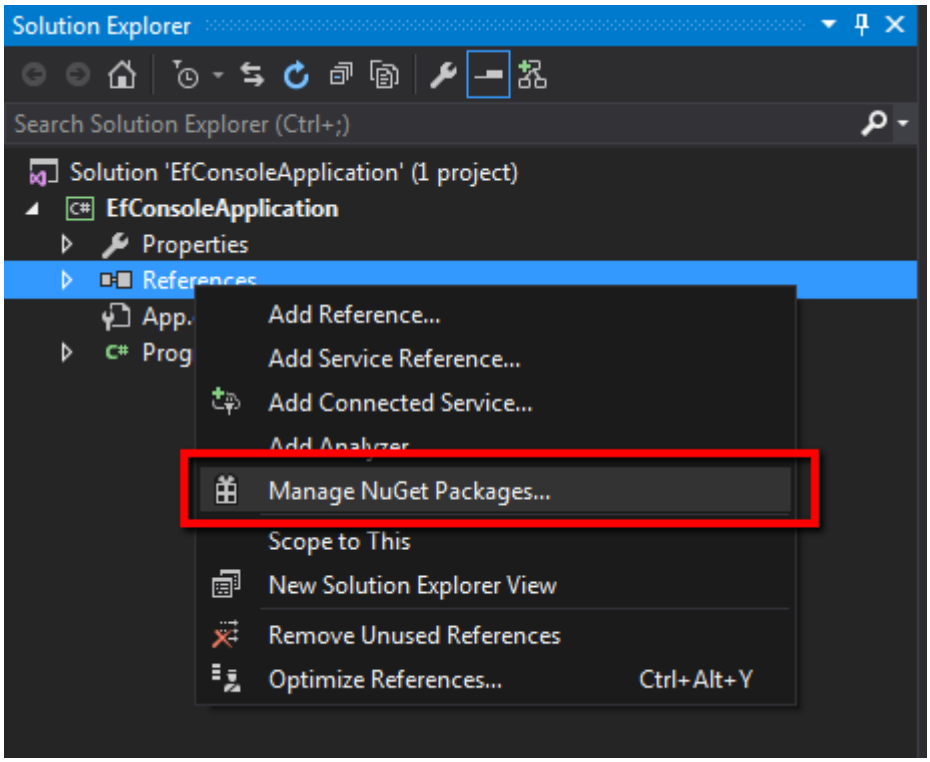
# Упражнения: Да направим CRUD приложение с ORM

## Създаване на просто приложение

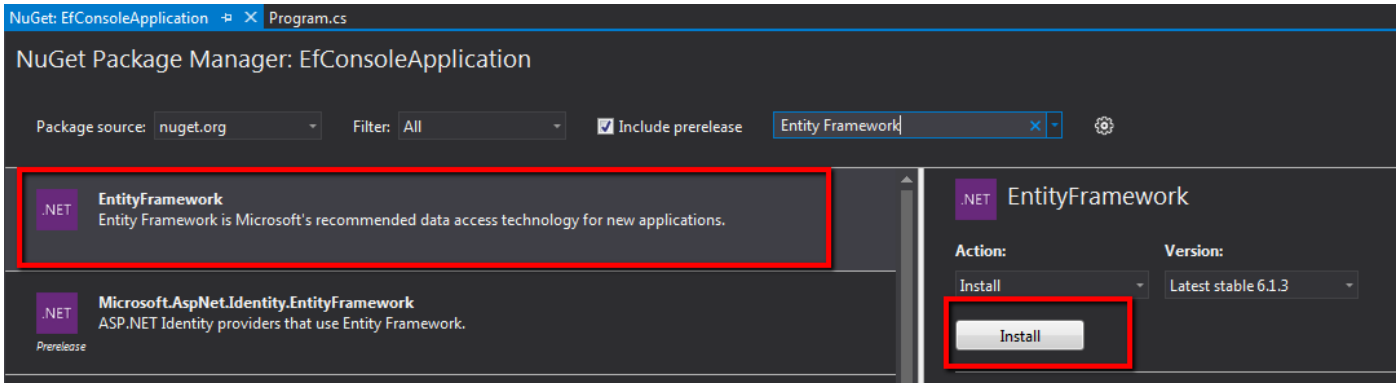
В рамките на това упражнение ще направим приложение аналогично с това от предното упражнение, с разликата че ще използваме **EntityFramework** като ORM.

## 1. Добавяне на EntityFramework

Ще започнем проекта с добавяне на **EntityFramework**. За тази цел, непосредствено след създаването на проекта, цъкнете с десен бутон върху **References** и изберете **Manage NuGet Packages…**




След това изберете **Entity Framework** и натиснете **Install:**




С това сме готови да продължим нататък ☺

## 2. База данни

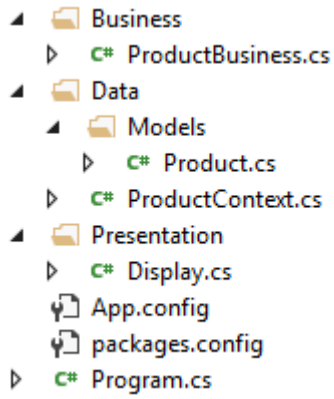
Тук ще използваме **Code-first** принципа, при който първо създаваме код, а по него автоматично се създава база данни. Съществува и **Database-first** принцип, но тук с цел лекота на упражнението се спираме на **Code-first**.

Единственото нещо, което трябва да направим е да отворим нашият App.config файл и там да добавим **connectionStrings** XML код, например преди **</configuration>.**

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=6.
  </configSections>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.LocalDbConnectionFactory, EntityFramework">
      <parameters>
        <parameter value="v11.0" />
      </parameters>
    </defaultConnectionFactory>
    <providers>
      <provider invariantName="System.Data.SqlClient" type="System.Data.Entity.SqlServer.SqlProviderServices, EntityFramework.SqlSer
    </providers>
  </entityFramework>
  <connectionStrings>
    <add name="ProductContext" connectionString="Data Source=.; Initial Catalog=ProductDb; Integrated Security=true"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

## 3. Структура на проекта

Структурата тук отново ще е трислойна, като тя всъщност доста прилича на предното приложение. Разбира се тук има разлика и тя е, че в папката **Data** има подпапка **Model –** повече за това по-късно.




## 4. Слой за данни

Слоят за данни тук се състои от папка с **модел** и **контекст**.

Моделът тук е клас **Product.cs,** находящ се в **Model** папката, като в него описваме единствено свойствата му, които всъщност съответстват на колоните от таблицата.

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Stock { get; set; }
}
```

В самата папка **Data** се намира и **ProductContext** класа. Той трябва да наследява **DbContext**. Тук ще се наложи да добавите и **using** директива, понеже класа **DbContext** е част от **EntityFramework**.
Самият клас ще съдържа конструктор, в който ще има обръщение към конструктора на базовия клас, където за параметър се подава името (**name** атрибутът) от низа за връзка, който добавихме в **App.config** по-рано.

Освен това в конструктора ще имаме и свойство, което ще е от **DbSet<Product>.**

Кодът е както следва:

```
public class ProductContext : DbContext
{
    public ProductContext()
        : base("name=ProductContext")
    {

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

С това сме готови с нашия слой за данни.

## 5. Бизнес слой

Бизнес слоят тук доста прилича на предното упражнение. Разбира се има някои разлики:

Тук ще имаме поле от тип **ProductContext**, което ще използваме в методите.

Методите като логика работят по абсолютно сходен начин с предното упражнение. Класът изглежда по следния начин:

```
class ProductBusiness
{
    private ProductContext productContext;

    public List<Product> GetAll()...

    public Product Get(int id)...

    public void Add(Product product)...

    public void Update(Product product)...

    public void Delete(int id)...
}
```

А методите са съответно:

```
public List<Product> GetAll()
{
    using (productContext = new ProductContext())
    {
        return productContext.Products.ToList();
    }
}

public Product Get(int id)
{
    using (productContext = new ProductContext())
    {
        return productContext.Products.Find(id);

    }
}

public void Add(Product product)
{
    using (productContext = new ProductContext())
    {
        productContext.Products.Add(product);
        productContext.SaveChanges();

    }
}

public void Update(Product product)
{
    using (productContext = new ProductContext())
    {

        var item = productContext.Products.Find(product.Id);
        if (item != null)
        {
            productContext.Entry(item).CurrentValues.SetValues(product);
            productContext.SaveChanges();
        }

    }
}

public void Delete(int id)
{
    using (productContext = new ProductContext())
    {
        var product = productContext.Products.Find(id);
        if (product != null)
        {
            productContext.Products.Remove(product);
            productContext.SaveChanges();
        }
    }
}
```

# 6. Презентационен слой

**Презентационния слой тук е абсолютно идентичен с предното упражнение.**

Тук ще имаме Display.cs клас. Той ще реализира конзолно меню, от което ще въвеждаме желана опция и съответно по този начин ще бъде контролиран вход/изхода на програмата.

Ще започнем със създаването на частен **ShowMenu()** метод, който ще бъде викан, за да показва какви са възможностите:

```
private void ShowMenu()
{
    Console.WriteLine(new string('-', 40));
    Console.WriteLine(new string(' ', 18)+"MENU"+new string(' ', 18));
    Console.WriteLine(new string('-', 40));
    Console.WriteLine("1. List all entries");
    Console.WriteLine("2. Add new entry");
    Console.WriteLine("3. Update entry");
    Console.WriteLine("4. Fetch entry by ID");
    Console.WriteLine("5. Delete entry by ID");
    Console.WriteLine("6. Exit");
}
```

Този метод ще се вика от друг частен метод – **Input().** Целта на този метод е да получим вход от потребителя – номер на желаната операция и според това да извикаме някой от другите методи. Методът може да бъде реализиран, чрез позната за вас **do/while** конструкция и **switch/case** конструкция. Една примерна реализация на метода би изглеждала ето така:

```
private void Input()
{
    var operation = -1;
    do
    {
        ShowMenu();
        operation = int.Parse(Console.ReadLine());
        switch (operation)
        {
            case 1:
                ListAll();
                break;
            case 2:
                Add();
                break;
            case 3:
                Update();
                break;
            case 4:
                Fetch();
                break;
            case 5:
                Delete();
                break;
            default:
                break;
        }
    } while (operation != closeOperationId);
}
```

В кода по-горе **closeOperationId** е поле на класа Display, в което задаваме номерът на операцията за затваряне на приложението, при въвеждането на който трябва да спрем да приемаме вход от потребителя:

```
private int closeOperationId = 6;
```

Самото извикване на **Input()** метода ще се случва в конструктора на класа Display:

```
public Display()
{
    Input();
}
```

Сега трябва да реализираме останалите методи. Ще започнем с **Add().** Водещата задача през този метод е той да въвежда информация за нов продукт. Съответно в него ние в отделни променливи ще въвеждаме името, цената и наличността на продукта, след което ще създаваме обект от клас **Product** и чрез обектът **productBusiness** ще извикваме **Add()** метода на бизнес логиката.

Кодът изглежда както следва:

```
private void Add()
{
    Product product = new Product();
    Console.WriteLine("Enter name: ");
    product.Name = Console.ReadLine();
    Console.WriteLine("Enter price: ");
    product.Price = decimal.Parse(Console.ReadLine());
    Console.WriteLine("Enter stock: ");
    product.Stock = int.Parse(Console.ReadLine());
    productBusiness.Add(product);
}
```

Следващият метод, който ще бъде реализиран е **ListAll().** В него трябва да си създадем променлива **products,** която ще получи своята стойност благодарение на метода **GetAll()** от бизнес логиката. След това трябва просто да обходим всички получени елементи и да ги изведем. Тук може да проявите въображение по начина на извеждане на елементите и да изведете информацията и по-красиво, тъй като примерната реализация не е направена особено красиво ☺

```
private void ListAll()
{
    Console.WriteLine(new string('-', 40));
    Console.WriteLine(new string(' ', 16)+"PRODUCTS"+new string(' ', 16));
    Console.WriteLine(new string('-', 40));
    var products = productBusiness.GetAll();
    foreach (var item in products)
    {
        Console.WriteLine("{0} {1} {2} {3}", item.Id, item.Name, item.Price, item.Stock);
    }
}
```

Сега, след като можем да виждаме вече всички елементи в таблицата, нека да преминем към възможността да ги редактираме. Това ще се случва в методът **Update(),** той ще иска от потребителя **id** на продукта за редактиране. Ако такъв продукт действително съществува, потребителят ще въвежда новите данни за него. Забележете, че този метод може да се направи значително по-удобен, отколкото в примера – например да показва досегашната стойност на свойствата в обекта... и не само ☺

```
private void Update()
{
    Console.WriteLine("Enter ID to update: ");
    int id = int.Parse(Console.ReadLine());
    Product product = productBusiness.Get(id);
    if (product != null)
    {
        Console.WriteLine("Enter name: ");
        product.Name = Console.ReadLine();
        Console.WriteLine("Enter price: ");
        product.Price = decimal.Parse(Console.ReadLine());
        Console.WriteLine("Enter stock: ");
        product.Stock = int.Parse(Console.ReadLine());
        productBusiness.Update(product);
    }
    else
    {
        Console.WriteLine("Product not found!");
    }
}
```

След като направихме методът за редактиране, е време да направим и метод за визуализиране на информацията по **id** на даден продукт. Тук ще използваме **Get()** метода от бизнес логиката.

```
private void Fetch()
{
    Console.WriteLine("Enter ID to fetch: ");
    int id = int.Parse(Console.ReadLine());
    Product product = productBusiness.Get(id);
    if (product != null)
    {
        Console.WriteLine(new string('-', 40));
        Console.WriteLine("ID: " + product.Id);
        Console.WriteLine("Name: " + product.Name);
        Console.WriteLine("Price: " + product.Price);
        Console.WriteLine("Stock: " + product.Stock);
        Console.WriteLine(new string('-', 40));
    }
}
```

Накрая ще реализираме и метод за изтриване на продукт по неговото **id.** Тук отново ще използваме съответния метод от бизнес логиката:

```
private void Delete()
{
    Console.WriteLine("Enter ID to delete: ");
    int id = int.Parse(Console.ReadLine());
    productBusiness.Delete(id);
    Console.WriteLine("Done.");
}
```

За да завършим нашето приложение имаме нужда единствено от създаването на обект от клас **Display** в рамките на **Main** метода в **Program.cs.**

# Министерство на образованието и науката (МОН)

- Настоящият курс (презентации, примери, задачи, упражнения и др.) е разработен за нуждите на Национална програма "**Обучение за ИТ кариера**" на МОН за подготовка по професия "Приложен програмист".



- Курсът е базиран на учебно съдържание и методика, предоставени от **фондация "Софтуерен университет"** и се разпространява под **свободен лиценз CC-BY-NC-SA** (Creative Commons Attribution-Non-Commercial-Share-Alike 4.0 International).