# *LINQ and Entity Framework*

# Entity Framework

- Writing and managing ADO.Net code for data access is a tedious and monotonous job.
- Microsoft has provided an O/RM framework called "Entity Framework" to automate database related activities for your application.
- ADO.NET entity is an ORM (object relational mapping) which creates a higher abstract object model over ADO.NET components. So rather than getting into dataset, datatables, command, and connection objects as shown in the below code, you work on higher level domain objects like customers, suppliers, etc.

# Entity Framework

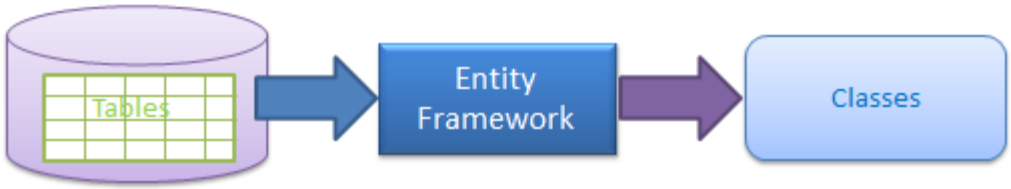
```
DataTable table = adoDs.Tables[0];
for (int j = 0; j < table.Rows.Count; j++)
{
   DataRow row = table.Rows[j];

   // Get the values of the fields
   string CustomerName =
      (string)row["Customername"];
   string CustomerCode =
      (string)row["CustomerCode"];
}
```

# Entity Framework

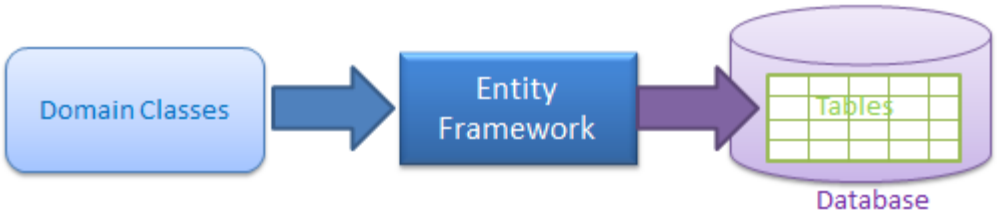
Below is the code for Entity Framework in which we are working on higher level domain objects like customer rather than with base level ADO.NET components (like dataset, datareader, command, connection objects, etc.).

```
foreach (Customer objCust in obj.Customers)
{                    }
```

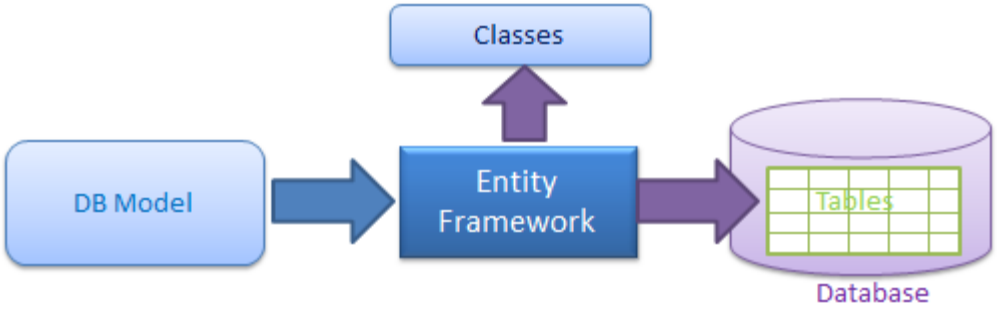
# Entity Framework Scenarios




Generate Data Access Classes for Existing Database

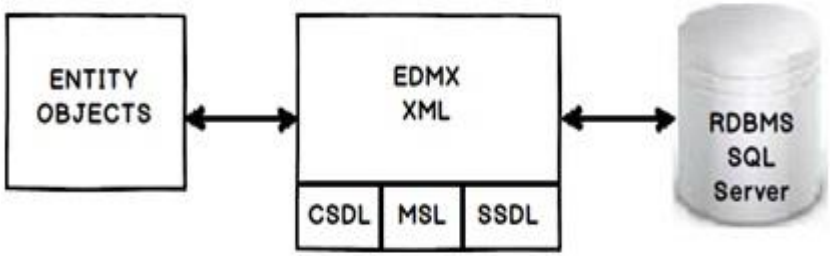



Create Database from the Domain Classes



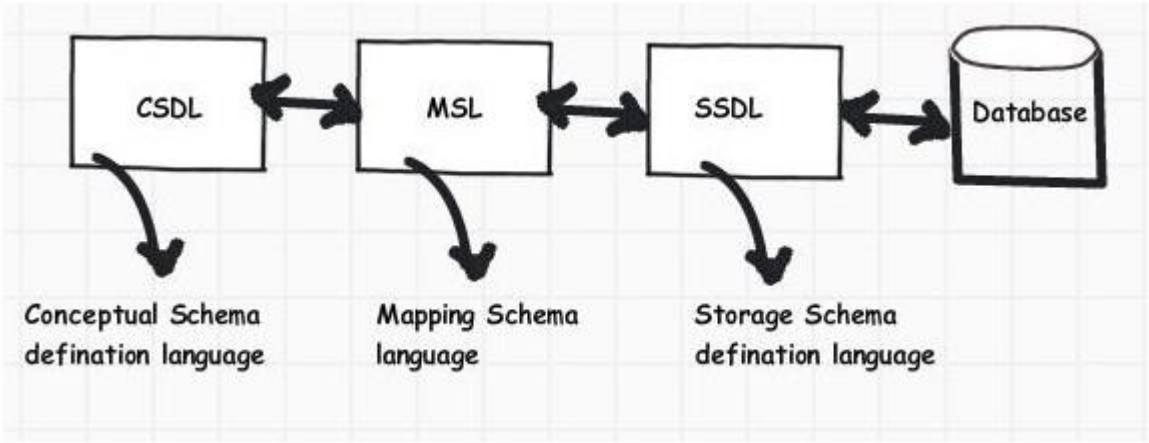

Create Database and Classes from the DB Model design

# EDMX file




•CSDL (Conceptual Schema definition language) is the conceptual abstraction which is exposed to the application.
•SSDL (Storage Schema Definition Language) defines the mapping with your RDBMS data structure.
•MSL (Mapping Schema Language) connects the CSDL and SSDL.

# EDMX file


CSDL
MSL
SSDL
Database
Conceptual Schema defination language
Mapping Schema language
Storage Schema defination language

# LINQ to Entities

LINQ Method syntax:

```
//Querying with LINQ to Entities
 using (var context = new SchoolDBEntities())
 {
    var L2EQuery = context.Students.where(s => s.StudentName == "Bill");

    var student = L2EQuery.FirstOrDefault<Student>();

 }
```

LINQ Query syntax:

```
 using (var context = new SchoolDBEntities())
 {
    var L2EQuery = from st in context.Students
                   where st.StudentName == "Bill"
                   select st;

    var student = L2EQuery.FirstOrDefault<Student>();
  }
```

# Entity SQL

```
//Querying with Object Services and Entity SQL
   string sqlString = "SELECT VALUE st FROM SchoolDBEntities.Students " +
                      "AS st WHERE st.StudentName == 'Bill'";

   var objctx = (ctx as IObjectContextAdapter).ObjectContext;

   ObjectQuery<Student> student = objctx.CreateQuery<Student>(sqlString);
               Student newStudent = student.First<Student>();
```

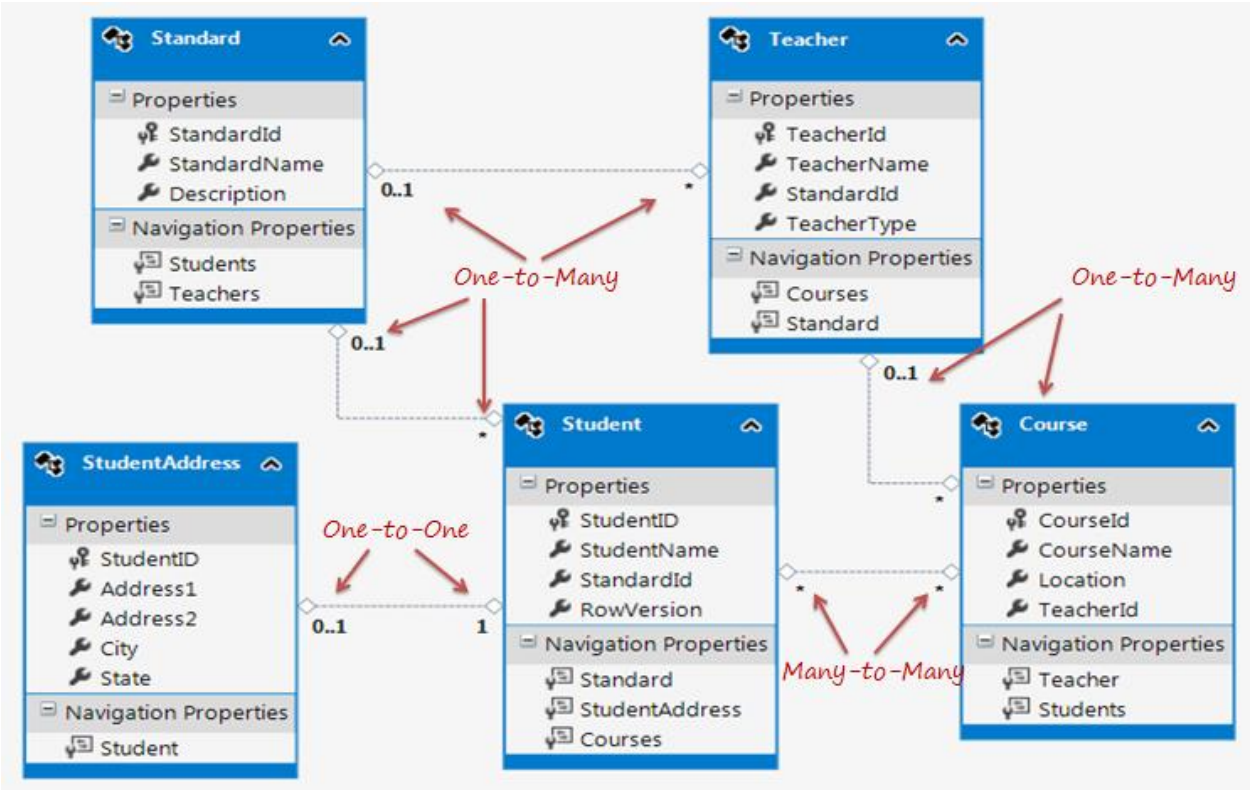
You can also use EntityConnection and EntityCommand to execute Entity SQL as shown below:

```
   using (var con = new EntityConnection("name=SchoolDBEntities"))
   {
      con.Open();
      EntityCommand cmd = con.CreateCommand();
      cmd.CommandText = "SELECT VALUE st FROM SchoolDBEntities.Students as st where st.StudentName='Bill'";
      Dictionary<int, string> dict = new Dictionary<int, string>();
      using (EntityDataReader rdr = cmd.ExecuteReader(CommandBehavior.SequentialAccess | CommandBehavior.CloseConnection))
      {
            while (rdr.Read())
            {
               int a = rdr.GetInt32(0);
               var b = rdr.GetString(1);
               dict.Add(a, b);
            }
      }
   }
```

# Native SQL

```
using (var ctx = new SchoolDBEntities())
   {
      var studentName = ctx.Students.SqlQuery("Select studentid,
studentname, standardId from Student where
studentname='Bill'").FirstOrDefault<Student>();
   }
```

# School Database Entity Data Model



# Linq-to-Entities Queries

**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.

# Linq-to-Entities Queries

**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.

# Linq-to-Entities Queries

**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>();
 }
```

# Linq-to-Entities Queries

**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;
}
```

# Linq-to-Entities Queries

**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;
}
```

# Linq-to-Entities Queries

**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
                           };
}
```

# Linq-to-Entities Queries

**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 | |

# 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.

# DBEntityEntry Class

```
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);
```

# DBEntityEntry Class

```
//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);
      }
   }
```

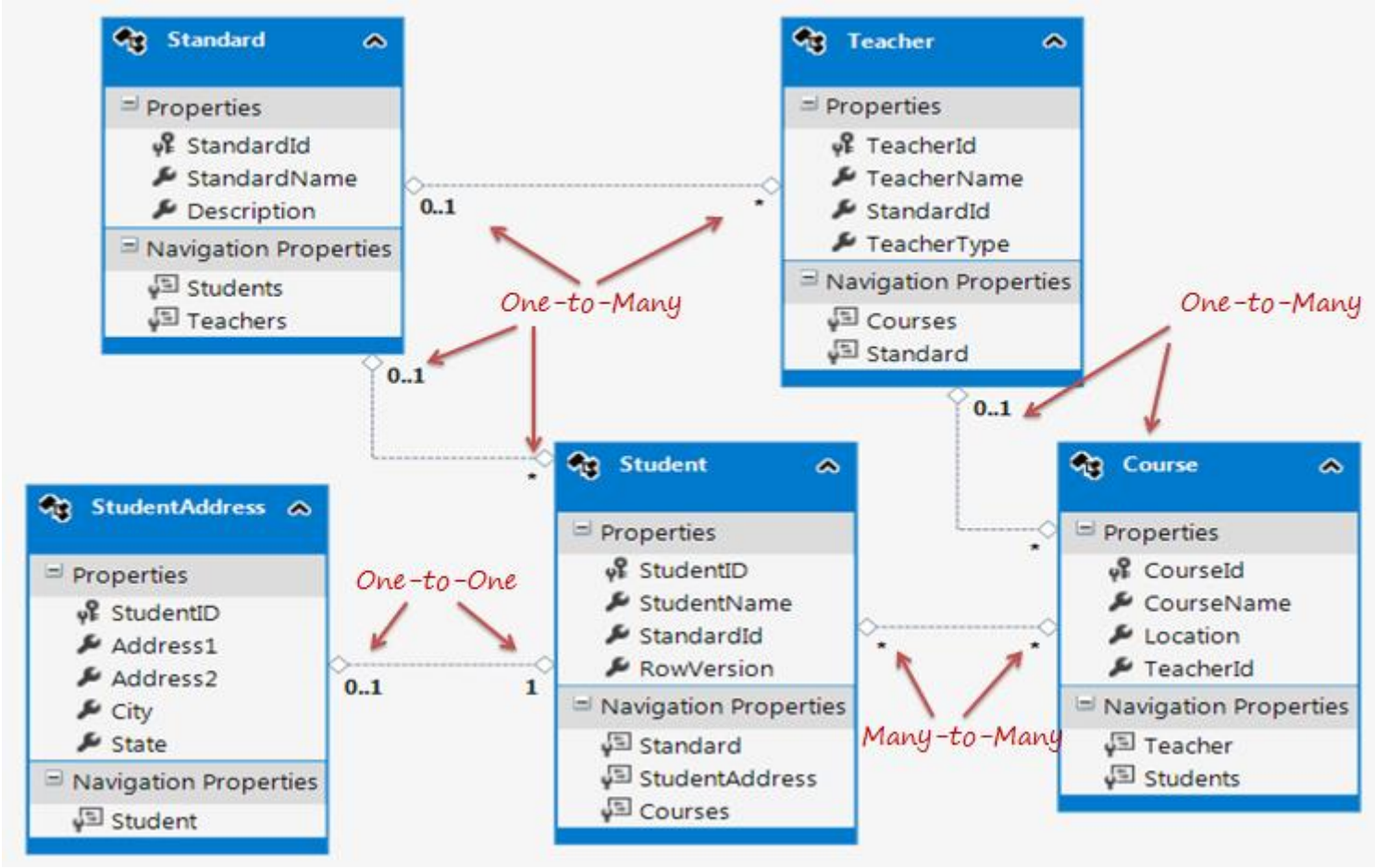
# DBEntityEntry Class

**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;
```

# SchoolDB Database ADO.NET Entity Model



# Entity Lifecycle

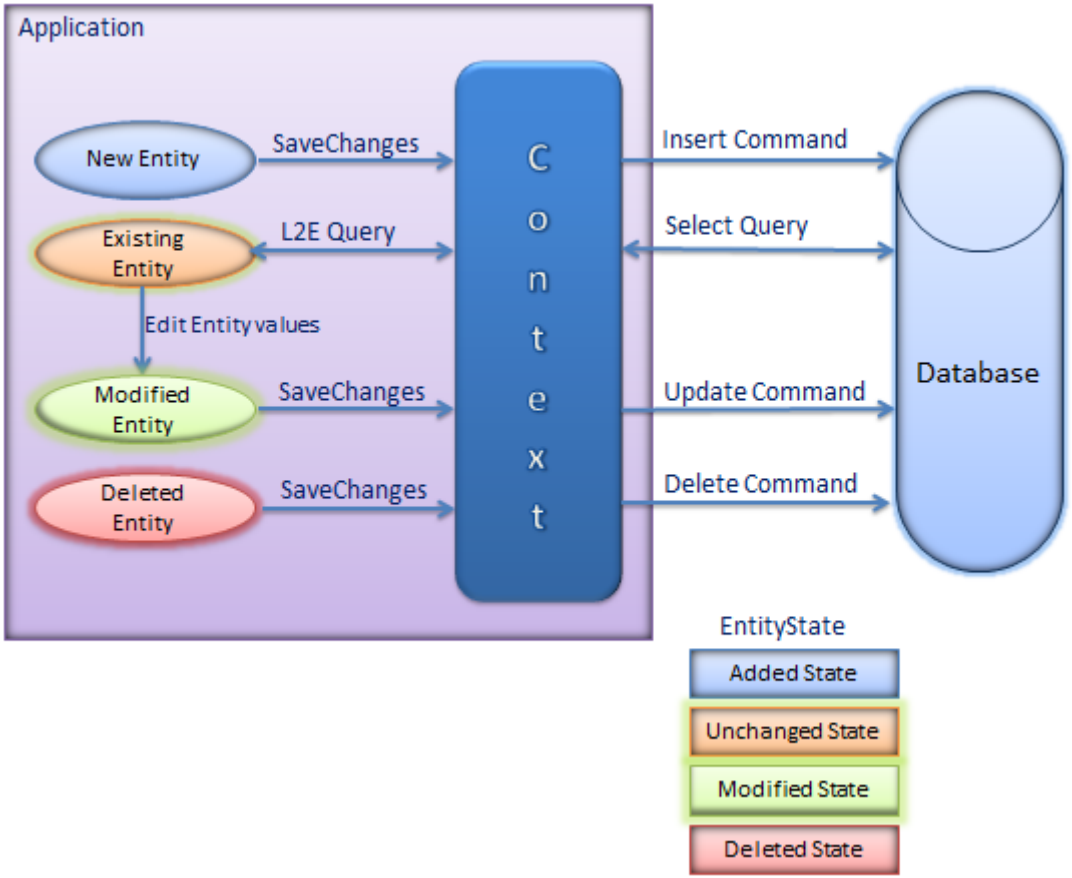
During an entity's lifetime, each entity has an entity state based on the operation performed on it via the context (DbContext). The entity state is an enum of type *System.Data.Entity.EntityState* that includes the following values:

- Added
- Deleted
- Modified
- Unchanged
- Detached

# Entity Lifecycle

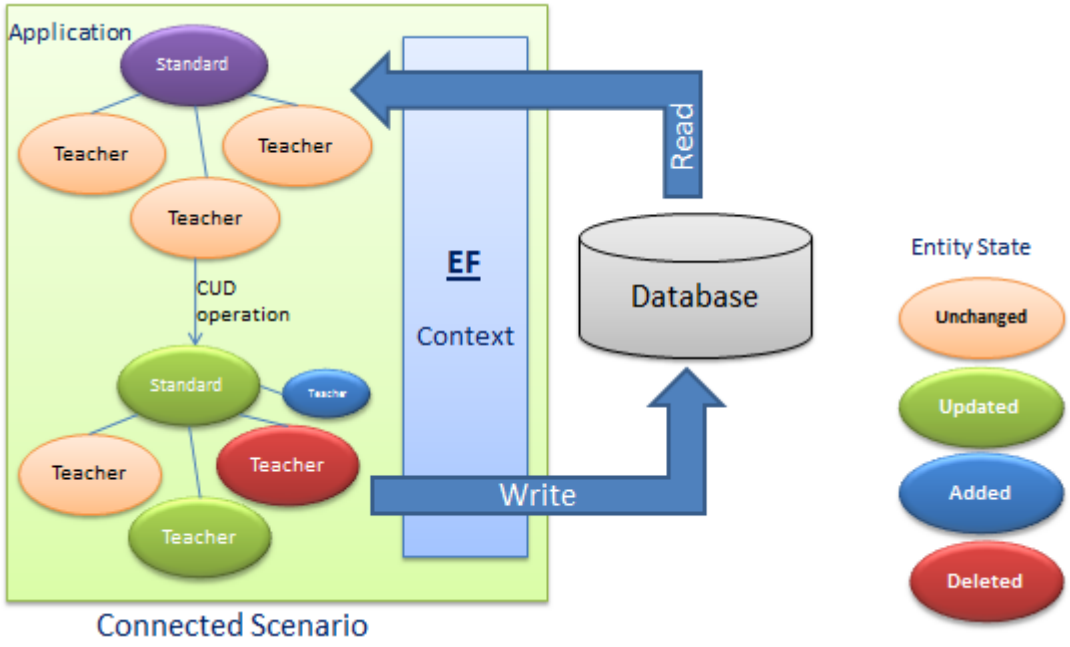
- The Context not only holds the reference to all the objects retrieved from the database but also it holds the entity states and maintains modifications made to the properties of the entity. This feature is known as *Change Tracking*.
- The change in entity state from the Unchanged to the Modified state is the only state that's automatically handled by the context. All other changes must be made explicitly using proper methods of DbContext and DbSet.
- The following figure illustrates how the operation performed on entity changes its' states which, in turn, affects database operation.

# Entity Lifecycle



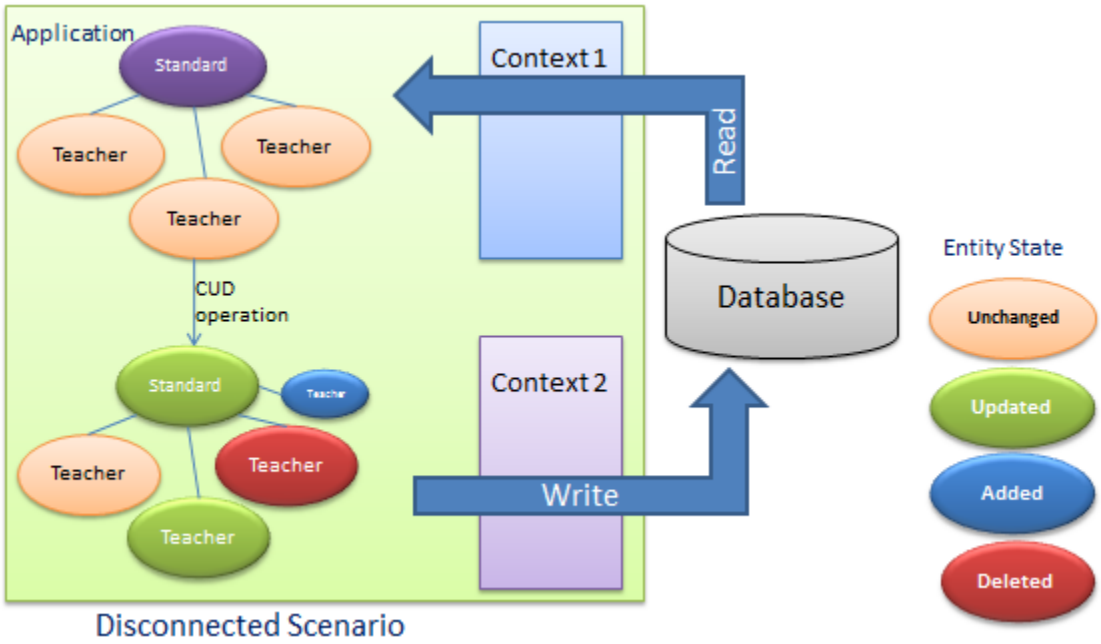
# Persistence in Entity Framework

- 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.



# Persistence in Entity Framework

- **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:



# Persistence in Entity Framework

- 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 previous figure, 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.

# CRUD Operation in Connected Scenario

- CRUD operation in connected scenario is a fairly easy task because the context automatically tracks the changes that happened in the entity during its lifetime, provided AutoDetectChangesEnabled is true, by default.
- The following example shows how you can add, update, and delete an entity in the connected scenario

```
using (var context = new SchoolDBEntities())
{
    var studentList = context.Students.ToList<Student>();

    //Perform create operation
    context.Students.Add(new Student() { StudentName = "New Student" });
```

# CRUD Operation in Connected Scenario

```
//Perform Update operation
   Student studentToUpdate = studentList.Where(s => s.StudentName
== "student1").FirstOrDefault<Student>();
   studentToUpdate.StudentName = "Edited student1";

   //Perform delete operation
   context.Students.Remove(studentList.ElementAt<Student>(0));

   //Execute Inser, Update & Delete queries in the database
   context.SaveChanges();
}
```

# CRUD Operation in Connected Scenario

**Note:** If *context.Configuration.AutoDetectChangesEnabled = false* then context cannot detect changes made to existing entities so do not execute update query. You need to call context.ChangeTracker.DetectChanges() before SaveChanges() in order to detect the edited entities and mark their status as Modified.

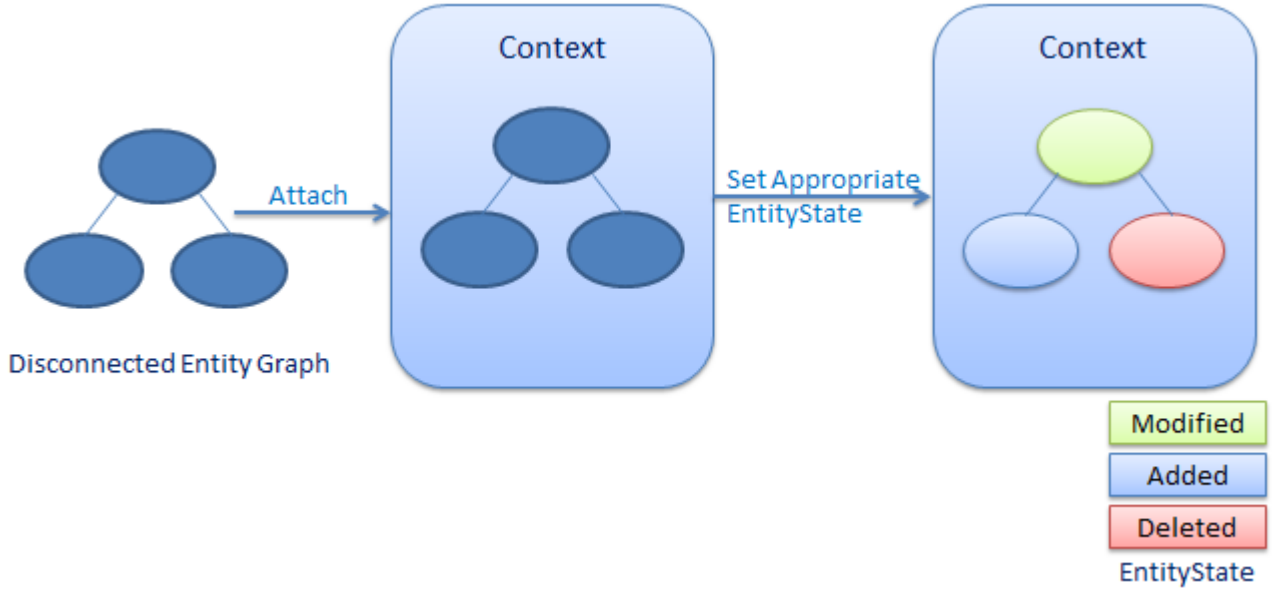
- Context detects adding and deleting entity, when the operation is performed only on **DbSet**. If you perform add and delete entity on a separate collection or list, then it won't detect these changes.
- The following code will **NOT** insert or delete student. It will only update the student entity because we are adding and deleting entities from the List and not from DbSet.

# CRUD Operation in Connected Scenario

```
    using (var context = new SchoolDBEntities())
{    var studentList = context.Students.ToList<Student>();
    //Add student in list
    studentList.Add(new Student() { StudentName = "New Student" });
    //Perform update operation
    Student studentToUpdate = studentList.Where(s => s.StudentName
== "Student1").FirstOrDefault<Student>();
    studentToUpdate.StudentName = "Edited student1";
    //Delete student from list
    if (studentList.Count > 0)
        studentList.Remove(studentList.ElementAt<Student>(0));
    //SaveChanges will only do update operation not add and delete
    context.SaveChanges();
}
```

# Disconnected Entities

There are two things we need to do when we get a disconnected entity graph or even a single disconnected entity. First, we need to attach entities with the new context instance and make context aware about these entities. Second, set appropriate EntityStates to these entities manually because the new context instance doesn't know anything about the operations performed on the disconnected entities, so the new context cannot apply the appropriate EntityState.



# Disconnected Entities

**DbSet.Add():**

DbSet.Add() method attaches the entire entity graph to the new context and automatically applies Added entity state to all the entities.

```
  //disconnected entity graph
    Student disconnectedStudent = new Student() { StudentName = "New Student" };
    disconnectedStudent.StudentAddress = new StudentAddress() { Address1 = "Address", City = "City1" };
    using (var ctx = new SchoolDBEntities())
    {       //add disconnected Student entity graph to new context instance - ctx
        ctx.Students.Add(disconnectedStudent);
        // get DbEntityEntry instance to check the EntityState of specified entity
        var studentEntry = ctx.Entry(disconnectedStudent);
        var addressEntry = ctx.Entry(disconnectedStudent.StudentAddress);
```

# Disconnected Entities

```
Console.WriteLine("Student EntityState: {0}",studentEntry.State);

    Console.WriteLine("StudentAddress EntityState: {0}",addressEntry.State);
}
```

**Output**:

Student EntityState: Added

StudentAddress EntityState: Added

Use Add method of parent DbSet entity to attach the entire entity graph to the new context instance with Added state to each entity. This will execute insert command for all the entities, which will insert new rows in the appropriate database table.

# Disconnected Entities

**DbSet.Attach():**

- DbSet.Attach method attaches a whole entity graph to the new context with Unchanged entity state.

```
//disconnected entity graph
   Student disconnectedStudent = new Student() { StudentName = "New Student" };
   disconnectedStudent.StudentAddress = new StudentAddress() { Address1 = "Address", City = "City1" };
   using (var ctx = new SchoolDBEntities())
   {  //attach disconnected Student entity graph to new context instance - ctx
      ctx.Students.Attach(disconnectedStudent);
      // get DbEntityEntry instance to check the EntityState of specified entity
      var studentEntry = ctx.Entry(disconnectedStudent);
      var addressEntry = ctx.Entry(disconnectedStudent.StudentAddress);
```

# Disconnected Entities

```
    Console.WriteLine("Student EntityState: {0}",studentEntry.State);
    Console.WriteLine("StudentAddress EntityState: {0}",addressEntry.State);
}
```

**Output**:

Student EntityState: Unchanged

StudentAddress EntityState: Unchanged

- As per the above code, we can attach disconnected entity graph using DbSet.Attach method. This will attach the entire entity graph to the new context with Unchanged entity state to all entities.
- Thus, Attach method will only attach entity graph to the context, so we need to find the appropriate entity state for each entity and apply it manually.

# Disconnected Entities

**DbContext.Entry()**

Entry method of DbContext returns DbEntityEntry instance for a specified entity. DbEntityEntry can be used to change the state of an entity.

DbContext.Entry(disconnectedEntity).state = EntityState.Added/Modified/Deleted/Unchanged

This method attaches a whole entity graph to the context with specified state to the parent entity and set the state of other entities, as shown in the foll

owing table.

| Parent Entity State | Entity State of child entities |
|---|---|
| Added | Added |
| Modified | Unchanged |
| Deleted | All child entities will be null |

# Disconnected Entities

**Add single Student entity (not entity graph)**

```
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();
        } } }
```

# Disconnected Entities

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;
```

# Disconnected Entities

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

# Disconnected Entities

Update a single Student entity (not entity graph)

```
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";
}
```

# Disconnected Entities

```
//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();
}
```

1. As you see in the above code snippet, we are doing the following steps:
2. Get the existing student from DB.
3. Change the student name out of Context scope (disconnected mode)
4. Pass the modified entity into the Entry method to get its DBEntityEntry object and then mark its state as Modified
5. Call SaveChanges() method to update student information into the database.

# Disconnected Entities

```
//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();
}
```

1. As you see in the above code snippet, we are doing the following steps:
2. Get the existing student from DB.
3. Change the student name out of Context scope (disconnected mode)
4. Pass the modified entity into the Entry method to get its DBEntityEntry object and then mark its state as Modified
5. Call SaveChanges() method to update student information into the database.

# Disconnected Entities

**Delete Entity using DBContext in Disconnected**

```
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();
}
```

# Entity states and SaveChanges

An entity can be in one of five states as defined by the EntityState enumeration. These states are:

- Added: the entity is being tracked by the context but does not yet exist in the database
- Unchanged: the entity is being tracked by the context and exists in the database, and its property values have not changed from the values in the database
- Modified: the entity is being tracked by the context and exists in the database, and some or all of its property values have been modified
- Deleted: the entity is being tracked by the context and exists in the database, but has been marked for deletion from the database the next time SaveChanges is called
- Detached: the entity is not being tracked by the context

# Entity states and SaveChanges

SaveChanges does different things for entities in different states:

- Unchanged entities are not touched by SaveChanges. Updates are not sent to the database for entities in the Unchanged state.
- Added entities are inserted into the database and then become Unchanged when SaveChanges returns.
- Modified entities are updated in the database and then become Unchanged when SaveChanges returns.
- Deleted entities are deleted from the database and are then detached from the context.