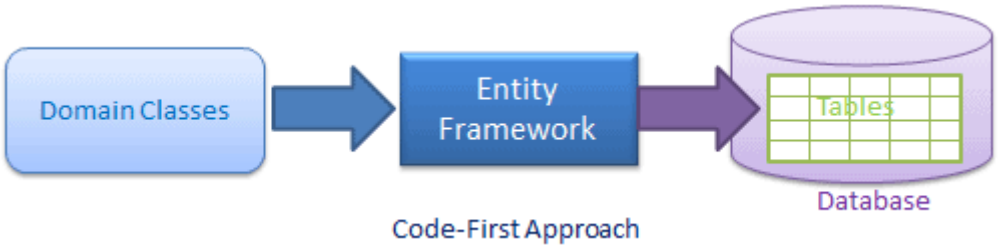
# Entity framework code first Approach

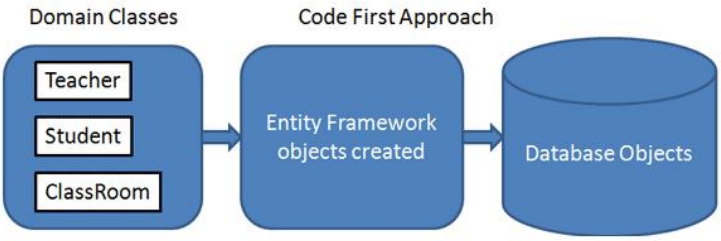
## What is code first approach :

Entity Framework introduced the Code-First approach with Entity Framework 4.1. In the Code-First approach, you focus on the domain of your application and start creating classes for your domain entity rather than design your database first and then create the classes which match your database design. The following figure illustrates the code-first approach.

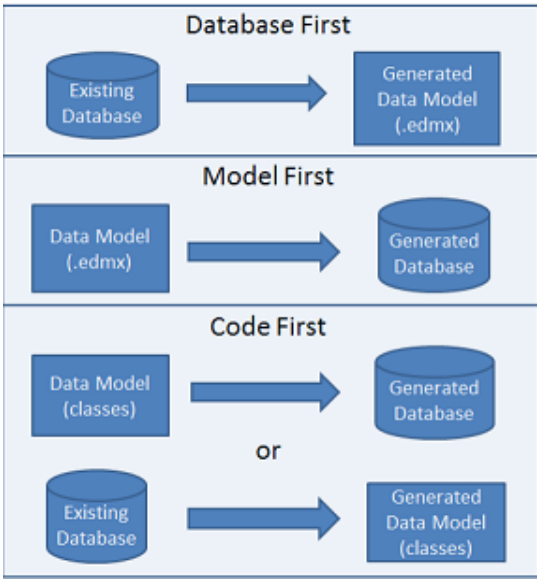



As you can see in the above figure, EF API(Application programming interface) will create the database based on your domain classes and configuration. This means you need to start to coding first in C-Sharp and then entity framework will create the database from your code.

## Example:




## Different between code-first approach and other approach:



Code first

- Very popular because hardcore programmers don't like any kind of designers and defining mapping in EDMX xml is too complex.
- Full control over the code (no autogenerated code which is hard to modify).
- General expectation is that you do not bother with DB. DB is just a storage with no logic. EF will handle creation and you don't want to know how it does the job.
- Manual changes to database will be most probably lost because your code defines the database.

Database first

- Very popular if you have DB designed by DBAs, developed separately or if you have existing DB.
- You will let EF create entities for you and after modification of mapping you will generate POCO entities.
- If you want additional features in POCO entities you must either T4 modify template or use partial classes.
- Manual changes to the database are possible because the database defines your domain model. You can always update model from database (this feature works quite well).
- I often use this together VS Database projects (only Premium and Ultimate version).

Model first

- IMHO popular if you are designer fan (= you don't like writing code or SQL).
- You will "draw" your model and let workflow generate your database script and T4 template generate your POCO entities. You will lose part of the control on both your entities and database but for small easy projects you will be very productive.
- If you want additional features in POCO entities you must either T4 modify template or use partial classes.
- Manual changes to database will be most probably lost because your model defines the database. This works better if you have Database generation power pack installed. It will allow you updating database schema (instead of recreating) or updating database projects in VS.

I expect that in case of EF 4.1 there are several other features related to Code First vs. Model/Database first. Fluent API used in Code first doesn't offer all features of EDMX. I expect that features like stored procedures mapping, query views, defining views etc. works when using Model/Database first and DbContext (I haven't tried it yet) but they don't in Code first.

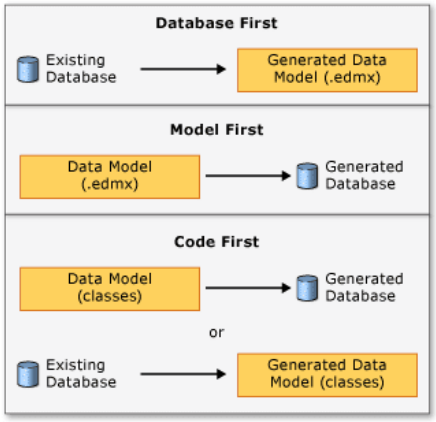
Why Code First? Not others approach:

## reasons why you might use the code first approach:

The workflows you have to choose from are:

1. Code first creating a new database
2. Code first to an existing database
3. Model designer creating a new database
4. Existing database to generated model

Code First can do (database first and model first )




### Greater Control

When you go DB first, you're at the mercy of what gets generated for your models for use in your application. Occasionally the naming convention is undesirable. Sometimes the relationships and associations aren't quite what you want. Other times non transient relationships with lazy loading wreak havoc on your API responses.

While there is almost always a solution for model generation problems you might run into, going code first gives you complete and fine grained control from the get go. You can control every aspect of both your code models and your database design from the comfort of your business object. You can precisely specify relationships, constraints, and associations. You can simultaneously set property character limits and database column sizes. You can specify which related collections are to be eager loaded, or not be serialized at all. In short, you are responsible for more stuff but you're in full control of your app design.

# Getting start with entity framework

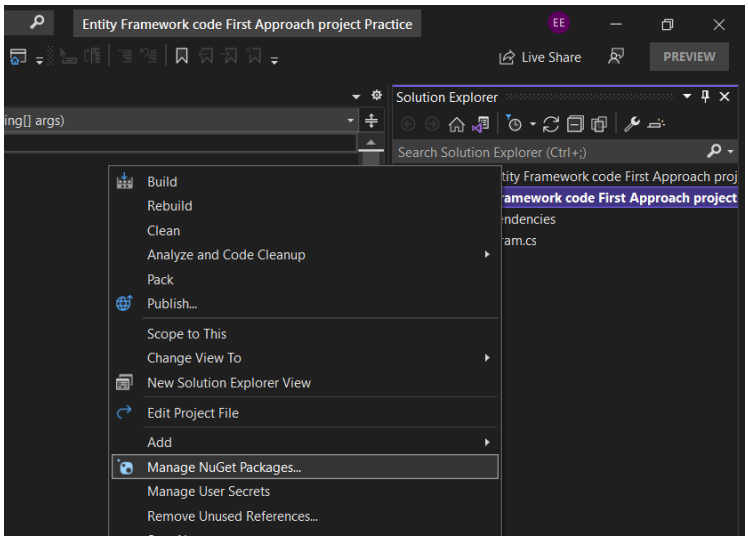
## Environnement Setup :

To start working with EF Code First approach you need the following tools to be installed on your system.

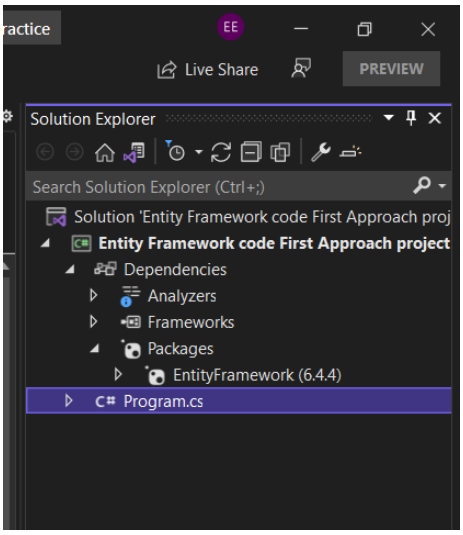
- Visual studio 2019 or 2022
- Sql server
- Entity Framework via NuGet Package. (EF6)

## Install EF internet Required:

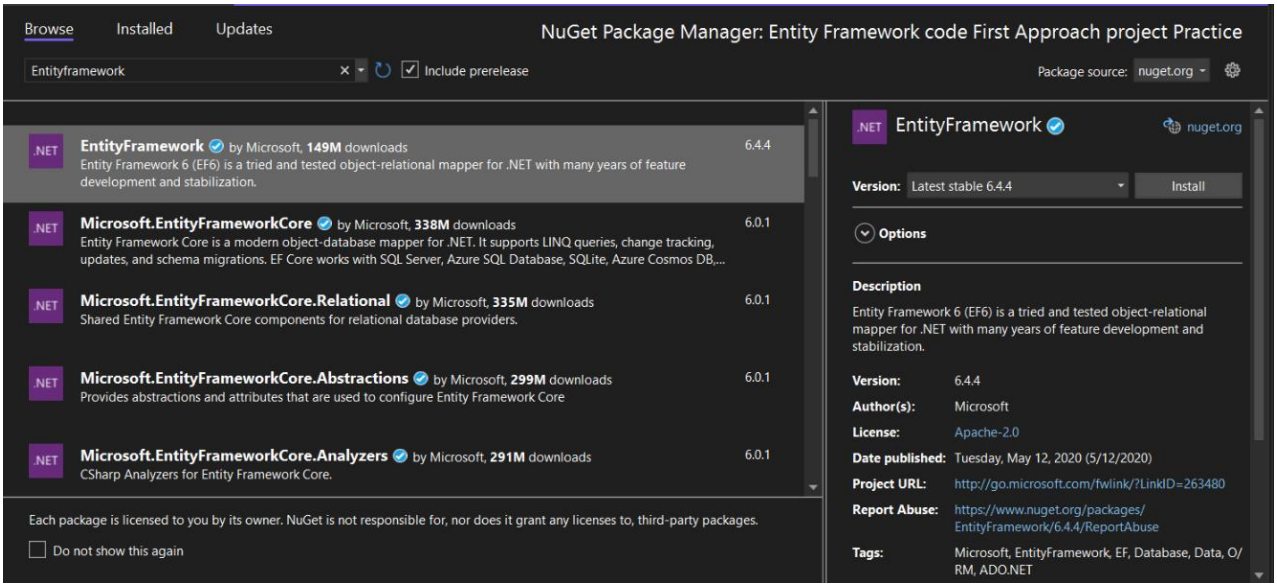
1




3




2




## Entity Framework code first Example:

### 1 – create the Application

### 2 - Create Models

```
public class Company
    {
        public int CompanyID { get; set; }
        public string CNAME { get; set; }
        public string ADDRESS { get; set; }
    }
```

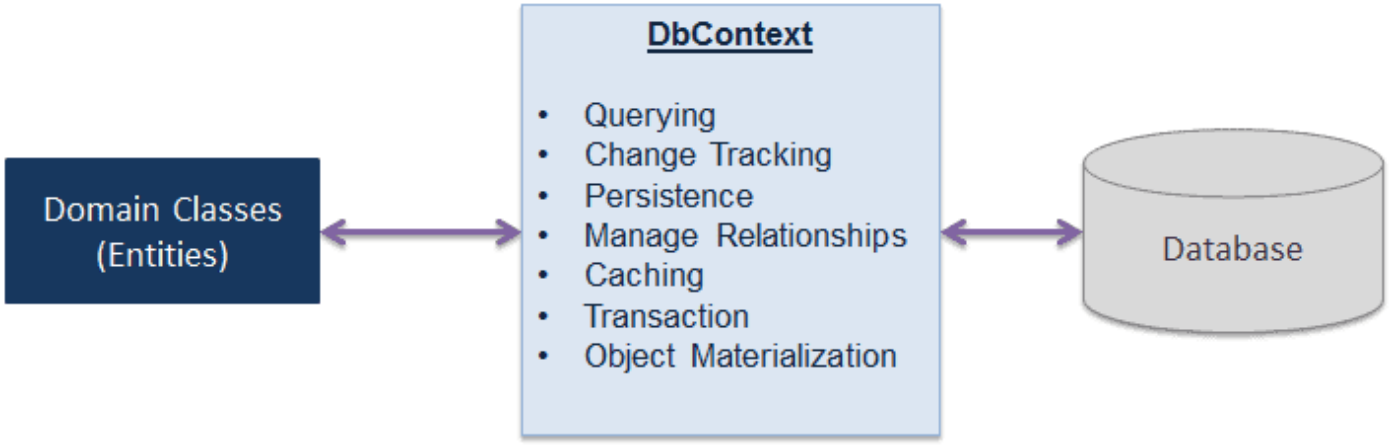
```
public class Employee
{
    public int Employeeid { get; set; }
    public string Name { get; set; }
    public string Phone { get; set; }
    public string Email { get; set; }
    public int CompanyID { get; set; }
}
```

## 3 -Create DbNameContext:DbContext (using System.Data.Entity)

Before we create DbContext les speak about what is dbcontext

# DbContext in Entity Framework 6

DbContext is an important class in Entity Framework API. It is a bridge between your domain(calsses) or entity classes and the database.



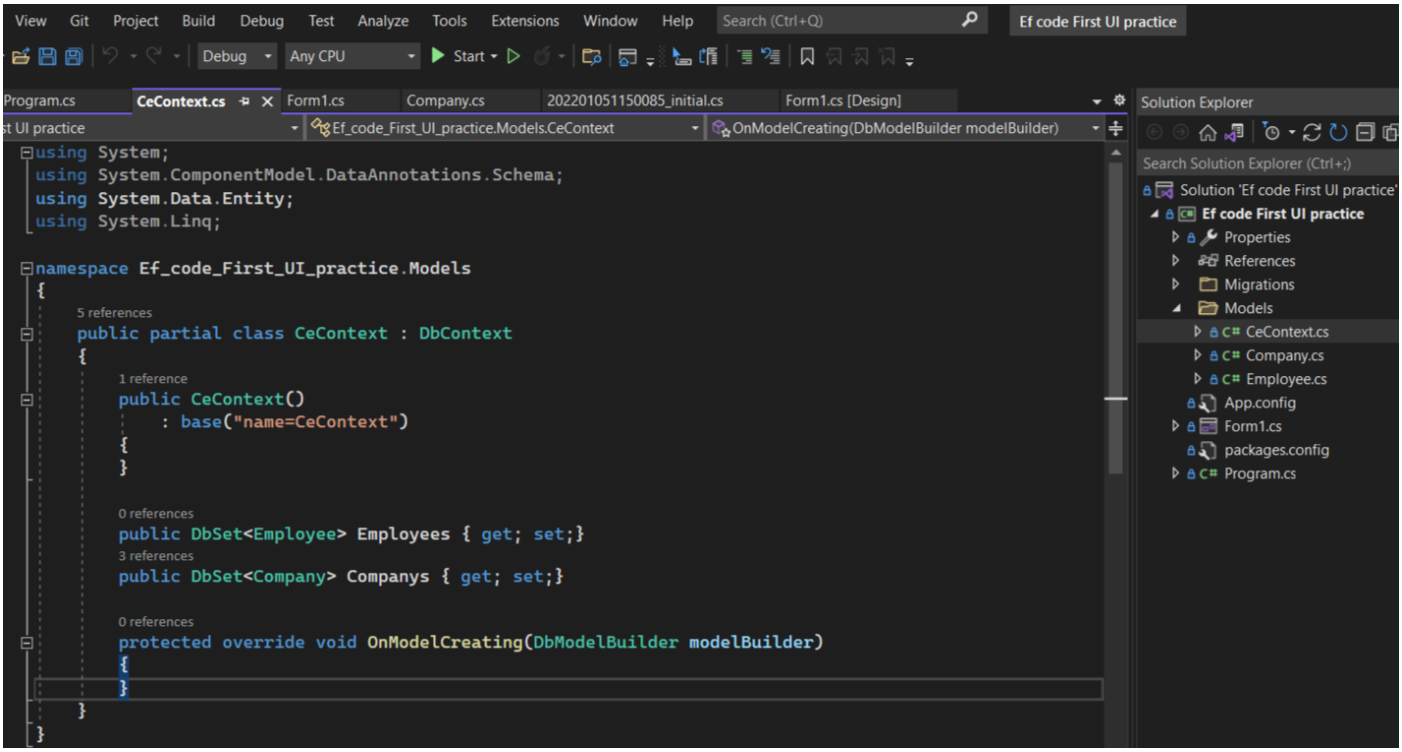

DbContext is the primary class that is responsible for interacting with the database. It is responsible for the following activities:

- 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.

dbContext Methods :

| Method | Usage |
|---|---|
| Entry | Gets an DbEntityEntry for the given entity. The entry provides access to change tracking information and operations for the entity. |
| SaveChanges | Executes INSERT, UPDATE and DELETE commands to the database for the entities with Added, Modified and Deleted state. |
| SaveChangesAsync | Asynchronous method of SaveChanges() |
| Set | Creates a DbSet<TEntity> that can be used to query and save instances of TEntity. |
| OnModelCreating | Override this method to further configure the model that was discovered by convention from the entity types exposed in DbSet<TEntity> properties on your derived context. |

Example :




## Connection string in Entity Framework 6 :

! Public CeContext:base(“name:CeContext”)

{

}

The connection string of my CeContext is name:CeContext you can find it in the App.config when you want to create database he used this connection string you can change it anytime you want feel free and if you don’t have one use there’s an option we will talk about later on .

## DbSet in Entity Framework 6 :

The DbSet class represents an entity set that can be used for create, read, update, and delete operations. As simple as that for example you have many class but you want to create database with just specific classes than you have to set only the classes you want in the dbcontext class.

## The following table lists important methods of the DbSet class:

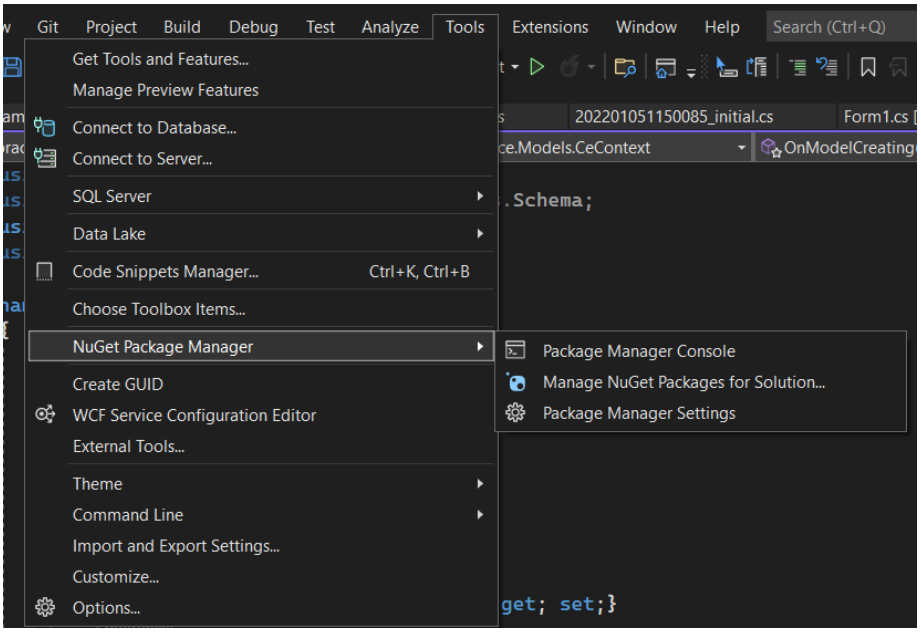
| Method Name | Return Type | Description |
|---|---|---|
| Add | Added entity type | Adds the given entity to the context with the Added state. When the changes are saved, the entities in the Added states are inserted into the database. After the changes are saved, the object state changes to Unchanged.<br><br>Example:<br>dbcontext.Students.Add(studentEntity) |
| AsNoTracking<Entity> | DBQuery<Entity> | Returns a new query where the entities returned will not be cached in the DbContext. (Inherited from DbQuery.)<br><br>Entities returned as AsNoTracking will not be tracked by DBContext. This will be a significant performance boost for read-only entities.<br><br>Example:<br>var studentList = dbcontext.Students.AsNoTracking<Student>().ToList<Student>(); |
| Attach(Entity) | Entity which was passed as parameter | Attaches the given entity to the context in the Unchanged state<br><br>Example:<br>dbcontext.Students.Attach(studentEntity); |
| 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.<br><br>Example:<br>var newStudentEntity = dbcontext.Students.Create(); |
| Find(int) | Entity type | Uses the primary key value 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.<br><br>Example:<br>Student studEntity = dbcontext.Students.Find(1); |
| Include | DBQuery | Returns the included non-generic LINQ to Entities query against a DbContext. (Inherited from DbQuery)<br><br>Example:<br>var studentList = dbcontext.Students.Include("StudentAddress").ToList<Student>();<br>var studentList = dbcontext.Students.Include(s => s.StudentAddress).ToList<Student>(); |
| 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.<br><br>Example:<br>dbcontext.Students.Remove(studentEntity); |
| 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.<br><br>Example:<br>var studentEntity = dbcontext.Students.SqlQuery("select * from student where studentid = 1").FirstOrDefault<Student>(); |

# Migration in EF 6 Code-First

In real world projects, data models change as features get implemented: new entities or properties are added and removed, and database schemas need to be changed accordingly to be kept in sync with the application. The migrations feature in Entity Framewrok provides a way to incrementally update the database schema to keep it in sync with the application's data model while preserving existing data in the database.

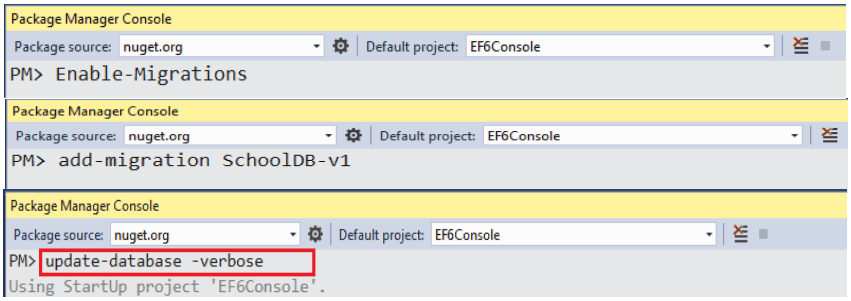
## Using Migaration in your projects :

To use migration in you project : Tools > NuGet Package Manager > Package Manager Console




## Migartion commands :

1. Enable-Migrations: Enables the migration in your project by creating a Configuration class.
2. Add-Migration: Creates a new migration class as per specified name with the Up() and Down() methods.
3. Update-Database: Executes the last migration file created by the Add-Migration command and applies changes to the database schema.



## Migartion Configuration:

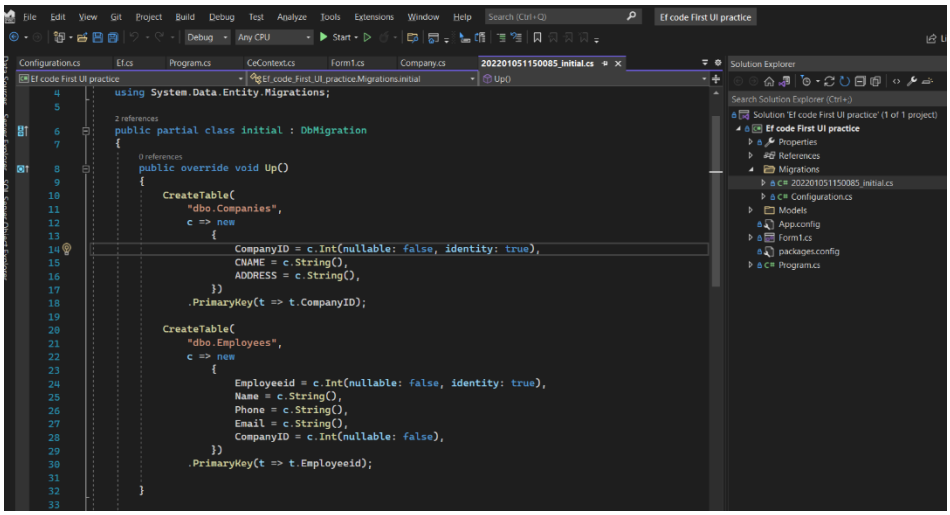
```
1 reference
internal sealed class Configuration : DbMigrationsConfiguration<Ef_code_First_UI_practice.Models.CeContext>
{
    0 references
    public Configuration()
    {
        AutomaticMigrationsEnabled = false;
    }

    0 references
    protected override void Seed(Ef_code_First_UI_practice.Models.CeContext context)
    {
        //  This method will be called after migrating to the latest version.

        //  You can use the DbSet<T>.AddOrUpdate() helper extension method
        //  to avoid creating duplicate seed data.
    }
}
```

## Migartion code :

Generated code based on Class CeContext:DbContext that have connection string on it




## Note :

Using Migartion make your work very smooth changing database by just update class

Add-Migartion NameOfMigartion → Update-database

# Data Annotations Attributes in EF 6

Data Annotations attributes are .NET attributes which can be applied on an entity class or properties to override default conventions in EF 6

This name space make you specified your attributes like who you want him to be the primary key length max and min foreign keys and tables  like your working directly with Sql Queries .

## System.ComponentModel.DataAnnotations Attributes :

| Attribute | Description |
|---|---|
| Key | Can be applied to a property to specify a key property in an entity and make the corresponding column a PrimaryKey column in the database. |
| Timestamp | Can be applied to a property to specify the data type of a corresponding column in the database as `rowversion`. |
| ConcurrencyCheck | Can be applied to a property to specify that the corresponding column should be included in the optimistic concurrency check. |
| Required | Can be applied to a property to specify that the corresponding column is a NotNull column in the database. |
| MinLength | Can be applied to a property to specify the minimum string length allowed in the corresponding column in the database. |
| MaxLength | Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database. |
| StringLength | Can be applied to a property to specify the maximum string length allowed in the corresponding column in the database. |

## System.ComponentModel.DataAnnotations.Schema Attributes

| Attribute | Description |
|---|---|
| Table | Can be applied to an entity class to configure the corresponding table name and schema in the database. |
| Column | Can be applied to a property to configure the corresponding column name, order and data type in the database. |
| Index | Can be applied to a property to configure that the corresponding column should have an Index in the database. (EF 6.1 onwards only) |
| ForeignKey | Can be applied to a property to mark it as a foreign key property. |
| NotMapped | Can be applied to a property or entity class which should be excluded from the model and should not generate a corresponding column or table in the database. |
| DatabaseGenerated | Can be applied to a property to configure how the underlying database should generate the value for the corresponding column e.g. identity, computed or none. |
| InverseProperty | Can be applied to a property to specify the inverse of a navigation property that represents the other end of the same relationship. |
| ComplexType | Marks the class as complex type in EF 6. EF Core 2.0 does not support this attribute. |

Example :

```
[Table("Department")]
class Department
{
    [Key]
    public int DeptID { get; set; }
    public string DeptName { get; set; }
    public string Location { get; set; }
    public List<Employee> Employee { get; set; }

}
```

```
[Table("tblEmployee")]
class Employee
{

    [Key]
    public int EmpID { get; set; }

    [Column("EmployeeName")]
    public string EmpName { get; set; }
    public string Job { get; set; }
    public DateTime HireDate { get; set; }
    public Decimal Salary { get; set; }
    public int DeptNO { get; set; }
    [ForeignKey("DeptNO")]
    public Department Department { get; set; }
}
```