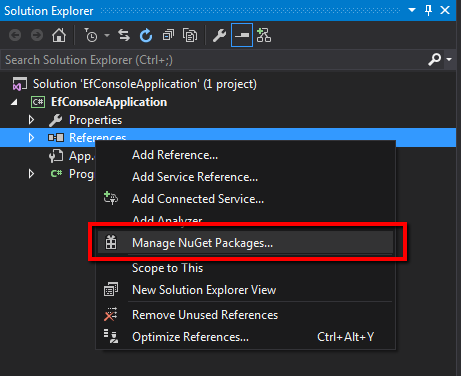
# Упражнения: Да направим CRUD приложение с ORM

## Създаване на просто приложение

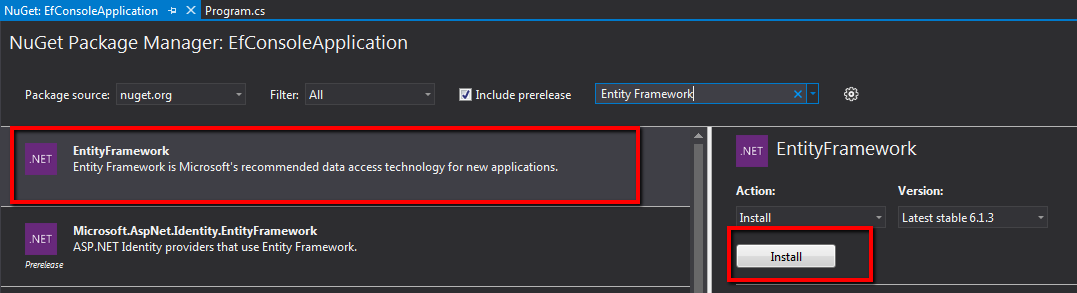
В рамките на това упражнение ще направим приложение аналогично с това от предното упражнение, с разликата че ще използваме **EntityFramework** като ORM.

## 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 ☺

## База данни

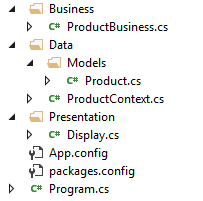
Тук ще използваме **Code-first** принципа, при който първо създаваме код, а по него автоматично се създава база данни. Съществува и **Database-first** принцип, но тук с цел лекота на упражнението се спираме на **Code-first**.

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



## Структура на проекта

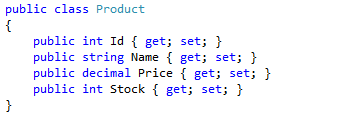
Структурата тук отново ще е трислойна, като тя всъщност доста прилича на предното приложение. Разбира се тук има разлика и тя е, че в папката **Data** има подпапка **Model –** повече за това по-късно.



## Слой за данни

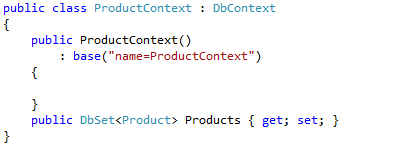
Слоят за данни тук се състои от папка с **модел** и **контекст**.

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.



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** e 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>.**

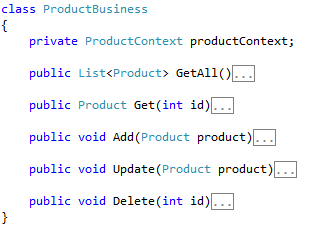
Кодът е както следва:  


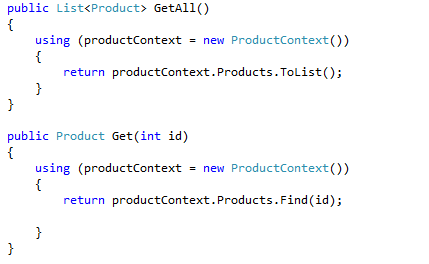
С това сме готови с нашия слой за данни.

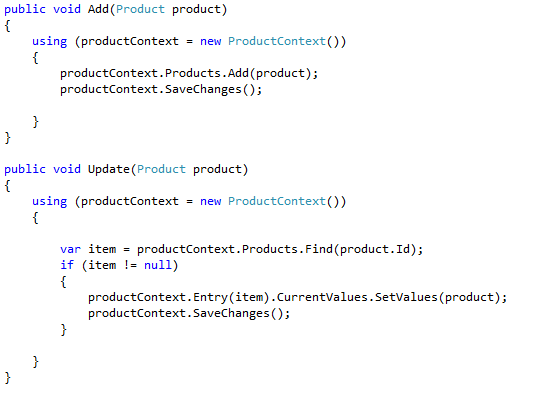
## Бизнес слой

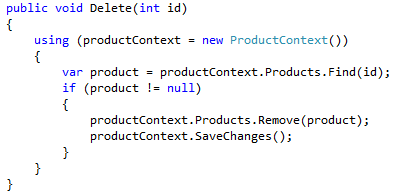
Бизнес слоят тук доста прилича на предното упражнение. Разбира се има някои разлики:

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:  


А методите са съответно:  




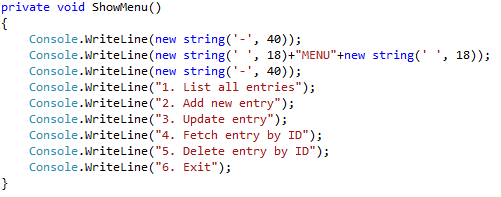


## Презентационен слой

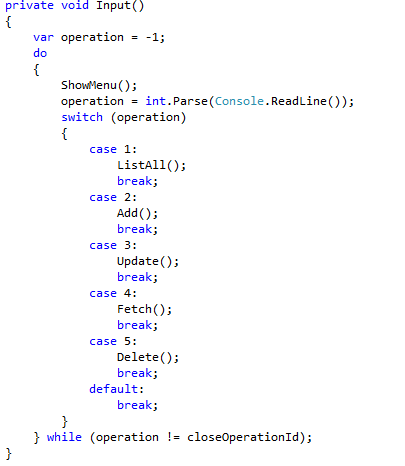
**Презентационния слой тук е абсолютно идентичен с предното упражнение.**

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:



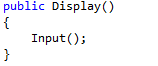
Този метод ще се вика от друг частен метод – **Input().** Целта на този метод е да получим вход от потребителя – номер на желаната операция и според това да извикаме някой от другите методи. Методът може да бъде реализиран, чрез позната за вас **do/while** конструкция и **switch/case** конструкция. Една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



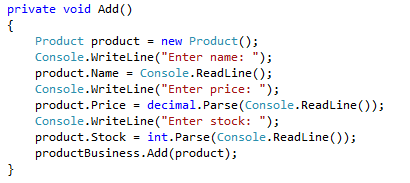
В кода по-горе **closeOperationId** е поле на класа Display, в което задаваме номерът на операцията за затваряне на приложението, при въвеждането на който трябва да спрем да приемаме вход от потребителя:



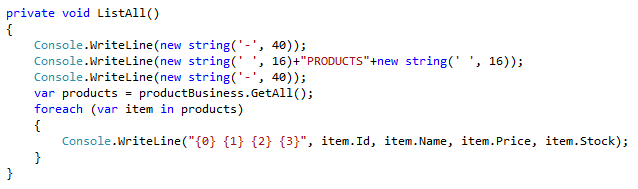
Самото извикване на **Input()** метода ще се случва в конструктора на класа Display:



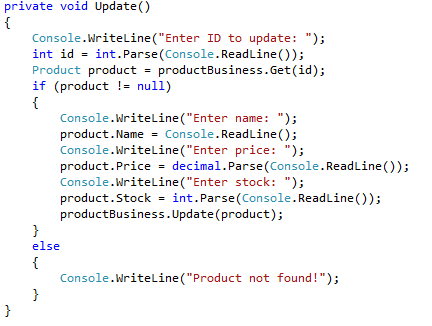
Сега трябва да реализираме останалите методи. Ще започнем с **Add().** Водещата задача през този метод е той да въвежда информация за нов продукт. Съответно в него ние в отделни променливи ще въвеждаме името, цената и наличността на продукта, след което ще създаваме обект от клас **Product** и чрез обектът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

Кодът изглежда както следва:  


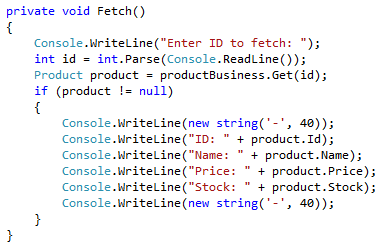
Следващият метод, който ще бъде реализиран е **ListAll().** В него трябва да си създадем променлива **products,** която ще получи своята стойност благодарение на метода **GetAll()** от бизнес логиката. След това трябва просто да обходим всички получени елементи и да ги изведем. Тук може да проявите въображение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



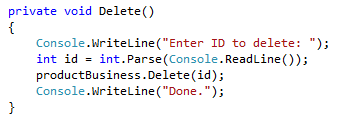
Сега, след като можем да виждаме вече всички елементи в таблицата, нека да преминем към възможността да ги редактираме. Това ще се случва в методът **Update(),** той ще иска от потребителя **id** на продукта за редактиране. Ако такъв продукт действително съществува, потребителят ще въвежда новите данни 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 ☺



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.



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



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