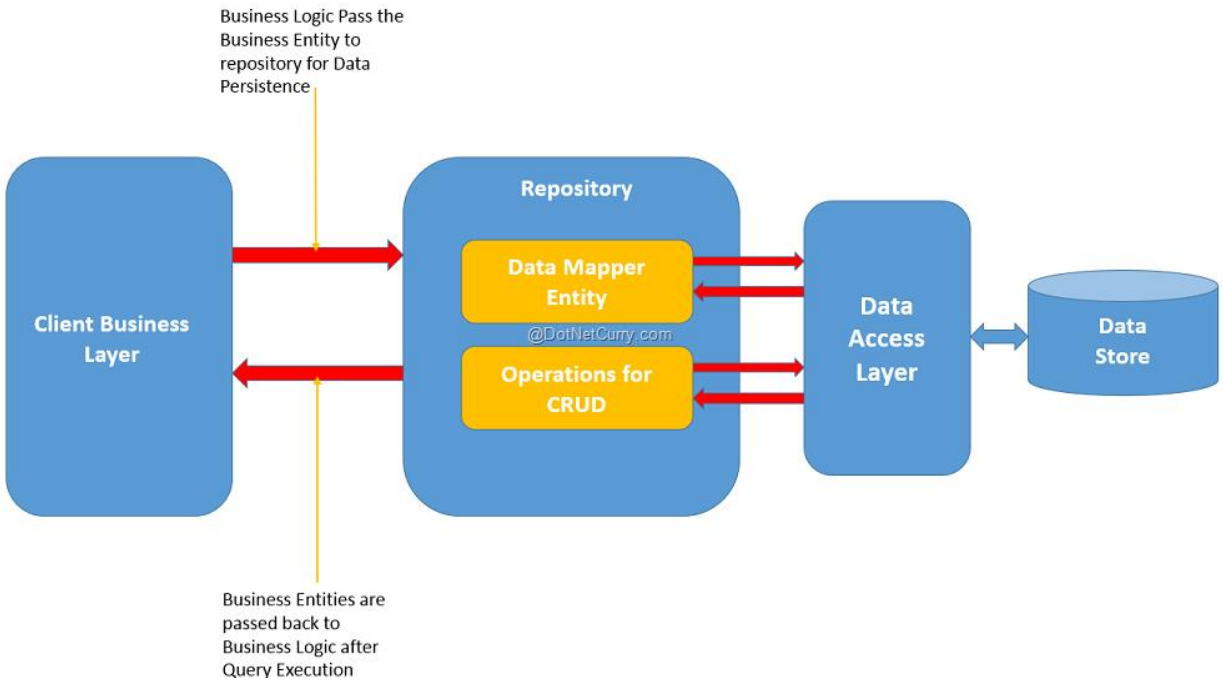
Repository pattern in c#. It is about **how an application stores and retrieves data from the data stores which actually stores the application data**. In a single line we can say "Repository pattern allows development of applications which are decoupled from the underlying data storage mechanisms"


Business Logic Pass the Business Entity to repository for Data Persistence
Client Business Layer
Repository
Data Mapper Entity
@DotNetCurry.com
Operations for CRUD
Data Access Layer
Data Store
Business Entities are passed back to Business Logic after Query Execution

Example:

What is 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 Client ID.

Relationship: Project HAS-a client.

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

```
class Client
{
   public string Name {get; set}
}

class Project
{
   public Client objClient {get; set}
}
```

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:

```
class Client
{

public
int Id {get; set ;}
    public string {get; set ;}
}

class Project
{
    public int Id {get; set ;}
    public Client client {get; set ;} // Project HAS-a client. Name of Field will be
    // [PropertyName]_[PropertyIDName] "client_Id"
}
```

<u>Note</u>

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:

```
Context.Projects.Where(p ->p.client.Name == 'ASD');
```

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

```
Select Projects .Id, Projects .client_Id
From Projects inner join Clients
On Projects.client_Id = Clients.Id
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 IQueryable 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 IQueryable.

How Navigation properties loading works,

```
class Client
{
    public int ID {get; set ;}
    public string Name {get; set ;}
}

Class Project
{
    public int ID {get; set ;}
    public Client Clients {get; set ;} // Project HAS-a client. Name of Field will be
    [PropertyName]_[PropertyIDName] “Clients_ID”
}
```

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.

```
Context.Projects.ToArray()


<ArrayOfProject>
    <Project>
        <ID>100</ID>
        <Clients>null</Clients>
    </Project>
    <Project>
        <ID>101</ID>
        <Clients>null</Clients>
    </Project>
</ArrayOfProjects>
```

2. Filter, using where clause, which works perfectly and filters the records, but the linked property clients are not loaded.

```
Context.Project.Where(p =>p.Clients.Name==”CMD”) .ToArray()
```

```

<ArrayOfProject>
    <Project>
        <ID>100</ID>
        <Clients>null</Clients>
    </Project>
</ArrayOfProjects>
```

3. Include statement is able to fetch or load the linked item of an entity.

```
Context.Project.include(p =>p.Clients).ToArray();

<ArrayOfProject>
    <Project>
        <ID>100 </ID>
        <Client>
            <ID>1</ID>
            <Name> CMD</Name>
        </Client>
    </Project>
    <Project>
        <ID>101 </ID>
        <Client>
            <ID>1</ID>
            <Name> CMD</Name>
        </Client>
    </Project>
<ArrayOfProject>
```

How Entity Framework detect Navigation properties

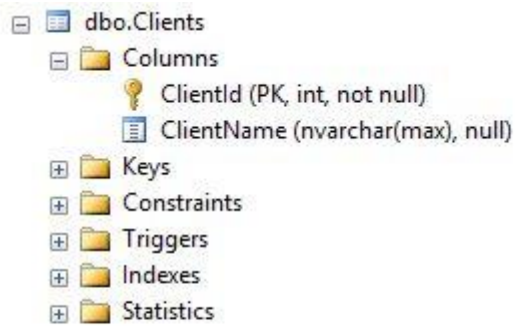
1. **Complex type**

   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.)

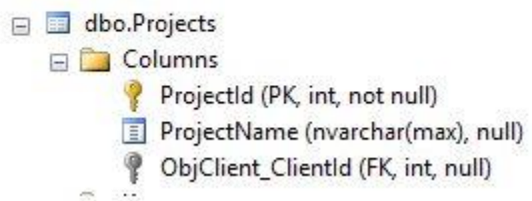
   Example

```
class Client
{
   public int ClientId {get; set;}
   public string ClientName {get; set;}

}

Class Project
{
   public int ProjectId {get; set;}
```

```
    public string ProjectName {get; set;}

    public Client ObjClient {get; set;}
}
```



dbo.Clients
Columns
ClientId (PK, int, not null)
ClientName (nvarchar(max), null)
Keys
Constraints
Triggers
Indexes
Statistics



dbo.Projects
Columns
ProjectId (PK, int, not null)
ProjectName (nvarchar(max), null)
ObjClient_ClientId (FK, int, null)

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 with the context by dbset.Thus, entity frame considers it as a Foreign key.

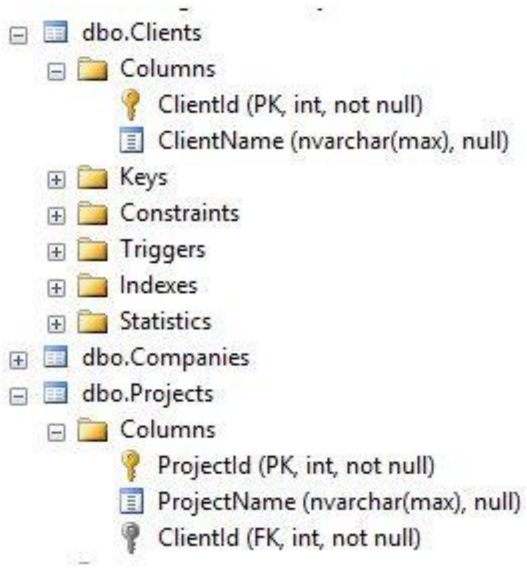
It will create a Foreign key inside the Project table by the following naming convention [PropertyName]_[PropertyIDName], which means, it will look like "ObjClient_ClientId"

2. **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 a property as a Foreign key of the table with the same name.

Example

```
class Client
{
    public int ClientId {get; set;}
    public string ClientName {get; set;}

}

class Project
{
    public int ProjectId {get; set;}
    public string ProjectName {get; set;}

    public int ClientId {get; set;} // Entity framework treat it as foreign key, and create same
    // name property as Foreign key into table.
    public Client ObjClient {get; set;}
}
```



dbo.Clients
Columns
ClientId (PK, int, not null)
ClientName (nvarchar(max), null)
Keys
Constraints
Triggers
Indexes
Statistics
dbo.Companies
dbo.Projects
Columns
ProjectId (PK, int, not null)
ProjectName (nvarchar(max), null)
ClientId (FK, int, not null)

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

Ways to define Foreign key in Entity Framework

1. By adding complex type as a property,

```
class Client
{
    public intClientId {get; set;}
    public string ClientName {get; set;}

}

class Project
{
    public intProjectId {get; set;}
    public string ProjectName {get; set;}

    public Client ObjClient {get; set;}
}
```

Only add property of complex type, and it will automatically create “ObjClient_ClientId” foreign key into project table.

2. By adding complex type as a property and adding the property by the same name and typing it as Primary key of a complex type,

```
class Client
{
   public int ClientId {get; set;}
   public string ClientName {get; set;}

}

class Project
{
   public int ProjectId {get; set;}
   public string ProjectName {get; set;}

   public int ClientId {get; set;} // Entity framework treat it as foreign
 key, and create same
   // name property as Foreign key into table.
   public Client ObjClient {get; set;}
}
```

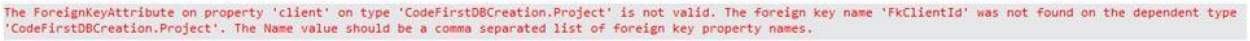
3. By using data annotation Foreign key attribute,

   Data annotation Foreign key attribute can be used in the 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.

```
class Project
{
   public int ProjectId {get;set;}
   public string ProjectName{get; set;}

   [ForeignKey("FkClientId")]
   public Client client {get; set;} // Navigation property
}
```

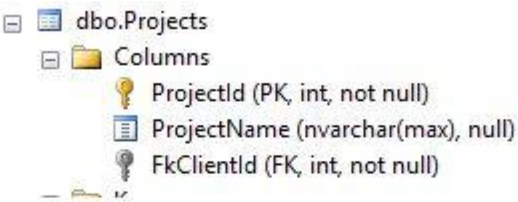


```
The ForeignKeyAttribute on property 'client' on type 'CodeFirstDBCreation.Project' is not valid. The foreign key name 'FkClientId' was not found on the dependent type 'CodeFirstDBCreation.Project'. The Name value should be a comma separated list of foreign key property names.
```

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 Foreign key, specified by 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.

```
class Project
{
   public int ProjectId {get;set;}
   public string ProjectName{get; set;}
   public int FkClientId {get; set;} // It will become foreign key.
   [ForeignKey("FkClientId")]
```

```
    public Client client {get; set;} // Navigation property
}
```

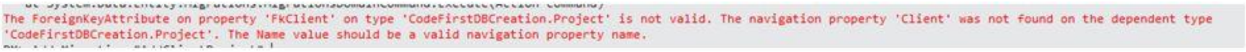


dbo.Projects
Columns
ProjectId (PK, int, not null)
ProjectName (nvarchar(max), null)
FkClientId (FK, int, not null)

Example

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

```
class Project
{
    public int ProjectId {get;set;}
    public string ProjectName{get; set;}
    [ForeignKey ("Client")]
    public int FkClient {get; set;}
}
```



```
The ForeignKeyAttribute on property 'FkClient' on type 'CodeFirstDBCreation.Project' is not valid. The navigation property 'Client' was not found on the dependent type 'CodeFirstDBCreation.Project'. The Name value should be a valid navigation property name.
```

On this stage, the code given above will give an error, because in the code given above, we are saying tha, FkClient is a Foreign key of 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.

```
class Project
{
    public int ProjectId {get;set;}
    public string ProjectName{get; set;}
    [ForeignKey ("Client")]
    public int FkClient {get; set;}
    public Client Client{get;set;} // Navigation Property.
}

```

4. By using data annotation Foreign key attribute, we can give different name to Foreign key property.

<u>Note</u>

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

One-to-One Relationship

Optional relationship (Null able 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 client into ClientAddress and then define the reverse navigation property of ClientAddress into the client. In this way, EF handles One to One relationships.

```
class Client
{
   public int ClientID{get;set;};
   public String Name{get;set;};
}
clientAddress
{

   public int ClientAddressId {get;set;}
   public String Address {get;set;}
   public string City {get;set;}
   [ForeignKey ("Client")]
   public int FkClient {get; set;}
   public Client Client{get;set;} // Navigation Property.
}
```

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

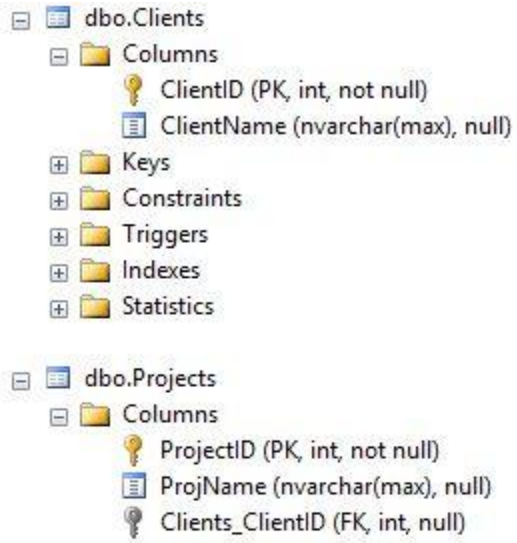
Suppose we have "Client" and "Project" entity.

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.

```
class Client
{
    public int ClientID{get;set;}
    public String ClientName{get;set;}
    // Second Part of expression define reverse navigation property
    public virtual ICollection<Project> Projects { get; set; }
}



class Project
{
    public int ProjectID{get;set;}
    public String ProjName {get;set;}
    // First Part of expression define navigation property
    public Client Clients {get;set}
}
```



```
⊟ dbo.Clients
    ⊟ Columns
        ClientID (PK, int, not null)
        ClientName (nvarchar(max), null)
    ⊞ Keys
    ⊞ Constraints
    ⊞ Triggers
    ⊞ Indexes
    ⊞ Statistics

⊟ dbo.Projects
    ⊟ Columns
        ProjectID (PK, int, not null)
        ProjName (nvarchar(max), null)
        Clients_ClientID (FK, int, null)
```

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

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

On the other hand, 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.

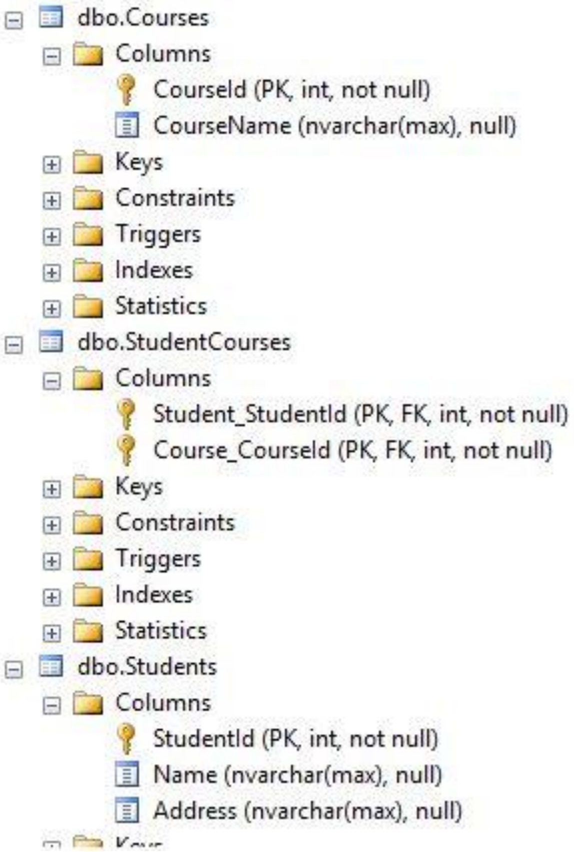
Many-to-many relationship require a third table, which is called linking or mapping the table.

Example: 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 Pk of both entities.

```
class Student
{
   public int StudentId {get;set;}
   public String Name {get;set;}
   public String Address {get; set; }
   public virtual ICollection<Course > Courses { get; set; }

}

class Course
{
   public int CourseId {get; set;}
   public String CourseName {get;set;}
   public virtual ICollection<Student>Students { get; set; }
}
```

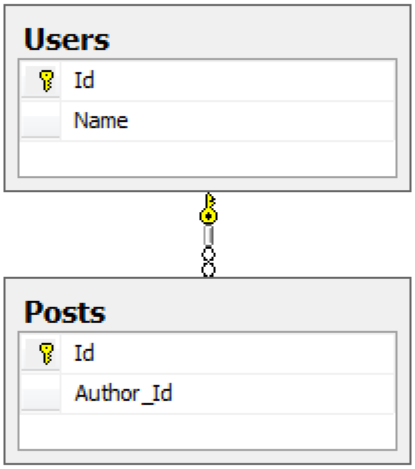


```
⊟ dbo.Courses
    ⊟ Columns
        CourseId (PK, int, not null)
        CourseName (nvarchar(max), null)
    ⊞ Keys
    ⊞ Constraints
    ⊞ Triggers
    ⊞ Indexes
    ⊞ Statistics
⊟ dbo.StudentCourses
    ⊟ Columns
        Student_StudentId (PK, FK, int, not null)
        Course_CourseId (PK, FK, int, not null)
    ⊞ Keys
    ⊞ Constraints
    ⊞ Triggers
    ⊞ Indexes
    ⊞ Statistics
⊟ dbo.Students
    ⊟ Columns
        StudentId (PK, int, not null)
        Name (nvarchar(max), null)
        Address (nvarchar(max), null)
    ⊞ Keys
```

Example: Entity Framework - Navigation Property Basics with Code First

What is a navigation property?

Navigation properties are Entity Frameworks way of representing Foreign Key relationships inside the database. Navigation properties allow you to define relationships between entities (rows in your database) in a way that makes sense in an object oriented language. Consider the following database:



As you can see a post has an author and that is relationaly linked inside our database. So how would we represent this same structure inside of an application (if we ignore the way this is implemented in a relational database)? It seems sensible that we would model this same structure with something like the following:

```
public class User

{

        public string Name { get; set; }

}



public class Post

{

        public User Author { get; set; }

}
```

Which means we can use it in code like this:

```
string.Format("{0} wrote this post", post.Author.Name);
```

What Entity Framework navigation proprieties do is to allow us to do just this with our database models. For example the above database structure could be represented as:

```
public class User

{

        public int Id { get; set; }

        public string Name { get; set; }

}



public class Post

{

        public int Id { get; set; }

        public User Author { get; set; }

}
```

How do Navigation Properties work?

When you are using navigation properties in your code you are asking Entity Framework to automatically perform a SQL join between your two tables. For example:

```
context.Posts.Where(p => p.Author.Name == "Luke");
```

will be translated into the following SQL*:

```
SELECT

         p.Id,

         p.Author_Id

```

```
   FROM  Posts AS p

         INNER JOIN Users AS u ON p.Author_Id = u.Id

   WHERE u.Name = N'Luke'
```

As you can see our relationship in c# has been converted into the equivalent SQL join.

How do I load my properties (Why is my navigation property null)?

Navigation properties are not loaded by default, so its important to know how and when you need to load navigation properties. So lets look at some scenarios and see what entity framework will give us. These scenarios are based off the following data:

User

Id	Name
1	Luke
2	Bob

Post

Id	Author_Id
1	1
2	2

Case 1 :

```
context.Posts.ToArray();
```

When we make this request we get the following:

```
{
        "posts": [
                {
                        "id": 1,
                        "author": null
                },
                {
                        "id": 2,
                        "author": null
                }
        ]
}
```

As you can see we have retrieved all the posts but haven’t received any of the linked authors.

Case 2:

```
context.Posts.Where(p => p.Author.Name == "Luke").ToArray();
```

When we make this request we get the following:

```
{
        "posts": [
                {
                        "id": 1,
                        "author": null
```

```
                 }
         ]
}
```

In this case the `where` clause successfully interacts with the author to filter by name, however when the results are returned the author is still not linked.

Case 3:

```
using System.Data.Entity; //this is required for .Include
...
context.Posts.Include(p => p.Author).ToArray();
```

When we make this request we get the following:

```
{
        "posts": [
                 {
                         "id": 1,
                         "author": {
                                  "id":1,
                                  "name": "Luke"
                         }
                 },
                 {
                         "id": 2,
{                         "author": {
                                  "id": 2,
                                  "name": "Bob"
                          }
                  }
          ]
}
```

As you can see by adding a `.Include` statement we are able to fetch the linked authors.

A note on lazy loading

Lazy loading is also an alternative to using the .Include syntax, however I recommend against using it in almost all cases as it makes it very easy to introduce subtle performance issues into your application. in addition using the include syntax makes it much more obvious what queries your code performs so increases readability.

How does Entity Framework detect Navigation Properties

When entity framework examines a class which is attached to the context it finds other complex type properties on the class and assumes that they are a foreign key to that table. Entity framework then creates a foreign key with the name `[PropertyName]_[PropertyIdName]` for example in the case of the post class the `Author` is `[PropertyName]` and in the user table `Id` is the Id. This means `Author_Id` is generated as the foreign key name.

The rules around navigation properties what's acceptable and how they are generated are defined by a set of conventions. I'm not going to go into any more detail on how these work but for more information you can take a look at the following pages:

- EF Feature CTP5: Pluggable Conventions
- MSDN - System.Data.Entity.ModelConfiguration.Conventions Namespace

- Entity Framework Navigation Property generation rules

More than the defaults with the Model Builder

Entity framework provides a mechanism to configure additional information about navigation properties. This additional configuration can be done using the Model Builder. The model builder allows control over how Entity Framework represents the database, one of the features of the model builder allows for control over how foreign keys in the database are translated to Navigation Properties in the Entity Framework Model.

To get access to the model builder you will need to override the `OnModelCreating` method on your `DbContext`. You can then use the `modelBuilder` argument to structure your Entity Framework Model.

```
public class MyContext : DbContext

{

        protected override void OnModelCreating(DbModelBuilder modelBuilder)

        {

        }

}
```

The Model Builder uses a two part fluent expression to define navigation properties within the model. 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. Navigation properties properties can be either **Optional** (ie *0..1 to x*), **Required** (ie *1 to x*) or **Many** (ie ** to x*). So lets take a look at some examples:

Optional relationship

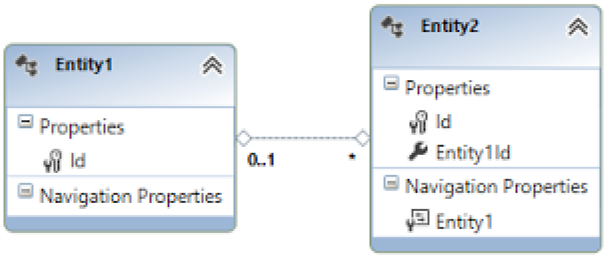
If you have the following classes:

```
public class Entity1

{
    public int Id { get; set; }

}



public class Entity2

{

    public int Id { get; set; }

    public Entity1 Entity1 { get; set; }

}
```

Using the following model builder statement

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

    modelBuilder.Entity<Entity2>().HasOptional(e => e.Entity1).WithMany();

}
```

Will create the following relationship



Note the two parts of the query the first to define `Entity1` navigation property, the first part defines an optional relationship (ie a nullable foreign key) and the second part `.WithMany()` defines the remote entity's (`Entity1`) multiplicity. Using `.WithMany()` with no argument tells Entity Framework that the relationship does not have a remote navigation property.

Required Relationship

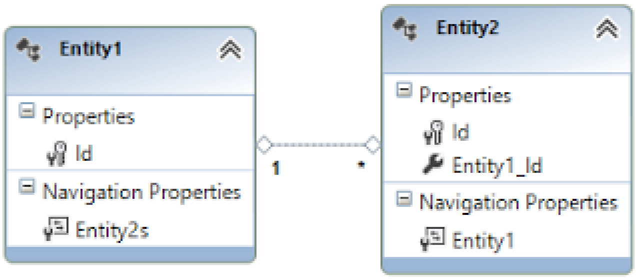
If you have the following classes:

```
public class Entity1
{
    public int Id { get; set; }
    public List<Entity2> Entity2s { get; set; }
}


public class Entity2
{
    public int Id { get; set; }
    public Entity1 Entity1 { get; set; }
}
```

Using the following model builder statement

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
        modelBuilder.Entity<Entity2>()
                 .HasRequired(e => e.Entity1)
                 .WithMany(e => e.Entity2s);
}
```

Will create the following relationship



Note that in this example we have specified a remote collection for the relationship. This means that you can use the `Entity2s` property to find all linked entities.

Many Relationship

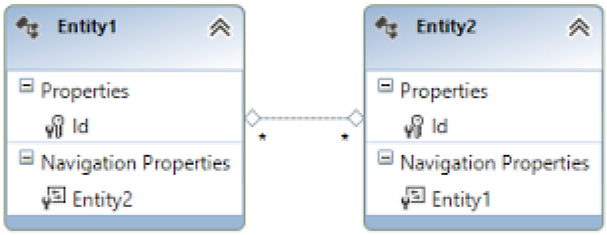
If you have the following classes:

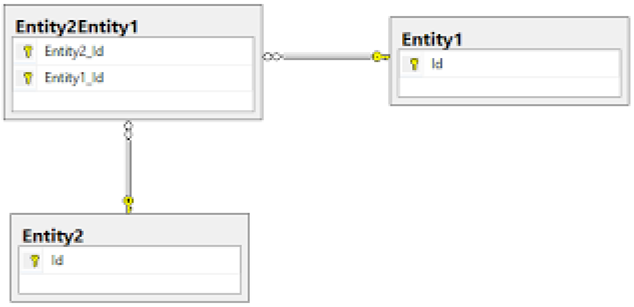
```
public class Entity1

{

        public int Id { get; set; }

        public List<Entity2> Entity2s { get; set; }

}



public class Entity2

{

        public int Id { get; set; }

        public List<Entity1> Entity1s { get; set; }

}
```

Using the following model builder statement

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
        modelBuilder.Entity<Entity2>()
                .HasMany(e => e.Entity1s)
                .WithMany(e => e.Entity2s);
}
```

Will produce the following relationship





With the following database tables

Note that here we have chained two `.Many` statements to produce a many to many relationship. As you can see the Entity Framework has created a link table in the database but the actual link in the model is created as two collections (without a link table) which makes more sense in c#.

Wrapping up

In this post we have looked in detail at navigation properties, what they are, how they are detected and how they are configured. As with many ORMs Entity Framework provides an abstraction over database Foreign Keys. As with any abstraction its really important to understand its limitations. I hope this post helps you better understand how Navigation Properties work and enables you to make good decisions when dealing with Entity Framework.

--text

Entity Framework - Navigation Property Basics with Code First

What is a navigation property?

Navigation properties are Entity Frameworks way of representing Foreign Key relationships inside the database. Navigation properties allow you to define relationships between entities (rows in your database) in a way that makes sense in an object oriented language. Consider the following database:

As you can see a post has an author and that is relationaly linked inside our database. So how would we represent this same structure inside of an application (if we ignore the way this is implemented in a relational database)? It seems sensible that we would model this same structure with something like the following:

```
public class User

{

        public string Name { get; set; }

}


public class Post

{

        public User Author { get; set; }

}
```

Which means we can use it in code like this:

```
String.Format("{0} wrote this post", post.Author.Name);
```

What Entity Framework navigation proprieties do is to allow us to do just this with our database models. For example the above database structure could be represented as:

```
public class User

{

        public int Id { get; set; }

        public string Name { get; set; }

}


public class Post
```

```
{
            public int Id { get; set; }
            public User Author { get; set; }
}
```

How do Navigation Properties work?

When you are using navigation properties in your code you are asking Entity Framework to automatically perform a SQL join between your two tables. For example:

```
context.Posts.Where(p => p.Author.Name == "Luke");
```

will be translated into the following SQL*:

```
SELECT
            p.Id,
            p.Author_Id
  FROM  Posts AS p
            INNER JOIN Users AS u ON p.Author_Id = u.Id
  WHERE u.Name = N'Luke'
```

As you can see our relationship in c# has been converted into the equivalent SQL join.

How do I load my properties (Why is my navigation property null)?

Navigation properties are not loaded by default, so its important to know how and when you need to load navigation properties. So lets look at some scenarios and see what entity framework will give us. These scenarios are based off the following data:

User

Id

Name

1

Luke

2

Bob

Post

Id

Author_Id

1

1

2

2

Case 1 :

```
context.Posts.ToArray();
```

When we make this request we get the following:

```
{
        "posts": [
                {
                        "id": 1,
                        "author": null
                },
                {
                        "id": 2,
                        "author": null
                }
        ]
}
```

As you can see we have retrieved all the posts but haven’t received any of the linked authors.

Case 2:

```
context.Posts.Where(p => p.Author.Name == "Luke").ToArray();
```

When we make this request we get the following:

```
{
        "posts": [
                {
                        "id": 1,
                        "author": null
                }
        ]
}
```

In this case the where clause successfully interacts with the author to filter by name, however when the results are returned the author is still not linked.

Case 3:

```
using System.Data.Entity; //this is required for .Include

...

context.Posts.Include(p => p.Author).ToArray();
```

When we make this request we get the following:

```
{
    "posts": [
        {
            "id": 1,
            "author": {
                "id":1,
                "name": "Luke"
            }
        },
        {
            "id": 2,
            "author":  {
                "id":2,
                "name": "Bob"
            }
        }
    ]
}
```

As you can see by adding a .Include statement we are able to fetch the linked authors.

A note on lazy loading

Lazy loading is also an alternative to using the .Include syntax, however I recommend against using it in almost all cases as it makes it very easy to introduce subtle performance issues into your application. in addition using the include syntax makes it much more obvious what queries your code performs so increases readability.

How does Entity Framework detect Navigation Properties

When entity framework examines a class which is attached to the context it finds other complex type properties on the class and assumes that they are a foreign key to that table. Entity framework then creates a foreign key with the name [PropertyName]_[PropertyIdName] for example in the case of the post class the Author is [PropertyName] and in the user table Id is the Id. This means Author_Id is generated as the foreign key name.

The rules around navigation properties what's acceptable and how they are generated are defined by a set of conventions. I'm not going to go into any more detail on how these work but for more information you can take a look at the following pages:

EF Feature CTP5: Pluggable Conventions

MSDN - System.Data.Entity.ModelConfiguration.Conventions Namespace

Entity Framework Navigation Property generation rules

More than the defaults with the Model Builder

Entity framework provides a mechanism to configure additional information about navigation properties. This additional configuration can be done using the Model Builder. The model builder allows control over how Entity Framework represents the database, one of the features of the model builder allows for control over how foreign keys in the database are translated to Navigation Properties in the Entity Framework Model.

To get access to the model builder you will need to override the OnModelCreating method on your DbContext. You can then use the modelBuilder argument to structure your Entity Framework Model.

```
public class MyContext : DbContext
{
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
        }
}
```

The Model Builder uses a two-part fluent expression to define navigation properties within the model. 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. Navigation properties properties can be either Optional (ie 0..1 to x), Required (ie 1 to x) or Many (ie ** to x*). So lets take a look at some examples:

Optional relationship

If you have the following classes:

```
public class Entity1
{
  public int Id { get; set; }
}

public class Entity2
{
  public int Id { get; set; }

  public Entity1 Entity1 { get; set; }

}
```

Using the following model builder statement

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

   modelBuilder.Entity<Entity2>().HasOptional(e => e.Entity1).WithMany();

}
```

Will create the following relationship

Note the two parts of the query the first to define Entity1 navigation property, the first part defines an optional relationship (ie a nullable foreign key) and the second part .WithMany() defines the remote entity's (Entity1) multiplicity. Using .WithMany() with no argument tells Entity Framework that the relationship does not have a remote navigation property.

Required Relationship

If you have the following classes:

```
public class Entity1

{

  public int Id { get; set; }

  public List<Entity2> Entity2s { get; set; }

}


public class Entity2

{

  public int Id { get; set; }

  public Entity1 Entity1 { get; set; }

}
```

Using the following model builder statement

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

            modelBuilder.Entity<Entity2>()

                        .HasRequired(e => e.Entity1)
            .WithMany(e => e.Entity2s);
}
```

Will create the following relationship

Note that in this example we have specified a remote collection for the relationship. This means that you can use the Entity2s property to find all linked entities.

Many Relationship

If you have the following classes:

```
public class Entity1
{
      public int Id { get; set; }
      public List<Entity2> Entity2s { get; set; }
}

public class Entity2
{
      public int Id { get; set; }
      public List<Entity1> Entity1s { get; set; }
}
```

Using the following model builder statement

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
      modelBuilder.Entity<Entity2>()
            .HasMany(e => e.Entity1s)
            .WithMany(e => e.Entity2s);
}
```

Will produce the following relationship

With the following database tables

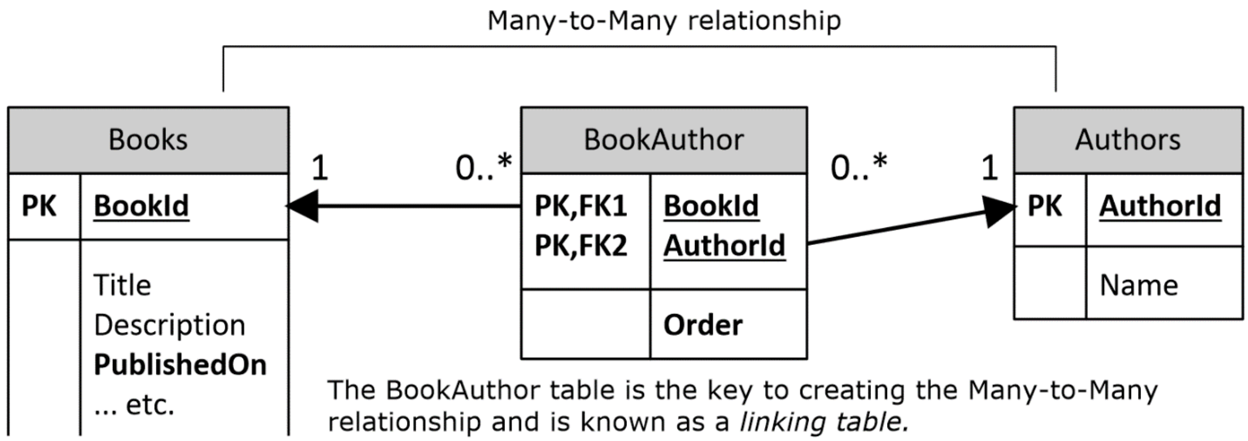
Note that here we have chained two .Many statements to produce a many to many relationship. As you can see the Entity Framework has created a link table in the database but the actual link in the model is created as two collections (without a link table) which makes more sense in c#.

Wrapping up

In this post we have looked in detail at navigation properties, what they are, how they are detected and how they are configured. As with many ORMs Entity Framework provides an abstraction over database Foreign Keys. As with any abstraction its really important to understand its limitations. I hope this post helps you better understand how Navigation Properties work and enables you to make good decisions when dealing with Entity Framework.

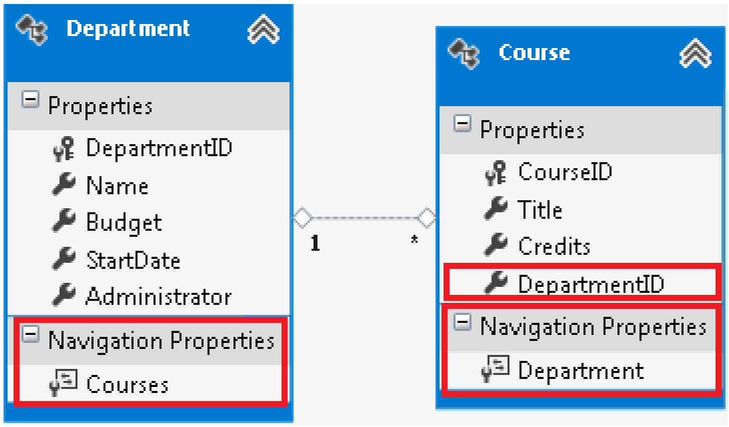
https://blog.staticvoid.co.nz/2012/entity_framework-navigation_property_basics_with_code_first/

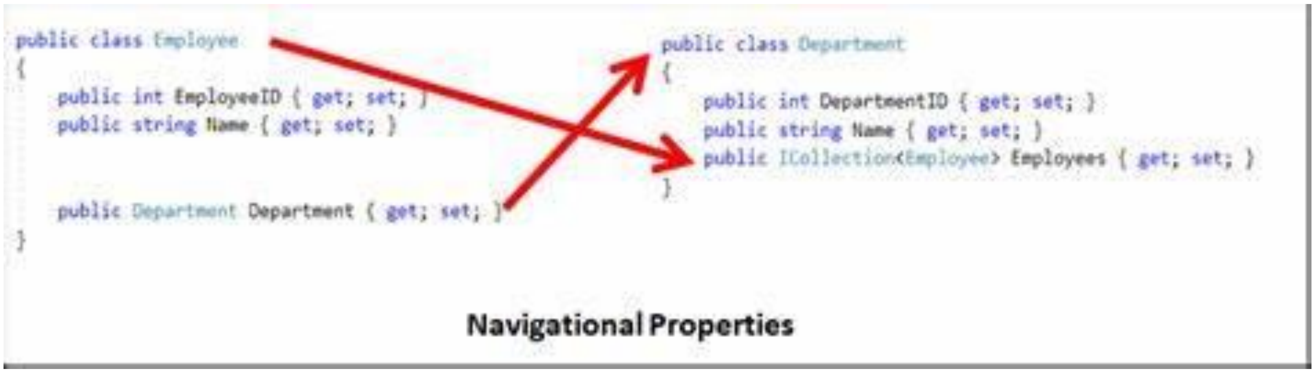
space



The BookAuthor table is the key to creating the Many-to-Many relationship and is known as a *linking table.*

Space

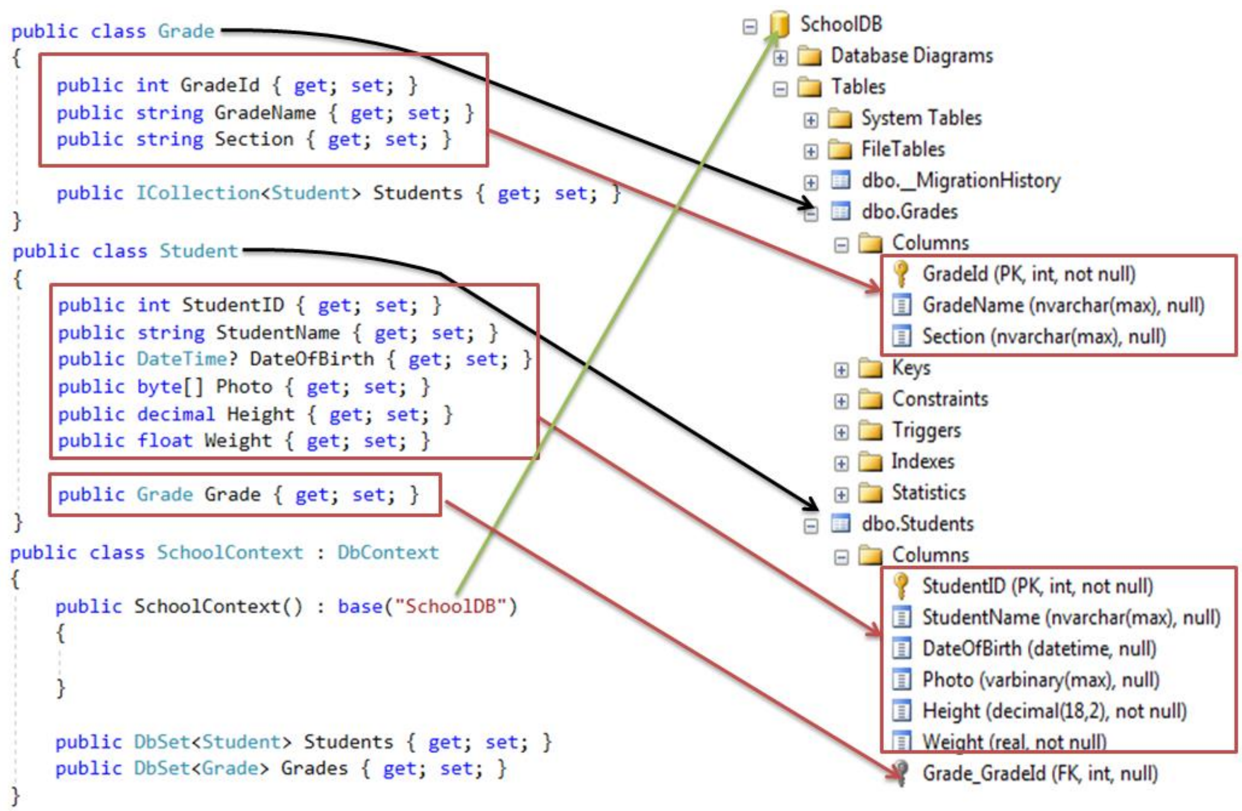




```
public class Employee
{
    public int EmployeeID { get; set; }
    public string Name { get; set; }


    public Department Department { get; set; }
}
```

```
public class Department
{
    public int DepartmentID { get; set; }
    public string Name { get; set; }
    public ICollection<Employee> Employees { get; set; }
}
```

Navigational Properties



```
public class Grade
{
    public int GradeId { get; set; }
    public string GradeName { get; set; }
    public string Section { get; set; }

    public ICollection<Student> Students { get; set; }
}
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime? DateOfBirth { get; set; }
    public byte[] Photo { get; set; }
    public decimal Height { get; set; }
    public float Weight { get; set; }

    public Grade Grade { get; set; }
}
public class SchoolContext : DbContext
{
    public SchoolContext() : base("SchoolDB")
    {

    }

    public DbSet<Student> Students { get; set; }
    public DbSet<Grade> Grades { get; set; }
}
```

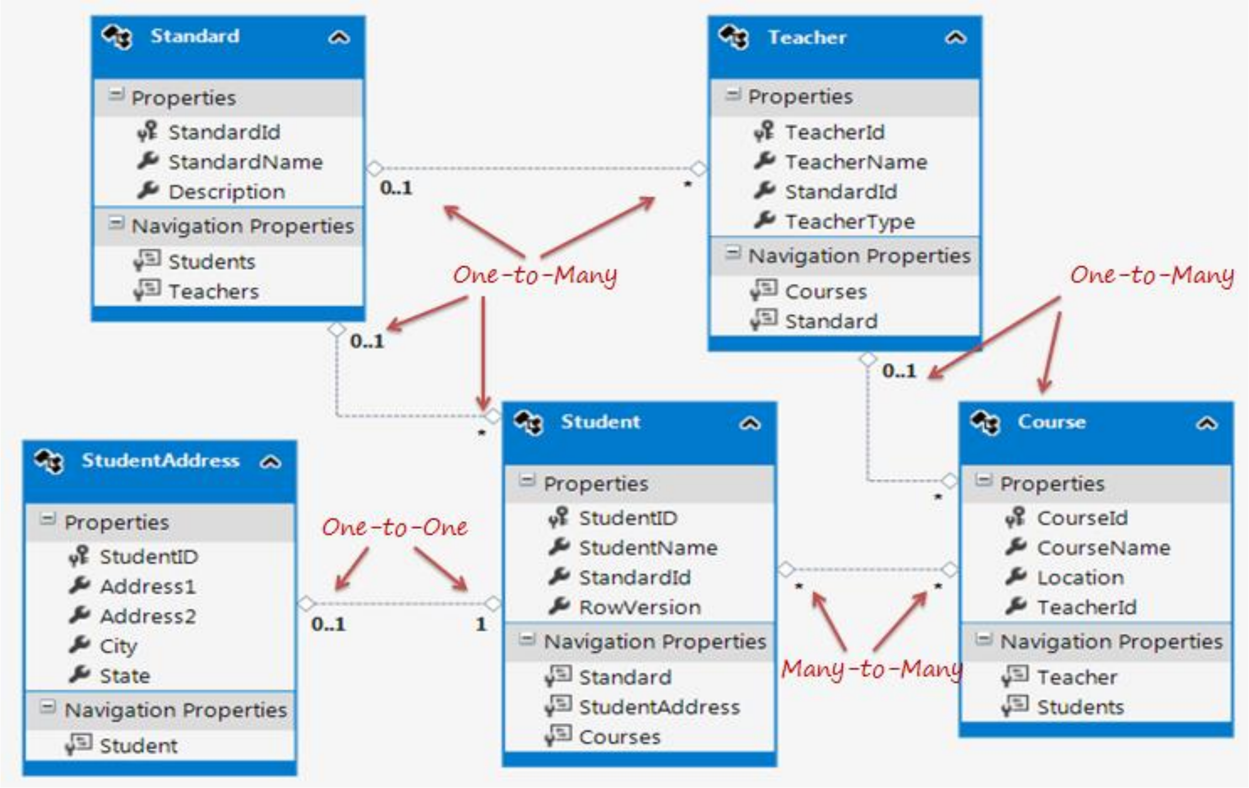
SchoolDB
Database Diagrams
Tables
System Tables
FileTables
dbo._MigrationHistory
dbo.Grades
Columns
GradeId (PK, int, not null)
GradeName (nvarchar(max), null)
Section (nvarchar(max), null)
Keys
Constraints
Triggers
Indexes
Statistics
dbo.Students
Columns
StudentID (PK, int, not null)
StudentName (nvarchar(max), null)
DateOfBirth (datetime, null)
Photo (varbinary(max), null)
Height (decimal(18,2), not null)
Weight (real, not null)
Grade_GradeId (FK, int, null)

Relationships between Entities in Entity Framework 6

Here, you will learn how entity framework manages the relationships between entities.

Entity framework supports three types of relationships, same as database: 1) One-to-One 2) One-to-Many, and 3) Many-to-Many.

We have created an Entity Data Model for the SchoolDB database in the Create Entity Data Model chapter. The following figure shows the visual designer for that EDM with all the entities and relationships among them.



Let's see how each relationship (association) is being managed by entity framework.

One-to-One Relationship

As you can see in the above figure, `Student` and `StudentAddress` have a One-to-One relationship (zero or one). A student can have only one or zero addresses. Entity framework adds the `Student` reference navigation property into the `StudentAddress` entity and the `StudentAddress` navigation entity into the `Student` entity. Also, the `StudentAddress` entity has both `StudentId` property as PrimaryKey and ForeignKey, which makes it a one-to-one relationship.

```
public partial class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public Nullable<int> StandardId { get; set; }
    public byte[] RowVersion { get; set; }
    public virtual StudentAddress StudentAddress { get; set; }
    }

public partial class StudentAddress
{
    public int StudentID { get; set; }
    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public virtual Student Student { get; set; }
}
```

In the above example, the `StudentId` property needs to be PrimaryKey as well as ForeignKey. This can be configured using Fluent API in the `OnModelCreating` method of the context class.

One-to-Many Relationship

The `Standard` and `Teacher` entities have a One-to-Many relationship marked by multiplicity where 1 is for One and * is for Many. This means that `Standard` can have many Teachers whereas `Teacher` can associate with only one `Standard`.

To represent this, the `Standard` entity has the collection navigation property `Teachers` (please notice that it's plural), which indicates that one `Standard` can have a collection of Teachers (many teachers). And the `Teacher` entity has a `Standard` navigation property (reference property), which indicates that `Teacher` is associated with one `Standard`. Also, it contains the `StandardId` foreign key (PK in `Standard` entity). This makes it a One-to-Many relationship.

```
public partial class Standard
{
    public Standard()
    {
        this.Teachers = new HashSet<Teacher>();
    }

    public int StandardId { get; set; }
    public string StandardName { get; set; }
    public string Description { get; set; }

    public virtual ICollection<Teacher> Teachers { get; set; }
}

public partial class Teacher
{
    public Teacher()
    {
        this.Courses = new HashSet<Course>();
    }

    public int TeacherId { get; set; }
    public string TeacherName { get; set; }
    public Nullable<int> TeacherType { get; set; }

    public Nullable<int> StandardId { get; set; }
    public virtual Standard Standard { get; set; }
}
```

Many-to-Many Relationship

The `Student` and `Course` have a Many-to-Many relationship marked by * multiplicity. It means one `Student` can enroll for many Courses and also, one `Course` can be taught to many Students.

The database includes the `StudentCourse` joining table which includes the primary key of both the tables (`Student` and `Course` tables). Entity Framework represents many-to-many relationships by not having the entity set (`DbSet property`) for the joining table in the CSDL and visual designer. Instead it manages this through mapping.

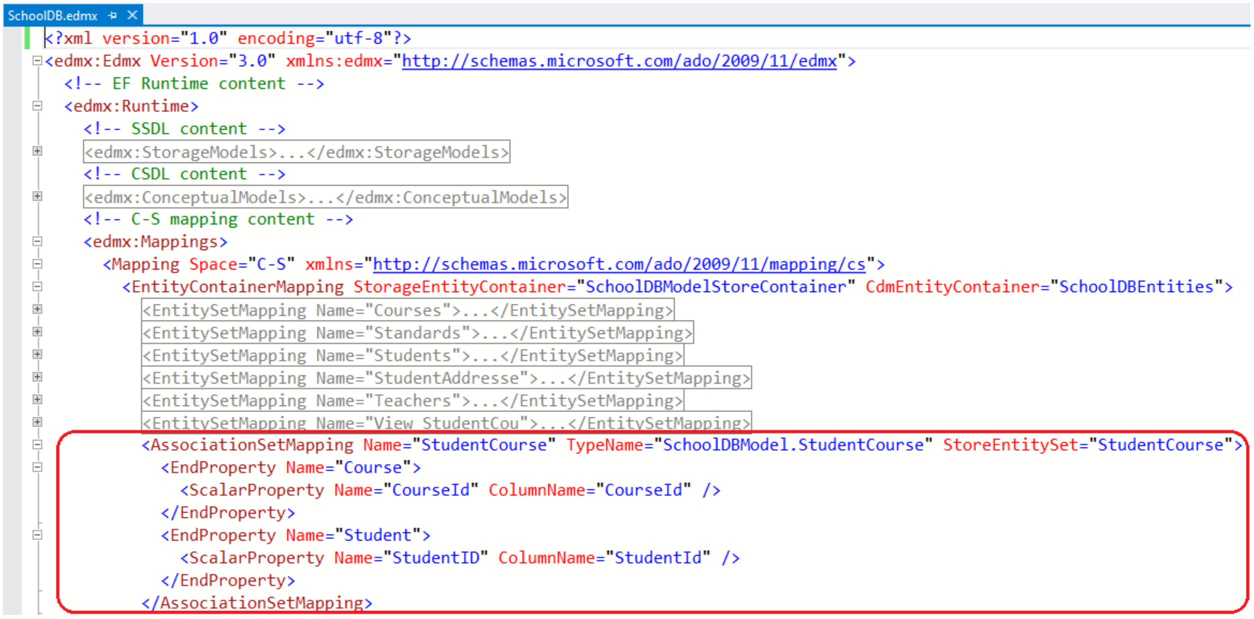
As you can see in the above figure, the `Student` entity includes the collection navigation property `Courses` and `Course` entity includes the collection navigation property `Students` to represent a many-to-many relationship between them.

The following code snippet shows the `Student` and `Course` entity classes.

```
public partial class Student
{
    public Student()
    {
        this.Courses = new HashSet<Course>();
    }

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public Nullable<int> StandardId { get; set; }
    public byte[] RowVersion { get; set; }
    public virtual ICollection<Course> Courses { get; set; }
}

public partial class Course
{
    public Course()
    {
        this.Students = new HashSet<Student>();
    }

    public int CourseId { get; set; }
    public string CourseName { get; set; }
    public System.Data.Entity.Spatial.DbGeography Location { get; set; }

    public virtual ICollection<Student> Students { get; set; }
}
```

Note: Entity framework supports many-to-many relationships only when the joining table (`StudentCourse` in this case) does **NOT** include any columns other than PKs of both tables. If the join tables contain additional columns, such as DateCreated, then the EDM creates an entity for the middle table as well and you will have to manage CRUD operations for many-to-many entities manually.

Open EDM in XML view. You can see that SSDL (storage schema) has the `StudentCourse` entityset, but CSDL doesn't have it. Instead, it's being mapped in the navigation property of the `Student` and `Course` entities. MSL (C-S Mapping) has mapping between `Student` and `Course` put into the `StudentCourse` table in the <AssociationSetMapping/> section.



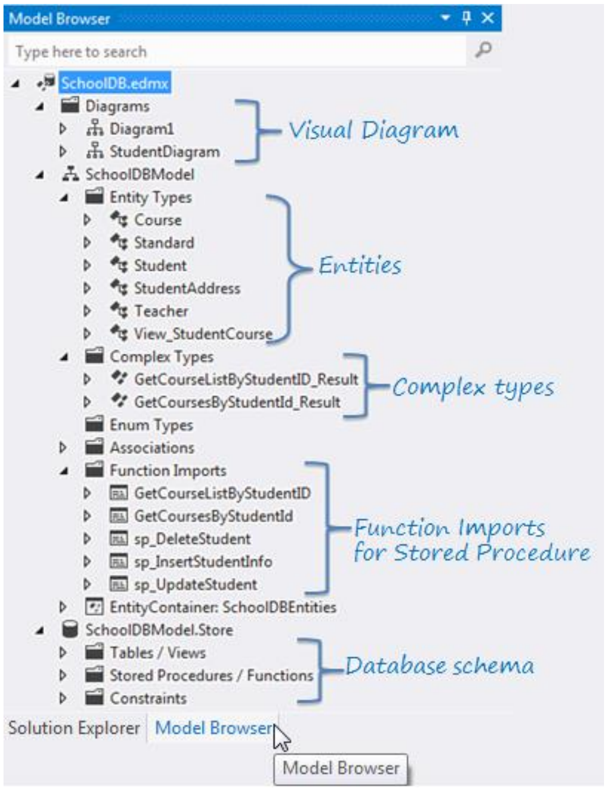
```
<?xml version="1.0" encoding="utf-8"?>
<edmx:Edmx Version="3.0" xmlns:edmx="http://schemas.microsoft.com/ado/2009/11/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>...</edmx:StorageModels>
    <!-- CSDL content -->
    <edmx:ConceptualModels>...</edmx:ConceptualModels>
    <!-- C-S mapping content -->
    <edmx:Mappings>
      <Mapping Space="C-S" xmlns="http://schemas.microsoft.com/ado/2009/11/mapping/cs">
        <EntityContainerMapping StorageEntityContainer="SchoolDBModelStoreContainer" CdmEntityContainer="SchoolDBEntities">
          <EntitySetMapping Name="Courses">...</EntitySetMapping>
          <EntitySetMapping Name="Standards">...</EntitySetMapping>
          <EntitySetMapping Name="Students">...</EntitySetMapping>
          <EntitySetMapping Name="StudentAddresse">...</EntitySetMapping>
          <EntitySetMapping Name="Teachers">...</EntitySetMapping>
          <EntitySetMapping Name="View_StudentCou">...</EntitySetMapping>
          <AssociationSetMapping Name="StudentCourse" TypeName="SchoolDBModel.StudentCourse" StoreEntitySet="StudentCourse">
            <EndProperty Name="Course">
              <ScalarProperty Name="CourseId" ColumnName="CourseId" />
            </EndProperty>
            <EndProperty Name="Student">
              <ScalarProperty Name="StudentID" ColumnName="StudentId" />
            </EndProperty>
          </AssociationSetMapping>
```

Thus, a many-to-many relationship is being managed by C-S mapping in EDM. So, when you add a `Student` in a `Course` or a `Course` in a `Student` entity and when you save it, it will then insert the PK of the added student and course in the `StudentCourse` table. So, this mapping not only enables a convenient association directly between the two entities, but also manages querying, inserts, and updates across these joins.

https://www.entityframeworktutorial.net/entity-relationships.aspx

The Model Browser contains all the information about the EDM, its conceptual model, storage model and mapping information, as shown below.

Bug: https://developercommunity.visualstudio.com/content/problem/578666/edmx-model-browser-not-working-in-version-161.html



As you can see in the above figure, the Model Browser contains the following objects:

Diagrams: The Model Browser contains visual diagrams of the EDM. We have seen a default visual diagram created by EDM. You can also create multiple diagrams for one EDM.

Entity Types: Entity Types lists all the class types which are mapped to the database tables.

Complex Types: Complex types are the classes which are generated by EDM to contain the result of stored procedures, table-valued functions etc. These complex types are customized classes for different purposes.

Enum Types: Enum Types lists all the entities which are used as Enum in the entity framework.

Associations: Associations lists all foreign key relationship between the entity types.

Function Imports: Function Imports lists all the functions which will be mapped to stored procedures, table-valued functions, etc. Stored procedures and table-valued functions will be used as functions and not as entities in EF.

.Store: Store represents the database schema (SSDL).

Baggage and Refs

----Skillpipe -

1. ViewBag property
    a. Module 4: Developing Controllers
    b. You are in chapter 6 at position 2570
2. Working with Forms - code examples
    a. Module 6: Developing Models
    b. You are in chapter 8 at position 3976
3. Migrations: https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/
    a. A data model changes during development and gets out of sync with the database. You can drop the database and let EF create a new one that matches the model, but this procedure results in the loss of data.
    b. The migrations feature in EF Core provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.
4. Module 7: Using Entity Framework Core in ASP.NET Core
    a. You are in chapter 9 at position 7373
5. Using Sections in a Layout
    a. Module 8: Using Layouts, CSS and JavaScript in ASP.NET Core MVC
    b. You are in chapter 10 at position 1132
6. Rendering and Executing JavaScript Code
7. Module 8: Using Layouts, CSS and JavaScript in ASP.NET Core MVC
    a. You are in chapter 10 at position 2518
8. jQuery
    a. Module 8: Using Layouts, CSS and JavaScript in ASP.NET Core MVC
    b. You are in chapter 10 at position 5433
    c. example:
    d. https://github.com/MicrosoftLearning/20486D-DevelopingASPNETMVCWebApplications/blob/master/Instructions/20486D_MOD08_LAK.md

References

9. https://blog.staticvoid.co.nz/2012/entity_framework-navigation_property_basics_with_code_first/
10. DbContext in Entity Framework 6
11. https://www.entityframeworktutorial.net/entityframework6/dbcontext.aspx
12. Relationships between Entities in Entity Framework 6
    a. https://www.entityframeworktutorial.net/entity-relationships.aspx
13. Navigation Property - good images

   a. https://www.c-sharpcorner.com/article/navigation-property-with-code-first-navigation-property-in-ef/

14. Relationships, navigation properties and foreign keys
    a. https://docs.microsoft.com/en-us/ef/ef6/fundamentals/relationships
15. Specifying attribute route optional parameters, default values, and constraints
    a. https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-2.2#specifying-attribute-route-optional-parameters-default-values-and-constraints