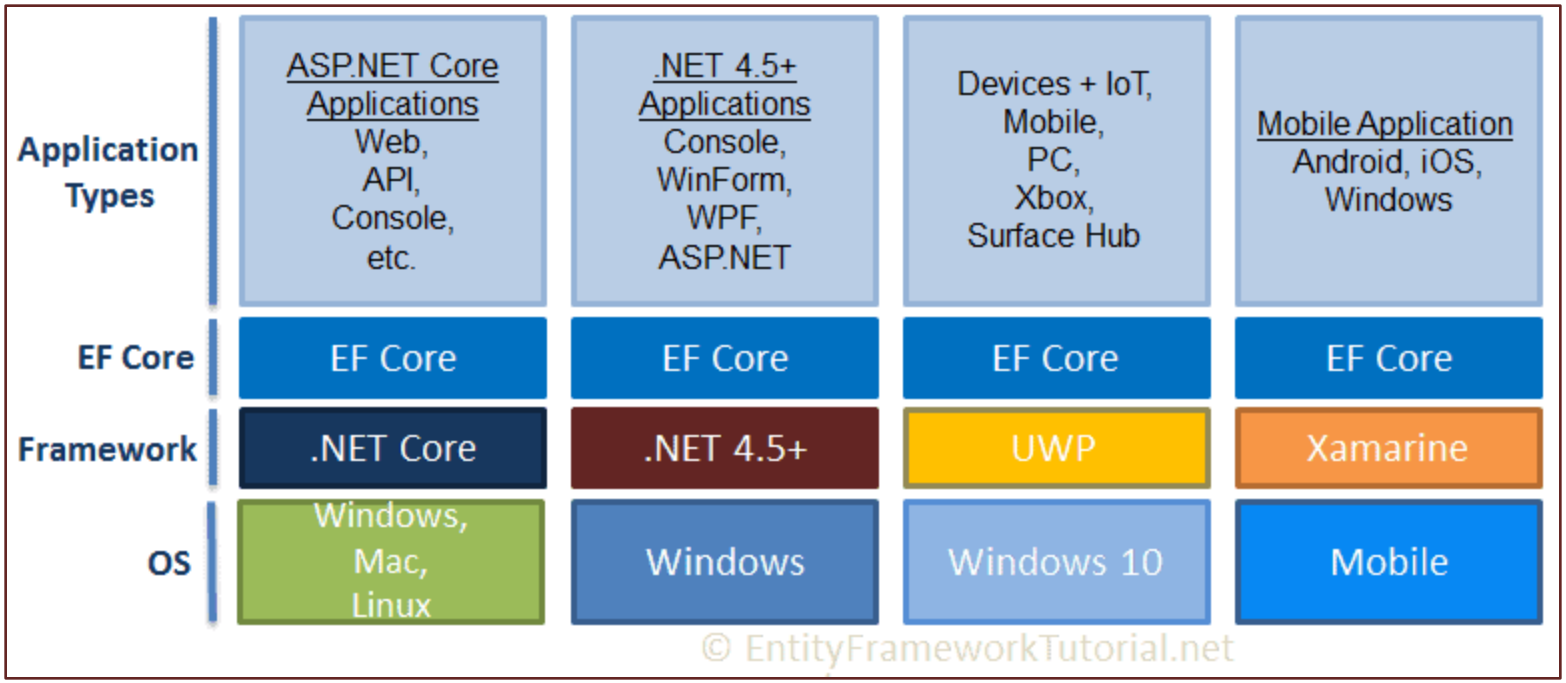
EF Core

Prepared for Vth semester DDU-CE students
2022-23 WAD

Apurva A Mehta

Entity Framework Core

- Entity Framework Core is the new version of Entity Framework after EF 6.x.
- It is open-source, lightweight, extensible and a cross-platform version of Entity Framework data access technology.
- Entity Framework is an Object/Relational Mapping (O/RM) framework.
- It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.
- EF Core is intended to be used with .NET Core applications. However, it can also be used with standard .NET 4.5+ framework based applications.

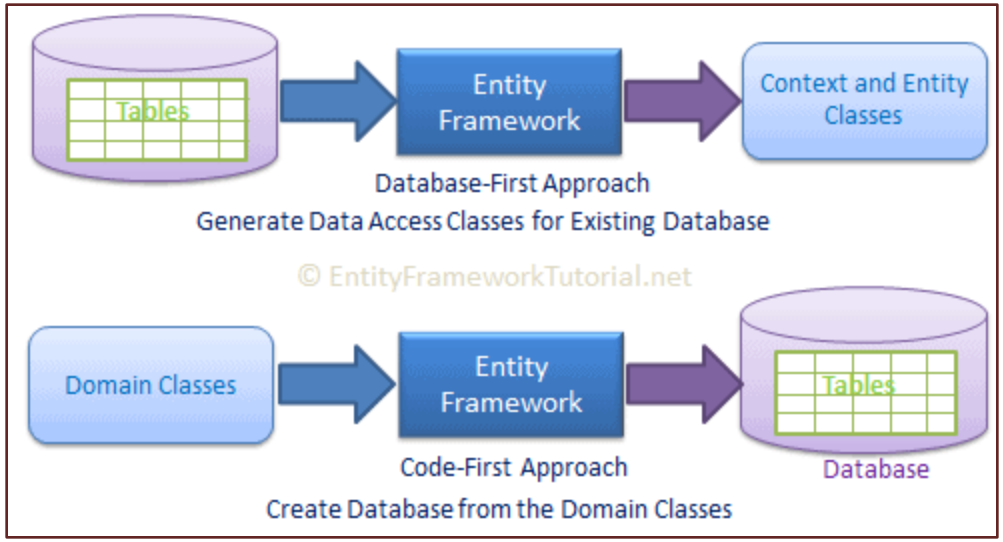

Application Types
ASP.NET Core Applications
Web,
API,
Console,
etc.
.NET 4.5+ Applications
Console,
WinForm,
WPF,
ASP.NET
Devices + IoT,
Mobile,
PC,
Xbox,
Surface Hub
Mobile Application
Android, iOS,
Windows
EF Core
EF Core
EF Core
EF Core
EF Core
Framework
.NET Core
.NET 4.5+
UWP
Xamarine
OS
Windows,
Mac,
Linux
Windows
Windows 10
Mobile
© EntityFrameworkTutorial.net

EF Core Development Approaches

- EF Core supports two development approaches

1) Code-First
2) Database-First.

- EF Core mainly targets the code-first approach and provides little support for the database-first approach.
  - as the visual designer or wizard for DB model is not supported as of EF Core 2.0.


Tables
Entity Framework
Context and Entity Classes
Database-First Approach
Generate Data Access Classes for Existing Database
© EntityFrameworkTutorial.net
Domain Classes
Entity Framework
Tables
Database
Code-First Approach
Create Database from the Domain Classes

Install Entity Framework Core

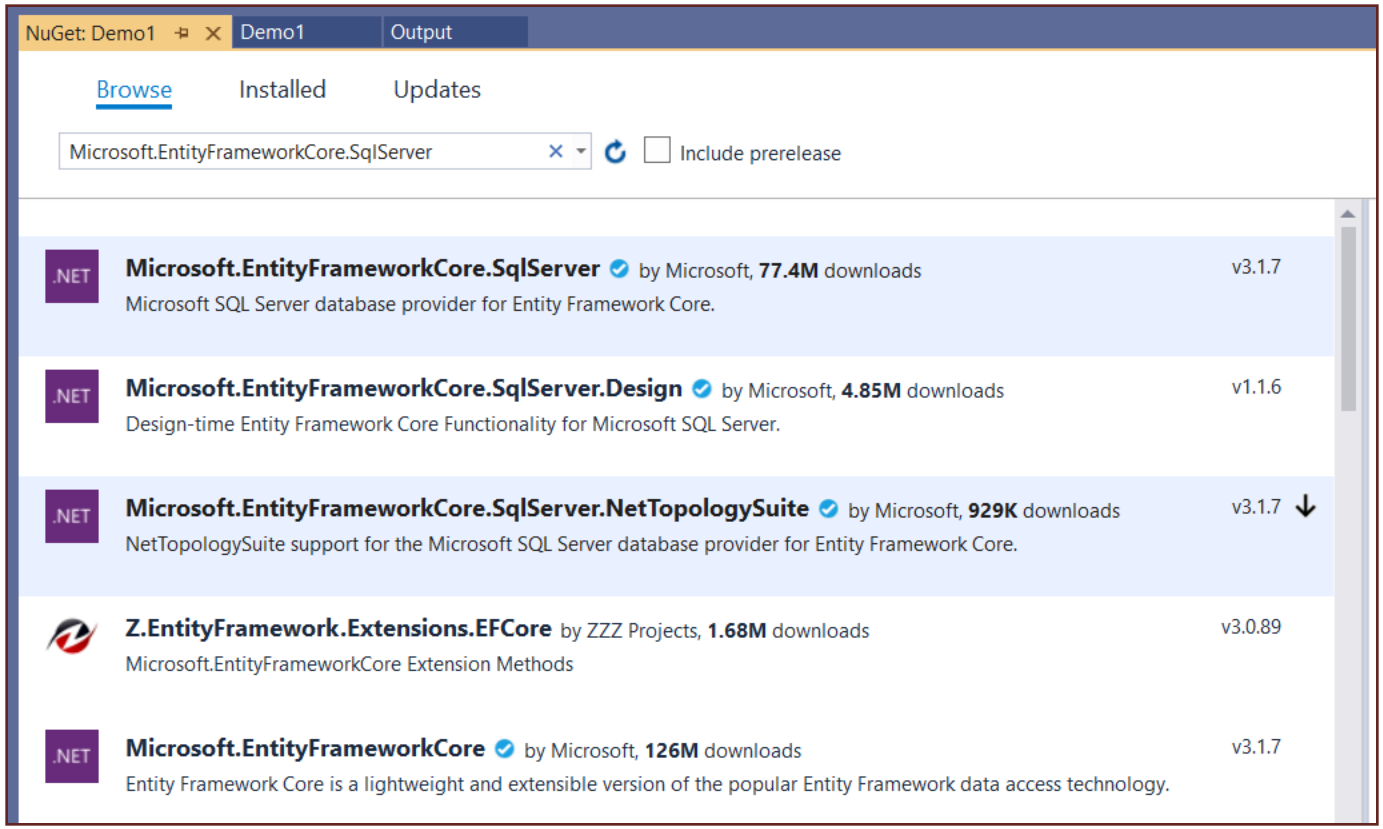
- EF Core is not a part of .NET Core and standard .NET framework. It is available as a NuGet package.
- You need to install NuGet packages for the following two things to use EF Core in your application:

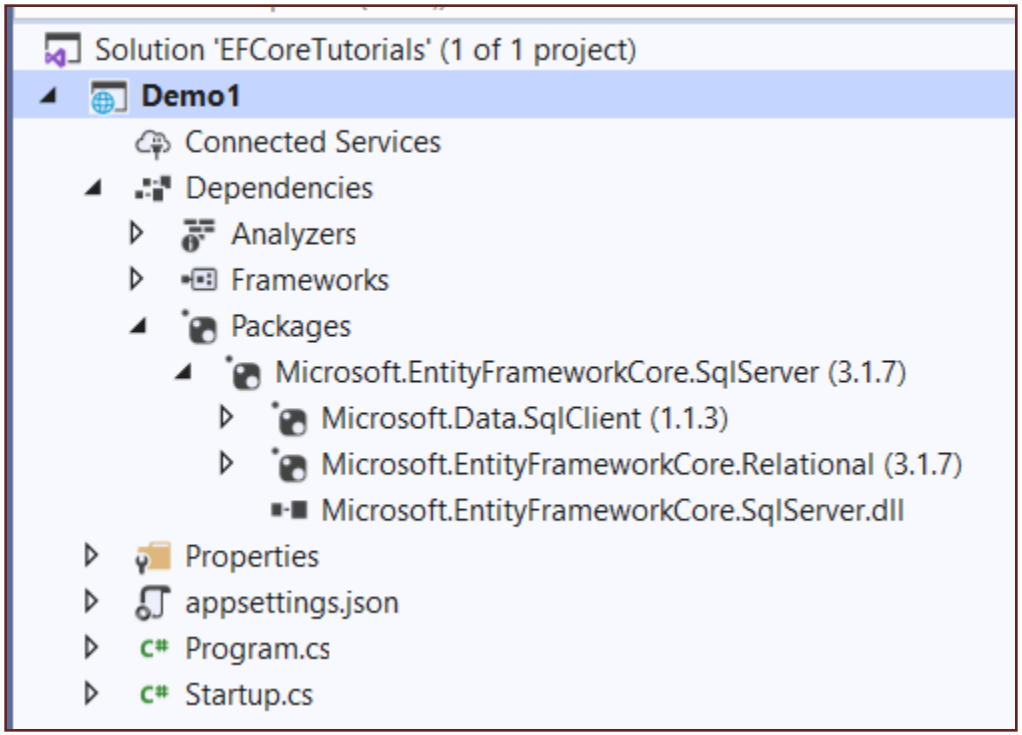
1. EF Core DB provider
2. EF Core tools

Install EF Core DB Provider

- EF Core allows us to access databases via the provider model. There are different EF Core DB providers available for the different databases. These providers are available as NuGet packages.
- First, we need to install the NuGet package for the provider of the database we want to access.
- Here, we want to access MS SQL Server database, so we need to install

  Microsoft.EntityFrameworkCore.SqlServer NuGet package.

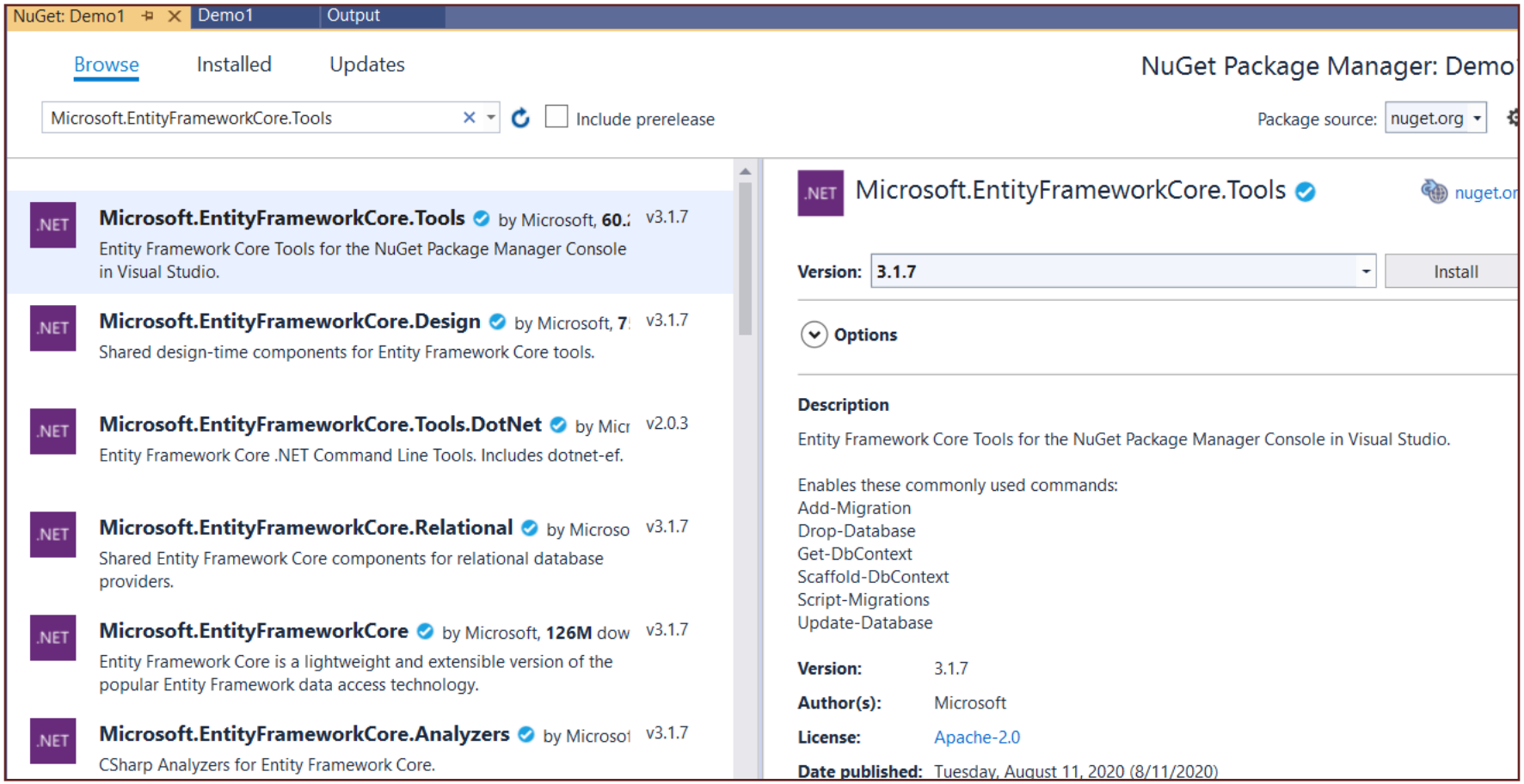

NuGet: Demo1
Demo1
Output
Browse
Installed
Updates
Microsoft.EntityFrameworkCore.SqlServer
Include prerelease
.NET
Microsoft.EntityFrameworkCore.SqlServer by Microsoft, 77.4M downloads
v3.1.7
Microsoft SQL Server database provider for Entity Framework Core.
.NET
Microsoft.EntityFrameworkCore.SqlServer.Design by Microsoft, 4.85M downloads
v1.1.6
Design-time Entity Framework Core Functionality for Microsoft SQL Server.
.NET
Microsoft.EntityFrameworkCore.SqlServer.NetTopologySuite by Microsoft, 929K downloads
v3.1.7
NetTopologySuite support for the Microsoft SQL Server database provider for Entity Framework Core.
Z.EntityFramework.Extensions.EFCore by ZZZ Projects, 1.68M downloads
v3.0.89
Microsoft.EntityFrameworkCore Extension Methods
.NET
Microsoft.EntityFrameworkCore by Microsoft, 126M downloads
v3.1.7
Entity Framework Core is a lightweight and extensible version of the popular Entity Framework data access technology.


Solution 'EFCoreTutorials' (1 of 1 project)
Demo1
Connected Services
Dependencies
Analyzers
Frameworks
Packages
Microsoft.EntityFrameworkCore.SqlServer (3.1.7)
Microsoft.Data.SqlClient (1.1.3)
Microsoft.EntityFrameworkCore.Relational (3.1.7)
Microsoft.EntityFrameworkCore.SqlServer.dll
Properties
appsettings.json
Program.cs
Startup.cs

Install EF Core Tools

- Along with the DB provider package, you also need to install EF tools to execute EF Core commands.
- These make it easier to perform several EF Core-related tasks in your project at design time, such as migrations, scaffolding, etc.
- EF Tools are available as NuGet packages.
- You can install NuGet package for EF tools depending on where you want to execute commands
  - either using Package Manager Console (PowerShell version of EF Core commands)
  - or using dotnet CLI.

Install EF Core Tools for PMC



Entity Framework Core: DbContext

- The DbContext class is an integral part of Entity Framework.
- An instance of DbContext represents a session with the database which can be used to query and save instances of your entities to a database.
- DbContext is a combination of the Unit Of Work and Repository patterns.

Cont.

- DbContext in EF Core allows us to perform following tasks:

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

SchoolContext.cs | NuGet: Demo1 | Demo1 | Output

Demo1 | Demo1.Models.SchoolContext | Students

```
7   namespace Demo1.Models
8   {
        1 reference
9       public class SchoolContext : DbContext
10      {
            0 references
11          public SchoolContext()
12          {
13          }
14
            0 references
15          protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
16          {
17          }
            0 references
18          protected override void OnModelCreating(ModelBuilder modelBuilder)
19          {
20          }
21          //entities
            0 references
22          public DbSet<Student> Students { get; set; }
            0 references
23          public DbSet<Course> Courses { get; set; }
24      }
25  }
```

```
namespace Demo1.Models
{
    1 reference
    public class Student
    {
        0 references
        public int StudentId { get; set; }
        0 references
        public string Name { get; set; }
    }
}
```

```
namespace Demo1.Models
{
    1 reference
    public class Course
    {
        0 references
        public int CourseId { get; set; }
        0 references
        public string CourseName { get; set; }
    }
}
```

_CreateSchoolDB.cs | Package Manager Console | SchoolContext.cs | NuGet: Demo1 | Demo1 | Output

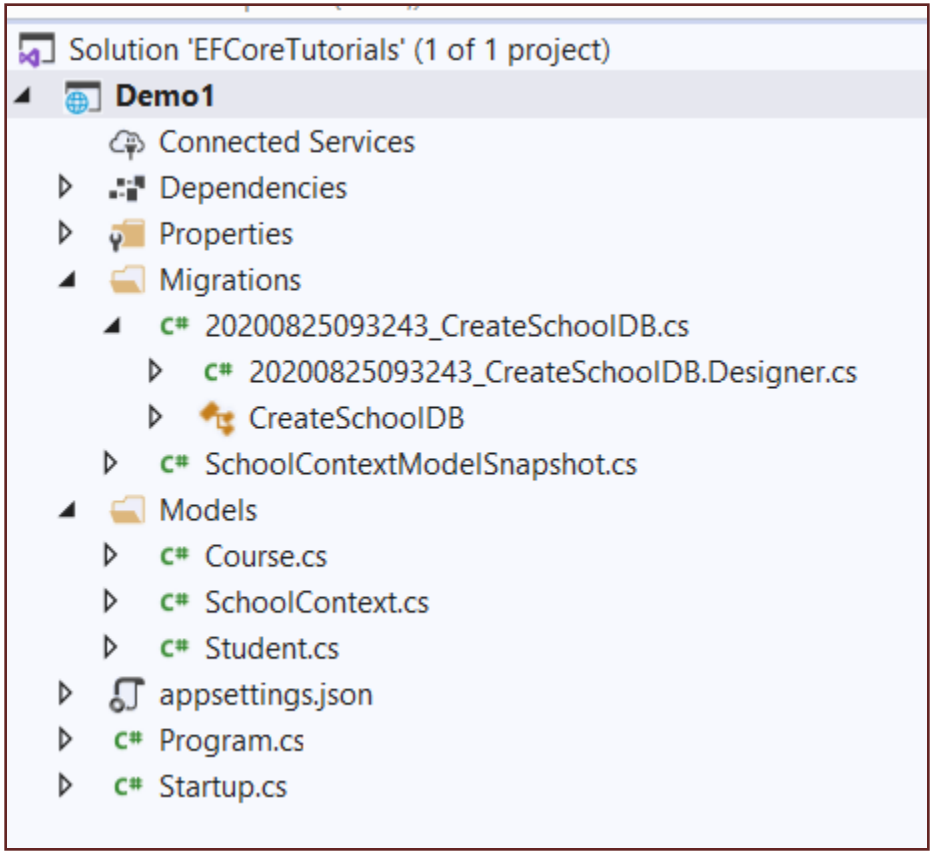
Demo1.Models.SchoolContext | Students

```
namespace Demo1.Models
{
    3 references
    public class SchoolContext : DbContext
    {
        0 references
        public SchoolContext()
        {
        }

        0 references
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Server=(localdb)\MSSQLLocalDB;Database=SchoolDB;
            Trusted_Connection=True;");
        }
        0 references
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
        }
        //entities
        0 references
        public DbSet<Student> Students { get; set; }
        0 references
        public DbSet<Course> Courses { get; set; }
    }
}
```

20200825093243_CreateSchoolDB.cs | Package Manager Console | SchoolContext.cs

Package source: All Default project: Demo1

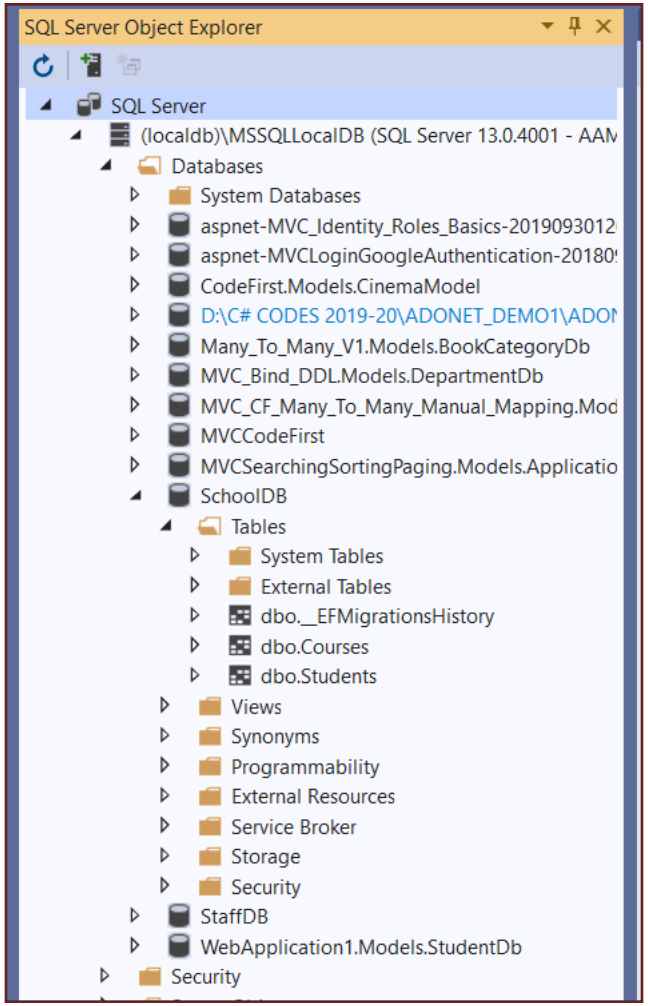
```
Each package is licensed to you by its owner. NuGet is not re
third-party packages. Some packages may include dependencies
package source (feed) URL to determine any dependencies.

Package Manager Console Host Version 5.6.0.6591

Type 'get-help NuGet' to see all available NuGet commands.

PM> add-migration CreateSchoolDB
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM>
```


Solution 'EFCoreTutorials' (1 of 1 project)
Demo1
Connected Services
Dependencies
Properties
Migrations
20200825093243_CreateSchoolDB.cs
20200825093243_CreateSchoolDB.Designer.cs
CreateSchoolDB
SchoolContextModelSnapshot.cs
Models
Course.cs
SchoolContext.cs
Student.cs
appsettings.json
Program.cs
Startup.cs

What is the use of Migration specific class generated with Add-Migration?

What is the use of Model snapshot class file?

```
PM> update-database
Build started...
Build succeeded.
Applying migration '20200825093243_CreateSchoolDB'.
Done.
PM>
```


SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 13.0.4001 - AAM
Databases
System Databases
aspnet-MVC_Identity_Roles_Basics-2019093012
aspnet-MVCLoginGoogleAuthentication-20180
CodeFirst.Models.CinemaModel
D:\C# CODES 2019-20\ADONET_DEMO1\ADO
Many_To_Many_V1.Models.BookCategoryDb
MVC_Bind_DDL.Models.DepartmentDb
MVC_CF_Many_To_Many_Manual_Mapping.Mod
MVCCodeFirst
MVCSearchingSortingPaging.Models.Applicatio
SchoolDB
Tables
System Tables
External Tables
dbo.__EFMigrationsHistory
dbo.Courses
dbo.Students
Views
Synonyms
Programmability
External Resources
Service Broker
Storage
Security
StaffDB
WebApplication1.Models.StudentDb
Security

Output | dbo.Students [Data] | Program.cs | 20200825093243_CreateSchoolDB.cs

Demo1.Program | CreateHostBuilder(string[]

```
0 references
public class Program
{
    0 references
    public static void Main(string[] args)
    {
        using (var context = new SchoolContext())
        {

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

            context.Students.Add(std);
            context.SaveChanges();
        }
        CreateHostBuilder(args).Build().Run();
    }

    1 reference
```

Package Manager Console | Output | dbo.Students [Data]

Max Rows: 1000

StudentId	Name
1	Bill
NULL	NULL

Conventions in Entity Framework Core

- Conventions are default rules using which Entity Framework builds a model based on your domain (entity) classes.
- In the previous slides, EF Core API creates a database schema based on domain and context classes, without any additional configurations because domain classes were following the conventions.

SchoolContext.cs | NuGet: Demo1 | Demo1

Demo1.Models.SchoolContext | SchoolContext()

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace Demo1.Models
{
    1 reference
    public class SchoolContext : DbContext
    {
        0 references
        public SchoolContext()
        {
        }

        0 references
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Server=(localdb)\MSSQLLocalDB;Database=SchoolDB;
            Trusted_Connection=True;");
        }
        0 references
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
        }
        //entities
```

SchoolContext.cs | NuGet: Demo1 | Demo1

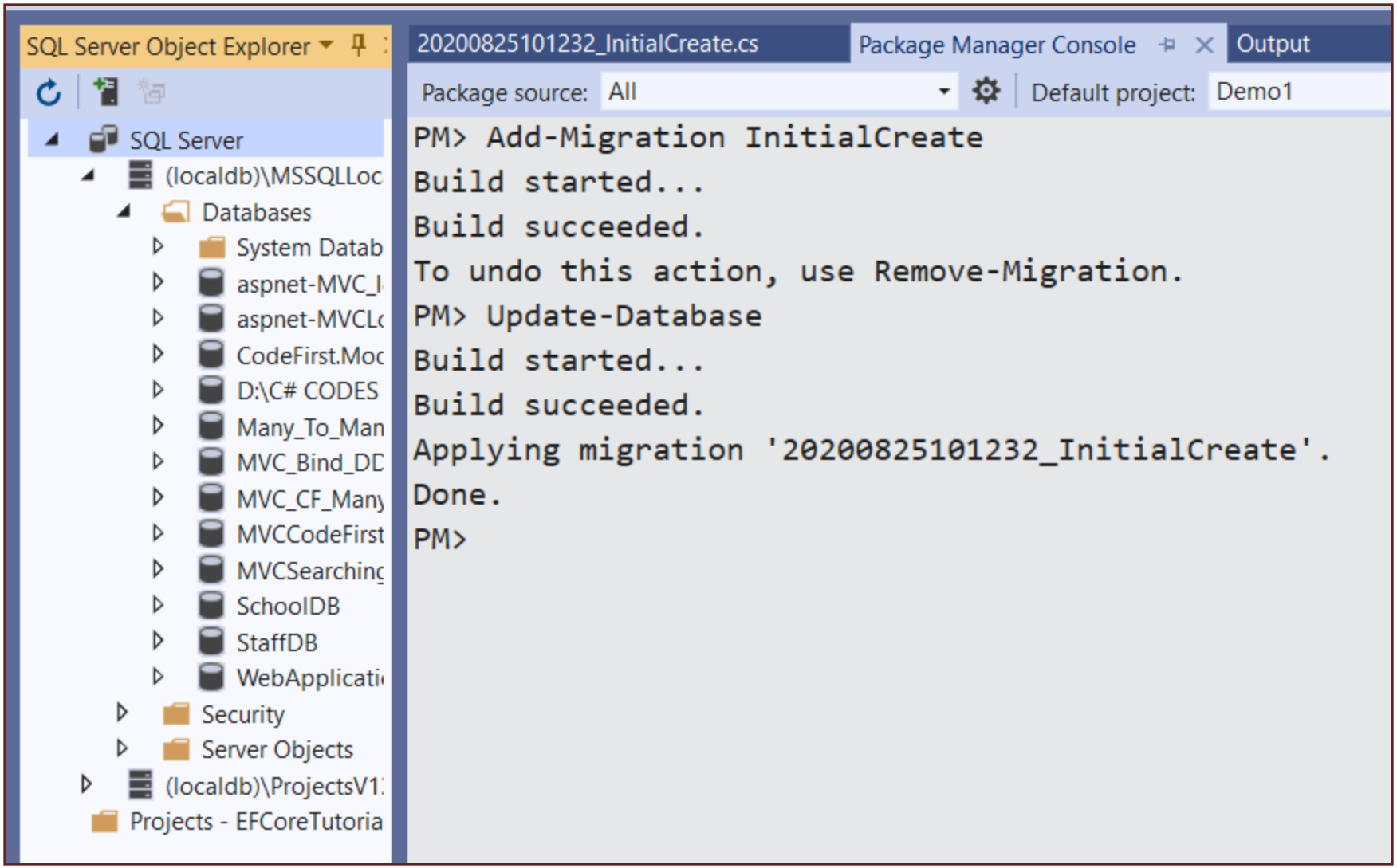
Demo1.Models.SchoolContext

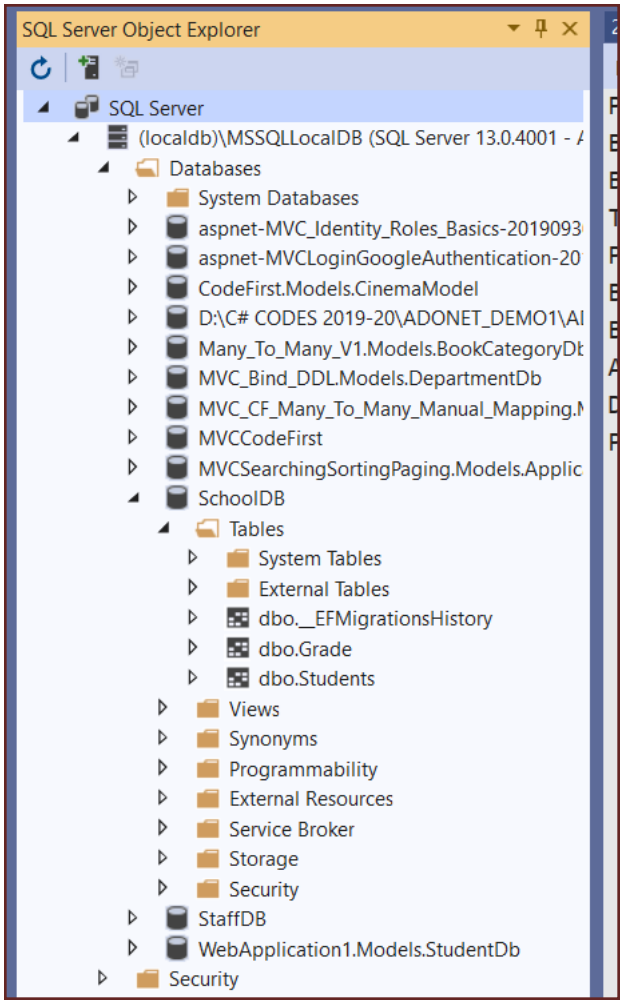
```
        //entities
        0 references
        public DbSet<Student> Students { get; set; }
    }

    2 references
    public class Student
    {
        0 references
        public int StudentId { get; set; }
        0 references
        public string FirstName { get; set; }
        0 references
        public string LastName { get; set; }
        0 references
        public DateTime DateOfBirth { get; set; }
        0 references
        public byte[] Photo { get; set; }
        0 references
        public decimal Height { get; set; }
        0 references
        public float Weight { get; set; }

        0 references
        public int GradeId { get; set; }
        0 references
        public Grade Grade { get; set; }
    }
```

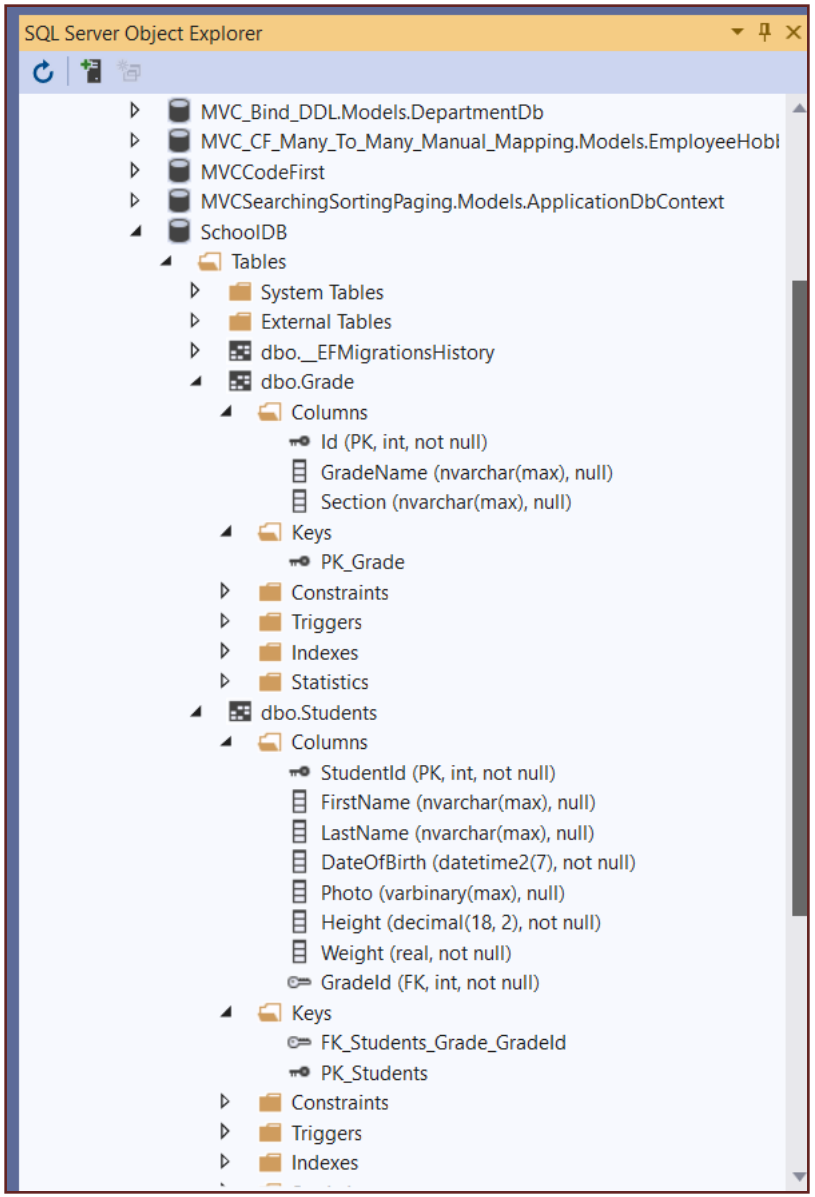
SchoolContext.cs | NuGet: Demo1 | Demo1

Demo1.Models.SchoolContext

```
    }

    1 reference
    public class Grade
    {
        0 references
        public int Id { get; set; }
        0 references
        public string GradeName { get; set; }
        0 references
        public string Section { get; set; }

        0 references
        public IList<Student> Students { get; set; }
    }

}
```


SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLoc
Databases
System Datab
aspnet-MVC_I
aspnet-MVCLc
CodeFirst.Moc
D:\C# CODES
Many_To_Man
MVC_Bind_DD
MVC_CF_Many
MVCCodeFirst
MVCSearching
SchoolDB
StaffDB
WebApplicati
Security
Server Objects
(localdb)\ProjectsV1
Projects - EFCoreTutoria
20200825101232_InitialCreate.cs
Package Manager Console
Output
Package source: All
Default project: Demo1
PM> Add-Migration InitialCreate
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> Update-Database
Build started...
Build succeeded.
Applying migration '20200825101232_InitialCreate'.
Done.
PM>


SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 13.0.4001 - A
Databases
System Databases
aspnet-MVC_Identity_Roles_Basics-2019093
aspnet-MVCLoginGoogleAuthentication-20
CodeFirst.Models.CinemaModel
D:\C# CODES 2019-20\ADONET_DEMO1\AI
Many_To_Many_V1.Models.BookCategoryDt
MVC_Bind_DDL.Models.DepartmentDb
MVC_CF_Many_To_Many_Manual_Mapping.M
MVCCodeFirst
MVCSearchingSortingPaging.Models.Applic
SchoolDB
Tables
System Tables
External Tables
dbo.__EFMigrationsHistory
dbo.Grade
dbo.Students
Views
Synonyms
Programmability
External Resources
Service Broker
Storage
Security
StaffDB
WebApplication1.Models.StudentDb
Security


SQL Server Object Explorer
MVC_Bind_DDL.Models.DepartmentDb
MVC_CF_Many_To_Many_Manual_Mapping.Models.EmployeeHobl
MVCCodeFirst
MVCSearchingSortingPaging.Models.ApplicationDbContext
SchoolDB
Tables
System Tables
External Tables
dbo.__EFMigrationsHistory
dbo.Grade
Columns
Id (PK, int, not null)
GradeName (nvarchar(max), null)
Section (nvarchar(max), null)
Keys
PK_Grade
Constraints
Triggers
Indexes
Statistics
dbo.Students
Columns
StudentId (PK, int, not null)
FirstName (nvarchar(max), null)
LastName (nvarchar(max), null)
DateOfBirth (datetime2(7), not null)
Photo (varbinary(max), null)
Height (decimal(18, 2), not null)
Weight (real, not null)
GradeId (FK, int, not null)
Keys
FK_Students_Grade_GradeId
PK_Students
Constraints
Triggers
Indexes

Scalar Property

- The primitive type properties are called scalar properties.
- Each scalar property maps to a column in the database table which stores an actual data.
- For example, StudentID, StudentName, DateOfBirth, Photo, Height, Weight are the scalar properties in the Student entity class.

Navigation Property

- The navigation property represents a relationship to another entity.
- There are two types of navigation properties:
  - Reference Navigation and Collection Navigation

Reference Navigation Property

```
public class Student
{
    // scalar properties
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[]  Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    //reference navigation property
    public Grade Grade { get; set; }
}
```

Collection Navigation Property

SchoolContext.cs | NuGet: Demo1 | Demo1

Demo1.Models.SchoolContext

```
    }

    1 reference
    public class Grade
    {
        0 references
        public int Id { get; set; }
        0 references
        public string GradeName { get; set; }
        0 references
        public string Section { get; set; }

        0 references
        public IList<Student> Students { get; set; }
    }

}
```

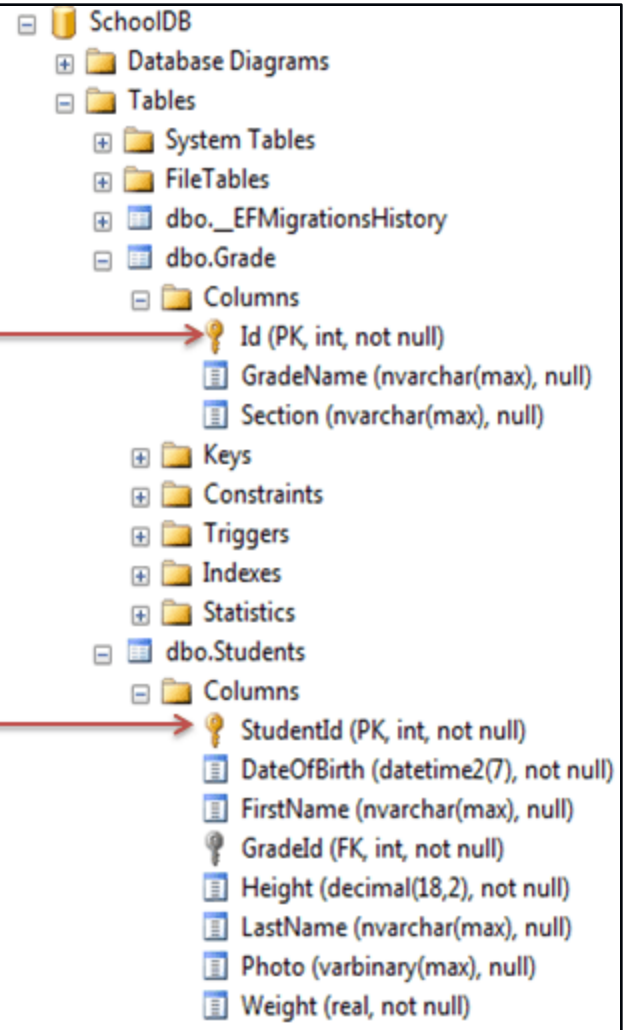
C# Data Type	Mapping to SQL Server Data Type
int	int
string	nvarchar(Max)
decimal	decimal(18,2)
float	real
byte[]	varbinary(Max)
datetime	datetime
bool	bit
byte	tinyint
short	smallint
long	bigint
double	float
char	No mapping
sbyte	No mapping (throws exception)
object	No mapping

Primary Key

- EF Core will create the primary key column for the property named
  - Id or
  - <Entity Class Name>Id (case insensitive).
- For example, EF Core will create a column as PrimaryKey in the Students table if the Student class includes a property named
  - id, ID, iD, Id,
  - studentid, StudentId, STUDENTID, or sTUdentID.

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

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

```
public class Student
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

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

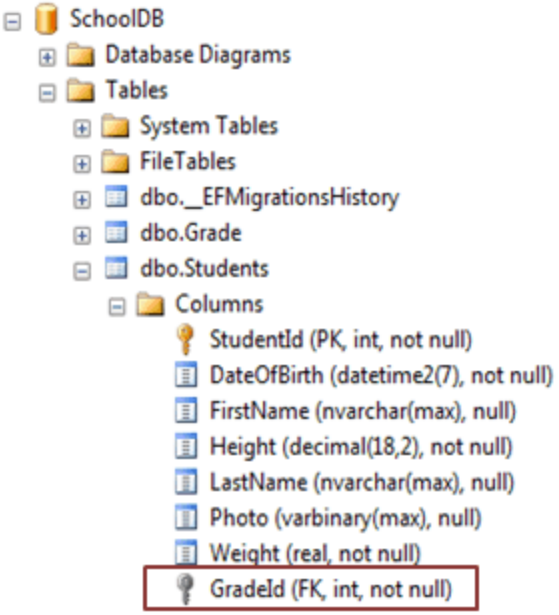


Foreign Key

- As per the foreign key convention, EF Core API will create a foreign key column for each reference navigation property in an entity with one of the following naming patterns.

```
<Reference Navigation Property Name>Id
<Reference Navigation Property Name><Principal Primary Key Property Name>
```

```
public class Student  Dependent Entity
{
    public int StudentId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public int GradeId { get; set; } Foreign Key Property
    public Grade Grade { get; set; } Reference Property
}

public class Grade  Principal Entity
{
    public int Id { get; set; } Primary Key Property
    public string GradeName { get; set; }
    public string Section { get; set; }

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



Reference Property Name in Dependent Entity	Foreign Key Property Name in Dependent Entity	Principal Primary Key Property Name	Foreign Key Column Name in DB
Grade	GradeId	GradeId	GradeId
Grade	-	GradeId	GradeId
Grade	-	Id	GradeId
CurrentGrade	CurrentGradeId	GradeId	CurrentGradeId
CurrentGrade	-	GradeId	CurrentGradeGradeId
CurrentGrade	-	Id	CurrentGradeId
CurrentGrade	GradeId	Id	GradeId

```
<Reference Navigation Property Name>Id
```

```
<Reference Navigation Property Name><Principal Primary Key Property Name>
```

One-to-Many Relationship Conventions in Entity Framework Core

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 here.

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

    public Grade Grade { get; set; }
}

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

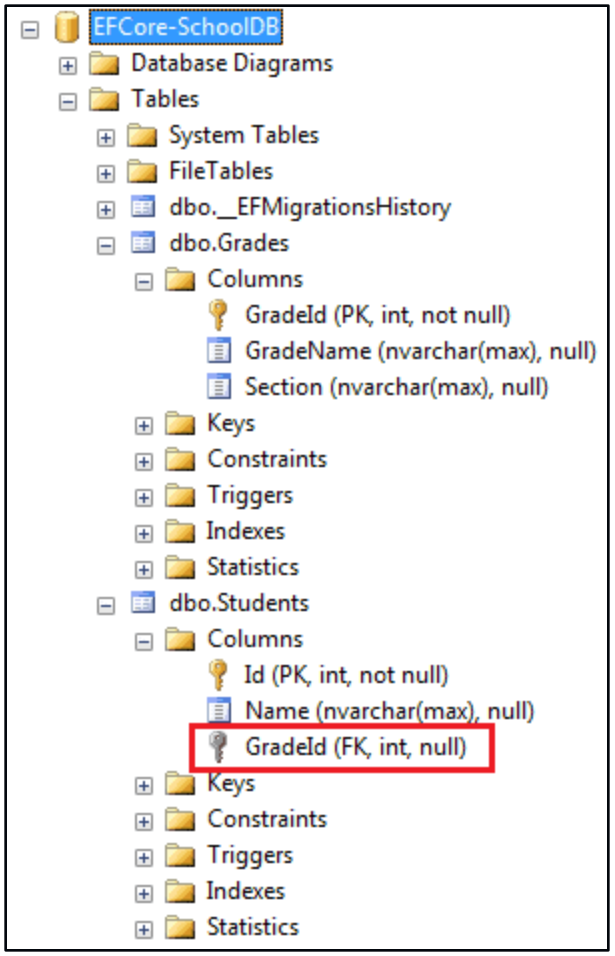
Convention 1

- 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.

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

    public Grade Grade { get; set; }
}

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

Convention 1

- 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 here.



Convention 2

- In the example here, the Grade entity includes a collection navigation property of type

ICollection<student>.

- This will allow us to add multiple Student entitie s to a Grade entity, which results in a one-to-many relationship between

Students and Grades table s in the database, same as in convention 1.

```
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; }
}
```

Convention 3

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

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

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }

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

Convention 3

- In the example here, 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.

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

    public Grade Grade { get; set; }
}

public class Grade
{
    public int GradeID { get; set; }
    public string GradeName { get; set; }

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

Convention 4

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

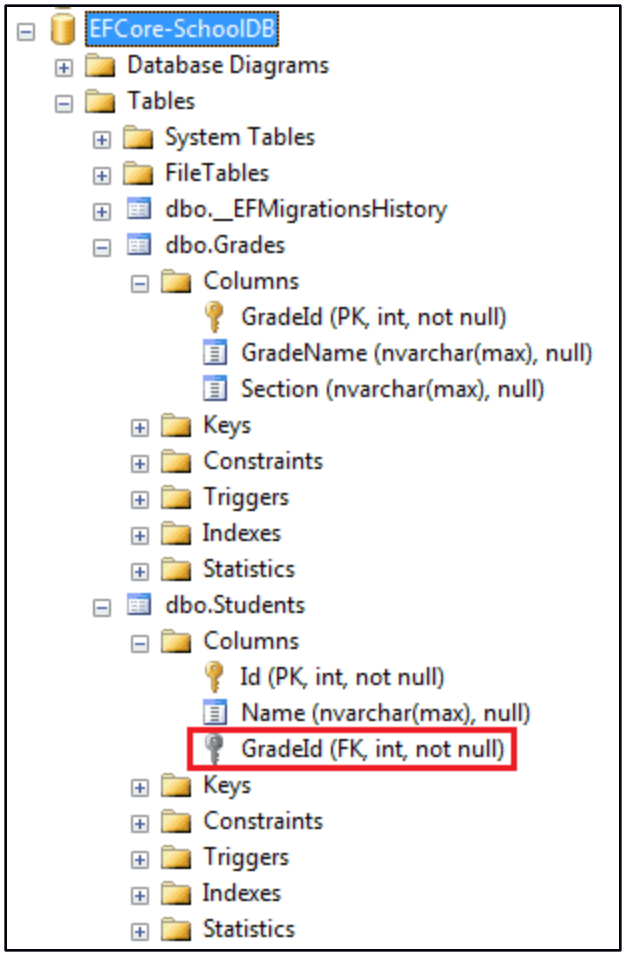
```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

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

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

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

Convention 4

- In this 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.



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.
- In EF Core, a one-to-one relationship requires a reference navigation property at both sides.

One to One

- In the example here, 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.

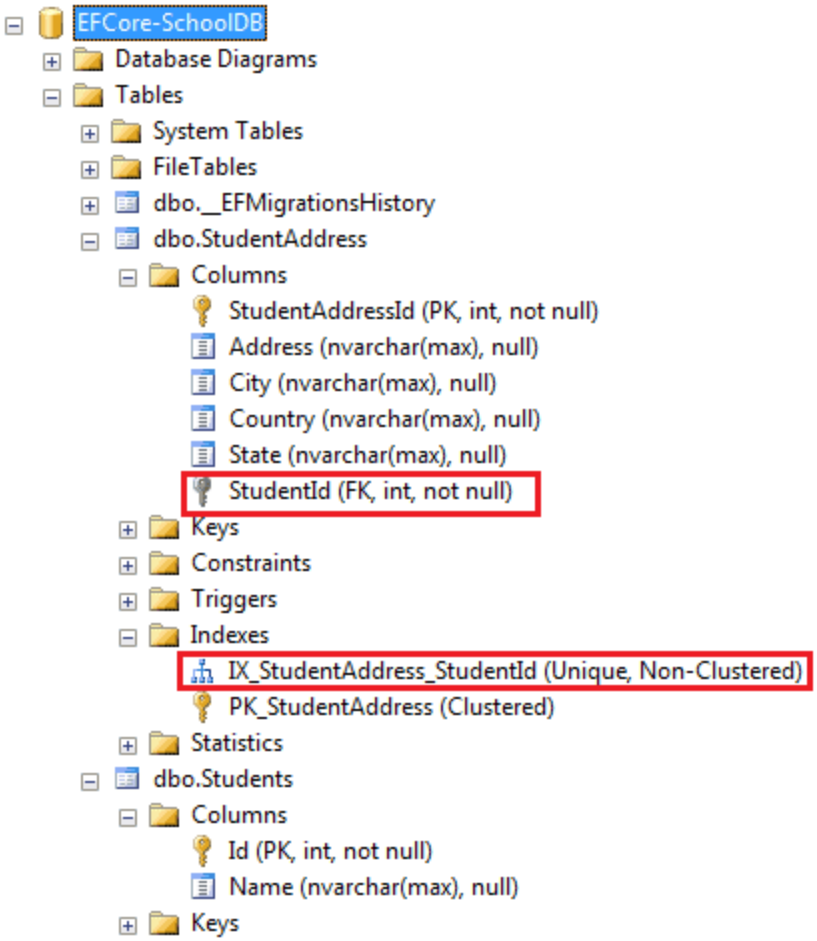
```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }

    public StudentAddress Address { get; set; }
}

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

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

One to One

- This will result in a one-to-one relationship in corresponding tables Students and StudentAddresses in the database, as shown here.
- EF Core creates a unique index on the NotNull foreign key column StudentId

in the StudentAddresses table, as shown here.

- This ensures that the value of the foreign key column StudentId must be unique in the StudentAddress table, which is necessary of a one-to-one relationship.



Configurations in Entity Framework Core

- We discussed about default Conventions in EF Core.
- 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 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.

1. By using Data Annotation Attributes
2. By using Fluent API

Data Annotation Attributes

- Data Annotations is a simple attribute based configuration method where different .NET attributes can be applied to domain classes and properties to configure the model.

```
[Table("StudentInfo")]
public class Student
{
    public Student() { }

    [Key]
    public int SID { get; set; }

    [Column("Name", TypeName="ntext")]
    [MaxLength(20)]
    public string StudentName { get; set; }

    [NotMapped]
    public int? Age { get; set; }


    public int StdId { get; set; }

    [ForeignKey("StdId")]
    public virtual Standard Standard { get; set; }
}
```

Fluent API

- Another way to configure domain classes is by using Entity Framework Fluent API.
- Entity Framework Fluent API is used to configure domain classes to override conventions.
- EF Fluent API is based on a Fluent API design pattern (a.k.a Fluent Interface) where the result is formulated by method chaining.
- In Entity Framework Core, the ModelBuilder class acts as a Fluent API. By using it, we can configure many different things, as it provides more configuration options than data annotation attributes.
- **Note:** Fluent API configurations have higher precedence than data annotation attributes.

Cont.

- Entity Framework Core Fluent API configures the following aspects of a model:

1. Model Configuration
   - Configures an EF model to database mappings.
   - Configures the default Schema, DB functions, additional data annotation attributes and entities to be excluded from mapping.
2. Entity Configuration
   - Configures entity to table and relationships mapping e.g. PrimaryKey, AlternateKey, Index, table name, one-to-one, one-to-many, many-to-many relationships etc.
3. Property Configuration
   - Configures property to column mapping e.g. column name, default value, nullability, Foreignkey, data type, concurrency column etc.

```
public class SchoolDBContext: DbContext
{
    public DbSet<Student> Students { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //Write Fluent API configurations here

        //Property Configurations
        modelBuilder.Entity<Student>()
                .Property(s => s.StudentId)
                .HasColumnName("Id")
                .HasDefaultValue(0)
                .IsRequired();
    }
}
```

```
//Fluent API method chained calls
modelBuilder.Entity<Student>()
        .Property(s => s.StudentId)
        .HasColumnName("Id")
        .HasDefaultValue(0)
        .IsRequired();

//Separate method calls
modelBuilder.Entity<Student>().Property(s => s.StudentId).HasColumnName("Id");
modelBuilder.Entity<Student>().Property(s => s.StudentId).HasDefaultValue(0);
modelBuilder.Entity<Student>().Property(s => s.StudentId).IsRequired();
```

One-to-Many Relationships using Fluent API in EF Core

- We learned about the Conventions for One-to-Many Relationship.
- Generally, we don't need to configure one-to-many relationships because EF Core includes enough conventions which will automatically configure them.
- However, we can use Fluent API to configure the one-to-many relationship if you decide to have all the EF configurations in Fluent API for easy maintenance.

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

    public int CurrentGradeId { 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; }
}
```

```
public class SchoolContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=.\\SQLEXPRESS;Database=EFCore-SchoolDB;Trusted_Co
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>()
            .HasOne<Grade>(s => s.Grade)
            .WithMany(g => g.Students)
            .HasForeignKey(s => s.CurrentGradeId);
    }

    public DbSet<Grade> Grades { get; set; }
    public DbSet<Student> Students { get; set; }
}
```

```
modelBuilder.Entity<Student>()
    .HasOne<Grade>(s => s.Grade)
    .WithMany(g => g.Students)
    .HasForeignKey(s => s.CurrentGradeId);
```

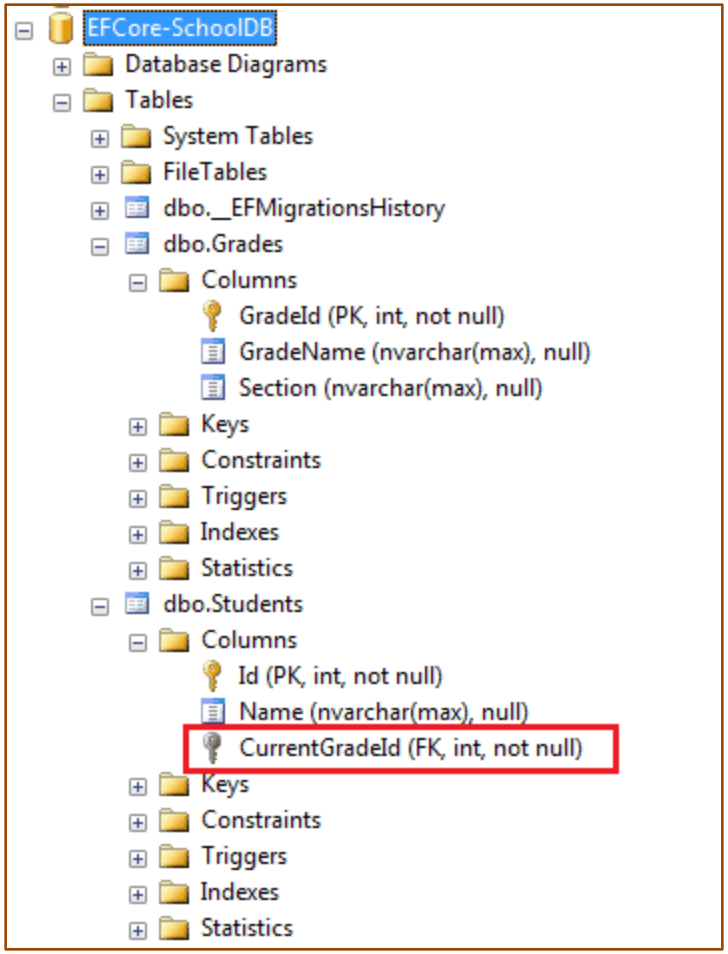
- First, we need to start configuring with one entity class, either Student or Grade. So, modelBuilder.Entity<student>() starts with the Student entity.
- Then, .HasOne<Grade>(s => s.Grade) specifies that the Student entity includes a Grade type property named Grade.
- Now, we need to configure the other end of the relationship, the Grade entity. The .WithMany(g => g.Students) specifies that the Grade entity class includes many Student entities. Here, WithMany infers collection navigation property.
- The .HasForeignKey<int>(s => s.CurrentGradeId); specifies the name of the foreign key property CurrentGradeId. This is optional. Use it only when you have the foreign key Id property in the dependent class.

```
modelBuilder.Entity<Student>()
    .HasOne<Grade>(s => s.Grade)
    .WithMany(g => g.Students)
    .HasForeignKey(s => s.CurrentGradeId);
```

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

    public int CurrentGradeId { get; set; }
    public Grade Grade { get; set; }
}
```



```
public class Grade
{
    public Grade()
    {
        Students = new HashSet<Student>();
    }

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

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


EFCore-SchoolDB
Database Diagrams
Tables
System Tables
FileTables
dbo.__EFMigrationsHistory
dbo.Grades
Columns
GradeId (PK, int, not null)
GradeName (nvarchar(max), null)
Section (nvarchar(max), null)
Keys
Constraints
Triggers
Indexes
Statistics
dbo.Students
Columns
Id (PK, int, not null)
Name (nvarchar(max), null)
CurrentGradeId (FK, int, not null)
Keys
Constraints
Triggers
Indexes
Statistics

```
modelBuilder.Entity<Grade>()
    .HasMany<Student>(g => g.Students)
    .WithOne(s => s.Grade)
    .HasForeignKey(s => s.CurrentGradeId);
```

Configure Cascade Delete using Fluent API

- Cascade delete automatically deletes the child row when the related parent row is deleted.
  - For example, if a Grade is deleted, then all the Students in that grade should also be deleted from the database automatically.

```
modelBuilder.Entity<Grade>()
    .HasMany<Student>(g => g.Students)
    .WithOne(s => s.Grade)
    .HasForeignKey(s => s.CurrentGradeId)
    .OnDelete(DeleteBehavior.Cascade);
```

One-to-One Relationships using Fluent API in EF Core

- Generally, you don't need to configure one-to-one relationships manually because EF Core includes Conventions for One-to-One Relationships.
- However, if the key or foreign key properties do not follow the convention, then you can use data annotation attributes or Fluent API to configure a one-to-one relationship between the two entities.

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

    public StudentAddress Address { get; set; }
}

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

    public int AddressOfStudentId { get; set; }
    public Student Student { get; set; }
}
```

```
public class SchoolContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=.\\SQLEXPRESS;Database=EFCore-SchoolDB;Truste
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>()
            .HasOne<StudentAddress>(s => s.Address)
            .WithOne(ad => ad.Student)
            .HasForeignKey<StudentAddress>(ad => ad.AddressOfStudentId);
    }

    public DbSet<Student> Students { get; set; }
    public DbSet<StudentAddress> StudentAddresses { get; set; }
}
```

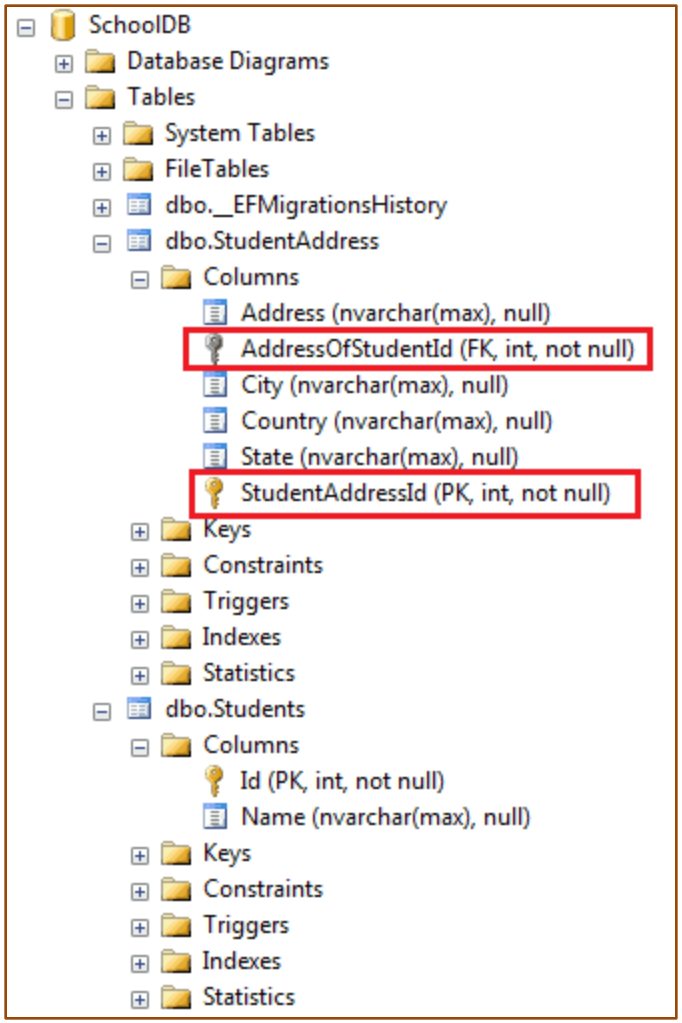
```
modelBuilder.Entity<Student>()
    .HasOne<StudentAddress>(s => s.Address)
    .WithOne(ad => ad.Student)
    .HasForeignKey<StudentAddress>(ad => ad.AddressOfStudentId);
```

- modelBuilder.Entity<Student>() starts configuring the Student entity.
- The .HasOne<StudentAddress>(s => s.Address) method specifies that the Student entity includes one StudentAddress reference property using a lambda expression.
- .WithOne(ad => ad.Student) configures the other end of the relationship, the StudentAddress entity. It specifies that the StudentAddress entity includes a reference navigation property of Student type.
- .HasForeignKey<StudentAddress>(ad => ad.AddressOfStudentId) specifies the foreign key property name.

```
modelBuilder.Entity<Student>()
    .HasOne<StudentAddress>(s => s.Address)
    .WithOne(sa => sa.Student)
    .HasForeignKey<StudentAddress>(sa => sa.AddressOfStudentId);
```

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

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

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

    public int AddressOfStudentId { get; set; }
    public Student Student { get; set; }
}
```


SchoolDB
Database Diagrams
Tables
System Tables
FileTables
dbo.__EFMigrationsHistory
dbo.StudentAddress
Columns
Address (nvarchar(max), null)
AddressOfStudentId (FK, int, not null)
City (nvarchar(max), null)
Country (nvarchar(max), null)
State (nvarchar(max), null)
StudentAddressId (PK, int, not null)
Keys
Constraints
Triggers
Indexes
Statistics
dbo.Students
Columns
Id (PK, int, not null)
Name (nvarchar(max), null)
Keys
Constraints
Triggers
Indexes
Statistics

```
modelBuilder.Entity<StudentAddress>()
    .HasOne<Student>(ad => ad.Student)
    .WithOne(s => s.Address)
    .HasForeignKey<StudentAddress>(ad => ad.AddressOfStudentId);
```

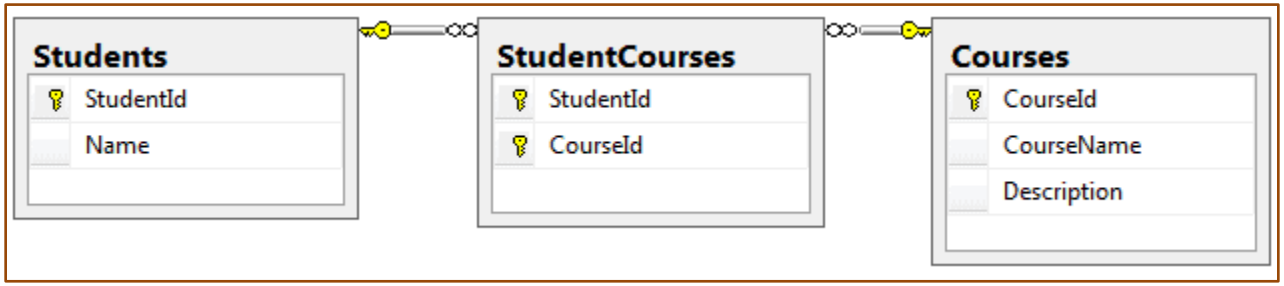
Many-to-Many Relationships in EF Core

- Let's implement a many-to-many relationship between Student and Course entities, where one student can enroll for many courses and, in the same way, one course can be joined by many students.

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

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

Cont.

- The many-to-many relationship in the database is represented by a joining table which includes the foreign keys of both tables.
- Also, these foreign keys are composite primary keys.



Convention

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

Fluent API

- In the Entity Framework 6.x or prior, EF API used to create the joining table for many-to-many relationships.
  - We need not to create a joining entity for a joining table (however, we can of course create a joining entity explicitly in EF 6).
- In Entity Framework Core, this has not been implemented yet.
  - We must create a joining entity class for a joining table.
  - The joining entity for the above Student and Course entities should include a foreign key property and a reference navigation property for each entity.

Cont.

- The steps for configuring many-to-many relationships

1. Define a new joining entity class which includes the foreign key property and the reference navigation property for each entity.
2. Define a one-to-many relationship between other two entities and the joining entity, by including a collection navigation property in entities at both sides (Student and Course, in this case).
3. Configure both the foreign keys in the joining entity as a composite key using Fluent API.

```
public class StudentCourse
{
    public int StudentId { get; set; }
    public Student Student { get; set; }

    public int CourseId { get; set; }
    public Course Course { get; set; }
}
```

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

    public IList<StudentCourse> StudentCourses { get; set; }
}

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

    public IList<StudentCourse> StudentCourses { get; set; }
}
```

```
public class SchoolContext : DbContext
{
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer("Server=.\\SQLEXPRESS;Database=EFCore-SchoolDB;Trus
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<StudentCourse>().HasKey(sc => new { sc.StudentId, sc.Course
    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Course> Courses { get; set; }
    public DbSet<StudentCourse> StudentCourses { get; set; }
}
```