Entity Framework

Entity Framework – Topic of Contents

What is Entity Framework

- **Entity Framework** (**EF**) is an open source object-relational mapping (ORM) framework for ADO.NET. It is one of the data access framework provided by Microsoft. We can use this framework to access the data from the database. We can also add, remove or update data into the database.
- The Entity Framework is a set of technologies in ADO.NET that support the development of data-oriented software applications with less code as compared to traditional database applications. The Entity Framework allows developers to work at a higher level of abstraction when they deal with data in the form of domain-specific object and properties (such as instance of customer and it's property customer address), without having to deal with the underlying database tables and columns where this data is stored.
- Entity Framework can be use with any type of application say Console Application, Windows Service, Web Forms, Web API, MVC etc.

Object Relational Mapping

Understanding Object-relational mapping (ORM)

- **Object-relational mapping (ORM)** is a mechanism that makes it possible to address, access and manipulate objects without having to consider how those objects relate to their data sources. ORM lets programmers maintain a consistent view of objects over time, even as the sources of data, the container that receive them and the applications that access them change.

- Based on abstraction, ORM manages the mapping details between a set of objects and underlying relational databases, XML repositories or other data sources

List of available ORM in .NET

- Dapper, open source
- **Entity Framework**
- **LINQ to SQL**
- NHibernate, open source
- nHydrate, open source

Benefits of Entity Framework

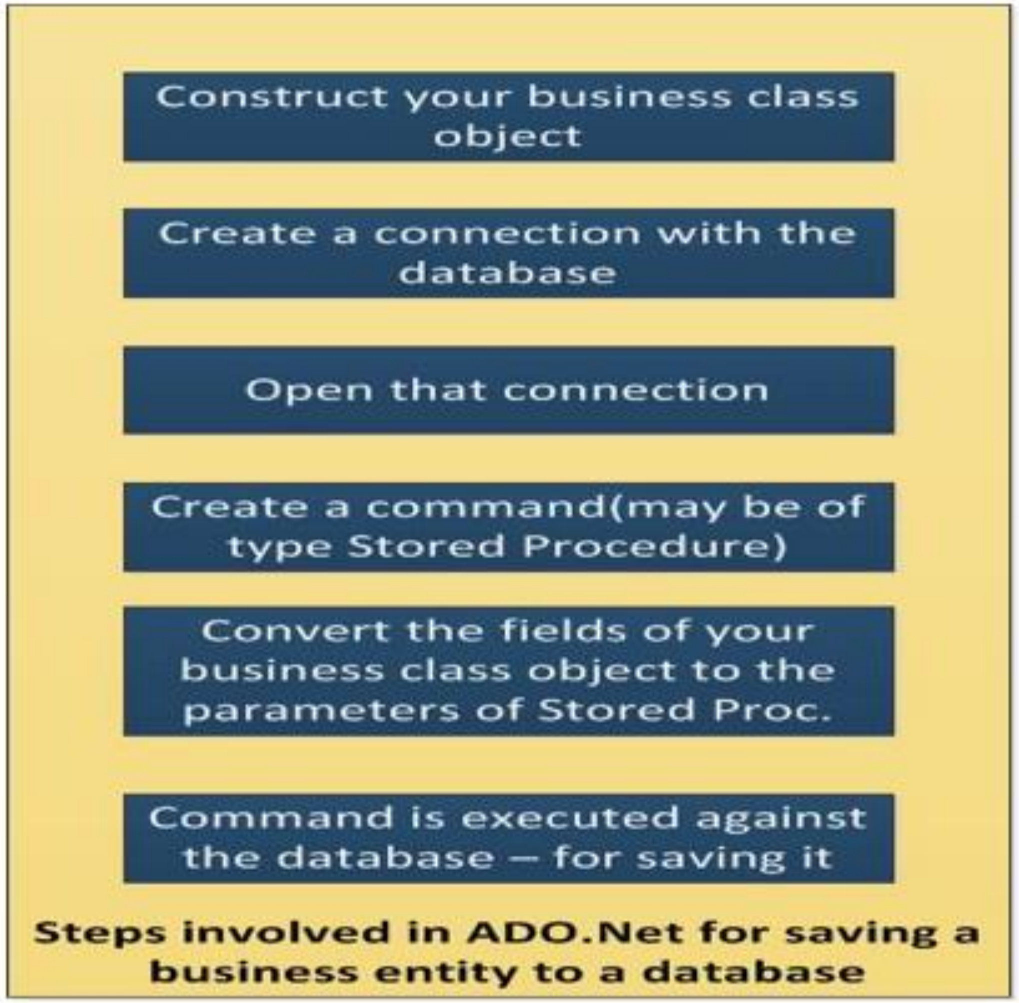
Why we have to go for this new framework when we have been doing fine with ADO.Net?

Entity framework is an ORM (object relational mapping) which creates a higher abstract object model over ADO.NET components. So rather than getting into dataset, data tables, command and connection objects, we work on higher level domain objects like instance of customers, suppliers, etc.

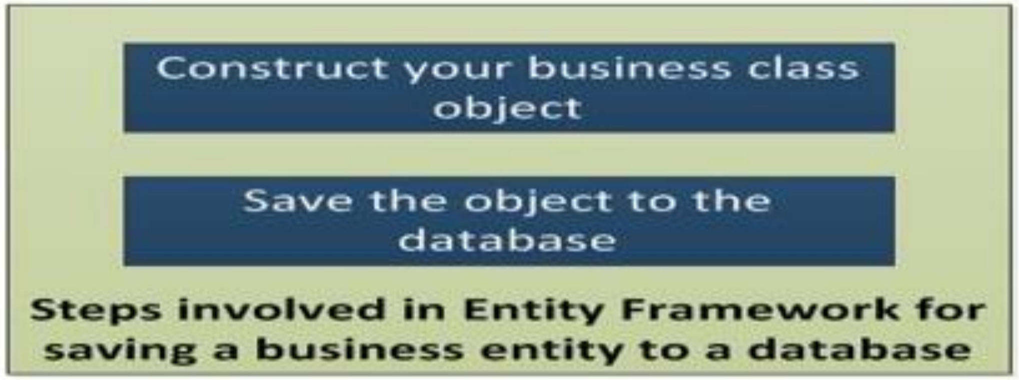
Important benefits of entity framework are

- Applications can work in terms of much more application-centric conceptual model, including types with inheritance, complex members and relationships.
- Applications are freed from hard-coded dependencies on a particular data engine or storage schema.
- Mappings between the conceptual model and the storage-specific schema can change without changing the application code.
- Multiple conceptual models can be mapped to a single storage schema.
- Language-integrated query (LINQ) support provides compile-time syntax validation for queries against a conceptual model.

Comparing with ADO.Net, how data can be accessed using Entity Framework?


Construct your business class object
Create a connection with the database
Open that connection
Create a command(may be of type Stored Procedure)
Convert the fields of your business class object to the parameters of Stored Proc.
Command is executed against the database – for saving it

Steps involved in ADO.Net for saving a business entity to a database


Construct your business class object
Save the object to the database

Steps involved in Entity Framework for saving a business entity to a database

Basic Workflow in Entity Framework

1. First, we need to define the model. Defining the model consists of domain classes, context classes derived from DbContext and configuration.

2. To insert data, we need to add a domain object for a context and call the savechanges() method. We need to use the insert command and execute it to the database.

3. For reading data, executing the LINQ-to-Entities query in your preferred language like C# or .NET will be useful. EF API will convert the query to the SQL query, which will be provided to the database for execution.

4. For editing, updating, deleting, and removing entities objects, we should call the savechanges() method. EF API will build and execute the commands in the database.

Types of Entity Framework approaches used for accessing data

We can deal with three approaches that Microsoft Entity Framework provides. The three approaches are as follows,

1. Model First,
2. Database First, and
3. Code First

Model-First approach says that we have a model with all kinds of entities and relations/associations using which we can generate a database that will eventually have entities and properties converted into database tables and the columns and associations and relations would be converted into foreign keys respectively.

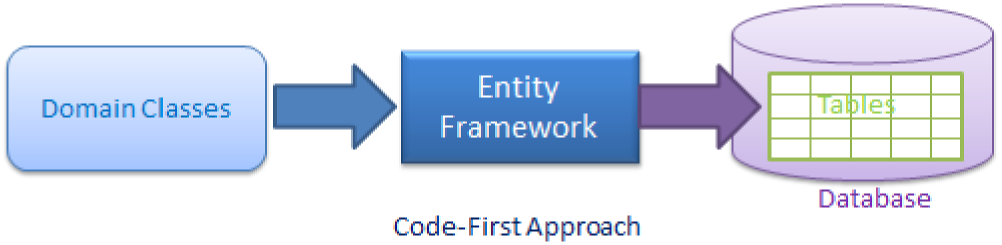
Using a Model-First approach, a developer may not need to write any code for generating a database. Entity Framework provides the designer tools that could help us make a model and then generate a database out of it. The tools are more of a drag and drop controls that just need inputs like what our entity name is, what properties it should have, how it is related to other entities and so.

Types of Entity Framework approaches used for accessing data

The **Database-First approach** says that we already have an existing database and we need to access that database in our application. We can create an entity data model along with its relationship directly from the database with just a few clicks and start accessing the database from our code. All the entities, i.e., classes, would be generated by EF that could be used in the application's data access layer to participate in DB operation queries. It is the opposite of a Model-First approach. Here, a model is created via a database and we have full control to choose what tables to include in the model, and what stored procedures, functions, or views to include.

The **Code-First approach** is the recommended approach with EF, especially when we are starting the development of an application from scratch. We can define the POCO classes in advance and their relationships and envision how our database structure and data model may look like by just defining the structure in the code. Entity Framework, at last, will take all the responsibility to generate a database for us for our POCO classes and for the data model and will take care of transactions, history, and migrations.

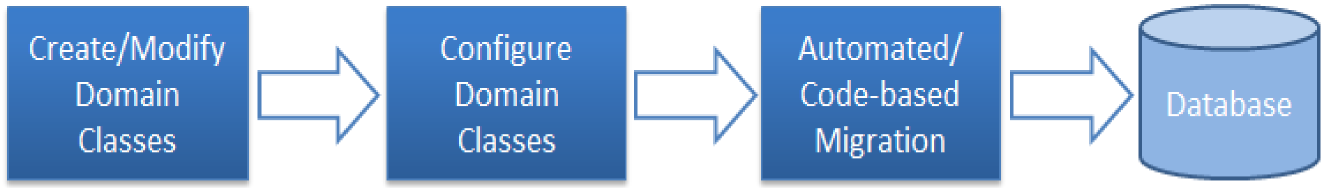
Entity Framework Code First Approach



As we can see in the above figure, EF API will create the database based on our domain classes and configuration. This means we need to start coding first in C# or VB.NET and then EF will create the database from our code.

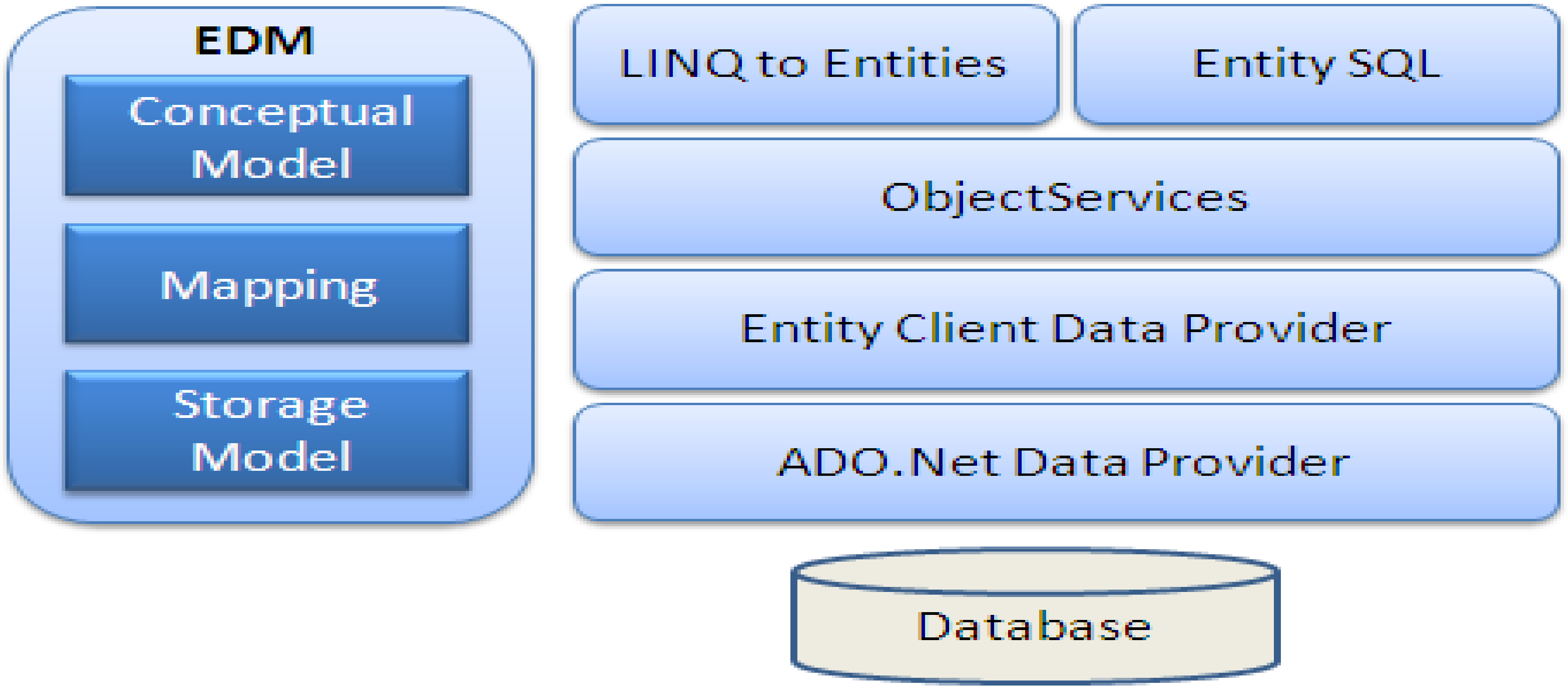
Code-First Workflow

The following figure illustrates the code-first development workflow.



The development workflow in the code-first approach would be: Create or modify domain classes -> configure these domain classes using Fluent-API or data annotation attributes -> Create or update the database schema using automated migration or code-based migration.

Entity Framework Architecture



Entity Framework Architecture

EDM (Entity Data Model): EDM consists of three main parts - Conceptual model, Mapping and Storage model.

Conceptual Model: The conceptual model contains the model classes and their relationships, which are independent from our database table design.

Storage Model: The storage model is the database design model which includes tables, views, stored procedures, and their relationships and keys.

Mapping: Mapping consists of information about how the conceptual model is mapped to the storage model.

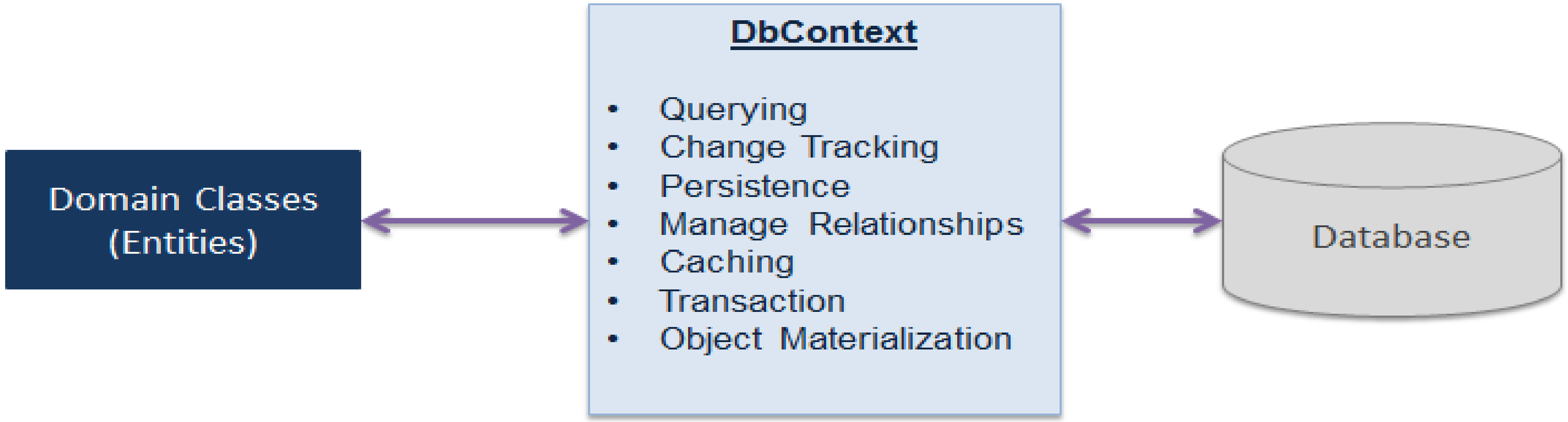
LINQ to Entities / Entity SQL: LINQ-to-Entities is a query language used to write queries against the object model. It returns entities, which are defined in the conceptual model.

Object Service: Object service is a main entry point for accessing data from the database and returning it back. Object service is responsible for materialization, which is the process of converting data returned from an entity client data provider (next layer) to an entity object structure.

Entity Client Data Provider: The main responsibility of this layer is to convert LINQ-to-Entities 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 the database.

ADO.Net Data Provider: This layer communicates with the database using standard ADO.Net.

DbContext is an important class in Entity Framework API. It acts as a bridge between our domain or entity classes and the database.



- Querying
- Change Tracking
- Persistence
- Manage Relationships
- Caching
- Transaction
- Object Materialization

DbContext is the primary class that is responsible for interacting with the database. It is responsible for the following activities:

- **Querying:** Converts LINQ-to-Entities queries to SQL query and sends them to the database.
- **Change Tracking:** Keeps track of changes that occurred on the entities after querying from the database.
- **Persisting Data:** Performs the Insert, Update and Delete operations to the database, based on entity states.
- **Caching:** Provides first level caching by default. It stores the entities which have been retrieved during the life time of a context class.
- **Manage Relationship:** Manages relationships using CSDL, MSL and SSDL in Db-First or Model-First approach, and using fluent API configurations in Code-First approach.
- **Object Materialization:** Converts raw data from the database into entity objects.

DbSet

The DbSet class represents an entity set that can be used for create, read, update and delete operations.

The context class (derived from DbContext) must include the DbSet type properties for the entities which map to database tables and views.

```
using System.Data.Entity;

public class BankAccContext : DbContext
{
    public BankAccContext()
    {
    }

    public DbSet<AccHolder> Accountholder { get; set; }
    public DbSet<AccNo> Accountnumber { get; set; }
    public DbSet<AccType> Accounttype { get; set; }
}
```

What is an **Entity** in Entity Framework?

An entity in Entity Framework is a class that maps to a database table. This class must be included as a DbSet<TEntity> type property in the DbContext class. EF API maps each entity to a table and each property of an entity to a column in the database.

Scalar and Navigation Properties

An Entity can include two types of properties: Scalar Properties and Navigation Properties.

Scalar Property
The primitive type properties are called scalar properties. Each scalar property maps to a column in the database table which stores an actual data. For example, StudentID, StudentName, DateOfBirth, Photo, Height, Weight are the scalar properties in the Student entity class.

Navigation Property
The navigation property represents a relationship to another entity. There are two types of navigation properties: Reference Navigation and Collection Navigation

Reference Navigation Property
If an entity includes a property of another entity type, it is called a Reference Navigation Property. It points to a single entity and represents multiplicity of one (1) in the entity relationships. EF API does will create a ForeignKey column in the table for the navigation properties that points to a PrimaryKey of another table in the database.

Collection Navigation Property

If an entity includes a property of generic collection of an entity type, it is called a collection navigation property. It represents multiplicity of many (*).
EF API does not create any column for the collection navigation property in the related table of an entity, but it creates a column in the table of an entity of generic collection.

Reference Navigation Property

```
    public class Player
    {
     public int Id { get; set; }                  // scalar property
     public string Name { get; set; }             // scalar property
     public string Code { get; set; }             // scalar property
     public int Matches { get; set; }             // scalar property
     public int GoalsScored { get; set; }         // scalar property

     public Country Country { get; set; } // reference navigation property
}
```

Collection Navigation Property

```
public class Country
{
        public string Code { get; set; }          // scalar property
        public string Name { get; set; }          // scalar property

        public ICollection<Player> Players { get; set; }

        // Collection navigation property to be specified later
        // once reference navigation property is specified in Player class
 }
```

Eager Loading in Entity Framework

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

LINQ Query Syntax:

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

LINQ Method Syntax:

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

Lazy Loading in Entity Framework

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

```
using (var ctx = new SchoolDBEntities())
{
IList<Student> studList = ctx.Students.ToList<Student>(); //Loading students only
Student std = studList[0];
//Loads Student address for particular Student only (seperate SQL query)
StudentAddress add = std.StudentAddress;
}
```

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

Rules for lazy loading:

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

Data Annotations in Entity Framework

Data Annotation refers to a simple attribute based configuration, which we can apply to our domain classes and its properties. .

Data annotation attributes are included in the System.ComponentModel.DataAnnotations and System.ComponentModel.DataAnnotations.Schema namespaces in Entity Framework.

System.ComponentModel.DataAnnotations Attributes-
Key, Required, MinLength, MaxLength, RegularExpression, Compare

System.ComponentModel.DataAnnotations.Schema Attributes-
DatabaseGenerated, Table, Column, ForeignKey

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ConsoleAppEF1.Models
{
    public class Country
    {
        [Key]
        public string Code { get; set; }

        [Required,MaxLength(30)]
        public string Name { get; set; }

        public ICollection<Player> Players { get; set; }

       }
}
```

```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace ConsoleAppEF1.Models {
    public class Player
    {
        [Key]
        public int Id { get; set; }
        [Required,MaxLength(30)]
        public string Name { get; set; }
        [ForeignKey("Country")]
        public string Code { get; set; }
        [Required]
        public int Matches { get; set; }
        [Required]
        public int GoalsScored { get; set; }
        public Country Country { get; set; } // reference navigation property
}
}
```

Fluent API

Fluent API is one of the way to configure domain classes by using Entity Framework Fluent API Extension Method. It is based on a Fluent API design pattern, where the result is formulated by method chaining.

Fluent API configuration can be applied when EF builds a model from our domain classes. We can inject the Fluent API configurations by overriding the OnModelCreating method of DbContext in Entity Framework 6.x, as shown below:

```
public class SchoolDBContext: DbContext
{
        public SchoolDBContext(): base("SchoolDBConnectionString") { }
        public DbSet<Student> Students { get; set; }
        public DbSet<Standard> Standards { get; set; }
        public DbSet<StudentAddress> StudentAddress { get; set; }
        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
                //Configure domain classes using modelBuilder here..
        }
}
```

Code First Migrations

Whenever we make some changes and thereby need to update the database schema, we may use the feature called Code First Migrations.

Migrations allows us to have an ordered set of steps that describe how to upgrade (and downgrade) our database schema. Each of these steps, known as a migration, contains some code that describes the changes to be applied.

The first step is to enable Code First Migrations

Tools -> Nuget Package Manager -> Package Manager Console

- 1> Run the **Enable-Migrations** command in Package Manager Console

A new Migrations folder will be added to our project that contains:

- **Configuration.cs** – This file contains the settings that Migrations will use for migrating CustomDbContext. We don't need to change anything for this, but here is where we can specify seed data, register providers for other databases, changes the namespace that migrations are generated in etc.

2> Run the next command as **Add-Migration InitialCreate**

- **<timestamp>_InitialCreate.cs** – This is the migration, it represents the changes that have already been applied to the database to take it from being an empty database to one that includes the tables. Although we let Code First automatically create these tables for us, now that we have opted in to Migrations they have been converted into a Migration. Code First has also recorded in our local database that this Migration has already been applied. The timestamp on the filename is used for ordering purposes.

3> Run the **Update-Database** command in Package Manager Console. This command will apply any pending migrations to the database. Our InitialCreate migration has already been applied so migrations will just apply our new added migration. We can use the **–Verbose** switch when calling Update-Database to see the SQL that is being executed against the database.

Practical Explanations using Visual Studio 2022

Thank you