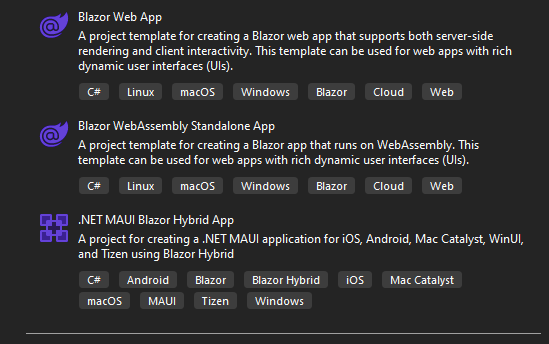
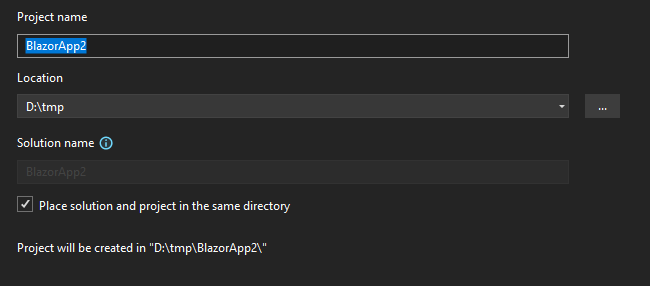
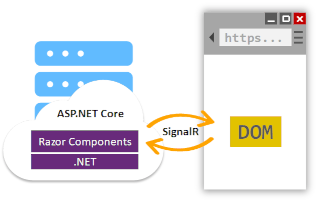
# Blazor 8

1. A whole bunch of great articles about Blazor have been published on the internet not so long ago, see the [Blazor University](https://blazor-university.com/) site. Microsoft has released a brand new version of Blazor. It has become a great WEB programming tool, but some of the statements and recommendations in the aforementioned series of articles are outdated and require corrections. I will describe here what, where and how it has changed.
2. First of all, it should be emphasized that the current version of **Blazor** requires:
3. Windows 11,
4. .NET 8
5. Visual Studio 2022 (Community Edition or any paid version).

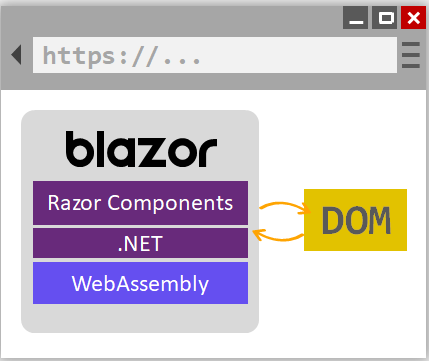
## Project templates

1. Visual Studio offers 3 different templates after creating new application:
2. 
3. **Blazor WebAssembly Standalone** application will create an application very similar to the one you had in your Windows 10 environment.
4. **.NET MAUI Blazor Hybrid App** is a Xamarin enhancement. This template creates applications that work as operating system applications and access the server directly, without a browser.
5. **Blazor Web App** is intended for the development of new WEB applications, and here the new **Blazor** variant works at full power. Visual studio asks for project parameters after selecting **Blazor Web App**. The dialog is the same as in the previous version of Blazor:
6. After setting the project parameters, the main dialog appears:
7. 
8. **Framework dropdown**: there are currently no options. More options will probably appear in this control later.
9. **Authentication type**: here you can choose **none** or **Individual Accounts**. Visual Studio will place the ..\Components\Account directory in the server part project after choosing the second option. The directory contains the files required for name/password authentication. The main menu is also expanded. We will talk about authentication and its configuration later, in a separate section.
10. **Configure for HTTPS**: this checkbox needs to be checked when you use standard authentication. The **SSL certificate** will be used by Blazor even after unchecking this control, as it is required when the page works in Server Side Rendering (SSR) mode. We'll talk about this later when we look at the different rendering modes.
11. **Interactive render mode** allows you to choose one of four options:
12. **None** - in this mode, applications will load extremely fast as there is no work required on the client, and no large WebAssembly assets to download. The server is just sending HTML that the browser then renders. This also means that each request for a new page results in a full page load. This kind of application is an enhancement of Razor pages: you can use Blazor components here, but you will have to use JavaScript for handling events.
13. **Server** - in this mode a page, or component, will be optionally pre-rendered on the server and then made interactive on the client via a **SignalR** connection. Once interactive, all events on the client will be transmitted back to the server over the SignalR connection to be processed on the server. Any updates required to the DOM will then be packaged up and sent to the client over the same SignalR connection where a small Blazor runtime will patch the updates into the DOM.
14. **WebAssembly** - this model is derived from the Blazor WebAssembly hosting model and fully capitalises on client-side capabilities, allowing C# code to run in the user’s browser. Initial data would be downloaded to the client along with the various framework DLLs and WebAssembly runtime. Once on the client, it would be bootstrapped and the page loaded. Any API calls to get data would be made and the UI would be re-rendered as necessary to display any data returned.
15. **Auto (Server and WebAssembly)** - it is main mode of Blazor applications in .NET 8. When setting a page or component to use Auto mode, the initial load of that component will be via server mode making it super fast. But in the background Blazor will download the necessary assets to the client so that on the next load it can be done using WebAssembly mode. This rendering mode will address the biggest pain point for developers when embarking on a new Blazor project, what hosting model should we use? Every component marked with **RenderMode.Auto** will need to execute on both the server and the client. Meaning that there will need to be some form of abstraction in place if the component needs to fetch any data.

### Blazor Server

1. 
2. **ASP.NET Core** apps and **Blazor Server** use the Razor language to describe HTML content for rendering, but they significantly differ in how markup is rendered.
3. When a Razor Page or view is rendered, every line of Razor code emits HTML in text form. After rendering, the server disposes of the page or view instance, including any state that was produced. When another request for the page occurs, the entire page is rerendered to HTML again and sent to the client.
4. Blazor Server produces a graph of components to display similar to an HTML or XML DOM. The component graph includes state held in properties and fields. Blazor evaluates the component graph to produce a binary representation of the markup, which is sent to the client for rendering. After the connection is made between the client and the server, the component's static prerendered elements are replaced with interactive elements. Prerendering content on the server in order to load HTML content on the client quickly makes the app feel more responsive to the client.
5. After the components are interactive on the client, UI updates are triggered by user interaction and app events. When an update occurs, the component graph is rerendered, and a UI diff (difference) is calculated. This diff is the smallest set of DOM edits required to update the UI on the client. The diff is sent to the client in a binary format and applied by the browser.

### Blazor WebAssembly

1. 
2. Running .NET code inside web browsers is made possible by **WebAssembly** (abbreviated wasm). **WebAssembly** is a compact bytecode format optimized for fast download and maximum execution speed. WebAssembly is an open web standard and supported in web browsers without plugins. WebAssembly works in all modern web browsers, including mobile browsers.
3. **WebAssembly** code can access the full functionality of the browser via JavaScript, called **JavaScript interoperability**, often shortened to JavaScript interop or JS interop. .NET code executed via WebAssembly in the browser runs in the browser's JavaScript sandbox with the protections that the sandbox provides against malicious actions on the client machine.
4. When a Blazor WebAssembly app is built and run:
5. C# code files and Razor files are compiled into .NET assemblies.
6. The assemblies and the [.NET runtime](https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview) are downloaded to the browser.
7. Blazor WebAssembly bootstraps the .NET runtime and configures the runtime to load the assemblies for the app. The Blazor WebAssembly runtime uses JavaScript interop to handle DOM manipulation and browser API calls.
8. The size of the published app, its payload size, is a critical performance factor for an app's usability. A large app takes a relatively long time to download to a browser, which diminishes the user experience.
9. For apps that require third-party JavaScript libraries and access to browser APIs, components interoperate with JavaScript. Components are capable of using any library or API that JavaScript is able to use. C# code can call into JavaScript code, and JavaScript code can call into C# code.
10. Blazor implements the .NET Standard, which enables Blazor projects to reference libraries that conform to .NET Standard specifications. .NET Standard is a formal specification of .NET APIs that are common across .NET implementations. .NET Standard class libraries can be shared across different .NET platforms, such as Blazor, .NET Framework, .NET Core, Xamarin, Mono, and Unity.APIs that aren't applicable inside of a web browser (for example, accessing the file system, opening a socket, and threading) throw a **PlatformNotSupportedException**.

### Auto

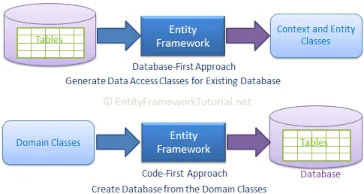
1. Set these parameters on next screen:
2. **Framework** - .NET 8.x,
3. **Authentication type** – Individual accounts,
4. **Configure for HTTPS** – true,
5. **Interactive render mode** – Auto (Server and WebAssembly),
6. **Interactivity location** – Per page/component,
7. **Include sample pages** – true,
8. **Do not use top-level statements** – false.
9. Visual Studio will create two projects after setting these parameters: server and client. Here I’ll demonstrate how these two projects may be used for processing any database.
10. The ~/Data directory has the ApplicationDbContext.cs file, which contains DB context for the Entity Framework:
    1. public class ApplicationDbContext(DbContextOptions<**ApplicationDbContext**> options)
    2. : IdentityDbContext<**ApplicationUser**>(options)
    3. {
    4. }
    5. Second file in this directory is **ApplicationUser.cs**. This file defines **ApplicationUser** class:
    6. public class ApplicationUser : IdentityUser
    7. {
    8. }
11. In the generated version, this class contains only those fields that are described in the standard class **Microsoft.AspNetCore.Identity**. You can add any number of additional fields.
12. The **appsettings.json** file contains the database connection: see to the "ConnectionStrings" section. The local database is specified in the generated task:
13. "ConnectionStrings": {
14. "DefaultConnection": "Server=(localdb)\\mssqllocaldb;Database=aspnet-BlazorApp4-0c7655a7-1f05-43cd-921a-72502c56eb35;Trusted\_Connection=True;MultipleActiveResultSets=true"
15. },
16. **"Server=…**" is one long string. JSON doesn't allow breaking lines for readability.

#### Entity Framework (EF)

Blazor server uses **EF Core 8** for work with SQL databases. EF Core supports two development approaches 1) Code-First 2) Database-First. EF Core mainly targets the code-first approach and provides little support for the database-first approach because the visual designer or wizard for DB model is not supported as of EF Core.

In the code-first approach, EF Core API creates the database and tables using migration based on the conventions and configuration provided in your domain classes. This approach is useful in Domain Driven Design (DDD).

In the database-first approach, EF Core API creates the domain and context classes based on your existing database using EF Core commands. This has limited support in EF Core as it does not support visual designer or wizard.



##### EF Core Database Providers

Entity Framework Core uses a provider model to access many different databases. EF Core includes providers as NuGet packages which you need to install.

Here is a list of database providers and NuGet packages for EF Core:

1. **SQL Server** Microsoft.EntityFrameworkCore.SqlServer
2. **MySQL** MySql.Data.EntityFrameworkCore
3. **PostgreSQL** Npgsql.EntityFrameworkCore.PostgreSQL
4. **SQLite** Microsoft.EntityFrameworkCore.SQLite
5. **SQL Compact** EntityFrameworkCore.SqlServerCompact40
6. **In-memory** Microsoft.EntityFrameworkCore.InMemory

**EF Core** is not included as a default package in .NET 7. We need to install it via the NuGet package.

Database provider is registered in the Program.cs file. Standart template registers DbContext service:

builder.Services.AddDbContext<ApplicationDbContext>(options =>

options.**UseSqlServer(connectionString)**);

and connection string in the **appsettings.json** file. You need to change both values for switching to another DB engine. It will be demonstrate in the topics about PostgreSQL database.

The **Microsoft.EntityFrameworkCore** is the base package for all the basic operations of EF Core. However, you have to install a database provider package from NuGet for the database you use in your project. For example, to use the MS SQL Server database, you need to install **Microsoft.EntityframeworkCore.SqlServer** package.

The database provider package of EF Core includes all the other dependent packages it needs. So, it includes **Microsoft.EntityFrameworkCore** too. So, no need to install it separately.

EF can be installed from Visual Studio by opening the '**Package manager**' console or by running the '**dotnet add package**' command from the terminal. Before running the latter command, change to the directory containing the \*.csproj file.

The '**Powershell**' window can also be opened from Visual Studio: **'right click**' on the project and select '**Open in terminal**'. Type this command in the window:

dotnet add package Microsoft.EntityFrameworkCore.SqlServer

EF can also be installed after modifying the \*.csproj file. Open this file in any text editor and add the command into ItemGroup:

<ItemGroup>

<ProjectReference Include="..\BlazorApp4.Client\BlazorApp4.Client.csproj" />

<ProjectReference Include="..\StudentsManagement\StudentsManagement.csproj" />

<PackageReference Include="Microsoft.AspNetCore.Components.WebAssembly.Server" Version="8.0.0" />

<PackageReference Include="Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore" Version="8.0.0" />

<PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="8.0.0" />

<PackageReference Include="Microsoft.EntityFrameworkCore" Version="8.0.1" />

<**PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="8.0.0"** />

<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.0" />

<PackageReference Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" Version="8.0.0" />

<PackageReference Include="System.Diagnostics.Debug" Version="4.3.0" />

</ItemGroup>

##### The model

With EF Core, data access is performed using a **model**. A model is made up of **entity classes** and a **context** object that represents a session with the database. The context object allows querying and saving data.

EF supports the following model development approaches:

* Generate a model from an existing database.
* Hand code a model to match the database. Once a model is created, use EF Migrations to create a database from the model. Migrations allow evolving the database as the model changes:

using System.Collections.Generic;

using Microsoft.EntityFrameworkCore;

namespace Intro;

public class BloggingContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

public DbSet<Post> Posts { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseSqlServer(

@"Server=(localdb)\mssqllocaldb;Database=Blogging;Trusted\_Connection=True");

}

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public int Rating { get; set; }

public List<Post> Posts { get; set; }

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

public int BlogId { get; set; }

public Blog Blog { get; set; }

}

This template is only suitable for small databases. For large databases with many relations, place a configuration into static class of each entity:

using Microsoft.EntityFrameworkCore;

namespace BlazorApp4.Models

{

public class Student

{

public int Id { get; set; }

public string? FirstName { get; set; }

public string? MiddleName { get; set; }

public string? LastName { get; set; }

public string? EmailAddress { get; set; }

public string? Address { get; set; }

public string? PhoneNumber { get; set; }

public string? Country { get; set; }

public Guid Version { get; set; }

public static void **Configure**(ModelBuilder builder)

{

builder.Entity<Student>().HasKey(s => s.Id);

builder.Entity<Student>().Property(s => s.Id).ValueGeneratedOnAdd();

builder.Entity<Student>().Property(s => s.Version).IsConcurrencyToken();

}

}

}

The static **Configure** function will be called from the same OnModelCreating event:

namespace BlazorApp4.Data

{

public class ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :

IdentityDbContext<ApplicationUser>(options)

{

public DbSet<Student> Students { get; set; }

protected override void **OnModelCreating**(ModelBuilder builder)

{

base.OnModelCreating(builder);

**Student.Configure(builder)**;

}

}

}

##### Sequences

A sequence generates unique, sequential numeric values in the database. Sequences are not associated with a specific table, and multiple tables can be set up to draw values from the same sequence. You can set up a sequence in the model, and then use it to generate values for properties:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

**modelBuilder.HasSequence<int>("OrderNumbers")**;

modelBuilder.Entity<Order>()

.Property(o => o.OrderNo)

.HasDefaultValueSql("NEXT VALUE FOR **OrderNumbers**");

}

You can also configure additional aspects of the sequence, such as its schema, start value, increment, etc.:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.HasSequence<int>("OrderNumbers", schema: "shared")

.StartsAt(1000)

.IncrementsBy(5);

}

**SEQUENCE** was introduced in **SQL Server 2012**. Entity Framework Core supports **Sequence** out of the box. Sequence is a user-defined object and it generates a sequence of numeric values according to the properties with which it is created. It is similar to Identity column, but there are many differences between them:

* The Sequence is used to generate database-wide sequential number, where identity column is tied to a table.
* Sequence is not associated with a table. You can associate it with multiple tables.
* It can be used in insert statement to insert identity values, it can also be used in T-SQL Scripts.

Consider this model:

public class Category

{

public int CategoryID { get; set; }

public string CategoryName { get; set; }

}

**CategoryID** is primary key by convention. The application needs “**Microsoft.EntityFrameworkCore.SqlServer**” nuget package. **DBContext** looks like this template:

public class SampleDBContext : DbContext

{

public SampleDBContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionbuilder)

{

string sConnString =

@"Server=localhost;Database=EFSampleDB;Trusted\_Connection=true;"

optionbuilder.**UseSqlServer**(sConnString);

}

protected override void OnModelCreating(ModelBuilder modelbuilder)

{

**modelbuilder.ForSqlServerHasSequence<int>("DBSequence")**

**.StartsAt(1000).IncrementsBy(2);**

modelbuilder.Entity<Category>()

.Property(x => x.CategoryID)

.**HasDefaultValueSql("NEXT VALUE FOR DBSequence")**;

}

public DbSet<Category> Categories { get; set; }

The **ForSqlServerHasSequence** extension method is used for creating **SQL Server Sequence**. You need to supply the name of the sequence. Additionally, you can also configure start number and increment option. There is also a **HasSequence** extension method which is similar to **ForSqlServerHasSequence**. The reason for having these 2 different sets of method is, EF Core supports many database providers. And functionality for these providers are slightly different so you are able to specify a different behavior for the same property depending on the provider being used. Read [this](https://stackoverflow.com/questions/37936956/what-is-the-difference-between-the-has-and-forsqlserverhas-extension-m/38845439" \l "38845439) to find out more

###### Use HiLo to generate keys with Entity Framework Core

**Entity Framework Core** supports different key generation strategies like **identity**, **Sequence** and **HiLo**. Database sequences are cached, scalable and address concurrency issues. But there would be a database round-trip for every new the sequence value. And in case of high number of inserts this becomes a little heavy. But you can optimize the sequence with HiLo pattern. And EF Core supports “HiLo” out of the box.

**HiLo** is a pattern where the primary key is made of 2 parts “**Hi**” and “**L**o”. Where the “Hi” part comes from database and “Lo” part is generated in memory to create unique value. Remember, “**Lo**” is a range number like 0-100. So when “Lo” range is exhausted for “Hi” number, then again a database call is made to get next “Hi number”. So the advantage of HiLo pattern is that you know the key value in advance. Let’s see how to use HiLo to generate keys with Entity Framework Core.

Consider a model of two classes:

public class Category

{

public int CategoryID { get; set; }

public string CategoryName { get; set; }

}

public class Product

{

public int ProductID { get; set; }

public string ProductName { get; set; }

}

Remember, EF Core by convention configures a property named **Id** or **<type name>Id** as the key of an entity. Now we need to create our DBContext. Add a new class file and name it **SampleDBContext.cs** and add the following code.

public class SampleDBContext : DbContext

{

public SampleDBContext()

{

Database.EnsureDeleted();

Database.EnsureCreated();

}

protected override void OnConfiguring(DbContextOptionsBuilder optionbuilder)

{

string sConnString =

@"Server=localhost;Database=EFSampleDB;Trusted\_Connection=true;"

optionbuilder.UseSqlServer(sConnString);

}

protected override void OnModelCreating(ModelBuilder modelbuilder)

{

modelbuilder.ForSqlServerUseSequenceHiLo("DBSequenceHiLo");

modelbuilder.Entity<Category>()

.Property(x => x.CategoryID)

.HasDefaultValueSql("NEXT VALUE FOR DBSequenceHiLo");

modelbuilder.Entity<Product>()

.Property(x => x.ProductID )

.HasDefaultValueSql("NEXT VALUE FOR DBSequenceHiLo");

}

public DbSet<Product> Products { get; set; }

public DbSet<Category> Categories { get; set; }

}

DBSequenceHiLo will be created in the database:

CREATE SEQUENCE [dbo].[DBSequenceHiLo]

AS [bigint]

START WITH 1

INCREMENT BY 10

MINVALUE -9223372036854775808

MAXVALUE 9223372036854775807

CACHE

GO

As you can see it starts with **1** and get increment by **10**. There is a difference between a **Sequence** and **HiLo Sequence** with respect to **INCREMENT BY** option. In **Sequence**, **INCREMENT BY** will add “increment by” value to previous sequence value to generate new value. So in this case, if your previous sequence value was 11, then next sequence value would be 11+10 = 21. And in case of **HiLo** Sequence, **INCREMENT BY** option denotes a **block value** which means that next sequence value will be fetched after first 10 values are used.

#### MS SQL Server

1. It is enough to change the **DefaultConnection** element for connecting to the MS SQL server. Here is connection to server on my computer:
2. "ConnectionStrings": {
3. "DefaultConnection": "Data Source=GedoDell;Initial Catalog=BlazorApp4;Integrated Security=True;Encrypt=False;MultipleActiveResultSets=true"
4. },
5. The connection string can be specified in several different ways. All they are all listed in the [ConnectionStrings](https://www.connectionstrings.com/sql-server/) site.

#### Migrations

1. Entity Framework knows how to create a database from C# classes. Create a Data directory and add the following class to it:
2. public class Student
3. {
4. public int Id { get; set; }
5. public string? FirstName { get; set; }
6. public string? MiddleName { get; set; }
7. public string? LastName { get; set; }
8. public string? EmailAddress { get; set; }
9. public string? Address { get; set; }
10. public string? PhoneNumber { get; set; }
11. public string? Country { get; set; }
12. public Guid Version { get; set; }
13. public static void Configure(ModelBuilder builder)
14. {
15. builder.Entity<Student>().HasKey(s => s.Id);
16. builder.Entity<Student>().Property(s => s.Id).ValueGeneratedOnAdd();
17. builder.Entity<Student>().Property(s => s.Version).IsConcurrencyToken();
18. }
19. }
20. The Configure static function provides the [Fluent API](https://www.learnentityframeworkcore.com/configuration/fluent-api) functions that describe the table columns. This example describes the primary key (**Id**) and a field for catching concurrency errors (**Version**). The **IsConcurrencyToken** method is used to specify that a property should be included in a WHERE clause in an **UPDATE** or **DELETE** statement as part of [concurrency management](https://www.learnentityframeworkcore.com/concurrency). Some properties of the model can be described by [attributes](https://www.learnentityframeworkcore.com/configuration/data-annotation-attributes) but **FluentAPI** provides much more configuration options.
21. Now all that remains is to modify the **ApplicationDbContext** class: list all data sets and the Configure functions of these sets (the **Data/ApplicationDbContext.cs** file ):
22. public class ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) :
23. IdentityDbContext<ApplicationUser>(options)
24. {
25. public DbSet<Student> Students { get; set; }
26. protected override void OnModelCreating(ModelBuilder builder)
27. {
28. base.OnModelCreating(builder);
29. **Student.Configure**(builder);
30. }
31. }
32. With this data, you can use [data migration tools](https://www.infoworld.com/article/3691114/how-to-work-with-ef-core-migrations-in-aspnet-core.html) and create a new database.
33. Now install two packages: **Microsoft.VisualStudio.Web.CodeGeneration.Design** and **Microsoft.EntityFrameworkCore**.
34. If you don't want a type to be included in the model, you can exclude it:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.**Ignore**<BlogMetadata>();

}

With this configuration migrations will not create the AspNetUsers table, but IdentityUser is still included in the model and can be used normally.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<IdentityUser>()

.ToTable("AspNetUsers", t => t.ExcludeFromMigrations());

}

##### Listing migrations

You can list all existing migrations as follows:

dotnet ef migrations list

##### Remove a migration

Sometimes you add a migration and realize you need to make additional changes to your EF Core model before applying it. To remove the last migration, use this command:

remove-migration

After removing the migration, you can make the additional model changes and add it again. Avoid removing any migrations which have already been applied to production databases. Doing so means you won't be able to revert those migrations from the databases, and may break the assumptions made by subsequent migrations.

##### Add a migration

After your model has been changed, you can add a migration for that change:

Add-Migration AddBlogCreatedTimestamp

The migration name can be used like a commit message in a version control system. For example, you might choose a name like AddBlogCreatedTimestamp if the change is a new **CreatedTimestamp** property on your Blog entity.

Three files are added to your project under the Migrations directory:

* XXXXXXXXXXXXXX\_AddCreatedTimestamp.cs--The main migrations file. Contains the operations necessary to apply the migration (in Up) and to revert it (in Down).
* XXXXXXXXXXXXXX\_AddCreatedTimestamp.Designer.cs--The migrations metadata file. Contains information used by EF.
* MyContextModelSnapshot.cs--A snapshot of your current model. Used to determine what changed when adding the next migration.

The timestamp in the filename helps keep them ordered chronologically so you can see the progression of changes.

##### Resetting all migrations

In some extreme cases, it may be necessary to remove all migrations and start over. This can be easily done by deleting your **Migrations** folder and dropping your database; at that point you can create a new initial migration, which will contain your entire current schema.

It's also possible to reset all migrations and create a single one without losing your data. This is sometimes called "squashing", and involves some manual work:

1. Back up your database, in case something goes wrong.
2. In your database, delete all rows from the migrations history table (e.g. DELETE FROM [\_\_EFMigrationsHistory] on SQL Server).
3. Delete your Migrations folder.
4. Create a new migration and generate a SQL script for it (dotnet ef migrations script).
5. Insert a single row into the migrations history, to record that the first migration has already been applied, since your tables are already there. The insert SQL is the last operation in the SQL script generated above, and resembles the following (don't forget to update the values):

##### Table name

By convention, each entity type will be set up to map to a database table with the same name as the **DbSet** property that exposes the entity. If no DbSet exists for the given entity, the class name is used. You can manually configure the table name:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.**ToTable**("blogs");

}

##### Table schema

When using a relational database, tables are by convention created in your database's default schema. For example, Microsoft SQL Server will use the **dbo** schema (SQLite does not support schemas).You can configure tables to be created in a specific schema as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.**ToTable**("blogs", **schema**: "blogging");

}

Rather than specifying the schema for each table, you can also define the default schema at the model level with the fluent API:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.HasDefaultSchema("blogging");

}

##### View mapping

Entity types can be mapped to database views using the Fluent API:

modelBuilder.Entity<Blog>()

.**ToView**("blogsView", schema: "blogging");

EF will assume that the referenced view already exists in the database, it will not create it automatically in a migration. Mapping to a view will remove the default table mapping, but the entity type can also be mapped to a table explicitly. In this case the query mapping will be used for queries and the table mapping will be used for updates.

##### Table-valued function mapping

It's possible to map an entity type to a table-valued function (**TVF**) instead of a table in the database. Here is an example from the Microsoft site. Let's define entity that represents blog with multiple posts:

public class BlogWithMultiplePosts

{

public string Url { get; set; }

public int PostCount { get; set; }

}

Let’s create the following table-valued function in the database, which returns only blogs with multiple posts as well as the number of posts associated with each of these blogs:

CREATE FUNCTION dbo.BlogsWithMultiplePosts()

RETURNS TABLE

AS

RETURN

(

SELECT b.Url, COUNT(p.BlogId) AS PostCount

FROM Blogs AS b

JOIN Posts AS p ON b.BlogId = p.BlogId

GROUP BY b.BlogId, b.Url

HAVING COUNT(p.BlogId) > 1

)

Now, the entity BlogWithMultiplePosts can be mapped to this function in a following way:

modelBuilder.Entity<BlogWithMultiplePosts>()

.HasNoKey()

.**ToFunction**("BlogsWithMultiplePosts");

This entity type mapping may be used in a query:

var query = from b in context.Set<BlogWithMultiplePosts>()

where b.PostCount > 3

select new { b.Url, b.PostCount };

##### Table comments

You can set an arbitrary text comment that gets set on the database table, allowing you to document your schema in the database:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>().ToTable(

tableBuilder => tableBuilder.**HasComment**("Blogs managed on the website"));

}

##### Included and excluded properties

By convention, all public properties with a getter and a setter will be included in the model. Specific properties can be excluded as follows:

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public DateTime LoadedFromDatabase { get; set; }

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.**Ignore**(b => b.LoadedFromDatabase);

}

##### Column names

By convention, when using a relational database, entity properties are mapped to table columns having the same name as the property. If you prefer to configure your columns with different names, you can do so as following code snippet:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.BlogId)

.**HasColumnName**("blog\_id");

}

##### Column data types

When using a relational database, the database provider selects a data type based on the .NET type of the property. It also takes into account other metadata, such as the configured maximum length, whether the property is part of a primary key, etc. You can also configure your columns to specify an exact data type for a column:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>(

eb =>

{

eb.Property(b => b.Url).**HasColumnType**("varchar(200)");

eb.Property(b => b.Rating).**HasColumnType**("decimal(5, 2)");

});

}

##### Maximum length

Configuring a maximum length provides a hint to the database provider about the appropriate column data type to choose for a given property. Maximum length only applies to array data types, such as **string** and **byte[]**. Entity Framework does not do any validation of maximum length before passing data to the provider. It is up to the provider or data store to validate if appropriate.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Url)

.**HasMaxLength**(500);

}

##### Precision and Scale

Some relational data types support the precision and scale facets; these control what values can be stored, and how much storage is needed for the column. Which data types support precision and scale is database-dependent, but in most databases **decimal** and **DateTime** types do support these facets. For decimal properties, precision defines the maximum number of digits needed to express any value the column will contain, and scale defines the maximum number of decimal places needed. For DateTime properties, precision defines the maximum number of digits needed to express fractions of seconds, and scale is not used.

Entity Framework does not do any validation of precision or scale before passing data to the provider. It is up to the provider or data store to validate as appropriate.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Score)

.**HasPrecision**(14, 2);

modelBuilder.Entity<Blog>()

.Property(b => b.LastUpdated)

.**HasPrecision**(3);

}

##### Unicode

In some relational databases, different types exist to represent **Unicode** and **non-Unicode** text data. For example, in SQL Server, nvarchar(x) is used to represent Unicode data in UTF-16, while varchar(x) is used to represent non-Unicode data (but see the notes on SQL Server UTF-8 support). For databases which don't support this concept, configuring this has no effect.Text properties are configured as Unicode by default. You can configure a column as non-Unicode:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Book>()

.Property(b => b.Isbn)

.**IsUnicode(false)**;

}

SQL Server 2019 introduced introduced UTF-8 support, which allows storing UTF-8 data in char and varchar columns by configuring them with special UTF-8 collations:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Name)

.HasColumnType("varchar(max)")

.**UseCollation**("LATIN1\_GENERAL\_100\_CI\_AS\_SC\_UTF8")

.**IsUnicode**();

}

##### Required and optional properties

A property is considered optional if it is valid for it to contain null. If null is not a valid value to be assigned to a property then it is considered to be a required property. When mapping to a relational database schema, required properties are created as non-nullable columns, and optional properties are created as nullable columns.

By convention, a property whose .NET type can contain null will be configured as optional, whereas properties whose .NET type cannot contain null will be configured as required. For example, all properties with .NET value types (int, decimal, bool, etc.) are configured as required, and all properties with nullable .NET value types (int?, decimal?, bool?, etc.) are configured as optional.

C# 8 introduced a new feature called nullable reference types (NRT), which allows reference types to be annotated, indicating whether it is valid for them to contain null or not. This feature is enabled by default in new project templates, but remains disabled in existing projects unless explicitly opted into. You can fix this issue adding **<Nullable>enable</Nullable**> element in the project file:

<PropertyGroup>

<TargetFramework>net8.0</TargetFramework>

**<Nullable>enable</Nullable>**

<ImplicitUsings>enable</ImplicitUsings>

<UserSecretsId>aspnet-BlazorApp4-0c7655a7-1f05-43cd-921a-72502c56eb35</UserSecretsId>

</PropertyGroup>

Nullable reference types affect EF Core's behavior in the following way:

* If nullable reference types are disabled, all properties with .NET reference types are configured as optional by convention (for example, string).
* If nullable reference types are enabled, properties will be configured based on the C# nullability of their .NET type: **string?** will be configured as optional, but **string** will be configured as required.

public class Customer

{

public int Id { get; set; }

public string FirstName { get; set; } // Required by convention

public string LastName { get; set; } // Required by convention

public string? MiddleName { get; set; } // Optional by convention

// Note the following use of constructor binding, which avoids compiled warnings

// for uninitialized non-nullable properties.

public Customer(string firstName, string lastName, string? middleName = null)

{

FirstName = firstName;

LastName = lastName;

MiddleName = middleName;

}

}

A property that would be optional by convention can be configured to be required as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Url)

.**IsRequired()**;

}

##### Database collation

In most database systems, a default collation is defined at the database level; unless overridden, that collation implicitly applies to all text operations occurring within that database. The database collation is typically set at database creation time (via the **CREATE DATABASE** DDL statement), and if not specified, defaults to a some server-level value determined at setup time. For example, the default server-level collation in SQL Server for the "English (United States)" machine locale is **SQL\_Latin1\_General\_CP1\_CI\_AS**, which is a case-insensitive, accent-sensitive collation. Although database systems usually do permit altering the collation of an existing database, doing so can lead to complications; it is recommended to pick a collation before database creation.

When using EF Core migrations to manage your database schema, the following in your model's OnModelCreating method configures a SQL Server database to use a case-sensitive collation:

modelBuilder.UseCollation("SQL\_Latin1\_General\_CP1\_CS\_AS");

Collations can also be defined on text columns, overriding the database default.

modelBuilder.Entity<Customer>().Property(c => c.Name)

.UseCollation("SQL\_Latin1\_General\_CP1\_CI\_AS");

In some cases, the same column needs to be queried using different collations by different queries. This can be accomplished by explicitly specifying a collation within the query itself:

var customers = context.Customers

.Where(c => EF.Functions.Collate(c.Name, "SQL\_Latin1\_General\_CP1\_CS\_AS") == "John")

.ToList();

Specifying an explicit collation in a query will generally prevent that query from using an index defined on that column, since the collations would no longer match; it is therefore recommended to exercise caution when using this feature.

In .NET, string equality is case-sensitive by default: s1 == s2 performs an ordinal comparison that requires the strings to be identical. Because the default collation of databases varies, and because it is desirable for simple equality to use indexes, EF Core makes no attempt to translate simple equality to a database case-sensitive operation: C# equality is translated directly to SQL equality, which may or may not be case-sensitive, depending on the specific database in use and its collation configuration. In addition, .NET provides overloads of string.Equals accepting a StringComparison enum, which allows specifying case-sensitivity and a culture for the comparison. By design, EF Core refrains from translating these overloads to SQL.

##### Column comments

You can set an arbitrary text comment that gets set on the database column, allowing you to document your schema in the database:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Url)

.HasComment("The URL of the blog");

}

##### Column order

By default when creating a table with Migrations, EF Core orders primary key columns first, followed by properties of the entity type and owned types, and finally properties from base types. You can, however, specify a different column order:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Employee>(x =>

{

x.Property(b => b.Id)

.HasColumnOrder(0);

x.Property(b => b.FirstName)

.HasColumnOrder(1);

x.Property(b => b.LastName)

.HasColumnOrder(2);

});

}

##### Keys and indexes

A **key** serves as a unique identifier for each entity instance. Most entities in EF have a single key, which maps to the concept of a primary key in relational databases. By convention, a property named **Id** or **<type name>Id** will be configured as the primary key of an entity.

internal class Car

{

**public string Id { get; set; }** // Primary key

public string Make { get; set; }

public string Model { get; set; }

}

internal class Truck

{

**public string TruckId { get; set; }** // Primary key

public string Make { get; set; }

public string Model { get; set; }

}

**Owned entity type**s use different rules to define keys. You can configure a single property to be the primary key of an entity as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Car>()

.**HasKey(c => c.LicensePlate)**;

}

The key may consists of multiple properties - this is known as a composite key.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Car>()

.**HasKey(c => new { c.State, c.LicensePlate })**;

}

By convention, on relational databases primary keys are created with the name **PK\_<type name>**. You can configure the name of the primary key constraint as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasKey(b => b.BlogId)

.**HasName("PrimaryKey\_BlogId")**;

}

Relational database tables have only one **primary key**, but any number of **unique constraints** may be defined there:

protected override void OnModelCreating(ModelBuilder builder)

{

builder.Entity<User>()

.HasIndex(u => u.Email)

.IsUnique();

}

An index can also span more than one column:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Person>()

.HasIndex(p => **new { p.FirstName, p.LastName }**);

}

By default, indexes aren't unique: multiple rows are allowed to have the same value(s) for the index's column set. You can make an index unique as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasIndex(b => b.Url)

.**IsUnique()**;

}

In most databases, each column covered by an index can be either ascending or descending. For indexes covering only one column, this typically does not matter: the database can traverse the index in reverse order as needed. However, for composite indexes, the ordering can be crucial for good performance, and can mean the difference between an index getting used by a query or not. In general, the index columns' sort orders should correspond to those specified in the **ORDER B**Y clause of your query.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasIndex(b => new { b.Url, b.Rating })

.**IsDescending()**;

}

You may also specify the sort order on a column-by-column basis as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasIndex(b => new { b.Url, b.Rating })

.IsDescending(false, true);

}

By convention, indexes created in a relational database are named **IX\_<type name>\_<property name>**. For composite indexes, <property name> becomes an underscore separated list of property names. You can set the name of the index created in the database:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasIndex(b => b.Url)

.HasDatabaseName("Index\_Url");

}

If you call HasIndex more than once on the same set of properties, that continues to configure a single index rather than create a new one:

modelBuilder.Entity<Blog>()

.HasIndex(b => new { b.FirstName, b.LastName })

.HasDatabaseName("IX\_Names\_Ascending");

modelBuilder.Entity<Blog>()

.HasIndex(b => new { b.FirstName, b.LastName })

.HasDatabaseName("IX\_Names\_Descending")

.IsDescending();

Since the second HasIndex call overrides the first one, this creates only a single, descending index.

To create multiple indexes over the same set of properties, pass a name to the HasIndex, which will be used to identify the index in the EF model, and to distinguish it from other indexes over the same properties:

modelBuilder.Entity<Blog>()

.HasIndex(b => new { b.FirstName, b.LastName }, "IX\_Names\_Ascending");

modelBuilder.Entity<Blog>()

.HasIndex(b => new { b.FirstName, b.LastName }, "IX\_Names\_Descending")

.IsDescending();

Note that this name is also used as a default for the database name, so explicitly calling **HasDatabaseName** isn't required.

By default, the Fluent API for EF Core adds

filter: "[ProductId] IS NOT NULL"

to the index created in the migration. In order to ensure that even NULL is unique, we have to modify our index, like so

config.Entity<Product>()

.HasIndex(b => b.ProductId)

.IsUnique()

.HasFilter(null)

This removes the filter, and allows NULL to be unique.

##### Keyless Entity Types

In addition to regular entity types, an EF Core model can contain keyless entity types, which can be used to carry out database queries against data that doesn't contain key values.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<BlogPostsCount>()

.HasNoKey();

}

Keyless entity types support many of the same mapping capabilities as regular entity types, like inheritance mapping and navigation properties. On relational stores, they can configure the target database objects and columns via fluent API methods or data annotations. However, they are different from regular entity types in that they:

* Cannot have a key defined.
* Are never tracked for changes in the DbContext and therefore are never inserted, updated or deleted on the database.
* Are never discovered by convention.
* Only support a subset of navigation mapping capabilities, specifically:
  + They may never act as the principal end of a relationship.
  + They may not have navigations to owned entities
  + They can only contain reference navigation properties pointing to regular entities.
  + Entities cannot contain navigation properties to keyless entity types.
* Need to be configured with a [**Keyless**] data annotation or a .**HasNoKey**() method call.
* May be mapped to a defining query. A defining query is a query declared in the model that acts as a data source for a keyless entity type.
* Can have a hierarchy, but it must be mapped as TPH (object-oriented hierarchy).
* Cannot use table splitting or entity splitting.

//[Indexes - EF Core | Microsoft Learn](https://learn.microsoft.com/en-us/ef/core/modeling/indexes?tabs=fluent-api)

##### Value generation

For non-composite numeric and GUID primary keys, EF Core sets up value generation for you by convention. For example, a numeric primary key in SQL Server is automatically set up to be an **IDENTITY** column. Generated values are described on [Microsoft](https://learn.microsoft.com/en-us/ef/core/modeling/generated-properties?tabs=data-annotations) site.

**EF Core** automatically sets up value generation for primary keys - but we may want to do the same for non-key properties. You can configure any property to have its value generated for inserted entities as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Inserted)

.ValueGeneratedOnAdd();

}

Similarly, a property can be configured to have its value generated on add or update:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.LastUpdated)

.ValueGeneratedOnAddOrUpdate();

}

Configuring a date/time column to have the creation timestamp of the row is usually a matter of configuring a default value with the appropriate SQL function. For example, on SQL Server you may use the following:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Created)

.HasDefaultValueSql("getdate()");

}

Be sure to select the appropriate function, as several may exist (e.g. **GETDATE()** vs. **GETUTCDATE()**).

Although stored computed columns seem like a good solution for managing last-updated timestamps, databases usually don't allow specifying functions such as GETDATE() in a computed column. As an alternative, you can set up a database trigger to achieve the same effect:

// SQL

CREATE TRIGGER [dbo].[Blogs\_UPDATE] ON [dbo].[Blogs]

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

IF ((SELECT TRIGGER\_NESTLEVEL()) > 1) RETURN;

UPDATE B

SET LastUpdated = GETDATE()

FROM dbo.Blogs AS B

INNER JOIN INSERTED AS I

ON B.BlogId = I.BlogId

END

For information on creating triggers, see the documentation on using [raw SQL](https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/managing?tabs=dotnet-core-cli" \l "adding-raw-sql) in migrations.

##### Primary key name

By convention, on relational databases primary keys are created with the name **PK\_<type name>**. You can configure the name of the primary key constraint as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasKey(b => b.BlogId)

.HasName("PrimaryKey\_BlogId");

}

##### Key types and values

While EF Core supports using properties of any primitive type as the primary key, including **string**, **Guid**, **byte[]** and others, not all databases support all types as keys. In some cases the key values can be converted to a supported type automatically, otherwise the conversion should be specified [manually](https://learn.microsoft.com/en-us/ef/core/modeling/value-conversions?tabs=data-annotations).

##### Overriding value generation

Key properties must always have a non-default value when adding a new entity to the context, but some types will be generated by the database. In that case EF will try to generate a temporary value when the entity is added for tracking purposes. After SaveChanges is called the temporary value will be replaced by the value generated by the database.

If a key property has its value generated by the database and a non-default value is specified when an entity is added, then EF will assume that the entity already exists in the database and will try to update it instead of inserting a new one. To provide an explicit value for properties that have been configured as value generated on add or update, you must also configure the property as follows:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>().Property(b => b.LastUpdated)

.ValueGeneratedOnAddOrUpdate()

.Metadata.SetAfterSaveBehavior(PropertySaveBehavior.Save);

}

Trying to insert explicit values into SQL Server IDENTITY fails by default; see these [docs](https://learn.microsoft.com/en-us/ef/core/providers/sql-server/value-generation?tabs=data-annotations" \l "inserting-explicit-values-into-identity-columns) for a workaround.

##### No value generation

In some cases you may want to disable value generation that has been set up by convention. For example, a primary key of type int is usually implicitly configured as value-generated-on-add (e.g. identity column on SQL Server). You can disable this via the following:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.BlogId)

.**ValueGeneratedNever()**;

}

##### Default values

On relational databases, a column can be configured with a default value; if a row is inserted without a value for that column, the default value will be used.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Rating)

.**HasDefaultValue(3)**;

}

You can also specify a SQL fragment that is used to calculate the default value:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.Property(b => b.Created)

.**HasDefaultValueSql**("getdate()");

}

##### Computed columns

On most relational databases, a column can be configured to have its value computed in the database, typically with an expression referring to other columns:

modelBuilder.Entity<Person>()

.Property(p => p.DisplayName)

.HasComputedColumnSql("[LastName] + ', ' + [FirstName]");

The above creates a virtual computed column, whose value is computed every time it is fetched from the database. You may also specify that a computed column be stored (sometimes called persisted), meaning that it is computed on every update of the row, and is stored on disk alongside regular columns:

modelBuilder.Entity<Person>()

.Property(p => p.NameLength)

.HasComputedColumnSql("LEN([LastName]) + LEN([FirstName])", stored: true);

##### Shadow Properties

Shadow properties are the properties that are not defined in your **.NET** entity class directly; instead, you configure them for the particular entity type in the entity data model. They can be configured in the **OnModelCreating()** method of the context class.

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public DateTime? DateOfBirth { get; set; }

public decimal Height { get; set; }

public float Weight { get; set; }

}

The Student class does not include CreatedDate and UpdatedDate properties to maintain **created** or **updated** time. Let configure them as shadow properties on the Student entity:

public class SchoolContext : DbContext

{

public SchoolContext() : base()

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder) {

**modelBuilder.Entity<Student>().Property<DateTime>("CreatedDate");**

**modelBuilder.Entity<Student>().Property<DateTime>("UpdatedDate");**

}

public DbSet<Student> Students { get; set; }

}

If the name specified in the Property() method matches the name of an existing property, then the EF Core will configure that existing property as a shadow property rather than introducing a new shadow property. Database will have fields **CreatedDate**, **UpdatedDate** after executing the following commands:

PM> add-migration <migration-name>

PM> update-database

You can get or set the values of the shadow properties using the **Property()** method of **EntityEntry**. The following code access the value of the shadow property:

using (var context = new SchoolContext())

{

var std = new Student(){ StudentName = "Bill" };

// sets the value to the shadow property

context.Entry(std).Property("CreatedDate").CurrentValue = DateTime.Now;

// gets the value of the shadow property

var createdDate = context.Entry(std).Property("CreatedDate").CurrentValue;

}

It is to set the value to these shadow properties automatically on the SaveChanges() method, so that we don't have to set them manually on each entity object:

public override int SaveChanges() {

var entries = ChangeTracker

.Entries()

.Where(e =>

e.State == EntityState.Added

|| e.State == EntityState.Modified);

foreach (var entityEntry in entries) {

entityEntry.Property("UpdatedDate").CurrentValue = DateTime.Now;

if (entityEntry.State == EntityState.Added) {

entityEntry.Property("CreatedDate").CurrentValue = DateTime.Now;

}

}

return base.SaveChanges();

}

You can configure shadow properties on all entities at once, rather than configuring them manually for all:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

var allEntities = modelBuilder.Model.GetEntityTypes();

foreach (var entity in allEntities)

{

**entity.AddProperty("CreatedDate",typeof(DateTime));**

**entity.AddProperty("UpdatedDate",typeof(DateTime));**

}

}

Shadow properties are most often used for foreign key properties, where they are added to the model by convention when no foreign key property has been found by convention or configured explicitly. The relationship is represented by navigation properties, but in the database it is enforced by a foreign key constraint, and the value for the foreign key column is stored in the corresponding shadow property.

The property will be named <navigation property name><principal key property name> (the navigation on the dependent entity, which points to the principal entity, is used for the naming). If the principal key property name starts with the name of the navigation property, then the name will just be <principal key property name>. If there is no navigation property on the dependent entity, then the principal type name is used in its place. For example, the following code listing will result in a **BlogId** shadow property being introduced to the Post entity:

internal class MyContext : DbContext

{

public DbSet<Blog> Blogs { get; set; }

public DbSet<Post> Posts { get; set; }

}

public class Blog

{

public int BlogId { get; set; }

public string Url { get; set; }

public List<Post> Posts { get; set; }

}

public class Post

{

public int PostId { get; set; }

public string Title { get; set; }

public string Content { get; set; }

**// Since there is no CLR property which holds the foreign**

**// key for this relationship, a shadow property is created.**

public Blog Blog { get; set; }

}

##### Indexers

An indexer is a special type of property that allows a class or a structure to be accessed like an array for its internal collection. C# allows us to define custom indexers, generic indexers, and also overload indexers.

An indexer can be defined the same way as property with this keyword and square brackets []:

[access\_modifier] [return\_type] this **[**argument\_list**]**

{

get

{

// get block code

}

set

{

// set block code

}

}

In the above syntax:

* emhesized brackets are part of the syntax,
* access\_modifier: It can be **public**, **private**, **protected** or **internal**.
* return\_type: It can be any valid C# type; you can use type parameter of the class here.
* this: It is the keyword which points to the object of the current class.
* argument\_list: This specifies the parameter list of the indexer.
* get{ } and set { }: These are the accessors.

class SampleCollection<T>

{

// Declare an array to store the data elements.

private T[] arr = new T[100];

// Define the indexer to allow client code to use [] notation.

**public T this[int i]**

{

get { return arr[i]; }

set { arr[i] = value; }

}

}

class Program

{

static void Main()

{

var stringCollection = new SampleCollection<string>();

stringCollection[0] = "Hello, World";

Console.WriteLine(stringCollection[0]);

}

}

// The example displays the following output:

// Hello, World.

It is common for an indexer's get or set accessor to consist of a single statement that either returns or sets a value. Expression-bodied members provide a simplified syntax to support this scenario. A read-only indexer can be implemented as an expression-bodied member, as the following example shows.

using System;

class SampleCollection<T>

{

// Declare an array to store the data elements.

private T[] arr = new T[100];

int nextIndex = 0;

// Define the indexer to allow client code to use [] notation.

**public T this[int i] => arr[i];**

public void Add(T value)

{

if (nextIndex >= arr.Length)

throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");

arr[nextIndex++] = value;

}

}

class Program

{

static void Main()

{

var stringCollection = new SampleCollection<string>();

stringCollection.Add("Hello, World");

System.Console.WriteLine(stringCollection[0]);

}

}

// The example displays the following output:

// Hello, World.

Note that => introduces the expression body, and that the get keyword is not used. Both the **get** and **set** accessor can be implemented as expression-bodied members. In this case, both get and set keywords must be used. For example:

using System;

class SampleCollection<T>

{

// Declare an array to store the data elements.

private T[] arr = new T[100];

// Define the indexer to allow client code to use [] notation.

public T this[int i]

{

get => arr[i];

set => arr[i] = value;

}

}

class Program

{

static void Main()

{

var stringCollection = new SampleCollection<string>();

stringCollection[0] = "Hello, World.";

Console.WriteLine(stringCollection[0]);

}

}

// The example displays the following output:

// Hello, World.

Indexers can be declared on an interface. Accessors of interface indexers differ from the accessors of class indexers in the following ways:

* Interface accessors do not use modifiers.
* An interface accessor typically does not have a body.

The purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only. You may provide an implementation for an indexer defined in an interface, but this is rare. Indexers typically define an API to access data fields, and data fields cannot be defined in an interface.

public interface ISomeInterface

{

//...

// Indexer declaration:

**string this[int index]**

{

get;

set;

}

}

EF Core migration creates correct structure of the table with indexer. Consider class blog:

public class Blog

{

private readonly Dictionary<string, object> \_data =

new Dictionary<string, object>();

public int BlogId { get; set; }

public string? Message { get; set; }

public object this[string key]

{

get => \_data[key];

set => \_data[key] = value;

}

public static void Configure(ModelBuilder builder)

{

builder.Entity<Blog>(entity =>

{

entity.HasKey(s => s.BlogId);

entity.Property(s => s.BlogId).ValueGeneratedOnAdd();

entity.IndexerProperty<string>("Author");

});

}

public string[] AllKeys => \_data.Keys.ToArray();

}

A function that returns all keys

public string[] AllKeys => \_data.Keys.ToArray();

is needed in the case when the indexer is not of integer type. When the indexer is of integer type and its values are 0, 1, 2, ... it is sufficient to return the number of elements in the dictionary \_data. You can also implement the **IEnumerable** interface. The structure of the table will not change, the data reading algorithm will remain the same, but you will be able to use the **foreach** loop when processing loaded data.

Migration will create table with columns **BlogId**, **Message**, **Author,** but column **Auhor** is a shadow property. Entity Blog has no this field. You can write into the dbo.Blogs all columns using raw SQL:

public void AddBlog(MyContext ctx, string message, string author) {

const string sql =

"insert into dbo.Blogs (Message, Author) values (@msq, @author);";

var p\_msq = new SqlParameter("@msq", message);

var p\_author = new SqlParameter("@author", author);

ctx.Database.**ExecuteSqlRaw**(sql, p\_msq, p\_author);

}

Expression **ctx.Blogs.AsNoTracking().ToList()** reads all fields fro database but field "Author" is a shadow property and it is not visible. You need to get value of this field using properties.

public Blog ReadBlogs(MyContext ctx) {

Blog rzlt = new Blog();

**var blogs = ctx.Blogs.AsNoTracking().ToList();**

foreach (var blog in blogs)

{

**var x = ctx.Entry<Blog>(blog).Property("Author").CurrentValue;**

if (x != null)

{

rzlt[(string)x] = blog;

}

}

return rzlt;

}

You can use the following template for processing the scanned data:

// allBlogs is object of a Blog type

foreach (string aKey in allBlogs.AllKeys)

{

Blog current = (Blog)allBlogs[aKey];

if (current != null)

{

Console.WriteLine($"{aKey} => {current.BlogId}\t{current.Message}");

}

}

#### Concurrency tokens

The IsConcurrencyToken method is used to specify that a property should be included in a WHERE clause in an UPDATE or DELETE statement as part of concurrency management.

public class SampleContext : DbContext

{

public DbSet<Author> Authors { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Author>()

.**Property(a => a.FirstName).IsConcurrencyToken()**

modelBuilder.Entity<Author>()

.**Property(a => a.LastName).IsConcurrencyToken()**;

}

}

public class Author

{

public int AuthorId { get; set; }

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

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

public string Biography { get; set; }

public ICollection<Book> Books { get; set; }

}

When applied to a byte array property in combination with the **ValueGeneratedOnAddOrUpdate** method, the **IsConcurrencyToken** method denotes that the property should map to a database type that provides automatic row-versioning, such as the SQL Server **rowversion** type:

public class SampleContext : DbContext

{

public DbSet<Author> Authors { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Author>()

.**Property(a => a.RowVersion)**

.**IsConcurrencyToken()**

.**ValueGeneratedOnAddOrUpdate()**;

}

}

public class Author

{

public int AuthorId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public string Biography { get; set; }

public ICollection<Book> Books { get; set; }

**public byte[] RowVersion { get; set; }**

}

Rather than have the database manage the concurrency token automatically, you can manage it in application code. This allows using optimistic concurrency on databases - like SQLite - where no native automatically-updating type exists. But even on SQL Server, an application-managed concurrency token can provide fine-grained control on exactly which column changes cause the token to be regenerated.

public class Person

{

public int PersonId { get; set; }

public string FirstName { get; set; }

**public Guid Version { get; set; }**

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Person>()

**.Property(p => p.Version)**

**.IsConcurrencyToken();**

}

Since this property isn't database-generated, you must assign it in application whenever persisting changes:

var person = context.People.Single(b => b.FirstName == "John");

person.FirstName = "Paul";

person.Version = Guid.NewGuid();

context.SaveChanges;

#### Relations

Relational databases represent relationships using foreign keys. For example, using SQL Server or Azure SQL, the following tables can be used to represent our Post and Blog classes:

CREATE TABLE [Posts] (

[Id] int NOT NULL IDENTITY,

[Title] nvarchar(max) NULL,

[Content] nvarchar(max) NULL,

[PublishedOn] datetime2 NOT NULL,

[Archived] bit NOT NULL,

[BlogId] int NOT NULL,

CONSTRAINT [PK\_Posts] PRIMARY KEY ([Id]),

CONSTRAINT [FK\_Posts\_Blogs\_BlogId] FOREIGN KEY ([BlogId]) REFERENCES [Blogs] ([Id]) ON DELETE CASCADE);

CREATE TABLE [Blogs] (

[Id] int NOT NULL IDENTITY,

[Name] nvarchar(max) NULL,

[SiteUri] nvarchar(max) NULL,

CONSTRAINT [PK\_Blogs] PRIMARY KEY ([Id]));

In this relational model, the Posts and Blogs tables are each given a "**primary key**" column. The value of the primary key uniquely identifies each post or blog. In addition, the Posts table is given a "**foreign key**" column. The Blogs primary key column Id is referenced by the BlogId foreign key column of the Posts table. This column is "constrained" such that any value in the BlogId column of Posts must match a value in the Id column of Blogs. This match determines which blog every post is related to. For example, if the BlogId value in one row of the Posts table is 7, then the post represented by that row is published in the blog with the primary key 7.

SQL uses these keys to creating joins:

select b.\*, p.\*

from Blogs b

**join** Posts p on p.BlogId = b.Id;

**ORM** tools create classes with cross references and configure these relationships with **Fluent API** tools. The structure and configuration of classes depends on the type of connection, so we will consider them separately.

##### One-to-many relationships

**One-to-many** relationships are used when a single entity is associated with any number of other entities. For example, a **Blog** can have many associated **Posts**, but each **Post** is associated with only one **Blog**. EF maps this relation into the two navigation properties:

// Principal (parent)

public class Blog

{

public int Id { get; set; } // Primary or unique key

// Collection navigation containing dependents

**public ICollection<Post> Posts { get; } = new List<Post>();**

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

**public int BlogId { get; set; }**  // Required foreign key property

**public Blog Blog { get; set; } = null!;** // Required reference navigation to principal

}

A one-to-many relationship is made up from:

* One or more **primary** or **alternate key** properties on the principal entity; that is the "one" end of the relationship. For example, **Blog.Id**.
* One or more **foreign key** properties on the dependent entity; that is the "many" end of the relationship. For example, **Post.BlogId**.
* Optionally, a collection navigation on the principal entity referencing the dependent entities. For example, **Blog.Posts**.
* Optionally, a reference navigation on the dependent entity referencing the principal entity. For example, **Post.Blog**.

**One-to-many** relation can be configured as from the **Principal** (entity Blog) side:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<**Blog**>()

.HasMany(b => b.**Posts**)

.WithOne(p => p.**Blog**)

.HasForeignKey(p => p.**BlogId**)

.IsRequired();

}

and also from the Dependent (entity Post) side:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<**Post**>()

.HasOne(p => p.**Blog**)

.WithMany(b => b.**Posts**)

.HasForeignKey(p => p.**BlogId**)

.IsRequired();

}

Neither of these options is better than the other; they both result in exactly the same configuration. It is never necessary to configure a relationship twice, once starting from the principal, and then again starting from the dependent. Also, attempting to configure the principal and dependent halves of a relationship separately generally does not work. Choose to configure each relationship from either one end or the other and then write the configuration code only once.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

// Collection navigation containing dependents

**public ICollection<Post> Posts { get; } = new List<Post>()**;

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

**public int? BlogId { get; set; }** // Optional foreign key property

**public Blog? Blog { get; set; }** // Optional reference navigation to principal

}

This is the same as the previous example, except that the foreign key property and navigation to the principal are now **nullable**. This makes the relationship "**optional**" because a dependent (**Post**) can exist without being related to any principal (Blog).

When using C# nullable reference types, the reference navigation must be nullable if the foreign key property is nullable. In this case, Post.BlogId is nullable, so Post.Blog must be nullable too. See Working with [Nullable Reference Types](https://learn.microsoft.com/en-us/ef/core/miscellaneous/nullable-reference-types) for more information.

Optional navigation may be configured by this template:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany(e => e.Posts)

.WithOne(e => e.Blog)

.HasForeignKey(e => e.BlogId)

.IsRequired(false);

}

The following example creates a required one-to-many relationship with a shadow foreign key (**BlogId**):

// Principal (parent)

public class Blog

{

public int Id { get; set; }

// Collection navigation containing dependents

public ICollection<Post> Posts { get; } = new List<Post>();

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

**public Blog Blog { get; set; } =** **null!**; // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany(e => e.Posts)

.WithOne(e => e.Blog)

.**HasForeignKey("BlogId")**

.**IsRequired()**;

}

Next example creates otional relation:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

// Collection navigation containing dependents

public ICollection<Post> Posts { get; } = new List<Post>();

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

**public Blog? Blog { get; set; }** // Optional reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany(e => e.Posts)

.WithOne(e => e.Blog)

**.HasForeignKey("BlogId")**

**.IsRequired(false);**

}

The next example creates a unidirectional relation:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

// Collection navigation containing dependents

**public ICollection<Post> Posts { get; } = new List<Post>();**

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

**public int BlogId { get; set; }** // Required foreign key property

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany(e => e.Posts)

.**WithOne()**

.**HasForeignKey(e => e.BlogId)**

.**IsRequired()**;

}

Notice that the call to **WithOne** has no arguments. This is the way to tell EF that there is no navigation from Post to Blog.

It is possible to configure **One-to-many** without navigation to principal and with **shadow foreign key**:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

**public ICollection<Post> Posts { get; } = new List<Post>()**;

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany(e => e.Posts)

.**WithOne()**

.**HasForeignKey("BlogId")**

.IsRequired();

}

The previous two examples had navigations from the principal to dependents, but no navigation from the dependent to principal. For the next couple of examples, the navigation on the dependent is re-introduced, while the navigation on the principal is removed instead.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

public int BlogId { get; set; } // Required foreign key property

**public Blog Blog { get; set; } = null!;** // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.**HasOne(e => e.Blog)**

.**WithMany()**

.**HasForeignKey(e => e.BlogId)**

.IsRequired();

}

Notice again that WithMany() is called with no arguments to indicate that there is no navigation in this direction.

If configuration starts from the entity with no navigation, then the type of the entity on the other end of the relationship must be explicitly specified using the generic **HasMany<>()** call.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.**HasMany<Post>()**

.**WithOne(e => e.Blog)**

.**HasForeignKey(e => e.BlogId)**

.IsRequired();

}

Occasionally, it can be useful to configure a relationship with no navigations. Such a relationship can only be manipulated by changing the foreign key value directly.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

public int BlogId { get; set; } // Required foreign key property

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany<Post>()

.WithOne()

.HasForeignKey(e => e.BlogId)

.IsRequired();

}

In all the examples so far, the foreign key property on the dependent is constrained to the primary key property on the principal. The foreign key can instead be constrained to a different property, which then becomes an alternate key for the principal entity type. For example:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

// Alternate key as target of the Post.BlogId foreign key

**public int AlternateId { get; set; }**

// Collection navigation containing dependents

**public ICollection<Post> Posts { get; } = new List<Post>();**

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

**public int BlogId { get; set; }** // Required foreign key property

**public Blog Blog { get; set; } = null!;** // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany(e => e.Posts)

.WithOne(e => e.Blog)

.**HasPrincipalKey(e => e.AlternateId)**

.**HasForeignKey(e => e.BlogId)**

.IsRequired();

}

In all the examples so far, the primary or alternate key property of the principal consisted of a single property. Primary or alternate keys can also be formed form more than one property--these are known as "composite keys". When the principal of a relationship has a composite key, then the foreign key of the dependent must also be a composite key with the same number of properties. The composite key itself needs to be configured explicitly:

// Principal (parent)

public class Blog

{

public int Id1 { get; set; } // Composite key part 1

public int Id2 { get; set; } // Composite key part 2

// Collection navigation containing dependents

public ICollection<Post> Posts { get; } = new List<Post>();

}

// Dependent (child)

public class Post

{

public int Id { get; set; }

**public int BlogId1 { get; set; } // Required foreign key property part 1**

**public int BlogId2 { get; set; } // Required foreign key property part 2**

public Blog Blog { get; set; } = null!; // Required reference navigation to principal

}

…

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>(

nestedBuilder =>

{

**nestedBuilder.HasKey(e => new { e.Id1, e.Id2 });**

nestedBuilder.HasMany(e => e.Posts)

.WithOne(e => e.Blog)

.**HasPrincipalKey(e => new { e.Id1, e.Id2 })**

.**HasForeignKey(e => new { e.BlogId1, e.BlogId2 })**

.IsRequired();

});

}

By convention, required relationships are configured to **cascade delete**; this means that when the principal is deleted, all of its dependents are deleted as well, since dependents cannot exist in the database without a principal. It's possible to configure EF to throw an exception instead of automatically deleting dependent rows that can no longer exist:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasMany(e => e.Posts)

.WithOne(e => e.Blog)

.**OnDelete(DeleteBehavior.Restrict)**;

}

In all the previous examples, the principal entity type was different from the dependent entity type. This does not have to be the case. For example, in the types below, each Employee is related to other Employees.

public class Employee

{

public int Id { get; set; }

**public int? ManagerId { get; set; }** // Optional foreign key property

**public Employee? Manager { get; set; }** // Optional reference navigation to principal

// Collection navigation containing dependents

**public ICollection<Employee> Reports { get; } = new List<Employee>();**

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Employee>()

.**HasOne(e => e.Manager)**

.**WithMany(e => e.Reports)**

.**HasForeignKey(e => e.ManagerId)**

.IsRequired(false);

}

##### One-to-one relationships

One-to-one relationships are used when one entity is associated with at most one other entity. For example, a Blog has one BlogHeader, and that BlogHeader belongs to a single Blog.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

**public BlogHeader? Header { get; set; }** // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public int BlogId { get; set; }** // Required foreign key property

**public Blog Blog { get; set; } = null!;** // Required reference navigation to principal

}

…

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.HasForeignKey<BlogHeader>(e => e.BlogId)

.IsRequired();

}

A one-to-one relationship is made up from:

* One or more primary or alternate key properties on the **principal** entity. For example, **Blog.Id**.
* One or more foreign key properties on the dependent entity. For example, **BlogHeader.BlogId**.
* Optionally, a reference navigation on the principal entity referencing the dependent entity. For example, **Blog.Header**.
* Optionally, a reference navigation on the dependent entity referencing the principal entity. For example, **BlogHeader.Blog**.

It is not always obvious which side of a one-to-one relationship should be the principal, and which side should be the dependent. Some considerations are:

* If the database tables for the two types already exist, then the table with the foreign key column(s) must map to the dependent type.
* A type is usually the dependent type if it cannot logically exist without the other type. For example, it makes no sense to have a header for a blog that does not exist, so BlogHeader is naturally the dependent type.
* If there is a natural parent/child relationship, then the child is usually the dependent type.

A required relationship ensures that every dependent entity must be associated with some principal entity. However, a principal entity can always exist without any dependent entity.

In the example above, configuration of the relationships starts the principal entity type (**Blog**). As with all relationships, it is exactly equivalent to start with dependent entity type (**BlogHeader**) instead:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<BlogHeader>()

.HasOne(e => e.Blog)

.WithOne(e => e.Header)

.HasForeignKey<BlogHeader>(e => e.BlogId)

.IsRequired();

}

Neither of these options is better than the other; they both result in exactly the same configuration. It is never necessary to configure a relationship twice, once starting from the principal, and then again starting from the dependent. Choose to configure each relationship from either one end or the other and then write the configuration code only **once**.

This example defines **optional** one-to-one relation:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

public BlogHeader? Header { get; set; } // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public int? BlogId { get; set; }** // Optional foreign key property

**public Blog? Blog { get; set; }** // Optional reference navigation to principal

}

…

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**HasForeignKey<BlogHeader>(e => e.BlogId)**

.**IsRequired(false)**;

}

This is the same as the previous example, except that the foreign key property and navigation to the principal are now nullable. This makes the relationship "optional" because a dependent (**BlogHeader**) can not be related to any principal (**Blog**) by setting its foreign key property and navigation to null. When using C# nullable reference types, the navigation property from dependent to principal must be nullable if the foreign key property is nullable. In this case, **BlogHeader.BlogId** is nullable, so **BlogHeader.Blog** must be nullable too.

Unlike with one-to-many relationships, the dependent end of a one-to-one relationship may use its primary key property or properties as the foreign key property or properties. This is often called a PK-to-PK relationship. This is only possible when the principal and dependent types have the same primary key types, and the resulting relationship is always required, since the primary key of the dependent cannot be nullable.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

**public BlogHeader? Header { get; set; }** // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public Blog Blog { get; set; } = null!**; // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**HasForeignKey<BlogHeader>(e => e.Id)**

.IsRequired();

}

When no property is specified in the call to HasForeignKey, and the primary key is suitable, then it is used as the foreign key.

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**HasForeignKey<BlogHeader>()**; // Foreign key is BlogHeader.Id

}

In some cases, you may not want a foreign key property in your model, since foreign keys are a detail of how the relationship is represented in the database, which is not needed when using the relationship in a purely object-oriented manner. Foreign key properties can be private, which is often a good compromise to avoid exposing the foreign key while allowing its value to travel with the entity. This example removes the foreign key property from the dependent entity type. However, instead of using the primary key, EF is instead instructed to create a **shadow foreign key** property called **BlogId** of type **int**:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

public BlogHeader? Header { get; set; } // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

public Blog Blog { get; set; } = null!; // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**HasForeignKey<BlogHeader>("BlogId")**

.**IsRequired()**;

}

It is possible to build optional one-one realation:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

public BlogHeader? Header { get; set; } // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public Blog? Blog { get; set; }** // Optional reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**HasForeignKey<BlogHeader>("BlogId")**

.**IsRequired(false)**;

}

For this example, the foreign key property has been re-introduced, but the navigation on the dependent has been removed:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

public BlogHeader? Header { get; set; } // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public int BlogId { get; set; }** // Required foreign key property

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.**WithOne()**

.**HasForeignKey<BlogHeader>(e => e.BlogId)**

.**IsRequired()**;

}

If configuration starts from the entity with no navigation, then the type of the entity on the other end of the relationship must be explicitly specified using the generic HasOne<>() call:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<BlogHeader>()

.**HasOne<Blog>()**

.WithOne(e => e.Header)

.**HasForeignKey<BlogHeader>(e => e.BlogId)**

.IsRequired();

}

It is possible to create One-to-one relation without navigation to principal and with shadow foreign key:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

**public BlogHeader? Header { get; set; }** // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.**HasOne(e => e.Header)**

.**WithOne()**

.**HasForeignKey<BlogHeader>("BlogId")**

.IsRequired();

}

For the next couple of examples, the navigation on the dependent is re-introduced, while the navigation on the principal is removed instead.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public int BlogId { get; set; }** // Required foreign key property

**public Blog Blog { get; set; } = null!;** // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<**BlogHeader**>()

.**HasOne(e => e.Blog)**

.**WithOne()**

.**HasForeignKey<BlogHeader>(e => e.BlogId)**

.**IsRequired()**;

}

Occasionally, it can be useful to configure a relationship with no navigations. Such a relationship can only be manipulated by changing the foreign key value directly.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

public int BlogId { get; set; } // Required foreign key property

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<**Blog**>()

.**HasOne<BlogHeader>()**

.**WithOne()**

.**HasForeignKey<BlogHeader>(e => e.BlogId)**

.IsRequired();

}

In all the examples so far, the foreign key property on the dependent is constrained to the primary key property on the principal. The foreign key can instead be constrained to a different property, which then becomes an alternate key for the principal entity type.

// Principal (parent)

public class Blog

{

public int Id { get; set; }

// Alternate key as target of the BlogHeader.BlogId foreign key

**public int AlternateId { get; set; }**

public BlogHeader? Header { get; set; } // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public int BlogId { get; set; }** // Required foreign key property

public Blog Blog { get; set; } = null!; // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**HasPrincipalKey<Blog>(e => e.AlternateId)**

.**HasForeignKey<BlogHeader>(e => e.BlogId)**

.IsRequired();

}

Primary or alternate keys can also be formed form more than one property--these are known as "**composite keys**". When the principal of a relationship has a composite key, then the foreign key of the dependent must also be a composite key with the same number of properties.

// Principal (parent)

public class Blog

{

**public int Id1 { get; set; }** // Composite key part 1

**public int Id2 { get; set; }** // Composite key part 2

public BlogHeader? Header { get; set; } // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

**public int BlogId1 { get; set; }** // Required foreign key property part 1

**public int BlogId2 { get; set; }** // Required foreign key property part 2

public Blog Blog { get; set; } = null!; // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>(

nestedBuilder =>

{

**nestedBuilder.HasKey(e => new { e.Id1, e.Id2 })**;

nestedBuilder.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**HasPrincipalKey<Blog>(e => new { e.Id1, e.Id2 })**

.**HasForeignKey<BlogHeader>(e => new { e.BlogId1, e.BlogId2 })**

.IsRequired();

});

}

In the code above, the calls to HasKey and HasOne have been grouped together into a nested builder. Nested builders remove the need to call Entity<>() multiple times for the same entity type, but are functionally equivalent to calling Entity<>() multiple times.

Composite key must be configured explicitly, since composite keys are not discovered automatically. A composite foreign key value is considered to be null if any of its property values are null. A composite foreign key with one property null and another non-null will not be considered a match for a primary or alternate key with the same values.

By convention, required relationships are configured to cascade delete. This is because the dependent cannot exist in the database once the principal has been deleted. The database can be configured to generate an error, typically crashing the application, instead of automatically deleting dependent rows that can no longer exist. This requires some configuration:

// Principal (parent)

public class Blog

{

public int Id { get; set; }

public BlogHeader? Header { get; set; } // Reference navigation to dependent

}

// Dependent (child)

public class BlogHeader

{

public int Id { get; set; }

public int BlogId { get; set; } // Required foreign key property

public Blog Blog { get; set; } = null!; // Required reference navigation to principal

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Blog>()

.HasOne(e => e.Header)

.WithOne(e => e.Blog)

.**OnDelete(DeleteBehavior.Restrict)**;

}

In all the previous examples, the principal entity type was different from the dependent entity type. This does not have to be the case. For example, in the types below, each Person is optionally related to another Person.

public class Person

{

public int Id { get; set; }

public int? HusbandId { get; set; } // Optional foreign key property

public Person? Husband { get; set; } // Optional reference navigation to principal

public Person? Wife { get; set; } // Reference navigation to dependent

}

...

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Person>()

.HasOne(e => e.Husband)

.WithOne(e => e.Wife)

.HasForeignKey<Person>(e => e.HusbandId)

.IsRequired(false);

}

For one-to-one self referencing relationships, since the principal and dependent entity types are the same, specifying which type contains the foreign key does not clarify the dependent end. In this case, the navigation specified in HasOne points from dependent to principal, and the navigation specified in WithOne points from principal to dependent.

##### Many-to-many relationships

Many-to-many relationships are used when any number entities of one entity type is associated with any number of entities of the same or another entity type. Many-to-many relationships are different from one-to-many and one-to-one relationships in that they cannot be represented in a simple way using just a foreign key. Instead, an additional entity type is needed to "join" the two sides of the relationship. This is known as the "join entity type" and maps to a "join table" in a relational database. The entities of this join entity type contain pairs of foreign key values, where one of each pair points to an entity on one side of the relationship, and the other points to an entity on the other side of the relationship. Each join entity, and therefore each row in the join table, therefore represents one association between the entity types in the relationship.

EF Core can hide the join entity type and manage it behind the scenes. This allows the navigations of a many-to-many relationship to be used in a natural manner, adding or removing entities from each side as needed.

CREATE TABLE "Posts" (

"Id" INTEGER NOT NULL CONSTRAINT "PK\_Posts" PRIMARY KEY AUTOINCREMENT);

CREATE TABLE "Tags" (

"Id" INTEGER NOT NULL CONSTRAINT "PK\_Tags" PRIMARY KEY AUTOINCREMENT);

CREATE TABLE "PostTag" (

"PostsId" INTEGER NOT NULL,

"TagsId" INTEGER NOT NULL,

CONSTRAINT "PK\_PostTag" PRIMARY KEY ("PostsId", "TagsId"),

CONSTRAINT "FK\_PostTag\_Posts\_PostsId" FOREIGN KEY ("PostsId") REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_PostTag\_Tags\_TagsId" FOREIGN KEY ("TagsId") REFERENCES "Tags" ("Id") ON DELETE CASCADE);

In this schema, **PostTag** is the join table. It contains two columns: **PostsId**, which is a foreign key to the primary key of the **Posts** table, and **TagsId**, which is a foreign key to primary key of the **Tags** table. Each row in this table therefore represents an association between one **Post** and one **Tag**.

A simplistic mapping for this schema in EF Core consists of three entity types--one for each table. If each of these entity types are represented by a .NET class, then those classes might look the following:

public class Post // **The next two examples supplies better mapping**

{

public int Id { get; set; }

public List<PostTag> PostTags { get; } = [];

}

public class Tag

{

public int Id { get; set; }

public List<PostTag> PostTags { get; } = [];

}

public class PostTag

{

public int PostsId { get; set; }

public int TagsId { get; set; }

public Post Post { get; set; } = null!;

public Tag Tag { get; set; } = null!;

}

EF allows for a more natural mapping through the introduction of two collection navigations, one on Post containing its related Tags, and an inverse on Tag containing its related Posts. For example:

public class Post

{

public int Id { get; set; }

public List<PostTag> PostTags { get; } = [];

public List<Tag> Tags { get; } = [];

}

public class Tag

{

public int Id { get; set; }

public List<PostTag> PostTags { get; } = [];

public List<Post> Posts { get; } = [];

}

public class PostTag

{

public int PostsId { get; set; }

public int TagsId { get; set; }

public Post Post { get; set; } = null!;

public Tag Tag { get; set; } = null!;

}

These new navigations are known as "skip navigations". EF can manage the join entity transparently, without a .NET class defined for it, and without navigations for the two one-to-many relationships. For example:

public class Post

{

public int Id { get; set; }

**public List<Tag> Tags { get; } = []**;

}

public class Tag

{

public int Id { get; set; }

**public List<Post> Posts { get; } = []**;

}

EF model building conventions will, by default, map the **Post** and **Tag** types shown here to the three tables in the database schema at the top of this section. This mapping, without explicit use of the join type, is what is typically meant by the term "**many-to-many**".

This relationship is mapped by convention. Even though it is not needed, an equivalent explicit configuration for this relationship is shown below as a learning tool:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.HasMany(e => e.Tags)

.WithMany(e => e.Posts);

}

In the previous example, the join table was named **PostTag** by convention. It can be given an explicit name with UsingEntity. For example:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.HasMany(e => e.Tags)

.WithMany(e => e.Posts)

.**UsingEntity("PostsToTagsJoinTable")**;

}

It is possible to leave the properties with their by-convention names, but then map these properties to different column names:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.HasMany(e => e.Tags)

.WithMany(e => e.Posts)

.UsingEntity(

j =>

{

j.Property("PostsId").HasColumnName("PostForeignKey");

j.Property("TagsId").HasColumnName("TagForeignKey");

});

}

This Migration will create SQL:

CREATE TABLE "PostTag" (

"PostForeignKey" INTEGER NOT NULL,

"TagForeignKey" INTEGER NOT NULL,

CONSTRAINT "PK\_PostTag"

PRIMARY KEY ("PostForeignKey", "TagForeignKey"),

CONSTRAINT "FK\_PostTag\_Posts\_PostForeignKey"

FOREIGN KEY ("PostForeignKey")

REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_PostTag\_Tags\_TagForeignKey"

FOREIGN KEY ("TagForeignKey")

REFERENCES "Tags" ("Id") ON DELETE CASCADE);

So far in the examples, the join table has been automatically mapped to a shared-type entity type. This removes the need for a dedicated class to be created for the entity type. However, it can be useful to have such a class so that it can be referenced easily, especially when navigations or a payload are added to the class, as is shown in later examples below:

public class Post

{

public int Id { get; set; }

public List<Tag> Tags { get; } = [];

}

public class Tag

{

public int Id { get; set; }

public List<Post> Posts { get; } = [];

}

**public class PostTag**

{

public int PostId { get; set; }

public int TagId { get; set; }

}

The class can have any name, but it is common to combine the names of the types at either end of the relationship. Now the UsingEntity method can be used to configure this as the join entity type for the relationship:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.HasMany(e => e.Tags)

.WithMany(e => e.Posts)

.UsingEntity<PostTag>();

}

The PostId and TagId are automatically picked up as the foreign keys and are configured as the composite primary key for the join entity type. The properties to use for the foreign keys can be explicitly configured for cases where they don't match the EF convention:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.HasMany(e => e.Tags)

.WithMany(e => e.Posts)

.UsingEntity<PostTag>(

l => l.HasOne<Tag>().WithMany().HasForeignKey(e => e.TagId),

r => r.HasOne<Post>().WithMany().HasForeignKey(e => e.PostId));

}

This migration creates SQL:

CREATE TABLE "PostTag" (

"PostId" INTEGER NOT NULL,

"TagId" INTEGER NOT NULL,

CONSTRAINT "PK\_PostTag" PRIMARY KEY ("PostId", "TagId"),

CONSTRAINT "FK\_PostTag\_Posts\_PostId" FOREIGN KEY ("PostId")

REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_PostTag\_Tags\_TagId" FOREIGN KEY ("TagId")

REFERENCES "Tags" ("Id") ON DELETE CASCADE);

Having a class representing the join entity, it becomes easy to add navigations to it:

public class Post

{

public int Id { get; set; }

public List<Tag> Tags { get; } = [];

**public List<PostTag> PostTags { get; } = []**;

}

public class Tag

{

public int Id { get; set; }

public List<Post> Posts { get; } = [];

**public List<PostTag> PostTags { get; } = []**;

}

public class PostTag

{

public int PostId { get; set; }

public int TagId { get; set; }

}

As shown in this example, navigations to the join entity type can be used in addition to the skip navigations between the two ends of the many-to-many relationship. This means that the skip navigations can be used to interact with the many-to-many relationship in a natural manner, while the navigations to the join entity type can be used when greater control over the join entities themselves is needed. Nothing needs to be changed in the UsingEntity call, since the navigations to the join entity are picked up by convention:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.HasMany(e => e.Tags)

.WithMany(e => e.Posts)

.UsingEntity<PostTag>();

}

The navigations can be configured explicitly for cases where they cannot be determined by convention:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Post>()

.HasMany(e => e.Tags)

.WithMany(e => e.Posts)

.UsingEntity<PostTag>(

l => l.HasOne<Tag>().WithMany(e => e.PostTags),

r => r.HasOne<Post>().WithMany(e => e.PostTags));

}

The mapped database schema is not affected by including navigations in the model:

CREATE TABLE "PostTag" (

"PostId" INTEGER NOT NULL,

"TagId" INTEGER NOT NULL,

CONSTRAINT "PK\_PostTag" PRIMARY KEY ("PostId", "TagId"),

CONSTRAINT "FK\_PostTag\_Posts\_PostId" FOREIGN KEY ("PostId")

REFERENCES "Posts" ("Id") ON DELETE CASCADE,

CONSTRAINT "FK\_PostTag\_Tags\_TagId" FOREIGN KEY ("TagId")

REFERENCES "Tags" ("Id") ON DELETE CASCADE);

### PostgreSQL

#### Add PostgreSQL database provider from NuGet

Run the following command from the project root folder to install the EF Core database provider for PostgreSQL from NuGet:

dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL

#### Add connection string to app settings

Open the appsettings.json file and add the entry "**ConnectionStrings**" with a child entry for the PostgreSQL connection string (e.g. "WebApiDatabase"), the connection string should be in the format

"Host=[URL]; Database=[DB NAME]; Username=[NAME]; Password=[PASSWD]"

When EF Core migrations generates the database, the database value will be the name of the database created in PostgreSQL.

[Many-to-many relationships - EF Core | Microsoft Learn](https://learn.microsoft.com/en-us/ef/core/modeling/relationships/many-to-many)

https://learn.microsoft.com/en-us/ef/core/modeling/relationships/foreign-and-principal-keys

https://github.com/dotnet/EntityFramework.Docs/blob/main/samples/core/Modeling/Relationships/FluentAPI/PrincipalKey.cs

[Microsoft SQL Server Database Provider - EF Core | Microsoft Learn](https://learn.microsoft.com/en-us/ef/core/providers/sql-server/?tabs=dotnet-core-cli)

1. https://learn.microsoft.com/en-us/ef/core/modeling/entity-types?tabs=fluent-api
2. https://www.entityframeworktutorial.net/efcore/create-entities.aspx
3. [EPISODE 1 On How to Create Simple CRUD .NET 8.0 Blazor With Auto Render Magic,EF Core, SQL Server. (youtube.com)](https://www.youtube.com/watch?v=xgeoNVsIwug)
4. [EPISODE 2 On How to Create Simple CRUD .NET 8.0 Blazor With Auto Render Magic,EF Core, SQL Server.](https://www.youtube.com/watch?v=7M6Pz86Vw3Q)