Modern Data Access with Entity Framework Core

Database Programming Techniques for .NET, .NET Core, UWP, and Xamarin with C#

Holger Schwichtenberg

Apress®

Modern Data Access with Entity Framework Core

Database Programming Techniques for .NET, .NET Core, UWP, and Xamarin with C#

Holger Schwichtenberg

Apress®

Modern Data Access with Entity Framework Core

Holger Schwichtenberg
Essen, Germany

ISBN-13 (pbk): 978-1-4842-3551-5 ISBN-13 (electronic): 978-1-4842-3552-2
https://doi.org/10.1007/978-1-4842-3552-2

Library of Congress Control Number: 2018947392

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Printed on acid-free paper

For Heidi, Felix, and Maja

Table of Contents

About the Author

Holger Schwichtenberg is a .NET MVP with more than 20 years of experience as a developer and trainer. He is currently a technical lead with the German company IT-Visions, where he consults and trains at companies throughout Europe. He also serves as a software architect for 5Minds IT-Solutions. Holger is a huge fan of Entity Framework (EF) and Entity Framework Core and regularly speaks about both. He has used EF in projects of all sizes, most recently on a big data project containing billions of records. He is a prolific writer, having published more than 65 books and 1,000 technical articles in well-known IT professional and developer journals, including MSDN. He has presented at events such as TechEd Europe, Microsoft IT Forum, Advanced Developer Conference, Microsoft Launch, MSDN Technical Summit, and others. Holger has a PhD in business informatics.

His company web sites are `www.IT-Visions.de` and `www.5minds.de`, and he regularly blogs at `dotnet-doktor.de`. His office can be reached at `office@IT-Visions.de`.

Introduction

I have always been a big fan of object-relational mapping (ORM); in fact, I developed my own OR mapper for my software development projects in the early days of .NET. I switched to the ADO.NET Entity Framework when Microsoft introduced it in .NET 3.5 Service Pack 1. Nowadays, I prefer its successor, Entity Framework Core. Interestingly, some of my projects are still running the classic Entity Framework. As Microsoft continues to do incremental releases of Entity Framework Core, many of the challenges and gripes developers had with earlier versions have gone away, so my plan is to switch the management of all my projects to Entity Framework Core.

The book you hold in your hands came from an idea I had to cover all the important database access scenarios. I hadn't found much collective information in one place and felt that a compendium could be of great value to others. In this book, you will be introduced to database access concepts, get hands-on experience installing Entity Framework Core, and learn about reverse engineering and forward engineering for existing or legacy databases. I'll delve into topics such as schema migrations, data reading, and data modification with LINQ, Dynamic LINQ, APIs, SQL, stored procedures and table-valued functions, object relationships, and asynchronous programming. I'll also talk about third-party products such as LINQPad, Entity Developer, Entity Framework Profiler, Entity Framework Plus, and AutoMapper.

I'll discuss how to apply Entity Framework Core through case studies using Universal Windows Platform (UWP) apps, Xamarin, and ASP.NET Core. Of course, no book would be complete without sharing a healthy dose of hard-earned tips and tricks from my experience with Entity Framework and Entity Framework Core over the years.

Expectations of the Reader

This book is intended for software developers who have experience with .NET and C# as well as some relational database experience and who now want to use Entity Framework Core to create data access code in .NET, .NET Core, UWP apps, and Xamarin. Previous knowledge in predecessor technologies such as ADO.NET and the classic ADO.NET Entity Framework is useful but not necessary to understand this book.

Programming Language Used in This Book

I chose to use C# in this book because it is by far the most commonly used programming language in .NET. While I still occasionally develop .NET applications in Visual Basic .NET, it doesn't make sense to print all the listings in both languages.

If you are interested, a language converter between C# and Visual Basic .NET is freely available on several web sites, including `http://converter.telerik.com` and `https://www.mindfusion.eu/convert-cs-vb.html`.

The Use of Case Studies and Fictitious Enterprises

Most of the sample code in this book revolves around the fictitious airline World Wide Wings, abbreviated as WWWings or just WWW (see Figure 1).

Figure 1. *Logo of the fictional airline World Wide Wings*

Note You'll see other case studies used in some chapters, such as the task management app MiracleList.

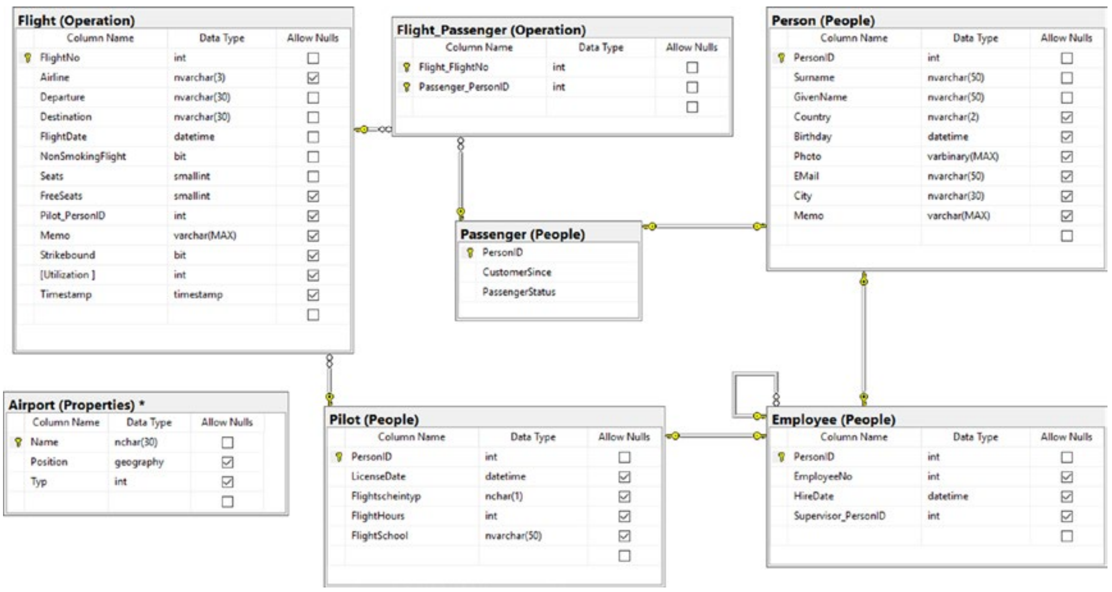
The World Wide Wings use case deals with the following entities:

- *Flights* between two places where the places were deliberately not modeled as separate entities but as strings (this simplifies the understanding of many examples).
- *Passengers* flying on a flight.
- *Employees* of the airline, who have supervisors who are also employees.
- *Pilots* as a specialization of employees. A flight has only one pilot. There are no copilots at World Wide Wings.

- *Persons* as a collection of common characteristics for all people in this example. A person is not available on their own, but only in one of three specializations: passenger, employee, and pilot. In the object-oriented sense, therefore, `Person` is an abstract base class that cannot own instances but is used only for inheritance.

The World Wide Wings use case has two data models, explained here:

- The slightly simpler model version 1 (see Figures 2 and 3) is the result of classic relational database design with normalization. The object model is created by reverse engineering.



Flight (Operation)

Column Name	Data Type	Allow Nulls
FlightNo	int	☐
Airline	nvarchar(3)	☑
Departure	nvarchar(30)	☐
Destination	nvarchar(30)	☐
FlightDate	datetime	☐
NonSmokingFlight	bit	☐
Seats	smallint	☐
FreeSeats	smallint	☑
Pilot_PersonID	int	☑
Memo	varchar(MAX)	☑
Strikebound	bit	☑
[Utilization]	int	☑
Timestamp	timestamp	☑
		☐

Flight_Passenger (Operation)

Column Name	Data Type	Allow Nulls
Flight_FlightNo	int	☐
Passenger_PersonID	int	☐
		☐

Person (People)

Column Name	Data Type	Allow Nulls
PersonID	int	☐
Surname	nvarchar(50)	☐
GivenName	nvarchar(50)	☐
Country	nvarchar(2)	☑
Birthday	datetime	☑
Photo	varbinary(MAX)	☑
EMail	nvarchar(50)	☑
City	nvarchar(30)	☑
Memo	varchar(MAX)	☑
		☐

Passenger (People)

PersonID
CustomerSince
PassengerStatus

Airport (Properties) *

Column Name	Data Type	Allow Nulls
Name	nchar(30)	☐
Position	geography	☑
Typ	int	☑
		☐

Pilot (People)

Column Name	Data Type	Allow Nulls
PersonID	int	☐
LicenseDate	datetime	☑
Flightscheintyp	nchar(1)	☑
FlightHours	int	☑
FlightSchool	nvarchar(50)	☑
		☐

Employee (People)

Column Name	Data Type	Allow Nulls
PersonID	int	☐
EmployeeNo	int	☑
HireDate	datetime	☑
Supervisor_PersonID	int	☑
		☐

***Figure 2.** World Wide Wings data model in the simpler version 1*

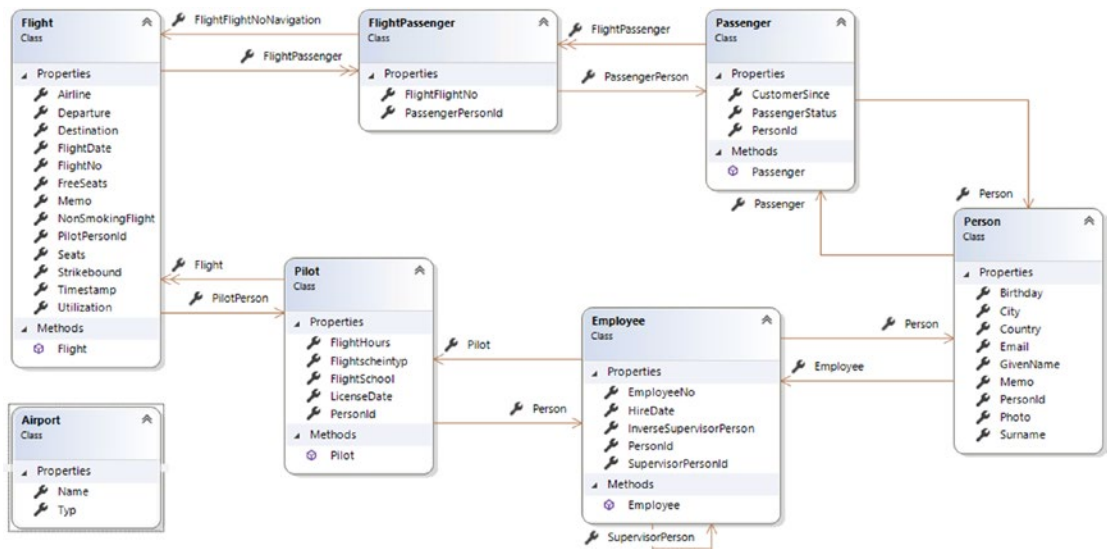


Figure 3. *Object model of the World Wide Wings data model in the simpler version 1*

- Model version 2 (see Figures 4 and 5) is the result of forward engineering with Entity Framework Core from an object model. In addition, there are other entities (`Airline`, `Persondetail`, `AircraftType`, and `AircraftTypeDetail`) in this model to show further modeling aspects. In this case, there is an optional copilot for each flight.

In model version 1 there is a separate table for people (called `Person`), staff, pilots, employee, and passengers. This separation corresponds to the classes in the object model.

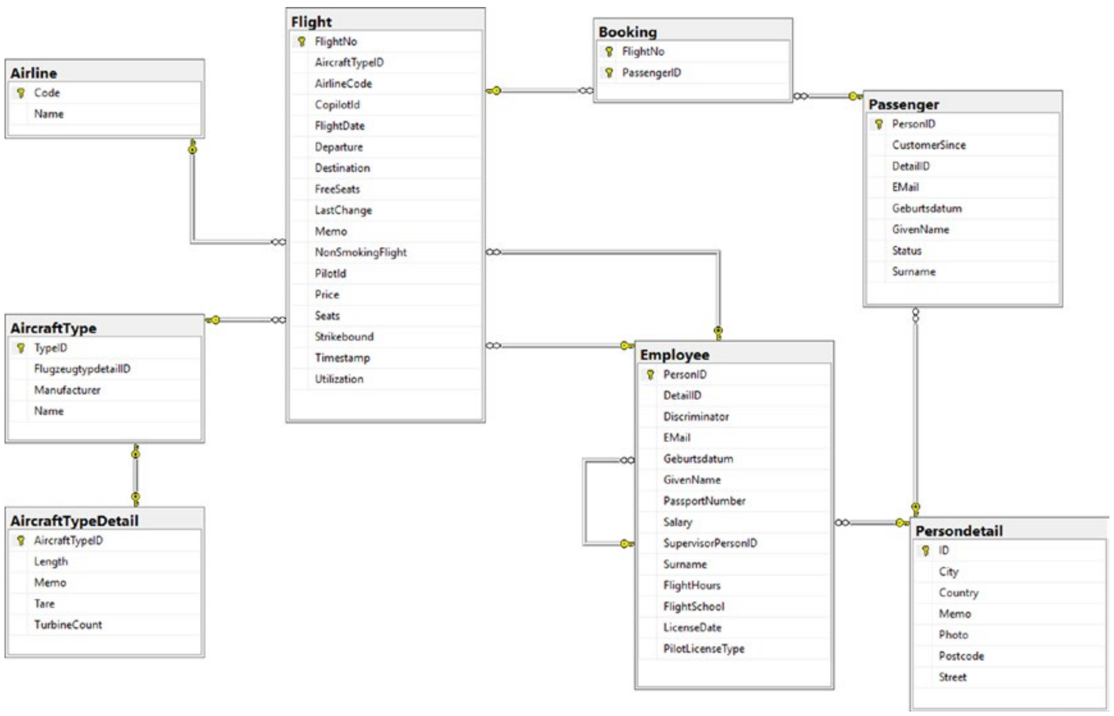


Figure 4. *World Wide Wings data model in the more complex version 2*

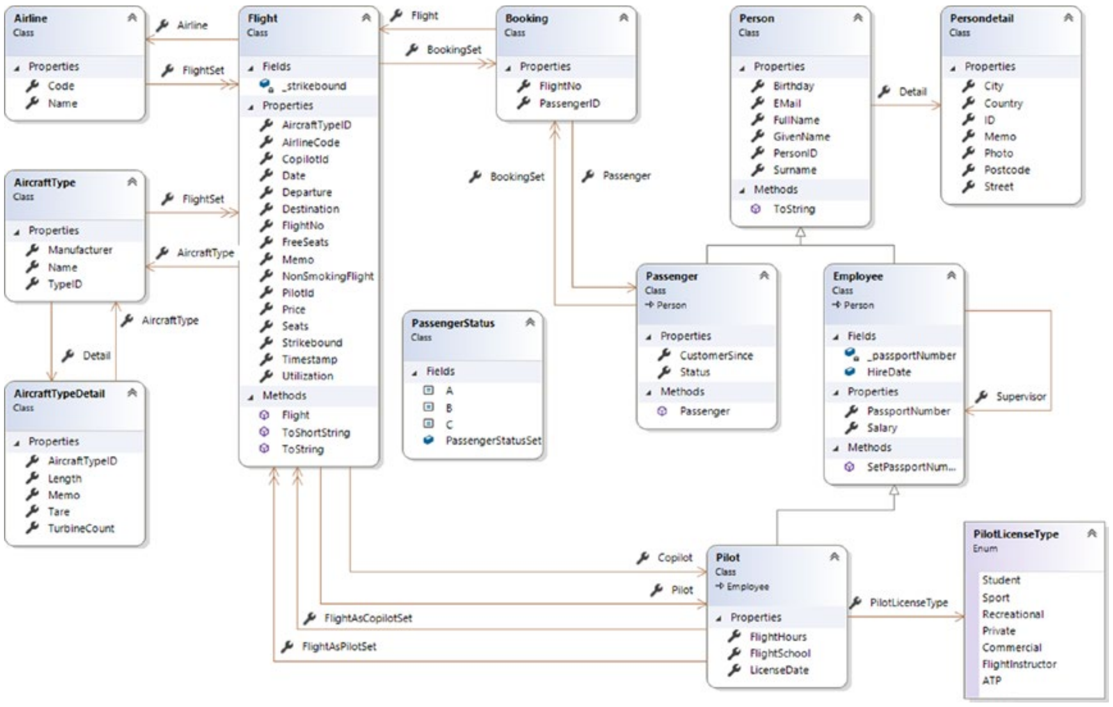


Figure 5. *Object model for the World Wide Wings data model in the more complex version 2*

Note The object models that were created in this book for the data models do not represent an ideal object model because Entity Framework Core does not support some mapping capabilities, such as N:M mapping, yet.

The object model for the data schema of World Wide Wings version 6.1 (Figure 3) was automatically generated by the Entity Framework Core from the database (through reverse engineering); I deliberately did not change it, even if some of the generated names are ugly.

In model version 2, there are only the `Passenger` and `Employee` tables for these four entities. Entity Framework Core is currently somewhat limited and does not support table per type mapping (a separate table for each class). Therefore, the table `Passenger` also includes all the characteristics of `Person`. In addition to the `Person` properties, the

Employee table includes the properties of the Employee and Pilot entities. In the table, a Discriminator column distinguishes between records that are an employee and those that are a pilot. Entity Framework Core mixes the concepts of table by concrete type (TPC) and table by hierarchy (TPH). The developer has no definite influence on the inheritance mapping in Entity Framework Core 1.*x*/2.0. The classic Entity Framework offers more options here.

The following are the extra dependencies in model version 2:

- A Flight belongs to Airline (there will be only World Wide Wings and its subsidiary Never Come Back Airline in this book).
- There a Copilot entity here, but it is optional.
- A Flight can optionally have an AircraftType object assigned. AircraftType must have an AircraftTypeDetail object.
- Each Person and therefore each Pilot and Passenger must own a Persondetail object.

In this book, both data models are used, partly in modified form, to show certain scenarios (for example, database schema migrations).

Application Types in This Book

In this book, the examples are for the most time shown via a text-based console in console applications because this allows me to focus on database access. When using graphical user interfaces such as WPF, Windows Forms, ASP.NET Web Forms, or ASP.NET MVC, the representation is decoupled by data binding, which means I would always need to show a second listing so you could understand that the data access was actually delivered. I simulate user inputs in the console examples by writing variables at the beginning of the program code.

I have provided training and consultancy on data access for many years and have learned that console editions are didactically the best tool for teaching because otherwise the listings are large and thus inferior.

Of course, console output is not common practice in 99 percent of software development, but graphical user interfaces are covered in other books, and data binding typically has no impact on the form of data access. Where data access is relevant, this book will also show data binding examples.

Helper Functions for Console Output

I will show the screen output on the console using the standard method Console.WriteLine() in several places; in addition, I use auxiliary routines that generate colored screen output. Listing 1 shows these auxiliary routines in the class CUI from ITV_DemoUtil.dll for a better understanding.

Listing 1. Class CUI with Subroutines for Screen Output to the Console

```
using System;
using System.Runtime.InteropServices;
using System.Web;
using ITVisions.UI;
using System.Diagnostics;

namespace ITVisions
{
 /// <summary>
 /// Helper utilities for Console UIs
 /// (C) Dr. Holger Schwichtenberg 2002-2018
 /// </summary>
 public static class CUI
 {
  public static bool IsDebug = false;
  public static bool IsVerbose = false;

  #region Print only under certain conditions
 public static void PrintDebug(object s)
  {
   PrintDebug(s, System.Console.ForegroundColor);
  }

  public static void PrintVerbose(object s)
  {
   PrintVerbose(s, System.Console.ForegroundColor);
  }
  #endregion
```

```
#region Print with predefined colors
public static void MainHeadline(string s)
{
 Print(s, ConsoleColor.Black, ConsoleColor.Yellow);

}
public static void Headline(string s)
{
 Print(s, ConsoleColor.Yellow);
}
public static void HeaderFooter(string s)
{
 Console.ForegroundColor = ConsoleColor.Green;
 Console.WriteLine(s);
 Console.ForegroundColor = ConsoleColor.Gray;
}

public static void PrintSuccess(object s)
{
 Print(s, ConsoleColor.Green);
}

public static void PrintStep(object s)
{
 Print(s, ConsoleColor.Cyan);
}

public static void PrintDebugSuccess(object s)
{
 PrintDebug(s, ConsoleColor.Green);
}

public static void PrintVerboseSuccess(object s)
{
 PrintVerbose(s, ConsoleColor.Green);
}
```

```
public static void PrintWarning(object s)
{
 Print(s, ConsoleColor.Cyan);
}

public static void PrintDebugWarning(object s)
{
 PrintDebug(s, ConsoleColor.Cyan);
}

public static void PrintVerboseWarning(object s)
{
 PrintVerbose(s, ConsoleColor.Cyan);
}

public static void PrintError(object s)
{
 Print(s, ConsoleColor.White, ConsoleColor.Red);
}

public static void PrintDebugError(object s)
{
 PrintDebug(s, ConsoleColor.White, ConsoleColor.Red);
}

public static void PrintVerboseError(object s)
{
 Print(s, ConsoleColor.White, ConsoleColor.Red);
}

public static void Print(object s)
{
 PrintInternal(s, null);
}
#endregion

#region Print with selectable color
```

```
public static void Print(object s, ConsoleColor frontcolor, ConsoleColor?
backcolor = null)
{
 PrintInternal(s, frontcolor, backcolor);
}

public static void PrintDebug(object s, ConsoleColor frontcolor,
ConsoleColor? backcolor = null)
{
 if (IsDebug || IsVerbose) PrintDebugOrVerbose(s, frontcolor, backcolor);
}

public static void PrintVerbose(object s, ConsoleColor frontcolor)
{
 if (!IsVerbose) return;
 PrintDebugOrVerbose(s, frontcolor);
}
#endregion

#region Print with additional data


/// <summary>
/// Print with Thread-ID
/// </summary>
public static void PrintWithThreadID(string s, ConsoleColor c =
ConsoleColor.White)
{
 var ausgabe = String.Format("Thread #{0:00} {1:}: {2}", System.Threading.
 Thread.CurrentThread.ManagedThreadId, DateTime.Now.ToLongTimeString(), s);
 CUI.Print(ausgabe, c);
}

/// <summary>
///  Print with time
/// </summary>
public static void PrintWithTime(object s, ConsoleColor c = ConsoleColor.
White)
```

```
{
 CUI.Print(DateTime.Now.Second + "." + DateTime.Now.Millisecond + ":" + s);
}

private static long count;
/// <summary>
/// Print with counter
/// </summary>
private static void PrintWithCounter(object s, ConsoleColor frontcolor,
ConsoleColor? backcolor = null)
{
 count += 1;
 s = $"{count:0000}: {s}";
 CUI.Print(s, frontcolor, backcolor);
}

#endregion

#region internal helper routines
private static void PrintDebugOrVerbose(object s, ConsoleColor
frontcolor, ConsoleColor? backcolor = null)
{
 count += 1;
 s = $"{count:0000}: {s}";
 Print(s, frontcolor, backcolor);
 Debug.WriteLine(s);
 Trace.WriteLine(s);
 Trace.Flush();
}

/// <summary>
/// Output to console, trace and file
/// </summary>
/// <param name="s"></param>
[DebuggerStepThrough()]
```

```
 private static void PrintInternal(object s, ConsoleColor? frontcolor =
 null, ConsoleColor? backcolor = null)
 {
  if (s == null) return;

  if (HttpContext.Current != null)
  {
   try
   {
if (frontcolor != null)
    {
     HttpContext.Current.Response.Write("<span style='color:" +
     frontcolor.Value.DrawingColor().Name + "'>");
    }
    if (!HttpContext.Current.Request.Url.ToString().ToLower().Contains(".
   asmx") && !HttpContext.Current.Request.Url.ToString().ToLower().
   Contains(".svc") && !HttpContext.Current.Request.Url.ToString().
   ToLower().Contains("/api/")) HttpContext.Current.Response.Write(s.
   ToString() + "<br>");

    if (frontcolor != null)
    {
     HttpContext.Current.Response.Write("</span>");
    }
   }
   catch (Exception)
   {
   }
  }
  else
  {
   object x = 1;
   lock (x)
   {
```