Entity Framework

Entity framework is essentially a persistent framework that freeze you from writing store procedures in most cases, Manage database connections, manually map the database tables, records or your domain objects. Entity framework do this all for you.

**S-2 Building Model using Database first workflow**

**Work Flows:-**

1. **Database first**

We design our tables using visual designers and then have entity framework generates domain classes base on the database.

1. **Code first**

we create our domain classes and then entity framework generates database tables

1. **Model first**

We create UML diagram and then entity framework generates domain classes and database

**Database first:-**

1. create database
2. Create console app
3. Add ado.net entity data model by right clicking on project name during adding select EF designer from database then specify new connection.

**Add new Table:-**

1. First add table in database
2. Then right click on .edmx diagram then select Update model from database then select new table from add tab then finish

**Update Table:-**

1. Make changes in database tables remove, editt or delete columns
2. Right click on empty area of .edmx diagram then select Update model from database then simple click on finish button
3. Then check errors by right clicking on empty area then click on validate
4. If have errors because of deleting columns or renamed then manually delete old columns from diagram and if data type changed then change also from properties of column in .edmx diagram.

**Delete Table:-**

1. delete table from database
2. Right click on .edmx diagram empty area then click on update model from database then click on delete tab then select deleted table then click on finish.
3. Then delete that table form diagram mean conceptual model

**Add Store Procedure:-**

1. adding store procedure is same as new table jus tick the stored procedure ad functions in update wizard then finish.

**Accessing the Tables and store procedures from database through plutoDbContext that is drived from DbContext**

class Program

{

static void Main(string[] args)

{

var dbContext = new PlutoDbContext();

var cour = dbContext.Authors;

foreach (var c in cour)

{

Console.WriteLine(c.Name);

}

}

}

**Code first With New Database:-**

1. type command in the nugget package manager console(install-package EntityFramework -Version:6.1.3)
2. Create model like this:-

public class Course

{

public int id { get; set; }

public string Title { get; set; }

public string Description { get; set; }

public CourseLevel Level { get; set; }

public float FullPrice { get; set; }

public Author Author { get; set; }

public IList<Tag> Tags { get; set; }

}

public class Author

{

public int Id { get; set; }

public string Name { get; set; }

public IList<Course> Courses { get; set; }

}

public class Tag

{

public int Id { get; set; }

public string Name { get; set; }

public IList<Course> Courses { get; set; }

}

public enum CourseLevel

{

Beginner = 1,

Intermediate = 2,

Advance = 3

}

1. create **DbContext:-** DbContext is simple abstraction over the database that hides all the complexity around working with connections, commands, reading data, transactions and so on.**DbSet** is a collection of objects that represent a table in the database.

public class PlutoContext : DbContext

{

public DbSet<Course> Courses { get; set; }

public DbSet<Author> Authors { get; set; }

public DbSet<Tag> Tags { get; set; }

}

1. Connections String: go to the app.config file this name is different in different applications like in web this is web.config.

<connectionStrings>

<add name="DefaultConnection" connectionString="data source=ZAHIDALI\SQLEXPRESS; initial catalog=PlutoCodeFirst; integrated security=SSPI" providerName="System.Data.SqlClient"/>

</connectionStrings>

</configuration>

Here initial catalog mean database name.

If the connection string name is different from over dbContext like here our DbContext name is PlutoContext and connection string name is DefaultConnection so we need to tell entity framework the name of our connection string for this add this constructor to your dbContext mean PlutoContext.

public PlutoContext()

:base("name=DefaultConnection")

{

}

1. create migration by going to nuget package manager console
2. run command enable-migrations
3. Run command add-migration nameOfMigration

6-apply migration on database run command update-database

**Code first With Existing Database:-**

1. create database
2. Create console app
3. Add ado.net entity data model by right clicking on project name during adding select Code First from database then specify new connection.

4-enable-migrations

5-add-migration migrationName

6-add-migration migrationName -IgnoreChanges -Force //ingnoreChanges is to tell entity framework that whatever you have in model now already exist in database

7-upate-database

8-if need to add new class for table then also add it to your DbContext

9-Use Sql(“”) to write sql commands in migration like this:-

10-update-database -TargetMigration: migrationName //to revert back to the specific migration

namespace CodeFirstExistingDatabase.Migrations

{

using System;

using System.Data.Entity.Migrations;

public partial class AddCatogriesTable : DbMigration

{

public override void Up()

{

CreateTable(

"dbo.Categories",

c => new

{

Id = c.Int(nullable: false, identity: false), //here we changed the identity to false to insert id manually

Name = c.String(),

})

.PrimaryKey(t => t.Id);

Sql("INSERT INTO Categories VALUES (1, 'Web Development')");

Sql("INSERT INTO Categories VALUES (2, 'Programming languages')");

}

public override void Down()

{

DropTable("dbo.Categories");

}

}

}

**Migrations:-**

* enable-migrations
* add-migration migrationName
* Revert back to prevous migration mean if you are in version 3 and want to go back to version 2 run this command (update-database -TargetMigration: migrationName)

**Get Script Of SQL From Visual Studio**

Run this command on nugget package manager console:

**Update-Database -Script -SourceMigration:0**

This command generates the SQL script from the very fifirst migration to the last

one. In a real-world scenario, you may want to change the range of migrations

included in the SQL script in each deployment. To do this, you can use:

**Update-Database -Script -SourceMigration:Migr1 -TargetMigration:Migr2**

**When Query executed:-**

Query executed when we

* Iterating over query variable
* Calling ToList, ToArray, ToDictionary
* Calling First, Last, Single, Count, Max, Min, Average

**Disabling Lazy Loading:-**

using System.Data.Entity;

using Queries.EntityConfigurations;

namespace Queries

{

public class PlutoContext : DbContext

{

public PlutoContext()

: base("name=PlutoContext")

{

this.Configuration.LazyLoadingEnabled = false; //disabled lazy loading

}

public virtual DbSet<Author> Authors { get; set; }

public virtual DbSet<Course> Courses { get; set; }

public virtual DbSet<Tag> Tags { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Configurations.Add(new CourseConfiguration());

}

}

}

**Enabling Lazy Loading:-**

Enable lazy loading by declaring properties with virtual key word

using System.Collections.Generic;

namespace Queries

{

public class Course

{

public Course()

{

Tags = new HashSet<Tag>();

}

public int Id { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public int Level { get; set; }

public float FullPrice { get; set; }

public virtual Author Author { get; set; } //for lazy loading

public int AuthorId { get; set; }

public virtual ICollection<Tag> Tags { get; set; } //for lazy loading

public Cover Cover { get; set; }

}

}

**Enabling Eager Loading:-**

using System.Linq;

using System;

using System.Data.Entity; //using this

namespace Queries

{

class Program

{

static void Main(string[] args)

{

var context = new PlutoContext();

var courses = context.Courses.Include(c => c.Author).ToList(); //using Include method with lambda expression and toList() to execute query when storing in courses

foreach (var item in courses)

{

Console.WriteLine(item.Name + " "+ item.Author.Name);

}

}

}

}

**Explicit Loading:-**

When queries are to big and become bulky we will use explicit loading to break into multiple queries

1. For single author

using System.Linq;

using System;

using System.Data.Entity;

namespace Queries

{

class Program

{

static void Main(string[] args)

{

var context = new PlutoContext();

var author = context.Authors.Single(a => a.Id == 1); //1st query single is use yo query at the time

//MSDN way works only for one author

context.Entry(author).Collection(a => a.Courses).Load(); //2nd query

//MOSH WAY

context.Courses.Where(c => c.AuthorId == author.Id && c.FullPrice ==0 ).Load(); //2nd query with filter

foreach (var item in author.Courses)

{

Console.WriteLine(item.Name + " "+ item.Author.Name);

}

}

}

}

1. For multiple authors

using System.Linq;

using System;

using System.Data.Entity;

namespace Queries

{

class Program

{

static void Main(string[] args)

{

var context = new PlutoContext();

var author = context.Authors.ToList(); //toList use to query at this time

var authorIds = author.Select(a => a.Id);

context.Courses.Where(c => authorIds.Contains(c.AuthorId) && c.FullPrice == 0).Load();

}

}

}