

# InverseProperty Attribute in Entity Framework Core

Dot Net Tutorials : 17-22 minutes

Back to: [ASP.NET Core Tutorials For Beginners and Professionals](#)

In this article, I will discuss the **InverseProperty Data Annotation Attribute in Entity Framework Core (EF Core)** with Examples. Please read our previous article discussing [Index Attributes in Entity Framework Core](#) with Examples.

## What is InverseProperty Attribute in Entity Framework Core?

The InverseProperty attribute in Entity Framework Core (EF Core) explicitly defines the inverse navigation properties between two related entities, especially when the relationship is ambiguous or complex, and EF Core cannot automatically determine which navigation properties are paired together. It helps EF Core understand how two entities relate to each other by specifying the corresponding navigation property on the other side of the relationship. This situation often arises when:

- **Multiple Relationships Between Two Entities:** If two entities have more than one relationship, EF Core might have difficulty determining which navigation properties should correspond to each other. The InverseProperty attribute clearly specifies these relationships.
- **Self-referencing Entities:** When an entity has a relationship with itself (like a hierarchical relationship), the InverseProperty attribute can help clarify which navigation properties correspond to each other.

## Example to Understand the InverseProperty Attribute in EF Core

Let us understand InverseProperty Attribute in EF Core with one example. To understand this concept, we will create two Entities, i.e., **Course** and **Teacher**. Here, a teacher can teach multiple courses, but a course has one online and one offline teacher, making it a scenario where multiple relationships exist between the two entities.

In our example, the Teacher Entity will be the Principal Entity, and the Course Entity will be the Dependent Entity. We will create the foreign keys inside the Dependent Entity to establish the relationships. First, I will show you the example using single relationships between these two entities. Then, I will show multiple relationships between these two entities. Then, we will see the problem, and finally, we will see how to overcome the problem using the InverseProperty Attribute in EF Core.

### Course Entity:

Create a class file named **Course.cs**, and copy and paste the following code. Here, we have created the following class with CourseId, CourseName, and Description properties along with the Teacher Reference Navigation Property, which makes the relationship One-To-One between the Course and Teacher, i.e., one course can be taught by a single teacher.

```
namespace EFCoreCodeFirstDemo.Entities

public int CourseId { get; set; }

public string? CourseName { get; set; }

public string? Description { get; set; }

public Teacher? OnlineTeacher { get; set; }

namespace EFCoreCodeFirstDemo.Entities { public class Course { public int CourseId { get; set; } public string? CourseName { get; set; } public string? Description { get; set; } public Teacher? OnlineTeacher { get; set; } } }

namespace EFCoreCodeFirstDemo.Entities
{
    public class Course
    {
```

```

        public int CourseId { get; set; }
        public string? CourseName { get; set; }
        public string? Description { get; set; }
        public Teacher? OnlineTeacher { get; set; }
    }
}

```

### Teacher Entity

Create another class file named **Teacher.cs** and copy and paste the following code. Here, you can see that we have created the following class with the TeacherId and Name properties along with the Course Collection Navigation Property, which makes the relationship between the Teacher and Course one-to-many, i.e., one Teacher can teach multiple courses.

```

namespace EFCoreCodeFirstDemo.Entities

public int TeacherId { get; set; }

public string? Name { get; set; }

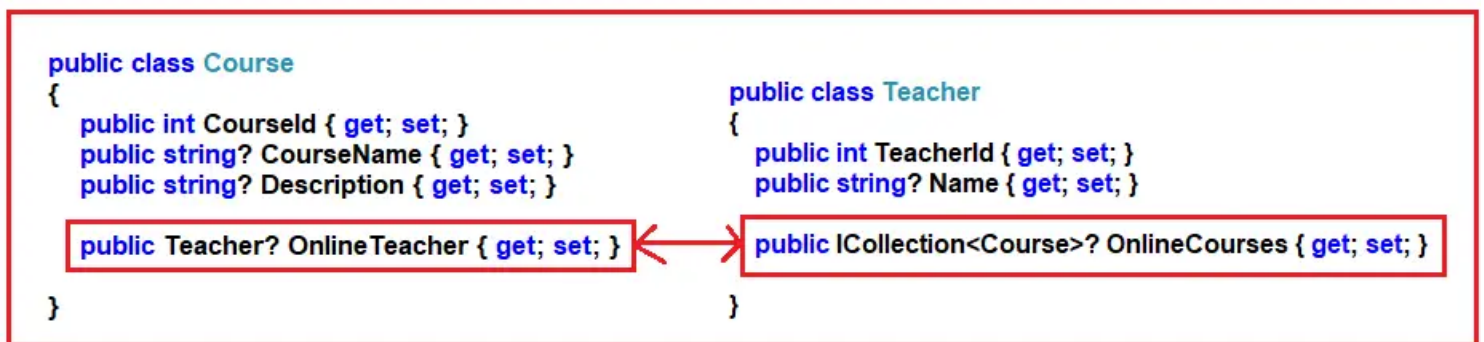
public ICollection<Course>? OnlineCourses { get; set; }

namespace EFCoreCodeFirstDemo.Entities { public class Teacher { public int TeacherId { get; set; } public string? Name { get; set; } public
ICollection<Course>? OnlineCourses { get; set; } } }

namespace EFCoreCodeFirstDemo.Entities
{
    public class Teacher
    {
        public int TeacherId { get; set; }
        public string? Name { get; set; }
        public ICollection<Course>? OnlineCourses { get; set; }
    }
}

```

Now, the two entities have a one-to-many relationship, and this is possible because of the Navigation Property, as shown in the image below. So, each entity must have a navigation property that maps to the navigation property of the other entity.



Here, in the Course entity, the Reference Navigation Property is OnlineTeacher, which points to the Teacher entity. On the other hand, in the Teacher entity, we have the OnlineCourse collection navigation property, which points to the Course entity. Here, it implements one-to-many relationships between Teachers and Courses, i.e., one teacher can teach multiple courses, and a single course can only be taught by one teacher.

### Modifying the Context Class:

Next, modify the EFCoreDbContext class as follows:

```
using Microsoft.EntityFrameworkCore;
```

```
namespace EFCoreCodeFirstDemo.Entities
```

```
public class EFCoreDbContext : DbContext
```

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
```

```
optionsBuilder.UseSqlServer(@"Server=LAPTOP-6P5NK25R\SQLSERVER2022DEV;Database=EFCoreDB;Trusted_Connection=True;TrustServerCertificate=True;");
```

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

```
public DbSet<Teacher> Teachers { get; set; }
```

```
using Microsoft.EntityFrameworkCore; namespace EFCoreCodeFirstDemo.Entities { public class EFCoreDbContext : DbContext { protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) { optionsBuilder.UseSqlServer(@"Server=LAPTOP-6P5NK25R\SQLSERVER2022DEV;Database=EFCoreDB;Trusted_Connection=True;TrustServerCertificate=True;"); } public DbSet<Course> Courses { get; set; } public DbSet<Teacher> Teachers { get; set; } }
```

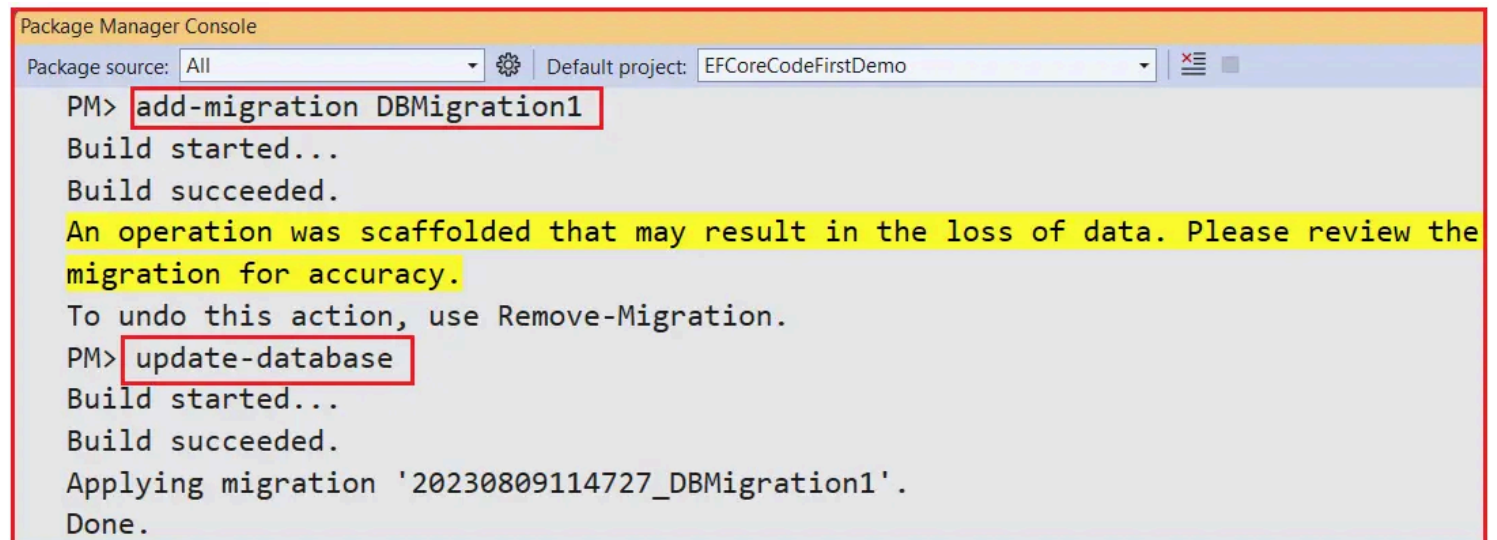
```
using Microsoft.EntityFrameworkCore; namespace EFCoreCodeFirstDemo.Entities
```

```
{
    public class EFCoreDbContext : DbContext
    {
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Server=LAPTOP-6P5NK25R\SQLSERVER2022DEV;Database=EFCoreDB;Trusted_Connection=True;TrustServerCertificate=True;");
        }

        public DbSet<Course> Courses { get; set; }
        public DbSet<Teacher> Teachers { get; set; }
    }
}
```

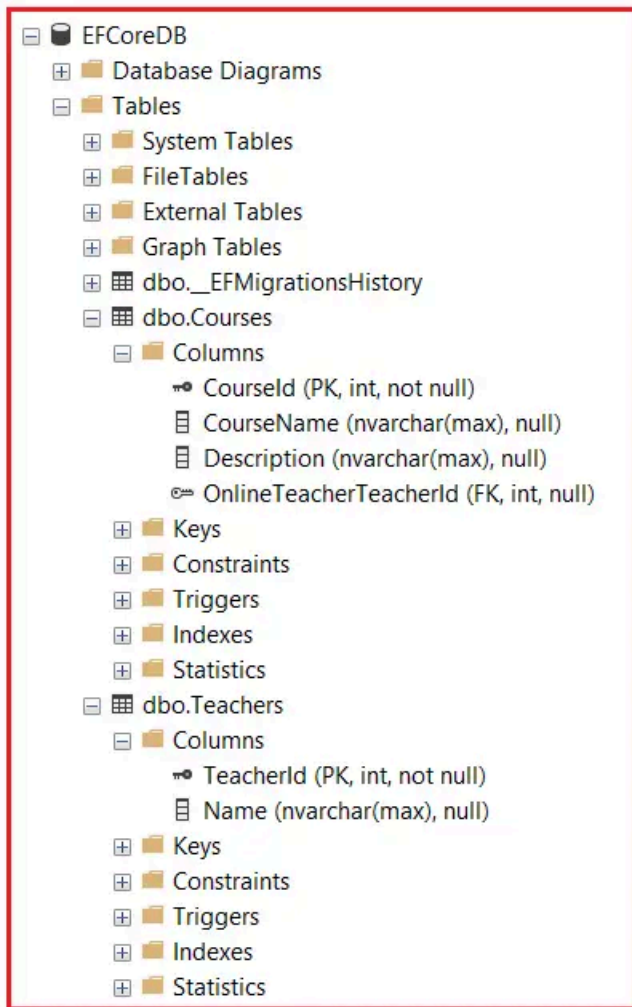
### Generating Migration and Updating Database

As we already discussed, whenever we add or update domain classes or configurations, we need to sync the database with the model using the **Add-Migration** and **Update-Database** commands using the Package Manager Console or .NET Core CLI. So, open the Package Manager Console and Execute the **Add-Migration** and **Update-Database** commands as follows. You can give your migration any name. Here, I am giving it DBMigration1. The name that you are giving it should not be given earlier.



```
Package Manager Console
Package source: All | Default project: EFCoreCodeFirstDemo
PM> add-migration DBMigration1
Build started...
Build succeeded.
An operation was scaffolded that may result in the loss of data. Please review the migration for accuracy.
To undo this action, use Remove-Migration.
PM> update-database
Build started...
Build succeeded.
Applying migration '20230809114727_DBMigration1'.
Done.
```

In our example, the Course and Teacher entities have a one-to-many relationship where one teacher can teach many online courses, and a single teacher can teach one course. As per the default conventions of Entity Framework Core, the above example would create the following tables in the database. Here, you can see OnlineTeacherTeacherId (Navigation Property Name plus TeacherId, OnlineTeacher plus TeacherId, i.e., OnlineTeacherTeacherId ) is the Foreign key created in the Courses table.



#### Creating Multiple Relationships Between Teacher and Course Entities:

Now, suppose our requirement is to add another one-to-many relationship between the Teacher and Course Entities. For example, apart from OnlineTeacher, we now need to add OfflineTeacher. First, modify the Teacher entity as follows to include the OfflineCourses collection Navigation Property.

```
namespace EFCoreCodeFirstDemo.Entities
```

```
public int TeacherId { get; set; }
```

```
public string? Name { get; set; }
```

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

```
public ICollection<Course>? OfflineCourses { get; set; }
```

```
namespace EFCoreCodeFirstDemo.Entities { public class Teacher { public int TeacherId { get; set; } public string? Name { get; set; } public
ICollection<Course>? OnlineCourses { get; set; } public ICollection<Course>? OfflineCourses { get; set; } }
```

```
namespace EFCoreCodeFirstDemo.Entities
```

```
{
```

```
    public class Teacher
```

```

    {
        public int TeacherId { get; set; }
        public string? Name { get; set; }
        public ICollection<Course>? OnlineCourses { get; set; }
        public ICollection<Course>? OfflineCourses { get; set; }
    }
}

```

Next, modify the Course Entity to include the Offline Teacher Reference Navigational property.

```
namespace EFCoreCodeFirstDemo.Entities
```

```
public int CourseId { get; set; }
```

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

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

```
public Teacher? OnlineTeacher { get; set; }
```

```
public Teacher? OfflineTeacher { get; set; }
```

```
namespace EFCoreCodeFirstDemo.Entities { 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? OfflineTeacher { get; set; } } }
```

```
namespace EFCoreCodeFirstDemo.Entities
```

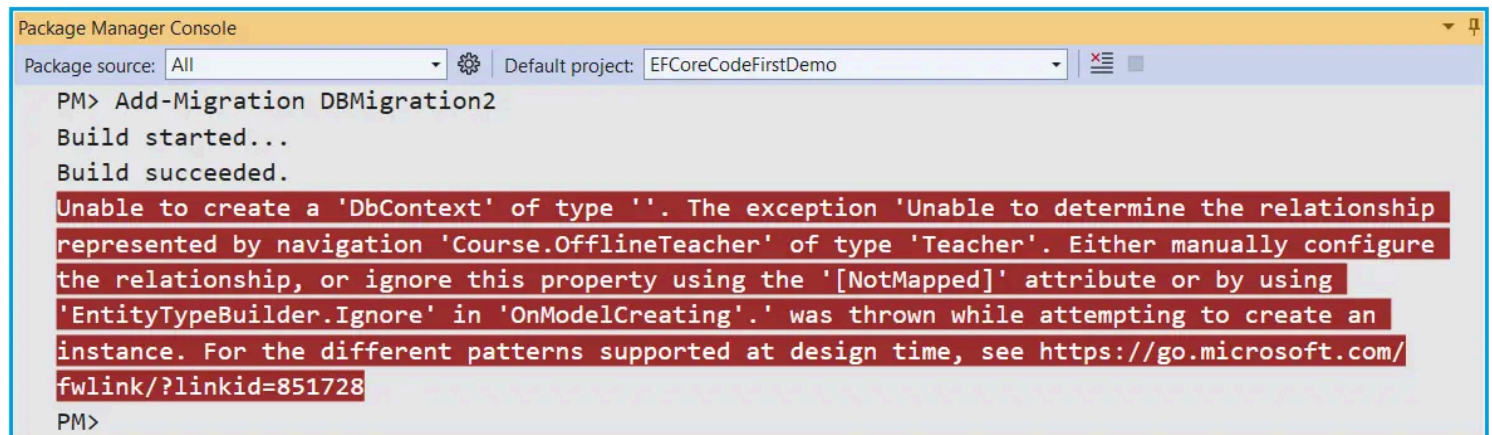
```

{
    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? OfflineTeacher { get; set; }
    }
}

```

Now, the Course and Teacher entities have two one-to-many relationships. An online or offline teacher can teach a course, and a teacher can teach multiple online and offline courses. So, if we have only one relationship between two entities, it works fine. However, when we have more than one relationship between two entities, EF Core throws an exception when executing the **Add-Migration** command.

With the above changes, open the Package Manager Console and Execute the **Add-Migration** command as follows. You should get the following error. It clearly says that it is unable to determine the relationship.



The screenshot shows the Package Manager Console window. At the top, the package source is set to 'All' and the default project is 'EFCoreCodeFirstDemo'. The command 'PM> Add-Migration DBMigration2' has been executed. The output shows 'Build started...' and 'Build succeeded.'. Below this, a red error message is displayed: 'Unable to create a 'DbContext' of type ''. The exception 'Unable to determine the relationship represented by navigation 'Course.OfflineTeacher' of type 'Teacher'. Either manually configure the relationship, or ignore this property using the '[NotMapped]' attribute or by using 'EntityTypeBuilder.Ignore' in 'OnModelCreating'.' was thrown while attempting to create an instance. For the different patterns supported at design time, see https://go.microsoft.com/fwlink/?linkid=851728'. The console prompt 'PM>' is visible at the bottom.

```

Package Manager Console
Package source: All [v] [g] Default project: EFCoreCodeFirstDemo [v] [x] [y]
PM> Add-Migration DBMigration2
Build started...
Build succeeded.
Unable to create a 'DbContext' of type ''. The exception 'Unable to determine the relationship
represented by navigation 'Course.OfflineTeacher' of type 'Teacher'. Either manually configure
the relationship, or ignore this property using the '[NotMapped]' attribute or by using
'EntityTypeBuilder.Ignore' in 'OnModelCreating'.' was thrown while attempting to create an
instance. For the different patterns supported at design time, see https://go.microsoft.com/
/fwlink/?linkid=851728
PM>

```

#### How Can We Overcome This Problem in EF Core?

To overcome this problem, we must use the **InverseProperty** Attribute in EF Core. If you go to the definition of InverseProperty class, you will see the following. As you can see, the InverseProperty class has one constructor, which takes a string property parameter and a read-only property to return the property name.

```
namespace System.ComponentModel.DataAnnotations.Schema
{
    ... public class InversePropertyAttribute : Attribute
    {
        ... public InversePropertyAttribute(string property);

        ... public string Property { get; }
    }
}
```

#### Using InverseProperty Attribute in Entity Framework Core

Let us modify the **Teacher** Entity class as follows to use the InverseProperty Attribute. As you can see in the code below, we have decorated the **InverseProperty** Attribute with both **OnlineCourses** and **OfflineCourses** property and also specified the Course Entity Navigation properties, which should point to make the relationship. With this, **OnlineCourses** will have a relationship with the **OnlineTeacher** property, and **OfflineCourses** will have a relationship with the **OfflineTeacher** property.

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

```
namespace EFCoreCodeFirstDemo.Entities
```

```
public int TeacherId { get; set; }
```

```
public string? Name { get; set; }
```

```
[InverseProperty("OnlineTeacher")]
```

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

```
[InverseProperty("OfflineTeacher")]
```

```
public ICollection<Course>? OfflineCourses { get; set; }
```

```
using System.ComponentModel.DataAnnotations.Schema; namespace EFCoreCodeFirstDemo.Entities { public class Teacher { public int TeacherId { get; set; } public string? Name { get; set; } [InverseProperty("OnlineTeacher")] public ICollection<Course>? OnlineCourses { get; set; } [InverseProperty("OfflineTeacher")] public ICollection<Course>? OfflineCourses { get; set; } }
```

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

```
namespace EFCoreCodeFirstDemo.Entities
```

```
{
    public class Teacher
    {
        public int TeacherId { get; set; }
        public string? Name { get; set; }
        [InverseProperty("OnlineTeacher")]
        public ICollection<Course>? OnlineCourses { get; set; }
        [InverseProperty("OfflineTeacher")]
        public ICollection<Course>? OfflineCourses { get; set; }
    }
}
```

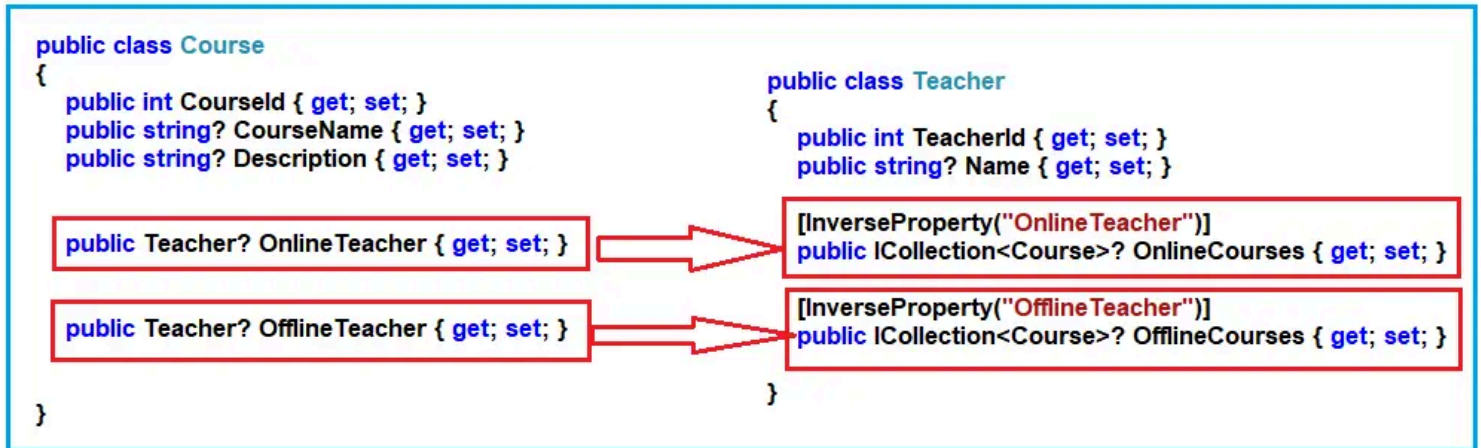
Now, the Course entity has two navigation properties (OnlineTeacher and OfflineTeacher), while the Teacher entity has two collection navigation properties (OnlineCourses and OfflineCourses). To explicitly associate these relationships, we use the [InverseProperty] attribute



on the Teacher entity:

- **[InverseProperty("OnlineTeacher")]** tells EF Core that the OnlineCourses collection is related to the OnlineTeacher property in the Course entity.
- **[InverseProperty("OfflineTeacher")]** tells EF Core that the OfflineCourses collection is related to the OfflineTeacher property in the Course entity.

For a better understanding, please have a look at the below image.



With the above changes, open the Package Manager Console and Execute the **Add-Migration** and **Update-Database** commands as follows. This time, they should be executed as expected.

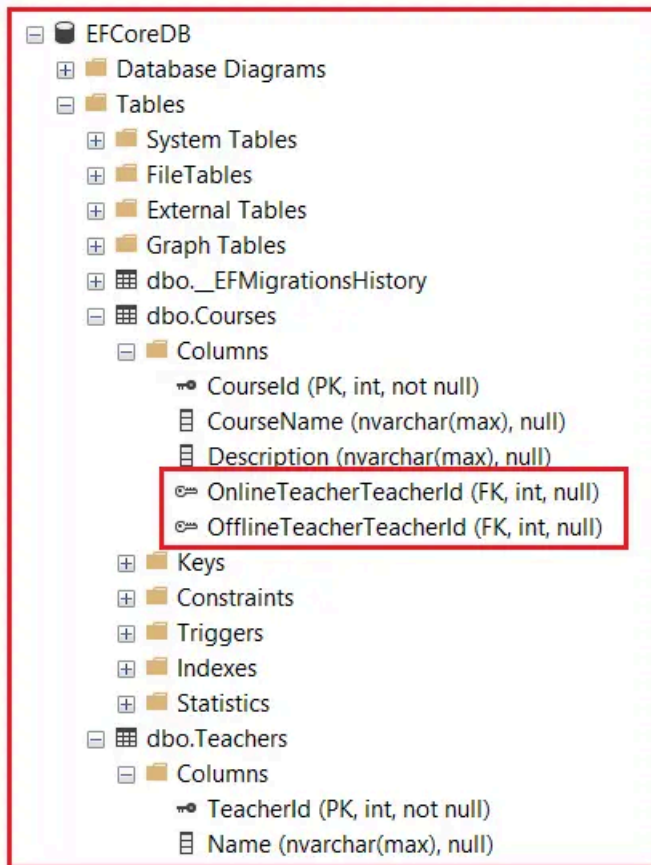
```

Package Manager Console
Package source: All Default project: EFCoreCodeFirstDemo

PM> add-migration DBMigration2
Build started...
Build succeeded.
To undo this action, use Remove-Migration.
PM> update-database
Build started...
Build succeeded.
Applying migration '20230809115945_DBMigration2'.
Done.

```

Now, verify the database, and you should see the following. As you can see, in this case, Entity Framework Core creates foreign keys OnlineTeacherTeacherId and OfflineTeacherTeacherId.



### Course Entity with Custom Foreign Key Names

We can use the [ForeignKey] attribute to specify the custom foreign key names. For a better understanding, please modify the Course entity as follows. Here, the [ForeignKey] attribute is applied to the navigation properties (OnlineTeacher and OfflineTeacher) to specify the custom foreign key column names (OnlineTeacherId and OfflineTeacherId).

using System.ComponentModel.DataAnnotations.Schema;

namespace EFCoreCodeFirstDemo.Entities

public int CourseId { get; set; }

public string? CourseName { get; set; }

public string? Description { get; set; }

// Navigation property for the online teacher

[ForeignKey("OnlineTeacherId")] // Custom Foreign Key for OnlineTeacher

public Teacher? OnlineTeacher { get; set; }

// Navigation property for the offline teacher

[ForeignKey("OfflineTeacherId")] // Custom Foreign Key for OfflineTeacher

public Teacher? OfflineTeacher { get; set; }

// Custom foreign key columns

public int? OnlineTeacherId { get; set; } // Custom FK column for OnlineTeacher

public int? OfflineTeacherId { get; set; } // Custom FK column for OfflineTeacher



```
using System.ComponentModel.DataAnnotations.Schema; namespace EFCoreCodeFirstDemo.Entities { public class Course { public int CourseId { get; set; } public string? CourseName { get; set; } public string? Description { get; set; } // Navigation property for the online teacher [ForeignKey("OnlineTeacherId")] // Custom Foreign Key for OnlineTeacher public Teacher? OnlineTeacher { get; set; } // Navigation property for the offline teacher [ForeignKey("OfflineTeacherId")] // Custom Foreign Key for OfflineTeacher public Teacher? OfflineTeacher { get; set; } // Custom foreign key columns public int? OnlineTeacherId { get; set; } // Custom FK column for OnlineTeacher public int? OfflineTeacherId { get; set; } // Custom FK column for OfflineTeacher }}
```

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

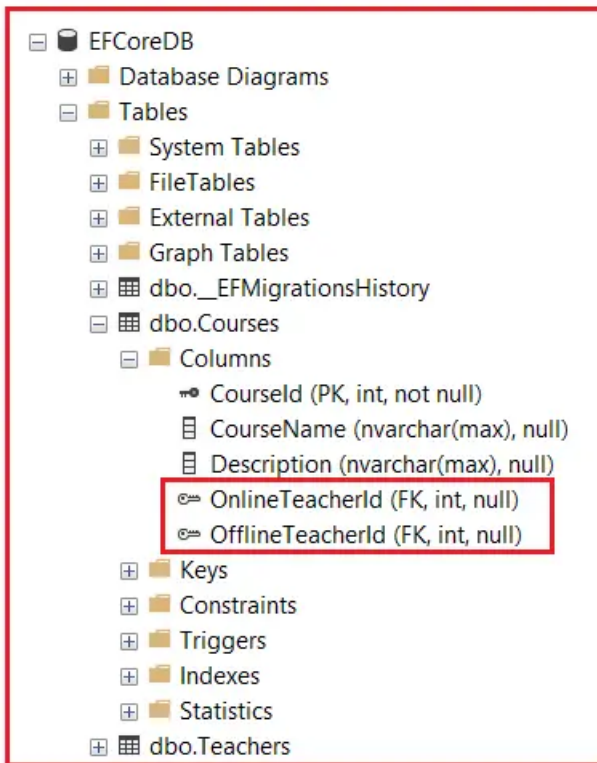
```
namespace EFCoreCodeFirstDemo.Entities
{
    public class Course
    {
        public int CourseId { get; set; }
        public string? CourseName { get; set; }
        public string? Description { get; set; }

        // Navigation property for the online teacher
        [ForeignKey("OnlineTeacherId")] // Custom Foreign Key for OnlineTeacher
        public Teacher? OnlineTeacher { get; set; }

        // Navigation property for the offline teacher
        [ForeignKey("OfflineTeacherId")] // Custom Foreign Key for OfflineTeacher
        public Teacher? OfflineTeacher { get; set; }

        // Custom foreign key columns
        public int? OnlineTeacherId { get; set; } // Custom FK column for OnlineTeacher
        public int? OfflineTeacherId { get; set; } // Custom FK column for OfflineTeacher
    }
}
```

In the Course class, the [ForeignKey("OnlineTeacherId")] and [ForeignKey("OfflineTeacherId")] attributes are used to specify custom column names for the foreign key properties (OnlineTeacherId and OfflineTeacherId). With the above changes, open the Package Manager Console, Execute the **Add-Migration** and **Update-Database** commands, and verify the database. You should see the Custom Foreign Key names as follows. The Courses table will have two custom foreign key columns: OnlineTeacherId and OfflineTeacherId.



#### Why Use the InverseProperty Attribute in Entity Framework Core?

By default, EF Core can only infer one relationship between two entities. When multiple relationships exist, EF Core cannot automatically determine how to map navigation properties, which leads to errors. The InverseProperty attribute resolves these issues by explicitly defining how navigation properties on both sides of the relationship are connected.

In the next article, I will discuss [NotMapped Attribute in Entity Framework Core](#) with Examples. Here, in this article, I try to explain the InverseProperty Data Annotation Attribute in Entity Framework Core with Examples. I hope you enjoyed this InverseProperty Attribute in EF Core with Examples article.



#### About the Author: Pranaya Rout

Pranaya Rout has published more than 3,000 articles in his 11-year career. Pranaya Rout has very good experience with Microsoft Technologies, Including C#, VB, ASP.NET MVC, ASP.NET Web API, EF, EF Core, ADO.NET, LINQ, SQL Server, MYSQL, Oracle, ASP.NET Core, Cloud Computing, Microservices, Design Patterns and still learning new technologies.

## Registration Open For New ASP.NET Core Online Training

Enhance Your Professional Journey with Our Upcoming Live Session starting on today, from 8.30 PM to 10 PM, IST. For complete information on Registration, Course Details, Syllabus, and Demo Session Zoom Credentials, please click on the below link.

- [ASP.NET Core Online Training with Real-time Application Development](#)

[Previous Chapter](#)

[Next Chapter](#)