Chapter 8
Entity Framework Core

Understanding Legacy Entity Framework

- **Entity Framework (EF) was first released as part of .NET Framework 3.5 with Service Pack 1 back in late 2008**

- **Entity Framework has evolved, as Microsoft has observed how programmers use an object-relational mapping (ORM) tool in the real world**

- **ORMs use a mapping definition to associate columns in tables to properties in classes and a programmer can interact with objects of different types in a way that they are familiar with, instead of having to deal with knowing how to store the values in a relational table or another structure provided by a NoSQL data store**

Understanding Legacy Entity Framework

◆ **The version of EF included with .NET Framework is Entity Framework 6 (EF6). It is mature, stable, and supports an old EDMX (XML file) way of defining the model as well as complex inheritance models, and a few other advanced features**

◆ **EF 6.3 and later have been extracted from .NET Framework as a separate package so it can be supported on .NET Core 3.0 and later, including .NET 5. This enables existing projects like web applications and services to be ported and run cross-platform**

◆ **EF6 should be considered a legacy technology because it has some limitations when running cross-platform and no new features will be added to it**

Understanding Entity Framework Core

- **Entity Framework Core (EF Core) is a lightweight, extensible, open source and cross-platform version of the popular Entity Framework data access technology**

- **EF Core allows us to interact with data from relational databases using an object model that maps directly to the business objects (or domain objects) in our application**

- EF Core can serve as an object-relational mapper (O/RM), which:

  - **Enables .NET developers to work with a database using .NET objects**

  - **Eliminates the need for most of the data-access code that typically needs to be written**

Understanding Entity Framework Core

◆ EF Core 5.0 runs on platforms that support .NET Standard 2.1, meaning .NET Core 3.0 and 3.1, as well as .NET 5. It will not run on.NET Standard 2.0 platforms like .NET Framework 4.8

◆ Entity Framework Core supports many database providers to access different databases and perform database operations:

- SQL Server (www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer)
- MySQL (**www.nuget.org/packages/MySQL.Data.EntityFrameworkCore**)
- PostgreSQL (**www.nuget.org/packages/Npgsql.EntityFrameworkCore.PostgreSQL**)
- SQLite (**www.nuget.org/packages/Microsoft.EntityFrameworkCore.Sqlite**)
- Oracle (**www.nuget.org/packages/Oracle.ManagedDataAccess.Core**)
- In-memory (www.nuget.org/packages/Microsoft.EntityFrameworkCore.InMemory)
- *More database provider: https://docs.microsoft.com/en-us/ef/core/providers/*

Understanding Entity Framework Core

◆ To manage data in a specific database, we need classes that know how to efficiently talk to that database

◆ EF Core database providers are sets of classes that are optimized for a specific data store. There is even a provider for storing the data in the memory of the current process, which is useful for high performance unit testing since it avoids hiing an external system

Understanding Entity Framework Core

◆ The distributed NuGet packages as shown in the following table:

Database	NuGet Package Name
Microsoft SQL Server 2012 or later	Microsoft.EntityFrameworkCore.SqlServer
SQLite 3.7 or later	Microsoft.EntityFrameworkCore.SQLite
MySQL	MySQL.Data.EntityFrameworkCore
In-memory	Microsoft.EntityFrameworkCore.InMemory
Azure Cosmos DB SQL API	Microsoft.EntityFrameworkCore.Cosmos
Oracle DB 11.2	Oracle.EntityFrameworkCore

What Is an Object-Relational (OR) Mapper?

- In the database world, relational databases are prevalent and the programming world is all about objects

- Working with objects as instances of classes in memory is at the core of object oriented programming (OOP)

- Most applications also include the requirement to permanently store data in objects, especially in databases. Basically, there are object oriented databases (OODBs) that are directly able to store objects, but OODBs have only a small distribution so far. Relational databases are more predominant, but they map the data structures differently than object models

What is an Object-Relational (OR) Mapper?

◆ To make the handling of relational databases more natural in object-oriented systems, the software industry has been relying on object-relational mappers

◆ The tools translate concepts from the object-oriented world, such as classes, attributes, or relationships between classes, to corresponding constructs of the relational world, such as tables, columns, and foreign keys

◆ Developers can thus remain in the object-oriented world and instruct the OR mapper to load or store certain objects that are in the form of records in tables of the relational database
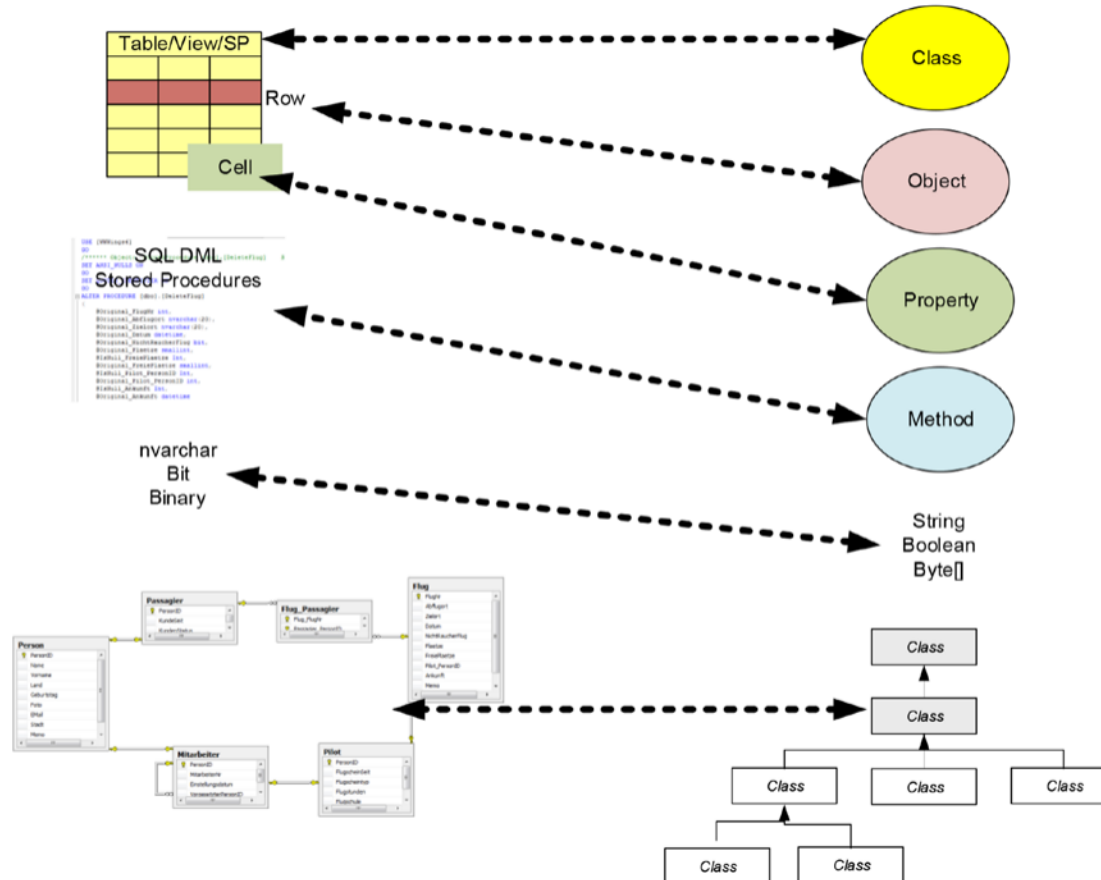
The OR mapper translates constructs of the OOP world to the relational world

New Features in Entity Framework Core

- Entity Framework Core runs not only on Windows, Linux, and macOS but also on mobile devices running Windows 10, iOS, and Android. On mobile devices, of course only access to local databases (such as SQLite) is provided

- Entity Framework Core provides faster execution speeds, especially when reading data (almost the same performance as manually copying data from a DataReader object to a typed .NET object)

- Batching allows the Entity Framework Core to merge INSERT, DELETE, and UPDATE operations into one database management system round-trip rather than sending each command one at a time

New Features in Entity Framework Core

◆ Projections with Select() can now be mapped directly to entity classes. The detour via anonymous .NET objects is no longer necessary

◆ Default values for columns in the database are now supported in both reverse engineering and forward engineering

◆ In addition to the classic auto-increment values, newer methods such as sequences are now also allowed for key generation

◆ The term shadow properties in Entity Framework Core refers to the now possible access to columns of the database table for which there are no attributes in the class

Process Models for Entity Framework Core

◆ **Entity Framework Core supports the following:**

- Reverse engineering of existing databases (an object model is created from an existing database schema)

- Forward engineering of databases (a database schema is generated from an object model)

◆ **Reverse engineering (often referred to as database first) is useful if we already have a database or if developers choose to create a database in a traditional way**

◆ **The second option, called forward engineering, gives the developer the ability to design an object model. From this, the developer can then generate a database schema**

Process Models for Entity Framework Core

◆ For the developer, forward engineering is usually better because we can design an object model that we need for programming

◆ Forward engineering can be used at development time (via so-called schema migrations) or at runtime

◆ A schema migration is the creation of the database with an initial schema or a later extension/modification of the schema
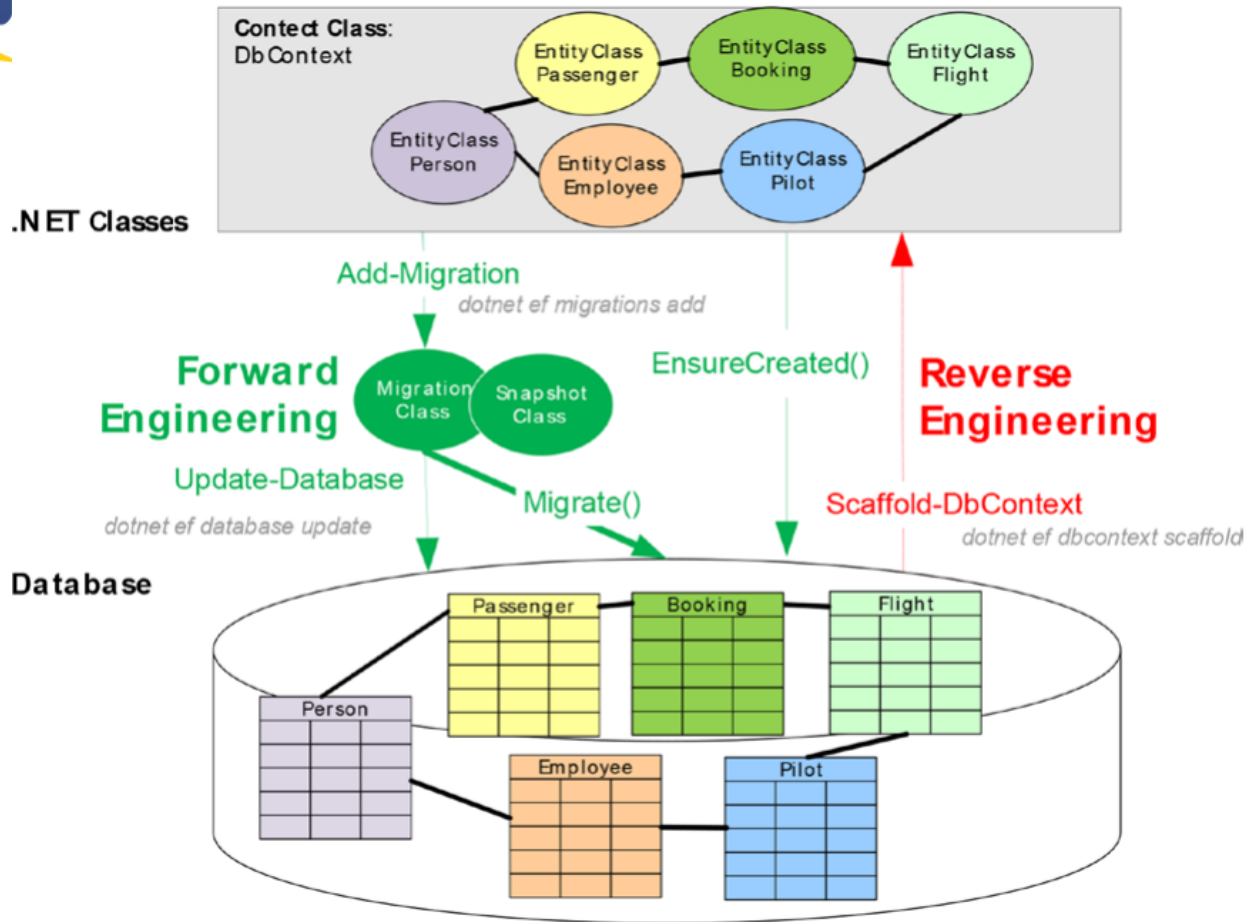
***Forward engineering versus reverse engineering for Entity Framework Core***

15

Components of Entity Framework Core

- Entity classes (domain object classes, business object classes, data classes, or persistent classes) are representations of tables and views. They contain properties or fields that are mapped to columns of the tables/views

- Entity classes can be plain old CLR objects (POCO classes); in other words, they need no base class and no interface

- A context class is a class always derived from the DbContext base class. It has properties of type DbSet for each of the entity classes

- The context class or DbSet properties take the commands of the self-created program code in the form of LINQ commands, SQL commands, stored procedure and table-valued function (TVF) calls, or special API calls for append, modify, and delete

Components of Entity Framework Core

- The context class sends the commands to the DBMS-specific provider, which sends the commands to the database via DbCommand objects and receives result sets in a DataReader from the database

- The context class transforms the contents of the DataReader object into instances of the entity class. This process is called *materialization*
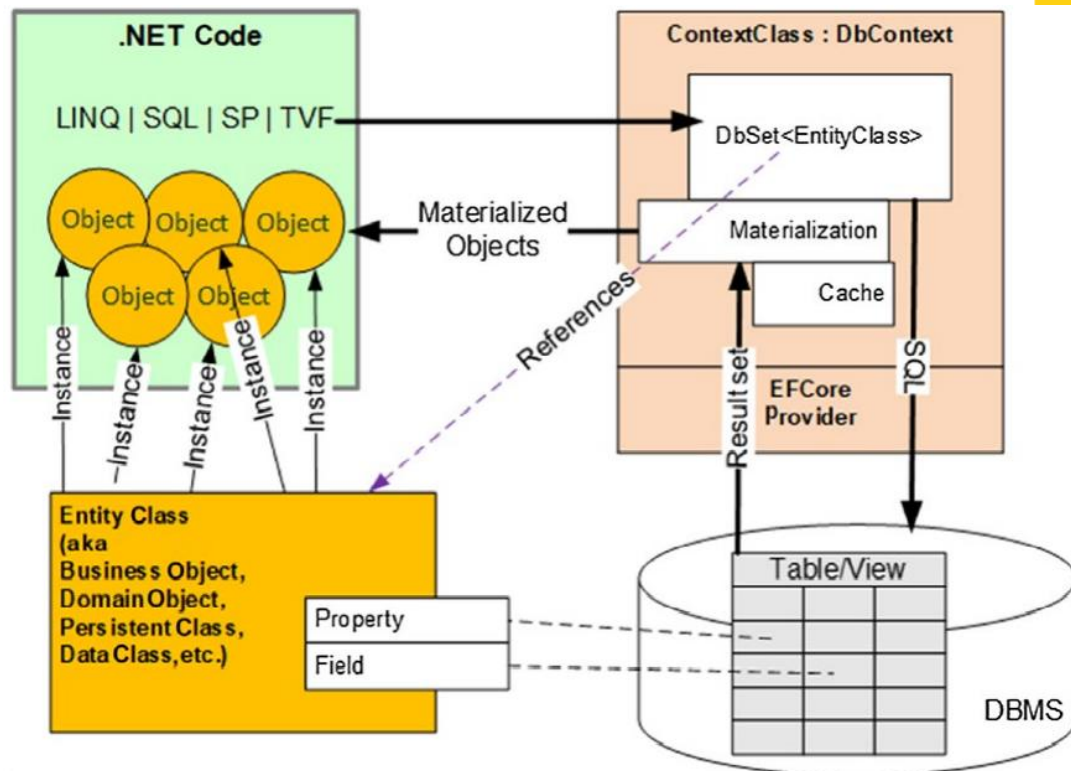
The central artifacts in Entity Framework Core and their context

DbContext Class

- The DbContext doesn't get used directly, but through classes that inherit from the DbContext class

- The entities that are mapped to the database are added as DbSet<T> properties on the derived class

- The OnModelCreating method is used to further define the mappings between the entities and the database

- The following table shows some of the more commonly used members of the DbContext:

DbContext Class

Member of DbContext	Description
Database	Provides access to database-related information and functionality, including execution of SQL statements
Model	The metadata about the shape of entities, the relationships between them, and how they map to the database. Note: This property is usually not interacted with directly
ChangeTracker	Provides access to information and operations for entity instances this context is tracking
DbSet<T>	Used to query and save instances of application entities. LINQ queries against DbSet properties are translated into SQL queries
EntryEntry<TEntity>	Provides access to change tracking information and operations (such as changing the EntityState) for the entity. Can also be called on an untracked entity to change the state to tracked
SaveChangesSaveChangesAsync	Saves all entity changes to the database and returns the number of records affected
OnConfiguring	A builder used to create or modify options for the context. Executes each time a DbContext instance is created. Note: It is recommended not to use this, and instead use the DbContextOptions to configure the context at runtime, and use an instance of the IDesignTimeDbContextFactory at design time
OnModelCreating	Called when a model has been initialized, but before it's finalized. Methods from the Fluent API are placed in this method to finalize the shape of the model

DbSet Class

- For each entity in our object model, we add a property of type DbSet<T>. The DbSet<T> is a specialized collection property used to interact with the database provider to get, add, update, or delete records in the database

- Each DbSet<T> provides a number of core services to each collection, such as creating, deleting, and finding records in the represented table

- The following table describes some of the core members of the DbSet<T> class:

◆ **The following table describes some of the core members of the DbSet<T> class:**

Member of DbSet<T>	Description
Add/AddRange	Begins tracking the entity/entities in the Added state. Item(s) will be added when SaveChanges is called. Async versions are available as well
Find	Searches for the entity in the ChangeTracker by primary key. If not found, the data store is queried for the object. An async version is available as well
Update/UpdateRange	Begins tracking the entity/entities in the Modified state. Item(s) will be updated when SaveChanges is called. Async versions are available as well
Remove/RemoveRange	Begins tracking the entity/entities in the Deleted state. Item(s) will be removed when SaveChanges is called. Async versions are available as well
Attach/AttachRange	Begins tracking the entity/entities in the Unchanged state. No operation will execute when SaveChanges is called. Async versions are available as well

Entities

◆ **Entities are a conceptual model of a physical database that maps to our business domain. This model is termed an entity data model (EDM). The EDM is a client-side set of classes that are mapped to a physical database by Entity Framework Core convention and configuration**

```csharp
[Table("Product", Schema = "dbo")]
public class Product{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }
    [Required]
    [StringLength(40)]
    public string ProductName { get; set; }
    [Required]
    public decimal UnitPrice { get; set; }
    [Required]
    public int UnitsInStock { get; set; }
}
```

http://vku.udn.vn/

Defining Entity Framework Core Models

◆ **An entity class represents the structure of a table and an instance of the class represents a row in that table**

◆ **EF Core uses a combination of Conventions, Annotation Atributes, and Fluent API statements to build an entity model at runtime so that any actions performed on the classes can later be automatically translated into actions performed on the actual database**

Defining Entity Framework Core Models

◆ **EF Core Conventions:** The code we will write will use the following conventions:

- The name of a table is assumed to match the name of a DbSet<T> property in the DbContext class, for example, Products

- The names of the columns are assumed to match the names of properties in the class, for example, ProductID

- The string .NET type is assumed to be a *nvarchar* type in the database

- The int .NET type is assumed to be an *int* type in the database

- A property that is named ID , or if the class is named Product, then the property can be named ProductID

10/08/2023

http://vku.udn.vn/

Defining Entity Framework Core Models

◆ **EF Core Annotation attributes:** Conventions often aren't enough to completely map the classes to the database objects. Another way of adding more smarts to our model is to apply annotation attributes

- For example, in the database, the maximum length of a ProductName is 40 , and the value cannot be null. In a Product class, we could apply aributes to specify this, as shown in the following code:

```
CREATE TABLE Products (
    ProductID        INTEGER        PRIMARY KEY,
    ProductName      NVARCHAR (40) NOT NULL
)
```

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

Data Annotations Supported by the Entity Framework

Data Annotation	Description
Table	Defines the schema and table name for the entity
Column	Defines the column name for the model property
Key	Defines the primary key for the model. Key fields are implicitly also [Required]
Required	Declares the property as not nullable in the database
ForeignKey	Declares a property that is used as the foreign key for a navigation property
InverseProperty	Declares the navigation property on the other end of a relationship
StringLength	Specifies the max length for a string property
TimeStamp	Declares a type as a rowversion in SQL Server and adds concurrency checks to database operations involving the entity
ConcurrencyCheck	Flags field to be used in concurrency checking when executing updates and deletes
DatabaseGenerated	Specifies if the field is database generated or not. Takes a DatabaseGeneratedOption value of Computed, Identity, or None
DataType	Provides for a more specific definition of a field than the intrinsic datatype
NotMapped	Excludes the property or class in regard to database fields and tables

Defining Entity Framework Core Models

◆ **EF Core Fluent API: The Fluent API configures the application entities through C# code.** The methods are exposed by the ModelBuilder instance available in the DbContext, OnModelCreating method

- The Fluent API is the most powerful of the configuration methods and overrides any data annotations or conventions that are in conflict. Some of the configuration options are only available using the Fluent API, such as complex keys and indices

- For example, the maximum length of a ProductName is 40 , and the value cannot be null, we could apply Fluent API Fluent API statement in the OnModelCreating method of a database context class, as shown in the following code:

```
[Required]
[StringLength(40)]
public string ProductName { get; set; }
```

```
modelBuilder.Entity<Product>()
    .Property(product => product.ProductName)
    .IsRequired()
    .HasMaxLength(40);
```

The EF Core Global Tool CLI Commands

◆ The dotnet-ef global CLI tool EF Core tooling contains the commands needed to scaffold existing databases into code, to create/remove migrations (changes in the data structure based on the entities), and to operate on a database (update, drop, etc.)

   ◆ Open Command Prompt (or Terminal) then run as the following command:

   ▪ Check if we have already installed dotnet-ef as a global tool

```
Command Prompt
C:\>dotnet tool list --global
Package Id              Version           Commands
------------------------------------------------------
dotnet-ef              5.0.0             dotnet-ef
```

The EF Core Global Tool CLI Commands

- If an old version is already installed, then uninstall the tool, as shown in the following command:

```
Command Prompt

C:\>dotnet tool uninstall --global dotnet-ef
Tool 'dotnet-ef' (version '5.0.0') was successfully uninstalled.
```

- Install the latest version, as shown in the following command:

```
Command Prompt

C:\>dotnet tool install --global dotnet-ef --version 5.0.1
You can invoke the tool using the following command: dotnet-ef
Tool 'dotnet-ef' (version '5.0.1') was successfully installed.

C:\>dotnet tool list --global
Package Id            Version         Commands
------------------------------------------------
dotnet-ef            5.0.1           dotnet-ef
```

The EF Core Global Tool CLI Commands

◆ The three main commands in the EF Core global tool are shown in the following table:

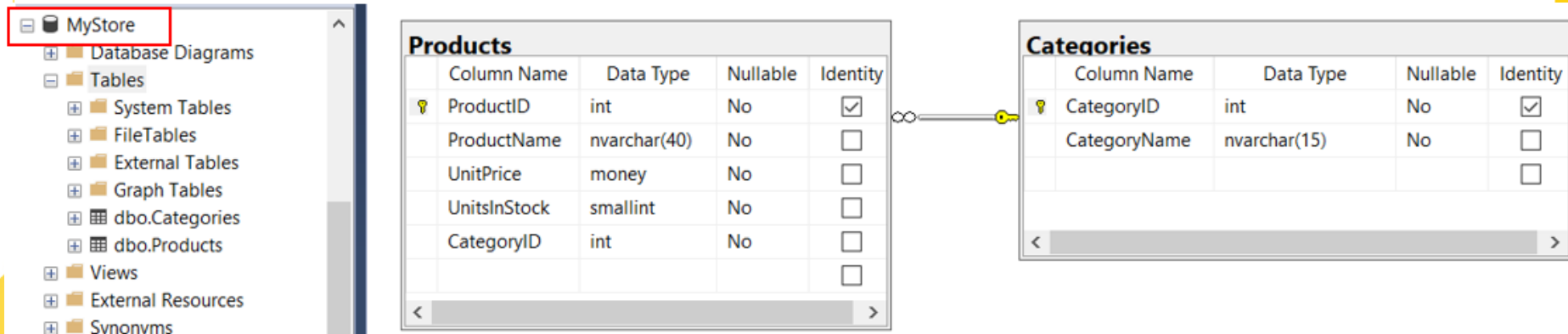| Command	Description
Database	Commands to manage the database. Sub-commands include drop and update
DbContext	Commands to manage the DbContext types. Sub-commands include scaffold, list, and info
Migrations	Commands to manage migrations. Sub-commands include add, list, remove, and script.

◆ Each main command has additional sub-commands. As with the all of the .NET Core commands, each command has a rich help system that can be accessed by entering -h along with the command

Reverse Engineering of Existing Databases Demonstration

## ◆ Create a sample database named MyStore for demonstrations

MyStore object explorer:
- □ MyStore
  - ⊞ Database Diagrams
  - □ Tables
    - ⊞ System Tables
    - ⊞ FileTables
    - ⊞ External Tables
    - ⊞ Graph Tables
    - ⊞ dbo.Categories
    - ⊞ dbo.Products
  - ⊞ Views
  - ⊞ External Resources
  - ⊞ Synonyms

Products

	Column Name	Data Type	Nullable	Identity
🔑	ProductID	int	No	☑
	ProductName	nvarchar(40)	No	☐
	UnitPrice	money	No	☐
	UnitsInStock	smallint	No	☐
	CategoryID	int	No	☐
				☐

Categories

	Column Name	Data Type	Nullable	Identity
🔑	CategoryID	int	No	☑
	CategoryName	nvarchar(15)	No	☐
				☐

T470S.MyStore - dbo.Products

ProductID	ProductName	UnitPrice	UnitsInStock	CategoryID
1	Genen Shouyu	50.0000	39	1
2	Alice Mutton	30.0000	17	1
3	Aniseed Syrup	40.0000	13	3
4	Perth Pasties	22.0000	53	2
5	Carnarvon Tigers	21.3500	0	4
6	Gula Malacca	25.0000	120	2
7	Steeleye Stout	30.0000	15	7
8	Chocolade	40.0000	6	5
9	Mishi Kobe Niku	97.0000	29	6
10	Ikura	31.0000	31	8

.MyStore - dbo.Categories

CategoryID	CategoryName
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood
NULL	NULL

1.Create a Console App named DemoDatabaseFirst then right-click on project , select Open In Terminal to install **packages**
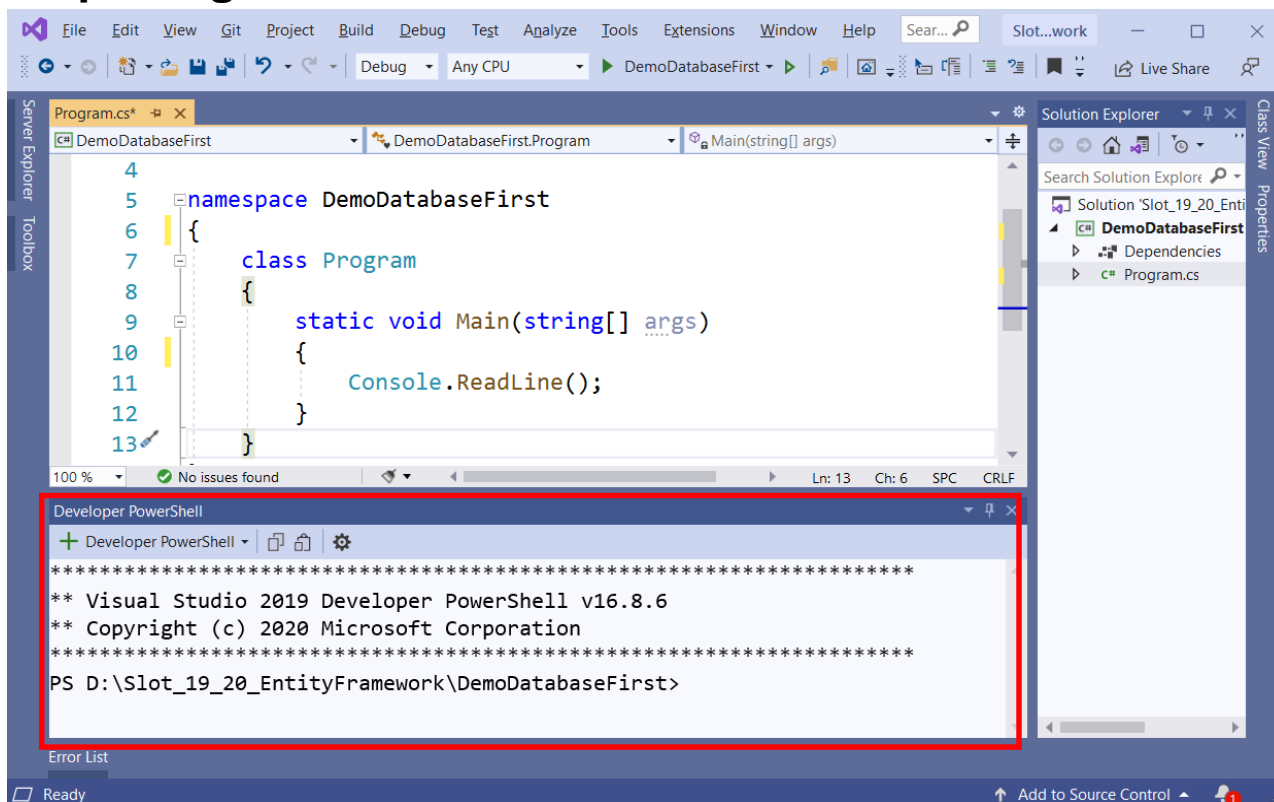
2.On Developer PowerShell dialog , execute the following commands to install packages:

- **dotnet add package Microsoft.EntityFrameworkCore.design**

```
***********************************************************************
PS D:\Slot_19_20_EntityFramework\DemoDatabaseFirst> dotnet add package Microsoft.EntityFrameworkCore.design
```

- **dotnet add package Microsoft.EntityFrameworkCore.SqlServer**

```
............................................................
PS D:\Slot_19_20_EntityFramework\DemoDatabaseFirst> dotnet add package Microsoft.EntityFrameworkCore.SqlServer
```

3.On Developer PowerShell dialog , execute the following commands to generate model:

dotnet ef dbcontext scaffold "server =(local); database = MyStore;uid=sa;pwd=123;" Microsoft.EntityFrameworkCore.SqlServer --output-dir Models

10/08/2023

34
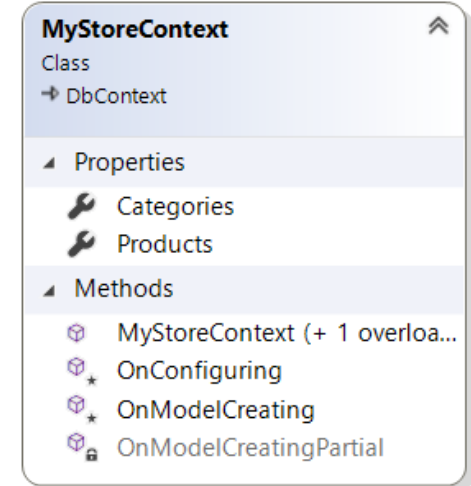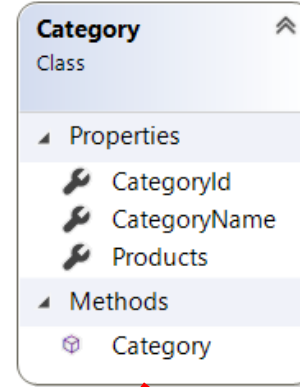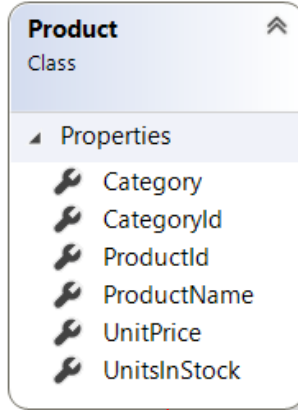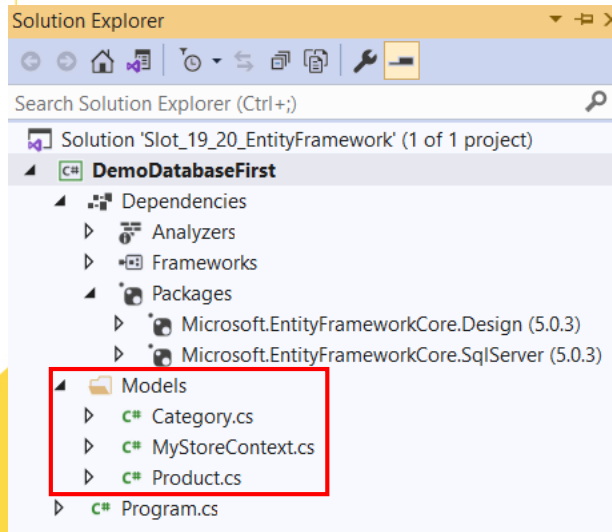
```
namespace DemoDatabaseFirst.Models{
    public partial class Product{
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public short UnitsInStock { get; set; }
        public int CategoryId { get; set; }
        public virtual Category Category { get; set; }
    }
}
```

```
namespace DemoDatabaseFirst.Models{
    public partial class Category{
        public Category() {
            Products = new HashSet<Product>();
        }
        public int CategoryId { get; set; }
        public string CategoryName { get; set; }
        public virtual ICollection<Product> Products { get; set; }
    }
}
```

10/08/2023

MyStoreConext Class

```csharp
public partial class MyStoreContext : DbContext {
    public MyStoreContext(){ }
    public MyStoreContext(DbContextOptions<MyStoreContext> options) : base(options){ }
    public virtual DbSet<Category> Categories { get; set; }
    public virtual DbSet<Product> Products { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder) {
        if (!optionsBuilder.IsConfigured) {
            optionsBuilder.UseSqlServer("server =(local); database = MyStore;uid=sa;pwd=123;");
        }
    }
    protected override void OnModelCreating(ModelBuilder modelBuilder){
        modelBuilder.HasAnnotation("Relational:Collation", "SQL_Latin1_General_CP1_CI_AS");
        modelBuilder.Entity<Category>(entity =>{
            entity.Property(e => e.CategoryId).HasColumnName("CategoryID");
            entity.Property(e => e.CategoryName)
                .IsRequired()
                .HasMaxLength(15)
                .UseCollation("Vietnamese_CI_AS");
        });
```

MyStoreContext
Class
→ DbContext

▲ Properties
  🔧 Categories
  🔧 Products

▲ Methods
  ⬡ MyStoreContext (+ 1 overloa...
  ⬡ OnConfiguring
  ⬡ OnModelCreating
  ⬡ OnModelCreatingPartial

```csharp
modelBuilder.Entity<Product>(entity =>
{
    entity.Property(e => e.ProductId).HasColumnName("ProductID");
    entity.Property(e => e.CategoryId).HasColumnName("CategoryID");
    entity.Property(e => e.ProductName)
        .IsRequired()
        .HasMaxLength(40)
        .UseCollation("Vietnamese_CI_AS");
    entity.Property(e => e.UnitPrice).HasColumnType("money");
    entity.HasOne(d => d.Category)
        .WithMany(p => p.Products)
        .HasForeignKey(d => d.CategoryId)
        .OnDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("FK_Products_Categories");
});

    OnModelCreatingPartial(modelBuilder);
}
partial void OnModelCreatingPartial(ModelBuilder modelBuilder);
}
```

4.Write codes for Program.cs then run project

```csharp
using System;
using System.Linq;
using DemoDatabaseFirst.Models;
using Microsoft.EntityFrameworkCore;
namespace DemoDatabaseFirst{
    class Program {
        static void Main(string[] args) {
            //Create a DbContext object
            MyStoreContext myStore = new MyStoreContext();
            //Print all Products
            var products = from p in myStore.Products
                           select new {p.ProductName,p.CategoryId};
            foreach (var p in products) {
                Console.WriteLine($"ProductName: {p.ProductName}, CategoryID: {p.CategoryId}");
            }
            Console.WriteLine("-------------------------------------------");
            // A query to get all Categories and their related Products
            IQueryable<Category> cats = myStore.Categories.Include(c => c.Products);
            foreach (Category c in cats){
                Console.WriteLine($"CategoryId: {c.CategoryId} has {c.Products.Count} products.");
            }
            Console.ReadLine();
        }//end Main
    }//end Class
}
```

```
D:\Slot_19_20_EntityFramework\DemoDatabaseFirst\bin\Debug\net5.0\DemoDatal

ProductName: Genen Shouyu, CategoryID: 1
ProductName: Alice Mutton, CategoryID: 1
ProductName: Aniseed Syrup, CategoryID: 3
ProductName: Perth Pasties, CategoryID: 2
ProductName: Carnarvon Tigers, CategoryID: 4
ProductName: Gula Malacca, CategoryID: 2
ProductName: Steeleye Stout, CategoryID: 7
ProductName: Chocolade, CategoryID: 5
ProductName: Mishi Kobe Niku, CategoryID: 6
ProductName: Ikura, CategoryID: 8
-------------------------------------------
CategoryId: 1 has 2 products.
CategoryId: 2 has 2 products.
CategoryId: 3 has 1 products.
CategoryId: 4 has 1 products.
CategoryId: 5 has 1 products.
CategoryId: 6 has 1 products.
CategoryId: 7 has 1 products.
CategoryId: 8 has 1 products.
```

http://vku.udn.vn/

The Weaknesses Reverse Engineering

- In the case of the temporal tables (called system-versioned tables) added in SQL Server , the history tables cannot be mapped using Entity Framework Core (this is already possible for the actual table)

- For database views and stored procedures, in contrast to the classic Entity Framework, classes and functions cannot be generated

- Once the object model is generated using the Entity Framework Core commandline tools, we cannot update it. The Update Model from Database command available for the Database First approach is currently not implemented

39

10/08/2023

Forward Engineering for New Databases

- Forward engineering is available in the classic Entity Framework in two variants: Model First and Code First

- In Model First, we graphically create an entity data model (EDM) to generate the database schema and .NET classes

- In Code First, we write classes directly, from which the database schema is created

- The EDM is invisible. In the redesigned Entity Framework Core, *there is only the second approach*, which however *is not called Code First* but code-based modeling and no longer uses an invisible EDM

Forward Engineering for New Databases

◆ Code-based modeling in Entity Framework Core happens through these two types of classes:

- We create entity classes, which store the data in RAM. We create navigation properties in the entity classes that represent the relationships between the entity classes. These are typically plain old CRL objects (POCOs) with properties for each database column

- We write a context class (derived from DbContext) that represents the database model, with each of the entities listed as a DBSet. This will be used for all queries and other operations

Forward Engineering for New Databases Demonstration

1.Create a Winform app named ManageCategoriesApp includes a form named frmManageCategories and has controls as follows :

Object Type	Object name	Properties / Events
Label	lbCategoryID	Text: lbCategoryID
Label	lbCategoryName	Text: CategoryName
TextBox	txtCategoryID	ReadOnly: True
TextBox	txtCategoryName	
Button	btnInsert	Text: Insert Event Handler: Click
Button	btnUpdate	Text: Update Event Handler: Click
Button	btnDelete	Text: Delete Event Handler: Click
DataGridView	dgvCategories	ReadOnly: True SelectionMode:FullRowSelect
Form	frmManageCategories	StartPosition: CenterScreen Text: Manage Categories Event Handler: Load

2.Right-click on the project | Add | New Item, select JavaScript JSON Configuration File then rename to appsettings.json , click Add and write contents as follows:

```
{
    "ConnectionStrings": {
     "MyStockDB": "Server=(local);uid=sa;pwd=123;database=MyStockDB"
    }
}
```

◆ Next , right-click on appsettings.json | Properties, select Copy if newer

Properties	▾ 📌 ✕
appsettings.json File Properties	▾
⊟ **Advanced**	
Build Action	**None**
Copy to Output Directory	**Copy if newer**
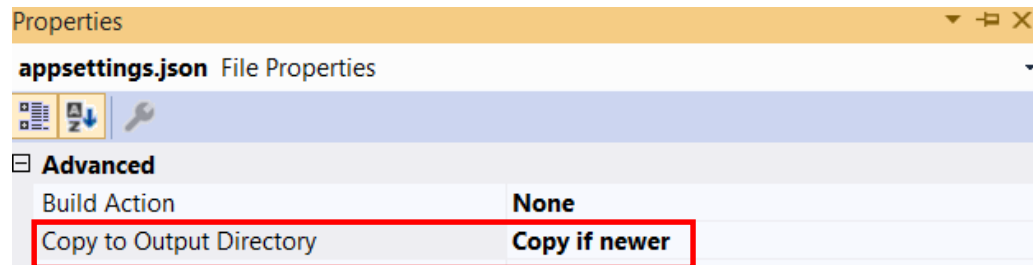
3.Install the following packages from Nuget

4.Add to the project **02 classes: ManageCategories.cs and MyStockDBContext.cs**

Solution Explorer

Search Solution Explorer (Ctrl+;)

- Solution 'Slot_19_20_EntityFramework' (2 of 2 projects)
  - **DemoCodeFirst**
    - Dependencies
      - Analyzers
      - Frameworks
      - Packages
        - Microsoft.EntityFrameworkCore.Design (5.0.3)
        - Microsoft.EntityFrameworkCore.SqlServer (5.0.3)
        - Microsoft.Extensions.Configuration (5.0.0)
        - Microsoft.Extensions.Configuration.Json (5.0.0)
    - appsettings.json
    - frmManageCategories.cs
    - ManageCategories.cs
    - MyStockDBContext.cs
    - Program.cs

5.Write codes **MyStockDBContext.cs** as follows:

```
//using more namespaces for Enitity Framwork Core
using System.IO;
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Configuration.Json;
```

```csharp
//Declare Category Entity
public class Category{
    public Category(){ }
    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
}//end Categories
public class MyStock : DbContext {
    public MyStock(){ }
    // These properties map to tables in the database
    public DbSet<Category> Categories { get; set; }
    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder){
        var builder = new ConfigurationBuilder()
          .SetBasePath(Directory.GetCurrentDirectory())
          .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true);
        IConfigurationRoot configuration = builder.Build();
        optionsBuilder.UseSqlServer(configuration.GetConnectionString("MyStockDB"));
    }
```

```csharp
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    // Using Fluent API instead of attributes
    // to limit the length of a Category Name to under 40
    modelBuilder.Entity<Category>()
        .Property(category => category.CategoryName)
        .IsRequired() // NOT NULL
        .HasMaxLength(40);
    //Insert data for Categories table
    modelBuilder.Entity<Category>().HasData(
        new Category { CategoryID = 1,  CategoryName= "Beverages" },
        new Category { CategoryID = 2, CategoryName= "Condiments" },
        new Category { CategoryID = 3, CategoryName= "Confections" }
    );
}
}//end MyStock class
```

6. Right-click on the project, select Open in Terminal. On Developer PowerShell dialog , execute the following commands to generate database:

- **dotnet ef migrations add "Initial"**

- **dotnet ef database update**

Developer PowerShell

```
************************************************************
PS D:\Slot_19_20_EntityFramework\DemoCodeFirst> dotnet ef migrations  add "Initial"
Build started...
Build succeeded.
Done. To undo this action, use 'ef migrations remove'
PS D:\Slot_19_20_EntityFramework\DemoCodeFirst> dotnet ef database update
Build started...
Build succeeded.
Applying migration '20210309082932_Initial'.
Done.
```

Solution 'Slot_19_20_EntityFramework' (
- DemoCodeFirst
  - Dependencies
    - Analyzers
    - Frameworks
    - Packages
  - Migrations
    - 20210309082932_Initial.cs
    - MyStockModelSnapshot.cs
  - appsettings.json
  - frmManageCategories.cs
  - ManageCategories.cs
  - MyStockDBContext.cs
  - Program.cs

6. Write codes **ManageCategories.cs** as follows:

```csharp
public sealed class ManageCategories{
    //Using Singleton Pattern
    private static ManageCategories instance = null;
    private static readonly object instanceLock = new object();
    private ManageCategories() { }
    public static ManageCategories Instance{
        get{
            lock (instanceLock){
                if (instance == null){
                    instance = new ManageCategories();
                }
                return instance;
            }
        }
    }
}
```

```csharp
//--------------------------------------
public List<Category> GetCategories() {
    List<Category> categories;
    try
    {
        using MyStock stock = new MyStock();
        categories = stock.Categories.ToList();
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }

    return categories;
}//end GetCategories
//--------------------------------------------
public void InsertCategory(Category category) {
    try {
        using MyStock stock = new MyStock();
        stock.Categories.Add(category);
        stock.SaveChanges();
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
}//end InsertCategory
```
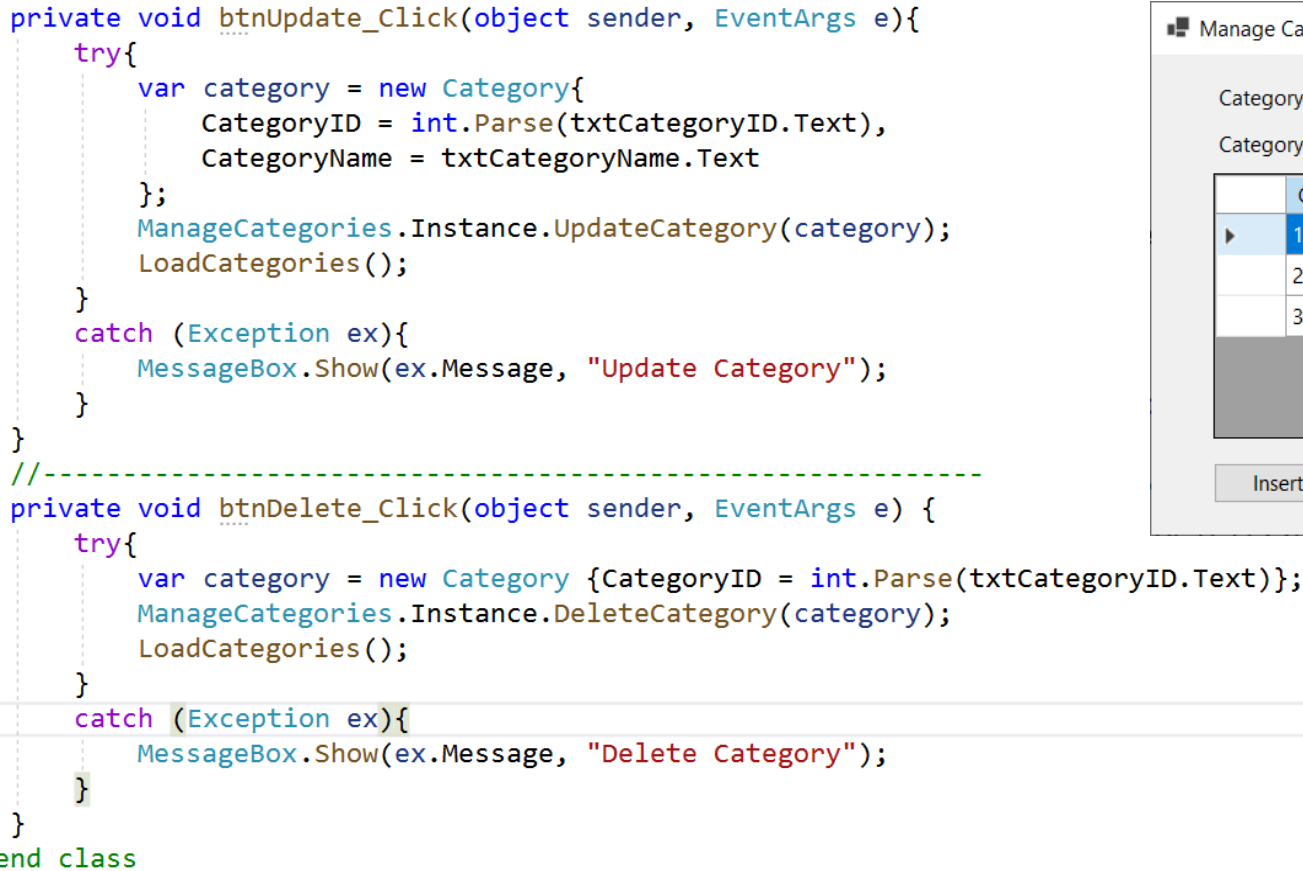
```csharp
    //----------------------------------------------
    public void UpdateCategory(Category category){
        try {
            using MyStock stock = new MyStock();
            stock.Entry<Category>(category).State = Microsoft.EntityFrameworkCore.EntityState.Modified;
            stock.SaveChanges();
        }
        catch (Exception ex){
            throw new Exception(ex.Message);
        }
    }//end UpdateCategory
    //----------------------------------------------
    public void DeleteCategory(Category category) {
        try {
            using MyStock stock = new MyStock();
            //Find Category by CategoryID
            var cate = stock.Categories.SingleOrDefault(c => c.CategoryID == category.CategoryID);
            stock.Categories.Remove(cate);
            stock.SaveChanges();
        }
        catch (Exception ex){
            throw new Exception(ex.Message);
        }
    }//end DeleteCategory
}//end ManageCategories
```

8.Write codes in **frmManageCategories.cs** as follows then press Ctrl+F5 to run project:

```csharp
public partial class frmManageCategories : Form{
    public frmManageCategories() ...
    private void LoadCategories(){
        var categories = ManageCategories.Instance.GetCategories();
        txtCategoryID.DataBindings.Clear();
        txtCategoryName.DataBindings.Clear();
        //Binding to TextBoxes
        txtCategoryID.DataBindings.Add("Text", categories, "CategoryID");
        txtCategoryName.DataBindings.Add("Text", categories, "CategoryName");
        //Binding to DataGridView
        dgvCategories.DataSource = categories;
    }
    //-------------------------------------------------------------------
    private void frmManageCategories_Load(object sender, EventArgs e) => LoadCategories();
    //-------------------------------------------------------------------
    private void btnInsert_Click(object sender, EventArgs e){
        try{
            var category = new Category { CategoryName = txtCategoryName.Text };
            ManageCategories.Instance.InsertCategory(category);
            LoadCategories();
        }
        catch (Exception ex){
            MessageBox.Show(ex.Message, "Insert Category");
        }
    }
    //-------------------------------------------------------------------
```

```csharp
private void btnUpdate_Click(object sender, EventArgs e){
    try{
        var category = new Category{
            CategoryID = int.Parse(txtCategoryID.Text),
            CategoryName = txtCategoryName.Text
        };
        ManageCategories.Instance.UpdateCategory(category);
        LoadCategories();
    }
    catch (Exception ex){
        MessageBox.Show(ex.Message, "Update Category");
    }
}
//-----------------------------------------------------------
private void btnDelete_Click(object sender, EventArgs e) {
    try{
        var category = new Category {CategoryID = int.Parse(txtCategoryID.Text)};
        ManageCategories.Instance.DeleteCategory(category);
        LoadCategories();
    }
    catch (Exception ex){
        MessageBox.Show(ex.Message, "Delete Category");
    }
}
}//end class
```

Manage Categories

CategoryID: 1
CategoryName: Beverages

| CategoryID	CategoryName
1	Beverages
2	Condiments
3	Confections

Insert   Update   Delete

Querying EF Core Models

◆ Entity Framework Core allows us to write database queries with Language Integrated Query (LINQ)

◆ The starting point for all LINQ queries in Entity Framework Core is the context class that we create either during the reverse engineering of an existing database or manually while forward engineering

◆ The context class in Entity Framework Core always inherits from the base class Microsoft.EntityFrameworkCore.DbContext. Accordingly, we have to use DbContext for all LINQ operations

◆ The DbContext class implements the IDisposable interface. As part of the Dispose() method, DbContext frees all allocated resources, including references to all objects loaded with change tracking

LINQ Queries

◆ After instantiating the context class, we can formulate a LINQ query. This query is not necessarily executed immediately; it is initially in the form of an object with the interface IQueryable<T>

◆ The LINQ query is executed when the result is actually used (for example, in a foreach loop) or when converted to another collection type

◆ We can force the execution of the query with a LINQ conversion operator with ToList(), ToArray(), ToLookup(), ToDictionary(), Single(), SingleOrDefault(), First(), FirstOrDefault(), or an aggregate operator such as Count(), Min(), Max(), or Sum()

10/08/2023

54

Internals for running a LINQ command through Entity Framework Core
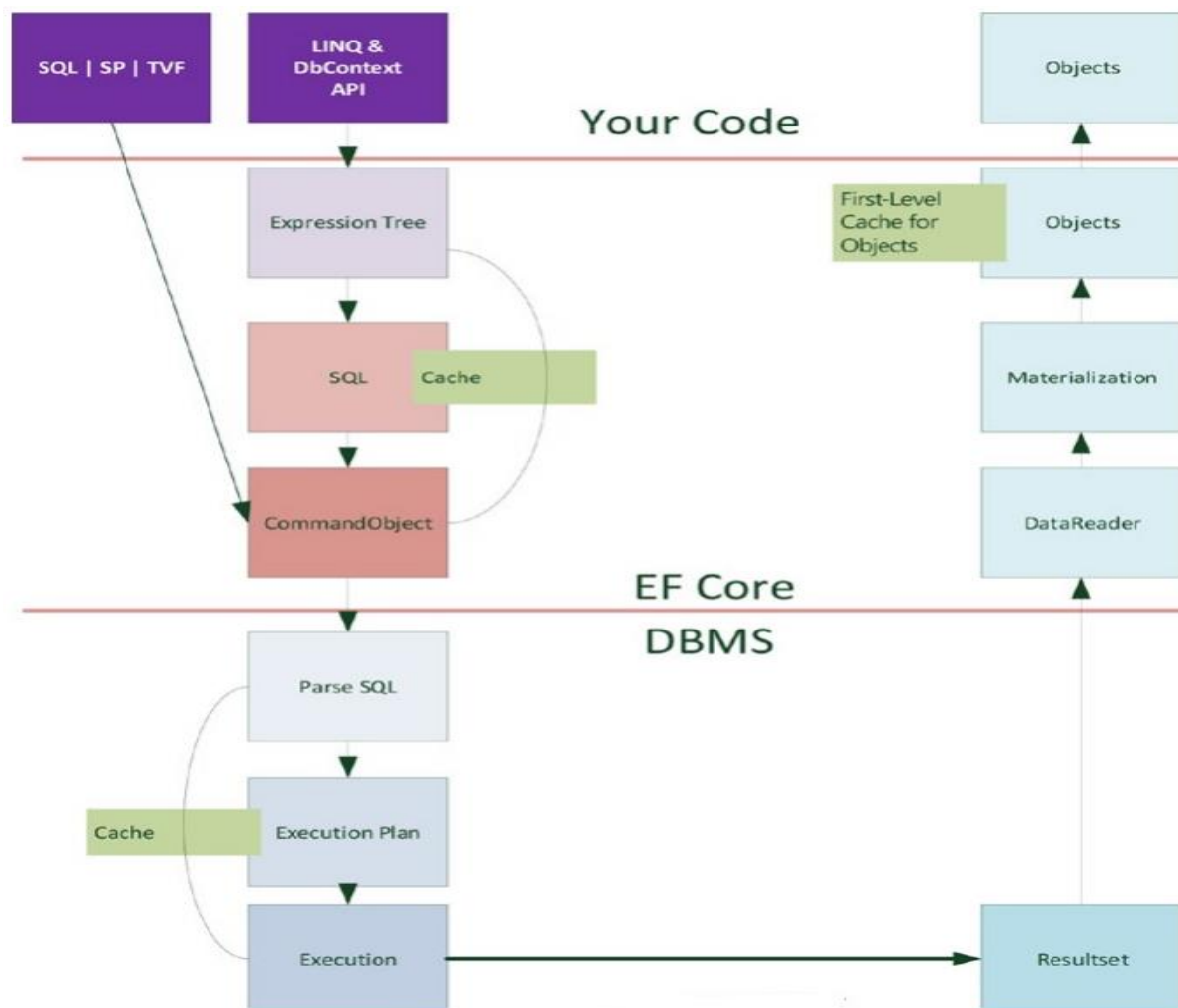
LINQ Queries Demonstrations
(using Reverse Engineering of Existing Databases Demo)

Create a query for categories that have products with that minimum number of units in stock. Enumerate through the categories and products, outpuing the name and units in stock for each one (using Reverse Engineering of Existing Databases

```csharp
static void FilteredIncludes(){
    using var db = new MyStoreContext();
    Console.Write("Enter a minimum for units in stock: ");
    string unitsInStock = Console.ReadLine();
    int stock = int.Parse(unitsInStock);
    IQueryable<Category> cats = db.Categories
    .Include(c => c.Products.Where(p => p.UnitsInStock >= stock));
    foreach (Category c in cats){
        Console.WriteLine($"{c.CategoryName} has {c.Products.Count} product");
        foreach (Product p in c.Products){
            Console.WriteLine($"--->{p.ProductName} has {p.UnitsInStock} units in stock");
        }
    }
}//end FilteredIncludes
static void Main(string[] args){
    FilteredIncludes();
    Console.ReadLine();
}//end Main
```

```
D:\Slot_19_20_EntityFramework\DemoDatabaseFirst\bin\Debug\net5.0\DemoD
Enter a minimum for units in stock: 20
Beverages has 1 product
--->Genen Shouyu has 39 units in stock
Condiments has 2 product
--->Perth Pasties has 53 units in stock
--->Gula Malacca has 120 units in stock
Confections has 0 product
Dairy Products has 0 product
Grains/Cereals has 0 product
Meat/Poultry has 1 product
--->Mishi Kobe Niku has 29 units in stock
Produce has 0 product
Seafood has 1 product
--->Ikura has 31 units in stock
```

10/08/2023

Create a query for products that cost more than the price

```csharp
static void QueryingProducts(){
    using (var db = new MyStoreContext()){
        Console.WriteLine("Products that cost more than a price, highest at top");
        string input;
        decimal price;
        do{
            Console.Write("Enter a product price: ");
            input = Console.ReadLine();
        } while (!decimal.TryParse(input, out price));
        IQueryable<Product> prods = db.Products
        .Where(product => product.UnitPrice > price)
        .OrderByDescending(product => product.UnitPrice);
        foreach (Product item in prods){
            Console.WriteLine($"ProductName: {item.ProductName} costs {item.UnitPrice:$#,##0.00} " +
                $"and has { item.UnitsInStock} in stock.");
        }
    }
}//end QueryingProducts
static void Main(string[] args){
    QueryingProducts();
    Console.ReadLine();
}//end Main
```

```
D:\Slot_19_20_EntityFramework\DemoDatabaseFirst\bin\Debug\net5.0\DemoDatabaseFirst.exe

Products that cost more than a price, highest at top
Enter a product price: 10
ProductName: Mishi Kobe Niku costs $97.00 and has 29 in stock.
ProductName: Genen Shouyu costs $50.00 and has 39 in stock.
ProductName: Aniseed Syrup costs $40.00 and has 13 in stock.
ProductName: Chocolade costs $40.00 and has 6 in stock.
ProductName: Ikura costs $31.00 and has 31 in stock.
ProductName: Steeleye Stout costs $30.00 and has 15 in stock.
ProductName: Alice Mutton costs $30.00 and has 17 in stock.
ProductName: Gula Malacca costs $25.00 and has 120 in stock.
ProductName: Perth Pasties costs $22.00 and has 53 in stock.
ProductName: Carnarvon Tigers costs $21.35 and has 0 in stock.
```

10/08/2023

58

Perform aggregation functions, such as Average and Sum on the Products table

```csharp
static void AggregateProducts()
{
    using (var db = new MyStoreContext())
    {
        Console.WriteLine("{0,-25} {1,10}", arg0: "Product count:", arg1: db.Products.Count());
        Console.WriteLine("{0,-25} {1,10:$#,##0.00}", arg0: "Highest product price:",arg1: db.Products.Max(p => p.UnitPrice));
        Console.WriteLine("{0,-25} {1,10:N0}",arg0: "Sum of units in stock:",arg1: db.Products.Sum(p => p.UnitsInStock));
        Console.WriteLine("{0,-25} {1,10:$#,##0.00}",arg0: "Average unit price:",arg1: db.Products.Average(p => p.UnitPrice));
        Console.WriteLine("{0,-25} {1,10:$#,##0.00}",arg0: "Value of units in stock:",
            arg1: db.Products.AsEnumerable().Sum(p => p.UnitPrice * p.UnitsInStock));
    }
}
static void Main(string[] args){
    AggregateProducts();
    Console.ReadLine();
}//end Main
```

D:\SLot_19_20_EntityFramework\DemoDatabaseFirst\bin\Debug\net5.

```
Product count:                    10
Highest product price:        $97.00
Sum of units in stock:           323
Average unit price:           $38.64
Value of units in stock:  $11,610.00
```

10/08/2023

59