What is EF?

- Entity Framework is an open source ORM framework from Microsoft.
- Entity framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database, and for working with the results, in addition to DataReader and DataSet.
- Using the Entity Framework, developers issue queries using LINQ, then retrieve and manipulate data as strongly typed objects.
- The Entity Framework's ORM implementation provides services like change tracking, identity resolution, lazy loading, and query translation so that developers can focus on their application-specific business logic rather than the data access fundamentals.
- ORM is a tool for storing data from domain objects to relational database like MS SQL Server, in an automated way, without much programming. O/RM includes three main parts: Domain class objects, Relational database objects and Mapping information on how domain objects map to relational database objects (tables, views & stored procedures).

Why Entity Framework

- It reduces development time and cost
- Provides auto generated code
- Allows you to use Domain objects to manage database
- Allows you to uses LINQ to query database using LINQ to Entities
- Improves performance with Caching and Lazy Loading
- No need to write SQL commands
- Automatic transaction management

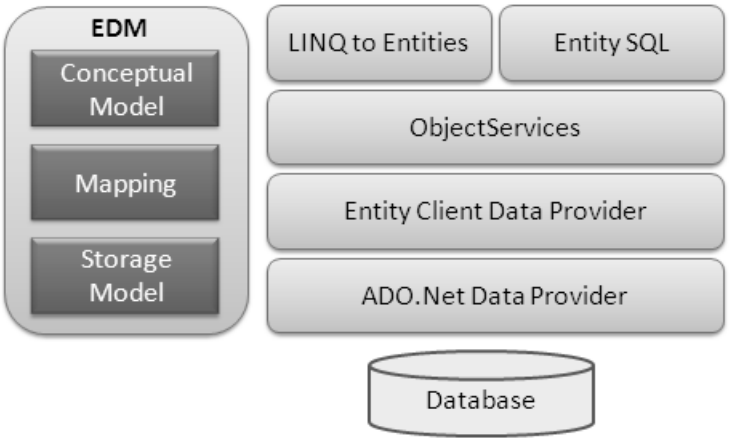
History of Entity Framework

The following are important features added in each major version of Entity Framework.

EF 6	This release can be used in Visual Studio 2013, Visual Studio 2012 and Visual Studio 2010 (runtime only) to write applications that target .NET 4.0 and .NET 4.5.
EF 5	❑ This release can be used in Visual Studio 2010 and Visual Studio 2012 to write applications that target .NET 4.0 and .NET 4.5. ❑ When targeting .NET 4.5 this release introduces some new features including enum support, table-valued functions, spatial data types and various performance improvements. ❑ The Entity Framework Designer in Visual Studio 2012 also introduces support for multiple-diagrams per model, coloring of shapes on the design surface and batch import of stored procedures.
EF 4.3	The EF 4.3 release included the new Code First Migrations feature that allows a database created by Code First to be incrementally changed as your Code First model evolves.
EF 4.1	The EF 4.1 release was the first to be published on NuGet. This release included the simplified DbContext API and the Code First workflow.
EF 4	❑ This release was included in .NET Framework 4 and Visual Studio 2010. ❑ New features in this release included POCO support, lazy loading. ❑ Added Model First workflow. ❑ Although it was the second release of Entity Framework it was named EF 4 to align with the .NET Framework version that it shipped with. ❑ After this release Microsoft started making Entity Framework available on NuGet and adopted semantic versioning since it is no longer tied to the .NET Framework Version.
EF (or EF 3.5)	The initial release of Entity Framework was included in .NET 3.5 SP1 and Visual Studio 2008 SP1. This release provided basic O/RM support using the Database First workflow.

Entity Framework Architecture

The following figure shows the overall architecture of the Entity Framework.



EDM (Entity Data Model)	EDM consists three main parts- Conceptual model, Mapping and Storage model. This is inside Entity Data Model XML (.EDMX) file.
Conceptual Model	The conceptual model contains the model classes and their relationships. This will be independent from your database table design.
Storage Model	Storage model is the database design model which includes tables, views, stored procedures and their relationships and keys.
Mapping	Mapping consists information about how the conceptual model is mapped to 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.
Object Service	Object service is a main entry point for accessing data from the database and to return it back. Object service is responsible for materialization, which is process of converting data returned from entity client data provider (next layer) to an entity object structure.
Entity Client Data Provider	The main responsibility of this layer is to convert L2E or Entity SQL queries into a SQL query which is understood by the underlying database. It communicates with the ADO.Net data provider which in turn sends or retrieves data from database.
ADO.Net Data Provider	This layer communicates with database using standard ADO.Net.

Entity-Table Mapping

Each entity in EDM (Entity Data Model) is mapped with the database table. You can check entity-table mapping by right click on any entity in the EDM designer -> select Table Mapping. Also, if you 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 table. Expand College.edmx to see important files - {EDM Name}.Context.tt and {EDM Name}.tt:

College.Context.tt

This template file generates a context class whenever you change Entity Data Model (.edmx file). You can see the context class file by expanding College.Context.tt. The context class resides in {EDM Name}.context.cs file. The default context class name is {DB Name} + Entities eg. The context class name for CollegeDB is CollegeDBEntities. Context class is derived from DBContext class in Entity Framework 5.0. Prior to EF 5.0 it had been derived from ObjectContext.

Types of Entities

The following are different types of Entities in EF 5.0.

POCO (Plain Old CLR Object)

POCO class is the class that doesn't depend on any framework specific base class. It is like any other normal .net class that is why it is called "Plain Old CLR Object".

Dynamic Proxy (POCO Proxy)

Dynamic Proxy is a runtime proxy class of POCO entity. POCO entity becomes POCO Proxy entity if it meets certain requirements in order to enable lazy loading and automatic change tracking.

It adds some methods at runtime to your POCO class which does instant change tracking and lazy loading stuff.

POCO entity should meet the following requirements to become a POCO proxy:

- A custom data class must be declared with public access
- A custom data class must not be sealed
- A custom data class must not be abstract
- *ProxyCreationEnabled* option should not set to false (default is true) in context class
- Each navigation property must be declared as public, virtual

Note: By default dynamic proxy is enabled for every entity. However, you can disable dynamic proxy by setting the **ProxyCreationEnabled** option to false in context class.

Note: EDM generates POCO entities which satisfy the above requirements for a dynamic proxy by default.

```
context.Configuration.ProxyCreationEnabled = false;
```

Get entity from Dynamic Proxy as follows:

```
ObjectContext.GetObjectType(employee.GetType());
```

Types of Properties

Entity can have two types of properties, Scalar properties and Navigation properties.

Scalar properties

Scalar properties are properties whose actual values are contained in the entity. For example, Student entity has scalar properties e.g. Id,Name. These correspond with the Student table columns.

Navigation properties

Navigation properties are pointers to other related entities. The Student has Course property as a navigation property that will enable application to navigate from a Student to related course entity.

MODELING

The following are different approaches to modeling in EF.

Code First
In the Code First approach, you avoid working with visual model designer (EDMX) completely. You write your POCO classes first and then create database from these POCO classes. Developers who follow the path of Domain-Driven Design (DDD) principles prefer to begin by coding their classes first and then generating the database required to persist their data.

Model First
In the Model First approach, you create Entities, relationships, and inheritance hierarchies directly on the design surface of EDMX. So in the Model First approach, when you add the ADO.NET Entity Data Model, you should select Empty EF Designer model. We use Entity Data Model Designer to design model from scratch.

Database First

In this we create the model and classes from an existing database. So when you generate EDMX from an existing database then it is a Database First approach.

EDMX File

When using Visual Studio design tools to create the .CSDL, Store Schema Definition Language (.SSDL), and Mapping Specification Language (.MSL) files, all three are stored in a single .EDMX file, rather than in separate files.

This file contains three sections:

- CSDL (Conceptual Schema Definition Language)
- SSDL (Store Schema Definition Language)
- MSL (Mapping Specification Language)

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.EntityState that declares the following values:

- Added
- Deleted
- Modified
- Unchanged
- 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.

Example - InventoryContext

This class extends DbContext and provides required properties of DbSet type.

```
public partial class InventoryContext : DbContext
    {
        public InventoryContext()
            : base("name=InventoryEntities")
        {
        }


        public virtual DbSet<Category> Categories { get; set; }
        public virtual DbSet<Product> Products { get; set; }
    }
```

Category.cs

Category class represents Categories table.

```
public partial class Category
    {
        public Category()
        {
            this.Products = new HashSet<Product>();
        }

        public string CatCode { get; set; }
        public string CatDesc { get; set; }

        public virtual ICollection<Product> Products { get; set; }
    }
```

Product.cs

Product class represents a product in Products table.

```
public partial class Product
    {
        public int ProdId { get; set; }
        public string ProdName { get; set; }
        public Nullable<decimal> Price { get; set; }
        public Nullable<int> Qoh { get; set; }
        public string Remarks { get; set; }
        public string CatCode { get; set; }

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

Listing entities

The following code displays category description from categories table using Categories DbSet.

```
class ListCategories
    {
        public static void Main()
        {
            InventoryContext ctx = new InventoryContext();
            var cats = from c in ctx.Categories
                       select c;

            foreach (var c in cats)
                Console.WriteLine(c.CatDesc);
        }
    }
```

Adding an Entity

The following code shows how to add a new category to Categories table through Categories DbSet.

```
class AddCategory
    {
        public static void Main()
        {
            InventoryContext ctx = new InventoryContext();
            Category c = new Category { CatCode = "tablet",CatDesc = "Tablets" };

            ctx.Categories.Add(c);
            ctx.SaveChanges();

            Console.WriteLine("A new category has been added!");
        }
    }
```

Modifying an Entity

The following code retrieves a category with code tablet and modifies its description.

```
class UpdateCategory
    {
        public static void Main()
        {
            InventoryContext ctx = new InventoryContext();
            Category c = ctx.Categories.Find("tablet");
            if ( c == null)
            {
                Console.WriteLine("Category not found!");
                return;
            }

            c.CatDesc = "10 Inch Tablets";
            ctx.SaveChanges();

            Console.WriteLine("Category has been updated!");

        }
    }
```

Deleting an Entity

In order to delete an entity, we have to first bring entity into context and then remove it from DbSet as follows:

```
class DeleteCategory
    {
        public static void Main()
        {
            InventoryContext ctx = new InventoryContext();
            Category c = ctx.Categories.Find("tablet");

            if (c == null)
            {
                Console.WriteLine("Category not found!");
                return;
            }

            ctx.Categories.Remove(c);
            ctx.SaveChanges();

            Console.WriteLine("Category has been deleted!");
        }
    }
```

List related information from other entities

It is possible to get related information from entities that are related to current entity.

The following code shows how to display product name and category description. Product name comes from Products table (Product object), but category description is to be taken from Categories table (Category object) through navigation property Category in Product class.

```
class ListProducts
{
        public static void Main()
        {
            InventoryContext ctx = new InventoryContext();

            var prods  = from p in ctx.Products
                         select p;

            foreach (var p in prods)
                Console.WriteLine("{0}  - {1}", p.ProdName, p.Category.CatDesc);
        }
}
```

Eager Loading

Eager loading is the process whereby a query for one type of entity also loads related entities as part of the query. Eager loading is achieved using the Include method of IQueryable**.**

The code snippet shown below demonstrates eager loading, where Course entity will also be loaded with Student entity using the Include() method:

```
using (var context = new CollegeEntities())
{
   var res = (from s in context.Students.Include("Course")
              where s.Name == "Bob"
              select s).FirstOrDefault<Student>();
}
```

Lazy Loading

- One of the important functions of Entity Framework is lazy loading.
- Lazy loading means delaying the loading of related data, until you specifically request it.
- When using POCO entity types, lazy loading is achieved by creating instances of derived proxy types and then overriding virtual properties to add the loading hook.
- Lazy loading of the Posts collection can be turned off by making the Posts property **non-virtual**. When lazy loading is turned off the only way to get related details is to explicitly load related data using Load() method.

It is possible to disable lazy loading for all entities as follows:

```
context.Configuration.LazyLoadingEnabled = false;   // default is true
```

It is possible to load an entity object explicitly using the following code :

```
context.Entry(student).Reference(s => s.Course).Load();
```

Use collection() method to load a collection explicitly :

```
context.Entry(student).Collection(s => s.Marks).Load();
```

Proxy

- When creating instances of POCO entity types, the Entity Framework often creates instances of a dynamically generated derived type that acts as a proxy for the entity.
- This proxy overrides some virtual properties of the entity to insert hooks for performing actions automatically when the property is accessed.
- This mechanism is used to support lazy loading of relationships.

A proxy is created only when entities satisfy the following rules:

- The class must be public and not sealed.
- Each property must be marked as virtual.
- Each property must have a public getter and setter.
- Any collection navigation property must be typed as ICollection<T>.

It is possible to get actual class instead of proxy class as follows:

```
HRContext ctx = new HRContext();
var dept = ctx.Departments.Find(1);

Console.WriteLine(dept.GetType().FullName);
Console.WriteLine(ObjectContext.GetObjectType(dept.GetType()));
```

DBContext Class

- EDM generates the Context class and Entity classes. This context class is derived from the System.Data.Entity.DbContext class.
- DbContext is the primary class that is responsible for interacting with data as object. It is 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.
- **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.

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

Property Name	Return Type	Description
ChangeTracker	DBChangeTracker	Provides access to features of the context that deal with changing tracking of entities.
Configuration	DBContextConfiguration	Provides access to configuration options for the context.
Database	Database	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 create, read, update, and delete operations.
- A generic version of DBSet (DbSet<tentity>) can be used when the type of entity is not known at build time.

Method Name	Return Type	Description
Add	Added entity type	Adds the given entity to the context with Added state. When the changes are being saved, the entities in the Added states are inserted into the database. After the changes are saved, the object state changes to Unchanged.
Attach(Entity)	Entity	Attaches the given entity to the context in the Unchanged state
Create	Entity	Creates a new instance of an entity for the type of this set. This instance is not added or attached to the set. The instance returned will be a proxy if the underlying context is configured to create proxies and the entity type meets the requirements for creating a proxy.
Find(int)	Entity type	Uses the primary key value to attempt to find an entity tracked by the context. If the entity is not in the context then a query will be executed and evaluated against the data in the data source, and null is returned if the entity is not found in the context or in the data source. Note that the Find also returns entities that have been added to the context but have not yet been saved to the database.
Include	DBQuery	Returns the included non generic LINQ to Entities query against a DbContext. (Inherited from DbQuery)
Remove	Removed entity	Marks the given entity as Deleted. When the changes are saved, the entity is deleted from the database. The entity must exist in the context in some other state before this method is called.
SqlQuery	DBSqlQuery	Creates a raw SQL query that will return entities in this set. By default, the entities returned are tracked by the context; this can be changed by calling AsNoTracking on theDbSqlQuery<TEntity> returned from this method.

DBEntityEntry Class

- DBEntityEntry class is used to retrieve various information about an entity.
- We can get an instance of DBEntityEntry of a particular entity by using **Entry** method of DBContext.

```
DBEntityEntry employeeEntry = dbcontext.Entry(employee);
```

Property	Description
CurrentValues	Gets the current property values for the tracked entity represented by this object.
Entity	Gets the entity.
OriginalValues	Gets the original property values for the tracked entity represented by this object. The original values are usually the entity's property values as they were when last queried from the database.
State	Gets or sets the state of the entity.

Method	Description
Collection(String)	Gets an object that represents the collection navigation property from this entity to a collection of related entities.
GetDatabaseValues()	Queries the database for copies of the values of the tracked entity as they currently exist in the database. Note that changing the values in the returned dictionary will not update the values in the database. If the entity is not found in the database then null is returned.
Property(String)	Gets an object that represents a scalar or complex property of this entity.
Reference(String)	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.

DbContextConfiguration class

Returned by the Configuration method of DbContext to provide access to configuration options for the context.

Property	Description
AutoDetectChangesEnabled	Gets or sets a value indicating whether the DetectChanges method is called automatically by methods of DbContext and related classes. The default value is true.
LazyLoadingEnabled	Gets or sets a value indicating whether lazy loading of relationships exposed as navigation properties is enabled. Lazy loading is enabled by default.
ProxyCreationEnabled	Gets or sets a value indicating whether or not the framework will create instances of dynamically generated proxy classes whenever it creates an instance of an entity type. Note that even if proxy creation is enabled with this flag, proxy instances will only be created for entity types that meet the requirements for being proxied. Proxy creation is enabled by default.
UseDatabaseNullSemantics	Gets or sets a value indicating whether database null semantics are exhibited when comparing two operands, both of which are potentially nullable. The default value is false. For example (operand1 == operand2) will be translated as: (operand1 = operand2) if UseDatabaseNullSemantics is true, respectively (((operand1 = operand2) AND (NOT (operand1 IS NULL OR operand2 IS NULL))) OR ((operand1 IS NULL) AND (operand2 IS NULL))) if UseDatabaseNullSemantics is false.
ValidateOnSaveEnabled	Gets or sets a value indicating whether tracked entities should be validated automatically when SaveChanges is invoked. The default value is true.

Database class

An instance of this class is obtained from an DbContext object and can be used to manage the actual database backing a DbContext or connection. This includes creating, deleting, and checking for the existence of a database.

Property	Description
CommandTimeout	Gets or sets the timeout value, in seconds, for all context operations. The default value is null, where null indicates that the default value of the underlying provider will be used.
Connection	Returns the connection being used by this context. This may cause the connection to be created if it does not already exist.
CurrentTransaction	Gets the transaction the underlying store connection is enlisted in. May be null.
Log	Set this property to log the SQL generated by the DbContextto the given delegate. For example, to log to the console, set this property to Write.

Method	Description
BeginTransaction()	Begins a transaction on the underlying store connection
ExecuteSqlCommand (string,Object[])	Executes the given DDL/DML command against the database.
Initialize(Boolean)	Runs the registered IDatabaseInitializer<TContext> on this context.
SetInitializer<TContext> (IdatabaseInitializer <TContext>)	Sets the database initializer to use for the given context type. The database initializer is called when a the given DbContext type is used to access a database for the first time.
SqlQuery(Type,String, Object[])	Creates a raw SQL query that will return elements of the given type.
SqlQuery<TElement> (String,Object[])	Creates a raw SQL query that will return elements of the given generic type.
UseTransaction (DbTransaction)	Enables the user to pass in a database transaction created outside of the Database object if you want the Entity Framework to execute commands within that external transaction. Alternatively, pass in null to clear the framework's knowledge of that transaction.

Querying

You can query EDM mainly by three ways, 1) LINQ to Entities 2) Entity SQL 3) Native SQL

LINQ to Entities

L2E query syntax is easier to learn than Entity SQL. You can use your LINQ skills for querying with EDM. These are LINQ Method Syntax with Lambda expression and LINQ query syntax.

```
using (var context = new College())
{
 var L2EQuery = context.Students.where(s => s.Name == "Bill");
 var L2EQuery2 = from st in context.Students
                 where st.Name == "Bill" select st;
}
```

Entity SQL

Entity SQL is another way to create a query. 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.

```
string sqlString = "SELECT VALUE st FROM College.Students AS st "  +
                   "WHERE st.Name == 'Bill'";

var objctx = (ctx as IObjectContextAdapter).ObjectContext;
ObjectQuery<Student> student = objctx.CreateQuery<Student>(sqlString);
Student newStudent = student.First<Student>();
```

Native SQL

You can execute native SQL queries for a relational database as shown below:

```
using (var ctx = new College())
{
   var courses = ctx.Courses.SqlQuery("select * from courses").ToList<Course>();

   // List Courses
   foreach(var c in courses)
   {
     Console.WriteLine(c.Title);
   }

   // Get count of courses
   var count = ctx.Database.SqlQuery<Int32>
               ("select count(*) from courses").First<Int32>();

   Console.WriteLine(count);

   // Hike course fee for all courses
   var updateCount = ctx.Database.ExecuteSqlCommand
           ("update courses set title = upper(title)");

   Console.WriteLine(updateCount);
}
```

It is possible to call stored procedures using SqlQuery method as follows :

```
var courses = ctx.Courses.SqlQuery("dbo.GetCostlyCourses").ToList();
```

TRANSACTION MANAGEMENT

- Whenever you execute SaveChanges() to insert, update or delete on the database the framework will wrap that operation in a transaction. This transaction lasts only long enough to execute the operation and then completes
- Database.ExecuteSqlCommand() by default will wrap the command in a transaction if one was not already present.

It is possible to create or use existing transaction as follows:

- Database.**BeginTransaction**() : An easier method for a user to start and complete transactions themselves within an existing DbContext – allowing several operations to be combined within the same transaction and hence either all committed or all rolled back as one. It also allows the user to more easily specify the isolation level for the transaction.
- Database.BeginTransaction() has two overrides – one which takes an explicit IsolationLevel and one which takes no arguments and uses the default IsolationLevel from the underlying database provider. Both overrides return a DbContextTransaction object which provides Commit() and Rollback() methods which perform commit and rollback on the underlying store transaction.
- Database.**UseTransaction**() : Allows the DbContext to use a transaction which was started outside of the Entity Framework.

```
using (var dbContextTransaction = context.Database.BeginTransaction())
{
    try
    {
        context.Database.ExecuteSqlCommand(
            @"UPDATE products SET price  =  price * 1.1 WHERE  price < 1000"
                            );
        var query = context.Products.Where(p => p.Price >= 20000);
        foreach (var prod in query)
        {
            prod.Category = "Costly";
        }

        context.SaveChanges();
        dbContextTransaction.Commit();
    }
    catch (Exception)
    {
         dbContextTransaction.Rollback();
    }
}
```

CODE FIRST CONVENTIONS

The following are important conventions related to Code First work flow.

Primary Key

- The default convention for primary key is that Code-First would create a primary key for a property if the property name is Id or <class name>Id (NOT case sensitive).
- The data type of a primary key property can be anything, but if the type of the primary key property is numeric or GUID, it will be configured as an identity column.

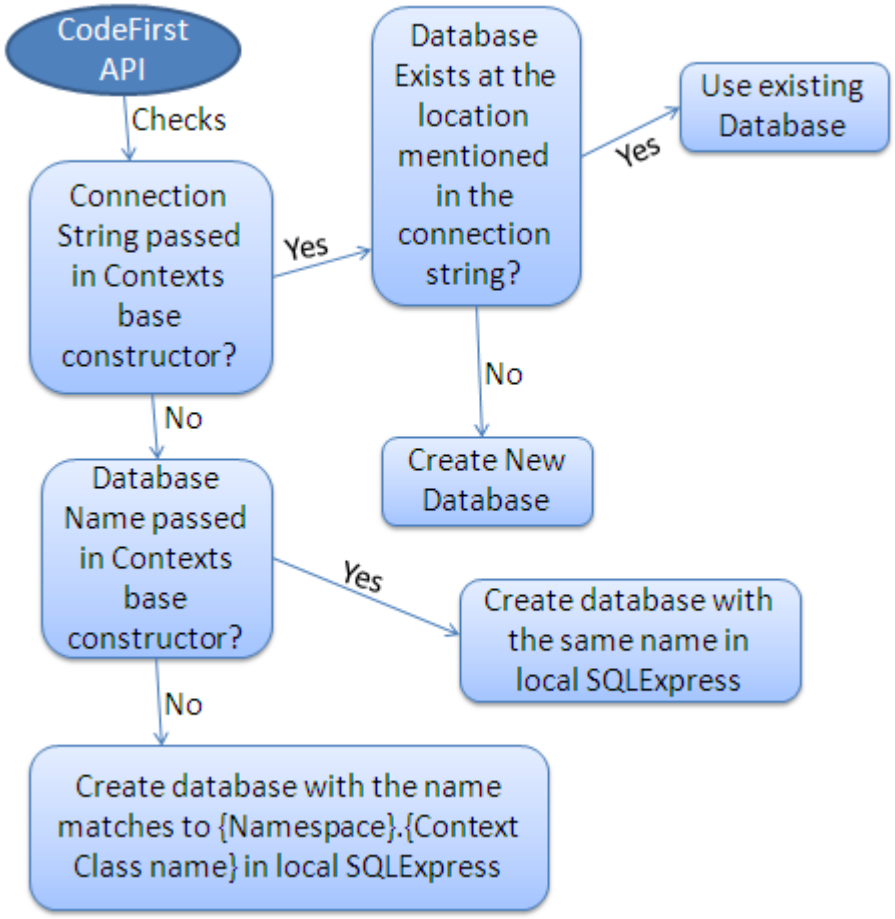
Relationship

- Code First infers the relationship between the two entities using navigation property.
- This navigation property can be simple reference type or collection type.
- Thus, the default code first convention for relationship automatically inserted a foreign key with <navigation property Name>_<primary key property name of navigation property type> e.g. Employee_DepartmentId.
- It is recommended to include a foreign key property on the dependent end of a relationship.
- If Employee class includes foreign key DepartmentID, which is the key property in Department class then Code First will create DepartmentId column in Employee class instead of Department_DepartmentId column.
- Code First infers the multiplicity of the relationship based on the nullability of the foreign key. If the property is nullable then the relationship is registered as null. Otherwise, the relationship is registered as NOT NULL. Modify data type of StandardId property from int to Nullable<int> in the Student class above to create a nullable foreign key column in the Students table.

Default Convention For	Description
Table Name	<Entity Class Name> + 's' EF will create DB table with entity class name suffixed by 's'
Primary key Name	1) Id 2) <Entity Class Name> + "Id" (case insensitive) EF will create primary key column for the property named Id or <Entity Class Name> + "Id" (case insensitive)
Foreign key property Name	By default EF will look for foreign key property with the same name as principal entity primary key name. If foreign key property does not exist then EF will create FK column in Db table with <Dependent Navigation Property Name> + "_" + <Principal Entity Primary Key Property Name>
Null column	EF creates null column for all reference type properties and nullable primitive properties.
Not Null Column	EF creates NotNull columns for PrimaryKey properties and non-nullable value type properties.
DB Columns order	EF will create DB columns same as order of properties in an entity class. However, primary key columns would be moved first.
Properties mapping to DB	By default all properties will map to database. Use [NotMapped] attribute to exclude property or class from DB mapping.
Cascade delete	Enabled By default for all types of relationships.

Database Creation

EF decides where and what database is to be created based on the following flow chart.



How to Enable Automatic Migration

- Whenever the model changes, it is possible for EF to update Database Schema using Migration feature.
- Automatic Migrations allows you to use Code First Migrations without having a code file in your project for each change you make

Take the following steps to turn on Auto Migration feature.

- Go to Package Console using ***Tools -> Nuget Package Manager -> Package Manager Console***
- Execute Enable-Migrations command to enable migrations feature

```
PM> enable-migrations -EnableAutomaticMigrations:$true
```

- This command has added a Migrations folder to our project. This new folder contains a Configuration class. This class allows you to configure how Migrations behave for your context.

```
internal sealed class Configuration:MigrationsConfiguration<examples.HRDbContext>
{
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
        }
        protected override void Seed(examples.HRDbContext context)
        {
        }
}
```

- EF has created one system table __MigrationHistory along with other tables. This is where automated migration maintains the history of database changes.
- Use **update-database** command in Package Manager Console to update changes made to model to corresponding tables in database.

Code First Migration

Code First Migrations has two commands that you are going to become familiar with.

Add-Migration

Will scaffold the next migration based on changes you have made to your model.

```
PM>add-migration "qty added"
```

Update-Database

This command will apply any pending changes to the database.

Note: It is better to avoid using Add-Migration (unless we really need to) and focus on letting Code First Migrations automatically calculate and apply the changes.

```
PM> update-database -verbose
```

Code-based Migration:

Code-based migration is useful when you want more control on the migration, i.e. set default value of the column, etc.

Code-First has two commands for code based migration:

Add-migration

It will scaffold the next migration for the changes you have made to your domain classes

Update-database

It will apply pending changes to the database based on latest scaffolding code file you created using "Add-Migration" command

Turn Off Database Initialization

It is possible to disable migration process that takes place automatically using the following code:

```
public class HRDBContext: DbContext
{
    public HRDBContext() : base("name=hrcon")
    {
        //Disable initializer
        Database.SetInitializer<HRDBContext>(null);
}
```

DATA ANNOTATIONS

The following are important data annotations using which we can configure Domain classes.

Attribute	Description
Key	Mark property as EntityKey which will be mapped to PK of the related table.
Timestamp	Mark the property as a non-nullable timestamp column in the database.
ConcurrencyCheck	ConcurrencyCheck annotation allows you to flag one or more properties to be used for concurrency checking in the database when a user edits or deletes an entity.
Required	The Required annotation will force EF (and MVC) to ensure that property has data in it.
MinLength	MinLength annotation validates property whether it has minimum length of array or string.
MaxLength	MaxLength annotation is the maximum length of property which in turn sets the maximum length of a column in the database
StringLength	Specifies the minimum and maximum length of characters that are allowed in a data field.
Table	Specify name of the DB table which will be mapped with the class
Column	Specify column name and datatype which will be mapped with the property
Index	Create an Index for specified column. (EF 6.1 onwards only)
ForeignKey	Specify Foreign key property for Navigation property
NotMapped	Specify that property will not be mapped with database
DatabaseGenerated	DatabaseGenerated attribute specifies that property will be mapped to computed column of the database table. So, the property will be read-only property. It can also be used to map the property to identity column (auto incremental column).
InverseProperty	InverseProperty is useful when you have multiple relationships between two classes.
ComplexType	Mark the class as complex type in EF.

FLUENT API

- There are two main ways you can configure EF to use something other than conventions, namely annotations or EFs fluent API.
- The annotations only cover a subset of the fluent API functionality, so there are mapping scenarios that cannot be achieved using annotations.
- The code first fluent API is most commonly accessed by overriding OnModelCreating method on your derived DbContext.

The following are different examples of fluent API.

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
   // changes default schema to the given schema
   modelBuilder.HasDefaultSchema("sales");
}
```

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t => t.InstructorID);
```

```
// Configure composite primary key
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name });
```

```
// Switch off Identity for numeric primary keys
modelBuilder.Entity<Department>().Property(t => t.DepartmentID)
    .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
```

```
// Not to map a property
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

```
// Map a property to specific column
modelBuilder.Entity<Department>()
    .Property(t => t.Name)
    .HasColumnName("DepartmentName");
```

```
// Specify type of column
modelBuilder.Entity<Department>()
    .Property(p => p.Name)
    .HasColumnType("varchar");
```

```
// Use a property as optimistic concurrency token
modelBuilder.Entity<OfficeAssignment>()
    .Property(t => t.Timestamp)
    .IsConcurrencyToken();
```

```
// Map an entity to specific table
modelBuilder.Entity<Department>()
    .ToTable("t_Department");
```

```
// Entity splitting – One entity properties map to multiple tables
modelBuilder.Entity<Department>()
    .Map(m =>
    {
        m.Properties(t => new { t.DepartmentID, t.Name });
        m.ToTable("Department");
    })
    .Map(m =>
    {
        m.Properties(t => new { t.DepartmentID, t.Administrator,
                                t.StartDate, t.Budget });
        m.ToTable("DepartmentDetails");
    });
```

```
// Table splitting - Maps two entity types that share a primary key to one table
modelBuilder.Entity<OfficeAssignment>()
    .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t => t.Instructor);

modelBuilder.Entity<Instructor>().ToTable("Instructor");

modelBuilder.Entity<OfficeAssignment>().ToTable("Instructor");
```

EntityTypeConfiguration class

Allows configuration to be performed for an entity type in a model. An EntityTypeConfiguration can be obtained via the Entity method on DbModelBuilder.

Method	Description
HasEntitySetName(String)	Configures the entity set name to be used for this entity type. The entity set name can only be configured for the base type in each set.
HasKey<TKey> (Expression<Func<TKey TEntityType,>>)	Configures the primary key property(s) for this entity type.
HasMany<TTargetEntity> (Expression<Func<ICollection TEntityType,<TTargetEntity>>>)	Configures a many relationship from this entity type.
HasOptional<TTargetEntity> (Expression<Func<TTargetEntity TEntityType,>>)	Configures an optional relationship from this entity type. Instances of the entity type will be able to be saved to the database without this relationship being specified. The foreign key in the database will be nullable.
HasRequired<TTargetEntity> (Expression<Func<TTargetEntity TEntityType,>>)	Configures a required relationship from this entity type. Instances of the entity type will not be able to be saved to the database unless this relationship is specified. The foreign key in the database will be non-nullable.
HasTableAnnotation (String,Object)	Sets an annotation in the model for the table to which this entity is mapped. The annotation value can later be used when processing the table such as when creating migrations.
Ignore<TProperty> (Expression<Func<TProperty TEntityType,>>)	Excludes a property from the model so that it will not be mapped to the database.
Map(Action <EntityMappingConfiguration <TEntityType>>)	Allows advanced configuration related to how this entity type is mapped to the database schema. By default, any configuration will also apply to any type derived from this entity type. Derived types can be configured via the overload of Map that configures a derived type or by using an EntityTypeConfiguration for the derived type. The properties of an entity can be split between multiple tables using multiple Map calls. Calls to Map are additive, subsequent calls will not override configuration already preformed via Map.
Map<TDerived>(Action <EntityMappingConfiguration <TDerived>>)	Allows advanced configuration related to how a derived entity type is mapped to the database schema. Calls to Map are additive, subsequent calls will not override configuration already preformed via Map.
MapToStoredProcedures()	Configures this type to use stored procedures for insert, update and delete. The default conventions for procedure and parameter names will be used.
Property<T>(Expression <Func<T TStructuralType,>>)	Configures a T:System.struct property that is defined on this type.(Inherited from StructuralTypeConfiguration<TStructuralType>.)
ToTable(String)	Configures the table name that this entity type is mapped to.

```
EntityTypeConfiguration<Course> courseConfig = modelBuilder.Entity<Course>();
courseConfig.HasKey(c => c.CourseId);
```

ENTITY SQL

Entity SQL is a text-based, provider independent, late bound query language used by Entity Framework to express queries in terms of EDM abstractions and to query data from the conceptual layer of the Entity Data Model.

```
class EntitySQLDemo
{
        static void Main(string[] args)
        {
            HRDbContext ctx = new HRDbContext();

            var qry = ctx.ObjectContext().CreateQuery<Department>
                    ("select VALUE dept from HRDBContext.Departments as dept");

            foreach (var d in qry.ToList())
                Console.WriteLine(d.DepartmentName);

            // Selection

            var empquery = ctx.ObjectContext().CreateQuery<Employee>
                ("select VALUE e from HRDBContext.Employees as e
                     where e.salary > 20000 order by e.EmployeeName ");

            foreach (var e in empquery.ToList())
                Console.WriteLine(e.EmployeeName);


            // Projection

            var namequery = ctx.ObjectContext().CreateQuery<DbDataRecord>
             ("select e.EmployeeName,e.Salary from HRDBContext.Employees as e");

            foreach (var e in namequery.ToList())
                Console.WriteLine("{0} - {1}" , e["EmployeeName"], e["Salary"]);
        }
}
```

CALLING STORED PROCEDURE

A stored procedure can be invoked using SqlQuery method of DbSet.

Assume we have the following stored procedure, which returns a collection of products. EF converts rows of Products to objects of Product class.

```
CREATE PROCEDURE [dbo].CostlyProducts

AS
      SELECT * from products where price > 10000
```

The following code invokes CostlyProducts procedure.

```
CREATE PROCEDURE [dbo].CostlyProducts(@price int)

AS
      SELECT * from products where price > @price
```

```
class CallCostlyProducts
{
    static void Main(string[] args)
    {
          HRContext ctx = new HRContext();

          var products = ctx.Products.SqlQuery("CostlyProducts").ToList();

          foreach (var p in products)
                Console.WriteLine(p.Name);

    }
}
```

If procedure expects any parameters then we can pass them as follows:

```
SqlParameter price  = new SqlParameter("@price",1000);
var products = ctx.Products.SqlQuery("CostlyProducts @price", price ).ToList();
```

Procedure returning non-entities

When a procedure returns non-entities then we have to map that to either a scalar type or a user-defined class that matches what procedure returns.

```
CREATE PROCEDURE [dbo].GetSellingPrice
      AS
select  Name, SellingPrice = price * 1.18 from products
order by Name
```

```
using examples;
using System;
using System.Linq;

namespace codefirstdemo
{
    class CallGetSellingPrice
    {
        static void Main(string[] args)
        {
            HRContext ctx = new HRContext();
            var products = ctx.Database.SqlQuery<ProductWithGst>
                                ("GetSellingPrice").ToList();

            foreach (var p in products)
                Console.WriteLine("{0} {1}", p.Name, p.SellingPrice);
        }
        private class ProductWithGst
        {
            public string Name { get; set; }
            public decimal SellingPrice { get; set; }
        }
    }
}
```

Calling a procedure that executes DML

When we need to execute non-query then we need to use ExecuteSqlCommand() method of Database object as shown in the following example:

```
CREATE PROCEDURE [dbo].[UpdatePrice](@id int,@price int)
AS
     Update products set price = @price where id = @id

     if @@rowcount = 0
       raiserror ('Invalid Product Id',16,1)
```

```
using System.Data.SqlClient;
namespace codefirstdemo
{
    class UpdatePrice
    {
        static void Main(string[] args)
        {
            EFContext ctx = new EFContext();

            SqlParameter id  = new SqlParameter("@id",3);
            SqlParameter price = new SqlParameter("@price", 3000);
            ctx.Database.ExecuteSqlCommand("updateprice @id, @price", id, price);
        }
    }
}
```

INHERITANCE

- Inheritance is available in Object model and not in Relational model.
- Entity framework supports three different approaches to represent inheritance of Object model in Relational model.
- Each model has its own pros and cons.

Below are three different approaches to represent an inheritance hierarchy in Code-First:

Table per Hierarchy (TPH)

This approach creates one table for the entire class inheritance hierarchy. The single table contains discriminator column which distinguishes between derived classes. This is a default inheritance mapping strategy in Entity Framework.

Table per Type (TPT)

This approach suggests a separate table for each domain class and one table for abstract class. Whenever we retrieve data, we join table related to base class with tables of derived class.

Table per Concrete class (TPC)

This approach suggests one table for one concrete class, but not for the abstract class. So, if you inherit the abstract class in multiple concrete classes, then the properties of the abstract class will be part of each table of the concrete class.

Example classes

The following are three classes used to demonstrate three different strategies.

```
class Course
{
  public int Id { get; set; }
  public string Name { get; set; }
  public int Fee { get; set; }
}
```

```
class OfflineCourse : Course
{
   public string Place { get; set; }
}
```

```
class OnlineCourse : Course
{
   public string Url { get; set; }
}
```

```
class TrainingContext : DbContext
{
  public TrainingContext():
      base(@"Data Source=(LocalDB)\MSSQLLocalDB;AttachDbFilename=
      C:\dev\msnet\ef\codefirstdemo\st.mdf;Integrated Security=True")
  {
  }
  public DbSet<Course> Courses { get; set; }
}
```

Table per Hierarchy (TPH)

- All properties of all classes are mapped to columns in one table
- This mapping strategy provides better performance and simplicity
- Columns representing properties of subclass must be nullable
- Violates Third normal form as discriminator column is not dependent on primary key

When Entity Framework creates database, it creates a single table with name COURSES with following structure:

```
CREATE TABLE [dbo].[Courses] (
    [Id]          INT             IDENTITY (1, 1) NOT NULL,
    [Name]        NVARCHAR (MAX) NULL,
    [Fee]         INT             NOT NULL,
    [Place]       NVARCHAR (MAX) NULL,
    [Url]         NVARCHAR (MAX) NULL,
    [Discriminator] NVARCHAR (128) NULL,
    CONSTRAINT [PK_dbo.Courses] PRIMARY KEY CLUSTERED ([Id] ASC)
);
```

The following program creates two courses and inserts them into same table.

```
TrainingContext ctx = new TrainingContext();
var c1 = new OfflineCourse { Name = "Java", Fee = 5500,
                     Place = "Srikanth Technologies, Vizag" };
var c2 = new OnlineCourse  { Name = "Angular", Fee = 4000,
                     Url = "https://global.gotomeeting.com/join/698399013" };
ctx.Courses.Add(c1);
ctx.Courses.Add(c2);
ctx.SaveChanges();
```

It is possible to change name and possible values of discriminator column using Fluent API as follows:

```
class TrainingContext : DbContext
{
   // other code
   protected override void OnModelCreating(DbModelBuilder modelBuilder)
   {
    // Configure discriminator column name and values for derived classes
    modelBuilder.Entity<Course>()
         .Map<OnlineCourse>(m => m.Requires("CourseType").HasValue("online"))
         .Map<OfflineCourse>(m => m.Requires("CourseType").HasValue("offline"));
  }

}
```

The structure and data in the course table will be as follows:

```
CREATE TABLE [dbo].[Courses] (
    [Id]          INT             IDENTITY (1, 1) NOT NULL,
    [Name]        NVARCHAR (MAX) NULL,
    [Fee]         INT             NOT NULL,
    [Place]       NVARCHAR (MAX) NULL,
    [Url]         NVARCHAR (MAX) NULL,
    [CourseType] NVARCHAR (128) NULL,
    CONSTRAINT   [PK_dbo.Courses] PRIMARY KEY CLUSTERED ([Id] ASC)
);
```

```
Id Name        Fee      Place                       Url
CourseType
```

```
-------------------------------------------------------------------------------
------------------------
1  Java    3500  Srikanth Technologies, Vizag NULL
offline
2  Angular 10000 NULL                        https://global.gotomeeting.com/join/698399013
online
```

Table per Type (TPT)

- In this strategy multiple tables are created. One table is created for each class in the hierarchy
- Specify table name for derived classes in the hierarchy to implement TPT strategy
- The table for subclasses contains columns only for each non-inherited property
- Every query needs a join between base class table and derived class tables
- Tables are normalized
- Tables can accommodate changes to corresponding class without effecting other tables
- Performance can be unacceptable for complex class hierarchies because queries always require a join across many tables

```
class Course
{
  public int Id { get; set; }
  public string Name { get; set; }
  public int Fee { get; set; }
}

[Table ("OfflineCourses")]
class OfflineCourse : Course
{
  public string Place { get; set; }
}

[Table("OnlineCourses")]
class OnlineCourse : Course
{
    public string Url { get; set; }
}
```

Alternatively you can use fluent API as follows:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<OfflineCourse>().ToTable("OfflineCourses");
    modelBuilder.Entity<OnlineCourse>().ToTable("OnlineCourses");
}
```

```
CREATE TABLE [dbo].[Courses] (
    [Id]   INT            IDENTITY (1, 1) NOT NULL,
    [Name] NVARCHAR (MAX) NULL,
    [Fee]  INT            NOT NULL,
    CONSTRAINT [PK_dbo.Courses] PRIMARY KEY CLUSTERED ([Id] ASC)
);

CREATE TABLE [dbo].[OfflineCourses] (
    [Id]    INT            NOT NULL,
    [Place] NVARCHAR (MAX) NULL,
    CONSTRAINT [PK_dbo.OfflineCourses] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.OfflineCourses_dbo.Courses_Id] FOREIGN KEY ([Id])
                                    REFERENCES [dbo].[Courses] ([Id])
);

CREATE TABLE [dbo].[OnlineCourses] (
    [Id]  INT            NOT NULL,
```

```
    [Url] NVARCHAR (MAX) NULL,
    CONSTRAINT [PK_dbo.OnlineCourses] PRIMARY KEY CLUSTERED ([Id] ASC),
    CONSTRAINT [FK_dbo.OnlineCourses_dbo.Courses_Id] FOREIGN KEY ([Id])
                                    REFERENCES [dbo].[Courses] ([Id])
);
```

Table per Concrete Type (TPC)

- In Table per Concrete type (aka Table per Concrete class), we create exactly one table for each (nonabstract) class.
- All properties of a class, including inherited properties, can be mapped to columns of this table.
- Any change to base class needs a change to all tables based on derived classes
- Any query related to base class will result in Union of all tables on derived classes

```
 abstract class Course
 {
        [DatabaseGenerated(DatabaseGeneratedOption.None)]
        public int Id { get; set; }
        public string Name { get; set; }
        public int Fee { get; set; }
 }
 class OfflineCourse : Course
 {
        public string Place { get; set; }
 }
 class OnlineCourse : Course
 {
       public string Url { get; set; }
 }
```

```
class TrainingContext : DbContext
{
        // other code

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<OfflineCourse>().Map(m =>
            {
                m.MapInheritedProperties();
                m.ToTable("OfflineCourses");
            });

            modelBuilder.Entity<OnlineCourse>().Map(m =>
            {
                m.MapInheritedProperties();
                m.ToTable("OnlineCourses");
            });
        }
}
```

```
class AddCourses
{
   static void Main(string[] args)
   {
     TrainingContext ctx = new TrainingContext();

     var c1 = new OfflineCourse {Id = 1, Name = "Java",
          Fee = 3500, Place = "Srikanth Technologies, Vizag" };
     var c2 = new OnlineCourse { Id = 2, Name = "Java",
          Fee = 10000, Url = "https://global.gotomeeting.com/join/698399013" };
```

```
     ctx.Courses.Add(c1);
     ctx.Courses.Add(c2);
     ctx.SaveChanges();
    }
}
```

```
CREATE TABLE [dbo].[OfflineCourses] (
    [Id]    INT            NOT NULL,
    [Name]  NVARCHAR (MAX) NULL,
    [Fee]   INT            NOT NULL,
    [Place] NVARCHAR (MAX) NULL,
    CONSTRAINT [PK_dbo.OfflineCourses] PRIMARY KEY CLUSTERED ([Id] ASC)
);

CREATE TABLE [dbo].[OnlineCourses] (
    [Id]   INT            NOT NULL,
    [Name] NVARCHAR (MAX) NULL,
    [Fee]  INT            NOT NULL,
    [Url]  NVARCHAR (MAX) NULL,
    CONSTRAINT [PK_dbo.OnlineCourses] PRIMARY KEY CLUSTERED ([Id] ASC)
);
```