

Mariusz Wieczorek

mariusz.wieczorek@kabat.pl

Streszczenie

Konwencje w Entity Framework Core  
Modyfikacja domyślnych zachowań framework’a za pomocą  
Data Annotations i FluentAPI

Entity Framework Core

Conventions in Entity Framework Core  
Relationships One to Many, Many to Many,   
One to One

# **Conventions in Entity Framework Core**

Conventions are default rules using which Entity Framework builds a model based on your domain (entity) classes. In the [First EF Core Application](https://www.entityframeworktutorial.net/efcore/entity-framework-core-console-application.aspx) chapter, EF Core API creates a database schema based on **domain** and **context** classes, without any additional configurations because domain classes were following the conventions.

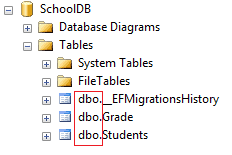
Consider the following sample entities and context class to understand the default conventions.

|  |
| --- |
| 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; }  }  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 SchoolContext : DbContext  {  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  optionsBuilder.UseSqlServer(@"Server=.\SQLEXPRESS;Database=SchoolDB;Trusted\_Connection=True;");  }  public DbSet<Student> Students { get; set; }  } |

Let's understand the EF Core conventions and how EF Core API will create a database for the above entities.

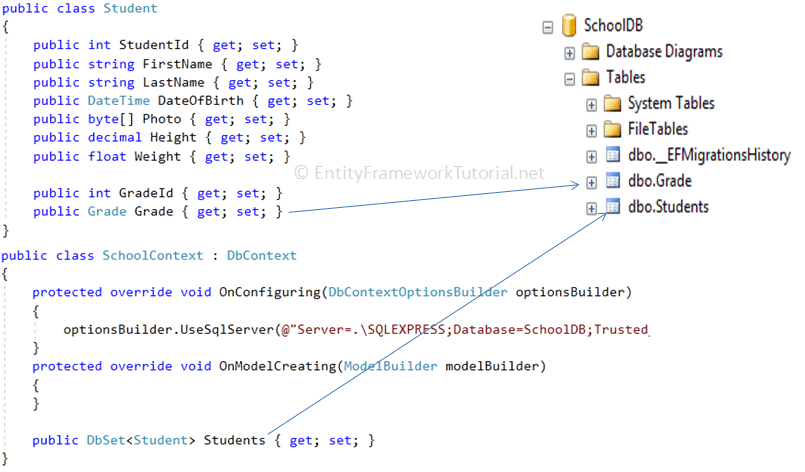
## **Schema**

EF Core will create all the database objects in the **dbo** schema by default.



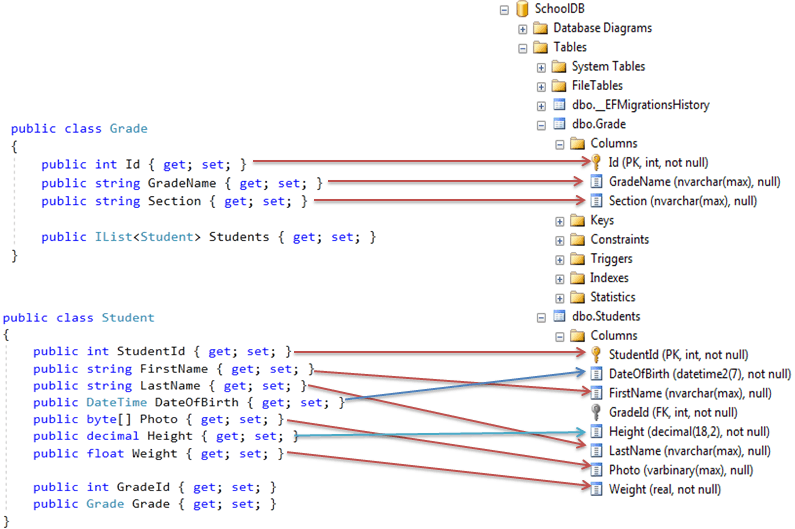
## **Table**

EF Core will create database tables for all DbSet<TEntity> properties in a context class with the same name as the property. It will also create tables for entities which are not included as DbSet properties but are reachable through reference properties in other DbSet entities.   
For the above example, EF Core will create the Students table for DbSet<Student> property in the SchoolContext class and the Grade table for a Grade property in the Student entity class, even though the SchoolContext class does not include the DbSet<Grade> property



## **Column**

EF Core will create columns for all the scalar properties of an entity class with the same name as the property, by default. It uses the reference and collection properties in building relationships among corresponding tables in the database.



## Column Data Type

The data type for columns in the database table is depending on how the provider for the database has mapped C# data type to the data type of a selected database. The following table lists mapping between C# data type to SQL Server column data type.

|  |  |
| --- | --- |
| 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 |

## **Nullable Column**

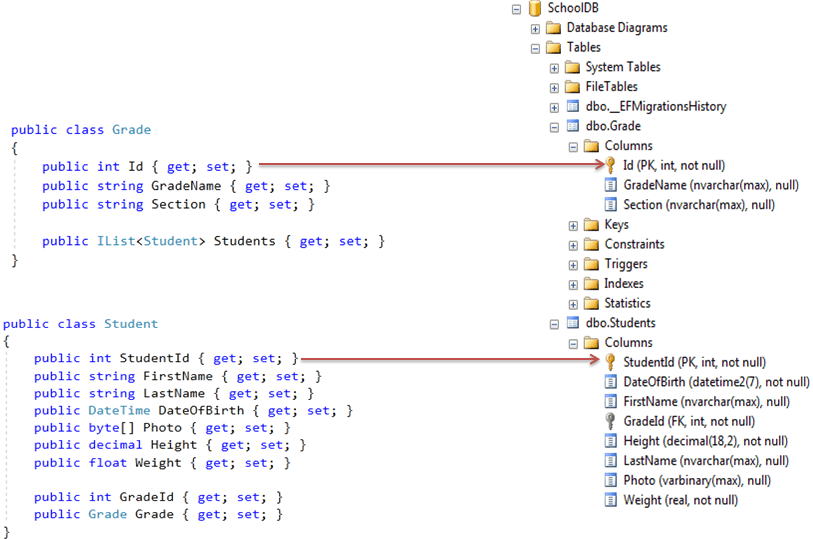
EF Core creates null columns for all reference data type and nullable primitive type properties e.g. string, Nullable<int>, decimal?.

## **NotNull Column**

EF Core creates NotNull columns in the database for all primary key properties, and primitive type properties e.g. int, float, decimal, DateTime etc..

## **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.



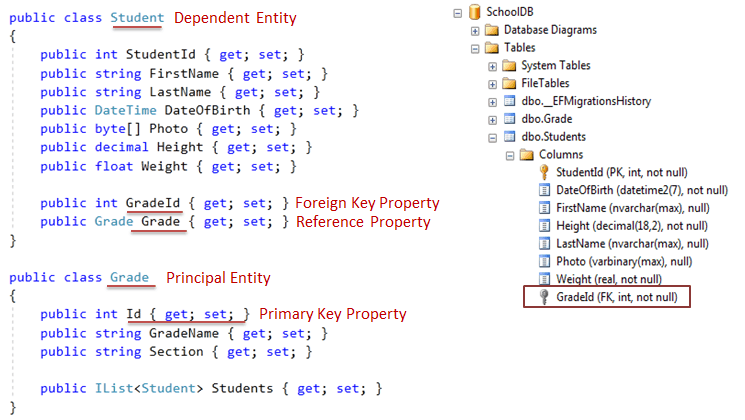
## **Foreign Key**

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>

In our example (Student and Grade entities), EF Core will create a foreign key column GradeId in the Students table, as depicted in the following figure.



The following table lists foreign key column names for different reference property names and primary key property names.

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

## **Index**

EF Core creates a clustered index on Primarykey columns and a non-clustered index on ForeignKey columns, by default.

# **One-to-Many Relationship Conventions in Entity Framework Core**

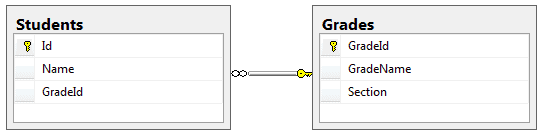
In the previous chapter, you learned about the EF conventions which map entities to different objects of the database. Here, you will learn about the relationship conventions between two entity classes that result in one-to-many relationships between corresponding tables in the database.

Entity Framework Core follows the same convention as [Entity Framework 6.x conventions for one-to-many relationship](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#conventions-for-one-to-many-ef6). The only difference is that EF Core creates a foreign key column with the same name as navigation property name and not as <NavigationPropertyName>\_<PrimaryKeyPropertyName>

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

|  |
| --- |
| **public class Student**  {  public int StudentId { get; set; }  public string StudentName { get; set; }  }  **public class Grade**  {  public int GradeId { get; set; }  public string GradeName { get; set; }  public string Section { get; set; }  } |

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

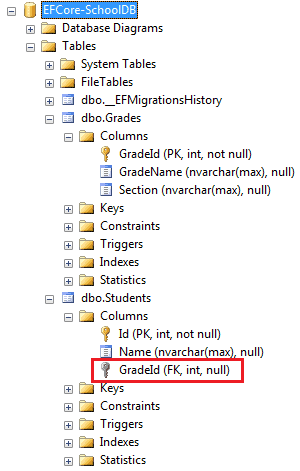


### **Convention 1**

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

|  |
| --- |
| // dependent entity  **public class Student**  {  public int Id { get; set; }  public string Name { get; set; }  public Grade Grade { get; set; } // a reference navigation property  }  // principal entity  **public class Grade**  {  public int GradeId { get; set; }  public string GradeName { get; set; }  public string Section { get; set; }  } |

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



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

### **Convention 2**

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

|  |
| --- |
| // dependent entity  **public class Student**  {  public int StudentId { get; set; }  public string StudentName { get; set; }  }  // principal entity  **public class Grade**  {  public int GradeId { get; set; }  public string GradeName { get; set; }  public string Section { get; set; }  **public ICollection<Student> Students { get; set; }** // a collection navigation property  } |

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

### **Convention 3**

Another EF 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).

|  |
| --- |
| // dependent entity  public class Student  {  public int Id { get; set; }  public string Name { get; set; }  public Grade Grade { get; set; } // a reference navigation property  }  // principal entity  public class Grade  {  public int GradeID { get; set; }  public string GradeName { get; set; }  public ICollection<Student> Students { get; set; } // a collection navigation property  } |

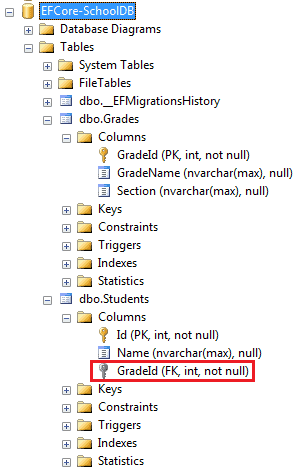
In the example above, the Student entity includes a reference navigation property of Grade type and the Grade entity class includes a collection navigation property ICollection<Student> , which results in a one-to-many relationship between corresponding database tables Students and Grades, same as in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

### **Convention 4**

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

|  |
| --- |
| // dependent entity  public class Student  {  public int Id { get; set; }  public string Name { get; set; }  **public int GradeId { get; set; }** // a foreign key property  **public Grade Grade { get; set; }** // a reference navigation property  }  // principal entity  public class Grade  {  public int GradeId { get; set; }  public string GradeName { get; set; }  **public ICollection<Student> Students { get; set; }** // a collection navigation property  } |

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



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

|  |
| --- |
| public class Student  {  public int Id { get; set; }  public string Name { get; set; }  public int? GradeId { get; set; }  public Grade Grade { get; set; }  } |

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

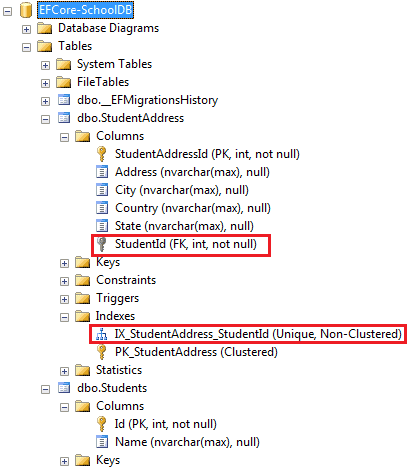
# One-to-One Relationship Conventions in Entity Framework Core

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

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

|  |
| --- |
| public class Student  {  public int Id { get; set; }  public string Name { get; set; }  public StudentAddress Address { get; set; } // a reference navigation property  }  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; } // a foreign key property  public Student Student { get; set; } // a reference navigation property  } |

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



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

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

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

# Configurations in Entity Framework Core

You learned about default [Conventions in EF Core](https://www.entityframeworktutorial.net/efcore/conventions-in-ef-core.aspx) in the previous chapter. 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 the EF model to database mappings.  
This programming pattern is referred to as [Convention over Configuration](https://en.wikipedia.org/wiki/Convention_over_configuration).

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

* By using Data Annotation Attributes
* By using Fluent API

## Data 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.

Data annotation attributes are not dedicated to Entity Framework, as they are also used in ASP.NET MVC. This is why these attributes are included in separate namespace : [*System.ComponentModel.DataAnnotations*](https://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations(v=vs.110).aspx).

The following example demonstrates how the data annotations attributes can be applied to a domain class and properties to override conventions. (mogą byś zastosowane do)

|  |
| --- |
| [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 annotation attributes are the same in EF 6 and EF Core. Visit [Data Annotations](https://www.entityframeworktutorial.net/code-first/dataannotation-in-code-first.aspx) chapter in the EF 6 section for more information.

## Fluent API

Another way to configure domain classes is by using Entity Framework Fluent API. EF Fluent API is based on a Fluent API design pattern (a.k.a [Fluent Interface](https://en.wikipedia.org/wiki/Fluent_interface)) where the result is formulated by [method chaining](https://en.wikipedia.org/wiki/Method_chaining).

# Fluent API in Entity Framework Core

In Entity Framework Core, the [ModelBuilder](https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.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.

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.

The following table lists important methods for each type of configuration.

|  |  |  |
| --- | --- | --- |
| Configurations | Fluent API Methods | Usage |
| Model Configurations | HasDbFunction() | Configures a database function when targeting a relational database. |
| HasDefaultSchema() | Specifies the database schema. |
| HasAnnotation() | Adds or updates data annotation attributes on the entity. |
| HasSequence() | Configures a database sequence when targeting a relational database. |
| Entity Configuration | HasAlternateKey() | Configures an alternate key in the EF model for the entity. |
| HasIndex() | Configures an index of the specified properties. |
| HasKey() | Configures the property or list of properties as Primary Key. |
| HasMany() | Configures the Many part of the relationship, where an entity contains the reference collection property of other type for one-to-Many or many-to-many relationships. |
| HasOne() | Configures the One part of the relationship, where an entity contains the reference property of other type for one-to-one or one-to-many relationships. |
| Ignore() | Configures that the class or property should not be mapped to a table or column. |
| OwnsOne() | Configures a relationship where the target entity is owned by this entity. The target entity key value is propagated from the entity it belongs to. |
| ToTable() | Configures the database table that the entity maps to. |
| Property Configuration | HasColumnName() | Configures the corresponding column name in the database for the property. |
| HasColumnType() | Configures the data type of the corresponding column in the database for the property. |
| HasComputedColumnSql() | Configures the property to map to computed column in the database when targeting a relational database. |
| HasDefaultValue() | Configures the default value for the column that the property maps to when targeting a relational database. |
| HasDefaultValueSql() | Configures the default value expression for the column that the property maps to when targeting relational database. |
| HasField() | Specifies the backing field to be used with a property. |
| HasMaxLength() | Configures the maximum length of data that can be stored in a property. |
| IsConcurrencyToken() | Configures the property to be used as an optimistic concurrency token. |
| IsRequired() | Configures whether the valid value of the property is required or whether null is a valid value. |
| IsRowVersion() | Configures the property to be used in optimistic concurrency detection. |
| IsUnicode() | Configures the string property which can contain unicode characters or not. |
| ValueGeneratedNever() | Configures a property which cannot have a generated value when an entity is saved. |
| ValueGeneratedOnAdd() | Configures that the property has a generated value when saving a new entity. |
| ValueGeneratedOnAddOrUpdate() | Configures that the property has a generated value when saving new or existing entity. |
| ValueGeneratedOnUpdate() | Configures that a property has a generated value when saving an existing entity. |

## Fluent API Configurations

Override the OnModelCreating method and use a parameter modelBuilder of type ModelBuilder to configure domain classes, as shown below.

|  |
| --- |
| 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();  }  } |

In the above example, the ModelBuilder Fluent API instance is used to configure a property by calling multiple methods in a chain. It configures the StudentId property of the Student entity; it configures the name using HasColumnName, the default value using HasDefaultValue and nullability using IsRequired method in a single statement instead of multiple statements. This increases the readability and also takes less time to write compare to multiple statements, as shown below.

|  |
| --- |
| //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(); |

|  |
| --- |
| **Note:** Fluent API configurations have higher precedence than data annotation attributes.  precedence = priority |

# Configure One-to-Many Relationships using Fluent API in Entity Framework Core

You learned about the [Conventions for One-to-Many Relationship](https://www.entityframeworktutorial.net/efcore/one-to-many-conventions-entity-framework-core.aspx).  
Generally, you don't need to configure one-to-many relationships because EF Core includes enough conventions which will automatically configure them.   
However, you 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.

Entity Framework Core made it easy to configure relationships using Fluent API.  
Consider the following Student and Grade classes where the Grade entity includes many Student entities.

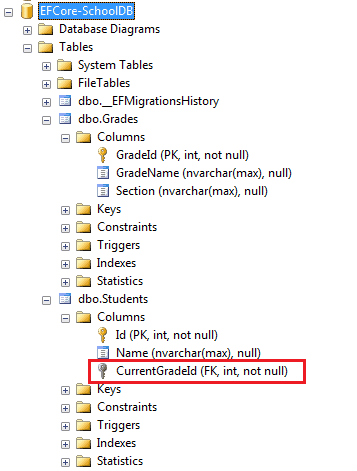
|  |
| --- |
| public class Student  {  public int Id { get; set; }  public string Name { get; set; }  public int CurrentGradeId { get; set; }  public Grade Grade { get; set; }  }  public class Grade  {  public int GradeId { get; set; }  public string GradeName { get; set; }  public string Section { get; set; }  public ICollection<Student> Students { get; set; }  } |

Configure the one-to-many relationship for the above entities using Fluent API by overriding  
the OnModelCreating method in the context class, as shown below.

|  |
| --- |
| public class SchoolContext : DbContext  {  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  optionsBuilder.UseSqlServer("Server=.\\SQLEXPRESS… ");  }  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; }  } |

In the example above, the following code snippet configures the one-to-many relationship:

Now, to reflect this in the database, execute [migration commands](https://www.entityframeworktutorial.net/efcore/entity-framework-core-migration.aspx), add-migration <name>   
and update-database . The database will include two tables with One-to-Many relationship as shown below.



Let's understand the above code step by step.

|  |
| --- |
| 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 .

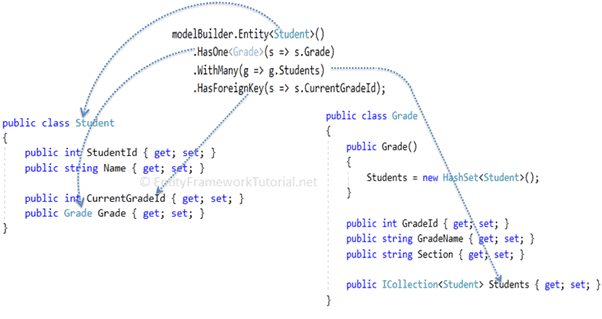
Now, we need to configure the other end of the relationship, the Grade entity.

The .WithMany(g => g.Students) specifies that the Grade entity class

includes many Student entities. Here, WithMany infers collection navigation property.  
( infer – wnioskować, infer from – wnioskować na podstawie czegoś )

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.

The following figure illustrates the above steps:



Alternatively, you can start configuring the relationship with the Grade entity instead of the Student entity, as shown below.

|  |
| --- |
| 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.

Use the OnDelete method to configure the cascade delete between Student and Grade entities, as shown below.

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

The OnDelete() method cascade delete behaviour uses the DeleteBehavior parameter.  
You can specify any of the following DeleteBehavior values, based on your requirement.

|  |  |
| --- | --- |
| Cascade | Dependent entities will be deleted when the principal entity is deleted. |
| ClientSetNull | The values of foreign key properties in the dependent entities will be set to null. |
| Restrict | Prevents Cascade delete. |
| SetNull | The values of foreign key properties in the dependent entities will be set to null. |

# Configure One-to-One Relationships using Fluent API in Entity Framework Core

Here you will learn how to configure one-to-one relationships between two entities using Fluent API, if they do not follow EF Core conventions.

Generally, you don't need to configure one-to-one relationships manually because EF Core includes [Conventions for One-to-One Relationships](https://www.entityframeworktutorial.net/efcore/one-to-one-conventions-entity-framework-core.aspx). 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.

Let's configure a one-to-one relationship between the following Student and StudentAddress entities, which do not follow the foreign key convention.

|  |
| --- |
| 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; }  } |

To configure a one-to-one relationship using Fluent API in EF Core, use:

the HasOne , WithOne and HasForeignKey methods, as shown below.

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

Let's understand it step by step.

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

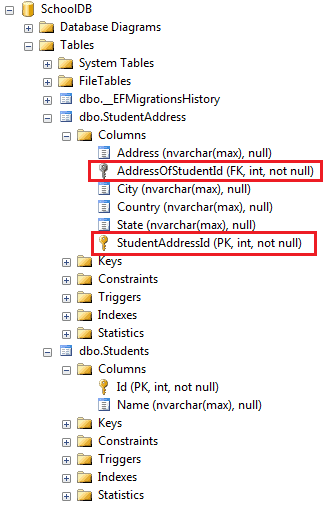
modelBuilder.Entity<Student>() starts configuring the Student entity.

The .HasOne<StudentAddress>(s => s.Address) method specifies that the Student entity includes one StudentAddress reference property using a lambda expression.

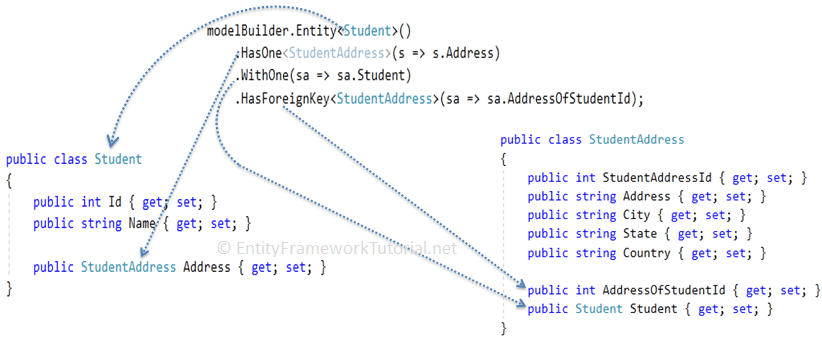
.WithOne(ad => ad.Student) configures the other end of the relationship, the   
 StudentAddress entity. It specifies that the StudentAddress entity includes a reference navigation property of Student type.

.HasForeignKey<StudentAddress>(ad => ad.AddressOfStudentId) specifies the foreign key property name.

Now, to reflect this in the database, execute [migration commands](https://www.entityframeworktutorial.net/efcore/entity-framework-core-migration.aspx), add-migration <name> and   
 update-database .   
The database will include two tables with one-to-one relationship as shown below.



The following figure illustrates the Fluent API configuration for a one-to-one relationship.



You can start configuring with the StudentAddress entity in the same way, as below.

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

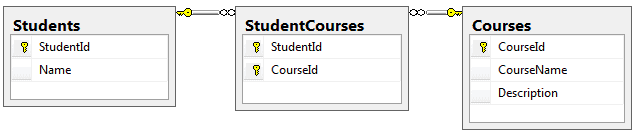
# Configure Many-to-Many Relationships in Entity Framework Core

Here you will learn how to configure many-to-many relationships between two entities using Fluent API in Entity Framework Core.

Let's implement a many-to-many relationship between the following Student and Course entities, where one student can enrols 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; }  } |

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.

The steps for configuring many-to-many relationships would the following:

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.

So, first of all, define the joining entity StudentCourse, as shown below.

|  |
| --- |
| public class StudentCourse  {  public int StudentId { get; set; }  public Student Student { get; set; }  public int CourseId { get; set; }  public Course Course { get; set; }  } |

The above joining entity StudentCourse includes reference navigation properties Student and Course and their foreign key properties StudentId and CourseId respectively (foreign key properties follow the convention).

Now, we also need to configure two separate one-to-many relationships between Student -> StudentCourse and Course -> StudentCourse entities. We can do it by just following the [convention for one-to-many relationships](https://www.entityframeworktutorial.net/efcore/one-to-many-conventions-entity-framework-core.aspx), as shown below.

|  |
| --- |
| 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; }**  } |

As you can see above, the Student and Course entities now include a collection navigation property of StudentCourse type. The StudentCourse entity already includes the foreign key property and navigation property for both, Student and Course. This makes it a fully defined one-to-many relationship between Student & StudentCourse and Course & StudentCourse.

Now, the foreign keys must be the composite primary key in the joining table. This can only be configured using Fluent API, as below.

|  |
| --- |
| public class SchoolContext : DbContext  {  protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)  {  optionsBuilder.UseSqlServer("Server=.\\SQLEXPRESS;Database=EFCore-SchoolDB;Trusted\_Connection=True");  }  protected override void OnModelCreating(ModelBuilder modelBuilder)  {  **base.OnModelCreating(modelBuilder);**  modelBuilder.Entity<StudentCourse>().HasKey(sc => new { sc.StudentId, sc.CourseId });  }  public DbSet<Student> Students { get; set; }  public DbSet<Course> Courses { get; set; }  public DbSet<StudentCourse> StudentCourses { get; set; }  } |

In the above code:  
modelBuilder.Entity<StudentCourse>().HasKey(sc =>new{ sc.StudentId, sc.CourseId })   
configures StudentId and CourseId as the composite key.

This is how you can configure many-to-many relationships if entities follow the conventions for one-to-many relationships with the joining entity. Suppose that the foreign key property names do not follow the convention (e.g. SID instead of StudentId and CID instead of CourseId), then you can configure it using Fluent API, as shown below.

|  |
| --- |
| protected override void OnModelCreating(ModelBuilder modelBuilder)  {  modelBuilder.Entity<StudentCourse>().HasKey(sc => new { sc.SId, sc.CId });  modelBuilder.Entity<StudentCourse>()  .HasOne<Student>(sc => sc.Student)  .WithMany(s => s.StudentCourses)  .HasForeignKey(sc => sc.SId);  modelBuilder.Entity<StudentCourse>()  .HasOne<Course>(sc => sc.Course)  .WithMany(s => s.StudentCourses)  .HasForeignKey(sc => sc.CId);  } |

**Note:** EF team will include a feature where we don't need to create a joining entity for many-to-many relationships in future. [Track this issue on GitHub](https://github.com/aspnet/EntityFrameworkCore/issues/1368).