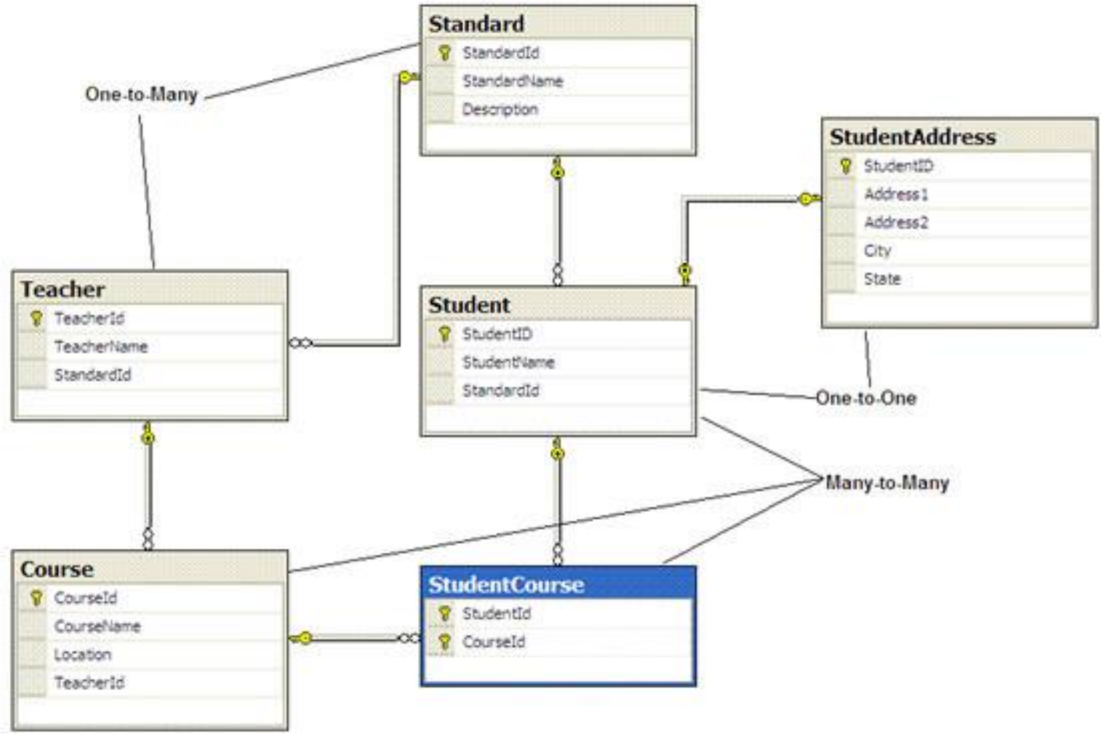


Linq-to-Entities Query

Here, you will learn how to write LINQ-to-Entities queries and get the result in Entity Framework 6.x as well as in Entity Framework Core. The `DbSet` class is derived from `IQuerayable`. So, we can use LINQ for querying against `DbSet`, which will be converted to an SQL query. EF API executes this SQL query to the underlying database, gets the flat result set, converts it into appropriate entity objects and returns it as a query result.

The following are some of the standard query operators (or extension methods) that can be used with LINQ-to-Entities queries.

LINQ Extension Methods
First()
FirstOrDefault()

Single()
SingleOrDefault()
ToList()
Count()
Min()
Max()
Last()
LastOrDefault()
Average()

Find():

In addition to LINQ extension methods, we can use the `Find()` method of `DbSet` to search the entity based on the primary key value.

Let's assume that `SchoolDbEntities` is our `DbContext` class and `Students` is the `DbSet` property.

```
var ctx = new SchoolDBEntities();
var student = ctx.Students.Find(1);
```

In the above example, `ctx.Student.Find(1)` returns a student record whose StudentId is 1 in the database. If no record is found, then it returns null. The above query will execute the following SQL query.

```
SELECT
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Extent1].[StandardId] AS [StandardId]
FROM [dbo].[Student] AS [Extent1]
WHERE [Extent1].[StudentId] = @p0',N'@p0 int',@p0=1
go
```

First/FirstOrDefault:

If you want to get a single student object, when there are many students, whose name is "Bill" in the database, then use First or FirstOrDefault, as shown below:

LINQ Query Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var student = (from s in ctx.Students
                  where s.StudentName == "Bill"
                  select s).FirstOrDefault<Student>();
}
```

LINQ Method Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var student = ctx.Students
                      .Where(s => s.StudentName == "Bill")
                      .FirstOrDefault<Student>();
}
```

EF 6 executes the following SQL query in the database for the above LINQ query.

```
SELECT TOP (1)
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Extent1].[StandardId] AS [StandardId]
FROM [dbo].[Student] AS [Extent1]
WHERE 'Bill' = [Extent1].[StudentName]
```

EF Core executes the following query in the database.

```
SELECT TOP (1)
[s].[StudentId], [s].[DoB], [s].[FirstName], [s].[GradeId],
[s].[LastName], [s].[MiddleName]
FROM [Students] AS [s]
WHERE [s].[FirstName] = N'Bill'
```

Parameterized Query:

EF builds and executes a parameterized query in the database if the LINQ-to-Entities query uses parameters, such as below.

```
using (var ctx = new SchoolDBEntities())
{
    string name = "Bill";
    var student = ctx.Students
                  .Where(s => s.StudentName == name)
                  .FirstOrDefault<Student>();
}
```

The above query will result into the following SQL query in EF 6.

```
SELECT TOP (1)
[Extent1].[StudentId] AS [StudentId],
[Extent1].[Name] AS [Name]
```

```
FROM [dbo].[Student] AS [Extent1]
WHERE ([Extent1].[Name] = @p__linq__0) OR (([Extent1].[Name] IS NULL)
         AND (@p__linq__0 IS NULL))',N'@p__linq__0 nvarchar(4000)',@p__linq__0=N'Bill'
```

The difference between `First` and `FirstOrDefault` is that `First()` will throw an exception if there is no result data for the supplied criteria, whereas `FirstOrDefault()` returns a default value (null) if there is no result data.

ToList:

The `ToList` method returns the collection result. If you want to list all the students with the same name then use `ToList()`:

```
using (var ctx = new SchoolDBEntities())
{
    var studentList = ctx.Students.Where(s => s.StudentName == "Bill").ToList();
}
```

We may also use ToArray, ToDictionary or ToLookup. The above query would result in the following database query:

```
SELECT
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Extent1].[StandardId] AS [StandardId]
FROM [dbo].[Student] AS [Extent1]
WHERE 'Bill' = [Extent1].[StudentName]
go
```

GroupBy:

Use the `group by` operator or `GroupBy` extension method to get the result based on the group by the particular property of an entity.

The following example gets the results grouped by each `Standard`. Use the foreach loop to iterate the group.

LINQ Query Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var students = from s in ctx.Students
                    group s by s.StandardId into studentsByStandard
                    select studentsByStandard;

    foreach (var groupItem in students)
    {
        Console.WriteLine(groupItem.Key);

        foreach (var stud in groupItem)
        {
            Console.WriteLine(stud.StudentId);
        }

    }
}
```

LINQ Method Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var students = ctx.Students.GroupBy(s => s.StandardId);

    foreach (var groupItem in students)
    {
        Console.WriteLine(groupItem.Key);

        foreach (var stud in groupItem)
        {
            Console.WriteLine(stud.StudentId);
        }

    }
}
```

The above query will execute the following database query:

```
SELECT
[Project2].[C1] AS [C1],
[Project2].[StandardId] AS [StandardId],
[Project2].[C2] AS [C2],
[Project2].[StudentID] AS [StudentID],
[Project2].[StudentName] AS [StudentName],
[Project2].[StandardId1] AS [StandardId1]
FROM ( SELECT
    [Distinct1].[StandardId] AS [StandardId],
    1 AS [C1],
    [Extent2].[StudentID] AS [StudentID],
    [Extent2].[StudentName] AS [StudentName],
    [Extent2].[StandardId] AS [StandardId1],
    CASE WHEN ([Extent2].[StudentID] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C2]
    FROM   (SELECT DISTINCT
        [Extent1].[StandardId] AS [StandardId]
        FROM [dbo].[Student] AS [Extent1] ) AS [Distinct1]
    LEFT OUTER JOIN [dbo].[Student] AS [Extent2] ON ([Distinct1].[StandardId] = [Extent2].[StandardId]) OR (([Distinct1].[StandardId] IS NULL) AND ([Extent2].[StandardId] IS NULL))

)  AS [Project2]

ORDER BY [Project2].[StandardId] ASC, [Project2].[C2] ASC

go
```

OrderBy:

Use the `OrderBy` operator with ascending/descending keywords in LINQ query syntax to get the sorted entity list.

```
using (var ctx = new SchoolDBEntities())

{

        var students = from s in ctx.Students

                       orderby s.StudentName ascending

                       select s;

}
```

Use the `OrderBy` or `OrderByDescending` method to get the sorted entity list.

```
using (var ctx = new SchoolDBEntities())

{

        var students = ctx.Students.OrderBy(s => s.StudentName).ToList();

        // or descending order

        var  descStudents = ctx.Students.OrderByDescending(s => s.StudentName).ToList();

}
```

The above query will execute the following database query:

```
SELECT

[Extent1].[StudentID] AS [StudentID],

[Extent1].[StudentName] AS [StudentName],

[Extent1].[StandardId] AS [StandardId]
```

```
FROM [dbo].[Student] AS [Extent1]
ORDER BY [Extent1].[StudentName] ASC
go
```

Anonymous Object Result:

LINQ-to-Entities queries do not always have to return entity objects. We may choose some of the properties of an entity as a result.

The following query returns a list of anonymous objects which contains `StudentId` and `StudentName` properties.

LINQ Query Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var anonymousObjResult = from s in ctx.Students
                             where s.StandardId == 1
                             select new {
                                Id = st.StudentId,
                                Name = st.StudentName
                             };

    foreach (var obj in anonymousObjResult)
    {
        Console.Write(obj.Name);
    }
}
```

LINQ Method Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var anonymousObjResult = ctx.Students
```

```
                                .Where(st => st.Standard == 1)
                                .Select(st => new {
                                            Id = st.StudentId,
                                            Name = st.StudentName });

    foreach (var obj in anonymousObjResult)
    {
        Console.Write(obj.Name);
    }
}
```

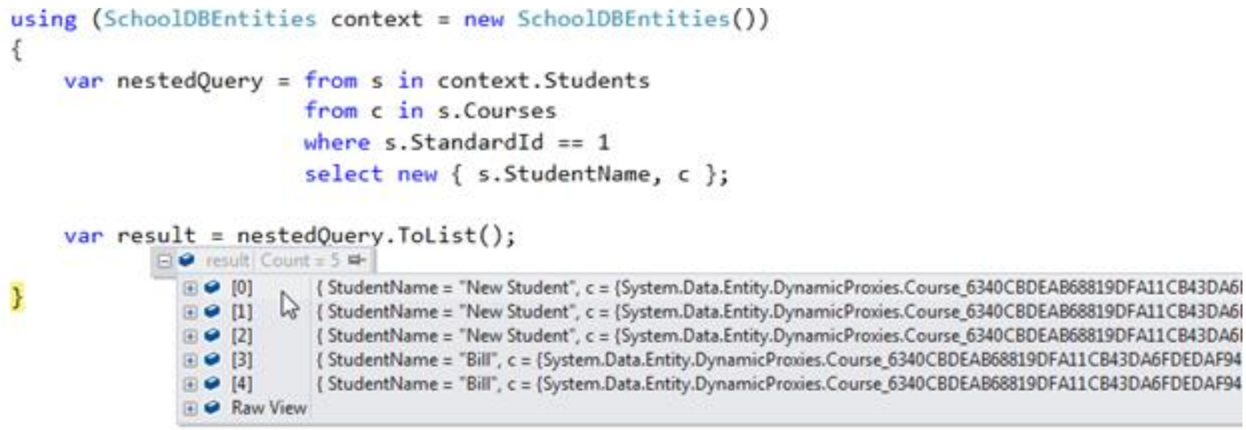
The above query will execute the following database query:

```
SELECT
[s].[StudentID] AS [Id], [s].[StudentName] AS [Name]
FROM [Student] AS [s]
WHERE [s].[StandardId] = 1
go
```

The projectionResult in the above query will be the anonymous type, because there is no class/entity which has these properties. So, the compiler will mark it as anonymous.

Nested queries:

You can also execute nested LINQ-to-entity queries as shown below:



```
using (SchoolDBEntities context = new SchoolDBEntities())
{
    var nestedQuery = from s in context.Students
                      from c in s.Courses
                      where s.StandardId == 1
                      select new { s.StudentName, c };

    var result = nestedQuery.ToList();
}
```

```
result  Count = 5
  [0]  { StudentName = "New Student", c = {System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA6
  [1]  { StudentName = "New Student", c = {System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA6
  [2]  { StudentName = "New Student", c = {System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA6
  [3]  { StudentName = "Bill", c = {System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA6FDEDAF94
  [4]  { StudentName = "Bill", c = {System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA6FDEDAF94
  Raw View
```

The nested query shown above will result in an anonymous list with a `StudentName` and `Course` object.

```
SELECT
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Join1].[CourseId1] AS [CourseId],
[Join1].[CourseName] AS [CourseName],
[Join1].[Location] AS [Location],
[Join1].[TeacherId] AS [TeacherId]
FROM  [dbo].[Student] AS [Extent1]
INNER JOIN  (SELECT [Extent2].[StudentId] AS [StudentId],
        [Extent3].[CourseId] AS [CourseId1], [Extent3].[CourseName] AS [CourseName],
        [Extent3].[Location] AS [Location], [Extent3].[TeacherId] AS [TeacherId]
    FROM  [dbo].[StudentCourse] AS [Extent2]
    INNER JOIN [dbo].[Course] AS [Extent3]
        ON [Extent3].[CourseId] = [Extent2].[CourseId] ) AS [Join1]
        ON [Extent1].[StudentID] = [Join1].[StudentId]
WHERE 1 = [Extent1].[StandardId]
go
```

In this way, you can do a projection of the result, as you want the data to be.

Eager Loading in Entity Framework

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query, so that we don't need to execute a separate query for related entities. Eager loading is achieved using the **Include()** method.

In the following example, it gets all the students from the database along with its standards using the `Include`() method.

LINQ Query Syntax:

```
using (var context = new SchoolDBEntities())
{
    var stud1 = (from s in context.Students.Include("Standard")
                where s.StudentName == "Bill"
                select s).FirstOrDefault<Student>();
}
```

LINQ Method Syntax:

```
using (var ctx = new SchoolDBEntities())
{
    var stud1 = ctx.Students
                   .Include("Standard")
                   .Where(s => s.StudentName == "Bill")
                   .FirstOrDefault<Student>();
}
```

The above LINQ queries will result in following SQL query:

```
SELECT TOP (1)
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Extent2].[StandardId] AS [StandardId],
[Extent2].[StandardName] AS [StandardName],
[Extent2].[Description] AS [Description]
FROM  [dbo].[Student] AS [Extent1]
LEFT OUTER JOIN [dbo].[Standard] AS [Extent2] ON [Extent1].[StandardId] =
[Extent2].[StandardId]
WHERE 'Bill' = [Extent1].[StudentName]
```

Use Lambda Expression:

You can also use the LINQ lambda expression as a parameter in the `Include` method. For this, take a reference of `System.Data.Entity`namespace and use the lambda expression as shown below:

```
using System;
using System.Data.Entity;

class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new SchoolDBEntities())
        {
            var stud1 = ctx.Students.Include(s => s.Standard)
                                .Where(s => s.StudentName == "Bill")
                                .FirstOrDefault<Student>();
        }
    }
}
```

Load Multiple Entities:

You can also eagerly load multiple levels of related entities. The following example query eagerly loads the `Student`, `Standard` and `Teacher`entities:

```
using (var ctx = new SchoolDBEntities())
{
    var stud1 = ctx.Students.Include("Standard.Teachers")
                    .Where(s => s.StudentName == "Bill")
                    .FirstOrDefault<Student>();
}
```

Or use the lambda expression as below:

```
using (var ctx = new SchoolDBEntities())
{
    var stud1 = ctx.Students.Include(s => s.Standard.Teachers)
                    .Where(s => s.StudentName == "Bill")
                    .FirstOrDefault<Student>();
}
```

The above query will execute the following SQL query in the database:

```
SELECT [Project2].[StudentID] AS [StudentID],
[Project2].[StudentName] AS [StudentName],
[Project2].[StandardId] AS [StandardId],
[Project2].[StandardName] AS [StandardName],
[Project2].[Description] AS [Description],
[Project2].[C1] AS [C1],
[Project2].[TeacherId] AS [TeacherId],
[Project2].[TeacherName] AS [TeacherName],
[Project2].[StandardId1] AS [StandardId1]
FROM ( SELECT
    [Limit1].[StudentID] AS [StudentID],
    [Limit1].[StudentName] AS [StudentName],
    [Limit1].[StandardId1] AS [StandardId],
    [Limit1].[StandardName] AS [StandardName],
    [Limit1].[Description] AS [Description],

    [Project1].[TeacherId] AS [TeacherId],

    [Project1].[TeacherName] AS [TeacherName],

    [Project1].[StandardId] AS [StandardId1],

    CASE WHEN ([Project1].[TeacherId] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS [C1]

    FROM   (SELECT TOP (1) [Extent1].[StudentID] AS [StudentID], [Extent1].[StudentName] AS [StudentName], [Extent1].[StandardId] AS [StandardId2], [Extent2].[StandardId] AS [StandardId1], [Extent2].[StandardName] AS [StandardName], [Extent2].[Description] AS [Description]

        FROM  [dbo].[Student] AS [Extent1]

        LEFT OUTER JOIN [dbo].[Standard] AS [Extent2] ON [Extent1].[StandardId] = [Extent2].[StandardId]

        WHERE 'updated student' = [Extent1].[StudentName] ) AS [Limit1]

    LEFT OUTER JOIN  (SELECT

        [Extent3].[TeacherId] AS [TeacherId],

        [Extent3].[TeacherName] AS [TeacherName],

        [Extent3].[StandardId] AS [StandardId]

        FROM [dbo].[Teacher] AS [Extent3]

        WHERE [Extent3].[StandardId] IS NOT NULL ) AS [Project1] ON [Limit1].[StandardId2] = [Project1].[StandardId]

)  AS [Project2]

ORDER BY [Project2].[StudentID] ASC, [Project2].[StandardId] ASC, [Project2].[C1] ASC
```

EF Core supports additional method `IncludeThen` for eager loading.

Lazy Loading in Entity Framework

Lazy loading is delaying the loading of related data, until you specifically request for it. It is the opposite of eager loading. For example, the `Student` entity contains the `StudentAddress` entity. In the lazy loading, the context first loads the `Student` entity data from the database, then it will load the `StudentAddress` entity when we access the `StudentAddress` property as shown below.

```
using (var ctx = new SchoolDBEntities())
{
    //Loading students only
    IList<Student> studList = ctx.Students.ToList<Student>();

    Student std = studList[0];

    //Loads Student address for particular Student only (seperate SQL quer
y)
    StudentAddress add = std.StudentAddress;
}
```

The code shown above will result in two SQL queries. First, it will fetch all students:

```
SELECT
[Extent1].[StudentID] AS [StudentID],
[Extent1].[StudentName] AS [StudentName],
[Extent1].[StandardId] AS [StandardId]
FROM [dbo].[Student] AS [Extent1]
```

Then, it will send the following query when we get the reference of StudentAddress:

```
exec sp_executesql N'SELECT
[Extent1].[StudentID] AS [StudentID],
[Extent1].[Address1] AS [Address1],
[Extent1].[Address2] AS [Address2],
[Extent1].[City] AS [City],
[Extent1].[State] AS [State]
```

```
FROM [dbo].[StudentAddress] AS [Extent1]

WHERE [Extent1].[StudentID] = @EntityKeyValue1',N'@EntityKeyValue1 int',@E
ntityKeyValue1=1
```

Disable Lazy Loading:

We can disable lazy loading for a particular entity or a context. To turn off lazy loading for a particular property, do not make it virtual. To turn off lazy loading for all entities in the context, set its configuration property to false.

```
using System;

using System.Data.Entity;

using System.Data.Entity.Infrastructure;

using System.Data.Entity.Core.Objects;

using System.Linq;


public partial class SchoolDBEntities : DbContext

{

    public SchoolDBEntities(): base("name=SchoolDBEntities")

    {

        this.Configuration.LazyLoadingEnabled = false;

    }


    protected override void OnModelCreating(DbModelBuilder modelBuilder)

    {

    }

}
```

Rules for lazy loading:

1. *context.Configuration.ProxyCreationEnabled* should be true.
2. *context.Configuration.LazyLoadingEnabled* should be true.
3. Navigation property should be defined as public, virtual. Context will **NOT** do lazy loading if the property is not defined as virtual.

Explicit Loading in Entity Framework

Here you will learn how to load related entities in an entity graph explicitly. Explicit loading is valid in EF 6 and EF Core both.

Even with lazy loading disabled (in EF 6), it is still possible to lazily load related entities, but it must be done with an explicit call. Use the `Load()` method to load related entities explicitly. Consider the following example.

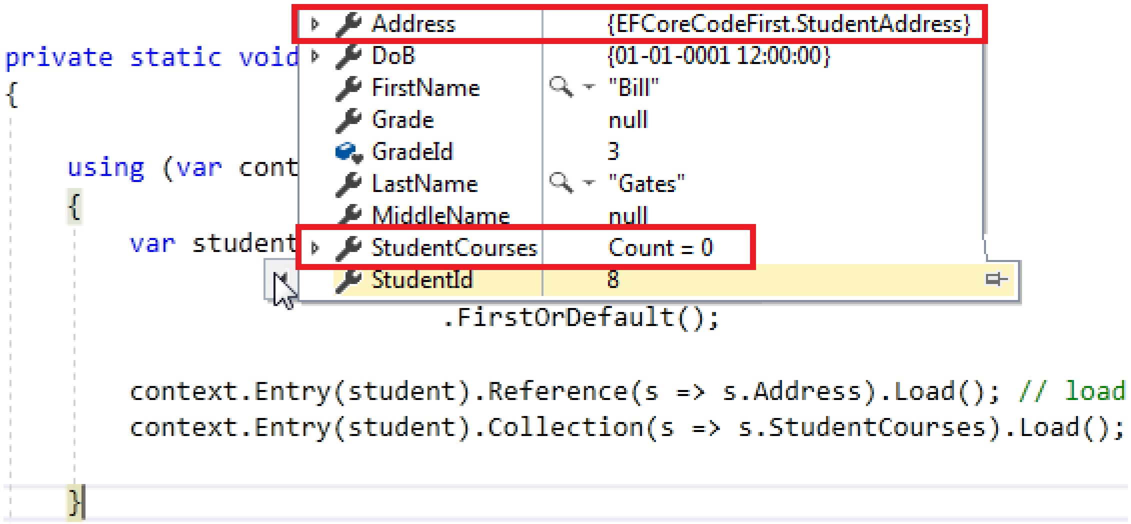
```
using (var context = new SchoolContext())
{
    var student = context.Students
                        .Where(s => s.FirstName == "Bill")
                        .FirstOrDefault<Student>();

    context.Entry(student).Reference(s => s.StudentAddress).Load(); // loads StudentAddress
    context.Entry(student).Collection(s => s.StudentCourses).Load(); // loads Courses collection
}
```

In the above example, `context.Entry(student).Reference(s => s.StudentAddress).Load()` loads the `StudentAddress` entity. The `Reference()` method is used to get an object of the specified reference navigation property and the `Load()` method loads it explicitly.

In the same way, `context.Entry(student).Collection(s => s.Courses).Load()` loads the collection navigation property `Courses` of the `Student`entity. The `Collection()` method gets an object that represents the collection navigation property.

The `Load()` method executes the SQL query in the database to get the data and fill up the specified reference or collection property in the memory, as shown below.



```
Address         {EFCoreCodeFirst.StudentAddress}
DoB             {01-01-0001 12:00:00}
FirstName       "Bill"
Grade           null
GradeId         3
LastName        "Gates"
MiddleName      null
StudentCourses  Count = 0
StudentId       8
```

```
private static void
{

    using (var cont
    {
        var student
                    .FirstOrDefault();

        context.Entry(student).Reference(s => s.Address).Load(); // load
        context.Entry(student).Collection(s => s.StudentCourses).Load();

    }
```

Query():

You can also write LINQ-to-Entities queries to filter the related data before loading. The `Query()` method enables us to write further LINQ queries for the related entities to filter out related data.

```
using (var context = new SchoolContext())
{
    var student = context.Students
                        .Where(s => s.FirstName == "Bill")
                        .FirstOrDefault<Student>();

    context.Entry(student)
           .Collection(s => s.StudentCourses)
           .Query()
               .Where(sc => sc.CourseName == "Maths")
               .FirstOrDefault();
}
```

In the above example, `.Collection(s => s.StudentCourses).Query()` allows us to write further queries for the `StudentCourses` entity.