Лекция №10. Оконные приложения

Рассмотрим фреймворки для разработки оконных приложений на C#.

WinForms (Windows Forms) появились как один из первых инструментов для создания графических приложений под Windows. Они предлагали довольно простой способ визуальной разработки через drag-and-drop в Visual Studio.

- **Преимущества:**
  - Простота и быстрота освоения.
  - Интеграция с Visual Studio, позволяющая быстро создавать прототипы и оформлять визуальные элементы.
- **Ограничения:**
  - Ограниченность функционала по сравнению с более современными технологиями.
  - Слабая поддержка современных UI-требований и сложная масштабируемость.

WPF (Windows Presentation Foundation) пришёл на смену WinForms, предлагая декларативный подход к разработке интерфейсов с использованием **XAML (Extensible Application Markup Language)**. Это позволило создать более гибкие и сложные графические интерфейсы, используя привязку данных, анимацию и стилизацию.

- **Преимущества:**
  - Богатые возможности для создания кастомного интерфейса.
  - Гибкость за счет использования XAML и поддержки разделения логики и представления (например, через паттерн MVVM).
- **Ограничения:**
  - Приложения работают только на Windows.
  - Более высокий порог входа по сравнению с WinForms.

Xamarin фреймворк для разработки на мобильные приложения, который позволял создавать нативные приложения для iOS, Android и других платформ с использованием C#.

- **Преимущества:**
  - Возможность переиспользования кода между платформами.
  - Нативный доступ ко всем функциям мобильных ОС.
- **Ограничения:**
  - Некоторые сложности с поддержанием единого интерфейса для всех платформ.
  - Ограничения в области десктопной разработки.

MAUI (Multi-platform App UI) — это современная эволюция Xamarin.Forms, созданная для поддержки разработки кроссплатформенных приложений сразу для мобильных (iOS, Android) и десктопных (Windows, macOS) устройств.

- **Преимущества:**
  - Единая кодовая база для мобильных и десктопных платформ.
  - Глубокая интеграция с экосистемой .NET и поддержка Microsoft.
  - Современные возможности XAML и привязки данных.
- **Ограничения:**
  - Нет официальной поддержки Linux.
  - Требования к современным инструментам разработки (Visual Studio, .NET 6/7/8).

Avalonia — это современный, полностью открытый кроссплатформенный UI-фреймворк для .NET, вдохновлённый WPF, но не зависящий от Windows. Он позволяет создавать приложения, работающие на Windows, Linux, macOS, а также экспериментально поддерживает веб и мобильные платформы.

- **Преимущества:**
  - Открытый исходный код и активное сообщество.
  - Поддержка Linux и других ОС, где нет официальной поддержки от Microsoft.
  - Знакомый разработчикам WPF синтаксис XAML и паттерн MVVM.
- **Ограничения:**
  - Некоторые возможности (например, мобильная или веб-поддержка) ещё находятся в стадии разработки.
  - Меньшая экосистема по сравнению с решениями от Microsoft.

Сравнение фреймворков.

Фреймворк	Платформы	Основные особенности	Целевая аудитория
WinForms	Windows	Простая разработка, drag-and-drop, быстрое прототипирование	Классические Windows-приложения
WPF	Windows	Богатый UI, XAML, поддержка MVVM, высокая кастомизация	Приложения с требованием сложной визуализации
Xamarin	iOS, Android, иногда Windows	Нативная мобильная разработка, разделяемый код	Мобильные приложения

Фреймворк	Платформы	Основные особенности	Целевая аудитория
.NET MAUI	iOS, Android, Windows, macOS	Единый код для мобильных и десктопных платформ, интеграция с .NET	Кроссплатформенные приложения (кроме Linux)
Avalonia	Windows, Linux, macOS, (экспериментально: веб, мобиль)	Открытый исходный код, XAML, MVVM, кроссплатформенность	Приложения с поддержкой Linux и других ОС

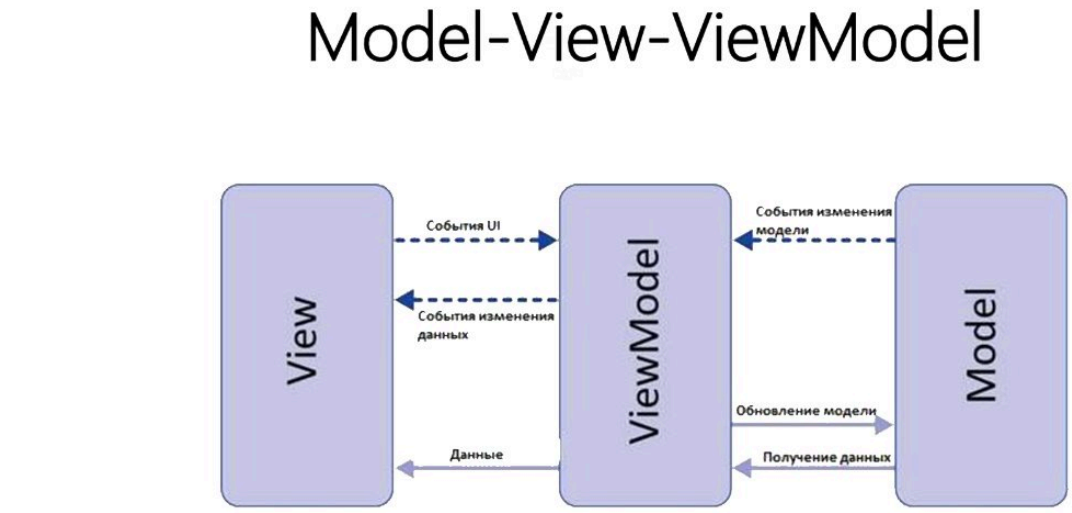
MVVM

MVVM (Model-View-ViewModel) — это архитектурный паттерн, ориентированный на разделение ответственности в приложении. Он помогает отделить логику бизнес-правил от представления пользовательского интерфейса.

- **Model (Модель):**
  Представляет данные и бизнес-логику. Это могут быть классы, работающие с данными, запросами к базе данных, API и т.д.
- **View (Представление):**
  Отвечает за визуальное отображение данных. Это XAML-разметка, которая определяет, какие элементы будут видны пользователю.
- **ViewModel (Модель представления):**
  Посредник между Model и View. Здесь реализуются команды, свойства для привязки данных (data binding) и логика взаимодействия между пользовательским интерфейсом и данными.

Этот паттерн широко используется при разработке UI-приложений, т.к. имеет ряд преимуществ:

- Чёткое разделение между логикой и представлением позволяет легче тестировать и поддерживать код.
- Автоматическое обновление представления при изменениях данных.
- Разработчики и дизайнеры могут работать параллельно над разными частями проекта.
- Повышенная модульность и переиспользуемость кода.

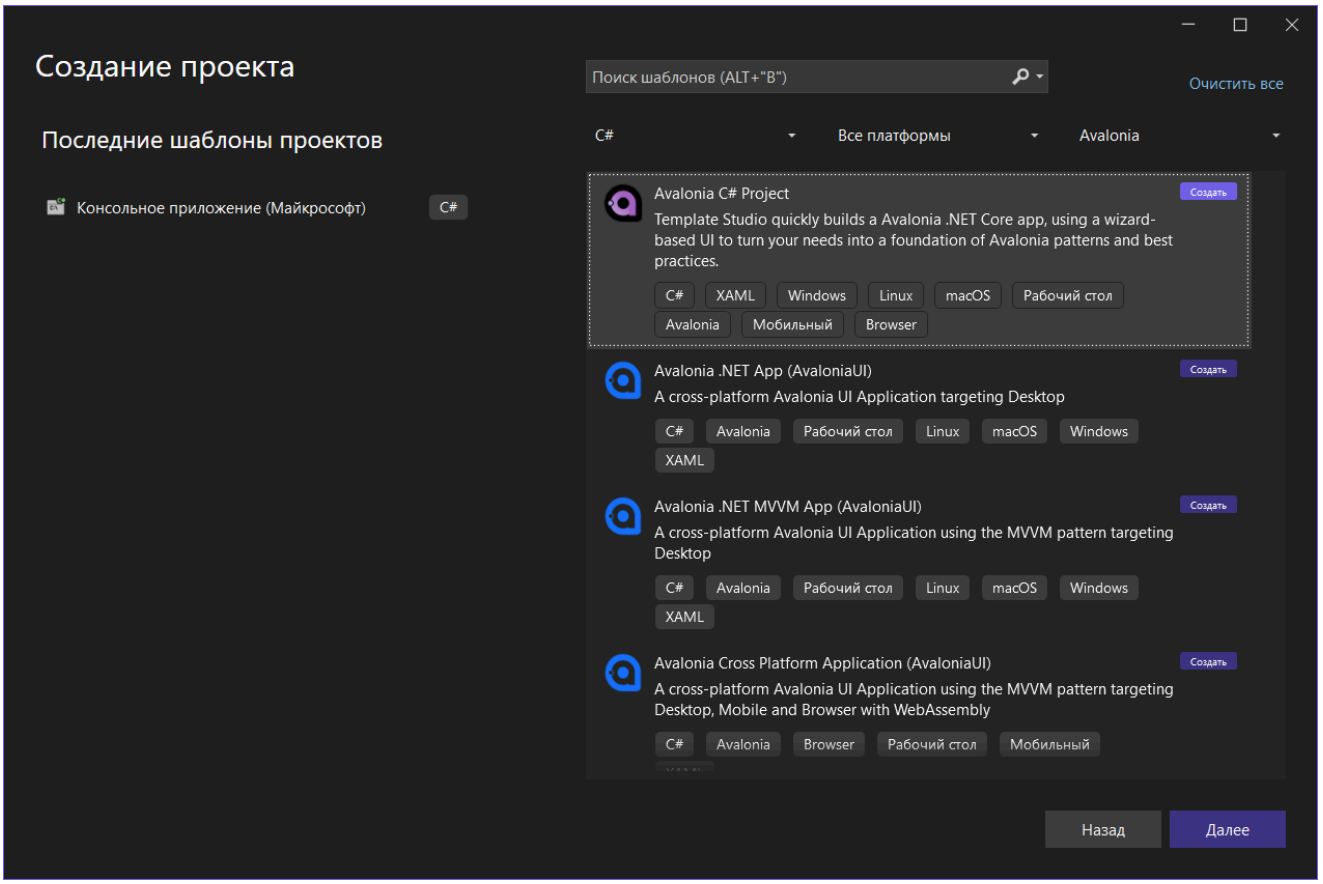


Avalonia

На этих сайтах можно найти информацию по фреймворку и документацию. В этом репозитории собраны примеры кода.

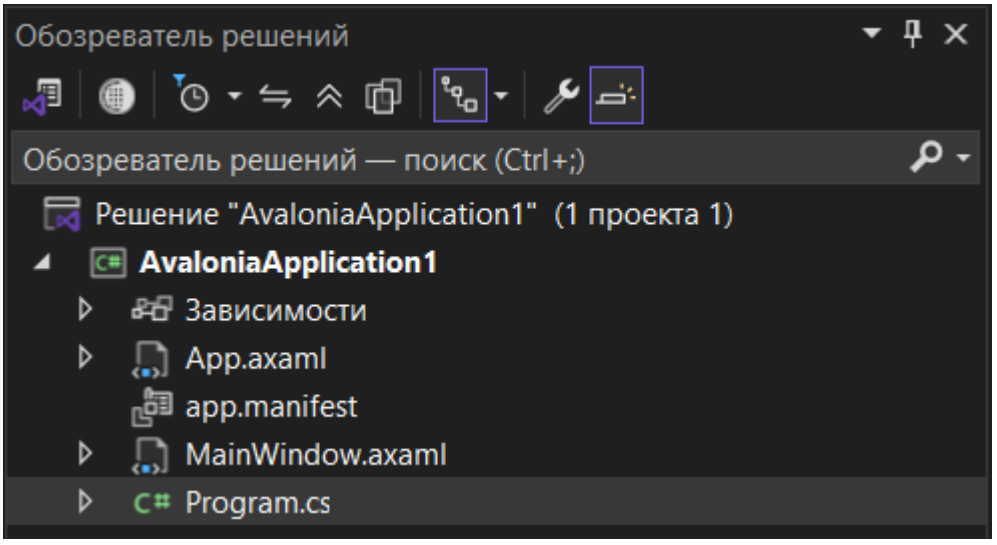
В этой лекции мы подробно рассмотрим только Avalonia. Для того чтобы иметь возможность создавать проекты у удобно работать с Avalonia, нужно установить соответствующее расширение в Visual Studio.

После того как мы установим это расширение и перезапустим Visual Studio, мы получим возможность создавать уже готовый шаблонные проекты на Avalonia.



Avalonia .NET App создаст простой проект, где есть следующие файлы:

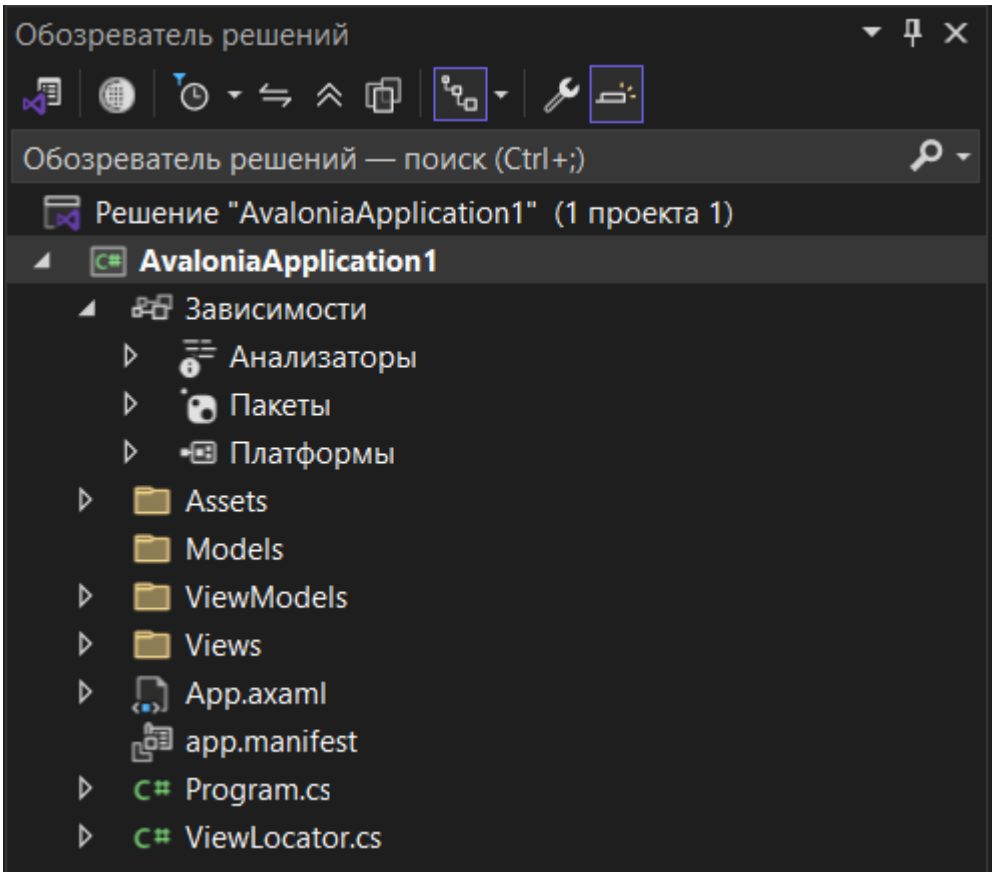
- **App.xaml и App.axaml.cs**
  Здесь задаются глобальные настройки приложения: темы, ресурсы и регистрация сервисов.
- **MainWindow.axaml и MainWindow.axaml.cs**
  Основное окно приложения, где описывается внешний вид в XAML, а также code-behind.



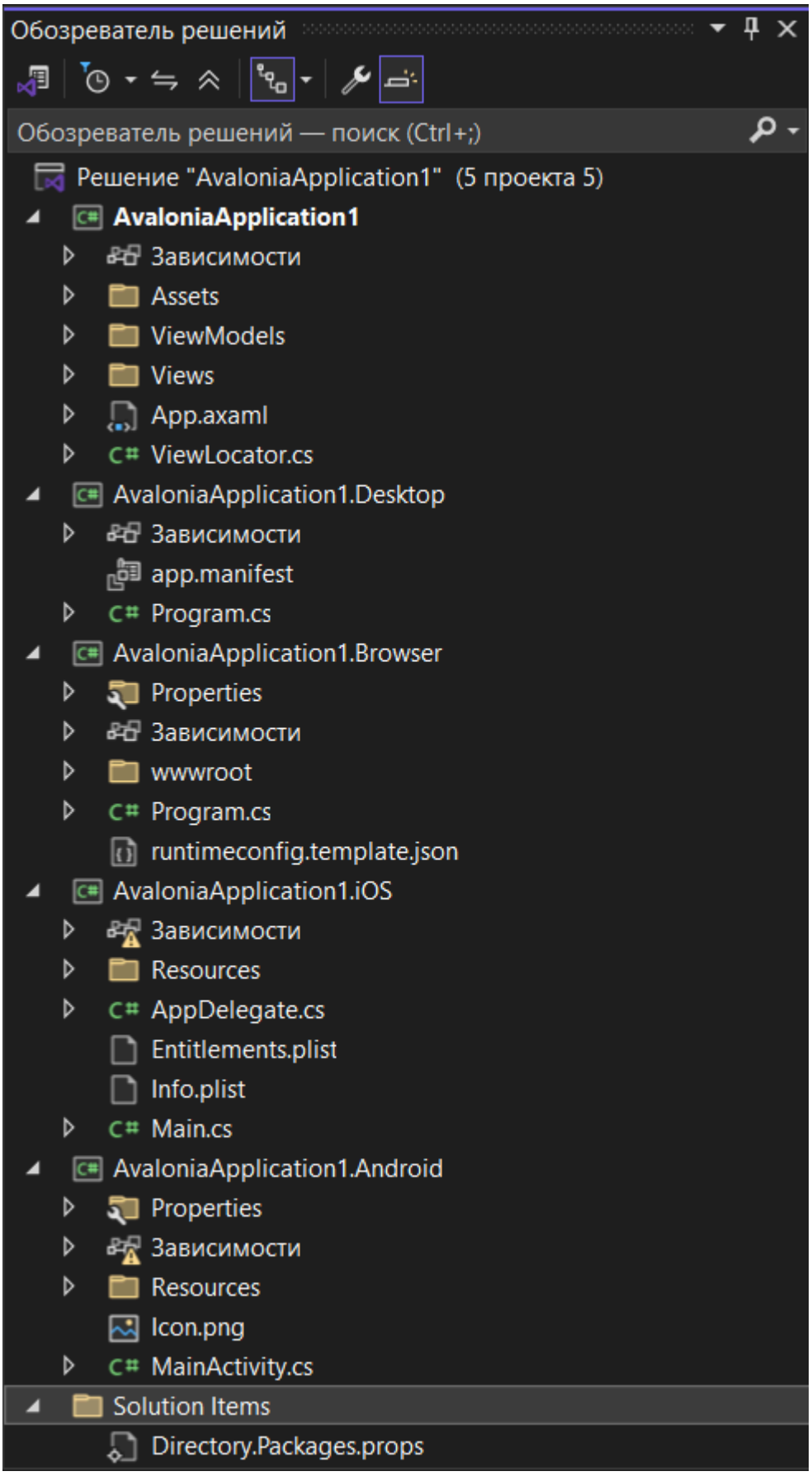
В шаблоне Avalonia .NET MVVM App, код будет разделен в соответствии с паттерном MVVM:

- В папке **Models** находятся классы используемых моделей данных.

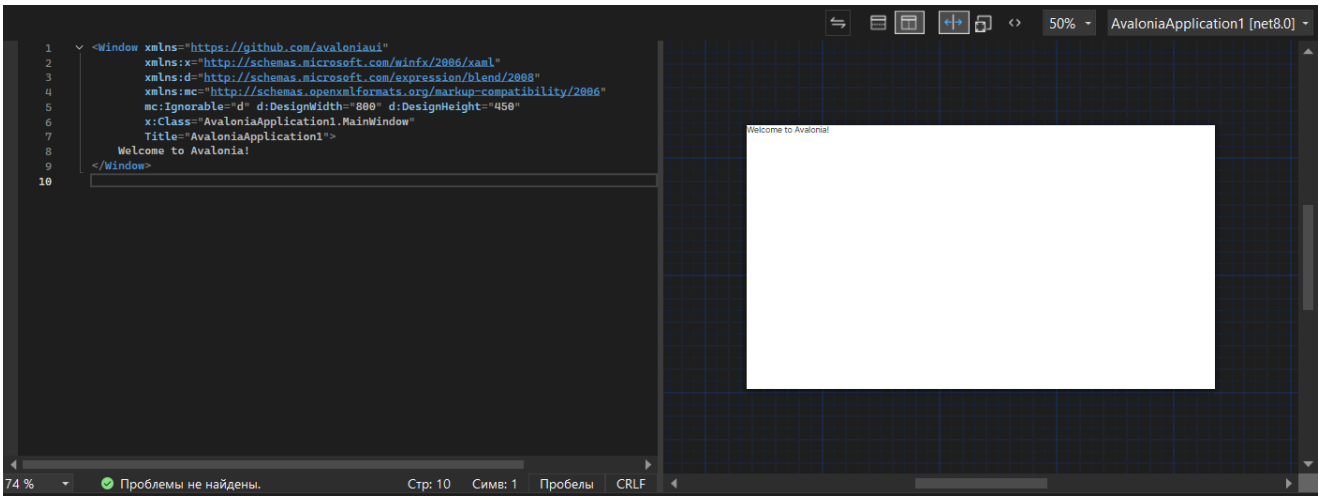
- В папке **Views** находятся файлы представления (например MainWindow.axaml)
- В папке **ViewModel** размещаются классы ViewModel, отвечающие за логику и данные, привязанные к UI.
- В папке **Assets** хранятся дополнительные ресурсы, например иконка оконного приложения или другие изображения.



Шаблон Avalonia Cross Platform Application создаст решение с проектами под каждую платформу (настольное, браузер, мобильные iOS и Android).



При работе с axaml-файлами будет отображаться макет вашего приложения, чтобы можно было оценить как оно будет выглядеть непосредственно во время разработке, без запуска.



AXAML

В то время как стандартное расширение файлов XAML — `.xaml`, в Avalonia UI используется собственное расширение `.axaml` (Avalonia XAML) из-за технических особенностей интеграции с Visual Studio.

Рассмотрим формат axaml-файла:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AvaloniaApplication1.MainWindow">
</Window>
```

Как и в любом XML-документе, в файле есть **корневой элемент**. В данном примере это `<Window></Window>`, который определяет тип корневого элемента. Этот элемент соответствует определённому **контролу** Avalonia UI, в данном случае окну.

Пример выше использует три важных атрибута:

- `xmlns="https://github.com/avaloniaui"` — объявляет пространство имён XAML для *Avalonia UI*. Без него файл не будет распознан как документ Avalonia XAML.
- `xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"` — объявляет пространство имён для языка XAML.
- `x:Class="AvaloniaApplication1.MainWindow"` — указывает XAML-компилятору, где находится связанный класс для данного файла. Этот класс обычно описывается в C#-файле (code-behind).

Вы можете создать пользовательский интерфейс (UI), добавляя XML-элементы, представляющие **элементы управления** Avalonia UI. Имя элемента совпадает с именем класса контрола. Например, следующий код добавляет кнопку в окно:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AvaloniaApplication1.MainWindow">
```

```
    <Button>Hello World!</Button>
</Window>
```

Атрибуты XML-элементов соответствуют свойствам контролов. Свойства можно задать, добавляя атрибуты к элементам.
Например, чтобы задать синий фон для кнопки, добавьте атрибут `Background`:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AvaloniaApplication1.MainWindow">
    <Button Background="Blue">Hello World!</Button>
</Window>
```

Этот код можно написать по другому перенеся содержимое кнопки в атрибут `Content`

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AvaloniaApplication1.MainWindow">
    <Button Background="Blue" Content="Hello World!"/>
</Window>
```

Система привязки данных Avalonia UI позволяет связывать свойства элементов управления с объектами. Это делается с помощью расширения разметки `{Binding}`.
Например:

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AvaloniaApplication1.MainWindow">
    <Button Background="Blue" Content="{Binding Greeting}"/>
</Window>
```

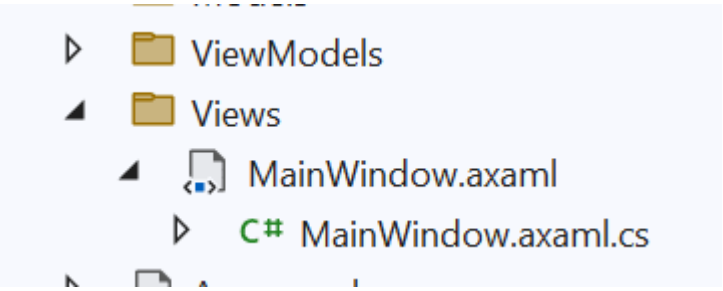
В Avalonia XAML можно объявлять пространства имён, как и в любом XML-документе. Это позволяет процессору XML находить определения элементов. Пространства имён добавляются с помощью атрибута `xmlns`. Для ссылок на код в текущей или подключённой сборке используется префикс `using`. например так подключается пространство с ViewModel для окон.

```
<Window
...
xmlns:vm="using:AvaloniaApplication1.ViewModels"
...>
...
</Window>
```

Code-behind

Code-behind — это файл C#, связанный с XAML-разметкой, который содержит императивную (обычную) логику окна или контрола. Обычно он создаётся автоматически и имеет то же имя, что и axaml, с расширением `.cs`. Его задачи:

- Инициализация компонентов (`InitializeComponent()`).
- Обработка событий напрямую.
- Хранение ссылок на элементы XAML, если указать им `x:Name`.
- При архитектуре MVVM он **обычно пустой** или содержит только инициализацию и привязку ViewModel (то есть как мост между XAML и ViewModel).



Файл code-behind содержит класс с тем же именем, что и у XAML-файла и указано в атрибуте `x:Class` элемента окна. Пример:

```
using Avalonia.Controls;

namespace AvaloniaApplication1.Views
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

При работе с code-behind часто возникает необходимость получить доступ к элементам управления, определенным в XAML. Для этого сначала нужно получить ссылку на нужный элемент управления. Назначьте элементу управления имя с помощью атрибута `Name` (или `x:Name`) в XAML.

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AvaloniaApplication5.MainWindow">
    <Button Name="greetingButton"/>
</Window>
```

Теперь можно получить доступ к кнопке через автоматически созданное поле `greetingButton` из code-behind:

```
using Avalonia.Controls;

namespace AvaloniaApplication1.Views
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            greetingButton.Content = "Hello World";
            greetingButton.Background = Brushes.Blue;
        }
    }
}
```

Полезное приложение требует выполнения каких-либо действий! При использовании подхода code-behind вы пишете обработчики событий в файле code-behind.

Обработчики событий пишутся как методы в файле code-behind и затем указываются в XAML с помощью атрибута события. Например, чтобы добавить обработчик для события клика кнопки:

```
<Window xmlns="https://github.com/avaloniaui"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="AvaloniaApplication1.MainWindow">
    <Button Click="GreetingButtonClickHandler">Hello World</Button>
</Window>
```

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
    }

    public void GreetingButtonClickHandler(object sender, RoutedEventArgs e)
    {
        // код здесь
    }
}
```

ViewModel

Мы рассмотрели как можно связать axaml элементы с кодом C# через code-behind, но если мы разрабатываем приложение с архитектурой MVVM, то представление нужно связывать с соответствующей ViewModel. Именно в этом классе должна располагаться вся логика работы приложения.

Название класса должно выглядеть как `Название_окна+ViewModel.cs`, располагаться он должен в соответствующем файле в папке ViewModels. Напишем код для `MainWindowViewModel`

```
using CommunityToolkit.Mvvm.ComponentModel;

namespace AvaloniaApplication.ViewModels
{
    public partial class MainWindowViewModel : ObservableObject
    {
    //...
    }
}
```

Теперь в `MainWindow.axaml` нужно указать ViewModel для этого представления.

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="using:AvaloniaApplication.ViewModels"
        x:Class="AvaloniaApplication.Views.MainWindow"
        x:DataType="vm:MainWindowViewModel">
        ...
</Window>
```

- С помощь `xmlns:vm="using:AvaloniaApplication.ViewModels"` создаем пространство имен, в котором находится `MainWindowViewModel`.
- Так указываем конкретный класс `x:DataType="vm:MainWindowViewModel"`.

Теперь нужно установить связь между ними. Для этого в `MainWindow.axaml.cs` нужно настроить `DataContext`.

```
using Avalonia.Controls;
using AvaloniaApplication.ViewModels;

namespace AvaloniaApplication.Views;

public partial class MainWindow : Window
{

```

```
    public MainWindow()
    {
        InitializeComponent();
        // Настраиваем ViewModel
        DataContext = new MainWindowViewModel();
    }
}
```

Но если вы создали проект из шаблона, то там будет класс `ViewLocator`, который сам будет находить связанные View и ViewModel и настраивать DataContext, поэтому последний шаг можно пропустить.

Теперь мы можем связывать значения из `MainWindow.axaml` с `MainWindowViewModel.cs`.

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:vm="using:AvaloniaApplication.ViewModels"
        x:Class="AvaloniaApplication.Views.MainWindow"
        x:DataType="vm:MainWindowViewModel">
  <StackPanel Margin="20" Spacing="10">
    <TextBlock Text="{Binding Message}" FontSize="16"/>
    <Button Content="Нажми меня" Command="{Binding ButtonClickCommand}" />
  </StackPanel>
</Window>
```

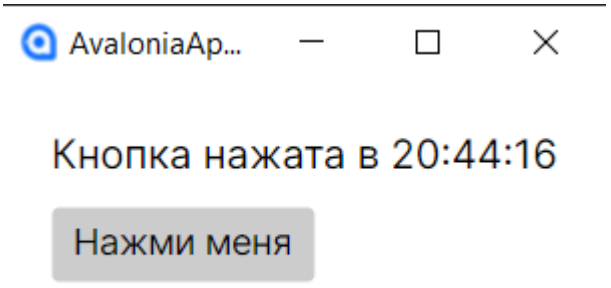
- `Text="{Binding Message}"` свяжет содержимое текстового блока с полем `Message`.
- `Command="{Binding ButtonClickCommand}"` свяжет с обработчиком нажатия на кнопу.

Код в `MainWindowViewModel.cs`.

```
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;
using System;

namespace AvaloniaApplication.ViewModels
{
    public partial class MainWindowViewModel : ObservableObject
    {
        // Автоматическая генерация свойства Message и вызов уведомлений об
изменении
        [ObservableProperty]
        private string message = "Начальное сообщение";

        // Автоматическая генерация команды ButtonClickedCommand
        // Префикс On говорит, что метод OnButtonClick будет вызываться при
```

```
выполнении команды.
        [RelayCommand]
        private void OnButtonClick()
        {
            Message = $"Кнопка нажата в {DateTime.Now:T}";
        }
    }
}
```

В итоге получим простое приложение.



Переключение между страницам

Рассмотрим как сделать оконное приложение с несколькими страницами. Создадим меню в верхней части окна, в котором можно выбрать страницу. Для вывода текущей страницы будем использовать `ContentControl`.

```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:views="clr-namespace:AvaloniaApplication.Views"
        xmlns:vm="clr-namespace:AvaloniaApplication.ViewModels"
        x:DataType="vm:MainWindowViewModel"
        x:Class="AvaloniaApplication.Views.MainWindow"
        Title="Multi Pages"
        Width="800"
        Height="600">

  <Design.DataContext>
    <vm:MainWindowViewModel/>
  </Design.DataContext>

  <DockPanel>
    <!-- Меню -->
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="_First" Command="{Binding
SwitchToFirstPageCommand}"/>
      <MenuItem Header="_Second" Command="{Binding
SwitchToSecondPageCommand}"/>
    </Menu>
```

```
    <!-- Контент страницы -->
    <ContentControl Content="{Binding CurrentPage}"/>
  </DockPanel>
</Window>
```

Создадим связанные поля и события в MainWindowViewModel.

```
using Avalonia.Controls;
using AvaloniaApplication.Views;
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;

namespace AvaloniaApplication.ViewModels;

public partial class MainWindowViewModel : ObservableObject
{
    // Текущая отображаемая страница
    [ObservableProperty]
    private UserControl _currentPage;

    // Команды переключения страниц
    [RelayCommand]
    private void SwitchToFirstPage() => CurrentPage = new FirstPageView();

    [RelayCommand]
    private void SwitchToSecondPage() => CurrentPage = new SecondPageView();
}
```

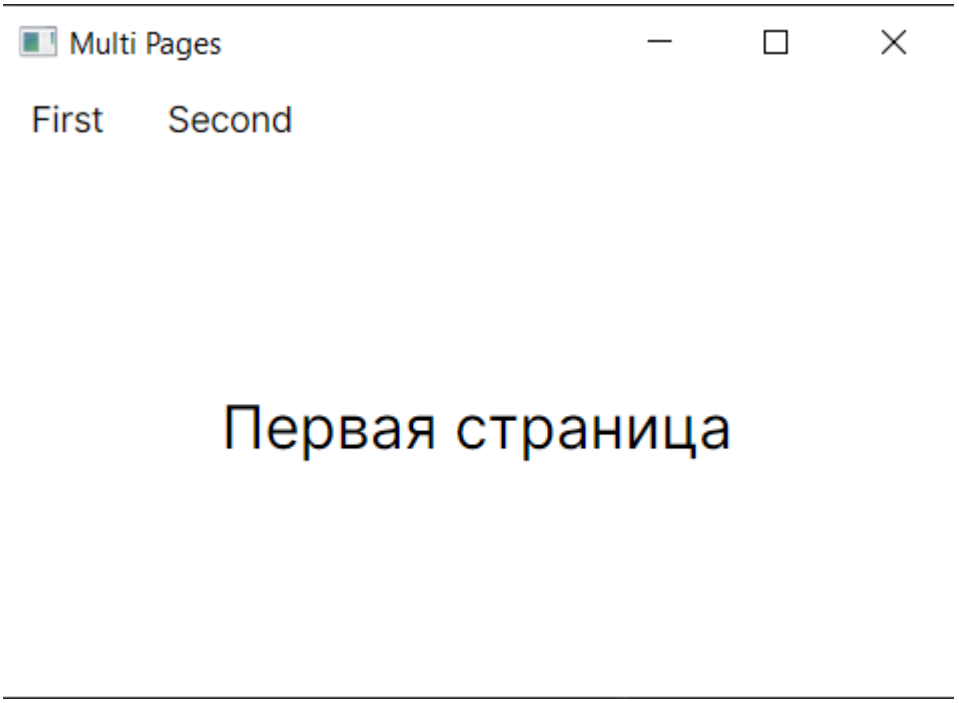
Создадим эти представления. Важно заметить, что вместо `Window` используется `UserControl`.

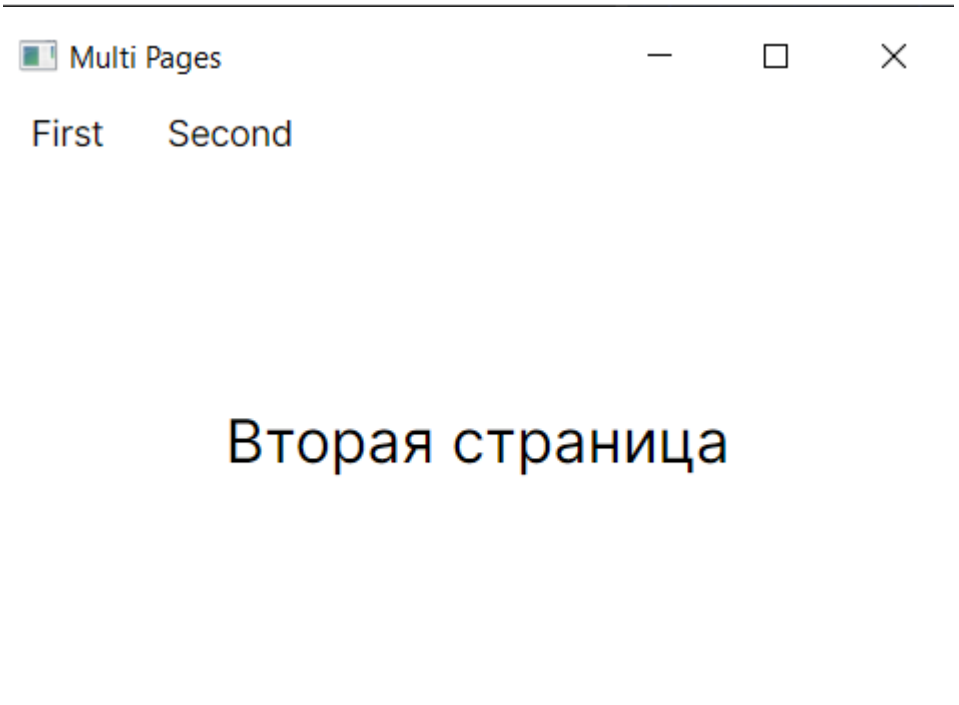
```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="AvaloniaApplication.Views.FirstPageView">
  <TextBlock Text="Первая страница"
             FontSize="24"
             HorizontalAlignment="Center"
             VerticalAlignment="Center"/>
</UserControl>
```

```
<UserControl xmlns="https://github.com/avaloniaui"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             x:Class="AvaloniaApplication.Views.SecondPageView">
  <TextBlock Text="Вторая страница"
             FontSize="24"
             HorizontalAlignment="Center"
```

```
            VerticalAlignment="Center"/>
</UserControl>
```

Теперь мы можем переключать страницы с помощью кнопок в меню. Если нужно будет реализовать более сложную логику для страниц, для каждой из них нужно будет создать свой класс ViewModel.





Вывод коллекции элементов

Давайте теперь научимся выводить список каких-нибудь элементов. Для этого нам понадобится использовать контрол `ItemsControl`, который мы свяжем с коллекцией. Так же нужно будет создать шаблон для отдельной записи. Для удобства так же добавим промотку, когда элементов слишком много. Элементы будем вводить с помощью текстового поля и кнопки "Добавить".

Код `MainWindow`.

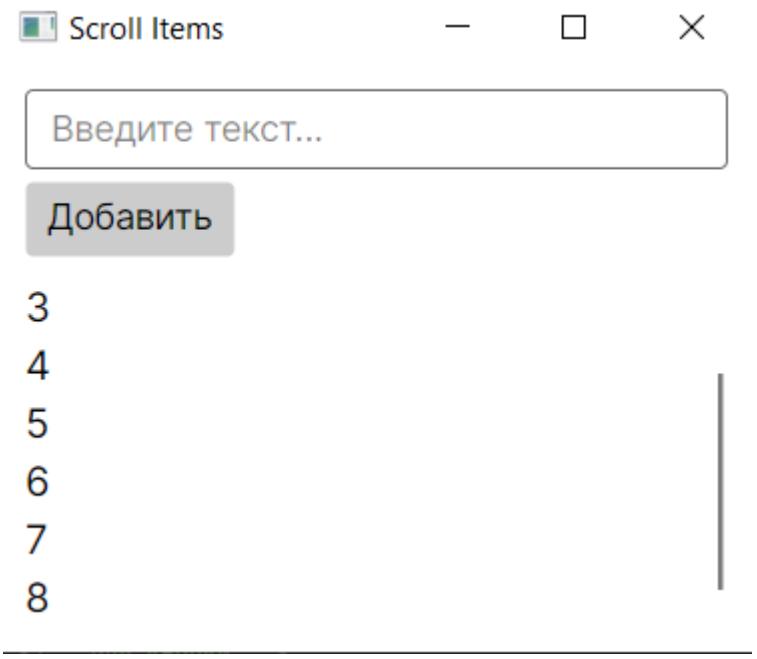
```
<Window xmlns="https://github.com/avaloniaui"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:views="clr-namespace:AvaloniaApplication.Views"
        xmlns:vm="clr-namespace:AvaloniaApplication.ViewModels"
        x:DataType="vm:MainWindowViewModel"
        x:Class="AvaloniaApplication.Views.MainWindow"
        Title="Scroll Items"
        Width="800"
        Height="600">

  <Design.DataContext>
    <vm:MainWindowViewModel/>
  </Design.DataContext>

  <Grid Margin="10">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <!-- Для полей ввода -->
      <RowDefinition Height="Auto"/>
      <!-- Для кнопки -->
      <RowDefinition Height="*"/>
      <!-- Для списка -->
    </Grid.RowDefinitions>

    <!-- Поле ввода -->
    <TextBox Grid.Row="0"
             Watermark="Введите текст..."
             Text="{Binding InputText, Mode=TwoWay}"/>

    <!-- Кнопка добавления -->
    <Button Grid.Row="1"
            Content="Добавить"
            Command="{Binding AddItemCommand}"
            Margin="0 5"/>

    <!-- Список записей с прокруткой -->
    <ScrollViewer Grid.Row="2"
                  VerticalScrollBarVisibility="Auto">
      <ItemsControl ItemsSource="{Binding Items}">
        <ItemsControl.ItemTemplate>
          <DataTemplate>
            <TextBlock Text="{Binding}"
                       Margin="0 2"
                       FontSize="16"/>
          </DataTemplate>
        </ItemsControl.ItemTemplate>
      </ItemsControl>
```

```
    </ScrollViewer>
  </Grid>
</Window>
```

Код для `MainWindowViewModel`.

```
using CommunityToolkit.Mvvm.ComponentModel;
using CommunityToolkit.Mvvm.Input;
using System.Collections.ObjectModel;

namespace AvaloniaApplication.ViewModels;

public partial class MainWindowViewModel : ObservableObject
{
    // Текст в поле ввода
    [ObservableProperty]
    private string _inputText = string.Empty;

    // Коллекция записей
    public ObservableCollection<string> Items { get; } = new();

    // Команда добавления
    [RelayCommand]
    private void AddItem()
    {
        if (!string.IsNullOrWhiteSpace(InputText))
        {
            Items.Add(InputText);
            InputText = string.Empty; // Очищаем поле ввода
        }
    }
}
```

В результате получим такое приложение.



Контролы

Эти контролы составляют базовый набор для создания пользовательских интерфейсов и покрывают большинство общих сценариев разработки настольных приложений:

- **Window**
  Основной контейнер для приложения, представляющий окно. Он служит корневым элементом для других элементов управления.
- **UserControl**
  Контейнер для создания повторно используемых блоков интерфейса. Позволяет инкапсулировать разметку и логику в один компонент.
- **ContentControl**
  Контрол, предназначенный для отображения одного дочернего элемента или содержимого. Его наследники (например, Button и GroupBox) используют подобную модель.
- **Button**
  Стандартная кнопка, используемая для вызова действий или команд.
- **TextBlock**
  Контрол для отображения текстовой информации. Чаще всего используется для вывода надписей, заголовков или сообщений.
- **TextBox**
  Элемент ввода текста, позволяющий пользователю вводить и редактировать строки.
- **CheckBox**
  Переключатель, позволяющий выбирать или отменять выбор (да/нет).
- **RadioButton**
  Контрол для выбора одного варианта из группы. Обычно используется, когда необходимо выбрать только один пункт из нескольких.

- **ComboBox**
  Выпадающий список, позволяющий выбрать элемент из предложенного набора данных.
- **ListBox**
  Контрол для отображения списка элементов с возможностью выбора одного или нескольких элементов.
- **ListView**
  Более функциональный список, позволяющий реализовывать различные шаблоны представления для элементов списка (например, с изображениями и дополнительным текстом).
- **DataGrid**
  Табличный контрол, который используется для отображения и редактирования данных в виде таблицы с колонками и строками.
- **TreeView**
  Контрол для отображения иерархических (деревовидных) данных, где элементы можно сворачивать и разворачивать.
- **ItemsControl**
  Универсальный контрол для отображения коллекций данных. Предоставляет базовую функциональность для рендеринга набора элементов с помощью шаблонов.
- **ScrollViewer**
  Контрол для организации прокрутки содержимого. Обычно применяется в случае, если вложенные элементы не помещаются на экране.
- **Slider**
  Элемент для выбора числового значения, представляющий собой ползунок с возможностью перемещения.
- **ProgressBar**
  Контрол, отображающий прогресс выполнения процесса в виде полоски заполнения.
- **Expander**
  Контрол, позволяющий сворачивать и разворачивать дополнительное содержимое, экономя место на экране.
- **TabControl** и **TabItem**
  Используются для создания интерфейсов с вкладками, где каждый TabItem является отдельной страницей с собственным содержимым.
- **Menu** и **MenuItem**
  Элементы для создания всплывающих или постоянных меню, типичных для оконных приложений.
- **Border**
  Контейнер, позволяющий добавлять рамки, отступы и фон вокруг вложенных элементов.

- **Panels (Grid, StackPanel, WrapPanel, DockPanel, Canvas)**
  Контейнеры, отвечающие за компоновку элементов на экране. Grid позволяет строить интерфейс в виде таблицы, StackPanel — упорядочивать элементы по вертикали или горизонтали, WrapPanel — автоматически переносить элементы, DockPanel — прикреплять элементы к краям, а Canvas — задавать абсолютное позиционирование.

Более подробную информацию и примеры ищите на сайте с документацией.