ASP.NET Core

Entity Framework Core

Rakesh Singh

RAKESHSOFTNET TECHNOLOGIES Hyderabad

http://www.rakeshsoftnet.com

https://www.facebook.com/RakeshSoftNet

https://www.facebook.com/RakeshSoftNetTech

https://twitter.com/RakeshSoftNet

+91 89191 36822

What is Entity Framework Core?

The Microsoft Entity Framework Core or EF Core is Microsoft's implementation of ORM Framework. The applications created using the EF Core does not work with the database directly. The application works only with the API provided by the EF Core for database related operations. The EF Core maps those operations to the database.

Entity Framework Core Application:

Entity Framework Core Console Application:

Creating Entity Framework Core Console Application:

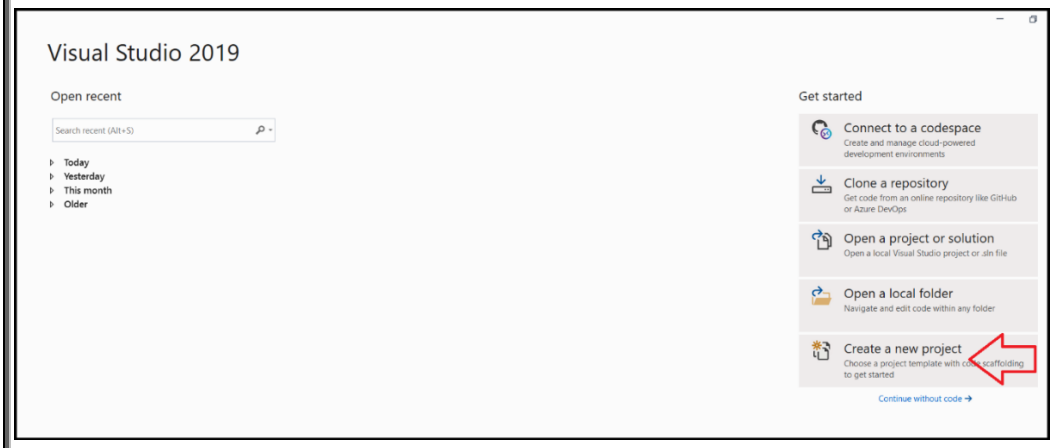
ConsoleApp1

Here, we will learn how to build the Entity Framework Core Console application. We will see how to create the console application and install the required dependencies. Then, we will create an entity model. We will create DBContext, which is used to manage the entity model by creating a DBSet Property. Next, we will use the migration features of the Entity Framework Core (EF Core) to create the database. Finally, we will see how to perform simple tasks like query, insert, update and delete operations on the model and persist the data into the database.

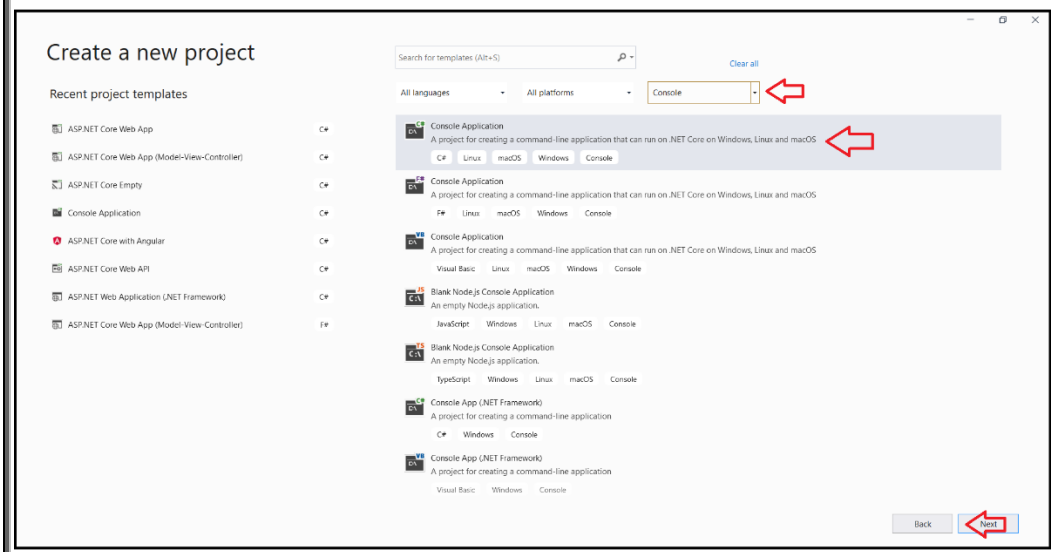
Creating the Entity framework core Console Application

The first step is to create the EF Core console application.

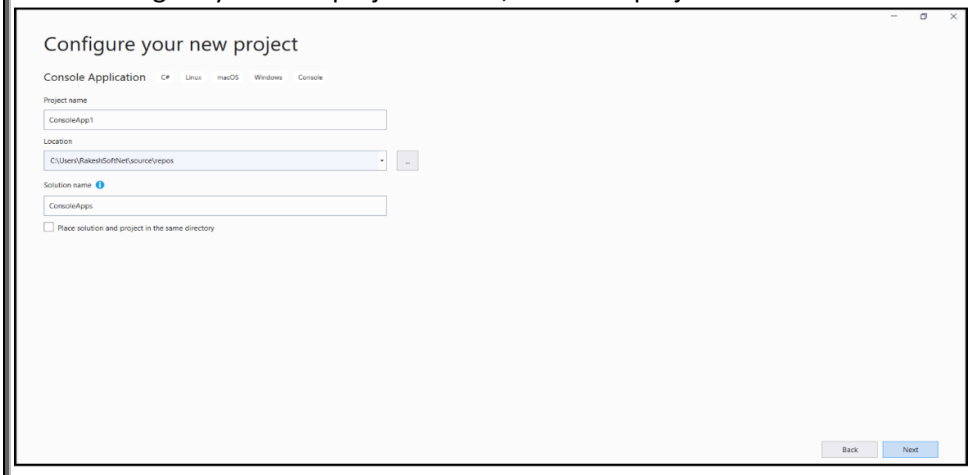
Open Visual Studio 2019. Click on the Create a new project option. (If Visual Studio is already open, then Select File -> New -> Project to open the New Project form)



In the Create a new project wizard, select the Console option, and select the Console App (.NET Core) template. Click on Next when done



In the Configure your new project wizard, enter the project name and click on Create to Create new Console Project.



Installing EF Core:

Now, we need to install Entity Framework Core.

The Microsoft.EntityFrameworkCore is the core library. But installing that alone is not sufficient. We also need to install the EF Core database provider(s). There are many different database providers currently available with the EF Core.

Entity Framework Core can access many different databases through plug-in libraries called database providers.

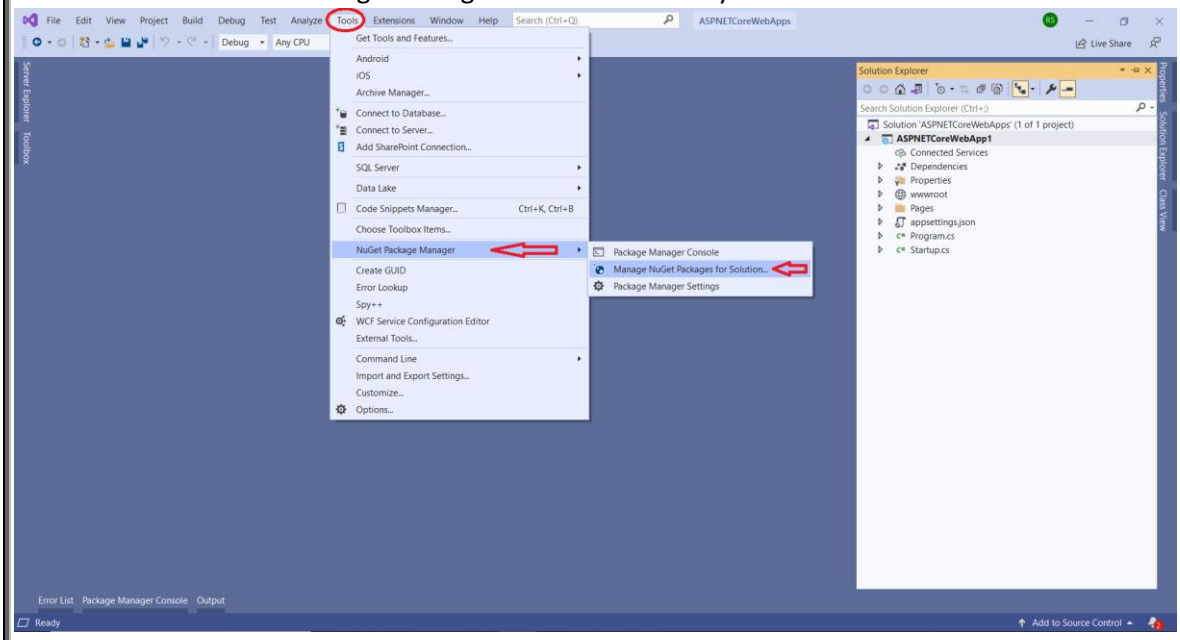
Current Database Providers:

Important Note:

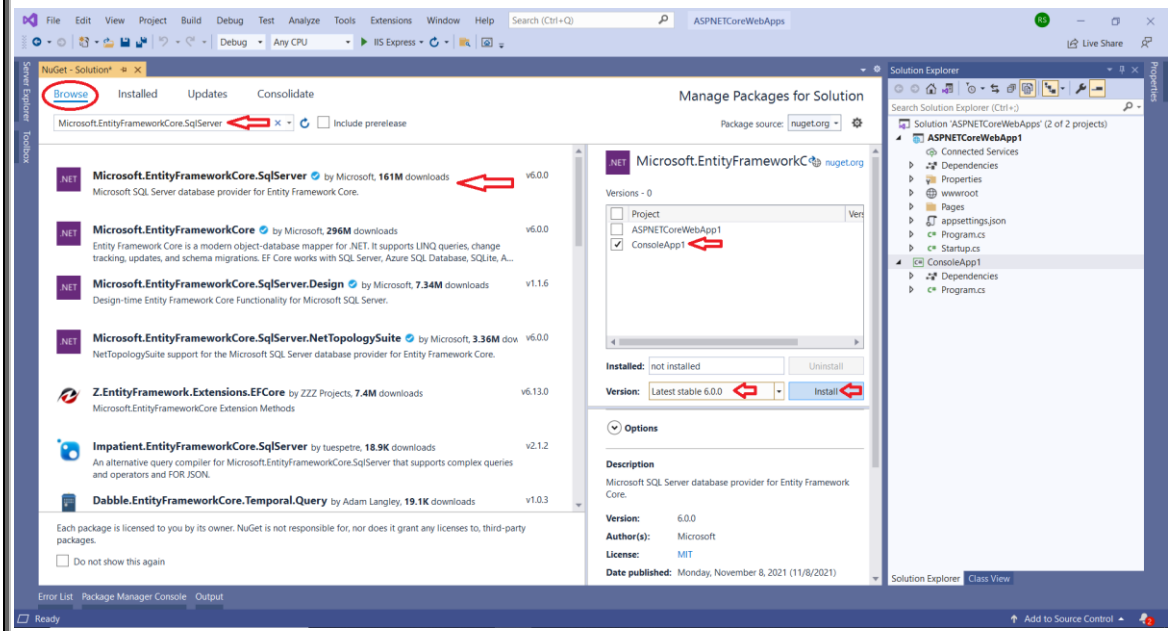
- EF Core providers are built by a variety of sources. Not all providers are maintained as part of the **Entity Framework Core Project**. When considering a provider, be sure to evaluate quality, licensing, support, etc. to ensure they meet your requirements. Also make sure you review each provider's documentation for detailed version compatibility information.
- EF Core providers typically work across minor versions, but not across major versions. For example, a provider released for EF Core 2.1 should work with EF Core 2.2, but will not work with EF Core 3.0.

NuGet Package	Supported database engines
Microsoft.EntityFrameworkCore.SqlServer	SQL Server 2012 onwards
Microsoft.EntityFrameworkCore.Sqlite	SQLite 3.7 onwards
Microsoft.EntityFrameworkCore.InMemory	EF Core in-memory database
Microsoft.EntityFrameworkCore.Cosmos	Azure Cosmos DB SQL API
Npgsql.EntityFrameworkCore.PostgreSQL	PostgreSQL
Pomelo.EntityFrameworkCore.MySql	MySQL, MariaDB
MySql.EntityFrameworkCore	MySQL
Oracle.EntityFrameworkCore	Oracle DB 11.2 onwards
Devart.Data.MySql.EFCore	MySQL 5 onwards
Devart.Data.Oracle.EFCore	Oracle DB 9.2.0.4 onwards
Devart.Data.PostgreSql.EFCore	PostgreSQL 8.0 onwards
Devart.Data.SQLite.EFCore	SQLite 3 onwards
FirebirdSql.EntityFrameworkCore.Firebird	Firebird 3.0 onwards
IBM.EntityFrameworkCore	Db2, Informix
IBM.EntityFrameworkCore-lnx	Db2, Informix
IBM.EntityFrameworkCore-osx	Db2, Informix
EntityFrameworkCore.Jet	Microsoft Access files
Teradata.EntityFrameworkCore	Teradata Database 16.10 onwards
Google.Cloud.EntityFrameworkCore.Spanner	Google Cloud Spanner
FileContextCore	Stores data in files
EntityFrameworkCore.SqlServerCompact35	SQL Server Compact 3.5
EntityFrameworkCore.SqlServerCompact40	SQL Server Compact 4.0
EntityFrameworkCore.OpenEdge	Progress OpenEdge

For example, to use the SQL Server, we need to install the Microsoft.EntityFrameworkCore.SqlServer
For SQLite install the Microsoft.EntityFrameworkCore.Sqlite. When we install the database provider(s), they automatically install the Microsoft.EntityFrameworkCore.
Here, we will be using the SQL Server, hence we will install the Microsoft.EntityFrameworkCore.SqlServer package.
We will use the NuGet Package Manager to install the Entity Framework Core.



- Click on Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution
- Click on Browse
- Enter Microsoft.EntityFrameworkCore.SqlServer and hit enter to search
- Select Microsoft.EntityFrameworkCore.SqlServer and on the right-hand side select the project 'ConsoleApp1'
- Select appropriate version and click to Install.
- Click on I Accept, when asked.
- The Package installs.



Installing Entity Framework Core Tools:

The EF Core Tools contains command-line interface tools (CLI). These tools contain the command to create migrations, apply migrations, generate script migrations, and generate code for a model based on an existing database.

Open the Manage NuGet Packages for Solution window again and search for the Microsoft.EntityFrameworkCore.Tools and install it.

Modelling the database:

EF Core performs data access using a model. A model is nothing but a POCO (Plain Old CLR Object) class. In EF Core we call them entity class.

The EF Core maps these entity classes to a table in the database.

Now, let us create a Product entity class.

- Right click on the solutions project folder and create the folder Models
- Under the Models folder right click and create the class Product
- Change the class to the public and add two properties **Id** and **Name** as shown below:

```
namespace ConsoleApp1.Models
{
    0 references
    public class Product
    {
        0 references
        public int Id { get; set; }
        0 references
        public string Name { get; set; }
    }
}
```

The DBContext class:

The DBContext class manages the entity classes or models. It is the heart of the Entity Framework Core. This class is responsible for:

- Connecting to the database
- Querying & updating the database
- Hold the information needed to configure the database etc...

We create our own context class by inheriting from the DBContext class.

Under the models folder, create the context class named ProductContext as shown below:

ProductContext.cs:

```
using Microsoft.EntityFrameworkCore;

namespace ConsoleApp1.Models
{
    0 references
    public class ProductContext : DbContext
    {
        private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET; Database=EFCoreDB; User Id=sa;
          Password=123";

        0 references
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(connectionString);
        }

        0 references
        public DbSet<Product> Products { get; set; }
    }
}
```

Now, let us understand each line of code.

Our ProductContext class inherits from the DBContext class

```
public class ProductContext : DbContext
```

Next, we have defined our connection string.

```
private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET; Database=EFCoreDB; User Id=sa; Password=123";
```

The OnConfiguring method allows us to configure the DBContext. EF Core calls this method when it instantiates the context for the first time.

This is where we configure the context class. For example, we configure the database providers, the connection string to use, etc...

The OnConfiguring method gets the instance of the **DbContextOptionsBuilder** as its argument. The **DbContextOptionsBuilder** provides API to configure the DBContext.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
```

Inside the **OnConfiguring** method, we call the **UseSqlServer** extension method provided by **Microsoft.EntityFrameworkCore.SqlServer**. The **UseSqlServer** method sets the SQL Server as our database provider. The first argument to the **UseSqlServer** method is the connection string to use.

```
optionsBuilder.UseSqlServer(connectionString);
```

We have hard coded connection string in this example. But you can use the Configuration system provided by the .NET Core to store the connection string in external file.

DbSet:

Creating the model (Entity Type) is not sufficient to map it to database. We must create a DbSet property for each model in the context class. EF Core includes only those types, which have a DbSet property in the model.

The DbSet provides methods like Add, Attach, Remove, etc. on the Entity Types. The context class maps these operations into a SQL query and runs it against the database using the Database Providers.

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

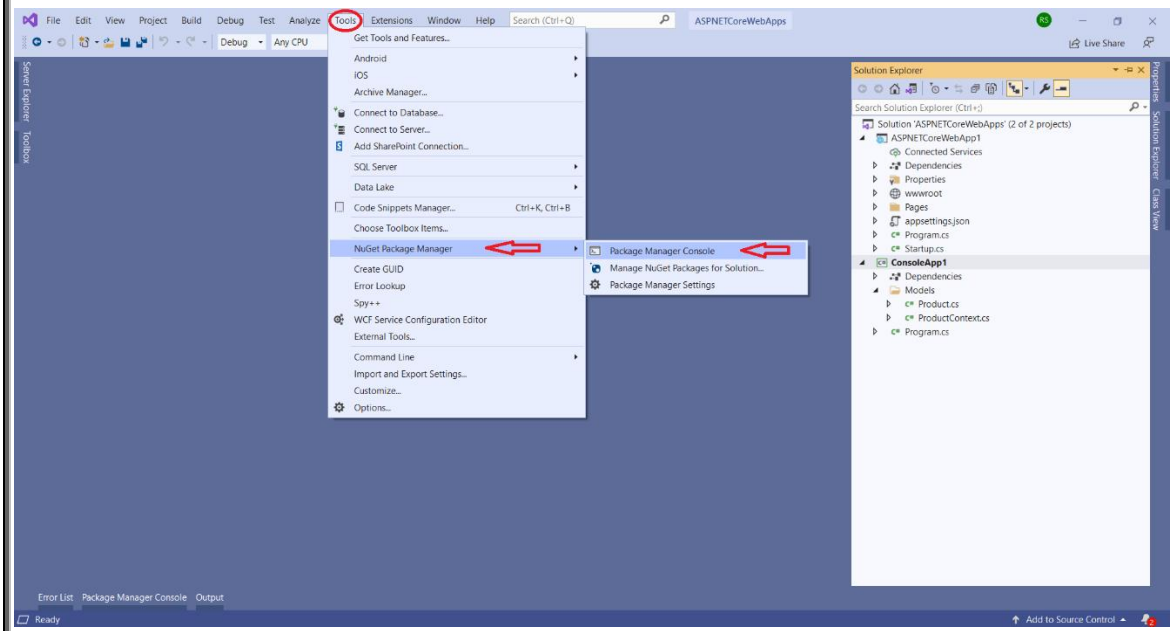
Creating the database:

Now, our model is ready. The model has one entity type Product. We have created ProductContext class to manage the Model. We have defined the DbSet Property of the Product so that it is included in Model. Now it is time to create the database.

In Entity Framework Core, We use **Migrations** to create the database.

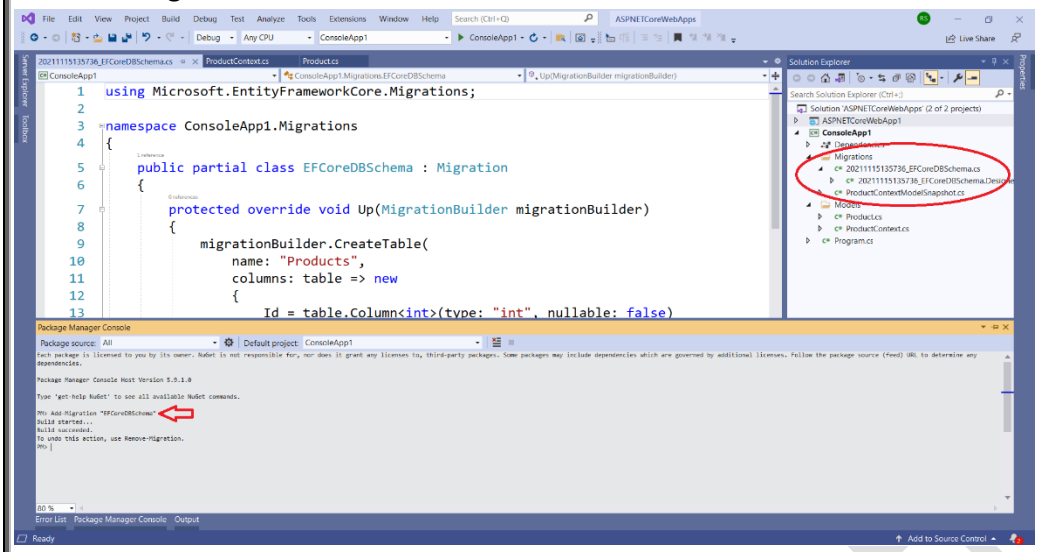
Adding Migration:

Click on Tools -> NuGet Package Manager -> Package Manager Console to open the console.



Run the **Add-Migration** to create the migration.

PM> **Add-Migration "EFCoreDBSchema"**



The Add-Migration generates the instructions to generate the SQL commands to update the underlying database to match the model. You can see that the three files created and added under the Migrations folder.

Create the database:

The next step is to create the database using the **Migrations** from the previous step.

Open the **Package Manager Console** and run the command **Update-Database**

The Update-Database uses the migrations to generate the SQL Queries to update the database. If the database does not exist it will create it. It uses the connection string provided while configuring the DBContext to connect to the database.

PM> **Update-Database**

Build started...

Build succeeded.

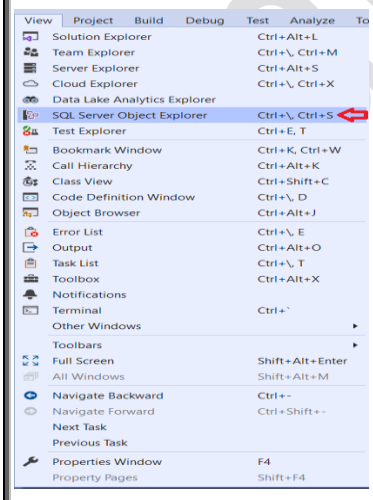
Applying migration ' 20211115135736_EFCoreDBSchema'.

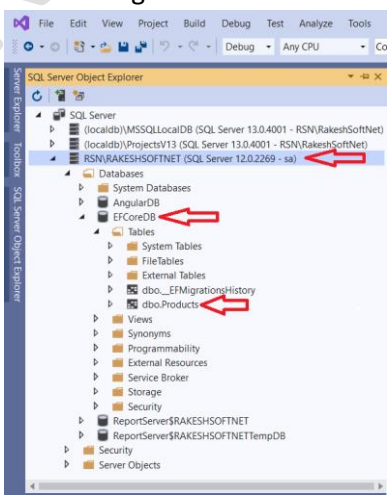
Done.

The Database:

Created database and their tables can be viewed using "**SQL Server Object Explorer**" or "**Server Explorer**" in Visual Studio.

- Click on **View** -> **SQL Server Object Explorer** to Open **SQL Server Object Explorer**.
- Go to the RSN\RAKESHSOFTNET
- You will see the EFCoreDB database is generated as shown in the image below





CRUD Operations

Let us now add simple create/read/update & delete operations on the Product Entity model and persist the data to the database.

Inserting Data:

Inserting to the database is handled by the SaveChanges method of the DBContext object. To do that, you need to follow these steps.

Create a new instance of the DBContext class.

```
using (var db = new ProductContext())
```

Create a new instance of the domain class Product and assign values to its properties.

```
Product product = new Product();
product.Name = "Monitor";
```

Next, add it to the DBContext class so that Context becomes aware of the entity.

```
db.Add(product);
```

Finally, call the SaveChanges method of the DBContext to persist the changes to the database.

```
db.SaveChanges();
```

Here is the list of the complete InsertProduct method, which is invoked from the Main method of the Program class in Program.cs file.

```
using System;
using ConsoleApp1.Models;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("---CRUD Example---");

            InsertProduct();
        }

        static void InsertProduct()
        {
            using(var db = new ProductContext())
            {
                Product product = new Product();
                product.Name = "Monitor";

                db.Products.Add(product);

                db.SaveChanges();

                Console.WriteLine("Product Details Inserted Successfully !!!");
            }
        }
    }
}
```

Run the code and Open the database to verify that the values are inserted.

Querying the Data:

The queries are written against the **DbSet** property of the entity. The queries are written using the **LINQ to Entities** API. There are two ways in which you can write queries in LINQ. One is **Method Syntax** & the other one is **Query Syntax**.

The following example code retrieves the Product using the **Method Syntax**. It uses the **ToList()** method of the **DbSet**. The **ToList()** sends the select query to the database and retrieves and convert the result into a List of Products as shown below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using ConsoleApp1.Models;
static void ViewProducts()
{
    using (var db = new ProductContext())
    {
        List<Product> products = db.Products.ToList();
        Console.WriteLine("---Product List---");
        foreach (var product in products)
        {
            Console.WriteLine("Product Id: {0}, Product Name: {1}", product.Id, product.Name);
        }
    }
}
```

And in the Main method call the ViewProducts method to see the list of Products.

```
static void Main(string[] args)
{
    Console.WriteLine("---CRUD Example---");

    ViewProducts();
}
```

Update the Record

The following code shows how to update a single entity.

First, we use the **Find** method to retrieve the single Product. The **Find** method takes the id (primary key) of the product as the argument and retrieves the product from the database and maps it into the Product entity.

Next, we update the Product entity.

Finally, we call **SaveChanges** to update the database.

```
static void UpdateProduct()
{
    using (var db = new ProductContext())
    {
        Product product = db.Products.Find(1);

        if (product != null)
        {
            product.Name = "Keyboard";

            db.SaveChanges();

            Console.WriteLine("Product Details Updated Successfully !!!");
        }
    }
}
```

In the Main method invoke the UpdateProduct and then ViewProducts to verify whether the values are changed.

```
static void Main(string[] args)
{
    Console.WriteLine("---CRUD Example---");

    ViewProducts();

    UpdateProduct();

    ViewProducts();
}
```

Delete the Record:

The following code demonstrates how to delete the record from the database.

Deleting is done using the Remove method of the DbSet. We need to pass the entity to delete as the argument to the remove method as shown below:

```
static void DeleteProduct()
{
    using (var db = new ProductContext())
    {
        Product product = db.Products.Find(1);

        if (product != null)
        {
            db.Products.Remove(product);

            db.SaveChanges();

            Console.WriteLine("Product Details Deleted Successfully !!!");
        }
    }
}
```

In the **Main** method invoke the **DeleteProduct** and then **ViewProducts** to verify whether the values are changed.

```
static void Main(string[] args)
{
    Console.WriteLine("---CRUD Example---");

    ViewProducts();

    DeleteProduct();

    ViewProducts();
}
```

Here, we learned how to create a console project using Entity Framework Core. We created a model. Added DBContext to the project and passed our connection string to it. Then, we added DbSet Property of the Product model to the Context class. The Context manages all models, which exposes the DbSet Property. Then, we used Migrations to create the database.

Real Time Example with User Interface in Console Application:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using ConsoleApp1.Models;
5
6  namespace ConsoleApp1
7  {
8      class Program
9      {
10         static void Main(string[] args)
11         {
12             Console.WriteLine("---CRUD Example---");
13
14             Options:
15                 Console.WriteLine("---Choose Option----");
16                 Console.WriteLine("1. Insert Product Data");
17                 Console.WriteLine("2. Update Product Data");
18                 Console.WriteLine("3. Delete Product Data");
19                 Console.WriteLine("4. View Products Data");
20                 Console.WriteLine("5. Exit");
```

```
21
22              Console.Write("Enter Option: ");
23              int option = int.Parse(Console.ReadLine());
24
25              if (option == 1)
26              {
27              AddProduct:
28                  Console.Write("Enter Product Name: ");
29                  string productName = Console.ReadLine();
30
31                  InsertProduct(productName);
32
33                  Console.Write("Do you want to add another product? (y/n)");
34                  char choice = Console.ReadKey().KeyChar;
35
36                  Console.WriteLine();
37
38                  if (choice == 'y' || choice == 'Y')
39                  {
40                      goto AddProduct;
41                  }
42
43                  goto Options;
44              }
45              else if (option == 2)
46              {
47                  Console.Write("Enter Product Id to Update their Details: ");
48                  int productId = int.Parse(Console.ReadLine());
49                  if (GetProduct(productId) != null)
50                  {
51                      Console.Write("Enter Product Name: ");
52                      string productName = Console.ReadLine();
53
54                      UpdateProduct(productId, productName);
55
56                  }
57                  else
58                  {
59                      Console.WriteLine("No Product Details Found to Update.");
60                  }
61
62                  goto Options;
63              }
64              else if (option == 3)
65              {
66                  Console.Write("Enter Product Id to Delete their Details: ");
67                  int productId = int.Parse(Console.ReadLine());
68                  if (GetProduct(productId) != null)
69                  {
70                      DeleteProduct(productId);
71                  }
72                  else
73                  {
74                      Console.WriteLine("No Product Details Found to Delete.");
75                  }
76
77                  goto Options;
78              }
79              else if (option == 4)
80              {
81                  ViewProducts();
82
83                  goto Options;
84              }
85          }
```

```
86
87          1 reference
            static void InsertProduct(string productName)
88          {
89              using (var db = new ProductContext())
90              {
91                  Product product = new Product();
92                  product.Name = productName;
93
94                  db.Products.Add(product);
95
96                  db.SaveChanges();
97
98                  Console.WriteLine("Product Added Successfully !!!");
99              }
100         }
101
102         1 reference
            static void UpdateProduct(int productId, string productName)
103         {
104             using (var db = new ProductContext())
105             {
106                 Product product = db.Products.Find(productId);
107
108                 product.Name = productName;
109
110                 db.SaveChanges();
111
112                 Console.WriteLine("Product Details Updated Successfully !!!");
113             }
114         }
115
116         1 reference
            static void DeleteProduct(int productId)
117         {
118             using (var db = new ProductContext())
119             {
120                 Product product = db.Products.Find(productId);
121
122                 db.Products.Remove(product);
123
124                 db.SaveChanges();
125
126                 Console.WriteLine("Product Details Deleted Successfully !!!");
127             }
128         }
129
130         1 reference
            static void ViewProducts()
131         {
132             using (var db = new ProductContext())
133             {
134                 List<Product> products = db.Products.ToList();
135                 Console.WriteLine("---Product List---");
136                 if (products.Count > 0)
137                 {
138                     foreach (Product product in products)
139                     {
140                         Console.WriteLine("Product Id: {0}, Product Name: {1}", product.Id, product.Name);
141                     }
142                 }
143                 else
144                 {
145                     Console.WriteLine("No Product Data Available.");
146                 }
147             }
148         }
149
```

```
2 references
        static Product GetProduct(int productId)
        {
            using (var db = new ProductContext())
            {
                Product product = db.Products.Find(productId);

                return product;
            }
        }
    }
}
```

Output:

```
C:\Users\RakeshSoftNet\source\repos\ASPNETCoreWebApps\ConsoleApp1\bin\Debug\net5.0\ConsoleApp1.exe
---CRUD Example---
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option:
```

Now enter option 1 to insert product data as shown below.

```
C:\Users\RakeshSoftNet\source\repos\ASPNETCoreWebApps\ConsoleApp1\bin\Debug\net5.0\ConsoleApp1.exe
---CRUD Example---
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option: 1
Enter Product Name: Monitor
Product Added Successfully !!!
Do you want to add another product? (y/n)
```

It prompts that "Do you want to add another product? (y/n)" so enter 'y' to add another product or 'n' to choose another option. Here we entered 'y' to add another product as sown below.

```
Do you want to add another product? (y/n)y
Enter Product Name: Mouse
Product Added Successfully !!!
Do you want to add another product? (y/n)
```

Now enter 'n' to choose another option like '4' to view the products.

```
Do you want to add another product? (y/n)n
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option: 4
---Product List---
Product Id: 1, Product Name: Monitor
Product Id: 2, Product Name: Mouse
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option:
```

Now enter option like '2' to update the product details by ProductId.

```
Enter Option: 2
Enter Product Id to Update their Details: 1
Enter Product Name: Desktop Monitor
Product Details Updated Successfully !!!
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option:
```

Now enter option like '4' to view the products with updated product details too.

```
Enter Option: 4
---Product List---
Product Id: 1, Product Name: Desktop Monitor
Product Id: 2, Product Name: Mouse
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option:
```

Now enter option like '3' to delete the product by ProductId.

```
Enter Option: 3
Enter Product Id to Delete their Details: 2
Product Details Deleted Successfully !!!
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option:
```

Now enter option like '4' to view the products.

```
Enter Option: 4
---Product List---
Product Id: 1, Product Name: Desktop Monitor
---Choose Option----
1. Insert Product Data
2. Update Product Data
3. Delete Product Data
4. View Products Data
5. Exit
Enter Option:
```

Finally enter option like '5' to Exit.

Installing Entity Framework Core

Here, we will learn the various ways to install the Entity Framework Core.

What to Install?

The Microsoft.EntityFrameworkCore is the core library. But installing that alone is not sufficient. We also need to install the EF Core database provider(s).

For Example, to use the SQL Server, we need to install the Microsoft.EntityFrameworkCore.SqlServer.

For SQLite install the Microsoft.EntityFrameworkCore.Sqlite.

When we install the database provider(s), they automatically install the Microsoft.EntityFrameworkCore.

The following are the some of the database provider(s)

- Microsoft.EntityFrameworkCore.SqlServer (SqlServer)
- Microsoft.EntityFrameworkCore.Sqlite (Sqlite)
- MySql.Data.EntityFrameworkCore (Official version for MySQL)
- Pomelo.EntityFrameworkCore.MySql (MySQL, MariaDB)
- Npgsql.EntityFrameworkCore.PostgreSQL (PostgreSQL)
- Oracle.EntityFrameworkCore (Oracle)
- EntityFrameworkCore.Jet (Microsoft Access)
- Microsoft.EntityFrameworkCore.Cosmos (Azure Cosmos DB SQL API)

We also need to install the **Entity Framework Core tools**. These tools contain the help us to create migrations, apply migrations, and generate code for a model based on an existing database. The tools are available in the package **Microsoft.EntityFrameworkCore.Tools**.

Various ways to Install Entity Framework Core

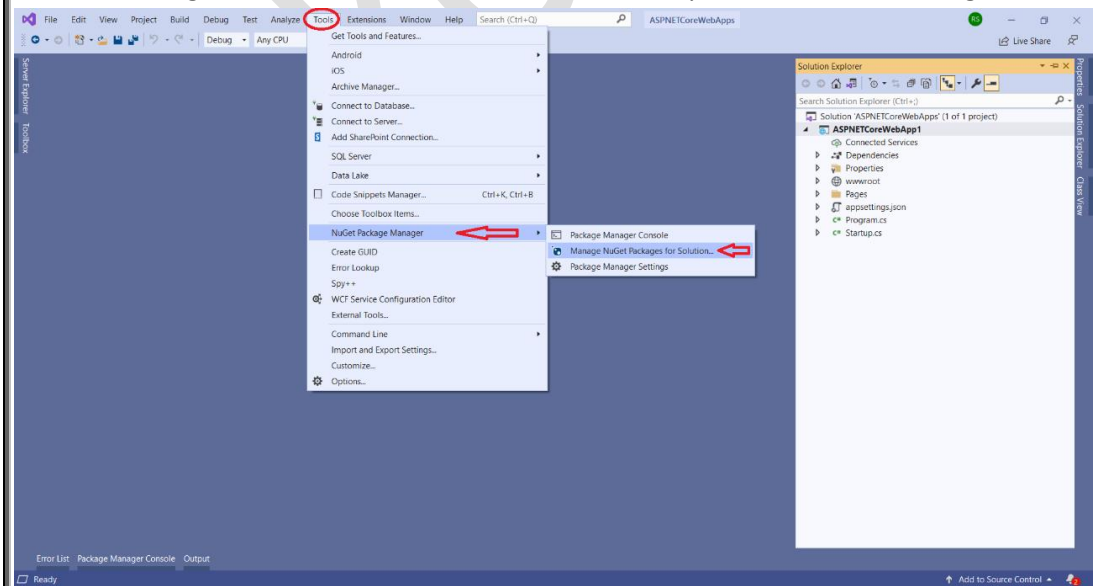
We can install Entity Framework Core in number of ways.

- Using Package Manager GUI Tools
- Using Package Manager Console
- Using .NET Core CLI (Command Line Interface) Tool
- Using Modify the Project file

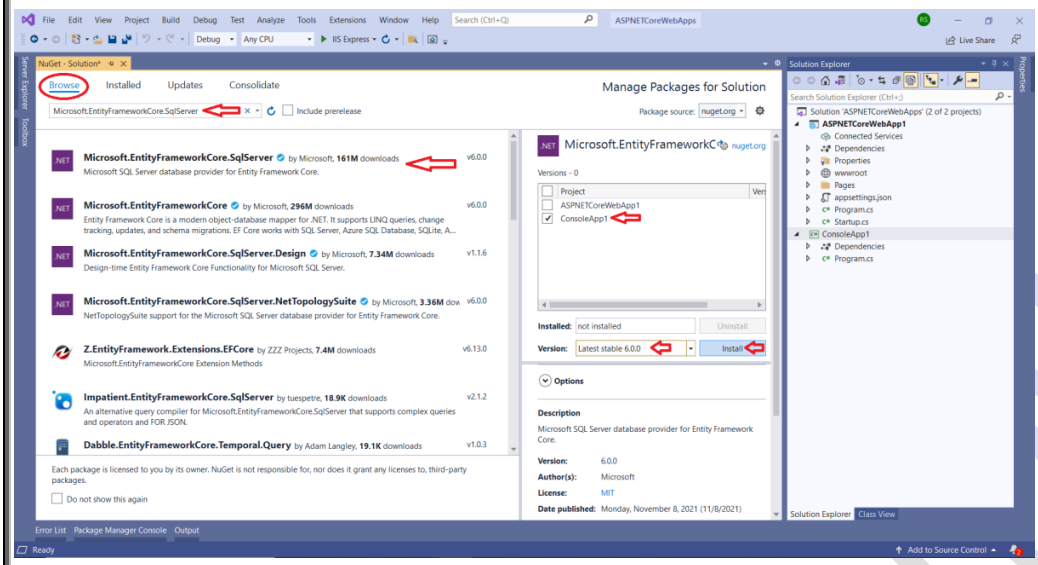
Using Package Manager GUI Tools:

In Visual Studio go to Tools menu select **NuGet Package Manager -> Manage NuGet Packages for Solution**.

You can also right-click on the Solution in the Solution Explorer and select Manage NuGet Packages for Solution.



Under **Browse** tab search for the **Microsoft.EntityFrameworkCore.SqlServer** Package. Select the package to install. On the right-hand side, select the projects to which you want to add the package. Also choose the version and click on install to begin the installation.



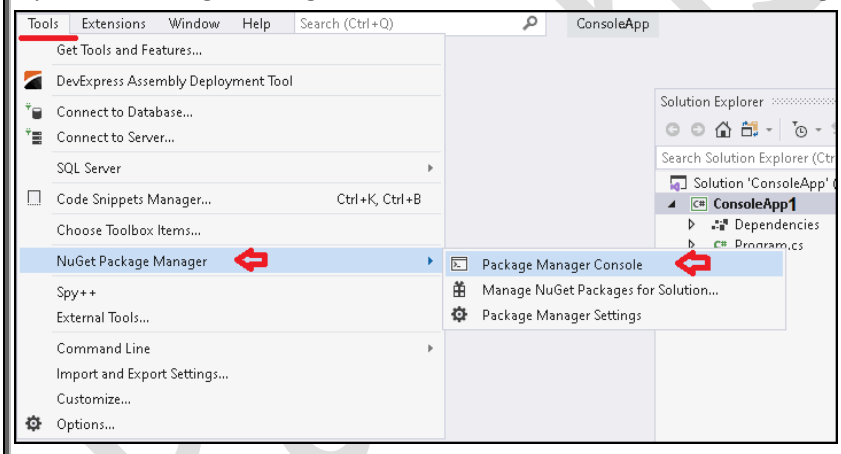
Installation will you to accept the license terms. Select accept to complete the installation.

Install Entity Framework Core Tools

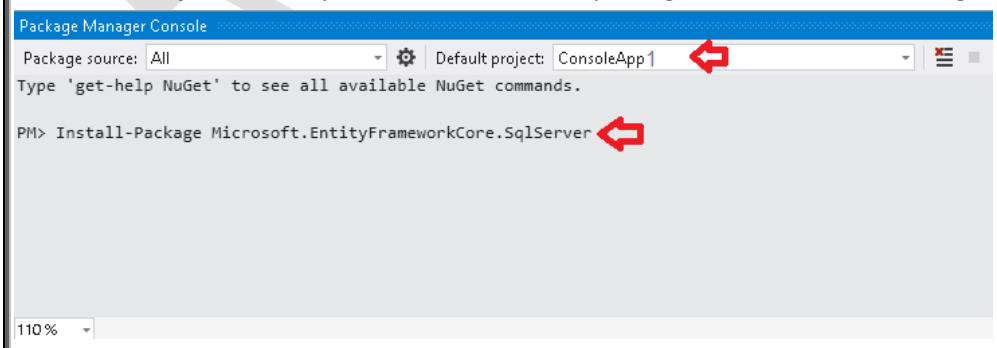
Search for the **Microsoft.EntityFrameworkCore.Tools** in the package manager and repeat the above steps to install the tools.

Using Package Manager Console

Open the Package Manager Console from the Tools -> NuGet Package Manager -> Package Manager Console



Select the Project, where you want to install the package. Use the Install-Package <PackageName> to install the package



Run the following commands. This will install the latest available version

- Install-Package Microsoft.EntityFrameworkCore.SqlServer
- Install-Package Microsoft.EntityFrameworkcore.Tools

Use the -Version flag to install the previous version

- Install-Package Microsoft.EntityFrameworkCore.SqlServer -Version 3.1.10

Using .Net Core CLI (Command Line Interface) Tool:

Using the .NET Core CLI command dotnet add package is another way to install the Entity Framework Core

Open the command prompt and cd into the project folder (folder with .csproj file). Run the following command:

- dotnet add package Microsoft.EntityFrameworkCore.SqlServer

Specify the name of the project after the add option to install to only in that project. This is useful, when you have multiple projects and in the solution folder.

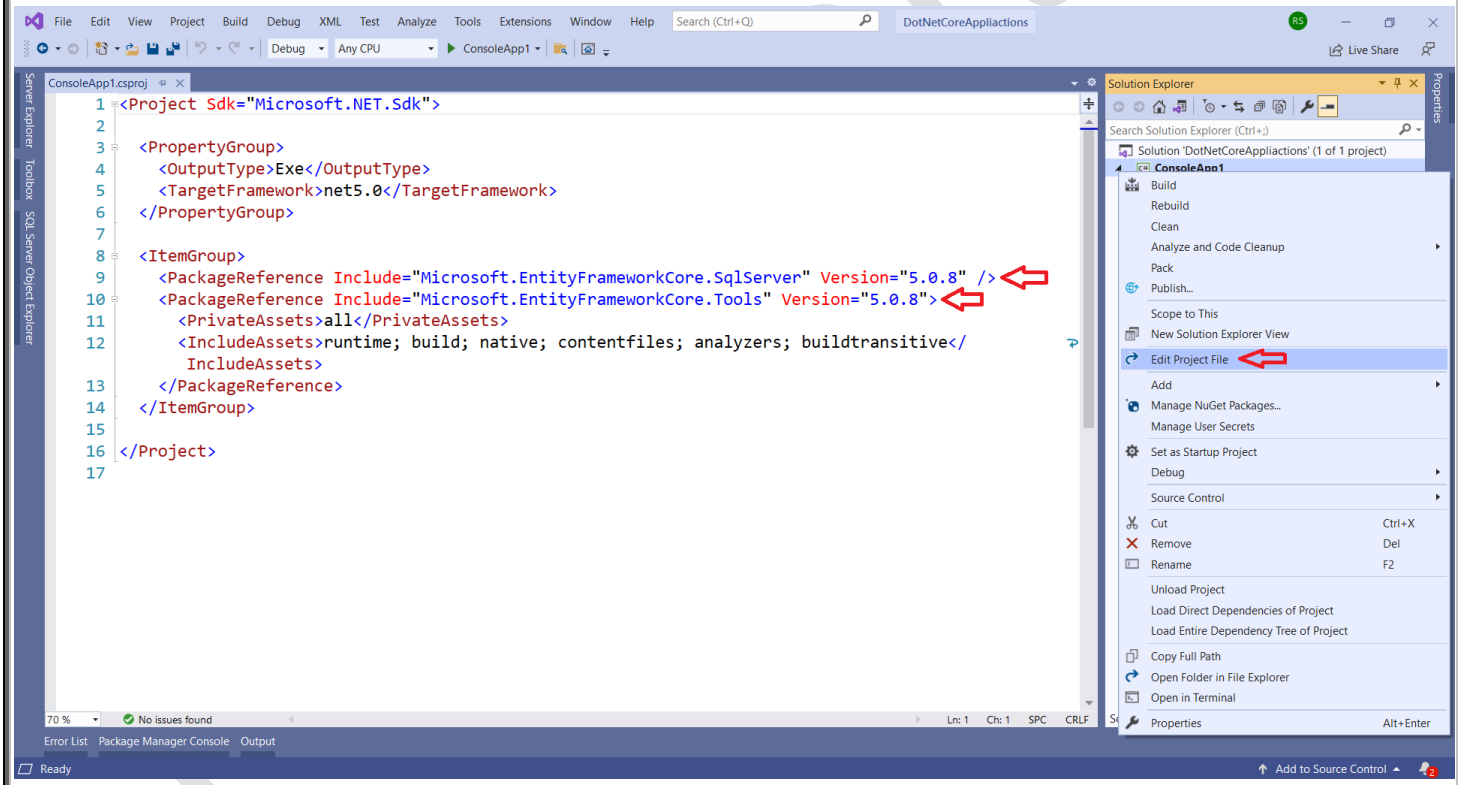
- dotnet add ConsoleApp1 package Microsoft.EntityFrameworkCore.SqlServer

Use the -v flag to choose the version.

- dotnet add ConsoleApp1 package Microsoft.EntityFrameworkCore.SqlServer -v 3.1.10

Using Modify the Project file:

Right click on the project and click on edit project file.



Update the project add the PackageReference under ItemGroup node

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.SqlServer" Version="5.0.8" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="5.0.8">
    <PrivateAssets>all</PrivateAssets>
    <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
  </PackageReference>
</ItemGroup>
```

Database Connection String in Entity Framework Core:

Here we will learn how to provide Database Connection String in Entity Framework Core Applications. The DBContext connects to the database using Database Providers. These Providers requires a connection string to connect to the database.

The way the connection string is specified has changed from the previous version of the Entity Framework.

There are several ways by which you can provide the connection string to Entity Framework Core application. We look at some of them in detail.

Where to Store the connection strings:

The connection strings were stored in **web.config** file in older version of ASP.NET Applications.

The newer ASP.NET Core applications can read the configurations from the various sources like **appsettings.json**, user secrets, environment variables, command line arguments etc... You can store connection strings anywhere you wish to. For this example, we will use the appsettings.json

Connection String in appsettings.json:

The appsettings.json can be created for each environment separately.

The appsettings.json holds the settings that are the common to all the environment like development, production & testing environment.

The appsettings.<environmentName>.json file holds the settings that are used in the environment specified by the ASPNETCORE_ENVIRONMENT variable. For example, appsettings.production.json holds the settings for the production environment and appsettings.development.json holds the settings for the development environment.

The configuration is stored in name-value pairs. These name-value pairs into a structured hierarchy of sections. Hence each connection string is stored as a separate node under the section ConnectionStrings as shown below:

```
{
        "ConnectionStrings":
        {
                "conStr": "Data Source=RSN\\RAKESHSOFTNET;Database=CoreDB;User Id=sa;Password=123"
        }
}
```

There is no requirement to name the section as "ConnectionStrings". You can name whatever you want it to be. But naming it ConnectionStrings allows us to make use of the GetConnectionString method of the IConfiguration object.

Passing Connection String to DBContext:

You can create DBContext and configure it by passing the connection string in several ways depending on the type of application like ASP.NET Core Razor Pages Apps, ASP.NET Core MVC Apps, ASP.NET Core API Apps or Console Apps and whether you want to make use of Dependency Injection or not.

ASP.NET Core MVC Application:

In MVC application, the DBContext is injected using the Dependency Injection. To do that we need to register the DBContext in ConfigureServices method of the Startup class. Hence we also need to read the connection string in startup class.

Reading the Connection String in the Startup class:

To read from the configuration, we need an instance of IConfiguration object. The IConfiguration is available to be injected via Dependency Injection.

Hence, we can inject it into the startup class using the constructor as shown below:

```
public IConfiguration Configuration { get; }
public Startup(IConfiguration configuration)
{
        Configuration = configuration;
}
```

You can read it as shown below:

Configuration.GetSection("ConnectionStrings")["conStr"]

Configuration.GetSection("ConnectionStrings:conStr")

Or by using the GetConnectionString() method as shown below. (If you have provided the section name as "ConnectionStrings")

Configuration.GetConnectionString("conStr")

Passing the Connection string to DBContext:

We create our own context class by inheriting from the DBContext:

StudentContext.cs

```
public class StudentContext : DbContext
{
   public StudentContext(DbContextOptions options): base(options)
   {
   }
   public DbSet<Student> Students { get; set; }
}
```

Note the StudentContext constructor requires the instance of the DbContextOptions contains the configuration information such as type of database to use, connection string etc.

Next, we need to register the StudentContext for dependency injection. This is done in the ConfigureServices method of the Startup class.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddControllersWithViews();

   var connectionString = Configuration.GetConnectionString("conStr");

   services.AddDbContext<StudentContext>(
      options => options.UseSqlServer(connectionString)
   );
}
```

The AddDbContext extension method provided by entity framework core is used to register our context class. The first argument is of Action<T>, where you get the reference to the DbContextOptionsBuilder. The DbContextOptionsBuilder is used to configure the DbContextOptions. Here we use UseSqlServer method to register SQL Server as the database provider, passing the connection string along with it.

Finally, you can use the DbContext by injecting it in the constructor of the Controller or services etc. as shown below:

```
public class StudentController : Controller
{
   private StudentContext db;

   public StudentController(StudentContext context)
   {
      db = context;
   }
   public IActionResult Index()
   {
      return View();
   }
}
```

DbContext in Entity Framework Core

Here, we will look at DBContext. We will also look at how to create a DBContext class in our ASP.NET Core application. Next, we will see you how to register DBContext for Dependency Injection. Later we will take a look at how to configure DBContext using DbContextOptions & DbContextOptionsBuilder. Finally, we will look at various functions performed by the DBContext

What is DbContext?

The DbContext is heart of the EF Core. It is the connection between our entity classes and the database. The DbContext is responsible for the database interaction like querying the database and loading the data into memory as entity. It also tracks changes made to the entity and persists the changes to the database.

How to Use/Create DbContext:

To use DbContext, we need to create a context class and derive it from the DbContext base class.

The following is the example of the Context class (StudentContext):

Creating the Context Class:

```
public class StudentContext : DbContext
{
        public StudentContext(DbContextOptions options): base(options)
        {
        }
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
                // Use this to configure the context
        }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
                //Use this to configure the model
        }
        public DbSet<Student> Students { get; set; }
}
```

The Context class above has a constructor which accepts the **DbContextOptions** as its argument. The **DbContextOptions** carries the configuration information needed to configure the DbContext.

The **DbContextOptions** can also be configured using the OnConfiguring method. This method gets the DbContextOptionsBuilder as its argument. It is then used to create the DbContextOptions.

The **OnModelCreating** is the method where you can configure the model. The instance of the **ModelBuilder** is passed as the argument to the **OnModelCreating** method. The **ModelBuilder** provides the API, which is used to configure the data type, relationships between the models etc...

Finally, we define the **DbSet** property for the each entity (database table). In the above example, Students represents the database table i.e. Student

Registering for the Dependency Injection:

Next, we need to register our context class to be available via dependency injection. This is done under the ConfigureServices method of the Startup class in the Startup.cs file.

```
public IConfiguration Configuration { get; }

public Startup(IConfiguration configuration)
{
        Configuration = configuration;
}

public void ConfigureServices(IServiceCollection services)
{
        services.AddControllersWithViews();

        var connectionString = Configuration.GetConnectionString("conStr");

        services.AddDbContext<StudentContext>(
           options => options.UseSqlServer(connectionString)
        );
}
```

First, we need connection string, which can be obtained from the IConfiguration instance.

```
var connectionString = Configuration.GetConnectionString("conStr");
```

Next, we use AddDbContext extension method to register the StudentContext in DI container. The first argument is of Action<T>, where you get the reference to the DbContextOptionsBuilder. The DbContextOptionsBuilder is used to configure the DbContextOptions.

```
services.AddDbContext<StudentContext>(
           options => options.UseSqlServer(connectionString)
        );
```

The DbContextOptionsBuilder uses the UseSqlServer extension method, which registers the SQL Server database provider to be used with EF Core. We pass the connection string to the UseSqlServer method.

Injecting Context via constructor:

Finally, we can use the context in the controller or in other services by using dependency injection as shown below:

```
private StudentContext db;

public StudentController(StudentContext context)
{
          db = context;
}
```

Database Providers:

The DBContext connects to the database using the Database Providers. In the example above we used the UseSqlServer extension method

The Database Provider are a set of API that used to connect to a particular database. There are many different database Providers currently available with the EF Core. You can find the complete list of database providers. These are the following most commonly used database providers:

Microsoft SQL Server - Microsoft.EntityFrameworkCore.SqlServer

SQLite - Microsoft.EntityFrameworkCore.Sqlite

MySQL (Official) - MySql.Data.EntityFrameworkCore

Pomelo (MySQL) - Pomelo.EntityFrameworkCore.MySql

Npgsql (PostgreSQL) - Npgsql.EntityFrameworkCore.PostgreSQL

IBM Data Server - IBM.EntityFrameworkCore

Once you install the database provider, then you can configure the database provider using the extension method provided them using the DBContextOptionsBuilder.

For Example to use SQL Server install the package:

- **Install-Package Microsoft.EntityFrameworkCore.SqlServer**

Then use the UseSqlServer extension method to register the SQL Server Database provider. This can be done while registering the service.

```
services.AddDbContext<StudentContext>(
          options => options.UseSqlServer(connectionString)
        );
```

OR by overriding the OnConfiguring method:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
        if(optionsBuilder.IsConfigured)
        {
                optionsBuilder.UseSqlServer("Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123");
        }
}
```

Configuring the DbContext:

The DbContext is configured using the DbContextOptions.

DbContextOptions:

The DbContext requires the DbContextOptions instance in order to perform any task.

The DbContextOptions instance carries configuration information such as database providers to use, connection strings and any database related configuration information.

DbContextOptionsBuilder:

We build DbContext options using the DbContextOptionsBuilder API.

There are two way you can build the DbContextOptions:

One option is to create **DbContextOptions** externally and pass it in the constructor of **DbContext** class

The second option is to override the **OnConfiguring(DbContextOptionsBuilder)** method and create the **DbContextOptions**

Using the Constructor:

To use the constructor first we need to define the context constructor as shown below:

```
public StudentContext(DbContextOptions options): base(options)
{
}
```

Here, we have two options.

One is to use the dependency injection, which is what we used in the example above.

```
services.AddDbContext<StudentContext>(
    options => options.UseSqlServer(connectionString)
);
```

The AddDbContext not only registers the StudentContext but also registers the DbContextOptions available for injection, which is provided as an Anonymous function

The second option is to create the DbContextOptions externally and pass it to while creating the context as shown in the following example:

```
var optionsBuilder = new DbContextOptionsBuilder<StudentContext>();
    optionsBuilder.UseSqlServer(connectionString);
    db = new StudentContext(optionsBuilder.Options);
```

Using the OnConfiguring Method

The second option is to use the OnConfiguring method to configure the DbContextOptions as shown below:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer("Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123");
    }
}
```

It is possible to use both the Constructor & OnConfiguring method to configure the DbContext. In such a scenario the OnConfiguring is executed last. Hence any changes applied in the constructor is overwritten.

The optionsBuilder.IsConfigured returns a boolean value indicating whether any options have been configured.

Functions of DBContext

Managing Database connection

The DBContext opens and manages the database connection pool. It will reuse the connections wherever possible and creates a new connection only when needed.

Configuring Model & relationships

The DBContext builds the model based on a set of conventions. You can override those conventions by providing additional configuration to build the model. The Fluent API Provides more control over the building of models

The model configuration is done in the OnModelCreating method using the ModelBuilder API

Querying & saving data to the database

In order to use Entity framework core, we need to define the DbSet property for each entity (or tables). Then we need to configure the model & define relationships between the entities using the ModelBuilder API

Once we have these in place, then we can Write and execute queries against those models, which gets translated to the database query and executed. The returned results are Materialized and converted to entity objects. Any changes made to those entities are persisted back to the database.

Change Tracking

The entities can be added/deleted or modified. The DBContext's change tracker keeps track of these operations and sets the EntityState of each entity. This state is then used while saving the entity into the database, by generating the proper insert, alter, and delete queries.

Transaction Management

By default each SaveChanges are wrapped in a single transaction.

You can control the transactions better by using DbContext.Database API. You can begin transaction commit, and rollback transactions

DbSet in Entity Framework Core

The DbSet represents the collection of all entities in the context. Every model must expose the DbSet property to become part of the Context and managed by it. We use the DbSet to query, insert, update & Delete entities. Here, we will look into DbSet and its methods in detail.

Modelling the Database

In EF Core we create POCO Classes to represent our database. Each table in the database, will gets its own class. For Example, the Product Table in the database can be represented using the following class. Here id & name represents the columns in the table.

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

It is a simple POCO class. The EF Core knows nothing about the class. It will not be included in the model and will not be mapped to the database.

DbSet

To include the above class in the model, we must define a DbSet property of the class. The DbSet property must be included in the Context class.

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

Typical Context class is as shown below. This is the Context class we used in the Entity Framework Core Console Application example.

```
public class ProductContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDemoDB;User Id=sa;Password=123";

    0 references
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

    5 references
    public DbSet<Product> Products { get; set; }
}
```

We need to create DbSet Property for each & every class, which we want to be part of the model.

EF Core scans all the types, which have a DbSet property and includes them in the model. It also looks for the public properties & types and uses it to determine the columns & types. We can also provide extra modelling information using conventions, data annotation attributes, & Fluent API.

The DBSet Provides methods like Add, Attach, remove, etc… on the Entity Types. The Context class maps these operations into a SQL query and runs it against the database using the Database Providers.

DbSet Implements the IQueryable & IEnumerable interfaces. This allows us to query the database using the LINQ query.

Using DbSet

The following are few of the examples on how to use DbSet.

Adding Single Record

The Add method adds the given entity to the context in the Added state. The EF Core generates the insert SQL query and updates the database when we call the SaveChanges method.

Syntax

Add(TEntity)

AddAsync(TEntity, CancellationToken)

Example:

```
using (var db = new ProductContext())
{
    Product product = new Product();
    product.Name = "Mouse";

    db.Products.Add(product); //Adding to the Context

    db.SaveChanges(); //Saving Changes to database
}
```

Adding Multiple Records

This AddRange method adds the collection of entities to the context. All the entities in the collection are marked as Added State. All the entities are inserted into the database table when saved by the context (by Calling SaveChanges method)

Note that entities that are already in the context in some other state will have their state set to Added

Syntax

AddRange(IEnumerable<TEntity>)

AddRange(TEntity[])

AddRangeAsync(IEnumerable<TEntity>, CancellationToken)

AddRangeAsync(TEntity[])

Example

```
using (var db = new ProductContext())
{
    List<Product> products = new List<Product>()
    {
        new Product(){Name="Mouse"},
        new Product(){Name="Keyboard"},
        new Product(){Name="Monitor"}
    };

    db.Products.AddRange(products);

    db.SaveChanges();

    Console.WriteLine("Products Added Successfully.");
}
```

Attaching Records

The Attach method attaches the given entity to the context. The records are marked as **Unchanged**. It means calling SaveChanges on the context will have no effect on the database

Attach(TEntity)

AttachRange(IEnumerable<TEntity>)

AttachRange(TEntity[])

Finding an Entity

EF Core Find method finds a record with the given primary key values. If the entity is already in the context (because of a previous query), then the Find method returns it. The Find method sends the query to the database if it does not find it in the context.

Syntax

Find(Object[])

FindAsync(Object[])

FindAsync(Object[], CancellationToken)

Example

```
using (var db = new ProductContext())
{
    Product product = db.Products.Find(1);

    if (product != null)
    {
        Console.WriteLine("Product Found !!!");
    }
}
```

Update a Single Record

To update the entity, first, we find/query and get the entity from the database, make the necessary changes, and then call the **SaveChanges** to persist the changes in the database.

Example

```
using (var db = new ProductContext())
{
    Product product = db.Products.Find(1);

    product.Name = "Web Cam";

    db.SaveChanges();

    Console.WriteLine("Product Details Updated Successfully !!!");
}
```

Deleting a Single Record

Remove method is used to delete the entity from the database. The entity is not deleted immediately. They are marked as deleted. The entity is deleted from the database when SaveChanges is called.

Syntax

Remove(TEntity)

```
using (var db = new ProductContext())
{
    Product product = db.Products.Find(1);

    db.Products.Remove(product);

    db.SaveChanges();

    Console.WriteLine("Product Details Deleted Successfully !!!");
}
```

Entity Framework Core Conventions:

EF Core Conventions or the default rules that you follow while creating the entity model. The EF Core uses these to infer and to configure the Database. It uses the information available in the POCO Classes to determine and infer the schema of the database that these classes are mapped to. For example, the table name, Column Name, Data Type, Primary keys are inferred from the class name, property name & property type by convention to build the database.

Conventions in EF Core

The EF Core makes certain assumptions based on how your code for domain model is written before creating the tables in the database. These are called Entity Framework core conventions or Entity Framework core naming conventions.

Preparing the Example Project

Create a new .NET Core Console application. Name the project as **CoreConsoleApp1**. Install the following packages via NuGet Package Manager.

- install-package Microsoft.EntityFrameworkcore.SqlServer
- install-package Microsoft.EntityFrameworkcore.Tools

Create Models folder

Under the Models folder create the Context class CustomerContext.cs as shown below:

```
using Microsoft.EntityFrameworkCore;

namespace CoreConsoleApp1.Models
{
    public class CustomerContext : DbContext
    {
        private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreCustomerDB;User Id=sa;Password=123";

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

Type Discovery

Every entity model must declare a DbSet property in the DbContext. EF Core builds the database by inspecting the DbSet Property using the reflection. It then creates tables for these types. It also includes the referenced types, which do not expose DbSet Property. Inherited types are also created if the base class exposed the DbSet Property.

Under the Models folder create the Customer class (Customer.cs) as shown below:

```
public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
}
```

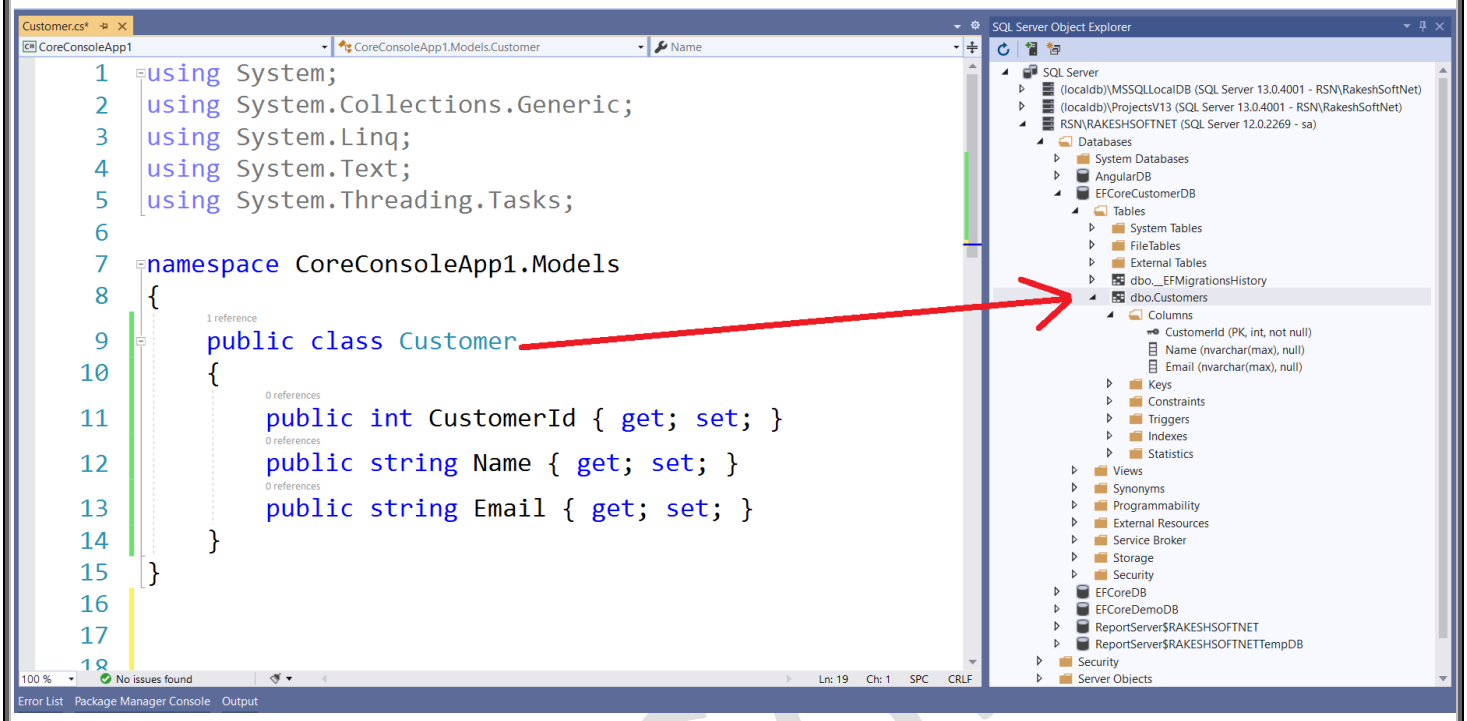
Open the Context class and add the DbSet Property for the Customer Model

```
public DbSet<Customer> Customers { get; set; }
```

Now, open the Package Console Manager and run the command

- add-migration "CustomerDB"
- update-database

Open the database and you should be able to see the following:



The Customer table is created. Note that the table name is taken from the DbSet Property as Customers:

```
public DbSet<Customer> Customers { get; set; }
```

Next, we will add another model class **CustomerAddress (CustomerAddress.cs)** and add the reference of **Customer** as shown below:

```
public class CustomerAddress
{
    public int CustomerAddressId { get; set; }
    public int CustomerId { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public Customer Customer { get; set; }
}
```

Next, we will add the reference of **CustomerAddress** in **Customer** model class as shown below:

```
public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }

    public CustomerAddress CustomerAddress { get; set; }
}
```

Do not add **DbSet** property for the **CustomerAddress** table.

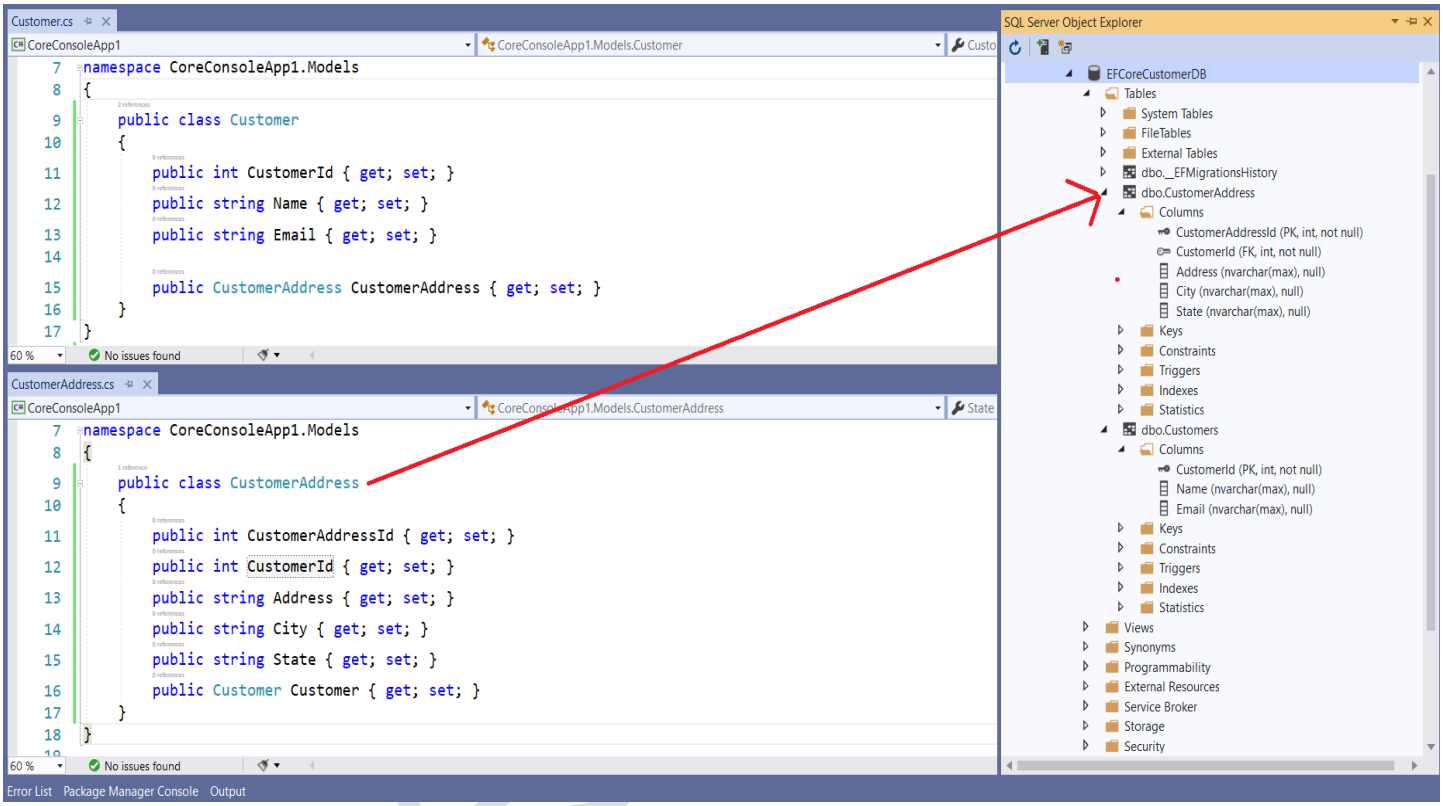
You can remove the migration by running the following commands in **Package Manager Console**.

- update-database 0
- Remove-migration

Then, apply the migration again.

- add-migration "CustomerDB"
- update-database

Check the Database. You will see the **CustomerAddress** table is created. This is because it is referenced by the **Customer**.



Entity Class Convention

As seen in the above example, the entity classes are mapped to the database. For Entity Classes to be mapped to the database, they need to follow few conventions.

- The class must be declared as public
- Static classes are not allowed. The EF must be able to create an instance of the class
- The Class should not have any constructor with a parameter. The parameterless constructor is allowed

Database Schema Name Convention:

The EF Core creates the database using the schema is **dbo**.

Table Name Convention

EF Core creates the tables using the **DbSet** property of the entity class names. In the above example, the table name for Customer class is Customers as it is defined as Customers in DbSet

If The DbSet property is not defined, then the class name is taken as default as in the case of **CustomerAddress** table

Remember, the **old entity framework code first conventions** used to pluralize the table names by default. This is dropped in EF Core.

Column Name

Table Column names are named after the property name. The property must have a public getter. The setter can have any access mode.

Data types and column length

The .NET type are mapped to corresponding SQL Datatype. For Example, the String Properties are mapped to nullable **nvarchar(max)**. The Byte array (byte[]) becomes **varbinary(max)** and Booleans get mapped to the **bit**.

The following is the DbTypeTest model with fields for various data types.

```
public class DbTypeTest
{
    public int id { get; set; }

    public string TestString { get; set; }

    public decimal TestDecimal { get; set; }
    public decimal? TestDecimalNull { get; set; }

    public double TestDouble { get; set; }
    public double? TestDoubleNull { get; set; }

    public int Testint { get; set; }
    public int? TestIntNull { get; set; }

    public bool Testbool { get; set; }
    public bool? TestboolNull { get; set; }

    public DateTime TestDateTime { get; set; }
    public DateTime? TestDateTimeNull { get; set; }

    public byte TestByte { get; set; }
    public byte? TestByteNull { get; set; }

    public byte[] TestByteA { get; set; }

    public uint TestUnit { get; set; }
    public uint? TestUnitNull { get; set; }

    public short TestShort { get; set; }
    public ushort TestUShort { get; set; }

    public char testChar { get; set; }
}
```

The list of Data types and their columns mappings is listed below.

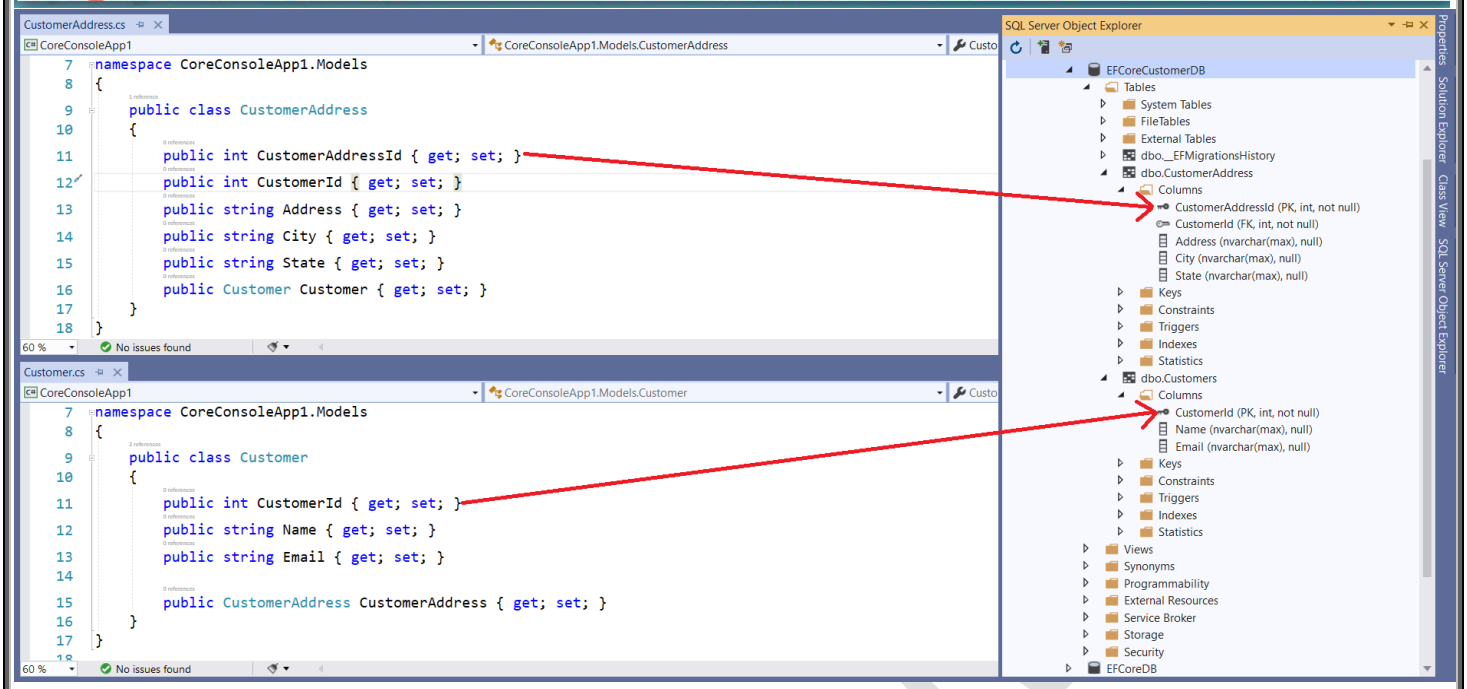
Data type	Mapped	Null?
string	nvarchar(max)	Null
decimal	decimal(18, 2)	Not Null
decimal?	decimal(18, 2)	Null
double	float	Not Null
double?	float	Null
int	int	Not Null
int?	Int	Null
bool	bit	Not Null
bool?	bit	Null
DateTime	datetime	Not Null
DateTime?	datetime	Null
byte[]	varbinary(max)	Null
byte	tinyint	Not Null
byte?	tinyint	Null
uint	bigint	Not Null
uint?	bigint	Null
short	smallint	Not Null
ushort	int	Not Null
char	nvarchar(1)	Not Null

Nullability of Column:

The Nullability of the data type depends on the .NET Data type. If the .NET Data type can be null then the corresponding column is mapped as NULL. The Primary keys, all primitive data types (like int), struct types are mapped to NOT NULL columns. All reference types like string & nullable primitive type properties are mapped as NULL columns

Primary Key Convention:

Entity Framework Core does not allow you to create the tables without the primary key. It follows the convention to identify the candidate for the Primary Key. It searches any property with the name ID or <ClassName>ID and uses it as Primary Key



In the above example, we have not specified the Primary key in the model definition. But the CustomerAddressId column from CustomerAddress table & CustomerId from the Customer table is chosen as primary key based on the convention.

If the model contains both Id & <ClassName>Id columns, then id is chosen as Primary key.

If you do not specify the primary key, then the add-migration will throw an error. For Example, try to remove the CustomerAddressId field from the CustomerAddress class and check. You will get the following error:

"System.InvalidOperationException: The entity type 'CustomerAddress' requires a primary key to be defined."

The EF core will create the identity column if the Primary Key is defined as numeric.

The guid data type in primary key is defined as uniqueidentifier.

The older Entity Framework did not support unsigned data types. The EF Core maps the uint type bigInt datatype.

Foreign Key Convention

In relational databases, data is divided between related tables. The relationship between these tables defined using the foreign keys. If you are looking for an employee working in a particular department, then you need to specify the relationship between employees and department table.

One-to-One Relationship Conventions in Entity Framework Core:

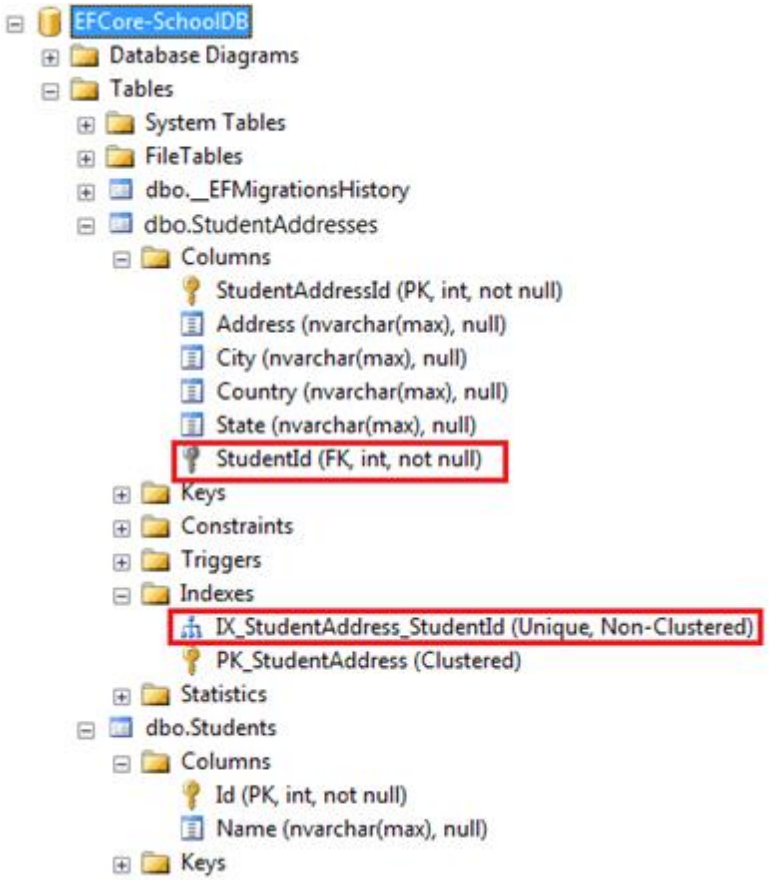
Entity Framework Core introduced default conventions which automatically configure a One-to-One relationship between two entities (EF 6.x or prior does not support conventions for One-to-One relationship).

In EF Core, a one-to-one relationship requires a reference navigation property at both sides. The following Student and StudentAddress entities follow the convention for the one-to-one relationship.

```
public class Student
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }

    0 references
    public StudentAddress Address { get; set; }
}
```

```
public class StudentAddress
{
    0 references
    public int StudentAddressId { get; set; }
    0 references
    public string Address { get; set; }
    0 references
    public string City { get; set; }
    0 references
    public string State { get; set; }
    0 references
    public string Country { get; set; }

    0 references
    public int StudentId { get; set; }
    0 references
    public Student Student { get; set; }
}
```

In the example above, the **Student** entity includes a reference navigation property of type **StudentAddress** and the **StudentAddress** entity includes a foreign key property **StudentId** and its corresponding reference property **Student**. This will result in a one-to-one relationship in corresponding tables **Students** and **StudentAddresses** in the database, as shown below.



EF Core creates a unique index on the **NotNull** foreign key column **StudentId** in the **StudentAddresses** table, as shown above. This ensures that the value of the foreign key column **StudentId** must be unique in the **StudentAddresses** table, which is necessary of a one-to-one relationship.

Note: Unique constraint is supported in Entity Framework Core but not in EF 6 and that's why EF Core includes conventions for one-to-one relationship but not EF 6.x.

Use Fluent API to configure one-to-one relationships if entities do not follow the conventions.

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

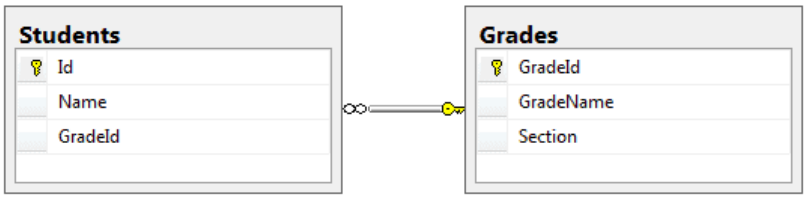
Here, we will learn about the relationship conventions between two entity classes that result in one-to-many relationships between corresponding tables in the database.

Entity Framework Core follows the same convention as Entity Framework 6.x conventions for one-to-many relationship. The only difference is that EF Core creates a foreign key column with the same name as navigation property name and not as **<NavigationPropertyName>_<PrimaryKeyPropertyName>**

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

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }
}
```

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

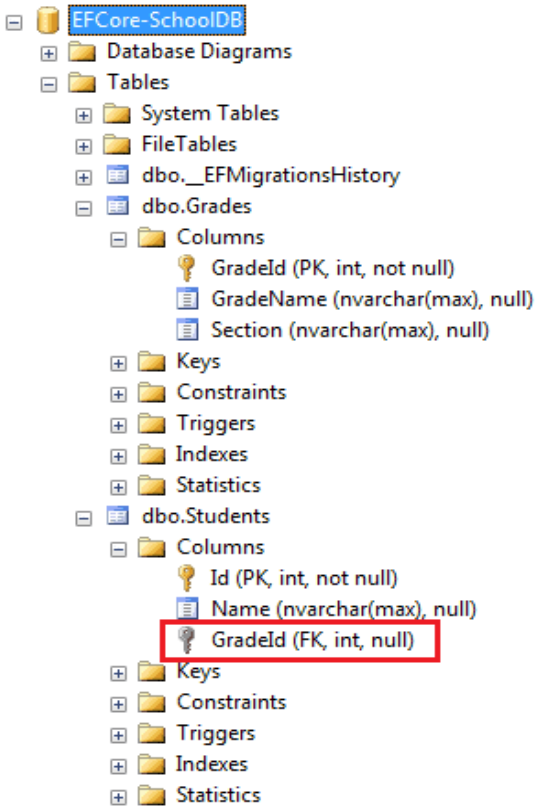


Convention 1

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

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }
}
```

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



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

Convention 2

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

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

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

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

Convention 3

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

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

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

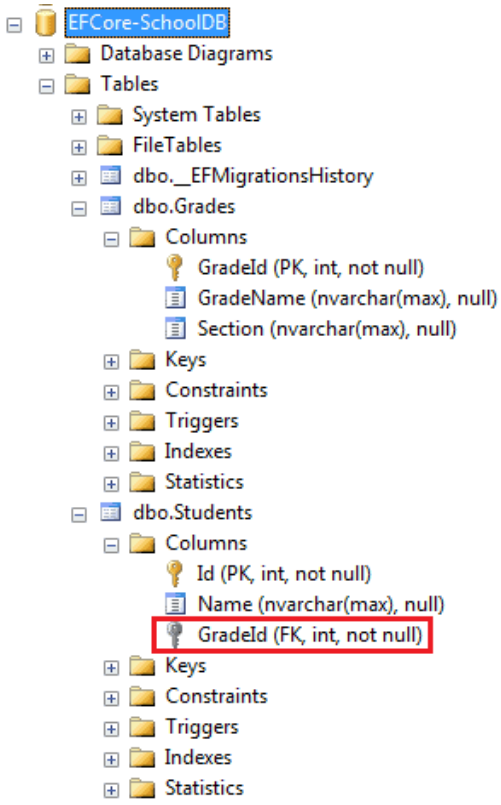
In the example above, the **Student** entity includes a reference navigation property of **Grade** type and the **Grade** entity class includes a collection navigation property **ICollection<Student>**, which results in a one-to-many relationship between corresponding database tables **Students** and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

Convention 4

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

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

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

public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

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

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



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

```
public class Student
{
    public int StudentId { get; set; }
    public string StudentName { get; set; }

    public int? GradeId { get; set; }
    public Grade Grade { get; set; }
}
```

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

Many-to-Many Relationship Conventions in Entity Framework Core:

There are no default conventions available in Entity Framework Core which automatically configure a many-to-many relationship. We must configure it using Fluent API.

Index

EF Core creates a clustered index on Primary Key columns and a non-clustered index on Foreign Key columns, by default.

Conclusion:

The EF Core conventions provide a starting point for configuring the model. You can further fine-tune the model using the Data **Annotations** & **Fluent API**

Configurations in Entity Framework Core

We learned about default Conventions in EF Core in the previous. Many times we want to customize the entity to table mapping and do not want to follow default conventions. EF Core allows us to configure domain classes in order to customize the EF Core model to database mappings. This programming pattern is referred to as "**Convention over Configuration"**.

There are two ways to configure domain classes in EF Core (same as in EF 6).

- By using Data Annotation Attributes
- By using Fluent API

Data Annotations in Entity Framework Core:

Data Annotations allow us to configure the model classes by adding metadata to the class and also the class properties. The Entity Framework Core (EF Core) recognizes these attributes and uses them to configure the models. We have seen how to use Entity Framework Core Convention to configure our models in the previous. The conventions have their limits in their functionalities. Data Annotations in Entity Framework Core allow us to further fine-tune the model. They override the conventions.

What is Data Annotation?

Data Annotations are the attributes that we apply to the class or on the properties of the class. They provide additional metadata about the class or its properties. These attributes are not specific to Entity Framework Core. They are part of a larger .NET/.NET Core Framework. The ASP.NET MVC or ASP.NET Core MVC Applications also uses these attributes to validate the model.

This is why these attributes are included in separate namespace **System.ComponentModel.DataAnnotations**.

The Data annotation attributes falls into two groups depending functionality provided by them.

- **Data Modelling Attributes**
- **Validation Related Attributes**

Data Modelling Attributes

Data Modelling Attributes specify the schema of the database. These attributes are present in the namespace **System.ComponentModel.DataAnnotations.Schema**. The following is the list of attributes are present in the namespace.

Attribute	Description
Table	You can specify the name of the table to which the entity class maps to using table attributes
Column	Allows us to specify the name of the column.
ComplexType	This attribute specifies that the class is a complex type.
DatabaseGenerated	We use this attribute to specify that the database automatically updates the value of this property.
ForeignKey	We apply Foreign Key Attribute to a property, which participates as a foreign key in a relationship
InverseProperty	Specifies the inverse of a navigation property. We use this when we have multiple relationships between two entities. Apply it on a property which is at the other end of the relationship
NotMapped	Specifies the property, which you do not want to be in the database table. The Entity Framework will not create a column for this property.
Index	Specify this attribute on a property to indicate that this property should have an index in the database

Validation Related Attributes

Validation related attributes reside in the **System.ComponentModel.DataAnnotations** namespace. We use these attributes to enforce validation rules for the entity properties. These attributes also define the size, Nullability & Primary Key etc…

Attribute	Description
ConcurrencyCheck	Apply this attribute to a property, which participates in concurrency check validation while updating or deleting an entity
Key	Specify the properties, that are part of the primary key
MaxLength	This validation attribute specifies the max length of the column in the database
MinLength	This validation attribute specifies the minimum length of the data allowed in a string or array property.
Required	Specifies that a data field value is required. Specify the column as non-nullable
StringLength	Specifies the minimum and maximum length of characters that are allowed in a data field. This attribute is similar to MaxLength & MinLength attribute
Timestamp	Specifies the data type of the column as a row version.

Conclusion:

Entity Framework Core allows us various ways to configure the model class. Using Conventions, Using data annotation attributes, or by using fluent API. Here, we looked at Data Annotation attributes. Now, we will look at each of the above data annotations attributes and how to use them with examples.

Data Annotations Table Attribute in Entity Framework Core:

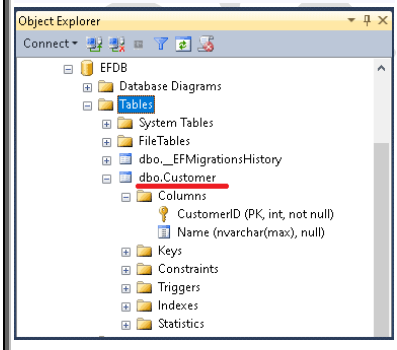
The **Table** attribute can be applied to a class to configure the corresponding table name in the database.

Table Name Convention

The Default EF Core conventions use the DbSet Property name as the table name or name of the class if DbSet property is not available.

Consider the following model.

```
public class Customer
{
  public int CustomerID { get; set; }
  public string Name { get; set; }
}

//Add this in Context class
public DbSet<Customer> Customer { get; set; }
```



The above entity model creates the table with the name Customer in EF Core as shown in the image above.

You can override this behavior using the **Table** attribute.

This attribute resides in **System.ComponentModel.DataAnnotations.Schema** namespace.

Table Attribute

Table attribute overrides the default EF Core Conventions, which creates the table with the name same as the Class Name. We apply the attribute to the class and not on the Property. The EF Core will create a table with the name specified in this attribute.

Syntax

```
Table(string name, Properties:[Schema = string])

Where
Name:  Sets the name of the table the class is mapped to.
Schema: Sets the schema of the table the class is mapped to.
```

Table Name

In the following example, we have updated our domain model **Customer** and applied the table attribute **CustomerMaster**

```
//Import required
using System.ComponentModel.DataAnnotations.Schema;

[Table("CustomerMaster")]
public class Customer
{
  public int CustomerID { get; set; }
  public string Name { get; set; }
}

//Add this in Context class
public DbSet<Customer> Customer { get; set; }
```

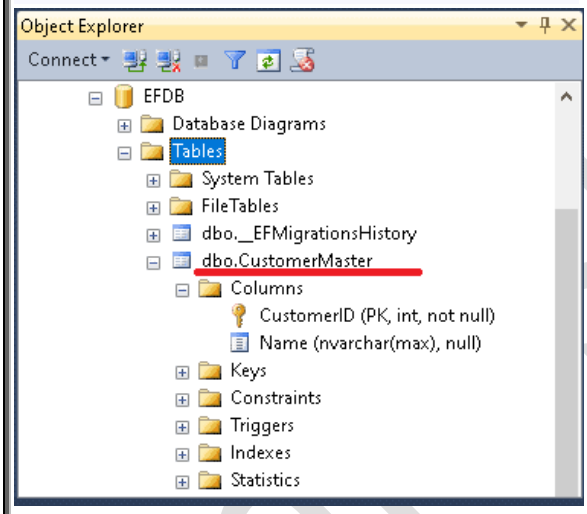
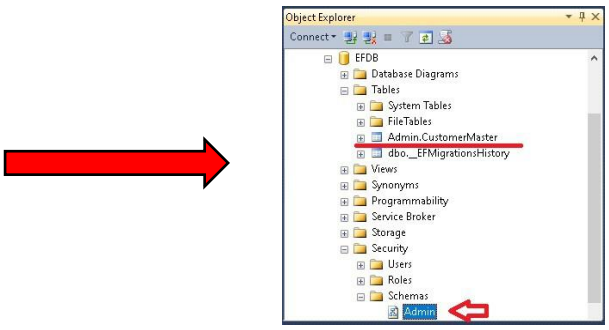


Table Schema

The above example creates the table **CustomerMaster** under the schema **dbo**. You can also specify a table schema using the Schema property as shown below. The example below creates the **CustomerMaster** table with the schema **Admin**

```
//Import required
using System.ComponentModel.DataAnnotations.Schema;

[Table("CustomerMaster", Schema = "Admin")]
public class Customer
{
  public int CustomerID { get; set; }
  public string Name { get; set; }
}

//Add this in Context class
public DbSet<Customer> Customer { get; set; }
```



The dbo is now changed to Admin. Also note that EF Core creates the Schema if does not exists.

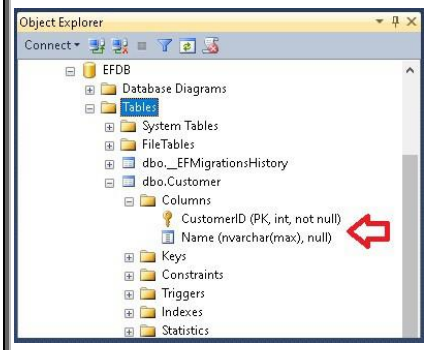
Data Annotation Column Attribute in EF Core:

The Column attribute can be applied to one or more properties in an entity class to configure the corresponding column name, data type and order in a database table.

Column Convention

Consider the following model. The default convention names the database fields after the property name. The Data type of property determines the data type of the field. The order of the fields in tables follows the order in which properties defined in the model.

```
public class Customer
{
   public int CustomerID { get; set; }
   public string Name { get; set; }
}
```



The following table shows data type to column type mapping. Since Name is a **string** so it is mapped to **nvarchar(max)**.

Data type	Mapped	Null ?
string	nvarchar(max)	Null
decimal	decimal(18, 2)	Not Null
decimal?	decimal(18, 2)	Null
double	float	Not Null
double?	float	Null
int	int	Not Null
int?	int	Null
bool	bit	Not Null
bool?	bit	Null
DateTime	datetime	Not Null
DateTime?	datetime	Null
byte[]	varbinary(max)	Null
byte	tinyint	Not Null
byte?	tinyint	Null
uint	biint	not null
uint?	biint ?	null
short	smallint	not null
ushort	int	not null
char	nvarchar(1)	not null

Column Attribute

By Applying the **Column** attribute, we can change the column name, datatype, and order of the column. The attribute takes the following argument

```
[Column (string name, Properties:[Order = int],[TypeName = string])
```

Where

Name: Name of the database column.

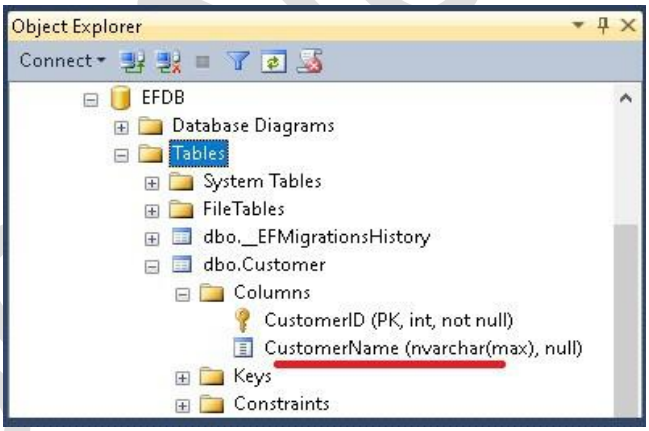
Order: Sets the zero-based Order of the field in the table. (Optional)

TypeName: Database Provider-specific data type of the column the property. (Optional)

Column Name

Column attribute used to specify the column names. EF by convention creates the columns using the property name. If the **Column** attribute is present, then the EF will use the attribute value as the column name when creating the table

```
public class Customer
{
    public int CustomerID { get; set; }

    [Column("CustomerName")]
    public string Name { get; set; }
}
```

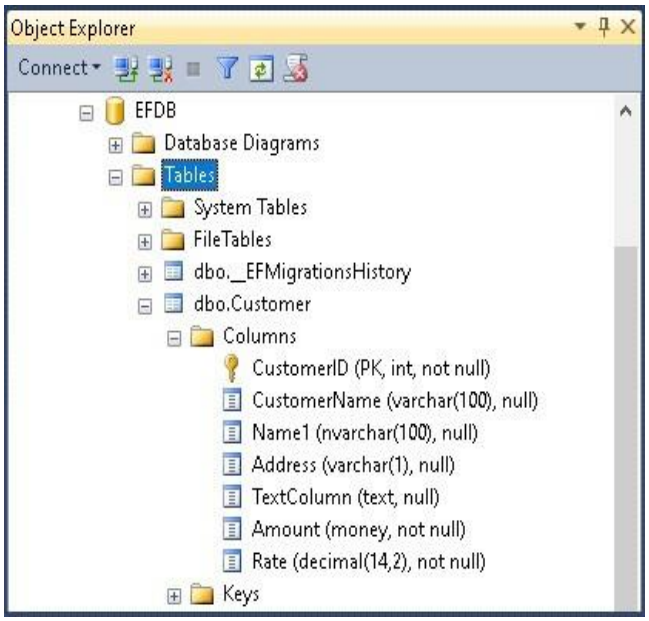


Note that **CustomerName** has field length set as **nvarchar(max)**

Data type

Use the **TypeName** option to set the data type of the column. In the following example, **CustomerName** is set as the **varchar(100)** datatype. **Name1** is set as the **nvarchar(100)** datatype etc.

```
public class Customer
{
    public int CustomerID { get; set; }

    [Column("CustomerName", TypeName = "varchar(100)")]
    public string Name { get; set; }

    [Column(TypeName = "nvarchar(100)")]
    public string Name1 { get; set; }

    [Column(TypeName = "varchar")]
    public string Address { get; set; }

    [Column(TypeName = "text")]
    public string TextColumn { get; set; }

    [Column(TypeName = "money")]
    public decimal Amount { get; set; }

    [Column(TypeName = "decimal(14,2)")]
    public decimal Rate { get; set; }
}
```



TypeName attribute is different from the **DataType** data annotation attribute, which is used only for the UI Validations. Note that **TypeName = "varchar"** converts to **varchar(1)** column

Column Order

Specify the order of the column using the Order option. The Order can be any integer value. The EF Core will create the column based on the Order specified. The Unordered column appears after the ordered column.

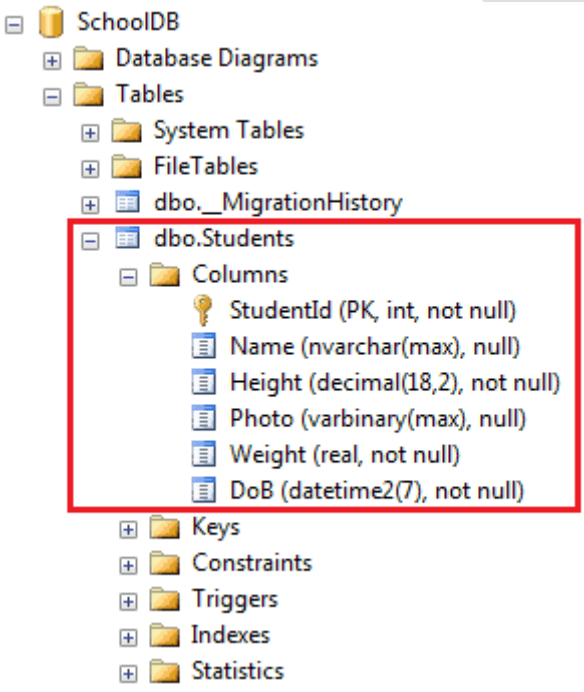
Use the zero-based Order parameter to set the order of columns in the database. As per the default convention, PK columns will come first and then the rest of the columns based on the order of their corresponding properties in an entity class.

Note: The Order parameter must be applied on all the properties with a different index, starting from zero.

This feature was introduced in EF Core 6.0.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    [Column(Order = 0)]
    public int StudentID { get; set; }

    [Column("Name", Order = 1)]
    public string StudentName { get; set; }

    [Column("DoB", Order = 5)]
    public DateTime DateOfBirth { get; set; }
    [Column(Order = 3)]
    public byte[] Photo { get; set; }
    [Column(Order = 2)]
    public decimal Height { get; set; }
    [Column(Order = 4)]
    public float Weight { get; set; }
}
```

The above example will create the columns in the specified order as shown below.



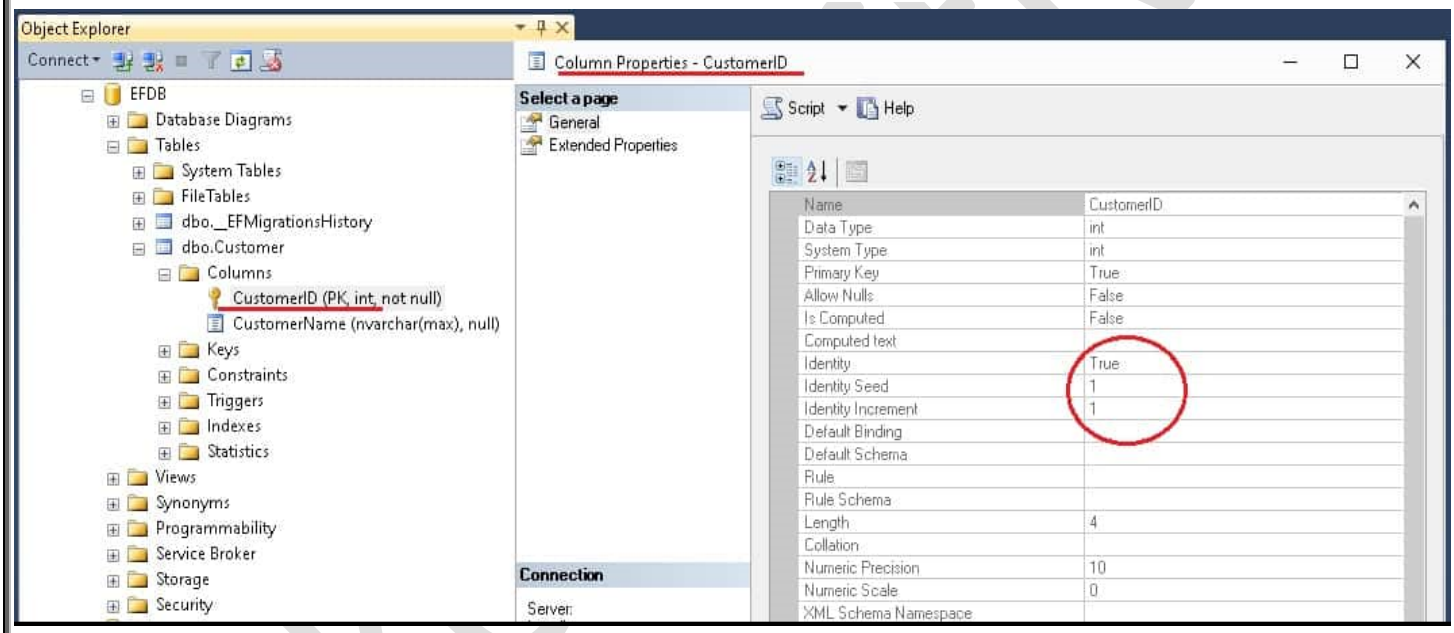
Data Annotations Key Attribute in EF Core:

The Key attribute can be applied to a property in an entity class to make it a key property and the corresponding column to a Primary Key column in the database.

Default Convention

The Entity Framework Core Convention look for the property with the name Id or with the name <Entity Class Name>Id. It then maps that property to the Primary Key. In case it finds both Id property and <Entity Class Name>Id property, then Id property is used. The following model creates the table with CustomerID As the primary key.

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    0 references
    public string CustomerName { get; set; }
}
```

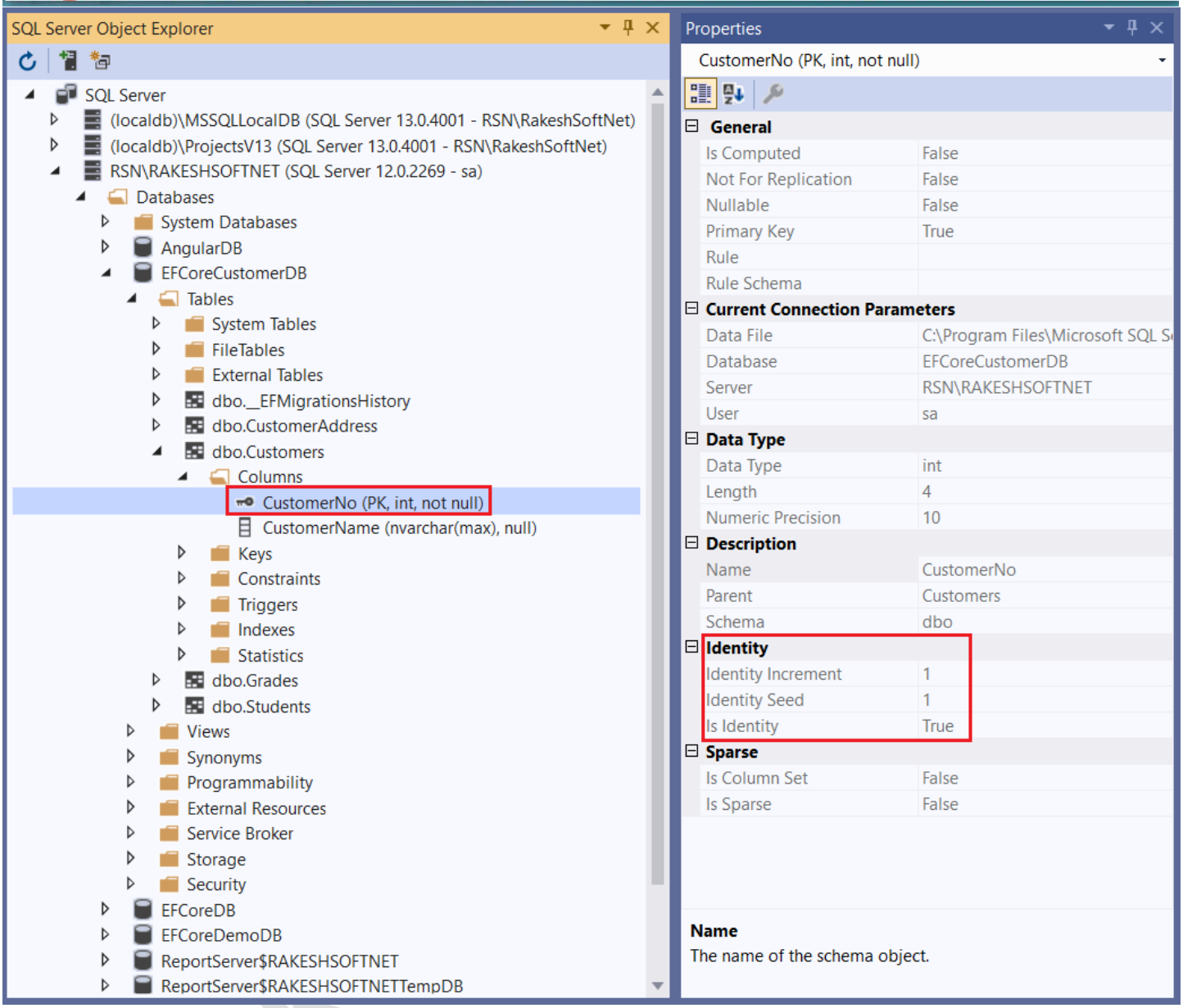


Key Attribute

You can override this behavior using the Data Annotation Key attribute. The Key attribute marks the property as the Primary Key. This will override the default Primary Key. The following code creates the table with **CustomerNo** as the Primary key instead if CustomerID

Do not forget to import the following namespace

```
using System.ComponentModel.DataAnnotations;
public class Customer
{
    [Key]
    0 references
    public int CustomerNo { get; set; }
    0 references
    public string CustomerName { get; set; }
}
```



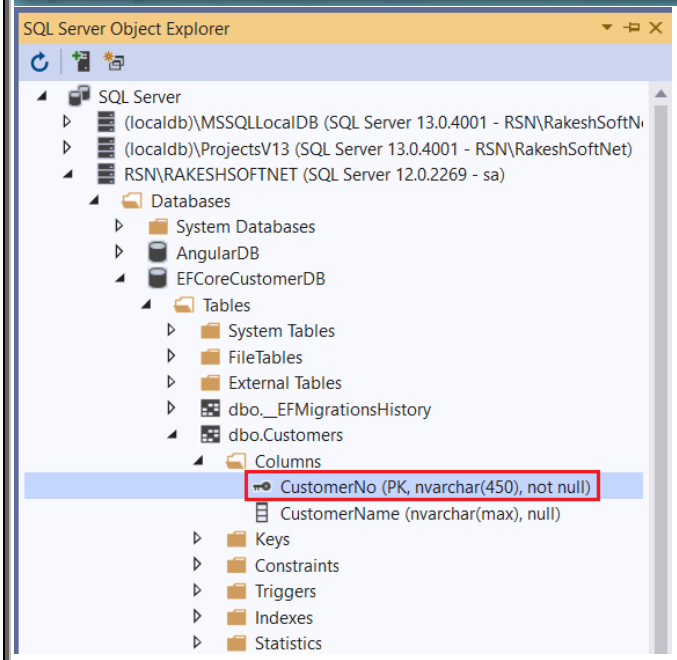
Note that the **CustomerNo** column created with the **Identity** enabled. Key attribute, if applied on the integer column will create the column with the **Identity** enabled (auto increment).

The **unsigned data types** (**uint**) is allowed in EF Core. It is mapped to **bigInt**. Also, EF Core does not create an **Identity** column for unsigned data types

Short data type is mapped to **SmallInt** with **Identity** column.

Key Attribute on a string property:

```
public class Customer
{
    [Key]
    0 references
    public string CustomerNo { get; set; }
    0 references
    public string CustomerName { get; set; }
}
```

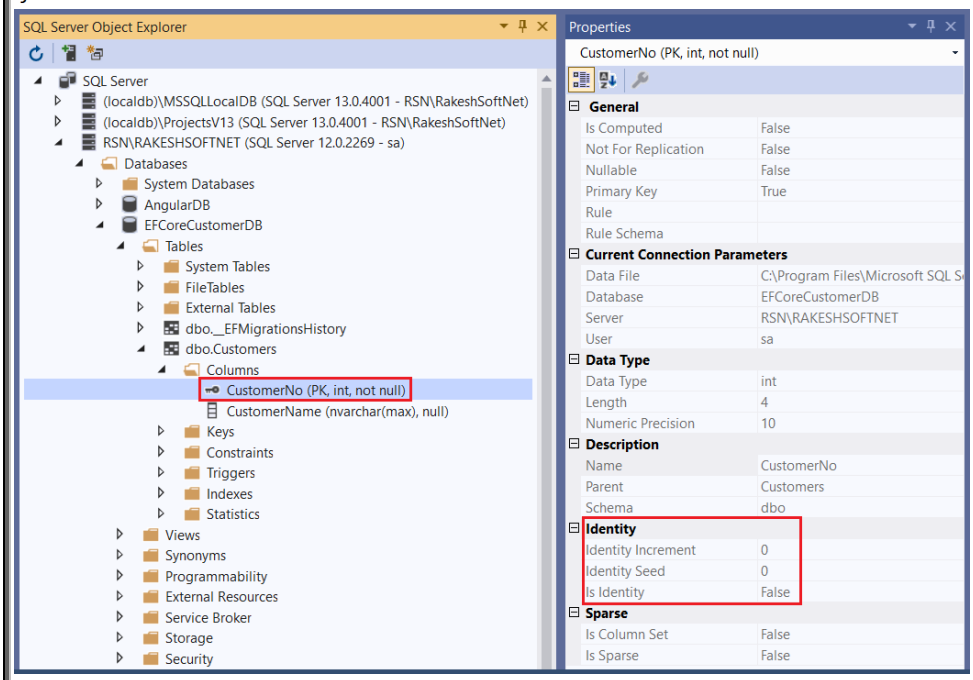


Enabling/Disabling identity column

You can enable/disable the **Identity** on the numeric column by using the **DatabaseGenerated** attribute.

The following code disables the identity column:

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
public class Customer
{
    [Key,DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int CustomerNo { get; set; }
    public string CustomerName { get; set; }
}
```



The following code enables the identity column

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
public class Customer
{
    [Key,DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int CustomerNo { get; set; }
    public string CustomerName { get; set; }
}
```

Composite Primary Key:

You can also configure multiple properties to be the key of an entity - this is known as a composite key.

In EF 6, the **Key** attribute can be applied to multiple properties of an entity class which will create composite primary key columns in the database.

EF Core does not support creating a composite key using the **Key** attribute. You have to use the Fluent API **HasKey()** function in EF Core. So in EF Core, composite key can only be configured using the Fluent API; conventions will never set up a composite key, and you cannot use Data Annotations to configure one.

For Example, the following model will result in an error.

```
public class Customer
{
    [Key]
    public int CustomerId { get; set; }
    public string Name { get; set; }
    [Key]
    public string PanNumber { get; set; }
}
```

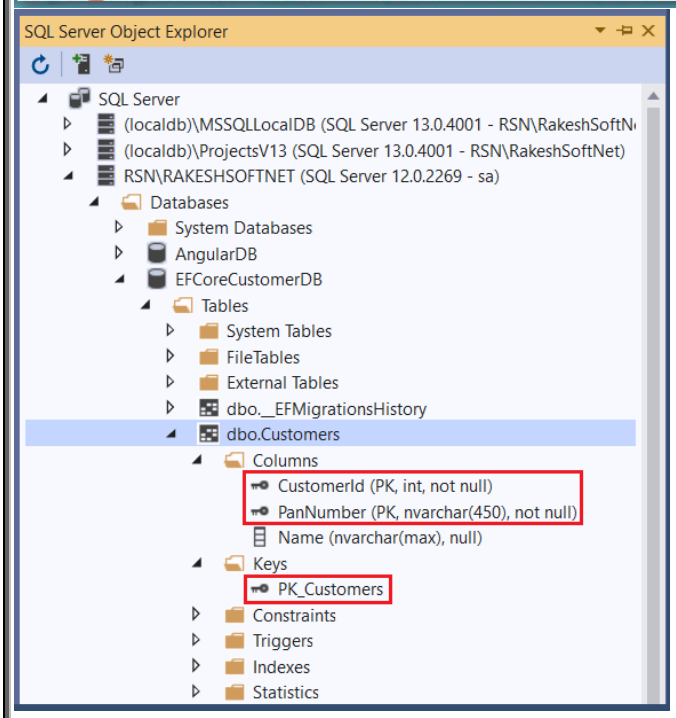
The entity type 'Customer' has multiple properties with the [Key] attribute. Composite primary keys can only be set using 'HasKey' in 'OnModelCreating'.

Solution:

```
public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string PanNumber { get; set; }
}
```

In Context Class:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .HasKey(c => new { c.CustomerId, c.PanNumber });
}
```



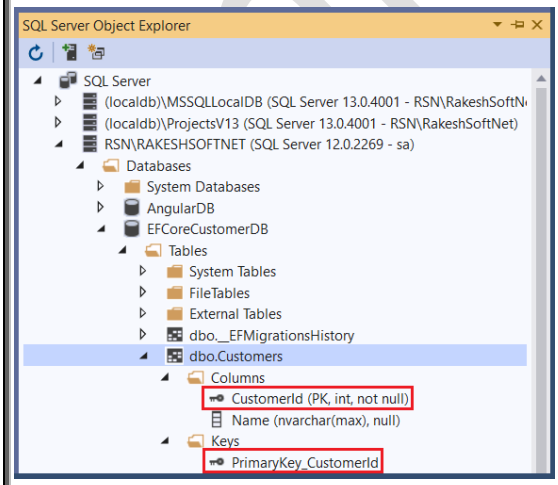
Primary key name

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

```
public class Customer
{
    0 references
    public int CustomerId { get; set; }
    0 references
    public string Name { get; set; }
}
```

In Context Class:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
        .HasKey("CustomerId")
        .HasName("PrimaryKey_CustomerId");
}
```



Data Annotations ComplexType Attribute in EF Core:

The ComplexType attribute denotes that the class is a complex type. A Complex Type is a class that has no primary key defined. Complex Types are not currently implemented in Entity Framework Core.

Data Annotations ConcurrencyCheck in EF Core:

ConcurrencyCheck attribute in EF Core is used to handle conflicts that result when multiple users are updating (or deleting) the table at the same time. So the ConcurrencyCheck attribute is used to specify that a property should be included in a WHERE clause in an UPDATE or DELETE statement as part of concurrency management.

You can add the ConcurrencyCheck attribute on any property, which you want to participate in the Concurrency Check.

What is Concurrency check?

Assume that two users simultaneously query for same data to edit from the Employee Table. One of the users saves his changes. Now, the other user is now looking at the data, which is invalid. If he also modifies the data and saves it, it will overwrite the first user's changes. What if both users save the data at the same time. We never know which data gets saved.

We use the Concurrency check precisely to avoid such situations. To do that we include additional fields in the where clause apart from the primary key. For Example, by including the name field in the where clause, we are ensuring that the value in the name field has not changed since we last queried it. If someone has changed the field, then the where clause fails the Entity Framework raises the exception.

Concurrency Management in Entity Framework Core

Concurrency conflicts occur when one user retrieves an entity's data in order to modify it, and then another user updates the same entity's data before the first user's changes are written to the database. How you handle those conflicts depends on the nature of the changes being made.

Last In Wins

In many cases, there is only one version of the truth, so it doesn't matter if one user's changes overwrite another's changes. In theory, the changes should result in the same update being made to the record. For example, it doesn't matter if two users attempt to update a sports fixture record with the final score. There is no need for any concurrency management strategy in this scenario. This is known as the last in wins approach to concurrency control. In this case, data could easily lead to inconsistency because of some network slowness when previously posted data arrives last.

Pessimistic Concurrency

Pessimistic concurrency involves locking database records to prevent other users being able to access/change them until the lock is released, much like when two users attempt to open the same file on a network share. However, the ability to lock records is not supported by all databases, and can be complex to program as well as highly resource intensive. It is simply not practical at all in disconnected scenarios such as web applications. Entity Framework Core provides no support for pessimistic concurrency control.

Optimistic Concurrency

Optimistic concurrency assumes that the update being made will be accepted, but prior to the change being made in the database, the original values of the record are compared to the existing row in the database and if any changes are detected, a concurrency exception is raised. This is useful in situations where allowing one user's changes to overwrite another's could lead to data loss. This could happen for example, if two users are looking at a customer record, and one of the users adds a missing telephone number. The second user alters the address, but the record that they alter was retrieved before the telephone number was added by the first user. When the second user commits their change (which won't include the telephone number), the first change will be lost. Entity Framework Core provides support for optimistic concurrency management.

Detecting Concurrency Conflicts:

Entity Framework Core supports two approaches to concurrency conflict detection: configuring existing properties as concurrency tokens; and adding an additional "rowversion" property to act as a concurrency token.

Configuring existing properties

Properties can be configured as concurrency tokens via data annotations by applying the **ConcurrencyCheck** attribute:

Customer.cs:

```
public class Customer
{
    0 references
    public int CustomerId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string Gender { get; set; }
    [ConcurrencyCheck]
    1 reference
    public string City { get; set; }
}
```

Alternatively, properties can be configured using the Fluent API **IsConcurrencyToken** method:

Customer.cs:

```
public class Customer
{
    0 references
    public int CustomerId { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public string Gender { get; set; }
    1 reference
    public string City { get; set; }
}
```

CustomerContext.cs:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
    .Property(c => c.City).IsConcurrencyToken();
}
```

Any existing properties that have been configured as concurrency tokens will be included with their original values in the **WHERE** clause of an **UPDATE** or **DELETE** statement. When the SQL command is executed, EF Core expects to find one row that matches the original values. If any of the configured columns have had their values changed between the time that the data was retrieved and the time that the changes are sent to the database, EF Core will throw a **DbUpdateConcurrencyException** with the message:

Database operation expected to affect 1 row(s) but actually affected 0 row(s). Data may have been modified or deleted since entities were loaded.

The concurrency token configuration can be applied to as many non-primary key properties as needed. Care needs to be taken as this approach can lead to very long **WHERE** clauses, or a lot of data being passed into them especially if any of the properties being configured as concurrency tokens are unlimited string values as is illustrated where, where the **Biography** field has been included as a concurrency token.

Adding a RowVersion property

The second approach to concurrency management involves adding a column to the database table to store a version stamp for the row of data. Different database systems approach this requirement in different ways. SQL Server offers the **rowversion** data type for this purpose. The column stores an incrementing number. Each time the data is inserted or modified, the number increments.

User A might retrieve a row of data, followed by User B. The **rowversion** value for the row will be the same for both users. If User B submits changes, the **rowversion** value in the table will increment by 1 for that row. If User A subsequently tries to modify the same record, the **rowversion** value in their **WHERE** clause combined with the primary key value will no longer match an existing row in the database and EF Core will throw a **DbUpdateConcurrencyException**.

A property must be a byte array data type to be mapped to a **rowversion** column. It can be configured to take part in concurrency checking by adding the **Timestamp** data annotations attribute:

Customer.cs:

```
public class Customer
{
    public int CustomerId { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string City { get; set; }
    [Timestamp]
    public byte[] RowVersion { get; set; }
}
```

If you prefer to use the Fluent API to configure the property, you will use the **IsRowVersion** method:

CustomerContext.cs:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
    .Property(c => c.RowVersion).IsRowVersion();
}
```

Note that the **IsRowVersion** method was added in EF Core 1.1. It is a convenience method that simplifies the previous approach which was to combine the **IsConcurrencyToken** method with the **ValueGeneratedOnAddOrUpdate** method:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
    .Property(c => c.RowVersion)
    .IsConcurrencyToken()
    .ValueGeneratedOnAddOrUpdate();
}
```

Either approach results in the same thing - the **RowVersion** column will be configured as a database type that provides automatic row-versioning (e.g. **rowversion** in SQL Server), rather than a varbinary type, which is the default mapping for **byte** array types.

The **SaveChanges** method should be called within a **try-catch** block so that any **DbUpdateConcurrencyException** exceptions can be caught and the appropriate action taken, such as presenting the newly updated record to the user:

```
try
{
    db.SaveChanges();
    //.....
    //.....

}
catch(DbUpdateConcurrencyException ex)
{
    //.....
}
```

In a disconnected scenario such as a web application, you will most likely store the values of any concurrency tokens in hidden fields if they are not included as regular form fields. This is to ensure that they are available as parameter values to the **WHERE** clause of any **UPDATE** or **DELETE** statement.

Data Annotations DatabaseGenerated Attribute in EF Core:

Use EF Core **DatabaseGenerated** Attribute on a property whose value is automatically generated by the Database. This attribute is part of the **System.ComponentModel.DataAnnotations.Schema** Namespace.

DatabaseGenerated Attribute

The **DatabaseGenerated** attribute specifies how values are generated for a property by the database. The attribute takes a **DatabaseGeneratedOption** enumeration value, which can be one of three values:

1. Computed - The database generates a value for the property when a row is inserted or updated.
2. Identity - The database generates a value when a row is inserted.
3. None - Database does not generate a value for the property either in insert or in an update.

Computed

The Computed option specifies that the property's value will be generated by the database when the value is first saved, and subsequently regenerated every time the value is updated. The practical effect of this is that Entity Framework will not include the property in INSERT or UPDATE statements, but will obtain the computed value from the database on retrieval.

Entity Framework Core will not implement a value generation strategy. Database providers differ in the way that values are automatically generated. Some will generate values for selected data types such as Identity, rowversion, GUID. Others may require manual configuration such as setting default values or triggers, or configuring the column as Computed.

This is useful in scenarios where you have computed columns in your database. The Database computes the value of these fields after each insert/update operation. The entity framework will not update these columns. But it will query and return the values of these fields after an insert or update operation.

For Example:

```
public class Customer
{
    0 references
    public int CustomerId { get; set; }
    1 reference
    public string Name { get; set; }
    1 reference
    public string Gender { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    0 references
    public DateTime? CreatedDate { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    0 references
    public DateTime? LastUpdatedDate { get; set; }
}
```

In the above model, we have two computed fields. **CreatedDate** & **LastUpdatedDate**.

Use the following code to insert data into the table.

```
using (var db = new CustomerContext())
{
    Customer customer1 = new Customer()
    {
        Name = "Smith",
        Gender = "Male"
    };

    db.Customers.Add(customer1);

    db.SaveChanges();
}
```

The FE Core generates the following SQL statement. As you can see the EF Core does not insert the values to the CreatedDate & LastUpdatedDate fields, but it tries to retrieve the value of these fields immediately after the insert i.e. because EF Core expects that the database to compute the value of these fields.

```
INSERT INTO [Customers] ([Gender], [Name])
VALUES (@p0, @p1);
SELECT [CustomerId], [CreatedDate], [LastUpdatedDate]
FROM [Customers]
WHERE @@ROWCOUNT = 1 AND [CustomerId] = scope_identity();

',N'@p0 nvarchar(4000),@p1 nvarchar(4000)',@p0=N'Male',@p1=N'Smith'
```

Use the following code to update data into the table.

```
using (var db = new CustomerContext())
{
    Customer customer = db.Customers.Find(1);
    customer.Name = "John Smith";

    db.SaveChanges();
}
```

The FE Core generates the following SQL statement. As you can see the EF Core does not update the values to the Created & LastUpdated fields, but it tries to retrieve the value of these fields immediately after the update i.e. because EF Core expects that the database to compute the value of these fields.

```
UPDATE [Customers] SET [Name] = @p0
WHERE [CustomerId] = @p1;
SELECT [CreatedDate], [LastUpdatedDate]
FROM [Customers]
WHERE @@ROWCOUNT = 1 AND [CustomerId] = @p1;

',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'John Smith'
```

In both, the above examples, the database inserts the null value into the CreatedDate & LastUpdatedDate fields as we are not generating any values for them in the database.

RSN\RAKESHSOFTNE...- dbo.Customers

	CustomerId	Name	Gender	CreatedDate	LastUpdatedDate
	1	Smith	Male	*NULL*	*NULL*

For testing purposes, you can use the following insert trigger to insert values & update trigger to update values and check the result.

OK, let's take a look at the triggers that can accomplish this. We'll need two triggers, and the first one to look at is the trigger that will fire after an INSERT statement.

```
CREATE TRIGGER InsertTrigger
ON dbo.Customers
AFTER INSERT AS
BEGIN
    SET NOCOUNT ON
    update Customers
    set CreatedDate = GETDATE(),
        LastUpdatedDate = NULL
    from Customers
    INNER JOIN inserted on Customers.CustomerId=inserted.CustomerId
END
```

In SQL Server, the pseudo table INSERTED lists all of the 'after' values of the rows that were affected by the DML statement, so in this case, like its name implies, the inserted rows. So what we need to do in order to update these columns is write an UPDATE statement like above where we join our target table to the INSERTED pseudo table in order to get the correct set of rows, and then set the value of CreatedDate to the correct time, which in this case we are using GETDATE() function that returns the current database system date and time, in a 'YYYY-MM-DD hh:mm:ss.mmm' format and set the value of LastUpdatedDate to null. So basically what this does is get the set of affected rows, and go back to the table and update the times to the times that we want. In this way, even if someone supplies a value for CreatedDate and/or LastUpdatedDate, we don't care. We'll make sure the correct datetime value gets put into the column via this trigger.

Now for the trigger that will fire after an UPDATE statement:

```
CREATE TRIGGER UpdateTrigger
ON dbo.Customers
AFTER UPDATE AS
BEGIN
    SET NOCOUNT ON
    IF ( (SELECT trigger_nestlevel() ) > 1 )
        RETURN
    update Customers
    set CreatedDate = deleted.CreatedDate,
        LastUpdatedDate = GETDATE()
    from Customers
    INNER JOIN deleted on Customers.CustomerId=deleted.CustomerId
END
```

The first thing you might notice about this trigger is the IF block wrapped around the SELECT **trigger_nestlevel()** function. This is necessary to prevent this trigger firing as a nested trigger. For example, during an INSERT statement, the **InsertTrigger** defined above will fire, which in turn does an UPDATE statement against the table. Without this code block, the **UpdateTigger** statement would also complete on the update statement fired by the first trigger, and that is not what we want. We only want this trigger to complete for user/application executed UPDATE statements against the table.

Next, we again need to update the **CreatedDate** and **LastUpdatedDate** columns in our table. This time, we'll use the DELETED pseudo table to join to, which includes the old values of all of the rows affected by the UPDATE statement. Joining to the DELETED pseudo table gives us the proper set of rows, and we can pull the existing (old) CreatedDate value out of these rows and use it in our trigger's UPDATE statement as shown above. What this does is keeps someone from updating the value of CreatedDate. As we'll see below, even if someone tries to update the value of CreatedDate, it doesn't matter, because our trigger will insure we retain the existing value in our table.

Let's test everything out and make sure it is working. At this point, table and both of triggers are created. So let's insert a few rows into the table.

```
INSERT INTO Customers (Name, Gender)
    VALUES ('Smith', 'Male');

INSERT INTO Customers (Name, Gender)
    VALUES ('Alina', 'Female');

INSERT INTO Customers (Name, Gender, CreatedDate, LastUpdatedDate)
    VALUES ('David', 'Male', '01/01/2021', '01/01/2010');
```

We see that in the third row, we are supplying a value for CreatedDate and LastUpdatedDate. However, as we are executing this statement, it is most certainly not January 1st, 2021 nor January 1st, 2010. This is exactly the type of situation we are designing for, to make sure we always have correct values in these columns and not allow them to be overridden by user supplied values like this that may potentially be incorrect. So let's see what is on our table at this point.

RSN\RAKESHSOFTNE...- dbo.Customers

	CustomerId	Name	Gender	CreatedDate	LastUpdatedDate
	1	Smith	Male	2021-12-31 02:29:48.4800000	*NULL*
	2	Alina	Female	2021-12-31 02:29:48.4870000	*NULL*
	3	David	Male	2021-12-31 02:29:48.4870000	*NULL*

This is exactly what we want. Even though we did not supply values in the first two INSERT statements, our trigger has automatically populated these columns for use with the correct value. And for the third statement when incorrect values were supplied, these values were not used but again, we have the correct values in these two columns.

Let's run an UPDATE statement now. And to start with, we'll just perform the most basic UPDATE statement, changing the Name of customer whose CustomerId is 1.

```
UPDATE Customers
    SET Name = 'John Smith'
    WHERE CustomerId = 1
```

And now we will check our table data once again.

RSN\RAKESHSOFTNE...- dbo.Customers

	CustomerId	Name	Gender	CreatedDate	LastUpdatedDate
	1	John Smith	Male	2021-12-31 02:29:48.4800000	2021-12-31 02:45:31.0730000
	2	Alina	Female	2021-12-31 02:29:48.4870000	*NULL*
	3	David	Male	2021-12-31 02:29:48.4870000	*NULL*

We see here that for "John Smith", the LastUpdatedDate has been automatically updated by our trigger to the appropriate time for when the row was changed. The CreatedDate record was left unchanged, as were the records for the other rows in the table. So this is exactly what we were looking for.

Finally, lets test the scenario where someone tries to run an UPDATE statement that will modify the CreatedDate column, which as we have discussed, we do not want to allow. So to test this out, we'll run the following statement:

```
UPDATE Customers
    SET Name = 'Alina DSouza',
        CreatedDate = '01/01/2010'
    WHERE CustomerId = 2
```

So we can see here, we are trying to backdate the CreatedDate of the row to a different value. However, with our trigger, we don't allow this to happen and we just retain the existing value. Here is the data:

RSN\RAKESHSOFTNE...- dbo.Customers

	CustomerId	Name	Gender	CreatedDate	LastUpdatedDate
	1	John Smith	Male	2021-12-31 02:29:48.4800000	2021-12-31 02:45:31.0730000
	2	Alina DSouza	Female	2021-12-31 02:29:48.4870000	2021-12-31 02:54:44.2300000
	3	David	Male	2021-12-31 02:29:48.4870000	*NULL*

We see here that row 2 for "Alina DSouza" has a new value for LastUpdatedDate (and this value is correct). But even though a value for CreatedDate was supplied in the UPDATE statement, the supplied value was ignored and the original CreatedDate value retained, which is the correct behavior.

DatabaseGenerated: DatabaseGeneratedOption.Identity

When we apply Identity attribute to a non-key property, Entity Framework Core expects that the database will compute its value when we insert a new row.

If we apply this attribute to the numeric property, the Entity Framework Core will create the identity column in the database. Remember that you can have only one identity column in the database.

The Identity property cannot be updated. Please note that the way the value of the Identity property will be generated by the database depends on the database provider. It can be identity, rowversion or GUID. SQL Server makes an identity column for an integer property.

```
using System.ComponentModel.DataAnnotations;

namespace CoreConsoleApp1.Models
{
    1 reference
    public class Customer
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        0 references
        public int CustomerID { get; set; }

        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        0 references
        public int SrNo { get; set; }
        0 references
        public string Name { get; set; }
        0 references
        public string City { get; set; }
    }
}
```

The above will create the SrNo as Identity column in the database. The entity framework core also retrieves the computed value from the database after the insert.

When you insert the data into the table, the EF Core generates the following queries. Note that the insert query does not contain SrNo field, because EF Core knows that the database generates the value only when inserting the values. Once inserted, the EF Core attempts to read the SrNo from the Database.

```
exec sp_executesql N'SET NOCOUNT ON;
INSERT INTO [Customers] ([CustomerID], [City], [Name])
VALUES (@p0, @p1, @p2);
SELECT [SrNo]
FROM [Customers]
WHERE @@ROWCOUNT = 1 AND [CustomerID] = @p0;

',N'@p0 int,@p1 nvarchar(4000),@p2 nvarchar(4000)',@p0=101,@p1=N'Pune',@p2=N'Smith'
```

While in case of update the EF Core generates the following query:

```
exec sp_executesql N'SET NOCOUNT ON;
UPDATE [Customers] SET [City] = @p0, [Name] = @p1
WHERE [CustomerID] = @p2;
SELECT @@ROWCOUNT;

',N'@p2 int,@p0 nvarchar(4000),@p1 nvarchar(4000)',@p2=101,@p0=N'Mumbai',@p1=N'John Smith'
```

Note: If you attempt to update SrNo field value, EF Core throws an exception i.e.

Microsoft.EntityFrameworkCore.DbUpdateException: 'An error occurred while updating the entries. See the inner exception for details.'

Inner Exception
SqlException: Cannot update identity column 'SrNo'.

DatabaseGenerated: DatabaseGeneratedOption.None

This prevents the database from creating the computed values. The user must provide the value.

This is useful when you want to disable the generation of identity for the integer primary key.

The following domain model Customer, with the DatabaseGeneratedOption.None attribute will create the CustomerID column with identity disabled.

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace CoreConsoleApp1.Models
{
    1 reference
    public class Customer
    {
        [Key]
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        0 references
        public int CustomerID { get; set; }
        0 references
        public string Name { get; set; }
        0 references
        public string City { get; set; }
    }
}
```

In the above example, EF Core will create the CustomerID column in the database and will not mark it as an IDENTITY column. So, each time you will have to provide the value of the CustomerID property before calling the SaveChanges() method.

```
static void Main(string[] args)
{
    using(var context = new CustomerContext())
    {
        Customer customer = new Customer()
        {
            CustomerID = 101,
            Name = "Smith",
            City = "Pune"
        };

        context.Customers.Add(customer);

        context.SaveChanges();
    }
}
```

Note: EF Core will throw an exception if you do not provide unique values each time because CustomerID is a primary key property.

Microsoft.EntityFrameworkCore.DbUpdateException: 'An error occurred while updating the entries. See the inner exception for details.'

Inner Exception
SqlException: Violation of PRIMARY KEY constraint 'PK_Customers'. Cannot insert duplicate key in object 'dbo.Customers'. The duplicate key value is (101).
The statement has been terminated.

ForeignKey Attribute in EF Core

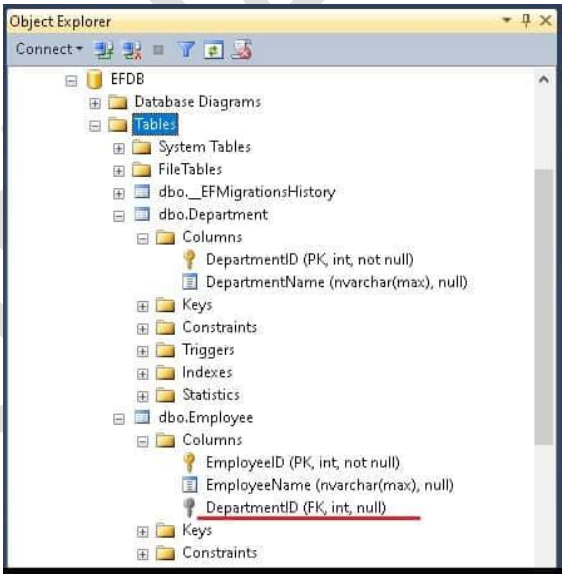
Here we will learn how to use the **ForeignKey** Attribute in Entity Framework Core to configure the Foreign Key Property. We use the Foreign Key to define the relationship between tables in the database. For Example, the Employee working in a Department is a relationship. We express this relationship by creating a DepartmentID field in the Employee table and marking it as Foreign Key. In this example, the Department is the Principal entity & Employee is a Dependent entity.

Foreign Key Default Convention

In the following model, the entity Employee has a Department navigational property that links it to the Department entity. We do not have any Property representing the Foreign Key field in Employee entity. The Entity framework Core automatically creates the Foreign Key field in the database for us.

The Entity Framework Core conventions will use the name of the Primary key field of the Principal class for the Foreign Key field.

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }

    //Navigation property
    public Department Department { get; set; }
}

public class Department
{
    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }

    //Navigation property
    public ICollection<Employee> Employeees { get; set; }
}
```



But, if you add the **DepartmentID** property in the Employee entity, Entity Framework Core conventions will make use of that field and will not create another field.

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }

    public int DepartmentID { get; set; }
    //Navigation property
    public Department Department { get; set; }
}

public class Department
{
    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }

    //Navigation property
    public ICollection<Employee> Employeees { get; set; }
}
```

ForeignKey Attribute

What if we wish to change the **DepartmentID** Property to **DeptID** in the employee table? The Default Convention in EF Core will not recognize the **DeptID** Property and will create **DepartmentID** column in the database.

We can override this behavior using the **ForeignKey** attribute on the navigational property. The following is the syntax of the **ForeignKey** Attribute.

ForeignKey(string name)

Where

Name: The name of the associated navigation property or of the associated foreign keys.

This attribute is from the **System.ComponentModel.DataAnnotations.Schema** namespace.

There are three ways you can apply this attribute

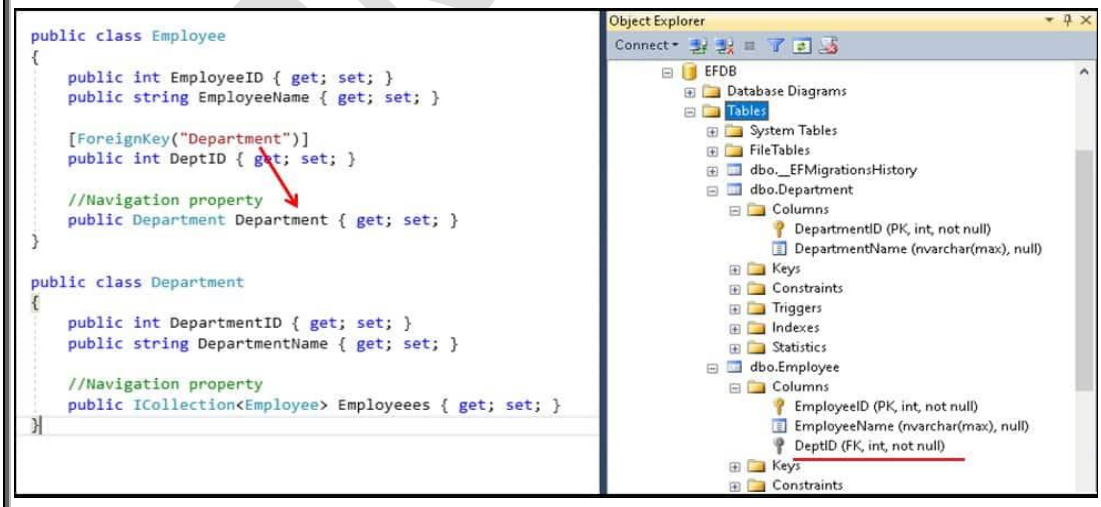
- ForeignKey property of the dependent class
- Navigational Property of the dependent class
- Navigational Property of the Principal class

In the above example, Employee is the dependent class as it depends on the Department. The Department is the Principal class.

Foreign Key property of the dependent class

The following example, we apply the ForeignKey attribute on the **DeptID** Property of the Employee class. In that case, the name must point to the navigational property.

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }

    [ForeignKey("Department")]
    public int DeptID { get; set; }

    //Navigation property
    public Department Department { get; set; }
}

1 reference
public class Department
{
    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }

    //Navigation property
    public ICollection<Employee> Employeees { get; set; }
}
```



Navigational Property of the dependent class

We can also place the ForeignKey Attribute Navigation property. When placed on navigation property, it should specify the associated foreign key

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }

    public int DeptID { get; set; }

    //Navigation property
    [ForeignKey("DeptID")]
    public Department Department { get; set; }
}

1 reference
public class Department
{
    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }

    //Navigation property
    public ICollection<Employee> Employeees { get; set; }
}
```

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }

    public int DeptID { get; set; }

    //Navigation property
    [ForeignKey("DeptID")]
    public Department Department { get; set; }
}

1 reference
public class Department
{
    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }

    //Navigation property
    public ICollection<Employee> Employeees { get; set; }
}
```

Navigational Property of the Principal class

We can also place the ForeignKey attribute on the Navigational property of the Principal class. The name argument must point to the Foreign Key of the dependent class.

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }

    public int DeptID { get; set; }

    //Navigation property
    public Department Department { get; set; }
}

public class Department
{
    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }

    //Navigation property
    [ForeignKey("DeptID")]
    public ICollection<Employee> Employeees { get; set; }
}
```

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string EmployeeName { get; set; }

    public int DeptID { get; set; }

    //Navigation property
    public Department Department { get; set; }
}

public class Department
{
    public int DepartmentID { get; set; }
    public string DepartmentName { get; set; }

    //Navigation property
    [ForeignKey("DeptID")]
    public ICollection<Employee> Employeees { get; set; }
}
```

Foreign Key Attribute on navigational property of the dependent and principal entity

MaxLength/MinLength Attribute in EF Core:

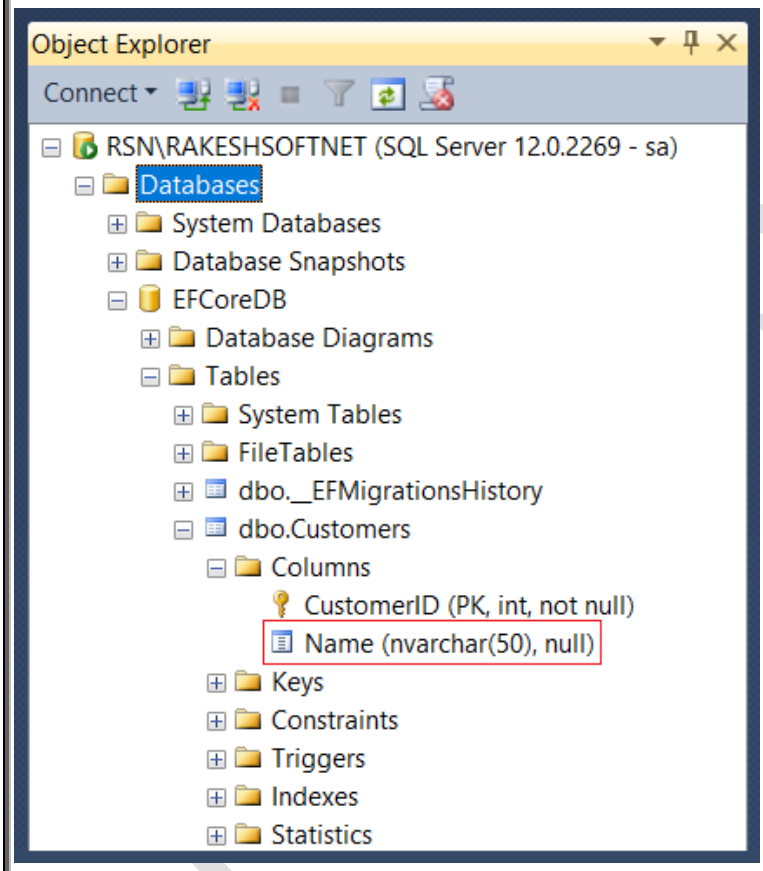
The **MaxLength** and **MinLength** Attributes are available in **System.ComponentModel.DataAnnotations** namespace.

MaxLength Attribute

MaxLength attribute used to set the size of the database column. We can apply this attribute only to the string or array type properties of an entity. The following example shows how to use the MaxLength Attribute.

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    [MaxLength(50)]
    0 references
    public string Name { get; set; }
}
```

In the above example, the **MaxLength(50)** attribute is applied on the **Name** property which specifies that the value of **Name** property cannot be more than 50 characters long. This will create a **Name** *column* with **nvarchar(50)** size in the SQL Server database, as shown below.



Entity Framework Core also validates the value of a property for the MaxLength attribute if you set a value higher than the specified size. For example, if you set more than 50 characters long string value, then Entity Framework Core will throw Microsoft.EntityFrameworkCore.DbUpdateException.

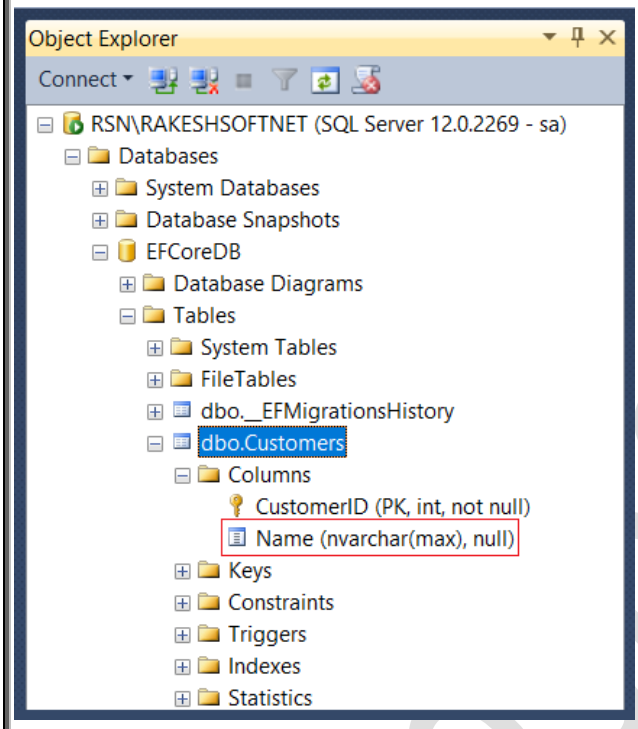
Microsoft.EntityFrameworkCore.DbUpdateException: 'An error occurred while updating the entries. See the inner exception for details.'

Inner Exception
SqlException: String or binary data would be truncated.
The statement has been terminated.

MinLength Attribute

MinLength attribute used to specify the minimum length of string or an array property. It is a *validation attribute* as it does not change the database schema. The following example shows how to use MinLength attribute.

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    [MinLength(5)]
    1 reference
    public string Name { get; set; }
}
```



MaxLength and **MinLength** attribute can be used together as shown below. In this example, **Name** cannot be greater than 10 characters and cannot be less than 2 characters.

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    [MaxLength(10), MinLength(2)]
    1 reference
    public string Name { get; set; }
}
```

You can customize the validation error message displayed to the user using the ErrorMessage parameter of the MinLength attribute as shown below:

```
public class Customer
{
    public int CustomerID { get; set; }
    [MaxLength(10), MinLength(2, ErrorMessage = "Name cannot be less than 2 characters")]
    public string Name { get; set; }
}
```

StringLength Attribute in EF Core

The **StringLength** Attribute allows us to specify the size of the column.

This attribute is available in the **System.ComponentModel.DataAnnotations**.

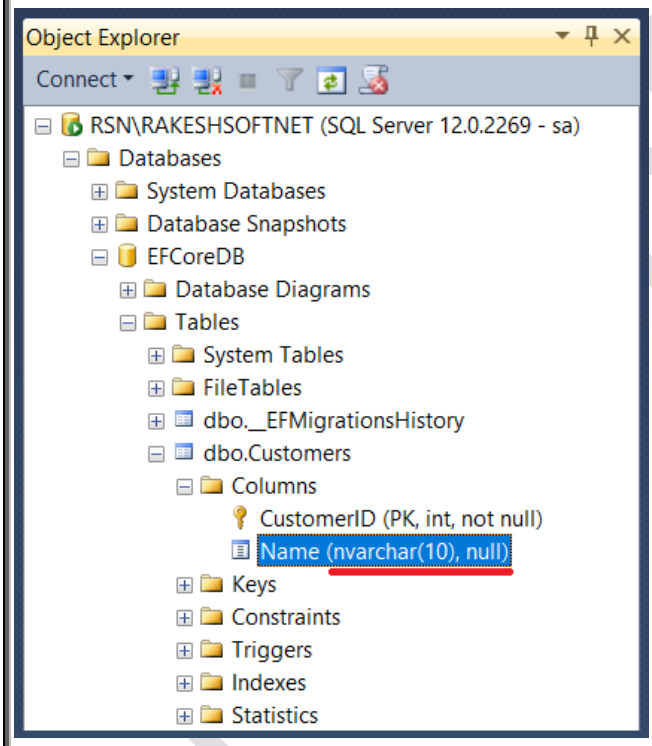
The **StringLength** Attribute specifies both the Min and Max length of characters that we can use in a field. **StringLength** is somewhat similar to **MaxLength** and **MinLength** attributes but operates only on string type properties of an entity class. EF Core will create the column with the MaxLength size set in this attribute. ASP.Net Core uses this attribute to validate the input.

Max Length

We can set the Max Length as shown below. This attribute affects the schema as it creates the database column with max length.

```
public class Customer
{
    public int CustomerID { get; set; }
    [StringLength(10)]
    public string Name { get; set; }
}
```

As you can see in the above code, we have applied the **StringLength** attribute to a **Name** property. So, EF Core will create a Name column with **nvarchar(10)** size in the database, as shown below.



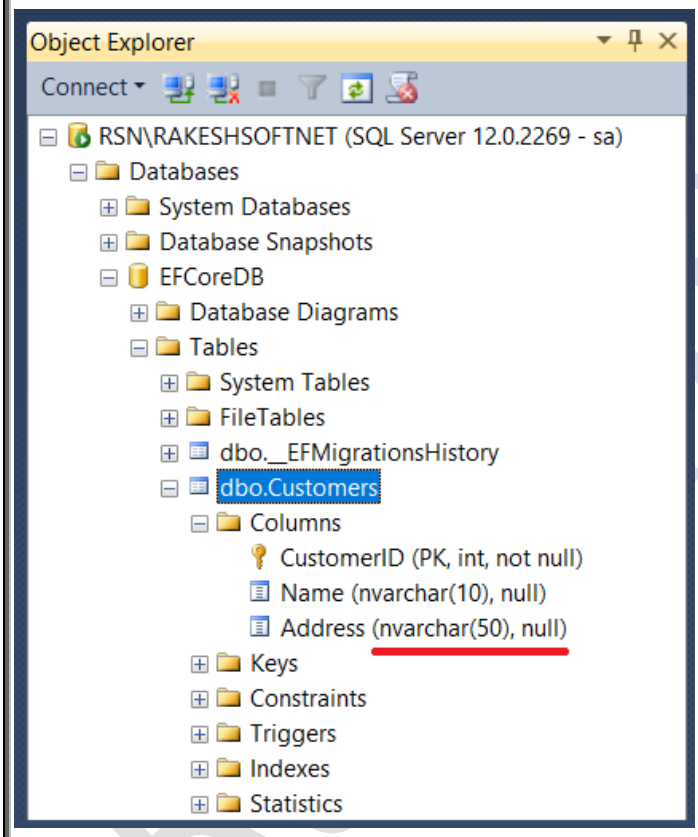
Entity Framework Core also validates the value of a property for the StringLength attribute if you set a value higher than the specified size. For example, if you set the string value to more than 10 characters, then Entity Framework Core will throw Microsoft.EntityFrameworkCore.DbUpdateException.

```
Microsoft.EntityFrameworkCore.DbUpdateException: 'An error occurred while updating the entries. See the inner exception for details.'

Inner Exception
SqlException: String or binary data would be truncated.
The statement has been terminated.
```

Min Length

The **StringLength** attribute also allows you to specify a minimum acceptable length for a string using **MinimumLength** property. So, we can use the **StringLength** Attribute to set both **Max** & **Min** Length of the string as shown below. The **Address** property cannot be greater than 50 and cannot be less than 10. While Max length affects the schema of the database. Whereas the **MinimumLength** property value has no effect on database configuration, but can be used to provide validation.

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    [StringLength(10)]
    1 reference
    public string Name { get; set; }
    [StringLength(50, MinimumLength = 10)]
    0 references
    public string Address { get; set; }
}
```



Validation Message

We can customize the validation messages using the error message parameter as shown below.

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    [StringLength(10)]
    1 reference
    public string Name { get; set; }
    [StringLength(50, MinimumLength = 10, ErrorMessage = "Address must have min length of 10 and max Length of 50")]
    0 references
    public string Address { get; set; }
}
```

NotMapped Attribute in EF Core

The NotMapped attribute can be applied to properties of an entity class for which we do not want to create corresponding columns in the database. By default, EF creates a column for each property (must have get; & set;) in an entity class. The **[NotMapped]** attribute overrides this default convention. You can apply the **[NotMapped]** attribute on one or more properties for which you do NOT want to create a corresponding column in a database table.

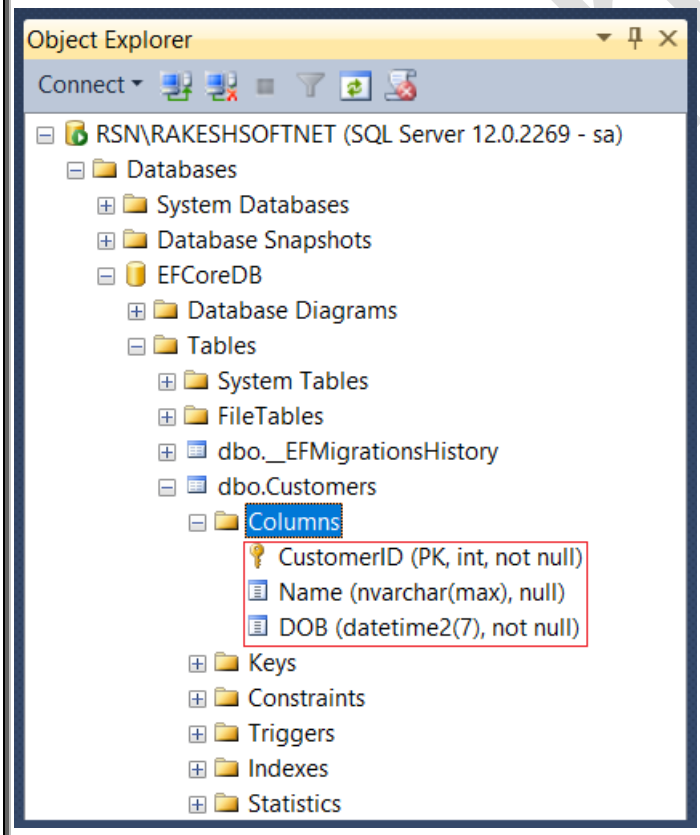
This attribute is from the **System.ComponentModel.DataAnnotations.Schema** namespace

NotMapped Attribute: [NotMapped]

In the following example, we have included age property. We can calculate the age property from the DOB Property. Hence no need to create the database column for the age property

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    1 reference
    public string Name { get; set; }
    0 references
    public DateTime DOB { get; set; }

    [NotMapped]
    0 references
    public int Age { get; set; }

}
```

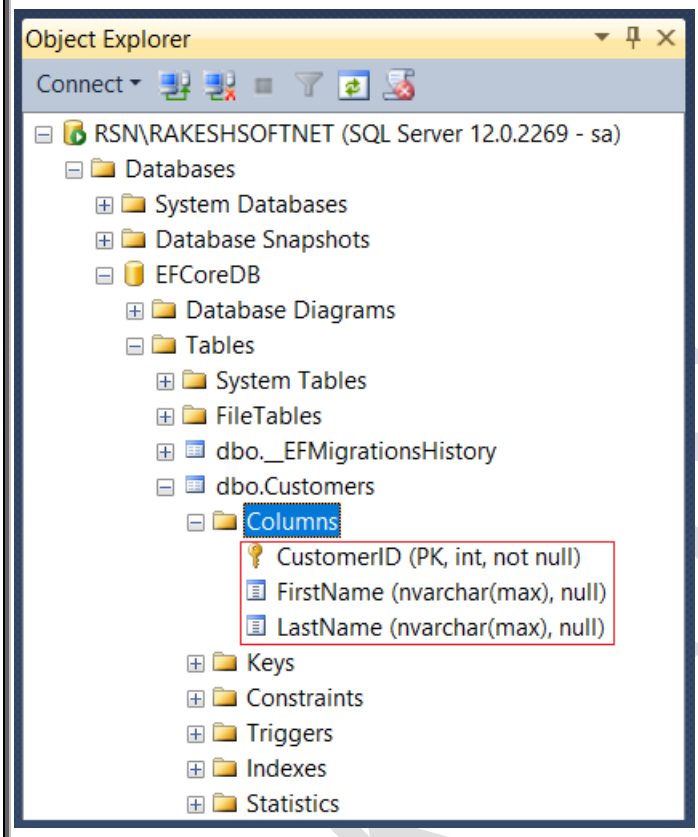
The above model maps to the database as shown below. The EF Core will not create the Age column in the database, because of the **NotMapped** attribute on the property. Without this attribute, the entity framework core will create the column in the database.



Note: EF Core also does not create a column for a property which does not have either getters or setters.

In this example, the **FullName** property will not be mapped to a column in the **Customers** table in the database:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName { get { return $"{FirstName} {LastName}"; } }
}
```



NotMapped Attribute on tables

You can also apply the **NotMapped** attribute on the class itself. This will exclude the model from the database.

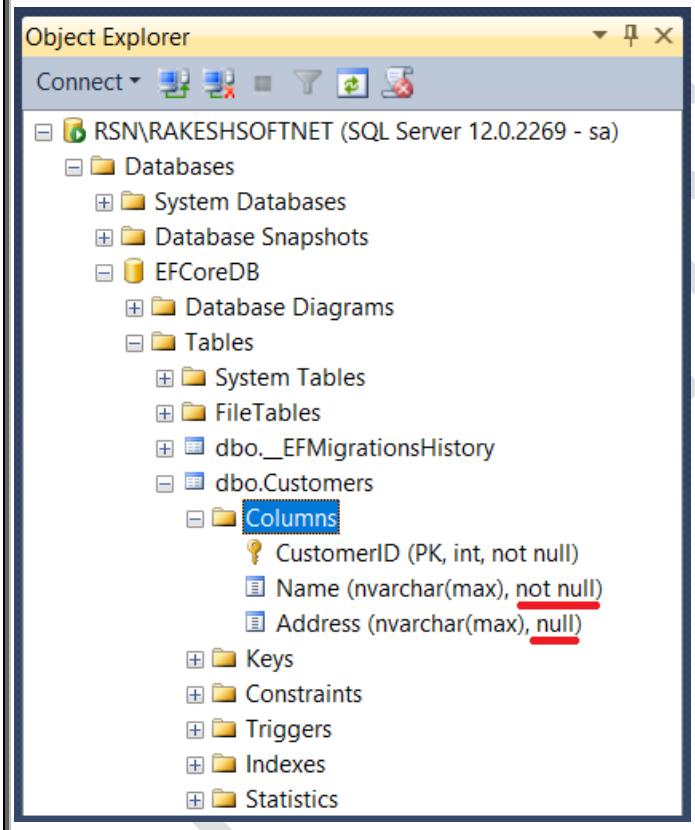
```
[NotMapped]
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
}
```

Required Attribute in EF Core

The Required attribute can be applied to one or more properties in an entity class. EF Core will create a **NOT NULL** column in a database table for a property on which the [**Required]** attribute is applied.

```
public class Customer
{
    0 references
    public int CustomerID { get; set; }
    [Required]
    0 references
    public string Name { get; set; }
    0 references
    public string Address { get; set; }
}
```

In the above example, the Name property is decorated with **[Required]** Attribute. This will create the Name column as **Not Null** in the database. Without the **[Required]** attribute, all string properties are mapped to **NULLABLE** columns (Example **Address** in the above model)



Now, if you try to save the Customer entity without assigning a value to the **Name** property then EF Core will throw the Microsoft.EntityFrameworkCore.DbUpdateException exception.

Microsoft.EntityFrameworkCore.DbUpdateException: 'An error occurred while updating the entries. See the inner exception for details.'

Inner Exception
SqlException: Cannot insert the value NULL into column 'Name', table 'EFCoreDB.dbo.Customers'; column does not allow nulls. INSERT fails.
The statement has been terminated.

Note: The Required attribute can also be used as a validation attribute.

InverseProperty Attribute in EF Core

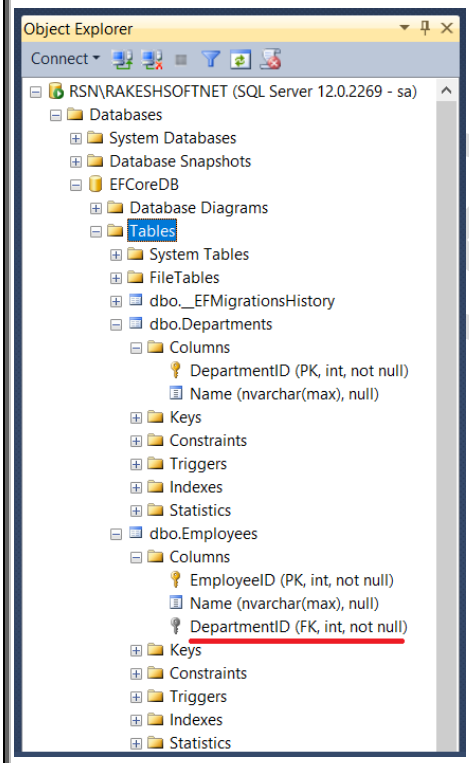
The **InverseProperty** informs the EF Core, which navigational property it relates to on the other end of the relationship. The default convention in EF Core correctly identifies only if there is a single relation between two entities. But in the case of multiple relationships, it fails to correctly identify them. In such cases, we use the InverseProperty to help the EF core correctly identify the relationship.

InverseProperty

A relationship in the Entity Framework Core always has two endpoints. Each end must return a navigational property that maps to the other end of the relationship. The Entity Framework by convention detects this relationship and creates the appropriate Foreign Key column.

Consider the following model

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string Name { get; set; }

    public int DepartmentID { get; set; }

    //Navigation property
    public Department Department { get; set; }
}

1 reference
public class Department
{
    public int DepartmentID { get; set; }
    public string Name { get; set; }

    //Navigation property
    public virtual ICollection<Employee> Employees { get; set; }
}
```



In the above example, we have employee belonging to a particular department. The default convention automatically detects the relationship and creates the DepartmentID foreign key column in the Employees table.

Multiple Relations

The employee and department is a single relationship. What if the employee belongs to multiple departments? Let us take the example of flight & airports. Flight departing from one airport and arrives at another. So the flight has multiple relationships with the Airport

```
public class Flight
{
    public int FlightID { get; set; }
    public string Name { get; set; }
    public Airport DepartureAirport { get; set; }
    public Airport ArrivalAirport { get; set; }
}

2 references
public class Airport
{
    public int AirportID { get; set; }
    public string Name { get; set; }
    public ICollection<Flight> DepartingFlights { get; set; }
    public ICollection<Flight> ArrivingFlights { get; set; }
}
```

The **DepartingFlights** property must map to **DepartureAirport** property in the airport model and **ArrivingFlights** property must map to **ArrivalAirport** property in the airport model.

The EF Core, throws the following error, when we run the migrations

Unable to determine the relationship represented by navigation 'Airport.DepartingFlights' of type 'ICollection<Flight>'. Either manually configure the relationship, or ignore this property using the '[NotMapped]' attribute or by using 'EntityTypeBuilder.Ignore' in 'OnModelCreating'.

Using InverseProperty

We solve this problem by using the **InverseProperty** attribute on any one side of the relationships. Apply **InverseProperty** attribute on that property. Specify the corresponding navigational property of the other end of the relationship as its argument.

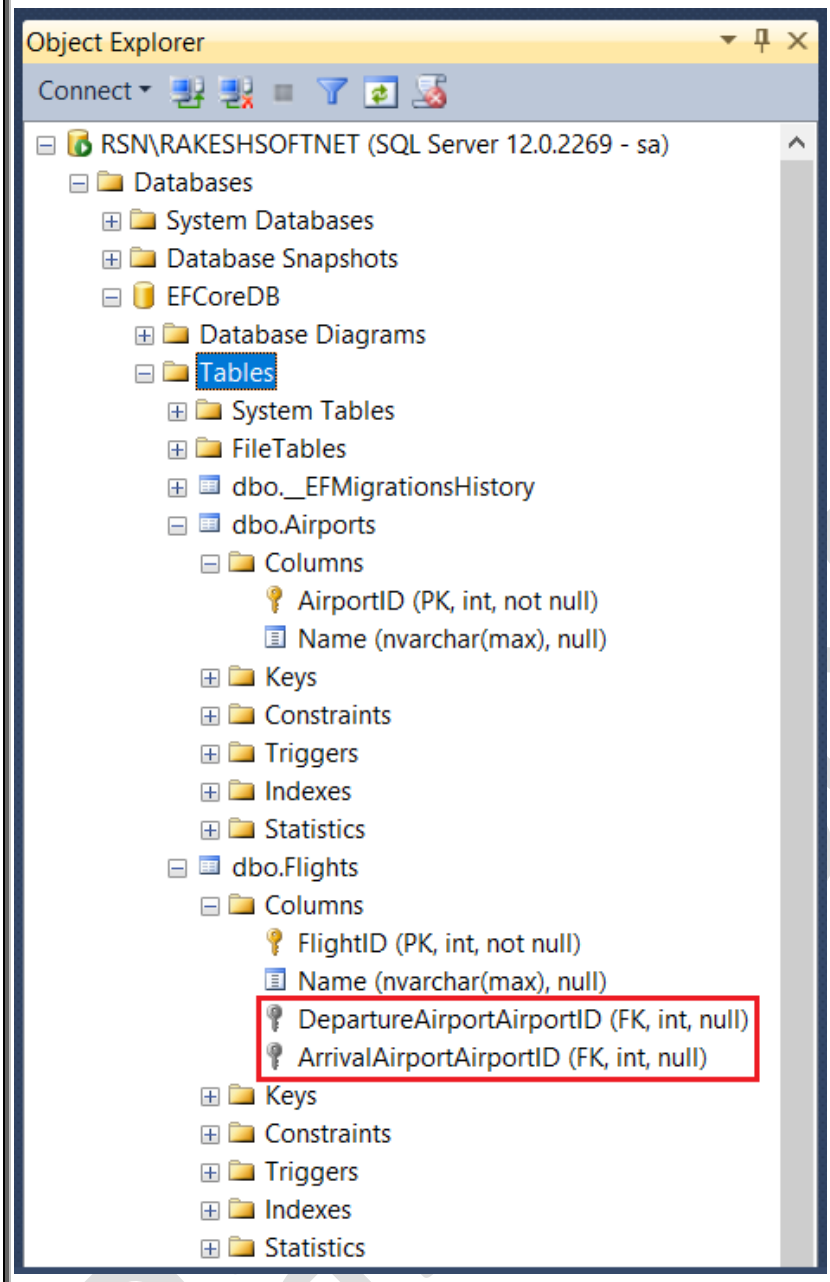
The **Airport** class after applying the InverseProperty is as shown below.

```
public class Airport
{
    public int AirportID { get; set; }
    public string Name { get; set; }

    [InverseProperty("DepartureAirport")]
    public ICollection<Flight> DepartingFlights { get; set; }

    [InverseProperty("ArrivalAirport")]
    public ICollection<Flight> ArrivingFlights { get; set; }
}
```

```
public class Flight
{
    public int FlightID { get; set; }
    public string Name { get; set; }
    public Airport DepartureAirport { get; set; }
    public Airport ArrivalAirport { get; set; }
}
```

```
public class Airport
{
    public int AirportID { get; set; }
    public string Name { get; set; }

    [InverseProperty("DepartureAirport")]
    public ICollection<Flight> DepartingFlights { get; set; }

    [InverseProperty("ArrivalAirport")]
    public ICollection<Flight> ArrivingFlights { get; set; }
}
```

Now the EF Core identifies the relationship correctly and creates only two fields as shown below.



You can apply InverseProperty on the Flight instead of the airport as shown below. Both will result in the same output.

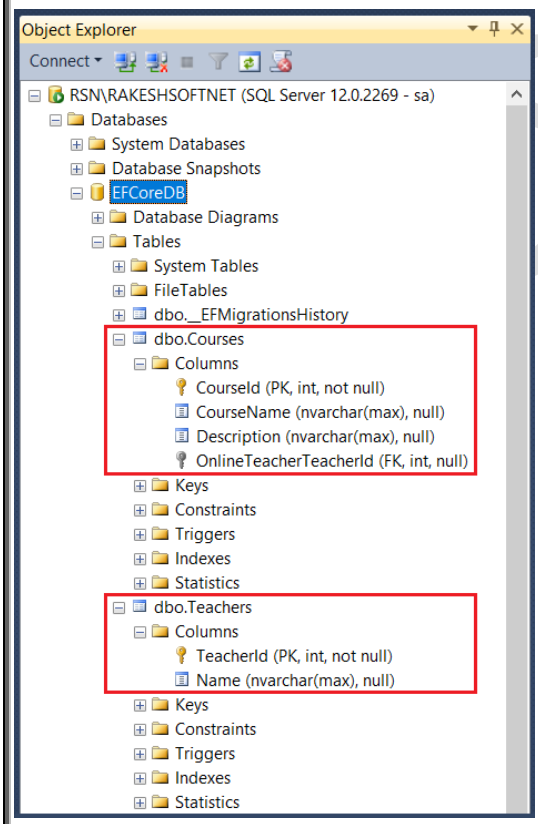
```
public class Flight
{
    public int FlightID { get; set; }
    public string Name { get; set; }
    [InverseProperty("DepartingFlights")]
    public Airport DepartureAirport { get; set; }
    [InverseProperty("ArrivingFlights")]
    public Airport ArrivalAirport { get; set; }
}
```

```
public class Airport
{
    public int AirportID { get; set; }
    public string Name { get; set; }
    public ICollection<Flight> DepartingFlights { get; set; }
    public ICollection<Flight> ArrivingFlights { get; set; }
}
```

Consider the following one more example of **Course** and **Teacher** entities.

```
public class Course
{
    0 references
    public int CourseId { get; set; }
    0 references
    public string CourseName { get; set; }
    0 references
    public string Description { get; set; }

    0 references
    public Teacher OnlineTeacher { get; set; }
}

public class Teacher
{
    0 references
    public int TeacherId { get; set; }
    0 references
    public string Name { get; set; }

    0 references
    public ICollection<Course> OnlineCourses { get; set; }
}
```

In the above example, the Course and Teacher entities have a one-to-many relationship where one teacher can teach many different online courses. As per the default conventions in EF Core, the above example would create the following tables in the database.



Now, suppose we add another one-to-many relationship between the Teacher and Course entities as below.

```
public class Course
{
    0 references
    public int CourseId { get; set; }
    0 references
    public string CourseName { get; set; }
    0 references
    public string Description { get; set; }

    0 references
    public Teacher OnlineTeacher { get; set; }
    0 references
    public Teacher ClassRoomTeacher { get; set; }
}

public class Teacher
{
    0 references
    public int TeacherId { get; set; }
    0 references
    public string Name { get; set; }

    0 references
    public ICollection<Course> OnlineCourses { get; set; }
    0 references
    public ICollection<Course> ClassRoomCourses { get; set; }
}
```

In the above example, the Course and Teacher entities have two one-to-many relationships. A Course can be taught by an online teacher as well as a class-room teacher. In the same way, a Teacher can teach multiple online courses as well as class room courses.

Here, EF Core API cannot determine the other end of the relationship. It will throw the following exception for the above example during adding migration.

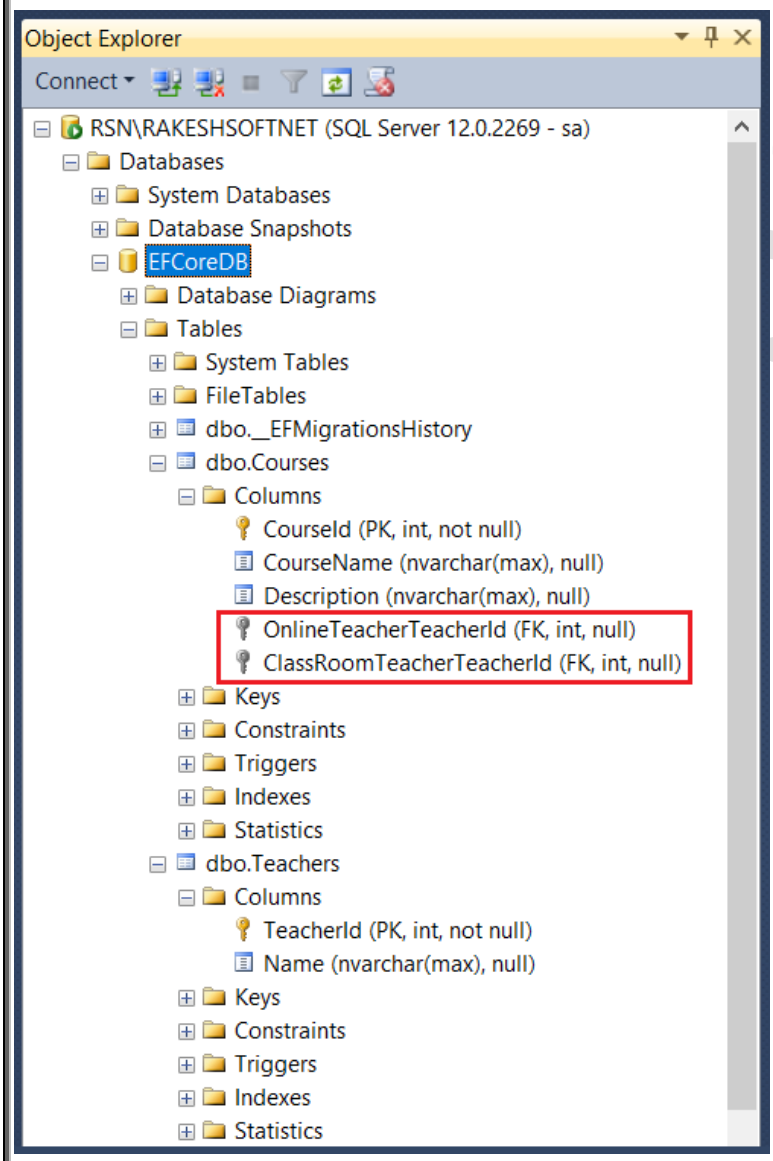
Unable to determine the relationship represented by navigation 'Course.OnlineTeacher' of type 'Teacher'. Either manually configure the relationship, or ignore this property using the '[NotMapped]' attribute or by using 'EntityTypeBuilder.Ignore' in 'OnModelCreating'.

To solve this issue, use the **[InverseProperty]** attribute in the above example to configure the other end of the relationship as shown below.

```
public class Course
{
    0 references
    public int CourseId { get; set; }
    0 references
    public string CourseName { get; set; }
    0 references
    public string Description { get; set; }

    0 references
    public Teacher OnlineTeacher { get; set; }
    0 references
    public Teacher ClassRoomTeacher { get; set; }
}
```

```
public class Teacher
{
    0 references
    public int TeacherId { get; set; }
    0 references
    public string Name { get; set; }

    [InverseProperty("OnlineTeacher")]
    0 references
    public ICollection<Course> OnlineCourses { get; set; }
    [InverseProperty("ClassRoomTeacher")]
    0 references
    public ICollection<Course> ClassRoomCourses { get; set; }
}
```

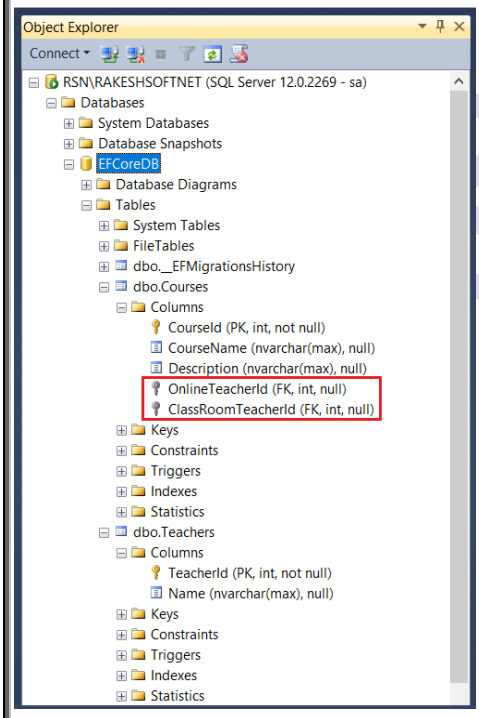
In the above example, the **[InverseProperty]** attribute is applied on two collection navigation properties **OnlineCourses** and **ClassRoomCourses** to specify their related navigation property in the **Course** entity. So now, EF Core will be able to figure out corresponding foreign key names. EF Core creates **OnlineTeacherTeacherId** and **ClassRoomTeacherTeacherId** as shown below.



You can use the **[ForeignKey]** attribute to configure the foreign key name as shown below.

```
public class Course
{
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public string Description { get; set; }

    [ForeignKey("OnlineTeacher")]
    public int? OnlineTeacherId { get; set; }
    [ForeignKey("ClassRoomTeacher")]
    public int? ClassRoomTeacherId { get; set; }

    public Teacher OnlineTeacher { get; set; }
    public Teacher ClassRoomTeacher { get; set; }
}
public class Teacher
{
    public int TeacherId { get; set; }
    public string Name { get; set; }

    [InverseProperty("OnlineTeacher")]
    public ICollection<Course> OnlineCourses { get; set; }
    [InverseProperty("ClassRoomTeacher")]
    public ICollection<Course> ClassRoomCourses { get; set; }
}
```

The above example will result in the following tables in the database.



Thus, you can use the **InverseProperty** and **ForeignKey** attributes to configure multiple relationships between the same entities.

Entity Framework Core Fluent API

Fluent API in Entity Framework Core is a way to configure the model classes. Fluent API uses the ModelBuilder instance to configure the domain model. We can get the reference to the ModelBuilder, when we override the OnModelCreating method of the DbContext. The ModelBuilder has several methods, which you can use to configure the model. These methods are more flexible and provide developers with more power to configure the database than the EF Core conventions and data annotation attributes.

What is Fluent API?

Fluent API is based on a Fluent API design pattern (Fluent Interface) where the result is formulated by method chaining.

In software engineering, a **fluent interface** is an object-oriented API whose design relies extensively on method chaining. Its goal is to increase code legibility by creating a domain-specific language (DSL)

Fluent Interface gives two distinct advantageous:

- Method chaining
- More readable API Code

The **ModelBuilder** class uses the **Fluent API** to build the model.

ModelBuilder

The **ModelBuilder** is the class which is responsible for building the Model.

The **ModelBuilder** builds the initial model from the entity classes that have **DbSet** Property in the context class, which we derive from the **DbContext** class.

It then uses the conventions to create primary keys, foreign keys, relationships etc.

Next, it will look for the Data Annotations attributes to further configure the model.

We can use the **ModelBuilder** to further configure the model. To do that we need to get the reference to the **ModelBuilder** in our context. To do that we need to override the **OnModelCreating** method of the **DbContext** class.

OnModelCreating

As mentioned above, the **DbContext** builds the model using the **ModelBuilder** class.

But, before freezing the model, it calls the **OnModelCreating** method and passes the instance of the **ModelBuilder** to it. This gives us a chance to further configure the model.

The initialization does not happen when EF Core creates the **DbContext**. It happens when we use the Context for the first time.

The EF Core also caches the resulting model. It uses the cached model whenever it creates a new instance of the Context.

We can override the **OnModelCreating** method in our code as shown in the example below. We get the reference to the **ModelBuilder** in our overridden class. Use the **ModelBuilder** along with the fluent API to configure our model.

```
public class CustomerContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    1 reference
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

    0 references
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //Configure domain classes using modelBuilder here
    }
}
```

ModelBuilder class includes important properties and methods to configure the domain model.

Fluent API Example in EF Core

Customer.cs:

```
public class Customer
{
    public int CustomerID { get; set; }
    public string Name { get; set; }
}
```

CustomerContext.cs

```
public class CustomerContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {

    }

    public DbSet<Customer> Customers { get; set; }
}
```

As mentioned earlier, we need to override the OnModelCreating method and use the instance of the ModelBuilder to configure the model

And then use the ModelBuilder instance. In the following code, we are invoking the HasDefaultSchema method to change the schema of the database from dbo to admin.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("Admin");
}
```

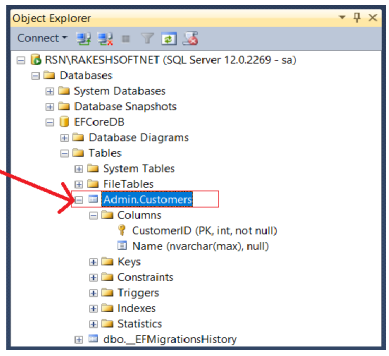
Next, we need to use the migrations to update/create the database as shown below:

Add-Migration "EFCoreDB"

Update-Database

You can see that the DbContext creates the Customers table under the Admin schema

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("Admin");
}

public DbSet<Customer> Customers { get; set; }
```



Method chaining is one of the main features of EF Core Fluent API. For Example, the following code renames table as Customer and defines CustomerID as Primary Key.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Customer>()
    .ToTable("Customer")
    .HasKey(e => e.CustomerID);
}
```

There are several methods available in EF Core Fluent API. These methods broadly classified into the three categories

- Model wide configuration (database)
- Entity Configuration (table)
- Property configuration

Model-wide configuration

You can configure a number of aspects of the model via the Entity Framework Core Fluent API. These options are made available through methods on the **ModelBuilder** type.

Schema

The default schema that EF Core uses to create database objects is **dbo**. You can change this behaviour using the ModelBuilder's **HasDefaultSchema** method:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.HasDefaultSchema("Admin");
}
```

Data Annotations

It is not possible to specify the default schema using Data Annotations attributes.

The **ModelBuilder** class exposes several methods to configure the model. Some of the important methods are listed below.

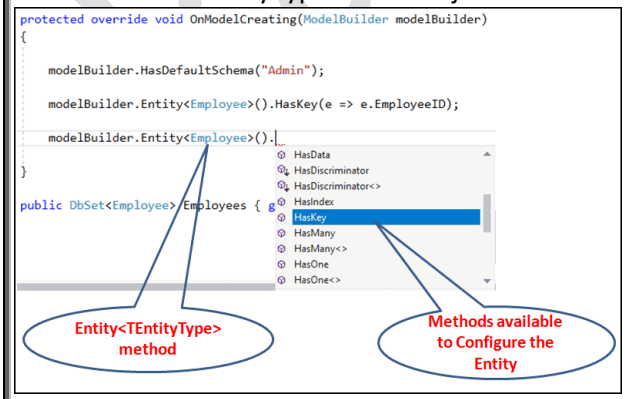
HasDefaultSchema	Configures the default schema that database objects should be created in, if no schema is explicitly configured.
RegisterEntityType	Registers an entity type as part of the model
HasAnnotation	Adds or updates an annotation on the model. If an annotation with the key specified in annotation already exists its value will be updated.
HasChangeTrackingStrategy	Configures the default ChangeTrackingStrategy to be used for this model. This strategy indicates how the context detects changes to properties for an instance of an entity type.
Ignore	Excludes the given entity type from the model. This method is typically used to remove types from the model that were added by convention.
HasDbFunction	Configures a database function when targeting a relational database.
HasSequence	Configures a database sequence when targeting a relational database.

Entity Configuration

The configuration of the entity (Table) is done using the method **Entity**. The following code is an example of how to configure the Primary Key using the HasKey method.

modelBuilder.Entity().HasKey(e => e.EmployeeID);

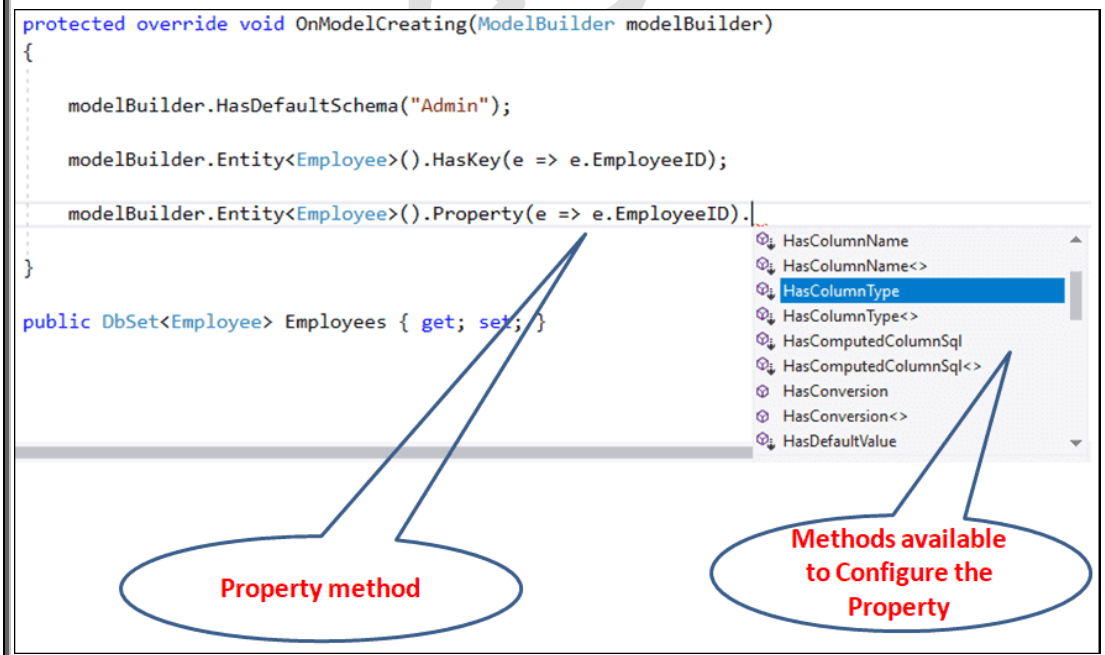
This method Entity returns the EntityTypeBuilder object to configure the entities. Some of the important methods available in the EntityTypeBuilder object are listed below



Method	Description
Ignore	Exclude the entity from the Model.
ToTable	Sets the table name for the entity type
HasKey	Sets the properties that make up the primary key for this entity type.
HasMany	Configures a relationship where this entity type has a collection that contains instances of the other type in the relationship.
HasOne	Configures a relationship where this entity type has a reference that points to a single instance of the other type in the relationship.
HasAlternateKey	Adds or updates an annotation on the entity type. If an annotation with the key specified in annotation already exists its value will be updated
HasChangeTrackingStrategy	Configures the ChangeTrackingStrategy to be used for this entity type. This strategy indicates how the context detects changes to properties for an instance of the entity type.
HasIndex	Configures an index on the specified properties. If there is an existing index on the given set of properties, then the existing index will be returned for configuration.
OwnsOne	Configures a relationship where the target entity is owned by (or part of) this entity. The target entity key value is always propagated from the entity it belongs to.

Property Configuration

The **EntityTypeBuilder** object, which is explained above returns the **Property** Method. This method is used to configure the attributes of the property of the selected entity. The Property method returns the **PropertyBuilder** object, which is specific to the type being configured.



Method	Description
Ignore	Exclude the Property from the Model.
HasColumnName	Configures database column name of the property
HasColumnType	Configures the database column data type of the property
HasDefaultValue	Configures the default value for the column that the property maps to when targeting a relational database.
HasComputedColumnSql	Configures the property to map to a computed column when targeting a relational database.
HasField	Specifies the backing field to be used with a property.
HasMaxLength	Specifies the maximum length of the property.
IsConcurrencyToken	Enables the property to be used in an optimistic concurrency updates
IsFixedLength	Configures the property to be fixed length. Use HasMaxLength to set the length that the property is fixed to.
IsMaxLength	Configures the property to allow the maximum length supported by the database provider
IsReguired	Specifies the database column as non-nullable.
IsUnicode	Configures the property to support Unicode string content
ValueGeneratedNever	Configures a property to never have a value generated when an instance of this entity type is saved.
ValueGeneratedOnAdd	Configures a property to have a value generated only when saving a new entity, unless a non-null, non-temporary value has been set, in which case the set value will be saved instead. The value may be generated by a client-side value generator or may be generated by the database as part of saving the entity.
ValueGeneratedOnAddOrUpdate	Configures a property to have a value generated when saving a new or existing entity.
ValueGeneratedOnUpdate	Configures a property to have a value generated when saving an existing entity.

Entity Framework Core Ignore Method

The **Ignore** method of the EF Core Fluent API is used to ignore a **Property** or **Entity (Table)** from being mapped to the database

Consider the following Models:

```
public class Employee
{
    public int EmployeeID { get; set; }
    public string Name { get; set; }

    public DateTime DOB { get; set; }

    public int Age { get; set; }

    public Dept Dept { get; set; }
}
public class Dept
{
    public int DeptId { get; set; }
    public string DeptName { get; set; }
}
```

Ignore Entity

The **Ignore<TEntity>()** method enables you to explicitly exclude an entity from the EF model, and therefore it will not be mapped to a database table.

In the above model classes, the **Dept** entity is referenced as a navigation property in the **Employee** entity, and will be included in the model by convention. The Ignore method is used to ensure that the **Dept** class will not be mapped to a table in the database:

```
public class EFCoreContext: DbContext
{
     ..............................
     ..............................
    public DbSet<Employee> Employees { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<Dept>();
    }
}
```

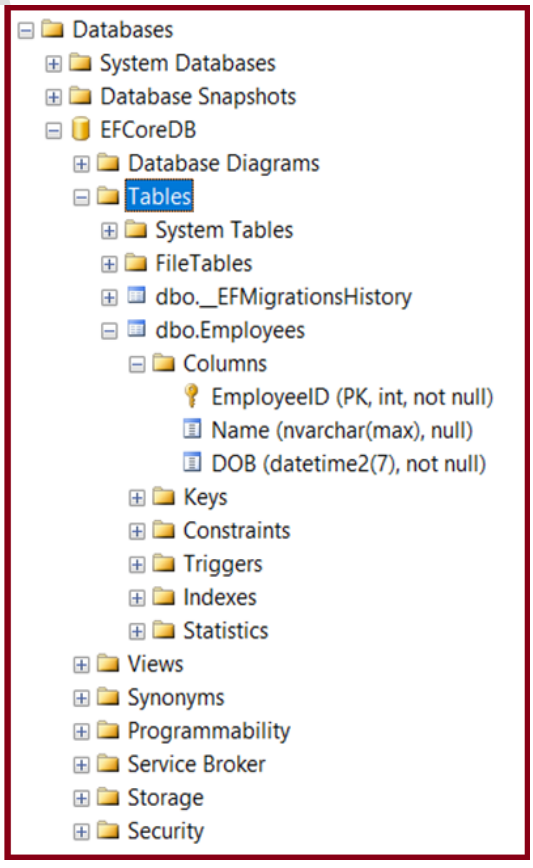
Ignore Property:

The Age field in the Employee Class is redundant as we can always calculate it from the DOB property. Hence it is not required to be mapped to the database.

```
public class EFCoreContext: DbContext
{
      ..............................
      ..............................
     public DbSet<Employee> Employees { get; set; }

     protected override void OnModelCreating(ModelBuilder modelBuilder)
     {
         modelBuilder.Entity<Employee>().Ignore("Age");
     }
}
```

The following image shows the model and the database generated.

```
public class Employee
{
     public int EmployeeID { get; set; }
     public string Name { get; set; }

     public DateTime DOB { get; set; }

     public int Age { get; set; }

     public Dept Dept { get; set; }
}

public class Dept
{
     public int DeptId { get; set; }
     public string DeptName { get; set; }
}
```



Data Annotation

The Data Annotations equivalent to the **Ignore** method is the **NotMapped** attribute.

Entity Framework Core HasAlternateKey Method

The Entity Framework Core Fluent API **HasAlternateKey** method creates the unique constraint for the property in the database. The Primary Key already has Unique Constraint defined, but you can have only one Primary Key in the table. Unique Constraints ensures that no duplicate values are entered in the columns.

Consider the following Model. By Convention, the **EmployeeID** is mapped as the Primary Key.

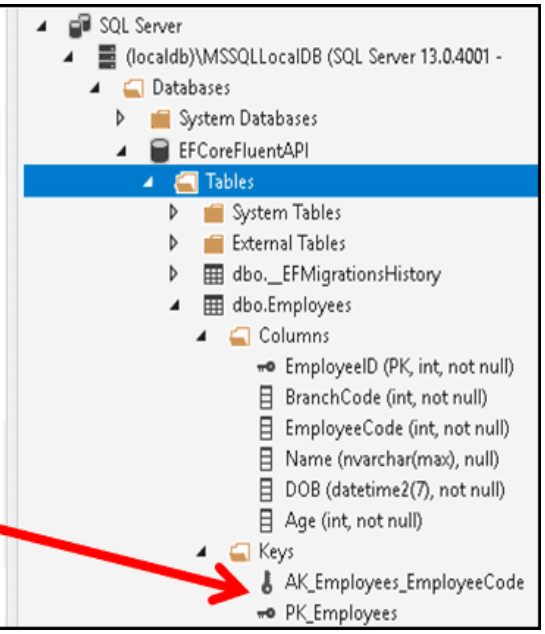
```
public class Employee
{
    public int EmployeeID { get; set; }

    public int BranchCode { get; set; }
    public int EmployeeCode { get; set; }

    public string Name { get; set; }

    public DateTime DOB { get; set; }

    public int Age { get; set; }
}
```

But we would like to ensure that the **EmployeeCode** must also be Unique i.e. no two employees should have the same Employee Code. This is the ideal case to use HasAlternateKey.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Employee>()
                .HasAlternateKey(e => e.EmployeeCode);
}
```

Update the database and you will see that the UNIQUE Constraint is added for the property **EmployeeCode**. The Constraint is named as **AK_<EntityName>_<PropertyName>**

```
protected override void OnConfiguring(DbContextOptionsBuilder option
{
    optionsBuilder.UseSqlServer(connectionString);
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{

    modelBuilder.Entity<Employee>()
        .HasAlternateKey(e=> e.EmployeeCode);

}

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

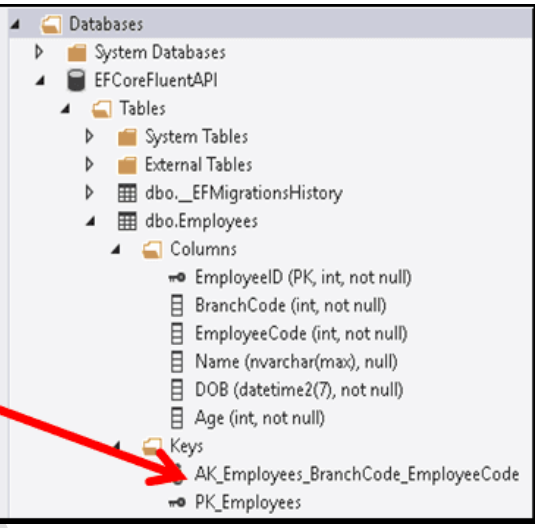


Composite Alternate Keys

The alternative key can be composed of multiple columns. In such cases, we need use an anonymous object as the argument to the **HasAlternateKey** as shown below.

```
modelBuilder.Entity<Employee>()
            .HasAlternateKey(e => new { e.BranchCode, e.EmployeeCode });
```

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{

    //modelBuilder.Entity<Employee>()
    //    .HasAlternateKey(e=> e.EmployeeCode);


    modelBuilder.Entity<Employee>()
        .HasAlternateKey(e => new { e.BranchCode, e.EmployeeCode });

}

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



The Constraint follows the naming convention as **AK_<EntityName>_<PropertyName1>_<PropertyName2>** as shown in the image above.

Data Annotations:

There is no direct alternative to the HasAlternateKey method in data annotations or in default convention.

HasKey Method in Entity Framework Core:

HasKey is a Fluent API method, which allows us to configure the primary key & composite primary of an entity in EF Core. Here we will see how to use HasKey in EF Core.

Primary Key

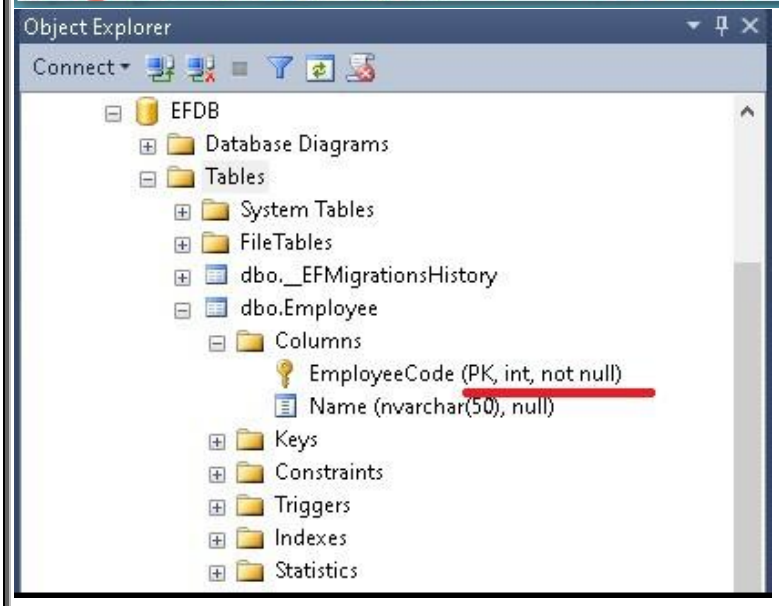
The default convention in EF Core uses the property with the name **id** or with the name **<className>ID**. In the absence of such properties it will not create the table, but raises an error

There are two ways you can create the primary key, one is using the data annotation **Key** Attribute. The other way is to use the **HasKey** method

```
public class Employee
{
    0 references
    public int EmployeeCode { get; set; }
    0 references
    public string Name { get; set; }
}
```

The following code configures the **EmployeeCode** as the Primary Key.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Employee>()
                .HasKey("EmployeeCode");
}
```



Composite Primary Key:

A primary key that consists of more than one column is called a **Composite Primary key**. **Default conventions** or **Key** attribute in Data Annotations do not support the creation of Composite Primary keys in EF Core.

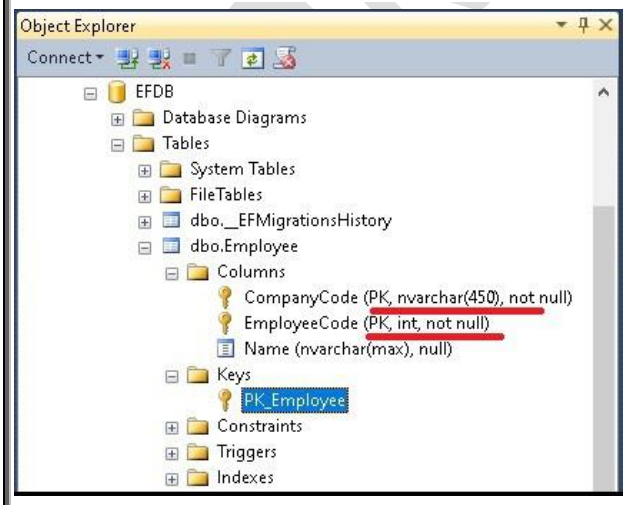
The only way we can create it using the **HasKey** method.

In the following model, we want both **CompanyCode** & **EmployeeCode** to be part of the primary key.

```
public class Employee
{
    public string CompanyCode { get; set; }
    public int EmployeeCode { get; set; }
    public string Name { get; set; }
}
```

We create the anonymous type consisting of the above properties of and pass it as the argument to the **HasKey** method.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Employee>()
        .HasKey(e => new { e.CompanyCode, e.EmployeeCode });
}
```



The Fluent API HasColumnName Method in Entity Framework Core:

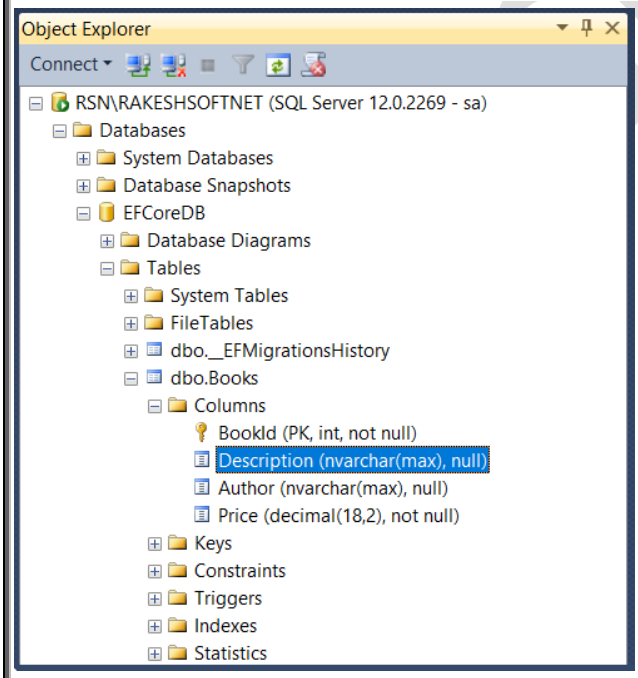
The **HasColumnName** method is applied to a property to specify the database column that the property should map to when the entity's property name and the database column name differ. The following example maps the **Title** property in the Book entity to a database column named **Description** in the Books table:

Book.cs:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public decimal Price { get; set; }
}
```

EFCoreContext.cs:

```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Book>()
                    .Property(b => b.Title).HasColumnName("Description");
    }
}
```



Data Annotations:

The data annotations equivalent to the **HasColumnName** method is the **Column** attribute.

The Fluent API HasColumnType Method in Entity Framework Core:

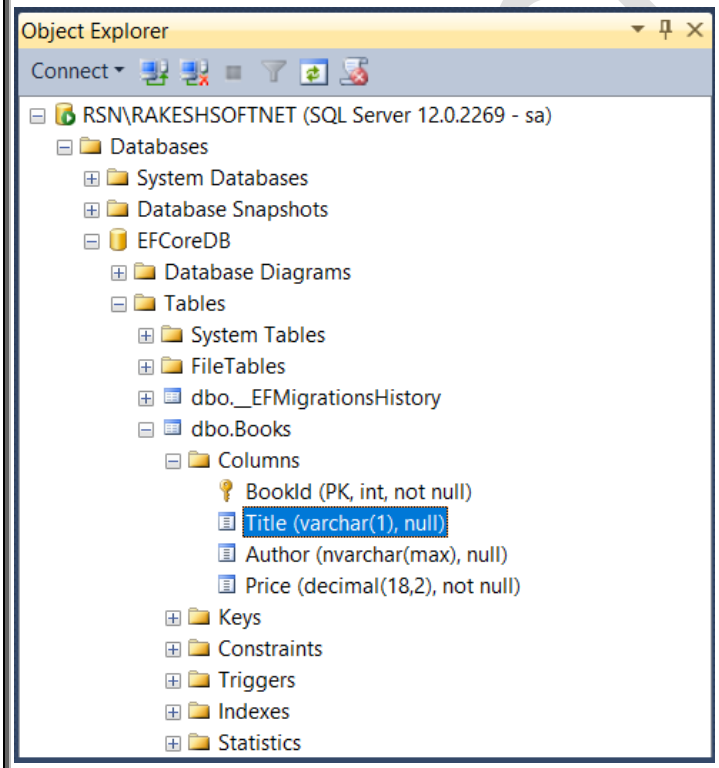
The **HasColumnType** method is applied to a property to specify the data type of the column that the property should map to when the type differs from convention. The following example specifies that the **Title** column in the Books table is to be configured as **varchar** instead of the default **nvarchar**:

Book.cs:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public decimal Price { get; set; }
}
```

EFCoreContext.cs:

```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Book>()
                    .Property(b => b.Title).HasColumnType("varchar");
    }
}
```



The **HasColumnType** method is also useful when mapping columns to user defined data types. In the next example, the **Title** property on the **Book** entity is mapped to the "Name" user defined data type:

First Create User Defined Data Types in SQL Server:

We can create a user defined data type using 2 methods:

1. T-SQL
2. Manually

In our example, we'll be creating a user defined data type named "**Name**" which stores max 10 characters only.

Creating an User-Defined Data Type using T-SQL

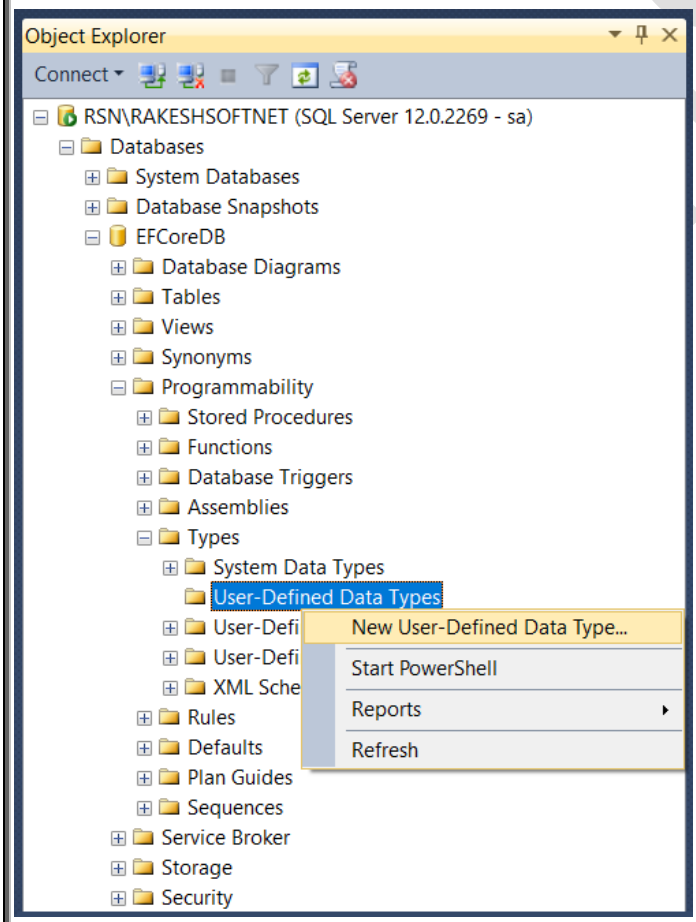
1. Launch SQL Server Management Studio.
2. Connect to your Server.
3. Click "New Query" in menu items.
4. Make sure you select the ideal database in which you want to create.
5. Write these TSQL codes:

```
Create Type Name from varchar(10) NULL
```

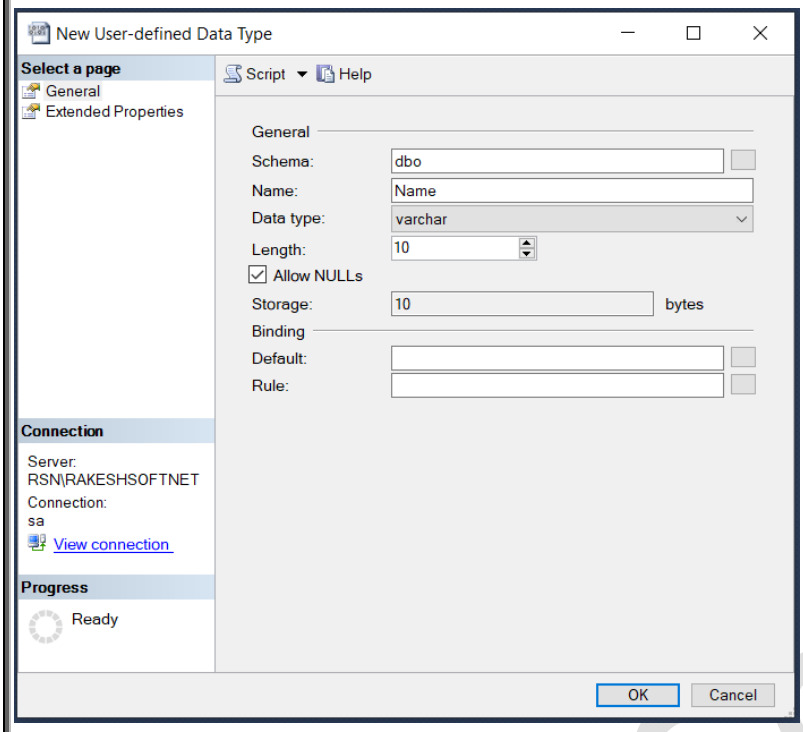
6. Execute
7. The UDDT will be created under {Database}-Programmability-Types-User Defined Data Types section.

Creating a User-Defined Data Type Manually

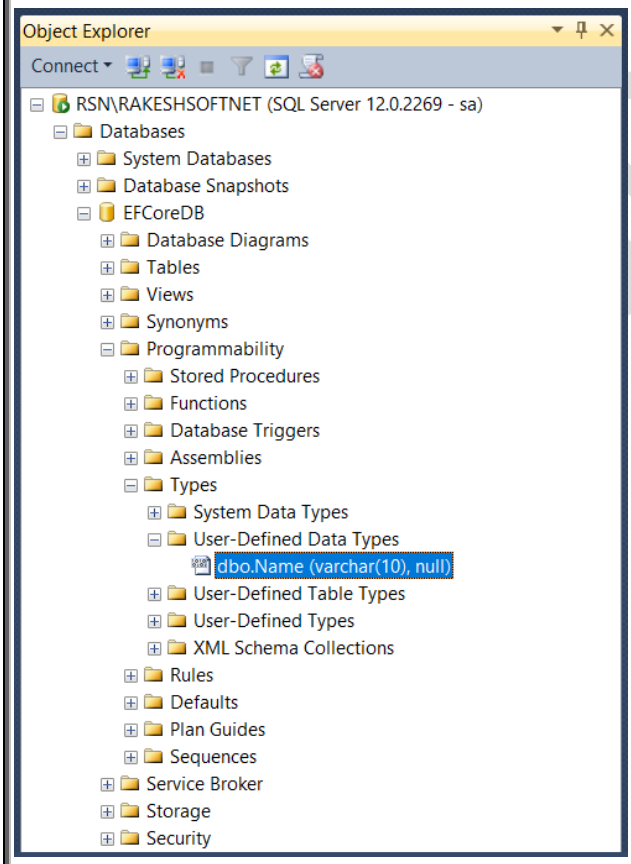
1. Launch SQL Server Management Studio.
2. Connect to your Server.
3. Follow path {Database}-Programmability-Types-User Defined Data Types section and right click on it.
4. Choose "New User-Defined Data Type."



5. Fill in specific information regarding creation of data type:



6. When you're done filling, click OK and this will create your User-Defined Data Type in the left panel.

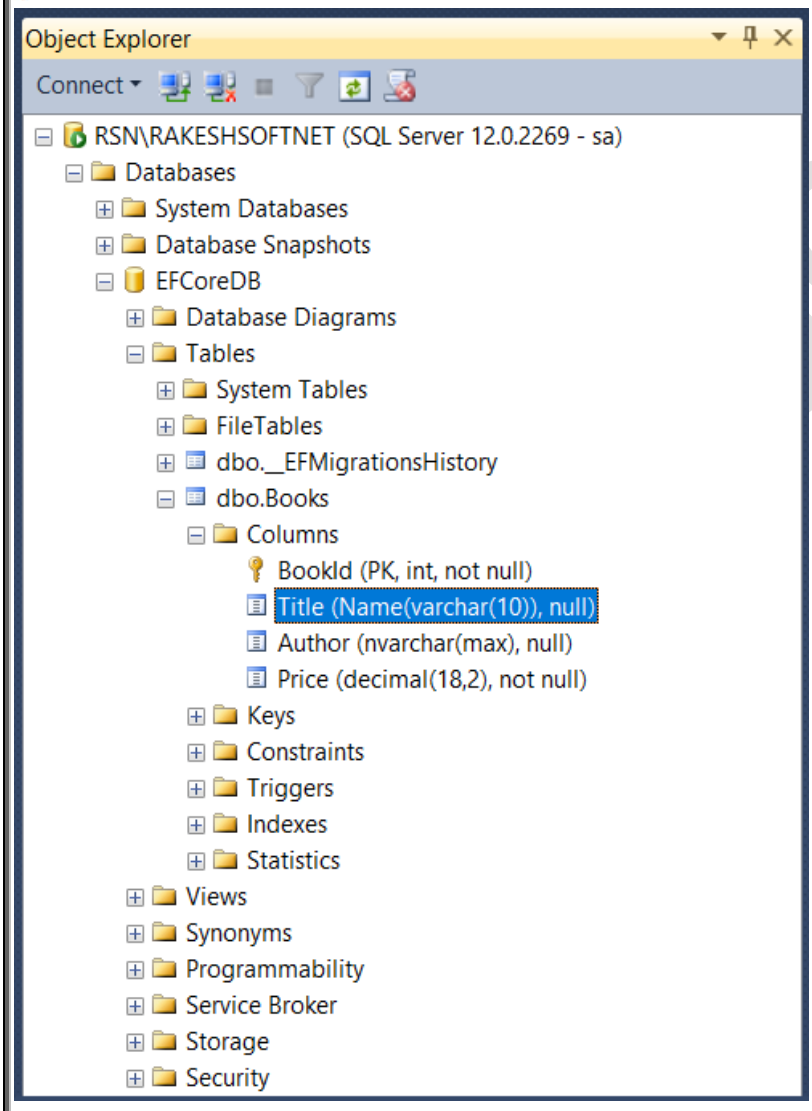


That's it!

Now you can use your custom datatype in a table or SQL query.

EFCoreContext.cs:

```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Book>()
                    .Property(b => b.Title).HasColumnType("Name");
    }
}
```



Data Annotations

The data annotations approach to specifying the data type of the mapped column is to provide a value for the **TypeName** property of the **Column** attribute.

The Fluent API HasMaxLength Method in Entity Framework Core:

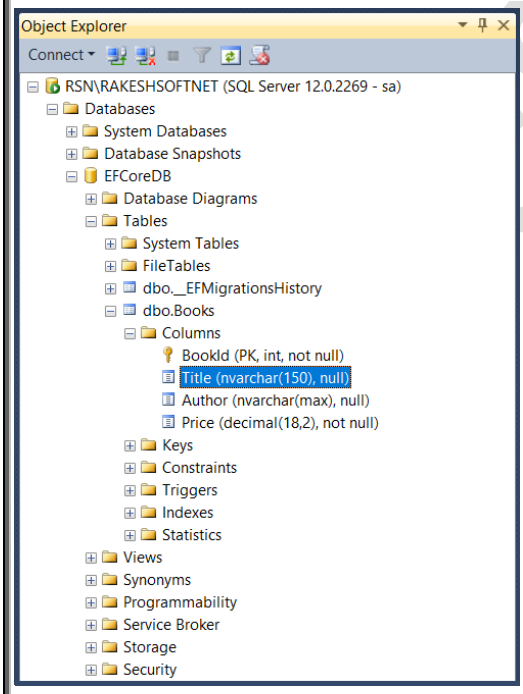
The HasMaxLength method is applied to a property to specify a maximum number of characters or bytes for the column that the property should map to. The following example specifies that the Title column in the Books table is to be have a maximum length of 150 characters instead of the default which is unlimited:

Book.cs:

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public string Author { get; set; }
    public decimal Price { get; set; }
}
```

EFCoreContext.cs:

```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Book>()
                    .Property(b => b.Title).HasMaxLength(150);
    }
}
```



Data Annotations:

The Data Annotations equivalents to the **HasMaxLength** method are the **MaxLength** attribute and the **StringLength** attribute.

The Fluent API HasComputedColumnSql Method in Entity Framework Core:

The Entity Framework Core Fluent API **HasComputedColumnSql** method is used to specify that the property should map to a computed column. The method takes a string indicating the expression used to generate the default value for a database column.

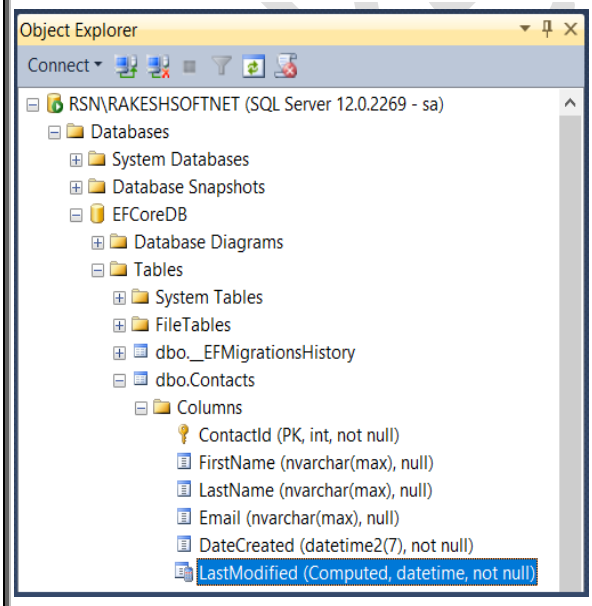
In the following example, the **LastModified** property of the **Contact** entity is mapped to a computed column. The value of the column is generated by the database's **GetUtcDate()** method whenever the row is created or updated:

Contact.cs:

```
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string Email { get; set; }
    public DateTime DateCreated { get; set; }
    public DateTime LastModified { get; set; }
}
```

EFCoreContext.cs:

```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

    public DbSet<Contact> Contacts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Contact>()
                                .Property(p => p.LastModified)
                                                        .HasComputedColumnSql("GetUtcDate()");
    }
}
```



In the following example, the **FullName** property of the **Contact** entity is mapped to a computed column. The value of the column is generated by the database by concatenating FirstName & LastName column whenever the row is created or updated:

Contact.cs:

```
public class Contact
{
    public int ContactId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string FullName { get; set; }
    public string Email { get; set; }
}
```

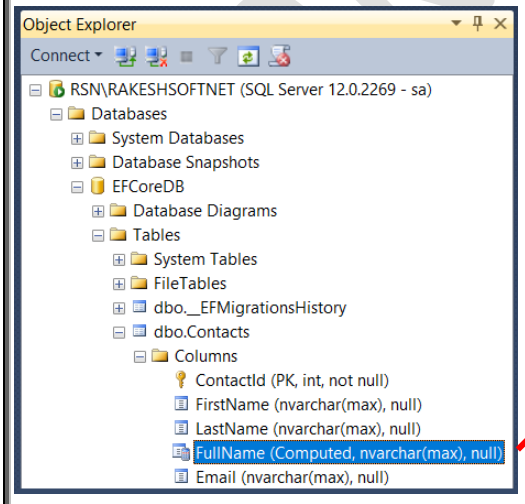
EFCoreContext.cs:

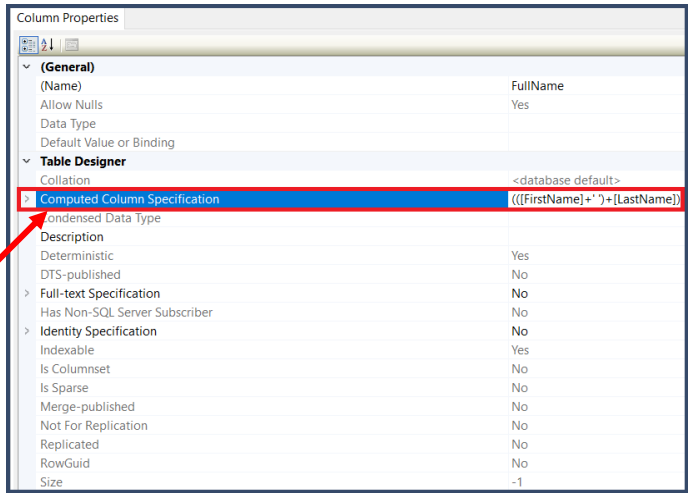
```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

    public DbSet<Contact> Contacts { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Contact>()
                    .Property(p => p.FullName)
                    .HasComputedColumnSql("[FirstName] + ' ' + [LastName]");
    }
}
```

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

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Contact>()
                .Property(p => p.FullName)
                .HasComputedColumnSql("[FirstName] + ' ' + [LastName]",stored:true);
}
```

Computed column specification added





Computed columns are very powerful. Entity Framework Core with its fluent API allows them to be easily added. You'll often see scenarios where a property is made up of underlying incrementing number along with a prefix or suffix. This is a perfect place to take advantage of computed columns.

Here's another quick example:

Employee entity has a primary key named **Id** of type int and also has a property named '**EmpCode**'.

'**EmpCode**' is made up of the primary key along with a prefix.

Example of how the EmpCode column might look in the table.

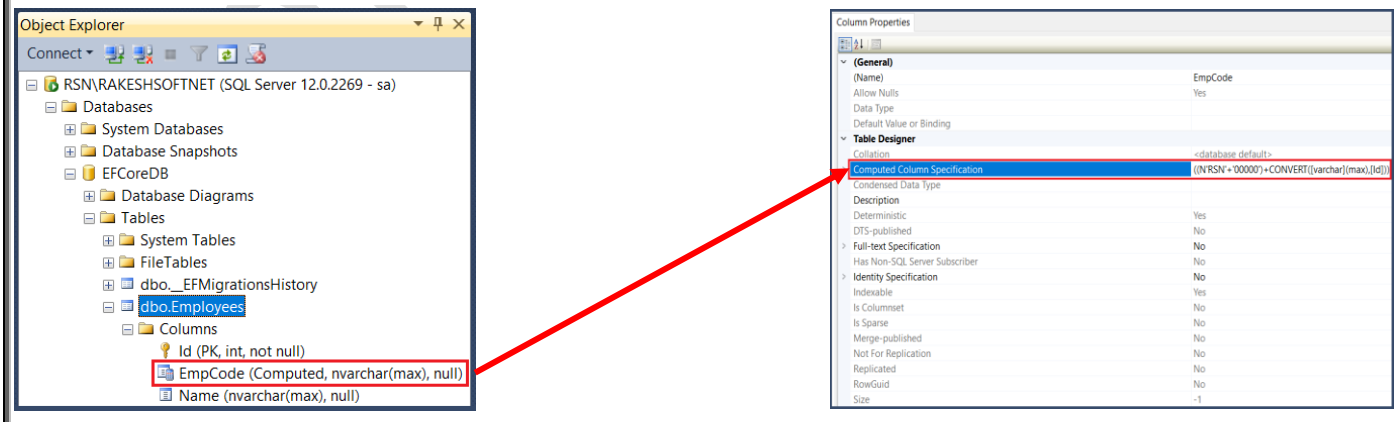
Id	EmpCode	Name
1	RSN000001	David
2	RSN000002	Smith
3	RSN000003	Peter
4	RSN000004	Alina
5	RSN000005	Fleming

Employee.cs:

```
public class Employee
{
    public int Id { get; set; }
    public string EmpCode { get; set; }
    public string Name { get; set; }
}
```

EFCoreContext.cs:

```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
                    .Property(e => e.EmpCode)
                    .HasComputedColumnSql("N'RSN'+ '00000' + CAST(Id AS VARCHAR(MAX))");
    }
}
```



Data Annotations

It is not possible to configure computed columns using data annotations.

The Fluent API HasDefaultValue Method in Entity Framework Core:

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

The Entity Framework Core Fluent API **HasDefaultValue** method is used to specify the default value for a database column mapped to a property. The value must be a constant.

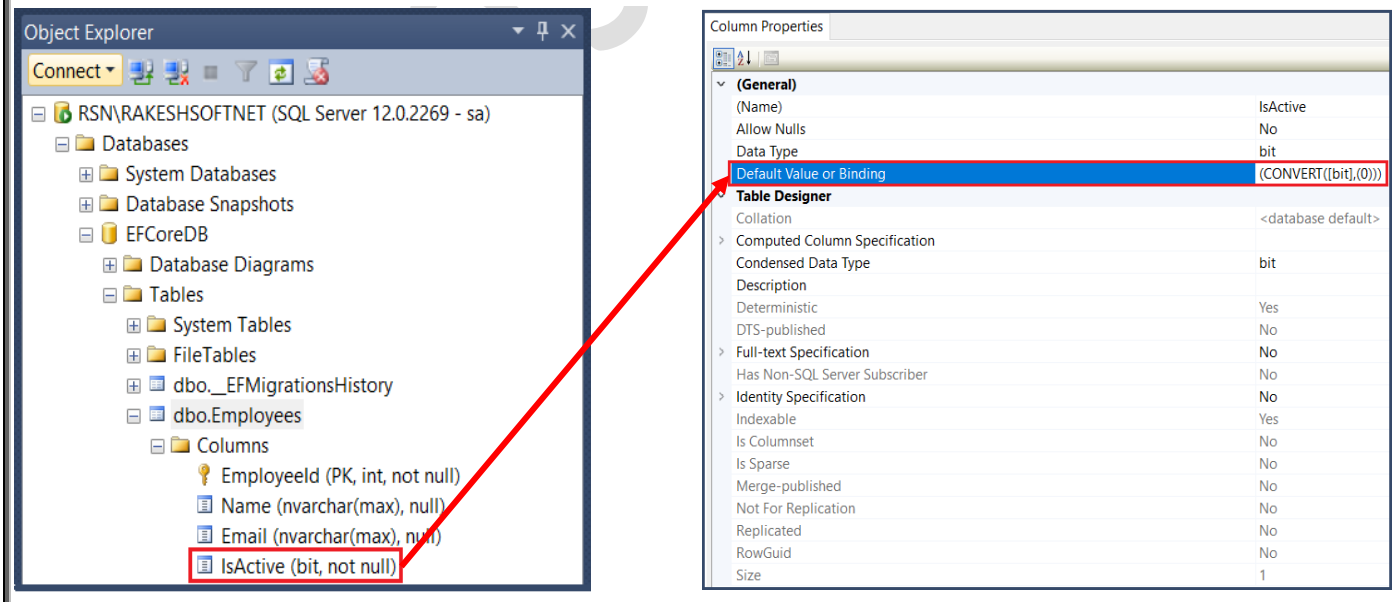
You can configure a default value on a property:

Employee.cs:

```
public class Employee
{
    public int EmployeeId { get; set; }
    public string Name { get; set; }
    public string Email { get; set; }
    public bool IsActive { get; set; }
}
```

EFCoreContext.cs:

```
public class EFCoreContext: DbContext
{
    private const string connectionString = "Data Source=RSN\\RAKESHSOFTNET;Database=EFCoreDB;User Id=sa;Password=123";

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(connectionString);
    }

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

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
                    .Property(e => e.IsActive)
                    .HasDefaultValue(false);
    }
}
```



Data Annotations

It is not possible to configure default database column values using data annotations.