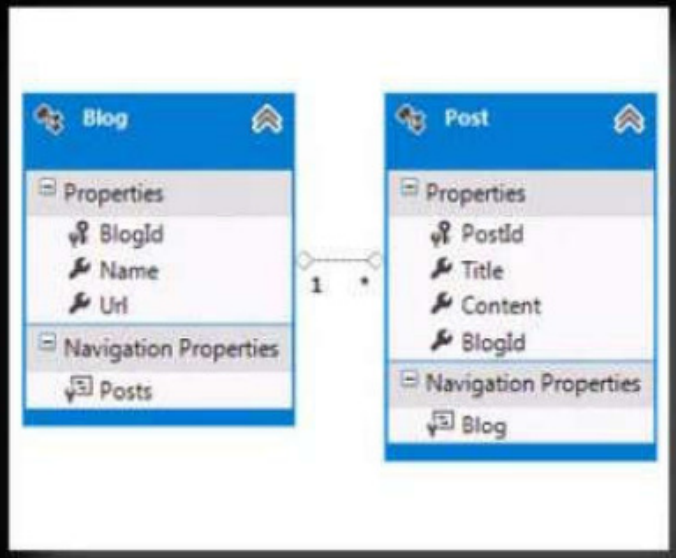
MCSE 541: Web Computing and Data Mining

Entity Framework Core in ASP.NET
Code First EF Convention

Prof. Dr. Shamim Akhter
shamimakhter@iubat.edu

Creating a Model

- An EF model
  - stores the details about how application classes and properties map to database tables and columns.
- There are two main ways to create an EF model:
  - **Using Code First**:
    - Developer writes code to specify the model.
    - EF generates the models and mappings at runtime based on entity classes and additional model configuration provided by the developer.
  - **Using the EF Designer**:
    - Developer draws boxes and lines to specify the model.
    - The resulting model is stored as XML in a file with the EDMX (Entity data model XML) extension.
    - The application's domain objects are typically generated automatically from the conceptual model.

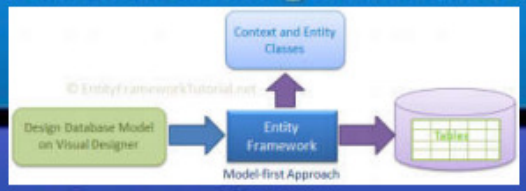



```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

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

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public Blog Blog { get; set; }
}
```

New Database

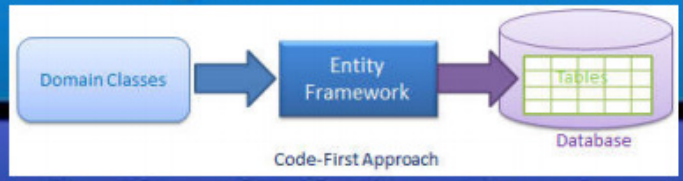
Model First

- Create model in designer
- Database created from model
- Classes auto-generated from model



Code First (New Database)

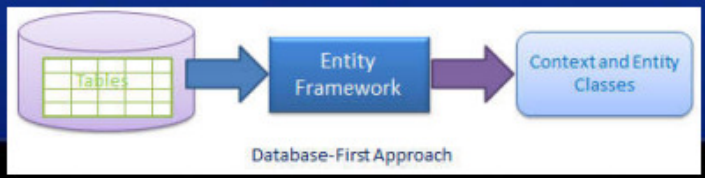
- Define classes & mapping in code
- Database created from model
- Use Migrations to evolve database



Existing Database

Database First

- Reverse engineer model in designer
- Classes auto-generated from model



Code First (Existing Database)

- Define classes & mapping in code
- Reverse engineer tools available

Code First Model-Simple1

Model Classes

```
1 reference
public class Student
{
    0 references | 0 exceptions
    public int StudentId { get; set; }
    0 references | 0 exceptions
    public string StudentName { get; set; }

    0 references | 0 exceptions
    public DateTime DateOfBirth { get; set; }

    0 references | 0 exceptions
    public byte[] Photo { get; set; }

    0 references | 0 exceptions
    public decimal Height { get; set; }

    0 references | 0 exceptions
    public float Weight { get; set; }

}
```

Domain Classes

Entity

- a class that maps to a database table.
- must be included as a DbSet<TEntity> type property in the DbContext class.
- EF API maps each entity to a table and each property of an entity to a column in the database.

Grade Model Class

```
1 reference
public class Grade
{
    0 references | 0 exceptions
    public int GradeId { get; set; }

    0 references | 0 exceptions
    public string GradeName { get; set; }

    0 references | 0 exceptions
    public string Section { get; set; }
}
```

Context Class in Entity Framework

- In order to use Entity Framework to query, insert, update, and delete data using .NET objects,
  - first need to Create a Model
  - maps the entities and relationships that are defined in the model to tables in a database.

- After model class, the primary class your application interacts with is context class.
  - It represents a session with the underlying database to perform CRUD (Create, Read, Update, Delete) operations.
  - Derives from System.Data.Entity.DbContext.
  - It is also used to configure domain classes, database related mappings, change tracking settings, caching, transaction etc.

- Model Context Class:

```
3 references
public class StudentContext:DbContext
{
    0 references | 0 exceptions
    public StudentContext(DbContextOptions<StudentContext> options)
        : base(options)
    {

    }
    0 references | 0 exceptions
    public DbSet<Student> Students { get; set; }
    0 references | 0 exceptions
    public DbSet<Grade> Grades { get; set; }


}
```

Configuration in the Connection String

AppDBContext.cs | Program.cs | Startup.cs | UpdateFriend.cshtml | SearchFriend.cshtml | RemoveFriend.cshtml | AddFriend.cshtml

Schema: https://json.schemastore.org/appsettings

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "AllowedHosts": "*",
  "MyKey":  "Value of myKey from appsettings.json",
  "ConnectionStrings": {
    "FriendDBConnection":  "server=(localdb)\\MSSQLLocalDB; database=FriendDB;Trusted_Connection=true"
  }
}
```

Configuration in the Startup class

- Working with SQL Server LocalDB

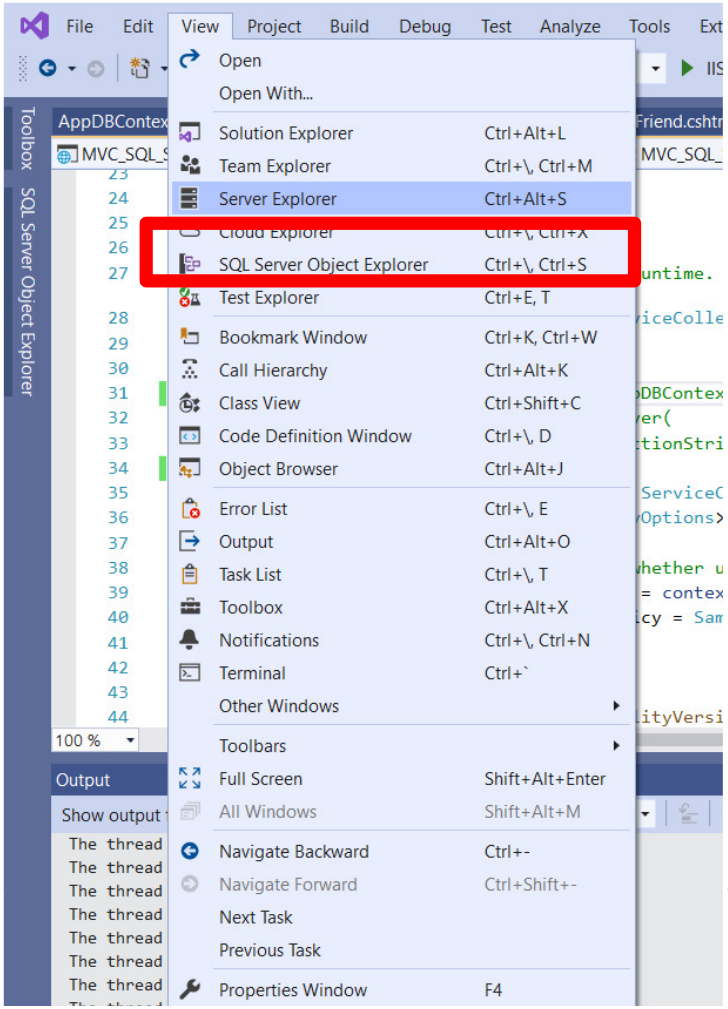
StudentContext

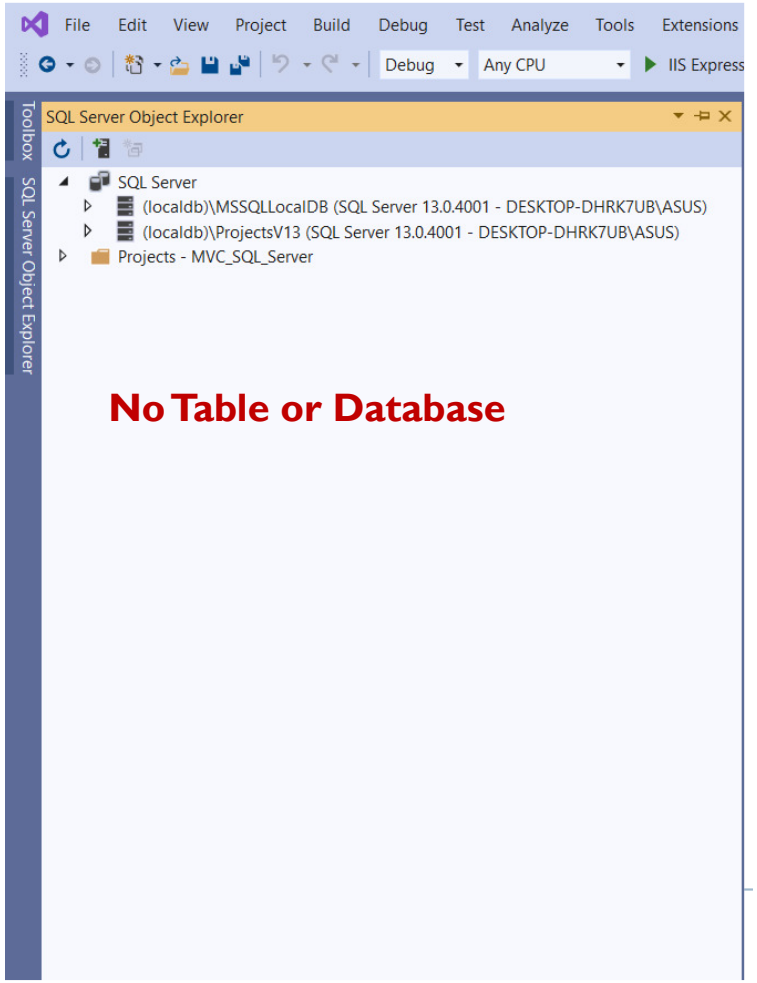
Use AddDb contextPool type AppDBContext method on services collection to add the DB context and the EF services at the beginning of the ConfigureService method.
Call the UseSqlServer method on the options action in its constructor to specify that you want to use SQL Server database provider. UseSqlServer @ Microsoft.EntityFrameworkCore namespace.
At first the connection string must be read.

```
0 references | 0 exceptions
public Startup(IConfiguration configuration)
{
    Configuration = configuration;
}

// This method gets called by the runtime. Use this method to add services to the container.
0 references | 0 exceptions
public void ConfigureServices(IServiceCollection services)
{

    /*services.AddDbContextPool<AppDBContext>(
        options=>options.UseSqlServer(
            Configuration.GetConnectionString("FriendDBConnection")));
    */
    services.AddSingleton<IFriend, ServiceClass>();
    services.Configure<CookiePolicyOptions>(options =>
    {
        // This lambda determines whether user consent for non-essential cookies is needed for a given request.
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });
```

SQL Server Express LocalDB

- LocalDB is a lightweight version of the SQL Server Express Database Engine that is targeted for program development. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB database creates "*.mdf" files in the *C:/Users/<user>* directory.





Adding Migration

Migration is a way to keep the database schema in sync with the EF Core model by preserving data.

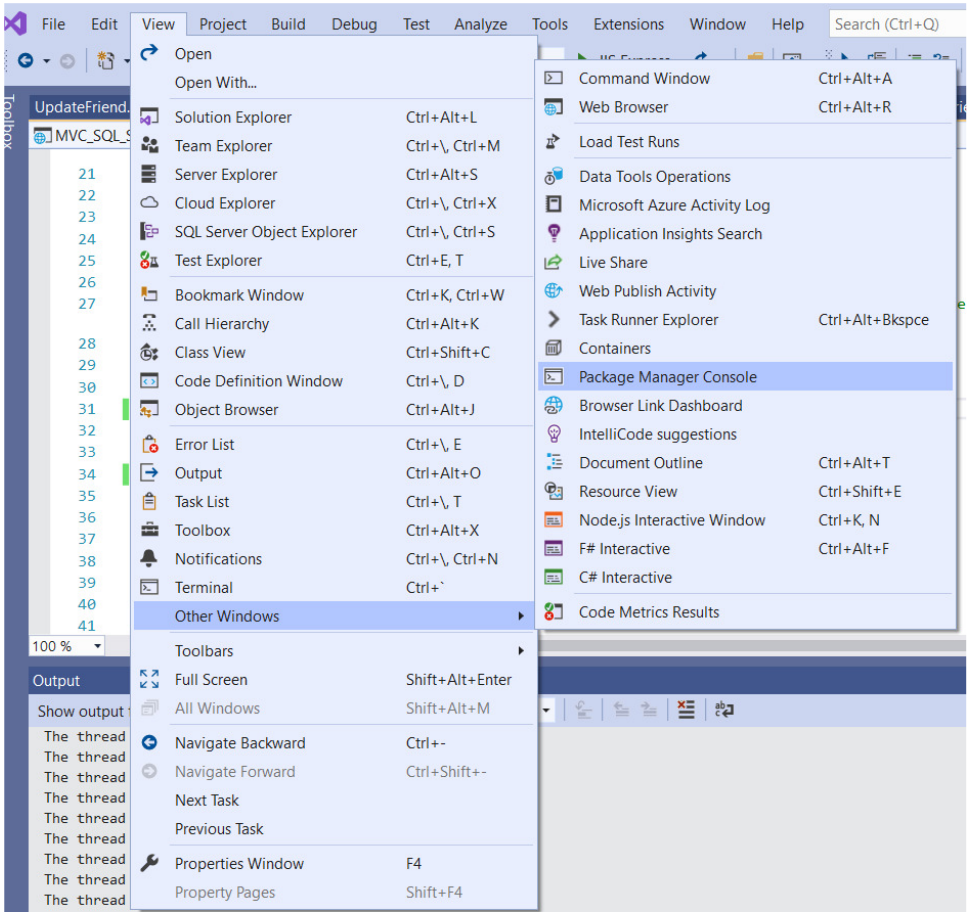
As per the above figure, EF Core API builds the EF Core model from the domain (entity) classes and EF Core migrations will create or update the database schema based on the EF Core model. Whenever you change the domain classes, you need to run migration to keep the database schema up to date.

EF Core migrations are a set of commands which you can execute in NuGet Package Manager Console or in dotnet Command Line Interface (CLI).

The following table lists important migration commands in EF Core.

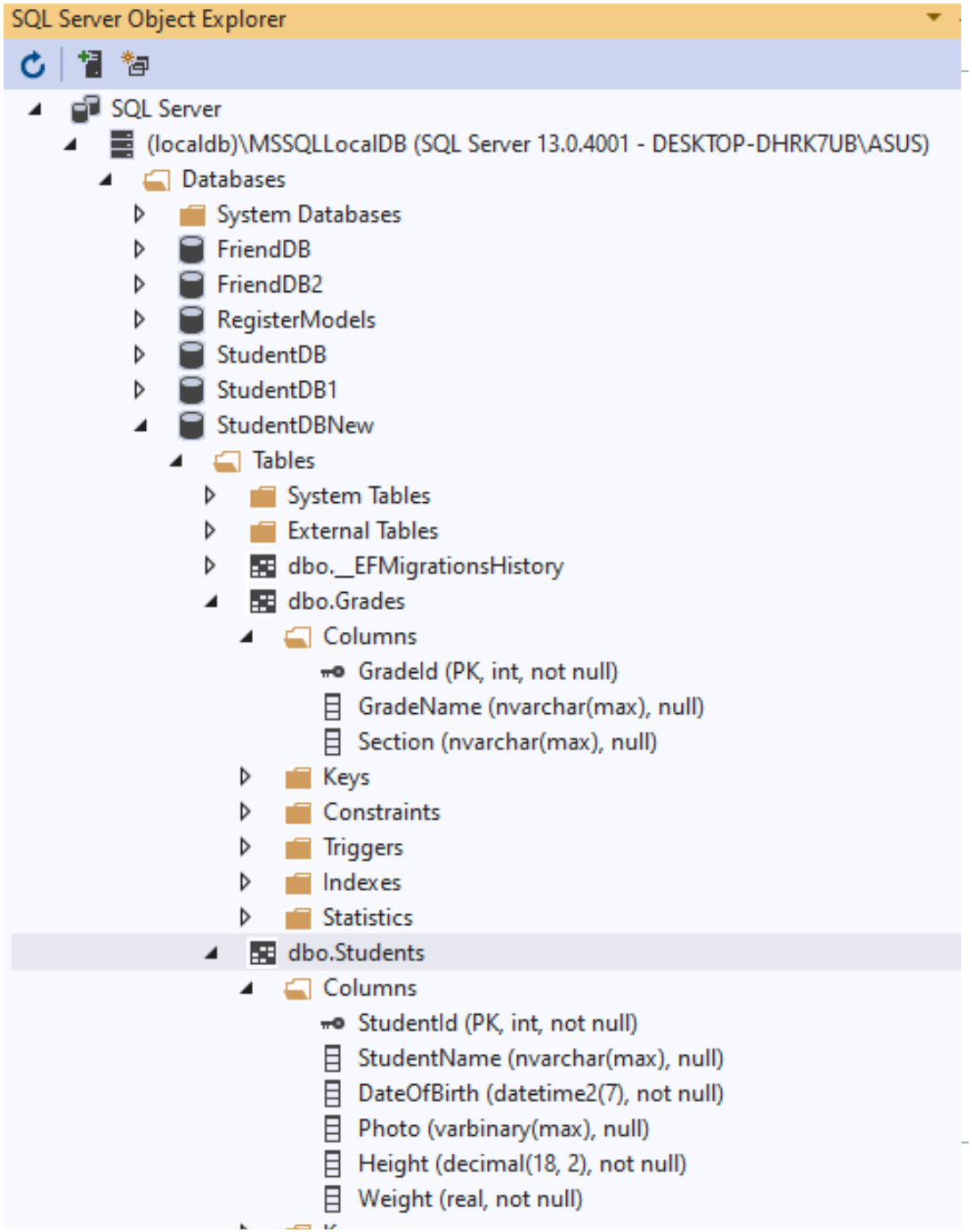
PMC Command	dotnet CLI command	Usage
add-migration <migration name>	Add <migration name>	Creates a migration by adding a migration snapshot.
Remove-migration	Remove	Removes the last migration snapshot.
Update-database	Update	Updates the database schema based on the last migration snapshot.
Script-migration	Script	Generates a SQL script using all the migration snapshots.

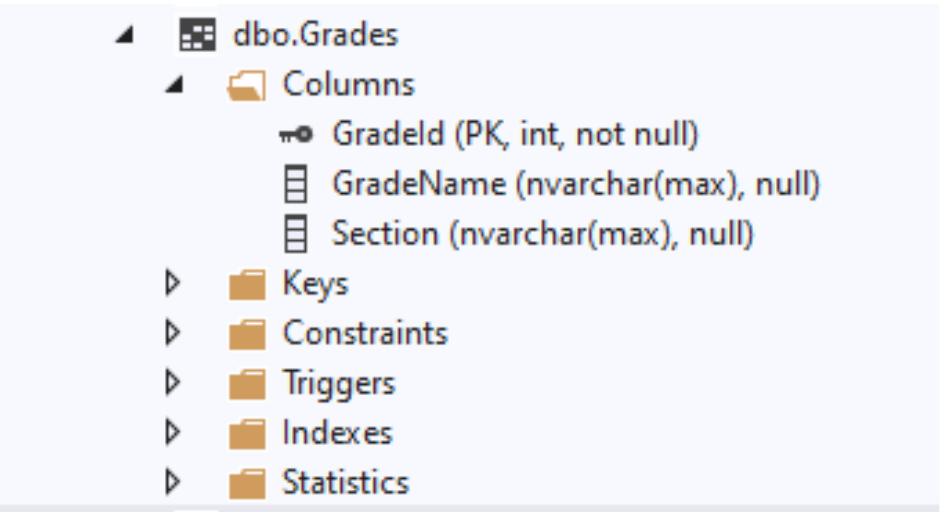
Adding Migration

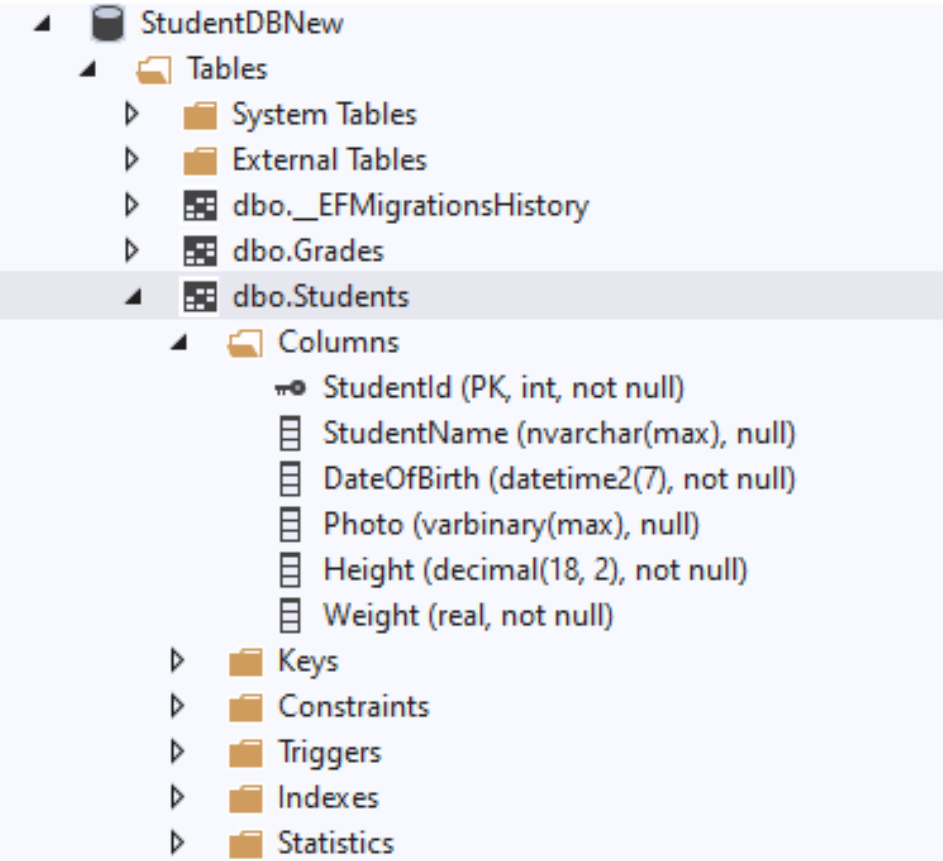


- Add-Migration Initialize
  - A folder will appear
  - Current and future migration will appear there.
- Update-Database

SQL Express Server



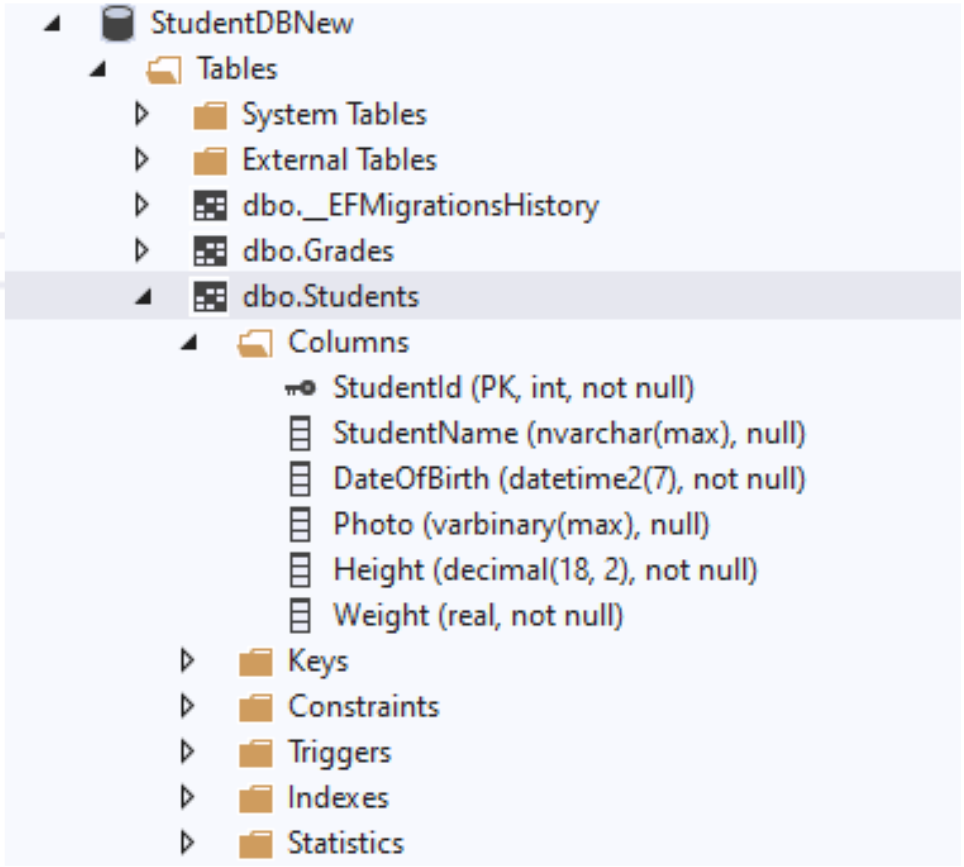




Code First Model-Simple1

Model Classes to Database Tables

```
1 reference
public class Student
{
    0 references | 0 exceptions
    public int StudentId { get; set; }
    0 references | 0 exceptions
    public string StudentName { get; set; }

    0 references | 0 exceptions
    public DateTime DateOfBirth { get; set; }

    0 references | 0 exceptions
    public byte[] Photo { get; set; }

    0 references | 0 exceptions
    public decimal Height { get; set; }

    0 references | 0 exceptions
    public float Weight { get; set; }

}
```



Code First Convention

Default Convention For	Description
Schema	By default, EF creates all the DB objects into the **dbo** schema.
Table Name	<Entity Class Name> + 's' EF will create a DB table with the entity class name suffixed by 's' e.g. `Student` domain class (entity) would map to the `Students` table.
Primary key Name	1) Id 2) <Entity Class Name> + "Id" (case insensitive) EF will create a primary key column for the property named Id or <Entity Class Name> + "Id" (case insensitive).
Foreign key property Name	By default EF will look for the foreign key property with the same name as the principal entity primary key name. If the foreign key property does not exist, then EF will create an FK column in the Db table with <Dependent Navigation Property Name> + "_" + <Principal Entity Primary Key Property Name> e.g. EF will create `Grade_GradeId` foreign key column in the `Students` table if the `Student` entity does not contain foreignkey property for `Grade`.

Null column	EF creates a null column for all reference type properties and nullable primitive properties e.g. string, Nullable<int>, Student, Grade (all class type properties)
Not Null Column	EF creates NotNull columns for Primary Key properties and non-nullable value type properties e.g. int, float, decimal, datetime etc.
DB Columns order	EF will create DB columns in the same order like the properties in an entity class. However, primary key columns would be moved first.
Properties mapping to DB	By default, all properties will map to the database. Use the `[NotMapped]` attribute to exclude property or class from DB mapping.
Cascade delete	Enabled by default for all types of relationships.

Entity Framework Core (EF Core) represents relationships using foreign keys. An entity with a foreign key is the child or dependent entity in the relationship. This entity's foreign key value must match the primary key value (or an alternate key value) of the related principal/parent entity.

If the principal/parent entity is deleted, then the foreign key values of the dependents/children will no longer match the primary or alternate key of *any* principal/parent. This is an invalid state, and will cause a referential constraint violation in most databases.

There are two options to avoid this referential constraint violation:

1. Set the FK values to null
2. Also delete the dependent/child entities

The first option is only valid for optional relationships where the foreign key property (and the database column to which it is mapped) must be nullable.

The second option is valid for any kind of relationship and is known as "cascade delete".

Domain Classes

```
public class Student
{
    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; }

    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; }
}
```

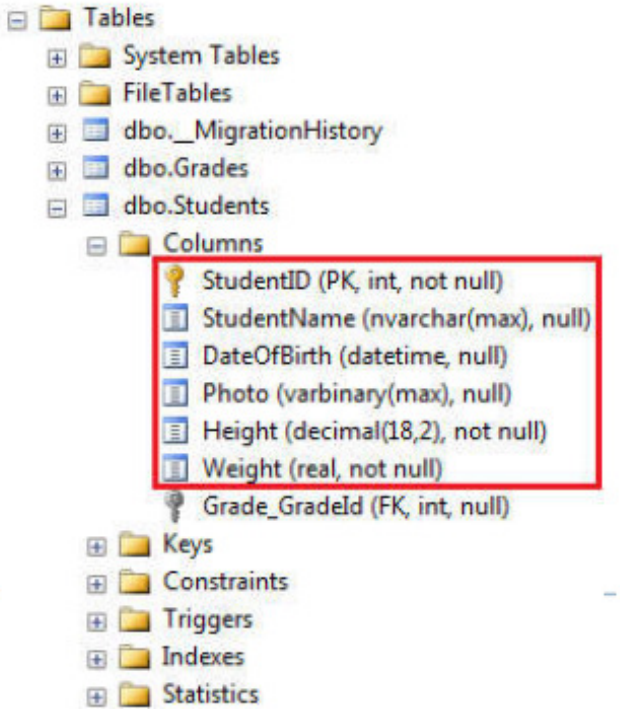
Entity

- a class that maps to a database table.
- must be included as a DbSet<TEntity> type property in the DbContext class.
- EF API maps each entity to a table and each property of an entity to a column in the database.

Entity can include two types of properties: Scalar Properties and Navigation Properties.

- 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.

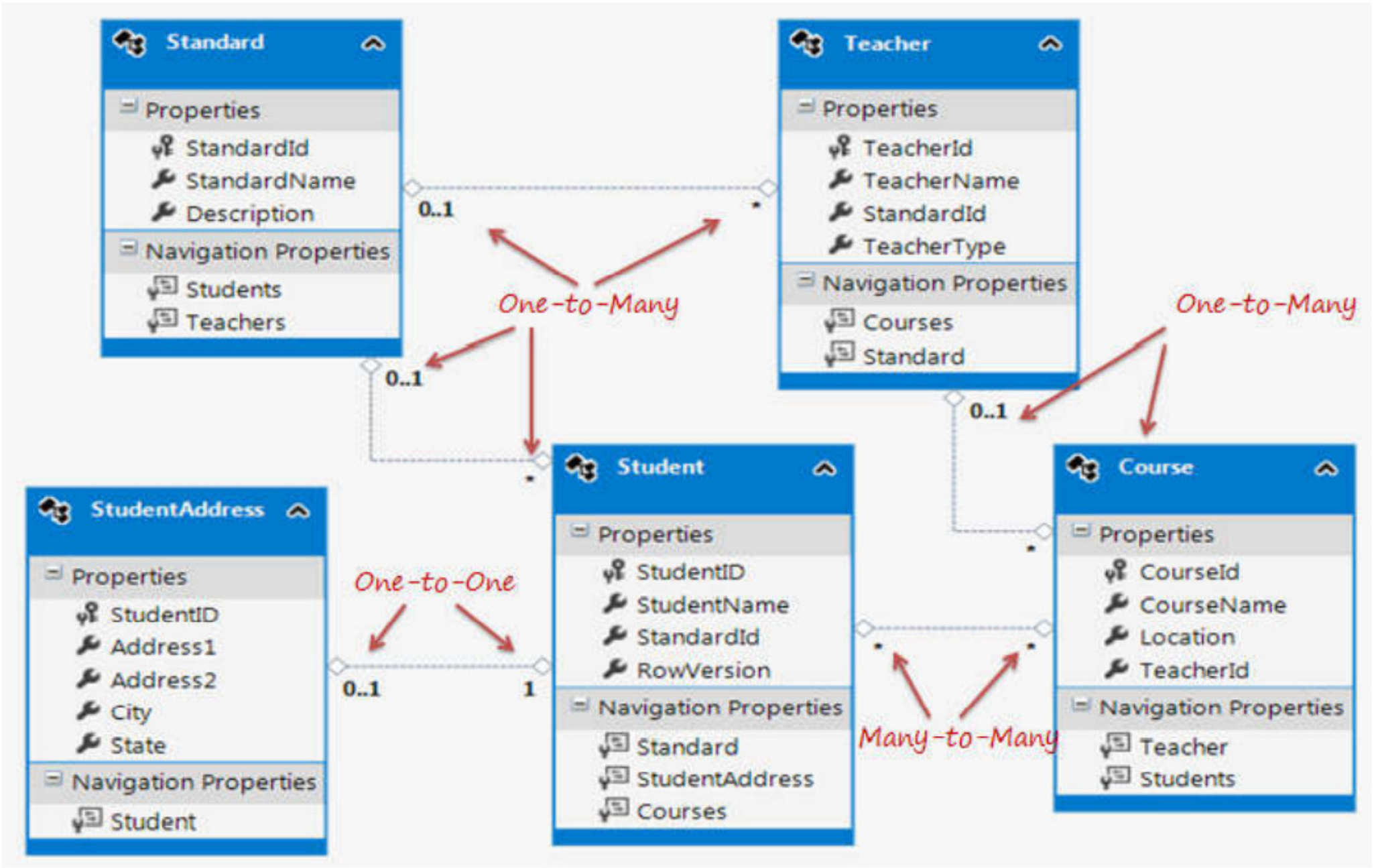
```
public class Student {
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set;}
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }
   // public Grade Grade { get;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



Introduction to Relationships

- Relational databases store data based on the relation between items of data.
- A key benefit to taking a relational view of data is to reduce duplication.
- In a relational database, each *entity* such as the person, the school, the place of work is stored in separate tables and unique instances if the entity are identified by a Primary Key value.
- Relationships or associations between entities are defined in a database by the existence of Foreign Keys.

Entity Relationship (association) Convention



Navigation Property

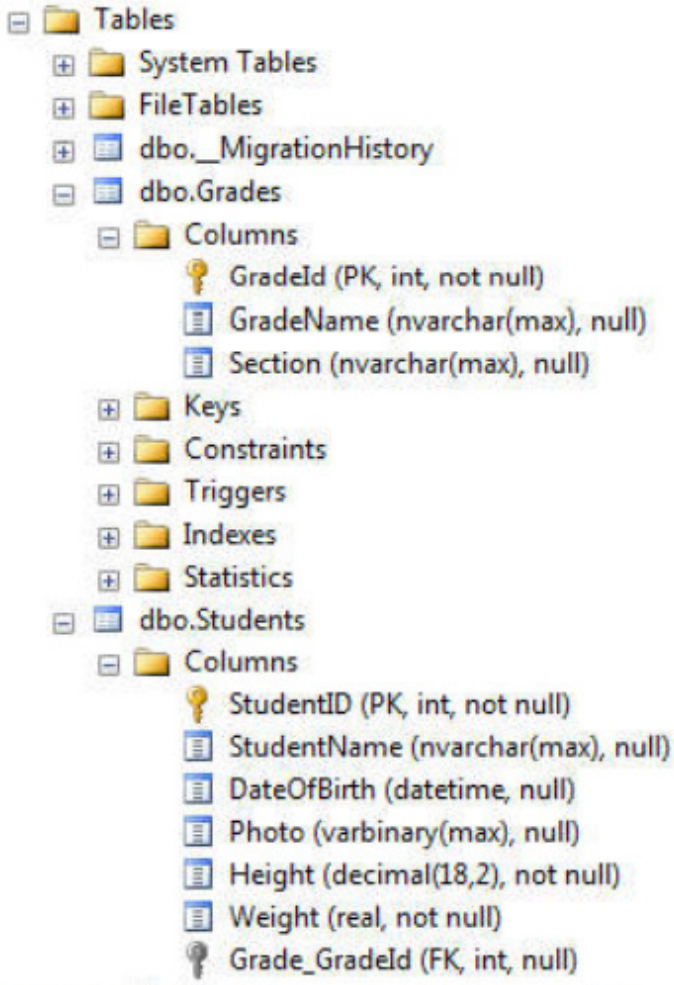
- The navigation property represents a relationship to another entity.
- There are two types of navigation properties: Reference Navigation and Collection Navigation

```
public class Student {
    //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; }

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

Reference Navigation Property

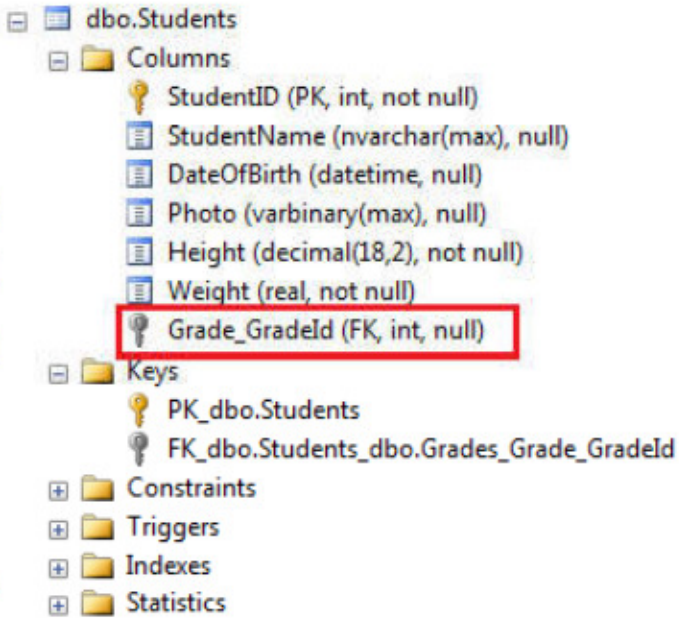
- If an entity includes a property of another entity type, it is called a Reference Navigation Property.
- It points to a single entity and represents multiplicity of one (1) in the entity relationships.
- EF API will create a ForeignKey column in the table for the navigation properties that points to a PrimaryKey of another table in the database.
- For example, Grade are reference navigation properties in the Student entity class.

```
public class Student {
…
    public Grade Grade { get; set; }
}
```



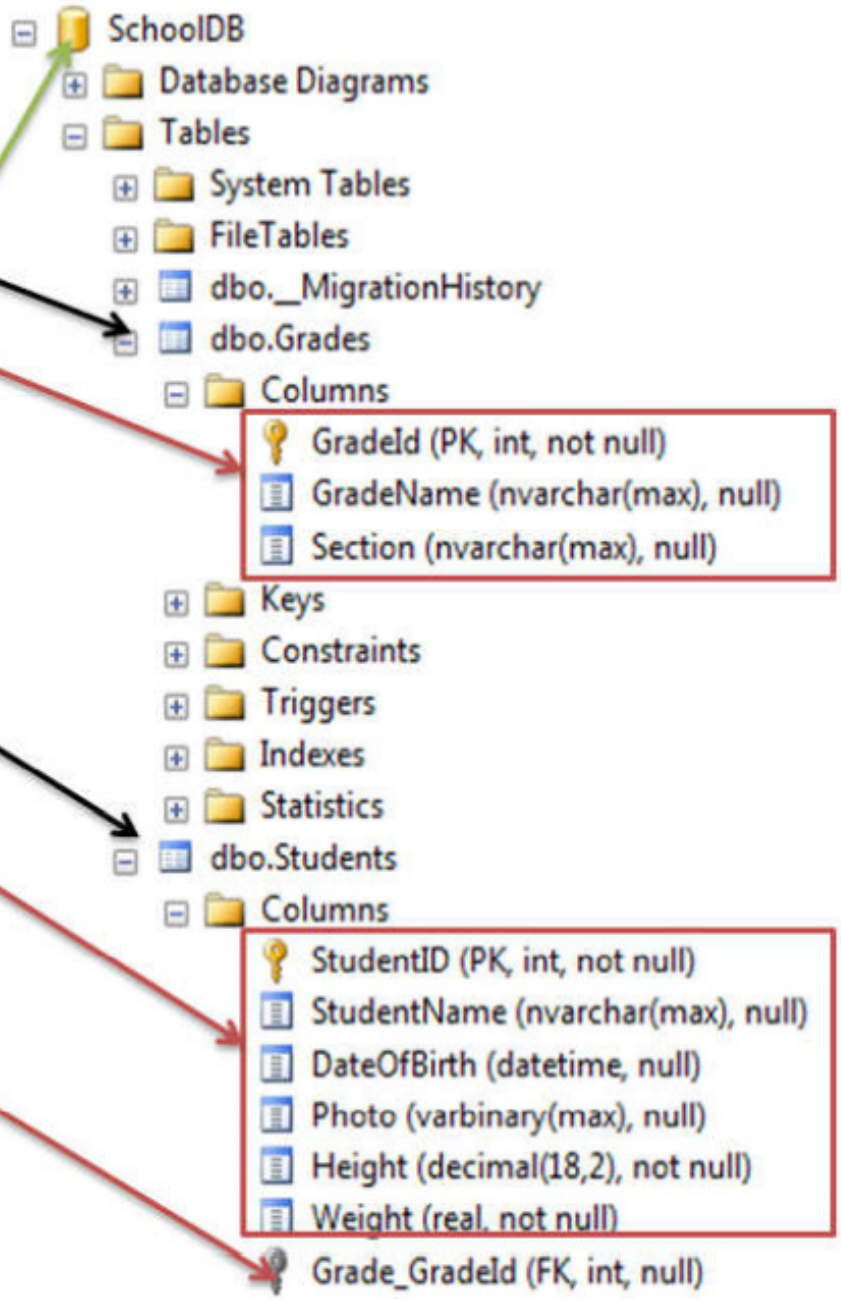
Collection Navigation Property

- If an entity includes a property of generic collection of an entity type, it is called a collection navigation property.
- It represents multiplicity of many (*).
- EF API does not create any column for the collection navigation property in the related table of an entity, but it creates a column in the table of an entity of generic collection.
- For example, the Grade entity contains a generic collection navigation property ICollection<Student>. Here, the Student entity is specified as generic type, so EF API will create a column Grade_GradeId in the Students table in the database.



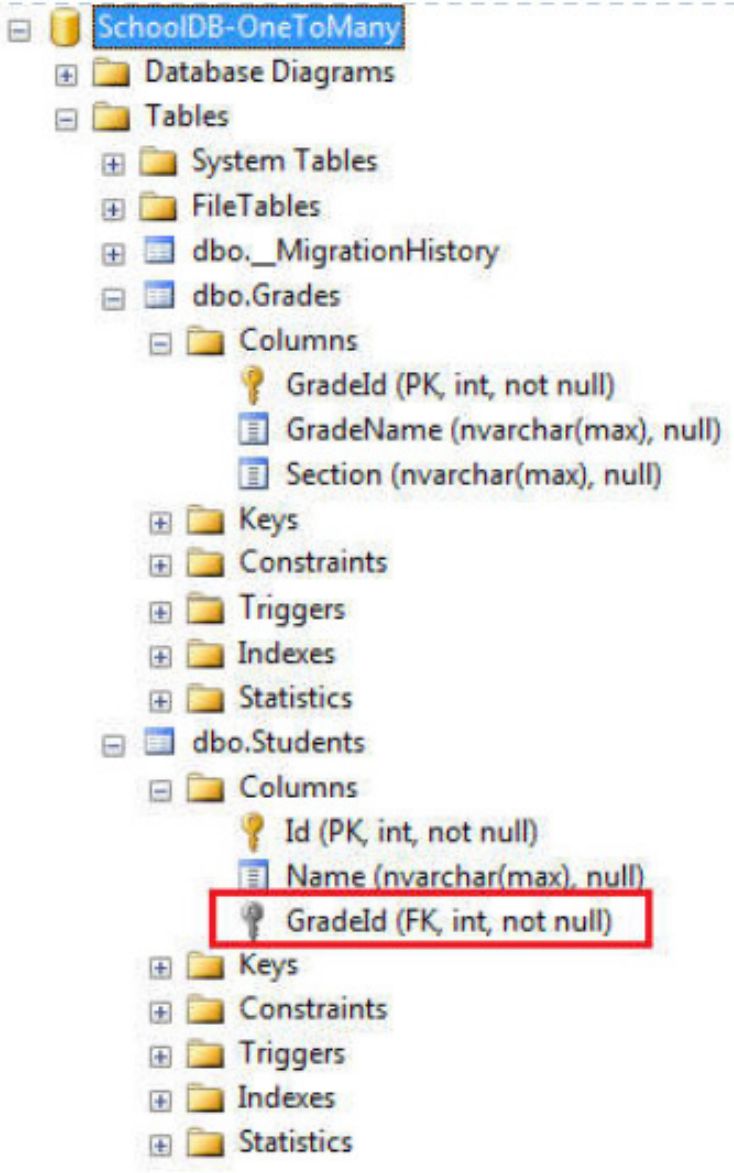

```
public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

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

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

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

    public Grade Grade { get; set; }
}
public class SchoolContext : DbContext
{
    public SchoolContext() : base("SchoolDB")
    {

    }

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



Convention 4

```
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> Student { get; set; }
}
```



public int? GradeId { get; set; }

GradeId is nullable integer, and create a null foreign key.

One-to-Zero-or-One relationships

- A student can have only one or zero addresses.

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

    public virtual StudentAddress Address { get; set; }
}

public class StudentAddress
{
    public int StudentAddressId { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public virtual Student Student { get; set; }
}
```

A one to one (or more usually a one to zero or one) relationship exists when only one row of data in the principal table is linked to zero or one row in a dependent table.

A one-to-zero-or-one relationship happens when a primary key of one table becomes PK & FK in another table in a relational database such as SQL Server.

One-to-Zero-or-One Relationship using Data Annotation Attributes

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

    public virtual StudentAddress Address { get; set; }
}

public class StudentAddress
{
    [ForeignKey("Student")]
    public int StudentAddressId { get; set; }

    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

    public virtual Student Student { get; set; }
}
```

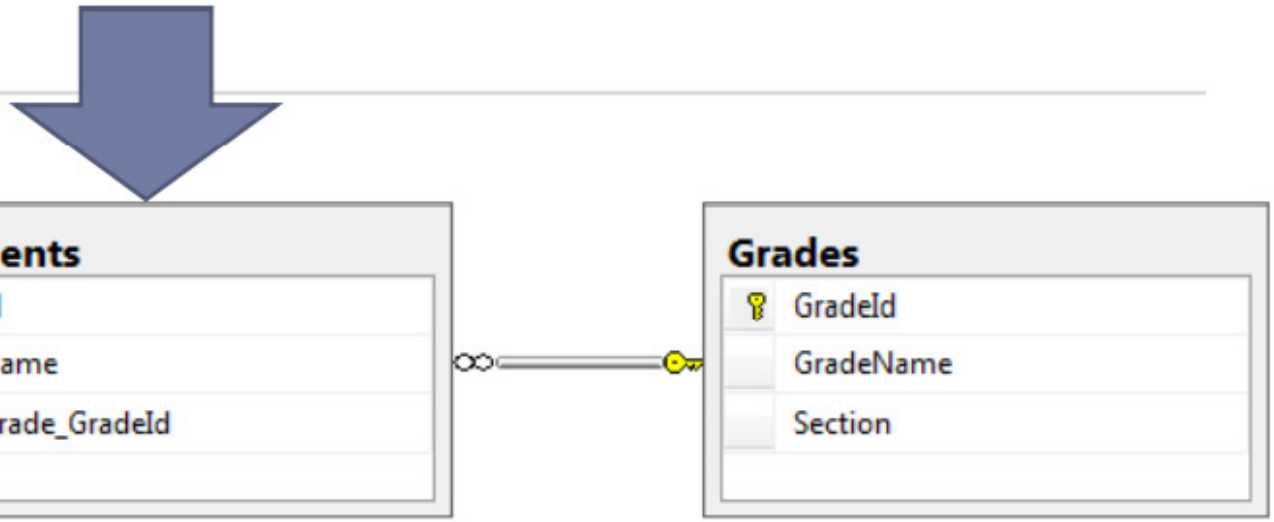
By convention StudentId and StudentAddressId are PK

But FK we define Data Annotation Attributes

Student includes the StudentAddress navigation property and StudentAddress includes the Student navigation property. With the one-to-zero-or-one relationship, a Student can be saved without StudentAddress but the StudentAddress entity cannot be saved without the Student entity. EF will throw an exception if you try to save the StudentAddress entity without the Student entity.

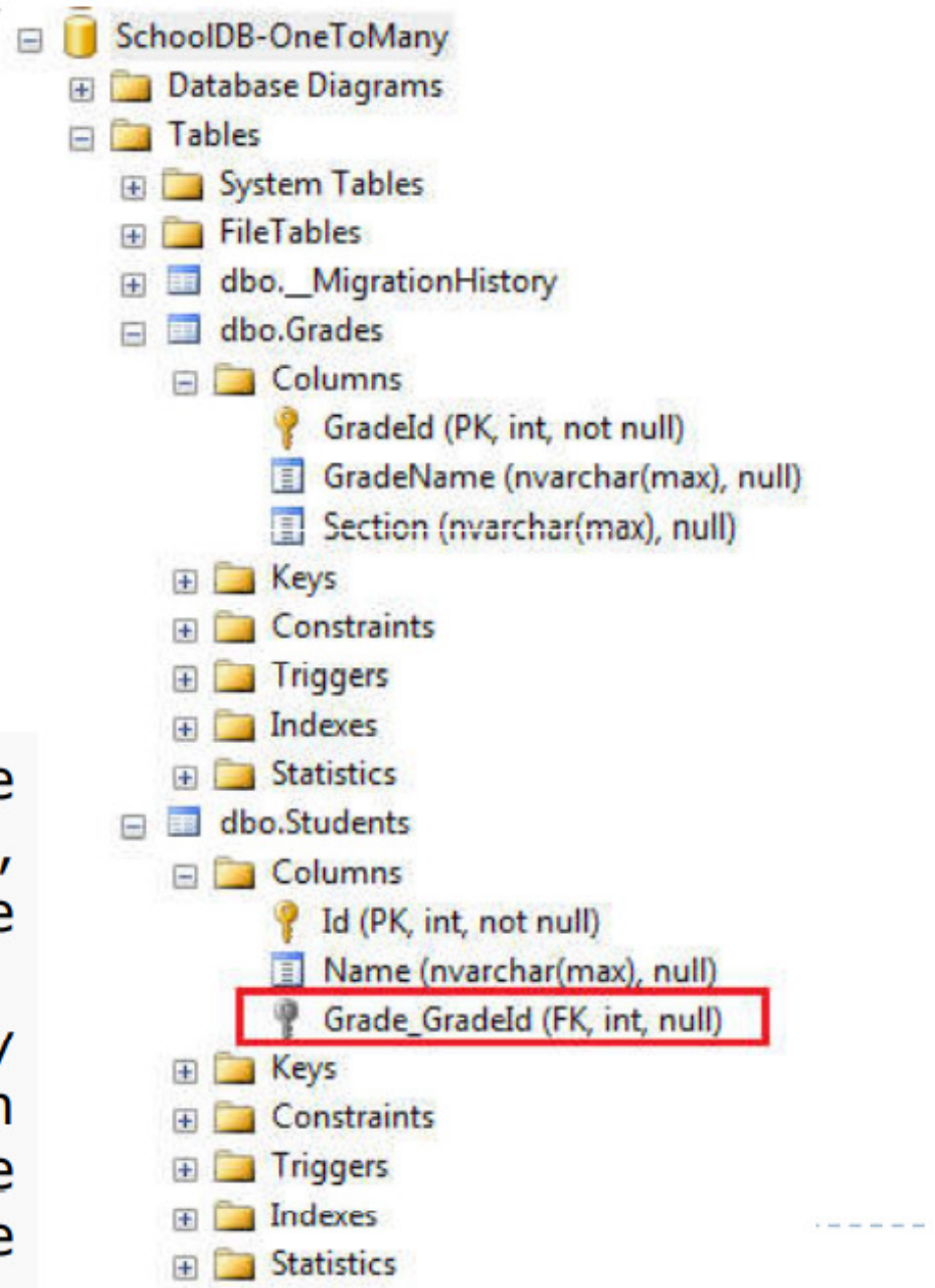
One-to-Many relationships

```
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; }
}
```



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 string Section { get; set; }
}
```



Student class includes a reference navigation property of Grade class. So, there can be many students in a single grade.

This will result in a one-to-many relationship between the Students and Grades table in the database, where the Students table includes foreign key Grade_GradeId

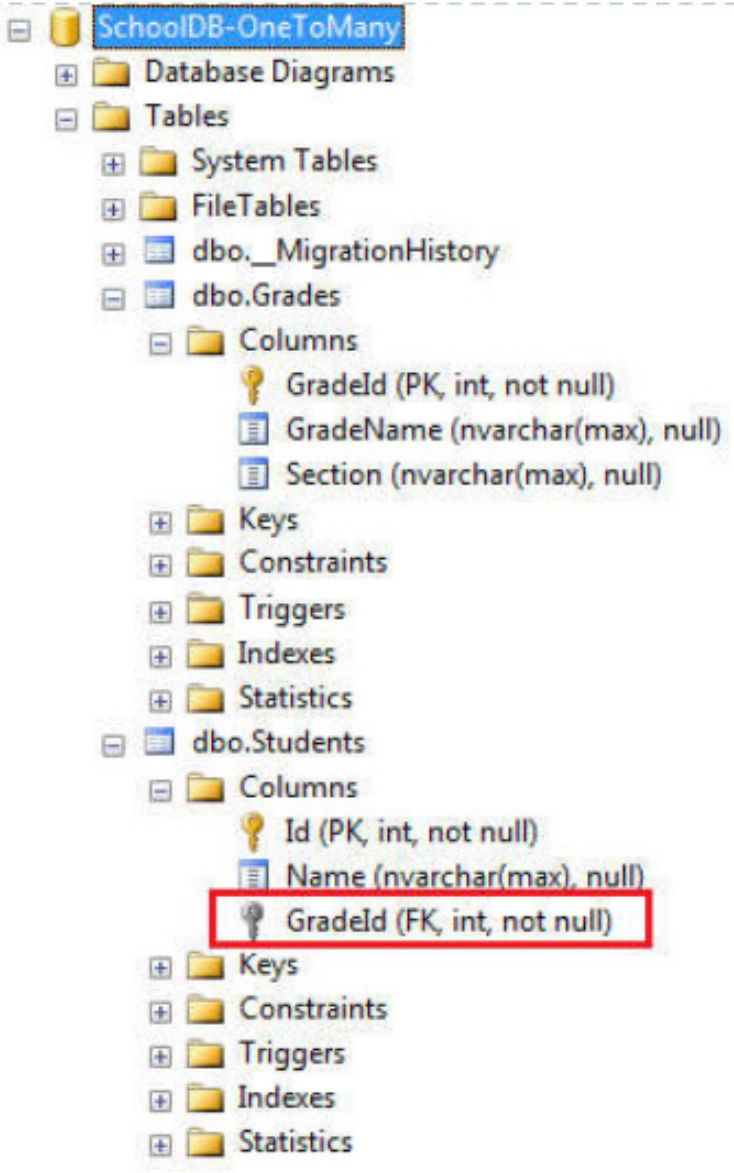
Convention 2

```
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; }
}
```

- Same result in the database as convention 1.

Convention 4

```
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> Student { get; set; }
}
```

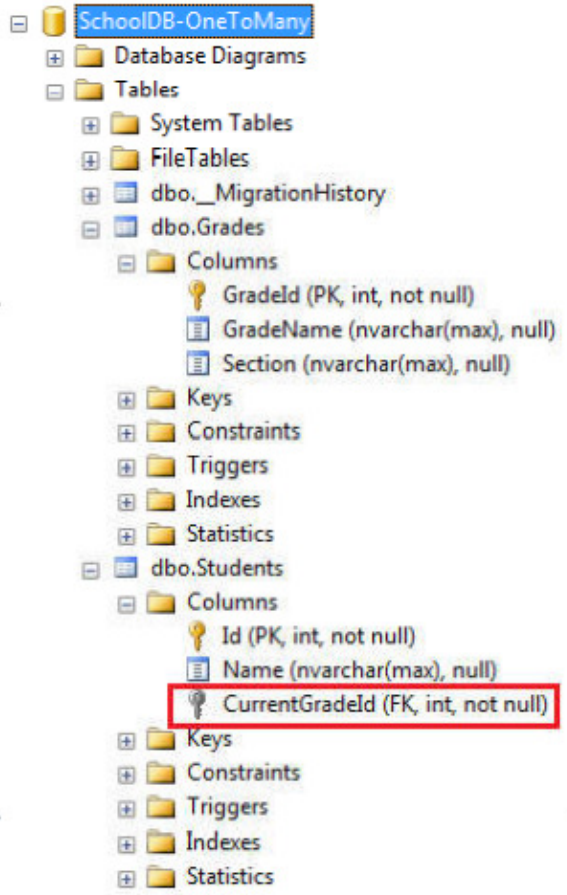


public int? GradeId { get; set; }

GradeId is nullable integer, and create a null foreign key.

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

    public int CurrentGradeId { get; set; }
    public Grade CurrentGrade { 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
{
    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // configures one-to-many relationship
        modelBuilder.Entity<Student>()
            .HasRequired<Grade>(s => s.CurrentGrade)
            .WithMany(g => g.Students)
            .HasForeignKey<int>(s => s.CurrentGradeId);          }
    }
}
```

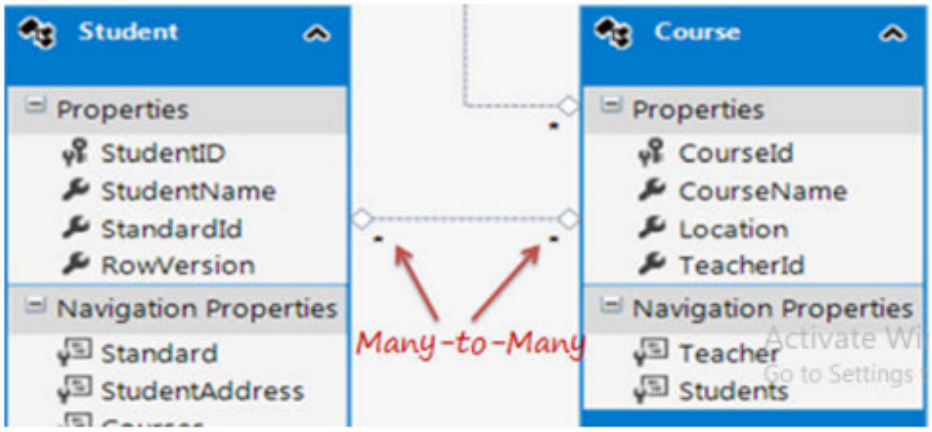



```
modelBuilder.Entity<Student>()
        .HasOne(s => s.CurrentGrade)
        .WithMany(g => g.Students)
        .HasForeignKey(s => s.CurrentGradeId);
```

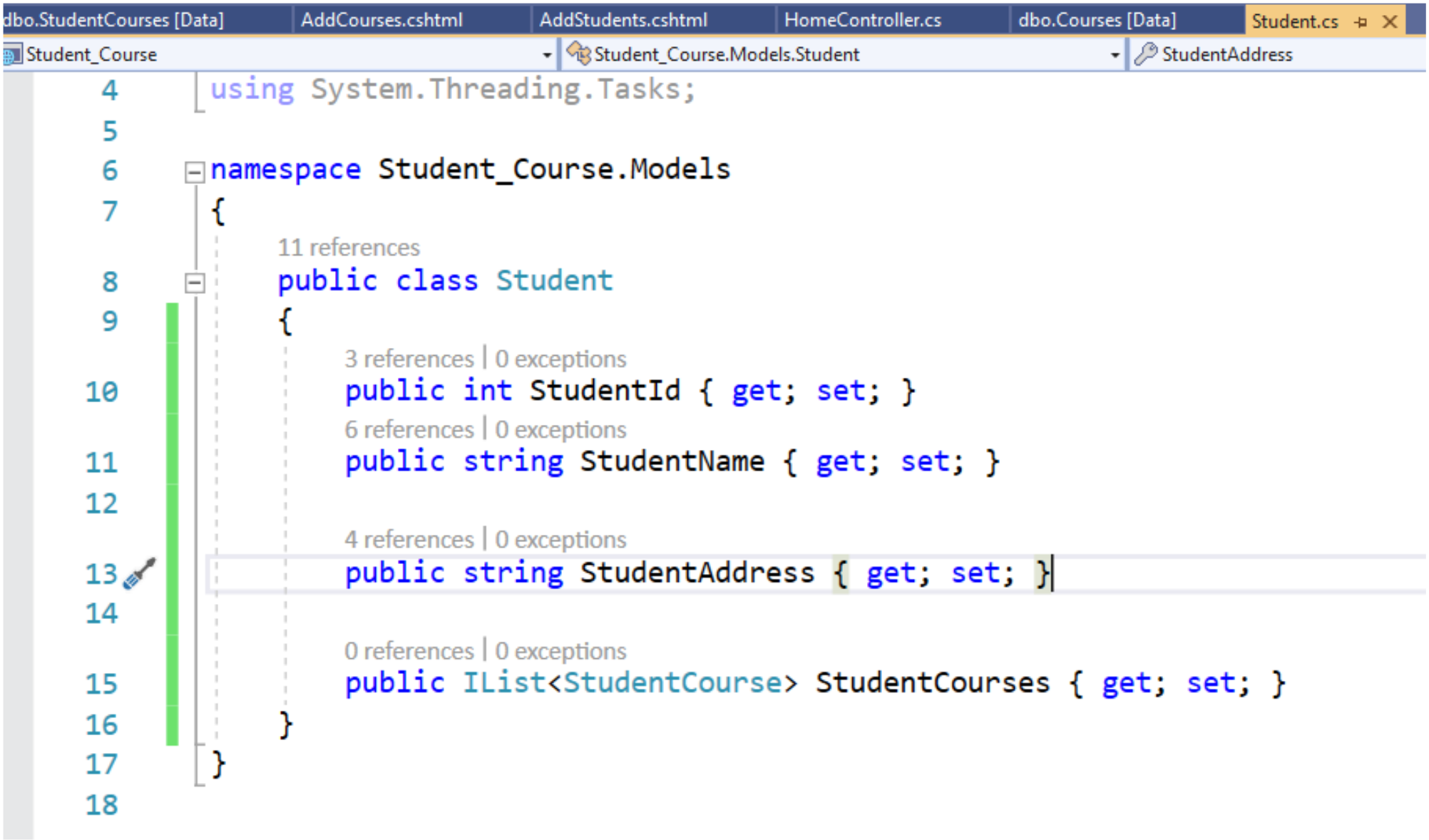
Many-to-Many relationship

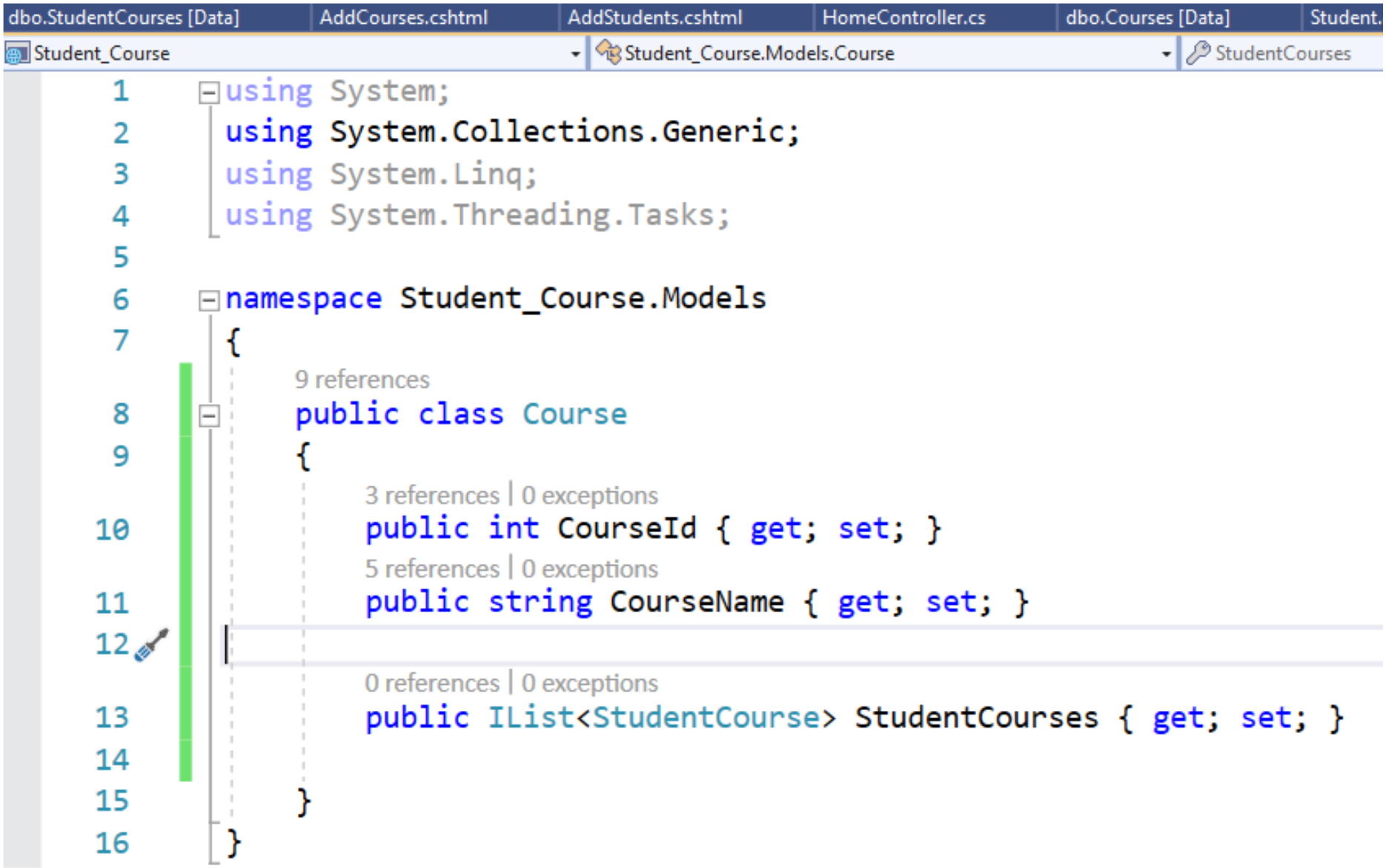
```
public class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }

    public virtual ICollection<Course> Courses { get; set; }
}

public class Course
{
    public Course()
    {
        this.Students = new HashSet<Student>();
    }

    public int CourseId { get; set; }
    public string CourseName { get; set; }

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

Student class should have a collection navigation property of Course type, and the Course class should have a collection navigation property of Student type to create a many-to-many relationship between them without any configuration.



Student can join multiple courses and multiple students can join one Course.


dbo.StudentCourses [Data]
AddCourses.cshtml
AddStudents.cshtml
HomeController.cs
dbo.Courses [Data]
Student.cs
Student_Course
Student_Course.Models.Student
StudentAddress
using System.Threading.Tasks;
namespace Student_Course.Models
{
11 references
public class Student
{
3 references | 0 exceptions
public int StudentId { get; set; }
6 references | 0 exceptions
public string StudentName { get; set; }
4 references | 0 exceptions
public string StudentAddress { get; set; }
0 references | 0 exceptions
public IList<StudentCourse> StudentCourses { get; set; }
}
}


dbo.StudentCourses [Data]
AddCourses.cshtml
AddStudents.cshtml
HomeController.cs
dbo.Courses [Data]
Student_Course
Student_Course.Models.Course
StudentCourses
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
namespace Student_Course.Models
{
9 references
public class Course
{
3 references | 0 exceptions
public int CourseId { get; set; }
5 references | 0 exceptions
public string CourseName { get; set; }
0 references | 0 exceptions
public IList<StudentCourse> StudentCourses { get; set; }
}
}

dbo.StudentCourses [Data] | AddStudents.cshtml | HomeController.cs | dbo.Courses [Data]

Student_Course | Student_Course.Models.StudentCourse

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Threading.Tasks;
5
6   namespace Student_Course.Models
7   {
        10 references
8       public class StudentCourse
9       {
            6 references | 0 exceptions
10          public int StudentId { get; set; }
            0 references | 0 exceptions
11          public Student Student { get; set; }
12
            6 references | 0 exceptions
13          public int CourseId { get; set; }
14
            0 references | 0 exceptions
15          public Course Course { get; set; }
16
17      }
18  }
19
```

dbo.StudentCourses [Data] | HomeController.cs | dbo.Courses [Data] | Student.cs | Course.cs | StudentCourse.cs | StudentContext.cs

Student_Course | Student_Course.Models.StudentContext | Courses

```
7   namespace Student_Course.Models
8   {
        7 references
9       public class StudentContext:DbContext
10      {
            0 references | 0 exceptions
11          public StudentContext(DbContextOptions<StudentContext> options) : base(options)
12          {
13
14          }
            0 references | 0 exceptions
15          protected override void OnModelCreating(ModelBuilder modelBuilder)
16          {
17              modelBuilder.Entity<StudentCourse>()
18                  .HasKey(sc => new { sc.StudentId, sc.CourseId });
19          }
20
            6 references | 0 exceptions
21          public DbSet<Student> Students { get; set; }
            6 references | 0 exceptions
22          public DbSet<Course> Courses { get; set; }
            5 references | 0 exceptions
23          public DbSet<StudentCourse> StudentCourses { get; set; }
24      }
25  }
```

Tables

dbo.StudentCourses [Data] | SC.cs | AddStudentCours

Max Rows: 1000

StudentId	StudentName	StudentAddress
1	Rahim	Uttara
2	Rahim	Uttara
3	Mounita	Gazipur
4	Moin	Mirpur
5	Moinul	Mirpur
NULL	NULL	NULL

dbo.StudentCourses [Data] | SC.cs

Max Rows: 1000

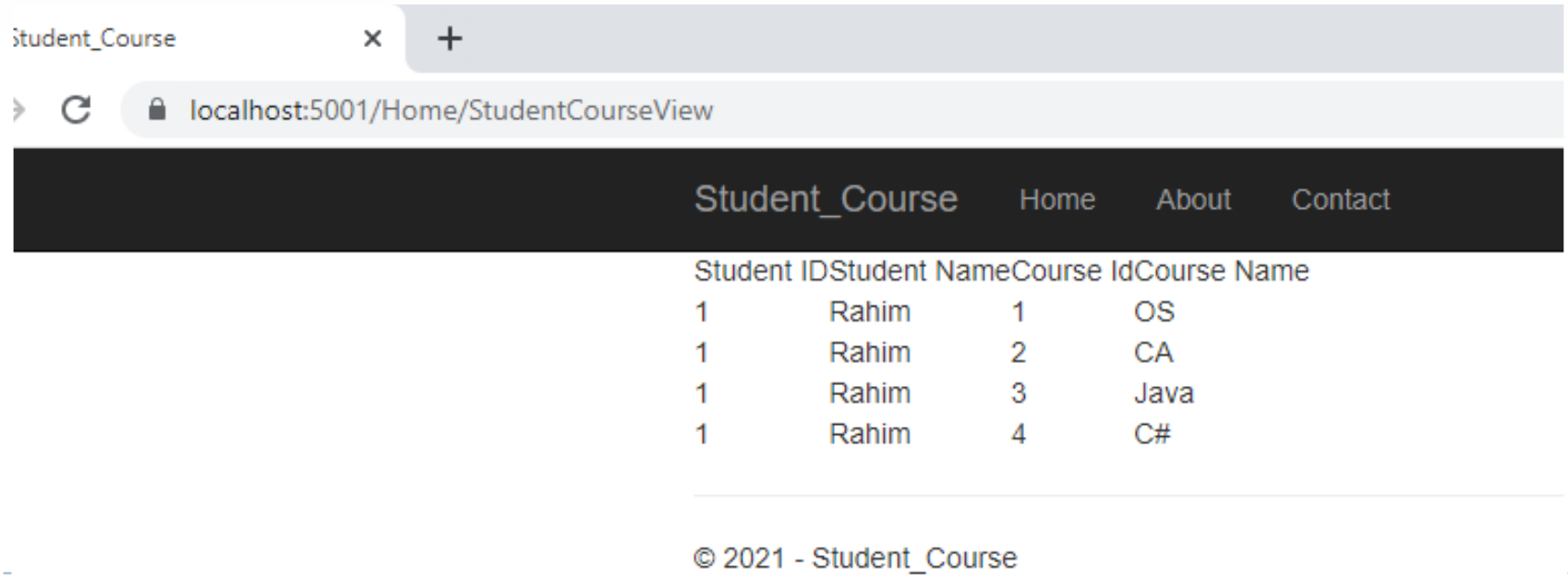
CourseId	CourseName
1	OS
2	CA
3	Java
4	C#
NULL	NULL

dbo.StudentCourses [Data] | SC.cs

Max Rows: 1000

StudentId	CourseId
1	1
2	1
1	2
2	2
1	3
1	4
NULL	NULL

```
0 references | 0 requests | 0 exceptions
public IActionResult StudentCourseView()
{
    var scv = from s in ctx.Students
              from c in ctx.Courses
              from sc in ctx.StudentCourses
              where s.StudentId == 1 && sc.StudentId==1 && sc.CourseId==c.CourseId
              select new SC {
                  StudentId = s.StudentId,
                  StudentName = s.StudentName,
                  CourseId = c.CourseId,
                  CourseName = c.CourseName
              };

    return View(scv);
}
```



AddStudent

```
public IActionResult AddStudent()
{

    var std = new Student
    {

        StudentName = "Mahin",
        DateOfBirth = new DateTime(2006, 1, 25),
        Height = 25.17m,
        Weight = 100.2F,
        GradeId=1
    };
    ctx.Students.Add(std);
    ctx.SaveChanges();
```

AddGrade

```
0 references | 0 requests | 0 exceptions
public IActionResult AddGrade() {
    var grd1= new Grade {  GradeName = "A", Section = "+" };
    ctx.Grades.Add(grd1);
    object p1 = ctx.Database.ExecuteSqlCommand(@"SET IDENTITY_INSERT [dbo].[Grades] ON");
    ctx.SaveChanges();
    ctx.Database.ExecuteSqlCommand(@"SET IDENTITY_INSERT [dbo].[Grades] OFF");
    var grd2 = new Grade {  GradeName = "A", Section = "-" };
    ctx.Grades.Add(grd2);
    object p2 = ctx.Database.ExecuteSqlCommand(@"SET IDENTITY_INSERT [dbo].[Grades] ON");
    ctx.SaveChanges();
    ctx.Database.ExecuteSqlCommand(@"SET IDENTITY_INSERT [dbo].[Grades] OFF");
    var grd3 = new Grade {  GradeName = "A", Section = " " };
    ctx.Grades.Add(grd3);
    object p3 = ctx.Database.ExecuteSqlCommand(@"SET IDENTITY_INSERT [dbo].[Grades] ON");
    ctx.SaveChanges();
    ctx.Database.ExecuteSqlCommand(@"SET IDENTITY_INSERT [dbo].[Grades] OFF");
```

Delete and Update Queries

```
0 references | 0 requests | 0 exceptions
public IActionResult DeleteStudent()
{
    var st = (from s in ctx.Students
              where (s.StudentId == 2)
              select s).FirstOrDefault();
    ctx.Students.Remove(st);
    ctx.SaveChanges();

    return View(ctx.Students);
}
```

```
public IActionResult UpdateStudent()
{
    var st = (from s in ctx.Students
              where (s.StudentId == 2)
              select s).FirstOrDefault();
    st.StudentName = "Moni";
    ctx.Students.Update(st);
    ctx.SaveChanges();

    return View(ctx.Students);
}
```

ShowGradeStudents

```
0 references | 0 requests | 0 exceptions
public IActionResult ShowGradeStudents()
{
    var sq = from g in ctx.Grades
             from s in ctx.Students
             where g.GradeName == "A" && g.GradeId==s.GradeId
             select new Student_Grade
             {
                 StudentId = s.StudentId,
                 StudentName = s.StudentName,
                 GradeName = g.GradeName,
                 Section = g.Section
             };
    //Student-Grade result= sq.ToList();
    return View(sq);
}
```

ShowStudentGrade

```
Student IdStudentNameStudent Grade
1        Rahim       A+
3        Mahin       A+
4        Rimi        A-

© 2021 - CodeFirstSimple
```

```
0 references | 0 requests | 0 exceptions
public IActionResult ShowStudentGrade()
{
    var sq = from s in ctx.Students
             from g in ctx.Grades
             where g.GradeId == s.GradeId
             select new Student_Grade{ StudentId=s.StudentId, StudentName=s.StudentName,
                 GradeName=g.GradeName, Section=g.Section };
   //Student-Grade result= sq.ToList();
    return View(sq);
}
```

```
namespace CodeFirstSimple.Models
{
    8 references
    public class Student_Grade
    {
        4 references | 0 exceptions
        public int StudentId { get; set; }
        4 references | 0 exceptions
        public string StudentName { get; set; }
        4 references | 0 exceptions
        public string GradeName { get; set; }
        4 references | 0 exceptions
        public string Section { get; set; }
    }
}
```

```
@model IEnumerable<CodeFirstSimple.Models.Student_Grade>
<html>
<body>
    <table>
        <tr>
            <td> Student Id </td>
            <td> StudentName</td>
            <td> Student Grade</td>
        </tr>
        @foreach (var s in Model)
        {
            <tr>
                <td> @s.StudentId </td>
                <td> @s.StudentName </td>
                <td>  @s.GradeName@s.Section</td>
</tr>
        }

    </table>

</body>

</html>
```