Asp.net core Web API микросервисы. Часть 1

WPF



# На этом уроке

На предыдущем уроке мы научились создавать автоматически клиентов для сервисов по сгенерированной документации по спецификации Open API. На этом уроке мы рассмотрим, как можно получать данные из микросервисов и использовать их не только для общения между микросервисами, а еще и для представления данных в графическом виде для пользователя. В качестве технологии представления данных возьмем WPF, с которым познакомимся подробнее в рамках этого урока.

Оглавление

[На этом уроке](#_4wfp8dsrmxhk)

[Вступление](#_n93g6yblf23c)

[Паттерн MVVM](#_nd9hodifih90)

[WPF](#_lm2421jacfq0)

[Markup Language (язык разметки XAML)](#_i12juw1tlwdg)

[CodeBehind](#_yxvekws9spek)

[Компоненты и события](#_pmpxhaguuk5b)

[Верстка](#_2e5ebnbkkhy4)

[Стили](#_ysd0nejz0xim)

[Ресурсы](#_alhw7mtdgwdw)

[Привязка данных](#_lbcfb2m0npfr)

[Свои компоненты](#_fv2tbkyk6dj7)

[Создание нового проекта WPF](#_tf1198bb3721)

[Добавление контролов](#_xkfxqmqkorq7)

[Практическое задание](#_c1qudcunbvzw)

[Список используемой литературы](#_54hssoocdbld)

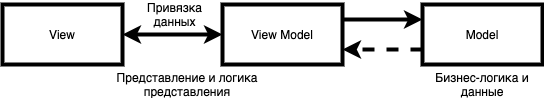
[Дополнительные материалы](#_oreyj64p9zko)

# Вступление

Представление данных в графическом виде является очень частой задачей. Когда мы, как программисты, делаем сервисы, данные из наших сервисов рано или поздно попадут к пользователю в виде графических интерфейсов. В нашем приложении сбора метрик мы уже сделали абсолютно весь бекенд и умеем собирать все метрики, хранить их и отдавать в нужном виде. Осталось собрать некоторую контроль-панель для нашего приложения сбора метрик с графическим пользовательским интерфейсом. Мы разработаем небольшое клиентское приложение, которое будет отображать данные из нашего менеджера сбора метрик.

# Паттерн MVVM

Паттерн MVVM (Model-View-ViewModel) позволяет отделить логику приложения от визуальной части (представления). Данный паттерн является архитектурным, то есть он задает общую архитектуру приложения.



Данный паттерн был представлен Джоном Госсманом в 2005 году как модификация шаблона Presentation Model и был первоначально нацелен на разработку приложений в WPF. И хотя сейчас данный паттерн вышел за пределы WPF и применяется в самых различных технологиях, в том числе при разработке под Android, iOS, тем не менее, WPF является довольно показательной технологией, которая раскрывает возможности данного паттерна.

MVVM состоит из трех компонентов: модели (Model), модели представления (ViewModel) и представления (View).

ViewModel не может общаться со View напрямую. Вместо этого она представляет легко связываемые свойства и методы в виде команд. View может привязываться к этим свойствам, чтобы получать информацию из ViewModel и вызывать на ней команды (методы). Это не требует того, чтобы View знала о ViewModel. XAML Databinding использует рефлексию, чтобы связать View и ViewModel. Таким образом, вы можете использовать любую ViewModel для View, которая предоставляет нужные свойства.

Модель описывает используемые в приложении данные. Модели могут содержать логику, непосредственно связанную этими данными. Модель не должна содержать никакой логики, связанной с отображением данных и взаимодействием с визуальными элементами управления.

Модель реализует интерфейсы INotifyPropertyChanged или INotifyCollectionChanged, которые позволяют уведомлять систему об изменениях свойств модели. Благодаря этому облегчается привязка к представлению, хотя прямое взаимодействие между моделью и представлением отсутствует.

# WPF

WPF — windows presentation foundation — это технология построения графических интерфейсов для приложений под операционную систему windows. WPF позволяет разрабатывать приложение, используя как разметку, и код на управляемом языке для реализации логики.

* Затраты на разработку и обслуживание снижаются, потому что разметка, специфичная для внешнего вида, не тесно связана с кодом, специфичным для поведения.
* Разработка более эффективна, потому что дизайнеры могут реализовать внешний вид приложения одновременно с разработчиками, которые реализуют поведение приложения.
* Глобализация и локализация приложений WPF упрощены.

## Markup Language (язык разметки XAML)

XAML - это язык разметки на основе XML, который декларативно реализует внешний вид приложения. Обычно вы используете его для определения окон, диалоговых окон, страниц и пользовательских элементов управления, а также для заполнения их элементами управления, фигурами и графикой.

В следующем примере XAML используется для реализации внешнего вида окна, содержащего одну кнопку:

|  |
| --- |
| <Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" Title="Окно с кнопочкой" Width="250" Height="100">  *<!-- Добавление кнопульки в окно -->*  <Button Name="button">Пожмакай меня!</Button>  </Window> |

Этот XAML определяет окно и кнопку с помощью элементов Window и Button. Каждый элемент конфигурируется с атрибутами, такими, как атрибут Title элемента Window, чтобы указать текст строки заголовка окна. Во время выполнения WPF преобразует элементы и атрибуты, определенные в разметке, в экземпляры классов WPF. Например, элемент Window преобразуется в экземпляр класса Window, свойство Title которого является значением атрибута Title.

Поскольку XAML основан на XML, пользовательский интерфейс, который вы составляете с его помощью, собран в иерархию вложенных элементов, известную как дерево элементов. Дерево элементов обеспечивает логичный и интуитивно понятный способ создания пользовательских интерфейсов и управления ими.

## CodeBehind

Основное назначение приложения - реализовать функциональность, которая реагирует на действия пользователя. Например, щелчок по меню или кнопке и вызов бизнес-логики и логики доступа к данным в ответ. В WPF это поведение реализовано в коде, связанном с разметкой. Этот тип кода известен как код программной части. В следующем примере показана обновленная разметка из предыдущего примера и код программной части:

|  |
| --- |
| <Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" x:Class="SDKSample.AWindow" Title="Окно с кнопкой" Width="250" Height="100">   <Button Name="button" Click="button\_Click">Пожамкай меня!</Button>  </Window> |

Обновленная разметка определяет пространство имен xmlns: x и сопоставляет его со схемой, которая добавляет поддержку типов кода программной части. Атрибут x: Class используется для связывания класса кода программной части с этой конкретной разметкой XAML. Учитывая, что этот атрибут объявлен в элементе <Window>, класс кода программной части должен наследовать от класса Window.

|  |
| --- |
| using System.Windows;  namespace Sample {  public partial class AWindow : Window  {  public AWindow()  {  InitializeComponent();  }   void button\_Click(object sender, RoutedEventArgs e)  {  *// Открывает мессадж-бокс и показывает сообщеньку.*  MessageBox.Show("Привет, Windows Presentation Foundation!");  }  } } |

InitializeComponent вызывается из конструктора класса для связки автоматически-сгенерированного кода построения ui и code-behind файла. (InitializeComponent создается для вас при построении приложения, поэтому вам не нужно реализовывать его вручную.) Комбинация x: Class и InitializeComponent гарантирует, что ваша реализация правильно инициализируется всякий раз, когда она создается.

Обратите внимание, что в разметке элемент <Button> определил значение button\_click для атрибута Click. Когда разметка и код программной части инициализированы и работают вместе, событие Click для кнопки автоматически сопоставляется с методом button\_click. При нажатии кнопки вызывается обработчик событий и отображается окно сообщения путем вызова метода System.Windows.MessageBox.Show.

## Компоненты и события

Чаще всего элементы управления обнаруживают действия пользователя и реагируют на них. Система ввода WPF использует как прямые, так и перенаправленные события для поддержки ввода текста, управления фокусом и позиционирования мыши.

Приложения часто имеют сложные входные требования. WPF предоставляет систему команд, которая отделяет действия, вводимые пользователем, от кода, который реагирует на эти действия. Система команд позволяет нескольким источникам вызывать одну и ту же логику команд. Например, возьмем общие операции редактирования, используемые различными приложениями: копирование, вырезание и вставка. Эти операции можно вызывать с помощью различных действий пользователя, если они реализованы с помощью команд.

## Верстка

Создавая пользовательский интерфейс, вы располагаете элементы управления по расположению и размеру, чтобы сформировать разметку. Ключевым требованием любой разметки является адаптация к изменениям размера окна и настроек отображения. Вместо того, чтобы заставлять вас писать код для адаптации разметки в этих обстоятельствах, WPF предоставляет вам расширенную систему разметки.

Основой системы компоновки является относительное позиционирование, которое увеличивает возможность адаптации к изменяющимся размерам окна и отображения. Система разметки также управляет согласованием между элементами управления для определения макета. Согласование - это двухэтапный процесс: сначала элемент управления сообщает своему родительскому элементу, какое расположение и размер ему требуется. Затем родительский элемент сообщает элементу управления, какое пространство у него может быть.

Пример верстки с несколькими полями ввода, объединенными компонентами dockPanel:

|  |
| --- |
| <Window  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.LayoutWindow"  Title="Layout with the DockPanel" Height="143" Width="319">    *<!--DockPanel to layout four text boxes-->*   <DockPanel>  <TextBox DockPanel.Dock="Top">Dock = "Top"</TextBox>  <TextBox DockPanel.Dock="Bottom">Dock = "Bottom"</TextBox>  <TextBox DockPanel.Dock="Left">Dock = "Left"</TextBox>  <TextBox Background="White">This TextBox "fills" the remaining space.</TextBox>  </DockPanel>  </Window> |

DockPanel позволяет дочерним элементам управления TextBox сообщать ему, как их расположить. Для этого DockPanel реализует присоединенное свойство Dock, которое предоставляется дочерним элементам управления, чтобы позволить каждому из них указать стиль закрепления.

## Стили

Стили позволяют разработчикам и дизайнерам стандартизировать внешний вид своего продукта. WPF предоставляет сильную стилевую модель, основу которой составляет элемент Style. Стили могут применять значения свойств к типам. Их можно автоматически применять ко всему в соответствии с типом или к отдельным объектам при ссылке.

В следующем примере создается стиль, который устанавливает оранжевый цвет фона для каждой кнопки в окне:

|  |
| --- |
| <Window  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.StyleWindow"  Title="Styles">   <Window.Resources>  *<!-- Style that will be applied to all buttons for this window -->*  <Style TargetType="{x:Type Button}">  <Setter Property="Background" Value="Orange" />  <Setter Property="BorderBrush" Value="Crimson" />  <Setter Property="FontSize" Value="20" />  <Setter Property="FontWeight" Value="Bold" />  <Setter Property="Margin" Value="5" />  </Style>  </Window.Resources>  <StackPanel>   *<!-- This button will have the style applied to it -->*  <Button>Click Me!</Button>   *<!-- This label will not have the style applied to it -->*  <Label>Don't Click Me!</Label>   *<!-- This button will have the style applied to it -->*  <Button>Click Me!</Button>    </StackPanel> </Window> |

Поскольку этот стиль нацелен на все элементы управления Button, он автоматически применяется ко всем кнопкам в окне.

## Ресурсы

Элементы управления в приложении должны иметь одинаковый внешний вид, который может включать в себя все, от шрифтов и цветов фона до шаблонов элементов управления, шаблонов данных и стилей. Для этого можно использовать поддержку WPF для ресурсов пользовательского интерфейса, чтобы инкапсулировать эти ресурсы в одном месте для повторного использования.

В примере кода ниже декларируется ресурс, который одновременно используется кнопкой и надписью:

|  |
| --- |
| <Window  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.ResourcesWindow"  Title="Resources Window">   *<!-- Define window-scoped background color resource -->*  <Window.Resources>  <SolidColorBrush x:Key="defaultBackground" Color="Red" />  </Window.Resources>   *<!-- Button background is defined by window-scoped resource -->*  <Button Background="{StaticResource defaultBackground}">One Button</Button>   *<!-- Label background is defined by window-scoped resource -->*  <Label Background="{StaticResource defaultBackground}">One Label</Label> </Window> |

## 

## Привязка данных

Большинство приложений созданы, чтобы предоставить пользователям средства для просмотра и редактирования данных. Для приложений WPF работа по хранению и доступу к данным уже обеспечивается множеством различных библиотек доступа к данным .NET, таких как SQL и Entity Framework Core. После доступа к данным и их загрузки в управляемые объекты приложения начинается тяжелая работа для приложений WPF. По сути, это включает в себя две вещи:

* Копирование данных из управляемых объектов в элементы управления, где данные можно отображать и редактировать.
* Обеспечение того, чтобы изменения, внесенные в данные с помощью элементов управления, были скопированы обратно в управляемые объекты.

Чтобы упростить разработку приложений, WPF предоставляет мощный механизм привязки данных для автоматической обработки этих шагов. Основным элементом механизма привязки данных является класс Binding, задача которого - привязать элемент управления (цель привязки) к объекту данных (источнику привязки).

WPF поддерживает прямое объявление привязок в разметке XAML. Вот например, код XAML связывает свойство Text TextBox со свойством Name объекта с помощью синтаксиса XAML «{Binding ...}». Это предполагает наличие объекта данных, установленного в свойстве DataContext окна со свойством Name.

|  |
| --- |
| <Window  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.DataBindingWindow">   *<!-- Bind the TextBox to the data source (TextBox.Text to Person.Name) -->*  <TextBox Name="personNameTextBox" Text="{Binding Path=Name}" />   </Window> |

Механизм привязки данных WPF предоставляет больше, чем просто привязку, он обеспечивает проверку, сортировку, фильтрацию и группировку. Кроме того, привязка данных поддерживает использование шаблонов данных для создания настраиваемого пользовательского интерфейса для привязанных данных.

## Свои компоненты

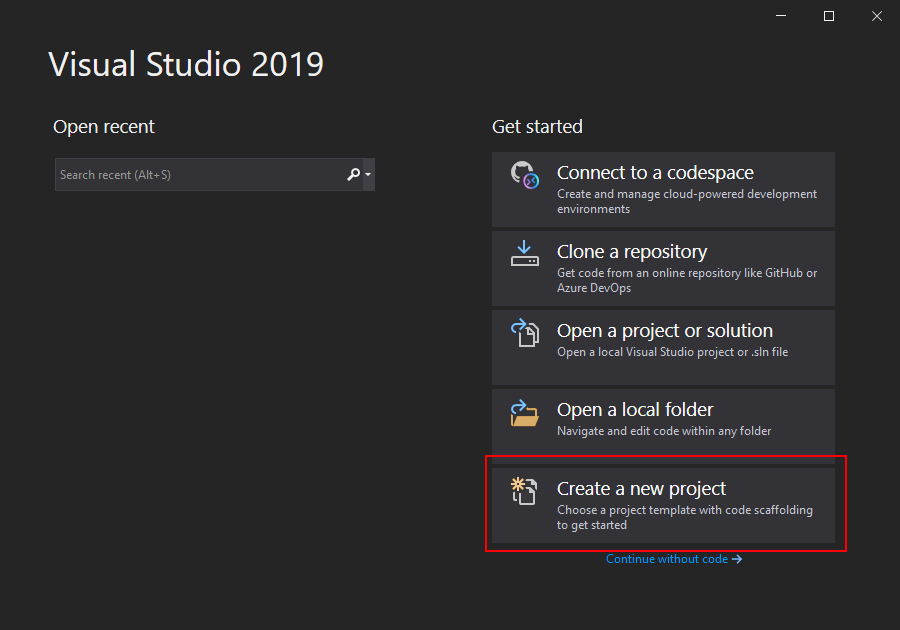
Хотя WPF предоставляет множество возможностей настройки, вы можете столкнуться с ситуациями, когда существующие элементы управления WPF не соответствуют потребностям вашего приложения или его пользователей. Именно для этого и можно создавать свои компоненты.

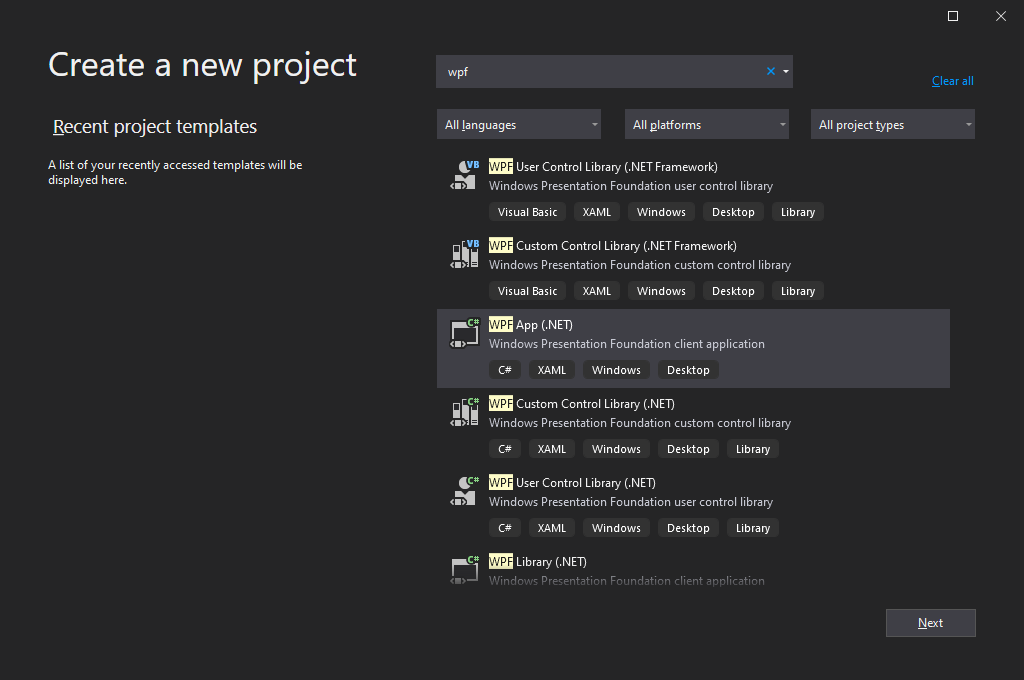
Однако на этом этапе вы можете воспользоваться одной из трех моделей WPF для создания нового элемента управления. Каждая модель нацелена на определенный сценарий и требует, чтобы ваш настраиваемый элемент управления был производным от определенного базового класса WPF. Здесь перечислены три модели:

1. Пользовательский элемент управления является производным от UserControl и состоит из одного или нескольких других элементов управления.
2. Пользовательский элемент управления является производным от Control и используется для создания реализаций, которые отделяют их поведение от внешнего вида с помощью шаблонов, как и большинство элементов управления WPF. Наследование из Control дает вам больше свободы для создания настраиваемого пользовательского интерфейса, чем пользовательские элементы управления, но это может потребовать больше усилий.
3. Пользовательский элемент управления является производным от FrameworkElement, если его внешний вид определяется пользовательской логикой отрисовки (а не шаблонами)

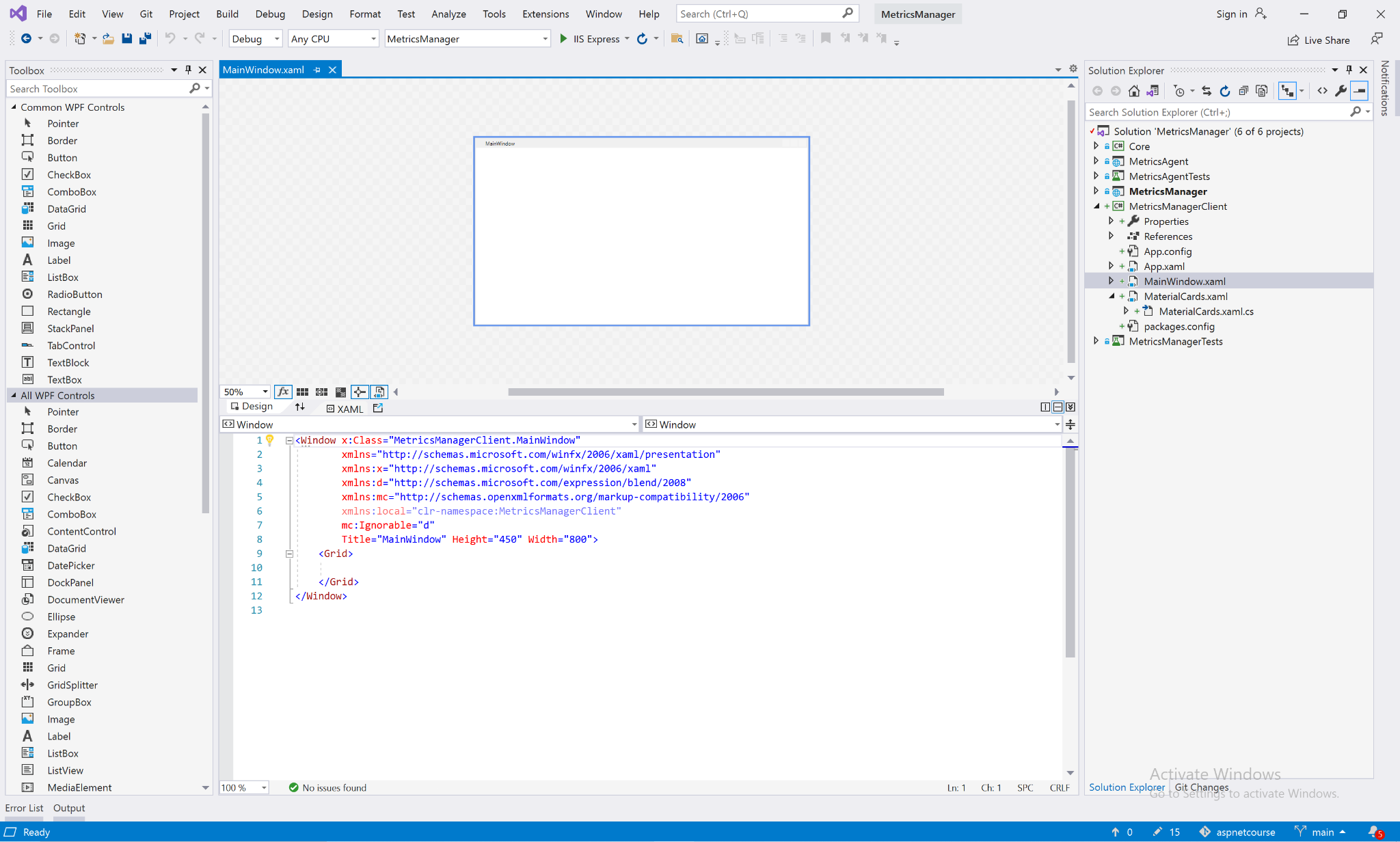
# Создание нового проекта WPF

Теперь мы можем создать новый проект на WPF. Для этого в Visual Studio есть соответствующий шаблон:





После того как вы создадите новый проект перед вами откроется следующее окно:

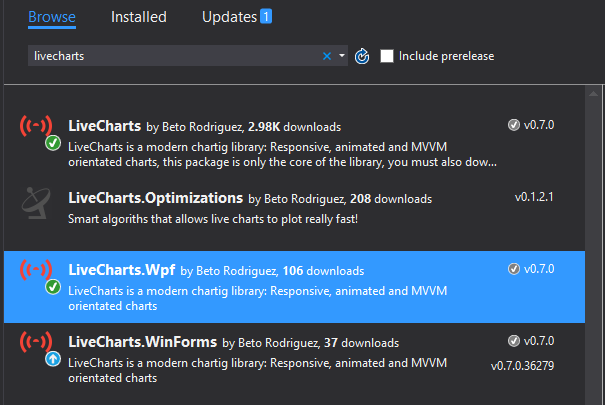


Рассмотрим, что мы видим в нем.

1. Визуальный редактор Xaml Designer. Это дизайнер документа XAML. Он интерактивен, и вы можете перетаскивать объекты из панели инструментов. Выбирая и перемещая элементы в конструкторе, вы можете визуально составить пользовательский интерфейс (UI) для своего приложения. Когда видны и дизайнер, и редактор, изменения в одном отражаются в другом. Когда вы выбираете элементы в конструкторе, на панели «Свойства» отображаются свойства и атрибуты этого объекта.
2. Это редактор кода XAML для документа XAML. Редактор кода XAML - это способ создать пользовательский интерфейс вручную без конструктора. Дизайнер может сделать вывод о значениях свойств элемента управления, когда элемент управления добавлен в конструктор. Редактор кода XAML дает вам гораздо больше контроля.
3. Панель готовых компонентов. На панели перечислены все стандартные компоненты WPF. Компоненты можно перетащить на форму, тем самым установив их к редактированию.
4. Уже знакомый вам Solution Explorer в котором отображена структура нашего решения.

## Добавление контролов

Теперь добавим свой UserControl, в котором мы будем отображать данные, которые пришли к нам от сервиса менеджера-сбора метрик. Мы воспользуемся сторонней библиотекой компонентов, так как в WPF нет стандартного встроенного компонента для отображения диаграмм. Подключите через nuget библиотеку LiveCharts.Wpf



Теперь добавим новый контрол к нашему проекту:

|  |
| --- |
| <UserControl x:Class="MetricsManagerClient.MaterialCards"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"    xmlns:lvc="clr-namespace:LiveCharts.Wpf;assembly=LiveCharts.Wpf"  xmlns:local="clr-namespace:MetricsManagerClient" d:DataContext="{d:DesignInstance Type=local:MaterialCards}"  mc:Ignorable="d"   d:DesignHeight="450" d:DesignWidth="800">  <Grid Height="500" Width="650" >  <Grid.ColumnDefinitions>  <ColumnDefinition></ColumnDefinition>  </Grid.ColumnDefinitions>   <Grid Margin="15, 20, 15, 15" MaxHeight="350">  <Grid.Effect>  <DropShadowEffect BlurRadius="15" Direction="-90" RenderingBias="Quality" Opacity=".2" ShadowDepth="1"/>  </Grid.Effect>  <Grid.OpacityMask>  <VisualBrush Visual="{Binding ElementName=Border2}" />  </Grid.OpacityMask>  <Grid.Resources>  <Style TargetType="lvc:ColumnSeries">  <Setter Property="StrokeThickness" Value="0"></Setter>  <Setter Property="Stroke" Value="White"></Setter>  <Setter Property="Fill" Value="White"></Setter>  <Setter Property="MaxColumnWidth" Value="5"></Setter>  </Style>  <Style TargetType="lvc:Axis">  <Setter Property="FontSize" Value="12"></Setter>  <Setter Property="Foreground" Value="#64FFFFFF"></Setter>  <Style.Triggers>  <Trigger Property="AxisOrientation" Value="Y">  <Setter Property="IsMerged" Value="True"></Setter>  <Setter Property="MaxValue" Value="100"></Setter>  </Trigger>  </Style.Triggers>  </Style>  <Style TargetType="lvc:Separator">  <Setter Property="StrokeThickness" Value="1"></Setter>  <Setter Property="Stroke" Value="#4BFFFFFF"></Setter>  <Style.Triggers>  <Trigger Property="AxisOrientation" Value="X">  <Setter Property="IsEnabled" Value="False"></Setter>  <Setter Property="Step" Value="1"></Setter>  </Trigger>  </Style.Triggers>  </Style>  </Grid.Resources>  <Grid.RowDefinitions>  <RowDefinition Height="Auto"></RowDefinition>  <RowDefinition Height="Auto"></RowDefinition>  <RowDefinition Height=".50\*"></RowDefinition>  <RowDefinition Height=".5\*"></RowDefinition>  </Grid.RowDefinitions>  <Border x:Name="Border2" Grid.Row="0" Grid.RowSpan="4" CornerRadius="5" Background="White" />  <Border Grid.Row="0" Grid.RowSpan="3" Background="#EB5A13" ></Border>  <TextBlock Grid.Row="0" TextAlignment="Center" Padding="10, 10, 0, 5" Foreground="White" FontSize="18">  Cpu Load  </TextBlock>  <Button Grid.Row="2" Width="40" Height="40" VerticalAlignment="Bottom"   HorizontalAlignment="Right" Margin="0,0,20,136" Panel.ZIndex="1"  Click="UpdateOnСlick" Grid.RowSpan="2">  <Button.Template>  <ControlTemplate TargetType="Button">  <Grid>  <Grid.Effect>  <DropShadowEffect BlurRadius="15" Direction="-90" RenderingBias="Quality" Opacity=".2" ShadowDepth="1"/>  </Grid.Effect>  <Ellipse Stroke="Black" StrokeThickness="0" Fill="#CD2156">   </Ellipse>  <Path Width="20" Height="20" Stretch="Fill" Fill="White" Data="F1 M 58,33.5001L 58,27L 49,19L 40,27.5001L 40,33.5001L 46,28.2097L 46,40.5C 46,46.299 41.299,51 35.5,51C 29.701,51 25,46.299 25,40.5C 25,34.8686 29.4332,30.2727 35,30.0117L 35,24.0074C 26.1186,24.2718 19,31.5546 19,40.5C 19,49.6127 26.3873,57 35.5,57C 44.6127,57 52,49.6127 52,40.5L 52,28.125L 58,33.5001 Z "/>  <ContentPresenter HorizontalAlignment="Center"  VerticalAlignment="Center"/>  </Grid>  </ControlTemplate>  </Button.Template>  </Button>  <lvc:CartesianChart Name="TimePowerChart" Grid.Row="2" Margin="10, 0, 10, 20"   Hoverable="True" DataTooltip="{x:Null}"  Series="{Binding ColumnServiesValues}">  </lvc:CartesianChart>  <StackPanel Grid.Row="3" VerticalAlignment="Center" Margin="25, 0">  <TextBlock Opacity=".4" FontSize="13">За последние 12 часов средняя загрузка <LineBreak /> Проценты</TextBlock>  <StackPanel Orientation="Horizontal">  <TextBlock Name="PercentTextBlock" Foreground="#303030" FontSize="40" Text="10" />  <TextBlock Foreground="#303030" FontSize="18" VerticalAlignment="Bottom" Margin="8, 6">%</TextBlock>  </StackPanel>  </StackPanel>  </Grid>  </Grid> </UserControl> |

Данный контрол является переработанным контролом, взятым из примера к коду библиотеки LiveCharts. Рассмотрим code behind файл:

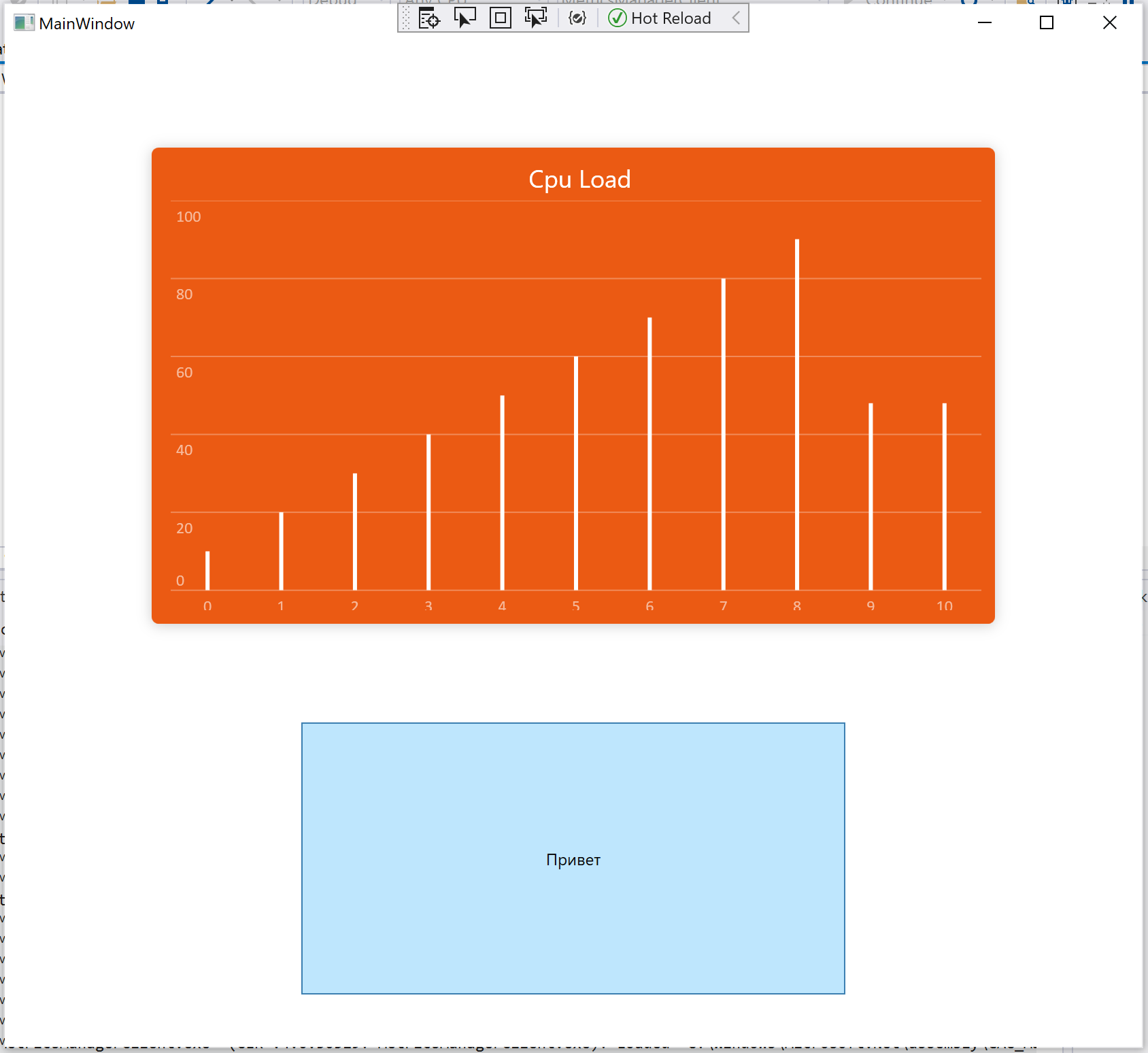
|  |
| --- |
| using LiveCharts; using LiveCharts.Wpf; using System.ComponentModel; using System.Windows; using System.Windows.Controls;  namespace MetricsManagerClient {  */// <summary>*  */// Interaction logic for MaterialCards.xaml*  */// </summary>*  public partial class MaterialCards : UserControl, INotifyPropertyChanged  {   public MaterialCards()  {  InitializeComponent();   ColumnServiesValues = new SeriesCollection  {  new ColumnSeries  {  Values = new ChartValues<double> { 10,20,30,40,50,60,70,80,90.100 }  }  };   DataContext = this;  }   public SeriesCollection ColumnServiesValues { get; set; }   public event PropertyChangedEventHandler PropertyChanged;   protected virtual void OnPropertyChanged(string propertyName = null)  {  var handler = PropertyChanged;  if (handler != null) handler(this, new PropertyChangedEventArgs(propertyName));  }   private void UpdateOnСlick(object sender, RoutedEventArgs e)  {  TimePowerChart.Update(true);  }  } } |

Для примера прямо в коде компонента задан биндинг на данные, которые добавляются в компонент. Теперь давайте сделаем основное окно программы. Для начала приведем код XAML файла главного окна программы:

|  |
| --- |
| <Window x:Class="MetricsManagerClient.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"  xmlns:local="clr-namespace:MetricsManagerClient"  mc:Ignorable="d"  Title="MainWindow" Height="800" Width="1600">  <Grid>  <StackPanel>  <local:MaterialCards Grid.Column="0" Grid.Row="0" x:Name="CpuChart"></local:MaterialCards>  <Button Width="400" Height="200" Click="Button\_Click">Привет</Button>  </StackPanel>  </Grid> </Window> |

А теперь посмотрим, как выглядит запущенная в режиме отладки программа. Попробуйте понажимать кнопку, на график будут добавляться значения. Для этого в Code behind файле сделаем обработчик нажатия на кнопку. Таким образом весь код code behind файла будет выглядеть так:

|  |
| --- |
| using System.Windows;  namespace MetricsManagerClient {  */// <summary>*  */// Interaction logic for MainWindow.xaml*  */// </summary>*  public partial class MainWindow : Window  {  public MainWindow()  {  InitializeComponent();  }   private void Button\_Click(object sender, RoutedEventArgs e)  {  CpuChart.ColumnServiesValues[0].Values.Add(48d);   }  } } |



Теперь мы можем по нажатию на кнопку сделать подгрузку метрик из нашего приложения по сбору метрик. Для этого подключите клиент для сервиса метрик в ваш проект и напишите код по получению данных от сервиса сбора метрик аналогично тому, как вы делали это для случая, когда менеджер метрик забирал данные у агента сбора метрик. Можно воспользоваться кодом добавления полученных значений на график из примера выше.

# Практическое задание

Практическое задание на данном уроке является творческим. Попробуйте поэкспериментировать со стандартными компонентами WPF и при помощи верстки и data bindings добавить данные, которые бы вам интересно было видеть от нашего агента сбора метрик. Например, попробуйте собрать grid с двумя колонками и двумя строками. В каждую ячейку получившейся сетки попробуйте разместить по графику, в которые можно вывести данные по различным метрикам.

# Список используемой литературы

<https://docs.microsoft.com/en-us/dotnet/desktop/wpf>

<https://docs.microsoft.com/en-us/dotnet/desktop/wpf/controls/controls-by-category>

<https://habr.com/ru/company/mobileup/blog/313538/>

<https://habr.com/ru/post/427325/>

<https://github.com/Live-Charts/Live-Charts>

# Дополнительные материалы

<https://lvcharts.net/App/examples/wpf/Install>

<https://docs.microsoft.com/en-us/dotnet/desktop/wpf/data/>

<https://docs.microsoft.com/en-us/dotnet/desktop/wpf/xaml/>