem.

Chapter 3: Defining the Database Structure

Q1. If you want to make a value optional, you need to use nullable types in Entity Framework. As we need to store an integer value, the correct answer is `Int`.

Q2. The statement is false because the string is a nullable type in .NET. Hence, by default the column will be nullable as well.

Q3. The statement is false. You can remove the conventions from the Entity Framework configuration using the `Remove` method on the `Conventions` collection in the model builder.

Q4. Many-to-Default is not a relationship type. One-to-Many (or One-to-Zero-to-Many), One-to-One (or One-to-Zero-to-One), and Many-to-Many are the correct relationship types.

Q5. The answer is false. This approach will become unwieldy if you have many tables in the database.

Q6. By default, Entity Framework will use Unicode types, such as `nvarchar` for string properties. As there are no constraints on the string property, the correct type will be `nvarchar(max)`.

Q7. Domain is not a relationship type.

Q8. `EntityTypeConfiguration` is the correct "buddy" class to be used to configure persistence for an entity.

Chapter 4: Querying, Inserting, Updating, and Deleting Data

Q1. LINQ supports two types of query syntaxes—method, which looks like any other method calls in your programming language, and Query, which resembles SQL in its appearance.

Q2. The answer is false. If an entity is tracked by the context after retrieval, all changes are tracked individually. Hence, Entity Framework will create an update query that only includes columns/properties touched by the code after the entity in question was retrieved.

Q3. Only the first property in the sort order is specified by the `OrderBy` method; all subsequent ones should be specified by the `ThenBy` method calls.

Q4. In order to specify multiple conditions, you need to use logical operators in a single `Where` method.

Q5. All of the approaches are valid, although you may find that the `AddRange` is a bit more readable.

Q6. The answer is false. `Insert` operations are different from other operations. You can add a root entity to its `DbSet`, and all child entities are assumed to be in new state as well.

Q7. True, since context was not tracking entities prior to the state being set, context has to assume that all properties have been changed.

Q8. The answer is false. If you want to issue a delete query, you need to attach an entity instead of adding it in order to simulate an existing entity in the unchanged state.

Q9. The detached state corresponds to any entity not tracked by the context. Since it is not tracked, `DbContext` will not look at this entity when `SaveChanges` is called. Entities in the unchanged state will also not result in any queries, but they are tracked by the context.

Q10. The local property of `DbSet` will give you access to in-memory data only and will never result in a database query to look for data.

Chapter 5: Advanced Modeling and Querying Techniques

Q1. Since we are not creating a new entity, but a complex type, we need to use `ComplexTypeConfiguration` of the `T` base class to configure it.

Q2. The answer is false. We can use the `ToTable` method in order to configure an entity to be stored in a table with a name that is different from the class name for this entity.

Q3. The answer is false. We can use the `Ignore` method to exclude some properties from the persistence engine.

Q4. The process of selecting a subset of columns from a table, that is, a subset of properties from an entity, is called projection.

Q5. We do not have to declare a type for a result set; we can always use anonymous types.

Q6. We do not have to use joins to get related data in a query, since relationships exist in properties inside entities. Thus, related data is available inside a query by walking through these association properties.

Q7. In order to repeat parent entity data along with child data in the result set, we need to use `SelectMany` method of LINQ.

Q8. The set operator `Distinct` can be used to create a set of unique values from a query.

Q9. We cannot accomplish `LEFT OUTER JOIN` in LINQ with a single method.

Q10. The `Skip` and `Take` methods are used to accomplish paging. The `Skip` method, as the name implies, excludes some number of records from the result set, even though they match the filter. The `Take` method only takes a specified number of rows to include in the result set, even though more rows match the filter.

Q11. We can definitely create grouping queries based on multiple properties. We can typically use the anonymous type to specify which properties the grouped data is based on.

Chapter 6: Working with Views, Stored Procedures, the Asynchronous API, and Concurrency

Q1. Although there is no first class support for views in Entity Framework, we can always retrieve data from a view using the `SqlQuery` method.

Q2. The `SqlQuery` method can be used to call an arbitrary SQL statement, including calling stored procedures or functions. Entity Framework will materialize the results based on the generic type provided to this method.

Q3. This is not correct. Insert, update, and delete operations can be automatically generated by Entity Framework. All we need to do is map an entity to stored procedures inside an entity type configuration class.

Q4. This is not correct. Arbitrary use of the asynchronous API can result in performance overhead.

Q5. `SaveChangesAsync` is the method on `DbContext` that can be called to flush changes to the database asynchronously.

Q6. `IsRowVersion` is the only method called that needs to be made on the property configuration class to mark a property as concurrency check property.

Q7. `DbUpdateConcurrencyException` is the correct type to catch from Entity Framework Code-First to handle concurrency errors.

Chapter 7: Database Migrations and Additional Features

Q1. This is correct. You have to run the `Enable-Migrations` commandlet to easily create all the necessary artifacts to support migrations.

Q2. The answer is false. Some operations, such as setting custom default values, cannot be done with automatic migrations. Neither can we create non-Entity Framework objects, such as stored procedures.

Q3. The answer is false. Entity Framework needs to compare our model, defined by entity classes and context, with the target database to know what structures have changed.

Q4. The answer is C – we need to create an empty migration which signals to Entity Framework that the target structure matches the model.

Q5. This is not correct. We can use the NuGet Package Manager Console window to run commandlets to maintain a local database.

Q6. The answer is false. The `DbMigration` class exposes many methods, including those that create stored procedures.

Q7. The answer is false. We can use conventions or global configuration methods of `DbModelBuilder` to achieve this task.

Q8. The answer is false. We can use the `Log` property of the `Database` object to log commands run by Entity Framework against the database.

Q9. The answer is false. We can create LINQ queries and use methods of the `DbGeometry` and `DbGeography` classes to execute native geospatial queries against the database.

Index

Symbol

.NET types
mapping, to SQL types 26, 27

A

actual migration code file 135
ADO.NET 2
advanced modeling techniques
about 74
column mappings 77
complex types 74-76
enumerations 79, 80
explicit table, using 77
multiple tables, using for
single entity 80-83
supporting columns, adding 78, 79
advanced querying techniques
about 83
aggregations 88, 89
data, paging with windowing
functions 92, 93
example 89-92
grouping 88, 89, 95-100
joins, using 93-95
left outer joins 95-100
projections 83-88
set operators 101, 102
aggregations
about 88, 89
Average method 88
Count method 88
Max method 88
Min method 88
Sum method 88

**American National Standards
Institute (ANSI)** 2
anonymous types 83
association properties 84
asynchronous API 115-118
Average method 88

B

Binary Large Objects (BLOBs) 80

C

CodePlex
URL 8
column mappings 77
complex types 74-76
composite primary keys 60
concurrency
about 119
handling 119-123
optimistic concurrency 119
pessimistic concurrency 119
configuration conventions 140
Count method 88
CRUD 2
custom conventions 139, 140

D

data
filtering, in queries 49, 50
paging, with windowing functions 92, 93
sorting, in queries 51
database
creating, based on .NET classes 9-12
data, deleting 65-67

- data, inserting 57-59
- data, querying 15, 16
- data, updating 60-65
- in-memory data, working with 67, 68
- migrations, adding 138, 139
- record, saving 12-14
- record, updating 16, 17
- row, deleting 17, 18
- schema changes 18-22

Data Definition Language (DDL) 134
Data Manipulation Language (DML) 134
dependency injection
 about 140, 141
 reference link 140
descending keyword 51
Distinct operator 101

E

eager loading 36, 55-57
EDMX file 4
element operations 52, 53
Entity Framework
 architecture 5
 capabilities 4, 5
 Code-First approach 4
 features 139
 history 3, 4
 Model-First approach 4
 new project, creating 8, 9
 URL, for tools 139
Entity Framework Power Tools
 about 141
 URL 138
entity splitting
 about 80
 versus table splitting 83
EntityState enumeration, state
 Added 59
 Deleted 59
 Detached 59
 Modified 60
 Unchanged 60
enumerations 79, 80
Except operator 102

explicit table
 using 77

F

features, Entity Framework
 custom conventions 139, 140
 dependency injection 140, 141
 geospatial data 140
 logging 140, 141
 multiple contexts per database 141
 startup performance 141

G

geospatial data 140
grouping 88, 89, 95-100

I

immediate execution 57
impedance mismatch 3
in-memory data
 working with 67, 68
Intersect operator 102

J

joins
 using 93-95
junction table 41

L

Language INtegrated Query. See LINQ
lazy loading 36, 55-57
left outer joins 95-100
LINQ
 about 3, 47
 method syntax 48
 query syntax 48
LINQ functions
 about 52
 element operations 52, 53
 quantifiers 53
logging 140, 141

M

- Many-to-Many relationship** 41
- mappings, .NET types**
 - reference link 27
- Max method** 88
- method syntax** 48
- migrate.exe** 136
- migrations**
 - about 5
 - adding, to existing database 138, 139
 - applying 135
 - applying, via initializer 137, 138
 - applying, via migrate.exe 136, 137
 - applying, via script 136
 - enabling 126-129
 - running 126-129
- migrations API**
 - using 130-135
- Min method** 88
- multiple contexts per database** 141
- multiple tables**
 - using, for single entity 80-83

N

- nullable properties**
 - handling 33, 34

O

- object graph** 57
- Object-Relational Mapping (ORM)** 1, 2
- One-to-Many relationship** 35-40
- One-to-One relationship** 42-44
- optimistic concurrency** 119
- Oracle mappings**
 - reference link 27
- Overridable** 36

P

- paging functions.** *See* **windowing functions**
- pessimistic concurrency** 119
- primitive properties**
 - configuring 27-32
- projections** 83-88

Q

- quantifiers**
 - about 53
 - All operation 53
 - Any operation 53
- query syntax** 48

R

- related entities**
 - about 54
 - eager loading 55-57
 - filtering 54
 - lazy loading 55-57
- Relational Database Management System (RDBMS)** 1
- relationships**
 - defining 35
 - Many-to-Many 35, 41
 - One-to-Many 35-40
 - One-to-One 35, 42-44
- resource file** 135

S

- schema changes** 18-22
- set operators**
 - about 101
 - Distinct 101
 - Except 102
 - Intersect 102
 - Union 101
- SQL injection** 109
- SQL Server Data Tools (SSDT)**
 - URL 14
- SQL Server Management Studio (SSMS)** 12, 128
- SQL types**
 - .NET types, mapping 26, 27
- startup performance** 141
- stored procedures**
 - about 110-112
 - entities, creating 112-114
 - entities, deleting 112-114
 - entities, updating 112-114
- Structured Query Language (SQL)** 2

Sum method 88

supporting columns

adding 78, 79

T

table splitting

versus entity splitting 83

table structures

.NET types, mapping to SQL types 26, 27

creating 26

nullable properties, handling 33, 34

primitive properties,

configuring 27-32

U

Unicode data 27

Union operator 101

V

views

about 106

working with 106-110

virtual keyword 36

Visual Studio

URL 138

W

windowing functions

used, for paging data 92, 93



Thank you for buying **Code-First Development with 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.

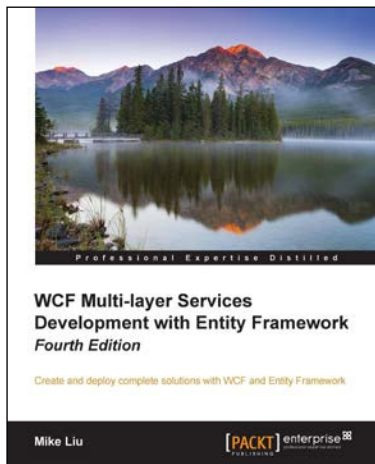
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books 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.

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.



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.



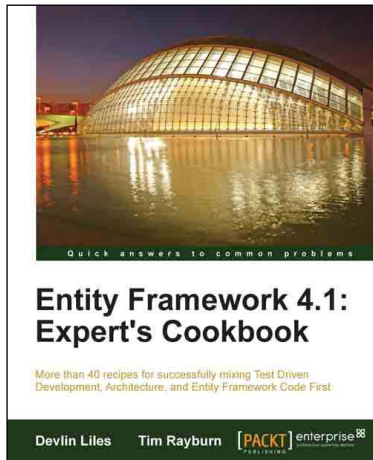
WCF 4.5 Multi-Layer Services Development with Entity Framework *Third Edition*

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.

Please check www.PacktPub.com for information on our titles

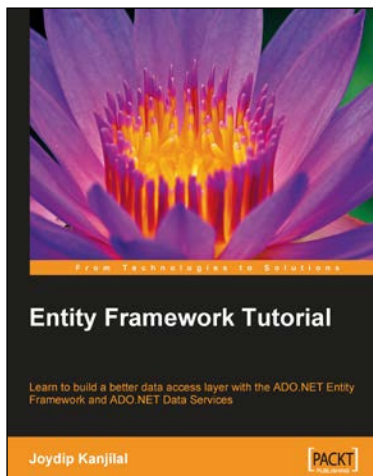


Entity Framework 4.1: Expert's Cookbook

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