EF

khanhdsp@gmail.com

Configure Domain Classes

- Code-First builds the conceptual model from your domain classes using default conventions
- However, you can override these conventions by configuring your domain classes to provide EF with the information it needs. There are two ways to configure your domain classes:
  - Data Annotation Attributes
  - Fluent API

Data Annotations Attributes

- Data Annotations is a simple attribute based configuration

```
[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;}
 }
```

Data Annotations Attributes

- Data annotation attributes do not support all the configuration options for Entity Framework. So, you can use Fluent API, which provides all the configuration options for EF.

Fluent API

- Fluent API configuration can be applied when EF builds a model from your domain classes
- You can inject the Fluent API configurations by **overriding** the **OnModelCreating** method of **DbContext** in Entity Framework 6.x, as shown below:

```
public class SchoolDBContext: DbContext
{
    public SchoolDBContext(): base("SchoolDBConnectionString")
    {
    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Standard> Standards { get; set; }
    public DbSet<StudentAddress> StudentAddress { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure domain classes using modelBuilder here..

    }
}
```

Data Annotations

- Data Annotations attributes are .NET attributes which can be applied on an entity class or properties to override default conventions in EF
- Data annotation attributes are included in the **System.ComponentModel.DataAnnotations** and **System.ComponentModel.DataAnnotations.Schema** namespaces in EF
- **Note:** Data annotations only give you a subset of configuration options. Fluent API provides a full set of configuration options available in Code-First.

DataAnnotations

Attribute	Description
Key	Can be applied to a property to specify a key property in an entity and make the corresponding column a PrimaryKey column in the database.
Timestamp	Can be applied to a property to specify the data type of a corresponding column in the database as `rowversion`.
ConcurrencyCheck	Can be applied to a property to specify that the corresponding column should be included in the optimistic concurrency check.
Required	Can be applied to a property to specify that the corresponding column is a NotNull column in the database.
MinLength	Can be applied to a property to specify the minimum string length allowed in the corresponding column in the database.
MaxLength	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database.
StringLength	Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database.

DataAnnotations.Schema

Attribute	Description
Table	Can be applied to an entity class to configure the corresponding table name and schema in the database.
Column	Can be applied to a property to configure the corresponding column name, order and data type in the database.
Index	Can be applied to a property to configure that the corresponding column should have an Index in the database. (EF 6.1 onwards only)
ForeignKey	Can be applied to a property to mark it as a foreign key property.
NotMapped	Can be applied to a property or entity class which should be excluded from the model and should not generate a corresponding column or table in the database.
DatabaseGenerated	Can be applied to a property to configure how the underlying database should generate the value for the corresponding column e.g. identity, computed or none.
InverseProperty	Can be applied to a property to specify the inverse of a navigation property that represents the other end of the same relationship.
ComplexType	Marks the class as complex type in EF 6. EF Core 2.0 does not support this attribute.

Table Attribute

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

[Table("StudentMaster", Schema="Admin")]
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
```

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

[Table("StudentMaster")]
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
}
```

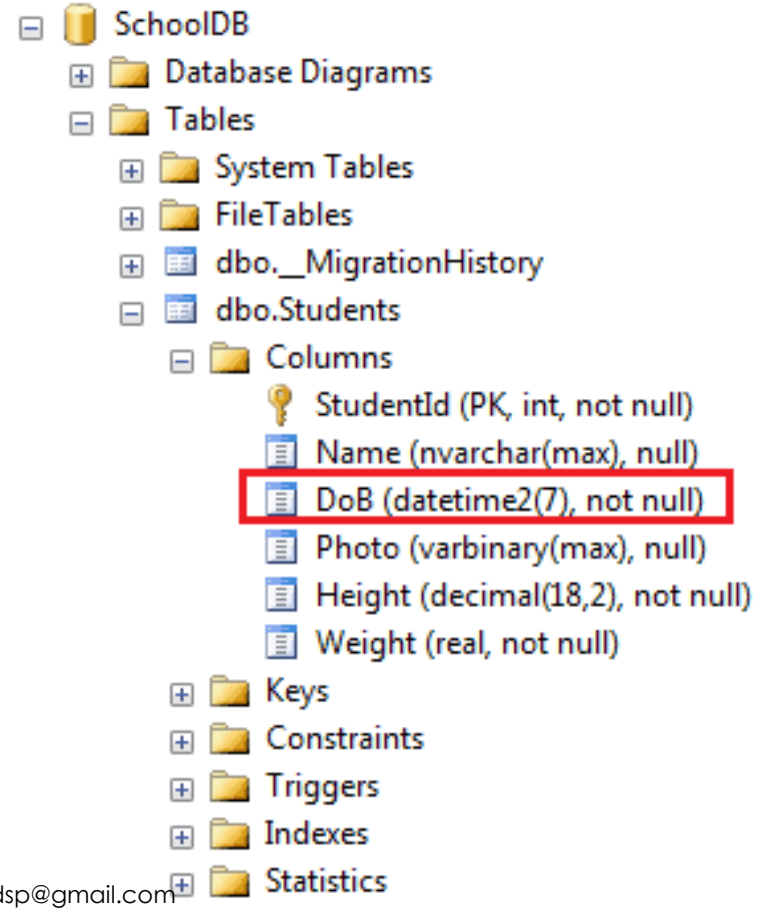
- It overrides the default convention in EF 6 and EF Core. As per the default conventions, EF 6 creates a table name matching with **<DbSet<TEntity>** property name> + 's' (or 'es') in a context class
- Table Attribute: **[Table(string name, Properties:[Schema = string])**
- **name**: Name of the Db table.
- **Schema**: Name of the Db Schema in which a specified table should be created. (Optional)

Column Attribute

```
public class Student
{
    public int StudentID { get; set; }

    [Column("Name")]
    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; }
}
```

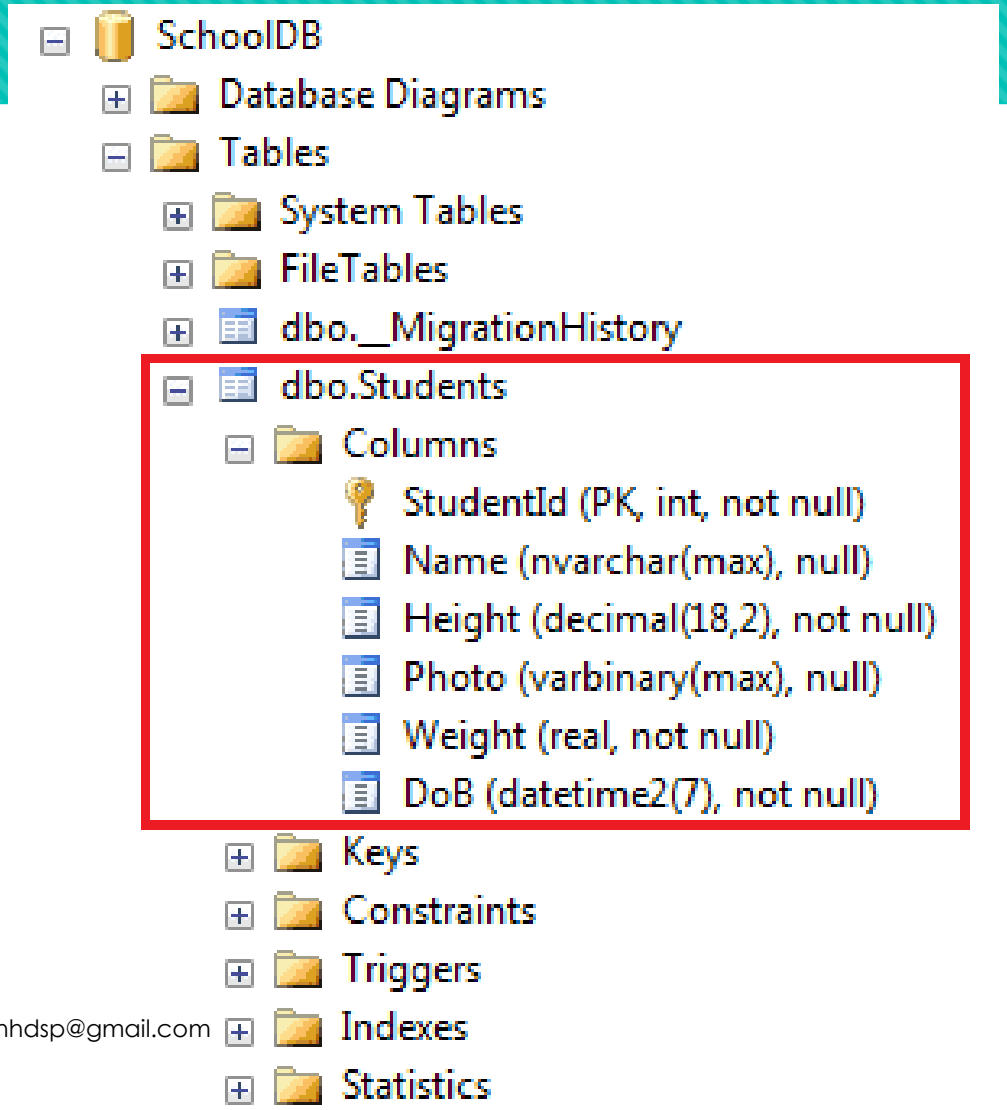
- The Column attribute overrides the default convention.
- it creates a column in a db table with the same name and order as the property names.
- **[Column (string name, Properties:[Order = int],[TypeName = string])**
- **name**: Name of a column in a db table.
- **Order**: Order of a column, starting with zero index. (Optional)
- **TypeName**: Data type of a column. (Optional)

Column Data Type




```
public class Student
{
    public int StudentID { get; set; }

    [Column("Name")]
    public string StudentName { get; set; }
    [Column("DoB", TypeName="DateTime2")]
    public DateTime DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }
}
```

Column Order

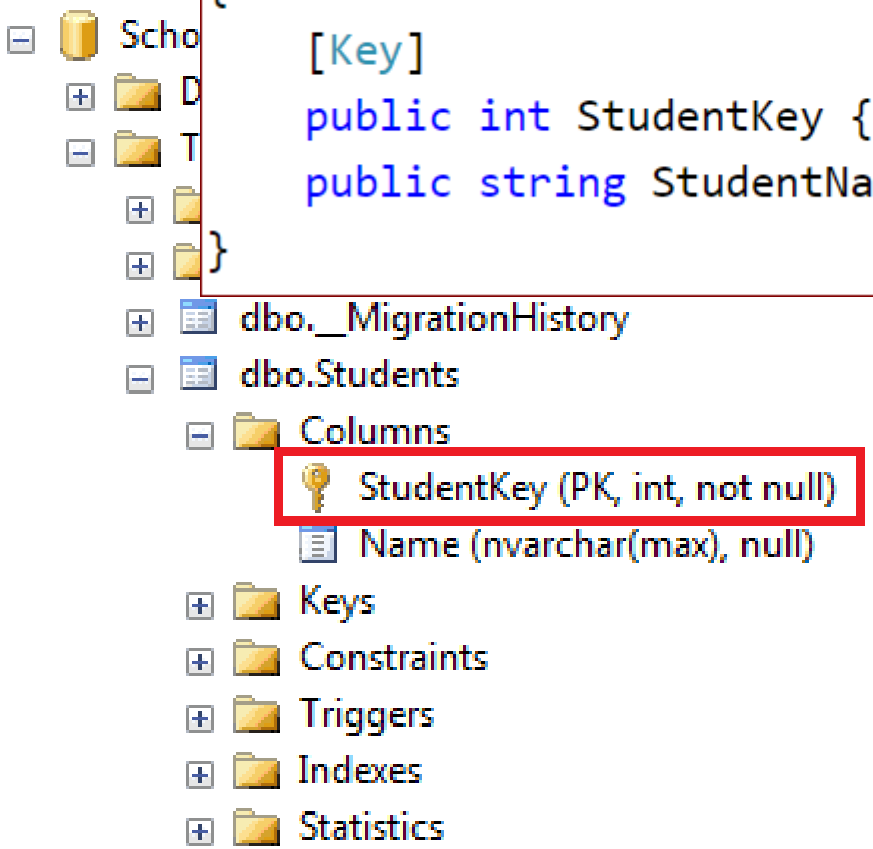



```
public class Student
{
    [Column(Order = 0)]
    public int StudentID { get; set; }

    [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; }
}
```

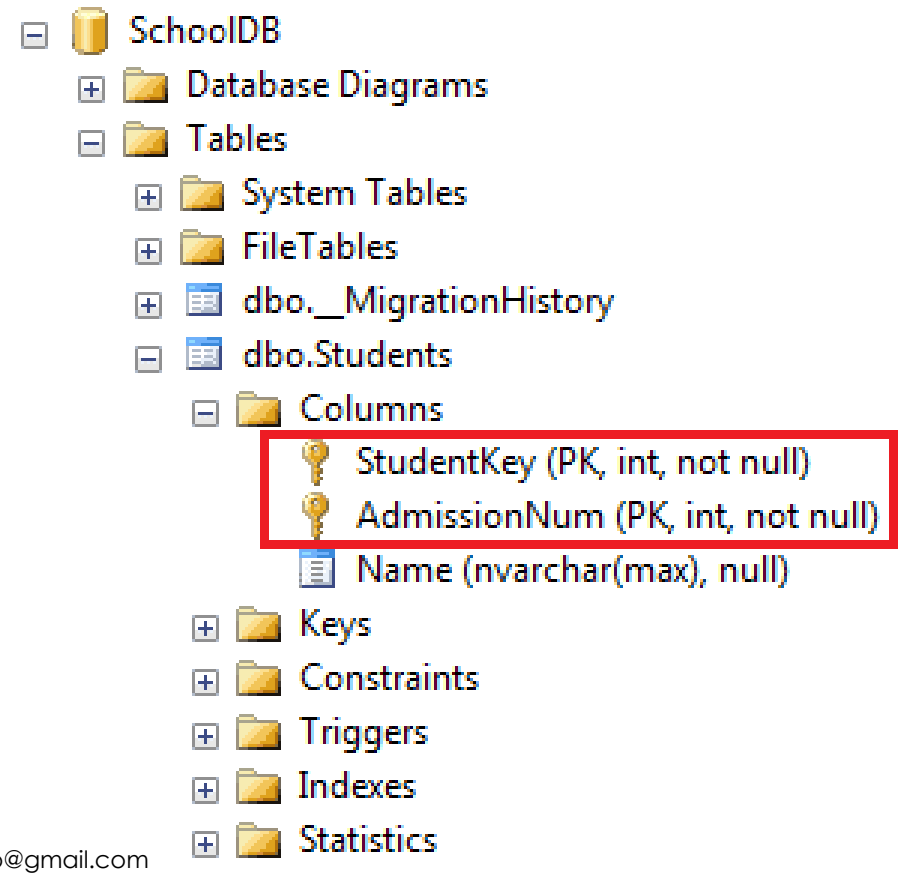
Key Attribute

- The default convention creates a primary key column for a property whose name is **Id** or **<Entity Class Name>Id**.
- The Key attribute overrides this default convention.

```
public class Student
{
    [Key]
    public int StudentKey { get; set; }
    public string StudentName { get; set; }
}
```



Composite key using the Key attribute




```
public class Student
{
    [Key]
    [Column(Order=1)]
    public int StudentKey { get; set; }

    [Key]
    [Column(Order=2)]
    public int AdmissionNum { get; set; }

    public string StudentName { get; set; }
}
```

NotMapped Attribute

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

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

- The [NotMapped] attribute overrides this default convention
- **Note:** EF also does not create a column for a property which does not have either getters or setters

```
public class Student
{
    private int _age = 0;

    public int StudentId { get; set; }
    public string StudentName { get; set; }
    public string City { get{ return StudentName;}  }
    public int Age { set{ _age = value;} }
}
```

ForeignKey Attribute

- It overrides the default conventions
- **Syntax: [ForeignKey(name string)]**
- ***name***: Name of the associated navigation property or the name of the associated foreign key(s).

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

    //Foreign key for Standard
    public int StandardId { get; set; }
    public Standard Standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }

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

ForeignKey

- The [ForeignKey] attribute overrides the default convention for a foreign key
- **[ForeignKey(NavigationPropertyName)]** on the foreign key scalar property in the dependent entity
- **[ForeignKey(ForeignKeyPropertyName)]** on the related reference navigation property in the dependent entity
- **[ForeignKey(ForeignKeyPropertyName)]** on the navigation property in the principal entity

```
public class Student
{
    public int StudentID {
    public string StudentN

    [ForeignKey("Standard"
    public int StandardRef
    public Standard Standa
}

public class Standard
{
    public int StandardId
    public string Standard

    public ICollection<Stu
}
```

```
public class St
{
    public int
    public stri

    public int

    [ForeignKey
    public Star
}

public class St
{
    public int
    public stri

    public ICol
}
```

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

    public int StandardRefId { get; set; }
    public Standard Standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }

    [ForeignKey("StandardRefId")]
    public ICollection<Student> Students { get; set; }
}
```

Index Attribute

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

    [Index]
    public int RegistrationNumber { get; set; }
}
```

```
[Index( "INDEX_REGNUM", IsClustered=true, IsUnique=true )]
public int RegistrationNumber { get; set; }
```

InverseProperty

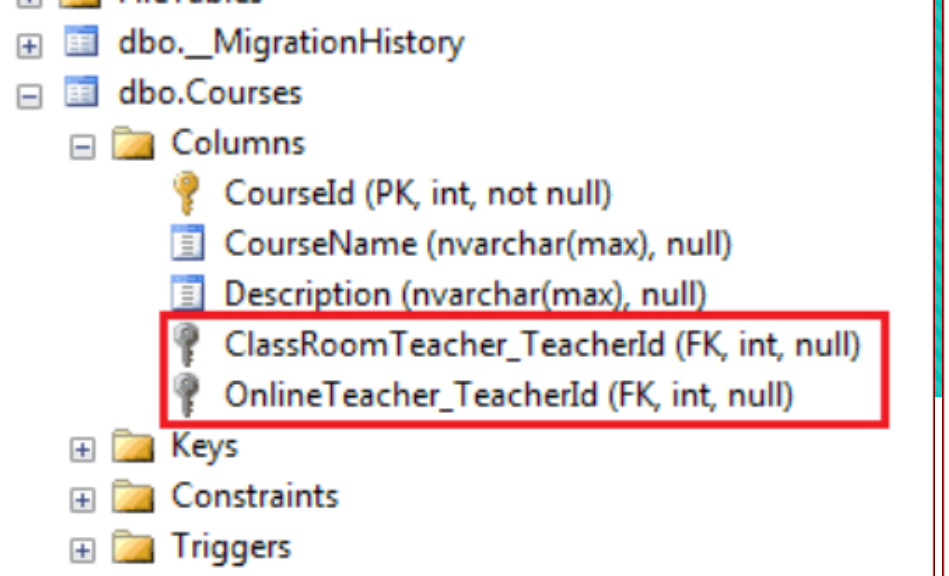
- used when two entities have more than one relationship
- the Course and Teacher entities have two one-to-many relationships.
- EF API cannot determine the other end of the relationship. It will throw the following exception for the above example during migration.

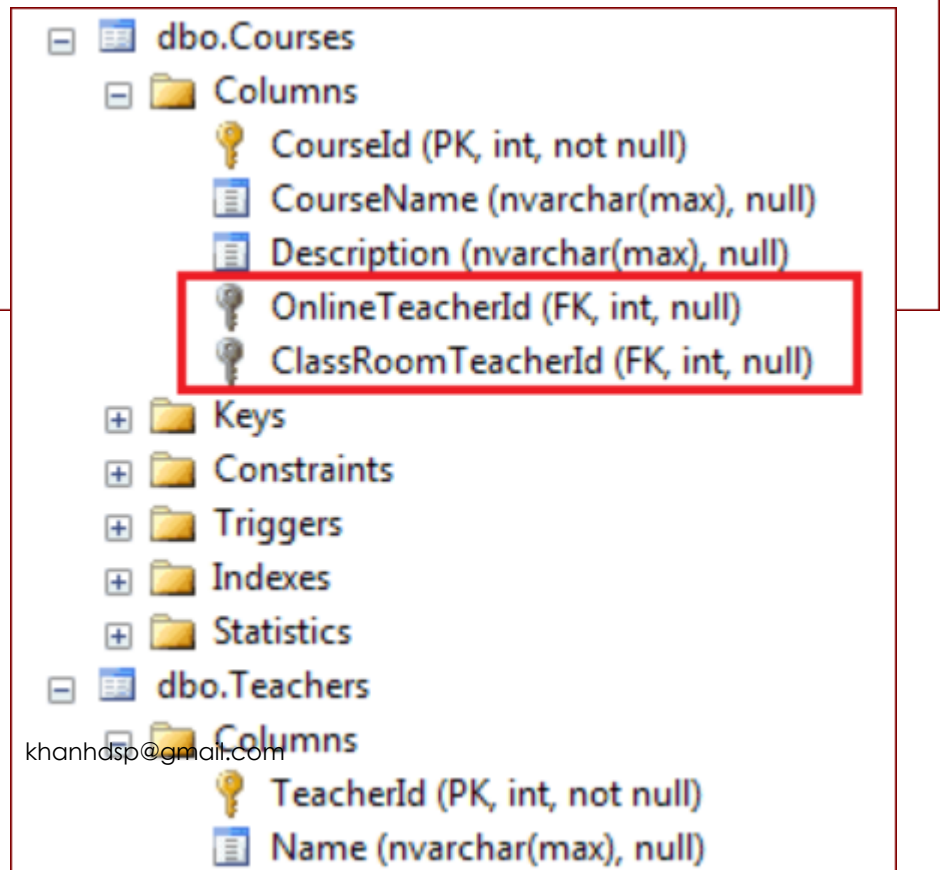
```
public class Course
{
    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public string Description { get; set; }

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

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

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

InverseProperty

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



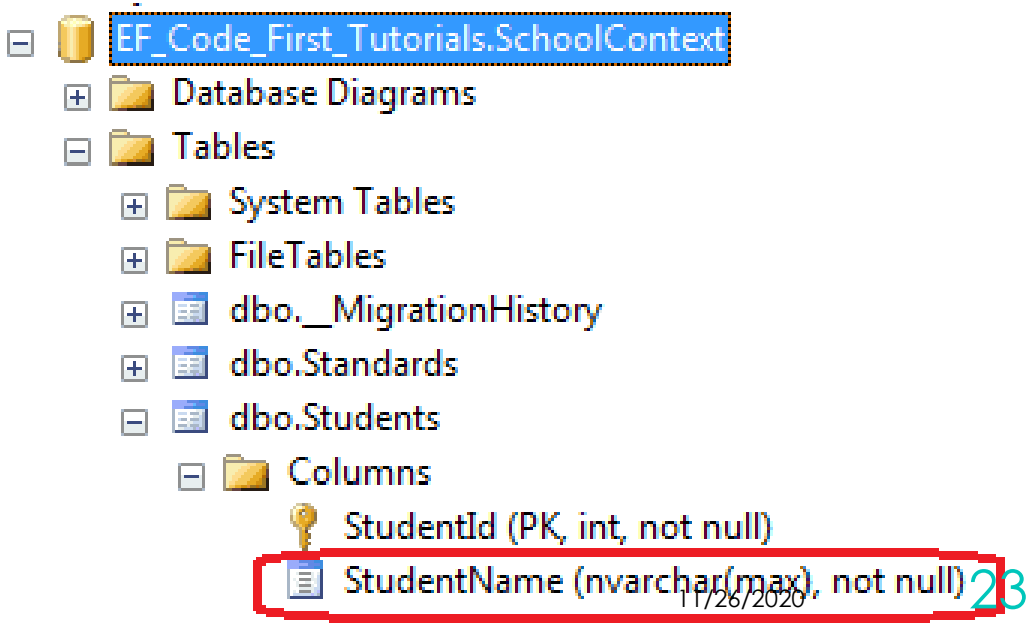



```
    public string CourseName { get; set; }
    public string Description { get; set; }

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

    [ForeignKey("ClassRoomTeacher")]
    public int? ClassRoomTeacherId { 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; }
}
```

Required Attribute

```
public class Student
{
    public int StudentID { get; set; }
    [Required]
    public string StudentName { get; set; }
}
```

- EF will create a NOT NULL column in a database table for a property on which the Required attribute is applied



MaxLength, [StringLength(50)]

- The MaxLength attribute specifies the maximum length of data value allowed for a property
- Sets the size of a corresponding column in the database
- It can be applied to the **string** or **byte**[] properties of an entity.

```
public class Student
{
    public int StudentID { get; set; }
    [StringLength(50)]
    public string StudentName { get; set; }
}
```

```
public class Student
{
    public int StudentID { get; set; }
    [MaxLength(50)]
    public string StudentName { get; set; }

}
```

DatabaseGenerated

- EF creates an IDENTITY column in the database for all the id (key) properties of the entity, by default
- DatabaseGenerated data annotation attribute to configure how the value of a property will be generated

1. DatabaseGeneratedOption.None
2. DatabaseGeneratedOption.Identity
3. DatabaseGeneratedOption.Computed

DatabaseGeneratedOption.Compute

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

    modelBuilder.Entity<Student>()
            .Property(s => s.CreatedDate)
            .HasDefaultValueSql("GETDATE()");
}
```

```
                                                    set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public DateTime CreatedDate { get; set; }
}
```

Fluent API Configurations

- Entity Framework Fluent API is used to configure domain classes to override conventions
- It provides more options of configurations than Data Annotation attributes.
- To write Fluent API configurations, override the OnModelCreating() method of DbContext in a context class, as shown below

```
public class SchoolContext: DbContext
{

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

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

    }
}
```

Fluent API Configurations (2)

- Fluent API configures the following aspect of a model in Entity Framework 6:
  - Model-wide Configuration: Configures the default Schema, entities to be excluded in mapping, etc.
  - Entity Configuration: Configures entity to table and relationship mappings e.g. PrimaryKey, Index, table name, one-to-one, one-to-many, many-to-many etc.
  - Property Configuration: Configures property to column mappings e.g. column name, nullability, Foreignkey, data type, concurrency column, etc.

Entity Mappings using Fluent API

- Configure Default Schema

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure default schema
        modelBuilder.HasDefaultSchema("Admin");
    }
}
```

Entity Mappings using Fluent API

- Map Entity to Table

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure default schema
        modelBuilder.HasDefaultSchema("Admin");

        //Map entity to table
        modelBuilder.Entity<Student>().ToTable("StudentInfo");
        modelBuilder.Entity<Standard>().ToTable("StandardInfo","dbo");
    }
}
```

Entity Mappings using Fluent API

- Map Entity to Multiple Tables

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>().Map(m =>
        {
            m.Properties(p => new { p.StudentId, p.StudentName});
            m.ToTable("StudentInfo");
        }).Map(m => {
            m.Properties(p => new { p.StudentId, p.Height, p.Weight, p.Phot
            m.ToTable("StudentInfoDetail");
        });

        modelBuilder.Entity<Standard>().ToTable("StandardInfo");
    }
}
```

Property Mappings using Fluent API

- Using Fluent API, you can change the corresponding column name, type, size, Null or NotNull, PrimaryKey, ForeignKey, concurrency column, etc.

```
public class Student
{
    public int StudentKey { 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 Standard Standard { get; set; }
}

public class Standard
{
    public int StandardKey { get; set; }
    public string StandardName { get; set; }

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

Configure Primary Key and Composite Primary Key

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure primary key
        modelBuilder.Entity<Student>().HasKey<int>(s => s.StudentKey);
        modelBuilder.Entity<Standard>().HasKey<int>(s => s.StandardKey);

        //Configure composite primary key
        modelBuilder.Entity<Student>().HasKey<int>(s => new { s.StudentKey, s.StudentName });
    }
}
```

Configure Column Name, Type and Order

- **Property()** method is used to configure a property of an entity.
- **HasColumnName()** method is used to change the column name of the DateOfBirth property.
- **HasColumnOrder()** methods change the order
- **HasColumnType()** methods change datatype of the corresponding column.

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        //Configure Column
        modelBuilder.Entity<Student>()
                    .Property(p => p.DateOfBirth)
                    .HasColumnName("DoB")
                    .HasColumnOrder(3)
                    .HasColumnType("datetime2");
    }
}
```

Configure Null or NotNull Column

- Use **IsOptional()** method to create a **nullable column** for a property.
- Use **IsRequired()** method to create a **NotNull column**.

```
public class SchoolContext: DbContext
{
    public SchoolDBContext(): base()
    {
    }

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

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
            //Configure Null Column
        modelBuilder.Entity<Student>()
                .Property(p => p.Heigth)
                .IsOptional();

            //Configure NotNull Column
            modelBuilder.Entity<Student>()
                .Property(p => p.Weight)
                .IsRequired();
    }
}
```

Configure Column Size

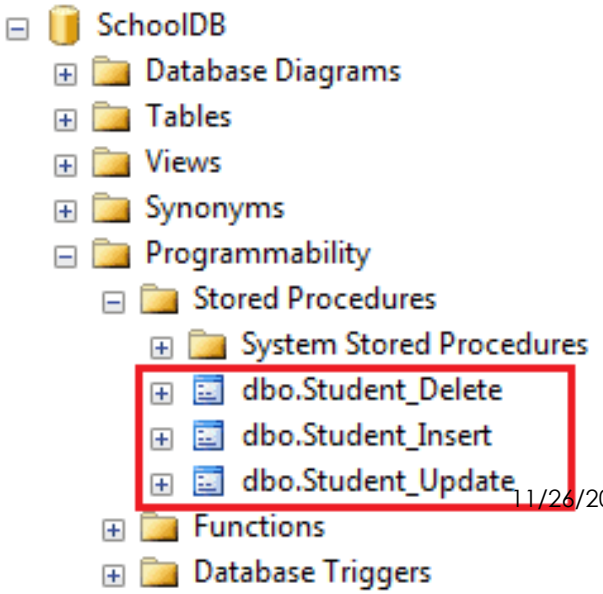
- **HasMaxLength()** method to set the size of a column.
- **IsFixedLength()** method converts nvarchar to nchar type.
- **HasPrecision()** method changed the precision of the decimal column.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    //Set StudentName column size to 50
    modelBuilder.Entity<Student>()
            .Property(p => p.StudentName)
            .HasMaxLength(50);

    //Set StudentName column size to 50 and change datatype to nchar
    //IsFixedLength() change datatype from nvarchar to nchar
    modelBuilder.Entity<Student>()
            .Property(p => p.StudentName)
            .HasMaxLength(50).IsFixedLength();

    //Set size decimal(2,2)
        modelBuilder.Entity<Student>()
            .Property(p => p.Height)
            .HasPrecision(2, 2);
}
```

Stored Procedures in Entity Framework

```
public class SchoolContext: DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Student>()
                    .MapToStoredProcedures();
    }

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

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



```sql
CREATE PROCEDURE [dbo].[Student_Insert]
    @StudentName [nvarchar](max),
    @DoB [datetime]
AS
BEGIN
    INSERT [dbo].[Students]([StudentName], [DoB])
    VALUES (@StudentName, @DoB)

    DECLARE @StudentId int
    SELECT @StudentId = [StudentId]
    FROM [dbo].[Students]
    WHERE @@ROWCOUNT > 0 AND [StudentId] = scope_identity()

    SELECT t0.[StudentId]
    FROM [dbo].[Students] AS t0
    WHERE @@ROWCOUNT > 0 AND t0.[StudentId] = @StudentId
END
```

```sql
CREATE PROCEDURE [dbo].[Student_Update]
    @StudentId [int],
    @StudentName [nvarchar](max),
    @DoB [datetime]
AS
BEGIN
    UPDATE [dbo].[Students]
    SET [StudentName] = @StudentName, [DoB] = @DoB
    WHERE ([StudentId] = @StudentId)
END
```

```sql
CREATE PROCEDURE [dbo].[Student_Delete]
    @StudentId [int]
AS
BEGIN
    DELETE [dbo].[Students]
    WHERE ([StudentId] = @StudentId)
END
```

Map Custom Stored Procedures to an Entity

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>()
            .MapToStoredProcedures(p => p.Insert(sp => sp.HasName("sp_InsertStudent").Parameter(pm => pm.
                    .Update(sp => sp.HasName("sp_UpdateStudent").Parameter(pm => pm.StudentName, "name"))
                    .Delete(sp => sp.HasName("sp_DeleteStudent").Parameter(pm => pm.StudentId, "Id"))
            );
}
```

```
protected override void OnModelCreating(DbModelBuilder modelBuilder) {
            modelBuilder.Entity<Student>()
            .MapToStoredProcedures(p => p.Insert(sp => sp.HasName("sp_InsertStudent").Parameter(pm => pm.StudentName, "name").Result(rs => rs.StudentId,
            .Update(sp => sp.HasName("sp_UpdateStudent").Parameter(pm => pm.StudentName, "name"))
            .Delete(sp => sp.HasName("sp_DeleteStudent").Parameter(pm => pm.StudentId, "Id")) );
}
```