**Introduction to Entity Framework Core**

**Entity Framework Core** also known as **EF Core** is the latest version of Microsoft’s Entity Framework. It is an **Object Relational Mapping (O/RM) framework**, an enhanced version of ADO.NET, that automates data storage and retrieval from the database. EF Core is open source, cross-platform, lightweight, extensible and very powerful in nature. It is also very easy to learn and use in our DOT NET Projects.

This tutorial is a part of **Entity Framework Core** series.

EF Core can be use with with any .NET applications. It runs on all OS – Windows, Mac, Linux.

EF Core Supported Application Types

We can use Entity Framework Core on all DOT NET applications like :

1. Console Applications
2. Windows Applications
3. ASP.NET Web Forms
4. ASP.NET MVC
5. ASP.NET Core MVC
6. ASP.NET Core Razor Pages
7. Blazor Apps
8. WPF
9. Xamarin Framework
10. Web API
11. .NET MAUI

EF Core Versions

| **EF Core Version** | **Release Date** |
| --- | --- |
| EF Core 8.0 | November 2023 |
| EF Core 7.0 | November 2022 |
| EF Core 6.0 | December 2021 |
| EF Core 5.0 | November 2020 |
| EF Core 3.1 | December 2019 |
| EF Core 2.0 | August 2017 |
| EF Core 1.1 | November 2016 |
| EF Core 1.0 | June 2016 |

One of the most important topics of EF Core is [DbContext Class in Entity Framework Core](https://www.yogihosting.com/dbcontext-entity-framework-core/). Make sure to understand it fully else coding will become hard.

Entity Framework Core vs Entity Framework

Entity Framework Core is the highly enhanced version of Entity Framework and contains a lot more features. It was released in the year 2016 when DOT NET Core came into existence.

Entity Framework has a visual designer tool called .edmx file which describes the database and the models along with the mappings between them.

Entity Framework Core does not have a visual designer tool. All the models and mappings are created in C# classes. Entity Framework is no longer actively developed by Microsoft so use only Entity Framework Core in your projects.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=4dde21cf-06e5-4a0c-5a32-8eb6e23f8423&d_id=122531&imp_id=2823895457026080&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

Entity Framework Core Approaches

There are 2 Entity Framework Core develpment approaches which we can follow. These are:

1. Database First .

2. Code First.

There is very little support of Database First approach since there is no visual designer, like .edmx file, in EF Core.

The EF Core mainly supports Code First approach.

Database First Approach

In Database First approach the domain & context classes are created based on the existing Database.

Code First Approach

In Code First approach the domain & context classes are created by you then EF Core created the database using these classes. The term Migration is used whenever EF Core creates or updates the Database based on the domain & context classes.

Ezoic

Entity Framework Core Features

Some important features of EF Core are:

1. DbContext & DbSet

2. LINQ Support

3. Tracking

4. [Migrations](https://www.yogihosting.com/migrations-entity-framework-core/)

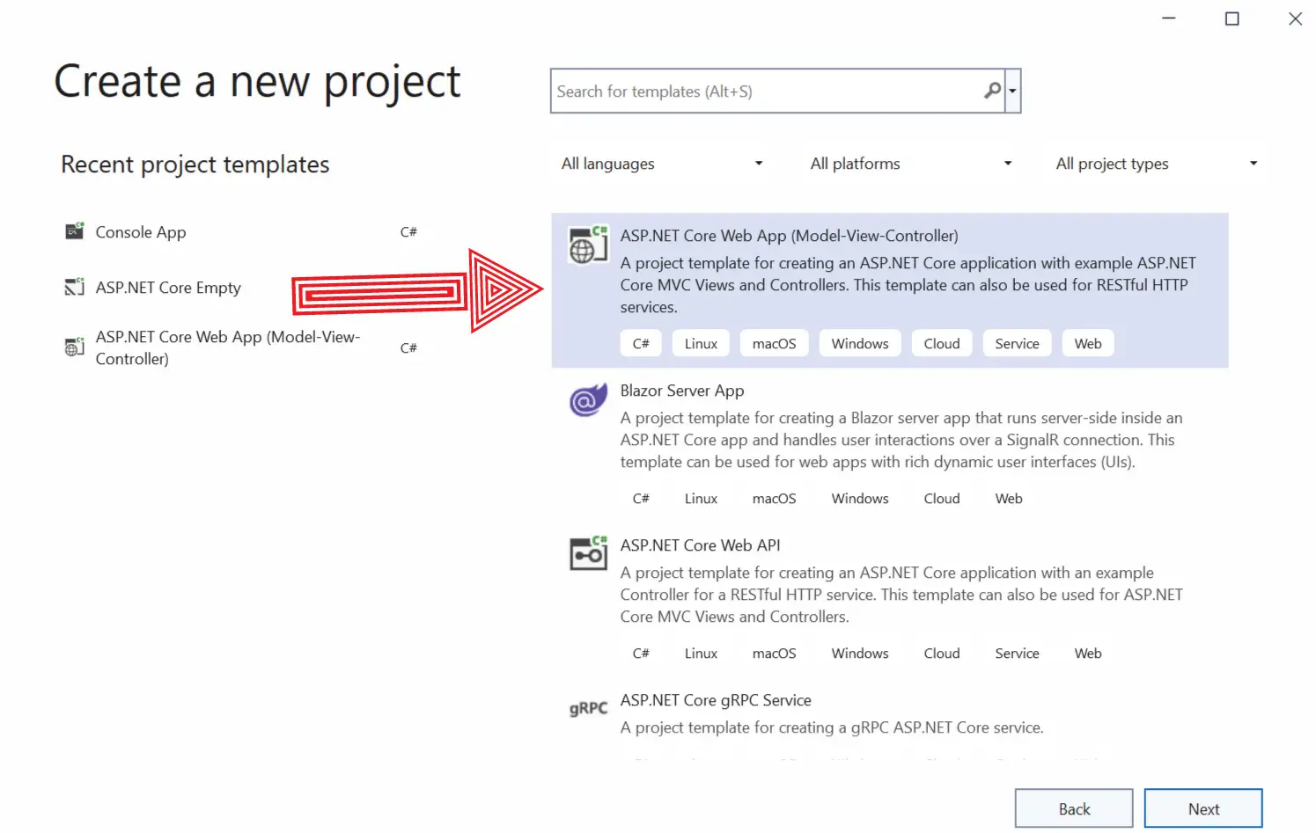
5. Batch Insert, Update and Delete operations

Supported Databases

Entity Framework Core works on many databases like:

1. SQL Server  
2. MySQL  
3. PostgreSQL  
4. SQLite  
5. SQL Compact  
6. Firebird  
7. Oracle  
8. Db2  
9. MongoDB

**Ins**First create a new web app in Visual Studio by selecting ASP.NET Core Web App (Model-View-Controller) template. See the image below.



To this app we will **Install Entity Framework Core** and work on database operations.

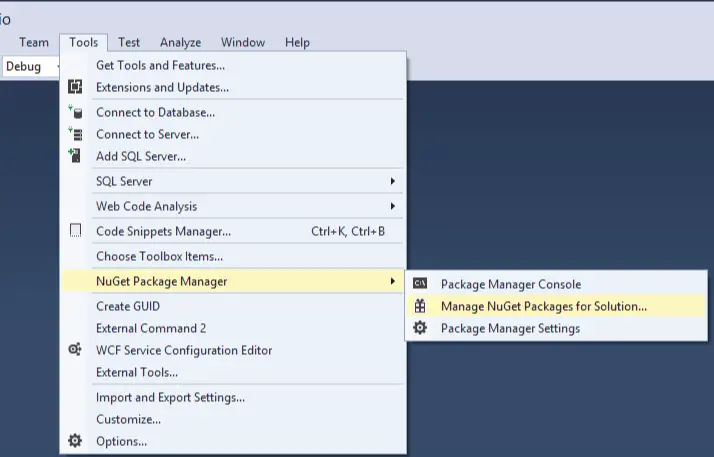
Ezoic

If you are just a beginner then check out my article on [Introduction to ASP.NET Core MVC](https://www.yogihosting.com/aspnet-core-introduction/) which is made specially for beginners.

Install EF Core SQL Server Provider

Entity Framework Core has Database Providers for all major databases. The installation of each of them can be done through NuGet. Here we will install SQL Server database provider since we will be working on a SQL Server Database.

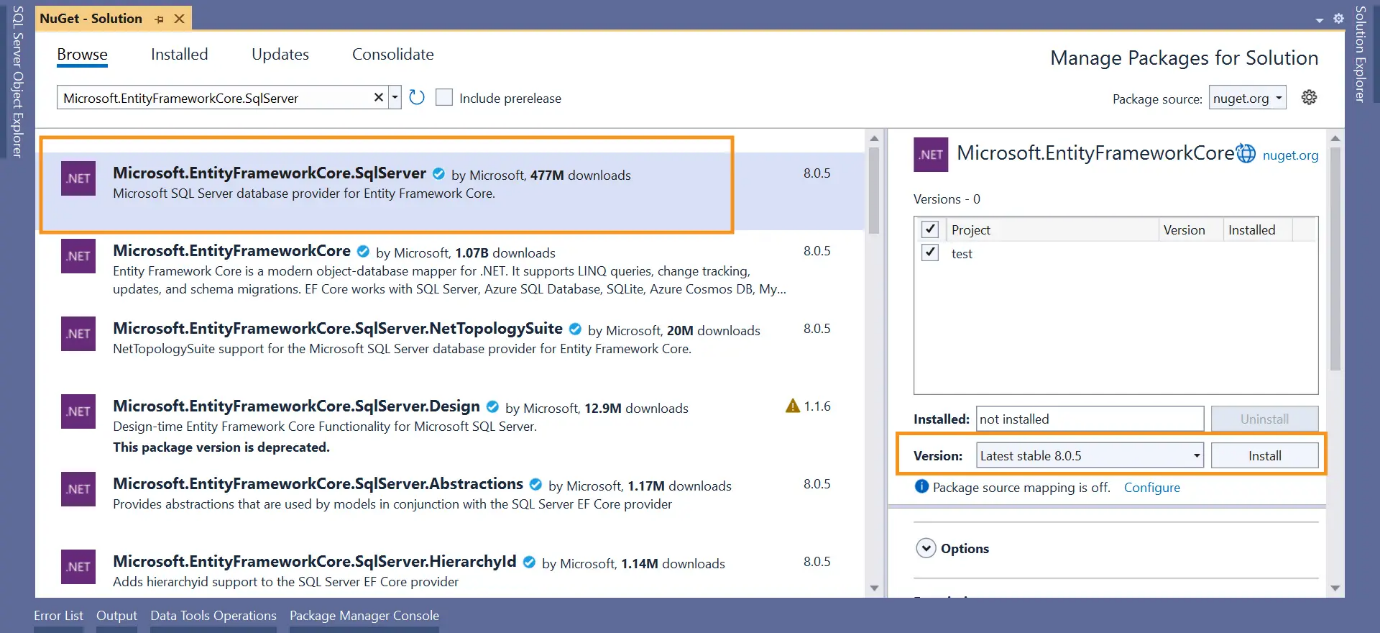
In Visual Studio go to Tools > NuGet Package Manager > Manage NuGet Packages for Solution, this will open the NuGet UI.



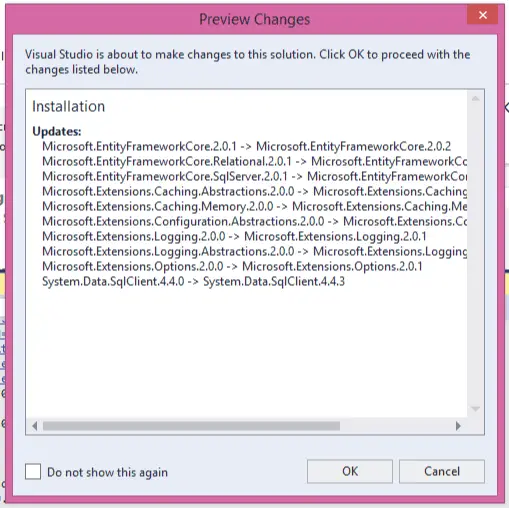
Here click the Browse link and search for Microsoft.EntityFrameworkCore.SqlServer on the text box.

Ezoic

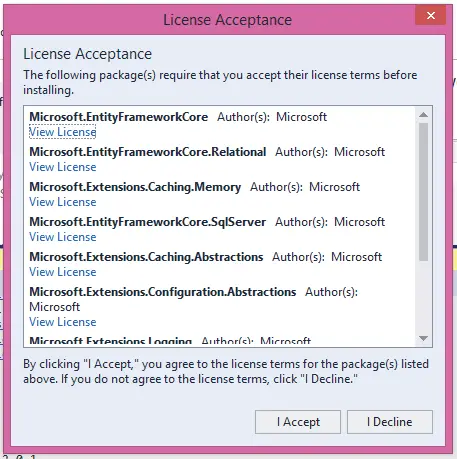
The package will show up. Click on it, then on the right side check the checkbox given against the project and click the Install button.



The installation of the package will start and we will see Preview Changes window. Click the OK button.



Next we will see the License Acceptance window, click the I Accept button.

Ezoic

Within few seconds the installation procedure will complete and the Microsoft.EntityFrameworkCore.SqlServer provider will be installed on the app.

You can verify it by seeing the Dependencies > NuGet section of your Solution Explorer.

Alternatively, go to Tools > NuGet Package Manager > Package Manager Console in Visual Studio, and execute the command PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer. This will install the SqlServer package of Entity Framework Core on the app.

*EF Core is made on top of ADO.NET. I have also written a complete series on ADO.NET which you can start from*[*Learn ADO.NET by building CRUD features in ASP.NET Core Application*](https://www.yogihosting.com/ado-net-aspnet-core/)

Install Entity Framework Core Tools

EzoicThere are many EF Core commands like Migration, scaffoldings that needs to be executed. For this we will need “any” of the two tools. These tools are:

* 1. .NET Core command-line interface (CLI) tools: it can be used on Windows, Linux, or macOS. Their commands begin with “dotnet ef”.
* 2. Package Manager Console (PMC) tools: it can be used only on Visual Studio on Windows. These commands start with a verb, for example “Add-Migration”, “Update-Database”, etc.

Let us understand their installation procedures.

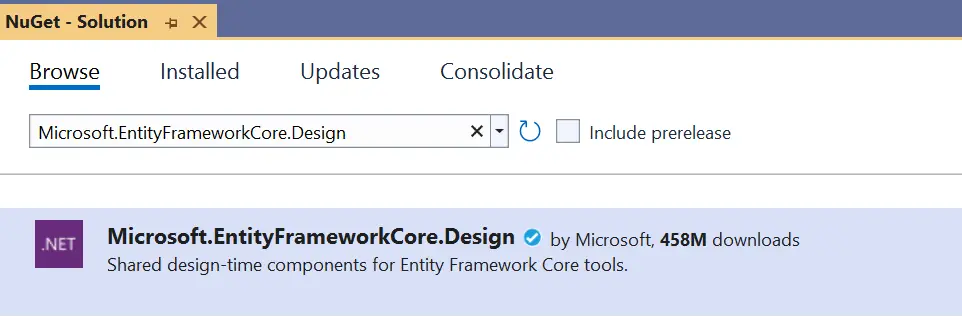
Install .NET Core command-line interface (CLI) tools

First open Package Manager Console window from Tools ➤ NuGet Package Manager ➤ Package Manager Console menu. Then run the following command to install it.

dotnet tool install --global dotnet-ef

Worth Mentioning – If you already had **dotnet ef** installed in your pc then it should be updated to the latest version. Run the following update command to do this job.

dotnet tool update --global dotnet-ef

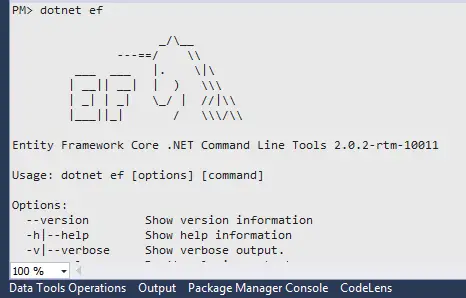
Next, install the Microsoft.EntityFrameworkCore.Design package from NuGet.

Test

Test the packages by running the following command on Package Manager Console.

|  |  |
| --- | --- |
| 1 | PM> dotnet ef |

You will see a horse picture telling the dotnet CLI has been installed successfully.



Install Package Manager Console (PMC) tools

Go to NuGet UI and search for Microsoft.EntityFrameworkCore.Tools, and install it. Check the below image.



Verify the installation by running the following command in Package Manager Console.

Get-Help about\_EntityFrameworkCore

It will show the horse images along with different commands and their work.

Conclusion

We learned how to perform the installation of SqlServer database provider package and EF Core tools in the app. Now it’s time to understand the workings of EF Core. Check the next tutorial whose link is given below and proceed.

In **Database-First** approach the entity and context classes are automatically created by the EF Core from the database. So this means you have to first create your database for the app.

This tutorial is a part of **Entity Framework Core** series.

Let me show you how to do this for a Company’s Database.

Creating Database in SQL Server

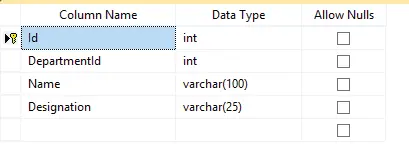
Open the View ➤ SQL Sever Object Explorer in Visual Studio. Then create a simple company’s database in your SQL Server and name it Company. Create 2 tables to it and name them as:

Ezoic

* 1. Employee
* 2. Department

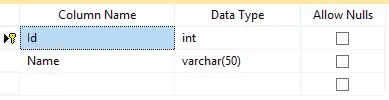
Important ASP.NET Core tutorial based on your interest – [Calling ASP.NET Core Web APIs with JavaScript and performing CRUD operations](https://www.yogihosting.com/aspnet-core-web-api-javascript/)

The Employee Table is shown below:



The id column is both Primary Key and Identity.

The Department Table is shown below:



The id column is both Primary Key and Identity.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=2733bda2-0a9e-4c3e-6487-af8d1ab44bda&d_id=122531&imp_id=8466912823044037&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

Relationship between Tables

There is **One-to-Many relationship** between the Employee & Department table. This means one department can have multiple employees.

Here we have created the DepartmentId column of Employee table as the foreign key for the Id column of the Department table.

Scripts of the Tables

We can create these 2 tables by running the create table scripts on the New Query Window. Right click on the Company database and select “New Query”. A window will open where we need to enter the given scripts and click the “Execute” button.

**Department table**

CREATE TABLE [dbo].[Department](

[Id] [int] IDENTITY(1,1) NOT NULL,

[Name] [varchar](50) NOT NULL,

CONSTRAINT [PK\_Department] PRIMARY KEY CLUSTERED

(

[Id] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

**Employee table**

CREATE TABLE [dbo].[Employee](

[Id] [int] IDENTITY(1,1) NOT NULL,

[DepartmentId] [int] NOT NULL,

[Name] [varchar](100) NOT NULL,

[Designation] [varchar](25) NOT NULL,

CONSTRAINT [PK\_Employee] PRIMARY KEY CLUSTERED

(

[Id] ASC

)WITH (PAD\_INDEX = OFF, STATISTICS\_NORECOMPUTE = OFF, IGNORE\_DUP\_KEY = OFF, ALLOW\_ROW\_LOCKS = ON, ALLOW\_PAGE\_LOCKS = ON) ON [PRIMARY]

) ON [PRIMARY]

GO

SET ANSI\_PADDING OFF

GO

ALTER TABLE [dbo].[Employee] WITH CHECK ADD CONSTRAINT [FK\_Employee\_Department] FOREIGN KEY([DepartmentId])

REFERENCES [dbo].[Department] ([Id])

GO

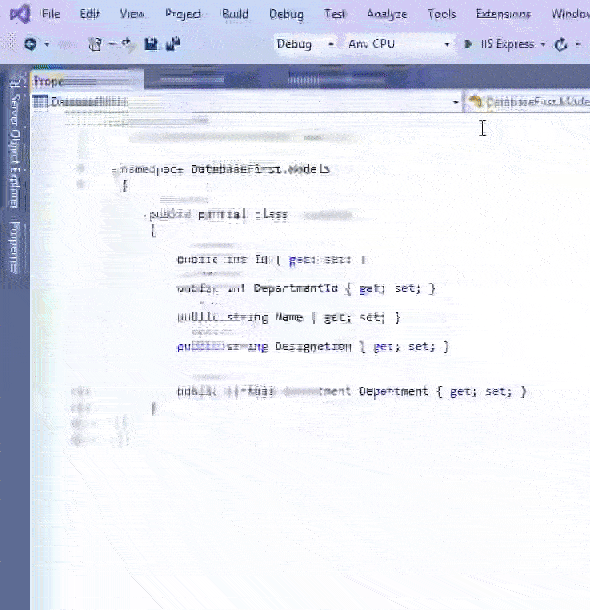
ALTER TABLE [dbo].[Employee] CHECK CONSTRAINT [FK\_Employee\_Department]

GO

Entity Framework Core Database Connection String

A database connection string contains information about a data source which is a database engine, as well as the information necessary to connect to it. Entity Framework Core needs connection string to communicate with the database and perform database operations like creating records, reading records and so on. We can easily find the **Database Connection String** by opening the SQL Sever Object Explorer then right click on the database to open the “Properties” options. Select it.

In the properties window find the Connection String field and simply copy it’s value from there. We have shown this in the below video.



The Database Connection String in our case is:

Server=vaio;Database=Company;Trusted\_Connection=True;

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=2733bda2-0a9e-4c3e-6487-af8d1ab44bda&d_id=122531&imp_id=508694815016891&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

EzoicThe connection string has following informations:

* The Database server name is given as Server=vaio.
* Database=Company is the name of the database.
* Trusted\_Connection=True specifies the Windows authentication i.e. it will use Windows credentials to connect to the SQL Server. On live servers we use SQL Server Authentication which has an SQL Server user name and password.

.NET Core command-line interface (CLI) Scaffold Command

Now we run the CLI Scaffold Command on the Package Manager Console window. Open this window from Tools ➤ NuGet Package Manager ➤ Package Manager Console menu in Visual Studio.

Before running the command make sure you have Install EF Core Tools installed in your project. I have covered this on [Installation of Entity Framework Core](https://www.yogihosting.com/install-entity-framework-core/) tutorial.

The command to run is shown below. Make sure to run this command from the directory of your project file.

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

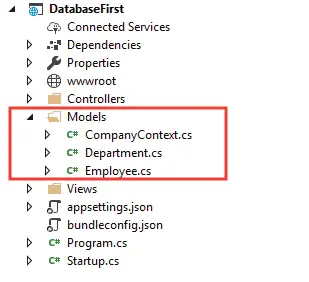
Note the following things:

* Inside the quotation marks (“”) is the Database Connection String which is Server=vaio;Database=Company;Trusted\_Connection=True;. It may be different on your case.
* Microsoft.EntityFrameworkCore.SqlServer; is the name of the provider for SQL Server.
* -o Models refers to the directory name where all the classes will be generated. Here it is the Models folder.

The command will take nearly 10 to 20 seconds to execute and it will generate the context & entity model classes inside the Models folder.

Context & Entity Classes

We will find the context & entity classes generated by DOT NET inside the Models folder.



The 2 entity Classes created are the Employee.cs & Department.cs.

Employee.cs

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **using** System;  **using** System.Collections.Generic;    **namespace** DatabaseFirst.Models;    **public** **partial** **class** Employee  {  **public** **int** Id { **get**; **set**; }    **public** **int** DepartmentId { **get**; **set**; }    **public** **string** Name { **get**; **set**; } = **null**!;    **public** **string** Designation { **get**; **set**; } = **null**!;    **public** **virtual** Department Department { **get**; **set**; } = **null**!;  } |

Department.cs

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **using** System;  **using** System.Collections.Generic;    **namespace** DatabaseFirst.Models;    **public** **partial** **class** Department  {  **public** **int** Id { **get**; **set**; }    **public** **string** Name { **get**; **set**; } = **null**!;    **public** **virtual** ICollection<Employee> Employees { **get**; **set**; } = **new** List<Employee>();  } |

[Xaero – Entity Framework Core Advanced Project](https://www.yogihosting.com/xaero-project-entity-framework-core/) is my latest project where I have created a full Movie Database based ASP.NET Core App in Entity Framework Core. In this project you will find lots and lots of reusable high quality codes.

The database context class created is CompanyContext.cs

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=2733bda2-0a9e-4c3e-6487-af8d1ab44bda&d_id=122531&imp_id=5379307037081006&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

CompanyContext.cs

using System;

using System.Collections.Generic;

using Microsoft.EntityFrameworkCore;

namespace DatabaseFirst.Models;

public partial class CompanyContext : DbContext

{

public CompanyContext()

{

}

public CompanyContext(DbContextOptions<CompanyContext> options)

: base(options)

{

}

public virtual DbSet<Department> Departments { get; set; }

public virtual DbSet<Employee> Employees { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

=> optionsBuilder.UseSqlServer("Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Company;Integrated Security=True;Connect Timeout=30;Encrypt=False;Trust Server Certificate=False;Application Intent=ReadWrite;Multi Subnet Failover=False");

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Department>(entity =>

{

entity.ToTable("Department");

entity.Property(e => e.Name)

.HasMaxLength(50)

.IsUnicode(false);

});

modelBuilder.Entity<Employee>(entity =>

{

entity.ToTable("Employee");

entity.Property(e => e.Designation)

.HasMaxLength(25)

.IsUnicode(false);

entity.Property(e => e.Name)

.HasMaxLength(100)

.IsUnicode(false);

entity.HasOne(d => d.Department).WithMany(p => p.Employees)

.HasForeignKey(d => d.DepartmentId)

.OnDelete(DeleteBehavior.ClientSetNull)

.HasConstraintName("FK\_Employee\_Department");

});

OnModelCreatingPartial(modelBuilder);

}

partial void OnModelCreatingPartial(ModelBuilder modelBuilder);

}

Package Manager Console (PMC) Scaffold-DbContext Command

We can skip the **DotNet CLI Scaffold Command** and use **Scaffold-DbContext Command** for creating context & entity classes from a database. This will do the exact same thing like before.

Run the following command on the Package Manager Console.

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

Download the source codes:

[DOWNLOAD](https://www.yogihosting.com/wp-content/themes/yogi-yogihosting/download/efcore/DatabaseFirst.zip)

Conclusion

We just finished understanding **Entity Framework Core Database First approach** and created the necessary model and database context from the sql server database by running the Scaffold commands. However just keep this in mind that Database First approach is not used anymore instead we will be using Code-First Approach as Microsoft itself put emphasis on using this approach.

Before that we need to understand how Database Context will work. This is the tutorial coming next – [DbContext Class in Entity Framework Core](https://www.yogihosting.com/dbcontext-entity-framework-core/)

**DbContext Class in Entity Framework Core**

Last Updated: October 10, 2022

**Entity Framework Core Database Context is an important class which is used to maintain session with the database. It thus help in performing all types database operations like creating, reading, updating and deleting records. We create a database context for the app by inheriting a class from the DbContext class of the Microsoft.EntityFrameworkCore namespace.**

This tutorial is a part of **Entity Framework Core** series.

* 1. [Introduction to Entity Framework Core](https://www.yogihosting.com/introduction-entity-framework-core/)
* 2. [Installation of Entity Framework Core](https://www.yogihosting.com/install-entity-framework-core/)
* 3. [Database-First approach in Entity Framework Core](https://www.yogihosting.com/database-first-approach-entity-framework-core/)
* 4. *DbContext Class in Entity Framework Core*
* 5. [Code-First Approach in Entity Framework Core](https://www.yogihosting.com/code-first-entity-framework-core/)
* 6. [Migrations in Entity Framework Core](https://www.yogihosting.com/migrations-entity-framework-core/)
* 7. [Insert Records in Entity Framework Core](https://www.yogihosting.com/insert-records-entity-framework-core/)
* 8. [Read Records in Entity Framework Core](https://www.yogihosting.com/read-records-entity-framework-core/)
* 9. [Update Records in Entity Framework Core](https://www.yogihosting.com/update-records-entity-framework-core/)
* 10. [Delete Records in Entity Framework Core](https://www.yogihosting.com/delete-records-entity-framework-core/)
* 11. [Conventions in Entity Framework Core](https://www.yogihosting.com/conventions-entity-framework-core/)
* 12. [Configurations in Entity Framework Core](https://www.yogihosting.com/configurations-entity-framework-core/)
* 13. [Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-entity-framework-core/)
* 14. [Configure One-to-Many relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-one-to-many-relationship-entity-framework-core/)
* 15. [Configure One-to-One relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-one-to-one-relationship-entity-framework-core/)
* 16. [Configure Many-to-Many relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-many-to-many-relationship-entity-framework-core/)
* 17. [Execute Raw SQL Queries using FromSqlRaw() method in Entity Framework Core](https://www.yogihosting.com/raw-sql-queries-entity-framework-core/)
* 18. [Execute SQL Stored Procedures using FromSqlRaw() & ExecuteSqlRawAsync() methods in Entity Framework Core](https://www.yogihosting.com/stored-procedures-entity-framework-core/)

Some common tasks performed through DbContext are:

* 1. Manage Database Connection.
* 2. Configure Entities and the Relationships between them.
* 3. Reading, Creating, Updating & Deleting data in the database.
* 4. Configure change tracking.
* 5. Caching.
* 6. Transaction management

**Page Contents**

* [Entity Framework Core Example project](https://www.yogihosting.com/dbcontext-entity-framework-core/#project)
* [Create DbContext for project](https://www.yogihosting.com/dbcontext-entity-framework-core/#create-dbcontext)
  + [The OnModelCreating() method](https://www.yogihosting.com/dbcontext-entity-framework-core/#onmodelcreating)
  + [Register the Database Context](https://www.yogihosting.com/dbcontext-entity-framework-core/#register)
  + [Storing Database Connection String in “appsettings.json”](https://www.yogihosting.com/dbcontext-entity-framework-core/#appsettings-json)
  + [Running EF Core Migrations](https://www.yogihosting.com/dbcontext-entity-framework-core/#migrations)
  + [DbContext Methods](https://www.yogihosting.com/dbcontext-entity-framework-core/#methods)
* [The OnConfiguring() method](https://www.yogihosting.com/dbcontext-entity-framework-core/#onconfiguring)
* [Download Source Codes](https://www.yogihosting.com/dbcontext-entity-framework-core/#download)

Entity Framework Core Example project

We will now create an **Entity Framework Core Example project** to understand how to work with Database Context (DBContext) class. Start by creating a new ASP.NET Core MVC project and name it EFCoreExample. Next install the NuGet package called Microsoft.EntityFrameworkCore.SqlServer. It is the database provider for SQL Server. Run the following command on the Package Manager Console to install the latest version of this package.

Ezoic

PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer

Note: On older DOT NET projects we have to install a supported version of this package. We do this by adding the version number after the above command. For example if our project is in DOT NET 5.0 version then we have to install the version 5.0.17 of this package which is compatible with 5.0 version of dot net. The installation command for this pacakge is:

PM> Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 5.0.17

All the versions and their commands are listed on [NuGet website](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer/).

I have covered all the details of EF Core installation details on my other tutorial – [Installation of Entity Framework Core](https://www.yogihosting.com/install-entity-framework-core/). Kindly read it as you will find it extremely useful.

This example project will work with a Database of a very small company. The database will have just 2 table which are:

* Department – for keeping information of the different departments of the company.
* Employee – for keeping information of the different employees of the company.

For representing these 2 tables we will need to create 2 entity model classes. So create 2 entity classes called Department.cs & Employee.cs inside the Models folder. The codes of these classes are given below:

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=5c305a7f-e127-490d-40cb-36d39e23ce8c&d_id=122531&imp_id=3828535557000185&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

1. Department.cs

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **namespace** EFCoreExample.Models  {  **public** **class** Department      {  **public** **int** Id { **get**; **set**; }  **public** **string** Name { **get**; **set**; }    **public** ICollection<Employee> Employee { **get**; **set**; }      }  } |

2. Employee.cs

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **namespace** EFCoreExample.Models  {  **public** **class** Employee      {  **public** **int** Id { **get**; **set**; }  **public** **int** DepartmentId { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** **string** Designation { **get**; **set**; }    **public** Department Department { **get**; **set**; }      }  } |

The “Employee” property in the Department.cs and “Department” property in the Employee.cs are navigation properties. A Navigation property hold other entities that are related to it. The entity classes are forming One-to-Many Relationship between them. We have covered relationship in [Conventions in Entity Framework Core](https://www.yogihosting.com/conventions-entity-framework-core/).

Now it’s time to create DbContext class.

Create DbContext for project

Start by creating a new class called CompanyContext.cs inside the “Models” folder of the app and defive it from the **DbContext** class of Microsoft.EntityFrameworkCore. Also add a constructor that calls the base class constructor. This class is the database context of the app.

Ezoic

The CompanyContext.cs will look like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Models  {  **public** **class** CompanyContext : DbContext      {  **public** CompanyContext(DbContextOptions<CompanyContext> options) : **base**(options)          {          }      }  } |

Next add the DbSet<T> for the 2 entity class which are Department and Employee. This will tell EF core 2 things:

1. These entities correspond to tables in the database.
2. These entities correspond to rows in the tables.

The DbSet<T> represents a collection for a given entity and is the gateway to perform database operations against the entity.

EzoicThis will make the CompanyContext.cs look this way:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Models  {  **public** **class** CompanyContext : DbContext      {  **public** CompanyContext(DbContextOptions<CompanyContext> options) : **base**(options)          {          }    **public** DbSet<Department> Department { **get**; **set**; }  **public** DbSet<Employee> Employee { **get**; **set**; }      }  } |

The OnModelCreating() method

The OnModelCreating() method of the DbContext class allows us to tell Entity Framework Core more about the entities like:

1. Length of a property of an entitiy.
2. Whether a property is required by default.
3. Relationships between the entities. One-to-Many, One-to-One, etc.

So add the OnModelCreating method to the CompanyContext.cs as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Models  {  **public** **class** CompanyContext : DbContext      {  **public** CompanyContext(DbContextOptions<CompanyContext> options) : **base**(options)          {          }    **public** DbSet<Department> Department { **get**; **set**; }  **public** DbSet<Employee> Employee { **get**; **set**; }    **protected** **override** **void** OnModelCreating(ModelBuilder modelBuilder)          {              modelBuilder.Entity<Department>(entity =>              {                  entity.Property(e => e.Name)                      .IsRequired()                      .HasMaxLength(50)                      .IsUnicode(**false**);              });                modelBuilder.Entity<Employee>(entity =>              {                  entity.Property(e => e.Designation)                      .IsRequired()                      .HasMaxLength(25)                      .IsUnicode(**false**);                    entity.Property(e => e.Name)                      .IsRequired()                      .HasMaxLength(100)                      .IsUnicode(**false**);                    entity.HasOne(d => d.Department)                      .WithMany(p => p.Employee)                      .HasForeignKey(d => d.DepartmentId)                      .OnDelete(DeleteBehavior.ClientSetNull)                      .HasConstraintName("FK\_Employee\_Department");              });          }      }  } |

We have configured both the Department and the Employee entities. By using IsRequired(), we am making the Name property of the Department class as the required one. Similarly the HasMaxLength(100) method is used to set the maximum length of Nameproperty of the employee class.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=5c305a7f-e127-490d-40cb-36d39e23ce8c&d_id=122531&imp_id=7075857109078388&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

We also set the One-to-Many relationship between these entities with a foreign key through the code.

.HasForeignKey(d => d.DepartmentId)

.OnDelete(DeleteBehavior.ClientSetNull)

.HasConstraintName("FK\_Employee\_Department");

*These things are known as FLuent API and we have covered this topic in full details at*[*Fluent API in Entity Framework Core*](https://www.yogihosting.com/fluent-api-entity-framework-core/)*.*

Register the Database Context

We need to register the DbContext of the app as a service in startup or program class depending upon the version of DOT NET we are using. On ASP.NET Core 6.0 and newer versions we register the DbContext on the Program.cs class by adding the highlighted code which is given below to it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **using** EFCoreExample.Models;  **using** Microsoft.EntityFrameworkCore;    **var** builder = WebApplication.CreateBuilder(args);    builder.Services.AddDbContext<CompanyContext>(options =>    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));    // Add services to the container.  builder.Services.AddControllersWithViews();    **var** app = builder.Build();  ..... |

In prior versions we do this in Startup.cs class of the app. Check the highlighted lines given below where we are doing this registration process. You will also need to inject IConfiguration by Dependency Injection to the Startup.cs class constructor.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | **using** Microsoft.AspNetCore.Builder;  **using** Microsoft.AspNetCore.Hosting;  **using** Microsoft.Extensions.DependencyInjection;  **using** Microsoft.Extensions.Hosting;  **using** DB\_Context.Models;  **using** Microsoft.EntityFrameworkCore;  **using** Microsoft.Extensions.Configuration;    **namespace** EFCoreExample  {  **public** **class** Startup      {  **public** Startup(IConfiguration configuration)          {              Configuration = configuration;          }  **public** IConfiguration Configuration { **get**; }    **public** **void** ConfigureServices(IServiceCollection services)          {              services.AddDbContext<CompanyContext>(options =>                  options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));                services.AddControllersWithViews();          }    **public** **void** Configure(IApplicationBuilder app, IWebHostEnvironment env)          {              // ...          }      }  } |

Storing Database Connection String in “appsettings.json”

We will be storing the database connection string in the appsettings.json file of the app. So add the **Database Connection String** to it as shown below.

{

"ConnectionStrings": {

"DefaultConnection": "Server=vaio;Database=Company;Trusted\_Connection=True;"

}

}

Running EF Core Migrations

It’s time to run the Migrations in order to create the database from the entity classes. Before that make sure you have the package “Microsoft.EntityFrameworkCore.Tools” installed in your project.

On the Package Manager Console window run these 2 commands one by one:

Ezoic

1. Add Migration Command

PM> add-migration Migration1

After this command is executed we will find a new folder called Migrations is created in the project folder. Check the Solution Explorer in Visual studio to find 2 .cs files inside the Migrations folder. We have shown this in the below image.

2. Update Migration Command

Next run the below given command which will create the database from the migration files.

PM> Update-Database

Now check the SQL server where we will find the Company database is created. We have shown this in the below image.

We have only introduced EF Core Migrations. In fact there are lots of Migration commands which are covered in a separate article, check it – [Migrations in Entity Framework Core](https://www.yogihosting.com/migrations-entity-framework-core/)

DbContext Methods

Some important methods of **Entity Framework Core DbContext** class are.

| **Method** | **Description** |
| --- | --- |
| Add | Adds a new entity with Added state |
| AddRange | Adds a collection of new entities with Added state |
| Attach | Attaches a new or existing entity with Unchanged state |
| AttachRange | Attaches a collection of new or existing entity with Unchanged state |
| Remove | Attaches an entity with Deleted state |
| RemoveRange | Attaches a collection of entities with Deleted state |
| Update | Attaches disconnected entity with Modified state |
| UpdateRange | Attaches collection of disconnected entity with Modified state |
| SaveChanges | Execute INSERT, UPDATE or DELETE command to the database for the entities with Added, Modified or Deleted state. |

Testing

Now it’s time to test if everything is working correctly or not. So let us insert a record into the database through Entity Framework Core. So add the below given code to the HomeController.cs file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | **using** EFCoreExample.Models;  **using** Microsoft.AspNetCore.Mvc;  **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Controllers  {  **public** **class** HomeController : Controller      {  **private** CompanyContext context;  **public** HomeController(CompanyContext cc)          {              context = cc;          }    **public** IActionResult Index()          {  **var** dept = **new** Department()              {                  Name = "Designing"              };              context.Entry(dept).State = EntityState.Added;              context.SaveChanges();    **return** View();          }      }  } |

First we added a dependency for the database context in the constructor of the controller. The dependency injection technique will provide us with the object of our DbContext which is registered on the program class as a service.

private CompanyContext context;

public HomeController(CompanyContext cc)

{

context = cc;

}

After that we are perfoming a new record creation in the database with EF Core.

Ezoic

var dept = new Department()

{

Name = "Designing"

};

context.Entry(dept).State = EntityState.Added;

context.SaveChanges();

Now run the project and initiate this action’s in the browser. This will insert a new record on the Department table of the database. Open the table to find the newly added record. See the below image of the database table.

EF core is super easy to work with and codes are very small. Writing the same thing in ADO.NET would have required 3 times more work. I have written a complete tutorial on this thing, you should check the tutorial called [Insert Records in Entity Framework Core](https://www.yogihosting.com/insert-records-entity-framework-core/).

The OnConfiguring() method

The OnConfiguring() method of the DbContext class can be used to configure the data source for Entity Framework Core. Add this method to the CompanyContext.cs class as shown below. Notice we are providing the database connection string to the UseSqlServer() method of the DbContextOptionsBuilder() class object.

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

if (!optionsBuilder.IsConfigured)

{

optionsBuilder.UseSqlServer(@"Server=vaio;Database=Company;Trusted\_Connection=True;");

}

}

If we use this method then we don’t have to add the connection string to the appsettings.json nor we have to register the DbContext file. However don’t use it on production since it is a poor coding style.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=5c305a7f-e127-490d-40cb-36d39e23ce8c&d_id=122531&imp_id=6746133841005614&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

You can download the source code from the below link.

[DOWNLOAD](https://www.yogihosting.com/wp-content/themes/yogi-yogihosting/download/efcore/EFCoreExample.zip)

Conclusion

In this tutorial we covered A-Z of EF Core Database Context which is going to help in the development of your projects. Now we can easily create, update, read and delete records from the database. So let’s move forward to our next tutorial.

**Code-First Approach in Entity Framework Core**

Last Updated: September 26, 2022

**The Entity Framework Core Code-First approach creates the database and tables based on entity classes and configurations given on DbContext. The Code-First Approach is helpful in situations where we are beginning a new project and don’t have a clear picture of the database. This is the preferred approach when working with EF Core and the creation of database & tables are done when migration commands are run.**

This tutorial is a part of **Entity Framework Core** series.

If you had worked with previous Entity Framework 6 then you would have used the Database-First approach and created [EDMX](https://www.yogihosting.com/entity-framework-create-edmx-file/) file. The EDMX file contains snapshot of the database, it’s table, stored procedures, relationships, etc. Now the shift is mode to Code-First approach, in-fact Microsoft has placed heavy emphasis on the Code-First approach. We can say Database-First approach is all but dead.

**Page Contents**

* [Example Project](https://www.yogihosting.com/code-first-entity-framework-core/#example)
* [Create a Record](https://www.yogihosting.com/code-first-entity-framework-core/#create)
* [Entity Framework Core Seed Data](https://www.yogihosting.com/code-first-entity-framework-core/#seed)
* [Download Source Codes](https://www.yogihosting.com/code-first-entity-framework-core/#download)

Example Project

To understand how the **Code-First approach** works, first create a new ASP.NET Core project and [Install Entity Framework Core](https://www.yogihosting.com/install-entity-framework-core/) package on it. In this project we will be dealing with Companies having areas like it’s employees, departments, etc.

Create Entity & DbContext

Create a class called Information.cs inside the Models folder. It will contain information about the different companies like company id, company name, license, year of establishment and yearly revenue generated by them.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ecea9c90-90fb-43bd-6dd7-9e1337411c67&d_id=122531&imp_id=3592833959012550&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **namespace** EFCoreExample.Models  {  **public** **class** Information      {  **public** **int** Id { **get**; **set**; }    **public** **string** Name { **get**; **set**; }    **public** **string** License { **get**; **set**; }    **public** DateTime Establshied { **get**; **set**; }    **public** **decimal** Revenue { **get**; **set**; }      }  } |

Next create a [**Database Context (DbContext)**](https://www.yogihosting.com/dbcontext-entity-framework-core/) for EF Core and name it CompanyContext.cs. Place it inside the Models folder.

using Microsoft.EntityFrameworkCore;

namespace EFCoreExample.Models

{

public class CompanyContext : DbContext

{

public CompanyContext(DbContextOptions<CompanyContext> options) : base(options)

{

}

public DbSet<Information> Information { get; set; }

}

}

We now register the DbContext as a service in the Program.cs class of the app. So we add the below given code line to it.

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

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

The connection string to the database is stored in the appsettings.json file and is shown below.

{

"ConnectionStrings": {

"DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Company;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False"

}

}

EF Core Migrations

With Migrations we can create the database and it’s tables based on the **Entity & Database Context classes**. Check that in the database connection string we have provided the database. Here the database name is Company and is located in the SQL Server Express LocalDB.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ecea9c90-90fb-43bd-6dd7-9e1337411c67&d_id=122531&imp_id=7406167511049886&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

In Visual Studio, open NuGet Package Manager Console window from Tools ➤ NuGet Package Manager ➤ Package Manager Console and enter the following PMC command to create a migration.

PM> add-migration CompanyDB

Or, we can do the same thing by executing the CLI command given below.

PM> dotnet ef migrations add CompanyDB

The command will create a Migrations folder on the project.

I have written complete tutorial on [EF Core Migrations](https://www.yogihosting.com/migrations-entity-framework-core/) which you will find very useful.

Now we create the database by executing the update-database PMC command on the Package Manager Console:

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ecea9c90-90fb-43bd-6dd7-9e1337411c67&d_id=122531&imp_id=6533049867052643&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

PM> update-database

Or, we can run the CLI command to do the same thing:

PM> dotnet ef database update

The command will execute and will create the Company database which we can open and see on the SQL Server Object Explorer windown in Visual Studio.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ecea9c90-90fb-43bd-6dd7-9e1337411c67&d_id=122531&imp_id=2090809801018525&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

EzoicCongrats, we have successfully created the database by using **Entity Framework Core Code-First approach**. Next we are going to create a new record on this database.

Create a Record

Let us Create a new Record on the Information table with Entity Framework Core. We add a new action method on a controller that will create a record. The code is given below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **public** **class** HomeController : Controller  {  **private** CompanyContext context;  **public** HomeController(CompanyContext cc)      {          context = cc;      }    **public** IActionResult CreateInformation()      {  **var** info = **new** Information()          {              Name = "YogiHosting",              License = "XXYY",              Revenue = 1000,              Establshied = Convert.ToDateTime("2014/06/24")          };          context.Entry(info).State = EntityState.Added;          context.SaveChanges();    **return** View();      }  } |

Once the Action is called, the code will execute, and a new record is added to the Information table. We can see the new record on the SQL Server database.

Entity Framework Core Seed Data

We can create the database and populate it with test data through our code. This is known as seeding the database. For doing this create a new static class called DbInitializer.cs inside the “Data” folder of the app. The Initialize method does the seeding part and contains test records which will be inserted to the database.

using EFCoreExample.Models;

namespace EFCoreExample.Data

{

public static class DbInitializer

{

public static void Initialize(CompanyContext context)

{

// Look for any students.

if (context.Information.Any())

{

return; // DB has been seeded

}

var infos = new Information[]

{

new Information { Name = "YogiHosting", License = "XXYY", Revenue = 1000, Establshied = Convert.ToDateTime("2014/06/24") },

new Information{ Name ="Microsoft", License ="XXXX", Revenue = 1000, Establshied = Convert.ToDateTime("2014/07/14") },

new Information{ Name ="Google", License ="RRRRR", Revenue = 1000, Establshied = Convert.ToDateTime("2019/06/18") },

new Information{ Name ="Apple", License ="XADFD", Revenue = 1000, Establshied = Convert.ToDateTime("2022/02/02") },

new Information{ Name ="SpaceX", License ="##@$", Revenue = 1000, Establshied = Convert.ToDateTime("2030/10/01") }

};

context.Information.AddRange(infos);

context.SaveChanges();

}

}

}

Now call this class from the Program class of the app by adding the below code.

using (var scope = app.Services.CreateScope())

{

var services = scope.ServiceProvider;

var context = services.GetRequiredService<CompanyContext>();

context.Database.EnsureCreated();

DbInitializer.Initialize(context);

}

The EnsureCreated method creates the database if no database exists. After that the DbInitializer.Initialize(context) is called and it does the seeding part.

Test it by first dropping the database and then running the app. The app checks for the database and finds that it does not exists. It then creates the database and adds test data. Using this approach we can bypass the Migration commands which we used earlier to create the database.

Download the source codes from here.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ecea9c90-90fb-43bd-6dd7-9e1337411c67&d_id=122531&imp_id=5009304441051730&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

[DOWNLOAD](https://www.yogihosting.com/wp-content/themes/yogi-yogihosting/download/efcore/EFCoreExample.zip)

Conclusion

We learned the concept of CODE-FIRST approach in Entity Framework Core. We created the database from entity classes and then seeded the database. Now we are ready to create the CRUD Operations in EF Core and this is the topic of the next tutorial.

**Entity Framework Core Migrations** keep the database synchronized with the domain entity classes and configurations given on DbContext. Migrations will create or update the database in a very easy manner. When a project in under development, the programmers keep on updating the entity classes, therefore they need to run migrations inorder to keep the database schema up to date.

This tutorial is a part of **Entity Framework Core** series.

How to run Migrations

EF Core Migrations are run from the Package Manager Console window. This window can be opened from Tools ➤ NuGet Package Manager ➤ Package Manager Console menu of Visual Studio. Migrations require either .NET Core command-line interface (CLI) tools or Package Manager Console (PMC) tools to be installed.

Run the following command to install CLI tools.

Ezoic

PM> dotnet tool install --global dotnet-ef

If it is already installed then kindly update it to the latest version. Run the following update command to do the updation.

PM> dotnet tool update --global dotnet-ef

To install PMC tools run the following command.

PM> Install-Package Microsoft.EntityFrameworkCore.Tools

More information can be optained from [Installation of Entity Framework Core](https://www.yogihosting.com/install-entity-framework-core/). Now we are ready to look into some of the most important migration commands.

Add Migration Command

The **Add Migration** command will **create Migration files** that store information from your entity classes and DbContext. On the Tools > NuGet Package Manager > Package Manager Console execute any one of the 2 commands, that are given below, to create the migration.

PM> dotnet ef migrations add Migration1

or

PM> add-migration Migration1

Here Migration1 is the name of the migration and can be any name of our choice. The migration command will create a folder named Migrations on the root of the app. This folder contains 3 files.

* **\_.cs**: It is the main migration file which includes migration operations named Up() and Down() methods. The Up method is responsible for creating DB objects while the Down method removes them.
* **\_.Designer.cs**: The migrations metadata file which contains some db related information.
* **ModelSnapshot.cs**: A snapshot of the current model. This is used to determine what has changed when creating the next migration.

Check the below given image of these files.

Ezoic



Now the migration is created and we can create the database through them.

You will be interested to read [Execute SQL Stored Procedures using FromSqlRaw() & ExecuteSqlRawAsync() methods in Entity Framework Core](https://www.yogihosting.com/stored-procedures-entity-framework-core/).

Update Migration Command

The **Update Migration** command will update the database to the latest migration. If the database is not present then it will create a new database else the database will be updated based the informations given on the migration.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=9c12db2e-9915-4e81-7ec3-fa5808317b52&d_id=122531&imp_id=5655711681062044&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

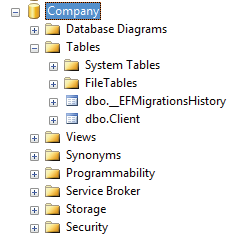
EzoicWe can run either of the 2 update migration command given below:

PM> dotnet ef database update

or

PM> Update-Database

Once the update migration command finish executing, we can find a new database is created. The database will have a table called \_EFMigrationsHistory which stores the name of all migrations that are applied.



Using the –context Keyword

If an app has **more than 1 Database Context file** then with the –context keyword we can specify which DbContext file the migration will target.

Example – There are 2 Database Context files which are – CompanyContext.cs & EmployeeContext.cs. Then we tell migrations to target “EmployeeContext.cs” as:

PM> dotnet ef migrations add Migration1 --context EmployeeContext

PM> dotnet ef database update --context EmployeeDbContext

How to Revert Database to Previous State

We can revert EF Core migrations quite easily. Let’s see with an example. We previously had an entity class Client.cs and we already had the database containing a table “Client” for this entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **public** **class** Client  {  **public** **int** Id { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** **string** Country { **get**; **set**; }  **public** **string** TelephoneNo { **get**; **set**; }  **public** **string** Email { **get**; **set**; }  } |

Suppose we decided to add an “Address” property to the entity class.

public string Address { get; set; }

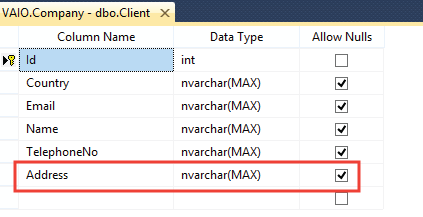
We now run the entity framework core migrations (shown below) to update the “Client” table.

|  |  |
| --- | --- |
| 1  2 | PM> dotnet ef migrations add Migration1  PM> dotnet ef database update |

The Client table on the database will get the address field added to it.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=9c12db2e-9915-4e81-7ec3-fa5808317b52&d_id=122531&imp_id=2795677569054832&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

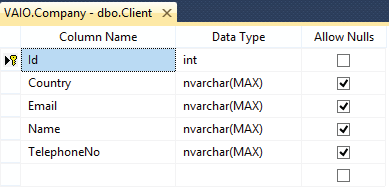
Ezoic



Suppose some situation arises and you need to **Revert the database to the previous state** i.e. when the Client table does not have the Address field. To do this, we remove the Address property from the Client table and rerun the add and update migration commands but this time we name the migration as “Migration2”.

|  |  |
| --- | --- |
| 1  2 | PM> dotnet ef migrations add Migration2  PM> dotnet ef database update Migration2 |

The Client table will have the Address column removed from it.



In this way **Entity Framework Core revert Migration** is achieved.

Remove Migration Command

With the Remove Migration command we can remove the last migration if it is not applied to the database. For doing this we can execute either of the 2 command given below:

Ezoic

PM> dotnet ef migrations remove

or

PM> remove-migration

If the migration has already been applied to the database then we will receive the following error – “The migration has already been applied to the database. Revert it and try again. If the migration has been applied to other databases, consider reverting its changes using a new migration.”

Drop Database

To drop the database use any of the following command.

PM> dotnet ef database drop

or

PM> Drop-Database

Generate SQL Script

We can also **Generate SQL Scripts of the Database**. To do this, execute either of the following 2 commands.

PM> dotnet ef migrations script

or

PM> script-migration

The script command will generate a script for all the migrations by default. This generated db script can be executed to form a duplicate database if we have a requirement.

Ezoic

**Insert Records in Entity Framework Core**

Last Updated: October 6, 2023

In this tutorial we will learn to **Insert Records in Database with Entity Framework Core**. EF Core can insert a single record or bulk records at the same time. Note that entities that have EntityState value as ‘Added’ are insterted to the database.

**Page Contents**

* [EntityState](https://www.yogihosting.com/insert-records-entity-framework-core/#es)
* [Entity Framework Core Create Records](https://www.yogihosting.com/insert-records-entity-framework-core/#cr)
* [Insert/Create a Single Record on the Database](https://www.yogihosting.com/insert-records-entity-framework-core/#single)
* [Entity Framework Core Bulk Insert](https://www.yogihosting.com/insert-records-entity-framework-core/#bulk)
* [TryUpdateModelAsync](https://www.yogihosting.com/insert-records-entity-framework-core/#modelupdate)
* [Entity Framework Core Insert Related Records](https://www.yogihosting.com/insert-records-entity-framework-core/#related)
* [Entity Framework Core CRUD Operations – CREATE RECORDS](https://www.yogihosting.com/insert-records-entity-framework-core/#crud)
* [Download Source Codes](https://www.yogihosting.com/insert-records-entity-framework-core/#download)

EntityState

EntityState is an enumeration that stores the state of the entity. It can have one out of the 5 different values, these are ‘Added’, ‘Deleted’, ‘Detached’, ‘Modified’ & ‘Unchanged’. When we want to create a new record in the database then the EntityState of the corresponding entity should be ‘Added’. This tells EF Core that it has to insert the given record. Similarly if we want to update an entity then it must be ‘Modified’, for deleting an entity it should be ‘Deleted’.

Unchanged entity state means that there isn’t any change done for a given entity. Also note that Entity Framework Core keeps track of all the entities for changes, a value of Detached tells that the given entity is not being tracked.

Ezoic

This tutorial is a part of **Entity Framework Core** series.

* 1. [Introduction to Entity Framework Core](https://www.yogihosting.com/introduction-entity-framework-core/)
* 2. [Installation of Entity Framework Core](https://www.yogihosting.com/install-entity-framework-core/)
* 3. [Database-First approach in Entity Framework Core](https://www.yogihosting.com/database-first-approach-entity-framework-core/)
* 4. [DbContext Class in Entity Framework Core](https://www.yogihosting.com/dbcontext-entity-framework-core/" \t "_blank)
* 5. [Code-First Approach in Entity Framework Core](https://www.yogihosting.com/code-first-entity-framework-core/)
* 6. [Migrations in Entity Framework Core](https://www.yogihosting.com/migrations-entity-framework-core/)
* 7. *Insert Records in Entity Framework Core*
* 8. [Read Records in Entity Framework Core](https://www.yogihosting.com/read-records-entity-framework-core/)
* 9. [Update Records in Entity Framework Core](https://www.yogihosting.com/update-records-entity-framework-core/)
* 10. [Delete Records in Entity Framework Core](https://www.yogihosting.com/delete-records-entity-framework-core/)
* 11. [Conventions in Entity Framework Core](https://www.yogihosting.com/conventions-entity-framework-core/)
* 12. [Configurations in Entity Framework Core](https://www.yogihosting.com/configurations-entity-framework-core/)
* 13. [Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-entity-framework-core/)
* 14. [Configure One-to-Many relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-one-to-many-relationship-entity-framework-core/)
* 15. [Configure One-to-One relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-one-to-one-relationship-entity-framework-core/)
* 16. [Configure Many-to-Many relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-many-to-many-relationship-entity-framework-core/)
* 17. [Execute Raw SQL Queries using FromSqlRaw() method in Entity Framework Core](https://www.yogihosting.com/raw-sql-queries-entity-framework-core/)
* 18. [Execute SQL Stored Procedures using FromSqlRaw() & ExecuteSqlRawAsync() methods in Entity Framework Core](https://www.yogihosting.com/stored-procedures-entity-framework-core/)

Entity Framework Core Create Records

We will now create records with entity framework core. There are 2 entities – Department & Employee. The entities are created in Department.cs and Employee.cs files inside the “Models” folder of the app.

A Department entity can have more than one Employee so there is **many-to-one relationship** between them.

The codes of these entity classes are given below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **public** **class** Department  {  **public** **int** Id { **get**; **set**; }  **public** **string** Name { **get**; **set**; }    **public** ICollection<Employee> Employee { **get**; **set**; }  }    **public** **class** Employee  {  **public** **int** Id { **get**; **set**; }  **public** **int** DepartmentId { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** **string** Designation { **get**; **set**; }    **public** Department Department { **get**; **set**; }  } |

**Navigation Property** is a property defined on the principal or dependent entity. This property references the related entity.

Notice the “Employee” property on the “Department” class – public ICollection Employee { get; set; }, which is a Navigation Property as it references the related Employee entity. To be more specific it is a **Collection Navigation Property** as it contains references to many related Employee entities.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ce7a1c67-aeab-450b-66f1-e8cac85b1053&d_id=122531&imp_id=6828221831006089&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

Similarly the property “Department” on the employee class which is our Navigation Property – public Department Department { get; set; }. This Navigation Property references the related Department entity.

The DbContext for the Entity Framework Core is created inside the “Models” folder of the app and is named as CompanyContext.cs. It defines the 2 entities which are Department and Employee.

public class CompanyContext : DbContext

{

public CompanyContext(DbContextOptions<CompanyContext> options) : base(options)

{

}

public DbSet<Department> Department { get; set; }

public DbSet<Employee> Employee { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

}

}

We registered the DbContext as a service in the program class with the below code.

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

options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

The above code provides DbContext with the connection string (located inside the “appsettings.json” file). It also registers the DbContext as a service. With this we can use the DbContext in our controllers through dependency injection technique. The connection string code is given below which will be different for your case.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ce7a1c67-aeab-450b-66f1-e8cac85b1053&d_id=122531&imp_id=3176501865028536&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

{

"ConnectionStrings": {

"DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB;Initial Catalog=Company;Integrated Security=True;Connect Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetFailover=False"

}

}

We explained how to create database context and register it as a service so that it can be used through dependency injection in the article [DbContext Class in Entity Framework Core](https://www.yogihosting.com/dbcontext-entity-framework-core/), so do check it.

Insert/Create a Single Record on the Database

First let me show how to Insert a single record on the Department table. The below code does this work.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **var** dept = **new** Department()  {      Name = "Designing"  };  context.Entry(dept).State = EntityState.Added;  context.SaveChanges(); |

We have created a new Designing object of the Department class and set its EntityState to Added. When the DbContext.SaveChanges() method is called this new record is inserted on the Department table by Entity Framework Core.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ce7a1c67-aeab-450b-66f1-e8cac85b1053&d_id=122531&imp_id=7498895657082033&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

EzoicThe variable “context” is the object of database context class which is provided to the controller by dependency injection technique. See the below code where we have shown this.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | **public** **class** DepartmentController : Controller  {  **private** CompanyContext context;  **public** DepartmentController(CompanyContext cc)      {          context = cc;      }        //... action methods creating record  **public** IActionResult Create()      {  **var** dept = **new** Department()          {              Name = "Designing"          };          context.Entry(dept).State = EntityState.Added;          context.SaveChanges();      }  } |

Check the below image where we have shown the newly created record on the database.

There is also a shorter way to insert a record. The below code will do the same thing.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** dept = **new** Department()  {      Name = "Designing"  };    context.Add(dept);  context.SaveChanges(); |

Note that we don’t have to explicitly set the EntityState to Added, as this is done by EF Core by it’s own.

Asynchronous code with “SaveChangesAsync()” method

The SaveChanges() method which we used earlier is a synchronous method. We can instead use the SaveChangesAsync() method which is an **asynchronous method**. Asynchronous methods make use of threading and enables server resources to be used more efficiently. So our codes handles more traffic without delays.

The below code does the record creation in the database in **Asynchronous** way.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** dept = **new** Department()  {      Name = "Designing"  };    context.Add(dept);  **await** context.SaveChangesAsync(); |

Web API Tutorial : [How to call Web API from JavaScript](https://www.yogihosting.com/aspnet-core-web-api-javascript/)

TryUpdateModelAsync

The best manner to create records and at the same time prevent overposting by malicious users is by the use of **TryUpdateModelAsync** method. Here, from the model, we can extract only those field values that we need to insert to our database. The remaining fields that are not needed are left out. A malicious user adds values to these fields through tools like fiddler or through JavaScript. But as we are filtering them therefore there is no chance they are making way to our database.

The below code is adding a new employee record by TryUpdateModelAsync method. We created an empty employee object first, and then added the Name, DepartmentId and Designation field values to it through lambda expression. These values are provided through Model Binding. This means the code is extracting only these 3 values from the Model and inserting it to the database. Other fields are totally ignored.

var emptyEmployee = new Employee();

if (await TryUpdateModelAsync<Employee>(emptyEmployee, "", s => s.Name, s => s.DepartmentId, s => s.Designation))

{

context.Employee.Add(emptyEmployee);

await context.SaveChangesAsync();

return RedirectToAction("Index");

}

Entity Framework Core Bulk Insert

The DbContext.AddRange() method is used to **bulk insert multiple records on the database** at the same time.

Here we are bulk inserting 3 department records.

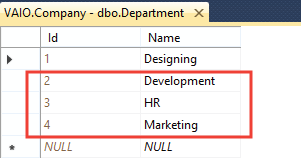
var dept1 = new Department() { Name = "Development" };

var dept2 = new Department() { Name = "HR" };

var dept3 = new Department() { Name = "Marketing" };

context.AddRange(dept1, dept2, dept3);

await context.SaveChangesAsync();



We can do the same thing from the below code:

var dept1 = new Department() { Name = "Development" };

var dept2 = new Department() { Name = "HR" };

var dept3 = new Department() { Name = "Marketing" };

var deps = new List<Department>() { dept1, dept2, dept3 };

context.AddRange(deps);

await context.SaveChangesAsync();

Entity Framework Core Insert Related Records

The Department & Employee tables have **many-to-one relationship** between them. Here we will insert one new record to each of these 2 tables (Department & Employee).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | **var** dept = **new** Department()  {      Name = "Admin"  };    **var** emp = **new** Employee()  {      Name = "Matt",      Designation = "Head",      Department = dept  };    context.Add(emp);  **await** context.SaveChangesAsync(); |

We have set the ‘Department’ property of the Employee class to the Department object. So in this way Entity Framework Core will know that the entities are related hence it will insert both of them to their respected database tables.

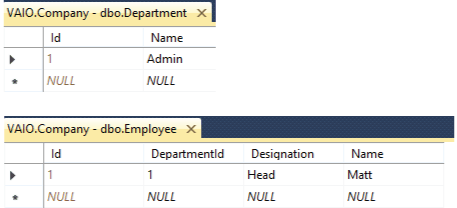
**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ce7a1c67-aeab-450b-66f1-e8cac85b1053&d_id=122531&imp_id=904611449006221&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=ce7a1c67-aeab-450b-66f1-e8cac85b1053&d_id=122531&imp_id=2282340309059395&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

Check the database to find both the new records on their respected tables.



Entity Framework Core CRUD Operations – CREATE RECORDS

We will now make **Entity Framework Core CRUD OPERATIONS** feature. Let us first perform the “Create Records” feature for the Department entity. So create a new controller file called DepartmentController.cs inside the “Controllers” folder of the app. Here add “Create” action methods which will take values for a new Department, from a form which is submitted from the view. The code of the controller is given below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | **using** EFCoreExample.Models;  **using** Microsoft.AspNetCore.Mvc;    **namespace** EFCoreExample.Controllers  {  **public** **class** DepartmentController : Controller      {  **private** CompanyContext context;  **public** DepartmentController(CompanyContext cc)          {              context = cc;          }    **public** IActionResult Create()          {  **return** View();          }            [HttpPost]  **public** **async** Task<IActionResult> Create(Department dept)          {              context.Add(dept);  **await** context.SaveChangesAsync();    **return** View();          }      }  } |

The controller gets the DbContext object from dependency injection and it uses it to add a new record to the database.

Next we need to create a small form where user can enter the department name whose records entity framework core will add to the database. So add a new razor view file called Create.cshtml inside the “Views/Department” folder. It’s code is given below.

@{

ViewData["Title"] = "Create Department";

}

@model Department

<h1 class="bg-info text-white">Create Department</h1>

<form method="post">

<div class="form-group">

<label asp-for="Name"></label>

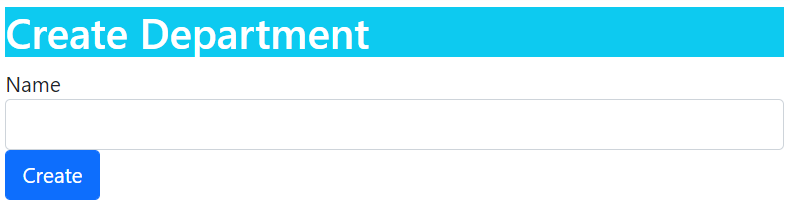
<input asp-for="Name" class="form-control" />

</div>

<button type="submit" class="btn btn-primary">Create</button>

</form>

Run the app and go to the url – https://localhost:7018/Department/Create. We will see the department form (screenshot given below). Fill in the name of the department and click the button to create the record.



Next we need to Create Employee Records. So we add a new controller file called EmployeeController.cs and add “Create” action methods which will create a new employee record. The code is given below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | **using** EFCoreExample.Models;  **using** Microsoft.AspNetCore.Mvc;  **using** Microsoft.AspNetCore.Mvc.Rendering;  **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Controllers  {  **public** **class** EmployeeController : Controller      {  **private** CompanyContext context;  **public** EmployeeController(CompanyContext cc)          {              context = cc;          }    **public** IActionResult Index()          {  **return** View();          }    **public** IActionResult Create()          {              List<SelectListItem> dept = **new** List<SelectListItem>();              dept = context.Department.Select(x => **new** SelectListItem { Text = x.Name, Value = x.Id.ToString() }).ToList();              ViewBag.Department = dept;    **return** View();          }            [HttpPost]  **public** **async** Task<IActionResult> Create(Employee emp)          {              context.Add(emp);  **await** context.SaveChangesAsync();  **return** RedirectToAction("Index");          }      }  } |

There is one-to-many relationship between employee and department. So we also have to insert a department for every empoyee. The departments are fetched from department entity, then they are converted to a List<SelectListItem> object and added to a Viewbag variable.

List<SelectListItem> dept = new List<SelectListItem>();

dept = context.Department.Select(x => new SelectListItem { Text = x.Name, Value = x.Id.ToString() }).ToList();

ViewBag.Department = dept;

On the view we will read the Viewbag value and show the departments in an HTML SELECT control.

We can also use the TryUpdateModelAsync method on the create action method. Remove the post version of Create action and replace with with the below given one.

[HttpPost]

[ActionName("Create")]

public async Task<IActionResult> Create\_Post()

{

var emptyEmployee = new Employee();

if (await TryUpdateModelAsync<Employee>(emptyEmployee, "", s => s.Name, s => s.DepartmentId, s => s.Designation))

{

context.Employee.Add(emptyEmployee);

await context.SaveChangesAsync();

return RedirectToAction("Index");

}

return View();

}

In the above code we are creating an empty employee object and then adding 3 value – Name, DepartmentId and Designation to it with lambda expression.

s => s.Name, s => s.DepartmentId, s => s.Designation

These employee values are provided by Model Binding when the form given on the razor view file is submitted. After that we are checking if the TryUpdateModelAsync returns true which is the case when the empty employee object is successfully provided with values through the lambda expression. So in that case we are adding the employee record to the database context and then inserting the record to the database by using Entity Framework Core SaveChangesAsync method.

context.SaveChangesAsync();

Next, add a razor view file called Create.cshtml inside the “Views/Employee” folder with the following code.

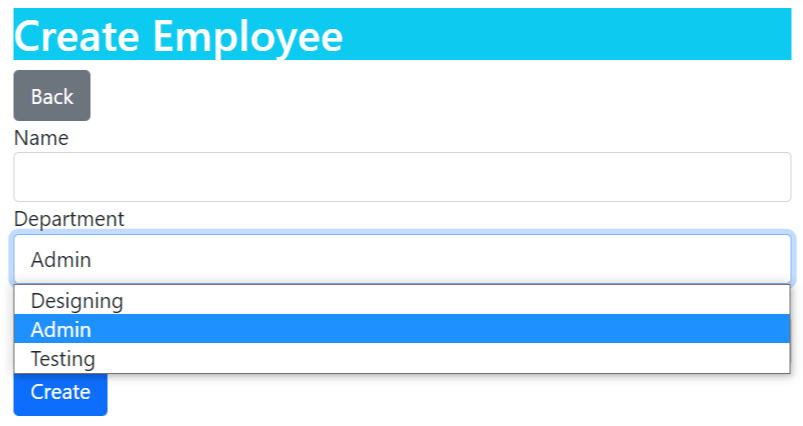
Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @{      ViewData["Title"] = "Create Employee";  }    @model Employee    <h1 **class**="bg-info text-white">Create Employee</h1>  <a asp-action="Index" **class**="btn btn-secondary">Back</a>    <form method="post">      <div **class**="form-group">          <label asp-**for**="Name"></label>          <input asp-**for**="Name" **class**="form-control" />      </div>      <div **class**="form-group">          <label asp-**for**="Department"></label>          <**select** asp-**for**="DepartmentId" asp-items="ViewBag.Department" **class**="form-control"></**select**>      </div>      <div **class**="form-group">          <label asp-**for**="Designation"></label>          <input asp-**for**="Designation" **class**="form-control" />      </div>      <button type="submit" **class**="btn btn-primary">Create</button>  </form> |

Notice how all the departments are shown in the select control with the following razor code.

<select asp-for="DepartmentId" asp-items="ViewBag.Department" class="form-control"></select>

We can now run the app and go to the url – https://localhost:7018/Employee/Create.. We will see the employee form with departments shown in a select control. Check the below image. Fill and submit the form to create a new record in the database.



Download the source codes:

[DOWNLOAD](https://www.yogihosting.com/wp-content/themes/yogi-yogihosting/download/efcore/EFCoreExample.zip)

Conclusion

We learned how to use Entity Framework Core to create records in the database. Records can be inserted one at a time or in batches. We also understood how to insert related records and created a CRUD OPERATIONS feature at the end. In the next tutorial we will cover the Reading part and add the read records feature to the CRUD OPERATIONS so continue this at [Read Records in Entity Framework Core](https://www.yogihosting.com/read-records-entity-framework-core/).

**Read Records in Entity Framework Core**

Last Updated: October 10, 2022

Entity Framework Core Reads Record from the database through the DbContext object. For example we can get all records from the database by using the below code.

var emp = context.Employee;

Here “context” is the object of DbContext class and “employee” is the entity whose reacords Entity Framework Core is reading from the database.

We can also fetch a particular employee from database. For example in the below code we are fetching the employee with name as Matt.

var emp = await context.Employee.Where(e => e.Name == "Matt").FirstOrDefaultAsync();

This tutorial is a part of **Entity Framework Core** series.

* 1. [Introduction to Entity Framework Core](https://www.yogihosting.com/introduction-entity-framework-core/)
* 2. [Installation of Entity Framework Core](https://www.yogihosting.com/install-entity-framework-core/)
* 3. [Database-First approach in Entity Framework Core](https://www.yogihosting.com/database-first-approach-entity-framework-core/)
* 4. [DbContext Class in Entity Framework Core](https://www.yogihosting.com/dbcontext-entity-framework-core/" \t "_blank)
* 5. [Code-First Approach in Entity Framework Core](https://www.yogihosting.com/code-first-entity-framework-core/)
* 6. [Migrations in Entity Framework Core](https://www.yogihosting.com/migrations-entity-framework-core/)
* 7. [Insert Records in Entity Framework Core](https://www.yogihosting.com/insert-records-entity-framework-core/)
* 8. *Read Records in Entity Framework Core*
* 9. [Update Records in Entity Framework Core](https://www.yogihosting.com/update-records-entity-framework-core/)
* 10. [Delete Records in Entity Framework Core](https://www.yogihosting.com/delete-records-entity-framework-core/)
* 11. [Conventions in Entity Framework Core](https://www.yogihosting.com/conventions-entity-framework-core/)
* 12. [Configurations in Entity Framework Core](https://www.yogihosting.com/configurations-entity-framework-core/)
* 13. [Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-entity-framework-core/)
* 14. [Configure One-to-Many relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-one-to-many-relationship-entity-framework-core/)
* 15. [Configure One-to-One relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-one-to-one-relationship-entity-framework-core/)
* 16. [Configure Many-to-Many relationship using Fluent API in Entity Framework Core](https://www.yogihosting.com/fluent-api-many-to-many-relationship-entity-framework-core/)
* 17. [Execute Raw SQL Queries using FromSqlRaw() method in Entity Framework Core](https://www.yogihosting.com/raw-sql-queries-entity-framework-core/)
* 18. [Execute SQL Stored Procedures using FromSqlRaw() & ExecuteSqlRawAsync() methods in Entity Framework Core](https://www.yogihosting.com/stored-procedures-entity-framework-core/)

**Page Contents**

* [Entity Framework Core Read Related Records](https://www.yogihosting.com/read-records-entity-framework-core/#read)
* [Eager Loading in EF Core](https://www.yogihosting.com/read-records-entity-framework-core/#eager)
* [Explicit Loading in EF Core](https://www.yogihosting.com/read-records-entity-framework-core/#explicit)
* [Lazy Loading in EF Core](https://www.yogihosting.com/read-records-entity-framework-core/#lazy)
* [Optimizing Entity Framework Core Codes](https://www.yogihosting.com/read-records-entity-framework-core/#optimize)
* [Entity Framework Core CRUD Operations – READ RECORDS](https://www.yogihosting.com/read-records-entity-framework-core/#crud)
* [Download Source Codes](https://www.yogihosting.com/read-records-entity-framework-core/#download)

Entity Framework Core Read Related Records

There are 3 common ORM Patterns used by Entity Framework Core to **Read Related Records**. These are:

* **Eager Loading**: In Eager Loading the related data is also retrieved at the same time when an Entity is read.
* **Explicit Loading**: In Explicit Loading the related data is not loaded at the same time when an Entity is read. It is explicitly loaded from the database at a later time.
* **Lazy Loading**: When the entity is first read, related data isn’t retrieved. However, the first time we attempt to access a navigation property, then the data required for that navigation property is automatically retrieved.

Here we will be taking the same Company Database which contains 2 tables – Employee & Department. There is Many-to-One Relationship between these 2 tables i.e. a department a can have one or more employees. So make sure you covered the previous tutorial on [EF Core – Insert Records](https://www.yogihosting.com/read-records-entity-framework-core/) beforehand.

Eager Loading in EF Core

On performing a normal reading of records in Entity Framework Core, the **Related Records are not Loaded**. But with Eager Loading we can use the Navigation property of an entity to load related records. Consider the below Employee entity which has a navigation property called “Deparment” which is pointing to another entity called “Department”.

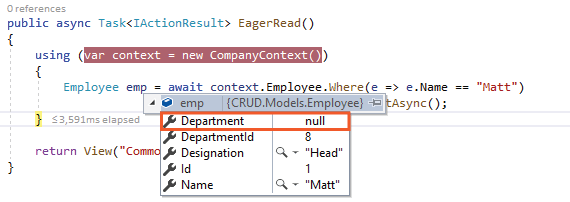
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **public** **class** Employee  {  **public** **int** Id { **get**; **set**; }  **public** **int** DepartmentId { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** **string** Designation { **get**; **set**; }    **public** Department Department { **get**; **set**; }  } |

The below code is used to read an Employee with name as “Matt”.

Employee emp = await context.Employee.Where(e => e.Name == "Matt").FirstOrDefaultAsync();

Apply the breakpoint on the above code and check the value of **Navigation Property** called Department. We will find it’s value as **“null”**. See the below image where we have marked this thing.

Ezoic



In **Eager Loading** we use the Include() method to load related enteries. The Related entity is loaded through the navigation property. The Employee entity contains a Navigation Property called Department for this purpose:

public Department Department { get; set; }

Similarly, the Department entity contains a Collection Navigation Property called Employee for this purpose.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **public** **class** Department  {  **public** **int** Id { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** ICollection<Employee> Employee { **get**; **set**; }  } |

The Employee entity has a related Department entity so the Include() method can be used to perform the **Eager Loading** of Department record like shown below:

|  |  |
| --- | --- |
| 1  2  3 | Employee emp = **await** context.Employee.Where(e => e.Name == "Matt")                              .Include(s => s.Department)                              .FirstOrDefaultAsync(); |

Check the above code by putting a breakpoint over it in Visual Studio and note the value of emp variable. We will find the value of Department property is filled with value this time. We have shown this on the below image:

Ezoic

It should be noted that Entity Framework Core executes SQL Queries behind the scene to read, create, delete and update data. The Include() method will execute a single SQL Join Query on the database to fetch the data. This SQL query is given below:

SELECT [e].[Id], [e].[Designation], [e].[Name], [e.Department].[Id], [e.Department].[Name]

FROM [Employee] AS [e]

LEFT JOIN [Department] AS [e.Department] ON [e].[DepartmentId] = [e.Department].[Id]

WHERE [e].[Name] = N'Matt'

Multiple “Include()” methods

We can use multiple Include() methods to load multiple levels of related entities with Entity Framework Core. For example, suppose the Employee entity also has another related entity called Project. Then the following Include code loads the Department & Project entities of the Employee.

|  |  |
| --- | --- |
| 1  2  3  4 | **var** emp = **await** context.Employee.Where(e => e.Name == "Matt")                         .Include(s=>s.Department)                         .Include(s=>s.Project)                         .FirstOrDefaultAsync(); |

*I have also written a similar article on ADO.NET see*[*Read Records using ADO.NET in ASP.NET Core Application*](https://www.yogihosting.com/read-records-ado-net-aspnet-core/)

“ThenInclude()” Method

EzoicEntity Framework core has method named ThenInclude() is used to **load multiple levels of related data**. For example suppose there is a Navigation Property named Report in the Department entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **public** **class** Department  {  **public** **int** Id { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** ICollection<Employee> Employee { **get**; **set**; }  **public** Report Report { **get**; **set**; }  } |

Then see the below code:

|  |  |
| --- | --- |
| 1  2  3  4 | **var** emp = **await** context.Employee.Where(e => e.Name == "Matt")                         .Include(s => s.Department)                         .ThenInclude(r => r.Report)                         .FirstOrDefaultAsync(); |

Here .Include(s => s.Department) will load the related entity called Department of the Employee entity. Next .ThenInclude(r => r.Report) will load the related entity called Report of the Department entity.

Explicit Loading in EF Core

In **Entity Framework Core Explicit Loading** the related data is explicitly loaded from the database at a later time. We write codes that retrieve the related data if it is needed. So as a result multiple queries are sent to the database thereforem making explicit Loading a heavy task in certain situations.

Ezoic

Here the Load() or it’s asynchronous method called LoadAsync() is used to load related entity explicitly.

Consider the below code:

|  |  |
| --- | --- |
| 1  2  3 | **var** emp = **await** context.Employee.Where(e => e.Name == "Matt")                         .FirstOrDefaultAsync();  **await** context.Entry(emp).Reference(s => s.Department).LoadAsync(); |

The code – await context.Entry(emp).Reference(s => s.Department).LoadAsync() loads the related entity called Department of the Employee entity. The Reference property gets the reference to the related data and the LoadAsync() method loads it explicitly. We have shown this in the below image.

We can also filter the related data before loading them. For this use the Query() method as shown below.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=275fa0f8-4fe7-4de4-7491-3efee7dbe954&d_id=122531&imp_id=696155455012941&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

await context.Entry(emp).Reference(s => s.Department).Query().Where(s => s.Name == "Admin").LoadAsync();

The below code will only load the Department having name as ‘Admin’.

Lazy Loading in EF Core

In **Entity Framework Core Lazy Loading** technique the related data isn’t retrieved when the entity is first read. However, when the first time we access a navigation property, the data required for that navigation property is automatically retrieved.

In order to use **Lazy Loading** we must do 2 things:

Ezoic

* 1. Install the Microsoft.EntityFrameworkCore.Proxies package and enable it with a call to UseLazyLoadingProxies method in the OnConfiguring method of Database Context file.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **protected** **override** **void** OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  **if** (!optionsBuilder.IsConfigured)      {          optionsBuilder.UseLazyLoadingProxies();      }  } |

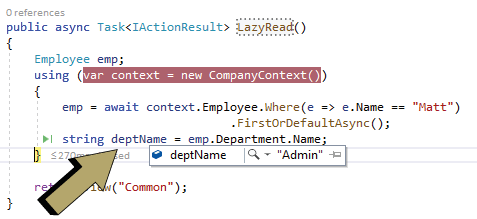
* 2. Make all the **Navigation Properties** as virtual.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | **public** **class** Employee  {  **public** **int** Id { **get**; **set**; }  **public** **int** DepartmentId { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** **string** Designation { **get**; **set**; }    **public** **virtual** Department Department { **get**; **set**; }  }    **public** **class** Department  {  **public** **int** Id { **get**; **set**; }  **public** **string** Name { **get**; **set**; }    **public** **virtual** ICollection<Employee> Employee { **get**; **set**; }  } |

Lazy Loading example

|  |  |
| --- | --- |
| 1  2  3 | Employee emp = **await** context.Employee.Where(e => e.Name == "Matt")                              .FirstOrDefaultAsync();  **string** deptName = emp.Department.Name; |

In the above code the Department which is related to Employee entity is **lazy loaded**. On putting a breakpoint over deptName variable we can see it’s value. Check the below given image.



Optimizing Entity Framework Core Codes

It is necessary that we Optimize our Entity Framework Core code so that the application codes remain light and at the same time execute faster. We can perform the optimization of Entity Framework codes in 3 manners which are:

1. No Tracking of Entities
2. Minimum call to Database
3. Limit the size of resultset

No Tracking of Entities

Entity Framework Core keeps track of all the entities that are returned from a LINQ query. This will cause unnecessary burden when we don’t require tracking particularly in read-only scenarios. The **AsNoTracking** method tells EF Core not to track the entity. We can use it in our code like shown below.

var emp = context.Employee.AsNoTracking();

Minimum call to Database

Whenever we access an entity through database context then Entity Framework Core calls the database to fetch the result set. We can make use of List type to store the result set and then extact the data from it, instead of making calls to the database again and again. See the below code where EF Core will be making database call 2 times.

var empall = context.Employee;

var empmatt = context.Employee.Where(e => e.Name == "Matt").FirstOrDefault();

We can reduce the calling to the database to just a single time by storing the result in a list type object and then subsiquently fetching a record from there. See the below code where we have done this thing.

var empall = context.Employee.ToList();

var empmatt = empall.Where(e => e.Name == "Matt").FirstOrDefault();

Limit the size of resultset

When we call an entity we are provided with all it’s fields. We should only pull back those fields that we need so that the unnecessary heaviness of the result set is reduced. For example in the below query we are only pulling the Name field for the entity.

var empmall = context.Employee.Select(b => b.Name);

Similary we are only needing the name and deignation of an employee which is done through the below linq code.

var empmatt = context.Employee.Where(e => e.Name == "Matt").Select(b => new {b.Name, b.Designation}).FirstOrDefault();

During reading of records, rather than fetching all at once, we should fetch them based on paga by page manner. This should be done when we implement pagination feature. LINQ Skip and Take operators are used to implement this thing. Skip tells to bipass a given number of records from the start and Take tells to fetch a given number of records from there.

See the below code which are providing us the records for the page numbers 1, 2 and 3. Page size being set to 10 records per page.

var emp\_page\_One = context.Employee.Skip(0).Take(10); // gets page 1 records

var emp\_page\_Two = context.Employee.Skip(20).Take(10); // gets page 2 records

var emp\_page\_Three = context.Employee.Skip(30).Take(10); // gets page 3 records

You can certainly read more about this concept of pagination in our article [Create Number Paging with Custom Tag Helper in ASP.NET Core](https://www.yogihosting.com/aspnet-core-paging/).

Entity Framework Core CRUD Operations – READ RECORDS

We will now perform **Entity Framework Core CRUD OPERATIONS for Reading Records from the Database**. We do this for both the Employee and Department entities.

Just a recall, we started this CRUD OPERATIONS in our article called [Insert Records in Entity Framework Core](https://www.yogihosting.com/insert-records-entity-framework-core/), make sure to read it first.

Open the DepartmentController.cs and add an “Index” action method that reads all the department records and returns them to the view with the code – context.Department.AsNoTracking().

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=275fa0f8-4fe7-4de4-7491-3efee7dbe954&d_id=122531&imp_id=2612013873078423&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **public** **class** DepartmentController : Controller  {  **private** CompanyContext context;  **public** DepartmentController(CompanyContext cc)      {          context = cc;      }    **public** IActionResult Index()      {  **return** View(context.Department.AsNoTracking());      }        //...  } |

Next, add the Index.cshtml razor view file inside the “Views/Department” folder which will shown the department records in a HTML Table.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | @{      ViewData["Title"] = "All Departments";  }  @model IEnumerable<Department>    <h1 **class**="bg-info text-white">All Departments</h1>  <a asp-action="Create" **class**="btn btn-secondary">Create</a>    <table **class**="table table-sm table-bordered">      <tr>          <th>ID</th>          <th>Name</th>      </tr>      @**foreach** (Department dept **in** Model)      {          <tr>              <td>@dept.Id</td>              <td>@dept.Name</td>          </tr>      }  </table> |

Run the app and open the url – https://localhost:7018/Department where we can see all the department records displayed nicely inside a table.

Next we will read Employee records and display them on the browser. So add a new Index action method to the “EmployeeController.cs” file as shown below.

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | **public** **class** EmployeeController : Controller  {  **private** CompanyContext context;  **public** EmployeeController(CompanyContext cc)      {          context = cc;      }    **public** IActionResult Index()      {  **return** View(context.Employee.Include(s => s.Department));      }        //...  } |

We used the Eager Loading concept to read Employee and it’s related Department recods by using the code.

context.Employee.Include(s => s.Department)

The records are returned to the browser where they are displayed to the user. Next add Index.cshtml razor view file inside “Views/Employee” folder with the following code.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | @{      ViewData["Title"] = "All Employees";  }  @model IEnumerable<Employee>    <h1 **class**="bg-info text-white">All Employees</h1>  <a asp-action="Create" **class**="btn btn-secondary">Create</a>    <table **class**="table table-sm table-bordered">      <tr>          <th>ID</th>          <th>Name</th>          <th>Designation</th>          <th>Department</th>      </tr>      @**foreach** (Employee emp **in** Model)      {          <tr>              <td>@emp.Id</td>              <td>@emp.Name</td>              <td>@emp.Designation</td>              <td>@emp.Department.Name</td>          </tr>      }  </table> |

The view has a model of type IEnumerable<Employee> and we are reading the related department record’s name as – @emp.Department.Name. Now run the app and go to the url – https://localhost:7018/Employee where we will see all the Employee records as shown below.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=275fa0f8-4fe7-4de4-7491-3efee7dbe954&d_id=122531&imp_id=7843753109000206&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=275fa0f8-4fe7-4de4-7491-3efee7dbe954&d_id=122531&imp_id=7843753109000206&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

Download the source codes:

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=275fa0f8-4fe7-4de4-7491-3efee7dbe954&d_id=122531&imp_id=4251363911018214&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

[DOWNLOAD](https://www.yogihosting.com/wp-content/themes/yogi-yogihosting/download/efcore/EFCoreExample.zip)

Conclusion

We covered Entity Framework Core Read Records concept and also learned how we can include child collection when reading related records. After that we also discussed optimization techniques that wil help us to speed our apps. In the end we create the Reading part of the CRUD OPERATIONS. We are now read to move to the next part which is [Update Records in Entity Framework Core](https://www.yogihosting.com/update-records-entity-framework-core/).

**Update Records in Entity Framework Core**

Last Updated: October 6, 2023

The **Entity Framework Core** executes UPDATE statement in the database for the entities whose EntityState is Modified. The DbContext’s Update() update method is used for performing the updation of records in the database.

This tutorial is a part of **Entity Framework Core** series.

**Page Contents**

* [Update Single Record](https://www.yogihosting.com/update-records-entity-framework-core/#single)
* [Update Multiple Records](https://www.yogihosting.com/update-records-entity-framework-core/#multiple)
* [Update Related Records](https://www.yogihosting.com/update-records-entity-framework-core/#related)
* [Entity Framework Core CRUD Operations – UPDATE RECORDS](https://www.yogihosting.com/update-records-entity-framework-core/#crud)
* [Download Source Codes](https://www.yogihosting.com/update-records-entity-framework-core/#download)

Update Single Record

Consider the below code where we are updating the Department name of the department with Id number 1. We are updating it’s name to Designing.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** dept = **new** Department()  {      Id = 1,      Name = "Designing"  };  context.Update(dept);  **await** context.SaveChangesAsync(); |

The EF Core is able to perform the update here is because we have given Id key value (which is ‘1’ in this case) for the department entity through which tracking of the entity is done.

Ezoic

Related to this tutorial: [Update Records using ADO.NET in ASP.NET Core Application](https://www.yogihosting.com/update-records-ado-net-aspnet-core/)

Case when the .Update() will Insert a new record

If there is no valid key value then EF core will will consider it a new entity and perform the Insert operation on the database.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** dept = **new** Department()  {      Name = "Research"  };    context.Update(dept);  **await** context.SaveChangesAsync(); |

Here the .Update() method will insert a new Research record on the Department table.

Update Multiple Records

The UpdateRange() method is used to **Update Multiple Records** at the same time. Consider the below code where we are updating 3 Department records at the same time.

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | **var** dept1 = **new** Department()  {      Id = 1,      Name = "New Designing"  };    **var** dept2 = **new** Department()  {      Id = 2,      Name = "New Research"  };    **var** dept3 = **new** Department()  {      Id = 3,      Name = "New HR"  };    List<Department> modifiedDept = **new** List<Department>() { dept1, dept2, dept3 };    context.UpdateRange(modifiedDept);  **await** context.SaveChangesAsync(); |

*I have covered very advanced model binding in my article called*[*Advanced Model Binding Concepts in ASP.NET Core*](https://www.yogihosting.com/aspnet-core-advanced-model-binding/)

Ezoic

Update Related Records

Entity Framework Core can also be used to Update Related Records. See the below code which updates the Employee and Department at the same time. The Department name is updated to ‘Admin\_1’. This Department is the related record for the Employee Entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **var** dept = **new** Department()  {      Id = 5,      Name = "Admin\_1"  };    **var** emp = **new** Employee()  {      Id = 1,      Name = "Matt\_1",      Designation = "Head\_1",      Department = dept  };    context.Update(emp);  **await** context.SaveChangesAsync(); |

When this code executes the employee id 1 gets his info changed to:

* 1. Name is changed to “Matt\_1”.
* 2. Designation is changed to “Head\_1”.

The related department also gets changed. This Employee Department name gets changed to “Admin\_1”.

Entity Framework Core CRUD Operations – UPDATE RECORDS

We will now perform the Update Records task in the CRUD OPERATIONS feature which we are building in the app. So first add the Update action methods on the DepartmentController.cs file. The code is shown below.

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | **using** EFCoreExample.Models;  **using** Microsoft.AspNetCore.Mvc;  **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Controllers  {  **public** **class** DepartmentController : Controller      {  **private** CompanyContext context;  **public** DepartmentController(CompanyContext cc)          {              context = cc;          }    **public** **async** Task<IActionResult> Update(**int** id)          {              Department dept = **await** context.Department.Where(e => e.Id == id).FirstOrDefaultAsync();  **return** View(dept);          }            [HttpPost]  **public** **async** Task<IActionResult> Update(Department dept)          {              context.Update(dept);  **await** context.SaveChangesAsync();    **return** RedirectToAction("Index");          }            //...      }  } |

The 2 actions are added which are the HTTP GET and HTTP POST versions. In the GET Version we are getting the id of the department on the method’s parameter and then we are fetching the corresponding department record from the database. This record is returned to the update view where it will be shown to the user in a form.

The user will update the values for the department record and submit the form. The POST Version of the Update action will then be called and it receives the updated value of the department in it’s parameter through Model Binding technique. It then calls the Update() method and saves the record to the database by using the DbContext SaveChangesAsync() method.

Next add a new razor view file called Update.cshtml inside the Views/Department folder with the following code.

@{

ViewData["Title"] = "Update Department";

}

@model Department

<h1 class="bg-info text-white">Update Department</h1>

<a asp-action="Index" class="btn btn-secondary">Back</a>

<form method="post">

<div class="form-group">

<label asp-for="Name"></label>

<input asp-for="Name" class="form-control" />

</div>

<button type="submit" class="btn btn-primary">Update</button>

</form>

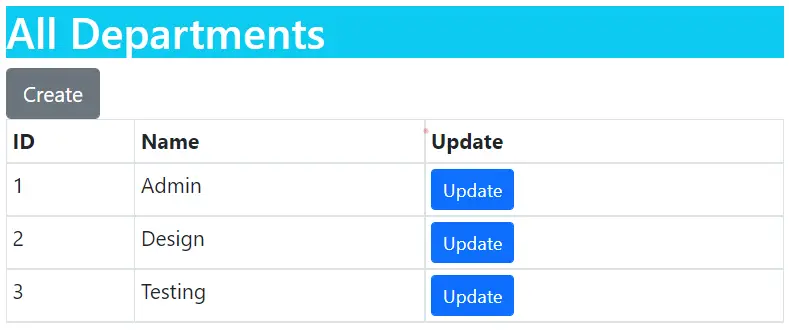
EzoicThe below image shows the update department form.

We will also need to link the update feature from the Index view file. Recall that the Index.cshtml view shows all the department records in a table. We built this feature on the previous tutorial – [Read Records in Entity Framework Core](https://www.yogihosting.com/read-records-entity-framework-core/). So we update the Index view by adding an update column with a anchor link that will call the Update action on the Department controller by passing the department id on it’s route.

The added column is shown in highlighted color below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | @{      ViewData["Title"] = "All Departments";  }  @model IEnumerable<Department>    <h1 **class**="bg-info text-white">All Departments</h1>  <a asp-action="Create" **class**="btn btn-secondary">Create</a>    <table **class**="table table-sm table-bordered">      <tr>          <th>ID</th>          <th>Name</th>          <th>Update</th>      </tr>      @**foreach** (Department dept **in** Model)      {          <tr>              <td>@dept.Id</td>              <td>@dept.Name</td>              <td>                  <a **class**="btn btn-sm btn-primary" asp-action="Update" asp-route-id="@dept.Id">                      Update                  </a>              </td>          </tr>      }  </table> |

Run the app and visit the department controller link – https://localhost:7018/Department where the departments records are shown along with the update button. Click on an update button to update the corresponding department record. Check the below image.



Lets do the same with the EmployeeController.cs by adding Update action methods to it.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | **using** EFCoreExample.Models;  **using** Microsoft.AspNetCore.Mvc;  **using** Microsoft.AspNetCore.Mvc.Rendering;  **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Controllers  {  **public** **class** EmployeeController : Controller      {  **private** CompanyContext context;  **public** EmployeeController(CompanyContext cc)          {              context = cc;          }    **public** **async** Task<IActionResult> Update(**int** id)          {              Employee emp = **await** context.Employee.Where(e => e.Id == id).FirstOrDefaultAsync();                List<SelectListItem> dept = **new** List<SelectListItem>();              dept = context.Department.Select(x => **new** SelectListItem { Text = x.Name, Value = x.Id.ToString() }).ToList();              ViewBag.Department = dept;    **return** View(emp);          }            [HttpPost]  **public** **async** Task<IActionResult> Update(Employee emp)          {              context.Update(emp);  **await** context.SaveChangesAsync();    **return** RedirectToAction("Index");          }            //...      }  } |

Similar to the department controller’s update action, we are doing the same thing here by fetching the employee record from the database whose id is provided on the method’s parameter.

We are also fetching all the departments and adding them to List<SelectListItem> object and then to a ViewBag. The departments will be shown in html select control on the view.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=eb650be8-90f1-4ce3-6066-fb3e6d5858f5&d_id=122531&imp_id=5927147759052129&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

List<SelectListItem> dept = new List<SelectListItem>();

dept = context.Department.Select(x => new SelectListItem { Text = x.Name, Value = x.Id.ToString() }).ToList();

ViewBag.Department = dept;

*We can also use [TryUpdateModelAsync](https://www.yogihosting.com/insert-records-entity-framework-core/" \l "modelupdate) during the updation of records.*

The update code with TryUpdateModelAsync is given below:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | [HttpPost]  **public** **async** Task<IActionResult> Update(**int**? id)  {  **if** (id == **null**)      {  **return** NotFound();      }  **var** employeeToUpdate = **await** context.Employee.FirstOrDefaultAsync(s => s.Id == id);  **if** (**await** TryUpdateModelAsync<Employee>(          employeeToUpdate,          "",          s => s.Name, s => s.DepartmentId, s => s.Designation))      {  **try**          {  **await** context.SaveChangesAsync();  **return** RedirectToAction(nameof(Index));          }  **catch** (DbUpdateException /\* ex \*/)          {              //Log the error (uncomment ex variable name and write a log.)              ModelState.AddModelError("", "Unable to save changes. " +                  "Try again, and if the problem persists, " +                  "see your system administrator.");          }      }  **return** View(employeeToUpdate);  } |

Here we have int type parameter for the employee id. We then are reading the employee entity with the given id from the database and then calling TryUpdateModel, passing in an explicit allowed properties list that needs to be updated. These are Name, DepartmentId and Designation.

Next, add the Update.cshtml razor view file inside the Views/Employee folder with the following it. The view will show the employee record on a form for the user to update it’s fields.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | @{      ViewData["Title"] = "Update Employee";  }    @model Employee    <**h1** class="bg-info text-white">Create Employee</**h1**>  <**a** asp-action="Index" class="btn btn-secondary">Back</**a**>    <**form** method="post">      <**div** class="form-group">          <**label** asp-for="Name"></**label**>          <**input** asp-for="Name" class="form-control" />      </**div**>      <**div** class="form-group">          <**label** asp-for="Department"></**label**>          <**select** asp-for="DepartmentId" asp-items="ViewBag.Department" class="form-control"></**select**>      </**div**>      <**div** class="form-group">          <**label** asp-for="Designation"></**label**>          <**input** asp-for="Designation" class="form-control" />      </**div**>      <**button** type="submit" class="btn btn-primary">Create</**button**>  </**form**> |

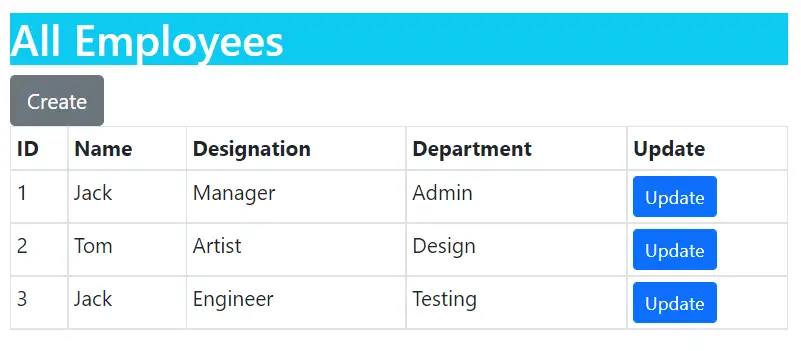
The update employee form will look as shown by the below image.



The final thing to do is to add the Update links to the Index view of the Employee controller. The code to add is shown in highlighted manner below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | @{      ViewData["Title"] = "All Employees";  }  @model IEnumerable<**Employee**>    <**h1** class="bg-info text-white">All Employees</**h1**>  <**a** asp-action="Create" class="btn btn-secondary">Create</**a**>    <**table** class="table table-sm table-bordered">      <**tr**>          <**th**>ID</**th**>          <**th**>Name</**th**>          <**th**>Designation</**th**>          <**th**>Department</**th**>          <**th**>Update</**th**>      </**tr**>      @foreach (Employee emp in Model)      {          <**tr**>              <**td**>@emp.Id</**td**>              <**td**>@emp.Name</**td**>              <**td**>@emp.Designation</**td**>              <**td**>@emp.Department.Name</**td**>              <**td**>                  <**a** class="btn btn-sm btn-primary" asp-action="Update" asp-route-id="@emp.Id">                      Update                  </**a**>              </**td**>          </**tr**>      }  </**table**> |

Run the app and visit the employee controller link – https://localhost:7018/Employee. Here see the update links against all the employees, click any one and update the employee.



Download the source codes:

[DOWNLOAD](https://www.yogihosting.com/wp-content/themes/yogi-yogihosting/download/efcore/EFCoreExample.zip)

Conclusion

In this Entity Framework Core Update Records tutorial we covered how to udpate a record, multiple records and related recods. We also created Update Records CRUD Operations feature at the end. Next tutorial will be on [Delete Records in Entity Framework Core](https://www.yogihosting.com/delete-records-entity-framework-core/).

**Delete Records in Entity Framework Core**

Last Updated: September 26, 2022

**Entity Framework Core** API executes the **DELETE** statement in the database for the entities whose EntityState is set as Deleted. The Remove() method of the DbContext is used for deleting records from the database.

This tutorial is a part of **Entity Framework Core** series.

**Page Contents**

* [Delete Single Record](https://www.yogihosting.com/delete-records-entity-framework-core/#single)
* [Delete Multiple Records](https://www.yogihosting.com/delete-records-entity-framework-core/#multiple)
* [Cascade Delete](https://www.yogihosting.com/delete-records-entity-framework-core/#cascade)
* [Entity Framework Core CRUD Operations – DELETE RECORDS](https://www.yogihosting.com/delete-records-entity-framework-core/#crud)
* [Download Source Codes](https://www.yogihosting.com/delete-records-entity-framework-core/#download)

Delete Single Record

Consider the following code where Entity Framework Core Deletes Record by id. We are deleting Department with Id ‘3’.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **var** dept = **new** Department()  {      Id = 3  };    context.Remove(dept);  **await** context.SaveChangesAsync(); |

Once the Remove() method is called for the department entity, the EF marks the 3rd id’s EntityState as Deleted. So when the SaveChangesAsync() method is called the 3rd department record is deleted from the database.

Ezoic

Interesting EF Core tutorial: [Configurations in Entity Framework Core](https://www.yogihosting.com/configurations-entity-framework-core/)

Delete Multiple Records

Use RemoveRange() method of DbContext to **delete multiple entities** on one go. The below codes remove 3 Department records with Id’s as – 1, 2 & 3 on one go.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | List<Department> dept = **new** List<Department>()  {  **new** Department() { Id=1 },  **new** Department() { Id=2 },  **new** Department() { Id=3 }  };    context.RemoveRange(dept);  **await** context.SaveChangesAsync(); |

If the key value specified in the Remove() or RemoveRange() method does not exist then EF Core will thrown exception of type DbUpdateConcurrencyException.

Cascade Delete

If an entity has relationship with other entities such as one-to-one or one-to-many then **child entities** may be affected when the **parent entity is deleted**. That is the child entities will be either deleted or their foreign key value is set to null. This is know as Cascade Delete behaviour in Entity Framework Core.

With the help of [Fluent API of Entity Framework Core](https://www.yogihosting.com/fluent-api-entity-framework-core/) we can define the Cascade Delete behaviour by 4 main ways. These are:

Ezoic

* **Cascade** : Related entities are also deleted when parent entity is deleted.
* **ClientSetNull** : It is the default value. Here foreign key is set to null.
* **Restrict** : Prevents Cascade delete.
* **SetNull** : The values of foreign key properties will be set to null.

Open the **Database Context File** of the application, there set the OnDelete() method to Cascade, ClientSetNull, Restrict, SetNull by using “DeleteBehavior” enum. Below we have maked it to Cascade.

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **protected** **override** **void** OnModelCreating(ModelBuilder modelBuilder)  {      modelBuilder.Entity<Employee>(entity =>      {          entity.HasOne(d => d.Department)              .WithMany(p => p.Employee)              .HasForeignKey(d => d.DepartmentId)              .OnDelete(DeleteBehavior.Cascade)              .HasConstraintName("FK\_Employee\_Department");      });  } |

Note that we have to re-run the migrations for this behaviour to apply.

Now when we delete a record in the Department table then all the **related records** in the Employee table are also deleted automatically. This is because Deparment is the parent entity and Employee is a child entity.

Check the below code where we are deleting the 5th department id. So all employees that are in the 5th department are also deleted automatically.

|  |  |
| --- | --- |
| 1  2  3 | **var** dept = **new** Department() { Id = 5 };  context.Remove(dept);  **await** context.SaveChangesAsync(); |

Entity Framework Core CRUD Operations – DELETE RECORDS

It’s time we complete the CRUD OPERATIONS with the DELETE RECORDS feature. Recall, on the last tutorial [Update Records in Entity Framework Core](https://www.yogihosting.com/update-records-entity-framework-core/), we added the update records feature to it.

Ezoic

The delete records feature is created by adding a new Delete column on the table given on the Index view where all the entities are shown. The new delete column will contain a small form with a “delete” button. When the user will click this button, the form will submit and calls the Delete action method. The Entity Framework Core will delete the record on this Delete action method.

So start by adding the delete column on the Index.cshtml file for the Department controller, location is Views/Department.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | @{      ViewData["Title"] = "All Departments";  }  @model IEnumerable<Department>    <h1 **class**="bg-info text-white">All Departments</h1>  <a asp-action="Create" **class**="btn btn-secondary">Create</a>    <table **class**="table table-sm table-bordered">      <tr>          <th>ID</th>          <th>Name</th>          <th>Update</th>          <th>Delete</th>      </tr>      @**foreach** (Department dept **in** Model)      {          <tr>              <td>@dept.Id</td>              <td>@dept.Name</td>              <td>                  <a **class**="btn btn-sm btn-primary" asp-action="Update" asp-route-id="@dept.Id">                      Update                  </a>              </td>              <td>                  <form asp-action="Delete" asp-route-id="@dept.Id" method="post">                      <button type="submit" **class**="btn btn-sm btn-danger">                          Delete                      </button>                  </form>              </td>          </tr>      }  </table> |

The delete column will look as shown below.

Next add the Delete action method to the DepartmentController.cs file as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | **using** EFCoreExample.Models;  **using** Microsoft.AspNetCore.Mvc;  **using** Microsoft.EntityFrameworkCore;    **namespace** EFCoreExample.Controllers  {  **public** **class** DepartmentController : Controller      {  **private** CompanyContext context;  **public** DepartmentController(CompanyContext cc)          {              context = cc;          }            [HttpPost]  **public** **async** Task<IActionResult> Delete(**int** id)          {  **var** dept = **new** Department() { Id = id };              context.Remove(dept);  **await** context.SaveChangesAsync();    **return** RedirectToAction("Index");          }            //...        }  } |

Check this feature on the url – https://localhost:7018/Department.

We do the same thing for the Employee entity. So first add the Delete column to the Index.cshtml view of Employee Controller whose location is Views/Employee.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | @{      ViewData["Title"] = "All Employees";  }  @model IEnumerable<Employee>    <h1 **class**="bg-info text-white">All Employees</h1>  <a asp-action="Create" **class**="btn btn-secondary">Create</a>    <table **class**="table table-sm table-bordered">      <tr>          <th>ID</th>          <th>Name</th>          <th>Designation</th>          <th>Department</th>          <th>Update</th>          <th>Delete</th>      </tr>      @**foreach** (Employee emp **in** Model)      {          <tr>              <td>@emp.Id</td>              <td>@emp.Name</td>              <td>@emp.Designation</td>              <td>@emp.Department.Name</td>              <td>                  <a **class**="btn btn-sm btn-primary" asp-action="Update" asp-route-id="@emp.Id">                      Update                  </a>              </td>              <td>                  <form asp-action="Delete" asp-route-id="@emp.Id" method="post">                      <button type="submit" **class**="btn btn-sm btn-danger">                          Delete                      </button>                  </form>              </td>          </tr>      }  </table> |

Next add the Delete action on the EmployeeController.cs file.

[HttpPost]

public async Task<IActionResult> Delete(int id)

{

var emp = new Employee() { Id = id };

context.Remove(emp);

await context.SaveChangesAsync();

return RedirectToAction("Index");

}

Run the app and go to – https://localhost:7018/Employee. Here we can see the delete button against every employee record. Click on any one to delete the employee.

You can get this CRUD feature on the app which is available for download.

[DOWNLOAD](https://www.yogihosting.com/wp-content/themes/yogi-yogihosting/download/efcore/EFCoreExample.zip)

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=f2d539be-0ab5-4f88-7188-67447adbccc5&d_id=122531&imp_id=4238440465053957&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

Conclusion

We learned how to delete records from EF Core. Both deletion of single and mulitple records covered. We also learned Cascade Delete topic and also completed the CRUD OPERATIONS app with the delete records feature.