ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ

ФАКУЛТЕТ КОМПЮТЪРНИ СИСТЕМИ И ТЕХНОЛОГИИ
СПЕЦИАЛНОСТ КОМПЮТЪРНО И СОФТУЕРНО ИНЖЕНЕРСТВО

Студент: Валери Ивайлов Райков

Фак. №: 121222139

Група: 42Б

Курсов проект по дисциплината:

ПРОГРАМНИ СРЕДИ

Тема:

Приложение на .NET MAUI следващо MVVM архитектура на проектиране с връзка към база данни(SQLite3)

С ъ д ъ р ж а н и е

1. Увод

Този документ съдържа документацията на разработения софтуерен проект, обхващайки анализ на съществуващите решения, проектиране, реализация и ръководство за правилна употреба на приложението. Освен това се разглеждат и възможни подобрения, които биха допринесли за допълнителна оптимизация на проекта.

2. Цел

Целта на проекта е да демонстрира практическото приложение на платформата .NET MAUI с програмиране на C#, както и да задълбочи разбирането на трислойния архитектурен модел MVVM (Model - View - ViewModel). Проектът включва и интеграция с база данни, която съхранява информация, добавена от потребителите, осигурявайки ефективно управление на данните и подобрена потребителска функционалност.

3. Необходимости


За успешната реализация на проекта е необходимо да бъдат инсталирани Visual Studio Community Edition 2022, .NET Framework и .NET MAUI. Тези технологии осигуряват средата за разработка, изпълнение и тестване на приложението, като позволяват ефективно изграждане на кросплатформени мобилни и десктоп решения. Освен тях, в последствие, когато преминем към разработката на конкретния проект, ще добавим още пакети (NuGet packages), които ще опиша в съответния раздел за разработка.

4. Анализ на съществуващи разработки

.NET MAUI (Multi-platform App UI) е модерна кросплатформена технология, разработена от Microsoft, която позволява създаването на приложения за Android, iOS, Windows и macOS с един споделен код. Тя е наследник на Xamarin.Forms и предоставя значителни подобрения в производителността, структурата на кода и интеграцията с платформени API. Като език за програмиране се използва C#. Технологията е основен конкурент в кросплатформената разработка на софтуер, която се конкурира с Flutter, React Native, Swift и т.н.

Предимства на .NET MAUI

- Единна кодова база – Позволява споделяне на по-голямата част от кода между различните платформи, което намалява времето и разходите за разработка.
- Подобрена производителност – MAUI предоставя директен достъп до платформени API и използва по-ефективни рендеринг механизми в сравнение с Xamarin.Forms.
- Интеграция с .NET екосистемата – Поддържа Entity Framework Core, Dependency Injection, Blazor Hybrid и други .NET технологии.
- MVVM архитектура – Вградена поддръжка на Model-View-ViewModel (MVVM) и Model-View-Update (MVU), което улеснява управлението на състоянието и разделянето на логиката от изгледа.
- Hot Reload – Позволява промени в кода да се отразяват в реално време без необходимост от повторно стартиране на приложението.

Недостатъци и предизвикателства

- Относително нова технология – MAUI все още се развива, като някои функционалности не са напълно стабилни или изискват допълнителна конфигурация.
- Ограничена документация и ресурси – В сравнение с Flutter и React Native, общността на .NET MAUI все още е по-малка, а поддръжката и примерите са по-ограничени.
- Размер на приложенията – Приложенията, базирани на .NET MAUI, могат да бъдат по-големи поради вградените зависимости на .NET.

5. Проектиране

Кой ще използва продукта

Разработеното приложение е създадено с тестови цели и може да бъде използвано от всеки потребител. То симулира работата на мониторингова система за управление на ученици, предоставяйки възможност за добавяне, редактиране и изтриване на данни в системата. Основната цел на приложението е да демонстрира работата с бази данни и архитектурния модел MVVM (Mode l- View - ViewModel), като същевременно предоставя удобен и интуитивен интерфейс за взаимодействие.

Какви данни ще се използват

В системата ще бъдат управлявани следните типове данни:

Данни за учениците – Всеки ученик в системата има следните задължителни атрибути:

- **Собствено име (First Name)** – Текстово поле (тип varchar (string в C#)), ограничено до 50 символа.
- **Фамилно име (Family Name)** – Текстово поле (тип varchar (string в C#)), ограничено до 50 символа.
- **Имейл (Email)** – Поле от тип varchar (string в C#), което трябва да съдържа валиден имейл адрес.

Уникален идентификатор (ID) – Генериран автоматично и използван за идентификация на ученика в базата данни.

Валидация на данните – За да може ученик да бъде успешно създаден или редактиран, полетата трябва да отговарят на предварително зададени правила за валидност. Това гарантира, че въведените данни са коректни и обработката им няма да доведе до грешки в системата. В проекта това е постигнато чрез употреба на регулярни изрази и шаблони за имена и валиден имейл.

Архитектура и програмна организация

Приложението е разработено, следвайки обектно-ориентирания програмен модел (ООР) на C# и прилагайки MVVM архитектурата. Основните програмни компоненти включват:

- **Разделяне на логиката** – Кодът е структуриран в отделни класове и модули, отговарящи за различни аспекти на приложението. Това спомага за по-лесна поддръжка и разширяемост.

- **Използване на Design Patterns** – За подобряване на организацията и гъвкавостта са имплементирани утвърдени шаблони на проектиране.

Допълнителни модули, които са включени:

- **Custom Exceptions** – Обработка на специфични грешки чрез персонализирани изключения.
- **Services** – Модул за комуникация с базата данни и управлението на данните.

Как ще бъдат достъпени функционалностите от потребителя?

Приложението предлага интуитивен и удобен интерфейс, състоящ се от:

- **Полета за въвеждане на данни** – Потребителите могат лесно да въвеждат и редактират информация чрез текстови полета.
- **Бутони за навигация и управление** – Използват се за добавяне, промяна и изтриване на записи в системата.
- **Flyout меню** – Осигурява бърз достъп до различните секции на приложението.
- **Динамични съобщения и нотификации** – При всяко действие (успешно или неуспешно) се показва детайлно описание, включително инструкции за следващи стъпки и информация за възникнали грешки.

Какво целим с предложените подобрения/разширения?

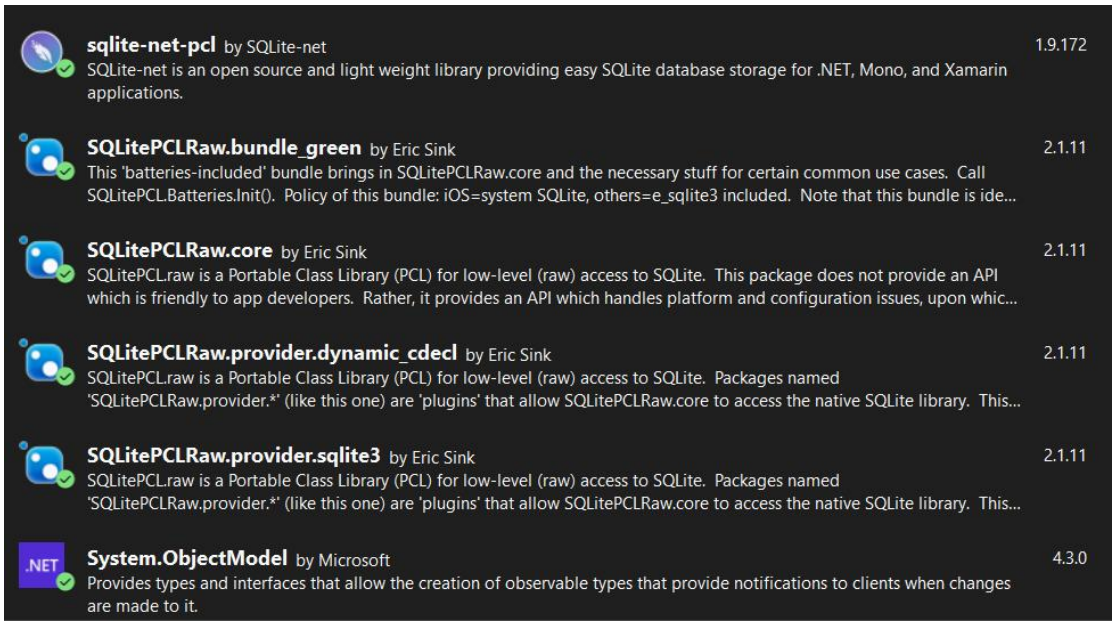
За да се увеличи функционалността и сложността на проекта, планираме добавянето на нови модули, включително:

- **Таблица за оценки** – Разширяване на базата данни с възможност за въвеждане на оценки по различни предмети. Това ще даде възможност за създаване на персонализиран профил на всеки ученик, подобно на академични системи като УИСС на ТУ-София.
- **Подобрена роля на администратора** – Добавяне на различни нива на достъп, така че различни потребители да имат различни разрешения за редактиране и управление на системата.

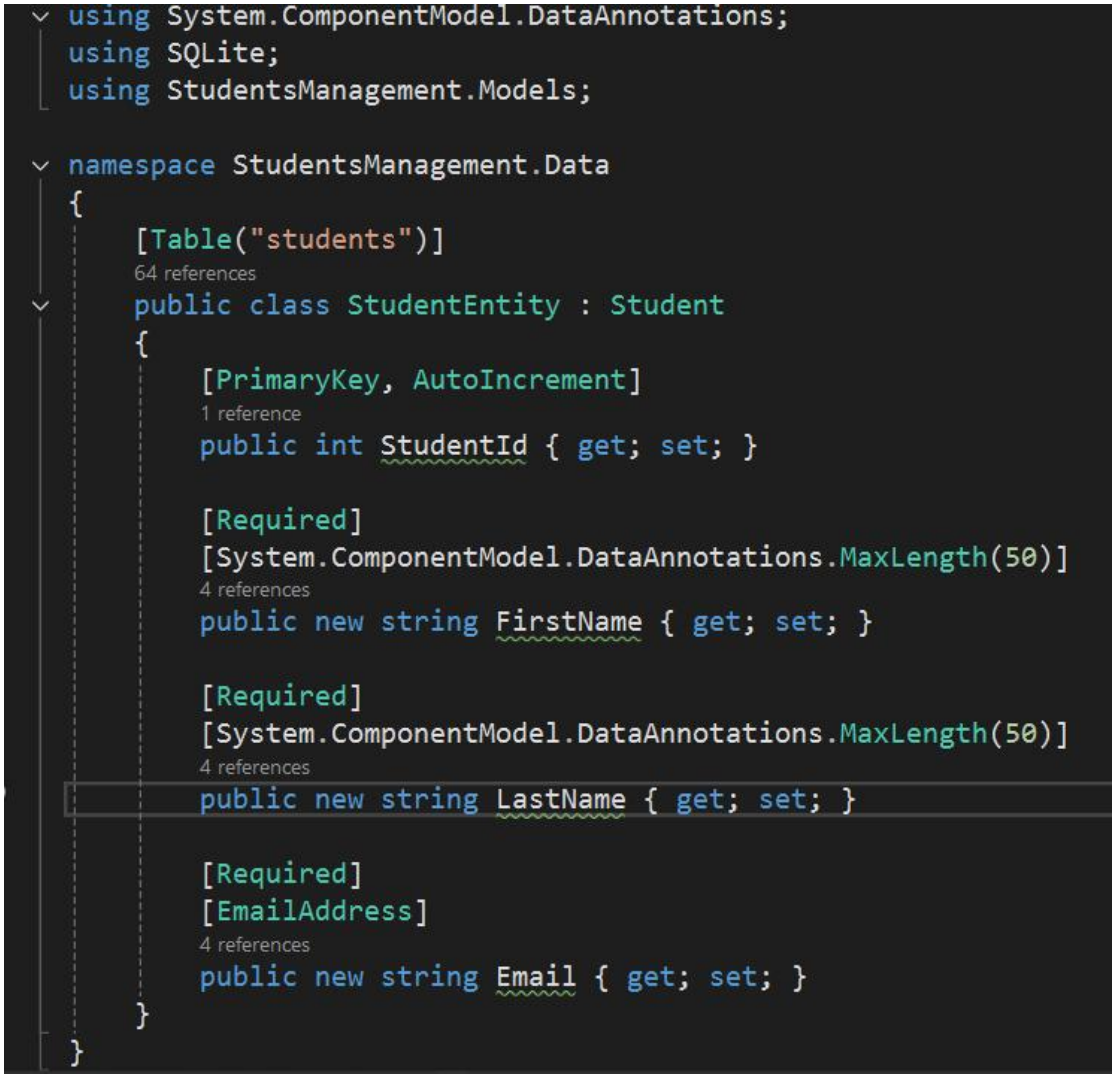
6. Реализация

В този раздел ще дам детайлно описание за стъпките на реализацията на проекта:

1. Инсталираме необходимите NuGet пакети, които са необходими за работа със SQLite база данни. Това включва SQLite.Net.Core-PCL и SQLite.Net.Async-PCL и т.н., осигуряващи интерфейси за синхронна и асинхронна работа с базата. Тези пакети ни позволяват да създаваме, четем, обновяваме и изтриваме данни (CRUD операции) ефективно в приложението.



2. Създаваме класа Student.cs, който дефинира основните полета, участващи в таблицата на базата данни. За да запазим структурата на този клас чиста и разделена от специфичните реализации, създаваме допълнителен клас StudentEntity.cs, който го наследява. Използваме библиотеките System.ComponentModel.DataAnnotations и SQLite, за да зададем необходимите ограничения и анотации за базата данни. Това ни позволява да дефинираме уникални идентификатори, дължини на полета и допълнителни валидиращи правила, които гарантират коректното съхраняване и управление на данните.



```
using System.ComponentModel.DataAnnotations;
using SQLite;
using StudentsManagement.Models;

namespace StudentsManagement.Data
{
    [Table("students")]
    64 references
    public class StudentEntity : Student
    {
        [PrimaryKey, AutoIncrement]
        1 reference
        public int StudentId { get; set; }

        [Required]
        [System.ComponentModel.DataAnnotations.MaxLength(50)]
        4 references
        public new string FirstName { get; set; }

        [Required]
        [System.ComponentModel.DataAnnotations.MaxLength(50)]
        4 references
        public new string LastName { get; set; }

        [Required]
        [EmailAddress]
        4 references
        public new string Email { get; set; }
    }
}
```

3. Създаваме интерфейса IStudentService.cs в директорията Services, който дефинира методите, необходими за управлението на данните в базата. Имплементираме интерфейса в класа StudentService.cs, който осъществява връзката с базата данни и изпълнява CRUD операциите (създаване, четене, актуализиране и изтриване на записи). Този клас отговаря за обработката на заявките към базата и гарантира ефективното управление на ученическата информация в приложението.

```
2 references
public class StudentService : IStudentService
{
    private readonly SQLiteAsyncConnection _dbConnection;

    0 references
    public StudentService()
    {
        _dbConnection = SetupDb().Result;
    }

    1 reference
    private async Task<SQLiteAsyncConnection> SetupDb()
    {
        try
        {
            string dbPath = Path.Combine(
                Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData), "students.db3"
            );

            var connection = new SQLiteAsyncConnection(dbPath);
            await connection.CreateTableAsync<StudentEntity>().ConfigureAwait(false);

            return connection;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Database initialization failed: {ex.Message}");
            throw new DatabaseException();
        }
    }
```

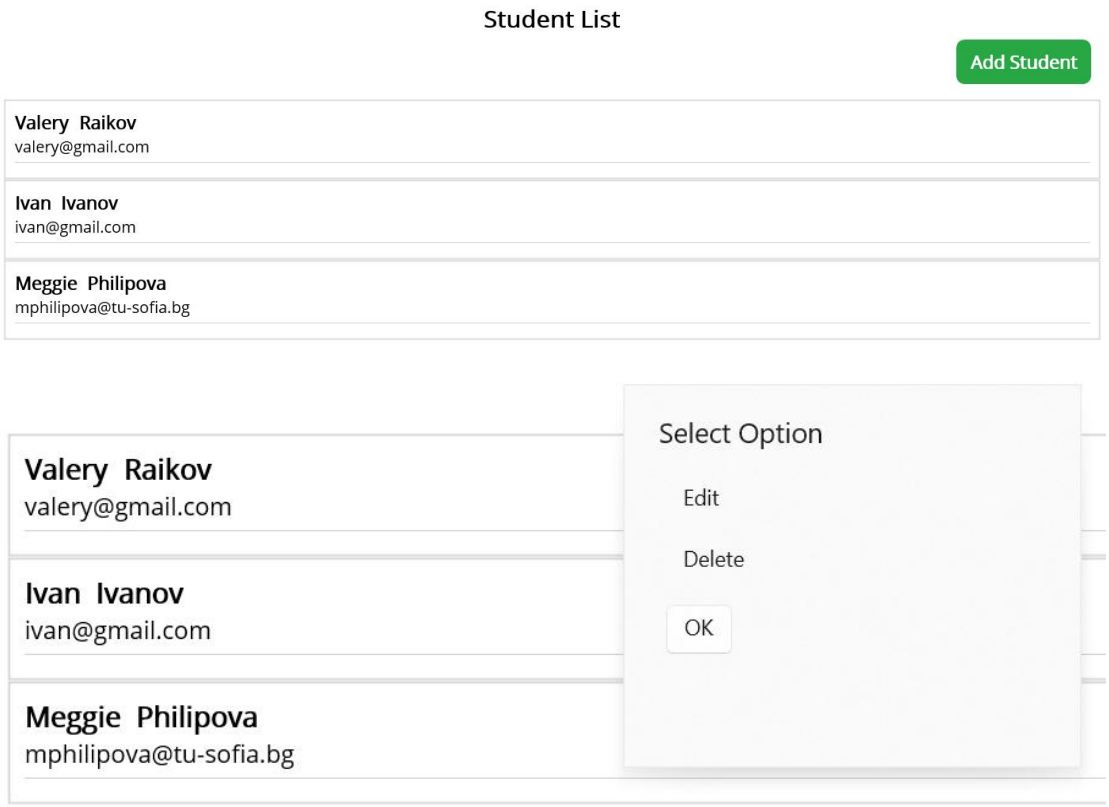
Други ключови методи в този файл са тези, свързани с CRUD операциите върху таблицата с учениците: GetStudentList, AddStudent, DeleteStudent и UpdateStudent. Във всеки от тях е внедрена обработка на грешки чрез try-catch, като се използват персонализирани (custom) изключения, създадени специално за приложението. Това позволява по-прецизно управление на грешките и предоставяне на ясна обратна връзка към потребителя.

Пример:

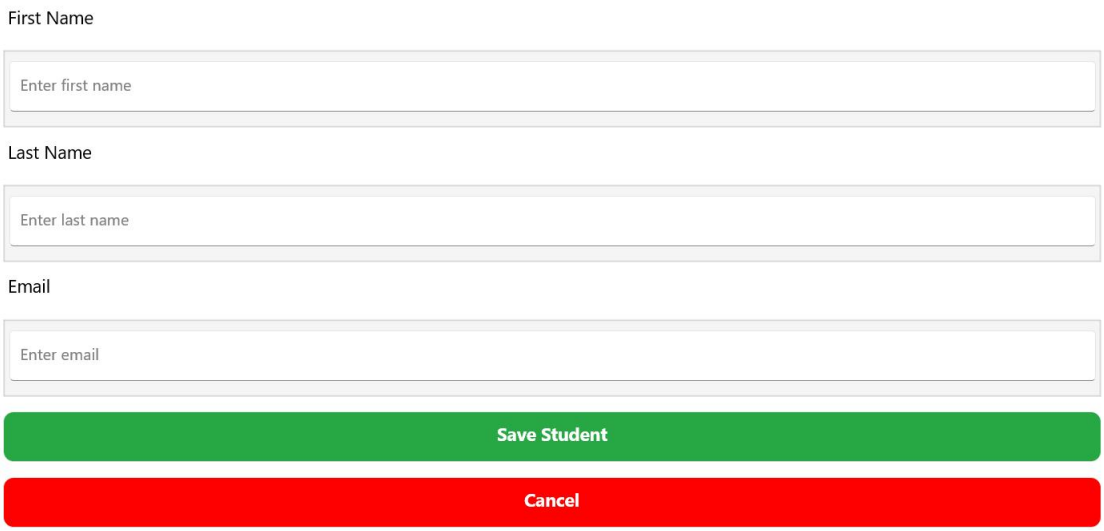
```
7 references
public class StudentException : Exception
{
    0 references
    public StudentException() { }

    0 references
    public StudentException(string message) : base(message) { }

    4 references
    public StudentException(string message, Exception innerException)
        : base(message, innerException) { }
}
```

4. Преминаваме към създаването на View слоя на проекта. Създаваме 3 Content Pages, които представляват потребителския интерфейс на приложението:

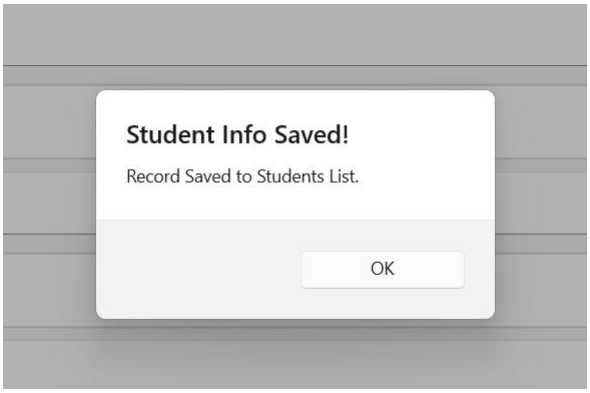
- **StudentListView.xaml** – Отговаря за визуализирането на списъка с всички ученици, съхранени в базата данни. При избор на конкретен ученик потребителят има възможност да редактира неговите данни или да го изтрие от списъка и базата. За добавяне на нов ученик е предвиден специален бутон, който пренасочва потребителя към страницата за въвеждане на нови записи. Това осигурява интуитивно и лесно управление на информацията в системата.



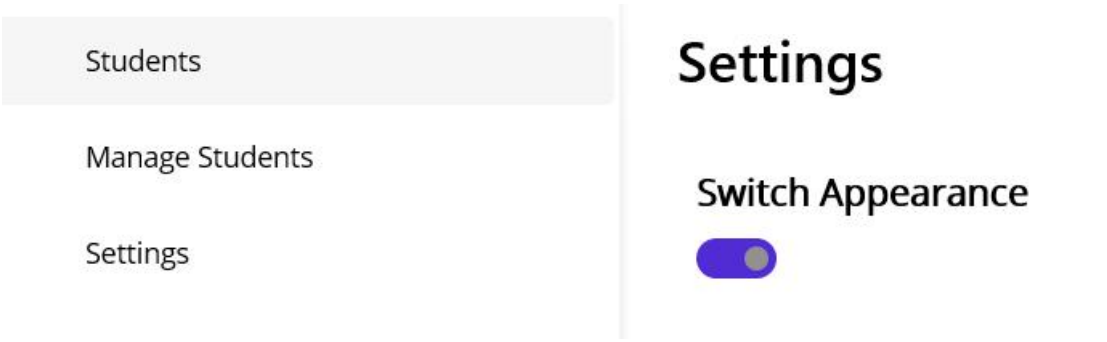
- **AddUpdateStudentView.xaml -** Отваря се нова страница, предназначена за добавяне на ученици в списъка. Тя представлява регистрационна форма с полета за собствено име, фамилно име и имейл, които трябва да бъдат попълнени коректно според зададените правила за валидация. Потребителят разполага с два бутона за действие:
- **Save Student** – Запазва въведените данни в базата.
- **Cancel** – Отменя операцията и връща потребителя обратно към списъка с ученици.



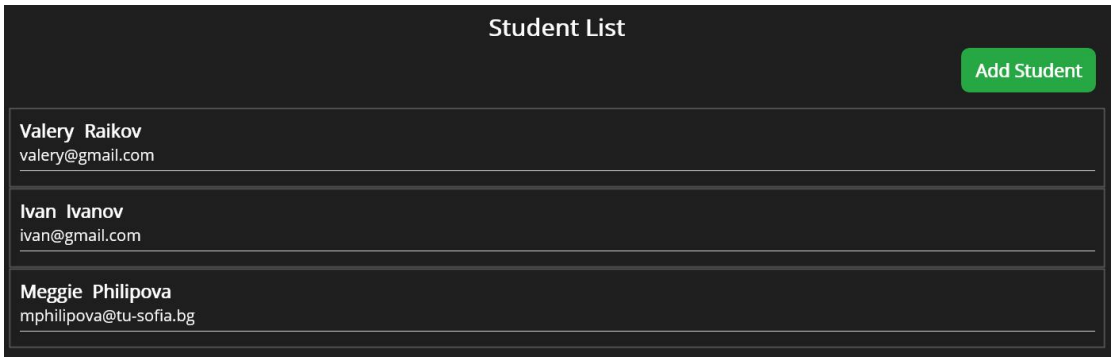
След като въведем правилни имена и имейл и запазим новия запис получаваме съобщение за успех и биваме автоматично прехвърлени към списъка с учениците:

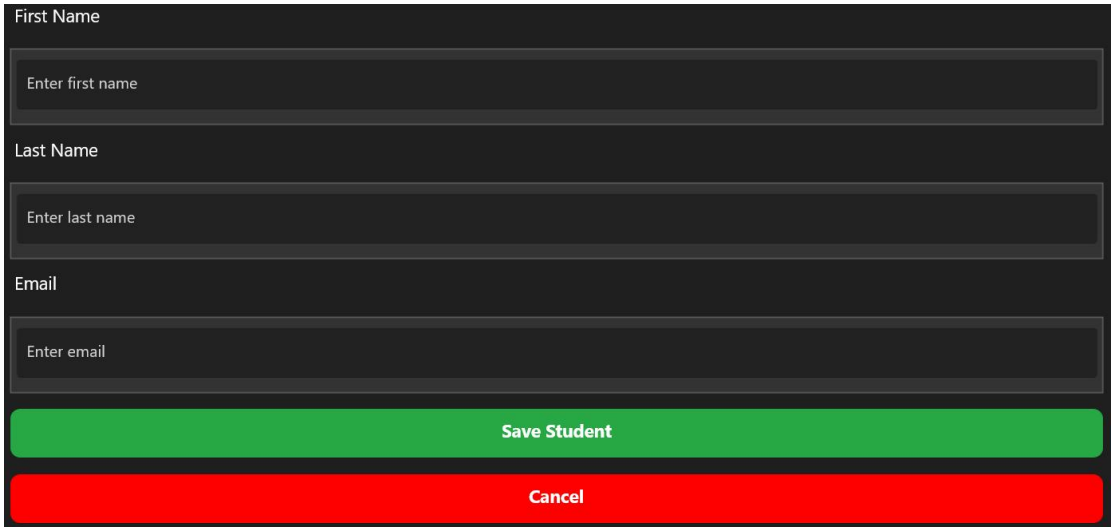


- **SettingsView.xaml -** Тази страница е достъпна единствено чрез Flyout менюто, което е интегрирано в приложението. Нейната основна цел е да предостави на потребителя възможност да избере предпочитания режим на визуализация – Light или Dark mode. Това се реализира чрез Switch компонент, който позволява динамична промяна на темата в реално време. При превключване приложението автоматично адаптира цветовата схема спрямо избрания режим, осигурявайки по-добро потребителско изживяване.



След смяна на изгледа сме автоматично прехвърлени към основния списък, но в тъмен режим:





Това са основните и най-важни компоненти по UI частта на програмата.

Ето някой по-интересни части от кода, които показват привързване (Binding):

```
xmlns:vm="clr-namespace:StudentsManagement.ViewModels"
xmlns:data="clr-namespace:StudentsManagement.Data"
x:DataType="vm:StudentListViewModel"
```

```
<CollectionView
    Grid.Row="1"
    ItemsSource="{Binding Students}"
    VerticalOptions="FillAndExpand"
>
    <CollectionView.ItemTemplate>
        <DataTemplate x:DataType="data:StudentEntity">
            <Border Margin="10" Padding="10">
                <StackLayout>
                    <HorizontalStackLayout Spacing="10">
                        <Label Text="{Binding FirstName}" FontSize="18" FontAttributes="Bold" />
                        <Label Text="{Binding LastName}" FontSize="18" FontAttributes="Bold" />
                    </HorizontalStackLayout>
                    <Label Text="{Binding Email}" FontSize="16" />

                    <BoxView HeightRequest="1" Color="LightGray" Margin="0, 5"/>
                </StackLayout>

                <Border.GestureRecognizers>
                    <TapGestureRecognizer
                        Command="{Binding Source={x:RelativeSource AncestorType={x:Type vm:StudentListViewModel}},
                        Path=DisplayActionCommand}"
                        CommandParameter="{Binding .}"
                        />
                </Border.GestureRecognizers>
            </Border>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

Дефиниране на стилове за двата вида режима:

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Style TargetType="Label">
            <Setter Property="FontSize" Value="18" />
            <Setter Property="TextColor" Value="{AppThemeBinding Light=Black, Dark=White}" />
            <Setter Property="Margin" Value="5,0,0,5" />
        </Style>

        <Style TargetType="Entry">
            <Setter Property="FontSize" Value="16" />
            <Setter Property="BackgroundColor" Value="{AppThemeBinding Light=White, Dark=#222222}" />
            <Setter Property="TextColor" Value="{AppThemeBinding Light=Black, Dark=White}" />
            <Setter Property="Margin" Value="0,5,0,10" />
            <Setter Property="HeightRequest" Value="50" />
            <Setter Property="PlaceholderColor" Value="{AppThemeBinding Light=Gray, Dark=LightGray}" />
        </Style>

        <Style TargetType="Button">
            <Setter Property="HeightRequest" Value="50" />
            <Setter Property="FontSize" Value="18" />
            <Setter Property="FontAttributes" Value="Bold" />
            <Setter Property="CornerRadius" Value="10" />
            <Setter Property="TextColor" Value="White" />
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>
```

5. Преминаваме към един от най-важните компоненти на проекта – ViewModel слоя, който играе ключова роля като свързващо звено между Model (бизнес логиката и данните) и View (потребителския интерфейс). Този слой гарантира, че потребителският интерфейс остава чист и независим от бизнес логиката, като подобрява разширяемостта и поддръжката на приложението.

* Роля на ViewModel слоя

- Осигурява двупосочна връзка (Data Binding) между интерфейса и данните.

- Управлява логиката за обработка на потребителски събития (напр. натискане на бутони).
- Минимизира директното взаимодействие между View и Model, следвайки принципите на MVVM.

Имплементация

- Създаваме клас StudentViewModel.cs, който ще съдържа свойствата и командите за работа със списъка от ученици.
- Добавяме ObservableCollection<StudentEntity>, за да управляваме динамично списъка и да следим промените.
- Дефинираме команди за добавяне, редактиране и изтриване на ученици. Използваме вградената в CommunityToolkit.Mvvm.Input [RelayCommand].
- Осигуряваме обработка на грешки, като валидираме въведените данни преди запис в базата.

```
public partial class StudentListViewModel : ObservableObject
{
    5 references
    public ObservableCollection<StudentEntity> Students { get; set; } = new ObservableCollection<StudentEntity>();

    private readonly IStudentService _studentService;

    0 references
    public StudentListViewModel(IStudentService studentService)
    {
        _studentService = studentService;
    }
```

```
[RelayCommand]
8 references
public async Task GetStudentList()
{
    try
    {
        var studentList = await _studentService.GetStudentList();

        if (studentList?.Any() == true)
        {
            Students.Clear();
            foreach (var student in studentList)
            {
                Students.Add(student);
            }
        }
        else
        {
            await Shell.Current.DisplayAlert("Info", "No student records found. Add new students.", "OK");
        }
    }
    catch (DatabaseException)
    {
        await Shell.Current.DisplayAlert("Error", "Failed to load student list. Please try again.", "OK");
    }
    catch (Exception)
    {
        await Shell.Current.DisplayAlert("Error", "Unexpected error occured. Please try again.", "OK");
    }
}
```

```
public async Task DisplayAction(StudentEntity studentEntity)
{
    try
    {
        var response = await Shell.Current.DisplayActionSheet("Select Option", "OK", null, "Edit", "Delete");

        if (response == "Edit")
        {
            var navParam = new Dictionary<string, object>();
            navParam?.Add("StudentDetail", studentEntity);

            await Shell.Current.GoToAsync(nameof(AddUpdateStudentView), navParam);
        }
        else if (response == "Delete")
        {
            bool confirm = await Shell.Current.DisplayAlert("Confirm",
                "Are you sure you want to delete this student?", "Yes", "No");

            if (confirm)
            {
                var delResponse = await _studentService.DeleteStudent(studentEntity);
```

След това създаваме AddUpdateStudentViewModel.cs, като неговата функционалност е сходна с тази на StudentViewModel, но отговаря за AddUpdateStudentView.xaml страницата и логиката за добавяне / редактиране на ученици.

*Забележка: Валидацията на данни е направена в самия метод за добавяне, като би било хубаво да е в отделен Code File, за да може да бъде преизползвана при необходимост (в случая не се преизползва).

```
const string NAME_PATTERN = @"^[A-Za-z\s]{2,50}$";
const string EMAIL_PATTERN = @"^[^@\s]+@[^@\s]+\.[^@\s]+$";

try
{
    if (string.IsNullOrWhiteSpace(StudentDetail.FirstName) || !Regex.IsMatch(StudentDetail.FirstName, NAME_PATTERN))
    {
        await Shell.Current.DisplayAlert("Validation Error", "First name must contain only letters and be at least 2 chara
        return;
    }

    if (string.IsNullOrWhiteSpace(StudentDetail.LastName) || !Regex.IsMatch(StudentDetail.LastName, NAME_PATTERN))
    {
        await Shell.Current.DisplayAlert("Validation Error", "Last name must contain only letters and be at least 2 charac
        return;
    }

    if (string.IsNullOrWhiteSpace(StudentDetail.Email) || !Regex.IsMatch(StudentDetail.Email, EMAIL_PATTERN))
    {
        await Shell.Current.DisplayAlert("Validation Error", "Please enter a valid email address.", "OK");
        return;
    }
```

И в двата ViewModel-а сме добавили StudentService.cs, от който преизползваме методите за CRUD операциите. Това позволява централизирано управление на данните, като логиката за достъп до базата данни остава в сервизния слой, а ViewModel-ите фокусират върху свързването на данните с потребителския интерфейс.

Завършваме със SettingsViewModel.cs.

За направата на Flyout-а е използван следния код във файла AppShell.xaml:

```
<ShellContent
    Title="Students"
    ContentTemplate="{DataTemplate views:StudentListView}"
    Route="StudentListView" />
<ShellContent
    Title="Manage Students"
    ContentTemplate="{DataTemplate views:AddUpdateStudentView}"
    Route="AddUpdateStudentView" />
<ShellContent
    Title="Settings"
    ContentTemplate="{DataTemplate views:SettingsView}"
    Route="SettingsView" />
```

```
public AppShell()
{
    InitializeComponent();

    Routing.RegisterRoute(nameof(AddUpdateStudentView), typeof(AddUpdateStudentView));
}
```

Навигацията в самия проект се извършва през Shell.

В MauiProgram.cs регистрираме зависимостите в .NET MAUI приложението чрез Dependency Injection (DI). Това позволява на приложението автоматично да управлява създаването и предаването на обекти, което подобрява модулността, тестируемостта и поддръжката на кода.

```
// Services
builder.Services.AddSingleton<IStudentService, StudentService>();

//Views Registration
builder.Services.AddSingleton<StudentListView>();
builder.Services.AddTransient<AddUpdateStudentView>();

// ViewModels Registration
builder.Services.AddSingleton<StudentListViewModel>();
builder.Services.AddTransient<AddUpdateStudentViewModel>();
```

След изпълнението на тези стъпки, проектът е в завършена фаза и е готов за тестване и използване.

7. Потребителско ръководство

Функционалността на приложението бе демонстрирана в предходната точка на документация. Ще бъде отново показана на живо при представяне на проекта. При желание за тестване от ваша може да свалите solution-а от моето github repository: https://github.com/ValeryRaikov/Maui-Course-Project

8. Литература и използвани източници

https://www.youtube.com/watch?v=Hh279ES_FNQ&list=PLdo4fOcmZ0oUBAdL2NwBpDs32zwGqb9DY

https://www.youtube.com/watch?v=DuNLR_NJv8U

https://learn.microsoft.com/en-us/dotnet/maui/?view=net-maui-9.0

https://stackoverflow.com/questions/tagged/maui

https://chatgpt.com/