robertmonden.com

Relationships in Entity Framework

Robert Monden

robertmonden.com

April 1, 2026

- Overview
- Relational databases and entity relations
- Entity relations in Entity Framework
- Deletion of entities subject to a relationship
- Conclusion

robertmonden.com

- A **relational database** is a collection of structured data, stored in **tables** that are organized into **columns** and **rows**.
- For example: a database for an online learning platform likely includes a `Student` table, to store the students, and a `Course` table, to store the available courses.
- Tables typically have a unique **primary key**, composed of one or more columns, in order to identify each row in the table.
- A column that refers to a primary key of another table, is referred to as a **foreign key**.

Let us define the `Student` and `Course` tables as follows:

Student	**Course**
ID	ID
EmailAddress	Title
FirstName	Teacher
LastName	

Using these table definitions as a base we can define the relations between the two tables. There are various relations we can specify:

- A **one-to-one relationship** to express that each student has exactly one course and each course has only one student.
- A **one-to-many relationship** to express that each student can take a multiple of courses, but each course has only one student.
- A **many-to-one relationship** to express that each student can take one course, possibly also taken by other students.
- A **many-to-many relationship** to express that a student can take any multiple of courses, with each course accepting any multiple of students.

Of course, an extension of the above types would allow for optional relations. For example: a student may have at most one course, but also not be enrolled in any course at all.

The first three relation types are trivial to assign:

- 1:1 relationship: add a foreign key to `Course.ID` in `Student`, such that `Student.CourseID` $\mapsto$ `Course.ID` and `Course.ID` $\exists$.
- 1:M relationship: add a foreign key to `Student.ID` in `Course`, such that `Course.StudentID` $\mapsto$ `Student.Id` and `Student.ID` $\exists$.

By the above it should be obvious that on a database level by default there is no real constraint on the maximum number of related entities.

For example, multiple students may have a foreign key reference to the same course.

We *can*, of course, make the foreign key column nullable, meaning that a student may not be enrolled in any course at all. In this case we have an **optional relationship**.

Many-to-many relationships are slightly more tricky to implement, since they require an additional table known as a **linking table**. Remember: cells can only contain a single value.

For instance, if students can enroll to multiple courses, which can in turn be attended by multiple students, we need a table to store the following values:

$$(\text{Student.ID}, \text{Course.ID}),$$

where both foreign keys refer to existing records in their respective tables.

Let us name this table `CourseEnrollments`. Note that this table may also have a primary key of its own, such that each record has the format

$$(\text{ID}, \text{Student.ID}, \text{Course.ID}),$$

Before moving on to relationship definition in Entity Framework, let us revisit how table structures can be defined in a code-first approach:

- Through data annotations, which are easier to set up but more limited (e.g., no composite keys);[1]
- Through model builders with **Fluent API**, to specify configuration without modifying the entity classes, in a more flexible and fine-grained manner. These override data annotations.

[1] Though setting up auto-incrementing ID fields might be easier with data annotations than with Fluent API.

Although we will focus on entity configuration through Fluent API, a simple example of a *data annotations-based approach*:

```
1  [Table("Student")]
2  public class Student
3  {
4      [Key]
5      public Guid Id { get; set; }
6
7      [Required]
8      [MaxLength(50)]
9      public string FirstName { get; set; }
10
11     [Required]
12     [MaxLength(50)]
13     public string LastName { get; set; }
14
15     ...
16 }
17
```

To set up the `Student` table in the same was as in the previous example, we would have to use the following code:

```
1  modelBuilder.Entity<Student>(entity =>
2  {
3      entity.HasKey(s => s.Id);
4
5      entity
6          .Property(s => s.FirstName)
7          .IsRequired()
8          .HasMaxLength(50);
9
10     entity
11         .Property(s => s.LastName)
12         .IsRequired()
13         .HasMaxLength(50);
14
15     ...
16 });
17
```

Of course, the entity model is in this case identical, with the difference that the class no longer needs any EF-attributes. We can declare the `Course` entity in the same manner.

Student

```
1  public class Student
2  {
3      public Guid Id { get; set; }
4
5      public string FirstName { get; set; }
6
7      public string LastName { get; set; }
8
9      ...
10 }
11
```

Course

```
1  public class Course
2  {
3      public Guid Id { get; set; }
4
5      public string Title { get; set; }
6
7      public string Teacher { get; set; }
8
9      ...
10 }
11
```

To implement a 1:1 relationship, we can extend both the `Student` and `Course` models with **so-called navigation properties**:

Student

```
1  public class Student
2  {
3      public Guid Id { get; set; }
4
5      public string FirstName { get; set; }
6
7      public string LastName { get; set; }
8
9      public Guid CourseId { get; set; }
10     public Course? Course { get; set; }
11
12     ...
13 }
14
```

Course

```
1  public class Course
2  {
3      public Guid Id { get; set; }
4
5      public string Title { get; set; }
6
7      public string Teacher { get; set; }
8
9      public Guid StudentId { get; set; }
10     public Student? Student { get; set; }
11
12     ...
13 }
14
```

In both tables, we have made the ID required and the object optional: we want EF to load the related data only if we explicitly tell it to.

M:M relationships can be implemented through two separate 1:M relationships, which an explicit linking table containing a reference to both tables. For instance:

```
1  public class CourseStudents
2  {
3      public Guid Id { get; set; }
4
5      public Guid CourseId { get; set; }
6      public Course Course { get; set; } = null!;
7
8      public Guid StudentId { get; set; }
9      public Student Student { get; set; } = null!;
10 }
11
```

Now we can modify `Course` such that:

```
1  public class Course
2  {
3      public Guid Id { get; set; }
4
5      public string Title { get; set; }
6
7      public string Teacher { get; set; }
8
9      public List<CourseStudents> Students { get; set; } = [];
10 }
11
```

A list of courses, including their students, can then be loaded through

```
1 var students = dbContext.Courses
2     .Include(c => c.CourseStudents)
3         .ThenInclude(cs => cs.Student)
4     .ToList()
```

However, it is not necessary to create an explicit joining table. In this case, `Course` has a property `Students: List<Student>`, meaning we can access the students directly:

```
1 var students = dbContext.Courses
2     .Include(c => c.Students)
3     .ToList()
```

In this case, EF has created a joining table for us.

Another alternative is to have EF 'detect' the joining table we have created ourselves. In that case we would also interact directly with the foreign entities, rather than through the joining table. Or we could even specify it manually using `.UsingEntity<T>()` in Fluent API.

robertmonden.com

As a quick detour, there are several ways to load related data, e.g., the `Students` associated with a `Course`:

- **Eager loading** through `.Include(c => Students)`, where any time we load a `Course`, the related student(s) are loaded as well (single query);
- **Lazy loading** through virtual navigation properties, deferring the loading of associated data until it is requested (implies multiple queries);
- **Explicit loading**, where data is loaded explicitly through '.Load()' after the parent entity has already been loaded.

We will assume eager loading from this point on.

The possible relations between `Course` and `Student` are simple. There is no real need to use Fluent API to configure the relations. Through so-called **mapping by convention**, the following code is generated:[2]

```
1 modelBuilder.Entity<Course>
2       .HasMany(c => c.Students)
3       .WithMany(s => s.Courses);
```

The above course tells us that a course **has** many students, **with** these students in turn being able to have multiple students.

Hence, this would mean that if each student can only attend one course, then:

```
1 modelBuilder.Entity<Course>
2       .HasMany(c => c.Students)
3       .WithOne(s => s.Courses);
```

The relationship can be defined on **either side**.

[2] Technically, even more is generated, but we'll ignore that for now.

Before we move on to deletion of entities subject to a relationship, let us talk about the **change tracker** (CT).

Each database context contains a CT to keep track of the changes to be executed as soon as `.SaveChanges()` is called.

Possible entity states include:

- `detached` - not tracked by the CT
- `unchanged` - no pending changes
- `deleted` - entity pending deletion

The **deletion behavior** describes what should happen if either party of the relationship is deleted, if anything at all:

- If A depends on B, but B is deleted, what should happen to A?
- If $B \subset A$ and A is deleted, what should happen to B?

Changing the deletion behavior of either party in the relationship does *not* require a new migration, since it affects the CT only.

Two situations:

- If $B \rightarrow A$ through non-nullable $FK_{B \rightarrow A}$, then deleting A implies B is also deleted: a profile must be associated with a user
- If $A \rightarrow B$ through a nullable $FK_{A \rightarrow B}$, then deleting B may result in either $FK_{A \rightarrow B}$ being set to `NULL` or A being deleted.

The latter is known as a **cascade delete**, triggered as a result of the parent entity's deletion.

In the first situation, we can refer to B (e.g., a profile) as an **orphan**, since it is no longer associated with its principal A (e.g., a user).

Consider the following lines of code:

```
1 var student = await dbContext.Students.Include(s => s.Grades).FirstAsync();
2 student.Grades.Clear();
3 await dbContext.SaveChangesAsync();
```

In this case we would sever the relationship by setting $FK_{\text{Grade}\rightarrow\text{Student}}$ to `NULL`. As a result, each record of `Grade` would be deleted.

Dependent entity deletion can happen in two situations:

- **Cascade delete** is triggered when the parent is deleted and $FK_{\text{Child}\rightarrow\text{Parent}}$ is non-nullable;
- **Orphan deletion** is triggered when both parent and child entities exist, but the relationship between then is severed.

By default, if a foreign key reference is non-nullable and all relevant dependent entities in a relationship are loaded, EF takes care of cascade deletion.

If some dependent entity B is not loaded, but deleting A would cause a foreign key constraint error, then an exception is thrown.

To get around this, dependent entity deletion can also be configured on a database level (but only to a certain extent)[3].

[3] For instance, since EF represents relationships both through navigations and foreign keys, whereas navigations do not exist in databases.

If $FK_{B \to A}$ is nullable and A is deleted, then EF does not perform any cascade delete. Hence, if we want to ensure that deleting A also results in the deletion of B, it needs to be configured manually through FluentAPI:

```
1 modelBuilder
2       .Entity<Course>()
3       .HasMany(c => c.Students)
4       .WithOne(s => s.Course)
5       .OnDelete(DeleteBehavior.ClientCascade);
```

This particular delete behavior (`ClientCascade`) is applied only on an EF level. Thus, it is only applied to tracked entities.

The following delete behaviors are supported:

DeleteBehavior	Database Changes	Effect
Cascade	`ON DELETE CASCADE`	Delete dependent entities
Restrict	`ON DELETE RESTRICT`	Disallow deletion if dependent entity exists
NoAction	none	None
SetNull	`ON DELETE SET NULL`	Set FK to `NULL`
ClientSetNull	none	Set FK to `NULL`
ClientCascade	none	Delete dependent entities
ClientNoAction	none	None

Recall that the client deletion behaviors require dependent entities to be tracked, in order for them to be cleaned up.

robertmonden.com

A table can refer to another through a relationship, which may be optional. These relationships are usually expressed through some foreign key $FK_{A \to B}$.

If B depends on A and A is deleted, B can no longer exist, otherwise the data integrity is violated. B should then be deleted.

If B has an optional dependency on A and A is deleted, the foreign key $FK_{A \to B}$ should be set to `NULL`.

The deletion behavior can be specified both on EF and a DB level.

”What Is A Relational Database (RDBMS)? | Google Cloud.” Google Cloud, 19 Feb. 2026, cloud.google.com/learn/what-is-a-relational-database.

”Relational Data Modeling - Cardinality.” Datacadamia - Data and Co, 30 Aug. 2024, www.datacadamia.com/data/type/relation/modeling/cardinality.

”Introduction to relationships - EF Core.” 20 Aug. 2025, learn.microsoft.com/en-us/ef/core/modeling/relationships.

”Creating and Configuring a Model - EF Core.” 30 Oct. 2025, learn.microsoft.com/en-us/ef/core/modeling.

Odoh, Benedict. ”Data Annotations vs Fluent API: Mastering Entity Relationships in Entity Framework Core.” Medium, 18 Sept. 2025, benedictodoh.medium.com/data-annotations-vs-fluent-api-mastering-entity-relationships-in-entity-framework-core-8a986fecccd4.

”Eager, Lazy and Explicit Loading with Entity Framework Core | The .NET Tools Blog.” JetBrains Blog, 1 Mar. 2026, blog.jetbrains.com/dotnet/2023/09/21/eager-lazy-and-explicit-loading-with-entity-framework-core.

”Many-to-many relationships - EF Core.” 30 Oct. 2025, learn.microsoft.com/en-us/ef/core/modeling/relationships/many-to-many.

”Conventions for relationship discovery - EF Core.” 30 Oct. 2025, learn.microsoft.com/en-us/ef/core/modeling/relationships/conventions.

”Cascade Delete - EF Core.” 30 Oct. 2025, learn.microsoft.com/en-us/ef/core/saving/cascade-delete

”Change Tracking - EF Core.” 30 Oct. 2025, learn.microsoft.com/en-us/ef/core/change-tracking.