

Chapitre 2

Accès aux données De ADO.Net à l'ORM Entity Framework



Saber BHAR

INSAT 2019 – GL3

Contenu

- **Introduction à l'accès aux données**
- **Les patterns « Repository » et « Unit Of Work »**
- **Retour sur l'infrastructure ADO.NET**
 - ADO.NET en mode connecté : Les DataReaders
 - ADO.NET en mode connecté : Les DataSets
- **Accès aux données avec Entity Framework (EF)**
 - ORM et Opérations CRUD
 - Gestion de la concurrence d'accès avec EF
 - Optimisation des performances de EF

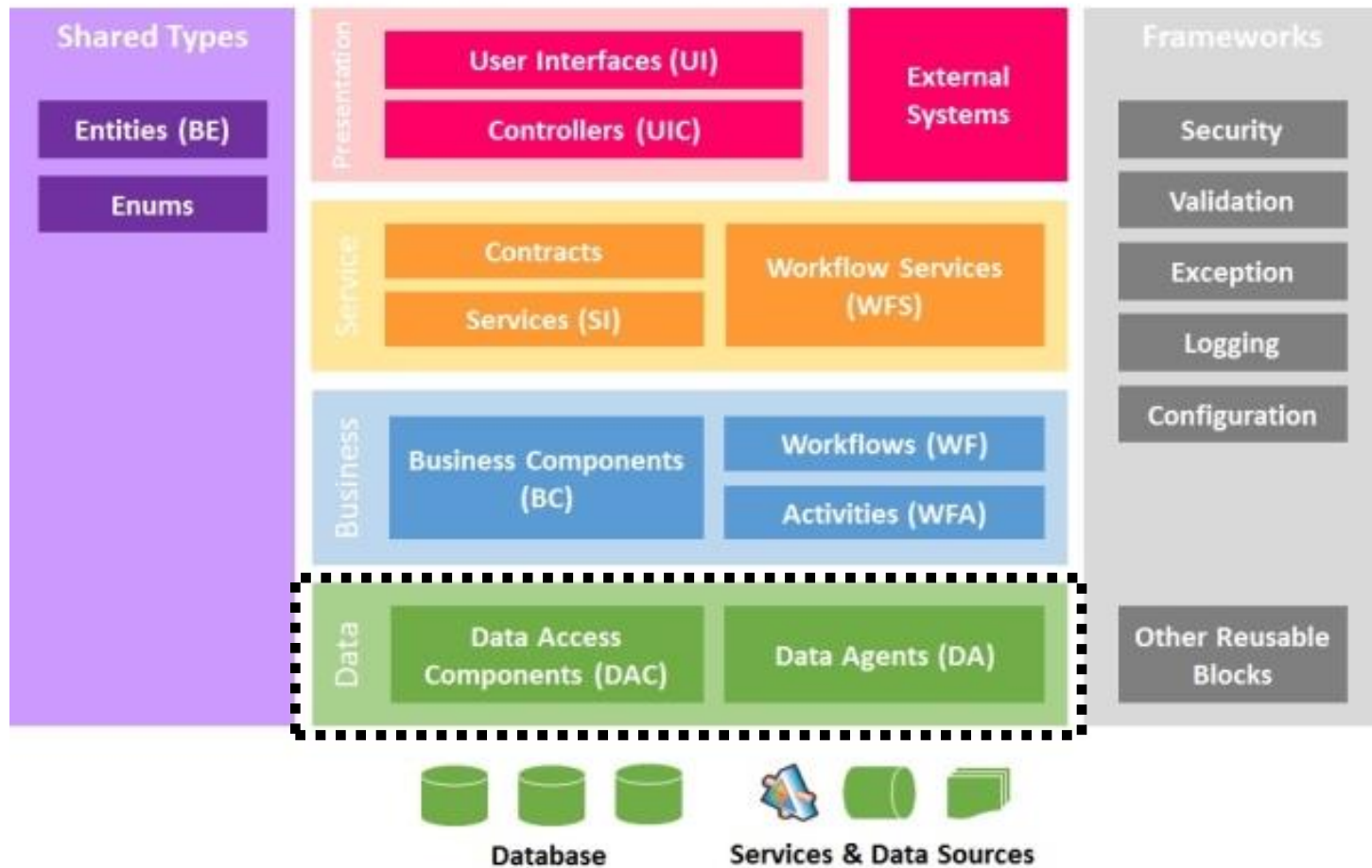
Introduction à l'accès aux données

Historique de l'accès aux données

- **L'objectif de l'API ADO.NET est de fournir un accès universel aux données et ce pour n'importe quelle base de données et à partir de n'importe quelle langue.**
- **Historique**
 - La toute première version de .NET Framework a inclut l'API ADO.NET
 - Assembly : System.Data
 - ADO.NET proposait les providers ODBC, OLE-DB, SqlConnection (pour SQL Server) et OracleClient
 - La version ADO.NET 3.5 a connu des améliorations importantes
 - Linq to SQL : une véritable solution ORM mais difficilement portable sur d'autres bases de données autres que SQL Server
 - La version ADO.NET 4.0
 - Entity Framework 4.0 : un nouvelle solution ORM portable
 - La version ADO.NET 4.5
 - La version 5 de Entity Framework ne fait plus partie du .NET Framework. Rajouté exclusivement avec l'outil Nuget

Positionnement de l'accès aux données

Layered Architecture



The Repository and Unif of Work Patterns

Describing the Data Access Layer

- **The DAL is the layer in your application that is solely responsible for talking to the data store and persisting and retrieving your business objects.**
 - The DAL typically includes all the create, read, update, and delete (CRUD) methods, transaction management, data concurrency, as well as a querying mechanism to enable your business logic layer to retrieve objects for any given criteria.
 - The DAL should not contain business logic and should be accessed via the business logic layer through interfaces.
 - This adheres to the separation of concerns principle and ensures that the business layer remains unaware of the underlying data access implementation strategy.

The Repository Pattern

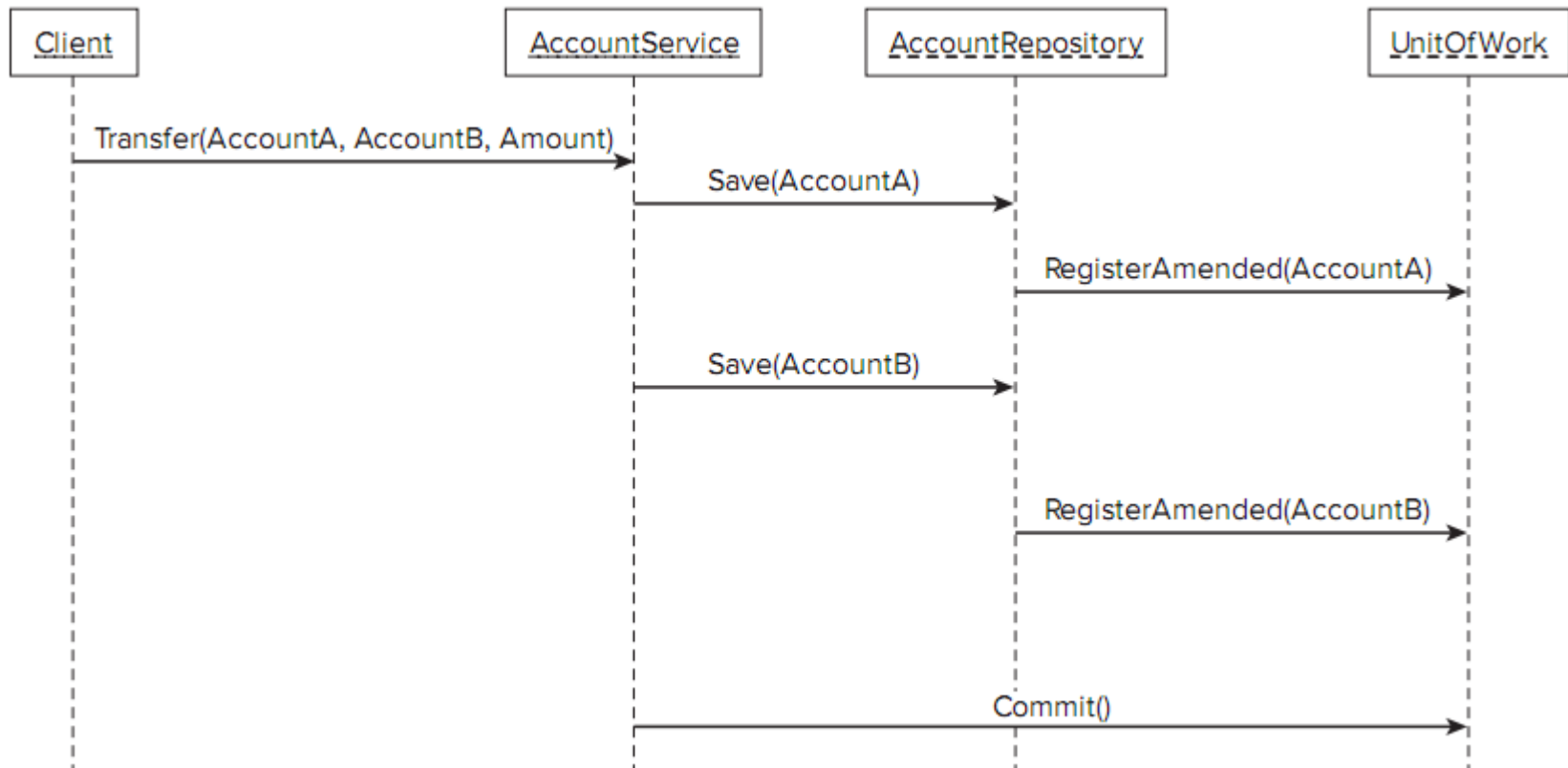
- With the advent of Language Integrated Query (LINQ) and the delayed execution model, Repositories can now expose an **IQueryable FindAll** method that allows the business layer to query a Repository directly, as in the code snippet that follows:

```
public interface IRepository<T>
{
    IQueryable<T> FindAll();
    T FindBy(Guid Id);
    void Add(T entity);
    void Save(T entity);
    void Remove(T entity);
}
```

- An IQueryable return type, however, is not universally viewed as such a good way to go when trying to keep persistence concerns out of your domain or business layer, because not all LINQ providers behave in the same manner or offer the same level of features.
- <https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>

Unit of Work

- The benefit of utilizing the Unit of Work in your DAL is to ensure **data integrity**; if an issue arises partway through persisting a series of business objects as part of a **transaction**, all changes should be rolled back to ensure that the data remains in a valid state.



Retour sur l'infrastructure d'accès aux données ADO.NET

ADO.NET Assembly et Namespaces

Assembly

- System.Data.dll

Namespaces:

- | | |
|----------------------------|------------------------------------|
| • System.Data | general types |
| • System.Data.Common | classes for implementing providers |
| • System.Data.OleDb | OLE DB provider |
| • System.Data.SqlClient | Microsoft SQL Server provider |
| • System.Data.SqlTypes | data types for SQL Server |
| • System.Data.Odbc | ODBC provider (since .NET 1.1) |
| • System.Data.OracleClient | Oracle provider (since .NET 1.1) |
| • System.Data.SqlServerCe | Compact Framework |

Program Pattern for Connection-oriented Data Access

```
//1. Chaîne de connexion
string connectionString =
@"Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=Northwind;Integrated Security=true";

//2. La requête sous forme de chaîne
string queryString =
"SELECT ProductID, UnitPrice, ProductName from dbo.products "
+ "WHERE UnitPrice > @pricePoint "
+ "ORDER BY UnitPrice DESC;";

//3. Créer une connexion
using (SqlConnection connection = new SqlConnection(connectionString))
{
    //4. Créer une commande
    SqlCommand command = new SqlCommand(queryString, connection);
    command.Parameters.AddWithValue("@pricePoint", 5);
    try
    {
        //5. Ouvrir une connexion
        connection.Open();
        //6. Résultat sous forme de reader // SI plusieurs résultats
        SqlDataReader reader = command.ExecuteReader();

        //7. Parcourir la liste
        while (reader.Read())
        {
            Console.WriteLine("\t{0}\t{1}\t{2}",
                reader[0], reader[1], reader[2]);
        }
        reader.Close();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        //8. Fermer connexion
        connection.Close();
    }
}
```

Class hierarchy

■ General interfaces

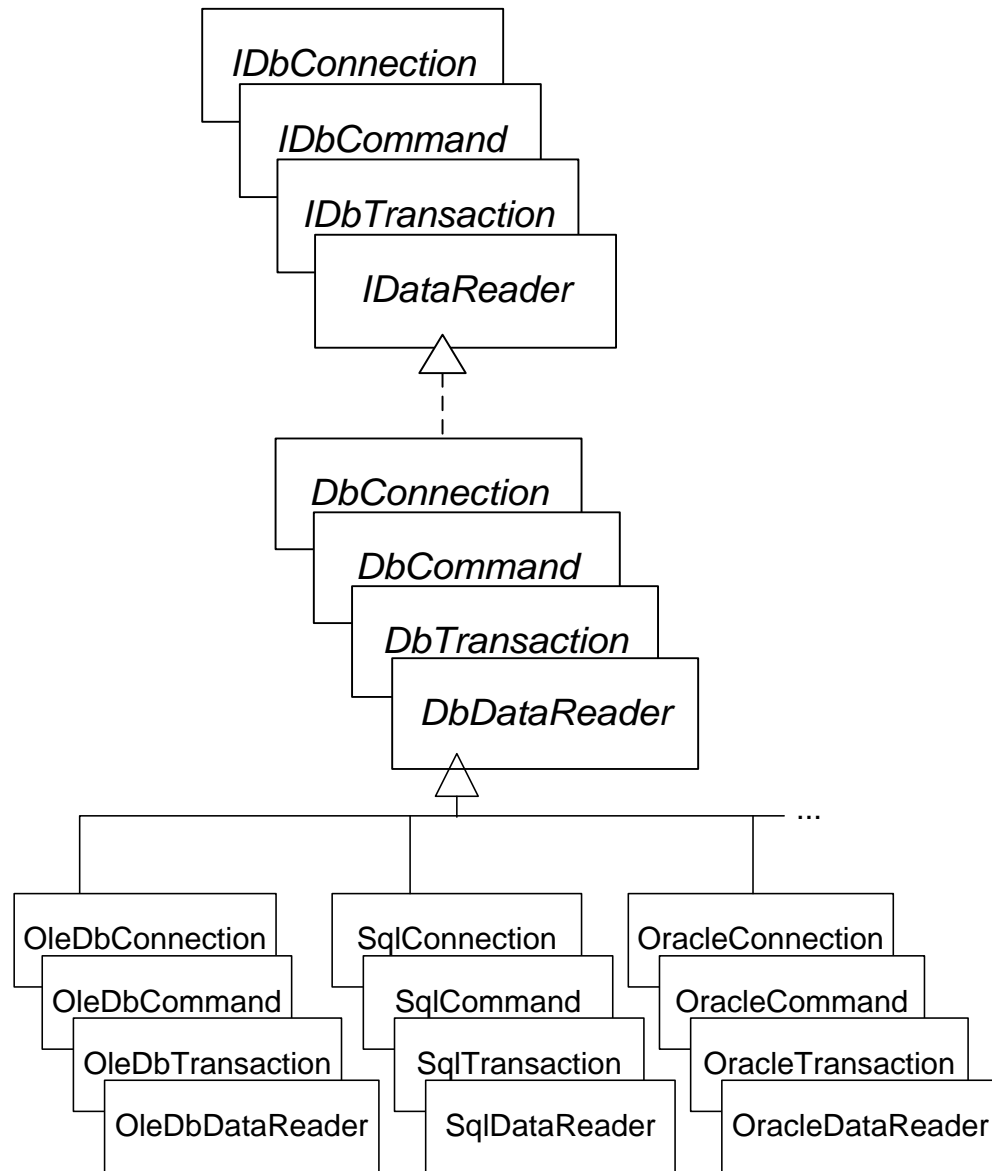
IDbConnection
IDbCommand
IDbTransaction
IDataReader

■ Abstract base classes

DbConnection
DbCommand
DbTransaction
DbDataReader

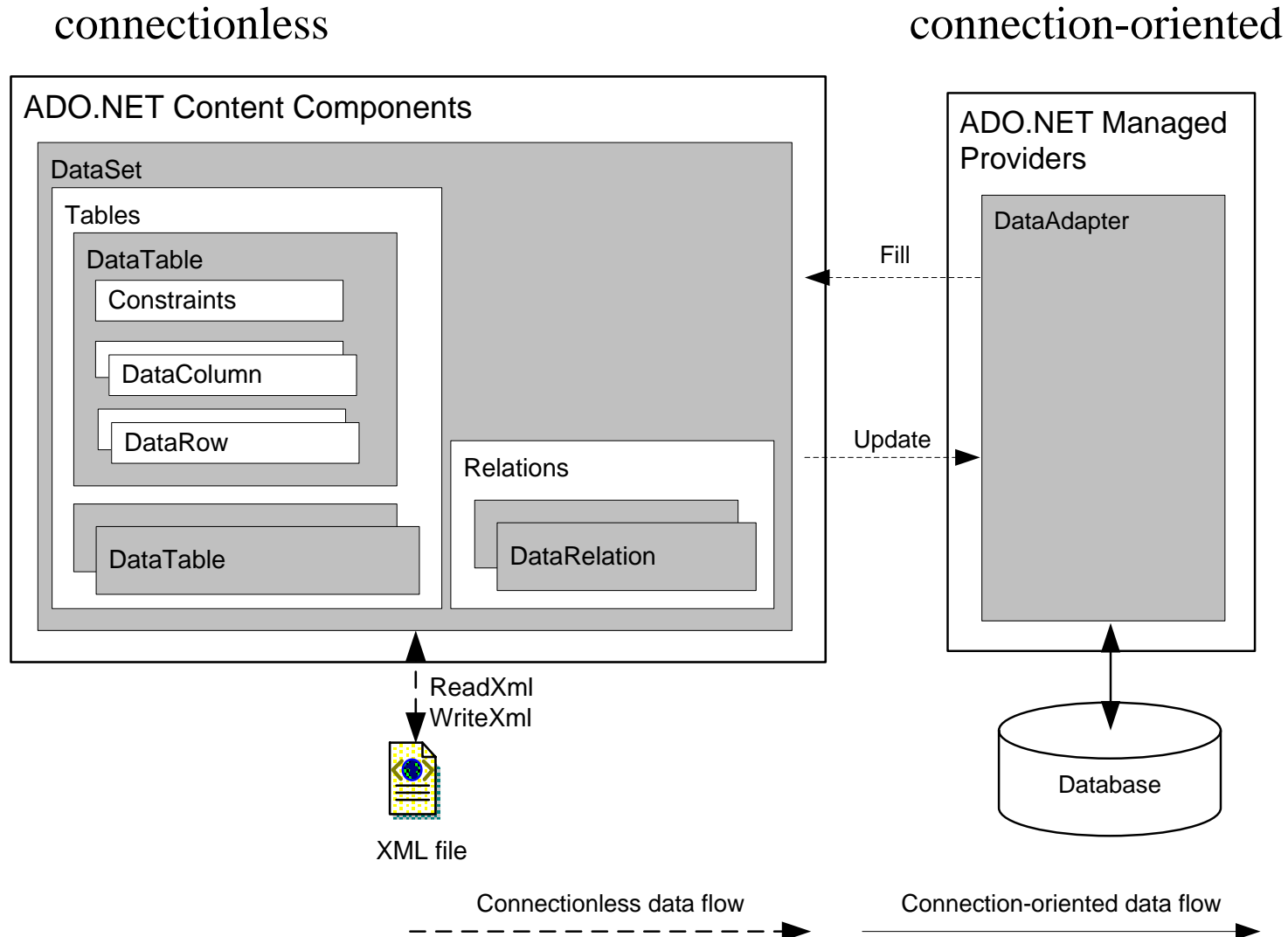
■ Special implementations

OleDb: implementation for OLEDB
Sql: implementation for SQL Server
Oracle: implementation for Oracle
Odbc: implementation for ODBC
SqlCe: implementation for SQL Server CE data base



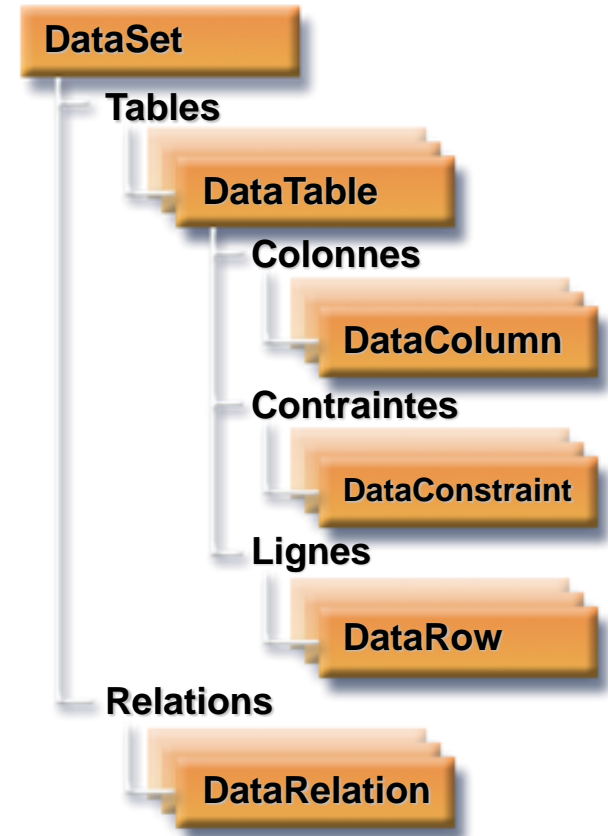
ADO.NET en mode déconnecté

Architecture of Connectionless Data Access



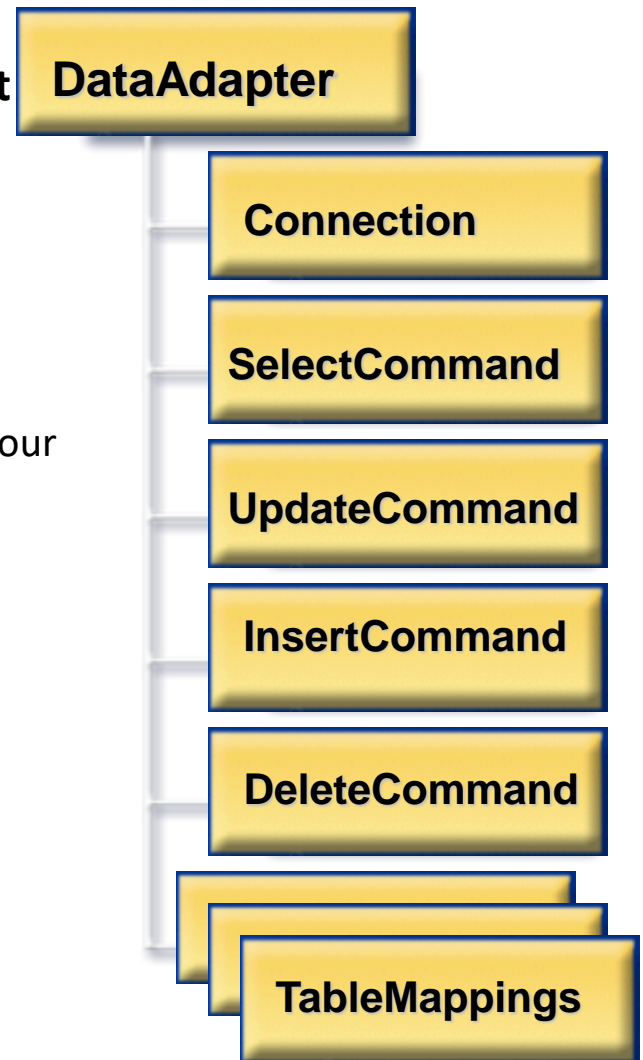
Le DataSet

- **Vue relationnelle des données**
 - Contient des tables, colonnes, lignes, contraintes, vues et relations
- **Modèle déconnecté**
 - N'est pas lié à la source de données
 - Accès par indice
 - Fortement typé
 - Liaison de données
 - Mise à jour/synchro par lots
 - Connexion à la source de données via DataAdapter



DataAdapter

- **Rôle : charger les données provenant d'une source et répercuter les modifications du DataSet**
 - Deux méthodes importantes :
 - **Fill** (Remplissage DataSet et Table)
 - **Update** (Mise à jour en base)
 - Définit les correspondances entre tables et colonnes
 - L'utilisateur peut spécifier des commandes explicites pour les INSERT/UPDATE/DELETE
 - ou des procédures stockées
 - Permet à un DataSet d'être rempli avec des données provenant de différentes sources



Exemple

```
// Assumes that customerConnection is a valid SqlConnection object.
// Assumes that orderConnection is a valid OleDbConnection object.
SqlDataAdapter custAdapter = new SqlDataAdapter(
    "SELECT * FROM dbo.Customers", customerConnection);
OleDbDataAdapter ordAdapter = new OleDbDataAdapter(
    "SELECT * FROM Orders", orderConnection);

DataSet customerOrders = new DataSet();

custAdapter.Fill(customerOrders, "Customers");
ordAdapter.Fill(customerOrders, "Orders");

DataRelation relation = customerOrders.Relations.Add("CustOrders",
    customerOrders.Tables["Customers"].Columns["CustomerID"],
    customerOrders.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow pRow in customerOrders.Tables["Customers"].Rows)
{
    Console.WriteLine(pRow["CustomerID"]);
    foreach (DataRow cRow in pRow.GetChildRows(relation))
        Console.WriteLine("\t" + cRow["OrderID"]);
}
```

```
private static void AdapterUpdate(string connectionString)
{
    using (SqlConnection connection =
        new SqlConnection(connectionString))
    {
        SqlDataAdapter dataAdapter = new SqlDataAdapter(
            "SELECT CategoryID, CategoryName FROM Categories",
            connection);

        dataAdapter.UpdateCommand = new SqlCommand(
            "UPDATE Categories SET CategoryName = @CategoryName " +
            "WHERE CategoryID = @CategoryID", connection);

        dataAdapter.UpdateCommand.Parameters.Add(
            "@CategoryName", SqlDbType.NVarChar, 15, "CategoryName");

        SqlParameter parameter = dataAdapter.UpdateCommand.Parameters.Add(
            "@CategoryID", SqlDbType.Int);
        parameter.SourceColumn = "CategoryID";
        parameter.SourceVersion = DataRowVersion.Original;

        DataTable categoryTable = new DataTable();
        dataAdapter.Fill(categoryTable);

        DataRow categoryRow = categoryTable.Rows[0];
        categoryRow["CategoryName"] = "New Beverages";

        dataAdapter.Update(categoryTable);

        Console.WriteLine("Rows after update.");
        foreach (DataRow row in categoryTable.Rows)
        {
            {
                Console.WriteLine("{0}: {1}", row[0], row[1]);
            }
        }
    }
}
```

ADO.Net Entity Framework

What is Entity Framework ?

- Entity Framework (EF) is an **object-relational mapper (ORM)** that enables .NET developers to work with relational data using domain-specific objects. It eliminates the need for most of the data-access code that developers usually need to write
 - Entity Framework is an open source software (EntityFramework.codeplex.com)
- **Object-Relational Mapping (ORM)** is a programming technique for automatic mapping and converting data between relational database tables and object-oriented classes and objects
 - ORM frameworks typically provide the following functionality:
 - Creating object model by database schema
 - Creating database schema by object model
 - Querying data by object-oriented API
 - Data manipulation operations : **CRUD** – create, retrieve/read, update, delete
 - ORM frameworks automatically generate SQL to perform the requested data operations

Entity Framework history

EntityFramework Version	Introduced Features
EF 3.5	Basic O/RM support with Database First approach.
EF 4.0	POCO Support, Lazy loading, testability improvements, customizable code generation and the Model First approach.
EF 4.1	First to available of NuGet package, Simplified DbContext API overObjectContext, Code First approach. EF 4.1.1 patch released with bug fixing of 4.1.
EF 4.3	Code First Migrations feature that allows a database created by Code First to be incrementally changed as your Code First model evolves. EF 4.3.1 patch released with bug fixing of EF 4.3.
EF 5.0	Announced EF as Open Source. Introduced Enum support, table-valued functions, spatial data types, multiple-diagrams per model, coloring of shapes on the design surface and batch import of stored procedures, EF Power Tools and various performance improvements.
EF 6.0 (the current release)	EF 6.0\6.1 is latest release of Entity Framework. It includes many new features related to Code First & EF designer like asynchronous query & save, connection Resiliency, dependency resolution etc.

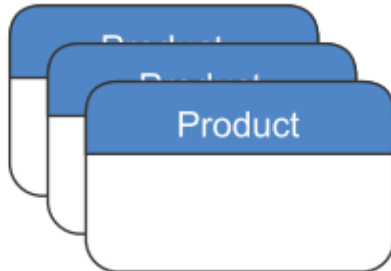
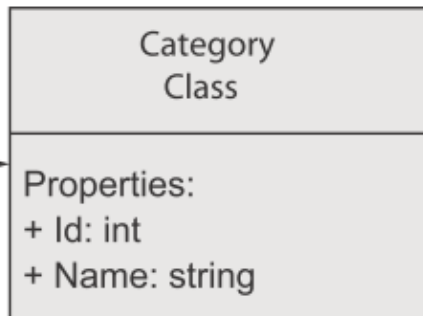
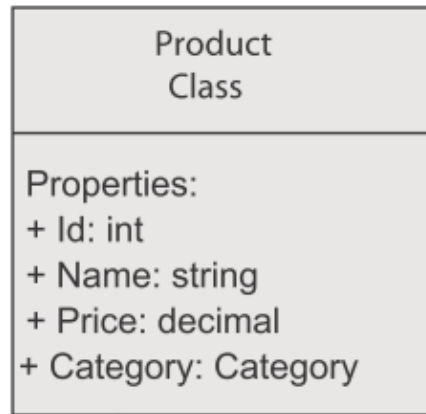
ORM Advantages

- **Developer productivity**
 - Writing less code
- **Abstract from differences between object and relational world**
 - Complexity hidden within ORM
- **Manageability of the CRUD operations for complex relationships**
- **Easier maintainability**

The Entity Framework ORM



.NET Application



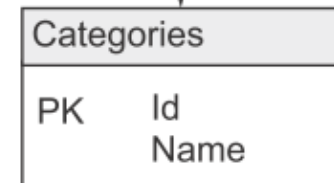
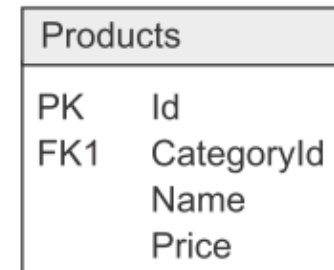
.NET classes map to tables and properties map to columns.

References between types map to foreign key relationships between tables.

Each object (an instance of a class) maps to a row in a table.

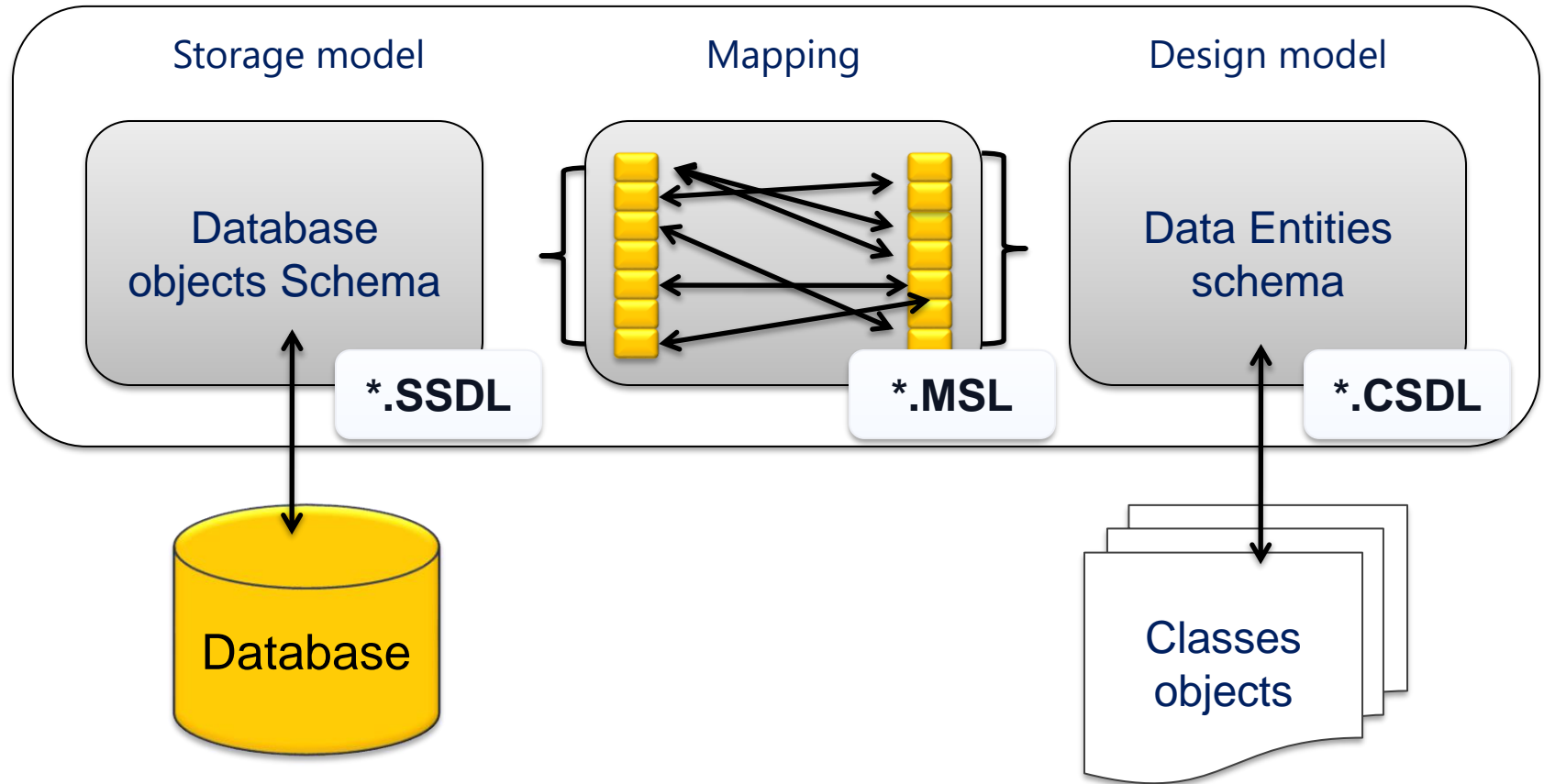


Database



Products			
Id	CategoryId	Name	Price
1	12	Mac mini	479.00
2	45	iPad	339.00
3	123	Xbox One	280.00

The Entity Framework ORM

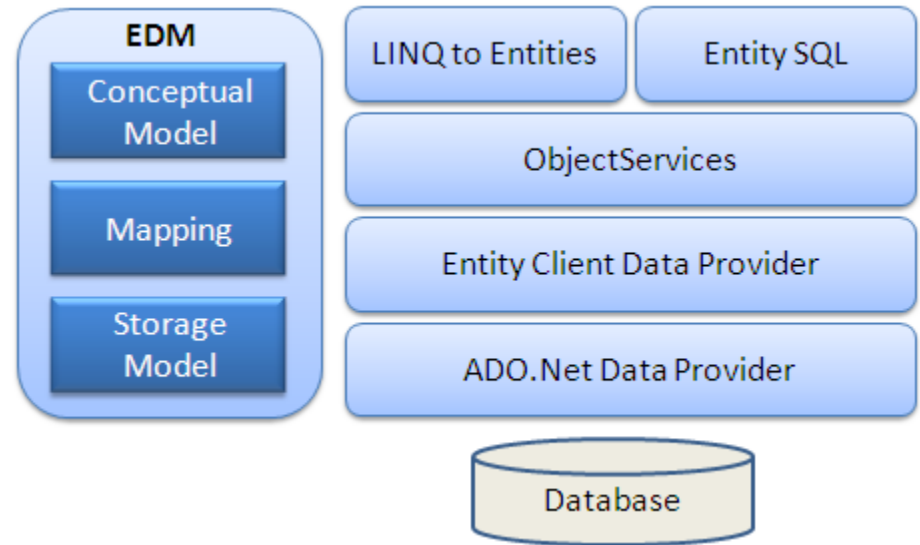


SSDL : Store Schema Definition Language
MSL : Mapping Schema Language
CSDL : Conceptual Schema Definition Language

} XML

Entity Framework Architecture

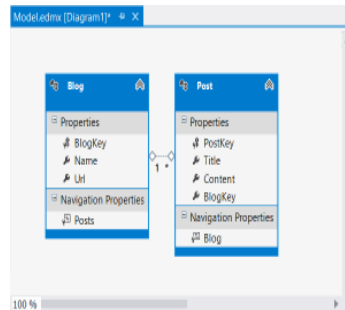
- **EDM (Entity Data Model):** EDM consist three main parts- Conceptual model, Mapping and Storage model.
- **LINQ to Entities:** LINQ to Entities is a query language used to write queries against the object model. It returns entities, which are defined in the conceptual model. You can use your LINQ skills here.
- **Entity SQL:** Entity SQL is another query language just like LINQ to Entities. However, it is a little more difficult than L2E and also the developer will need to learn it separately.



Development workflows with Entity framework

Developer Workflows

Designer Centric



Code Centric

```
public class Blog
{
    public int BlogKey { get; set; }
    public string Name { get; set; }
    public string Url { get; set; }

    public virtual List<Post> Posts {
    }
}

public class Post
```



New
Database

Model First

Create model in EF Designer
Generate database from model
Classes auto-generated from model

Code First

Define classes and mapping in code
Database created from code
Migrations apply model changes to database



Existing
Database

Database First

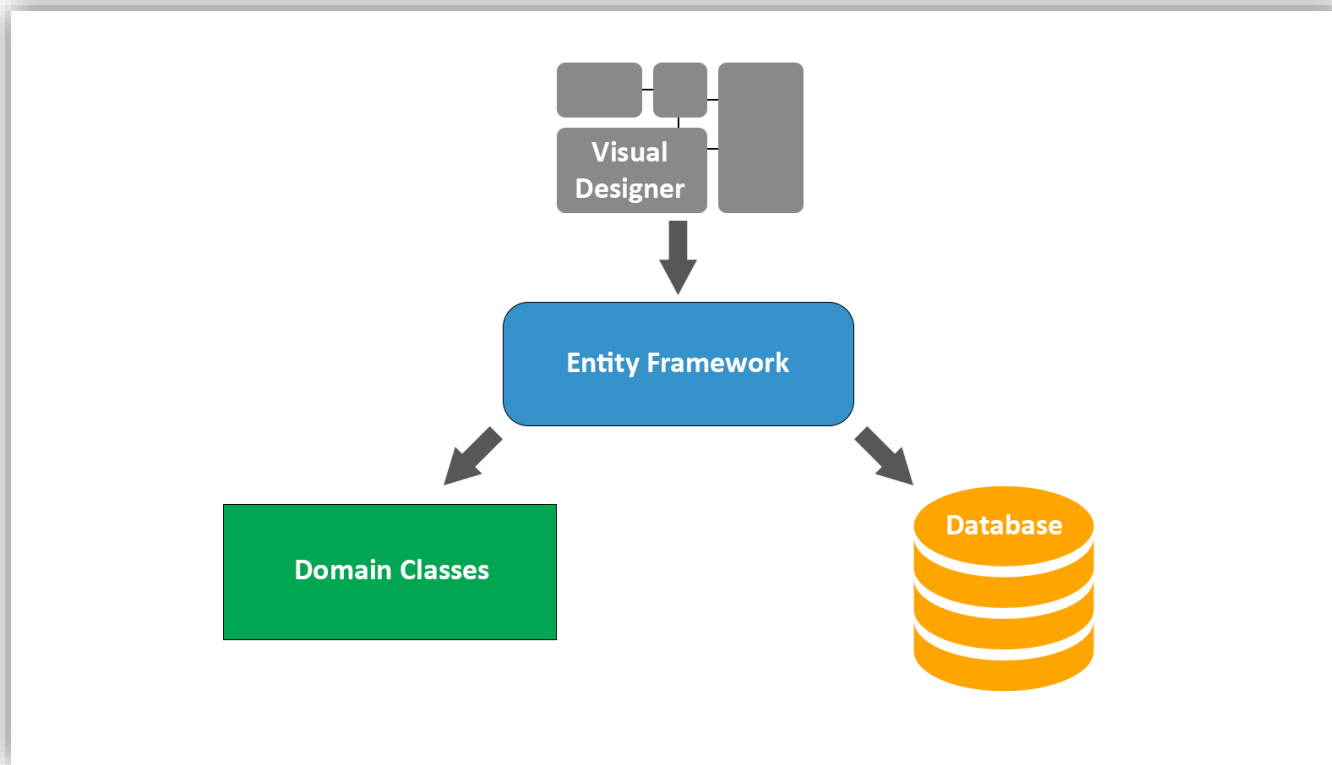
Reverse engineer model in EF Designer
Classes auto-generated from model

Code First

Define classes and mapping in code
EF Power Tools provide reverse engineer

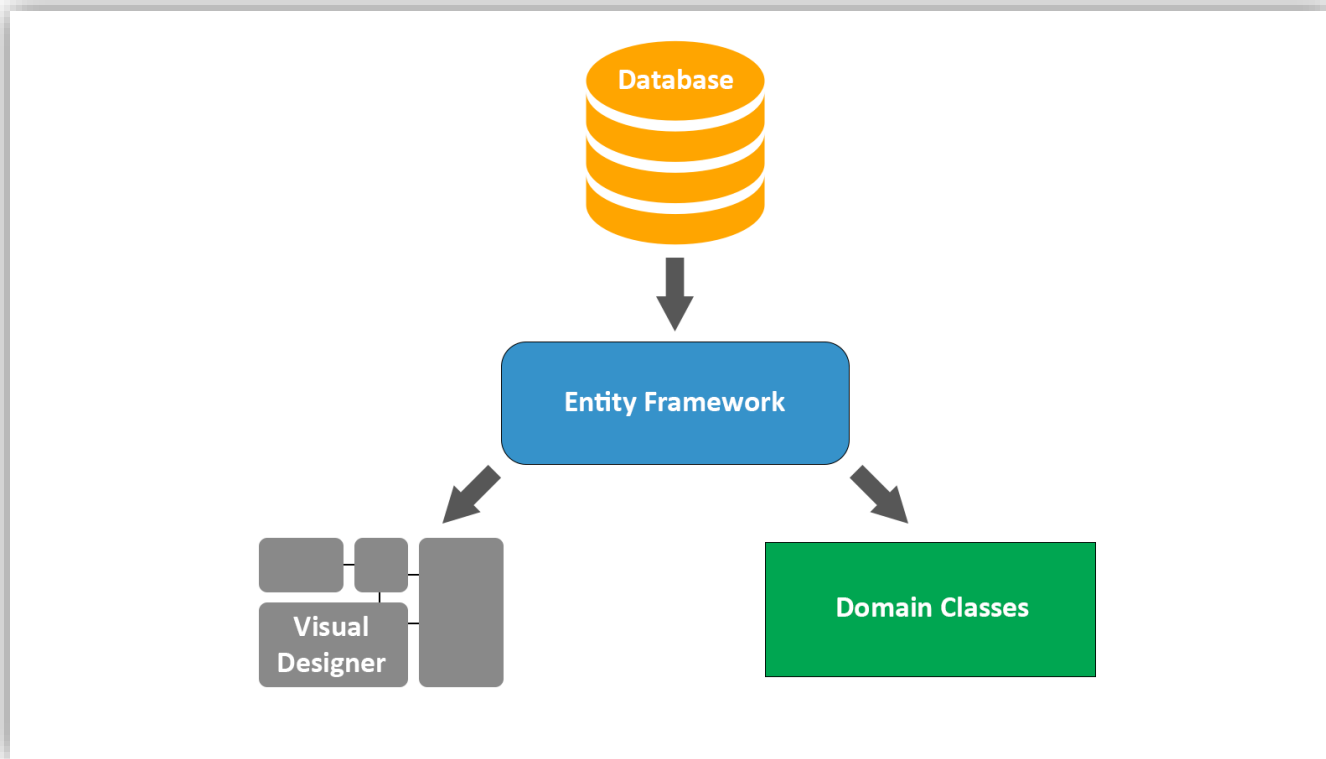
Model First

- working on a visual diagram and letting the ORM framework – Entity Framework – create/update the rest accordingly



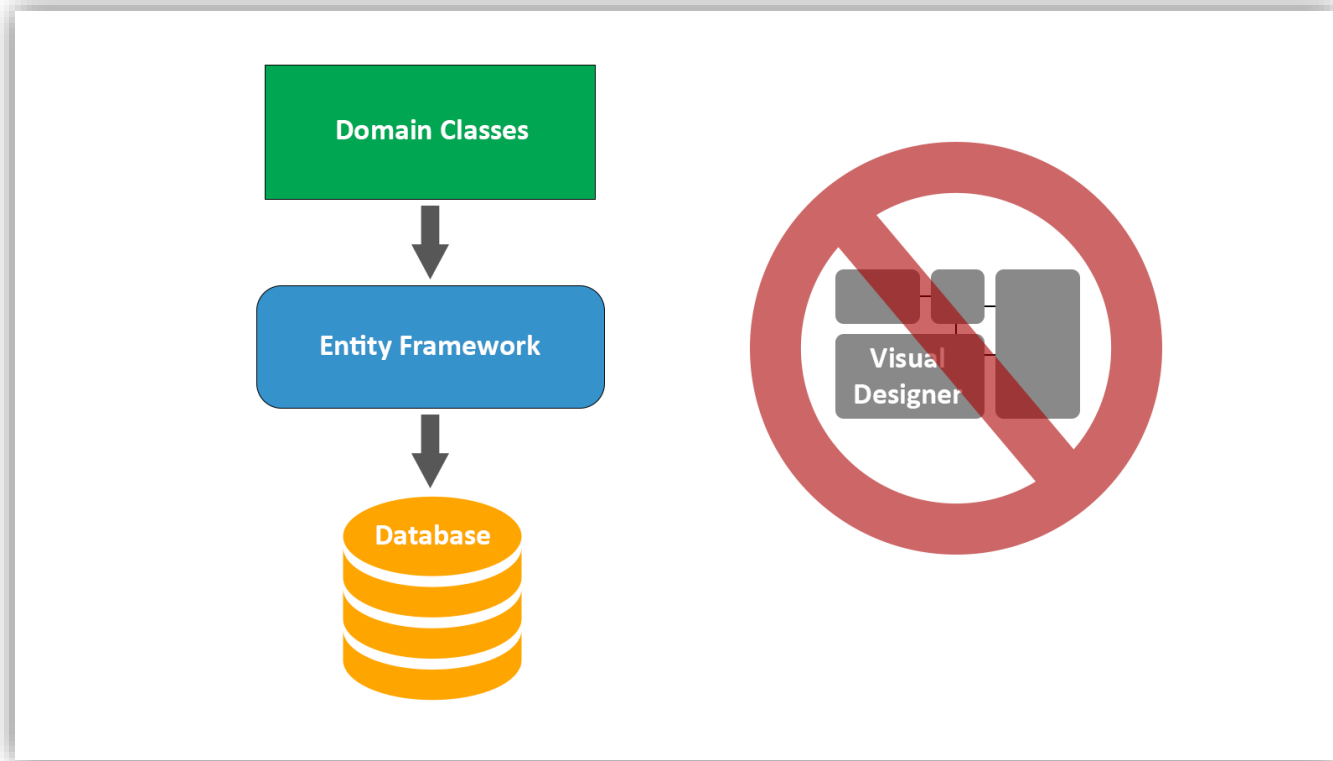
Database First

- building the Database and letting Entity Framework create/update the rest accordingly



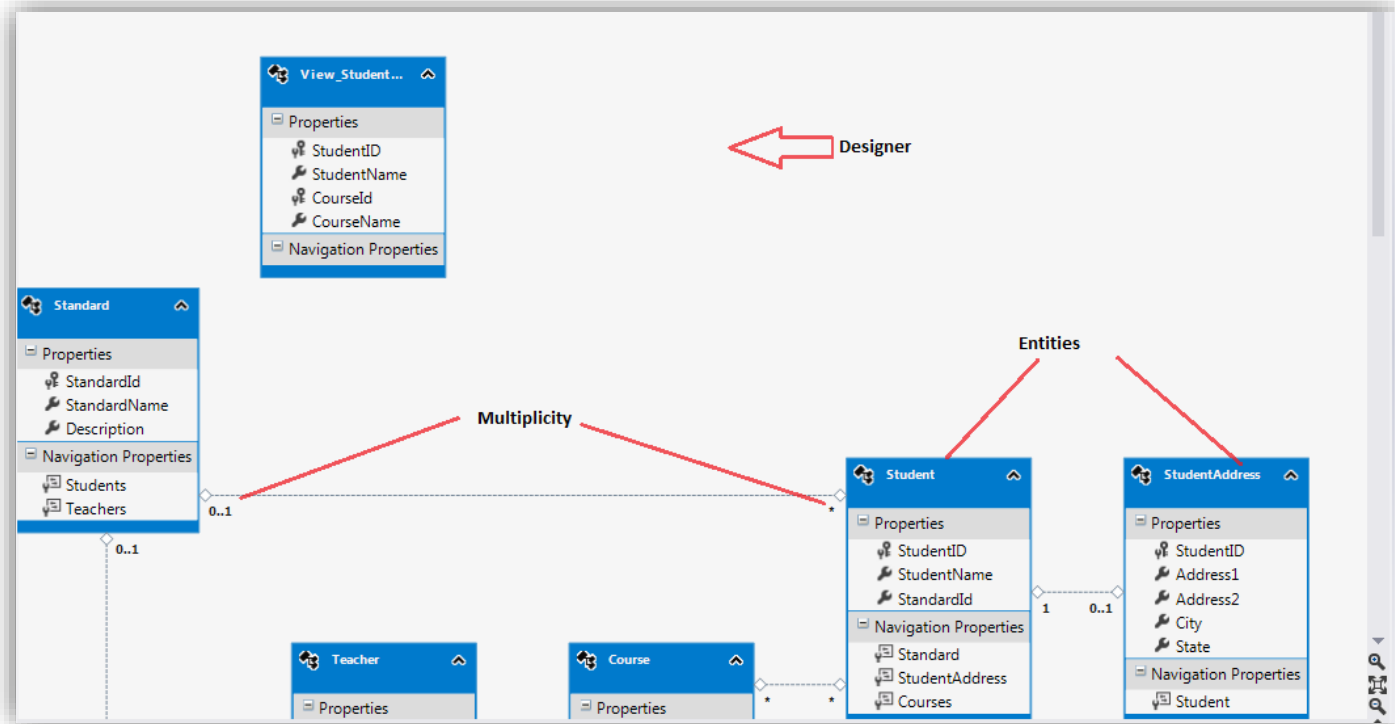
Code First

- writing the Data Model entity classes we'll be using within our project and let Entity Framework generate the Database accordingly



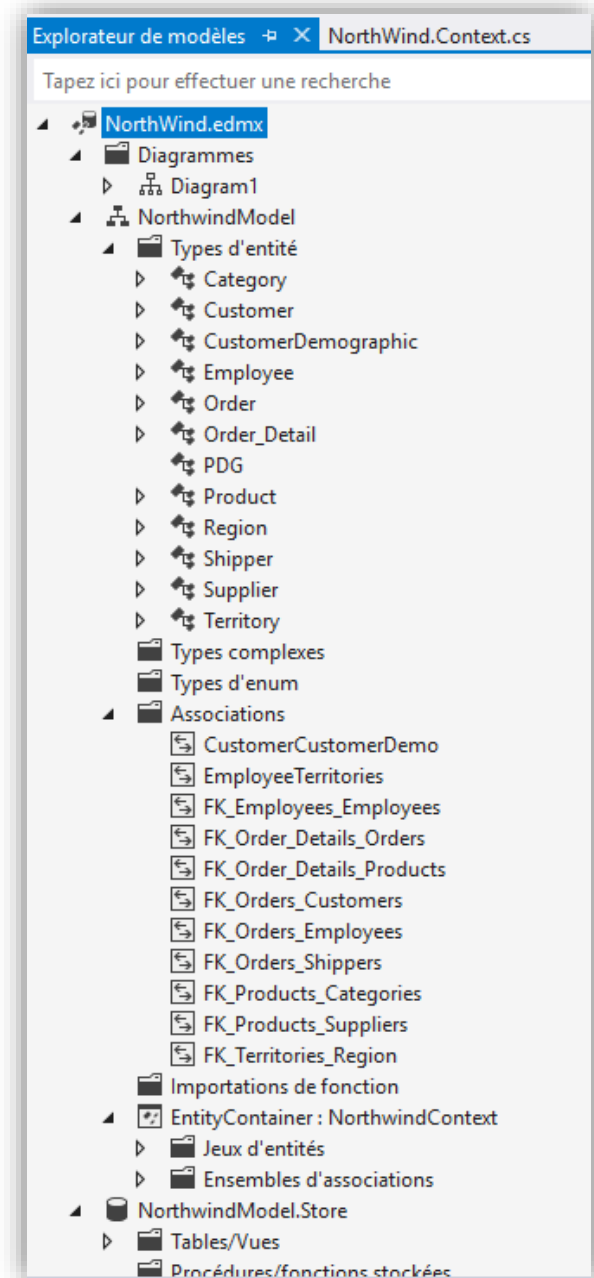
EDM Creation

- Entity Data Model Wizard in VS2017 opens with four options to select.



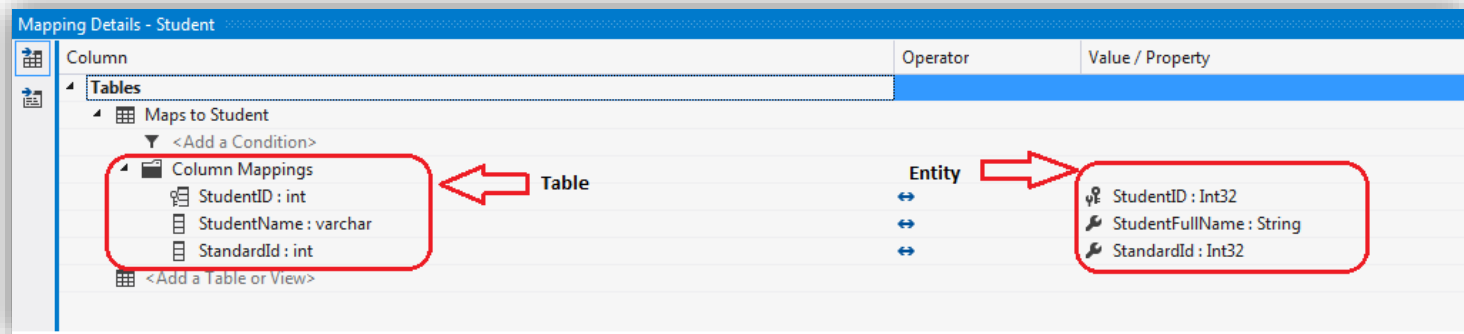
The EDM as a conceptual model

- **The model browser :** Model browser contains all the information about the EDM, its conceptual model, storage model and mapping information.
 - Open Model Browser by right clicking on the EDM Designer and select Model Browser from context menu
- **EDM structure**
 - **EntityContainer** EntityContainer is a wrapper for EntitySets and AssociationSets. It is a critical entry point for querying the model.
 - **EntitySet:** EntitySet is a container for EntityType. It is set of the same EntityTypes. You can think it of like the db table.
 - **EntityType:** EntityType is a datatype in the model. You can see each EntityType for your conceptual model in XML. If you expand EntityType node in XML, you can see each properties and its type and other info.
 - **AssociationSet:** AssociationSet defines the relation between each EntitySet.



Entity Data Model :Entity-Table Mapping

- Each entity in EDM is mapped with the database table.
 - right click on any entity in the EDM designer -> select Table Mapping.
 - change any property name of entity from designer then the table mapping would reflect that change automatically



- Context & Entity Classes:
 - Every Entity Data Model generates one context class and multiple entity classes for each DB.
 - Expand School.edmx and see two important files can be seen, {EDM Name}.Context.tt and {EDM Name}.tt:

Querying with EDM

- You can query EDM mainly by three ways

LINQ to Entities: L2E returns IQueryable.

- LINQ Method syntax (Lambda Expressions):

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

- LINQ Query syntax:

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

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

Querying with EDM

Entity SQL: It is processed by the Entity Framework's Object Services directly. It returns `ObjectQuery` instead of `IQueryable`.
You need `ObjectContext` to create a query using 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>();
```

Native SQL

```
using (var ctx = new SchoolDBEntities())
{
    var student = ctx.Students
        .SqlQuery("Select * from Students where StudentId=@id", new SqlParameter("@id", 1))
        .FirstOrDefault();
}
```

LINQ to Entities

- **Projection Queries**

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

- **First/FirstOrDefault:**

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

The difference = First() will throw an **exception** if there is no result data for the supplied criteria
FirstOrDefault() returns default value (null) if there is no result data

- **Single/SingleOrDefault:**

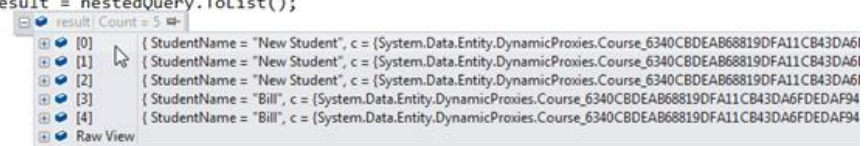
```
var student = (from s in context.Students where s.StudentID == 1 select s).SingleOrDefault<Student>();
```

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.

- **ToList:** converts the results to a List
 - **GroupBy:** groups the results by a criteria
 - **OrderBy :** sorts the results by a criteria
 - **Nested queries:** The nested query shown above will result in an anonymous list with a StudentName and Course object.

```
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();
}
```



Index	StudentName	Course
[0]	New Student	{ StudentName = "New Student", c = { System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA61...
[1]	New Student	{ StudentName = "New Student", c = { System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA61...
[2]	New Student	{ StudentName = "New Student", c = { System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA61...
[3]	Bill	{ StudentName = "Bill", c = { System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA61...
[4]	Bill	{ StudentName = "Bill", c = { System.Data.Entity.DynamicProxies.Course_6340CBDEAB68819DFA11CB43DA61...

The DBContext

DbContext

- **DbContext is the primary class that is responsible for interacting with data as object. It often referred to as context.**
- **It does following activities:**
 - **Querying:** It can be useful in querying the database. It converts database values into entity objects and vice versa.
 - **Change Tracking:** It keeps track of changes occurred in the entities after it has been querying from the database.
 - **Persisting Data:** It also performs the Insert, update and delete operations to the database, based on the entity states.
- **Prior to EntityFramework 4.1, EDM used to generate context class, that were derived from the **ObjectContext** class.**
 - DbContext is conceptually similar to ObjectContext. It is a wrapper around ObjectContext which is useful in all the development models: Code First, Model First and Database First.
 - DbContext API is easier to use than ObjectContext API for all common tasks. However, you can get the reference of ObjectContext from DbContext in order to use some of the features of ObjectContext. This can be done by using **IObjectContextAdapter** as shown below:

```
using (var ctx = new SchoolDBEntities())  
{ var objectContext =  
    (ctx as IObjectContextAdapter).ObjectContext; //use objectContext now...
```

DBContext

- **DBContext represents a combination of the Unit-Of-Work and Repository patterns and enables you to query a database, and group together changes that will then be written back to the store as a unit.**
 - Public Methods

Method Name	Return Type	Description
Entry(Object)	<i>DbEntityEntry</i>	Gets a <i>DbEntityEntry</i> object for the given entity, providing access to information about the entity and the ability to perform actions on the entity.
Entry <TEntity> (TEntity)	<i>DbEntityEntry<TEntity></i>	Gets a <i>DbEntityEntry<TEntity></i> object for the given entity, providing access to information about the entity and the ability to perform actions on the entity.
Set(Type)	<i>DbSet</i>	Returns a <i>DbSet</i> for the specified type, this allows CRUD operations to be performed for the given entity in the context.
Set<TEntity>()	<i>DbSet</i>	Returns a <i>DbSet</i> for the specified type, this allows CRUD operations to be performed for the given entity in the context.
SaveChanges()	<i>int</i>	Saves all changes made in this context to the underlying database.

- **Public Properties:**

Property Name	Return Type	Description
ChangeTracker	<i>DbChangeTracker</i>	Provides access to features of the context that deal with changing tracking of entities.
Configuration	<i>DbContextConfiguration</i>	Provides access to configuration options for the context.
Database	<i>Database</i>	Creates a database instance for this context and allows you to perform creation, deletion or existence checks for the underlying database.

DBSet Class

- **DBSet class represents an entity set that is used for the create, read, update, and delete operations. A generic version of DBSet (DbSet<entity>) can be used when the type of entity is not known at build time.**
 - You can get the reference of DBSet by using DbContext eg. dbContext.Categories or dbContext.Posts or any entity set. DbContext class includes DbSet as shown below:

```
public class ForumContext : DbContext
{
    public DbSet<Category> Categories { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<PostAnswer> PostAnswer { get; set; }
    public DbSet<Tag> Tags { get; set; }
}
```

- **The important methods of DBSet class are**
 - **Add**
 - **Attach(Entity)**
 - **Create**
 - **Find(int)**
 - **Include**
 - **Remove**
 - **SqlQuery**

DBEntityEntry

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

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

- **DBEntityEntry** class has following important methods:
 - **Collection** : Gets an object that represents the collection navigation property from this entity to a collection of related entities

```
var studentDBEntityEntry = dbContext.Entry(studentEntity);  
var collectionProperty = studentDBEntityEntry.Collection(s => s.Courses);
```
 - **GetDatabaseValues** : Queries the database for copies of the values of the tracked entity as they currently exist in the database.
 - **Property** : Gets an object that represents a scalar or complex property of this entity.

```
var studentDBEntityEntry = dbContext.Entry(studentEntity);  
string propertyName = studentDBEntityEntry.Property("StudentName").Name;
```
 - **Reference** : Gets an object that represents the reference (i.e. non-collection) navigation property from this entity to another entity.
 - **Reload** : Reloads the entity from the database overwriting any property values with values from the database. The entity will be in the Unchanged state after calling this method.

CRUD Operations

Entity Lifecycle

- **Before we work on CRUD operation (Create, Read, Update, Delete), it's important to understand the entity lifecycle and how it's being managed by the EntityFramework.**
 - 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.EntityState* that declares the following values:
 1. **Added**
 2. **Deleted**
 3. **Modified**
 4. **Unchanged**
 5. **Detached**
- **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.

Entity using DbContext

- We will learn how to add and save single a entity using DbContext in the disconnected scenario which will in tern insert a single row in database table.
 - We will see how to add single 'Standard' entity

```
// create new Standard entity object
var newStandard = new Standard();
// Assign standard name
newStandard.StandardName = "Standard 1";
//create DbContext object
using (var dbCtx = new SchoolDBEntities())
{ //Add standard object into Standard DBset
  dbCtx.Standards.Add(newStandard);
  // call SaveChanges method to save standard into database
  dbCtx.SaveChanges();
}
```
 - Alternatively, we can also add Standard entity into DbContext.Entry and mark it as Added which results in same insert query:

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

Update Entity using DbContext

- As we have learned about DbContext.Entry method in the previous chapter, the Entry() method is useful to get DBEntityEntry for given Entity.

- The following statement attaches entity to context and marks its state as Modified:

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

- Let's see how to update an existing single 'Standard' entity

```
Student stud ;
// Get student from DB
using (var ctx = new SchoolDBEntities())
{ stud = ctx.Students.Where(s => s.StudentName == "New
                                Student1").FirstOrDefault<Student>();
}
// change student name in disconnected mode (out of DbContext scope)
if (stud != null) { stud.StudentName = "Updated Student1"; }
//save modified entity using new DbContext
using (var dbCtx = new SchoolDBEntities())
{
    //Mark entity as modified
    dbCtx.Entry(stud).State = System.Data.EntityState.Modified;
    dbCtx.SaveChanges();
}
```

- As you see in the above code snippet, we are doing the following steps:
 - Get the existing student
 - Change the student name out of DbContext scope (disconnected mode)
 - We pass the modified entity into the Entry method to get its DBEntityEntry object and then mark its state as Modified
 - Call SaveChanges() method to update student information into the database.

Delete Entity using DbContext

- We can use the **Entry()** method to attach a disconnected entity to the context and mark its state to **Deleted**

```
using (var context = new SchoolDBEntities())
{
    context.Entry(disconnectedTeacher).State =
        System.Data.EntityState.Deleted;
    context.SaveChanges();
}
```

- The code shown above results in the following delete query which deletes the row from Teacher table.

```
delete [dbo].[Teacher] where ([TeacherId] = @@0)',N'@@0 int',@@0=1
```

Configure Domain Classes in Code-First

Configure Domain Classes in Code-First

- **In Database-First , you can create entity data model from existing database.**
 - This EDM has all the metadata information in SSDL, CSDL and MSL, so that EF can use this model in querying, change tracking, updating functionality etc..
- **The same way, entity framework Code-First allows you to use your domain classes to build the model, which in turn will be used by EF in different activity.**
 - Code-First suggests certain **conventions** for your domain classes to follow by, so that EF can interpret it and build the model out of it.
 - However, if your domain classes don't follow the conventions, then you also have the **ability to configure your domain classes**, so that EF can understand it and build the model out of it.
 - There are two ways by which you can configure your domain classes:
 1. **DataAnnotation**
 2. **Fluent API**

DataAnnotation

- **DataAnnotation is a simple attribute based configuration, which you can apply on your domain classes and its properties.**
 - You can find most of the attributes in the `System.ComponentModel.DataAnnotations` namespace.
 - However, DataAnnotation provides **only a subset of Fluent API configurations**. So if you don't find some attributes in DataAnnotation, then you have to use Fluent API to configure it.

```
[Table("StudentInfo")]
public class Student
{ public Student() { }
  [Key]
  public int SID { get; set; }
  [Column("Name", TypeName="ntext")]
  [MaxLength(20)]
  public string StudentName { get; set; }
  [NotMapped] public int? Age { get; set; }
  public int StdId { get; set; }
  [ForeignKey("StdId")]
  public virtual Standard Standard { get; set; }
}
```

DataAnnotation

- **[Key]** – specifies the primary key of the table
- For validation: **[StringLength], [MaxLength], [MinLength], [Required]**
- Schema: **[Column], [Table], [ComplexType], [ConcurrencyCheck], [Timestamp], [ComplexType], [InverseProperty], [ForeignKey], [DatabaseGenerated], [NotMapped]**
- In EF 6 we will be able to add custom attributes by using custom conventions

Configure One-to-One Relationship

■ We are going to configure a One-to-One relationship between **Student** and **StudentAddress**

- As you may know that one to one relationship happens when primary key of one table becomes PK & FK in another table.
- Here, StudentId is a Primary key of Student table so StudentId should be PK and FK in StudentAddress table in order to have one-to-one (one to zero or one) relationship between them

■ Configure one to zero or one relationship using DataAnnotation

```
public class Student
{
    public Student() { }

    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }

    [Required]
    public virtual StudentAddress StudentAddress { get; set; }
}

public class StudentAddress
{
    [Key, ForeignKey("Student")]
    public int StudentId { get; set; }

    public string Address1 { get; set; }
    public string Address2 { get; set; }
    public string City { get; set; }
    public int Zipcode { get; set; }
    public string State { get; set; }
    public string Country { get; set; }

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

■ Configure one to zero or one relationship using Fluent API

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<StudentAddress>()
        .HasKey(e => e.StudentId);
    modelBuilder.Entity<StudentAddress>()
        .Property(e => e.StudentId)
        .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
    modelBuilder.Entity<StudentAddress>()
        .HasRequired(e => e.Student)
        .WithRequiredDependent(s => s.StudentAddress);

    base.OnModelCreating(modelBuilder);
}
```

Configure One-to-Many Relationship

■ Configure One-to-Many relationship using DataAnnotation

```
public class Student
{
    public Student() { }

    public int StudentId { get; set; }
    [Required]
    public string StudentName { get; set; }

    public int StdandardId { get; set; }

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

```
public class Standard
{
    public Standard()
    {
        StudentsList = new List<Student>();
    }
    public int StandardId { get; set; }
    public string StandardName { get; set; }
    public string Description { get; set; }

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

■ Configure One-to-Many relationship using Fluent API

- Suppose your Student and Standard entity class don't follow Code-First conventions and they have different property names, for example:

```
public class Student { ..
    //StdId is not following code first conventions name
    public int StdId { get; set; } public virtual Standard Standard { get; set; } .. }
public class Standard { ..
    public virtual ICollection<Student> StudentsList { get; set; }.. }
```

- Two possible ways

```
//one-to-many
modelBuilder.Entity<Student>().HasRequired<Standard>(s => s.Standard)
    .WithMany(s => s.StudentsList).HasForeignKey(s => s.StdId);

//one-to-many
modelBuilder.Entity<Standard>().HasMany<Student>(s => s.StudentsList)
    .WithRequired(s => s.Standard).HasForeignKey(s => s.StdId);
```

Configure Many-to-Many relationship

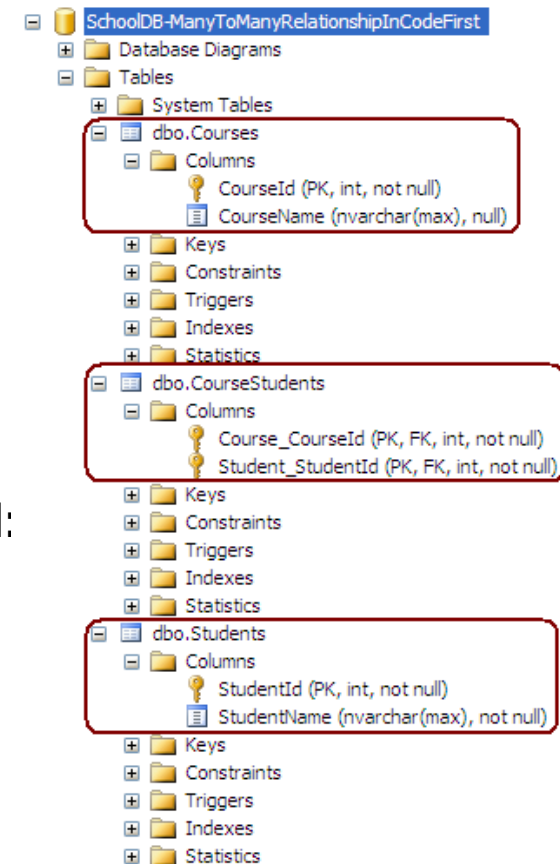
■ Configure Many-to-Many relationship using DataAnnotation:

- Student class should have a collection navigation property for Course, and Course should have a collection navigation property for student, which will create a Many-to-Many relationship between student and course as shown below:

```
public class Student {  
    public int StudentId { get; set; }  
    [Required] public string StudentName { get; set; }  
    public int StdandardId { get; set; }  
    public virtual ICollection<Course> Courses { get; set; }  
}  
  
public class Course {  
    public Course() { this.Students = new HashSet<Student>(); }  
    public int CourseId { get; set; }  
    public string CourseName { get; set; }  
    public virtual ICollection<Student> Students { get; set; }  
}
```

■ Configure Many-to-Many relationship using Fluent API:

```
modelBuilder.Entity<Student>().  
    HasMany<Course>(s => s.Courses).  
    WithMany(c => c.Students).Map(c =>  
    { c.MapLeftKey("Student_id");  
      c.MapRightKey("Course_id");  
      c.ToTable("StudentAndCourse");  
    });
```

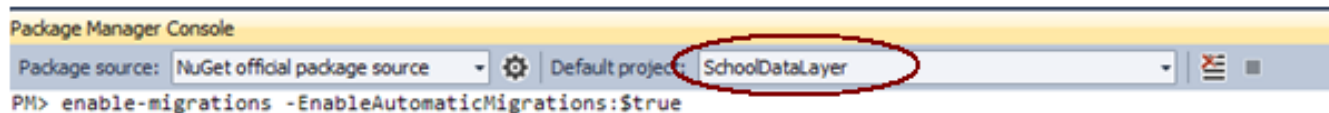


Migration in Code-First:

- Entity framework Code-First had different database initialization strategies prior to EF 4.3 like **CreateDatabaseIfNotExists**, **DropCreateDatabaseIfModelChanges** or **DropCreateDatabaseAlways**.
 - However, there were some problems with these strategies, for example if you already have data (other than seed data) or existing Stored Procedures, triggers etc in your database, then these strategies used to drop the entire database and recreate it, so you would **lose the data** and other db objects
- Entity framework 4.3 has introduced migration that automatically updates database schema, when your model changes without losing any existing data or other database objects. It uses new database initializer called **MigrateDatabaseToLatestVersion**
 - There are two kinds of Migration:
 1. Automated Migration
 2. Code based Migration

Automated Migration

- Entity framework 4.3 has introduced Automated Migration so that you don't have to maintain database migration manually in the code file, for each change you make in your domain classes.
 - You just need to run a command in Package Manger Console to accomplish this
 - Before running the application which does not have its database created yet, **you have to enable automated migration by running the 'enable-migrations' command** in Package Manager Console as shown below :



- Once the command runs successfully, it creates an internal sealed Configuration class in the Migration folder in your project:
- If you open this and see the class shown below, then you will find AutomaticMigrationsEnabled = true in the constructor.

```
internal sealed class Configuration :  
    DbMigrationsConfiguration<SchoolDataLayer.SchoolDbContext> {  
    public Configuration() { AutomaticMigrationsEnabled = true; }  
    protected override void Seed(SchoolDataLayer.SchoolDbContext context) {...}  
}
```

