ENTITY FRAMEWORK CORE

SUCCINCTLY

BY RICARDO PERES

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Entity Framework Core Succinctly

By
Ricardo Peres

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

Important licensing information. Please read.

Technical Reviewer: Jeff Boenig
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

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

Ricardo Peres is a Portuguese developer who has been working with .NET since 2001. He's a technology enthusiast, and has worked in many areas, from games to enterprise applications. His main interests currently are enterprise application integration and web technologies. He works for a UK company called Simplify Digital as a technical evangelist, and authored Entity Framework Core Cookbook – Second Edition. Ricardo contributed to the *Succinctly* collection with books on Entity Framework Code First, NHibernate, Microsoft Unity, and multitenancy with ASP.NET. He keeps a blog on technical subjects called Development With A Dot, and can be followed on Twitter as @rjperes75.

Introduction

Object/Relation Mappers (O/RMs) exist to bridge a gap between object-oriented programming (OOP) and relational databases. At the expense of being less specific, O/RMs abstract away database-specific technicalities—and hide those scary SQL queries from you, the OOP developer. O/RMs are very popular in whatever language you work with, be it Java, C#, PHP, or Python, because they actually can increase productivity a lot.

Entity Framework is Microsoft's flagship O/RM, and the recommended way to access relational databases. It has been around since 2009, and has experienced some changes, the most obvious of which is the introduction of the Code First model, with which developers would first design their model classes and have Entity Framework create the actual database structures.

Entity Framework Core is the brand-new data access technology from Microsoft. It is a complete rewrite from the "classic" Entity Framework, yet follows the Code First direction.

Entity Framework Core (EFC) builds on Entity Framework Code First, a very popular release that was introduced with the Code First workflow in mind: developers would first create POCO classes to map the concepts they wanted to reproduce, and Entity Framework would take care of their persistence in a semi-transparent way.

So, what is new? Well, a lot, actually. For one, it builds on the new .NET Core framework, which is multiplatform. Yeah, you heard it right: your code will run on Linux and Mac, in addition to good, old Windows! But that is not all. Entity Framework Core will feature the capability to connect to nonrelational data sources—think NoSQL databases. It also adds a couple of interesting features, while keeping most (but not all, as of now) features that made Code First—and will probably make Core—so popular.

Entity Framework Core, although recent, has already had three major versions: 1, 1.1 and 2. With 2, it has finally gained some maturity.

In this book, I will try to make clear what is new, what has changed, and what was dropped as of Entity Framework Core 2.0. Find this book's source code here. Stick with me as we explore Entity Framework Core!

Chapter 1 Setting Up

Before we start

Before you start using EF Core, you need to have its assemblies deployed locally. The distribution model followed by Microsoft and a number of other companies does not depend on old-school Windows installers, but instead relies on new technologies such as NuGet and Git. We'll try to make sense of each of these options in a moment, but before we get to that, make sure you have Visual Studio 2015 or 2017 installed (any edition including Visual Studio Community Edition will work), as well as SQL Server 2012 (any edition, including Express) or higher. In SQL Server Management Studio, create a new database called **Succinctly**.

Getting Entity Framework Core from NuGet

NuGet is to .NET package management what Entity Framework is to data access. In a nutshell, it allows Visual Studio projects to have dependencies on software packages—assemblies, source code files, PowerShell scripts, etc.—stored in remote repositories. Being highly modular, EF comes in a number of assemblies, which are deployed out-of-band, unrelated to regular .NET releases. In order to install it to an existing project, first run the Package Manager Console from Tools > Library Package Manager and enter the following command.

```
Install-package Microsoft.EntityFrameworkCore
```

Because of the new modular architecture, you will also need the provider for SQL Server (**Microsoft.EntityFramework.MicrosoftSqlServer**), which is no longer included in the base package (**Microsoft.EntityFramework**):

```
Install-package Microsoft.EntityFrameworkCore.SqlServer
```

This is by far the preferred option for deploying Entity Framework Core, the other being getting the source code from GitHub and manually building it. I'll explain this option next.

Getting Entity Framework Core from GitHub

The second option, for advanced users, is to clone the Entity Framework Core repository on GitHub, build the binaries yourself, and manually add a reference to the generated assembly.

First things first, let's start by cloning the Git repository using your preferred Git client.

Code Listing 1

```
https://github.com/aspnet/EntityFramework.git
```

Next, build everything from the command line using the following two commands.

Code Listing 2

```
build /t:RestorePackages /t:EnableSkipStrongNames
build
```

You can also fire up Visual Studio 2015 or 2017 and open the **EntityFramework.sln** solution file. This way, you can do your own experimentations with the source code, compile the debug version of the assembly, run the unit tests, etc.

Contexts

A context is a non-abstract class that inherits from DbContext. This is where all the fun happens: it exposes collections of entities that you can manipulate to return data from the store or add stuff to it. It needs to be properly configured before it is used. You will need to implement a context and make it your own, by making available your model entities and all of the necessary configuration.

Infrastructure methods

Entity Framework Core's DbContext class has some infrastructure methods that it calls automatically at certain times:

- **`OnConfiguring`**: Called automatically when the context needs to configure itself—setting up providers and connection strings, for example—giving developers a chance to intervene.
- **`OnModelCreating`**: Called automatically when Entity Framework Core is assembling the data model.
- **`SaveChanges`**: Called explicitly when we want changes to be persisted to the underlying data store. Returns the number of records affected by the saving operations.

Make yourself acquainted with these methods, as you will see them several times throughout the book.

Configuring the database provider

Entity Framework is database-agnostic, but that means that each interested party—database manufacturers or others—must release their own providers so that Entity Framework can use them. Out of the box, Microsoft makes available providers for SQL Server 2012, including Azure SQL Database, SQL Server Express, and SQL Server Express LocalDB, but also for SQLite and In Memory. The examples in this book will work on any of these providers. Make sure you have at least one of them installed and that you have the appropriate administrative permissions.

Entity Framework determines which connection to use through a new infrastructure method of DbContext, **`OnConfiguring`**, where we can explicitly configure it. This is substantially different from previous versions. Also, you can pass the configuration using the constructor that takes a **`DbContextOptions`** parameter.

SQL Server

Before we can connect to SQL Server, we need to have locally the Microsoft.EntityFrameworkCore.SqlServer NuGet package. See Appendix A for other options.

Configuring the service provider

An Entity Framework Core context uses a service provider to keep a list of its required services. When you configure a specific provider, the process registers all of its specific services, which are then combined with the generic ones. It is possible to supply our own service provider or just replace one or more of the services. We will see an example of this.

Sample domain model

Let's consider the following scenario as the basis for our study.

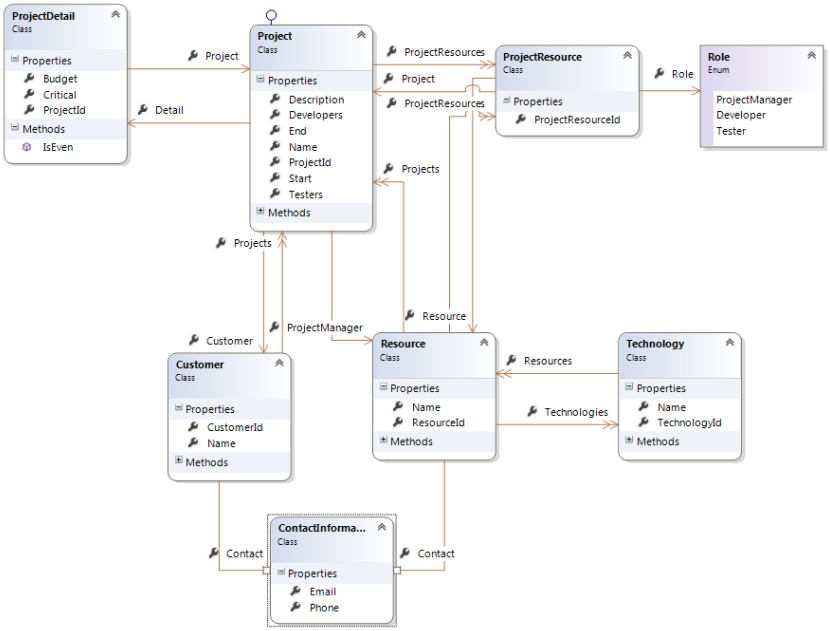

Figure 1: The domain model

You will find all these classes in the accompanying source code. Let's try to make some sense out of them:

- A **`Customer`** has a number of **`Projects`**.

- Each **`Project`** has a collection of **`ProjectResources`**, belongs to a single **`Customer`**, and has a **`ProjectDetail`** with additional information.
- A **`ProjectDetail`** refers to a single **`Project`**.
- A **`ProjectResource`** always points to an existing **`Resource`** and is assigned to a **`Project`** with a given **`Role`**.
- A **`Resource`** knows some **`Technologies`** and can be involved in several **`Projects`**.
- A **`Technology`** can be collectively shared by several **`Resources`**.
- Both **`Customers`** and **`Resources`** have **`Contact`** information.

Note: You can find the full source code in my Git repository** here**.

Core concepts

Before a class model can be used to query a database or to insert values into it, Entity Framework needs to know how it should translate entities (classes, properties, and instances) back and forth into the database (specifically, tables, columns, and records). For that, it uses a mapping, for which two APIs exist. More on this later, but first, some fundamental concepts.

Contexts

Again, a context is a class that inherits from DbContext and exposes a number of entity collections in the form of DbSet<T> properties. Nothing prevents you from exposing all entity types, but normally you only expose aggregate roots, because these are the ones that make sense querying on their own. Another important function of a context is to track changes to entities so that when we are ready to save our changes, it knows what to do. Each entity tracked by the context will be in one of the following states: unchanged, modified, added, deleted, or detached. A context can be thought of as a sandbox in which we can make changes to a collection of entities and then apply those changes with one **`save`** operation.

An example context might be the following.

Code Listing 3

```
public class ProjectsContext : DbContext
{
  public DbSet<Resource> Resources { get;  private set; }
public DbSet<Resource> Resources { get;  private set; }
  public DbSet<Project> Projects { get;  private set; }
  public DbSet<Customer> Customers { get;  private set; }
  public DbSet<Technology> Technologies { get;  private set; }
}
```

Note: Feel free to add your own methods, business or others, to the context class.

The DbContext class offers two public constructors, allowing the passing of context options:

Code Listing 4

```
public class ProjectsContext : DbContext
{
  public ProjectsContext(string connectionString) : base
(GetOptions(connectionString))
  {
  }

  public ProjectsContext(DbContextOptions options) : base(options)
  {
  }

  private static DbContextOptions GetOptions(string connectionString)
  {
    var modelBuilder = new DbContextOptionsBuilder();
    return modelBuilder.UseSqlServer(connectionString).Options;
  }
}
```

Context options include the specific database provider to use, its connection string, and other applicable properties.

This is just one way to pass a connection string to a context, we can also use the `OnConfiguring` method that will be described shortly.

Entities

At the very heart of the mapping is the concept of an entity. An entity is just a class that is mapped to an Entity Framework context, and has an identity (a property that uniquely identifies instances of it). In domain-driven design (DDD) parlance, it is said to be an aggregate root if it is meant to be directly queried. Think of an entity detail that is loaded together with an aggregate root and not generally considerable on its own, such as project details or customer addresses. An entity is usually persisted on its own table and may have any number of business or validation methods.

Code Listing 5

```
public class Project
{
  public int ProjectId { get; set; }

  public string Name { get; set; }

  public DateTime Start { get; set; }

  public DateTime? End { get; set; }

  public ProjectDetail Detail { get; set; }
```

```
  public Customer Customer { get; set; }

  public void AddResource(Resource resource, Role role)
  {
    resource.ProjectResources.Add(new ProjectResource()
      { Project = this, Resource = resource, Role = role });
  }

  public Resource ProjectManager
  {
    get
    {
      return ProjectResources.ToList()
        .Where(x => x.Role == Role.ProjectManager)
        .Select(x => x.Resource).SingleOrDefault();
    }
  }

  public IEnumerable<Resource> Developers
  {
    get
    {
      return ProjectResources.Where(x => x.Role == Role.Developer)
        .Select(x => x.Resource).ToList();
    }
  }

  public IEnumerable<Resource> Testers
  {
    get
    {
      return ProjectResources.Where(x => x.Role == Role.Tester)
        .Select(x => x.Resource).ToList();
    }
  }

  public ICollection<ProjectResource> ProjectResources { get; } =
new HashSet<ProjectResource>();

  public override String ToString()
  {
    return Name;
  }
}
```

Here you can see some patterns that we will be using throughout the book:

- An entity needs to have at least a public parameterless constructor.
- An entity always has an identifier property, which has the same name and ends with **`Id`**.

- Collections are always generic, have protected setters, and are given a value in the constructor in the form of an actual collection (like HashSet<T>).
- Calculated properties are used to expose filtered sets of persisted properties.
- Business methods are used for enforcing business rules.
- A textual representation of the entity is supplied by overriding the ToString.

Tip: Seasoned Entity Framework developers will notice the lack of the virtual qualifier for properties. It is not required since Entity Framework Core 1.1 does not support lazy properties, as we will see.

Tip: In the first version of Entity Framework Core, complex types are not supported, so nonscalar properties will always be references to other entities or to collections of them.

A domain model where its entities have only properties (data) and no methods (behavior) is sometimes called an Anemic Domain Model. You can find a good description for this anti-pattern on Martin Fowler's website.

Complex types

Complex types, or owned entities, were introduced in EF Core 2.0. They provide a way to better organize our code by grouping together properties in classes. These classes are not entities because they do not have identity (no primary key) and their contents are not stored in a different table, but on the table of the entity that declares them. A good example is an **`Address`** class: it can have several properties and can be repeated several times (e.g., work address, personal address).

Scalar properties

Scalars are simple values like strings, dates, and numbers. They are where actual entity data is stored—in relational databases and table columns—and can be of one of any of the following types.

Table 1 : Scalar Properties

.NET Type	SQL Server Type	Description
`Boolean`	BIT	Single bit
`Byte`	TINYINT	Single byte (8 bits)
`Char`	CHAR, NCHAR	ASCII or UNICODE char (8 or 16 bits)

.NET Type	SQL Server Type	Description
Int16	SMALLINT	Short integer (16 bits)
Int32	INT	Integer (32 bits)
Int64	BIGINT	Long (64 bits)
Single	REAL	Floating point number (32 bits)
Double	FLOAT	Double precision floating point number (64 bits)
Decimal	MONEY, SMALLMONEY	Currency (64 bits) or small currency (32 bits)
Guid	UNIQUEIDENTIFIER	Globally Unique Identifier (GUID)
DateTime	DATE, DATETIME, SMALLDATETIME, DATETIME2	Date with or without time
DateTimeOffset	DATETIMEOFFSET	Date and time with timezone information
TimeSpan	TIME	Time
String	VARCHAR, NVARCHAR, XML	ASCII (8 bits per character), UNICODE (16 bits) or XML character string. Can also represent a Character Long Object (CLOB)
Byte[]	BINARY, VARBINARY,	Binary Large Object (BLOB)

.NET Type	SQL Server Type	Description
	ROWVERSION	
Enum	INT	Enumerated value

Tip: Spatial data types are not supported yet, but they will be in a future version.

The types **Byte**, **Char**, and **String** can have a maximum length specified. A value of **-1** translates to **MAX**.

All scalar types can be made "nullable," meaning they might have no value set. In the database, this is represented by a **NULL** value.

Scalar properties need to have both a getter and a setter, but the setter can have a more restricted visibility than the getter: internal, protected internal, or protected.

Some examples of scalar properties are as follows.

Code Listing 6

```
public class Project
{
  public int ProjectId { get; set; }

  public string Name { get; set; }

  public DateTime Start { get; set; }

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

All public properties are, by default, included in the model that Entity Framework uses to represent its entities. You can exclude them either by using attributes or by code configuration.

Identity properties

One or more of the scalar properties of your entity must represent the underlying table's primary key, which can be single or composite.

Primary key properties can only be of one of the basic types (any type in Table 1, except arrays and enumerations), but cannot be complex types or other entity's types.

Fields

New since Entity Framework Core 1.1 is the possibility of mapping fields, of any visibility. This comes in handy because it allows better encapsulation of inner data and helps prevent bad data. Fields, unlike public properties, are not automatically mapped and have to be explicitly added to the model. More on this in a moment.

References

A reference from one entity to another defines a bidirectional relation. There are two types of reference relations:

- **Many-to-one**: Several instances of an entity can be associated with the same instance of another type (such as projects that are owned by a customer).

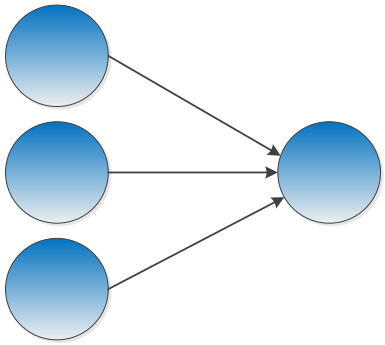

Figure 2: Many-to-one relationship

- **One-to-one**: An instance of an entity is associated with another instance of another entity; this other instance is only associated with the first one (such as a project and its detail).

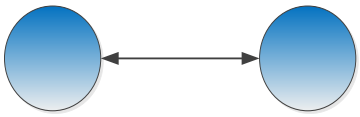

Figure 3: One-to-one relationship

In EF, we represent an association by using a property of the other entity's type.

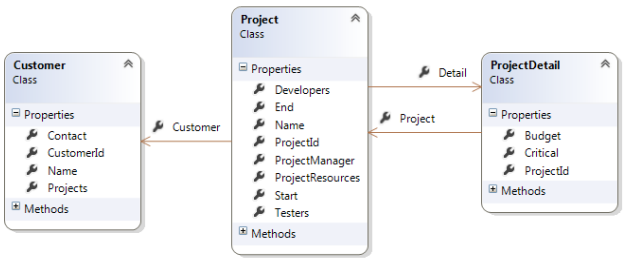

Figure 4: References: one-to-one, many-to-one

We call an entity's property that refers to another entity as an endpoint of the relation between the two entities.

Code Listing 7

```
public class Project
{
  //one endpoint of a many-to-one relation.
  public Customer Customer { get; set; }

  //one endpoint of a one-to-one relation.
  public ProjectDetail Detail { get; set; }
}

public class ProjectDetail
{
  //the other endpoint of a one-to-one relation.
  public Project Project { get; set; }
}

public class Customer
{
  //the other endpoint of a many-to-one relation.
  public ICollection<Project> Projects { get; protected set; }
}
```

Note: By merely looking at one endpoint, we cannot immediately tell what its type is (one-to-one or many-to-one), we need to look at both endpoints.

Collections

Collections of entities represent one of two possible types of bidirectional relations:

- **One-to-many**: A single instance of an entity is related to multiple instances of some other entity's type (such as a project and its resources).

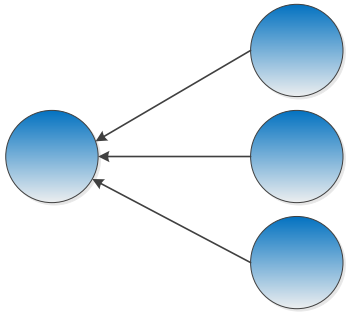

Figure 5: One-to-many relationship

- **Many-to-many**: A number of instances of a type can be related with any number of instances of another type (such as resources and the technologies they know). Entity Framework Core currently does not support this kind of relation directly.

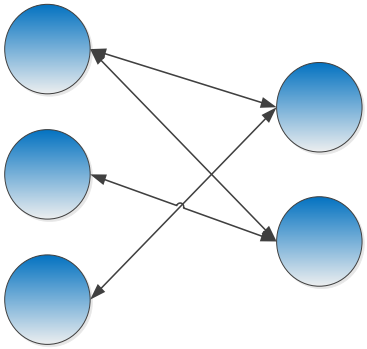

Figure 6: Many-to-many relationship

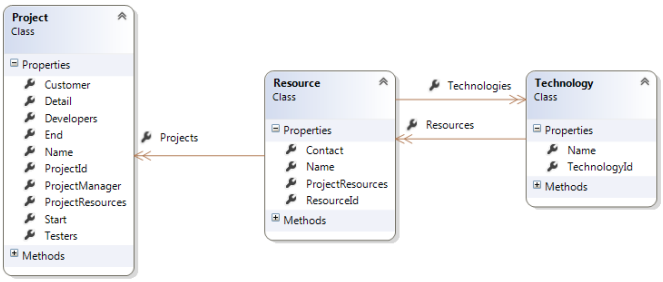

Figure 7: Collections: one-to-many

Entity Framework only supports declaring collections as ICollection<T> (or some derived class or interface) properties. In the entity, we should always initialize the collections properties in the constructor.

Code Listing 8

```
public class Project
{
  public Project()
  {
    ProjectResources = new HashSet<ProjectResource>();
  }

  public ICollection<ProjectResource> ProjectResources
  { get; protected set; }
}
```

Note: References and collections are collectively known as navigation properties, as opposed to scalar properties.

Tip: As of now, Entity Framework Core does not support many-to-many relations. The way to go around this limitation is to have two one-to-many relations, which implies that we need to map the association table.

Shadow properties

Entity Framework Core introduces a new concept, shadow properties, which didn't exist in previous versions. In a nutshell, a shadow property is a column that exists in a table but has no corresponding property in the POCO class. Whenever Entity Framework fetches an entity of a class containing shadow properties, it asks for the columns belonging to them.

What are they used for, then? Well, shadow properties are kept in Entity Framework's internal state for the loaded entities, and can be used when inserting or updating records, for example, to set auditing fields. Consider this case where we have an interface for specifying entities to be auditable, let's call it, then, `IAuditable`:

Code Listing 9

```
public interface IAuditable { }

public class ProjectsContext : DbContext
{
  public Func<string> UserProvider { get; set; } = () =>
    WindowsIdentity.GetCurrent().Name;
  public Func<DateTime> TimestampProvider {  get; set ; } = () =>
DateTime.UtcNow;

  public DbSet<Project> Projects { get; protected set; }

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    foreach (var entity in modelBuilder.Model.GetEntityTypes()
      .Where(x => typeof(IAuditable).IsAssignableFrom(x.ClrType)))
    {
      entity.AddProperty("CreatedBy", typeof(string));
      entity.AddProperty("CreatedAt", typeof(DateTime));
      entity.AddProperty("UpdatedBy", typeof(string));
      entity.AddProperty("UpdatedAt", typeof(DateTime?));
    }

    base.OnModelCreating(modelBuilder);
  }

  public override int SaveChanges()
  {
    foreach (var entry in ChangeTracker.Entries().Where(e => e.State ==
      EntityState.Added || e.State == EntityState.Modified))
    {
      if (entry.Entity is IAuditable)
      {
        if (entry.State == EntityState.Added)
        {
          entry.Property("CreatedBy").CurrentValue = UserProvider();
          entry.Property("CreatedAt").CurrentValue = TimestampProvider();
```

```
        }
        else
        {
          entry.Property("UpdatedBy").CurrentValue = UserProvider();
          entry.Property("UpdatedAt").CurrentValue = TimestampProvider();
        }
      }
    }

    return base.SaveChanges();
  }
}
```

For the **`WindowsIdentity`** class we need to add a reference to the **System.Security.Principal.Windows** NuGet package. By default, we will be getting the current user's identity from it.

In the OnModelCreating method, we look for any classes implementing **`IAuditable`**, and for each, we add a couple of auditing properties. Then in SaveChanges, we iterate through all of the entities waiting to be persisted that implement **`IAuditable`** and we set the audit values accordingly. To make this more flexible—and unit-testable—I made the properties **`UserProvider`** and **`TimestampProvider`** configurable, so that you change the values that are returned.

Querying shadow properties is also possible, but it requires a special syntax—remember, we don't have "physical" properties:

Code Listing 10

```
var resultsModifiedToday = ctx
  .Projects
  .Where(x => EF.Property<DateTime>(x, "UpdatedAt") == DateTime.Today)
  .ToList();
```

Mapping by attributes

Overview

The most commonly used way to express our mapping intent is to apply attributes to properties and classes. The advantage is that, by merely looking at a class, one can immediately infer its database structure.

Schema

Unless explicitly set, the table where an entity type is to be stored is determined by a convention (more on this later on), but it is possible to set the type explicitly by applying a TableAttribute to the entity's class.

Code Listing 11

```
[Table("MY_SILLY_TABLE", Schema = "dbo")]
public class MySillyType { }
```

The Schema property is optional, and should be used to specify a schema name other than the default. A schema is a collection of database objects (tables, views, stored procedures, functions, etc.) in the same database. In SQL Server, the default schema is **dbo**.

For controlling how a property is stored (column name, physical order, and database type), we apply a ColumnAttribute.

Code Listing 12

```
[Column(Order = 2, TypeName = "VARCHAR")]
public string Surname { get; set; }
[Column(Name = "FIRST_NAME", Order = 1, TypeName = "VARCHAR")]
public string FirstName { get; set; }
```

If the TypeName is not specified, Entity Framework will use the engine's default for the property type. SQL Server will use **NVARCHAR** for **String** properties, **INT** for **Int32**, **BIT** for **Boolean**, etc. We can use it for overriding this default.

The Order applies a physical order to the generated columns that might be different from the order by which properties appear on the class. When the Order property is used, there should be no two properties with the same value in the same class.

Marking a scalar property as mandatory requires the usage of the RequiredAttribute.

Code Listing 13

```
[Required]
public string Name { get; set; }
```

Tip: When this attribute is applied to a String property, it not only prevents the property from being null, but also from taking an empty string.

Tip: For value types, the actual property type should be chosen appropriately. If the column is non-nullable, one should not choose a property type that is nullable, such as Int32?.

For a required entity reference, the same rule applies.

Code Listing 14

```
[Required]
public Customer Customer { get; set; }
```

Setting the maximum allowed length of a string column is achieved by means of the MaxLengthAttribute.

Code Listing 15

```
[MaxLength(50)]
public string Name { get; set; }
```

The MaxLengthAttribute can also be used to set a column as being a **CLOB**, a column containing a large amount of text. SQL Server uses the types **NVARCHAR(MAX)** (for UNICODE) and **VARCHAR(MAX)** (ASCII). For that, we pass a length of -1.

Code Listing 16

```
[MaxLength(-1)]
public string LargeText { get; set; }
```

It can also be used to set the size of a **BLOB** (in SQL Server, **VARBINARY**) column.

Code Listing 17

```
[MaxLength(-1)]
public byte[] Picture { get; set; }
```

Like in the previous example, the -1 size will effectively be translated to **MAX**.

Ignoring a property and having Entity Framework never consider it for any operations is as easy as setting a NotMappedAttribute on the property.

Code Listing 18

```
[NotMapped]
public string MySillyProperty { get; set; }
```

Fully ignoring a type, including any properties that might refer to it, is also possible by applying the NotMappedAttribute to its class instead.

Code Listing 19

```
[NotMapped]
public class MySillyType { }
```

Fields

Mapping fields needs to be done using mapping by code, which we'll cover shortly.

Primary keys

While database tables don't strictly require a primary key, Entity Framework requires it. Both single-column and multicolumn (composite) primary keys are supported. You can mark a property, or properties, as the primary key by applying a KeyAttribute.

Code Listing 20

```
[Key]
public int ProductId { get; set; }
```

If we have a composite primary key, we need to use mapping by code. The order of the keys is important so that EF knows which argument refers to which property when an entity is loaded by the Find method.

Code Listing 21

```
//composite id[Column(Order = 1)]
public int ColumnAId { get; set; } [Column(Order = 2)]
public int ColumnBId { get; set; }
```

Primary keys can also be decorated with an attribute that tells Entity Framework how keys are to be generated (by the database or manually). This attribute is DatabaseGeneratedAttribute, and its values are explained in further detail in an upcoming section.

Navigation properties

We typically don't need to include foreign keys in our entities; instead, we use references to the other entity, but we can have them as well. That's what the ForeignKeyAttribute is for.

Code Listing 22

```
public Customer Customer { get; set; }

[ForeignKey("Customer")]
public int CustomerId { get; set; }
```

The argument to ForeignKeyAttribute is the name of the navigation property that the foreign key relates to.

Now, suppose we have several relations from one entity to another. For example, a customer might have two collections of projects: one for the current, and another for the past projects. It could be represented in code as this:

Code Listing 23

```
public class Customer
{
  //the other endpoint will be the CurrentCustomer.
  [InverseProperty("CurrentCustomer")]
```

```
  public ICollection<Project> CurrentProjects { get; protected set; }

  //the other endpoint will be the PastCustomer.
  [InverseProperty("PastCustomer")]
  public ICollection<Project> PastProjects { get; protected set; }
}

public class Project
{
  public Customer CurrentCustomer { get; set; }

  public Customer PastCustomer { get; set; }
}
```

In this case, it is impossible for EF to figure out which property should be the endpoint for each of these collections, hence the need for the InversePropertyAttribute. When applied to a collection navigation property, it tells Entity Framework the name of the other endpoint's reference property that will point back to it.

Note: When configuring relationships, you only need to configure one endpoint.

Computed columns

Entity Framework Core does not support implicit mapping to computed columns, which are columns whose values are not physically stored in a table but instead come from SQL formulas. An example of a computed column is combining first and last name into a full name column, which can be achieved in SQL Server very easily. However, you can map computed columns onto properties in your entity explicitly.

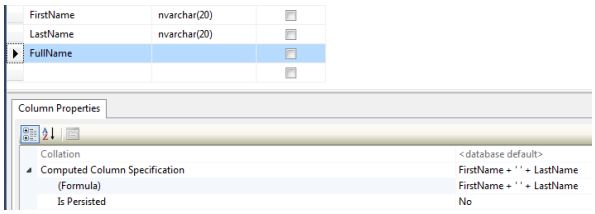

Figure 8: Computed columns

Another example of a column that is generated on the database is when we use a trigger for generating its values. You can map server-generated columns to an entity, but you must tell Entity Framework that this property is never to be inserted. For that, we use the DatabaseGeneratedAttribute with the option DatabaseGeneratedOption.Computed.

Code Listing 24

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

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

```
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public string FullName { get;  protected set; }
```

Since the property will never be set, we can have the setter as a protected method, and we mark it as **`DatabaseGeneratedOption.Computed`** to let Entity Framework know that it should never try to **`INSERT`** or **`UPDATE`** this column.

With this approach, you can query the **`FullName`** computed property with both LINQ to Objects and LINQ to Entities.

Code Listing 25

```
//this is executed by the database.
var me = ctx.Resources.SingleOrDefault(x => x.FullName == "Ricardo Peres");

//this is executed by the process.
var me = ctx.Resources.ToList().SingleOrDefault(x => x.FullName == "Ricardo
 Peres");
```

In this example, **`FullName`** would be a concatenation of the **`FirstName`** and the **`LastName`** columns, specified as a SQL Server T-SQL expression, so it's never meant to be inserted or updated.

Computed columns can be one of the following:

- Generated at insert time (**`ValueGeneratedOnAdd`**)
- Generated at insert or update time (**`ValueGeneratedOnAddOrUpdate`**)
- Never (**`ValueGeneratedNever`**)

Limitations

As of the current version of EF, there are some mapping concepts that cannot be achieved with attributes:

- Configuring cascading deletes (see "Cascading")
- Applying Inheritance patterns (see "Inheritance Strategies")
- Cascading
- Defining owned entities
- Defining composite primary keys

For these, we need to resort to code configuration, which is explained next.

Mapping by code

Overview

Convenient as attribute mapping may be, it has some drawbacks:

- We need to add references in our domain model to the namespaces and assemblies where the attributes are defined (sometimes called domain pollution).
- We cannot change things dynamically; attributes are statically defined and cannot be changed at runtime.
- There isn't a centralized location where we can enforce our own conventions.

To help with these limitations, Entity Framework Core offers an additional mapping API: code or fluent mapping. All functionality of the attribute-based mapping is present, and more. Let's see how we implement the most common scenarios.

Fluent, or code, mapping is configured on an instance of the ModelBuilder class, and normally the place where we can access one is in the OnModelCreating method of the DbContext.

Code Listing 26

```
public class ProjectsContext : DbContext
{
  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
    //configuration goes here.

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

This infrastructure method is called by Entity Framework when it is initializing a context, after it has automatically mapped the entity classes declared as **`DbSet<T>`** collections in the context, as well as entity classes referenced by them.

Schema

Here's how to configure the entity mappings by code.

Code Listing 27

```
//set the table and schema.
modelBuilder
  .Entity<Project>()
  .ToTable("project", "dbo");

//ignoring an entity and all properties of its type.
```

```
modelBuilder.Ignore<Project>();
```

This is an example of mapping individual properties. Notice how the API allows chaining multiple calls together by having each method return the ModelBuilder instance. In this example, we chain together multiple operations to set the column name, type, maximum length, and required flag. This is very convenient, and arguably makes the code more readable.

Entities

You need to declare a class to be an entity (recognized by EF) if it's not present in a context collection property.

Code Listing 28

```
//ignore a property.
modelBuilder
  .Entity<MyClass>()
  .ToTable("MY_TABLE");
```

Since EF Core 2.0, more than one entity can share the same table (table splitting), with different properties mapped to each of them.

Primary keys

The primary key and the associated generation strategy are set as follows:

Code Listing 29

```
//setting a property as the key.
modelBuilder
  .Entity<Project>()
  .HasKey(x => x.ProjectId);

//and the generation strategy.
modelBuilder
  .Entity<Project>()
  .Property(x => x.ProjectId)
  .UseSqlServerIdentityColumn();

//composite keys
modelBuilder
  .Entity<CustomerManager>()
  .HasKey(x => new { x.ResourceId, x.CustomerId });
```

Instead of **`UseSqlServerIdentityColumn`**, you could instead have **`ForSqlServerUseSequenceHiLo`** for using sequences (High-Low algorithm).

Properties

You can configure individual properties in the **`OnModelCreating`** method, which includes ignoring them:

Code Listing 30

```
//ignore a property.
modelBuilder
  .Entity<Project>()
  .Ignore(x => x.MyUselessProperty);
```

Or setting nondefault values:

Code Listing 31

```
//set the maximum length of a property.
modelBuilder
  .Entity<Project>()
  .Property(x => x.Name)
  .HasMaxLength(50);
```

Setting a property's database attributes:

Code Listing 32

```
//set a property's values (column name, type, length, nullability).
modelBuilder
  .Entity<Project>()
  .Property(x => x.Name)
  .HasColumnName("NAME")
  .HasColumnType("VARCHAR")
  .HasMaxLength(50)
  .IsRequired();
```

Fields

Mapping fields needs to be done using their names and types explicitly:

Code Listing 33

```
//map a field.
modelBuilder
  .Entity<Project>()
  .Property<String>("SomeName")
  .HasField("_someName");
```

Mind you, the visibility of the fields doesn't matter—even **`private`** will work.

Navigation properties

Navigation properties (references and collections) are defined as follows.

Code Listing 34

```
//a bidirectional many-to-one and its inverse with cascade.
modelBuilder
  .Entity<Project>()
  .HasOne(x => x.Customer)
  .WithMany(x => x.Projects)
  .OnDelete(DeleteBehavior.Cascade);

//a bidirectional one-to-many.
modelBuilder
  .Entity<Customer>()
  .HasMany(x => x.Projects)
  .WithOne(x => x.Customer)
  .IsRequired();

//a bidirectional one-to-one-or-zero with cascade.
modelBuilder
  .Entity<Project>()
  .HasOptional(x => x.Detail)
  .WithOne(x => x.Project)
  .IsRequired()
  .OnDelete(DeleteBehavior.Cascade);

//a bidirectional one-to-one (both sides required) with cascade.
modelBuilder
  .Entity<Project>()
  .HasOne(x => x.Detail)
  .WithOne(x => x.Project)
  .IsRequired()
  .OnDelete(DeleteBehavior.Cascade);

//a bidirectional one-to-many with a foreign key property (CustomerId).
modelBuilder
  .Entity<Project>()
  .HasOne(x => x.Customer)
  .WithMany(x => x.Projects)
  .HasForeignKey(x => x.CustomerId);

//a bidirectional one-to-many with a non-conventional foreign key column.
modelBuilder
  .Entity<Project>()
  .HasOne(x => x.Customer)
  .WithMany(x => x.Projects)
  .Map(x => x.MapKey("FK_Customer_Id"));
```

Note: When configuring relationships, you only need to configure one endpoint.

Computed columns

Not all properties need to come from physical columns. A computed column is one that is generated at the database by a formula (in which case it's not actually physically stored); it can be specified with the following database generation option:

Code Listing 35

```
modelBuilder
  .Entity<Resource>()
  .Property(x => x.FullName)
  .ValueGeneratedNever();
```

In this example, **`FullName`** would be a concatenation of the **`FirstName`** and the **`LastName`** columns, specified as a SQL Server T-SQL expression, so it's never meant to be inserted or updated.

Computed columns can be one of the following:

- Generated at insert time (**`ValueGeneratedOnAdd`**).
- Generated at insert or update time (**`ValueGeneratedOnAddOrUpdate`**).
- Never (**`ValueGeneratedNever`**).

Default values

It's also possible to specify the default value for a column to be used when a new record is inserted:

Code Listing 36

```
modelBuilder
  .Entity<Resource>()
  .Property(x => x.FullName)
  .ForSqlServerHasDefaultValueSql("SomeFunction()");
```

Here, SomeFunction will be included in the CREATE TABLE statement if the database is created by Entity Framework. This configuration cannot be done using attributes.

Global Filters

In version 2.0, a very handy feature was introduced: global filters. Global filters are useful in a couple of scenarios, such as:

- **Multi-tenant code**: if we have a column that contains the identifier for the current tenant, we can filter by it automatically.

- **Soft-deletes**: instead of deleting records, we mark them as deleted, and filter them out automatically.

Global filters are defined in the `OnModelCreating` method, and its configuration is done like this:

Code Listing 37

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder
    .Entity<Project>()
    .HasQueryFilter(x => x.IsDeleted == false);

  base.OnModelCreating(builder);
}
```

Here we are saying that entities of type **`Project`**, whether loaded in a query or from a collection (one-to-many), are automatically filtered by the value in its **`IsDeleted`** property. Two exceptions:

- When an entity is loaded explicitly by its id (**`Find`**).

- When a one-to-one or many-to-one related entity is loaded.

You can have an arbitrary condition in the filter clause, as long as it can be executed by LINQ. For multi-tenant code, you would have something like this instead:

Code Listing 38

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

protected override void OnModelCreating(ModelBuilder builder)
{
  builder
    .Entity<Project>()
    .HasQueryFilter(x => x.TenantId == this.TenantId);

  base.OnModelCreating(builder);
}
```

Self-contained configuration

New in EF Core 2 is the possibility of having configuration classes for storing the code configuration for an entity; it's a good way to have your code organized. You need to inherit from **`IEntityTypeConfiguration<T>`** and add your mapping code there:

Code Listing 39

```
public class ProjectEntityTypeConfiguration :
IEntityTypeConfiguration<Project>
{
  public void Configure(EntityTypeBuilder<Project> builder)
  {
    builder
      .HasOne(x => x.Customer)
      .WithMany(x => x.Projects)
      .OnDelete(DeleteBehavior.Cascade);
  }
}
```

But these are not loaded automatically, so you need to do so explicitly:

Code Listing 40

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.ApplyConfiguration(new ProjectEntityTypeConfiguration());
}
```

Identifier strategies

Overview

Entity Framework requires that all entities have an identifier property that will map to the table's primary key. If this primary key is composite, multiple properties can be collectively designated as the identifier.

Identity

Although Entity Framework is not tied to any specific database engine, out of the box it works better with SQL Server. Specifically, it knows how to work with IDENTITY columns, arguably the most common way in the SQL Server world to generate primary keys. Until recently, it was not supported by some major database engines, such as Oracle.

By convention, whenever Entity Framework encounters a primary key of an integer type (**`Int32`** or **`Int64`**), it will assume that it is an IDENTITY column. When generating the database, it will start with value 1 and use the increase step of 1. It is not possible to change these parameters.

Tip: Although similar concepts exist in other database engines, Entity Framework can only use IDENTITY with SQL Server out of the box.

Manually generated primary keys

In the event that the identifier value is not automatically generated by the database, it must be manually set for each entity to be saved. If it is **`Int32`** or **`Int64`**, and you want to use attributes for the mapping, then mark the identifier property with a DatabaseGeneratedAttribute and pass it the DatabaseGeneratedOption.None. This will avoid the built-in convention that will assume IDENTITY.

Code Listing 41

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

Use the following if you prefer fluent mapping.

Code Listing 42

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder
    .Entity<Project>()
    .HasKey(x => x.ProjectId);

  modelBuilder
    .Property(x => x.ProjectId)
    .ValueGeneratedNever();

  base.OnModelCreating(modelBuilder);
}
```

In this case, it is your responsibility to assign a valid identifier that doesn't already exist in the database. This is quite complex, mostly because of concurrent accesses and transactions. A popular alternative consists of using a **`Guid`** for the primary key column. You still have to initialize its value yourself, but the generation algorithm ensures that there won't ever be two identical values.

Code Listing 43

```
public Project()
{
  //always set the ID for every new instance of a Project.
  ProjectId = Guid.NewGuid();
}

[Key]
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public Guid ProjectId { get; set; }
```

Note: When using non-integral identifier properties, the default is not to have them generated by the database, so you can safely skip the DatabaseGeneratedAttribute.

Note: Another benefit of using Guids for primary keys is that you can merge records from different databases into the same table; the records will never have conflicting keys.

High-Low

Another very popular identifier generation strategy is High-Low (or Hi-Lo). This one is very useful when we want the client to know beforehand what the identifier will be, while avoiding collisions. Here's a simple algorithm for it:

- Upon start, or when the need first arises, the ORM asks the database for a next range of values—the **High** part—which is then reserved.
- The ORM has either configured a number of **Lows** to use, or it will continue generating new ones until it exhausts the numbers available (reaches maximum number capacity).
- When the ORM has to insert a number, it combines the **High** part—which was reserved for this instance—with the next **Low**. The ORM knows what the last one was, and just adds 1 to it.
- When all the **Lows** are exhausted, the ORM goes to the database and reserves another **High** value.

There are many ways in which this algorithm can be implemented, but, somewhat sadly, it uses sequences for storing the current High values. What this means is that it can only be used with SQL Server 2012 or higher, which, granted, is probably not a big problem nowadays.

This is the way to configure sequences for primary key generation:

Code Listing 44

```
modelBuilder
  .Entity<Resource>()
  .Property(x => x.FullName)
  .ForSqlServerUseSequenceHiLo("sequencename");
```

Note: The sequence specified is set on the property, not the primary key, although the property should be the primary key.

Note: The sequence name is optional; it is used when you want to specify different sequences per entity.

Tip: Sequences for primary key generation cannot be specified by attributes; you need to use the fluent API.

Inheritance

Consider the following class hierarchy.

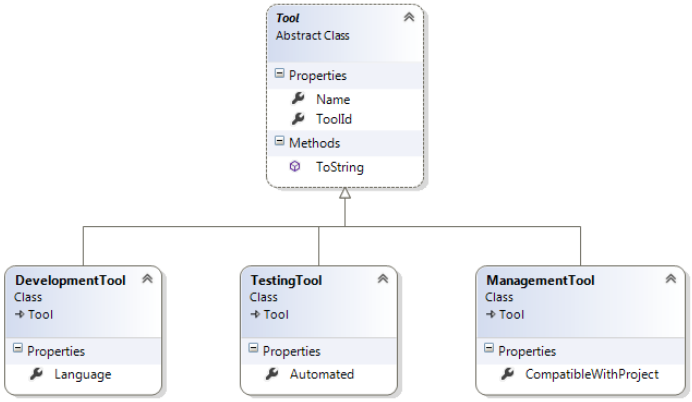

Figure 9: An Inheritance model

In this example, we have an abstract concept, **`Tool`**, and three concrete representations of it: **`DevelopmentTool`**, **`TestingTool`**, and **`ManagementTool`**. Each **`Tool`** must be one of these types.

In object-oriented languages, we have class inheritance, which is something relational databases don't natively support. How can we store this in a relational database?

Martin Fowler, in his seminal work *Patterns of Enterprise Application Architecture*, described three patterns for persisting class hierarchies in relational databases:

- **Single Table Inheritance** or **Table Per Class Hierarchy**: A single table is used to represent the entire hierarchy; it contains columns for all mapped properties of all classes. Many of these will be **`NULL`** because they will only exist for one particular class; one discriminating column will store a value that will tell Entity Framework which class a particular record will map to.

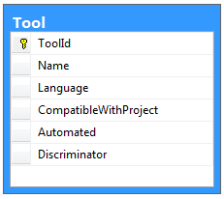

Figure 10: Single Table Inheritance data model

- **Class Table Inheritance** or **Table Per Class**: A table will be used for the columns for all mapped base-class properties, and additional tables will exist for all concrete classes; the additional tables will be linked by foreign keys to the base table.

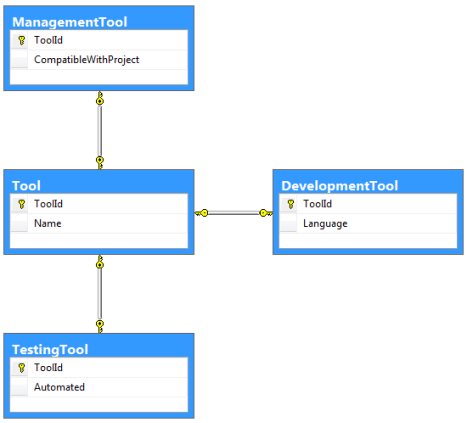

Figure 11: Class Table Inheritance data model

- **Concrete Table Inheritance** or **Table Per Concrete Class**: One table is used for each concrete class, each with columns for all mapped properties, either specific or inherited by each class.

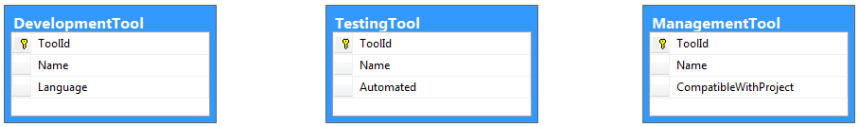

Figure 12: Concrete Table Inheritance data model

You can see a more detailed explanation of these patterns on Martin's website. For now, I'll leave you with some thoughts:

- **Single Table Inheritance**, when it comes to querying from a base class, offers the fastest performance because all information is contained in a single table. However, if you have lots of properties in all of the classes, it will be a difficult read, and you will have many "nullable" columns. In all of the concrete classes, all properties must be optional because they must allow null values. This is because different entities will be stored in the same class, and not all share the same columns.
- **Class Table Inheritance** offers a good balance between table tidiness and performance. When querying a base class, a **`LEFT JOIN`** will be required to join each table from derived classes to the base class table. A record will exist in the base class table and in exactly one of the derived class tables.
- **Concrete Table Inheritance** for a query for a base class requires several **`UNION`**s, one for each table of each derived class, because Entity Framework does not know beforehand in which table to look. This means that you cannot use **`IDENTITY`** as the identifier generation pattern or any one that might generate identical values for any two tables. Entity Framework would be confused if it found two records with the same ID. Also, you will have the same columns—those from the base class, duplicated on all tables.

As far as Entity Framework is concerned, there really isn't any difference; classes are naturally polymorphic. However, Entity Framework Core 1.1 only supports the Single Table Inheritance pattern. See 0 to find out how we can perform queries on class hierarchies. Here's how we can apply Single Table Inheritance:

Code Listing 45

```
public abstract class Tool
{
  public string Name { get; set; }

  public int ToolId { get; set; }
}

public class DevelopmentTool : Tool
{
  //String is inherently nullable.
  public string Language { get; set; }
}

public class ManagementTool : Tool
{
  //nullable Boolean
  public bool? CompatibleWithProject { get; set; }
}

public class TestingTool : Tool
{
  //nullable Boolean
  public bool? Automated { get; set; }
}
```

As you can see, there's nothing special you need to do. Single table inheritance is the default strategy. One important thing, though: because all properties of each derived class will be stored in the same table, all of them need to be nullable. It's easy to understand why. Each record in the table will potentially correspond to any of the derived classes, and their specific properties only have meaning to them, not to the others, so they may be undefined (**NULL**). In this example, I have declared all properties in the derived classes as nullable. Notice that I am not mapping the discriminator column here (for the Single Table Inheritance pattern); it belongs to the database only.

Some notes:

- All properties in derived classes need to be marked as nullable, because all of them will be stored in the same table, and may not exist for all concrete classes.
- The Single Table Inheritance pattern will be applied by convention; there's no need to explicitly state it.
- Other inheritance mapping patterns will come in the next versions of Entity Framework Core.

Conventions

The current version of Entity Framework Core at the time this book was written (1.1) comes with a number of conventions. Conventions dictate how EF will configure some aspects of your model when they are not explicitly defined.

The built-in conventions are:

- All types exposed from a DbSet<T> collection in the DbContext-derived class with public getters are mapped automatically.
- All classes that appear in DbSet<T> properties on a DbContext-derived class are mapped to a table with the name of the property.
- All types for which there is no DbSet<T> property will be mapped to tables with the name of the class.
- All public properties of all mapped types with a getter and a setter are mapped automatically, unless explicitly excluded.
- All properties of nullable types are not required; those from non-nullable types (value types in .NET) are required.
- Single primary keys of integer types will use **`IDENTITY`** as the generation strategy.
- Associations to other entities are discovered automatically, and the foreign key columns are built by composing the foreign entity name and its primary key.
- Child entities are deleted from the database whenever their parent is, if the relation is set to **`required`**.

For now, there is no easy way to add our own custom conventions or disable existing ones, which is rather annoying, but will improve in future versions. You can, however, implement your own mechanism by building an API that configures values in the **`ModelBuilder`** instance. In this case, you should plug this API in the **`OnModelCreating`** virtual method.

And, of course, we can always override the conventional table and column names using attributes.

Code Listing 46

```
//change the physical table name.
[Table("MY_PROJECT")]
public class Project
{
  //change the physical column name.
  [Column("ID")]
  public int ProjectId { get; set; }
}
```

Generating entities automatically

One thing often requested is the ability to generate entity types that Entity Framework Core can use straightaway. This normally occurs when we have an existing, large database with many tables for which it would be time consuming to create the classes manually. This is not exactly the purpose of the Code First approach, but we have that option. You should be thankful that we do—imagine how it would be if you had to generate hundreds of classes by hand.

In order to generate your classes, you need to run the following command:

```
dotnet ef dbcontext scaffold “Data Source=.\SQLEXPRESS; Initial
Catalog=Succinctly; Integrated Security=SSPI”
Microsoft.EntityFrameworkCore.SqlServer
```

Note: Make sure to replace the connection string with one that is specific to your system.

After executing this command, Entity Framework will generate classes for all the tables in your database that have a primary key defined. Because these are generated as partial classes, you can easily extend them without the risk of losing your changes in the next generation.

Chapter 2 Database

Configuring the connection string

We need to explicitly pass Entity Framework the connection string to use, which is related to a specific provider. We can do it in the **`OnConfiguring`** method, as follows:

Code Listing 47

```
public class ProjectsContext : DbContext
{
  private readonly String _nameOrConnectionString;

  public ProjectsContext(String nameOrConnectionString)
  {
    _nameOrConnectionString = nameOrConnectionString;
  }

  protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
  {
    optionsBuilder.UseSqlServer(_nameOrConnectionString);
    base.OnConfiguring(optionsBuilder);
  }
}
```

Another option is to use the base constructor that takes a **`DbContextOptions`** object:

Code Listing 48

```
public class ProjectsContext : DbContext
{
  public ProjectsContext(DbContextOptions options) : base(options)
  {
  }

  private static DbContextOptions GetOptions(string connectionString)
  {
    var optionsBuilder = new DbContextOptionsBuilder();
    return optionsBuilder.UseSqlServer(connectionString).Options;
  }
}
```

Generating the database

Explicit creation

The Code First model, as its name implies, comes before the database. Nevertheless, we still need it and have to create it. Entity Framework offers an API to do just that. If we decide that it should be created, we need to start a new context and ask it to do that for us by calling **`EnsureCreated`**:

Code Listing 49

```
using (var ctx = new ProjectsContext())
{
  //will create the database if it doesn't already exist.
  ctx.Database.EnsureCreated();
}
```

Tip: The user specified by the connection string, which can even be the current Windows user, must have access rights to create a database.

Tip: If the database already exists, `Create` will fail, and `CreateIfNotExists` will return `false`.

Database initializers

Entity Framework Core dropped database initializers—they are no longer included in the code.

Migrations

We all know that schema and data change over time, be it the addition of another column or a modification in a base record. Entity Framework Core offers a code-based approach for dealing with changes in schema: migrations.

Migrations offer fine-grained control over the generated model. We start by creating a migration with a name that describes our purpose, for which we will need some NuGet packages:

- https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Tools.DotNet as a **DotNetCliToolReference** in the .csproj file
- Microsoft.EntityFrameworkCore.Tools

For additional information, please refer to https://docs.microsoft.com/en-us/ef/core/get-started/aspnetcore/new-db.

And we execute `Add-Migration` in the Package Manager Console:

```
Add-Migration InitialVersion
```

Or, using the command line, do:

```
dotnet ef migrations add InitialCreate
```

This will only create the migrations classes, on the Migrations folder whose name reflects the name passed to **`Add-Migration`** and the timestamp of its creation, which inherits from class Migration. This new class is empty except for two method declarations, which are overridden from the base class:

- Up: will specify the changes that will be applied to the database when this migration is run.
- Down: will contain the reverse of the changes declared in the Up method, for the case when this migration is rolled back.

Note: This class is not specific to any Entity Framework Core context; it only cares about the database.

When Entity Framework creates a migration, it creates a **`__MigrationHistory`** table in the target database for data stores that support it, like relational databases, of course. This is where all migrations are tracked.

Let's see a simple example.

Code Listing 50

```
public partial class InitialCreate : Migration
{
  public override void Up()
  {
    /* … */
  }

  public override void Down()
  {
    /* … */
  }
}
```

As you can see, I didn't show all contents of the Up method, all operations are reversed in the Down method.

The **`Migration`** class contains helper methods for most typical database operations, but in case we need something different, we can always use the Sql method:

Code Listing 51

```
Sql("-- some SQL command");
```

Now that we have this migration, we might as well execute it, using the command line:

```
dotnet ef database update
```

Or the **Package Manager Console**:

```
Update-Database -TargetMigration InitialCreate
```

Tip: Unlike previous versions, there is no longer a `Seed` method.

Once again, we can see that the **`__MigrationHistory`** table was updated.

	MigrationId	Model	ProductVersion
1	201309211453190_InitialCreate	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
2	201309211516207_AutomaticMigration	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
3	201309211538380_AutomaticMigration	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45
4	201309211556555_AddProjectStatus	0x1F8B0800000000000400ED1D5D6FDCB8F1BD40FFC3629F...	5.0.0.net45

Figure 13: The __MigrationHistory table after the named migration

We add new migrations by calling **`Add-Migration`** with some name. Of course, we can always go back to the previous state, or the state identified by any named migration that has been run, by using **`Update-Database`**.

```
PM> Update-Database -ProjectName Model -TargetMigration $InitialDatabase
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Reverting migrations: [201309211556555_AddProjectStatus, 201309211538380_AutomaticMigration, 201309211516207_AutomaticMigration, 201309211453190_InitialCreate].
Reverting code-based migration: 201309211556555_AddProjectStatus.
Reverting automatic migration: 201309211538380_AutomaticMigration.
Reverting automatic migration: 201309211516207_AutomaticMigration.
Reverting automatic migration: 201309211453190_InitialCreate.
```

Figure 14: Returning to the initial database version

If you want to revert to a named migration, just pass its name as a parameter to **`Update-Database`**.

```
PM> update-database -ProjectName Model -TargetMigration SomeOldVersion
```

Figure 15: Reverting to a named version

At any time, you can see what migrations have been executed in the database by invoking the **`Get-Migrations`** command.

```
PM> get-migrations -ProjectName Model
Retrieving migrations that have been applied to the target database.
201309211858284_InitialCreate
```

Figure 16: Listing all applied migrations

One final word of caution: to use migrations, your context needs to have a public, parameter-less constructor. Why? Because it needs to be instantiated through reflection by the migrations infrastructure, which has no way of knowing which parameters it should take. Strictly speaking, this is not entirely true: if the migrations framework can find a public implementation of **`IDesignTimeDbContextFactory<T>`** (used to be **`IDbContextFactory<T>,`** but this interface was deprecated in EF Core 2), where **`T`** is the type of your DbContext, then it will call its Create method automatically:

Code Listing 52

```
public class ProjectsContextFactory :
IDesignTimeDbContextFactory<ProjectsContext>
{
  public ProjectsContext Create(string [] args)
  {
    //create and return a new ProjectsContext.
    return new ProjectsContext(/* … */);
  }
}
```

Chapter 3 Getting Data from the Database

Overview

As you might expect, Entity Framework offers a small number of APIs to get data from the database into objects, designed to cover slightly different scenarios.

By Id

All data access layers must support loading by primary key, and certainly EFCF does so. For that, we have the Find method:

Code Listing 53

```
//retrieving a record by a single primary key consisting of an integer.
var project = ctx.Projects.Find(1);
```

This method can take any number of parameters; it supports entities having composite primary keys. The following example shows just that:

Code Listing 54

```
//retrieving a record by a composite primary key consisting of two
integers.
var project = ctx.SomeEntities.Find(1, 2);
```

Find will return **NULL** if no matching record is found; no exception will be thrown, since this is a perfectly valid result.

Tip: The order and type of the parameters must match the order of primary key properties defined in the mapping for this entity.

LINQ

Since its introduction with .NET 3.5, Language Integrated Querying (LINQ) has become the *de facto* standard for querying data of any kind, so it's no surprise that Entity Framework has LINQ support. It will probably be your API of choice for most scenarios. It is strongly typed (meaning you can tell at compile time that some things are not right), refactor friendly, and its syntax is easy to understand. Let's see some examples.

A LINQ to Entities query is built from the entity collection properties of the DbSet<T> type exposed by the context, and it consists of an IQueryable<T> implementation.

Tip: Don't forget that LINQ is about querying. You won't find any way to change data here.

Execution of LINQ queries is deferred until **`GetEnumerator`** is invoked, which occurs in a **`foreach`** loop or when a terminal operator such as ToList, ToArray, ToDictionary, Any, Count, LongCount, Single, SingleOrDefault, First, FirstOrDefault, Last, or LastOrDefault is used. You can pick up this query and start adding restrictions such as paging or sorting.

Code Listing 55

```
//create a base query.
var projectsQuery = from p in ctx.Projects select p;

//add sorting.
var projectsSortedByDateQuery = projectsQuery.OrderBy(x => x.Start);

//execute and get the sorted results.
var projectsSortedByDateResults = projectsSortedByDateQuery.ToList();

//add paging and ordering (required for paging).
var projectsWithPagingQuery = projectsQuery
  .OrderBy(x => x.Start)
  .Take(5)
  .Skip(0);

//execute and get the first 5 results.
var projectsWithPagingResults = projectsWithPagingQuery.ToList();

//add a restriction
var projectsStartingAWeekAgoQuery = projectsQuery
       .Where(x => x.Start.Year >= DateTime.Today.Year);

//execute and get the projects that started a week ago.
var projectsStartingAWeekAgoResults = projectsStartingAWeekAgoQuery.ToList(
);
```

You will get at most a single result.

Code Listing 56

```
//retrieving at most a single record with a simple filter.
var project = ctx.Projects.SingleOrDefault(x => x.ProjectId == 1);
```

Restricting by several properties is just as easy:

Code Listing 57

```
//retrieving multiple record with two filters.
var projects = ctx.Projects
```

```
  .Where(x => x.Name.Contains("Something") && x.Start >= DateTime.Today)
  .ToList();
```

Or having one of two conditions matched:

Code Listing 58

```
//or
var resourcesKnowingVBOrCS = ctx.Technologies
  .Where(t => t.Name == "VB.NET" || t.Name == "C#")
  .SelectMany(x => x.Resources)
  .Select(x => x.Name)
  .ToList();
```

Count results:

Code Listing 59

```
//count
var numberOfClosedProjects = ctx.Projects
  .Where(x => x.End != null && x.End < DateTime.Now)
  .Count();
```

Check record existence:

Code Listing 60

```
//check existence
var existsProjectBySomeCustomer = ctx.Projects
  .Any(x => x.Customer.Name == "Some Customer");
```

Perform a projection (only get some parts of an entity):

Code Listing 61

```
//get only the name of the resource and the name of the associated project.
var resourcesXprojects = ctx.Projects
  .SelectMany(x => x.ProjectResources)

.Select(x => new { Resource = x.Resource.Name, Project = x.Project.Name })
  .ToList();
```

Do aggregations:

Code Listing 62

```
var avgResources = ctx.Projects.Average(p => p.ProjectResources.Count());
```

Get distinct values:

Code Listing 63

```
//distinct roles performed by a resource.
var roles = ctx.Resources
  .SelectMany(x => x.ProjectResources)
  .Where(x => x.Resource.Name == "Ricardo Peres")
  .Select(x => x.Role)
  .Distinct()
  .ToList();
```

Group on a property:

Code Listing 64

```
//grouping and projecting.
var resourcesGroupedByProjectRole = ctx.Projects
  .SelectMany(x => x.ProjectResources)
  .Select(x => new { Role = x.Role, Resource = x.Resource.Name })
  .GroupBy(x => x.Role)
  .Select(x => new { Role = x.Key, Resources = x })
  .ToList();

//grouping and counting.
var projectsByCustomer = ctx.Projects
  .GroupBy(x => x.Customer)
  .Select(x => new { Customer = x.Key.Name, Count = x.Count() })
  .ToList();

//top 10 customers having more projects in descending order.
var top10CustomersWithMoreProjects = ctx.Projects
  .GroupBy(x => x.Customer.Name)
  .Select(x => new { x.Key, Count = x.Count() })
  .OrderByDescending(x => x.Count)
  .Take(10)
  .ToList();
```

Use the results of a subquery:

Code Listing 65

```
//subquery
var usersKnowingATechnology = (from r in ctx.Resources where r.Technologies
.Any(x
=> (from t in ctx.Technologies where t.Name == "ASP.NET" select t).Contains
(x))
select r)
  .ToList();
```

Partial matches (**LIKE**):

Code Listing 66

```
//like
var aspTechnologies = (from t in ctx.Technologies
                       where EF.Functions.Like(t.Name, "asp%")
                       select t)
  .ToList();
```

Tip: Notice the static** Like **method in the** EF.Functions **class.

Finally, you can check for one of a set of values:

Code Listing 67

```
//contains
var customersToFind = new string[] { "Some Customer", "Another Customer" };
var projectsOfCustomers = ctx.Projects
  .Where(x => customersToFind.Contains(x.Customer.Name))
  .ToList();
```

Note: In case you are wondering, all literals present in LINQ queries (strings, numbers, dates, etc.) will be turned into parameters for proper execution plan reusing.

Tip: In EF Core 2.0, it is not possible to perform mathematical operations nor operations over dates and times.

Executing code on the client-side

Since EF Core 1.0, it has been possible to mix server and client-side code on your queries. For example, if the LINQ parser finds some method call that it does not know how to translate to a database call, it will execute it silently on the client-side. Depending on the call, it can either prove useful or result in a performance penalty. As an example of the latter, imagine the case where you are filtering by a client method: you need to bring all records, apply filtering to them, and only after return the results. But if used sparingly, it can indeed turn out useful; here's an example:

Code Listing 68

```
   var projects = from p in ctx.Projects select new { p.Name, Age =
   CalculateAge(p.StartDate) };
```

As a side note, if you want to disable this, effectively reverting to the old, pre-Core behavior, all you need to do is configure logging to throw an exception on the event of a client method call:

Code Listing 69

```
services.AddDbContext<ProjectsContext>(options =>
{
  options
    .UseSqlServer("<connection string>")
    .ConfigureWarnings(options =>
        options.Throw(RelationalEventId.QueryClientEvaluationWarning));
});
```

SQL

Try as we might, the truth is that when you're working with relational databases, it's impossible to escape SQL. This may be because performance is typically better or because some query is difficult or even impossible to express using any of the other APIs, but that's just the way it is. Entity Framework Code First (and Core) has full support for SQL, including:

- Getting entities and values.
- Executing **`INSERT`**, **`UPDATE`** and **`DELETE`**.
- Calling functions and stored procedures.

It does have all the disadvantages you might expect:

- It's not strongly typed.
- There is no compile-time checking.
- If you use database-specific functions and you target a new database, you must rewrite your SQL.
- You must know the right syntax for even simple things, such as paging or limiting the number of records to return.

The first case I'm demonstrating is how to execute a **`SELECT`** and convert the result into an entity. For that, we shall use the **`FromSql`** method of the DbSet<T>.

Code Listing 70

```
//simple select
var projectFromSQL = ctx.Projects
  .FromSql("SELECT * FROM Project WHERE Name = @p0", "Big Project")
  .SingleOrDefault();
```

Tip: Notice how we pass parameters directly after the SQL; each must be named `@p0`, `@p1`, and so on.

If we wanted to retrieve an entity from a table-valued function, we would use the following:

Code Listing 71

```
//table-valued function
```

```
var projectFromFunction = ctx.Projects
  .FromSql("SELECT * FROM dbo.GetProjectById @p0", 1)
  .SingleOrDefault();
```

Where the **GetProjectById** function might be something like this:

Code Listing 72

```
CREATE FUNCTION dbo.GetProjectById
(
      @ProjectID INT
)
RETURNS TABLE
AS
RETURN
(
      SELECT *
      FROM Project
      WHERE ProjectId = @ProjectID
)
GO
```

Tip: Don't forget that if you want to return entities, your SQL must return columns that match the properties of these entities, as specified in its mapping.

If we want to execute arbitrary SQL modification commands (**UPDATE**, **DELETE**, **INSERT**), we will need to retrieve the underlying DbConnection as shown in the following example.

Code Listing 73

```
//get the ADO.NET connection.
var con = ctx.Database.GetDbConnection();

//create a command.
var cmd = con.CreateCommand();
cmd.CommandText =
  "UPDATE ProjectDetail SET Budget = Budget * 1.1 WHERE ProjectId = @p0";

//create a parameter.
var parm = cmd.CreateParameter();
parm.ParameterName = "p0";
cmd.Parameters.Add(parm);

//update records.
var updatedRecords = cmd.ExecuteUpdate();
```

Using Database Functions

Also new in EF Core 2 is something that existed in pre-Core EF: the ability to execute database functions. Apply a **[DbFunction]** attribute to a static method, like this:

Code Listing 74

```
[DbFunction("ComputeHash")]
public static int ComputeHash(this string phrase)
{
  throw new NotImplementedException();
}
```

As you can see, it's implementation is irrelevant, as it will never be called by code. Four things to keep in mind:

- It can only receive scalars as parameters.
- It can only return a scalar type.
- It needs to be declared in the **DbContext** class.
- It needs to be static.

The name parameter to the **[DbFunction]** attribute is not necessary, if the database function has the same name as the method, but you can also add a schema name, if the function is not to be found under the default schema. Now you can use it like this:

Code Listing 75

```
   var hash = from p in ctx.Projects select p.Name.ComputeHash();
```

This will execute the database function on the database and return its result. You can also use it in other contexts, like for filtering, for example.

Mixing LINQ and SQL

A new feature coming in EF Core is the ability to mix LINQ with SQL; this allows you to get the best of both worlds:

- Execute arbitrarily complex SQL queries from inside EF.
- Have them materialize into .NET classes.
- Apply strongly typed LINQ queries on top.

Let's see an example:

Code Listing 76

```
var ps = ctx.Projects
  .FromSql("SELECT p.* FROM Project p")
  .OrderBy(p => p.Start)
  .ToList();
```

You can feed the **FromSql** method from any valid SQL, including stored procedures or functions. Notice that this method is prototyped to return instances of **Project**, so any queries to be executed need to return columns that can be mapped to **Projects**, per the mapping.

Eager loading

There will be cases in which you want all data from the main entity as well as associated entities to be loaded at the same time. This will most likely be for one of two reasons:

- You are certain that you are going to have to access some of the navigation properties and, for performance reasons, you load them beforehand (for example, you need to go through all order details).
- The entity's (and its associated entities') lifecycle will probably outlive the context from which it was obtained (for example, you are going to store the entity in some cache), so it won't have access to it, and thus lazy loading will not be possible.

Enter *eager loading*. What eager loading means is, when issuing a query, you explicitly declare the expansion paths that Entity Framework will bring along with the root entities. EF will then generate a different SQL expression than it would normally with using a **JOIN** for all the required associations.

For example, the following query brings along a **Customer** and all of its **Projects**, and it introduces the Include method.

Code Listing 77

```
//explicitly eager load the Customer for each project.
var projectsAndTheirCustomers = ctx.Projects
  .Include(x => x.Customer)
  .ToList();
```

For the record, this will produce the following SQL statement:

Code Listing 78

```
SELECT
[Extent1].[ProjectId] AS [ProjectId],
[Extent1].[Name] AS [Name],
[Extent1].[Start] AS [Start],
[Extent1].[End] AS [End],
[Extent2].[CustomerId] AS [CustomerId],
[Extent2].[Contact_Email] AS [Contact_Email],
```

```
[Extent2].[Contact_Phone] AS [Contact_Phone],
[Extent2].[Name] AS [Name1]
FROM  [dbo].[Project] AS [Extent1]
INNER JOIN [dbo].[Customer] AS [Extent2] ON
[Extent1].[Customer_CustomerId] = [Extent2].[CustomerId]
```

The Include method can also take a **`String`** as its parameter, which must be the name of a navigation property (a reference or a collection).

Code Listing 79

```
//explicitly eager load the Customer for each project.
var projectsAndTheirCustomers = ctx.Projects
  .Include("Customer")
  .ToList();
```

Multiple paths can be specified.

Code Listing 80

```
//two independent include paths.
var resourcesProjectResourcesAndTechnologies = ctx.Resources
  .Include(x => x.ProjectResources)
  .Include(x => x.Technologies)
  .ToList();
```

In this case, the SQL will look like the following:

Code Listing 81

```
SELECT
[UnionAll1].[ResourceId] AS [C1],
[UnionAll1].[ResourceId1] AS [C2],
[UnionAll1].[ResourceId2] AS [C3],
[UnionAll1].[Contact_Email] AS [C4],
[UnionAll1].[Contact_Phone] AS [C5],
[UnionAll1].[Name] AS [C6],
[UnionAll1].[C1] AS [C7],
[UnionAll1].[ProjectResourceId] AS [C8],
[UnionAll1].[ProjectResourceId1] AS [C9],
[UnionAll1].[Role] AS [C10],
[UnionAll1].[Project_ProjectId] AS [C11],
[UnionAll1].[Resource_ResourceId] AS [C12],
[UnionAll1].[C2] AS [C13],
[UnionAll1].[C3] AS [C14]
FROM  (SELECT
        CASE WHEN ([Extent2].[ProjectResourceId] IS NULL) THEN CAST(NULL AS
int)
 ELSE 1 END AS [C1],
        [Extent1].[ResourceId] AS [ResourceId],
        [Extent1].[ResourceId] AS [ResourceId1],
```

```
        [Extent1].[ResourceId] AS [ResourceId2],
        [Extent1].[Contact_Email] AS [Contact_Email],
        [Extent1].[Contact_Phone] AS [Contact_Phone],
        [Extent1].[Name] AS [Name],
        [Extent2].[ProjectResourceId] AS [ProjectResourceId],
        [Extent2].[ProjectResourceId] AS [ProjectResourceId1],
        [Extent2].[Role] AS [Role],
        [Extent2].[Project_ProjectId] AS [Project_ProjectId],
        [Extent2].[Resource_ResourceId] AS [Resource_ResourceId],
        CAST(NULL AS int) AS [C2],
        CAST(NULL AS varchar(1)) AS [C3]
        FROM  [dbo].[Resource] AS [Extent1]
        LEFT OUTER JOIN [dbo].[ProjectResource] AS [Extent2] ON
[Extent1].[ResourceId] = [Extent2].[Resource_ResourceId]
UNION ALL
        SELECT
        2 AS [C1],
        [Extent3].[ResourceId] AS [ResourceId],
        [Extent3].[ResourceId] AS [ResourceId1],
        [Extent3].[ResourceId] AS [ResourceId2],
        [Extent3].[Contact_Email] AS [Contact_Email],
        [Extent3].[Contact_Phone] AS [Contact_Phone],
        [Extent3].[Name] AS [Name],
        CAST(NULL AS int) AS [C2],
        CAST(NULL AS int) AS [C3],
        CAST(NULL AS int) AS [C4],
        CAST(NULL AS int) AS [C5],
        CAST(NULL AS int) AS [C6],
        [Join2].[TechnologyId] AS [TechnologyId],
        [Join2].[Name] AS [Name1]
        FROM  [dbo].[Resource] AS [Extent3]
        INNER JOIN  (SELECT [Extent4].[Resource_ResourceId] AS
[Resource_ResourceId], [Extent5].[TechnologyId] AS [TechnologyId],
[Extent5].[Name] AS [Name]
                FROM  [dbo].[TechnologyResource] AS [Extent4]
                INNER JOIN [dbo].[Technology] AS [Extent5] ON
[Extent5].[TechnologyId] = [Extent4].[Technology_TechnologyId] ) AS [Join2]
ON [Extent3].[ResourceId] = [Join2].[Resource_ResourceId]) AS [UnionAll1]
ORDER BY [UnionAll1].[ResourceId1] ASC, [UnionAll1].[C1] ASC
```

Here's a final example with multilevel inclusion:

Code Listing 82

```
//multilevel include paths.
var resourcesProjectResourcesCustomers = ctx
  .Resources
  .Include(x => x.ProjectResources.Select(y => y.Project.Customer))
  .ToList();
```

The generated SQL will look like this:

Code Listing 83

```
SELECT
[Project1].[ResourceId] AS [ResourceId],
[Project1].[Contact_Email] AS [Contact_Email],
[Project1].[Contact_Phone] AS [Contact_Phone],
[Project1].[Name] AS [Name],
[Project1].[C1] AS [C1],
[Project1].[ProjectResourceId] AS [ProjectResourceId],
[Project1].[Role] AS [Role],
[Project1].[ProjectId] AS [ProjectId],
[Project1].[Name1] AS [Name1],
[Project1].[Start] AS [Start],
[Project1].[End] AS [End],
[Project1].[CustomerId] AS [CustomerId],
[Project1].[Contact_Email1] AS [Contact_Email1],
[Project1].[Contact_Phone1] AS [Contact_Phone1],
[Project1].[Name2] AS [Name2],
[Project1].[Resource_ResourceId] AS [Resource_ResourceId]
FROM ( SELECT
      [Extent1].[ResourceId] AS [ResourceId],
      [Extent1].[Contact_Email] AS [Contact_Email],
      [Extent1].[Contact_Phone] AS [Contact_Phone],
      [Extent1].[Name] AS [Name],
      [Join2].[ProjectResourceId] AS [ProjectResourceId],
      [Join2].[Role] AS [Role],
      [Join2].[Resource_ResourceId] AS [Resource_ResourceId],
      [Join2].[ProjectId] AS [ProjectId],
      [Join2].[Name1] AS [Name1],
      [Join2].[Start] AS [Start],
      [Join2].[End] AS [End],
      [Join2].[CustomerId] AS [CustomerId],
      [Join2].[Contact_Email] AS [Contact_Email1],
      [Join2].[Contact_Phone] AS [Contact_Phone1],
      [Join2].[Name2] AS [Name2],
      CASE WHEN ([Join2].[ProjectResourceId] IS NULL) THEN CAST(NULL AS
int) ELSE 1 END AS [C1]
      FROM  [dbo].[Resource] AS [Extent1]
      LEFT OUTER JOIN  (SELECT [Extent2].[ProjectResourceId] AS
[ProjectResourceId], [Extent2].[Role] AS [Role],
[Extent2].[Resource_ResourceId] AS [Resource_ResourceId],
[Extent3].[ProjectId] AS [ProjectId], [Extent3].[Name] AS [Name1],
[Extent3].[Start] AS [Start], [Extent3].[End] AS [End],
[Extent4].[CustomerId] AS [CustomerId], [Extent4].[Contact_Email] AS
[Contact_Email], [Extent4].[Contact_Phone] AS [Contact_Phone],
[Extent4].[Name] AS [Name2]
            FROM   [dbo].[ProjectResource] AS [Extent2]
```

```
            INNER JOIN [dbo].[Project] AS [Extent3] ON
[Extent2].[Project_ProjectId] = [Extent3].[ProjectId]
            INNER JOIN [dbo].[Customer] AS [Extent4] ON
[Extent3].[Customer_CustomerId] = [Extent4].[CustomerId] ) AS [Join2] ON
[Extent1].[ResourceId] = [Join2].[Resource_ResourceId]
)  AS [Project1]
ORDER BY [Project1].[ResourceId] ASC, [Project1].[C1] ASC
```

As you can imagine, EF goes through a lot of work to **JOIN** all data that it needs to fetch at the same time, hence the quite complicated SQL.

Multiple levels

Here's a final example with multilevel inclusion using **ThenInclude**:

Code Listing 84

```
//multilevel include paths
var resourcesProjectResourcesCustomers = ctx
  .Resources
  .Include(x => x.ProjectResources)
  .ThenInclude(x => x.Role)
  .ToList();
```

ThenInclude can be used only following an **Include** call to force loading a nested path on the included one.

As you can imagine, EF goes to a lot of work to **JOIN** all data that it needs to fetch at the same time, hence the quite complicated SQL.

Explicit loading

In a case where a reference property was not eagerly loaded, we can still force it to load explicitly:

Code Listing 85

```
//explicitly load the Customer property.
ctx.Entry(project).Reference(x => x.Customer).Load();
```

The same also applies to collections:

Code Listing 86

```
//see if the ProjectResources collection is loaded.
var resourcesLoaded = ctx
  .Entry(project)
  .Collection(x => x.ProjectResources)
```

```
  .IsLoaded;

if (resourcesLoaded == false)
{
  //explicitly load the ProjectResources collection.
  ctx.Entry(project).Collection(x => x.ProjectResources).Load();
}
```

Another interesting case is where you want to load just a part of some collection by filtering out the entities that do not match a given condition, or even count its members without actually loading them. It is possible to do it with EF, and for this you would issue queries like this:

Code Listing 87

```
//count an entity's collection entities without loading them.
var countDevelopersInProject = ctx
  .Entry(project)
  .Collection(x => x.ProjectResources)
  .Query()
  .Where(x => x.Role == Role.Developer)
  .Count();

//filter an entity's collection without loading it.
var developersInProject = ctx
  .Entry(project)
  .Collection(x => x.ProjectResources)
  .Query()
  .Where(x => x.Role == Role.Developer)
  .ToList();
```

And you can also force loading and bring along related references or collections—notice the call to **`Include`**:

Code Listing 88

```
//filter an entity's collection without loading it.
var developersInProject = ctx
  .Entry(project)
  .Collection(x => x.ProjectResources)
  .Query()
  .Include(x => x.Resource)
  .ToList();
```

So, the difference between lazy and eager loading is that with lazy loading, you don't need to do anything explicit—you just access the navigation properties without even thinking about it, whereas with explicit loading, you have to perform some action.

Local data

Entities known by an Entity Framework context—either loaded from it or marked for deletion or insertion— are stored in what is called a local or first-level cache. Martin Fowler calls it the Identity Map, and you can read more about the concept here. Basically, the context keeps track of all these entities, so that it doesn't need to materialize them whenever a query that returns their associated records is executed. It is possible to access this cache by means of the **`ChangeTracker`** instance. Two extension methods make it easier:

Code Listing 89

```
public static class DbContextExtensions
{

public static IEnumerable<EntityEntry<T>> Local<T>(this DbContext context)
    where T : class
  {
    return context.ChangeTracker.Entries<T>();
  }

  public static IEnumerable<EntityEntry<T>> Local<T>(this DbSet<T> set)
    where T : class
  {
    if (set is InternalDbSet<T>)
    {
      var svcs = (set as InternalDbSet<T>)
        .GetInfrastructure()
        .GetService<IDbContextServices>();

      var ctx = svcs.CurrentContext.Context;

      return Local<T>(ctx);
    }

    throw new ArgumentException("Invalid set", "set");
  }
}

//local Projects.
var projectsAlreadyLoaded = ctx.Projects.Local();

//filtered local Projects - no need to call ToList.
var projectsAlreadyLoadedBelongingToACustomer = projectsAlreadyLoaded
  .Where(x => x.Customer.Name == "Some Customer");
```

It is possible to know all entities that are present in the local cache, and to see their state, as seen by the context. It's the responsibility of the **`ChangeTracker`** to keep track of all these entities.

Code Listing 90

```
//get the projects in local cache that have been modified.
var modifiedProjects = ctx.ChangeTracker
  .Entries<Project>()
  .Where(x => x.State == EntityState.Modified)
  .Select(x => x.Entity);
```

As you can guess, it's considerably faster to get an entity from the local cache than it is to load it from the database. With that in mind, we can write a method like the following one, for transparently returning a local entity or fetching it with SQL.

Code Listing 91

```
//retrieve from cache or the database.
public static IQueryable<T> LocalOrDatabase<T>(this DbContext context,
      Expression<Func<T, bool>> expression) where T : class
{
  var localResults = context
    .Set<T>()
    .Local()
    .Where(expression.Compile());

    if (localResults.Any() == true)
    {
      return localResults.AsQueryable();
    }

  return context.Set<T>().Where(expression);
}
```

Implementing LINQ extension methods

Another useful technique consists of leveraging LINQ expressions to build complex queries from extension methods.

The BETWEEN SQL operator does not have a corresponding LINQ expression. We can use two simultaneous conditions in our LINQ expression, one for the low end of the range (> X), and one for the high end (< Y). We can also implement a LINQ extension method to provide us this functionality with a single expression.

Code Listing 92

```
public static class QueryableExtensions
{
  public static IQueryable<TSource> Between<TSource, TKey>(
    this IQueryable<TSource> source,
    Expression<Func<TSource, TKey>> property, TKey low, TKey high
  ) where TKey : IComparable<TKey>
```

```
  {
    var sourceParameter = Expression.Parameter(typeof(TSource));
    var body = property.Body;
    var parameter = property.Parameters[0];
    var compareMethod = typeof(TKey).GetMethod("CompareTo",
      new Type[] { typeof(TKey) });
    var zero = Expression.Constant(0, typeof(int));

var upper = Expression.LessThanOrEqual(Expression.Call(body, compareMethod,
      Expression.Constant(high)), zero);

var lower = Expression.GreaterThanOrEqual(Expression.Call(body, compareMeth
od,
      Expression.Constant(low)), zero);
    var andExpression = Expression.AndAlso(upper, lower);
    var whereCallExpression = Expression.Call
    (
      typeof(Queryable),
      "Where",
      new Type[] { source.ElementType },
      source.Expression,
      Expression.Lambda<Func<TSource, Boolean>>(andExpression,
        new ParameterExpression[] { parameter })
    );

    return source.Provider.CreateQuery<TSource>(whereCallExpression);
  }
}
```

For a good understanding of how this is implemented, it is crucial to understand LINQ expressions. There are some good links on the Internet. This technology, although complex to master, has great potential and has drawn a lot of attention.

This is an extension method on IQueryable<T>, and it can be used like this:

Code Listing 93

```
//get projects starting between two dates.
var projectsBetweenTodayAndTheDayBefore = ctx
  .Projects
  .Between(x => x.Start, DateTime.Today.AddDays(-1), DateTime.Today)
  .ToList();

//projects with 10 to 20 resources.
var projectsWithTwoOrThreeResources = ctx
  .Projects
  .Select(x => new { x.Name, ResourceCount = x.ProjectResources.Count() })
  .Between(x => x.ResourceCount, 10, 20)
  .ToList();
```

The LINQ provider will happily chew the new expression and translate it into the appropriate SQL.

Both DbContext and DbSet<T> implement the **`IInfrastructure<IServiceProvider>`** interface. This method exposes the internal service provider, and from it you can obtain references to all services used by Entity Framework Core.

Chapter 4 Writing Data to the Database

Saving, updating, and deleting entities

Saving entities

Because EF works with POCOs, creating a new entity is just a matter of instantiating it with the new operator. If we want it to eventually get to the database, we need to attach it to an existing context.

Code Listing 94

```
var developmentTool = new DevelopmentTool() { Name = "Visual Studio 2017",
Language = "C#" };

ctx.Tools.Add(developmentTool);
```

New entities must be added to the DbSet<T> property of the same type, which is also your gateway for querying. Another option is to add a batch of new entities, maybe of different types, to the DbContext itself:

Code Listing 95

```
DevelopmentTool tool = /* something */;
Project project = /* something */;
Customer customer = /* something */;

ctx.AddRange(tool, project, customer);
```

However, this new entity is not immediately sent to the database. The EF context implements the Unit of Work pattern, a term coined by Martin Fowler, about which you can read more here. In a nutshell, this pattern states that the Unit of Work container will keep internally a list of items in need of persistence (new, modified, or deleted entities), and will save them all in an atomic manner, taking care of any eventual dependencies between them. The moment when these entities are persisted in Entity Framework Code First happens when we call the DbContext's SaveChanges method.

Code Listing 96

```
var affectedRecords = ctx.SaveChanges();
```

At this moment, all the pending changes are sent to the database. Entity Framework employs a first-level (or local) cache, which is where all the "dirty" entities—like those added to the context—sit waiting for the time to persist. The SaveChanges method returns the number of records that were successfully inserted, and will throw an exception if some error occurred in the process. In that case, all changes are rolled back, and you really should take this scenario into consideration.

Updating entities

As for updates, Entity Framework tracks changes to loaded entities automatically. For each entity, it knows what their initial values were, and if they differ from the current ones, the entity is considered “dirty.” A sample code follows.

Code Listing 97

```
//load some entity.
var tool = ctx.Tools.FirstOrDefault();

ctx.SaveChanges(); //0

//change something.
tool.Name += "_changed";

//send changes.
var affectedRecords = ctx.SaveChanges(); //1
```

As you can see, no separate update method is necessary, nor does it exist, since all types of changes (inserts, updates, and deletes) are detected automatically and performed by the SaveChanges method. SaveChanges still needs to be called, and it will return the combined count of all inserted and updated entities. If some sort of integrity constraint is violated, then an exception will be thrown, and this needs to be dealt with appropriately.

Deleting entities

When you have a reference to a loaded entity, you can mark it as deleted, so that when changes are persisted, EF will delete the corresponding record. Deleting an entity in EF consists of removing it from the DbSet<T> collection.

Code Listing 98

```
//load some entity.
var tool = ctx.Tools.FirstOrDefault();

//remove the entity.
ctx.Tools.Remove(tool);

//send changes.
var affectedRecords = ctx.SaveChanges(); //1
```

Of course, SaveChanges still needs to be called to make the changes permanent. If any integrity constraint is violated, an exception will be thrown.

Note: Entity Framework will apply all the pending changes (inserts, updates, and deletes) in an appropriate order, including entities that depend on other entities.

Inspecting the tracked entities

When we talked about the local cache, you may have asked yourself where this cache is—and what can be done with it.

You access the local cache entry for an entity with the **`Entry`** method. This returns an instance of **`EntityEntry`**, which contains lots of useful information, such as the current state of the entity (as seen by the context), the initial and current values, and so on.

Code Listing 99

```
//load some entity.
var tool = ctx.Tools.FirstOrDefault();

//get the cache entry.
var entry = ctx.Entry(tool);

//get the entity state.
var state = entry.State; //EntityState.Unchanged

//get the original value of the Name property.
var originalName = entry.OriginalValues["Name"] as String; //Visual Studio
2017

//change something.
tool.Name += "_changed";

//get the current state
state = entry.State; //EntityState.Modified

//get the current value of the Name property.
var currentName = entry.CurrentValues["Name"] as String; //Visual Studio
2017_changed
```

If you want to inspect all the entries currently being tracked, there is the ChangeTracker property.

Code Listing 100

```
//get all the added entities of type Project.
var addedProjects = ctx
  .ChangeTracker
  .Entries()
  .Where(x => x.State == EntityState.Added)
  .Select(x => x.Entity)
  .OfType<Project>();
```

Using SQL to make changes

Sometimes you need to do bulk modifications, and, in this case, nothing beats good old SQL. EF Core offers the **`ExecuteSqlCommand`** in the Database property that you can use just for that:

Code Listing 101

```
var rows = ctx.Database.ExecuteSqlCommand($"DELETE FROM Project WHERE
ProjectId = {id}");
```

As you can see, you can even use interpolated strings, EF Core will translate them to safe parameterized strings.

Firing events when an entity's state changes

There's a special infrastructure interface called `ILocalViewListener` that is registered in Entity Framework and is called whenever the state of an entity changes (like when it's about to be saved, deleted, updated, etc.). We use it like this:

Code Listing 102

```
var events = ctx.GetService<ILocalViewListener>();
events.RegisterView((entry, state) =>
{
  //entry contains the entity's details.
  //state is the new state.
});
```

Cascading deletes

Two related tables can be created in the database in such a way that when one record of the parent table is deleted, all corresponding records in the child table are also deleted. This is called cascading deletes.

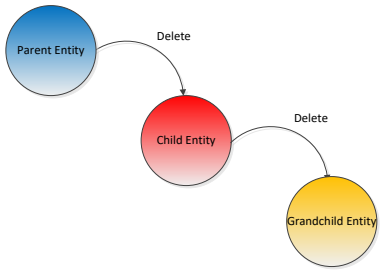

Figure 17: Cascading deletes

This is useful for automatically keeping the database integrity. If the database didn't do this for us, we would have to do it manually; otherwise, we would end up with orphan records. This is only useful for parent-child or master-detail relationships where one endpoint cannot exist without the other. Not all relationships should be created this way; for example, when the parent endpoint is optional, we typically won't cascade. Think of a customer-project relationship: it doesn't make sense to have projects without a customer. On the other hand, it does make sense to have a bug without an assigned developer.

When Entity Framework creates the database, it will create the appropriate cascading constraints depending on the mapping.

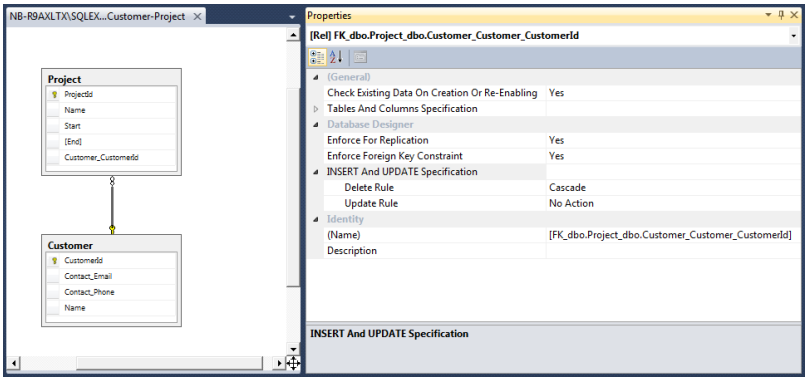

Figure 18: A cascade delete

As of now, EF applies a convention for that, but it can be overridden by fluent mapping.

Table 2

Relationship	Default Cascade
One-to-one	No
One-to-many	Only if the one endpoint is required
Many-to-one	No

We can explicitly configure the cascading option in fluent configuration like this:

Code Listing 103

```
//when deleting a Project, delete its ProjectDetail.
builder
  .Entity<Project>()
  .HasOne(b => b.Detail)
  .WithOne(d => d.Project)
  .OnDelete(DeleteBehavior.Cascade);
```

```
//when deleting a ProjectResource, do not delete the Project.
builder
  .Entity<ProjectResource>()
  .HasOne(x => x.Project)
  .WithMany(x => x.ProjectResources)
  .OnDelete(DeleteBehavior.SetNull);

//when deleting a Project, delete its ProjectResources.
builder
  .Entity<Project>()
  .HasMany(x => x.ProjectResources)
  .WithOne(x => x.Project)
  .OnDelete(DeleteBehavior.Cascade);
```

Tip: You can have multiple levels of cascades, just make sure you don't have circular references.

Note: Cascade deletes occur at the database level; Entity Framework does not issue any SQL for that purpose.

Refreshing entities

When a record is loaded by EF as the result of a query, an entity is created and placed in local cache. When a new query is executed that returns records associated with an entity already in local cache, no new entity is created; instead, the one from the cache is returned. This has an occasionally undesirable side effect: even if something changed in the entity's record, the local entity is not updated. This is an optimization that Entity Framework performs, but sometimes it can lead to unexpected results. If we want to make sure we have the latest data, we need to force an explicit refresh, first, removing the entity from the local cache:

Code Listing 104

```
//load some entity.
var project = ctx.Projects.Find(1);

//set it to detached.
ctx.Entry(project).State = EntityState.Detached;

//time passes…

//load entity again.
project = ctx.Projects.Find(1);
```

Entity Framework Core allows us to refresh it:

Code Listing 105

```
ctx.Entry(project).Reload();
```

Even for just a specific property:

Code Listing 106

```
ctx
  .Entry(project)
  .Property(x => x.Name)
  .EntityEntry
  .Reload();
```

Or collection:

Code Listing 107

```
ctx
  .Entry(customer)
  .Collection(x => x.Projects)
  .EntityEntry
  .Reload();
```

Concurrency control

Optimistic concurrency control is a method for working with databases that assumes multiple transactions can complete without affecting each other; no locking is required. Each update transaction will check to see if any records have been modified in the database since they were read, and if so, will fail. This is very useful for dealing with multiple accesses to data in the context of web applications.

There are two ways for dealing with the situation where data has been changed:

- First one wins: The second transaction will detect that data has been changed, and will throw an exception.
- Last one wins: While it detects that data has changed, the second transaction chooses to overwrite it.

Entity Framework supports both of these methods.

First one wins

We have an entity instance obtained from the database, we change it, and we tell the EF context to persist it. Because of optimistic concurrency, the SaveChanges method will throw a DbUpdateConcurrencyException if the data was changed, so make sure you wrap it in a **`try…catch`**.

Code Listing 108

```
try
{
  ctx.SaveChanges();
}
catch (DbUpdateConcurrencyException)
{
  //the record was changed in the database, notify the user and fail.
}
```

The “first one wins” approach is just this: fail if a change has occurred.

Last one wins

For this one, we will detect that a change has been made, and we’ll overwrite it explicitly. However, in Entity Framework Core, we cannot do this automatically in an easy way.

Applying optimistic concurrency

Entity Framework by default does not perform the optimistic concurrency check. You can enable it by choosing the property or properties whose values will be compared with the current database values. This is done by applying a ConcurrencyCheckAttribute when mapping by attributes.

Code Listing 109

```
public class Project
{
  [Concurrency][ConcurrencyCheck]
  public DateTime Timestamp { get; set; }
}
```

Or in mapping by code.

Code Listing 110

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder
    .Entity<Project>()
    .Property(x => x.Name)
    .IsConcurrencyToken();
}
```

What happens is this: when EF generates the SQL for an **UPDATE** operation, it will not only include a **WHERE** restriction for the primary key, but also for any properties marked for concurrency check, comparing their columns with the original values.

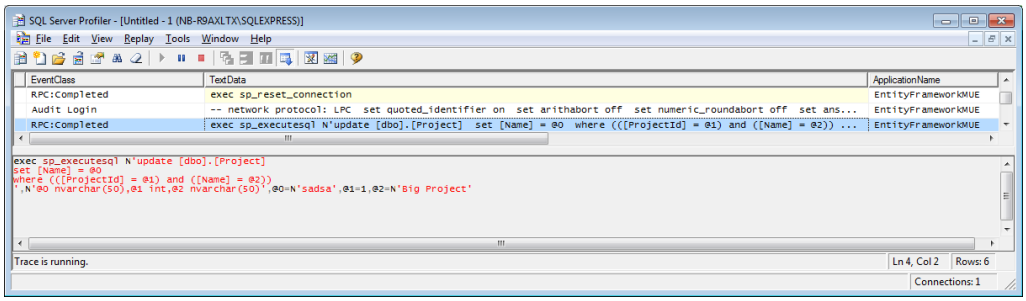

Figure 19: An update with a concurrency control check

If the number of affected records is not **1**, this will likely be because the values in the database will not match the original values known by Entity Framework, because they have been modified by a third party outside of Entity Framework.

SQL Server has a data type whose values cannot be explicitly set, but instead change automatically whenever the record they belong to changes: ROWVERSION. Other databases offer similar functionality.

Because Entity Framework has a nice integration with SQL Server, columns of type ROWVERSION are supported for optimistic concurrency checks. For that, we need to map one such column into our model as a timestamp. First, we'll do so with attributes by applying a TimestampAttribute to a property, which needs to be of type **`byte array`**, and doesn't need a public setter.

Code Listing 111

```
[Timestamp]
public byte [] RowVersion { get; protected set; }
```

And, for completeness, with fluent configuration.

Code Listing 112

```
modelBuilder
  .Entity<Project>()
  .Property(x => x.RowVersion)
  .IsRowVersion();
```

The behavior of TimestampAttribute is exactly identical to that of ConcurrencyCheckAttribute, but there can be only one property marked as timestamp per entity, and ConcurrencyCheckAttribute is not tied to a specific database.

Detached entities

A common scenario in web applications is this: you load some entity from the database, store it in the session, and in a subsequent request, get it from there and resume using it. This is all fine except that if you are using Entity Framework Core, you won't be using the same context instance on the two requests. This new context knows nothing about this instance. In this case, it is said that the entity is detached in relation to this context. The effect is that any changes to this instance won't be tracked, and any lazy loaded properties that weren't loaded when it was stored in the session won't be loaded.

What we need to do is associate this instance with the new context.

Code Listing 113

```
//retrieve the instance from the ASP.NET context.
var project = Session["StoredProject"] as Project;

var ctx = new ProjectsContext();

//attach it to the current context with a state of unchanged.
ctx.Entry(project).State = EntityState.Unchanged;
```

After this, everything will work as expected.

If, however, we need to attach a graph of entities in which some may be new and others modified, we can use the Graph API introduced in Entity Framework Core. For this scenario, Entity Framework lets you traverse through all of an entity's associated entities and set each's state individually. This is done using the **`TrackGraph`** method of the **`DbContext`**'s **`ChangeTracker`** member:

Code Listing 114

```
ctx.ChangeTracker.TrackGraph(rootEntity, node =>
{
  if (node.Entry.Entity is Project)
  {
    if ((node.Entry.Entity as Project).ProjectId != 0)
    {
      node.Entry.State = EntityState.Unchanged;
    }
    else
    {
      //other conditions
    }
  }
);
```

Validation

Unlike previous versions, Entity Framework Core does not perform validation of the tracked entities when the SaveChanges method is called. Entity Framework pre-Core used to validate entities with the Data Annotations API. Fortunately, it is easy to bring this behavior back if we override SaveChanges and plug in our validation algorithm—in this case, Data Annotations validation:

Code Listing 115

```
public override int SaveChanges()
{
  var serviceProvider = GetService<IServiceProvider>();
  var items = new Dictionary<object, object>();

  foreach (var entry in ChangeTracker.Entries().Where(e =>

(e.State == EntityState.Added) || (e.State == EntityState.Modified))
  {
    var entity = entry.Entity;
    var context = new ValidationContext(entity, serviceProvider, items);
    var results = new List<ValidationResult>();

    if (!Validator.TryValidateObject(entity, context, results, true))
    {
      foreach (var result in results)
      {
        if (result != ValidationResult.Success)
        {
          throw new ValidationException(result.ErrorMessage);
        }
      }
    }
  }

  return base.SaveChanges();
}
```

In order for the **GetService** extension method to be recognized, you need to add a namespace reference to **Microsoft.EntityFrameworkCore.Infrastructure**. This is possible because **DbContext** implements **IInfrastructure<IServiceProvider>**, which exposes the internal service provider.

In order to validate entities before committing them to the database, we use the change tracker to loop through each entity that has been added or updated in the context. Then we use **Validator.TryValidateObject** to check for validation errors. If we find a validation error, we throw a **ValidationException**.

Because we call the base SaveChanges method, if there are no validation errors, everything works as expected.

A validation result consists of instances of DbEntityValidationResult, of which there will be only one per invalid entity. This class offers the following properties.

Table 3: Validation result properties

Property	Purpose
Entry	The entity to which this validation refers.
IsValid	Indicates whether the entity is valid or not.
ValidationErrors	A collection of individual errors.

The ValidationErrors property contains a collection of DbValidationError entries, each exposing the following.

Table 4: Result error properties

Property	Purpose
ErrorMessage	The error message.
PropertyName	The name of the property on the entity that was considered invalid (can be empty if what was considered invalid was the entity as a whole).

If we attempt to save an entity with invalid values, a DbEntityValidationException will be thrown, and inside of it, there is the EntityValidationErrors collection, which exposes all DbEntityValidationResult found.

Code Listing 116

```
try
{
  //try to save all changes.
  ctx.SaveChanges();
}
catch (DbEntityValidationException ex)
{
  //validation errors were found that prevented saving changes.
  var errors = ex.EntityValidationErrors.ToList();
}
```

Validation attributes

Similar to the way we can use attributes to declare mapping options, we can also use attributes for declaring validation rules. A validation attribute must inherit from ValidationAttribute in the System.ComponentModel.DataAnnotations namespace and override one of its IsValid methods. There are some simple validation attributes we can use out of the box that are in no way tied to Entity Framework.

Table 5: Validation attributes

Validation Attribute	Purpose
CompareAttribute	Compares two properties and fails if they are different.
CustomValidationAttribute	Executes a custom validation function and returns its value.
MaxLengthAttribute	Checks whether a string property has a length greater than a given value.
MinLengthAttribute	Checks whether a string property has a length smaller than a given value.
RangeAttribute	Checks whether the property's value is included in a given range.
RegularExpressionAttribute	Checks whether a string matches a given regular expression.
RequiredAttribute	Checks whether a property has a value; if the property is of type string, also checks whether it is empty.
StringLengthAttribute	Checks whether a string property's length is contained within a given threshold.
MembershipPasswordAttribute	Checks whether a string property (typically a password) matches the requirements of the default Membership Provider.

It is easy to implement a custom validation attribute. Here we can see a simple example that checks whether a number is even.

Code Listing 117

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false,
        Inherited = true)]
public sealed class IsEvenAttribute : ValidationAttribute
{
  protected override ValidationResult IsValid(object value,
      ValidationContext validationContext)
  {
    //check if the value is null or empty.
    if ((value != null) && (!string.IsNullOrWhiteSpace(value.ToString())))
    {
      //check if the value can be converted to a long one.
      var number = Convert.ToDouble(value);
      //fail if the number is even.
      if ((number % 2) == 0)
      {
        return new ValidationResult(ErrorMessage, new []
          { validationContext.MemberName });
      }
    }
    return ValidationResult.Success;
  }
}
```

It can be applied to any property whose type can be converted to a long integer—it probably doesn't make sense in the case of a budget, but let's pretend it does.

Code Listing 118

```
[IsEven(ErrorMessage = "Number must be even")]
public int Number { get; set; }
```

We can also supply a custom validation method by applying a CustomValidationAttribute. Let's see how the same validation ("is even") can be implemented using this technique. First, use the following attribute declaration.

Code Listing 119

```
[CustomValidation(typeof(CustomValidationRules), "IsEven",
  ErrorMessage = "Number must be even")]
public int Number { get; set; }
```

Next, use the following actual validation rule implementation.

Code Listing 120

```
public static ValidationResult IsEven(Object value,
ValidationContext context)
{
  //check if the value is not null or empty.
```

```
  if ((value != null) && (!string.IsNullOrWhiteSpace(value.ToString())))
  {
    //check if the value can be converted to a long one.
    var number = Convert.ToDouble(value);
    //fail if the number is even.
    if ((number % 2) == 0)
    {
      return new ValidationResult(ErrorMessage, new []
        { validationContext.MemberName });
    }
    return ValidationResult.Success;
  }
}
```

I chose to implement the validation function as static, but it is not required. In that case, the class where the function is declared must be safe to instantiate (not abstract with a public parameterless constructor).

Implementing self-validation

Another option for performing custom validations lies in the IValidatableObject interface. By implementing this interface, an entity can be self-validatable; that is, all validation logic is contained within itself. Let's see how.

Code Listing 121

```
public class Project : IValidatableObject
{
  //other members go here.
  public IEnumerable<ValidationResult> Validate(ValidationContext context)
  {
    if (ProjectManager == null)
    {
      yield return new ValidationResult("No project manager specified");
    }
    if (Developers.Any() == false)
    {
      yield return new ValidationResult("No developers specified");
    }
    if ((End != null) && (End.Value < Start))
    {
      yield return new ValidationResult("End of project is before start");
    }
  }
}
```

Wrapping up

You might have noticed that all these custom validation techniques—custom attributes, custom validation functions, and IValidatableObject implementation—all return ValidationResult instances, whereas Entity Framework Code First exposes validation results as collections of DbEntityValidationResult and DbValidationError. Don't worry: Entity Framework will take care of it for you!

So, which validation option is best? In my opinion, all have strong points, and all can be used together. I'll just leave some final remarks:

- If a validation attribute is sufficiently generic, it can be reused in many places.
- When we look at a class that uses attributes to express validation concerns, it is easy to see what we want.
- It also makes sense to have general purpose validation functions available as static methods, which may be invoked from either a validation attribute or otherwise.
- Finally, a class can self-validate in ways that are hard or even impossible to express using attributes. For example, think of properties whose values depend on other properties' values.

Tip: Keep in mind that this only works because we explicitly called Validator.TryValidateObject, since EF Core no longer automatically performs validation.

Transactions

Transactions in Entity Framework Core come in three flavors:

- Implicit: The method SaveChanges creates a transaction for wrapping all change sets that it will send to the database, if no ambient transaction exists. This is necessary for properly implementing a Unit of Work, where either all or no changes are applied simultaneously.
- Explicit: We need to start the transaction explicitly ourselves, and then either commit or "roll it back."
- External: A transaction was started outside of Entity Framework, yet we want it to use it.

You should use transactions primarily for two reasons:

- For maintaining consistency when performing operations that must be simultaneously successful (think of a bank transfer where money leaving an account must enter another account).
- For assuring identical results for read operations where data can be simultaneously accessed and possibly changed by third parties.

In order to start an explicit transaction on a relational database, you need to call **`BeginTransaction`** or **`BeginTransactionAsync`** on the **`Database`** property of the context:

Code Listing 122

```
ctx.Database.BeginTransaction();
```

This actually returns a **IDbContextTransaction** object, which wraps the ADO.NET transaction object (**DbTransaction**).

Likewise, you either commit or roll it back using methods in **Database**:

Code Listing 123

```
if (/*some condition*/)
{
  ctx.Database.CommitTransaction();
}
else
{
  ctx.Database.RollbackTransaction();
}
```

Calling **RollbackTransaction** explicitly is redundant if you just dispose of the transaction returned by **BeginTransaction** without actually calling **CommitTransaction**.

If, on the other hand, you have a transaction started elsewhere, you need to pass the DbTransaction instance to the **UseTransaction** method of **IRelationalTransactionManager**:

Code Listing 124

```
ctx.GetService<IRelationalTransactionManager>().UseTransaction(tx);
```

Or, pass it to the underlying DbConnection instance:

Code Listing 125

```
var con = ctx.Database.GetDbConnection();
con.Transaction = tx;
```

Connection resiliency

Another thing you need to be aware of is that connections may be dropped, and commands may fail due to broken connections. This will happen not only in cloud scenarios, but also elsewhere. You need to be aware of this possibility and program defensively.

Entity Framework Core—as, to some degree, its predecessor—offers something called connection resiliency. In a nutshell, it is a mechanism by which EF will retry a failed operation a number of times, with some interval between them, until it either succeeds or fails. This only applies to what the provider considers transient errors.

This needs to go in two stages:

1. We first need to configure it in the provider-specific configuration code:

Code Listing 126

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
  optionsBuilder
    .UseSqlServer(
      connectionString: _nameOrConnectionString,
      sqlServerOptionsAction: opts =>
      {
        opts.EnableRetryOnFailure(3, TimeSpan.FromSeconds(3), new int[]
{});
      });
    );

  base.OnConfiguring(optionsBuilder);
}
```

2. Then we need to register all the operations that we want to retry:

Code Listing 127

```
var strategy = ctx.Database.CreateExecutionStrategy();
strategy.Execute(() =>
{
  using (var tx = ctx.Database.BeginTransaction())
  {
    tx.Projects.Add(new Project { Name = "Big Project",
Customer = new Customer { CustomerId = 1 }, Start = DateTime.UtcNow });
    tx.Commit();
  }
});
```

This tells EF to retry this operation up to three times with an interval of three seconds between each. The last parameter to **`EnableRetryOnFailure`** is an optional list of provider-specific error codes that are to be treated as transient errors. Instead of executing an operation immediately, we go through the **`strategy`** object.

Chapter 5 Logging

Entity Framework Core makes use of the logging infrastructure of .NET Core. This can come in handy because it offers a glimpse (or even more) into what Entity Framework is doing, like the SQL it is generating, how long the operations take, etc.

The logging framework of .NET Core—used by Entity Framework Core—consists of the following:

- **Logger factories**: The top-level registration point. Normally, we can just use the built-in **`LoggerFactory`** and add logger providers to it.
- **Logger providers**: These are implementations of **`ILoggerProvider`** that are registered to the .NET Core-provided **`ILoggerFactory`**.
- **Actual loggers**: These are **`ILogger`** implementations returned by a logger provider, and do the actual logging.

A logger is returned for a concrete logger category. Logger categories come from infrastructure class names, the ones that provide the logging information. Because Entity Framework Core is built using a modular, layered approach, some of its operations will come from the datastore-agnostic core (DbContext, **`InternalQueryCompiler`**). Others will come from the relational layers (**`RelationalCommandBuilderFactory`**, **`RelationalModelValidator`**, **`SqlServerQueryCompilationContextFactory`**, **`SqlServerCompositeMethodCallTranslator`**), and finally, others will come from database-specific classes (**`SqlServerConnection`**).

Each logging entry actually consists of:

- **Log Level**: The severity of the log entry, such as **`Critical`**, **`Debug`**, **`Error`**, **`Information`**, or **`Trace`**.
- **Event ID**: A provider-specific code that represents the type of event being logged (more on this in a second).
- **State**: An optional contextual object to pass more information to the logger.
- **Exception**: An exception, to use in the case of an error (normally for the **`Critical`** or **`Error`**) log levels.
- **Formatter**: An optional formatter object to help format the log output, in cases where it is necessary.

The event ID is specific to the infrastructure. Some common values are as follows:

Table 6

ID	Meaning	Sender	State
1	Execute SQL	**`RelationalCommandBuilderFactory`**	**`DbCommandLogData`**

ID	Meaning	Sender	State
2	Create database	**SqlServerConnection**	Database and server names as strings
3	Open connection	**SqlServerConnection**	Database and server names as strings
4	Close connection	**SqlServerConnection**	Database and server names as strings
5	Begin transaction	**SqlServerConnection**	**IsolationLevel**
6	Commit transaction	**SqlServerConnection**	**IsolationLevel**
7	Rollback transaction	**SqlServerConnection**	**IsolationLevel**
> 7	Warnings		

In the case of relational data sources, these values are specified in the **RelationalEventId** enumeration. There is also **CoreEventId** for EF Core generic events (context initialized, etc.) and **SqlServerRelationalId** (for SQL Server-specific events, of course). Other databases will feature similar constants.

When we add logging to an Entity Framework context through the **OnConfiguring** method, we start to get things in the logging target of our choice—in this example, it is the console. For example, we issue a query such as this:

Code Listing 128

```
var projects = ctx
  .Projects
  .Where(x => x.Start == DateTime.Today)
  .Select(x => new { x.ProjectId, x.Start })
  .ToList();
```

We're likely to get output like the following in the console:

```
info:
Microsoft.EntityFrameworkCore.Storage.Internal.RelationalCommandBuilderFactory[1]
```

```
Executed DbCommand (2ms) [Parameters=[], CommandType='Text', CommandTimeout='30']
   SELECT [x].[ProjectId], [x].[Start]
   FROM [Project] AS [x]
   WHERE [x].[Start] = @__Today_0
```

Notice that the SQL does not include the actual filtering value (**DateTime.Today**); instead, it references the parameter name that was used. Besides the actual SQL, we can see that the execution took two milliseconds.

How do we interpret this? Well, first, we can see the log level, **info** in this case. Then we have the provider class that logged this (the category name), **RelationalCommandBuilderFactory**, and then the event inside **[]**, which in this example is **1**. Finally, we have the actual message, which is specific to the log event parameters (Executed DbCommand).

Microsoft makes some logging providers for .NET Core available:

Table 7

Provider	Purpose
Microsoft.Extensions.Logging.Console	Logs all messages with log level equal to or greater than information to the console of the current application.
Microsoft.Extensions.Logging.Debug	Logs to the debug window of the current attached debugger (like Visual Studio while debugging) with log level equal to or greater than information.
Microsoft.Extensions.Logging.EventLog	Writes to the Windows Event Log.
Microsoft.Extensions.Logging.EventSource	Logs to registered **EventListeners**.
Microsoft.Extensions.Logging.TraceSource	Writes to all registered trace listeners.

You can also write your own logging provider. You need to create a logging provider factory, register it with an Entity Framework Core context, and it will be used automatically. Here is an example for writing a (simple) log provider that calls a supplied delegate:

Code Listing 129

```
public static class LoggerFactoryExtensions
{
  public static ILoggerFactory AddAction(this ILoggerFactory loggerFactory,
Action<LogLevel, EventId, Exception, string> action, Func<string, LogLevel,
bool> filter)
```

```
  {
    if (action == null)
    {
      throw new ArgumentNullException(nameof(action));
    }

    loggerFactory.AddProvider(new ActionLoggerProvider(action, filter));
    return loggerFactory;
  }

  public static ILoggerFactory AddAction(this ILoggerFactory loggerFactory,
Action<LogLevel, EventId, Exception, string> action)
  {
    return AddAction(loggerFactory, action, null);
  }
}

public sealed class ActionLoggerProvider : ILoggerProvider
{
  private readonly Action<LogLevel, EventId, Exception, string> _action;
  private readonly Func<string, LogLevel, bool> _filter;

  public ActionLoggerProvider(Action<LogLevel, EventId, Exception, string>
action,
    Func<string, LogLevel, bool> filter)
  {
    this._action = action;
    this._filter = filter ?? delegate (string cn, LogLevel ll) { return
true; };
  }

  public ILogger CreateLogger(string categoryName)
  {
    return new ActionLogger(categoryName, this._action, this._filter);
  }

  public void Dispose()
  {
  }
}

public sealed class ActionLogger : ILogger
{
  private readonly string _categoryName;
  private Action<LogLevel, EventId, Exception, string> _action;
  private Func<string, LogLevel, bool> _filter;

  public ActionLogger(string categoryName, Action<LogLevel, EventId,
    Exception, string> action, Func<string, LogLevel, bool> filter)
  {
```

```
    this._categoryName = categoryName;
    this._action = action;
    this._filter = filter;
  }

  public IDisposable BeginScope<TState>(TState state)
  {
    return NullDisposable.Instance;
  }

  public bool IsEnabled(LogLevel logLevel)s
  {
    return this._filter(this._categoryName, logLevel);
  }

  public void Log<TState>(LogLevel logLevel, EventId eventId, TState state,
    Exception exception, Func<TState, Exception, string> formatter)
  {
    this._action(logLevel, eventId, exception, formatter(state,
exception));
  }
}

internal sealed class NullDisposable : IDisposable
{
  public static readonly IDisposable Instance = new NullDisposable();

  private NullDisposable()
  {
  }

  public void Dispose()
  {
  }
}
```

So, to add this provider, you need to do this:

Code Listing 130

```
loggerFactory.AddAction((logLevel, eventId, exception, message) =>
  {
    //do something
  });
```

You can either pass a filter to **AddAction** or not. If you don't, anything will be logged.

Entity Framework Core will use the logger factory provided in the “default” dependency injection configuration, if you are using ASP.NET Core, but you can also provide your own, with your specific settings. For example, in .NET Core 2.x, you can filter out which events to log, based on the log category (the class name bound to the logger), the log level or the provider itself. For example, if you wish to log to the console all the SQL being produced by EF Core, you can do something like this:

Code Listing 131

```
var loggerFactory = new LoggerFactory()
  .AddConsole((categoryName, logLevel) => (logLevel ==
LogLevel.Information) && (categoryName ==
DbLoggerCategory.Database.Command.Name));

var optionsBuilder = new DbContextOptionsBuilder<ProjectsContext>()
  .UseLoggerFactory(loggerFactory)
  .UseSqlServer("<connection string>");

var ctx = new ProjectsContext(optionsBuilder.Options);
```

So, the **`UseLoggerFactory`** method allows you to supply your own logger provider to a **`DbContext`**. You can both set it the way I’ve shown (using a **`DbContextOptionsBuilder<T>`**) or through the **`OnConfiguring`** method. The **`DbLoggerCategory`** class and its subclasses contain names for logger category names, and they are:

Table 8

Class	Purpose
`DbLoggerCategory.Database`	Events related to database interactions
`DbLoggerCategory.Database.Command`	SQL sent to the database
`DbLoggerCategory.Database.Connection`	Connection-related events
`DbLoggerCategory.Database.Transaction`	Transaction operations
`DbLoggerCategory.Infrastructure`	Miscelaneous messages
`DbLoggerCategory.Migrations`	Migration events
`DbLoggerCategory.Model`	Model building and metadata events

Class	Purpose
DbLoggerCategory.Model.Validation	Model validation events
DbLoggerCategory.Query	Messages related to queries, excluding the SQL generated
DbLoggerCategory.Scaffolding	Scaffolding/reverse engineer events
DbLoggerCategory.Update	Database updates

The **Name** property contains the category name which is also output by the logger.

Finally, you can change the behavior of certain events–ignore, log, throw an exception–by making a call to **ConfigureWarnings** inside the **OnConfiguring** method:

Code Listing 132

```
optionsBuilder.ConfigureWarnings(
  warnings =>
  {
    warnings.Ignore(RelationalEventId.OpeningConnection,
      RelationalEventId.ClosingConnection);
    warnings.Throw(RelationalEventId.RollingbackTransaction);
});
```

Setting a default is as simple as this:

Code Listing 133

```
optionsBuilder.ConfigureWarnings(
  warnings =>
  {
    warnings.Default(WarningBehavior.Ignore);
    warnings.Log(RelationalEventId.CommitTransaction);
  });
```

In this case, the default log behavior is to ignore everything and only log transaction commits.

Diagnostics

EF Core also leverages the diagnostics framework. In order to understand it, you must know that all .NET Core components produce diagnostic information (events), which can be consumed by listeners registered to the default **`DiagnosticListener`** (instantiated automatically and registered to the dependency injection framework). For example, EF Core produces events for the "**Microsoft.EntityFrameworkCore.Database.Command.CommandExecuted**" and "**Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting**" occurrences. So, we can hook a listener to the **`DiagnosticListener`**, a reference to which we can obtain from the DI framework:

Code Listing 134

```
DiagnosticListener listener = …;
listener.SubscribeWithAdapter(new CommandListener());
```

And a sample listener could look like this:

Code Listing 135

```
public class CommandListener
{
  [DiagnosticName(
    "Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting")]
  public void OnCommandExecuting(DbCommand command,
    DbCommandMethod executeMethod,
    Guid commandId, Guid connectionId, bool async,
    DateTimeOffset startTime)
  {
  }

  [DiagnosticName(
    "Microsoft.EntityFrameworkCore.Database.Command.CommandExecuted")]
  public void OnCommandExecuted(object result, bool async)
  {
  }
}
```

These two methods can be named whatever you want, but they must be decorated with **`[DiagnosticName]`** attributes with these names, and take the same-name parameters. When EF Core executes commands (SQL), they will be called, and the developer is even given a chance to modify the generated SQL or the parameters to the **`DbCommand`**.

There are many events that EF Core produces, and they match the names of **`RelationalEventId`** and **`CoreEventId`** (and **`SqlServerEventId`**, for SQL Server):

Table 9

Name
RelationalEventId.AmbientTransactionWarning
RelationalEventId.BoolWithDefaultWarning
RelationalEventId.CommandError
RelationalEventId.CommandExecuted
RelationalEventId.CommandExecuting
RelationalEventId.ConnectionClosed
RelationalEventId.ConnectionClosing
RelationalEventId.ConnectionError
RelationalEventId.ConnectionOpened
RelationalEventId.ConnectionOpening
RelationalEventId.DataReaderDisposing
RelationalEventId.MigrateUsingConnection
RelationalEventId.MigrationApplying
RelationalEventId.MigrationGeneratingDownScript
RelationalEventId.MigrationGeneratingUpScript
RelationalEventId.MigrationReverting
RelationalEventId.MigrationsNotApplied
RelationalEventId.MigrationsNotFound
RelationalEventId.ModelValidationKeyDefaultValueWarning
RelationalEventId.QueryClientEvaluationWarning
RelationalEventId.QueryPossibleExceptionWithAggregateOperator
RelationalEventId.QueryPossibleUnintendedUseOfEqualsWarning
RelationalEventId.TransactionCommitted
RelationalEventId.TransactionDisposed
RelationalEventId.TransactionError
RelationalEventId.TransactionRolledBack
RelationalEventId.TransactionStarted
RelationalEventId.TransactionUsed

These names should be meaningful enough for you to understand, but a problem with them is that they all take different parameters (the ones that need to go in the listener method marked with **`[DiagnosticName]`**). You will need to consult the documentation on each of these properties to find out which they are.

Chapter 6 Performance Optimizations

Having a pool of DbContexts

Entity Framework Core 2 allows us to precreate a pool of **`DbContexts`**. Instead of creating one new instance for each time it is requested—normally, at least once per request—you can have a number of contexts precreated and ready to be used. It is configured like this:

Code Listing 136

```
services.AddDbContextPool<ProjectsContext>(options =>
{
  options.UseSqlServer("<connection string>");
}, 128);
```

In this example, we are registering a pool of 128 contexts, which is actually the default.

Using a profiler

There are some profilers, specific for Entity Framework Core or otherwise, that can assist you in writing better queries and understanding what is going on below. The most-used one is probably Entity Framework Profiler, by Hibernating Rhinos.

But you also have to look directly at the database. For that purpose, in the case of SQL Server, you have:

- SQL Server Profiler (included with the SQL Server package)
- Express Profiler (free and open source)

These allow you to hook directly up to the database and see all SQL sent to it.

Filter entities in the database

You are certainly aware that Entity Framework works with two flavors of LINQ:

- LINQ to Objects: Operations are performed in memory.
- LINQ to Entities: Operations are performed in the database.

LINQ to Entities queries that return collections are executed immediately after calling a "terminal" method—one of ToList, ToArray, or ToDictionary—or when enumerating a collection (when **GetEnumerator** is called). After that, the results are materialized, and are therefore stored in the process's memory space. Being instances of IEnumerable<T>, they can be manipulated with LINQ to Objects standard operators without us even noticing. This means it's totally different to issue these two queries, because one will be executed fully by the database, while the other will be executed in memory after retrieving all entities.

Code Listing 137

```
//LINQ to Objects: all Technologies are retrieved from the database and
filtered in memory.
var technologies = ctx.Technologies.ToList().Where(x => x.Resources.Any());

//LINQ to Entities: Technologies are filtered in the database, and only
after retrieved into memory.
var technologies = ctx.Technologies.Where(x => x.Resources.Any()).ToList();
```

Beware, you may be bringing a lot more records than you expected!

Do not track entities not meant to be changed

Entity Framework has a first-level (or local) cache where all the entities known by an EF context, loaded from queries or explicitly added, are stored. This happens so that when the time comes to save changes to the database, EF goes through this cache and checks which ones need to be saved (which ones need to be inserted, updated, or deleted). What happens if you load a number of entities from queries when EF has to save? It needs to go through all of them to see which have changed, and that constitutes the memory increase.

If you don't need to keep track of the entities that result from a query, since they are for displaying only, you should apply the AsNoTracking extension method:

Code Listing 138

```
//no caching
var technologies = ctx.Technologies.AsNoTracking().ToList();

var technologiesWithResources = ctx
  .Technologies
  .Where(x => x.Resources.Any())
  .AsNoTracking()
  .ToList();

var localTechnologies = ctx.Technologies.Local.Any();  //false
```

This even causes the query execution to run faster, because EF doesn't have to store each resulting entity in the cache.

Only use eager loading where appropriate

As you saw in the Eager section, you have only two options when it comes to loading navigation properties: either load them together with the root entity, or live without them. Generally speaking, when you are certain of the need to access a reference property or to go through all of the child entities present in a collection, you should eager-load them with their containing entity. There's a known problem called **`SELECT N + 1`** that illustrates this: you issue one base query that returns **`N`** elements, and then you issue another **`N`** queries, one for each reference or collection that you want to access.

Performing eager loading is achieved by applying the Include extension method.

Code Listing 139

```
//eager load the Technologies for each Resource.
var resourcesIncludingTechnologies = ctx
  .Resources
  .Include(x => x.Technologies)
  .ToList();

//eager load the Customer for each Project.
var projectsIncludingCustomers = ctx
  .Projects
  .Include("Customer")
  .ToList();
```

This way you can potentially save a lot of queries, but it can also bring much more data than the one you need.

Use paging

Instead of retrieving all records from the database, you should only return the useful part of them. For that, you use paging, which is implemented in LINQ through the Skip and Take methods:

Code Listing 140

```
//get from the 11th to the 20th projects ordered by their start date.
var projects = ctx
  .Projects
  .OrderBy(x => x.Start)
  .Skip(10)
  .Take(10)
  .ToList();
```

Don't forget an OrderBy clause; otherwise, you will get an exception.

Use projections

As you know, normally the LINQ queries return full entities; that is, for each entity, they bring along all of its mapped properties (except references and collections). Sometimes we don't need the full entity, but just some parts of it, or even something calculated from some parts of the entity. For that purpose, we use projections.

Projections allow us to significantly reduce the data returned by handpicking just what we need, which is particularly useful when we have entities with a large number of properties. Here are some examples:

Code Listing 141

```
//return the resources and project names only with LINQ.
var resourcesXprojects = ctx
  .Projects
  .SelectMany(x => x.ProjectResources)

.Select(x => new { Resource = x.Resource.Name, Project = x.Project.Name })
  .ToList();

//return the customer names and their project counts with LINQ.
var customersAndProjectCount = ctx
  .Customers
  .Select(x => new { x.Name, Count = x.Projects.Count() })
  .ToList();
```

Projections with LINQ depend on anonymous types, but you can also select the result into a .NET class for better access to its properties.

Code Listing 142

```
//return the customer names and their project counts into a dictionary with
LINQ.
var customersAndProjectCountDictionary = ctx.Customers
  .Select(x => new { x.Name, Count = x.Projects.Count() })
  .ToDictionary(x => x.Name, x => x.Count);
```

Use disconnected entities

When saving an entity, if you need to store in a property a reference to another entity for which you know the primary key, then instead of loading it with Entity Framework, just assign it a blank entity with only the identifier property filled.

Code Listing 143

```
//save a new project referencing an existing Customer.
var newProject = new Project { Name = "Some Project", Customer =
new Customer { CustomerId = 1 } };
```

```
//instead of Customer = ctx.Customers.SingleOrDefault(x => x.CustomerId ==
1)

ctx.Projects.Add(newProject);
ctx.SaveChanges();
```

This will work fine because Entity Framework only needs the foreign key set.

This also applies to deletes—no need to load the entity beforehand; its primary key will do.

Code Listing 144

```
//delete a Customer by id.
//ctx.Customers.Remove(ctx.Customers.SingleOrDefault(x => x.CustomerId ==
1));
ctx.Entry(new Customer { CustomerId = 1 }).State = EntityState.Deleted;
ctx.SaveChanges();
```

Use compiled queries

Even though EF Core compiles LINQ queries the first time they are used, there is still a minor performance penalty each time we issue a query, as the cache needs to be looked up. We can use compiled queries—which were present in old versions of Entity Framework—to get around this:

Code Listing 145

```
private static readonly Func<ProjectsContext, int, Project>
_projectsByCustomer = EF.CompileQuery((ProjectsContext ctx, int customerId)
=> ctx.Projects.Where(x => x.Customer.Id == customerId).Include(x =>
x.Resources).ToList();

var projects = _projectsByCustomer(ctx, 100);
```

As you can see, you can even eagerly include collections. Make sure you add a terminating method, like ToList.

Use SQL where appropriate

If you have looked at the SQL generated for some queries, and you know your SQL, you can probably tell that it is far from optimized. This is because EF uses a generic algorithm for constructing SQL that automatically picks up the parameters from the specified query and puts it all blindly together. Of course, understanding what we want and how the database is designed, we may find a better way to achieve the same purpose.

When you are absolutely certain that you can write your SQL in a better way than EF can, feel free to experiment with SqlQuery and compare the response times. You should repeat the tests a number of times, because other factors may affect the results, such as other accesses to the database, the Visual Studio debugger, and the number of processes running in the test machines. All of these can cause impact.

One thing that definitely has better performance is batch deletes or updates. Always do them with SQL instead of loading the entities, changing or deleting them one by one, and then saving the changes. Use the ExecuteSqlCommand method for that purpose.

Chapter 7 Common Pitfalls

Overview

Whenever you start using a new technology, there's always the possibility that you will fall into one of its traps. Here I will list some of them.

Group By is performed in memory

In this version of Entity Framework Core, the translation of **`Group By`** to SQL hasn't been implemented. The result is that EF will fetch all the records from the database and then perform the grouping in memory on the client-side, which is a performance nightmare. You will need to use plain SQL to execute your query, if you really need it.

Changes are not sent to the database unless SaveChanges is called

This is probably obvious, but still people sometimes forget about it.

No support for date and time or mathematical operations

You need to use SQL for this. Any nontrivial queries over **`DateTime`**, **`DateTimeOffset`**, **`TimeSpan`**, or using the **`Math`** static methods will fail.

No implicit loading

This is not implemented yet. You need to force the loading of all the collections and references you need by adding Include calls in your queries, or load them explicitly afterwards using **`Load`**. See the " Explicit" section.

LINQ queries over unmapped properties

Visual Studio IntelliSense makes it very easy to write LINQ queries because it automatically shows all available properties. It may happen that some of these properties are not actually mapped. For example, they are read-only, calculated columns that are computed from other properties. Access to these properties cannot be translated to SQL queries, so any attempt to use them in an Entity Framework query will result in an exception.

Null navigation properties

If you have a null navigation property in a loaded entity, this is because you didn't explicitly fetch it through a call to Include.

For more information, revisit the "Eager" section.

Cannot have non-nullable columns in single-table inheritance

When using the Single Table Inheritance pattern (the only supported inheritance pattern as of EF 1.1), you cannot have non-nullable properties in derived classes, because it all will be created in the same table, and they won't exist for all derived classes.

Cannot use the integrated service provider in an infrastructure method

The Entity Framework Core context includes its own service provider. You can replace it when you build the context; otherwise you get a default one. You may be tempted to try to get services from it in one of the infrastructure methods like **`OnConfiguring`** or **`OnModelCreating`**, but you won't be lucky, because you will get an exception. The service provider can only be accessed after the context is fully initialized, which includes these methods having run.

Using the integrated service provider requires a special constructor

You can register your EF contexts, for example, so that they are injected into an MVC controller. This is normally done in the **`ConfigureServices`** method of the **`Startup`** class through the **`AddDbContext`** extension method:

Code Listing 146

```
services.AddDbContext<ProjectsContext>(options =>
```

```
{
  options.UseSqlServer("<connection string>");
});
```

For this to work, however, you need to add a specific constructor to your context and call the base implementation:

Code Listing 147

```
public class ProjectsContext :  DbContext
{
  public ProjectsContext(DbContextOptions options) : base(options)
  {
  }
}
```

Otherwise, the included service provider will not be able to build your context.

Migrations need to go on a .NET Core project

You cannot have migrations on a .NET Standard project. If, for example, your context belongs in a .NET Standard project like “Domain Model” or similar, you will need to reference a .NET Core project (like “Web” or “Console Application”) where migrations are to be created:

Code Listing 148

```
protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
{
  optionsBuilder .UseSqlServer(connectionString, b =>
b.MigrationsAssembly("Web"));
  base.OnConfiguring(optionsBuilder);
}
```

Contexts without parameterless constructors cannot be used in migrations

If you have a context that does not feature a public parameterless constructor, you need to implement a context factory so that it can be instantiated by the migrations tooling:

Code Listing 149

```
public class ProjectsContextFactory :
IDesignTimeDbContextFactory<ProjectsContext>
{
  public ProjectsContext CreateDbContext(string[] args)
  {
```

```
    var configurationBuilder = new ConfigurationBuilder()
      .AddJsonFile("appSettings.json", false);

    var configuration = configurationBuilder.Build();

    return new ProjectsContext(
      configuration["ConnectionStrings:Projects"]);
  }
}
```

Appendix A Working with Other Databases

Entity Framework is agnostic regarding databases, which means it can potentially be used with any database for which there is an Entity Framework Core provider. Having said that, the harsh truth is that it can be cumbersome for databases other than SQL Server. The major problems are:

- The supported primary key generation strategies are IDENTITY, manual assigned, and the **High-Low** algorithm. Unfortunately, IDENTITY is inherently SQL Server-based, and the High-Low implementation also requires SQL Server 2012 or higher for sequences.
- Some .NET types may not be supported by some other databases, such as enumerated types or **`Guid`**.
- The database data types, even if conceptually identical, have different names. For example, a variable length Unicode string in SQL Server is called **`NVARCHAR`**, where in Oracle it is called **`VARCHAR2`**. Be careful when you need to specify it.
- Some equivalent types are slightly different. For example, a **`DateTime`** property can be translated to a **`DATETIME`** column in SQL Server, which has both a date and a time part, but when translated to a **`DATE`** column in Oracle, it will only have the date part. Another example, **`DateTimeOffset`** even has equivalents in Oracle and others, but not on SQL Server 2005.
- The ROWVERSION type implied by the TimestampAttribute for concurrency checks, or better, the way it works, also only exists in the SQL Server family.
- There may be different accuracies for floating-point or decimal types.

It is certainly possible to use Entity Framework to target other databases. Having the same code base for that almost certainly won't work. I'll just leave you with some guidelines for cross-database projects:

- Do use only "safe" base types, such as strings, numbers, byte arrays, and date/times.
- Do specify the physical column name for each property.
- Do specify the physical name and schema for each entity.
- Do use manually assigned identifiers.
- Do not specify a database type name for a property.

For an up-to-date list of third-party Entity Framework providers, check out this resource.

Generally speaking, to work with a database other than SQL Server (or even to select SQL Server) you need to install a NuGet package for that database, and then configure it on a context by leveraging the **`OnConfiguring`** method:

Code Listing 150

```
protected override void OnConfiguring(
  DbContextOptionsBuilder optionsBuilder)
{
  optionsBuilder.UseSqlServer(_nameOrConnectionString);
  base.OnConfiguring(optionsBuilder);
}
```

Here you plug the provider you wish to use to **optionsBuilder**. Each provider may add some specific options: for SQL Server, it is the connection string; for SQLite, the location of the database file, etc.

For SQLite, you need to install the Microsoft.EntityFrameworkCore.Sqlite NuGet package. After you do, you can have this:

Code Listing 151

```
protected override void OnConfiguring(
  DbContextOptionsBuilder optionsBuilder)
{
  optionsBuilder.UseSqlite("Filename=./Projects.db");
  base.OnConfiguring(optionsBuilder);
}
```

Another included provider is **InMemory**: a good addition when it comes to testing. Its NuGet package is Microsoft.EntityFrameworkCore.InMemory:

Code Listing 152

```
protected override void OnConfiguring(
  DbContextOptionsBuilder optionsBuilder)
{
  optionsBuilder.UseInMemoryDatabase();
  base.OnConfiguring(optionsBuilder);
}
```

Appendix B Features Not in Entity Framework Core 2.0

Because Entity Framework Core was a total rewrite of Entity Framework—and, in fact, all of .NET Core was a rewrite of .NET—not all of its features are available in version 2.0. Some of them will be available in future releases, and others will not be replaced at all. In the following table, I am going to show what they are.

Table 10

Feature	Description
Lazy loading	For now, we need to use eager loading (`Include`). Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/3797
Support for Group By	Currently, it falls back silently to LINQ to Objects, meaning it brings everything from the database and groups in memory. For now, use plain SQL. Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/2341
Many to Many Collections	For now, we need a middle entity. Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/1368
Table Per Type Inheritance Strategy	Use Table Per Class Hierarchy/ Single Table Inheritance. Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/2266
Table Per Concrete Type Inheritance Strategy	Use Table Per Class Hierarchy / Single Table Inheritance. Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/3170

Feature	Description
Common SQL and DateTime Operations	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/2850
Mapping CUD with stored procedures	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/245
Map database views	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/1679 https://github.com/aspnet/EntityFramework/issues/827
Spatial data types	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/1100
Custom conventions	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/214
Populate non-model types from SQL	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/240
Seeding data in migrations	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/629
Command and query interception	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/626 https://github.com/aspnet/EntityFramework/issues/4048 https://github.com/aspnet/EntityFramework/issues/737

Feature	Description
Visual Studio support for generating/updating entities from the database and viewing the model	Will be available in a future version. https://github.com/aspnet/EntityFramework/issues/5837
Support for System.Transactions	Will be available in a future version. https://github.com/aspnet/EntityFrameworkCore/issues/5595
Lifecycle Events (SavingChanges, ObjectMaterialized)	Will be available in a future version. https://github.com/aspnet/EntityFrameworkCore/issues/3204 https://github.com/aspnet/EntityFrameworkCore/issues/626
Database initializers	Dropped
Automatic migrations	Dropped
ObjectContext (Entity SQL, events)	Dropped
Model first	Dropped
Data Annotations validations	Dropped

For the most up-to-date roadmap, please consult the Entity Framework Roadmap on GitHub. Entity Framework Core issues are maintained here.