



SQL Server for C# Developers

Succinctly[®]

by Sander Rossel

SQL Server for C# Developers Succinctly

By
Sander Rossel

Foreword by Daniel Jebaraj

Copyright © 2016 by Syncfusion, Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

I mportant licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Morgan Weston, social media marketing manager, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	6
About the Author	8
Introduction.....	9
Tools	9
Chapter 1 SQL Server Management Studio.....	10
Chapter 2 ADO.NET	18
Establishing a connection	18
The connection string.....	18
Creating a connection	19
Configuring your connection string.....	20
Building a connection string	21
Querying your database	21
Reading data	22
Parameterization.....	26
Create, update, delete	29
Stored procedures	32
Data sets	33
ADO.NET abstractions	36
Chapter 3 Entity Framework Database First	38
Database First.....	38
IQueryable.....	41
Expression trees.....	45
CRUD.....	47
Chapter 4 Entity Framework Code First	50

Migrations.....	53
Automatic migrations	54
Code First migrations.....	56
Seeds.....	58
Chapter 5 SQL Server Data Tools	61
Database Project.....	61
Publishing	63
Updating	63
Comparing	64
Chapter 6 Troubleshooting	67
Profiling	68
Query plans.....	70
Indexing	73
Parameter sniffing	75
Chapter 7 Interception, Locking, Dynamic Management Views	82
Interception	82
Locking and deadlocks	83
Dynamic management views	86
Chapter 8 Continuous Integration	91
Source control	91
Testing	92
Deployment	94
Conclusion	96

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge
As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Sander Rossel is a professional developer with working experience in .NET (VB and C#, WinForms, MVC, WebAPI, Entity Framework), JavaScript, Git, Jenkins, Oracle, and SQL Server.

He has an interest in various technologies including, but not limited to, functional programming, NoSQL, Continuous Integration (and more generally, Software Quality), and software design.

He's written another e-book in the *Succinctly* series that you can download for free: [Object-Oriented Programming in C# Succinctly](#).

He seeks to educate others on his blog, [Sander's bits - Writing the code you need](#), on his [CodeProject profile](#), and through his continuing book-writing.

Introduction

When writing many applications, you will almost certainly need some kind of database. And when you use a database, you're going to have to work with it. Whether you have database administrators (DBAs) or develop both the database and the application, a little database knowledge always comes in handy.

This book is for everyone who develops C# applications with a SQL Server database. In this book, we're going to connect to a database using classic ADO.NET, and using the Entity Framework, we're going to look at different methods to develop your database. We'll see common pitfalls, why your database is running slow, how we can troubleshoot performance issues, and how we can test and deploy our SQL Server database. The first half of the book has a focus on C# and development. The second half of the book has a focus on troubleshooting using the SQL Server tools.

C# and SQL (the query language) knowledge is assumed.

Tools

In this book we'll make use of [Visual Studio 2015](#) Community Edition and [SQL Server](#) 2014 Express on a Windows machine (I'm using Windows 8.1 Pro and Windows 10). You will need a Microsoft account to download these files. When prompted for a specific SQL Server Express version, pick **SQL Server Express with Tools** (or SQL Server Express with Advanced Services, if you want to play around with Reporting Services and Full Text Search, which aren't a part of this book).

The provided links take you to the product website and will probably let you download the newest version of the respective tool. You may download the newest version at your own risk (the examples in this book probably still apply), or you may search for the version I have used throughout this book. Most of the examples also apply to older versions of the tools, so if you already have something installed, just continue and see if it works.

Installation is pretty straightforward. When installing Visual Studio, make sure you select the **SQL Server Data Tools** check box, and when installing SQL Server, make sure you select **Management Tools - Complete**. When prompted for an instance name in the SQL Server installer, pick any name you like, or choose **Default instance** (my instance is called SQLEXPRESS). Other than that, you can leave on all the defaults, accept the license terms, and just click **Next, Next, Next**.

Chapter 1 SQL Server Management Studio

If you followed the steps in the introduction, you now have a clean installation of both Visual Studio and SQL Server. So let's open up SQL Server Management Studio (SSMS for short). Log in to your default instance, which is localhostname_of_your_instance (default name is MSSQLEXPRESS) with Windows Authentication if you followed the defaults. You should now see the following window:

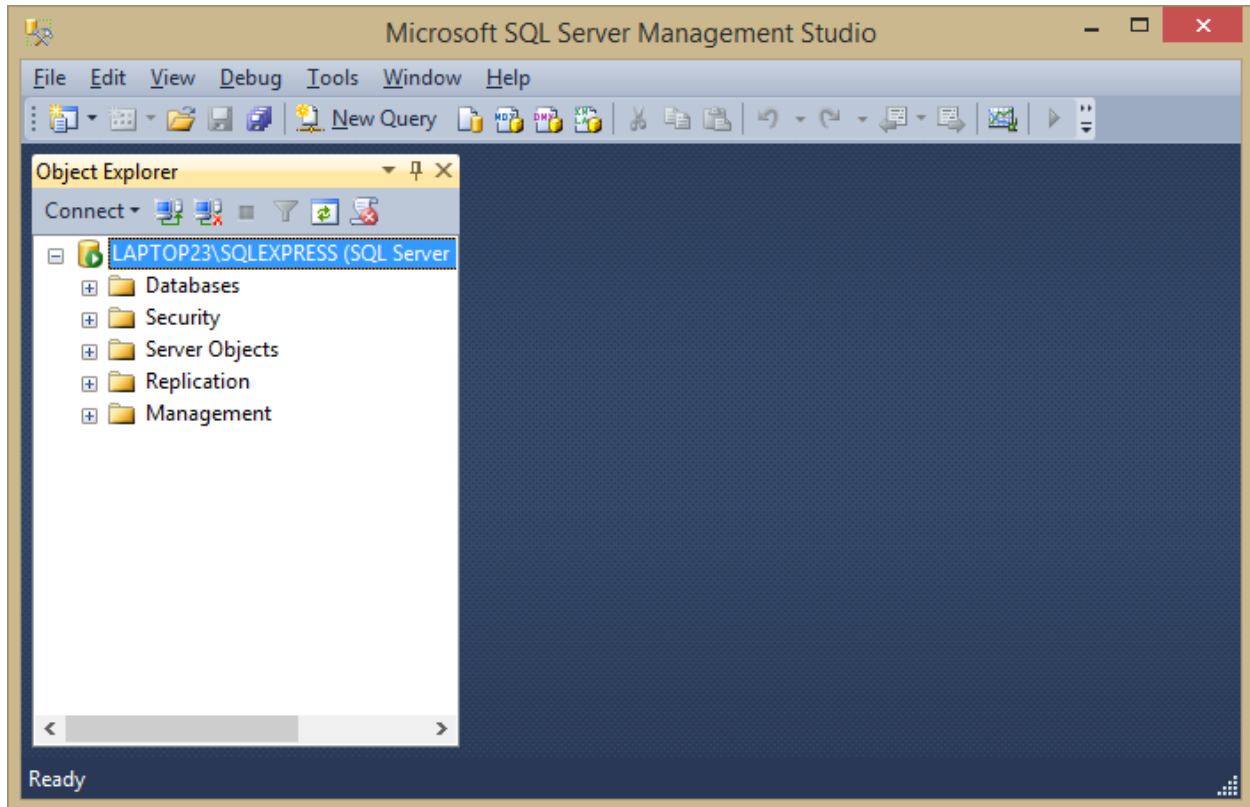


Figure 1: Microsoft SQL Server Management Studio

I've always found SSMS very intuitive to use. To execute any script, simply click **New Query** in the toolbar. Make sure you select the correct database (if applicable) in the upper-left corner (or put **USE [database_name] GO** in your script). Now let's create a database to use in future examples. In the Object Explorer, open up the **Databases** node. It's probably empty, so right-click on the **Databases** node and click **New database**. Pick a name (for this book we'll use **SuccinctlyExamples**), leave all the defaults, and click **OK**. Now that you have a database, you can open that node and view all tables, views, and stored procedures.

So let's create a table. Right-click the **Tables** node and click **Table**. Now create the table in the following figure. Make sure to set the **Id** column as Primary Key and make it auto-increment (which is what the **Identity Specification** property does).

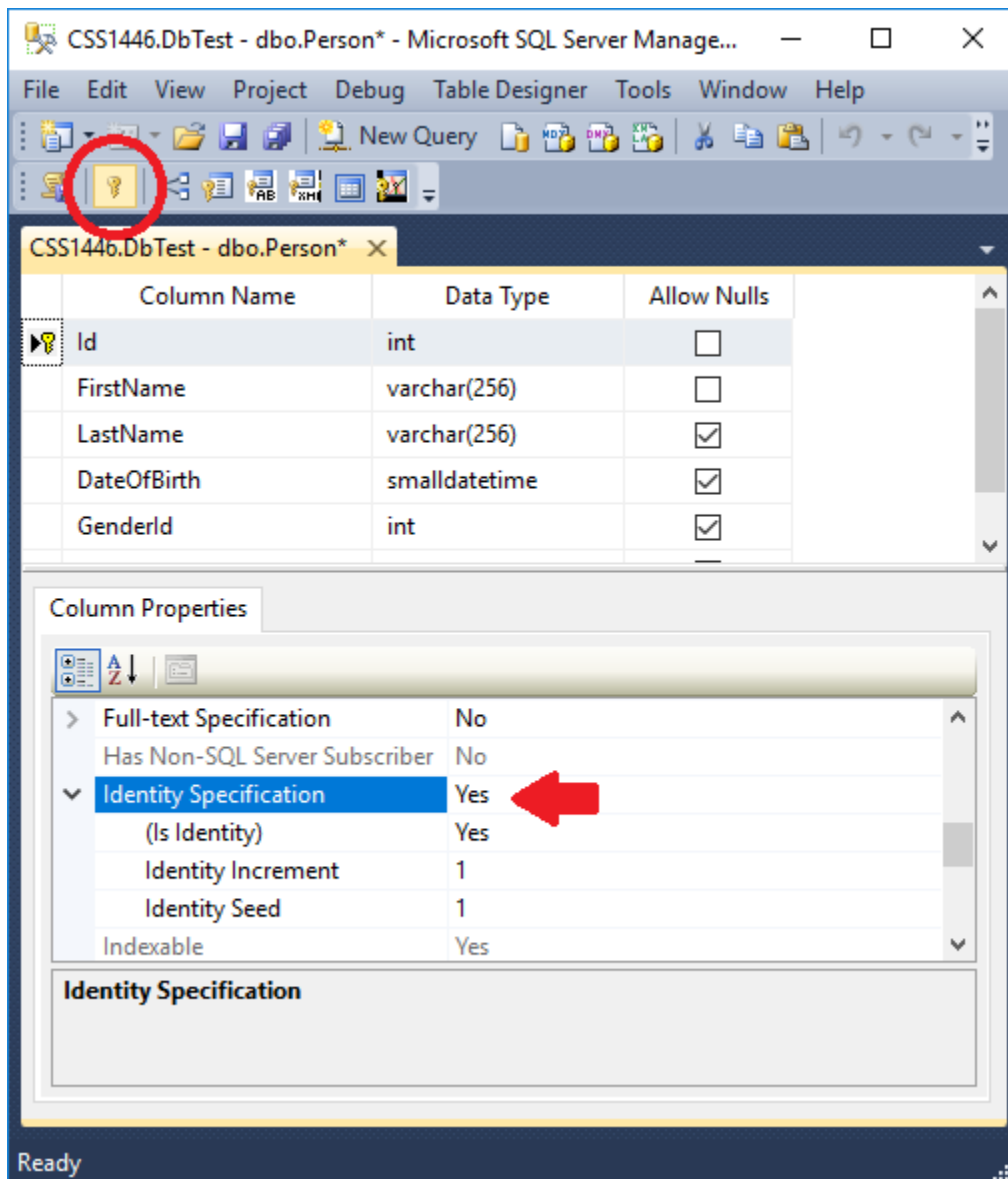


Figure 2: Person table

When saving, name the table Person. To see the table in your Object Explorer, you may have to refresh (right-click the **Tables** node and click **Refresh**). Notice that I've picked datatype **int** for **GenderId**, which seems a little odd at first. However, I'd like to create another table that has all the possible genders and refer to that table here.



Note: I've called the table *Person* (singular) and not *Persons* or *People* (plural). The naming of tables can be rather religious for some. One compelling reason for singular naming is that it will make your life a lot easier when you're going to use an Object

Relational Mapper (ORM). For more arguments (for and against), check out this thread on [StackOverflow](#).



Note: Related is the naming for GenderId in Person (rather than just Gender). One reason for this naming is that it really is an ID referring to a gender. And again, this naming convention will make your life easier when using an ORM. That said, I've seen (and used) just Gender many times as well.

In the same manner as before, create the following table (no Identity Specification on Id this time):

	Column Name	Data Type	Allow Nulls
▶ 🔑	Id	int	<input type="checkbox"/>
	Code	varchar(16)	<input type="checkbox"/>
	Description	varchar(64)	<input type="checkbox"/>
			<input type="checkbox"/>

Figure 3: Gender table

When saving this table, name it Gender.

Now go back to your **Person** table. If you've closed the tab, you can right-click the table and click **Design**. Now right-click anywhere, and then click **Relationships**. Add a new one, click **Tables and Columns Specification**, and make sure it looks as follows:

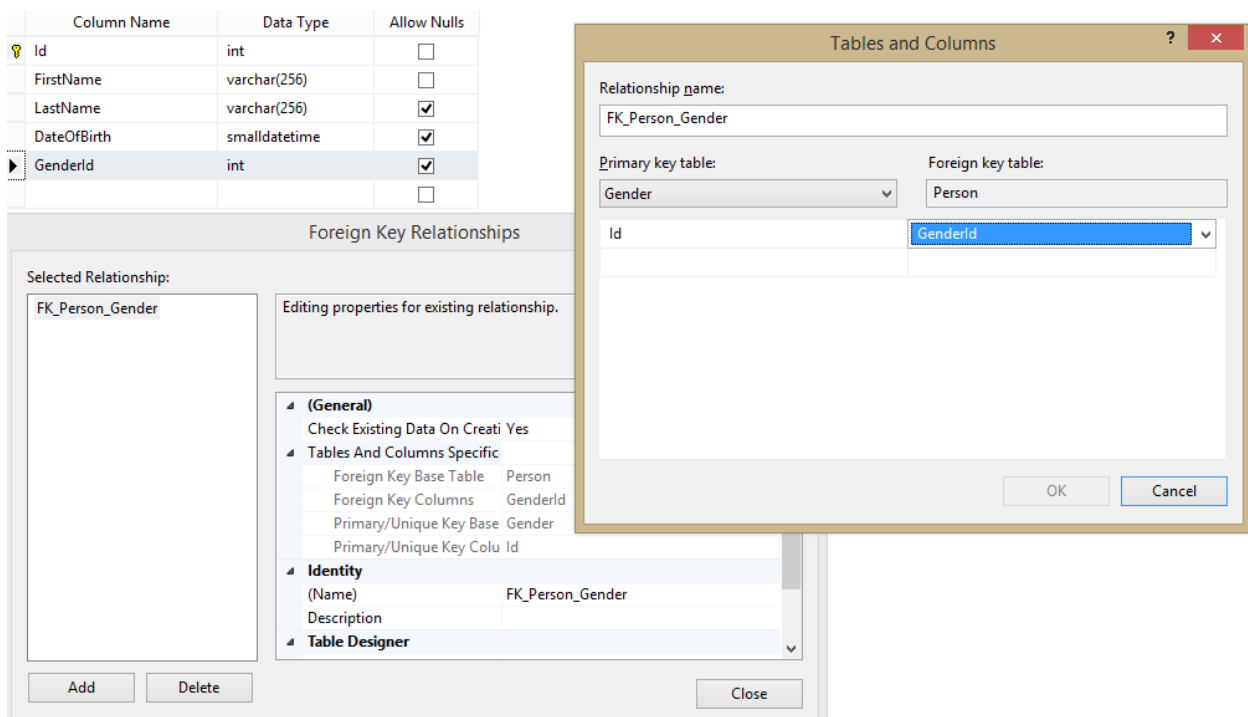


Figure 4: A foreign key relationship

SSMS will automatically name your foreign key FK_Person_Gender, so you can leave all the rest and save. You have now told SQL Server that the **GenderId** column in the **Person** table may only contain values that are also a value in the **Id** column of the **Gender** table.



Note: Many people believe that relational databases are called so because it is possible to define foreign key relationships between tables. That is not true though. A foreign key is nothing more than a constraint, guaranteeing data integrity. The actual reason relational databases are called so is because the mathematical term for a table is a relation. A relation is an unordered set of values. A set is a collection of unique values. A value in SQL Server is a tuple, which is a collection of values. Uniqueness of a value is guaranteed by a primary key. That also means that any table (or view) without a unique primary key is actually not a relation!

Now, right-click the **Gender** table and click **Edit Top 200 Rows**. It doesn't really matter what you insert, but maybe you should follow my example.

Id	Code	Description
0	UNKNOWN	Not saying
1	MALE	Male
2	FEMALE	Female
NULL	NULL	NULL

Figure 5: Genders

Now we can also insert one or more people in the **Person** table.

	Id	FirstName	LastName	DateOfBirth	GenderId
	1	Sander	Rossel	1987-11-08 00:00:00	1
	2	Bill	Gates	1955-10-28	1
*	NULL	NULL	NULL	NULL	NULL

Figure 6: People

So that was easy. We've set up two tables, created a foreign key, and inserted some data. We are missing one thing though... the scripts! You're going to need the scripts if you ever want to do these steps on a different database (or the same database on another environment). Luckily, SSMS has a few options for generating scripts.

The first method is to right-click an object, select **Script [Object] as >**, and make your pick. This only scripts out entire objects, though. This is probably fine for everything but tables (which can't be overwritten without losing data). For example, the **Person** table **CREATE** script is as follows:

Code Listing 1: CREATE TABLE [dbo].[Person]

```
CREATE TABLE [dbo].[Person](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [varchar](256) NOT NULL,
```

```

        [LastName] [varchar](256) NULL,
        [DateOfBirth] [smalldatetime] NULL,
        [GenderId] [int] NULL,
        CONSTRAINT [PK_Person] PRIMARY KEY CLUSTERED
    (
        [Id] ASC
    )WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
    ) ON [PRIMARY]

GO

SET ANSI_PADDING OFF
GO

ALTER TABLE [dbo].[Person] WITH CHECK ADD CONSTRAINT [FK_Person_Gender]
FOREIGN KEY([GenderId])
REFERENCES [dbo].[Gender] ([Id])
GO

ALTER TABLE [dbo].[Person] CHECK CONSTRAINT [FK_Person_Gender]
GO

```

That's awesome, but do note that we need the **Gender** table first, or this script will fail. We added the foreign key constraint later, but it's all here in this one script.

For updates to tables, we can go into the Design window, add a column, change a type, add a constraint, and then right-click, and choose **Generate Change Script**. For example, I've added a **LastLoginTime** column, and I've changed **GenderId** to be non-nullable.

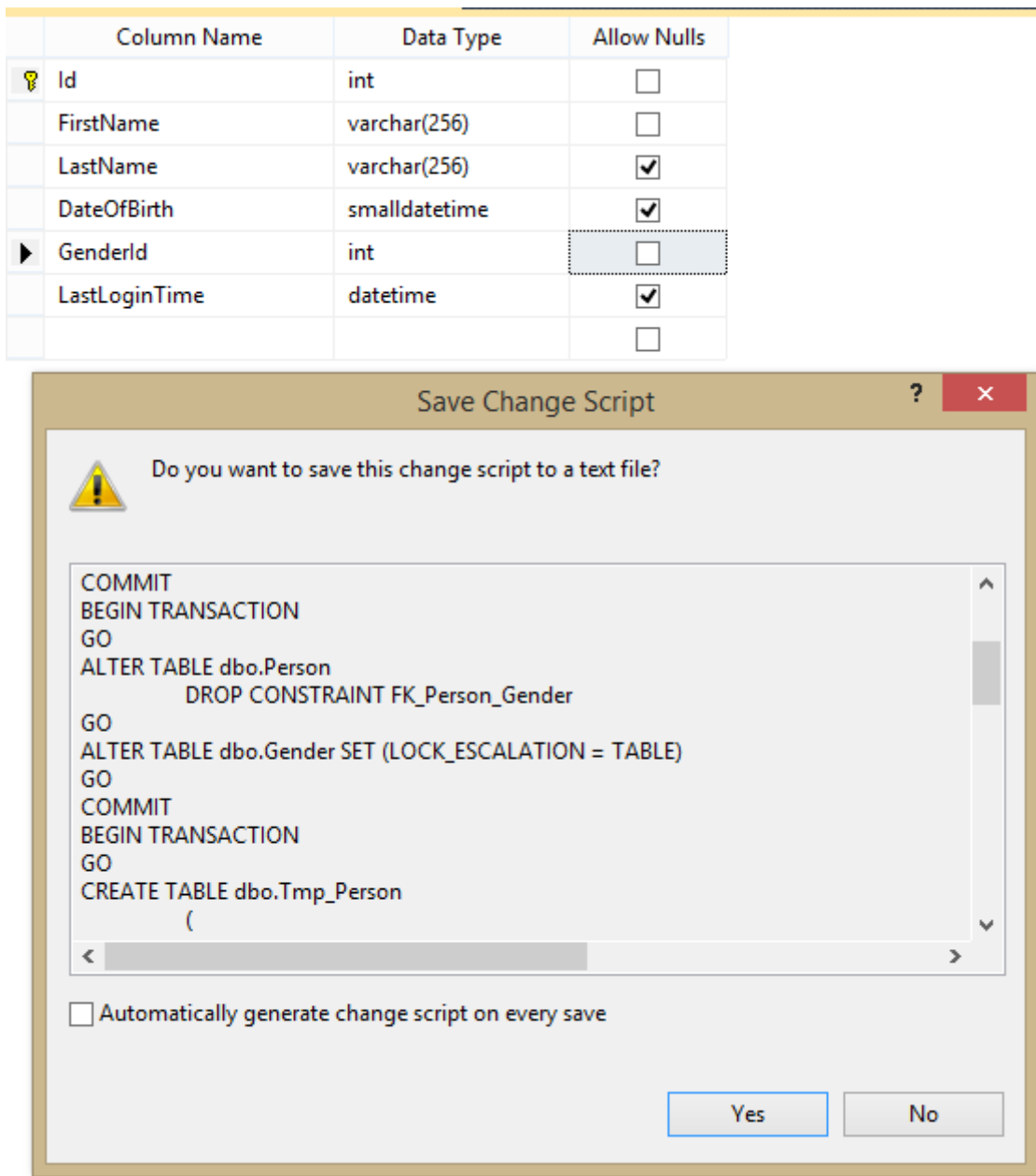


Figure 7: Save Change Script

As you can see, SQL Server actually creates a temporary table, drops the table, and makes the changes. It does more than you'd think! You can save this script and then execute it on another environment effortlessly (or even automated). Unfortunately, dropping and recreating a table is pretty slow and resource-heavy, so you might want to write your own change scripts anyway.

You can open a new query window using the **New Query** button in the menu near the upper-left side of the screen, or by using **Ctrl + N**. Be mindful to make sure you execute scripts (using the **Execute** button or **F5**) on the correct database—a common mistake is executing scripts on the master database, which is selected as default.

The following simple script will also add the **LastLoginTime** column and make **GenderId** non-nullable.

Code Listing 2: ALTER TABLE

```
ALTER TABLE dbo.Person
ADD LastLoginTime DATETIME

ALTER TABLE dbo.Person
ALTER COLUMN GenderId INT NOT NULL
```



Tip: *It is possible to execute only certain parts of scripts in a single query window. Simply select the part of the query that you want to execute and hit F5. This is especially handy when you're testing some software and you'd like to inspect the data, reset the data, run your code again, and repeat. Write a SELECT statement and an UPDATE/INSERT/DELETE statement in a single window, and you can execute the two (or more) statements independently, without having to switch windows all the time.*

Last, but not least, you can script a database with all or some selected objects, and with or without data. Right-click your database, go to **Tasks** >, and then choose **Generate Scripts**. Go through the wizard and experiment with it (it's not very complicated, but it has a lot of options). Here is the generated script for the **Gender** table, including data.

Code Listing 3: CREATE TABLE [dbo].[Gender] including data

```
CREATE TABLE [dbo].[Gender](
    [Id] [int] NOT NULL,
    [Code] [varchar](16) NOT NULL,
    [Description] [varchar](64) NOT NULL,
    CONSTRAINT [PK_Gender] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO
INSERT [dbo].[Gender] ([Id], [Code], [Description]) VALUES (0, N'UNKNOWN',
N'Not saying')
GO
INSERT [dbo].[Gender] ([Id], [Code], [Description]) VALUES (1, N'MALE',
N'Male')
GO
INSERT [dbo].[Gender] ([Id], [Code], [Description]) VALUES (2, N'FEMALE',
N'Female')
GO
```


That was a brief introduction to SQL Server Management Studio. The tree view combined with right-click context menus work very well. Of course, you can use all kinds of shortcut keys if you want; just look them up in the menus. We're going to use SSMS more often throughout this book.



Tip: When creating new objects in your database, you will not get full design time support when writing queries against these newly created objects. They won't show up in autocomplete, and SSMS will tell you these objects don't exist if you manually type them in your query. Your queries will run successfully, though. Even a database refresh will not help you here. For a long time, I thought restarting SSMS was the only way to load new objects into the SSMS object cache. It turned out Ctrl + Shift + R also does the trick—no more red squiggly lines under objects you've just created!

Chapter 2 ADO.NET

Now that we have a database, let's see how we can get data from that database into our application, or from our application into our database. An obvious choice nowadays would be an Object Relational Mapper (ORM) that automatically creates classes from our tables and knows how to read and store data with a simple, single command. It can be that easy. However, a lot is still happening that you do not know about. It is my belief that you can't truly understand an ORM if you haven't done the same thing without an ORM. In fact, I've met developers who have never used anything other than an ORM, and were absolutely clueless when it didn't support some feature they needed. Or who were amazed when their ORM spewed out huge queries at 100 times a second! If you don't really understand your ORM, you're going to have a hard time (and a slow application). So let's work without an ORM first, and see what happens at a (somewhat) lower level.

Open up Visual Studio and create a new Console Application in C# (or Visual Basic if you like, but my examples will be in C#). The .NET Framework has a set of components for working with data sources (such as databases or spreadsheets), collectively called ADO.NET. Microsoft first introduced ADO, which is short for ActiveX Data Objects, long before the .NET Framework. Most of ADO.NET is in the **System.Data** namespace, which is automatically available when you create a new project.

Let's say we want to retrieve a person from the database and use it in our software. The first thing we'd need is a connection to the database. For now, we're assuming we'll always use a SQL Server database. In this case, we're going to make extensive use of the **System.Data.SqlClient** namespace.

Establishing a connection

The first thing we'll need is an actual connection. Without a connection, we'll never be able to do anything with our database.

The connection string

In order to set up a connection, we'll need a *connection string*: a text value specifying various options necessary to set up a connection, such as the server on which our instance is running, the database to connect to, login credentials, a name for the connection, timeout, language, and more.

You can connect to any type of database using a connection string, and every database vendor has their own format. Luckily, we have websites such as connectionstrings.com to help us out. The typical connection string looks as follows:

```
Server=server\instance; Database=database; User Id=username;  
Password=password.
```

Or, in case you choose integrated security, instead of User Id and Password you can put in **Trusted_Connection=True**; or **Integrated Security=True**; or even **Integrated Security=SSPI** (they're all the same).

Likewise, **Data Source=server\instance** is a much-used alternative to **Server=server\instance**, and **Integrated Security=database** is a much-used alternative to **Database=database**.

So let's fill in the blanks and create a connection. My connection string looks as follows: **Server=Laptop23\SQLEXPRESS; Database=SuccinctlyExamples; Integrated Security=SSPI**.

Creating a connection

Now let's create a connection in code and open it. A short note on the code: I'm using `$"..."` syntax for strings, which is a new feature in C# 6.0. It's basically a shorthand for `string.Format`.

Code Listing 4: Creating a SqlConnection

```
try
{
    using (SqlConnection connection = new SqlConnection(
        @"Server=LAPTOP23\SQLEXPRESS; Database=SuccinctlyExamples;
        Integrated Security=SSPI"))
    {
        connection.Open();
        // The database is closed upon Dispose() (or Close()).
    }
    Console.WriteLine("Successfully opened and closed the database.");
}
catch (Exception ex)
{
    Console.WriteLine($"Something went wrong while opening a connection to
    the database: { ex.Message }");
}
Console.WriteLine("Press any key to close.");
Console.ReadKey();
```

The first line of code creates a new **SqlConnection** instance (found in **System.Data.SqlClient**) and passes in the connection string. A lot of objects we will use need to be properly disposed to release resources (in this case the database connection), so we will properly wrap the **SqlConnection** in a using block. The connection is automatically closed when the **Close()** or **Dispose()** method is called (by the using block). The **connection.Open()** opens the connection.

A lot can go wrong in these few lines of code, hence the **try-catch** block. Try making a typo in the connection string (for example, **SServer**), use a non-existing database (any random value),

supply invalid credentials, or try to open the connection twice (without closing in between). All of those will result in an **Exception** being thrown.

Configuring your connection string

Having the connection string hard-coded in C# may be convenient during development, but is not very practical (or secure) in a production environment. You will need to change it, rebuild, and redeploy every time your database or environment (development, test, production) changes.

Connection strings are often stored in config files. The configuration file for your application is usually different for each environment. The big advantage to storing various settings in a configuration file is that you can change the behavior of your application without actually changing and redeploying your application, allowing different settings in multiple environments. Storing configuration strings in configuration files is so common that the .NET config file actually has a **connectionStrings** section. I've added the **SuccinctlyDB** connection string to the app.config file.

Code Listing 5: connectionStrings section in config file

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add name="SuccinctlyDB" connectionString="Server=LAPTOP23\SQLEXPRESS;
Database=SuccinctlyExamples; Integrated Security=SSPI" />
  </connectionStrings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6" />
  </startup>
</configuration>
```

We can now easily retrieve the connection string from the config file. First we will need to add a project reference to **System.Configuration**. Now we can easily replace the first line of code to use the value from the config file.

Code Listing 6: Use connection string from the config file

```
string connectionString =
    ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
using (SqlConnection connection = new SqlConnection(connectionString))
```

Notice that when the connection string is not present in the config file, **connectionString** will be empty and an **Exception** will be raised when trying to open the connection ("The ConnectionString property was not initialized"). Without the **?.** null-conditional operator, we would get a **NullReferenceException** when trying to get the **ConnectionString** property instead.

Building a connection string

In some use cases, you might want to edit the connection string or create it from scratch (for example, when writing the SSMS login form). For this use case, we can use a **SqlConnectionStringBuilder**. The **SqlConnectionStringBuilder** does pretty much what its name implies; it creates connection strings using various input parameters. Let's look at a simple example.

Code Listing 7: The *SqlConnectionStringBuilder*

```
SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
builder.DataSource = @"LAPTOP23\SQLEXPRESS";
builder.InitialCatalog = "SuccinctlyExamples";
builder.IntegratedSecurity = true;
using (SqlConnection connection = new
SqlConnection(builder.ConnectionString))
```

Notice that the **SqlConnectionStringBuilder** actually uses the terms **DataSource** and **InitialCatalog** instead of **Server** and **Database**. Also, **IntegratedSecurity** is set to **true** instead of SSPI (or **Trusted_Connection**).

We can also use the **SqlConnectionStringBuilder** to edit connection strings. The constructor is overloaded, so you can insert a connection string to be used as a base.

Code Listing 8: Editing a connection string

```
string baseConnectionString =
ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
SqlConnectionStringBuilder builder = new
SqlConnectionStringBuilder(baseConnectionString);
builder.IntegratedSecurity = false;
builder.UserID = "sander";
builder.Password = "password";
using (SqlConnection connection = new
SqlConnection(builder.ConnectionString))
```

In this example we're setting **UserID** and **Password** in favor of **Integrated Security**. Of course, trying to open the connection will now fail, as these credentials are clearly invalid.

Querying your database

The next step is using our connection to retrieve some data. This requires a **SqlCommand** to hold and execute our queries. Let's select our **Person** table using the query **SELECT Id, FirstName, LastName, DateOfBirth, GenderId FROM Person**. To do this, we'll need to create the **SqlCommand**, pass it our query and a **SqlConnection**, and have it execute the query.



Tip: Never use *SELECT ** in your production code or SQL queries. *SELECT ** retrieves all columns in a table, which is not always what you want. By using *SELECT*

** you're losing control over your code and what it retrieves. `SELECT *` may yield unintended results. For example, when used in a view, `SELECT *` will select all fields that were present at the time the view was created, not all fields that are present when the view is queried (so removing a column will still break your view). Overall, it's best to avoid `SELECT *` except for some simple, ad-hoc queries.*

Reading data

When using the `ExecuteReader` function on a `SqlCommand` instance, its select query is executed and we get a `SqlDataReader` that holds the retrieved data. A `SqlDataReader`, unfortunately, is not very easy to work with. I don't know why, but it requires quite a bit of typing. The `SqlDataReader` reads the result set row by row (forward only), and you can access columns by index or name. Obviously, selecting by index is not very readable, especially in big queries (who knows what column index 43 is?), but getting the index by name returns the value as object and requires the programmer to make the proper conversions. You can get the index of a column by name as well, and then use the `SqlDataReader` methods to get the correct values. Then, if no value is present (some say the value is `NULL`), you will have to make a conversion to a default value or `null` yourself. Fun fact: `NULL` from your database is a whole different something than `null` in C#. A database `NULL` is represented in C# by the static `DBNull.Value`. Tedious, to say the least. We'll probably want to store the results in some custom objects too, so we'll need to create a `Person` class. Let's see what this looks like.

First, let's create a simple `Person` class.

Code Listing 9: A Person class

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public int? GenderId { get; set; }
}
```

And now for the code to get all people from the database:

Code Listing 10: Usage of the `SqlDataReader`

```
try
{
    List<Person> people = new List<Person>();
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "SELECT Id, FirstName, LastName, DateOfBirth, GenderId FROM
        dbo.Person",
        connection))
```

```

{
    connection.Open();
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            var p = new Person();

            // Get Id by index...
            object idByIndex = reader[0];
            // ...And make the correct conversion.
            int idByIndexCast = Convert.ToInt32(idByIndex);

            // Get Id by name...
            object idByName = reader[nameof(Person.Id)];
            // ...And make the correct conversion.
            int idByNameCast = Convert.ToInt32(idByName);

            // Or get the Id index by name...
            int idIndex = reader.GetOrdinal(nameof(Person.Id));
            // ...And use the SqlDataReader methods.
            p.Id = reader.GetInt32(idIndex);

            int firstNameIndex =
reader.GetOrdinal(nameof(Person.FirstName));
            p.FirstName = reader.GetString(firstNameIndex);

            int lastNameIndex =
reader.GetOrdinal(nameof(Person.LastName));
            if (!reader.IsDBNull(lastNameIndex))
            {
                p.LastName = reader.GetString(lastNameIndex);
            }

            int dateOfBirthIndex =
reader.GetOrdinal(nameof(Person.DateOfBirth));
            if (!reader.IsDBNull(dateOfBirthIndex))
            {
                p.DateOfBirth = reader.GetDateTime(dateOfBirthIndex);
            }

            int genderIdIndex =
reader.GetOrdinal(nameof(Person.GenderId));
            if (!reader.IsDBNull(genderIdIndex))
            {
                p.GenderId = reader.GetInt32(genderIdIndex);
            }

            people.Add(p);
        }
    }
}

```

```

    }
    // The database is closed upon Dispose() (or Close()).
}
Console.WriteLine("Successfully opened and closed the database.");

foreach (Person p in people)
{
    Console.WriteLine($"{p.FirstName} {p.LastName} was born on
{p.DateOfBirth}");
}
}
catch (Exception ex)
{
    Console.WriteLine($"Something went wrong while opening a connection to
the database: { ex.Message }");
}
Console.WriteLine("Press any key to close.");
Console.ReadKey();

```

First of all, notice that I'm declaring my **List<Person>** at the top so I can use it outside of the **using** blocks. It's generally best practice to open a connection, get your data right away, and immediately close your connection. Your database connection remains open until it is explicitly closed, and when using a **SqlDataReader**, you can't close it until everything is read. So just read, close, and use your data later!

Creating the **SqlCommand** is pretty straightforward. And as promised, the **SqlDataReader** is a little less straightforward. The **SqlDataReader** works with a table, of which a single row is accessible at a time. The first row (if any) will become accessible when **SqlDataReader.Read** is called. Any subsequent call on **Read** will move to the next row, if any. If no (next) row is available, **Read** will return **false**. To check if any rows are available, you can use the **SqlDataReader.HasRows** property. The **GetOrdinal** function returns the index of the column name (**nameof** only works here because I've named my properties the same as my database columns; if one of them changes, this code will break, but if both change, all your code will keep working).

In this example we see **GetInt32**, **GetString**, and **GetDateTime**. There are more **Get[Type]** methods, like **GetInt16**, **GetInt64**, **GetBoolean**, **GetChar**, and more. They convert the SQL type to the appropriate CLR type. If a value is missing, the **Get[Type]** methods will throw an **Exception**. So for nullable fields, we'll need to check for **NULL** using **IsDBNull**. Alternatively, there's a **GetValue** function that will just return an object. You can use it if you don't need the type of a value. To check for **NULL**, use **DBNull.Value**.

Code Listing 11: SqlDataReader.GetValue(string)

```

object value = reader.GetValue("SomeColumn");
bool isDBNull = value == DBNull.Value;

```

Now here's another interesting one. What if we wanted to return multiple result sets? Say we change the select query and retrieve all genders. The **SqlDataReader** can retrieve multiple result sets and access them in the same manner it can access rows. The reader is on the first

result set by default, but if you have any subsequent result sets, you can call **NextResult** and the reader will move to the next result set. You can call **Read** again to move to the first row (if any).

Code Listing 12: SqlDataReader with multiple result sets

```
List<Person> people = new List<Person>();
List<Gender> genders = new List<Gender>();
string connectionString =

ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
using (SqlConnection connection = new SqlConnection(connectionString))
using (SqlCommand command = new SqlCommand(
    "SELECT Id, FirstName, LastName, DateOfBirth, GenderId FROM
    dbo.Person;"
    + "SELECT Id, Code, Description FROM Gender;",
    connection))
{
    connection.Open();
    using (SqlDataReader reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            // Process people...
        }
        reader.NextResult();
        while (reader.Read())
        {
            Gender g = new Gender();
            g.Id = reader.GetInt32(0);
            g.Code = reader.GetString(1);
            g.Description = reader.GetString(2);
            genders.Add(g);
        }
    }
}
```

So does it have to be this difficult to read data from a database? There are some alternatives, which we'll look at later in this book. Using a **SqlDataReader** is, however, the best-performing way (when done correctly) to read data from a database.

By the way, if your query returns a single value (a table of one row and one column), you can use **ExecuteScalar** instead. This will return an object that can be cast to the correct type.

Code Listing 13: Usage of ExecuteScalar

```
public string GetPersonName(int id)
{
    string connectionString =
```

```

    ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "SELECT FirstName FROM dbo.Person WHERE Id = @Id", connection))
    {
        command.Parameters.Add("Id", SqlDbType.Int).Value = id;
        connection.Open();
        object result = command.ExecuteScalar();
        string firstName = null;
        if (result != DBNull.Value)
        {
            firstName = (string)result;
        }
        return firstName;
    }
}

```

Parameterization

Before we continue, I'd like to address a very, VERY important topic: parameterization. I can't tell you enough how important it is to use parameters. This is the "practice safe sex" of programming. Unfortunately, just as people have unsafe sex, programmers still don't always use parameters. The result is slow queries, and even more important, security breaches and leaked data. The worst part is that it happens to really big IT companies like Sony, Symantec, and SAP. There is this fun [SQL Injection Hall of Shame](#) that you really don't want to be in. Really—**use parameters**.

Now that you know parameterization is really important, let's look at what it is exactly, and how to implement it. Usually, when working with a database, you're not selecting entire tables. Most of the time you're going to use some filter, like `WHERE Id = x`, where `x` is a variable. With the knowledge I've shared so far, you might be tempted to implement a `GetPeopleByName` function as follows.

Code Listing 14: Non-parameterized query

```

public List<Person> GetPeopleByName(string firstName)
{
    List<Person> people = new List<Person>();
    string connectionString =

    ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "SELECT Id, FirstName, LastName, DateOfBirth, GenderId FROM
        dbo.Person "
        + $"WHERE FirstName = '{ firstName }'",
        connection))
    {

```

```

        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                Person p = new Person();
                // Read the data...
                people.Add(p);
            }
        }
    }
    return people;
}

```

Sure, it gets you a list of people with the given first name. It also generates the exact SQL statement that you would use in SSMS to get people with the given first name. So what's the problem? The `firstName` variable is probably a value the user entered in some text field, either on the web or in a desktop application. If the user enters a name like "D'artagnan," this will break your code because the apostrophe ends the string in SQL Server (pretty annoying). Don't go around and escape names yourself (replace ' with " or some such); that's just a lot of hassle.

Far more dangerous is if a user (or more likely, a malicious hacker) enters the first name **"John'; USE master; DROP DATABASE SuccinctlyExamples; GO --"**? Try it out and bam! There goes your database (yes, it's really gone). This is called SQL Injection. A user is now able to alter SQL statements directly, and by doing this, can get access to sensitive data and destroy that data. There is actually a legendary [xkcd comic](#) about SQL Injection that I think every developer should have hanging in the office.

This is really just database basics, yet many people get it wrong—and not just beginners. SAP, Yahoo, LinkedIn, and even the FBI and NASA have [fallen victim](#) to this simple yet so dangerous exploit. All in all, tens of thousands of applications (and programmers) have been left unsecure because people don't parameterize their queries. Unbelievable, right?

Let's see how we can parameterize in .NET. Luckily, this is very easy (making it even more amazing that so many people don't do it). We can simply place a parameter in the SQL query and add its value using the `SqlCommand (Sql)Parameters` collection.

Code Listing 15: Parameterized query

```

public List<Person> GetPeopleByName(string firstName)
{
    List<Person> people = new List<Person>();
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "SELECT Id, FirstName, LastName, DateOfBirth, GenderId FROM
        dbo.Person "
        + "WHERE FirstName = @FirstName",

```

```

        connection))
    {
        command.Parameters.AddWithValue("FirstName", firstName);
        connection.Open();
        using (SqlDataReader reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                Person p = new Person();
                // Read the data...
                people.Add(p);
            }
        }
    }
    return people;
}

```

Notice that the query now has a placeholder, **@FirstName** (parameters in SQL Server start with **@**), where the name is supposed to be. As an added bonus, we don't have to remember to put quotes around it (since it's a string) because SQL Server will do that for us. The actual value for **@FirstName** is pushed to the **SqlCommand** using **command.Parameters.AddWithValue("FirstName", firstName);** (the **@** in the parameter name is optional in this context). Not often do I see a more self-explanatory piece of code. If you push in a value like **D'artagnan** now, everything will be fine because SQL Server knows this is a single value, and will escape it for you. The same goes for **"John"; USE master; DROP DATABASE SuccinctlyExamples; GO --"**.

Yet this is still not completely right. It's safe, but not optimal. The thing is, and I'll get back to it, SQL Server creates an execution plan for each query. This plan basically contains instructions for the SQL engine to get the queried data in a manner that's probably pretty fast. The creation of an execution plan takes some time, but luckily SQL Server caches the plan and will reuse it in the future if the same query comes along. This is a double-edged sword, but I'll get back to that. For now, we want to reuse query plans as much as possible.

So for a query **"SELECT FirstName, LastName FROM dbo.Person"** this is straightforward. If the same query comes along, the cached plan will be used. But what if the plan contains parameters such as **"SELECT FirstName, LastName FROM dbo.Person WHERE Id = @Id"**? The value for **@Id** will probably be different the next time we call this query. Still, as long as **@Id** has the same *type*, but not necessarily the same *value*, the query plan will be reused. And there's the crux: we never specified a *type* for our parameter. The **AddWithValue** method infers the type, but does so from the data passed in to it—not the type of column in the database. The problem here is that the value **"Sander"** gets inferred to an **nvarchar(6)** while the value **"Bill"** becomes an **nvarchar(4)**.

Another problem, and one possibly more devastating to performance, is if a type gets inferred incorrectly. For example, the **FirstName** database field is a **varchar(256)**, but your parameter has the value **"Sander"**. This will be converted to **nvarchar(6)** by .NET. Those are different types, and in order to compare values of these types, SQL Server will have to implicitly cast every value in the **FirstName** column to **nvarchar(6)** before comparing. Although SQL Server

can optimize some of these differences, there comes a time when this is going to bite you in the backside.

So in order to fix this issue, we should call **Parameters.Add** instead. Unfortunately, Microsoft didn't leave us with an easy overload where we can specify type, size, precision, and scale along with the value. In **varchar(5)**, **varchar** is the type and 5 the size. In **decimal(10, 5)**, **decimal** is the type, 10 the precision, and 5 the scale. **Parameters.Add** has a few overloads; most will create the **SqlParameter** for you, and one simply takes a **SqlParameter** as input. All of them return the added **SqlParameter**.

Code Listing 16: A SqlParameter with the correct type and size

```
command.Parameters.Add("FirstName", SqlDbType.VarChar, 256).Value =  
"Sander";
```

So you see, parameterizing your queries is not only secure, robust, and performing, but also not very hard.

Create, update, delete

Inserting, updating, and deleting data goes pretty much the same way. Instead of **ExecuteReader**, we use **ExecuteNonQuery**, which only returns the number of rows affected.

Code Listing 17: Insert statement

```
public int InsertPerson(Person person)  
{  
    string connectionString =  
  
    ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    using (SqlCommand command = new SqlCommand(  
        "INSERT INTO dbo.Person (FirstName, LastName, DateOfBirth,  
        GenderId) "  
        + "VALUES (@FirstName, @LastName, @DateOfBirth, @GenderId)",  
        connection))  
    {  
        command.Parameters.Add("FirstName", SqlDbType.VarChar, 256).Value =  
person.FirstName;  
        object dbLastName = person.LastName;  
        if (dbLastName == null)  
        {  
            dbLastName = DBNull.Value;  
        }  
        command.Parameters.Add("LastName", SqlDbType.VarChar, 256).Value =  
dbLastName;  
  
        object dbDateOfBirth = person.DateOfBirth;  
        if (dbDateOfBirth == null)
```

```

        {
            dbDateOfBirth = DBNull.Value;
        }
        command.Parameters.Add("DateOfBirth",
SqlDbType.SmallDateTime).Value = dbDateOfBirth;

        object dbGenderId = person.GenderId;
        if (dbGenderId == null)
        {
            dbGenderId = DBNull.Value;
        }
        command.Parameters.Add("GenderId", SqlDbType.Int).Value =
dbGenderId;
        connection.Open();
        return command.ExecuteNonQuery();
    }
}

```

Again, the whole null to **DBNull.Value** conversion is pretty annoying. Here are two nifty (extension) methods that can make this kind of code a lot shorter. Unfortunately, they work on ALL types, whether they're database-compatible or not, so wrong usage will result in an **Exception**.

Code Listing 18: Some parameter utils

```

public static class DbUtils
{
    public static object ToDbParameter<T>(this T? value)
        where T : struct
    {
        object dbValue = value;
        if (dbValue == null)
        {
            dbValue = DBNull.Value;
        }
        return dbValue;
    }

    public static object ToDbParameter(this object value)
    {
        object dbValue = value;
        if (dbValue == null)
        {
            dbValue = DBNull.Value;
        }
        return dbValue;
    }
}

```

The usage now looks as follows.

Code Listing 19: Usage of the utils

```
command.Parameters.Add("FirstName", SqlDbType.VarChar, 256).Value =
person.FirstName;
command.Parameters.Add("LastName", SqlDbType.VarChar, 256).Value =
person.LastName.ToDbParameter();
command.Parameters.Add("DateOfBirth", SqlDbType.SmallDateTime).Value =
person.DateOfBirth.ToDbParameter();
command.Parameters.Add("GenderId", SqlDbType.Int).Value =
person.GenderId.ToDbParameter();
```

This looks a lot nicer.

The **UPDATE** method looks pretty much the same; the biggest difference is that we now also need an **Id** parameter for the **WHERE** clause.

Code Listing 20: Update statement

```
public int UpdatePerson(Person person)
{
    string connectionString =
ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "UPDATE dbo.Person "
        + "SET FirstName = @FirstName,"
        + " LastName = @LastName,"
        + " DateOfBirth = @DateOfBirth,"
        + " GenderId = @GenderId "
        + "WHERE Id = @Id",
        connection))
    {
        command.Parameters.Add("Id", SqlDbType.Int).Value = person.Id;
        command.Parameters.Add("FirstName", SqlDbType.VarChar, 256).Value =
person.FirstName;
        command.Parameters.Add("LastName", SqlDbType.VarChar, 256).Value =
person.LastName.ToDbParameter();
        command.Parameters.Add("DateOfBirth",
SqlDbType.SmallDateTime).Value = person.DateOfBirth.ToDbParameter();
        command.Parameters.Add("GenderId", SqlDbType.Int).Value =
person.GenderId.ToDbParameter();
        connection.Open();
        return command.ExecuteNonQuery();
    }
}
```

And finally, the **DELETE** method.

Code Listing 21: Delete statement

```
public int DeletePerson(Person person)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "DELETE FROM dbo.Person "
        + "WHERE Id = @Id",
        connection))
    {
        command.Parameters.Add("Id", SqlDbType.Int).Value = person.Id;
        connection.Open();
        return command.ExecuteNonQuery();
    }
}
```

Stored procedures

Executing stored procedures is really not that different from all the previous examples. Whether you need to call **ExecuteReader**, **ExecuteScalar**, or **ExecuteNonQuery** depends on the nature of the procedure. You can pass in the name of the procedure to the command and change the **CommandType** to **StoredProcedure**. Suppose we created a stored procedure, **GetFirstName**, that returns the first name of a person based on ID. The call would look as follows.

Code Listing 22: Calling a Stored Procedure

```
public string GetPersonName(int id)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand("GetFirstName", connection))
    {
        command.CommandType = CommandType.StoredProcedure;
        command.Parameters.Add("Id", SqlDbType.Int).Value = id;
        connection.Open();
        object result = command.ExecuteScalar();
        string firstName = null;
        if (result != DBNull.Value)
        {
            firstName = (string)result;
        }
        return firstName;
    }
}
```


Stored procedures can make use of output parameters, which are also easy to implement. Suppose we used an output parameter for the first name. Simply create a parameter, set its **Direction** to **Output**, and read the value after execution.

Code Listing 23: An output parameter

```
public string GetPersonName(int id)
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand("GetFirstName", connection))
    {
        command.CommandType = CommandType.StoredProcedure;
        command.Parameters.Add("Id", SqlDbType.Int).Value = id;
        SqlParameter param = command.Parameters.Add("FirstName",
            SqlDbType.VarChar, 256);
        param.Direction = ParameterDirection.Output;
        connection.Open();
        command.ExecuteNonQuery();
        string firstName = null;
        if (param.Value != DBNull.Value)
        {
            firstName = (string)param.Value;
        }
        return firstName;
    }
}
```

Data sets

Retrieving data and manually mapping to your own C# classes is quite a bit of tedious work. An easier method of getting the data from your database directly into memory is by using the **SqlDataAdapter**. The **SqlDataAdapter** is pretty much obsolete technology, but I want to briefly discuss it for completeness, and because you might still find code that uses it. It's also a little introduction to the next chapter, Entity Framework.

The **SqlDataAdapter** maps the data in your database to a **DataSet** or **DataTable**. A **DataTable** is like your database table in C# memory. A **DataSet** is a collection of **DataTables** with references, constraints, etc. What's awesome is that **CREATE**, **UPDATE**, and **DELETE** statements can be generated automatically. Using **DataTables**, you can also pass table parameters to SQL Server.

To populate a **DataTable** with people from the database, we simply call the **Fill** method on the **SqlDataAdapter** and give it a **DataTable**.

Code Listing 24: Populating a DataTable

```
public DataTable GetPeople()
{
    DataTable people = new DataTable();
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "SELECT Id, FirstName, LastName, DateOfBirth, GenderId FROM
        dbo.Person", connection))
    using (SqlDataAdapter adapter = new SqlDataAdapter(command))
    {
        adapter.Fill(people);
    }
    return people;
}
```

The **SqlDataAdapter** takes a **SqlCommand** in its constructor. The **SqlCommand** is the command that's used to select the data. There are a few overloads, one taking the SQL **SELECT** statement and a connection string. The adapter manages the connection, so there is no need to open and close it explicitly.

Now this looks wonderful. The code is a lot shorter than what we had previously, but how can we read the data from a **DataTable**? Unfortunately, the **DataTable** is a collection of rows, which in turn is a collection of columns. So to read everything, we have to loop through the rows and then loop through the columns. The columns can be accessed by index or name. The values are all objects—so there is still (un)boxing, and we still need to cast—and even **DBNull** is still **DBNull**. So basically, mapping to custom C# objects is just as hard as with the **SqlDataReader**.

The strength of **DataSets** becomes apparent when you use them directly in your code. This makes sense, for example, in WinForms environments with binding. You can update your **DataSet** directly and the **DataSet** keeps track of changes.

Code Listing 25: Update, insert, and delete with the adapter

```
public void UpdatePeople()
{
    DataTable people = new DataTable();
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand(
        "SELECT Id, FirstName, LastName, DateOfBirth, GenderId FROM
        dbo.Person", connection))
    using (SqlDataAdapter adapter = new SqlDataAdapter(command))
    using (SqlCommandBuilder builder = new SqlCommandBuilder(adapter))
    {
        adapter.Fill(people);
        builder.Update(people);
    }
}
```

```

{
    // Creates columns, defines primary keys, foreign keys, etc.
    adapter.FillSchema(people, SchemaType.Source);

    adapter.Fill(people);

    // If you followed my examples, I am the first
    // person in the database, let's change my name.
    people.Rows[0]["FirstName"] = "John";

    // Delete Bill.
    people.Rows[1].Delete();

    // And add Satya.
    people.Rows.Add(new object[] { null, "Satya", "Nadella", new
DateTime(1967, 8, 19), 1 });

    // Sander is updated, Bill is deleted, Satya is added.
    // All in a single line of code!
    adapter.Update(people);
}
}

```

First, I've added the **SqlCommandBuilder** and passed it the **SqlDataAdapter**. The **SqlCommandBuilder** builds **CREATE**, **UPDATE**, and **DELETE** **SqlCommand**s based on the **SELECT** query. It doesn't do this particularly well, by the way, so you may want to do this manually (the commands are properties of the adapter). After that, we must fill the schema using **FillSchema**, of the **DataSet** or **DataTable**. This will fetch the table's schema from the database and provide the **DataTable** with information on columns, types, primary keys, etc. The **Fill** method gets the data. After that we can update, delete, and insert rows to the **DataTable**. Each **DataRow** will keep track of its own state and changes. Last, but not least, the **SqlDataAdapter** is able to insert, update, and delete the data (in that order) from the **DataTable**.

As I mentioned, you can use **DataTables** to pass in table-valued parameters to stored procedures. You can create the following type and procedure in your database to try this one out.

Code Listing 26: Create type and stored procedure

```

CREATE TYPE FirstAndLastName AS TABLE
(
    FirstName VARCHAR(256) NOT NULL,
    LastName VARCHAR(256) NULL
)
GO

ALTER PROCEDURE InsertPeople
    @People FirstAndLastName READONLY
AS
BEGIN

```

```

SET NOCOUNT ON;

INSERT INTO dbo.Person
    (FirstName, LastName)
SELECT FirstName, LastName
FROM @People

END
GO

```

Now, to execute this stored procedure from code, we can pass in a **DataTable**.

Code Listing 27: DataTable as parameter

```

public void ExecInsertPeople()
{
    string connectionString =
        ConfigurationManager.ConnectionStrings["SuccinctlyDB"]?.ConnectionString;
    using (SqlConnection connection = new SqlConnection(connectionString))
    using (SqlCommand command = new SqlCommand("InsertPeople", connection))
    {
        command.CommandType = CommandType.StoredProcedure;

        DataTable people = new DataTable();
        people.Columns.Add("FirstName", typeof(string));
        people.Columns.Add("LastName", typeof(string));
        people.Rows.Add(new object[] { "Tim", "Cook" });
        people.Rows.Add(new object[] { "Mark", "Zuckerberg" });

        command.Parameters.Add("People", SqlDbType.Structured).Value =
people;

        connection.Open();
        command.ExecuteNonQuery();
    }
}

```

There's a lot more you can do with **DataTables** and **DataSets**, but I wouldn't recommend using this technology for database access unless you absolutely have to.

ADO.NET abstractions

As you've seen, I've used **SqlConnection**, **SqlCommand**, and **SqlDataReaders** in the previous examples. They work well with SQL Server, but what if you're using Oracle, MySQL, or another relational database?

One thing you should know about ADO.NET is that there are a lot of abstractions. The **SqlConnection**, for example, inherits from **DbConnection**, which implements the

IDbConnection interface. If you're working with Oracle, you're probably going to make use of an **OracleConnection**, which inherits from **DbConnection**. Likewise, we have an **OleDbConnection** and an **OdbcConnection**. They work together with **OracleCommand**, **OleDbCommand**, and **OdbcCommand**, which all inherit from **DbCommand**. Other vendors have implemented their own versions of these classes.

So you see, if you know how to use one, you pretty much know how to use them all. You can use one of these base classes throughout your software and get the correct type using Dependency Injection instead. The **DbProviderFactory** class is a base factory for constructing specific types of connections, commands, parameters, etc.

Chapter 3 Entity Framework Database First

Everything in the previous chapter was a lot of typing, a little tedious, and not very easy to do. It's not very hard to create an abstraction layer that automatically maps fields from your select statements to properties on C# objects, but it won't be very pretty. To do something like that properly is a little harder, but fortunately a few such abstraction layers already exist. In this chapter we'll take a look at Microsoft's Entity Framework (EF), an Object Relational Mapper (ORM) that does everything we did in the previous chapter, but a lot easier.

An ORM, as the name implies, maps objects from a relational database to objects in your code. The Entity Framework has two modes of development. First, and most traditional, is that EF generates C# classes from an already-existing database. Second is that you code out your classes with properties, foreign key relations, and primary keys, and have EF generate the database for you. These two modes are called Database First and Code First, respectively. A third option exists where EF creates C# classes from an existing database, but pretends that you've used code first, allowing you to use Code First with an existing database. In this chapter I'm going to show you Database First and Code First. From there it shouldn't be difficult to use the Code First from an existing database option as well.

To extensively discuss EF, we'd need a not-so-succinct book. In fact, I've got a book counting over 600 pages from an older version of EF when Code First wasn't even an option. Add to that the LINQ queries you can write, for which you can read a completely different book, and the T4 Template technology that EF uses, and you'll understand that I can really only scratch the surface of EF in this chapter. The goal of this chapter is not to make you an expert of EF, but to apply the lessons from the previous chapter to an ORM and discuss some of the finer "gotcha!" details of EF that will save you a lot of gray hairs later on.

Database First

Let's start with Database First, as it is pretty straightforward. Create a new C# Console Application Project, save it, and install EntityFramework through NuGet. In the menu, go to **Tools > NuGet Package Manager > Manage NuGet Packages for Solution**. Browse and search for **EntityFramework**, select your project in the list of projects on the right side of the window, and install it. Now, add a new item to your project and choose **ADO.NET Entity Data Model**, and name it SuccinctlyExamplesModel.

The Entity Data Model Wizard presents you with a few options; select **EF designer from database**. In the next screen, you can set up your connection properties and pick a database. Pick the **SuccinctlyExamples** database you created. After you've created your connection, you get a tree view with database objects. Check the tables (**dbo.Gender** and **dbo.Person**), make sure you check **Pluralize or singularize generated object names**, and click **Finish**.

You'll see a file called **SuccinctlyModels.edmx** added to your project. The edmx (Entity Data Model eXtension) file holds all your mappings from database to code. Right-clicking the edmx file and selecting **Open with** allows you to select an XML Editor and view your mappings in XML. You should never need the XML, but if you get deeper into EF, you'll find yourself editing

or inspecting it more than you'd like. For this tutorial, though, we're not going to look at it. So if you double-click on the edmx file, you'll get a diagram showing all the database objects you imported earlier. If you want to import more objects or update existing objects, simply right-click somewhere in the model and click **Update Model from Database**. Unfortunately, if you import a table, you will always get all properties, and an update will also update all already imported tables.

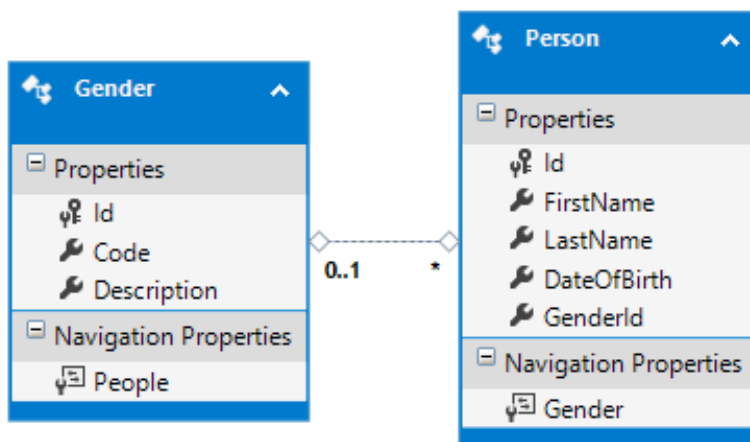


Figure 8: Entity Framework diagram

Notice that **Gender** has a **People** attribute and **Person** has a **Gender** attribute (if you named **GenderId** just **Gender**, the **Gender** attribute would've been named **Gender1**, yuck!). The thin line between the entities describes the relation. A **Gender** has 0 to n **People** (the * part, 0..n) and a **Person** has 0 or 1 **Gender** (the 0..1 part, 0 because it's nullable). As you can see, these are called **Navigation Properties**. You can rename everything here if you like (for example, if you had **Gender1**, you could rename it to **GenderEntity**, or you could rename **Gender** to **GenderId** and **Gender1** to just **Gender**). You may also remove properties from the model if they're nullable, but if you ever want them back, you'll have to add them manually. The best part is that these are now C# classes you can use.

Code Listing 28: Using your entities in C#

```

Person p = new Person();
p.FirstName = "Mark";
p.LastName = "Zuckerberg";
p.Gender = new Gender();
  
```

The generated code can be found when you expand the edmx file. This is where you will find two T4 files (Text Template Transformation Toolkit); they have the .tt extension. You can expand those as well. One will have your generated entities, and the other will have your Context class, the code representation of your database. The T4 files are templates that will generate your classes using the edmx file. Remember, you can edit the generated files, but once they're regenerated, your changes will be overwritten. If you want to generate some additional code, your best bet is to edit the T4 files.

The connection string, as you might have guessed, is added to your app.config file. The EF connection string has some extra metadata included. It's good practice to put your entities in a

separate project and reference that project from other projects. The App.config or Web.config of the startup project will need this connection string, or you'll get a runtime error. So be sure to copy it (including metadata).

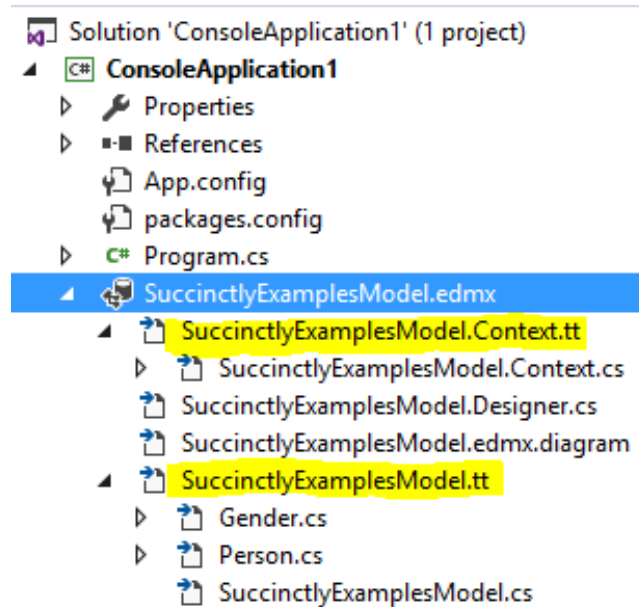


Figure 9: T4 files

If you want to add some code to a single class, you can make use of the **partial** keyword. All your entities are created as partial classes. That means you can “extend” them in a different file.

Code Listing 29: A partial class

```
public partial class Person
{
    public string FullName
    {
        get
        {
            // FirstName and LastName are defined
            // in this class, but in another file.
            return $"{FirstName} {LastName}";
        }
    }
}
```

Let's get some data from our database. The first thing we need is our context class, which inherits from **DbContext**. It has a property for each of our tables and a method for each of our stored procedures.

Code Listing 30: Use the DbContext

```
using (var context = new SuccinctlyExamplesEntities())
{
    // Access the database.
    List<Person> people = context.People.ToList();
}
```

That's how easy it is to fetch all people from your database and load them into memory!

IQueryable

Unfortunately, there's a lot going on in that one line of code, so let's elaborate. First of all, **context.People** doesn't really do anything yet—it's the **ToList** that forces database access. This is a VERY important detail. The **context.People** property is a **DbSet<Person>**, which in turn is an **IQueryable<Person>**. The **IQueryable<T>** interface looks like any collection (**IEnumerable**) on the outside, but acts very differently. When using **IQueryable**, a query is built for the data source, in this case SQL Server. The query is only executed when the **IQueryable** is enumerated, like when **ToList** or **foreach** is called. That allows you to create queries using **Where**, **OrderBy**, or **Select** without ever going to the database. If we had to access the database many times, you could imagine this becoming very slow indeed!

Let's build a more advanced query.

Code Listing 31: An IQueryable<Person>

```
using (var context = new SuccinctlyExamplesEntities())
{
    // Don't access the database just yet...
    IQueryable<Person> query = context.People
        .Where(p => p.GenderId == 2)
        .OrderBy(p => p.LastName);

    // Access the database.
    List<Person> women = query.ToList();
}
```

This also allows you to create queries based on conditions.

Code Listing 32: Build queries using conditionals

```
using (var context = new SuccinctlyExamplesEntities())
{
    bool orderResults = false;

    // Don't access the database just yet...
    IQueryable<Person> query = context.People
        .Where(p => p.GenderId == 2);
```

```

    if (orderResults)
    {
        query = query.OrderBy(p => p.LastName);
    }

    // Access the database.
    List<Person> women = query.ToList();
}

```

As you can see, this is a *lot* less code than we used in Chapter 1. As an added bonus, the code reads a lot easier, too (well, once you get used to the LINQ syntax). Creating different queries, depending on your needs, becomes a breeze, as you can reuse parts of your queries (like object-oriented SQL queries!).

Another concern here is parameterization. Don't worry, everything is parameterized just fine. It's just that in the previous example there are no parameters. We query for the people with a **GenderId** of 2. The value 2 is a constant, so our query won't have any parameters. If, somewhere else in the code, we query for a **GenderId** of 1, we'll get a new query with a new plan. So suppose we want to parameterize this query. What would we need to do? Easy—create a variable!

Code Listing 33: A parameterized query

```

using (var context = new SuccinctlyExamplesEntities())
{
    // Not parameterized.
    context.People.Where(p => p.GenderId == 2).ToList();

    int female = 2;
    // New, parameterized query.
    // Doesn't re-use the previous plan.
    context.People.Where(p => p.GenderId == female).ToList();

    int male = 1;
    // Parameterized, uses same query plan as above.
    context.People.Where(p => p.GenderId == male).ToList();
}

```

Of course there are a few downsides to **IQueryable**s as well. You don't have any influence on what queries are generated. Some queries are so monstrous that it really pays off to just write them yourself, and this is where some SQL knowledge really comes in handy. Sometimes EF just can't generate clean SQL queries from your **IQueryable**, for example, when joining two completely unrelated tables. EF will generate a query and it will get you your data, but it won't be fast or elegant. Speaking of performance, using EF comes with a performance penalty. If those few milliseconds matter (and often they don't), use the techniques we discussed in Chapter 2.

Another thing you cannot do in an **IQueryable** is use any function or property that is not known to the data source. That makes sense, as the query will have to be translated to SQL. The following code, for example, will not work.

Code Listing 34: Invalid query

```
using (var context = new SuccinctlyExamplesEntities())
{
    Person sander = context.People.SingleOrDefault(p => p.FullName ==
    "Sander Rossel");
}
```

Unfortunately, it compiles just fine. Only at runtime, when actually compiling the query, will you get an error because there is no **FullName** column in your table, so be sure to always run your queries against your actual data source. You might be surprised that the following query will work as expected.

Code Listing 35: Valid code

```
using (var context = new SuccinctlyExamplesEntities())
{
    List<Person> peopleWithS = context.People
        .Where(p => p.FirstName.ToLower() == "sander")
        .ToList();
}
```

The **ToLower** function is known in SQL Server, and the EF team made sure that this function translates correctly. In theory, the **StartsWith** function could be translated to SQL (using SQL's **LEFT**) as well, but this is not supported. For some additional supported functions (such as **LEFT**, **RIGHT**, **DIFFDAYS**, etc.), check out the **System.Data.Entity.DbFunctions** and **System.Data.Entity.SqlServer.SqlFunction** classes. It's also possible to add your own functions to EF, but this is outside the scope of this book.

There's another pitfall to the code in the previous example. Consider the following code.

Code Listing 36: Lazy-loading

```
using (var context = new SuccinctlyExamplesEntities())
{
    List<Gender> genders = context.Genders.ToList();
    foreach (Gender g in genders)
    {
        foreach (Person p in g.People)
        {
            Console.WriteLine($"{p.FirstName} is a {g.Description}.");
        }
    }
}
```

How often will this code access the database? The only correct answer is "I don't know." We first get all the possible genders, which is one database call. But then we loop through the genders and we're accessing the **People** of that **Gender**. Guess what? The **People** are not yet loaded into our memory, so each time we access **People**, EF will access the database to get all people for that **Gender**.

In this case we have three genders, so we have the call to get all genders, and then an additional call for each gender to get the people, which is a total of four database calls. In many scenarios you really don't know how many entities you're getting, and before you know it, you're doing thousands of database calls! This is called lazy-loading, and can be turned off in your EF diagram. However, with lazy-loading turned off, your **People** will not be loaded and it may seem you have no people at all. I've seen systems get very slow because of lazy-loading, but when used correctly, it's pretty awesome. Luckily, EF does some caching for us, so once a Navigation Property is loaded, EF won't access the database again. "Think before you query" is the message!

Let's take a look at some cool extensions, which make working with **IQueryable** a lot easier. Working with dates is always a pain, so let's create an **OlderThan** extension method. I must confess I got the SQL query from Stack Overflow and just converted it to LINQ. It's accurate enough for the example.

Code Listing 37: OlderThan extension method

```
public static class Extensions
{
    public static IQueryable<Person> OlderThan(this IQueryable<Person> q,
    int age)
    {
        return q
            .Where(p => (DbFunctions.DiffHours(p.DateOfBirth,
    DateTime.Today) / 8766) > age);
    }
}
```

The usage is simple and readable. The original query would be unreadable to anyone, but it's clear what it does from the name, (**OlderThan**). It's a win-win situation.

Code Listing 35: OlderThan usage

```
using (var context = new SuccinctlyExamplesEntities())
{
    List<Person> people = context.People.OlderThan(45).ToList();
}
```

Remember that the function or property you're using in an **IQueryable** must be present in the data source. That means creating an interface and using that on different types to reuse your extension method is only possible when the interface member has the same name as your entity member. So if you have multiple entities with a **DateOfBirth** and you want to reuse your extension method, that's possible, but only when you implement the extension method as a generic function (after all, **IQueryable** works on entities, not on interface types). The generic must be a **class**, because a **struct** may never be an entity.

Code Listing 36: Using an interface

```
public interface IDateOfBirth
{
    DateTime? DateOfBirth { get; set; }
```

```

}

public partial class Person : IDateOfBirth
{ }

public static class Extensions
{
    public static IQueryable<T> OlderThan<T>(this IQueryable<T> q, int age)
        where T : class, IDateOfBirth
    {
        return q
            .Where(p => (DbFunctions.DiffHours(p.DateOfBirth,
DateTime.Today) / 8766) > age);
    }
}

```

If you have multiple entities with different property names (for example, **Machine** with **ManufacturedDate**), you're out of luck. Maybe **Expression Trees** can help you there.

To get the query that is generated by EF and send to the database, you can use your **DbContext.Database.Log** property, which is an **Action<string>** and outputs any generated SQL to the delegate. You could, for example, log to the Console.

Code Listing 40: Output SQL to the Console

```

using (CodeFirstContext context = new CodeFirstContext())
{
    context.Database.Log = Console.Write;
    context.People.Where(p => p.GenderId == 1).ToList();
}
Console.ReadKey();

```

Expression trees

Another detail in working with **IQueryable** is that it contains many extension methods that are also defined on **IEnumerable**. Or so it seems. The following code looks the same for the **IEnumerable** and the **IQueryable**, but only one of the two will make your application come to a grinding halt!

Code Listing 37: IEnumerable vs. IQueryable

```

using (var context = new SuccinctlyExamplesEntities())
{
    IQueryable<Person> peopleQuery = context.People;
    IEnumerable<Person> peopleEnumr = context.People;
    peopleQuery.Where(p => p.FirstName == "Sander").ToList();
    peopleEnumr.Where(p => p.FirstName == "Sander").ToList();
}

```

The two **Where** methods differ in input parameter, and in what they return. The **IQueryable** takes an **Expression<Func<Person, bool>>** as input and returns an **IQueryable**, whereas the **IEnumerable** takes a **Func<Person, bool>** as input and returns an **IEnumerable**.

```
peopleQuery.Where(p => p.FirstName == "Sander").ToList();
peopleEnumr. (extension) IQueryable<Person> IQueryable<Person>.Where<Person>(System.Linq.Expressions.Expression<Func<Person, bool>> predicate)
Filters a sequence of values based on a predicate.
```

Figure 10: *IQueryable.Where*

```
peopleEnumr.Where(p => p.FirstName == "Sander").ToList();
 (extension) IEnumerable<Person> IEnumerable<Person>.Where<Person>(Func<Person, bool> predicate)
Filters a sequence of values based on a predicate.
```

Figure 11: *IEnumerable.Where*

The **Expression<Func<SomeType, bool>>** is basically a representation of the **Func<SomeType, bool>** in an object graph. So instead of passing in a method like in the **IEnumerable.Where**, you pass in some object that represents the method. In this case the compiler creates the object graph, so you don't need to worry about that.

In the previous example, that means the **IEnumerable.Where** enumerates over the collection, forcing a roundtrip to the database (after all, the **IEnumerable** variable is actually an **IQueryable**), and only after that does it execute the **Where** function. As a result, you'll get ALL people from the database instead of just those with the **FirstName** "Sander". So be very careful that you don't accidentally break your **IQueryable** chain.

Just for fun, here's the **Expression<Func<Person, bool>>** the compiler actually created from that lambda expression.

Code Listing 38: *An Expression<Func<Person, bool>>*

```
ParameterExpression parameter = Expression.Parameter(typeof(Person), "p");
Expression property = Expression.Property(parameter,
typeof(Person).GetProperty("FirstName"));
Expression constant = Expression.Constant("Sander");
Expression binary = Expression.Equal(property, constant);
Expression<Func<Person, bool>> lambda = Expression.Lambda<Func<Person,
bool>>(binary, parameter);
peopleQuery.Where(lambda).ToList();
```

This means that when you want to add your own (extension) methods, you'll have to keep in mind that you're dealing with **Expressions**. And if you want to create some extra layers of abstraction, you might even need to create your own **Expressions** manually. Creating **Expressions** manually is a little too advanced for this book, but let's take a look at an extension method.

What happens if you want to include, for example, additional select logic in your extension method? Just be sure to put in an **Expression**. Failing to do so will give no errors or warnings, but will call the method on **IEnumerable**, forcing a database call before your action executes.

Code Listing 39: Extension with selector

```
public static List<T> SelectOlderThan<T>(this IQueryable<Person> q, int
age,
    Expression<Func<Person, T>> selector)
{
    return q
        .Where(p => (DbFunctions.DiffHours(p.DateOfBirth, DateTime.Today) /
8766) > age)
        .Select(selector)
        .ToList();
}
```

Usage is, again, pretty straightforward.

Code Listing 40: Usage of SelectOlderThan

```
var people = context.People.SelectOlderThan(45, p =>
    new
    {
        Name = p.FirstName,
        DateOfBirth = p.DateOfBirth
    });
```

CRUD

Let's check out the CRUD operations. You won't believe how easy it is.

Code Listing 41: CRUD operations

```
using (var context = new SuccinctlyExamplesEntities())
{
    // Update Sander.
    Person sander = context.People.FirstOrDefault(p => p.FirstName ==
"Sander");
    sander.FirstName = "John";

    // Create Google CEO.
    Person sundar = new Person();
    sundar.FirstName = "Sundar";
    sundar.LastName = "Pichai";
    sundar.DateOfBirth = new DateTime(1972, 7, 12);
    sundar.GenderId = 1;
    context.People.Add(sundar);

    // Delete Mark Zuckerberg.
    Person mark = context.People.FirstOrDefault(p => p.FirstName ==
"Mark");
    context.People.Remove(mark);
}
```

```

    context.SaveChanges();
}

```

That's all there is to it! It's so easy, you don't really need further explanation.

Remember that I said “**SELECT * ...**” can be very harmful to a system. What we're doing now is basically that. If we update our model and **Person** gets 100 new fields, our system suddenly gets a lot slower. We can fix this using **Select**.

Code Listing 42: Using Select

```

using (var context = new SuccinctlyExamplesEntities())
{
    // Using an anonymous type.
    var satyaAnon = context.People.Where(p => p.FirstName == "Satya")
        .Select(p => new
        {
            Id = p.Id,
            FirstName = p.FirstName,
            LastName = p.LastName
        }).SingleOrDefault();
    // Use the anonymous type here...

    // Using a known type.
    PersonModel satyaKnown = context.People.Where(p => p.FirstName ==
"Satya")
        .Select(p => new PersonModel
        {
            Id = p.Id,
            FirstName = p.FirstName,
            LastName = p.LastName
        }).SingleOrDefault();
    // Use the known type here,
    // pass it to other functions,
    // or return it from this function.
}

```

Using this style of querying makes a difference for your update and delete statements as well. Since you don't want to get the entire entity anymore, you'll have to get the ID (given you don't have it yet) and use that to do your updates and deletes. In most scenarios, you'll have the IDs at your disposal, so the selects are not necessary.

Code Listing 43: CRUD without getting entire entities

```

using (var context = new SuccinctlyExamplesEntities())
{
    // Update Sander.
    // Get Sander's ID.
    var sander = context.People.Where(p => p.FirstName == "Sander")

```



```

        .Select(p => new { Id = p.Id }).SingleOrDefault();

    // Create a new Person, but set the Id to an existing one.
    Person update = new Person();
    update.Id = sander.Id;

    // Attach Person with the given Id.
    context.People.Attach(update);

    // Fields that are set after attach will be tracked for updates.
    update.FirstName = "John";

    // Delete Mark Zuckerberg.
    // Get Mark's ID.
    var mark = context.People.Where(p => p.FirstName == "Mark")
        .Select(p => new { Id = p.Id }).SingleOrDefault();
    Person delete = new Person();
    delete.Id = mark.Id;
    context.People.Attach(delete);
    context.People.Remove(delete);

    context.SaveChanges();
}

```

That was a quick overview of the Entity Framework. We've addressed creating a model, importing tables, creating queries, parameterization, lazy-loading, and CRUD operations. Unfortunately, as you've seen, there are many "gotchas" when working with EF (or any ORM, for that matter). I know these things because I've done them wrong in the past. I've seen entire teams, with years of experience, still fall for some of the pitfalls laid out in this chapter. Often have I heard "EF sucks because it is so slow," but more often than not, this was the result of not parameterizing, not knowing when **IQueryable**s are executed (or even just not knowing about **IQueryable** at all), lazy-loading, and selecting too much data (or just poor database design, but that's another subject altogether).

Chapter 4 Entity Framework Code First

Now that we've seen EF Database First, how to use it, and some of its quirks, let's try Code First. You can simply write the classes that would be generated by Database First yourself. Code First is especially awesome when you're working in a team and everyone has their own local copy of the database. Updating databases and inserting data is very easy, and can even be done automatically—no more having to run and update scripts manually. Once again, create a new Console Application and install EF (using NuGet).

Now, add the following classes to your project. Also make sure you add the **CodeFirst** connection string to your config file (without EF metadata, just a “plain” connection string).

Code Listing 44: EF Code First classes

```
public class Person
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public int? GenderId { get; set; }
    public Gender Gender { get; set; }
}

public class Gender
{
    public int Id { get; set; }
    public string Code { get; set; }
    public string Description { get; set; }
    public virtual ICollection<Person> People { get; set; }
}

public class CodeFirstContext : DbContext
{
    public CodeFirstContext()
        : base("CodeFirst")
    { }

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

Those classes may look familiar. It's pretty much what EF database first generated for us earlier. Please note the **virtual** keyword for the **ICollection**. At runtime, EF will create a proxy class for **Gender**, overriding the **People** property, and have it return some **ICollection** that supports lazy-loading. Now simply run your application with the following code.

Code Listing 49: Run EF Code First

```
using (CodeFirstContext context = new CodeFirstContext())
{
    context.People.Add(new Person
    {
        FirstName = "Sander",
        Gender = new Gender
        {
            Id = 1,
            Code = "MALE",
            Description = "A true man!"
        }
    });
    context.SaveChanges();
}
```

This will actually create your database and insert a new person and a new gender.

If you take a look at the database, you'll notice your tables are called **dbo.People** and **dbo.Genders**. **Genders** is not a **DbSet** in your **DbContext**, but it is still a table in the database. EF figured that out through **Person**. There is also a table called **dbo.__MigrationHistory** (note there are two underscore characters); we'll look at that in a bit. Let's first fix the plural naming by overriding **OnModelCreating** in the **DbContext**.

Code Listing 50: Non-pluralizing DbContext

```
public class CodeFirstContext : DbContext
{
    public CodeFirstContext()
        : base("CodeFirst")
    { }

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

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

There are a lot of conventions like that, and you can add your own. Delete the database by right-clicking on it in SSMS and click **Delete**. In the dialog, check **Close existing connections**. When you run the application again, your tables will now be named **dbo.Person** and **dbo.Gender**. If you don't delete your database, you'll get a runtime exception—we'll fix that in a minute.

Looking further at the database, you'll notice that all string properties are translated to **nvarchar(max)**. You probably don't want this. Furthermore, **FirstName** is non-nullable, so let's fix that too. This is actually pretty easy using data annotations (in the

`System.ComponentModel.DataAnnotations` namespace). Simply add some **Attributes** to your model.

Code Listing 45: DataAnnotations

```
[Required]
[MaxLength(256)]
public string FirstName { get; set; }
[MaxLength(256)]
public string LastName { get; set; }
```

Again, delete your database and run the code again. These fields will now be generated as **nvarchar(256) not null** and **nvarchar(256) null**, respectively. You can tweak a lot more, for example, by not having an autogenerated key, mapping to another column name in the database, changing generated types, or making properties named **Key** a primary key by default.

Code Listing 52: Non-autogenerated key

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public int Id { get; set; }
```

By overriding **OnModelCreating** of **DbContext**, you have a lot of flexibility in customizing generation.

Code Listing 46: Customizing in OnModelCreating

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

    modelBuilder.Entity<Person>()
        .Property(p => p.DateOfBirth)
        .HasColumnName("DOB")
        .HasColumnType("smalldatetime");

    modelBuilder.Properties()
        .Where(p => p.Name == "Key")
        .Configure(p => p.IsKey());
}
```

By using the **NotMappedAttribute**, a property is ignored by EF altogether (for example, a **FullName** property that you're going to compose using **FirstName** and **LastName**).

Code Listing 47: A property that's not mapped to the database

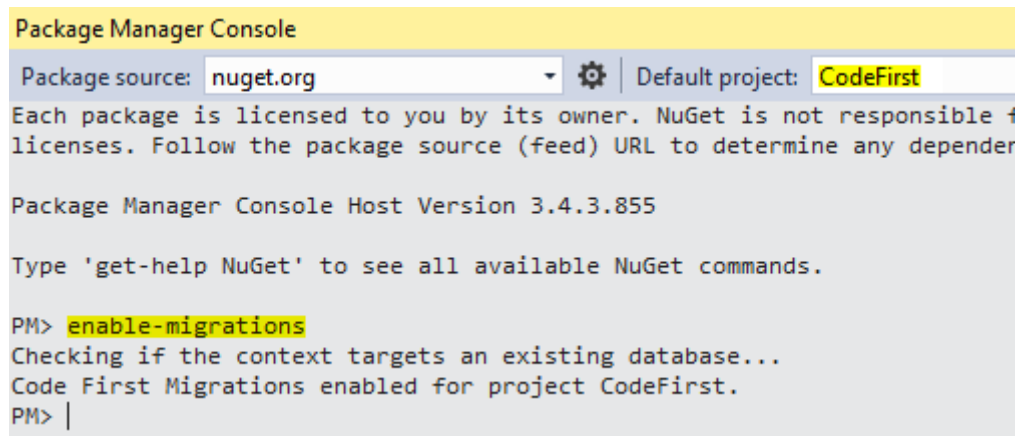
```
[NotMapped]
public int FullName { get; set; }
```

You probably get the gist of Code First: write code and configuration and have EF create a database for you.

Migrations

Up to now, we had to delete the database every time we made a change to our model. That, of course, won't work very well in the real world. EF has a few options for the creation of a database: **CreateDatabaseIfNotExists** (the default), **DropCreateDatabaseIfModelChanges**, and **DropCreateDatabaseAlways**. These are called **DatabaseInitializers**. You probably see some trouble with those. If you have any data or nonmapped entities in your database, you'll lose them when EF drops the database and recreates them (imagine that happening in production). Newer versions of EF introduced the **MigrateDatabaseToLatestVersion**. Up to now that hasn't worked so well—after all, you get an exception when your model changes.

To enable Migrations, use the NuGet Package Manager Console. In the menu, select **Tools > NuGet Package Manager > Package Manager Console**. Make sure the console targets the project that has your **DbContext** class. Next, type **enable-migrations**.



```
Package Manager Console
Package source: nuget.org | Default project: CodeFirst
Each package is licensed to you by its owner. NuGet is not responsible for
licenses. Follow the package source (feed) URL to determine any dependencies.
Package Manager Console Host Version 3.4.3.855
Type 'get-help NuGet' to see all available NuGet commands.
PM> enable-migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project CodeFirst.
PM> |
```

Figure 12: Enable Migrations using NuGet Console

You'll find that a class called **Configuration** will be created for you.

Code Listing 48: Generated Configuration class

```
namespace CodeFirst.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration :
        DbMigrationsConfiguration<CodeFirst.CodeFirstContext>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }
    }
}
```

```

        protected override void Seed(CodeFirst.CodeFirstContext context)
        {
            // ...
        }
    }
}

```

Automatic migrations

For now, you'll want to change **AutomaticMigrationsEnabled** to **true**.

Code Listing 49: Enable automatic migrations

```
AutomaticMigrationsEnabled = true;
```

The next thing we need to do is use this configuration in our **DbContext**. We can set it in the constructor.

Code Listing 50: Configure the DbContext for migrations

```

public CodeFirstContext()
    : base("CodeFirst")
{
    Database.SetInitializer(new
    MigrateDatabaseToLatestVersion<CodeFirstContext,
    Migrations.Configuration>("CodeFirst"));
}

```

Now add a new class, **Company**, and give **Person** two new properties, **CompanyId** and **Company**. Make sure **CompanyId** is nullable, or you might get an error saying that an already existing person cannot be updated (after all, it would have **NULL** for non-nullable **CompanyId**).

Code Listing 51: Make a change to the model

```

public class Person
{
    // ...
    public int? CompanyId { get; set; }
    public Company Company { get; set; }
}
public class Company
{
    public int Id { get; set; }
    public string Name { get; set; }
}

```

Now, run the application using the following code (the previous code won't work, as **Gender** 1 will be re-inserted, but it already exists and we do need some query, or EF won't update the database).

Code Listing 52: Force automatic migration

```
using (CodeFirstContext context = new CodeFirstContext())
{
    context.People.ToList();
}
```

Now, do a few migrations (make some more changes) and check out the **dbo.__MigrationHistory** table. That's where everything is stored, so you can check if some migration was successful.

You may have noticed that adding classes and properties is no problem, but when you try to delete something, you will get an **AutomaticDataLossException**. That may not come as a surprise, as you don't expect that removing a property in a model leads to data loss in a database. However, that may still be exactly what you want. You can configure this in the **Configuration** class.

Code Listing 60: AutomaticMigrationDataLossAllowed

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

Because automatic migrations may be dangerous (after all, you have little control over what's happening), there is an option to disable database initialization (including migrations) in your app.config (or web.config) file. It's actually not a bad idea to disable automatic migrations on your production environment. Chances are that your customers don't even allow it because database changes must go through a DBA or be requested in writing.

Code Listing 53: Disable database initialization in config

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <!-- ... -->
  <entityFramework>
    <!-- ... -->
    <contexts>
      <context type="CodeFirst.CodeFirstContext, CodeFirst"
        disableDatabaseInitialization="true" />
    </contexts>
  </entityFramework>
  <!-- ... -->
</configuration>
```

Another option is to disable it in code.

Code Listing 54: Disable database initialization in code

```
public CodeFirstContext()
```

```

        : base("CodeFirst")
    {
        Database.SetInitializer<CodeFirstContext>(null);
    }

```

By doing this, you must update your database manually. We'll look at some deployment options later.

Code First migrations

An alternative to automatic migrations is Code First migrations. Disable your automatic migrations by setting **AutomaticMigrationsEnabled** on **false** in the **Configuration** class. Now make some changes to your models, for example, by adding **FullName** to your **Person** class.

Code Listing 55: FullName property

```

[MaxLength(512)]
public string FullName { get; set; }

```

Now open up the NuGet Package Manager Console again, make sure the project with your **DbContext** is selected, and type **add-migration "Add Person.FullName"**. In the Migrations folder, you now get a new file called **[current_date_and_time]_Add Person.FullName** containing a class named **AddPersonFullName**. In it, there's some C# update "script" since your last migration. It also has a "script" to revert those changes.

Code Listing 56: A code first migration

```

namespace CodeFirst.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    public partial class AddPersonFullName : DbMigration
    {
        public override void Up()
        {
            AddColumn("dbo.Person", "FullName", c => c.String(maxLength:
256));
        }

        public override void Down()
        {
            DropColumn("dbo.Person", "FullName");
        }
    }
}

```


In the Package Manager Console, type **update-database**, and your database will be updated. You can run your code and query again against an updated database. Make sure you don't give two migrations the same name, or you will get name conflicts in your C# code.

Now add another property on **Person**, for example, **int Age**, and run another **add-migration** "**Add Person.Age**" and **update-database**. Unfortunately, for whatever reason, your last database change has to be rolled back. Luckily, **Up** and **Down** methods are generated, so this should be easy. To revert updates, simply update to an older migration and newer migrations will be rolled back. In the console, type "**update-database -TargetMigration:'Add Person.FullName'**". Run it and **Add Person.Age** will be reverted.

As I was playing around with migrations, I created **FullName**, removed it, added it again, and as a result I was left with two C# classes **AddPersonFullName**. No problem, I just deleted the oldest and ran my migration. However, when rolling back, I got a conflict as **Add Person.FullName** was ambiguous. Luckily, everything you do is saved in **dbo.__MigrationHistory** and you can easily get the unique ID for every migration from there. Simply run "**update-database -TargetMigration:'2016052111803411_Add Person.FullName'**" instead to remove the ambiguity. That said, it's best not to mess up your migrations too much, or you'll find yourself doing a lot of tedious manual fixing.

	MigrationId	ContextKey	Model	ProductVersion
1	201605211411034_InitialCreate	CodeFirst.Migrations.Configuration	0x1F8B080000000000400E559DB6EE336107D2FD07F10F458...	6.1.3-40302
2	201605211411572_AutomaticMigration	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5ADB6EDB3618BE1FB0771074B...	6.1.3-40302
3	201605211413008_AutomaticMigration	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED1ADB6EDB36F47DC0FE41D0D3...	6.1.3-40302
4	201605211416519_AutomaticMigration	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5ADB6EDB3618BE1FB0771074B...	6.1.3-40302
5	201605211423045_AutomaticMigration	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5A596FDC36107E2FD0FF20E8A...	6.1.3-40302
6	201605211444306_AutomaticMigration	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5ADB6EDB3618BE1FB0771074B...	6.1.3-40302
7	201605211514397_Add Person.FullName	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5A596FDC36107E2FD0FF20E8A...	6.1.3-40302
8	201605211751478_Remove Person.FullName	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5ADB6EDB3618BE1FB0771074B...	6.1.3-40302
9	201605211754438_Add Person.FullName	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5ADB6EDB3618BE1FB0771074B...	6.1.3-40302
10	201605211803058_AutomaticMigration	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5ADB6EDB3618BE1FB0771074B...	6.1.3-40302
11	201605211803411_Add Person.FullName	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5ADB6EDC36107D2FD07F10F4D...	6.1.3-40302
12	201605211804453_Add Person.Age	CodeFirst.Migrations.Configuration	0x1F8B080000000000400ED5A5B6FDB36147E1FB0FF20E869...	6.1.3-40302

Figure 13: **dbo.__MigrationHistory**

Another option for you to consider is to get the actual SQL script that Migrations generates. For example, I added **Person.FullName** and generated the script. You can generate the script using **update-database -Script** from the Package Manager Console. You'll get something like the following script.

Code Listing 57: *Generated SQL script*

```
ALTER TABLE [dbo].[Person] ADD [FullName] [nvarchar](512)
INSERT [dbo].[__MigrationHistory]([MigrationId], [ContextKey], [Model],
[ProductVersion])
VALUES (N'201605221001289_AutomaticMigration',
N'CodeFirst.Migrations.Configuration', 0x1F/*...*/ , N'6.1.3-40302')
```

Seeds

Another thing that makes Code First pretty awesome is the ability to seed your database with some initial data. You may have already noticed it when we created the **Configuration** class, but there is a **Seed** method with some example in a comment. So let's make sure that we at least have our **Genders** in place and a **Person** to use for testing.

Code Listing 58: The Seed method

```
protected override void Seed(CodeFirstContext context)
{
    context.Genders.AddOrUpdate(new[]
    {
        new Gender { Id = 0, Code = "UNKNOWN", Description = "Not saying" },
        new Gender { Id = 1, Code = "MALE", Description = "Male" },
        new Gender { Id = 2, Code = "FEMALE", Description = "Female" }
    });
    context.People.AddOrUpdate(p =>
        new
        {
            FirstName = p.FirstName,
            LastName = p.LastName
        },
        new Person
        {
            FirstName = "Sander",
            LastName = "Rossel",
            GenderId = 1
        }
    );
}
```

We can simply insert our three **Genders**. EF will check if the ID is already present in the database and do an insert or update based on that. The **Id** of **Gender** is not an autonumbering field, so we have to set it ourselves. The **Id** of a **Person** is not known to us, though, since it's autonumbering. With autonumbering, there is no way for us to insert a **Person** with a specific **Id**, and EF will just happily insert the same **Person** every time (giving it some autonumber **Id**). We can, however, specify an alternative means of looking up a specific entity. This is the first parameter to **AddOrUpdate**, which takes an anonymous object containing all the fields that make up for a unique **Person**. In this example, I've taken the combination of **FirstName** and **LastName** to be the alternative key. If the combination of **FirstName** and **LastName** isn't present in the database, EF will insert the **Person**, otherwise it will do an update. Now if you run the code, these rows will be inserted into the database if they're not already present.

If you would like to start with a fresh database (including an initial test set) every time you start your debug session, you may opt for a custom **IDatabaseInitializer**. I already mentioned a few, so let's simply create one. The following initializer will always recreate a database and seed some data.

Code Listing 59: A custom DatabaseInitializer

```
public class CleanDbInitializer :
DropCreateDatabaseAlways<CodeFirstContext>
{
    protected override void Seed(CodeFirstContext context)
    {
        context.Genders.AddOrUpdate(new[]
        {
            new Gender { Id = 0, Code = "UNKNOWN", Description = "Not
saying" },
            new Gender { Id = 1, Code = "MALE", Description = "Male" },
            new Gender { Id = 2, Code = "FEMALE", Description = "Female" }
        });
        context.People.AddOrUpdate(p =>
            new
            {
                FirstName = p.FirstName,
                LastName = p.LastName
            },
            new Person
            {
                FirstName = "Sander",
                LastName = "Rossel",
                GenderId = 1
            });
    }
}
```

To get it working, simply go to your **DbContext** and change the constructor. Make sure you don't have any migrations (delete the Migrations folder), or your database won't actually get deleted, and you will get an exception.

Code Listing 60: Usage of CleanDbInitializer

```
public CodeFirstContext()
    : base("CodeFirst")
{
    Database.SetInitializer(new CleanDbInitializer());
}
```

It is also possible to change the initialization using your App.config or Web.config file. Simply remove the **SetInitializer** from your **DbContext** constructor and add the following value in your config file.

Code Listing 61: MigrateDatabaseToLatestVersion with Configuration in config

```
<contexts>
  <context type="CodeFirst.CodeFirstContext, CodeFirst">
    <databaseInitializer
type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[[CodeFirst.CodeFi
rstContext, CodeFirst],
  [CodeFirst.Migrations.Configuration, CodeFirst]], EntityFramework"/>
    </context>
  </contexts>
```

Now we're switching to our custom initializer.

Code Listing 70: Custom initializer in config

```
<contexts>
  <context type="CodeFirst.CodeFirstContext, CodeFirst">
    <databaseInitializer type="CodeFirst.CleanDbInitializer, CodeFirst" />
  </context>
</contexts>
```

As you can see, by using Code First, Migrations, and Seeds, working with multiple local database instances while still making daily database updates becomes a breeze.

Chapter 5 SQL Server Data Tools

Using the Visual Studio SQL Server Data Tools, you can create databases in pretty much the same way you can in SQL Server Management Studio. It gets better though—you can compare and deploy databases with ease. For this chapter, I have undone all the changes we made to the SuccinctlyExamples database in the previous chapters, so I'm starting fresh again.

If you followed all my steps during the Visual Studio installation, you should have the SQL Server Data Tools. If you haven't, you can change your installation by going to the Windows Control Panel, then **Programs and Features**. In the list of programs, look for **Microsoft Visual Studio X (with Update Y)**, select it, and click **Change**. Now make sure you select the Data Tools.

I should mention that I got a new laptop between writing the previous chapters and this one, so instead of LAPTOP23\SQLEXPRESS, you'll see CSS1446. All the above still applies, of course; it's just a name change.

Database Project

Start Visual Studio—you don't even have to start or open a project. In the menu, go to **View** and then **SQL Server Object Explorer**. You should now get a window on the right or left side of the screen (depending on your settings) showing two nodes, SQL Server and Projects. Right-click **SQL Server** and then click **Add SQL Server**. You should now see a tree view similar to that in SSMS.

As you right-click on the objects, you'll notice that you can add tables, columns, and other objects. You can generate scripts of objects, execute scripts, and add or alter data. It's all pretty cool that you can do this in Visual Studio, but you could do this in SSMS as well.

Now here comes the awesome part—right-click the **SuccinctlyExamples** database, and click **Create New Project**. Choose a name for your project, ideally the name of your database (I was lazy and kept the default, **Database1**). You'll probably want to check **Create new solution**, and possibly **Create directory for solution**; they're off by default, so I'll leave it to you. Click **Start**, and a new database project will be created. In the new solution, your Solution Explorer should now look something like this.

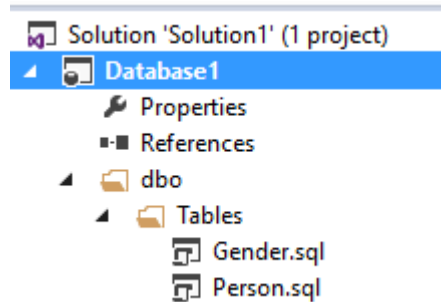


Figure 14: Database project Solution Explorer

An alternative method for creating a database project, even when you don't have an existing database, is by simply starting a new project and choosing **SQL Server Database Project** from the installed templates. In the new project, you can create objects and deploy them, or import an existing database by right-clicking your database project in the Solution Explorer, selecting **Import**, and then selecting **Database**.

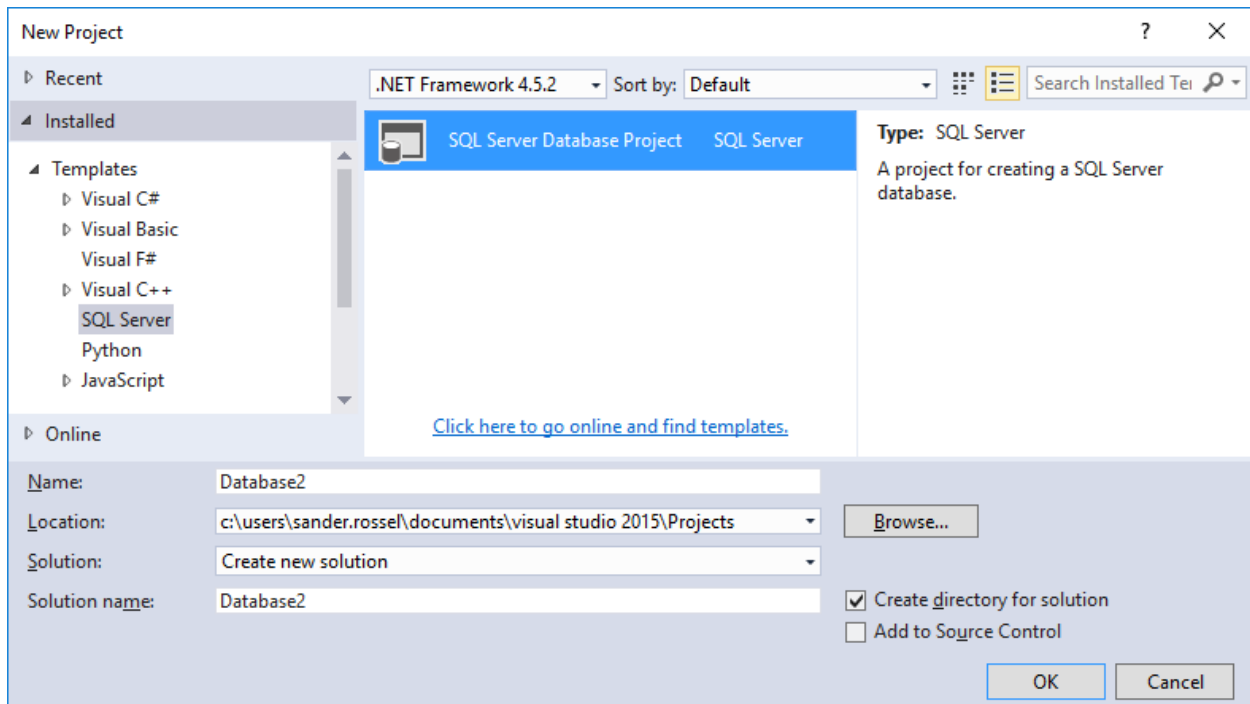


Figure 15: The SQL Server Database Project template

Let's continue with the Database1 project. By double-clicking on a table, you get to the designer, which is pretty similar to that of SSMS. Here you can add new columns and foreign keys, change columns, etc. In the lower part of the screen you'll see the SQL script for the table; it changes as you change the table.

Publishing

Before you do anything, let's assume the SuccinctlyExamples database is production ready, so we want to deploy it. Right-click your database project in the Solution Explorer and choose **Deploy**. In the **Publish Database** window, select a database connection. The following window is a bit confusing, as you're presented some recent connections, but your current connection is hidden. To show your current connection details, click **Show Connection Properties**. You can just pick the **SuccinctlyExamples** database for now.

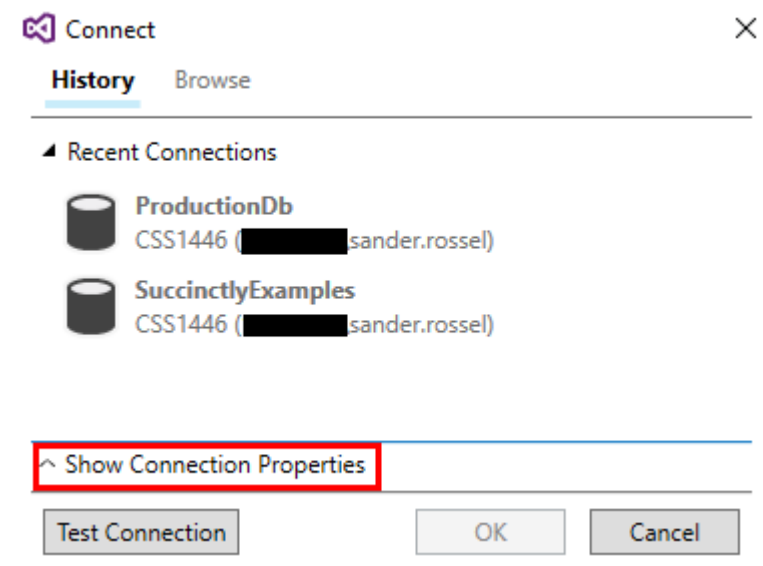


Figure 16: The Connect window

Back in the Publish database window, pick a database name. We're going to create a new database, so let's pick a name that doesn't exist yet. I've called it **ProductionDb**. In a real environment you'd probably have the same database name, but another server instance. Since we don't have spare servers lying around, we're going to use the same server instance, but using a different database name. You can choose a name for the generated script as well. You can enable or disable numerous options under the Advanced menu as well, most notably if data loss is allowed. Unless you uncheck **Block incremental deployment if data loss might occur**, you'll never be able to delete columns in tables that have rows. There are too many options to discuss them all, but be sure to look at them.

Now you can either view the script, examine it, change it, and run it (in the upper left), or just run the script directly. If you run the script directly, it will be placed in the bin folder of your project. Also, a Data Tools Operations window opens, allowing you to open the script directly. Whichever method you choose, you should now have a new ProductionDb database that is the same as the SuccinctlyExamples database. That's pretty awesome!

Updating

Now that we have deployed our database, we're going to get change requests. Let's add a column to the **Person** table. Double-click the **Person.sql** script in the Visual Studio Solution Explorer. You can right-click in the white space of the table designer to add or remove designer

columns such as **Length**, **Description**, and **Identity**. Now simply add a column, let's say **Title**. Save it.

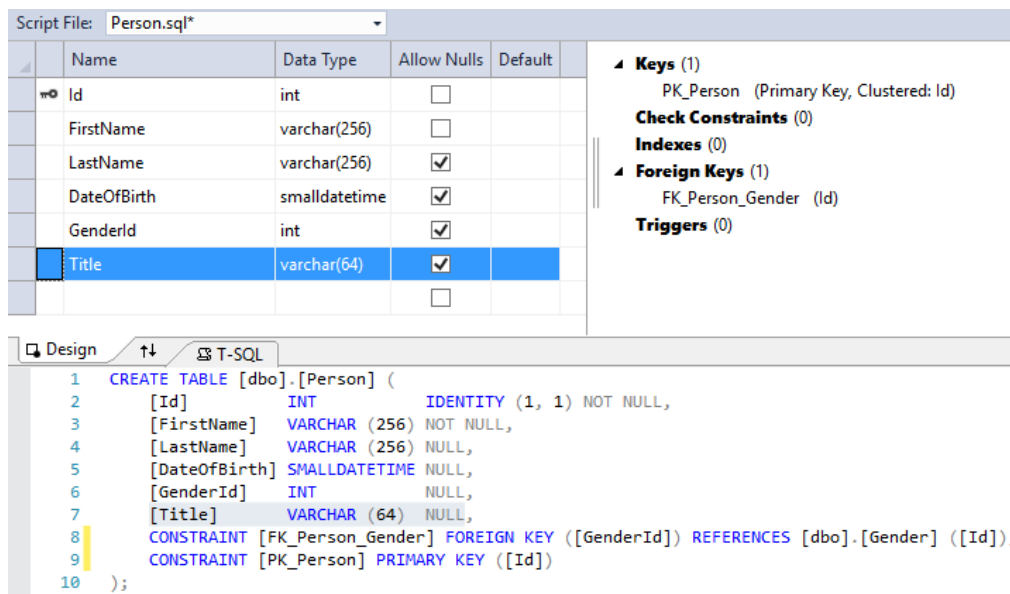


Figure 17: Title column

Now publish the database again, but this time, simply publish to the SuccinctlyExamples database. Instead of a **CREATE** script, Visual Studio will now generate an **ALTER** script. If you run it (directly or by generating the script and then running), your database will now have the new **Title** column.

Updating a table or column—that is, changing it—results in a refactorlog file (or an update thereof). This file is necessary for Visual Studio to figure out that it has to rename the table or column rather than drop the table or column with the old name and create the table or column with the new name. Once you publish the change, a system table is created on your database called **__RefactorLog** (note the two underscores). The table contains a single column indicating which schema changes have been published.

Comparing

Another cool feature of the Data Tools is to compare databases. We have a few options: compare database schemas against other databases, compare schemas against database projects, and compare data between databases.

Let's start with comparing schemas between databases. Right-click on the database project and then click **Schema Compare**. In the window that opens, you can now choose a source and a target. The source is your current project by default. Change the source to the **SuccinctlyExamples** database and the target to the **ProductionDb** database. Now simply click **Compare** in the upper-left corner, and Visual Studio will visually show you the differences. You can now, again, update the target automatically or generate a SQL script and run it manually. You can also pick objects to exclude from the generated script in the **Action** column. So now, simply update the ProductionDb any way you want.

So let's open up SSMS and make a change to the **Person** table. The Visual Studio table designer has no support for Computed Columns, so we're going to add one. In SSMS, open the designer for the **Person** table and add the following column.



The data compare is a little different. Let's make sure we have some data in our database. Run the following script on the SuccinctlyExamples database to insert some data.

```
INSERT INTO [dbo].[Gender] ([Id], [Code], [Description]) VALUES (1,
N'UNKNOWN', N'Not saying')
INSERT INTO [dbo].[Gender] ([Id], [Code], [Description]) VALUES (2,
N'MALE', N'Male')
INSERT INTO [dbo].[Gender] ([Id], [Code], [Description]) VALUES (3,
N'FEMALE', N'Female')
```

This is the kind of data you want to keep synchronized between your databases. Users, even admins, may not be able to enter this data anywhere in the application, and without it your application does not work correctly. So let's make sure we have all the data we need. For some reason we can't do a data compare from our database project, so go to the SQL Server Object Explorer instead. Right-click on either the **SuccinctlyExamples** database or the **Gender** table and select **Data compare**. Set the **SuccinctlyExamples** database as your source (it probably already is) and the **ProductionDb** as your target. Click **Next** to pick the tables and views you wish to compare, and then click **Finish**. You now get an overview of tables that were compared. Only tables and views that have matching names, column names, column types, and primary keys can be compared. Names are case-sensitive. Once compared, you can (as always) create a script and manually execute it, or automatically update the target database.

1 tables and/or views were compared.

Object (check to include in update)	Different Records	Only in Source
<input checked="" type="checkbox"/> Tables <input checked="" type="checkbox"/> [dbo].[Gender]	0	3 (Add 3)
<input type="checkbox"/> Views		
<input type="checkbox"/> Table/View Combination		

Different Records (0)	Only in Source (3)	Only in Target (0)	Identical Records (0)
-----------------------	--------------------	--------------------	-----------------------

3 records exist on the source but not on the target database; 3 records will be added to target (CSS1446.ProductionDb (

Update	Id	Code	Description
<input checked="" type="checkbox"/>	1	UNKNOWN	Not saying
<input checked="" type="checkbox"/>	2	MALE	Male
<input checked="" type="checkbox"/>	3	FEMALE	Female

Figure 19: SQL data compare

While what we've seen here is incredibly useful, it comes at a cost, and with certain dangers. There was a time when I did all my deployments manually using schema compare. And then I accidentally deployed a change that was not meant to go into production yet, and caused an entire factory to go down for about five minutes. Oops! If you're not careful, you might execute scripts that you didn't quite mean to execute. A lot of scripts are generated, and they can be executed with a single button press, so be careful.

Last, but not least, I wanted to mention the data-tier application (for some reason abbreviated as DAC). A DAC is basically a sort of database project. The difference between the two is mostly the way in which it is published. In data-tier applications, publish packages are created by developers, which can then be used by DBAs to perform the updates. If you right-click on the **Databases** node in the Object Explorer in SSMS, you'll notice the options "Deploy Data-tier Application..." and "Import Data-tier Application." It is possible to convert a database project to a DAC, and [vice versa](#). I must confess I've never used it, so I won't say much about it. If you want to know more about them, I suggest you read about it on [MSDN](#) and try it out for yourself.

Chapter 6 Troubleshooting

Once in a while queries go awry. Unfortunately, there can be many reasons for this. Maybe your database has grown to the point where a query that always worked well is now very slow. Maybe one of your environments is missing an index. Maybe a query is using a cached query plan that is not optimal for the current parameter value. In this chapter we're going to look at how we can identify troublesome queries and how we can solve these issues.

For a few examples, we're going to need a database with a little body. Microsoft has a sample database we can use called Adventure Works. You can download it from [CodePlex](#). Alternatively, you can Google (or Bing) AdventureWorks2014 and see what shows up. Download the **Adventure Works 2014 Full Database Backup.zip** file and unzip the .bak file. In SSMS, right-click the **Databases** node in your Object Explorer and then click **Restore Database...** in the context menu. In the Restore Database window, enable **Device**, click the button "...", and add the .bak file you just unzipped. Then simply click **OK**, and the database should now be available.

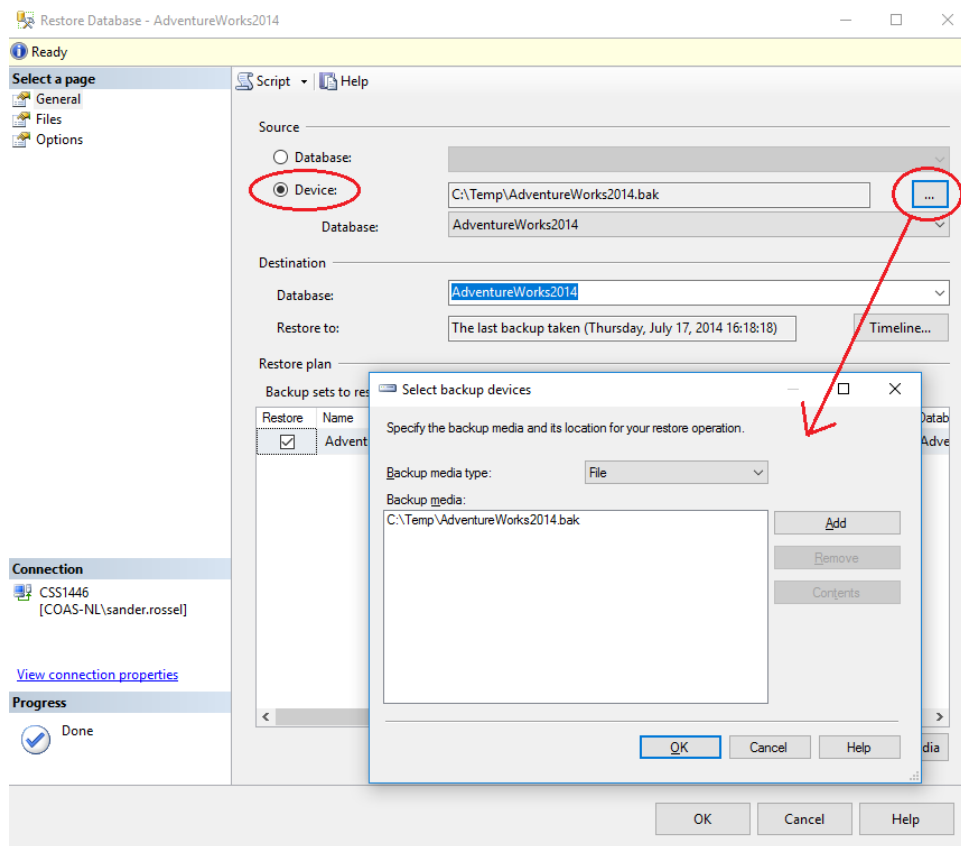


Figure 20: Restoring Adventure Works

Profiling

First of all, we'll need to know which queries are running slowly. For this (and many more) we can use the profiler. I recommend using the profiler often when developing, as it shows you just about everything that happens in a database. For example, you may have a well-performing piece of code, but maybe it only performs well because you're testing locally with few data. The profiler may show you that you're actually sending thousands of queries to your database (maybe because of EF lazy-loading, or because you're executing queries in a loop). I've often run into such scenarios that run smoothly on a test environment, but bring the database to a grinding halt in production (and that's never fun).

To start the profiler, start up SSMS, and in the menu, go to **Tools** and then **SQL Server Profiler**. You can also run it as a stand-alone application from your start menu (or wherever you have it installed). If you start it from SSMS, it will automatically start a new trace. If you're starting it from Windows directly, you can find **New Trace** under the **File** menu. Next, you will have to log in to the instance you want to trace. Be careful when tracing a production environment, as it will put a load on the system. Connect to your local instance, and you should get a window captioned "Trace Properties." On the **General** tab, you can name your trace, save to file, or pick a template. We're going to ignore this tab. Instead, go to the **Events Selection** tab. On this tab you can find all kinds of events that are raised by the database. A few are already selected for you, but there are many more. Check the **Show all events** and **Show all columns** boxes, and you'll see what I mean. For now we're interested in the following events: "**Stored Procedures -> RPC:Completed** and **SP:StmtCompleted**" and "**TSQL -> SQL:BatchCompleted**". Now click **Run** to run the trace.

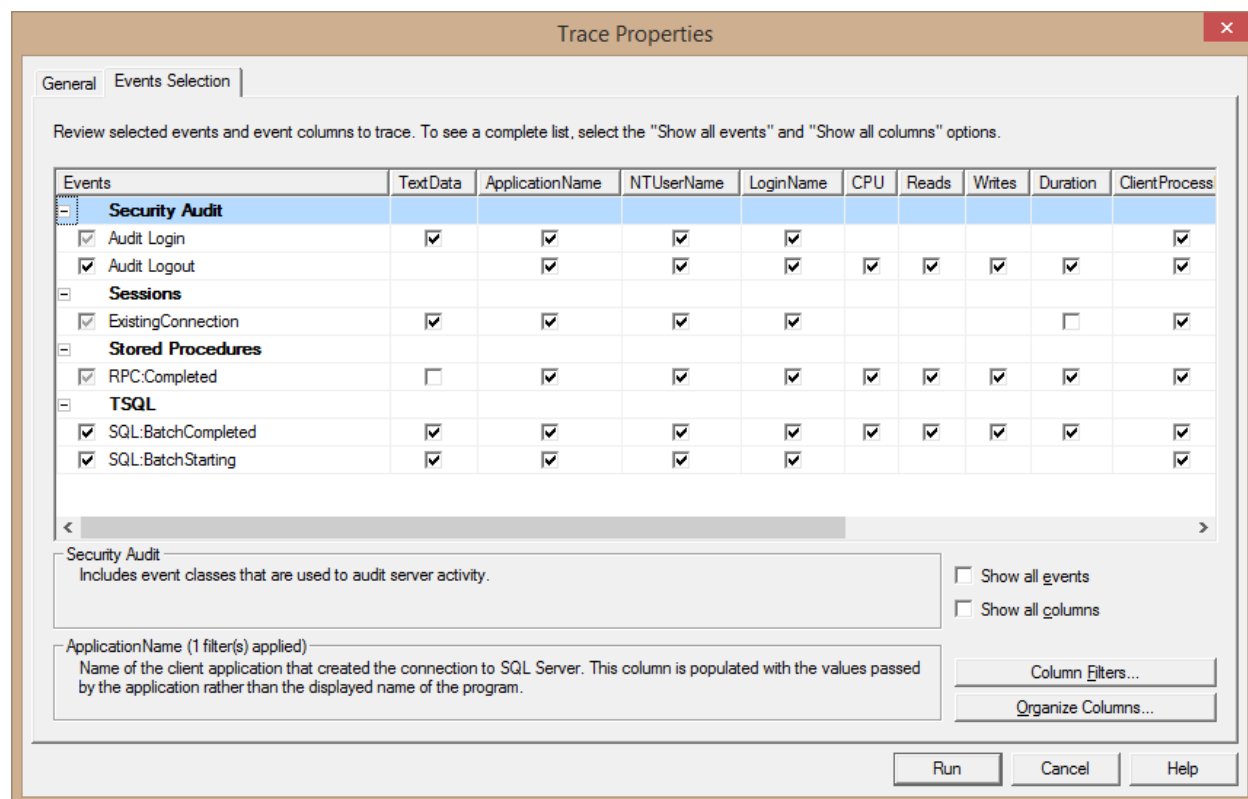


Figure 21: Events Selection tab of Trace

If you now go into SSMS and query a table, or you run the application we created in the previous chapter, you will see the queries in the profiler. And, unfortunately, you may see a whole lot more (I'm also seeing queries from my Report Server and some other stuff I've got running). You can pause the trace and put some filters in place.



Figure 22: Trace Pause and Properties

In the **Events Selection** tab, go to **Column Filters...** and filter on ApplicationName like "EntityFramework". If you haven't manually changed the Application in the connection string of your DbContext, you will now only see queries that are fired from your application. A filter on LoginName and/or NTUserName can also come in handy, especially when you have millions of queries a day and you're only interested in a single user. Another filter I use quite often is Duration Greater than or equal; set it to 5000 (milliseconds) and you'll get all the queries that take at least five seconds. If one query pops up a lot, you may need to have a look at it. Anyway, with your ApplicationName filter in place, run some query from your application and look it up in the profiler. I ran `context.People.Where(p => p.GenderId == 1).ToList();`.

EventClass	TextData	ApplicationName	NTUserName	LoginName	CPU	Reads	Writes	Duration
SQL:BatchCompleted	select cast(serverproperty('EngineEdition') as int)	EntityFramework	SR	Laptop...	0	0	0	0
RPC:Completed	exec sp_reset_connection	EntityFramework	SR	Laptop...	0	0	0	0
SP:StmtCompleted	SELECT StatMan([SCO], [SC1]) FROM (SELECT TOP 100 PERCE...	EntityFramework	SR	Laptop...	0	2	0	0
SP:StmtCompleted	SELECT StatMan([SCO]) FROM (SELECT TOP 100 PERCENT [Id]...	EntityFramework	SR	Laptop...	0	2	0	0
SQL:BatchCompleted	SELECT [Extent1].[Id] AS [Id], [Extent1].[F...	EntityFramework	SR	Laptop...	0	18	1	19


```

SELECT
  [Extent1].[Id] AS [Id],
  [Extent1].[FirstName] AS [FirstName],
  [Extent1].[LastName] AS [LastName],
  [Extent1].[FullName] AS [FullName],
  [Extent1].[Age] AS [Age],
  [Extent1].[DateOfBirth] AS [DateOfBirth],
  [Extent1].[GenderId] AS [GenderId],
  [Extent1].[CompanyId] AS [CompanyId]
FROM [dbo].[Person] AS [Extent1]
WHERE 1 = [Extent1].[GenderId]

```

Figure 23: Query in the profiler

As you can see, EF has some overhead, and only the fifth row is my actual query. By looking at the generated query, I can see `1 = [Extent1].[GenderId]` was not parameterized (as I intended in my query). Looking further at the columns, we can see the database needed 18 reads, a single write, and 19 milliseconds to come up with the result. When looking for troublesome queries, you might want to keep an eye on those values. Higher isn't necessarily bad, but it may be an indication that something can be optimized. Now we can also copy and paste the query and run it in SSMS manually.

The profiler can be used for many things, for example, trace deadlocks (event "Locks -> Deadlock graph"). Just look at all the events and columns and you'll see that we'd need another book to fully understand just this one tool. For my day-to-day activities, the **BatchCompleted** and **StmtCompleted** events, together with the Duration filter, prove to be invaluable.

Sometimes you might just want to run the trace for a while. Luckily, you can set up your trace and then in the **File** menu, go to **Export** and then **Script Trace Definition** and **For SQL Server 2005 - 2014...** Save the script somewhere and open it. Be sure to replace **InsertFileNameHere** (for example, **C:\Traces\MyTrace** will do, but make sure the folder exists) and run the script.

Now it doesn't matter whether the profiler is running or not; everything will be saved to the trace file. Of course you could do this using the Trace Properties, but I wanted to show you the script as well. By the way, if you're having trouble opening the file, try to run the profiler as administrator.

To see which traces are running, you can use the following script.

Code Listing 63: View SQL traces

```
SELECT * FROM fn_trace_getinfo(default)
```

To stop, start, and delete a trace, you can use the following.

Code Listing 64: Stop, start, and delete a trace file

```
SELECT * FROM fn_trace_getinfo(default)
-- The file we created got traceid 3.
-- Yours may be different, check first.
DECLARE @TraceId INT = 3

-- Stop the trace.
EXEC sp_trace_setstatus @TraceId, 0

-- Start the trace.
EXEC sp_trace_setstatus @TraceId, 1

-- To delete the trace, first stop it.
EXEC sp_trace_setstatus @TraceId, 0
EXEC sp_trace_setstatus @TraceId, 2
```

Query plans

Now that you know how to get the queries that may cause trouble, you want to find out why exactly they cause trouble. The most obvious way to do this is by looking at the query plan. Before a query is executed, SQL Server determines the approximate quickest way to fetch all the data. I say approximate because SQL Server knows how to do one thing in different ways, so for example, a **JOIN** can be realized internally by a **LOOP**, **MERGE**, or **HASH JOIN**. Take two **JOINS** in a single query and we can have $3 * 3 = 9$ different ways to execute this query. Add a **WHERE** clause, and things grow exponentially. A single query can have hundreds or even thousands of different plans. SQL Server takes an educated guess as to what plan might be optimal. We've already talked a bit about this a bit in the parameterization chapter, but now we're going to look at some actual plans.

Unfortunately, Syncfusion doesn't have a *Succinctly* e-book on query plans (yet?), but you could always check out this [free e-book](#) on query plans by SimpleTalk.

For now, open SSMS, browse to the **AdventureWorks** database, and open a new query window. Right-click on the empty query window, and in the context menu, check **Include Actual**

Execution Plan. For some extra information, run the following statements in your query window.

Code Listing 74: Enabling statistics

```
SET STATISTICS IO ON
SET STATISTICS TIME ON
```

These statistics will be shown in your Messages tab after you run a query. You can delete the statements from your window; they will be active in this window until you close it or until you explicitly turn it off again (**SET STATISTICS X OFF**). Now run the following query (in the same query window).

Code Listing 75: A SQL query with JOIN

```
SELECT p.FirstName + ' ' + p.LastName AS FullName, *
FROM Person.Person p
JOIN Person.EmailAddress e ON
    e.BusinessEntityID = p.BusinessEntityID
```

You now get the Results, Messages, and an additional tab, the Execution plan.

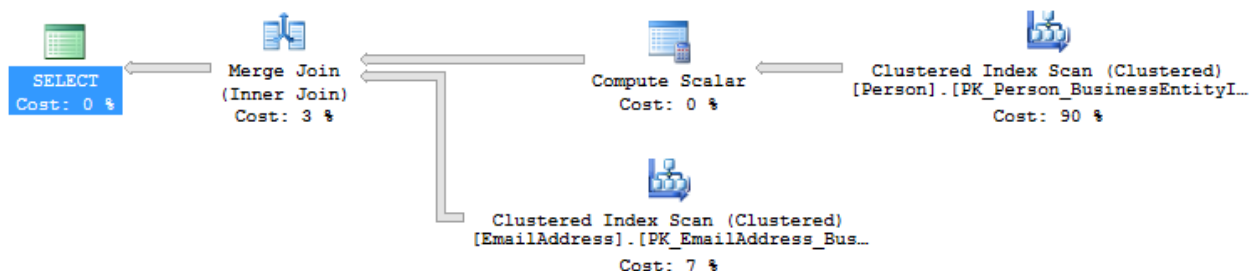


Figure 24: An execution plan

You should read the execution plan from right to left (the **SELECT** is executed last). The thicker an arrow between two nodes, the more data that was handled by that node. SQL Server starts by doing a scan on the **Person** table (using the primary key). It then computes the scalar **FullName**. It does another scan on **EmailAddress** using the primary key for that table. It then does a merge join between **Person** and **EmailAddress**. Finally, it selects the data. To interpret this, you must have a little knowledge on the inner workings of SQL Server, and quite possibly, algorithms. Knowing the alternatives is a must. So let's dissect this plan and check out the alternatives.

We first start with an Index Scan. The alternative is an Index Seek. An Index Scan simply reads through the rows in your input (in this case an index table) and takes out the rows that match the queries criteria. In this case we want all rows, so we can't really get faster than just fetching all rows like we do now. Let's say we expect only one row to be returned because we filter on an ID—surely an entire table scan would be a waste of resources. That's when SQL Server uses an Index Seek, which only goes through pages and rows that contain qualifying data. Somewhere in between all rows and none or one row is a turning point where one of the two is faster than the alternative.

The **MERGE JOIN** is the fastest join around. It only works when two sets of data are ordered in the same way, and when two rows have to match on a key. In this case, both tables are ordered on **BusinessEntityID**. **MERGE JOIN** reads a row from one input and compares it with a row from the other input. If the rows don't match, the row with the smaller value can be discarded and the next row is read until all rows have been read. The maximum number of reads necessary for this operation is the number of rows in the first input, plus the number of rows in the second input.

The alternatives here are **LOOP JOIN** and **HASH JOIN**. **LOOP JOIN** loops through all rows in the first input, and for each row, it loops through all rows of the second input until a match is found. This is less efficient than a **MERGE JOIN**, as the maximum number of reads is now the number of rows in input one times the number of rows in input two.

The **HASH JOIN** can require the most processing behind the scenes, and if you encounter it, you probably should check your design. The **HASH JOIN** algorithm creates a hash table for the smaller input of the join (and possibly writes this to disk if short on memory), and then compares them to the other values using a hash match function. For small inputs this can be very efficient, but if you have bigger inputs, this might slow down your query considerably.

As we can see, this query is optimal; we have the optimal search algorithm and the optimal **JOIN** algorithm. The next query is far from optimal, and since the data set is relatively small, even though it's not giving us any troubles, we're going to optimize it for the sake of practice.

Code Listing 65: A query with a suboptimal plan

```
SELECT FirstName, MiddleName, LastName, Title
FROM Person.Person
WHERE MiddleName > 'Z'
OPTION (RECOMPILE)
```

I have added the **OPTION (RECOMPILE)** so that we're sure SQL Server always generates a new plan for this query (so we're not looking at cached plans). Here is the generated plan.

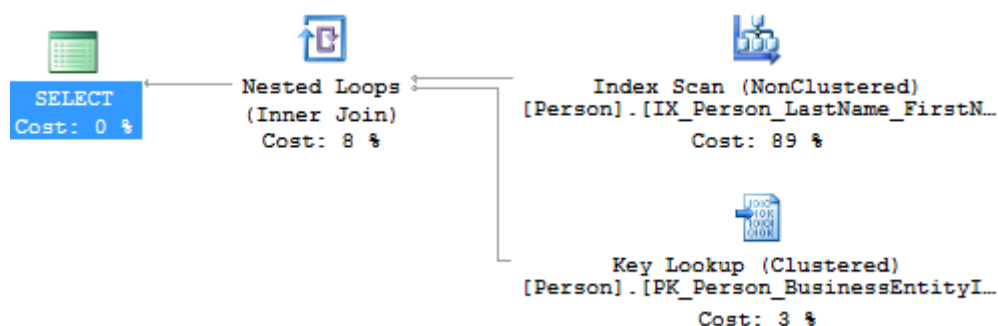


Figure 25: A suboptimal query plan

Here we see that we're doing an Index Scan, but when we execute the query, we only get one result. So we're scanning the entire table (almost 20,000 rows) for just one row. Also notice the Key Lookup. This happens when an index is used that does not have all the requested values. In this case, the Index Scan is done on **IX_Person_LastName_FirstName_MiddleName**, but we're requesting more fields than just those three because we also want **Title**. The Key

Lookup searches for the row index and retrieves all missing fields. Of course, SQL Server needs to join the results of the Index Scan and the Key Lookup, which costs even more time and resources.



Tip: *The problem with slowly executing queries is that sometimes they run very slowly (like maybe an hour or more). In such cases, “Include Actual Execution Plan” is not an option, as it requires you to run the query, but you can’t wait for hours for it to finish, taking up resources and possibly blocking other queries. For such cases, SQL Server has an option in the context menu, “Display Estimated Execution Plan.” This option does not run the query and only creates the estimated plan. It may not give you the actual plan, but more often than not it’s pretty accurate, and you can at least inspect the plan and identify possible bottlenecks.*

Indexing

So we have this query that uses an index, but still needs to look up the master row in the primary key index. That happens when an index is not *covering*. When all requested fields from a table are contained in a single index, this index is called a covering index. In this case, **Title** is missing from the index. So let’s add **Title** to the index. We can do two things: add **Title** to the actual index, or include it in the index.

To understand the difference, we must understand that an index is a virtual sorting of one or more fields. **IX_Person_LastName_FirstName_MiddleName** is sorted on **LastName**, then **FirstName**, and then **MiddleName**. That means SQL Server can use the [Binary Search algorithm](#) to quickly find all rows with a specific **LastName**, then for all people with that **LastName**, it can quickly find all people with a specific **FirstName**. Finally, for people with that **LastName** and **FirstName**, it can find all people with a specific **MiddleName**.

If we added **Title** to the index, we could find people with a specific name and specific **Title** as well. Unfortunately, indexing comes with a price. Every time one of the fields is updated, the index must be updated as well. Adding **Title** to the index means the index must be updated when any **LastName**, **FirstName**, **MiddleName**, or **Title** is updated. So inserts, updates, and deletes on this table will now take the performance hit. We’re probably never going to query for such a specific name with a specific title, so we’d rather not do that.

Luckily we have a second choice: include **Title** as an included field. It’s not included in the sorting, but it is included in the index, meaning that whenever **Title** is updated, the index doesn’t have to update. The only cost we have then is that of some extra memory usage, a fair price to pay if it speeds up our query. To include a field in an index, find the index in the Object Explorer, right-click, choose **Properties**, and add the field on the **Included columns** tab (and NOT the “Index key columns” tab). Now let’s execute the query again.

Changing the index worked, since our plan no longer requires a Key Lookup.

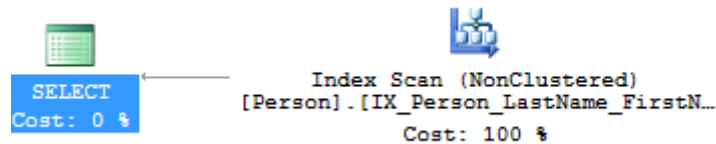


Figure 26: Key Lookup eliminated

Unfortunately, we still have an Index Scan instead of an Index Seek. Apparently our current indexes just don't cut it. You might as well remove the **Title** from the included columns now, as we're going to create an index that's completely optimized for this query. Simply add an index and put **MiddleName** in the Index key columns, and **FirstName**, **LastName**, and **Title** in the Included columns.

Code Listing 66: Index on MiddleName

```
CREATE NONCLUSTERED INDEX IX_Person_MiddleName
ON Person.Person
(
    MiddleName
)
INCLUDE (FirstName, LastName, Title)
```

If we run the query again, we will see that we have eliminated the Index Scan as well, and the query is now optimal.

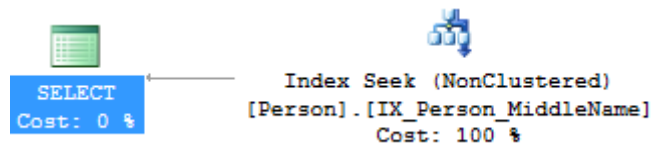


Figure 27: Optimal query plan

In some cases, SQL Server may suggest indexes for you. For example, run the following query and inspect the query plan.

Code Listing 67: Non-indexed query

```
SELECT *
FROM Sales.SalesOrderHeader
WHERE OrderDate = '20120101'
```

Obviously it would be nice if we had an index on **OrderDate**. The **SalesOrderHeader** table has over 30,000 rows, and currently **OrderDate** has no index, and an index scan is performed. A specific order date never returns huge result sets, though, so scanning the entire index is never optimal for this query. Luckily, SQL Server sees that, too, and gives you a hint. You can right-click it and select **Missing index details**, and it will open a new query window with the index script. Be sure to take a good look at it, as it will not always be a desired index! For example, a **SELECT *** may result in an index having all columns that are not part of the ordering to be included in the index. In such cases, it's probably better to not include any columns at all, and just stick to the index columns.

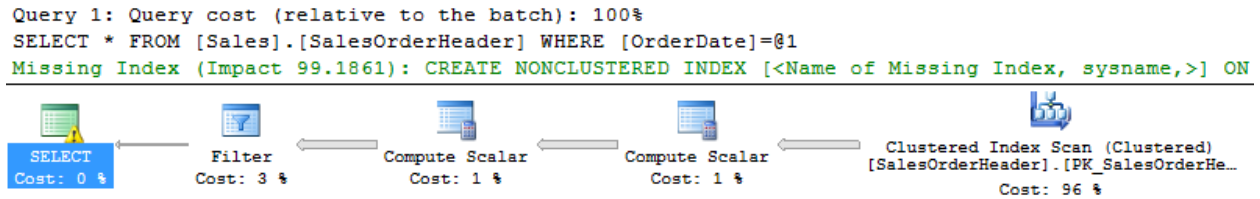


Figure 28: A missing index hint

Now, you may think it's a good idea to optimize every query, but it's not. Imagine a query taking five minutes to complete—surely that must be optimized, right? What if the query runs only once a month? Suddenly those five minutes are not so important anymore. Optimizing it may take hours of work that you'll have back after years of running the query. If the five minutes is a daily routine and users are waiting for it, by all means, optimize! Even a query that takes an hour can be fine if it is, for example, an automated job that runs at night. Optimizing it might mean that other processes that run constantly during production may incur a performance hit. Finally, as we will see in the next section, an optimization may not always be optimal for all cases.

Parameter sniffing

Sometimes your query runs really slow when executed from your C# code, while other times it completes in a few milliseconds. Multiple things can happen here. First, and we'll get to this, is blocking queries. Another option is parameter sniffing. What SQL Server does when it encounters a new query that uses parameters is that it creates a query plan that is (approximately) optimal for the current value of the parameter. Remember that query plans are cached and reused if the same query is executed again. However, if the same parameterized query now has a different value for its parameter, the plan may now be suboptimal. Luckily we can replicate this issue quite easily (although we won't see the performance hit, as our database is still too small).

First we need some parameterized query. An ad hoc query in a query window is not parameterized unless you use `sp_executesql`, an in-built procedure that runs a string as SQL and can optionally take parameters. This is what C# sends to the database if you run a parameterized query. Running a string isn't very easy to type, though (as you have to escape it yourself), so let's go with the alternative, a stored procedure. Create the following procedure in the **AdventureWorks** database (never mind that this procedure is pretty useless and becomes literally unusable after some million orders).

Code Listing 68: SP GetOrdersByDate

```

CREATE PROCEDURE GetOrdersByDate
    @OrderDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
    FROM Sales.SalesOrderHeader
    WHERE OrderDate < @OrderDate

```

END

You have to know one thing about SQL Server when dealing with plans. SQL Server keeps all kinds of statistics internally. When you're going to execute this procedure, SQL Server will inspect **@OrderDate** and make some assumptions based on its statistics. That makes things a little easier for us, too. At this point we know there is no index on **OrderDate**. As a result, with so many rows, SQL Server will probably just scan the table for each value we assign to **@OrderDate**. We can simply test this by executing this query with a few values and inspecting the plan.

Code Listing 80: Multiple plans for the same query

```
DBCC FREEPROCCACHE
EXECUTE GetOrdersByDate @OrderDate = '20100101'
DBCC FREEPROCCACHE
EXECUTE GetOrdersByDate @OrderDate = '20120101'
DBCC FREEPROCCACHE
EXECUTE GetOrdersByDate @OrderDate = '20150101'
```

DBCC FREEPROCCACHE clears all query plans from the cache, so we know SQL Server isn't reusing any plans. We get the same plan for each parameter.

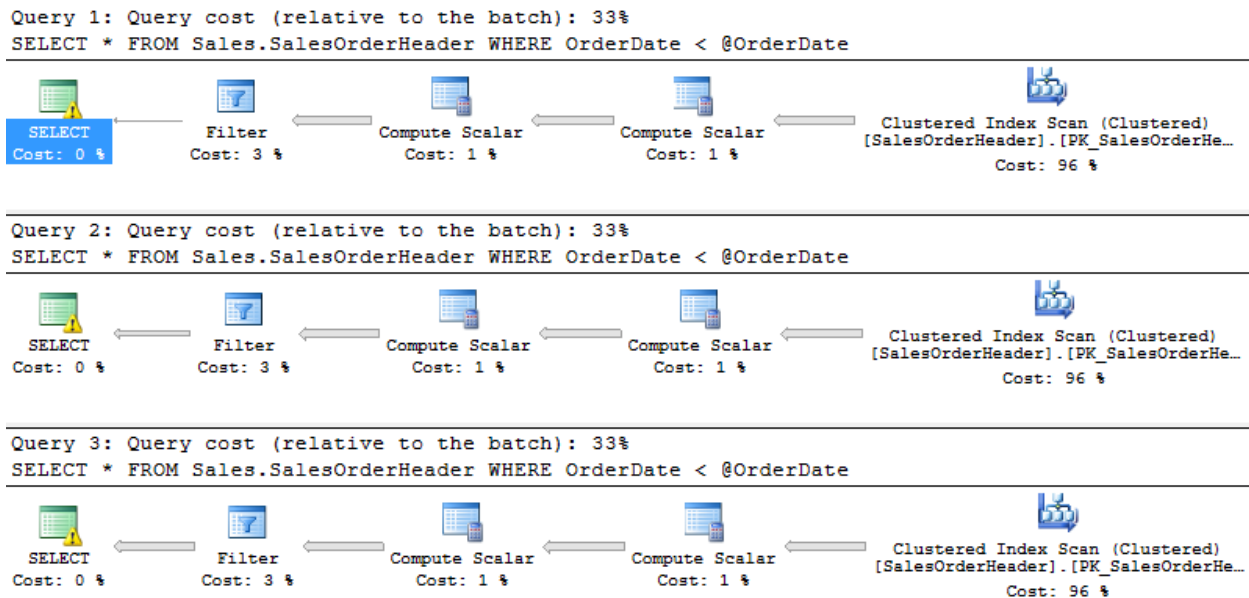


Figure 29: Equal plans

That isn't helping our example very much. It also seems, since **@OrderDate = '20100101'** returns 0 rows, that a table scan isn't optimal. So let's make SQL Server know this, too. Let's put an index on **OrderDate**.

Code Listing 81: Index on OrderDate

```
CREATE NONCLUSTERED INDEX IX_SalesOrderHeader_OrderDate
ON Sales.SalesOrderHeader
```

```
(  
    [OrderDate] ASC  
)
```

With this index in place, run the previous query again, with all the **DBCC FREEPROCCACHE**, and surely the first query plan will look a little different! It now does an Index Seek on **IX_SalesOrderHeader_OrderDate**, which is a lot better for this particular query.

Unfortunately, we can't go around and execute **DBCC FREEPROCCACHE** before every query. In fact, *never run DBCC FREEPROCCACHE* on a production server unless you know what you're doing. Run the following query, check the query plans, and you will see the effects of parameter sniffing.

Code Listing 69: Parameter sniffing

```
DBCC FREEPROCCACHE  
EXECUTE GetOrdersByDate @OrderDate = '20100101'  
EXECUTE GetOrdersByDate @OrderDate = '20120101'  
EXECUTE GetOrdersByDate @OrderDate = '20150101'
```

The query plans are now equal again for each execution. Unfortunately, this is far from optimal for order dates 2012 and 2015. At this point you may be wondering, "Why did it use such suboptimal plans?" or "Why is my query so slow?" When you hover over a node in the query plan, you get to see all kinds of statistics that may be helpful in figuring out what went wrong.





 Index Seek (NonClustered) [SalesOrderHeader].[IX_SalesOrderDate] Cost: 50 %	<div> <div>Index Seek (NonClustered)</div> <div>Scan a particular range of rows from a nonclustered index.</div> </div> <table> <tr><td>Physical Operation</td><td>Index Seek</td></tr> <tr><td>Logical Operation</td><td>Index Seek</td></tr> <tr><td>Actual Execution Mode</td><td>Row</td></tr> <tr><td>Estimated Execution Mode</td><td>Row</td></tr> <tr><td>Storage</td><td>RowStore</td></tr> <tr><td>Actual Number of Rows</td><td>31465</td></tr> <tr><td>Actual Number of Batches</td><td>0</td></tr> <tr><td>Estimated Operator Cost</td><td>0,0032831 (50%)</td></tr> <tr><td>Estimated I/O Cost</td><td>0,003125</td></tr> <tr><td>Estimated CPU Cost</td><td>0,0001581</td></tr> <tr><td>Estimated Subtree Cost</td><td>0,0032831</td></tr> <tr><td>Number of Executions</td><td>1</td></tr> <tr><td>Estimated Number of Executions</td><td>1</td></tr> <tr><td>Estimated Number of Rows</td><td>1</td></tr> <tr><td>Estimated Row Size</td><td>19 B</td></tr> <tr><td>Actual Rebinds</td><td>0</td></tr> <tr><td>Actual Rewinds</td><td>0</td></tr> <tr><td>Ordered</td><td>True</td></tr> <tr><td>Node ID</td><td>3</td></tr> </table> <div> <div>Object</div> <div>[AdventureWorks2014].[Sales].[SalesOrderHeader]. [IX_SalesOrderHeader_OrderDate]</div> </div> <div> <div>Output List</div> <div>[AdventureWorks2014].[Sales]. [SalesOrderHeader].SalesOrderID; [AdventureWorks2014].[Sales]. [SalesOrderHeader].OrderDate</div> </div> <div> <div>Seek Predicates</div> <div>Seek Keys[1]: End: [AdventureWorks2014].[Sales]. [SalesOrderHeader].OrderDate < Scalar Operator ([@OrderDate])</div> </div>	Physical Operation	Index Seek	Logical Operation	Index Seek	Actual Execution Mode	Row	Estimated Execution Mode	Row	Storage	RowStore	Actual Number of Rows	31465	Actual Number of Batches	0	Estimated Operator Cost	0,0032831 (50%)	Estimated I/O Cost	0,003125	Estimated CPU Cost	0,0001581	Estimated Subtree Cost	0,0032831	Number of Executions	1	Estimated Number of Executions	1	Estimated Number of Rows	1	Estimated Row Size	19 B	Actual Rebinds	0	Actual Rewinds	0	Ordered	True	Node ID	3
Physical Operation	Index Seek																																						
Logical Operation	Index Seek																																						
Actual Execution Mode	Row																																						
Estimated Execution Mode	Row																																						
Storage	RowStore																																						
Actual Number of Rows	31465																																						
Actual Number of Batches	0																																						
Estimated Operator Cost	0,0032831 (50%)																																						
Estimated I/O Cost	0,003125																																						
Estimated CPU Cost	0,0001581																																						
Estimated Subtree Cost	0,0032831																																						
Number of Executions	1																																						
Estimated Number of Executions	1																																						
Estimated Number of Rows	1																																						
Estimated Row Size	19 B																																						
Actual Rebinds	0																																						
Actual Rewinds	0																																						
Ordered	True																																						
Node ID	3																																						
 Key Lookup (Clustered) [SalesOrderHeader].[PK_SalesOrderID] Cost: 50 %																																							
 Index Seek (NonClustered) [SalesOrderHeader].[IX_SalesOrderDate] Cost: 50 %																																							
 Key Lookup (Clustered) [SalesOrderHeader].[PK_SalesOrderID] Cost: 50 %																																							

Figure 30: Plan step statistics

The really interesting values here are the **Estimated Number of Rows** and the **Actual Number of Rows**. When SQL Server expects no rows to be returned, the estimated number will be 1, and SQL Server will optimize for that. However, the actual number of rows returned was 31,465. That's a huge difference, and it's an indicator that you're probably missing an index, your statistics are out-of-sync, or you've become victim of parameter sniffing.

Parameter sniffing isn't a bad thing by itself, but it has its quirks. There are a few things we can do to solve parameter sniffing. First, when you're pretty sure each plan will be different, you can recompile your queries. Recompilation makes sure that a new plan is created every time the query is executed. You will get an optimal plan for order dates 2010, 2012, and 2015, but every execution will incur a performance hit for the addition plan creation. You can recompile single queries or entire procedures.

Code Listing 70: Recompiling queries and procedures

```
ALTER PROCEDURE GetOrdersByDate
    @OrderDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
    FROM Sales.SalesOrderHeader
    WHERE OrderDate < @OrderDate
    OPTION (RECOMPILE)
END
GO
-- OR
ALTER PROCEDURE GetOrdersByDate
    @OrderDate DATETIME
WITH RECOMPILE
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
    FROM Sales.SalesOrderHeader
    WHERE OrderDate < @OrderDate
END
```

Another option is to tell SQL Server to optimize for a specific value. In this case, we're pretty sure this procedure will be called with an order date that's pretty close to today, so 99 out of 100 times we expect to fetch thousands of rows.

Code Listing 71: Optimize for value

```
ALTER PROCEDURE GetOrdersByDate
    @OrderDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
    FROM Sales.SalesOrderHeader
    WHERE OrderDate < @OrderDate
    OPTION (OPTIMIZE FOR (@OrderDate = '20150101'))
END
```

Order date 2010 will have a suboptimal query, but 2012 and 2015 run fine and don't incur the performance penalty for creating the plan. In case you're not sure about an ideal value, you may optimize for unknown. SQL Server will generate a plan based on some statistical average of your data distribution. The downside, of course, is that this optimization is unpredictable and may still result in parameter sniffing issues.

Code Listing 72: Optimize for unknown

```
OPTION (OPTIMIZE FOR UNKNOWN)
```

In some cases, SQL Server makes a wrong call considering the index to use. Or maybe it's not wrong, just not what you'd like it to be. A while ago I found myself in such a situation. A coworker had created a rather lengthy stored procedure and the execution wasn't all that it was supposed to be. Sure enough, we were missing an index. We added the index and cleared the query cache, but nothing happened. A quick look at the execution plan revealed that SQL Server wasn't using our index! In this instance, we just wanted it to use our index.

Code Listing 73: Forcing an index

```
ALTER PROCEDURE [dbo].[GetOrdersByDate]
    @OrderDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    SELECT *
    FROM Sales.SalesOrderHeader
        WITH (INDEX (IX_SalesOrderHeader_CustomerID))
    WHERE OrderDate < @OrderDate
END
```

Surely that index makes no sense! The fun part is that if you run it now and look at the query plan, SQL Server will give you a hint that you're missing an index on **OrderDate** (with all other fields included, which we don't want).

Yet another option is to make a local variable from your input parameter. The idea is that SQL Server optimizes queries for a given value: the one that is passed to the stored procedure. However, when you use a local variable, SQL Server cannot know the value of that variable in the query plan building stage. As a result, using a local variable will create the same output as **OPTION (OPTIMIZE FOR UNKNOWN)**.

Code Listing 74: Using a local variable

```
ALTER PROCEDURE [dbo].[GetOrdersByDate]
    @OrderDate DATETIME
AS
BEGIN
    SET NOCOUNT ON;

    DECLARE @LocalOrderDate DATETIME = @OrderDate

    SELECT *
    FROM Sales.SalesOrderHeader
    WHERE OrderDate < @LocalOrderDate
END
```


As you can see, there are some solutions to parameter sniffing when working in SQL Server. You can apply the same methods in C# when working with ADO.NET. You're building the query, so you can also include hints like **OPTION (RECOMPILE)**, **OPTION (OPTIMIZE FOR ...)**, or **WITH (INDEX(...))**. Unfortunately, when using Entity Framework, you're not building your own queries. But you have a few options. You can use stored procedures for troublesome queries (in fact, some people *a/ways* use stored procedures for everything, but that's another discussion). You can also use plain ADO.NET for troublesome queries. If you're stuck with EF, you can use EF Interceptors (EF6 and onwards only), as discussed in the next chapter.

Chapter 7 Interception, Locking, Dynamic Management Views

In this chapter we'll look at three relatively advanced topics: customizing default behaviors by intercepting SQL execution from C# code, the issues around SQL locking, and performance tuning using Dynamic Management Views.

Interception

SQL interception using C# is not easy, and is probably best avoided. It does give you some low-level control over the generated SQL and additional actions EF should take when it's executing, or has executed, queries. Interception is based around a couple of interfaces, most notably **System.Data.Entity.Infrastructure.Interception.IDbInterceptor**.

There are different interceptors for different use cases, such as executing queries (**IDbCommandInterceptor**), working with transactions (**IDbTransactionInterceptor**), and connections (**IDbConnectionInterceptor**). For our case, we need the **IDbCommandInterceptor**, but we derive from the base class **DbCommandInterceptor** (which implements **IDbCommandInterceptor**).

As you can see, each method (**SomethingExecuting** or **SomethingExecuted**) gets a **DbCommand**, which can be manipulated, and a **DbCommandInterceptionContext**. Now we can simply concatenate " **OPTION (RECOMPILE)**" to any **SELECT** query.

Code Listing 75: RecompileInterceptor

```
public class RecompileInterceptor : DbCommandInterceptor
{
    public override void ReaderExecuting(DbCommand command,
    DbCommandInterceptionContext<DbDataReader> interceptionContext)
    {
        base.ReaderExecuting(command, interceptionContext);
        AddOptionRecompile(command, interceptionContext);
    }

    public override void NonQueryExecuting(DbCommand command,
    DbCommandInterceptionContext<int> interceptionContext)
    {
        base.NonQueryExecuting(command, interceptionContext);
        AddOptionRecompile(command, interceptionContext);
    }

    public override void ScalarExecuting(DbCommand command,
    DbCommandInterceptionContext<object> interceptionContext)
    {

```

```

        base.ScalarExecuting(command, interceptionContext);
        AddOptionRecompile(command, interceptionContext);
    }

    private void AddOptionRecompile<T>(DbCommand command,
    DbCommandInterceptionContext<T> interceptionContext)
    {
        // DbContexts.Any indicates the query is not EF overhead.
        if (interceptionContext.DbContexts.Any() &&
            command.CommandText.StartsWith("SELECT",
            StringComparison.OrdinalIgnoreCase))
        {
            command.CommandText += " OPTION(RECOMPILE)";
        }
    }
}

```

To use an **IDbInterceptor**, we can use the **DbInterception** class. Unfortunately, this class registers **IDbInterceptors** globally, so once registered, an interceptor will be used for every query in your entire application (including other threads). You can, of course, also unregister them.

Code Listing 76: Add an interceptor

```

RecompileInterceptor interceptor = new RecompileInterceptor();
DbInterception.Add(interceptor);
using (CodeFirstContext context = new CodeFirstContext())
{
    context.Database.Log = Console.Write;
    context.People.Where(p => p.GenderId == 1).ToList();
}
DbInterception.Remove(interceptor);
Console.ReadKey();

```

Interceptors are powerful tools. They can be used for logging, additional security, canceling queries, or altering queries, among other things.

Locking and deadlocks

Another issue you may be facing when queries run slowly is locking. I've been in situations where queries such as **SELECT Field FROM Table WHERE Id = X** were running very slowly. We know now that such queries should be very fast, and that parameter sniffing shouldn't be a problem since the plan should be the same for every value of X. Chances are this query is simply not running because some other query is blocking access to the specified table. We can easily simulate such scenarios. First of all, I want to remind you that it's possible to run only parts of a script in a single query window. Simply select the part you want to execute, and execute it.

Let's create a blocking query. Open up a query window and execute the following.

Code Listing 90: A blocking query

```
BEGIN TRAN  
  
UPDATE Production.Product  
SET Name = 'Adjustable Racer'  
WHERE ProductID = 1
```

SQL Server locks the row for **ProductID 1**. Different kinds of locks exist, and SQL Server will determine the best locking strategy, depending on your query. For example, not specifying a **WHERE** clause would probably end up in a table lock. While a certain object is locked, rows, pages, tables, and other objects can't usually access it. Usually a lock is lifted when a transaction completes, but in this case we started a transaction and never committed or rolled back. So the transaction will just stay open and lock this row. Locking is not a bad thing, though; it prevents other processes from querying data that is not committed yet, preventing dirty reads.

Open up a new window and execute the following query.

Code Listing 91: A blocked query

```
SELECT *  
FROM Production.Product  
WHERE ProductID = 1
```

You will notice this query will start executing and simply never complete. It's actually waiting for our first query to complete.

There are a few fixes here. First of all, we should examine why our first query is taking so long to complete and see if we can rewrite it so it finishes sooner. Second, we could set our **TRANSACTION ISOLATION LEVEL** to **(READ COMMITTED) SNAPSHOT**. This means that whenever SQL Server locks a resource, it makes a copy of that resource and then uses the copy when selecting data, so your **SELECT** queries aren't blocked, but still return committed data.

SNAPSHOT ISOLATION is not in the scope of this book, but a quick Google search will no doubt give you a lot of resources on the subject. The solution I'm going to show you here is to set the **ISOLATION LEVEL** of the **SELECT** query to **READ UNCOMMITTED** so that it will read uncommitted data. In many cases this is a fine solution, since the chances of dirty reads (reading uncommitted data) are very small, and dirty reads may even be acceptable. If your **SELECT** query is still running, you can stop it by hitting the **Cancel Executing Query** button or with Alt + Break. Alternatively, you could return to the **UPDATE** query window and there execute **ROLLBACK TRAN**, causing the transaction to end and the **SELECT** query to finish. If you decide to end your transaction, be sure to start it again for the following example.

So, assuming you have a lock on **ProductID 1**, you can tell a **SELECT** query to do dirty reads by giving it the **NOLOCK** hint.

Code Listing 77: Dirty reads WITH (NOLOCK)

```
SELECT *  
FROM Production.Product WITH (NOLOCK)  
WHERE ProductID = 1
```

If you have multiple statements and would like to execute them all using dirty reads, you can set your **TRANSACTION ISOLATION LEVEL** instead.

Code Listing 78: Dirty reads with READ UNCOMMITTED

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

SELECT *
FROM Production.Product
WHERE ProductID = 1
```

Both queries will return instantly, even when your other transaction is still running. However, both queries will return **Name Adjustable Racers**, which isn't yet committed. Now, rollback your **UPDATE** query, and no one will ever be able to get **Adjustable Racers** ever again.

A connection string usually has a timeout defined (the default is 30 seconds). This means that when a query is started from C# and takes over 30 seconds to complete, or is blocked for at least 30 seconds, you'll get a **TimeoutException**.

When one transaction is waiting for another transaction, and then that other transaction requests a resource that is locked by the first transaction, you're having a deadlock. The concept of deadlocks should be familiar to programmers, as multithreaded applications can run into the same problems.

Let's fake a deadlock. Open up a query window and execute the following query.

Code Listing 79: A locking query

```
BEGIN TRAN

UPDATE Production.Product
SET Name = 'Adjustable Racer'
WHERE ProductID = 1
```

Now open up another query window, make it wait for the open transaction, and lock another resource.

Code Listing 80: A locking and waiting query

```
BEGIN TRAN

UPDATE Production.Location
SET Name = 'Tool Cribs'
WHERE LocationID = 1

SELECT *
FROM Production.Product
WHERE ProductID = 1
```

Go back to the first window, where the transaction is still open, and request **LocationID 1**, which is locked.

Code Listing 81: Deadlock ensues

```
SELECT *  
FROM Production.Location  
WHERE LocationID = 1
```

One of the two query windows should now return an error: Msg 1205, Level 13, State 51, Line 7

Transaction (Process ID 54) was deadlocked on lock resources with another process and has been chosen as the deadlock victim. Rerun the transaction.

The transaction of that query is automatically rolled back. In the other window, you can manually execute **ROLLBACK TRAN**.

You can track locks and deadlocks using the SQL Server Profiler. Open up a new session, unselect all events, and select the **Deadlock graph** event under **Locks**. If you repeat the previous steps, you will get an event and a graph. It should look something like as follows.

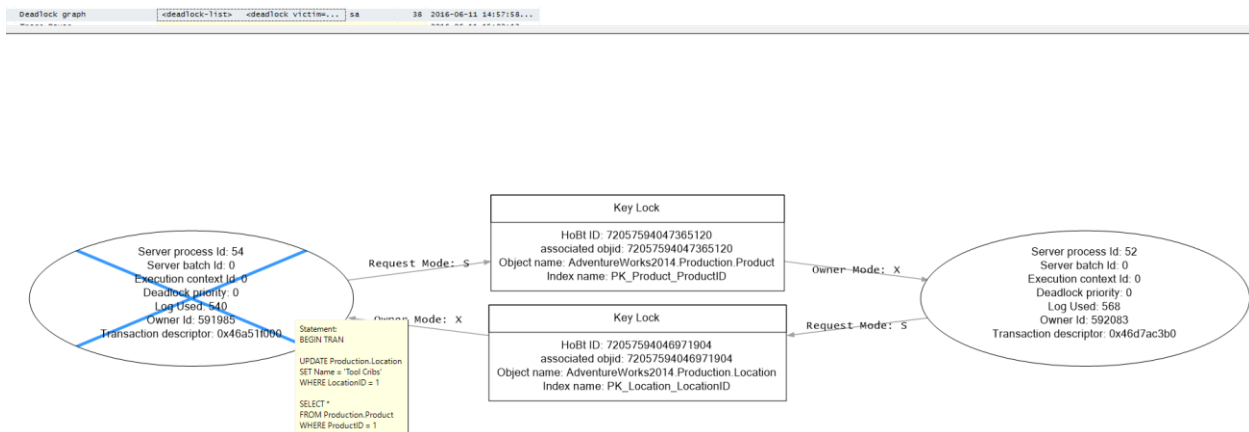


Figure 31: A deadlock graph

The ovals are the queries involved in the deadlock, and the rectangles are the objects that were locked and requested. Deadlock graphs can get very complicated, having tens of ovals and rectangles.

We have already seen two methods of fixing locks that will also fix deadlocks: using **SNAPSHOT ISOLATION**, and lowering your **ISOLATION LEVEL** to **READ UNCOMMITTED**. Another method that will prevent deadlocks is to always access database objects in the same order. Unfortunately, that's not always easy or even possible. Try keeping transactions short so the chances of (dead) locking are minimized.

Dynamic management views

Knowing a little more on how to get a grip on queries, plans, and other data may be very useful. SQL Server has what are called dynamic management views, or DMVs. The structure of your database, tables, procedures, and practically everything in your database is stored in these views. For example, the following query lists all tables in your database (twice).

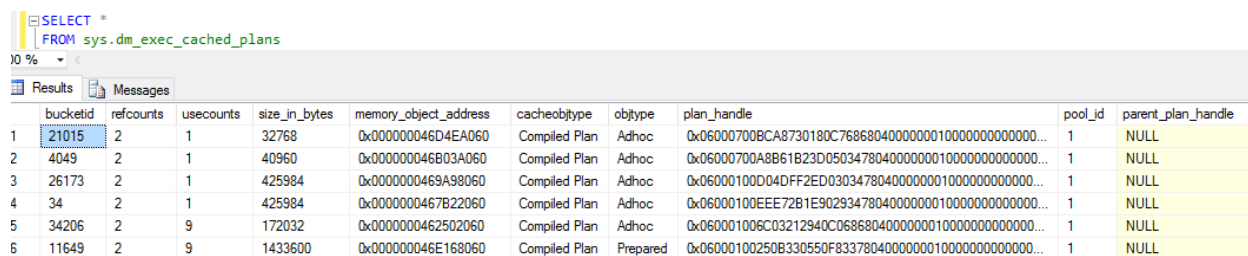
Code Listing 82: List all tables

```
SELECT * FROM sys.tables
SELECT * FROM INFORMATION_SCHEMA.TABLES
```

You can find the views in the Object Explorer under **Views** and then **System Views**. You'll find views called **sys.something** and a few **INFORMATION_SCHEMA.SOMETHING** views. The **INFORMATION_SCHEMA** views are ISO standards, and using them in your queries should, in theory, make them work in Oracle, MySQL, or any other SQL database that adheres to ISO standards. There are about twenty **INFORMATION_SCHEMA** DMVs, and (I'm guessing) well over a hundred **sys** DMVs, so you just go with the **sys** views, as they have all you need.

Next to the DMVs, there are system SPs and functions, as well. You can find them under **Programmability > Stored Procedures > System Stored Procedures > Programmability > Functions > System Functions**. You can simply scroll through them and try them out. You'll probably find you've been using many functions already, such as **CAST**, **MAX**, and **CURRENT_TIMESTAMP**.

Let's move on to the good stuff. We were talking about query plans. There is a view called **sys.dm_exec_cached_plans**, which has a list of all cached plans. The data in it is a little cryptic, though. In its current form, it's of no use to us.



	bucketid	refcounts	usecounts	size_in_bytes	memory_object_address	cacheobjtype	objtype	plan_handle	pool_id	parent_plan_handle
1	21015	2	1	32768	0x000000046D4EA060	Compiled Plan	Adhoc	0x06000700BCA8730180C76868040000000100000000000000...	1	NULL
2	4049	2	1	40960	0x0000000046B03A060	Compiled Plan	Adhoc	0x06000700A8B61B23D0503478040000000100000000000000...	1	NULL
3	26173	2	1	425984	0x00000000469A98060	Compiled Plan	Adhoc	0x06000100D04DFF2ED030347804000000010000000000000...	1	NULL
4	34	2	1	425984	0x00000000467B22060	Compiled Plan	Adhoc	0x06000100EEE72B1E90293478040000000100000000000000...	1	NULL
5	34206	2	9	172032	0x00000000462502060	Compiled Plan	Adhoc	0x060001006C0321294DC06868040000000100000000000000...	1	NULL
6	11649	2	9	1433600	0x0000000046E168060	Compiled Plan	Prepared	0x06000100250B330550F83378040000000100000000000000...	1	NULL

Figure 32: sys.dm_exec_cached_plans

The **plan_handle** is very useful, however. We will need this once we know which plan and query it actually belongs to. We can get the actual plan with the function **sys.dm_exec_query_plan**, which takes a **plan_handle** as parameter. For some reason this also returns the database ID, which is nice if we want to filter on database (which we do, as it also has plans of the master database).

Now that we have the **plan_handle** and the **query_plan**, we want the actual query. This is another function, called **sys.dm_exec_sql_text**, which also has a **plan_handle** as input parameter. We now also have the **text** and, again, database ID. We can get the database ID from a database name using **DB_ID(database_name)**.

Code Listing 83: List query plans

```
SELECT cp.plan_handle, qp.query_plan, st.[text]
FROM sys.dm_exec_cached_plans cp
CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) AS qp
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) AS st
WHERE qp.[dbid] = DB_ID('AdventureWorks2014')
```

Now here is a most peculiar thing: open up a new query window and run **DBCC FREEPROCCACHE**. Remove it and, in the same window, run **SELECT * FROM Person.Person** (or whatever you like). Empty the query window again and run **SET ARITHABORT OFF**. Clear the query window and run **SELECT * FROM Person.Person** again. Now list the query plan, and you will see that **SELECT * FROM Person.Person** is listed twice! But how could this be? Isn't a query plan reused?

SQL Server reuses query plans, but only if the query is exactly the same, and certain settings are the same as well. **ARITHABORT** is a setting that will trigger SQL Server to create a new plan if the setting value is different from a currently stored plan. These settings can be retrieved using the function **sys.dm_exec_plan_attributes**. There are a lot of attributes, and their values are stored together in a single value, so it's completely unreadable. Luckily for us, SQL Server MVP Erland Sommarskog has written a rather lengthy (but very good and detailed) [article](#) about this phenomenon. This part is especially relevant in his "Different Plans for Different Settings" section. I'm going to use his trick to get all these attributes on one line; you can read his blog to make sense of it all.

Code Listing 84: List plan attributes

```
SELECT cp.plan_handle, qp.query_plan, st.[text], attrlist
FROM sys.dm_exec_cached_plans cp
CROSS APPLY sys.dm_exec_query_plan(cp.plan_handle) AS qp
CROSS APPLY sys.dm_exec_sql_text(cp.plan_handle) AS st
CROSS APPLY (SELECT epa.attribute + '=' + convert(nvarchar(127),
epa.value) + '
FROM sys.dm_exec_plan_attributes(cp.plan_handle) epa
WHERE epa.is_cache_key = 1
ORDER BY epa.attribute
FOR XML PATH('')) AS a(attrlist)
WHERE qp.[dbid] = DB_ID('AdventureWorks2014')
```

The takeaway here is that SQL Server creates a plan based on certain settings. **ARITHABORT** is a very nasty one, too. When you're running a query from C# code, be it through ADO.NET or Entity Framework, **ARITHABORT** is set to **OFF** by default. However, if you run a query in SQL Server, **ARITHABORT** is **ON** by default. So whenever your application is running slowly because of some query, and you run the query in SSMS, it will probably be lightning-fast—it will create a new plan because the **ARITHABORT** setting is different. I have been stumbling around in the dark for years not knowing about this!

Now that you know how to get the query plan for a single query, it is also possible to delete this plan from the cache. We have used **DBCC FREEPROCCACHE** to delete all plans from the cache, but you can use the **plan_handle** to delete just that one plan. Just grab the handle and pass it to **DBCC FREEPROCCACHE**.

Code Listing 100: DBCC FREEPROCCACHE (plan_handle)

[illegible]

Another DMV we can use is **sys.dm_tran_locks**. It shows currently active requests to the lock manager. Additionally, **sys.dm_os_waiting_tasks** shows all sessions that are currently waiting on a lock resource. So combining the two, we get the sessions that are currently waiting to acquire a lock along with the lock information.

Code Listing 101: List blocked sessions

```
SELECT *
FROM sys.dm_tran_locks t1
JOIN sys.dm_os_waiting_tasks AS owt ON t1.lock_owner_address =
owt.resource_address
```

The most interesting columns right now are the **request_session_id** in **dm_tran_locks** (which should correspond to the **session_id** in **dm_os_waiting_tasks**) and the **blocking_session_id** in **dm_os_waiting_tasks**. We can use these columns to get the associated sessions from **sys.dm_exec_connections**.

Code Listing 85: List blocked connections

```
SELECT *
FROM sys.dm_tran_locks t1
JOIN sys.dm_os_waiting_tasks AS owt ON t1.lock_owner_address =
owt.resource_address
JOIN sys.dm_exec_connections ec_waiting ON ec_waiting.session_id =
t1.request_session_id
JOIN sys.dm_exec_connections ec_blocking ON ec_blocking.session_id =
owt.blocking_session_id
```

The **dm_exec_connections** view has a column **most_recent_sql_handle**. This is the last query that was executed by the session, and must be the queries that are currently blocking and waiting. We have already seen how to get the query text and plan from a **sql_handle**, so we can apply that here as well. Furthermore, a lock is requested on some partition, as indicated by the **resource_associated_entity_id** in **dm_tran_locks**. The partition belongs to an object, so this is where we get the object ID, which we can then use to get the object name. If we filter out some fields, we're left with a resulting table that's actually pretty readable.

Code Listing 86: List blocked queries

```
SELECT t1.request_session_id
      , owt.blocking_session_id
      , t1.resource_type
      , waiting_text.[text] AS WaitingText
      , blocking_text.[text] AS BlockingText
      , OBJECT_NAME(p.object_id) AS ObjectName
FROM sys.dm_tran_locks t1
JOIN sys.dm_os_waiting_tasks AS owt ON t1.lock_owner_address =
owt.resource_address
JOIN sys.dm_exec_connections ec_waiting ON ec_waiting.session_id =
t1.request_session_id
JOIN sys.dm_exec_connections ec_blocking ON ec_blocking.session_id =
owt.blocking_session_id
```

```

CROSS APPLY sys.dm_exec_sql_text(ec_waiting.most_recent_sql_handle) AS
waiting_text
CROSS APPLY sys.dm_exec_sql_text(ec_blocking.most_recent_sql_handle) AS
blocking_text
JOIN sys.partitions p ON p.hobt_id = tl.resource_associated_entity_id

```

	request_session_id	blocking_session_id	resource_type	WaitingText	BlockingTest	ObjectName
1	54	52	KEY	(@1 tinyint)SELECT * FROM [Production].[Product] ...	BEGIN TRAN UPDATE Production.Product SET Na...	Product

Figure 33: Currently blocked query

If you decide that the blocker is less important than the waiter, or that the blocker is breaking too much, you can easily kill the blocker if you want to.

Code Listing 87: List and kill a session

```

EXEC sp_who -- Lists all sessions.
EXEC sp_who2 -- Lists all sessions+.

KILL 54 -- Kill a session.

```

DMVs are an invaluable tool when trying to troubleshoot problems. Unfortunately, there are a whole lot of them, and most of them just don't make sense when you don't know exactly what's going on. One person who does know what's going on, and who has helped me and my coworkers a lot, is Pinal Dave. I've never met him in person, but luckily he shares his knowledge in his [blogs](#). No doubt you've already come across him—he's the (self-proclaimed) SQL Authority. Hopefully, next time you're troubleshooting some query, you know what to do.



Tip: We have seen various methods of finding out what's going on in your SQL Server database. Another way of seeing what's up is by using the Activity Monitor. It shows processor time, waits, I/O, and recent expensive queries. It's less detailed than the methods laid out in this chapter, but gives a nice overall picture of your database performance. To open it, simply right-click on your instance in the Object Explorer and choose Activity Monitor from the context menu, or click Activity Monitor on the top menu.

Chapter 8 Continuous Integration

Up to now, we have seen how we can develop applications using C# and SQL Server using ADO.NET and the Entity Framework. But developing applications is only one aspect of the typical developer's job. At the start of the day we check out the newest code, at the end of the day we commit our own changes, and we probably do that a few times in between as well. Source control has been an invaluable tool in our day-to-day lives. Next to that, we have to make sure that our changes don't break any existing code, and if they do, we want to know as soon as possible. Writing unit tests for our code—whether upfront, using test-driven development, or after we've written the code—has become a big part of our jobs. Last, but not least, we want to deploy our application to any environment, and the fewer manual actions we have to take, the better.

Yet, when we do all this, the database is usually just a bunch of scripts that we have to save, test, and deploy manually. I have to admit that's how I do it, too. Unfortunately, databases aren't just a bunch of code files that can easily be compiled and copied for deployment. Even though I have little experience with the topics in this chapter, that doesn't mean I can't point you in the right direction.

Source control

When you're using Entity Framework Code First or Database Projects to develop your database, you already get some sort of database source control. After all, it's pretty easy to update a database or go back to a specific version. However, when you're not using either of those, you're probably stuck saving SQL scripts, and they probably go one way only, so going back to a previous version isn't really an option. None of those are examples of true source control, though. Unfortunately, all SQL source control systems I could find were paid ones, and I'm not discussing paid products in this book. So although SSMS has an options page for source control, this is only to select a third-party plugin, and the choices are limited.

It is possible to put your database into source control systems such as SVN or Git. Probably the best one out there is SQL Source Control by [Redgate](#). I have used Redgate products in the past, and they're very good tools. Unfortunately, they aren't cheap.

Other tools I've found are [GitSQL](#), [ApexSQL](#) (I've used their Diff tool), [AgentSvn](#), and [Liquibase](#). All have free trials or (very) limited free versions, so you can try them at your own risk and leisure.

For now, it seems database projects are the easiest and, most of all, cheapest method of maintaining some sort of source control for your database. And that's only your schema—getting (some) data into source control seems, for now, impossible.

Testing

When it comes to testing our code, we usually mock our data access layer. After all, our tests should run independent of the database. We test if our logic is correct, if $1 + 1$ really equals 2. But what we often don't test is whether the database really returns what we expect it to return.

Once again, Redgate has a rather pricey [testing tool](#) for SQL Server unit testing. I'm not discussing it here, but if you really want a good testing tool, that's probably the way to go. For testing, however, we don't need a third-party tool. The Visual Studio Database Project includes testing tools.

Start Visual Studio and create a new SQL Server Database Project. I've called mine **DbTest**, but name it whatever you like. Now add a table, and call it **Person**. Make the default **Id** field an auto-incrementing field by setting **Identity Specification** to **True** in the options pane. Then add a non-nullable field, **FirstName varchar(256)**, and a nullable field, **LastName varchar(256)**. Next, add a stored procedure to the project, call it **GetFullName**, and change the contents of the created file to the following.

Code Listing 88: GetPeople stored procedure

```
CREATE PROCEDURE [dbo].[GetFullName]
    @PersonId INT
AS
SET NOCOUNT ON
SELECT FirstName + ' ' + LastName
FROM Person
WHERE Id = @PersonId
```

Now publish the database. If it fails, you may need to set the Target Platform to SQL Server 2014 in the project properties. If everything went well, you should now have a database with a single table and a single stored procedure.

To create a unit test, right-click on the stored procedure, and in the context menu, choose **Create Unit Tests...** You now get a window where you can pick stored procedures to test, select a type and name for your test project, and pick a name for your test class. Be sure to change the default VB project to a C# project. Call the project whatever you like; I've called it **DbTest.Tests**. I've called my unit test **GetFullNameTests**. In the next part, you can pick a connection, a secondary connection, and whether or not your changes should automatically be deployed before a test run. Set the connection to your new **DbTest** and select **Automatically deploy...** Pick your **DbTest** project as Database project and debug as your Deployment configuration.

You now get a Test Project with a SQL Server unit test and a regular unit test class. You can delete the regular unit test class. The SQL Server unit test contains a SQL script that says **-- database unit test for dbo.GetFullName** at the top. It should be shown by default after you create the project. Underneath the script are the test conditions. You will want to delete the default **Inconclusive** condition, as it will always make your test fail. Now change the script to the following.

Code Listing 89: SQL unit test

```
EXEC GetFullName 1
```

Add a test condition of the type **Scalar Value**. In the options of the condition, set **Null expected** to **False** and **Expected value** to **Sander Rosse1** (or your own name). You can rename your test in the top right above your query; I've renamed it **FirstAndLastName**.

Now, in SSMS, add some data to your database. If you didn't test for **Sander Rosse1** in the condition, make sure you change the value in the following script as well.

Code Listing 90: Insert test data

```
INSERT INTO Person
(FirstName, LastName)
VALUES
('Sander', 'Rosse1'),
('Satya', NULL)
```

Now you are ready to run your test. In Visual Studio, go to the menu and select **Test > Windows > Test Explorer**. In the Test Explorer, simply click **Run All**. Your test should now run and pass, if you did everything correctly.

Now let's add a failing test. Go to your **GetFullNameTests** again, and add a test by clicking the green plus sign somewhere in the bar above the query. Name your test **OnlyFirstName**. Again, remove the Inconclusive condition add a Scalar Value, and test for **Satya**. Change your script to the following.

Code Listing 91: Failing unit test

```
EXEC GetFullName 2
```

Now run your tests again, and **OnlyFirstName** should fail. It will tell you that value **Satya** was expected, but the returned value was null. That's right, because the procedure simply concatenates **FirstName** and **LastName**, but **LastName** is **NULL**, and by default, anything + **NULL** = **NULL**. We should use the **CONCAT** function instead. So in Visual Studio, go to the **GetFullName** file and change the **SELECT** line with the following.

Code Listing 92: Fixed procedure

```
SELECT CONCAT(FirstName, ' ' + LastName)
```

Run your tests again, and this time they should both pass. The change is automatically published because that's what we specified when we created the project. If someone changed this procedure back to what it was, because it has less verbose syntax, and he doesn't quite get **CONCAT**, our test will fail again. More likely, if someone adds **MiddleName**, we'll know pretty soon if their change broke our cases where **MiddleName** is **NULL**. Likewise, they can add unit tests for **MiddleName**.

Deployment

Deploying a database is often a manual task. DBAs want to keep total control of their database. The IT department doesn't trust automated deployment, or is afraid it's triggered at a wrong time. They're not confident that the development team has their scripts in order so that an automatic database update goes exactly as planned, always. When the database goes awry, it's often a costly exercise to get everything running again. In short, automated database updates make a lot of people very nervous. Manually running scripts is the way to go. We have seen that (semi-) automatic deployment is pretty easy using Entity Framework Code First or Database Projects. When using a Database Project, you could even go for (manual) deployment using a Data-tier application package.

Be that as it may, it is possible to just automate database deployment. Perhaps you'd like this for noncritical environments, such as a test environment. Again, [Redgate](#) has a good, but expensive, solution. We've talked about Database Projects, so I'm skipping that here. One last thing I'd like to discuss is the SQL Server command line tool, or the **sqlcmd**. Chances are you need to download it, especially when you're using a separate server that handles updates. You can download the Command Line Utilities for SQL Server in the [Microsoft Download Center](#). With this tool, you can execute scripts from the command line, batch scripts, and your favorite CI tools. The **sqlcmd** tool was introduced with SQL Server 2005, when it replaced the functionally similar **isql** and **osql** tools.

Once you have the **sqlcmd** installed, open up a command window. Now, in the command window type the following (hit enter for a new line).

Code Listing 93: SELECT using sqlcmd

```
sqlcmd -S [servername\instancename] -E -d AdventureWorks2014
SELECT *
FROM Person.Person
WHERE BusinessEntityID = 1
GO
```

On the first line, we connect to our database. Each line after that makes up your query, which is executed after a **GO** command. The **-S** parameter is the name of your server, **-E** indicates we're logging in with a trusted connection, and **-d** specifies the database. To exit the **sqlcmd**, simply type **exit**. Note that all parameters are case sensitive. If you're not on a trusted connection and you would like to log in using a username and password, you can use the following command instead.

Code Listing 94: sqlcmd using SQL Server login

```
sqlcmd -S [servername\instancename] -U username -P password -d
AdventureWorks2014
```

When you run the **sqlcmd** from the command prompt, you can stretch your query over multiple lines and execute using **GO**. When you're not in the command prompt, you need to use **-q** (query) or **-Q** (query and exit) and a single line query.

Code Listing 95: sqlcmd using a direct query

```
sqlcmd -S [server\instance] -E -d AdventureWorks2014 -Q "SELECT * FROM  
Person.Person WHERE BusinessEntityID = 1"
```

Of course, you can also specify a script file to run using **-i**. Create a file somewhere (I put it in C:\Temp), name it something (like **SelectPerson1.sql**), and then you can run the following command.

Code Listing 96: sqlcmd using an input file

```
sqlcmd -S [server\instance] -E -d AdventureWorks2014 -i  
C:\Temp\SelectPerson1.sql
```

Now the results are a little hard to read in the command prompt. You can also output them to a file using **-o**.

Code Listing 97: sqlcmd using an output file

```
sqlcmd -S [server\instance] -E -d AdventureWorks2014 -i  
C:\Temp\SelectPerson1.sql -o C:\Temp\result.txt
```

Another option you may like to use is **-b**, which terminates the job if an error occurs. You can find all options in the **sqlcmd** [documentation](#).

Using the **sqlcmd**, you can automate certain SQL tasks by running them from the command line. I should mention that this in no way replaces the SQL Agent, which also runs automated SQL jobs. The thing is, you can't run the SQL Agent from your CI tool, while you can run the **sqlcmd** utility. The **sqlcmd** tool still does not automate your deployments, but it may be useful in running scripts automatically before your tests start off, or after you commit some new scripts to your source control system.

Conclusion

The tools laid out in this book do not make you the best C# developer around, nor do they teach you how to be an awesome DBA. They give you the middle ground of both. I've seen many developers who lack even the most basic database skills. SQL injection still happens way too often. Software becomes slow after a year of production because the SQL queries are everything but optimal. Companies have to invest in expensive server hardware because developers put queries in loops, get surprised by the lazy-loading ability of ORMs, or just write plain awful SQL queries. While this book has not helped you much in the ways of writing your queries, it hopefully helped in giving you a broader perspective on the database in general. Parameterize your queries, but be wary of parameter sniffing. Know that your one query that takes five minutes to run is possibly blocking the entire system. Identify such bottlenecks and fix them, or prevent them from ever happening.