Entity Framework Core

Entity Framework Core is the new version of Entity Framework after EF 6.x. It is open-source, lightweight, extensible and a cross-platform version of Entity Framework data access technology.

Entity Framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.

EF Core is intended to be used with .NET Core applications. However, it can also be used with standard .NET 4.5+ framework based applications.

The following figure illustrates the supported application types, .NET Frameworks and OSs.

[A screenshot of a computer

Description automatically generated with medium confidence](https://www.entityframeworktutorial.net/Images/efcore/ef-core.png)

EF Core Version History

| EF Core Version | Release Date |
| --- | --- |
| EF Core 2.0 | August 2017 |
| EF Core 1.1 | November 2016 |
| EF Core 1.0 | June 2016 |

EF Core on GitHub: <https://github.com/aspnet/EntityFrameworkCore>

EF Core Roadmap: [docs.microsoft.com/en-us/ef/core/what-is-new/roadmap](https://docs.microsoft.com/en-us/ef/core/what-is-new/roadmap)

Track EF Core's issues at <https://github.com/aspnet/EntityFrameworkCore/issues>

EF Core Official Documentation: <https://docs.microsoft.com/ef/core>

EF Core Development Approaches

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 2.0.

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.

[Diagram

Description automatically generated](https://www.entityframeworktutorial.net/Images/efcore/ef-core-dev-approaces.png)

EF Core vs EF 6

Entity Framework Core is the new and improved version of Entity Framework for .NET Core applications. EF Core is new, so still not as mature as EF 6.

EF Core continues to support the following features and concepts, same as EF 6.

1. DbContext & DbSet
2. Data Model
3. Querying using Linq-to-Entities
4. Change Tracking
5. SaveChanges
6. Migrations

EF Core will include most of the features of EF 6 gradually. However, there are some features of EF 6 which are not supported in EF Core 2.0 such as:

1. EDMX/ Graphical Visualization of Model
2. Entity Data Model Wizard (for DB-First approach)
3. ObjectContext API
4. Querying using Entity SQL.
5. Automated Migration
6. Inheritance: Table per type (TPT)
7. Inheritance: Table per concrete class (TPC)
8. Many-to-Many without join entity
9. Entity Splitting
10. Spatial Data
11. Lazy loading of related data
12. Stored procedure mapping with DbContext for CUD operation
13. Seed data
14. Automatic migration

EF Core includes the following new features which are not supported in EF 6.x:

1. Easy relationship configuration
2. Batch INSERT, UPDATE, and DELETE operations
3. In-memory provider for testing
4. Support for IoC (Inversion of Control)
5. Unique constraints
6. Shadow properties
7. Alternate keys
8. Global query filter
9. Field mapping
10. DbContext pooling
11. Better patterns for handling disconnected entity graphs

Learn more on EF Core and EF 6 differences at [here](https://docs.microsoft.com/en-us/ef/efcore-and-ef6/features).

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.

The following table lists database providers and NuGet packages for EF Core.

| Database | NuGet Package |
| --- | --- |
| SQL Server | [Microsoft.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer) |
| MySQL | [MySql.Data.EntityFrameworkCore](https://www.nuget.org/packages/MySql.Data.EntityFrameworkCore) |
| PostgreSQL | [Npgsql.EntityFrameworkCore.PostgreSQL](https://www.nuget.org/packages/Npgsql.EntityFrameworkCore.PostgreSQL) |
| SQLite | [Microsoft.EntityFrameworkCore.SQLite](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SQLite) |
| SQL Compact | [EntityFrameworkCore.SqlServerCompact40](https://www.nuget.org/packages/EntityFrameworkCore.SqlServerCompact40) |
| In-memory | [Microsoft.EntityFrameworkCore.InMemory](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.InMemory) |

Learn to install EF Core in the next chapter.

# Install Entity Framework Core

Entity Framework Core can be used with .NET Core or .NET 4.6 based applications. Here, you will learn to install and use Entity Framework Core 2.0 in .NET Core applications using Visual Studio 2017.

EF Core is not a part of .NET Core and standard .NET framework. It is available as a NuGet package. You need to install NuGet packages for the following two things to use EF Core in your application:

1. EF Core DB provider
2. EF Core tools

Let's install the above NuGet packages in the .NET Core console application in Visual Studio 2017.

## Install EF Core DB Provider

As mentioned in the previous chapter, EF Core allows us to access databases via the provider model. There are different [EF Core DB providers](https://www.entityframeworktutorial.net/efcore/entity-framework-core.aspx#efcore-db-providers) available for the different databases. These providers are available as NuGet packages.

First, we need to install the NuGet package for the provider of the database we want to access. Here, we want to access MS SQL Server database, so we need to install Microsoft.EntityFrameworkCore.SqlServer NuGet package.

To install the DB provider NuGet package, right click on the project in the Solution Explorer in Visual Studio and select Manage NuGet Packages.. (or select on the menu: Tools -> NuGet Package Manager -> Manage NuGet Packages For Solution).

[Graphical user interface, text, application

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/install-efcore-1.png)

This will open NuGet Package Manager UI. Click on the Browse or the Updates tab and search for Microsoft.entityframeworkcore in the search box at the top left corner, as shown below.

[Graphical user interface, text, application

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/install-efcore-2.png)

Choose the provider package for the database you want to access. In this case select Microsoft.EntityFrameworkCore.SqlServer for MS SQL Server as shown above. (make sure that it has the .NET symbol and the Author is Microsoft). Click Install to start the installation.

The preview popup displays the list of packages it is going to install in your application. Review the changes and click OK.

[Graphical user interface, text, application

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/install-efcore-3.png)

Finally, accept the license terms associated with the packages that are going to be installed.

[Graphical user interface, text, application, email

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/install-efcore-4.png)

This will install the Microsoft.EntityFrameworkCore.SqlServer package. Verify it in Dependencies -> NuGet, as shown below.

[Graphical user interface, text, application, email

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/install-efcore-5.png)

Notice that the provider NuGet package also installed other dependent packages such as Microsoft.EntityFrameworkCore.Relational and System.Data.SqlClient.

**Alternatively,** you can also install provider's NuGet package using Package Manager Console. Go to Tools -> NuGet Package Manager -> Package Manager Console and execute the following command to install SQL Server provider package:

PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer

## Install EF Core Tools

Along with the DB provider package, you also need to install EF tools to execute EF Core commands. These make it easier to perform several EF Core-related tasks in your project at design time, such as migrations, scaffolding, etc.

EF Tools are available as NuGet packages. You can install NuGet package for EF tools depending on where you want to execute commands: either using Package Manager Console (PowerShell version of EF Core commands) or using dotnet CLI.

### **Install EF Core Tools for PMC**

In order to execute EF Core commands from Package Manager Console, search for the Microsoft.EntityFrameworkCore.Tools package from NuGet UI and install it as shown below.

[Graphical user interface, text, application, email

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/install-efcore-6.png)

This will allow you to execute EF Core commands for scaffolding, migration etc. directly from Package Manager Console (PMC) within Visual Studio.

### **Install EF Core Tools for dotnet CLI**

If you want to execute EF Core commands from .NET Core's CLI (Command Line Interface), first install the NuGet package Microsoft.EntityFrameworkCore.Tools.DotNet using NuGet UI.

After installing Microsoft.EntityFrameworkCore.Tools.DotNet package, edit the .csproj file by right clicking on the project in the Solution Explorer and select Edit <projectname>.csproj. Add <DotNetCliToolReference> node as shown below. This is an extra step you need to perform in order to execute EF Core 2.0 commands from dotnet CLI in VS2017.

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<OutputType>Exe</OutputType>

<TargetFramework>netcoreapp2.0</TargetFramework>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="2.0.0" />

<PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="2.0.0" />

**<DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet" Version="2.0.0" />**

</ItemGroup>

</Project>

Now, open the command prompt (or terminal) from the root folder of your project and execute EF Core commands from CLI starting with **dotnet ef**, as shown below.

[Text

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/dotnet-cli.png)

Thus, you can install required packages for EF Core 2.0 to get started.

# Creating a Model for an Existing Database in Entity Framework Core

Here you will learn how to create the context and entity classes for an existing database in Entity Framework Core. Creating entity & context classes for an existing database is called Database-First approach.

EF Core does not support visual designer for DB model and wizard to create the entity and context classes similar to EF 6. So, we need to do reverse engineering using the Scaffold-DbContext command. This reverse engineering command creates entity and context classes (by deriving DbContext) based on the schema of the existing database.

Let's create entity and context classes for the following SchoolDB database in the local MS SQL Server shown below.

[Graphical user interface

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/schooldb.png)

## Scaffold-DbContext Command

Use Scaffold-DbContext to create a model based on your existing database. The following parameters can be specified with Scaffold-DbContext in Package Manager Console:

Scaffold-DbContext [-Connection] [-Provider] [-OutputDir] [-Context] [-Schemas>] [-Tables>]

[-DataAnnotations] [-Force] [-Project] [-StartupProject] [<CommonParameters>]

In Visual Studio, select menu Tools -> NuGet Package Manger -> Package Manger Console and run the following command:

PM> Scaffold-DbContext "Server=.\SQLExpress;Database=SchoolDB;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

In the above command, the first parameter is a connection string which includes three parts: DB Server, database name and security info. Here, Server=.\SQLExpress; refers to local SQLEXPRESS database server. Database=SchoolDB; specifies the database name "SchoolDB" for which we are going to create classes. Trusted\_Connection=True; specifies the Windows authentication. It will use Windows credentials to connect to the SQL Server. The second parameter is the provider name. We use provider for the SQL Server, so it is Microsoft.EntityFrameworkCore.SqlServer. The -OutputDir parameter specifies the directory where we want to generate all the classes which is the Models folder in this case.

Use the following command to get the detailed help on Scaffold-DbContext command:

PM> get-help scaffold-dbcontext –detailed

The above Scaffold-DbContext command creates entity classes for each table in the SchoolDB database and context class (by deriving DbContext) with Fluent API configurations for all the entities in the Models folder.

[Graphical user interface, text, application

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/dbfirst-model.png)

The following is the generated Student entity class for the Student table.

using System;

using System.Collections.Generic;

namespace EFCoreTutorials.Models

{

public partial class Student

{

public Student()

{

StudentCourse = new HashSet<StudentCourse>();

}

public int StudentId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public int? StandardId { get; set; }

public Standard Standard { get; set; }

public StudentAddress StudentAddress { get; set; }

public ICollection<StudentCourse> StudentCourse { get; set; }

}

}

The following is the SchoolDBContext class which you can use to save or retrieve data.

using System;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata;

namespace EFCoreTutorials.Models

{

public partial class SchoolDBContext : DbContext

{

public virtual DbSet<Course> Course { get; set; }

public virtual DbSet<Standard> Standard { get; set; }

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

public virtual DbSet<StudentAddress> StudentAddress { get; set; }

public virtual DbSet<StudentCourse> StudentCourse { get; set; }

public virtual DbSet<Teacher> Teacher { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

#warning To protect potentially sensitive information in your connection string, you should move it out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection strings.

optionsBuilder.UseSqlServer(@"Server=.\SQLExpress;Database=SchoolDB;Trusted\_Connection=True;");

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Course>(entity =>

{

entity.Property(e => e.CourseName)

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Teacher)

.WithMany(p => p.Course)

.HasForeignKey(d => d.TeacherId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Course\_Teacher");

});

modelBuilder.Entity<Standard>(entity =>

{

entity.Property(e => e.Description)

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.StandardName)

.HasMaxLength(50)

.IsUnicode(false);

});

modelBuilder.Entity<Student>(entity =>

{

entity.Property(e => e.StudentId).HasColumnName("StudentID");

entity.Property(e => e.FirstName)

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.LastName)

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Standard)

.WithMany(p => p.Student)

.HasForeignKey(d => d.StandardId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Student\_Standard");

});

modelBuilder.Entity<StudentAddress>(entity =>

{

entity.HasKey(e => e.StudentId);

entity.Property(e => e.StudentId)

.HasColumnName("StudentID")

.ValueGeneratedNever();

entity.Property(e => e.Address1)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.Address2)

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.City)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.State)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Student)

.WithOne(p => p.StudentAddress)

.HasForeignKey<StudentAddress>(d => d.StudentId)

.HasConstraintName("FK\_StudentAddress\_Student");

});

modelBuilder.Entity<StudentCourse>(entity =>

{

entity.HasKey(e => new { e.StudentId, e.CourseId });

entity.HasOne(d => d.Course)

.WithMany(p => p.StudentCourse)

.HasForeignKey(d => d.CourseId)

.OnDelete(DeleteBehavior.ClientSetNull)

.HasConstraintName("FK\_StudentCourse\_Course");

entity.HasOne(d => d.Student)

.WithMany(p => p.StudentCourse)

.HasForeignKey(d => d.StudentId)

.HasConstraintName("FK\_StudentCourse\_Student");

});

modelBuilder.Entity<Teacher>(entity =>

{

entity.Property(e => e.StandardId).HasDefaultValueSql("((0))");

entity.Property(e => e.TeacherName)

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Standard)

.WithMany(p => p.Teacher)

.HasForeignKey(d => d.StandardId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Teacher\_Standard");

});

}

}

}

**Note:** EF Core creates entity classes only for tables and not for StoredProcedures or Views.

## DotNet CLI

If you use dotnet command line interface to execute EF Core commands then open command prompt and navigate to the root folder and execute the following dotnet ef dbcontext scaffold command:

> dotnet ef dbcontext scaffold "Server=.\SQLEXPRESS;Database=SchoolDB;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -o Models

Thus, you can create EF Core model for an existing database.

**Note:** Once you have created the model, you must use the Migration commands whenever you change the model to keep the database up to date with the model.

# One-to-Many Relationship Conventions in Entity Framework Core

In the previous chapter, you learned about the EF conventions which map entities to different objects of the database. Here, you will learn about the relationship conventions between two entity classes that result in one-to-many relationships between corresponding tables in the database.

Entity Framework Core follows the same convention as [Entity Framework 6.x conventions for one-to-many relationship](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#conventions-for-one-to-many-ef6). The only difference is that EF Core creates a foreign key column with the same name as navigation property name and not as <NavigationPropertyName>\_<PrimaryKeyPropertyName>

Let's look at the different conventions which automatically configure a one-to-many relationship between the following Student and Grade entities.

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

}

After applying the conventions for one-to-many relationship in the entities above, the database tables for Student and Grade entities will look like below, where the Students table includes a foreign key GradeId.

[Diagram

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/onetomany-entities.png)

### **Convention 1**

We want to establish a one-to-many relationship where many students are associated with one grade. This can be achieved by including a reference navigation property in the dependent entity as shown below. (here, the Student entity is the dependent entity and the Grade entity is the principal entity).

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

**public Grade Grade { get; set; }**

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

}

In the example above, the Student entity class includes a reference navigation property of Grade type. This allows us to link the same Grade to many different Student entities, which creates a one-to-many relationship between them. This will produce a one-to-many relationship between the Students and Grades tables in the database, where Students table includes a nullable foreign key GradeId, as shown below. EF Core will create a shadow property for the foreign key named GradeId in the conceptual model, which will be mapped to the GradeId foreign key column in the Students table.

[Graphical user interface

Description automatically generated with low confidence](https://www.entityframeworktutorial.net/images/efcore/onetomany-conventions1.png)

**Note:** The reference property Grade is nullable, so it creates a nullable ForeignKey GradeId in the Students table. You can configure NotNull foreign keys using fluent API.

### **Convention 2**

Another convention is to include a collection navigation property in the principal entity as shown below.

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

**public ICollection<Student> Students { get; set; }**

}

In the example above, the Grade entity includes a collection navigation property of type ICollection<student>. This will allow us to add multiple Student entities to a Grade entity, which results in a one-to-many relationship between Students and Grades tables in the database, same as in convention 1.

### **Convention 3**

Another EF convention for the one-to-many relationship is to include navigation property at both ends, which will also result in a one-to-many relationship (convention 1 + convention 2).

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

**public Grade Grade { get; set; }**

}

public class Grade

{

public int GradeID { get; set; }

public string GradeName { get; set; }

**public ICollection<Student> Students { get; set; }**

}

In the example above, the Student entity includes a reference navigation property of Grade type and the Grade entity class includes a collection navigation property ICollection<Student>, which results in a one-to-many relationship between corresponding database tables Students and Grades, same as in convention 1.

### **Convention 4**

Defining the relationship fully at both ends with the foreign key property in the dependent entity creates a one-to-many relationship.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

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

**public Grade Grade { get; set; }**

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

**public ICollection<Student> Students { get; set; }**

}

In the above example, the Student entity includes a foreign key property GradeId of type int and its reference navigation property Grade. At the other end, the Grade entity also includes a collection navigation property ICollection<Student>. This will create a one-to-many relationship with the NotNull foreign key column in the Students table, as shown below.

[Graphical user interface

Description automatically generated with medium confidence](https://www.entityframeworktutorial.net/images/efcore/onetomany-conventions2.png)

If you want to make the foreign key GradeId as nullable, then use nullable int data type (Nullable<int> or int?), as shown below.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

**public int? GradeId { get; set; }**

public Grade Grade { get; set; }

}

Therefore, these are the conventions which automatically create a one-to-many relationship in the corresponding database tables. If entities do not follow the above conventions, then you can use Fluent API to configure the one-to-many relationship.

# Creating a Model for an Existing Database in Entity Framework Core

Here you will learn how to create the context and entity classes for an existing database in Entity Framework Core. Creating entity & context classes for an existing database is called Database-First approach.

EF Core does not support visual designer for DB model and wizard to create the entity and context classes similar to EF 6. So, we need to do reverse engineering using the Scaffold-DbContext command. This reverse engineering command creates entity and context classes (by deriving DbContext) based on the schema of the existing database.

Let's create entity and context classes for the following SchoolDB database in the local MS SQL Server shown below.

[Graphical user interface

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/schooldb.png)

## Scaffold-DbContext Command

Use Scaffold-DbContext to create a model based on your existing database. The following parameters can be specified with Scaffold-DbContext in Package Manager Console:

Scaffold-DbContext [-Connection] [-Provider] [-OutputDir] [-Context] [-Schemas>] [-Tables>]

[-DataAnnotations] [-Force] [-Project] [-StartupProject] [<CommonParameters>]

In Visual Studio, select menu Tools -> NuGet Package Manger -> Package Manger Console and run the following command:

PM> Scaffold-DbContext "Server=.\SQLExpress;Database=SchoolDB;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models

In the above command, the first parameter is a connection string which includes three parts: DB Server, database name and security info. Here, Server=.\SQLExpress; refers to local SQLEXPRESS database server. Database=SchoolDB; specifies the database name "SchoolDB" for which we are going to create classes. Trusted\_Connection=True; specifies the Windows authentication. It will use Windows credentials to connect to the SQL Server. The second parameter is the provider name. We use provider for the SQL Server, so it is Microsoft.EntityFrameworkCore.SqlServer. The -OutputDir parameter specifies the directory where we want to generate all the classes which is the Models folder in this case.

Use the following command to get the detailed help on Scaffold-DbContext command:

PM> get-help scaffold-dbcontext –detailed

The above Scaffold-DbContext command creates entity classes for each table in the SchoolDB database and context class (by deriving DbContext) with Fluent API configurations for all the entities in the Models folder.

[Graphical user interface, text, application

Description automatically generated](https://www.entityframeworktutorial.net/images/efcore/dbfirst-model.png)

The following is the generated Student entity class for the Student table.

using System;

using System.Collections.Generic;

namespace EFCoreTutorials.Models

{

public partial class Student

{

public Student()

{

StudentCourse = new HashSet<StudentCourse>();

}

public int StudentId { get; set; }

public string FirstName { get; set; }

public string LastName { get; set; }

public int? StandardId { get; set; }

public Standard Standard { get; set; }

public StudentAddress StudentAddress { get; set; }

public ICollection<StudentCourse> StudentCourse { get; set; }

}

}

The following is the SchoolDBContext class which you can use to save or retrieve data.

using System;

using Microsoft.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore.Metadata;

namespace EFCoreTutorials.Models

{

public partial class SchoolDBContext : DbContext

{

public virtual DbSet<Course> Course { get; set; }

public virtual DbSet<Standard> Standard { get; set; }

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

public virtual DbSet<StudentAddress> StudentAddress { get; set; }

public virtual DbSet<StudentCourse> StudentCourse { get; set; }

public virtual DbSet<Teacher> Teacher { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

#warning To protect potentially sensitive information in your connection string, you should move it out of source code. See http://go.microsoft.com/fwlink/?LinkId=723263 for guidance on storing connection strings.

optionsBuilder.UseSqlServer(@"Server=.\SQLExpress;Database=SchoolDB;Trusted\_Connection=True;");

}

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Course>(entity =>

{

entity.Property(e => e.CourseName)

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Teacher)

.WithMany(p => p.Course)

.HasForeignKey(d => d.TeacherId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Course\_Teacher");

});

modelBuilder.Entity<Standard>(entity =>

{

entity.Property(e => e.Description)

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.StandardName)

.HasMaxLength(50)

.IsUnicode(false);

});

modelBuilder.Entity<Student>(entity =>

{

entity.Property(e => e.StudentId).HasColumnName("StudentID");

entity.Property(e => e.FirstName)

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.LastName)

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Standard)

.WithMany(p => p.Student)

.HasForeignKey(d => d.StandardId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Student\_Standard");

});

modelBuilder.Entity<StudentAddress>(entity =>

{

entity.HasKey(e => e.StudentId);

entity.Property(e => e.StudentId)

.HasColumnName("StudentID")

.ValueGeneratedNever();

entity.Property(e => e.Address1)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.Address2)

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.City)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.Property(e => e.State)

.IsRequired()

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Student)

.WithOne(p => p.StudentAddress)

.HasForeignKey<StudentAddress>(d => d.StudentId)

.HasConstraintName("FK\_StudentAddress\_Student");

});

modelBuilder.Entity<StudentCourse>(entity =>

{

entity.HasKey(e => new { e.StudentId, e.CourseId });

entity.HasOne(d => d.Course)

.WithMany(p => p.StudentCourse)

.HasForeignKey(d => d.CourseId)

.OnDelete(DeleteBehavior.ClientSetNull)

.HasConstraintName("FK\_StudentCourse\_Course");

entity.HasOne(d => d.Student)

.WithMany(p => p.StudentCourse)

.HasForeignKey(d => d.StudentId)

.HasConstraintName("FK\_StudentCourse\_Student");

});

modelBuilder.Entity<Teacher>(entity =>

{

entity.Property(e => e.StandardId).HasDefaultValueSql("((0))");

entity.Property(e => e.TeacherName)

.HasMaxLength(50)

.IsUnicode(false);

entity.HasOne(d => d.Standard)

.WithMany(p => p.Teacher)

.HasForeignKey(d => d.StandardId)

.OnDelete(DeleteBehavior.Cascade)

.HasConstraintName("FK\_Teacher\_Standard");

});

}

}

}

**Note:** EF Core creates entity classes only for tables and not for StoredProcedures or Views.

## DotNet CLI

If you use dotnet command line interface to execute EF Core commands then open command prompt and navigate to the root folder and execute the following dotnet ef dbcontext scaffold command:

> dotnet ef dbcontext scaffold "Server=.\SQLEXPRESS;Database=SchoolDB;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -o Models

Thus, you can create EF Core model for an existing database.

**Note:** Once you have created the model, you must use the Migration commands whenever you change the model to keep the database up to date with the model.

Entity Framework Core: DbContext

The [DbContext](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext?view=efcore-2.0" \t "_blank) class is an integral part of Entity Framework. An instance of DbContext represents a session with the database which can be used to query and save instances of your entities to a database. DbContext is a combination of the Unit Of Work and Repository patterns.

DbContext in EF Core allows us to perform following tasks:

1. Manage database connection
2. Configure model & relationship
3. Querying database
4. Saving data to the database
5. Configure change tracking
6. Caching
7. Transaction management

To use DbContext in our application, we need to create the class that derives from DbContext, also known as context class. This context class typically includes [DbSet<TEntity>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbset-1?view=efcore-2.0" \t "_blank) properties for each entity in the model. Consider the following example of context class in EF Core.

public class SchoolContext : DbContext

{

public SchoolContext()

{

}

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

}

//entities

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

public DbSet<Course> Courses { get; set; }

}

In the example above, the SchoolContext class is derived from the DbContext class and contains the DbSet<TEntity> properties of Student and Course type. It also overrides the OnConfiguring and OnModelCreating methods. We must create an instance of SchoolContext to connect to the database and save or retrieve Student or Course data.

The OnConfiguring() method allows us to select and configure the data source to be used with a context using DbContextOptionsBuilder. Learn how to configure a DbContext class at [here](https://docs.microsoft.com/en-us/ef/core/miscellaneous/configuring-dbcontext).

The OnModelCreating() method allows us to configure the model using ModelBuilder Fluent API.

### Relationship types

There are three kinds of relationships are possible

A picture containing shape

Description automatically generatedA picture containing text, electronics, circuit

Description automatically generatedGraphical user interface

Description automatically generated with low confidenceA picture containing text, person, birthday, child

Description automatically generatedDiagram

Description automatically generated with medium confidenceA picture containing bottle, indoor, drink, beverage

Description automatically generatedGraphical user interface

Description automatically generated with low confidenceA picture containing shape

Description automatically generatedA picture containing text, electronics, circuit

Description automatically generated

Graphical user interface

Description automatically generated with low confidence

1. One to One
2. One to Many
3. Many to Many

Entity Framework uses the navigational properties to describe the relationship between two entity types. A relationship in the Entity Framework always has two endpoints. Each endpoint, which participates in the relationship must return a navigation property describing the relationship. Entity Framework maps these relationships together using the convention.

There are two types of navigation properties that can be returned by the entity types depending on the type of relationship they participate.

1. Reference Object (If the relationship is one or Zero-or-One)
2. Collection (If the relationship is Many)

Code first uses the navigational properties to navigate to the relationship.

### One to One relationship

The Relationship between husband and wife is an example of one to one relationship. You can have only one spouse. One to one relationship usually does not happen in database design. Because you can combine them into one table easily without breaking any rule.

For our example,  consider the domain class Employee and EmployeeAddress.  Let us assume that employee can have only one address.  This will create a one to one relationship between employee and EmployeeAddress table. In a One to one relationship PrimaryKey of one table (employeeID of employee table) is both Primary key and Foreign key in the second table (EmployeeAddress). Hence our entity classes will be as shown below

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | public **class** Employee      {          public **int** EmployeeID { **get**; **set**; }          public **string** Name { **get**; **set**; }    *//Navigation property Returns the Employee Address*          public virtual EmployeeAddress EmployeeAddress { **get**; **set**; }      }        public **class** EmployeeAddress      {          [Key, ForeignKey("Employee")]          public **int** EmployeeID { **get**; **set**; }          public **string** Address { **get**; **set**; }    *//Navigation property Returns the Employee object*          public virtual Employee Employee { **get**; **set**; }      } |

Note that we have created the navigational property which returns the reference to the related entity. In Employee class navigational property returns the reference to the EmployeeAddress object. In EmployeeAddress class navigational property returns the reference to the Employee object.

[Configure one to one relationship in entity framework](https://www.tektutorialshub.com/entity-framework/ef-one-to-one-relationship/)

### One to Many Relationship

The Relationship between mother and children is an example of one to Many relationships. A Mother can have many children, but a child can have only one mother.

The relationship between employee and the department is one to many. The employee belongs to only one department. The department can have many employees. In a One to Many relationship Primary key of the department table (DepartmentID) is defined as Foreign key in the employee’s table

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public **class** Employee  {      public **int** EmployeeID { **get**; **set**; }      public **string** Name { **get**; **set**; }      public virtual Department Department { **get**; **set**; }  }    public **class** Department  {      public **int** DepartmentID { **get**; **set**; }      public **string** Name { **get**; **set**; }      public virtual ICollection<Employee> Employees { **get**; **set**; }  } |

Note that navigational property in the employee class returns the reference to the department object. The navigational property in Department class returns the employee’s collection. Code first uses this information to determine which class is dependent on which.

[Configure One to Many Relationship in Entity Framework](https://www.tektutorialshub.com/entity-framework/ef-one-to-many-relationship/)

### Many to Many Relationship

The Relationship between siblings is Many to Many relationships. Each of your siblings can have many siblings

The relationship between employee and projects is many to many. The employee can be part of more than one project. The Projects can have many employees. The Many to Many relationships usually involves the creation of a join table. The join table will have a composite primary key consisting combination of the primary key of the both employees and project table.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | public **class** Employee  {      public **int** EmployeeID { **get**; **set**; }      public **string** Name { **get**; **set**; }  *//Navigational Property*      public virtual ICollection<Project> Projects { **get**; **set**; }  }  public **class** Project  {      public **int** ProjectID { **get**; **set**; }      public **string** Name { **get**; **set**; }  *//Navigational Property*      public virtual ICollection<Employee> Employees { **get**; **set**; }  } |

[Configure Many to Many relationships in Entity Framework](https://www.tektutorialshub.com/entity-framework/ef-many-many-relationship/)

## Configuring the Relationship in Entity Framework

The Relationships in entity framework can be done using the following methods

1. [Default Code First Conventions](https://www.tektutorialshub.com/entity-framework/ef-code-first-conventions/)
2. [Data Annotations attributes](https://www.tektutorialshub.com/entity-framework/ef-data-annotations/)
3. [Fluent API](https://www.tektutorialshub.com/entity-framework/ef-fluent-api-tutorial/)

### Default Code First Conventions

We learned about [Code First Conventions](https://www.tektutorialshub.com/entity-framework/ef-code-first-conventions/) from the Tutorial. Code First conventions use the default rules to build a model based on your domain classes. Code First infers the relationships using the navigational properties and use the Conventions and builds the model.

### Data Annotations Attributes

Data Annotations allow us to fine tune the model by Attributes. [Key Attribute](https://www.tektutorialshub.com/entity-framework/ef-data-annotations-key-attribute/) & [ForeginKey Attributes](https://www.tektutorialshub.com/entity-framework/ef-data-annotations-foreignkey-attribute/) are used to further configure the relationship between models

### Fluent API

Fluent API Provides more power to configure the relationships in entity models.  DbModelbuilder returns the entitytypeconfiguration to configure the entities. entitytypeconfiguration exposes HasMany, HasOptional, HasRequired methods which used to configure relationships. The method to be used depends on the type of relationship (one or many). These methods take navigational properties as their parameter

#### HasMany

Use this when entity type participates in **many relationship**. This takes the collection navigational property. This method returns **ManyNavigationPropertyConfiguration**.

#### HasOptional

Use this when entity type participates in **one relationship**. This takes the reference navigation property. Use this when the **relationship is optional**. This method creates the nullable foreign key. This means that you can save the data to database without this relationship being created. This method returns OptionalNavigationPropertyConfiguration

#### HasRequired

This is similar to HasOptional. The difference is that the foreign key is not null, Which means that entity type and its relation must be created otherwise data will not be saved to the database. This method returns RequiredNavigationPropertyConfiguration  
The return types of the above methods are defined in the **System.Data.Entity.ModelConfiguration.Configuration**. These return types then used to configure the relationship at the other end of the relationship. The return types expose the following methods.

#### WithOptional

Configures an optional navigation property on the other side of the relationship.

#### WithRequired

Configures the relationship to be required on the other side of the relationship.

#### WithMany

Configures the relationship to be many on the other side of the relationship.

### Conclusion

In the next tutorials, we will explore each of three relationships in detail.

* [Configure one to one relationship in entity framework](https://www.tektutorialshub.com/entity-framework/ef-one-to-one-relationship/)
* [Configure One to Many Relationship in Entity Framework](https://www.tektutorialshub.com/entity-framework/ef-one-to-many-relationship/)
* [Configure Many to Many relationships in Entity Framework](https://www.tektutorialshub.com/entity-framework/ef-many-many-relationship/)