<https://docs.microsoft.com/en-us/dotnet/standard/base-types/>

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; }

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

// Configure Student & StudentAddress entity

modelBuilder.Entity<Student>()

.HasOptional(s => s.Address) // Mark Address property optional in Student entity

.WithRequired(ad => ad.Student); // mark Student property as required in StudentAddress entity. Cannot save StudentAddress without Student

}

# **Configure One-to-Zero-or-One Relationship in Entity Framework 6**

Here, you will learn to configure One-to-Zero-or-One relationships between two entities.

We will implement a one-to-Zero-or-One relationship between the following Studentand StudentAddress entities.

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; }

}

Visit the [Entity Relationship](https://www.entityframeworktutorial.net/entity-relationships.aspx) section to understand how EF manages one-to-one, one-to-many, and many-to-many relationships.

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. So, we need to configure the above entities in such a way that EF creates the Students and StudentAddresses tables in the DB and makes the StudentId column in Student table as PrimaryKey (PK) and StudentAddressId column in the StudentAddresses table as PK and ForeignKey (FK) both.

## **Configure One-to-Zero-or-One Relationship using Data Annotation Attributes**

Here, we will apply data annotation attributes on the Student and StudentAddressentities to establish a one-to-zero-or-one relationship.

The Student entity follows the default [code-first convention](https://www.entityframeworktutorial.net/code-first/code-first-conventions.aspx) as it includes the StudentId property which will be the key property. So, we don't need to apply any attributes on it because EF will make the StudentId column as a PrimaryKey in the Students table in the database.

For the StudentAddress entity, we need to configure the StudentAddressId as PK & FK both. The StudentAddressId property follows the default convention for primary key. So, we don't need to apply any attribute for PK. However, we also need to make it a foreign key which points to StudentId of the Student entity. So, apply [ForeignKey("Student")] on the StudentAddressId property which will make it a foreign key for the Student entity, as shown below.

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; }

}

Thus, you can use data annotation attributes to configure a one-to-zero-or-one relationship between two entities.

**Note:** Student includes the StudentAddress navigation property and StudentAddressincludes 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.

## **Configure a One-to-Zero-or-One relationship using Fluent API**

Here, we will use Fluent API to configure a one-to-zero-or-one relationship between the Student and StudentAddress entities.

The following example sets a one-to-zero or one relationship between Student and StudentAddress using Fluent API.

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

// Configure Student & StudentAddress entity

modelBuilder.Entity<Student>()

.HasOptional(s => s.Address) // Mark Address property optional in Student entity

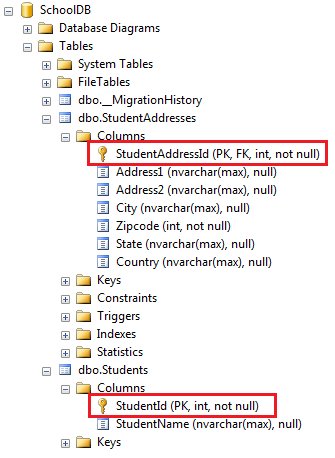
.WithRequired(ad => ad.Student); // mark Student property as required in StudentAddress entity. Cannot save StudentAddress without Student

}

In the above example, we start with the Student entity. The HasOptional() method configures the Address navigation property in Student entity as optional (not required when saving the Student entity). Then, the WithRequired() method makes the Student navigation property of StudentAddress as required (required when saving the StudentAddress entity; it will throw an exception when the StudentAddress entity is saved without the Student navigation property). This will also make the StudentAddressId as ForeignKey.

Thus, you can configure a one-to-Zero-or-one relationship between two entities where the Student entity can be saved without attaching the StudentAddress object to it but the StudentAddress entity cannot be saved without attaching an object of the Studententity. This makes one end required.

EF API will create the following tables in the database.

[](https://www.entityframeworktutorial.net/images/codefirst/onetoone-1.PNG)

## **Configure a One-to-One relationship using Fluent API**

We can configure a one-to-One relationship between entities using Fluent API where both ends are required, meaning that the Student entity object must include the StudentAddress entity object and the StudentAddress entity must include the Student entity object in order to save it.

**Note:**One-to-one relationships are technically not possible in MS SQL Server. These will always be one-to-zero-or-one relationships. EF forms One-to-One relationships on entities not in the DB.

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

// Configure StudentId as FK for StudentAddress

modelBuilder.Entity<Student>()

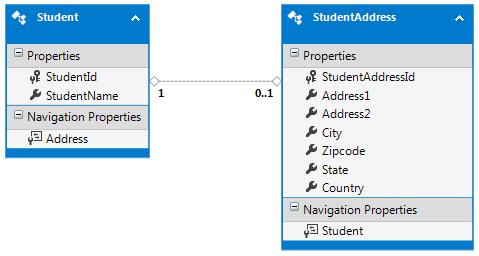
.HasRequired(s => s.Address)

.WithRequiredPrincipal(ad => ad.Student);

}

In the above example, modelBuilder.Entity<Student>().HasRequired(s => s.Address) makes the Address property of StudentAddress as required and .WithRequiredPrincipal(ad => ad.Student) makes the Student property of the StudentAddress entity as required. Thus it configures both ends as required. So now, when you try to save the Student entity without the StudentAddress entity without the Student, it will throw an exception.

Create a read-only Entity Data Model for the above example using [EF Power Tools](https://www.entityframeworktutorial.net/code-first/entity-framework-power-tools.aspx). The entities will appear like the diagram shown below:

[](https://www.entityframeworktutorial.net/images/codefirst/onetoone-2.PNG)

Learn how to configure a one-to-many relationship in the next section.

# **Configure One-to-Many Relationships in EF 6**

Here, we will learn how to configure One-to-Many relationships between two entities (domain classes) in Entity Framework 6.x using the code-first approach.

Let's configure a one-to-many relationship between the following Student and Gradeentities where there can be many students in one grade.

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

}

public class Grade

{

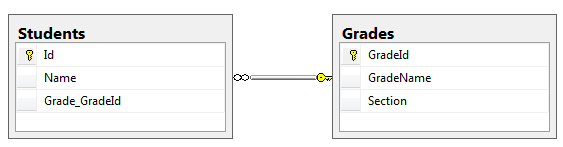
public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

}

After implementing the one-to-many relationship in the above entities, the database tables for Student and Grade will look like below.

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-db.PNG)

The one-to-many relationship can be configured in the following ways.

1. [By following Conventions](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#conventions-for-one-to-many-ef6)
2. [By using Fluent API Configurations](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#configure-one-to-many-using-fluent-api)

## **Conventions for One-to-Many Relationships**

There are certain conventions in Entity Framework which if followed in entity classes (domain classes) will automatically result in a one-to-many relationship between two tables in the database. You don't need to configure anything else.

Let's look at an example of all the conventions which create a one-to-many relationship.

### Convention 1

We want to establish a one-to-many relationship between the Student and Gradeentities where many students are associated with one Grade. It means that each Student entity points to a Grade. This can be achieved by including a reference navigation property of type Grade in the Student entity class, as shown below.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

**public Grade Grade { get; set; }**

}

public class Grade

{

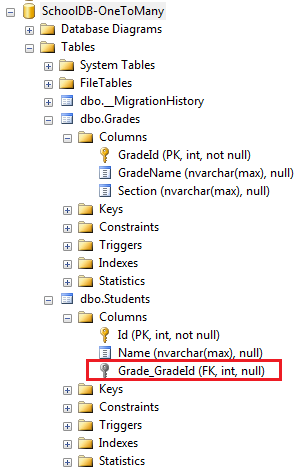
public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

}

In the above example, the 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 as shown below.

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-1.PNG)

Notice that the reference property is nullable, so it creates a nullable foreign key column Grade\_GradeId in the Students table. You can [configure NotNull foreign key using fluent API](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#notnull-foreignkey-using-fluent-api).

### Convention 2

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

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

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

}

In the above example, the Grade entity includes a collection navigation property of type ICollection<Student>. This also results in a one-to-many relationship between the Student and Grade entities. This example produces the same result in the database as convention 1.

### Convention 3

Including navigation properties at both ends will also result in a one-to-many relationship, as shown below.

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; }

**public ICollection<Student> Student { get; set; }**

}

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

### Convention 4

A fully defined relationship at both ends will create a one-to-many relationship, 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; }**

}

public class Grade

{

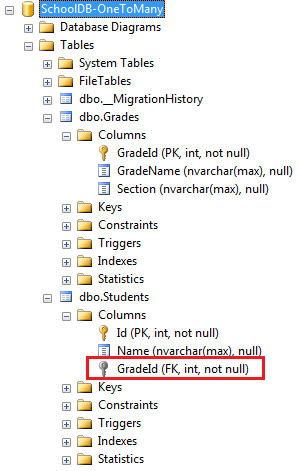
public int GradeId { get; set; }

public string GradeName { get; set; }

**public ICollection<Student> Student { get; set; }**

}

In the above example, the Student entity includes foreign key property GradeId with its reference property Grade. This will create a one-to-many relationship with the NotNull foreign key column in the Students table, as shown below.

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-2.PNG)

If the data type of GradeId is nullable integer, then it will create a null foreign key.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public int? GradeId { get; set; }

public Grade Grade { get; set; }

}

The above code snippet will create a nullable GradeId column in the database because we have used Nullable<int> type (? is a shortcut for Nullable<int>)

## **Configure a One-to-Many Relationship using Fluent API**

Generally, you don't need to configure the one-to-many relationship in entity framework because one-to-many relationship conventions cover all combinations. However, you may configure relationships using Fluent API at one place to make it more maintainable.

Consider the following Student and Grade entity classes.

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; }

}

You can configure a 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

{

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); }

}

}

Let's understand the above code step by step.

* First, we need to start configuring with any one entity class. So, modelBuilder.Entity<student>() starts with the Student entity.
* Then, .HasRequired<Grade>(s => s.CurrentGrade) specifies that the Student entity has required the CurrentGrade property. This will create a NotNull foreign key column in the DB.
* Now, it's time to configure the other end of the relationship - the Grade entity.
* .WithMany(g => g.Students) specifies that the Grade entity class includes many Student entities. Here, many infers the ICollection type property.
* Now, if the Student entity does not follow the Id property convention for foreign key, then we can specify the name of the foreign key using the HasForeignKeymethod.
* .HasForeignKey<int>(s => s.CurrentGradeId); specifies the foreign key property in the Student entity.

Alternatively, you can start configuring the relationship with the Grade entity instead of the Student entity. The following code produces the same result as above.

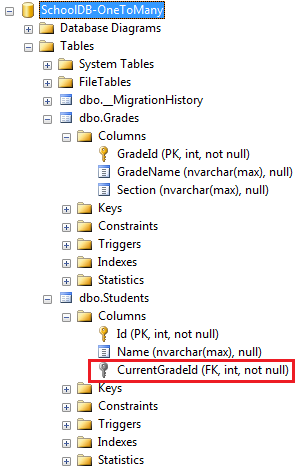
modelBuilder.Entity<Grade>()

.HasMany<Student>(g => g.Students)

.WithRequired(s => s.CurrentGrade)

.HasForeignKey<int>(s => s.CurrentGradeId);

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

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-4.PNG)

### Configure the NotNull ForeignKey using Fluent API

In convention 1, we have seen that it creates an optional one-to-many relationship which in turn creates a nullable foreign key column in the database. To make it a NotNull column, use the HasRequired() method as shown below.

modelBuilder.Entity<Student>()

.HasRequired<Grade>(s => s.CurrentGrade)

.WithMany(g => g.Students);

### Configure Cascade Delete using Fluent API

Cascade delete means automatically deleting child rows when the related parent row is deleted. For example, if Grade is deleted then all the students in that Grade should also be deleted automatically. The following code configures the cascade delete using the WillCascadeOnDelete method.

modelBuilder.Entity<Grade>()

.HasMany<Student>(g => g.Students)

.WithRequired(s => s.CurrentGrade)

.WillCascadeOnDelete();

# **Configure Many-to-Many Relationships in Code-First**

Here, we will learn how to configure a Many-to-Many relationship between the Studentand Course entity classes. Student can join multiple courses and multiple students can join one Course.

Visit the [Entity Relationship](https://www.entityframeworktutorial.net/entity-relationships.aspx) chapter to understand how EF manages one-to-one, one-to-many and many-to-many relationships between entities.

## **Many-to-Many Relationship by Following Conventions**

EF 6 includes default conventions for many-to-many relationships. You need to include a collection navigation property at both ends. For example, the 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, as shown below:

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; }

}

The following is the context class that includes the Student and Course entities.

public class SchoolDBContext : DBContext

{

public SchoolDBContext() : base("SchoolDB-DataAnnotations")

{

}

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

public DbSet<Course> Courses { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

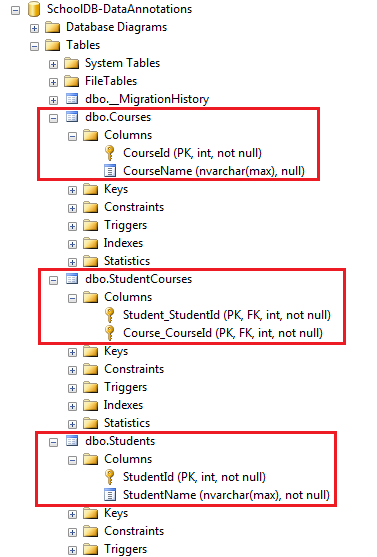
{

base.OnModelCreating(modelBuilder);

}

}

EF API will create Students, Courses and also the joining table StudentCourses in the database for the above example. The StudentCourses table will include the PK (Primary Key) of both tables - Student\_StudentId & Course\_CourseId, as shown below.

[](https://www.entityframeworktutorial.net/images/codefirst/manytomany-fg.PNG)

**Note:** EF automatically creates a joining table with the name of the both entities and the suffix 's'.

## **Configure a Many-to-Many Relationship using Fluent API**

As you have seen above, the default conventions for many-to-many relationships creates a joining table with the default naming conventions. Use Fluent API to customize a joining table name and column names, as shown below:

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<Student>()

.HasMany<Course>(s => s.Courses)

.WithMany(c => c.Students)

.Map(cs =>

{

cs.MapLeftKey("StudentRefId");

cs.MapRightKey("CourseRefId");

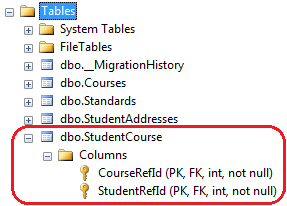
cs.ToTable("StudentCourse");

});

}

In the above example, the HasMany() and WithMany() methods are used to configure a many-to-many relationship between the Student and Course entities. The Map()method takes Action type delegate, hence, we can pass the lambda expression to customize column names in a joining table. We can specify the PK property name of Student in MapLeftKey() (we started with the Student entity, so it will be the left table) and the PK of the Course table in MapRightKey() method. The ToTable()method specifies the name of a joining table (StudentCourse in this case).

The above code will create a joining table StudentCourse with two Primary Keys StudentRefId and CourseRefId which will also be Foreign Keys, as shown below:

[](https://www.entityframeworktutorial.net/images/codefirst/manytomany-fg2.PNG)

In this way, you can override the default conventions for many-to-many relationship and customize a joining table name and its columns.

Create table course

(

Course\_Id int primary key,

CourseName nvarchar(25)

)

Create table Grade

(

GradeId int primary key,

GradeName nvarchar(20),

Section nvarchar(20)

)

Create table Student

(

StudentID int primary key,

StudentName nvarchar(20),

DateOfBirth datetime,

Height longint,

Weight smallint,

RowVersion smallint,

GradeId int not null

)

Create table StudentAddress

(

StudentAddressID int,

Address1 nvarchar(20),

Address2 nvarchar(20),

City nvarchar(20),

Zipcode int ,

State nvarchar(20),

Country nvarchar(20)

)

Create table Teacher

(

Id int primary key,

TeacherName nvarchar(25),

TeachingMode nvarchar(20)

)

### /\* How to work with Regular Expression Example 1: Replacing Substrings

Assume that a mailing list contains names that sometimes include a title (Mr., Mrs., Miss, or Ms.) along with a first and last name. If you do not want to include the titles when you generate envelope labels from the list, you can use a regular expression to remove the titles, as the following example illustrates.

\*/

using System;

using System.Text.RegularExpressions;

public class Example

{

public static void Main()

{

string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";

string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",

"Abraham Adams", "Ms. Nicole Norris" };

foreach (string name in names)

Console.WriteLine(Regex.Replace(name, pattern, String.Empty));

}

}

// The example displays the following output:

// Henry Hunt

// Sara Samuels

// Abraham Adams

// Nicole Norris

/\* how to identify duplicate words \*

The regular expression pattern \b(\w+?)\s\1\b can be interpreted as follows:

|  |  |
| --- | --- |
| \b | Start at a word boundary. |
| (\w+?) | Match one or more word characters, but as few characters as possible. Together, they form a group that can be referred to as \1. |
| \s | Match a white-space character. |
| \1 | Match the substring that is equal to the group named \1. |
| \b | Match a word boundary. |

\*/

using System;

using System.Text.RegularExpressions;

public class Class1

{

public static void Main()

{

string pattern = @"\b(\w+?)\s\1\b";

string input = "This this is a nice day. What about this? This tastes good. I saw a a dog.";

foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))

Console.WriteLine("{0} (duplicates '{1}') at position {2}",

match.Value, match.Groups[1].Value, match.Index);

}

}

// The example displays the following output:

// This this (duplicates 'This') at position 0

// a a (duplicates 'a') at position 66