

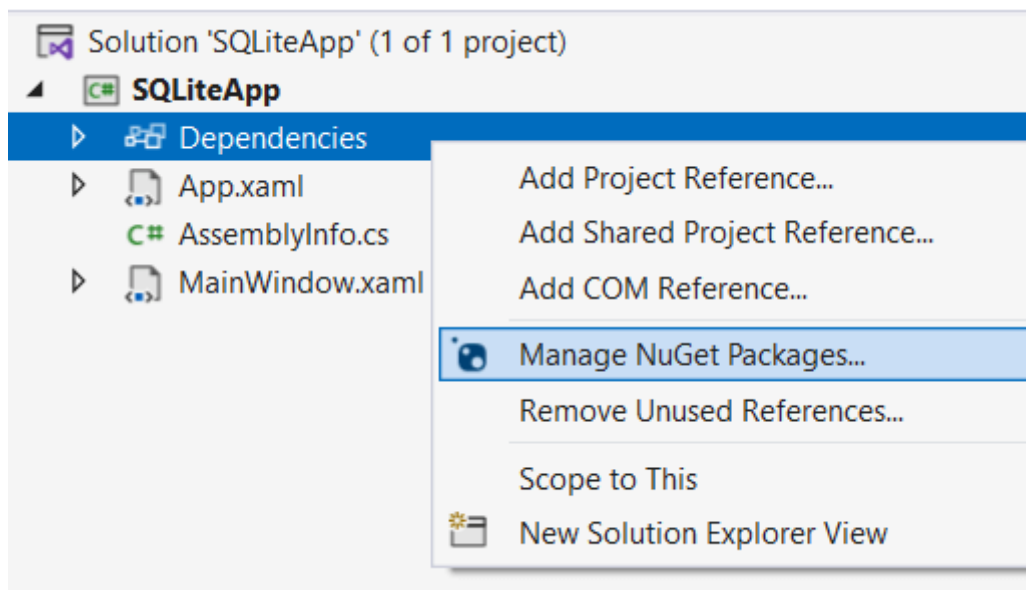
Работа с SQLite через Entity Framework

SQLite является одной из наиболее используемых систем управления базами данных. Главным преимуществом SQLite является то, что для базы данных не нужно сервера. База данных представляет собой обычный локальный файл, который мы можем перемещать вместе с главным файлом приложения. Кроме того, для запросов к базе данных мы можем использовать стандартные выражения языка SQL, которые равным образом с некоторыми изменениями могут применяться и в других СУБД как Oracle, MS SQL Server, MySQL, Postgres и т.д.

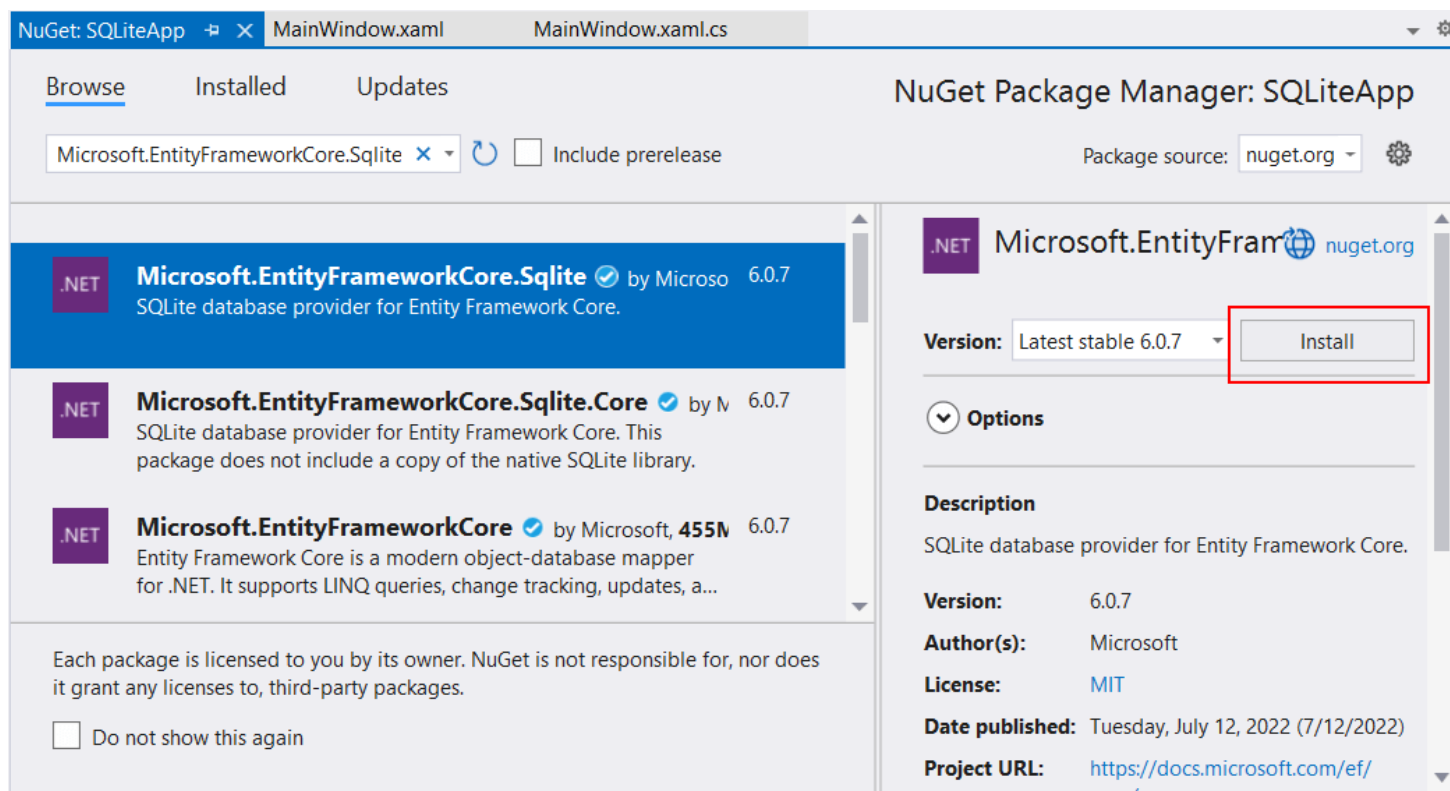
Еще одним преимуществом является широкое распространение SQLite - область применения охватывает множество платформ и технологий: WPF, Windows Forms, UWP, Android, iOS и т.д. И если мы, скажем, захотим создать копию приложения для ОС Android, то базу данных мы можем перенести в новое приложение такой, как она определена для приложения на WPF.

Но в данном случае для упрощения мы будем работать с базой данных SQLite через специальный ORM-фреймворк Entity Framework. В данном случае мы сосредоточимся на тех моментах, которые характерны для проекта WPF. Но при необходимости дополнительную информацию можно найти по следующим ссылкам: [SQLite в C# и .NET](#) и [Руководство по Entity Framework](#)

Итак, создадим новый проект WPF, который назовем SQLiteApp. В первую очередь нам надо добавить функциональность SQLite и Entity Framework в проект. Для этого необходимо добавить в проект через пакетный менеджер Nuget пакет **Microsoft.EntityFrameworkCore.Sqlite**. Итак, перейдем к Nuget, нажав в проекте правой кнопкой на узел Dependencies и выбрав в открывшемся контекстном меню пункт **Manage NuGet Packages...**:



В окне менеджера NuGet введем в окно поиска "Microsoft.EntityFrameworkCore.Sqlite", и менеджер отобразит нам ряд результатов. Из них нам надо установить пакет под названием **Microsoft.EntityFrameworkCore.Sqlite**:



Итак, для работы с базой данных нам естественно понадобится сама база данных. Однако если у нас нет никакой БД, как в данном случае, мы можем воспользоваться подходом Code First, чтобы Entity Framework сам автоматически создал нам базу данных по определению используемых классов.

Вначале определим в проекте класс, объекты которого будут храниться в базе данных. Пусть это будет класс `User` :

```
namespace SQLiteApp
{
    public class User
    {
        public int Id { get; set; }
        public string? Name { get; set; }
        public int Age { get; set; }
    }
}
```

В данном случае класс `User` содержит свойство `Id`, которое будет выполнять роль уникального идентификатора пользователя. Кроме того, класс определяет свойство `Name` (имя пользователя) и `Age` (возраст пользователя).

Ключевым компонентом для взаимодействия с базой данных является контекст данных. Поэтому добавим в проект новый класс, который назовем **ApplicationContext** и который будет выполнять роль контекста данных:

```
using Microsoft.EntityFrameworkCore;

namespace SQLiteApp
{
    public class ApplicationContext : DbContext
    {
        public DbSet<User> Users {get;set; } = null!;
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=helloapp.db");
        }
    }
}
```

Класс контекста должен наследоваться от DbContext. И для взаимодействия с таблицей, которая будет хранить данные объектов User, здесь также определено свойство Users. Чтобы указать анализатору кода, что данное свойство не будет равно null, свойству Users присваивается значение null!

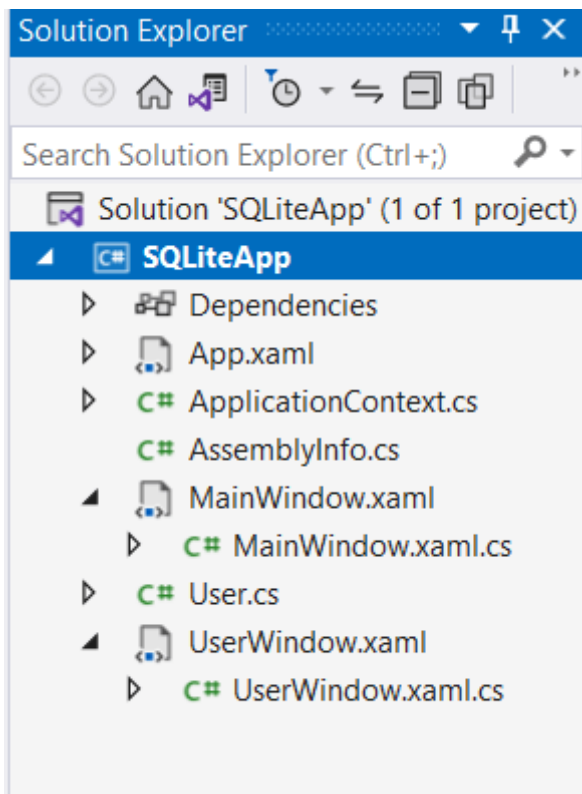
Кроме того, чтобы связать контекст данных с конкретной базой данных, в методе OnConfiguring с помощью вызова

```
optionsBuilder.UseSqlite("Data Source=helloapp.db");
```

устанавливается используемая база данных. В частности методу UseSqlite передается строка подключения, в которой есть только один параметр - Data Source, который указывает на имя базы данных. То есть приложение будет использовать базу данных с именем "helloapp.db". Причем не важно, что сейчас такой базы данных не существует - она будет создана автоматически.

Теперь все готово для работы с базой данных. Стандартный набор операций по работе с БД включает получение объектов, добавление, изменение и удаление. Для получения и просмотра списка объектов из бд мы будем использовать главное окно. А для добавления и изменения создадим новое окно.

Итак, добавим в проект новое окно, которое назовем **UserWindow.xaml**. В итоге общая конфигурация проекта у нас будет выглядеть следующим образом:



В коде xaml у страницы UserWindow.xaml определим следующее содержимое:

```

<Window x:Class="SQLiteApp.UserWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="UserWindow" Height="200" Width="300">
<Window.Resources>
    <Style TargetType="TextBlock">
        <Setter Property="Margin" Value="8" />
    </Style>
    <Style TargetType="TextBox">
        <Setter Property="Margin" Value="8" />
    </Style>
    <Style TargetType="Button">
        <Setter Property="MinWidth" Value="60" />
        <Setter Property="Margin" Value="8" />
    </Style>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBlock Text="Имя" />
    <TextBlock Text="Возраст" Grid.Row="1" />

    <TextBox Text="{Binding Name}" Grid.Column="1" />
    <TextBox Text="{Binding Age}" Grid.Column="1" Grid.Row="1" />

    <StackPanel HorizontalAlignment="Center" Orientation="Horizontal" Grid.Row="3" Grid.Column="1">
        <Button IsDefault="True" Click="Accept_Click" >OK</Button>
        <Button IsCancel="True" >Отмена</Button>
    </StackPanel>
</Grid>
</Window>

```

Здесь определены два поля ввода для каждого свойства модели User и две кнопки для сохранения и отмены.

В файле связанного кода **UserWindow.xaml.cs** определим контекст для этой страницы:

```

using System.Windows;
namespace SQLiteApp
{
    public partial class UserWindow : Window
    {
        public User User { get; private set; }
        public UserWindow(User user)
        {
            InitializeComponent();
            User = user;
            DataContext = User;
        }

        void Accept_Click(object sender, RoutedEventArgs e)
        {
            DialogResult = true;
        }
    }
}

```

Данное окно будет диалоговым. Через конструктор оно будет получать объект User, который устанавливается в качестве контекста данных.

В коде xaml у главного окна MainWindow определим вывод списка телефонов и набор кнопок для управления этим списком:

```

<Window x:Class="SQLiteApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="MainWindow" Height="300" Width="425">
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="50" />
    </Grid.RowDefinitions>
    <ListBox x:Name="usersList" ItemsSource="{Binding}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding Name}" FontSize="16" />
                    <TextBlock Text="{Binding Age}" FontSize="13" />
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
    <StackPanel Grid.Row="1" Orientation="Horizontal" HorizontalAlignment="Center">
        <Button Content="Добавить" Margin="10" Click="Add_Click" />
        <Button Content="Изменить" Margin="10" Click="Edit_Click" />
        <Button Content="Удалить" Margin="10" Click="Delete_Click" />
    </StackPanel>
</Grid>
</Window>

```

В коде C# у этого окна пропишем обработчики кнопок, через которые будем взаимодействовать с базой данных SQLite:

```

using Microsoft.EntityFrameworkCore;
using System.Windows;
namespace SQLiteApp
{
    public partial class MainWindow : Window
    {
        ApplicationContext db = new ApplicationContext();
        public MainWindow()
        {
            InitializeComponent();

            Loaded += MainWindow_Loaded;
        }

        // при загрузке окна
        private void MainWindow_Loaded(object sender, RoutedEventArgs e)
        {
            // гарантируем, что база данных создана
            db.Database.EnsureCreated();
            // загружаем данные из БД
            db.Users.Load();
            // и устанавливаем данные в качестве контекста
            DataContext = db.Users.Local.ToObservableCollection();
        }

        // добавление
        private void Add_Click(object sender, RoutedEventArgs e)
        {
            UserWindow UserWindow = new UserWindow(new User());
            if (UserWindow.ShowDialog() == true)
            {
                User User = UserWindow.User;
                db.Users.Add(User);
                db.SaveChanges();
            }
        }

        // редактирование
        private void Edit_Click(object sender, RoutedEventArgs e)
        {
            // получаем выделенный объект
            User? user = usersList.SelectedItem as User;
            // если ни одного объекта не выделено, выходим
            if (user is null) return;

            UserWindow UserWindow = new UserWindow(new User
            {
                Id = user.Id,
                Age = user.Age,
                Name = user.Name
            });
        }
    }
}

```



```

        if (UserWindow.ShowDialog() == true)
        {
            // получаем измененный объект
            user = db.Users.Find(UserWindow.User.Id);
            if (user != null)
            {
                user.Age = UserWindow.User.Age;
                user.Name = UserWindow.User.Name;
                db.SaveChanges();
                usersList.Items.Refresh();
            }
        }
    }
    // удаление
    private void Delete_Click(object sender, RoutedEventArgs e)
    {
        // получаем выделенный объект
        User? user = usersList.SelectedItem as User;
        // если ни одного объекта не выделено, выходим
        if (user is null) return;
        db.Users.Remove(user);
        db.SaveChanges();
    }
}

```

В коде класса прежде всего определяется переменная `db`, которая представляет контекст данных `ApplicationContext` и через которую будет идти взаимодействие с базой данных.

В конструкторе окна определяем обработчик загрузки окна, в котором, во-первых, гарантируем, что база данных создана, и для этого вызываем метод

```
db.Database.EnsureCreated();
```

В итоге даже если базы данных не существует, она будет создана в той же папке, где расположен файл приложения. Если бд уже имеется, тогда она просто будет использоваться.

Далее выражение `db.Users.Load()` загружает данные из таблицы `Users` в локальный кэш контекста данных. И затем список загруженных объектов устанавливается в качестве контекста данных:

```
DataContext = db.Users.Local.ToObservableCollection();
```

Для добавления вызывается метод `Add`:

```
db.Users.Add(user);  
db.SaveChanges();
```

Для удаления - метод Remove:

```
db.Users.Remove(user);  
db.SaveChanges();
```

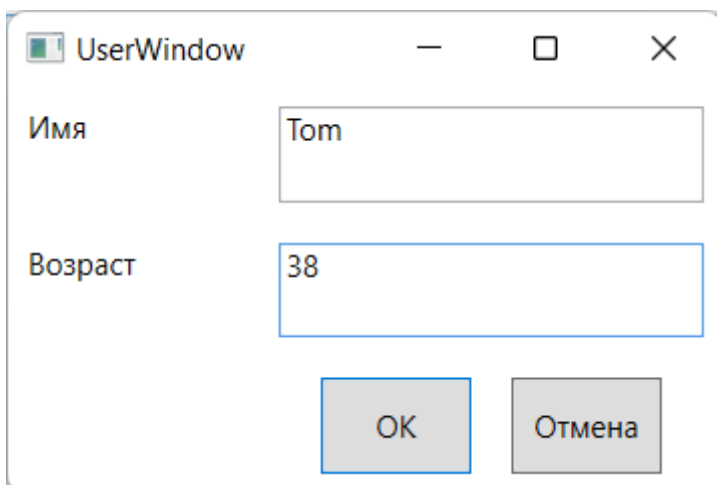
При изменении мы передаем в UserWindow копию выбранного объекта. Если мы передавали бы сам выделенный объект, то все изменения на форме автоматически синхронизировались со списком, и не было бы смысла в кнопке отмены.

После получения измененного объекта мы находим его в базе данных и устанавливаем у него новые значения свойств, после чего сохраняем все изменения:

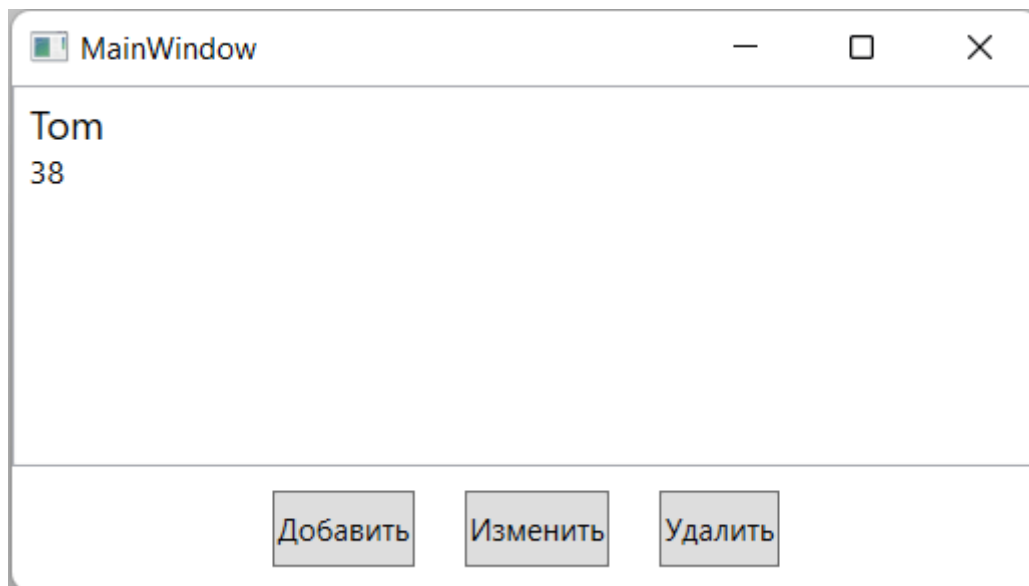
```
user = db.Users.Find(UserWindow.User.Id);  
if (user != null)  
{  
    user.Age = UserWindow.User.Age;  
    user.Name = UserWindow.User.Name;  
    db.SaveChanges();  
    usersList.Items.Refresh();  
}
```

Здесь надо обратить внимание на вызов usersList.Items.Refresh(), который после обновления объекта в бд обновляется отображение списка. В качестве альтернативы мы бы могли реализовать интерфейс INotifyPropertyChanged и уведомлять систему при изменении значения свойства Name и/или Age.

Запустим приложение. И добавим какой нибудь объект:



И после добавления объект отобразится в списке:



MVVM и SQLite

В прошлой теме было рассмотрено, как подключаться к SQLite. Теперь посмотрим, как мы можем совместить это с паттерном MVVM.

Итак, возьмем проект WPF, который пусть называется SQLiteApp. В первую очередь нам надо добавить функциональность SQLite и Entity Framework в проект. Для этого добавим в проект через пакетный менеджер Nuget пакет **Microsoft.EntityFrameworkCore.Sqlite**, как это было показано в прошлой статье.

Model

Вначале определим в проекте класс, объекты которого будут храниться в базе данных. Пусть это будет класс **User**:

```

using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace SQLiteApp
{
    public class User : INotifyPropertyChanged
    {
        string? name;
        int age;
        public int Id { get; set; }
        public string? Name
        {
            get { return name; }
            set
            {
                name = value;
                OnPropertyChanged("Name");
            }
        }
        public int Age
        {
            get { return age; }
            set
            {
                age = value;
                OnPropertyChanged("Age");
            }
        }

        public event PropertyChangedEventHandler? PropertyChanged;
        public void OnPropertyChanged([CallerMemberName] string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(prop));
        }
    }
}

```

Класс User содержит свойство Id, которое будет выполнять роль уникального идентификатора пользователя. Кроме того, класс определяет свойство Name (имя пользователя) и Age (возраст пользователя). Класс реализует интерфейс INotifyPropertyChanged, что позволяет ему уведомлять систему об изменении значений свойств Name и Age.

Теперь добавим в проект новый класс, который назовем **ApplicationContext** и который будет выполнять роль контекста данных:

```
using Microsoft.EntityFrameworkCore;

namespace SQLiteApp
{
    public class ApplicationContext : DbContext
    {
        public DbSet<User> Users {get;set; } = null!;
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlite("Data Source=helloapp.db");
        }
    }
}
```

И для взаимодействия с таблицей, которая будет хранить данные объектов User, здесь определено свойство Users. Чтобы связать контекст данных с конкретной базой данных, в методе OnConfiguring устанавливается используемая база данных. В частности методу UseSqlite передается строка подключения, в которой есть только один параметр - Data Source, который указывает на имя базы данных. То есть приложение будет использовать базу данных с именем "helloapp.db".

View для отдельного объекта

Далее определим в проекте новое окно, которое назовем **UserWindow.xaml**. В коде xaml у страницы **UserWindow.xaml** определим следующее содержимое:

```

<Window x:Class="SQLiteApp.UserWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="UserWindow" Height="200" Width="300">
<Window.Resources>
    <Style TargetType="TextBlock">
        <Setter Property="Margin" Value="8" />
    </Style>
    <Style TargetType="TextBox">
        <Setter Property="Margin" Value="8" />
    </Style>
    <Style TargetType="Button">
        <Setter Property="MinWidth" Value="60" />
        <Setter Property="Margin" Value="8" />
    </Style>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBlock Text="Имя" />
    <TextBlock Text="Возраст" Grid.Row="1" />

    <TextBox Text="{Binding Name}" Grid.Column="1" />
    <TextBox Text="{Binding Age}" Grid.Column="1" Grid.Row="1" />

    <StackPanel HorizontalAlignment="Center" Orientation="Horizontal" Grid.Row="3" Grid.Column="1">
        <Button IsDefault="True" Click="Accept_Click" >OK</Button>
        <Button IsCancel="True" >Отмена</Button>
    </StackPanel>
</Grid>
</Window>

```

Здесь определены два поля ввода для каждого свойства модели User и две кнопки для сохранения и отмены.

В файле связанного кода **UserWindow.xaml.cs** определим контекст для этой страницы:

```
using System.Windows;
namespace SQLiteApp
{
    public partial class UserWindow : Window
    {
        public User User { get; private set; }
        public UserWindow(User user)
        {
            InitializeComponent();
            User = user;
            DataContext = User;
        }

        void Accept_Click(object sender, RoutedEventArgs e)
        {
            DialogResult = true;
        }
    }
}
```

Данное окно будет диалоговым. Через конструктор оно будет получать объект User, который устанавливается в качестве контекста данных.

ViewModel

Основная логика в MVVM заключена в компоненте ViewModel, с которым взаимодействует представление или графический интерфейс посредством команд. Поэтому вначале добавим в проект новый класс RelayCommand, который будет представлять команду:

```

using System;
using System.Windows.Input;

namespace SQLiteApp
{
    public class RelayCommand : ICommand
    {
        Action<object?> execute;
        Func<object?, bool>? canExecute;

        public event EventHandler? CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }

        public RelayCommand(Action<object?> execute, Func<object?, bool>? canExecute = null)
        {
            this.execute = execute;
            this.canExecute = canExecute;
        }

        public bool CanExecute(object? parameter)
        {
            return canExecute == null || canExecute(parameter);
        }

        public void Execute(object? parameter)
        {
            execute(parameter);
        }
    }
}

```

Этот класс реализует интерфейс **ICommand**, благодаря чему с помощью подобных команды мы сможем направлять вызовы к ViewModel. Ключевым здесь является метод Execute(), который получает параметр и выполняет действие, переданное через конструктор команды.

Затем определим в проекте собственно компонент ViewModel. Добавим новый класс **ApplicationViewModel**:


```

using System.Collections.ObjectModel;
using Microsoft.EntityFrameworkCore;

namespace SQLiteApp
{
    public class ApplicationViewModel
    {
        ApplicationDbContext db = new ApplicationDbContext();
        RelayCommand? addCommand;
        RelayCommand? editCommand;
        RelayCommand? deleteCommand;
        public ObservableCollection<User> Users { get; set; }
        public ApplicationViewModel()
        {
            db.Database.EnsureCreated();
            db.Users.Load();
            Users = db.Users.Local.ToObservableCollection();
        }
        // команда добавления
        public RelayCommand AddCommand
        {
            get
            {
                return addCommand ??
                    (addCommand = new RelayCommand((o) =>
                    {
                        UserWindow userWindow = new UserWindow(new User());
                        if (userWindow.ShowDialog() == true)
                        {
                            User user = userWindow.User;
                            db.Users.Add(user);
                            db.SaveChanges();
                        }
                    }));
            }
        }
        // команда редактирования
        public RelayCommand EditCommand
        {
            get
            {
                return editCommand ??
                    (editCommand = new RelayCommand((selectedItem) =>
                    {
                        // получаем выделенный объект
                        User? user = selectedItem as User;
                        if (user == null) return;

                        User vm = new User
                        {

```


Команда добавления отображает диалоговое окно UserWindow, которое было создано в прошлой теме, и добавляет в базу данных новый объект:

```
public RelayCommand AddCommand
{
    get
    {
        return addCommand ??
            (addCommand = new RelayCommand((o) =>
            {
                UserWindow userWindow = new UserWindow(new User());
                if (userWindow.ShowDialog() == true)
                {
                    User user = userWindow.User;
                    db.Users.Add(user);
                    db.SaveChanges();
                }
            }));
    }
}
```

Команда изменения получает объект, который надо изменить, также вызывает для него диалоговое окно UserWindow и сохраняет изменения в базу данных:

```

public RelayCommand EditCommand
{
    get
    {
        return editCommand ??
            (editCommand = new RelayCommand((selectedItem) =>
            {
                // получаем выделенный объект
                User? user = selectedItem as User;
                if (user == null) return;

                User vm = new User
                {
                    Id = user.Id,
                    Name = user.Name,
                    Age = user.Age
                };
                UserWindow userWindow = new UserWindow(vm);

                if (userWindow.ShowDialog() == true)
                {
                    user.Name = userWindow.User.Name;
                    user.Age = userWindow.User.Age;
                    db.Entry(user).State = EntityState.Modified;
                    db.SaveChanges();
                }
            }));
    }
}

```

В команде удаления также получаем объект, который надо удалить, и удаляем его из БД:

```

public RelayCommand DeleteCommand
{
    get
    {
        return deleteCommand ??
            (deleteCommand = new RelayCommand((selectedItem) =>
            {
                User? user = selectedItem as User;
                if (user == null) return;
                db.Users.Remove(user);
                db.SaveChanges();
            }));
    }
}

```

То есть практически все действия, которые в прошлой теме производились в коде главного окна, теперь вынесены в команды в ApplicationViewModel.

Теперь код файла **MainWindow.xaml.cs**:

```
using System.Windows;

namespace SQLiteApp
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            DataContext = new ApplicationViewModel();
        }
    }
}
```

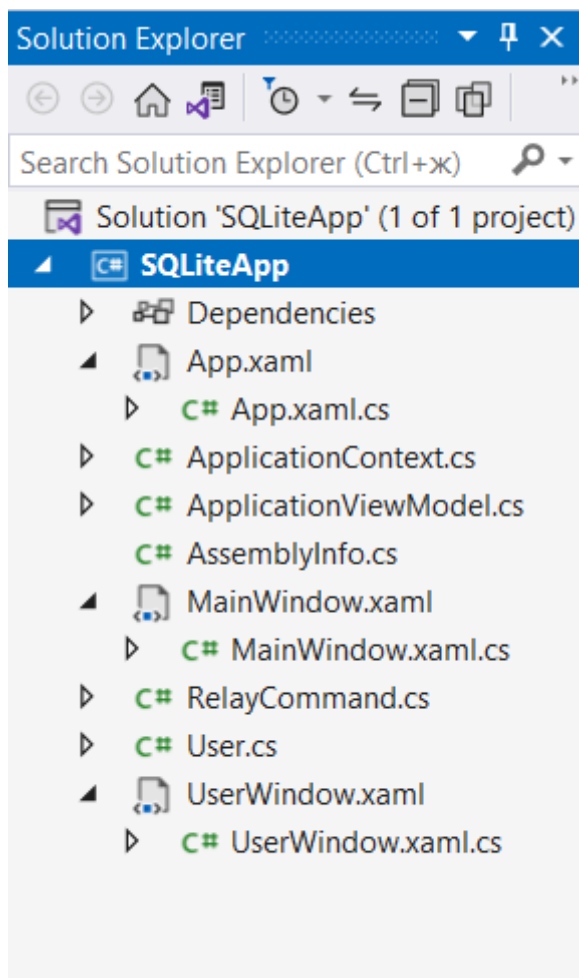
Теперь в качестве контекста данных выступает объект ApplicationViewModel. И далее в коде XAML у MainWindow определим выражения привязки к командам и к списку Users:

```

<Window x:Class="SQLiteApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="MainWindow" Height="300" Width="425">
<Window.Resources>
    <Style TargetType="Button">
        <Setter Property="MinWidth" Value="60" />
        <Setter Property="Margin" Value="8" />
    </Style>
</Window.Resources>
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition Height="50" />
    </Grid.RowDefinitions>
    <ListBox x:Name="usersList" ItemsSource="{Binding Users}">
        <ListBox.ItemTemplate>
            <DataTemplate>
                <StackPanel>
                    <TextBlock Text="{Binding Name}" FontSize="16" />
                    <TextBlock Text="{Binding Age}" FontSize="13" />
                </StackPanel>
            </DataTemplate>
        </ListBox.ItemTemplate>
    </ListBox>
    <StackPanel Grid.Row="1" Orientation="Horizontal" HorizontalAlignment="Center">
        <Button Content="Добавить" Margin="10" Padding="3" Command="{Binding AddCommand}" /
        <Button Content="Изменить" Margin="10" Command="{Binding EditCommand}"
            CommandParameter="{Binding ElementName=usersList, Path=SelectedItem}" />
        <Button Content="Удалить" Margin="10" Command="{Binding DeleteCommand}"
            CommandParameter="{Binding ElementName=usersList, Path=SelectedItem}" />
    </StackPanel>
</Grid>
</Window>

```

В итоге получится следующая структура проекта:



И теперь все вызовы к базе данных SQLite будут идти через ApplicationViewModel.

