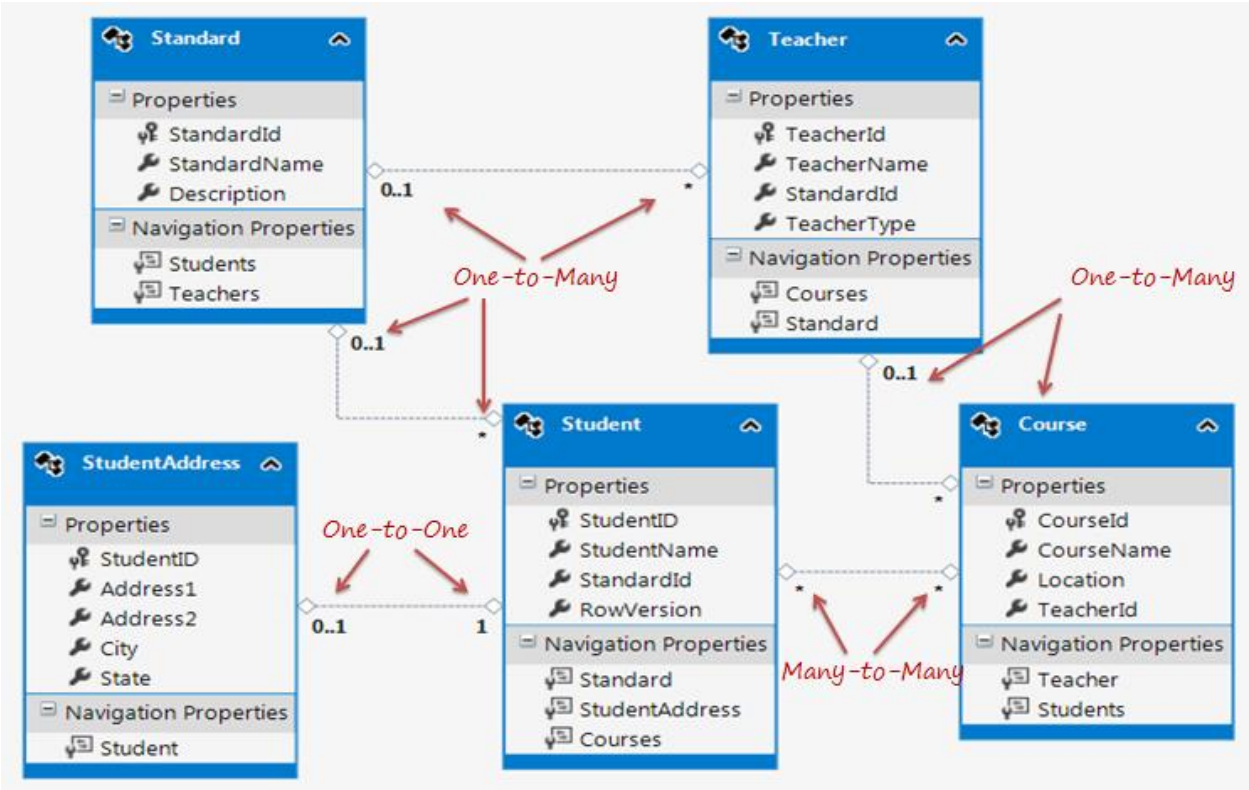


Linq-to-Entities Projection Queries

Here, you will learn how to write LINQ-to-Entities queries and get the result entities. Knowledge of LINQ is a prerequisite.

Projection is a process of selecting data in a different shape rather than a specific entity being queried. There are many ways of projection. We will now see some projection styling:

First/FirstOrDefault:

If you want to get a single student object, when there are many students, whose name is "Student1" in the database, then use First or FirstOrDefault, as shown below:

```
using (var ctx = new SchoolDBEntities())
{
    var student = (from s in ctx.Students
                   where s.StudentName == "Student1"
                   select s).FirstOrDefault<Student>();
 }
```

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 default value (null) if there is no result data.

Single/SingleOrDefault:

You can also use Single or SingleOrDefault to get a single student object as shown below:

```
using (var ctx = new SchoolDBEntities())
{
    var student = (from s in context.Students
                   where s.StudentID == 1
                   select s).SingleOrDefault<Student>();
}
```

Single or SingleOrDefault will throw an exception, if the result contains more than one element. Use Single or SingleOrDefault where you are sure that the result would contain only one element. If the result has multiple elements then there must be some problem.

ToList:

If you want to list all the students whose name is 'Student1' (provided there are many students has same name) then use ToList():

```
using (var ctx = new SchoolDBEntities())
  {
      var studentList = (from s in ctx.Students
      where s.StudentName == "Student1"
      select s).ToList<Student>();
  }
```

GroupBy:

If you want to group students by standardId, then use groupby:

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

OrderBy:

If you want to get the list of students sorted by StudentName, then use OrderBy:

```
using (var ctx = new SchoolDBEntities())
{
     var student1 = from s in ctx.Students
                    orderby s.StudentName ascending
                    select s;
}
```

Anonymous Class result:

If you want to get only StudentName, StandardName and list of Courses for that student in a single object, then write the following projection:

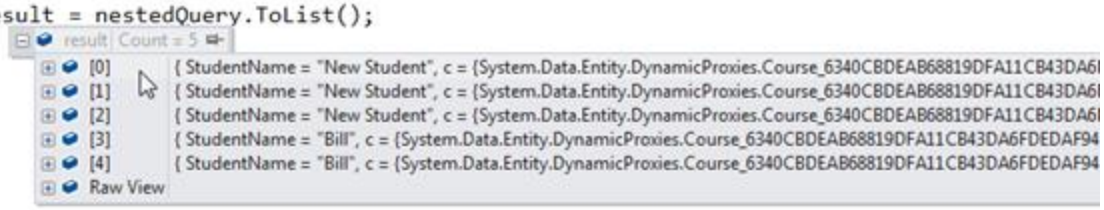
```
using (var ctx = new SchoolDBEntities())
{
    var projectionResult = from s in ctx.Students
                           where s.StudentName == "Student1"
                           select new {
                           s.StudentName, s.Standard.StandardName, s.Courses
                           };
}
```

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();
}
```



In this way, you can do a projection of the result, in the way that you would like the data to be.

DBEntityEntry Class

DBEntityEntry is an important class, which is useful in retrieving various information about an entity. You can get an instance of DBEntityEntry of a particular entity by using Entry method of DBContext. For example:

```
DBEntityEntry studentEntry = dbcontext.Entry(StudentEntity);
```

DBEntityEntry enables you to access entity state, current, and original values of all the property of a given entity. The following example code shows how to retrieve important information of a particular entity.

```
   using (var dbCtx = new SchoolDBEntities())
    {
        //get student whose StudentId is 1
        var student = dbCtx.Students.Find(1);

        //edit student name
        student.StudentName = "Edited name";

        //get DbEntityEntry object for student entity object
        var entry = dbCtx.Entry(student);

        //get entity information e.g. full name
        Console.WriteLine("Entity Name: {0}",
entry.Entity.GetType().FullName);

        //get current EntityState
        Console.WriteLine("Entity State: {0}", entry.State );

        Console.WriteLine("********Property Values********");

        foreach (var propertyName in entry.CurrentValues.PropertyNames )
        {
            Console.WriteLine("Property Name: {0}", propertyName);

            //get original value
            var orgVal = entry.OriginalValues[propertyName];
            Console.WriteLine("     Original Value: {0}", orgVal);

            //get current values
            var curVal = entry.CurrentValues[propertyName];
            Console.WriteLine("     Current Value: {0}", curVal);
        }

    }
```

Output: `Entity Name: Student`
`Entity State: Modified`

```
********Property Values********
Property Name: StudentID
Original Value: 1
Current Value: 1
Property Name: StudentName
Original Value: First Student Name
Current Value: Edited name
Property Name: StandardId
Original Value:
Current Value:
```

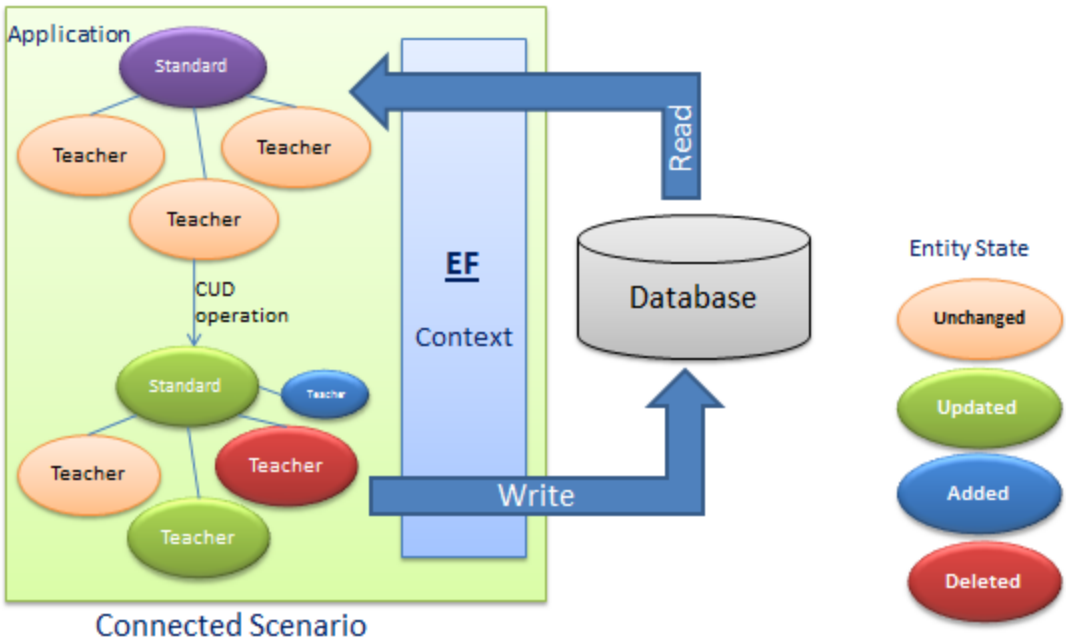
DbEntityEntry enables you to set Added, Modified or Deleted EntityState to an entity as shown below.

```
context.Entry(student).State = System.Data.Entity.EntityState.Modified;
```

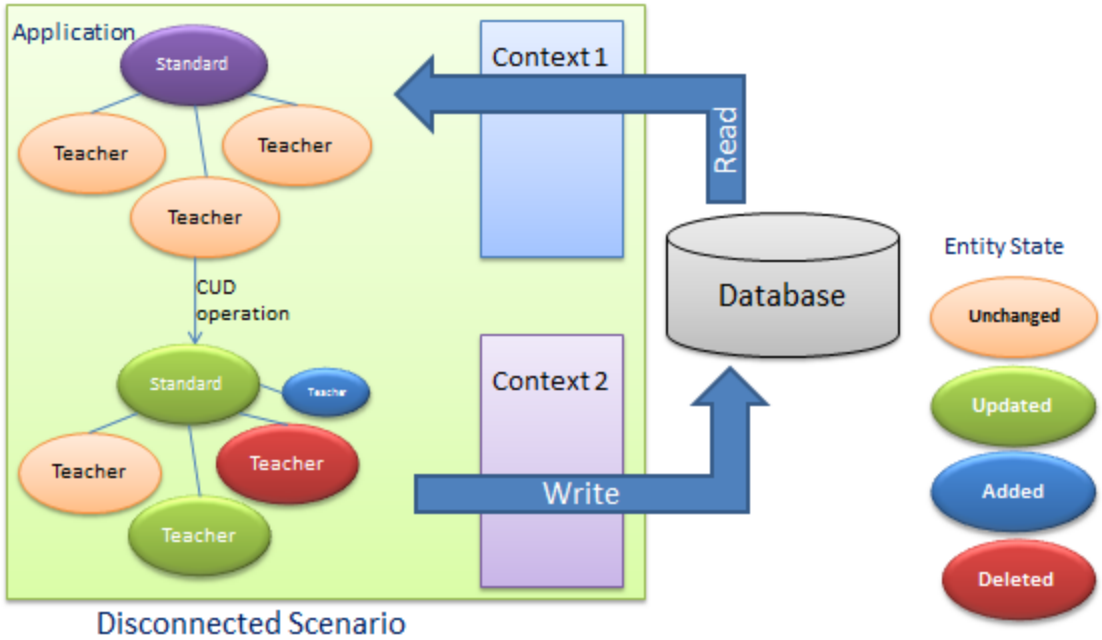
Persistence in Entity Framework

There are two scenarios when persisting an entity using EntityFramework, connected and disconnected scenarios.

Connected Scenario: This is when an entity is retrieved from the database and persist is used in the same context. Context object doesn't destroy between entity retrieval and persistence of entities.



Disconnected Scenario: Disconnected scenario is when an entity is retrieved from the database and the changed entities are submitted using the different objects in the context. The following example illustrates disconnected scenario:



As per the above scenario, Context1 is used for read operation and then Context1 is destroyed. Once entities change, the application submits entities using Context2 - a different context object.

Disconnected scenario is complex because the new context doesn't know anything about the modified entity so you will have to instruct the context of what has changed in the entity. In the figure below, the application retrieves an entity graph using Context 1 and then the application performs some CUD (Create, Update, Delete) operations on it and finally, it saves the entity graph using Context 2. Context 2 doesn't know what operation has been performed on the entity graph in this scenario.

Add New Entity using DBContext in Disconnected Scenario

Add new entity in DbContext in the **disconnected scenario**, which in turn inserts a new row in a database table.

The following code shows how to save a single entity.

```
class Program
{
    static void Main(string[] args)
    {
```

```
            // create new Student entity object in disconnected scenario (out
of the scope of DbContext)
            var newStudent = new Student();

            //set student name
            newStudent.StudentName = "Bill";

            //create DBContext object
            using (var dbCtx = new SchoolDBEntities())
            {
                //Add Student object into Students DBset
                dbCtx.Students.Add(newStudent);

                // call SaveChanges method to save student into database
                dbCtx.SaveChanges();
            }
        }
    }
```

As you can see in the above code snippet, first, we have created a new Student entity object and set StudentName to 'Bill'. Second, we have created a new DBContext object and added newStudent into Students EntitySet. Third, we called SaveChanges method of DBContext which will execute the following insert query to the database.

Alternatively, we can also add entity into DBContext.Entry and mark it as Added which results in the same insert query:

```
    class Program
    {
        static void Main(string[] args)
        {
           // create new Student entity object in disconnected scenario (out
of the scope of DbContext)
            var newStudent = new Student();

            //set student name
            newStudent.StudentName = "Bill";

            //create DBContext object
            using (var dbCtx = new SchoolDBEntities())
            {
                //Add newStudent entity into DbEntityEntry and mark
EntityState to Added
                dbCtx.Entry(newStudent).State =
System.Data.Entity.EntityState.Added;

                // call SaveChanges method to save new Student into database
                dbCtx.SaveChanges();
            }
        }
    }
```

So, in this way, you can add a new single entity in the disconnected scenario.

Update Existing Entity using DBContext in Disconnected Scenario

The following example shows how to update a Student entity in the disconnected scenario:

```
    Student stud;
    //1. Get student from DB
    using (var ctx = new SchoolDBEntities())
    {
        stud = ctx.Students.Where(s => s.StudentName == "New
Student1").FirstOrDefault<Student>();
    }

    //2. change student name in disconnected mode (out of ctx scope)
    if (stud != null)
    {
        stud.StudentName = "Updated Student1";
    }

    //save modified entity using new Context
    using (var dbCtx = new SchoolDBEntities())
    {
        //3. Mark entity as modified
        dbCtx.Entry(stud).State = System.Data.Entity.EntityState.Modified;

        //4. call SaveChanges
        dbCtx.SaveChanges();
    }
```

As you see in the above code snippet, we are doing the following steps:

1. Get the existing student from DB.
2. Change the student name out of Context scope (disconnected mode)
3. Pass the modified entity into the Entry method to get its DBEntityEntry object and then mark its state as Modified
4. Call SaveChanges() method to update student information into the database.

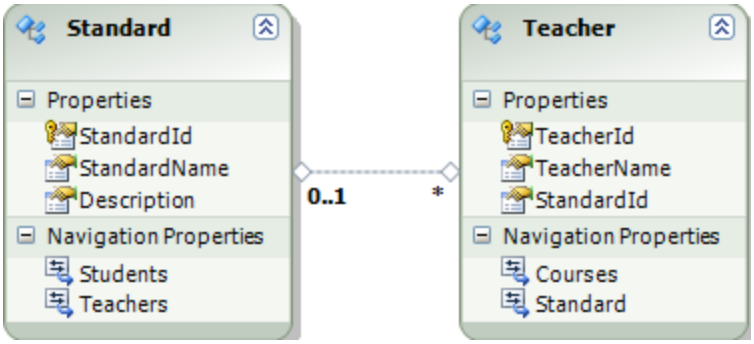
Delete Entity using DBContext in Disconnected Scenario

We used the Entry() method of DbContext to mark EntityState as Modified in the previous chapter. In the same way, we can use the Entry() method to attach a disconnected entity to the context and mark its state to Deleted.

```
    Student studentToDelete;
    //1. Get student from DB
    using (var ctx = new SchoolDBEntities())
    {
        studentToDelete = ctx.Students.Where(s => s.StudentName ==
"Student1").FirstOrDefault<Student>();
    }

    //Create new context for disconnected scenario
    using (var newContext = new SchoolDBEntities())
    {
        newContext.Entry(studentToDelete).State =
System.Data.Entity.EntityState.Deleted;

        newContext.SaveChanges();
    }
```

Add One-to-Many Relationship Entity Graph using DBContext

We will see how to add new Standard and Teacher entities which has One-to-Many relationship which results in single entry in 'Standard' database table and multiple entry in 'Teacher' table.



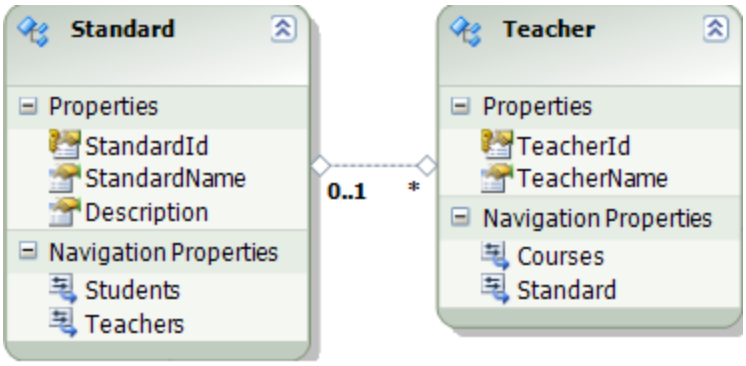
[Standard and Teacher has One-to-Many relationship]

```
    //Create new standard
    var standard = new Standard();
```

```
standard.StandardName = "Standard1";

//create three new teachers
var teacher1 = new Teacher();
teacher1.TeacherName = "New Teacher1";

var teacher2 = new Teacher();
teacher2.TeacherName = "New Teacher2";

var teacher3 = new Teacher();
teacher3.TeacherName = "New Teacher3";

//add teachers for new standard
standard.Teachers.Add(teacher1);
standard.Teachers.Add(teacher2);
standard.Teachers.Add(teacher3);

using (var dbCtx = new SchoolDBEntities())
{
    //add standard entity into standards entitySet
    dbCtx.Standards.Add(standard);
    //Save whole entity graph to the database
    dbCtx.SaveChanges();
}
```

Update One-to-Many Entity using DBContext

We will see how to update existing Standard and Teacher entities in disconnected scenario which has One-to-Many relationship.



[Standard and Teacher has One-to-Many relationship]

The same way as we did it in the case of One-to-One entity relationship, here also we have to find which teacher entities are added, modified or removed from the collection property of Standard entity in disconnected scenario.

Consider following scenario in disconnected mode for One-to-Many entity relationship:

- User might have added new Teachers in the collection
- User might have modified existing Teachers of the collection
- User might have removed existing Teachers from the collection

So you have to find entity state of each entities in the collection.

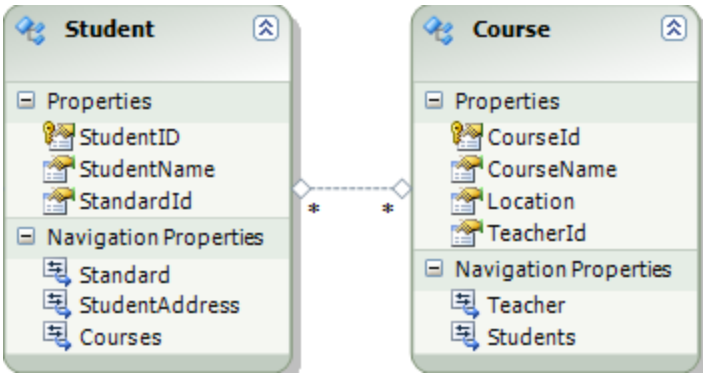
Below code snippet shows how you can handle above scenario and update One-to-Many entities:

```
Standard std = null;

        using (var ctx = new SchoolDBContext())
        {
            //fetching existing standard from the db
            std = (from s in ctx.Standards.Include("Teachers")
                   where s.StandardName == "standard1"
                   select s).FirstOrDefault<Standard>();
        }
std.StandardName = "Updated standard3";
        std.Description = "Updated standard description";

        if (std.Teachers  != null)
        {
            if (std.Teachers.Count >= 2)
            {
                //get the first element to be updated
                Teacher updateTchr =
std.Teachers.ElementAt<Teacher>(0);

                //get the second element to be removed
                Teacher deletedTchr =
std.Teachers.ElementAt<Teacher>(1);

                //remove updated teacher to re-add later
                std.Teachers.Remove(updateTchr);

                //delete second teacher from the list
                // deleted second teacher
                std.Teachers.Remove(deletedTchr);

                //Update first teacher in the list
                updateTchr.TeacherName = "Updated Teacher1";

                // re-add first teacher
                std.Teachers.Add(updateTchr);
```

```
            }
        }

        // adding new teacher for selected standard
        Teacher newTeacher = new Teacher();
        newTeacher.TeacherName = "NewTeacher";
        std.Teachers.Add(newTeacher);
```

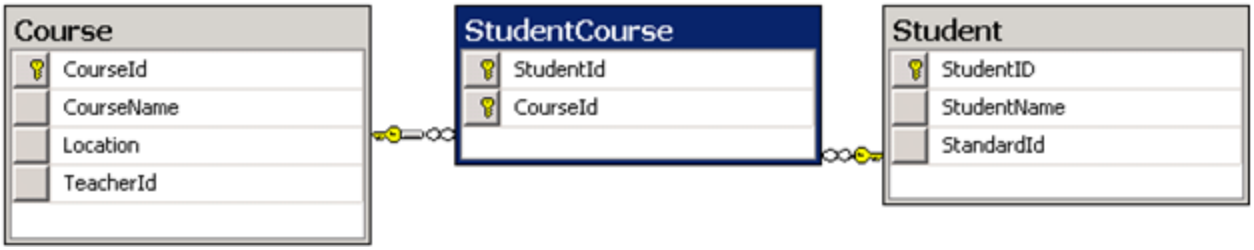
Add Many-to-Many Relationship Entity Graph using DBContext

We will see how to add new courses in student's course collection. Student and Course has Many-to-Many relationship which results in insert new rows in Student and StudentCourse tables.



[Student and Course has Many-to-Many relationship]

If you see database design, actually there are three tables participates in Many-to-Many relationship between Student and Course, Student, Course and StudentCourse tables. StudentCourse table consist StudentID and CourseId where both StudentId and CourseId is composite key (combined primary key).



Now let's see code to add these entities into DBContext:

```
//Create student entity
    var student1 = new Student();
    student1.StudentName = "New Student2";

    //Create course entities
    var course1 = new Course();
    course1.CourseName = "New Course1";
    course1.Location = "City1";

    var course2 = new Course();
    course2.CourseName = "New Course2";
    course2.Location = "City2";

    var course3 = new Course();
    course3.CourseName = "New Course3";
    course3.Location = "City1";

    // add multiple courses for student entity
    student1.Courses.Add(course1);
    student1.Courses.Add(course2);
    student1.Courses.Add(course3);

    using (var dbCtx = new SchoolDBEntities())
    {
        //add student into DBContext
        dbCtx.Students.Add(student1);
        //call SaveChanges
        dbCtx.SaveChanges();
    }
```

SaveChanges results in seven inserts query, 1 for student, 3 for Course and 3 for StudentCourse table.

Alternatively, we can also add Student entity into DBContext.Entry and mark it as Added which result in same insert query:

dbCtx.Entry(student1).State = System.Data.EntityState.Added;

Thus, we have just added Student entity into DBContext and it has inserted Studnet as well as Course information into respective database tables including StudentCourse table. We don't need to add Course entity into DBContext separately because Student and Course entity has Many-to-Many relationship.