Картина, която съдържа текст, Шрифт, лого, символ

Описанието е генерирано автоматично

**Курсов проект по ПС Computer Service**

Тема 1: Система за търсене и показване на годишни списъци

Изготвил: Стоян Стойков  
Специалност: КСИ  
Фак. номер: 121221087  
Група: 45

22.05.2024

Съдържание

[Анализ на заданието 3](#_Toc167296877)

[Компоненти на проекта 3](#_Toc167296878)

[ER Диаграма на класовете 3](#_Toc167296879)

[Файлова стриктура 3](#_Toc167296880)

[App.xaml 4](#_Toc167296881)

[Database folder 4](#_Toc167296882)

[DatabaseContext.cs 4](#_Toc167296883)

[DatabaseService.cs 5](#_Toc167296884)

[Helpers folder 6](#_Toc167296885)

[DelegateCommands.cs 6](#_Toc167296886)

[Model Folder 7](#_Toc167296887)

[Computer.cs 7](#_Toc167296888)

[Repair.cs 7](#_Toc167296889)

[View folder 8](#_Toc167296890)

[SearchControl View 8](#_Toc167296891)

[AddRepairWindow 8](#_Toc167296892)

[MainWindow 9](#_Toc167296893)

[ViewModels folder 9](#_Toc167296894)

[SearchViewModel 9](#_Toc167296895)

[AddRepairViewModel 10](#_Toc167296896)

# Анализ на заданието

Приложението е направено за сервиз за компютри, с което да следи направените ремонти. Базирано на WPF. Позволява на потребителите да търсят ремонти въз основа на специфични критерии и да добавят нови ремонти, ако не съществуват в базата данни. Приложението следва модела на проектиране MVVM (Model-View-ViewModel), за да поддържа чисто разделение между потребителския интерфейс и бизнес логиката.

# Компоненти на проекта

## ER Диаграма на класовете

## Файлова стриктура

ComputerServiceApplication/

│

├── Computer\_Serivce/

│ ├── Database/

│ │ ├── DatabaseContext.cs

│ │ ├── DatabaseService.cs

│ │

│ ├── Helpers/

│ │ ├── DelegateCommand.cs

│ │

│ ├── Model/

│ │ ├── Computer.cs

│ │ └── Repair.cs

│ │

│ ├── View/

│ │ ├── Controls/

│ │ │ ├── SearchControl.xaml

│ │ │ └── SearchControl.xaml.cs

│ │ ├── Window/

│ │ │ ├── AddRepairWindow.xaml

│ │ │ └── AddRepairWindow.xaml.cs

│ │ │ ├── MainWindow.xaml

│ │ │ └── MainWindow.xaml.cs

│ │

│ ├── ViewModel/

│ │ ├── AddRepairViewModel.cs

│ │ └── SearchViewModel.cs

│ │

│ └── App.xaml

│ └── App.xaml.cs

│

└── README.md

## Функционални изисквания

* **Търсене и филтриране:** Потребителят трябва да може да задава критерии за търсене (филтри) за полетата на обектите и да е възможно търсенето по конкретна година и връщане на резултати за тази година
* **Алтернативни резултати:** Ако няма резултати за годината, приложението трябва да връща резултати от последната налична година, за която има данни, отговарящи на останалите критерии.
* **Спазване на MVVM**: Архитектурата на системата трябва да спазва MVVM модела.

## Указания за бъдещия програмист

Функцията Search<T> е общ метод, който позволява гъвкави заявки за всеки тип обект в база данни. За да използвате тази функция в друг проект, трябва да следвате няколко стъпки, за да се уверите, че функцията е правилно интегрирана и може да работи с различни класове обекти. Преди да може да се използва функцията Search<T>, трябва Entity Framework Core (EF Core) да е инсталиран на проекта. EF Core ще обработва операции с бази данни и ще предоставя контекста, необходим за заявки за обекти.

Трябва да се създаде клас DbContext в новия проект. Този клас трябва да дефинира DbSet свойства за всеки тип обект, който искате да включите във вашата база данни. Класът DbContext служи като мост между вашия код и базата данни, позволявайки ви да извършвате CRUD (създаване, четене, актуализиране, изтриване) операции.

Създайте класове обекти, които представляват таблиците във вашата база данни. Всеки клас на обект трябва да има свойства, които се съпоставят с колоните в съответната таблица. Уверете се, че тези класове включват необходимите анотации (като [Key] за първични ключове и [ForeignKey] за външни ключове), за да дефинират връзките между обектите.

Функцията Search<T> може да се ползва, както е написана, без допълнителни изменения. Проектирана е да бъде обща, което означава, че може да работи с всеки тип обект. Функцията приема няколко параметъра, включително:

* Filters: списък с изрази, които определят условията, на които трябва да отговарят обектите.
* yearExpression: Израз, който указва свойството година на обекта.
* searchYear: Незадължителна година за филтриране на обекти въз основа на тяхното свойство година.
* includeExpression: незадължителна функция за включване на свързани обекти (например както е в този проект, връзката между Repairs и Computers).

За да използвате функцията Search<T> трябва да се дефинират филтри, например създайте списък с условия (филтри), на които обектите трябва да отговарят. Може да филтрирате обекти въз основа на тяхното име, тип или друго свойство. След това се дефинира израз за година, който указва кое свойство представлява годината във вашите обекти. Следват незадължителните параметри, като годината за търсене на обекти от тази година или по-рано. И накрая са включени свързани обекти, ако е необходимо. Функцията Search<T> се извиква, като се подадават филтрите, израза за годината и всички незадължителни параметри. Функцията ще върне списък с обекти, които отговарят на посочените условия.

## App.xaml

В проекта Computer\_Serivce всички стилове за цялото приложение са дефинирани във файла App.xaml. Този централизиран подход гарантира, че стиловете се прилагат в цялото приложение. Дефинира ресурсите, стиловете и настройките за цялото приложение.

<Application x:Class="Computer\_Serivce.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:local="clr-namespace:Computer\_Serivce"

StartupUri="View/Windows/MainWindow.xaml">

<Application.Resources>

<Style TargetType="Grid">

<Setter Property="Background" Value="WhiteSmoke"/>

</Style>

<Style x:Key="Content" TargetType="Grid">

<Setter Property="Background" Value="#FFFFFF"/>

<Setter Property="Width" Value="1030"/>

<Setter Property="Margin" Value="0 50 0 50"/>

</Style>

<Style x:Key="Header" TargetType="TextBlock">

<Setter Property="Padding" Value="10"/>

<Setter Property="FontSize" Value="24" />

</Style>

<Style x:Key="NameRow" TargetType="StackPanel">

<Setter Property="HorizontalAlignment" Value="Center"/>

</Style>

<Style TargetType="Label">

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="Padding" Value="50 0 45 0"/>

<Setter Property="FontSize" Value="16"/>

</Style>

<Style x:Key="InputRow" TargetType="StackPanel">

<Setter Property="HorizontalAlignment" Value="Center"/>

</Style>

<Style TargetType="TextBox">

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="Width" Value="100"/>

<Setter Property="Margin" Value="20 0 20 0"/>

<Setter Property="FontSize" Value="16"/>

</Style>

<Style x:Key="ButtonsRow" TargetType="StackPanel">

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="Margin" Value="0 20 0 10"/>

</Style>

<!-- Button Style -->

<Style x:Key="ButtonStyle" TargetType="Button">

<Setter Property="Background">

<Setter.Value>

<LinearGradientBrush StartPoint="0,0" EndPoint="1,1">

<GradientStop Color="#FF57A773" Offset="0.0"/>

<GradientStop Color="#FF29A19C" Offset="1.0"/>

</LinearGradientBrush>

</Setter.Value>

</Setter>

<Setter Property="Foreground" Value="White"/>

<Setter Property="Padding" Value="15 5 15 5"/>

<Setter Property="FontSize" Value="16"/>

<Setter Property="FontWeight" Value="Bold"/>

<Setter Property="BorderBrush" Value="#FF29A19C"/>

<Setter Property="BorderThickness" Value="2"/>

<Setter Property="Cursor" Value="Hand"/>

<Setter Property="HorizontalAlignment" Value="Center"/>

<Setter Property="VerticalAlignment" Value="Center"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Button">

<Border x:Name="border" Background="{TemplateBinding Background}" Padding="{TemplateBinding Padding}" Margin="10 0 10 0" BorderBrush="{TemplateBinding BorderBrush}" BorderThickness="{TemplateBinding BorderThickness}" CornerRadius="5">

<ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center"/>

</Border>

<ControlTemplate.Triggers>

<Trigger Property="IsMouseOver" Value="True">

<Setter TargetName="border" Property="Background">

<Setter.Value>

<LinearGradientBrush StartPoint="0,0" EndPoint="1,1">

<GradientStop Color="#FF29A19C" Offset="0.0"/>

<GradientStop Color="#FF57A773" Offset="1.0"/>

</LinearGradientBrush>

</Setter.Value>

</Setter>

</Trigger>

<Trigger Property="IsPressed" Value="True">

<Setter TargetName="border" Property="Background">

<Setter.Value>

<LinearGradientBrush StartPoint="0,0" EndPoint="1,1">

<GradientStop Color="#FF1C7878" Offset="0.0"/>

<GradientStop Color="#FF147363" Offset="1.0"/>

</LinearGradientBrush>

</Setter.Value>

</Setter>

</Trigger>

</ControlTemplate.Triggers>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

<Style x:Key="Datagrid" TargetType="DataGrid">

<Setter Property="Foreground" Value="#191308"/>

<Setter Property="Width" Value="Auto"/>

<Setter Property="Height" Value="300"/>

</Style>

<Style x:Key="DatagridHeader" TargetType="DataGridColumnHeader">

<Setter Property="Width" Value="Auto"/>

<Setter Property="FontSize" Value="16"/>

<Setter Property="FontWeight" Value="Bold"/>

<Setter Property="Padding" Value="5"/>

</Style>

<Style x:Key="DataGridCellStyle" TargetType="DataGridCell">

<Setter Property="Background" Value="#C0FDFB"/>

<Setter Property="Foreground" Value="#001B2E"/>

<Setter Property="Width" Value="Auto"/>

<Setter Property="FontSize" Value="14"/>

<Setter Property="Padding" Value="5" />

<Style.Triggers>

<Trigger Property="IsSelected" Value="True">

<Setter Property="Background" Value="#FFCCE8FF"/>

<Setter Property="Foreground" Value="#FF000000"/>

</Trigger>

</Style.Triggers>

</Style>

</Application.Resources>

</Application>

## Database folder

Папката Database съдържа класовете и конфигурациите, необходими за взаимодействие с базата данни. Това включва настройка на контекста на базата данни, дефиниране на начина, по който обектите се съхраняват и извличат, и предоставяне на методи за заявки и актуализиране на базата данни. Това се случва в два файла DatabaseContext.cs и DatabaseService.cs.

### DatabaseContext.cs

Класът DatabaseContext наследява от DbContext и отговаря за конфигурирането на връзката към базата данни и настройването на моделите на обекти. И накрая се добавят няколко обекта при първото създаване на таблиците.

public class DatabaseContext : DbContext

{

public DbSet<Repair> Repairs { get; set; }

public DbSet<Computer> Computers { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

string baseDir = AppDomain.CurrentDomain.BaseDirectory;

string databaseFile = "ComputerRepairs.db";

string projectRoot = Path.GetFullPath(Path.Combine(baseDir, @"..\..\..\.."));

string databaseDir = Path.Combine(projectRoot, "db");

if (!Directory.Exists(databaseDir))

{

Directory.CreateDirectory(databaseDir);

}

string databasePath = databaseDir + $"\\{databaseFile}";

optionsBuilder.UseSqlite($"Data Source={databasePath}")

.LogTo(Console.WriteLine);

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Repair>().Property(e => e.Id).ValueGeneratedOnAdd();

modelBuilder.Entity<Computer>().Property(e => e.Id).ValueGeneratedOnAdd();

base.OnModelCreating(modelBuilder);

// Configure one-to-many relationship

modelBuilder.Entity<Repair>()

.HasOne(r => r.Computer)

.WithMany(v => v.Repairs)

.HasForeignKey("ComputerId");

modelBuilder.Entity<Repair>().Property(e => e.Id).ValueGeneratedOnAdd();

modelBuilder.Entity<Computer>().Property(e => e.Id).ValueGeneratedOnAdd();

// Seed data for Computer

modelBuilder.Entity<Computer>().HasData(

new Computer { Id = 1, Brand = "Lenovo", Model = "Yoga X1", YearMade = "2018", SerialNumber = "LE2018y" },

// ...

);

// Seed data for Repair

modelBuilder.Entity<Repair>().HasData(

new Repair { Id = 1, ComputerId = 1, YearOfService = 2023, ServiceType = "Hardware Maintenance", Description = "Full system diagnostics and cleaning" },

// ...

);

}

### DatabaseService.cs

Класът DatabaseService предоставя методи за взаимодействие с базата данни. Той използва DatabaseContext за извършване на CRUD операции (Създаване, Четене, Актуализиране, Изтриване). Най-забележителният метод в този клас е общият метод за търсене Search<T>.

**Search<T> метод**

Методът Search<T> е общ метод, който позволява да търсите обекти от произволен тип (T) въз основа на набор от условия (филтри), конкретна година и незадължителни включвания за свързани обекти (include). В този проект се използва, за да покаже свързаните към Repairs компютри. Другите две функции за търсене не се използват, тъй като тази ги замества.

**Параметри:**

* List<Expression<Func<T, bool>>> filters: Списък от ламбда изрази, представящи условията, на които трябва да отговарят обектите.
* Expression<Func<T, int>> yearExpression: Ламбда израз за извличане на свойството година от обекта.
* int? searchYear: незадължителна година за търсене на обекти от тази година или по-рано.
* Func<IQueryable<T>, IIncludableQueryable<T, object>>? includeExpression: Незадължителен ламбда израз за включване на свързани обекти в заявката.

public IEnumerable<T> Search<T>(

List<Expression<Func<T, bool>>> filters,

Expression<Func<T, int>> yearExpression,

int? searchYear,

Func<IQueryable<T>, IIncludableQueryable<T, object>>? includesExression = null

) where T : class

{

using (var context = new DatabaseContext())

{

var query = context.Set<T>().AsQueryable();

// Apply includes if provided

if (includesExression != null)

{

query = includesExression(query);

}

// Apply filters to the query

foreach (var filter in filters)

{

query = query.Where(filter);

}

// Search for entities based on the provided year

if (searchYear.HasValue)

{

var yearQuery = query.Select(yearExpression);

var closestYear = yearQuery.Where(y => y <= searchYear.Value).OrderByDescending(y => y).FirstOrDefault();

if(closestYear == 0)

{

return Enumerable.Empty<T>();

}

var parametar = yearExpression.Parameters[0];

var equalsClosestYear = Expression.Equal(yearExpression.Body, Expression.Constant(closestYear, typeof(int)));

var lambda = Expression.Lambda<Func<T, bool>>(equalsClosestYear, parametar);

query = query.Where(lambda);

}

return query.ToList();

}

}

**Начин на работа:**

* Initialize Query: Започва със създаване на колекция с възможност за запитване от посочения тип обект от контекста на базата данни.
* Apply includes: Ако е предоставен includeExpression, той се прилага към заявката, за да включи свързани обекти.
* Apply filters: Преминава през списъка с филтри и прилага всеки филтър към заявката с помощта на клаузата Where.
* Search by year: Ако е предоставено searchYear, то филтрира обектите въз основа на указаното свойство за година. Той намира най-близката година, която е по-малка или равна на годината на търсене, и съответно филтрира резултатите.
* Връщане на резултати: Преобразува крайната заявка в списък и връща резултатите.

**AddRepair метод**

Методът AddRepair добавя нов Repair запис към базата данни, заедно със свързания компютърен запис, ако е необходимо.

Параметри:

* Repair repair: Обектът за поправка, който трябва да бъде добавен.
* Computer computer: Компютърният обект, свързан с ремонта.

public void AddRepair(Repair repair, Computer computer)

{

using (var context = new DatabaseContext())

{

var existingComputer = context.Computers.FirstOrDefault(c => c.SerialNumber == computer.SerialNumber);

if (existingComputer == null)

{

context.Computers.Add(computer);

context.SaveChanges();

}

else

{

computer = existingComputer;

}

repair.ComputerId = computer.Id;

context.Repairs.Add(repair);

context.SaveChanges();

}

}

**Начин на работа:**

* Проверка на съществуващ компютър: Проверява дали предоставеният компютър вече съществува в базата данни, като използва неговия сериен номер.
* Добавяне на компютър: Ако компютърът не съществува, той се добавя към базата данни.
* Set Computer ID: Свързва поправката с правилното ID на компютър.
* Добавяне на ремонт: Добавя ремонтния обект към базата данни.
* Запазване на промените: Записва промените в базата данни, за да запази новите записи.

**GetAllComputers метод**

Методът GetAllComputers извлича всички компютърни записи от базата данни.

* Извличане на данни: Изпълнява заявка за получаване на всички компютри от таблицата Компютри.
* Връщане на резултати: Връща списъка с компютърни обекти.

public List<Computer> GetAllComputers()

{

using (var context = new DatabaseContext())

{

return context.Computers.Include(c => c.Repairs).ToList();

}

}

**GetAllRepairsWithComputers метод:**

Методът GetAllRepairsWithComputers извлича всички Repair записи от базата данни, включително свързаните с тях компютърни записи.

public List<Repair> GetAllRepairsWIthComputers()

{

using (var context = new DatabaseContext())

{

return context.Repairs.Include(r => r.Computer).ToList();

}

}

**Начин на работи:**

* Извличане на данни с включвания: Изпълнява заявка за получаване на всички поправки, включително свързани компютърни данни, използвайки метода Включване.
* Връщане на резултати: Връща списъка с ремонтни обекти с техните свързани компютърни обекти.

## Helpers folder

### DelegateCommands.cs

Класът DelegateCommand е реализация на интерфейса ICommand. Използва се за обработка на командната логика във ViewModel, позволявайки на ViewModel да реагира на потребителските взаимодействия. Този клас е от съществено значение за обвързване на команди в XAML, което отделя потребителския интерфейс от бизнес логиката.

**Основни методи:**

1. Конструктор: Конструкторът DelegateCommand приема два параметъра:
   * execute: действие, което определя какво се случва, когато командата се изпълни.
   * canExecute: Функция, която връща булево значение, показващо дали командата може да бъде изпълнена. Този параметър не е задължителен.

public DelegateCommand(Action execute, Func<bool> canExecute = null)

{

\_execute = execute ?? throw new ArgumentNullException(nameof(execute));

\_canExecute = canExecute;

}

1. CanExecute: Този метод проверява дали командата може да бъде изпълнена. Ако е предоставена функцията canExecute, тя връща своя резултат. В противен случай връща true, което показва, че командата винаги може да се изпълни.

public bool CanExecute(object parameter)

{

return \_canExecute == null || \_canExecute();

}

1. Execute: Този метод изпълнява действието, което е подадено.

public void Execute(object parameter)

{

\_execute();

}

1. RaiseCanExecuteChanged: Този метод задейства събитието CanExecuteChanged, което уведомява WPF, че възможността за изпълнение на командата може да се е променила. Това е важно за активиране или деактивиране на бутони или други източници на команди в потребителския интерфейс.

internal void RaiseCanExecuteChanged()

{

throw new NotImplementedException();

}

1. CanExecuteChanged event: Това събитие е част от интерфейса на ICommand и се използва от WPF за проследяване на промени в състоянието на изпълнение на командата. DelegateCommand се връзва за събитието CommandManager. RequerySuggested за автоматично преоценяване на метода CanExecute при определени условия (напр. когато фокусът на потребителския интерфейс се промени).

public event EventHandler CanExecuteChanged

{

add { CommandManager.RequerySuggested += value; }

remove { CommandManager.RequerySuggested -= value; }

}

## Model Folder

### Computer.cs

Компютърният клас представлява компютърен обект с различни свойства, описващи неговите атрибути. Той също така поддържа връзка с класа Repair за проследяване на ремонтите, свързани с всеки компютър. Класът Computer има връзка "един към много" с класа Repair, което означава, че един компютър може да има няколко ремонта. Това е представено от ICollection<Repair> в класа Computer и свойството Computer в класа Repair.

Атрибути:

* Id: Първичният ключ за компютърния обект. Той се генерира автоматично от базата данни.
* Brand: Марката на компютъра (напр. Dell, HP, Apple). Това поле е задължително.
* Model: Моделът на компютъра (напр. XPS 13, MacBook Pro). Това поле е задължително.
* YearMade: Годината на производство на компютъра. Това поле не е задължително.
* SerialNumber: Серийният номер на компютъра, който го идентифицира уникално. Това поле е задължително.
* Repairs: Колекция от поправки, свързани с компютъра.

public class Computer

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

[Required]

public required string Brand { get; set; }

[Required]

public required string Model { get; set; }

public string? YearMade { get; set; }

[Required]

public required string SerialNumber { get; set; }

public virtual ICollection<Repair>? Repairs { get; set; }

}

### Repair.cs

Класът Repair представлява запис за ремонт на компютър. Той включва подробности за услугата за ремонт и поддържа препратка към свързания компютър.

Атрибути:

* Id: Първичният ключ за обекта за поправка. Той се генерира автоматично от базата данни.
* YearOfService: Годината на извършване на ремонтната услуга. Това поле е задължително.
* ServiceType: Типът извършена услуга (напр. ремонт на хардуер, актуализация на софтуера). Това поле е задължително.
* Description: Подробно описание на ремонтната услуга. Това поле не е задължително.
* ComputerId: Външният ключ, който свързва поправката с конкретен компютър.
* Компютър: Свойство за навигация към асоциирания компютърен обект.

public class Repair

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

[Required]

public required int YearOfService { get; set; }

[Required]

public required string ServiceType { get; set; }

public string? Description { get; set; }

public int ComputerId { get; set; }

public Computer Computer { get; set; }

public Repair(){}

}

## View folder

### SearchControl View

Изгледът SearchControl е потребителски контрол в проекта Computer\_Serivce, който предоставя потребителски интерфейс за търсене на записи за ремонт на компютри въз основа на специфични критерии като година на обслужване, марка и модел.

XAML файлът дефинира оформлението и външния вид на потребителския интерфейс SearchControl. Той включва различни елементи на потребителския интерфейс, като етикети, текстови полета, бутони и мрежа с данни.

Ключови раздели:

* Grid Layout: Организира елементите на потребителския интерфейс в редове и колони.
* TextBlocks: Показване на заглавия и текст.
* StackPanels: Групирайте свързани полета за въвеждане и бутони.
* TextBoxes: Позволява потребителско въвеждане за критерии за търсене.
* Buttons: Задействайте търсене и ясни действия.
* DataGrid: Показва резултатите от търсенето.

Изгледът SearchControl обвързва своите UI елементи със свойства и команди, дефинирани в SearchViewModel. Това обвързване позволява динамични актуализации и потребителски взаимодействия, без да се изисква директно манипулиране на UI елементи в задния код.

Ключови връзки:

* Text Boxes: Свързване към свойствата RepairedYear, Brand и Model в SearchViewModel.
* Buttons: Свързване към SearchRepairsCommand и ClearCommand в SearchViewModel.
* DataGrid: Свързва се с колекцията Repairs

Обработката на събития се извършва в code-behind-а за специфични случаи, като например валидиране на въведената година от потребителя. Събитието е NumberValidationTextBox - гарантира, че в текстовото поле за годината е разрешено само въвеждане на цифри.

<UserControl.DataContext>

<vm:SearchViewModel />

</UserControl.DataContext>

<Grid Style="{StaticResource Content}">

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="Auto" />

<RowDefinition Height="\*" />

</Grid.RowDefinitions>

<TextBlock Style="{StaticResource Header}" Text="Computer Service" Grid.Row="0" VerticalAlignment="Center" HorizontalAlignment="Center"/>

<StackPanel Style="{StaticResource NameRow}" Grid.Row="1" Orientation="Horizontal">

<Label Content=" Year " VerticalAlignment="Center"/>

<Label Content=" Brand" VerticalAlignment="Center" />

<Label Content=" Model" VerticalAlignment="Center" />

</StackPanel>

<StackPanel Style="{StaticResource InputRow}" Grid.Row="2" Orientation="Horizontal" Margin="10" HorizontalAlignment="Center">

<TextBox PreviewTextInput="NumberValidationTextBox" Text="{Binding RepairedYear, Mode=TwoWay}" />

<TextBox Text="{Binding Brand}" />

<TextBox Text="{Binding Model}" />

</StackPanel>

<StackPanel Style="{StaticResource ButtonsRow}" Grid.Row="3" Orientation="Horizontal">

<Button

Style="{StaticResource ButtonStyle}"

Content="Search"

Command="{Binding SearchRepairsCommand}"

Padding="5" />

<Button

Style="{StaticResource ButtonStyle}"

Content="Clear"

Command="{Binding ClearCommand}"

Padding="5" />

</StackPanel>

<TextBlock Style="{StaticResource Header}" Text="Results" Grid.Row="4" VerticalAlignment="Center" HorizontalAlignment="Center"/>

<DataGrid

Style="{StaticResource Datagrid}"

ColumnHeaderStyle="{StaticResource DatagridHeader}"

CellStyle="{StaticResource DataGridCellStyle}"

Grid.Row="5" AutoGenerateColumns="False"

ItemsSource="{Binding Repairs}"

Margin="30,10,30,100"

IsReadOnly="True"

>

<DataGrid.Columns>

<DataGridTextColumn Header="RepairID" Binding="{Binding Id}" />

<DataGridTextColumn Header="Year of service" Binding="{Binding YearOfService}" />

<DataGridTextColumn Header="Repair Descripion" Binding="{Binding Description}" />

<DataGridTextColumn Header="ComputerID" Binding="{Binding Computer.Id}" />

<DataGridTextColumn Header="Brand" Binding="{Binding Computer.Brand}" />

<DataGridTextColumn Header="Model" Binding="{Binding Computer.Model}" />

<DataGridTextColumn Header="Serial Number" Binding="{Binding Computer.SerialNumber}" />

</DataGrid.Columns>

</DataGrid>

</Grid>

### AddRepairWindow

AddRepairWindow е изскачащ прозорец в проекта, който позволява на потребителите да добавят нови записи за ремонт за компютър, ако по търсените критерии няма такъв. Този прозорец предоставя удобен за потребителя интерфейс за въвеждане на ремонт и след това за запис в базата данни.

XAML файлът дефинира оформлението и външния вид на AddRepairWindow. Той включва етикети, текстови полета за въвеждане и бутон за добавяне на поправката.

Изгледът AddRepairWindow свързва своите UI елементи със свойства, дефинирани в AddRepairViewModel. Това свърззване позволява динамични актуализации и потребителски взаимодействия, без да се изисква директно манипулиране на UI елементи в задния код. Обработката на събития се управлява чрез кода за специфични случаи, като валидиране на въведените от потребителя данни за година, да е възможно въвеждането само на цифри и обработка на щракването върху бутона за добавяне. Не е направено с команда, защото имаше проблем при затваряне на изкачващия прозорец.

Ключови връзки:

* Текстови полета: Свързване към свойствата SerialNumber, RepairedYear, Brand, Model и Description в AddRepairViewModel.
* Бутон: Свързва се с AddRepairCommand в AddRepairViewModel.

<Window.DataContext>

<vm:AddRepairViewModel />

</Window.DataContext>

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="Auto"/>

</Grid.RowDefinitions>

<StackPanel Grid.Row="0" Orientation="Vertical" HorizontalAlignment="Center">

<Label Content="Serial number" VerticalAlignment="Center"/>

<TextBox x:Name="SerialNumberBox" Text="{Binding SerialNumber, UpdateSourceTrigger=PropertyChanged}" Margin="10"/>

</StackPanel>

<StackPanel Grid.Row="1" Orientation="Vertical" HorizontalAlignment="Center">

<Label Content="Repaired Year" VerticalAlignment="Center"/>

<TextBox x:Name="YearTextBox" PreviewTextInput="NumberValidationTextBox" Text="{Binding RepairedYear, UpdateSourceTrigger=PropertyChanged}" Margin="10"/>

</StackPanel>

<StackPanel Grid.Row="2" Orientation="Vertical" HorizontalAlignment="Center">

<Label Content="Brand" VerticalAlignment="Center" />

<TextBox x:Name="BrandTextBox" Text="{Binding Brand, UpdateSourceTrigger=PropertyChanged}" Grid.Row="1" Margin="10"/>

</StackPanel>

<StackPanel Grid.Row="3" Orientation="Vertical" HorizontalAlignment="Center">

<Label Content="Model" VerticalAlignment="Center" />

<TextBox x:Name="ModelTextBox" Text="{Binding Model, UpdateSourceTrigger=PropertyChanged}" Grid.Row="2" Margin="10"/>

</StackPanel>

<StackPanel Grid.Row="4" Orientation="Vertical" HorizontalAlignment="Center">

<Label Content="Description" VerticalAlignment="Center" />

<TextBox x:Name="DescritionTextBox" Text="{Binding Description}" TextWrapping="WrapWithOverflow" Grid.Row="2" Margin="10" Width="200" Height="50" VerticalAlignment="Top"/>

</StackPanel>

<Button Style="{StaticResource ButtonStyle}" Content="Add Repair" Grid.Row="5" HorizontalAlignment="Center" Click="AddRepairButton\_Click"/>

</Grid>

Code-behind:

private void AddRepairButton\_Click(object sender, RoutedEventArgs e)

{

if (DataContext is AddRepairViewModel viewModel)

{

if (viewModel.AddRepairCommand.CanExecute(null))

{

viewModel.AddRepairCommand.Execute(null);

DialogResult = true;

Close();

}

}

}

Целта на този метод е да обработи събитието за щракване на бутона „Добавяне на поправка“, като гарантира, че новият запис за поправка е добавен чрез командата на ViewModel и след това затваря прозореца, ако операцията е успешна.

Първо проверяваме дали DataContext на прозореца е от тип AddRepairViewModel. DataContext, дали е източникът на данни е обвързванията в изгледа. След това проверяваме дали AddRepairCommand може да бъде изпълнена. Методът CanExecute(null) се извиква, за да се определи дали командата е активирана в момента (т.е. дали отговаря на критериите за изпълнение). И ако командата може да се изпълни, изпълняваме метода Execute на AddRepairCommand. Това задейства логиката в метода AddRepair на AddRepairViewModel, който управлява процеса на добавяне на нова поправка. Задаваме диалоговия резултат на true, което показва, че операцията е била успешна и затваряме AddRepairWindow, връщайки контрола на извикващия прозорец или диалог.

### MainWindow

Това е основният прозорец, който обединява SearchControl-а и изкачващия прозорец за добавяне на поправка.

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<components:SearchControl Grid.Row="0" />

</Grid>

## ViewModels folder

### SearchViewModel

SearchViewModel е отговорен за управлението на логиката за търсене и показване на компютърни ремонти. Той обработва въведените от потребителя критерии за търсене, извършва търсения и управлява резултатите.

**Properties:**

* Computers: ObservableCollection<Computer>, която съдържа списъка с компютри.
* Repairs: ObservableCollection<Repair>, която съдържа резултатите от търсенето за поправки.
* RepairedYear, Brand, Model: низове, които съдържат въведеното от потребителя за критерии за търсене.

public ObservableCollection<Computer> Computers

{

get => \_computers;

set

{

\_computers = value;

OnPropertyChanged();

}

}

public ObservableCollection<Repair> Repairs

{

get => \_repairs;

set

{

\_repairs = value;

OnPropertyChanged();

}

}

public string RepairedYear

{

get => \_repairedYear;

set

{

\_repairedYear = value;

OnPropertyChanged();

}

}

public string Brand

{

get => \_brand;

set

{

\_brand = value;

OnPropertyChanged();

}

}

public string Model

{

get => \_model;

set

{

\_model = value;

OnPropertyChanged();

}

}

**Commands:**

* SearchRepairsCommand: Команда за изпълнение на търсенето въз основа на въвеждане от потребителя.

public ICommand SearchRepairsCommand { get; private set; }

* ClearCommand: Команда за изчистване на критериите за търсене и резултатите.

public ICommand ClearCommand { get; private set; }

**Ключови методи:**

* SearchViewModel (Конструктор): Инициализира колекциите Компютри и Ремонти. Настройва SearchRepairsCommand и ClearCommand със съответните методи. Извиква LoadComputers и LoadRepairs за извличане на първоначални данни от базата данни.

public SearchViewModel()

{

Computers = new ObservableCollection<Computer>();

Repairs = new ObservableCollection<Repair>();

SearchRepairsCommand = new DelegateCommand(SearchRepairs);

ClearCommand = new DelegateCommand(ClearFields);

LoadComputers();

LoadRepairs();

}

* LoadComputers: Зарежда всички компютри от базата данни с помощта на dbService.GetAllComputers() и попълва колекцията Computers.

private void LoadComputers()

{

try

{

var records = dbService.GetAllComputers();

foreach (var record in records)

{

Computers.Add(record);

}

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred: {ex.Message}");

}

}

* LoadRepairs: Зарежда всички поправки от базата данни със свързаните с тях компютри, с помощта на dbService.GetAllRepairsWIthComputers() и добавя извлечените записи в колекцията Repairs.

private void LoadRepairs()

{

try

{

var records = dbService.GetAllRepairsWIthComputers();

foreach (var record in records)

{

Repairs.Add(record);

}

}

catch (Exception ex)

{

Console.WriteLine($"An error occurred: {ex.Message}");

}

}

* SearchRepairs: Конструира филтърни изрази на базата на входни данни за марка и модел. По избор включва филтър за години, ако е посочена - RepairedYear. Използва метода за търсене от dbService за извличане на съответстващи записи. Ако не бъдат намерени резултати, подканва потребителя да добави нова поправка чрез диалогов прозорец (AddRepairWindow). Актуализира колекцията Repairs с резултатите от търсенето.

private void SearchRepairs()

{

// Old search

// Repairs = new ObservableCollection<Repair>(dbService.SearchRepairs(\_repairedYear, \_brand, \_model));

// OnPropertyChanged(nameof(Repairs));

var filters = new List<Expression<Func<Repair, bool>>>();

int? searchYear = string.IsNullOrWhiteSpace(RepairedYear) ? null : int.Parse(RepairedYear);

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

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

if (!string.IsNullOrWhiteSpace(Brand))

{

filters.Add(r => EF.Functions.Like(r.Computer.Brand, $"%{Brand}%"));

}

if (!string.IsNullOrWhiteSpace(Model))

{

filters.Add(r => EF.Functions.Like(r.Computer.Model, $"%{Model}%"));

}

var results = dbService.Search<Repair>(filters, yearExpression, searchYear, include);

if (!results.Any())

{

Application.Current.Dispatcher.Invoke(() =>

{

var addRepairWindow = new AddRepairWindow();

if (addRepairWindow.ShowDialog() == true)

{

var viewModel = addRepairWindow.DataContext as AddRepairViewModel;

if (viewModel != null)

{

var newRepair = new Repair

{

YearOfService = int.Parse(viewModel.RepairedYear),

Computer = new Computer

{

SerialNumber = viewModel.SerialNumber,

Brand = viewModel.Brand,

Model = viewModel.Model,

},

ServiceType = ""

};

LoadRepairs();

}

}

});

}

else

{

Repairs = new ObservableCollection<Repair>(results);

OnPropertyChanged(nameof(Repairs));

}

}

Този ред: ***Func<IQueryable<Repair>, IIncludableQueryable<Repair, object>> include = query => query.Include(r => r.Computer);*** е LINQ израз, който указва как да се включат свързани обекти в резултатите от заявката. Включва свързани обекти в резултата от заявката. В този случай той включва компютърния обект, свързан с всеки ремонтен обект. Това е полезно в сценарии, при които искаме да извлечем данни от свързани таблици в една заявка, като по този начин намаляваме броя на извикванията на базата данни и подобряваме производителността.

Компоненти:

* Func<IQueryable<Repair>, IIncludableQueryable<Repair, object>>:
  + Това дефинира делегат, който приема **IQueryable<Repair>** като вход и връща **IIncludableQueryable<Repair, object>**.
  + **IQueryable<Repair>**: Представлява подлежаща на запитване колекция от обекти Repair.
  + **IIncludableQueryable<Repair, object>**: Разширява IQueryable <Repair>, за да включва свързани обекти (Компютър в този случай).
* ClearFields: Изчиства критериите за търсене и презарежда всички поправки.

private void ClearFields()

{

RepairedYear = string.Empty;

Brand = string.Empty;

Model = string.Empty;

Repairs.Clear();

LoadRepairs();

}

### AddRepairViewModel

AddRepairViewModel е отговорен за управлението на логиката за добавяне на нов ремонт. Той обработва въведените от потребителя данни за новия ремонт и записва новия ремонт в базата данни.

**Properties:**

* Brand, Model, Description, RepairedYear, SerialNumber, YearMade: Низове, които съдържат въведените от потребителя данни за новия ремонт.

public string Brand

{

get => \_brand;

set

{

\_brand = value;

OnPropertyChanged(nameof(Brand));

}

}

public string Model

{

get => \_model;

set

{

\_model = value;

OnPropertyChanged(nameof(Model));

}

}

public string Description

{

get => \_description;

set

{

\_description = value;

OnPropertyChanged(nameof(Description));

}

}

public string RepairedYear

{

get => \_repairedYear;

set

{

\_repairedYear = value;

OnPropertyChanged(nameof(RepairedYear));

}

}

public string SerialNumber

{

get => \_serialNumber;

set

{

\_serialNumber = value;

OnPropertyChanged(nameof(SerialNumber));

}

}

public string YearMade

{

get => \_yearMade;

set

{

\_yearMade = value;

OnPropertyChanged(nameof(YearMade));

}

}

**Команди:**

* AddRepairCommand: Команда за изпълнение на логиката за добавяне на нова поправка. Инициира процеса на добавяне на нов запис за ремонт към базата данни. Свързан с бутон в потребителския интерфейс, той изпълнява метода AddRepair, когато бъде извикан.

public ICommand AddRepairCommand;

**Ключови методи:**

* AddRepairViewModel (Констрултор): Инициализира AddRepairCommand с метода AddRepair. Така ViewModel-а е готов да обработва потребителски взаимодействия и да добавя записи за ремонт.

public AddRepairViewModel()

{

AddRepairCommand = new DelegateCommand(AddRepair);

}

* AddRepair: Създава нов Repair обект и задава свойствата му въз основа на въведените от потребителя. Създава нов компютърен обект и задава свойствата му въз основа на въведените от потребителя. Използваме DatabaseService за добавяне на нови ремонтни и компютърни записи към базата данни. И накрая се извиква CloseAction за затваряне на прозореца, ако добавянето е успешно(В момента не работи, прозореца не се затваря, използва Click във View-то).

private void AddRepair()

{

Repair repair = new Repair

{

ServiceType = "",

Description = Description,

YearOfService = int.Parse(RepairedYear),

};

Computer computer = new Computer

{

Brand = Brand,

Model = Model,

SerialNumber = SerialNumber

};

dbService.AddRepair(repair, computer);

// CloseAction?.Invoke();

}

* ClearProps: Задава всички свойства (RepairedYear, Brand, Model, Description, SerialNumber, YearMade) на празни низове. Така полетата на формуляра се нулират, готови за ново въвеждане.

public void ClearProps()

{

RepairedYear = string.Empty;

Brand = string.Empty;

Model = string.Empty;

Description = string.Empty;

SerialNumber = string.Empty;

YearMade = string.Empty;

}

* OnPropertyChanged: Уведомява потребителския интерфейс за промени в свойствата, за да осигури актуализации на данните. Извиква събитието PropertyChanged с името на модифицираното свойство, което задейства актуализации на потребителския интерфейс.

public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged([CallerMemberName] string propertyName = null)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}