Технически университет – София
Факултет по компютърни системи и технологии

Курсова работа

Програмни среди

Тема 1: Система за търсене и показване на годишни списъци

Изготвил: Кристиян Любомиров Стойков
Специалност: КСИ
Фак. номер: 121221086
Група: 45

21.05.2024

Contents

Увод

Настоящият проект представлява приложение, разработено на C# с използване на Windows Presentation Foundation (WPF) и архитектурния модел Model-View-ViewModel (MVVM). Приложението е предназначено за управление на информация за автомобилни ремонти и свързаните с тях превозни средства, като използва база данни SQLite и EntityFramework за съхранение на данните.

Подобни разработки

AutoFluent

AutoFluent е автоматизирана система за управление на автосервизи, която предлага богат набор от функционалности, насочени към улесняване и оптимизиране на операциите в автомобилни сервизи и гаражи. Системата е създадена да подпомогне управлението на различни аспекти на бизнеса, включително инвентар, поръчки, клиенти и финансови отчети.

Програмата предлага следните функционалности:

- **Управление на клиенти**: Поддържане на детайлна база данни с информация за клиентите, включително история на поръчките и комуникациите.
- **Управление на превозни средства**: Съхранение на информация за превозните средства на клиентите, включително модел, марка, регистрационен номер и история на ремонтите.
- **Управление на инвентар**: Следене на наличностите и автоматизирано управление на запасите от части и консумативи.
- **Управление на поръчки**: Създаване и проследяване на поръчки за ремонт и обслужване, включително детайли за извършените услуги и използваните части.
- **Фактуриране и плащания**: Генериране на фактури и управление на плащанията, включително интеграция с различни платежни системи.
- **Отчети и анализи**: Генериране на различни отчети и анализи, които подпомагат управлението и вземането на бизнес решения.

Някои отрицателни черти:

- **Сложност на конфигурацията**: За нови потребители и малки автосервизи, първоначалната конфигурация може да бъде сложна и време отнемаща.
- **Цена**: Високата цена на лицензите може да бъде пречка за малки автосервизи, които разполагат с ограничен бюджет.

Auto Repair Cloud

Auto Repair Cloud е облачно базирана система за управление на автосервизи, предназначена да улесни и автоматизира процесите в автомобилните сервизи и гаражи. Системата предлага множество функционалности, които покриват всички аспекти на управлението на автосервизи, като същевременно предоставя удобство и гъвкавост чрез облачната си архитектура.

Функционалност

- **Управление на клиенти**: Съхранение на детайлна информация за клиентите, включително история на поръчките, комуникациите и предпочитанията.
- **Управление на превозни средства**: Съхранение на информация за превозните средства на клиентите, включително марка, модел, регистрационен номер и история на ремонтите.
- **Управление на инвентар**: Следене на наличностите от части и консумативи, автоматизирано попълване на запаси и управление на доставките.
- **Управление на поръчки**: Създаване и проследяване на поръчки за ремонт и обслужване, включително детайли за извършените услуги и използваните части.
- **Фактуриране и плащания**: Генериране на фактури и управление на плащанията, включително интеграция с различни платежни системи.
- **Комуникация с клиенти**: Възможност за изпращане на съобщения и уведомления до клиентите чрез различни канали, включително SMS и имейл.
- **Мобилно приложение**: Предоставя мобилно приложение за лесен достъп и управление на информацията от всяко място.

Отрицателни страни

- **Зависимост от интернет връзка**: Необходимостта от постоянна интернет връзка може да бъде пречка в райони с нестабилна интернет връзка.
- **Цена**: Високата цена на абонамента може да бъде пречка за малки автосервизи с ограничен бюджет.
- **Ограничена офлайн функционалност**: Липсата на офлайн режим може да затрудни работата при липса на интернет връзка.

Анализ на заданието

Целта на проекта е да се разработи система за управление на информация за автомобилни ремонти и свързаните с тях превозни средства. Системата ще предоставя функционалности за търсене, показване и управление на годишни списъци с ремонти, като използва C# с WPF, архитектурния модел MVVM и база данни SQLite с EntityFramework.

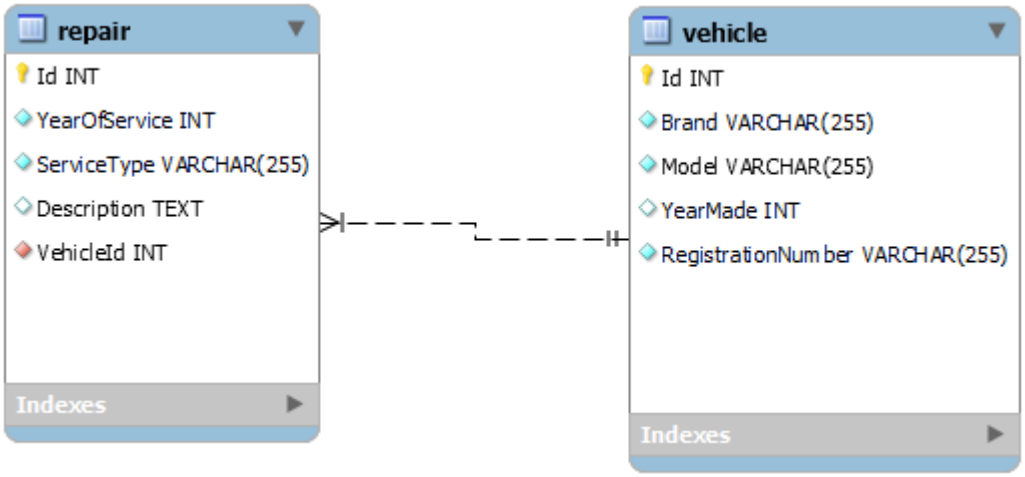
Функционални изисквания

- **Търсене и филтриране**: Потребителят трябва да може да задава критерии за търсене (филтри) за полетата на обектите. Системата трябва да позволява търсене по конкретна година и връщане на резултати за тази година
- **Алтернативни резултати**: Ако няма резултати за посочената година, системата трябва да връща резултати от последната налична година, за която има данни, отговарящи на останалите критерии.

- **Гъвкавост и повторна употреба**: Системата трябва да може да се прилага върху различни типове годишни списъци и да се интегрира лесно в други проекти.
- **Спазване на MVVM**: Архитектурата на системата трябва да предразполага към спазване на MVVM модела.
- **Примерни данни:** Базата данни трябва да съдържа примерни данни за демонстрация.

Проектиране и реализация

ER диаграма на клас models



Файлова структура

```
Vehicle_Repairs
├── Dependencies
├── Commands
│   └── DelegateCommand.cs
├── Converter
│   └── ReverseBooleanVisibilityConverter.cs
├── Database
│   ├── DatabaseContext.cs
│   └── DatabaseService.cs
├── Model
│   ├── Repair.cs
│   └── Vehicle.cs
├── Observable
│   └── ObservableObject.cs
├── Theme
│   └── Fonts
│       ├── Poppins-Bold.ttf
│       ├── Poppins-Medium.ttf
│       ├── Poppins-MediumItalic.ttf
│       └── Poppins-Regular.ttf
├── View
│   ├── Controls
│   │   ├── AddRepairControl.xaml
│   │   └── AddRepairControl.xaml.cs
│   ├── Windows
│   │   ├── MainWindow.xaml
│   │   └── MainWindow.xaml.cs
│   ├── SearchControl.xaml
│   └── SearchControl.xaml.cs
├── ViewModel
│   ├── AddRepairViewModel.cs
│   ├── MainViewModel.cs
```

```
|   └── SearchViewModel.cs
└── App.xaml
```

Commands

DelegateCommand.cs

DelegateCommand наследява класа ICommand. Това е проста реализация на интерфейса ICommand, който позволява да дефинираме логиката на командите с помощта на делегат за действие.

Методът Execute е част от интерфейса ICommand, който трябва да се изпълни при извикване на командата.

Методът CanExecute се използва за определяне дали командата може да се изпълни в текущото си състояние.

Converter

ReverseBooleanVisibilityConverter.cs

Класът ReverseBooleanVisibilityConverter е персонализиран конвертор, който наследява интерфейса IValueConverter за използване data binding на данни на WPF. Този преобразувател обръща типичната логика от bool to visibility, при който ture стойност трябва да скрие елемент на потребителския интерфейс, а false стойност трябва да го покаже. Използва се да се скрие таблицата на Repairs, когато не са намерени записи.

Database

DatabaseContext.cs

Класът DatabaseContext е основен компонент за управление на взаимодействията на базата данни в приложение за класовете Repair и Vehicle. Той разширява класа DbContext от Entity Framework.

OnConfiguring метод конфигурира връзката с базата данни. Определя основната директория и конструира пътя до файла на базата данни SQLite.

OnModelCreating - конфигурира моделите на обекти и техните взаимоотношения. Указва, че свойството Id на обектите Repair и Vehicle трябва да се генерира автоматично. Дефинира връзка "един към много" между обекти Repair и Vehicle. Зарежда първоначални данни за обектите Vehicle и Repair, за да попълни базата данни с примерни записи.

DatabaseService.cs

Класът DatabaseService предоставя клас, който взаимодействие с базата данни, улеснявайки CRUD операции и по-сложните заявки.

Generic методи:

- **GetAll<T>():** Извлича всички записи от определен тип T от базата данни.
- **Add<T>(T entity):** Добавя нов запис от тип T към базата данни.
- **Search<T>(List<Expression<Func<T, bool>>> stringFilters, Expression<Func<T, int>> yearExpr, int? searchYear, Func<IQueryable<T>, IIncludableQueryable<T, object>>? includesExpr =**

null): Извършва по-общо търсене с персонализирани филтри, съвпадение по години и незадължителни includes изрази за свързани обекти.

Специфични методи:

- **GetAllVehicles():** Извлича всички записи за превозни средства, включително свързаните с тях записи за ремонт.
- **GetAllRepairs():** Извлича всички записи за ремонт.
- **GetAllRepairsWithVehicles():** Извлича всички записи за ремонт, включително свързаните с тях записи за превозни средства.
- **AddRepair(Repair repair, Vehicle vehicle):** Добавя нов запис за ремонт, свързан с превозно средство. Ако превозното средство не съществува, то се добавя към базата данни; в противен случай се използва съществуващото превозно средство.

Model

Repair

Класът Repair представлява запис за ремонт. Той дефинира структурата на repair entity, включително неговите свойства, анотации на данни за валидиране и връзка с Vehicle.

Vehicle

Класът Vehicle в представлява обект превозно средство. Той дефинира структурата на vehicle entity, включително неговите свойства, анотации на данни за валидиране и връзка с Repair.

Observable

ObservableObject.cs

Класът ObservableObject е основен клас, предназначен да реализира известие за промяна на свойствата.

Theme

Fonts

В тази папка живеят custom шрифтове за приложението. Това включва Poppins-Bold, Poppins-Medium, Poppins-MediumItalic, Poppins-Regular.

View

Controls

SearchControl

SearchControl е персонализиран UserControl, предназначен да предостави интерфейс за търсене на записи за ремонт на превозни средства.

Оформлението на контролата се дефинира с помощта на grid, разпределен за различни части на интерфейса. В горната част TextBlock показва заглавието. Под заглавието се създава зона за търсене, състояща се от етикети и текстови полета за въвеждане на критерии за търсене като „Година (задължително)“, „Марка“, „Модел“ и „Описание на ремонта“.

Два бутона, "Търсене" и "Изчистване", са поставени до полетата за търсене, съответно за изпълнение на командата за търсене и изчистване на полетата за въвеждане. Хоризонтална линия разделя областта за търсене от секцията с резултати, където се показват резултатие от търсенето

Резултатите от търсенето се показват в DataGrid в рамките на StackPanel. DataGrid е оформен така, че да променя цветовете на редовете и включва колони за показване на подробности за ремонта, като RepairID, година на обслужване, марка, модел, регистрационен номер и описание на ремонта.

Във code-behind се включва метод NumberValidationTextBox, който валидира въвеждането в текстовото поле „Година“, като гарантира, че са въведени само числови стойности. Този метод използва регулярен израз за филтриране на нечислов вход.

AddUserControl

AddRepairControl е персонализиран предназначен да улесни добавянето на нови записи за ремонт. Контролата се показва само когато няма точно съвпадение от търсенето в SearchControl. От там се предават, като подсказки за добавяне, текста от полетата за търсене.

Оформлението на контролата е организирано с помощта на грид.

В горната част на контролата се показва съобщение, подканващо потребителя да добави нов Repair, ако не бъдат намерени точни съвпадения. Под това има форма за добавяне на Repair със следните полета:

- „Година на ремонт“ е задължително поле с цифрова проверка, за да се гарантира, че са въведени само цифри.
- Полетата „Марка“ и „Модел“.
- „Описание на ремонта“ позволява на потребителя да опише ремонта.
- Полетата „Регистрационен номер“ и „Година на производство“ с цифрово валидиране за годината.

В долната част има бутон с надпис „Добавяне на нов ремонт“. Този бутон е обвързан с AddRepairCommand.

Във code-behind AddRepairControl включва метод NumberValidationTextBox, който гарантира, че са позволени само числови стойности в текстовите полета „Година на ремонт“ и „Година направено“. Този метод използва регулярен израз за филтриране на нечислов вход.

Windows

MainWindow

MainWindow служи като основен интерфейс за приложението Vehicle Repairs. Той интегрира различни контролите за търсене и добавяне на Repair и превозното средсвто свързано с него.

Контекстът на данните на прозореца е зададен на MainViewModel, който служи като основен view model за приложението, управлявайки данните и командите за потребителския интерфейс. Освен това ресурса BooleanToVisibilityConverter е дефиниран за видимостта въз основа на булева стойност.

Съдържанието на MainWindow е обвито в ScrollViewer, което гарантира, че потребителите могат да превъртат през съдържанието, ако надхвърля видимата област. Вътре в ScrollViewer се използва грид за оформление на съдържанието.

В първия ред на грида е поставен SearchControl компонент, който е обвързан с SearchViewModel.

Във втория ред е поставен компонент AddRepairControl, който е обвързан с AddRepairViewModel. Този контрол предоставя интерфейс за добавяне на нови записи за ремонт, когато няма точно съвпадение. Видимостта на AddRepairControl се управлява динамично с помощта на свойството IsAddRepairVisible в MainViewModel, преобразувано чрез BooleanToVisibilityConverter. Това гарантира, че контролът е видим само когато е няма точно съвпадение.

ViewModel

MainViewModel.cs

Класът MainViewModel е основен компонент на приложението, функциониращ като основен view model, който организира взаимодействието между потребителския интерфейс и логиката на приложението. Този клас разширява ObservableObject, за да поддържа уведомления за промяна на свойствата, за динамичното актуализиране на потребителския интерфейс. Този view model обединява останалите два view model-а SearchViewModel и AddRepairViewModel. Тази връзка е нужна, за да могат да се инициализират пропъртитата за добавяне на Repair (като подсказка)

Ключови свойства и методи

Конструкторът инициализира свойствата SearchViewModel и AddRepairViewModel. Те се инициализират с this към самия MainViewModel, което позволява взаимодействие между view моделите.

Методи:

- **UpdateAddRepairProps(string serviceYear, string brand, string model, string repairDescription):** Актуализира свойствата (като подсказка) на AddRepairViewModel с предоставените стойности за година на обслужване, марка, модел и описание на ремонта.
- **Clear():** Извиква **HideAddRepairControl**, за да скрие контролата за добавяне на ремонт. Изчиства свойствата както в SearchViewModel и в AddRepairViewModel, за да нулира състоянието.
- **ClearAddRepairProps():** Изчиства свойствата в AddRepairViewModel.
- **ShowAddRepairControl():** Задава IsAddRepairVisible на true, което прави контролата за добавяне на ремонт видима.
- **HideAddRepairControl():** Задава IsAddRepairVisible на false, скривайки контролата за добавяне на ремонт.

SearchViewModel.cs

Класът SearchViewModel осигурява функционалността и обвързването на данни, необходими за търсене на записи за ремонт. Той разширява ObservableObject, за да поддържа известия за промяна на свойствата, които са от значение за динамичното актуализиране на потребителския интерфейс.

Полета и свойства:

_mainViewModel: Препратка към MainViewModel.

_repairs: Колекции за съхраняване на данни за превозни средства и ремонти.

_repairedYear, _brand, _model, _repairDescription: Полета за съхранение на критерии за търсене.

dbService: инстаниця на DatabaseService за взаимодействие с базата данни.

_isRepairsEmpty: Булева стойност, показваща дали резултатите от търсенето са празни. Това се използва за контролиране на видимостта на елементи на потребителския интерфейс.

Конструктор

Конструкторът инициализира SearchViewModel с препратка към MainViewModel и настройва колекцията Repairs. Той също така извиква LoadRepairs(), за да попълни първоначалния списък с данни.

Команди

SearchCommand: Изпълнява метода SearchRepairs.

ClearCommand: Изпълнява метода Clear.

Методи

LoadRepairs(): Зарежда всички записи за ремонт със свързаните с тях превозни средства от базата данни в колекцията Repairs.

SearchRepairs(): Създава динамична заявка за търсене въз основа на предоставените критерии. Той използва функции на Entity Framework за съвпадение на шаблони и включва свързани данни за превозни средства в резултатите. Ако не бъдат намерени записи, той актуализира основния модел на изглед, за да покаже контрола за добавяне на ремонт.

ClearProps(): Нулира всички свойства на критериите за търсене и презарежда списъка за поправки.

Clear(): Скрива контролата за добавяне на ремонт и изчиства всички свойства в основния view model

Свързване на данни и известия за промяна на свойства

SearchViewModel използва обвързване на данни, за да свърже UI елементи със своите свойства. Когато стойността на свойство се промени, съответният сетер предизвиква събитие за промяна на свойство, използвайки RaisePropertyChangedEvent. Това гарантира, че потребителският интерфейс се актуализира автоматично.

Интеграция с MainViewModel

SearchViewModel взаимодейства с MainViewModel. Когато не бъдат намерени резултати от търсенето, той актуализира свойствата в AddRepairViewModel, чрез функцията UpdateAddRepairProps в MainViewModel, за да подкани потребителя да добави нов repair. Също така контролира видимостта на контролата за добавяне на ремонт въз основа на резултатите от търсенето.

AddRepairViewModel.cs

Класът AddRepairViewModel е отговорен за управлението на данните и логиката, необходими за добавяне на нови записи за ремонт. Класа разширява ObservableObject, който позволява известяване за промяна на свойства, което е от значение за динамичното актуализиране на потребителския интерфейс.

Полета и свойства

_mainViewModel: Инстанция на MainViewModel.

_brand, _model, _repairDescription, _repairedYear, _registrationNumber, _yearMade: Полета за съхраняване на входните стойности за новия ремонт. Тези полета се синхронизират с полетата за търсене като подсказка за добавяне при неточен резултат.

dbService: Инстаниця DatabaseService, използва се за взаимодействие с базата данни.

Команди

AddRepairCommand: Връща инстанция на DelegateCommand, който изпълнява метода AddRepair, когато се задейства. Тази команда е свързана с бутон в потребителския интерфейс.

Методи

AddRepair(): Този метод подготвя добавянето на Repair и Vehicle, като използва стойностите от свойствата. След това използва dbService, за да изпрати данните към метода за добавяне в базата данни. След успешно добавяне на записите, той извиква метода **Clear()** за нулиране на полетата за въвеждане.

Clear(): Извиква метода **Clear** на **MainViewModel**, като гарантира, че състоянието на приложението се нулира по подходящ начин.

ClearProps(): Нулира всички полета за въвеждане до техните стойности по подразбиране, изчиствайки всички данни, въведени от потребителя.

AddRepairViewModel използва data binding, за да свърже своите свойства със съответните елементи на потребителския интерфейс.

AddRepairViewModel взаимодейства с MainViewModel, за да поддържа последователно състояние на приложението. Например, когато се добави нова поправка, методът Clear() се извиква от MainViewModel, за да нулира състоянието на приложението.

По сложни имплементации

По сложни типове

Expression: Това представлява LINQ изразно дърво, което може да бъде „parsed“ парснато и изпълнено от query providers като Entity Framework.

Func<in T, out TResult>: Капсулира метод, който има един параметър и връща стойност от типа, определен от параметъра TResult

Func<T, bool>: Това е делегат, който представлява функция, приемаща вход от тип T и връщаща булева стойност. В контекста на заявка T представлява тип обект (напр. Repair), а булевата върната стойност показва дали обектът отговаря на критериите, посочени от функцията.

Func<T, int>: Това е делегат, представляващ функция, която приема един параметър от тип T и връща цяло число. В контекста на LINQ заявка, T обикновено е тип обект (напр. Repair) и функцията указва как да се осъществи достъп до целочислено свойство на този обект.

IQueryable<T>: Представлява queryable колекция от T обекти. Това е типът, използван за LINQ заявки към база данни.

IIncludableQueryable<T, object>: Разширява **IQueryable<T>** и се използва за заявки, които включват свързани обекти. Вторият тип параметър (в този случай обект) представлява типа на включеното свързано entity.

Func<IQueryable<T>, IIncludableQueryable<T, object>>: Това е тип делегат, който представлява функция. Функцията приема **IQueryable<T>** като вход и връща **IIncludableQueryable<T, обект>**.

Функции и пример

По сложна е имплементацията на обща заявка за търсене:

```
public IEnumerable<T> Search<T>(

        List<Expression<Func<T, bool>>> stringFilters,
        Expression<Func<T, int>> yearExpr,
        int? searchYear,
        Func<IQueryable<T>, IIncludableQueryable<T, object>>? includesExpr = null
    ) where T : class
    {
        using (var context = new DatabaseContext())
        {
            var query = context.Set<T>().AsQueryable();

            if (includesExpr != null)
            {
                query = includesExpr(query);
            }

            foreach (var filter in stringFilters)
            {
                query = query.Where(filter);
            }

            if (searchYear.HasValue)
            {
                // This subquery fetches all years, then selects the closest one less than or equal to the search year
                var yearQuery = query.Select(yearExpr);
                var closestYear = yearQuery.Where(y => y <= searchYear.Value).OrderByDescending(y => y).FirstOrDefault();

                // Ensure that a valid year was found, otherwise return an empty collection
                if (closestYear == 0)
                {
                    return Enumerable.Empty<T>();
                }

                var parameter = yearExpr.Parameters[0];
                var equalsClosestYear = Expression.Equal(yearExpr.Body, Expression.Constant(closestYear, typeof(int)));
                var lambda = Expression.Lambda<Func<T, bool>>(equalsClosestYear, parameter);
                query = query.Where(lambda);
            }

            return query.ToList();
        }
    }
```

List<Expression<Func<T, bool>>> stringFilters: Този лист съдържа всички филтри, които ще се ползват в заявката. Пример:

```
if (!string.IsNullOrWhiteSpace(Brand))
{
   stringFilters.Add(r => EF.Functions.Like(r.Vehicle.Brand, $"%{Brand}%"));
}

if (!string.IsNullOrWhiteSpace(Model))
{
   stringFilters.Add(r => EF.Functions.Like(r.Vehicle.Model, $"%{Model}%"));
}

if (!string.IsNullOrWhiteSpace(RepairDescription))
{
   stringFilters.Add(r => EF.Functions.Like(r.Description, $"%{RepairDescription}%"));
}
```

Expression<Func<T, int>> yearExpr: Това ще се използва за филтриране по година. Пример:

```
Expression<Func<Repair, int>> yearExpr = r => r.YearOfService;
```

Func<IQueryable<T>, IIncludableQueryable<T, object>>? includesExpr: Това се използва за да окаже на Entity Framework, че искаме да включване свързаните обекти в резултата от заявката. Например в приложението Repair е свързан с Vehicle. В модела на Repair имаме Vehicle обект и Vehcile ID. Vehicle обекта по подразбиране остава празен при обикновена заявка. За да се инициализира Vehicle пропъртито трябва да използваме include. Това трябва да се направи ПРЕДИ останалите заявки (като първа заявка)! По подразбиране е null, защото не е задължително. Пример:

```
Func<IQueryable<Repair>, IIncludableQueryable<Repair, object>> include = query => query.Include(r => r.Vehicle);
```

Архив и код на приложението

Приложението е качено в github на този линк.

Заключение

Приложението за Vehicle Repairs е цялостна и напълно функционална система, предназначена да управлява ефективно записите за ремонт на превозни средства. Създадено с WPF и Entity Framework, приложението предоставя функции за търсене, добавяне и управление на записи за Repair, като използва разширено обвързване на данни и възможности за динамични заявки.

Използвана литература

https://stackoverflow.com/

https://stackoverflow.com/questions/17233651/wpf-data-binding-label-content

https://programmingistheway.wordpress.com/2017/02/17/only-numbers-in-a-wpf-textbox-with-regular-expressions/

https://learn.microsoft.com/en-us/ef/core/get-started/wpf

https://www.codeproject.com/Articles/873592/Tutorial-for-a-Basic-WPF-MVVM-Project-Using-Entity