Navigation Property With Code First (Navigation Property In EF)

What is the Navigation Property?

It is a way to represent a foreign key relationship in the database or define the relationship between the two entities.

**Example**

Client - Client ID, Name.

Project - Project ID, Client ID.

It means the project is linked with the client on the basis of the Client ID.

Relationship: Project HAS-a client.

If we model this relationship, it would look like as shown below:

1. class Client
2. {
3. **public** **string** Name {**get**; **set**}
4. }
6. class Project
7. {
8. **public** Client objClient {**get**; **set**}
9. }

It means we can do this in the following manner:

Project.objClient.Name;

Now, if we want to achieve this structure in the database, we need to define our model in the way shown below:

1. class Client
2. {

**public**

 int Id {**get**; **set** ;}

1. **public** **string** {**get**; **set** ;}
2. }
4. class Project
5. {
6. public **int** Id {**get**; **set** ;}
7. public Client client {**get**; **set** ;} // Project HAS-a client. Name of Field will be
8. // [PropertyName]\_[PropertyIDName] “client\_Id”
9. }

**Note**

Association between two entities defines three things: type of entity, type of association, and referential integrity.

**How Navigation property works**

When we apply navigation property in our code, it means we are asking EF to automatically perform a join between the two tables.

**Example**

Code accesses project info of the client name ASD, as shown below:

1. Context.Projects.Where(p ->p.client.Name == ‘ASD’);

Now, SQL translation of the code, given above is as follows

1. **Select** Projects .Id, Projects .client\_Id
2. **From** Projects **inner** join Clients
3. **On** Projects.client\_Id = Clients.Id
4. **Where** Clients.**Name** = ‘ASD’

**How and when navigation properties load**

By default, navigation properties are null, they are not loaded by default.  For loading navigation property, we use “include” method of IQuearable and this type of loading is called Eager loading.

Eager loading: It is a process by which a query for one type of entity loads the related entities as a part of query and it is achieved by “include” method of IQuerable.

**How Navigation properties loading works,**

1. class Client
2. {
3. **public** **int** ID {get; **set** ;}
4. **public** string **Name** {get; **set** ;}
5. }
7. Class Project
8. {
9. **public** **int** ID {get; **set** ;}
10. **public** Client Clients {get; **set** ;} // Project HAS-a client. **Name** **of** Field will be
11. [PropertyName]\_[PropertyIDName] “Clients\_ID”
12. }

**Clients**

|  |  |
| --- | --- |
| ID | Name |
| 1 | CMD |
| 2 | NSH |

**Projects**

|  |  |
| --- | --- |
| ID | Clients\_ID |
| 100 | 1 |
| 101 | 1 |

1. Get all the projects but do not get the linked clients.
   1. Context.Projects.ToArray()
   2. <ArrayOfProject>
   3. <Project>
   4. <ID>100</ID>
   5. <Clients>null</Clients>
   6. </Project>
   7. <Project>
   8. <ID>101</ID>
   9. <Clients>null</Clients>
   10. </Project>
   11. </ArrayOfProjects>
2. Filter, using where clause, which works perfectly and filters the records, but the linked property clients are not loaded.
   1. Context.Project.Where(p =>p.Clients.Name==”CMD”) .ToArray()
   3. <ArrayOfProject>
   4. <Project>
   5. <ID>100</ID>
   6. <Clients>**null**</Clients>
   7. </Project>
   8. </ArrayOfProjects>
3. Include statement is able to fetch or load the linked item of an entity.
   1. Context.Project.include(p =>p.Clients).ToArray();
   3. <ArrayOfProject>
   4. <Project>
   5. <ID>100 </ID>
   6. <Client>
   7. <ID>1</ID>
   8. <Name> CMD</Name>
   9. </Client>
   10. </Project>
   11. <Project>
   12. <ID>101 </ID>
   13. <Client>
   14. <ID>1</ID>
   15. <Name> CMD</Name>
   16. </Client>
   17. </Project>
   18. <ArrayOfProject>

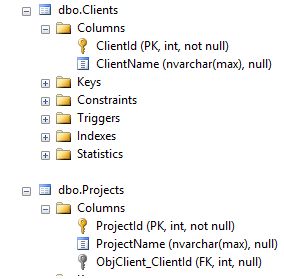
**How Entity Framework detects Navigation properties**

1. **Complex type**

When we define complex type inside class, which is attached with the context for Entity framework, consider this complex type as a foreign key inside this class. Hence, it creates Foreign key by name [PropertyName]\_[PropertyIDName] (Name by which property is defined and the name of Primary key is of complex type.)

**Example**

* 1. class Client
  2. {
  3. **public** **int** ClientId {**get**; **set**;}
  4. **public** **string** ClientName {**get**; **set**;}
  6. }
  8. Class Project
  9. {
  10. **public** **int** ProjectId {**get**; **set**;}
  11. **public** **string** ProjectName {**get**; **set**;}
  13. **public** Client ObjClient {**get**; **set**;}
  14. }



In the above example, we have two classes: Client and Project.

Now, the client is defined inside the Project class. Hence, these two classes are attached to the context by dbset. Thus, the entity frame considers it a Foreign key.

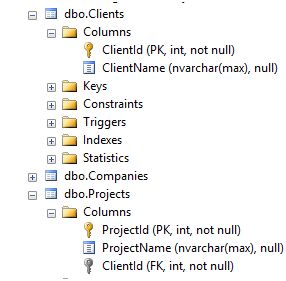
It will create a Foreign key inside the Project table by the following naming convention [PropertyName]\_[PropertyIDName], which means, it will look like “ObjClient\_ClientId”

1. **By Naming conventions**

When we define the complex type property inside a class and also add the property with the same name as a Primary key of complex type, the Entity Framework considers this and defines property as a Foreign key of the table with the same name.

**Example**

* 1. class Client
  2. {
  3. **public** **int** ClientId {**get**; **set**;}
  4. **public** **string** ClientName {**get**; **set**;}
  6. }
  8. class Project
  9. {
  10. **public** **int** ProjectId {**get**; **set**;}
  11. **public** **string** ProjectName {**get**; **set**;}
  13. **public** **int** ClientId {**get**; **set**;} // Entity framework treat it as foreign key, and create same
  14. // name property as Foreign key into table.
  15. **public** Client ObjClient {**get**; **set**;}
  16. }



In the above example in the Project table, the Entity framework creates a Foreign key by the name “ClientId”.

**Ways to define Foreign key in Entity Framework**

1. By adding complex type as a property,
   1. class Client
   2. {
   3. public intClientId {**get**; **set**;}
   4. public **string** ClientName {**get**; **set**;}
   6. }
   8. class Project
   9. {
   10. public intProjectId {**get**; **set**;}
   11. public **string** ProjectName {**get**; **set**;}
   13. public Client ObjClient {**get**; **set**;}
   14. }

Only add a property of complex type, and it will automatically create an “ObjClient\_ClientId” foreign key into project table.

1. By adding complex type as property and adding the property by the same name and typing it as the Primary key of a complex type,
   1. class Client
   2. {
   3. public **int** ClientId {**get**; **set**;}
   4. public **string** ClientName {**get**; **set**;}
   6. }
   8. class Project
   9. {
   10. public **int** ProjectId {**get**; **set**;}
   11. public **string** ProjectName {**get**; **set**;}
   13. public **int** ClientId {**get**; **set**;} // Entity framework treat it as foreign key, and create same
   14. // name property as Foreign key into table.
   15. public Client ObjClient {**get**; **set**;}
   16. }
2. By using data annotation Foreign key attribute,

Data annotation Foreign key attribute can be used in two ways -- one with the navigation property defined inside the class or with a Foreign key, defined inside the class.

**Example**

Foreign key attribute is used over a navigation property.

* 1. class Project
  2. {
  3. **public** **int** ProjectId {**get**;**set**;}
  4. **public** **string** ProjectName{**get**; **set**;}
  6. [ForeignKey("FkClientId")]
  7. **public** Client client {**get**; **set**;} // Navigation property
  8. }



Hence, at this stage, it will give an error, because here we have specified a Foreign key. Hence, EF will not create a default foreign key by [PropertyName]\_[PropertyIdName]. Hence, EF tries to find a Foreign key, specified by the name “FkClientId”. It will give an error for not finding Foreign key by the FkClientId inside the Project class. Hence, we need to define Foreign key property.

* 1. class Project
  2. {
  3. **public** **int** ProjectId {**get**;**set**;}
  4. **public** **string** ProjectName{**get**; **set**;}
  5. **public** **int** FkClientId {**get**; **set**;} // It will become foreign key.
  6. [ForeignKey("FkClientId")]
  7. **public** Client client {**get**; **set**;} // Navigation property
  8. }

A close up of a text

Description automatically generated

**Example**

Foreign key attribute is used over the property with which you want to make Foreign key.

* 1. class Project
  2. {
  3. **public** **int** ProjectId {**get**;**set**;}
  4. **public** **string** ProjectName{**get**; **set**;}
  5. [ForeignKey ("Client")]
  6. **public** **int** FkClient {**get**; **set**;}
  7. }



At this stage, the code given above will give an error, because in the code given above, we are saying that, FkClient is a Foreign key of the Client table, but EF will not create this foreign key due to the absence of the navigation property of Client, it means at least one navigation property of client should be there adding its key as a foreign key into the project.

* 1. class Project
  2. {
  3. **public** **int** ProjectId {**get**;**set**;}
  4. **public** **string** ProjectName{**get**; **set**;}
  5. [ForeignKey ("Client")]
  6. **public** **int** FkClient {**get**; **set**;}
  7. **public** Client Client{**get**;**set**;} // Navigation Property.
  8. }

1. By using data annotation Foreign key attributes, we can give different names to Foreign key property.

**Note**

The first part of the expression defines the navigation property on the current entity, the second part of the expression defines the reverse navigation property.

**What is Reverse Navigation property?**

Here, we will discuss the reverse navigation property.

Two navigation properties that we have defined on each end of a relationship are in fact different ends of the same relationship.

It has been able to do this because there has been only one possible match.

Type of Navigation properties,

1. Optional relationship (Nullable foreign key and multiplicity, 0...1)(One to one relationship )(Zero or one).
2. Required relationship (One to many relationships).
3. Many relationships (Many to Many relationships).

**One-to-One Relationship**

Optional relationship (Nullable foreign key and multiplicity, 0...1)(One to one relationship )(Zero or one).

Primary key value table contains one record and its related table contains one or zero records related to this table. This relationship is called a one to one relationship.

Suppose, take an example of “Client” and “ClientAddress”, which has a one to one relationship (zero or one),this means the client can have one or zero addresses.

Hence, if we create a class of the client having ID and Name; create Class ClientAddress contains ID , Address and City.

For setting up one to one relationships between the Client and ClientAddress, we need to define the navigation property of the client into ClientAddress and specify a Foreign Key of the client into ClientAddress and then define the reverse navigation property of ClientAddress into the client. In this way, EF handles One One relationships.

1. class Client
2. {
3. **public** **int** ClientID{**get**;**set**;};
4. **public** String Name{**get**;**set**;};
5. }
6. clientAddress
7. {
9. **public** **int** ClientAddressId {**get**;**set**;}
10. **public** String Address {**get**;**set**;}
11. **public** **string** City {**get**;**set**;}
12. [ForeignKey ("Client")]
13. **public** **int** FkClient {**get**; **set**;}
14. **public** Client Client{**get**;**set**;} // Navigation Property.
15. }

**One-to-many relationships**

Primary key value table contains one record and its related table contains one, zero, or many records related to it. This relationship is called a One-to-Many relationship.

**Example**

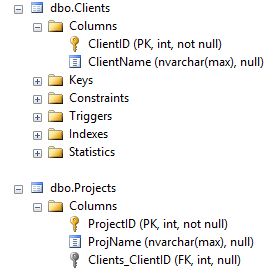
Suppose we have the “Client” and “Project” entity.

One client can have multiple projects, but one project can only link with one project.

**Note**

Reverse navigation property of the expression is defined as a collection. In this case, there is a one-to-many relationship.

1. class Client
2. {
3. **public** **int** ClientID{**get**;**set**;}
4. **public** String ClientName{**get**;**set**;}
5. // Second Part of expression define reverse navigation property
6. **public** **virtual** ICollection<Project> Projects { **get**; **set**; }
7. }
9. class Project
10. {
11. **public** **int** ProjectID{**get**;**set**;}
12. **public** String ProjName {**get**;**set**;}
13. // First Part of expression define navigation property
14. **public** Client Clients {**get**;**set**}
15. }



Hence, in the code given above, you can see the client type contains a collection of Project types. Project type contains a client-type entity.

These are the two ends of the expression of the navigation properties and it shows that the client can have multiple projects due to which, the project is defined in the collection inside the client.

On the other hand, the project only links with the single client, due to which it contains the client type entity.

**Many-to-Many relationship**

One table records are able to relate with any numbers (or zero records) of records of the second, the second table records are able to relate with any number of records of the first table.

Many-to-many relationships require a third table, which is called linking or mapping the table.

Example: the relationship between “Student” and “Course” entity,

One student can be enrolled for multiple courses and one course can be taught to many students.

Hence, in this case, a third table is required, which will contain the Pk of both entities.

1. class Student
2. {
3. **public** **int** StudentId {**get**;**set**;}
4. **public** String Name {**get**;**set**;}
5. **public** String Address {**get**; **set**; }
6. **public** **virtual** ICollection<Course > Courses { **get**; **set**; }
8. }
10. class Course
11. {
12. **public** **int** CourseId {**get**; **set**;}
13. **public** String CourseName {**get**;**set**;}
14. **public** **virtual** ICollection<Student>Students { **get**; **set**; }
15. }

