Daj nam znać, co sądzisz o środowisku pobierania plików PDF.

Dokumentacja platformy Windows Presentation Foundation

Dowiedz się więcej o korzystaniu z Windows Presentation Foundation (WPF), graficznego interfejsu użytkownika typu open source dla systemu Windows na platformie .NET 7.

Zapoznanie się z platformą WPF

OMÓWIENIE

Omówienie podsystemu WPF

CO NOWEGO

Różnice w porównaniu z platformą .NET Framework

SAMOUCZEK

Tworzenie nowej aplikacji

Migracja z platformy .NET Framework

POBIERZ

Visual Studio 2022 w wersji 17.4

Okna WPF

OMÓWIENIE

Okna platformy WPF — informacje

Okna dialogowe — informacje

PRZEWODNIK Z INSTRUKCJAMI

Otwieranie okna komunikatu

Otwieranie wspólnego okna dialogowego

Pobieranie lub ustawianie głównego okna aplikacji

Sterowanie stylami i szablonami

OMÓWIENIE

Style i szablony — informacje

PRZEWODNIK Z INSTRUKCJAMI

Tworzenie stylu

Tworzenie szablonu

Powiązanie danych

OMÓWIENIE

Powiązanie danych — informacje

Deklarowanie powiązań

XAML w WPF

OMÓWIENIE

XAML — informacje

Przewodnik dotyczący aplikacji klasycznych (WPF .NET)

Artykuł • 18.02.2023 • Czas czytania: 16 min

Witamy w przewodniku dotyczącym aplikacji klasycznych Windows Presentation Foundation (WPF), struktury interfejsu użytkownika, która jest niezależna od rozdzielczości i używa aparatu renderowania opartego na wektorach, opracowanego w celu korzystania z nowoczesnego sprzętu graficznego. WPF zawiera kompleksowy zestaw funkcji programowania aplikacji, obejmujący między innymi język Extensible Application Markup Language (XAML), kontrolki, powiązanie danych, układ, grafikę 2D i 3D, animację, style, szablony, dokumenty, multimedia, tekst i typografię. Struktura WPF jest częścią platformy .NET, co umożliwia tworzenie aplikacji zawierających inne elementy interfejsu API platformy .NET.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Istnieją dwie implementacje struktury WPF:

1. Wersja dla platformy **.NET** (ten przewodnik):

   Implementacja platformy WPF typu open source hostowana w usłudze GitHub, która działa na platformie .NET. Projektant XAML wymaga programu Visual Studio 2019 co najmniej w wersji 16.8. Jednak w zależności od używanej wersji platformy .NET może być wymagane użycie nowszej wersji programu Visual Studio.

   Mimo że platforma .NET jest technologią międzyplatformową, platforma WPF działa tylko w systemie Windows.

2. Wersja dla platformy **.NET Framework 4**:

   Implementacja struktury WPF dla platformy .NET Framework obsługiwana przez programy Visual Studio 2019 i Visual Studio 2017.

   .NET Framework 4 jest wersją platformy .NET przeznaczoną tylko dla systemu Windows, która jest traktowana jak składnik systemu operacyjnego Windows. Ta wersja struktury WPF jest rozpowszechniana razem z platformą .NET Framework. Aby uzyskać więcej informacji na temat wersji struktury WPF dla platformy .NET

Framework, zobacz Wprowadzenie do struktury WPF dla platformy .NET Framework.

To omówienie jest przeznaczone dla nowych użytkowników, dlatego obejmuje najważniejsze możliwości i pojęcia dotyczące struktury WPF. Aby dowiedzieć się, jak utworzyć aplikację WPF, zobacz Samouczek: Tworzenie nowej aplikacji WPF.

Po co migrować z platformy .NET Framework

Platforma WPF dla platformy .NET udostępnia nowe funkcje i ulepszenia w .NET Framework. Aby dowiedzieć się, jak przeprowadzić migrację aplikacji, zobacz Jak przeprowadzić migrację aplikacji klasycznej WPF na platformę .NET 5.

Program ze strukturą WPF

Struktura WPF istnieje jako podzestaw typów platformy .NET, które znajdują się głównie w przestrzeni nazw System.Windows. Użytkownicy, którzy kompilowali już wcześniej aplikacje na platformie .NET za pomocą takich struktur jak ASP.NET i Windows Forms, powinni już znać podstawowe środowisko programowania WPF:

- Tworzenie wystąpień klas
- Ustawianie właściwości
- Wywoływanie metod
- Obsługa zdarzeń

Struktura WPF zawiera dodatkowe konstrukcje programistyczne, które rozszerzają możliwości właściwości i zdarzeń: właściwości zależności i zdarzenia trasowane.

Znaczniki i plik codebehind

Struktura WPF umożliwia tworzenie aplikacji przy użyciu zarówno *znaczników*, jak i *pliku codebehind*, środowiska, które powinno być znane deweloperom ASP.NET. Na ogół używa się znaczników XAML do implementowania wyglądu aplikacji oraz zarządzanych języków programowania (plik codebehind) do implementowania ich zachowania. Rozdzielenie wyglądu od zachowania przynosi następujące korzyści:

- Zmniejszenie kosztów programowania i konserwacji dzięki temu, że znaczniki specyficzne dla wyglądu nie są ściśle powiązane z kodem specyficznym dla zachowania.

- Programowanie jest bardziej wydajne, ponieważ projektanci mogą implementować wygląd aplikacji jednocześnie z deweloperami, którzy implementują zachowanie aplikacji.
- Uproszczona globalizacja i lokalizacja aplikacji WPF.

Znaczniki

XAML to język znaczników oparty na języku XML, który deklaratywnie implementuje wygląd aplikacji. Zazwyczaj służy do definiowania okien, okien dialogowych, stron i kontrolek użytkownika oraz wypełniania ich kontrolkami, kształtami i grafiką.

W poniższym przykładzie za pomocą języka XAML zaimplementowano wygląd okna zawierającego jeden przycisk:

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button">Click Me!</Button>

</Window>
```

Konkretnie ten kod XAML definiuje okno i przycisk przy użyciu elementów `Window` i `Button`. Każdy element jest skonfigurowany za pomocą atrybutów, takich jak atrybut `Title` elementu `Window` określający tekst na pasku tytułu okna. W czasie wykonywania struktura WPF konwertuje elementy i atrybuty zdefiniowane w znacznikach na wystąpienia klas WPF. Na przykład element `Window` jest konwertowany na wystąpienie klasy Window, której właściwość Title jest wartością atrybutu `Title`.

Na poniższej ilustracji przedstawiono interfejs użytkownika zdefiniowany przez kod XAML w poprzednim przykładzie:

Ponieważ kod XAML jest oparty na języku XML, interfejs użytkownika, który jest z nim komponowany, jest zestawiony w hierarchii zagnieżdżonych elementów, która jest nazywana drzewem elementów. Drzewo elementów zapewnia logiczny i intuicyjny sposób tworzenia interfejsów użytkownika i zarządzania nimi.

Plik codebehind

Głównym zachowaniem aplikacji jest zaimplementowanie funkcji reagującej na interakcje użytkowników. Na przykład kliknięcie menu lub przycisku oraz wywołanie w odpowiedzi logiki biznesowej i logiki dostępu do danych. W ramach struktury WPF to zachowanie jest implementowane w kodzie skojarzonym ze znacznikami. Ten typ kodu nosi nazwę pliku codebehind. W poniższym przykładzie przedstawiono zaktualizowany znacznik z poprzedniego przykładu i plik codebehind:

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

Zaktualizowany znacznik definiuje przestrzeń nazw `xmlns:x` i mapuje ją na schemat dodający obsługę typów pliku codebehind. Do skojarzenia klasy pliku codebehind z tym konkretnym znacznikiem XAML służy atrybut `x:Class`. Biorąc pod uwagę, że ten atrybut jest zadeklarowany w elemecie `<Window>`, klasa pliku codebehind musi dziedziczyć z klasy `Window`.

C#

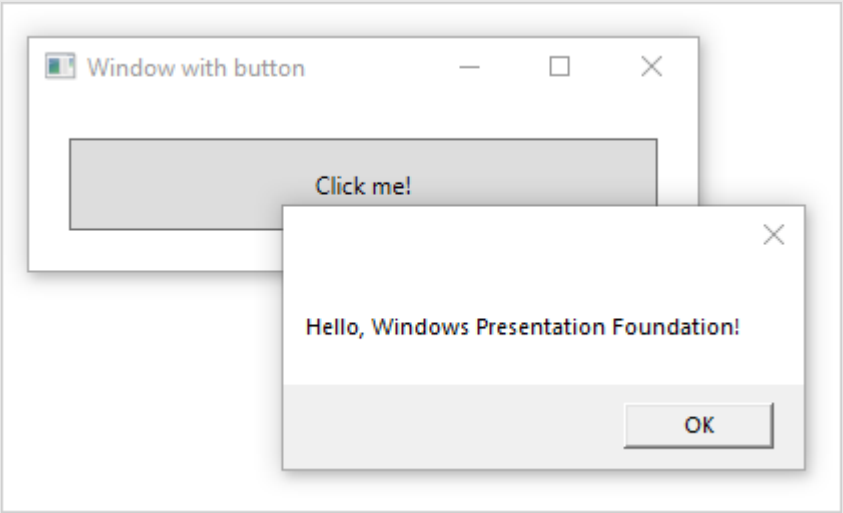
```
using System.Windows;

namespace SDKSample
{
    public partial class AWindow : Window
    {
        public AWindow()
        {
            // InitializeComponent call is required to merge the UI
            // that is defined in markup with this class, including
```

```
            // setting properties and registering event handlers
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}
```

Element `InitializeComponent` jest wywoływany z konstruktora klasy pliku codebehind w celu scalenia interfejsu użytkownika zdefiniowanego w znacznikach z klasą pliku codebehind. (Element `InitializeComponent` jest generowany dla Ciebie podczas kompilowania aplikacji, dlatego nie trzeba implementować go ręcznie). Kombinacja elementów `x:Class` i `InitializeComponent` zapewnia poprawne zainicjowanie implementacji po jej utworzeniu.

Zwróć uwagę, że element `<Button>` w znaczniku zdefiniował wartość `button_click` atrybutu `Click`. Po zainicjowaniu i rozpoczęciu współpracy znacznika i pliku codebehind zdarzenie Click dotyczące przycisku jest automatycznie mapowane na metodę `button_click`. Po kliknięciu przycisku jest wywoływana procedura obsługi zdarzeń, a wywołanie metody System.Windows.MessageBox.Show powoduje wyświetlenie pola komunikatu.

Na poniższej ilustracji przedstawiono rezultat kliknięcia przycisku:



Dane wejściowe i polecenia

Kontrolki najczęściej wykrywają dane wejściowe użytkownika i reagują na nie. System wprowadzania danych struktury WPF używa zarówno zdarzeń bezpośrednich, jak i

trasowanych do obsługi wprowadzania tekstu, zarządzania fokusem i pozycjonowania myszy.

Aplikacje często mają złożone wymagania dotyczące danych wejściowych. Struktura WPF udostępnia system poleceń, który oddziela akcje wprowadzania danych przez użytkownika od kodu, który reaguje na te akcje. System poleceń umożliwia wywoływanie tej samej logiki poleceń przez wiele źródeł. Weźmy na przykład typowe operacje edycji używane przez różne aplikacje: **kopiowanie**, **wycinanie** i **wklejanie**. Te operacje można wywoływać przy użyciu różnych akcji użytkownika, jeśli są one zaimplementowane przy użyciu poleceń.

Formanty

Środowiska użytkownika dostarczane przez model aplikacji są skonstruowanymi kontrolkami. W strukturze WPF *kontrolka* jest terminem ogólnym, który dotyczy kategorii klas WPF o następujących właściwościach:

- Są hostowane w oknie lub na stronie.
- Mają interfejs użytkownika.
- Implementują jakieś zachowanie.

Aby uzyskać więcej informacji, zobacz Kontrolki.

Kontrolki WPF według funkcji

Poniżej wymieniono wbudowane kontrolki WPF:

- **Przyciski**: Button i RepeatButton.
- **Wyświetlanie danych**: DataGrid, ListView i TreeView.
- **Wyświetlanie i wybieranie danych**: Calendar i DatePicker.
- **Okna dialogowe**: OpenFileDialog, PrintDialog i SaveFileDialog.
- **Pismo cyfrowe**: InkCanvas i InkPresenter.
- **Dokumenty**: DocumentViewer, FlowDocumentPageViewer, FlowDocumentReader, FlowDocumentScrollViewer i StickyNoteControl.
- **Dane wejściowe**: TextBox, RichTextBox i PasswordBox.
- **Układ**: Border, BulletDecorator, Canvas, DockPanel, Expander, Grid, GridView, GridSplitter, GroupBox, Panel, ResizeGrip, Separator, ScrollBar, ScrollViewer,

StackPanel, Thumb, Viewbox, VirtualizingStackPanel, Window i WrapPanel.

- **Multimedia**: Image, MediaElement i SoundPlayerAction.
- **Menu**: ContextMenu, Menu i ToolBar.
- **Nawigacja**: Frame, Hyperlink, Page, NavigationWindow i TabControl.
- **Wybór**: CheckBox, ComboBox, ListBox, RadioButton i Slider.
- **Informacje o użytkowniku**: AccessText, Label, Popup, ProgressBar, StatusBar, TextBlock i ToolTip.

Układ

Podczas tworzenia interfejsu użytkownika rozmieszcza się kontrolki według lokalizacji i rozmiaru, aby utworzyć układ. Kluczowym wymaganiem dla każdego układu jest dostosowanie się do zmian rozmiaru okna i ustawień wyświetlania. Zamiast wymuszać pisanie kodu w celu dostosowania układu w tych okolicznościach, struktura WPF zapewnia najwyższej klasy, rozszerzalny system układu.

Podstawą systemu układu jest pozycjonowanie względne, które zwiększa możliwość adaptacji do zmieniających się warunków okna i wyświetlania. System układu zarządza również negocjacjami między kontrolkami w celu określenia układu. Negocjacje są procesem dwuetapowym: najpierw kontrolka informuje swój element nadrzędny, jakiej lokalizacji i rozmiaru wymaga. Następnie element nadrzędny informuje kontrolkę, ile miejsca może zająć.

System układu jest uwidaczniany dla kontrolek podrzędnych za pomocą podstawowych klas WPF. Na potrzeby typowych układów, takich jak siatki, umieszczanie na stosie i dokowanie, struktura WPF zawiera kilka kontrolek układu:

- Canvas: Kontrolki podrzędne mają własny układ.
- DockPanel: Kontrolki podrzędne są wyrównane do krawędzi panelu.
- Grid: Kontrolki podrzędne są pozycjonowane według wierszy i kolumn.
- StackPanel: Kontrolki podrzędne są umieszczane na stosie pionowo lub poziomo.
- VirtualizingStackPanel: Kontrolki podrzędne są wirtualizowane i rozmieszczane w jednym wierszu o orientacji poziomej lub pionowej.
- WrapPanel: Kontrolki podrzędne są umieszczone w kolejności od lewej do prawej i opakowane do następnego wiersza, gdy nie ma wystarczającej ilości miejsca w

bieżącym wierszu.

W poniższym przykładzie użyto elementu DockPanel, aby utworzyć układ kilku kontrolek TextBox:

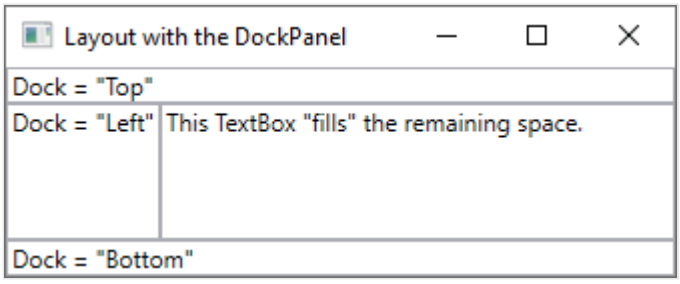
XAML

```
<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>
```

Element DockPanel umożliwia kontrolkom podrzędnym TextBox informowanie o sposobie ich rozmieszczenia. W tym celu element DockPanel implementuje właściwość dołączoną `Dock` uwidocznioną dla kontrolek podrzędnych, aby umożliwić każdej z nich określenie stylu dokowania.

> ⓘ **Uwaga**
>
> Właściwość zaimplementowana przez kontrolkę nadrzędną do użycia przez kontrolki podrzędne to konstrukcja WPF nazywana **właściwością dołączoną**.

Na poniższej ilustracji przedstawiono wynik działania znaczników XAML z poprzedniego przykładu:

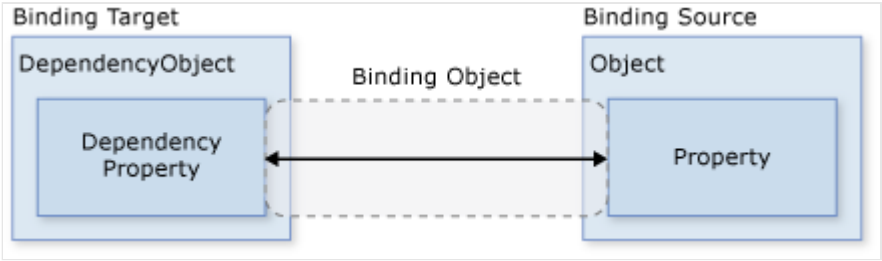


Powiązanie danych

Większość aplikacji tworzy się w celu zapewnienia użytkownikom środków do wyświetlania i edytowania danych. W przypadku aplikacji WPF praca nad przechowywaniem danych i uzyskiwaniem do nich dostępu jest już wykonywana przez wiele różnych bibliotek dostępu do danych platformy .NET, takich jak SQL i Entity Framework Core. Po uzyskaniu dostępu do danych i załadowaniu ich do obiektów zarządzanych aplikacji rozpoczyna się ciężka praca dla aplikacji WPF. Zasadniczo obejmuje ona dwie rzeczy:

1. Kopiowanie danych z obiektów zarządzanych do kontrolek, w których można wyświetlać i edytować dane.
2. Zapewnianie, że zmiany danych wprowadzone przy użyciu kontrolek zostaną skopiowane z powrotem do obiektów zarządzanych.

Aby uprościć tworzenie aplikacji, struktura WPF udostępnia zaawansowany aparat powiązania danych, który umożliwia automatyczną obsługę tych kroków. Podstawową jednostką aparatu powiązania danych jest klasa Binding, której zadaniem jest powiązanie kontrolki (obiektu docelowego powiązania) z obiektem danych (obiektem źródłowym powiązania). Tę relację pokazano na poniższej ilustracji:



Struktura WPF obsługuje deklarowanie powiązań bezpośrednio w znacznikach XAML. Na przykład poniższy kod XAML wiąże właściwość Text elementu TextBox z właściwością `Name` obiektu przy użyciu składni XAML „`{Binding ... }`”. Przyjęto założenie, że istnieje obiekt danych ustawiony na właściwość DataContext elementu `Window` za pomocą właściwości `Name`.

XAML

```
<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>
```

Aparat powiązania danych WPF zapewnia nie tylko powiązanie, ale także walidację, sortowanie, filtrowanie i grupowanie. Ponadto powiązanie danych umożliwia korzystanie z szablonów danych do tworzenia niestandardowego interfejsu użytkownika dla powiązanych danych.

Aby uzyskać więcej informacji, zobacz Omówienie powiązania danych.

Grafiki & animacja

Struktura WPF oferuje rozbudowany i elastyczny zestaw funkcji graficznych, które przynoszą następujące korzyści:

- **Grafika niezależna od rozdzielczości i niezależna od urządzenia**. Podstawową jednostką pomiaru w systemie graficznym WPF jest niezależny od urządzenia piksel o rozmiarze 1/96 cala, który stanowi podstawę renderowania niezależnego od rozdzielczości i niezależnego od urządzenia. Każdy piksel niezależny od urządzenia jest automatycznie skalowany w celu dopasowania do ustawienia liczby punktów na cal (dpi) systemu, na który jest renderowany.

- **Zwiększona precyzja**. System współrzędnych WPF jest mierzony przy użyciu liczb zmiennoprzecinkowych o podwójnej (a nie pojedynczej) precyzji. Przekształcenia i wartości nieprzezroczystości również są wyrażane z podwójną precyzją. Struktura WPF obsługuje również szeroką gamę kolorów (scRGB) i zapewnia zintegrowaną obsługę zarządzania danymi wejściowymi z różnych przestrzeni kolorów.

- **Obsługa zaawansowanej grafiki i animacji**. Struktura WPF upraszcza programowanie grafiki, zarządzając scenami animacji za Ciebie; nie musisz martwić się o przetwarzanie scen, pętle renderowania i interpolację dwuliniową. Ponadto struktura WPF zapewnia obsługę testowania trafień i pełną obsługę składania alfa.

- **Przyspieszanie sprzętowe**. System graficzny WPF wykorzystuje sprzęt graficzny, aby zminimalizować użycie procesora CPU.

Grafika 2D

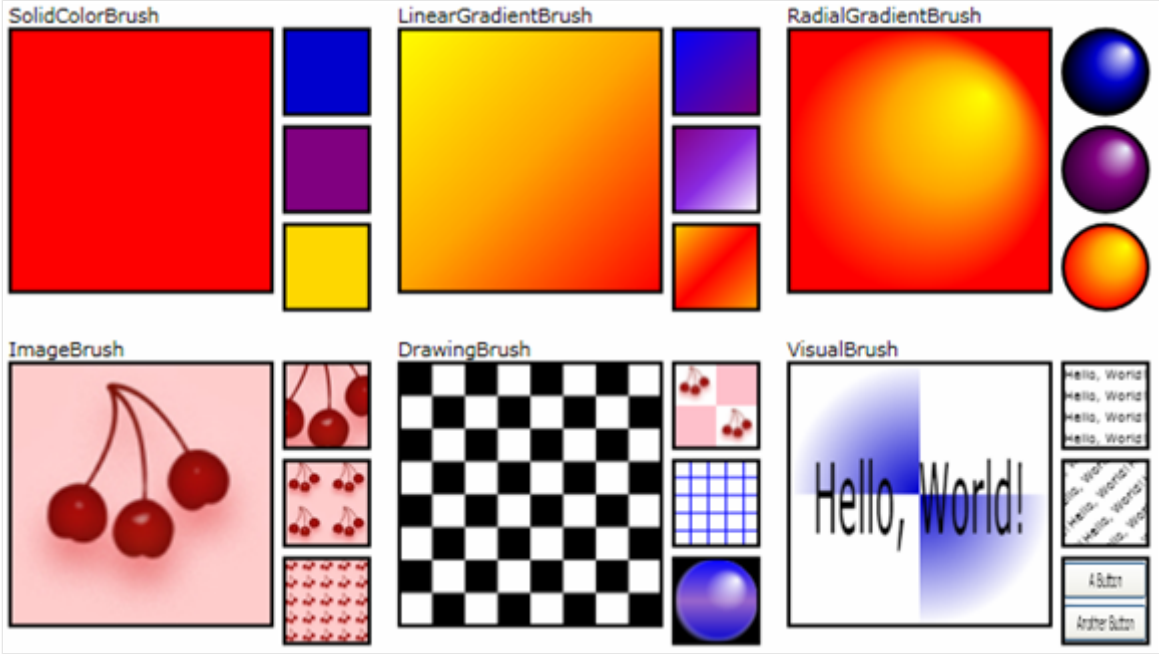
Struktura WPF udostępnia bibliotekę typowych kształtów wektorowych 2D, takich jak prostokąty i elipsy. Kształty te nie służą tylko do wyświetlania; implementują one wiele funkcji, których oczekujesz od kontrolek, w tym wprowadzanie danych za pomocą klawiatury i myszy.

Kształty 2D dostarczane przez strukturę WPF obejmują standardowy zestaw kształtów podstawowych. Jednak w celu ułatwienia projektowania dostosowanego interfejsu

użytkownika może być konieczne utworzenie kształtów niestandardowych. Struktura WPF udostępnia geometrie do tworzenia niestandardowego kształtu, który może być rysowany bezpośrednio, używany jako pędzel lub używany do przycinania innych kształtów i kontrolek.

Aby uzyskać więcej informacji, zobacz Omówienie geometrii.

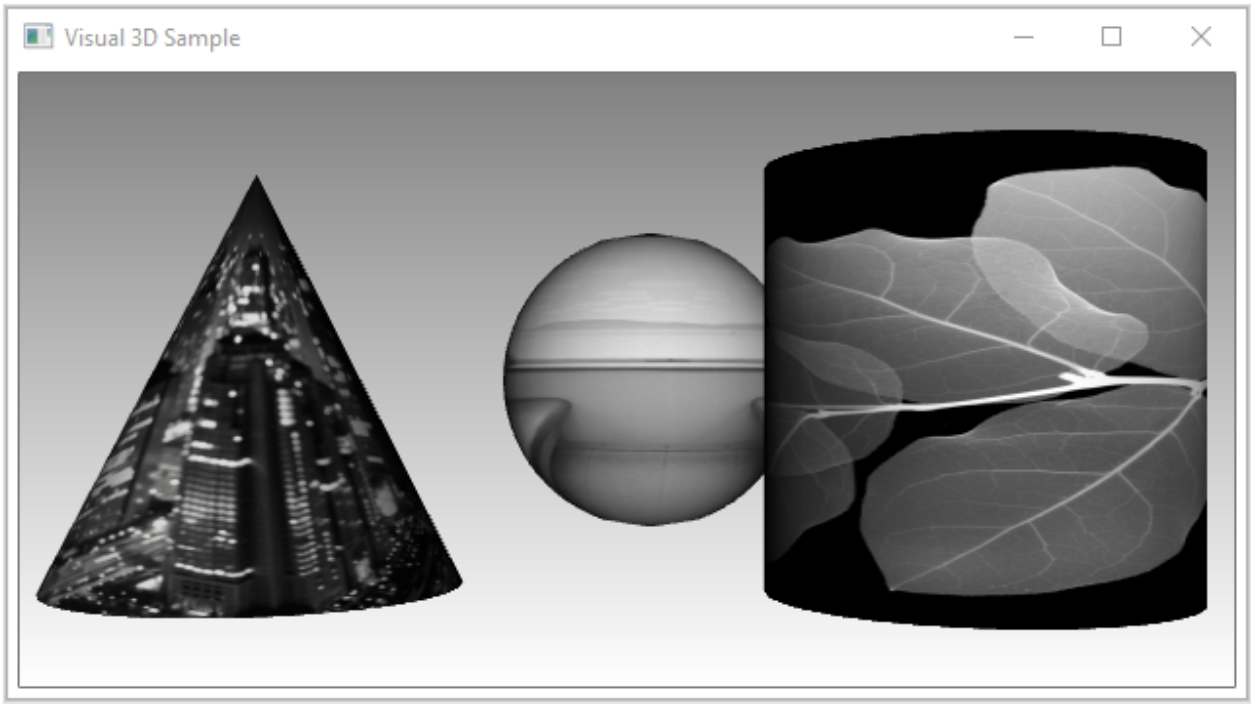
Podzestaw możliwości 2D struktury WPF obejmuje efekty wizualne, takie jak gradienty, mapy bitowe, rysunki, malowanie wideo, rotacja, skalowanie i pochylanie. Wszystkie te efekty uzyskuje się za pomocą pędzli. Na poniższej ilustracji przedstawiono kilka przykładów:



Aby uzyskać więcej informacji, zobacz Omówienie pędzli WPF.

Renderowanie 3D

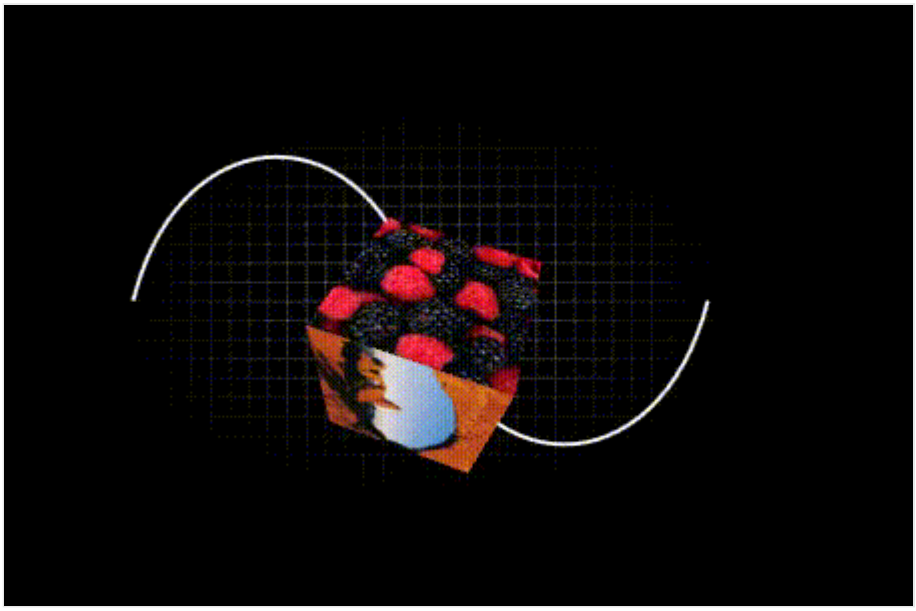
Struktura WPF obejmuje również możliwości renderowania 3D, które integrują się z grafiką 2D, aby umożliwić tworzenie bardziej ekscytujących i interesujących interfejsów użytkownika. Na przykład na poniższej ilustracji przedstawiono obrazy 2D renderowane na kształty 3D:



Aby uzyskać więcej informacji, zobacz Omówienie grafiki 3D.

Animacja

Obsługa animacji WPF umożliwia stosowanie efektów powiększania, wstrząsania, obracania i zanikania dla kontrolek w celu tworzenia interesujących przejść między stronami i nie tylko. Większość klas WPF można animować, nawet klasy niestandardowe. Na poniższej ilustracji przedstawiono prostą animację w działaniu:



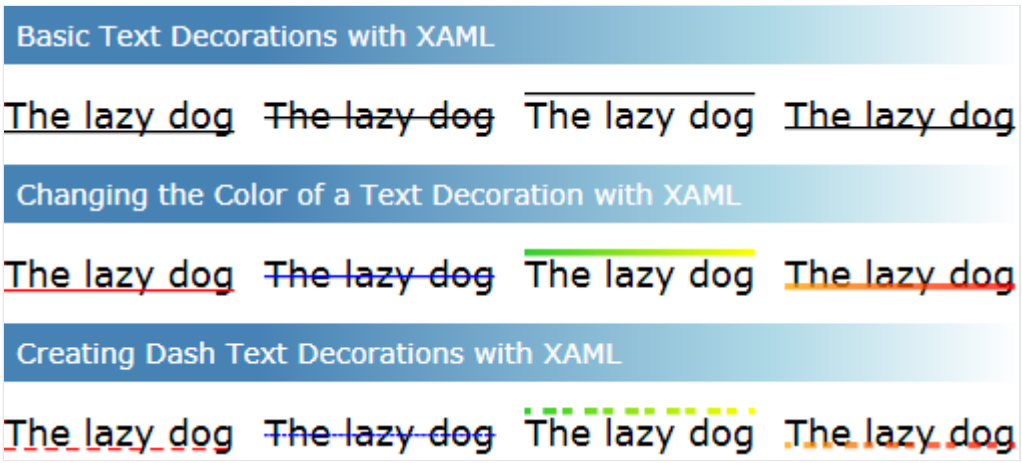
Aby uzyskać więcej informacji, zobacz Omówienie animacji.

Tekst i typografia

Aby zapewnić wysokiej jakości renderowanie tekstu, struktura WPF oferuje następujące funkcje:

- Obsługa czcionek OpenType.
- Ulepszenia technologii ClearType.
- Wysoka wydajność wykorzystująca przyspieszanie sprzętowe.
- Integracja tekstu z multimediami, grafiką i animacją.
- Obsługa czcionek międzynarodowych i mechanizmy awaryjne.

W ramach pokazu integracji tekstu z grafiką na poniższej ilustracji przedstawiono zastosowanie dekoracji tekstu:



Aby uzyskać więcej informacji, zobacz Typografia w strukturze Windows Presentation Foundation.

Dostosowywanie aplikacji WPF

Do tej pory przedstawiliśmy podstawowe bloki konstrukcyjne struktury WPF umożliwiające tworzenie aplikacji:

- Model aplikacji służy do hostowania i dostarczania zawartości aplikacji, która składa się głównie z kontrolek.
- W celu uproszczenia rozmieszczenia kontrolek w interfejsie użytkownika użyj systemu układu WPF.
- Powiązanie danych pozwala zmniejszyć nakład pracy wymaganej do integracji interfejsu użytkownika z danymi.
- Aby poprawić wygląd wizualny aplikacji, użyj kompleksowego zakresu grafiki, animacji i obsługi multimediów dostarczanego przez strukturę WPF.

Jednak często podstawowe funkcje nie wystarczają do tworzenia naprawdę wyróżniających się i wizualnie atrakcyjnych środowisk użytkownika oraz zarządzania nimi. Standardowe kontrolki WPF mogą nie integrować się z pożądanym wyglądem

aplikacji. Dane mogą nie być wyświetlane w najbardziej efektywny sposób. Ogólne środowisko użytkownika aplikacji może nie być dostosowane do domyślnego wyglądu i sposobu działania motywów systemu Windows.

Z tego powodu struktura WPF udostępnia różne mechanizmy tworzenia unikatowych środowisk użytkownika.

Model zawartości

Głównym zadaniem większości kontrolek WPF jest wyświetlanie zawartości. W strukturze WPF typ i liczbę elementów, które mogą stanowić zawartość kontrolki, określa się jako *model zawartości* kontrolki. Niektóre kontrolki mogą zawierać pojedynczy element i typ zawartości. Na przykład zawartość obiektu TextBox jest wartością ciągu przypisaną do właściwości Text.

Jednak inne kontrolki mogą zawierać wiele elementów różnego typu zawartości; zawartość obiektu Button, określona przez właściwość Content, może zawierać różne elementy, w tym kontrolki układu, tekst, obrazy i kształty.

Aby uzyskać więcej informacji na temat rodzajów zawartości obsługiwanej przez różne kontrolki, zobacz Model zawartości WPF.

Wyzwalacze

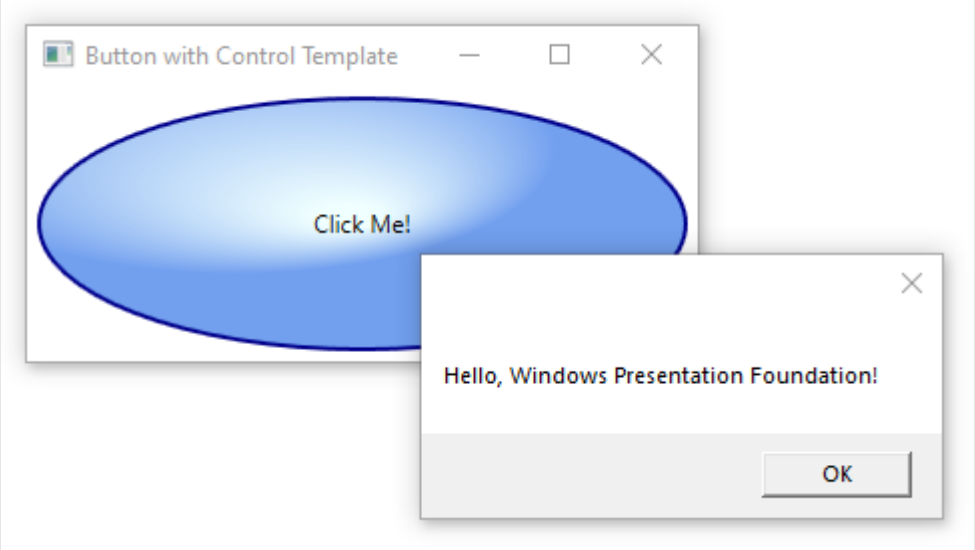
Chociaż głównym zadaniem znaczników XAML jest implementowanie wyglądu aplikacji, języka XAML można również używać do implementowania niektórych aspektów zachowania aplikacji. Jednym z przykładów jest użycie wyzwalaczy w celu zmiany wyglądu aplikacji na podstawie interakcji użytkownika. Aby uzyskać więcej informacji, zobacz Style i szablony.

Szablony

Domyślne interfejsy użytkownika kontrolek WPF są zwykle konstruowane z innych kontrolek i kształtów. Na przykład element Button składa się z kontrolek ButtonChrome i ContentPresenter. Element ButtonChrome zapewnia standardowy wygląd przycisku, a element ContentPresenter wyświetla zawartość przycisku określoną przez właściwość Content.

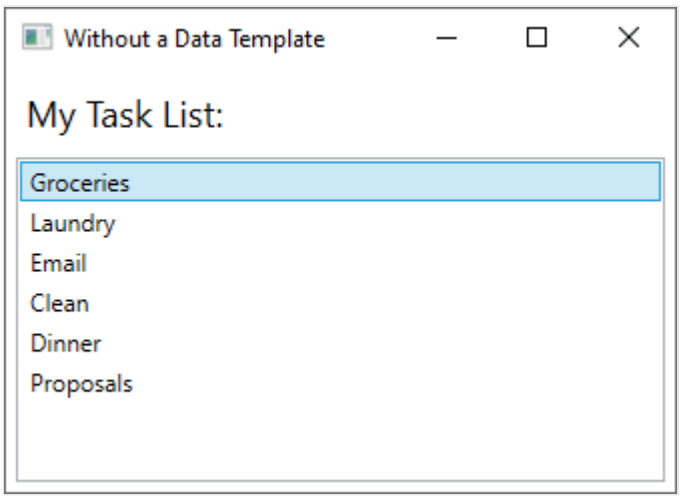
Czasami domyślny wygląd kontrolki może kolidować z ogólnym wyglądem aplikacji. W takim przypadku można użyć elementu ControlTemplate, aby zmienić wygląd interfejsu użytkownika kontrolki bez zmiany jego zawartości i zachowania.

Na przykład element Button zgłasza zdarzenie Click po kliknięciu. Zmiana szablonu przycisku w celu wyświetlenia kształtu Ellipse powoduje zmianę aspektu wizualnego kontrolki, ale funkcjonalność pozostaje niezmieniona. Nadal możesz kliknąć aspekt wizualny kontrolki, a zdarzenie Click zostanie wywołane zgodnie z oczekiwaniami.

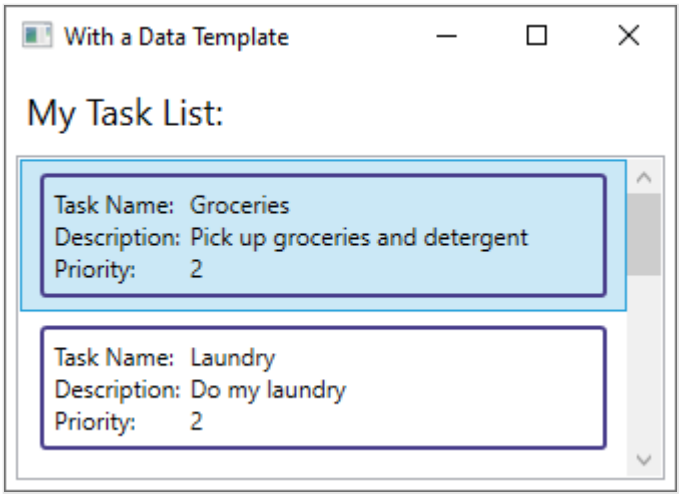


Szablony danych

Podczas gdy szablon kontrolki pozwala określić jej wygląd, szablon danych umożliwia określenie wyglądu zawartości kontrolki. Szablony danych są często używane do ulepszania sposobu wyświetlania powiązanych danych. Na poniższej ilustracji przedstawiono domyślny wygląd elementu ListBox powiązanego z kolekcją obiektów `Task`, gdzie każde zadanie ma nazwę, opis i priorytet:



Wygląd domyślny to wygląd, jakiego można oczekiwać od elementu ListBox. Jednak wygląd domyślny każdego zadania zawiera tylko nazwę zadania. Aby wyświetlić nazwę, opis i priorytet zadania, wygląd domyślny powiązanych elementów listy kontrolki ListBox należy zmienić przy użyciu elementu DataTemplate. Oto przykład zastosowania szablonu danych utworzonego dla obiektu `Task`.



Element ListBox nie zmienia zachowania i wyglądu ogólnego — zmienia się tylko wygląd zawartości wyświetlanej przez pole listy.

Aby uzyskać więcej informacji, zobacz Omówienie tworzenia szablonów danych.

Style

Style umożliwiają deweloperom i projektantom standaryzację określonego wyglądu produktu. Struktura WPF zapewnia silny model stylu, którego podstawą jest element Style. Style mogą stosować wartości właściwości do typów. Można je stosować automatycznie do wszystkich elementów zgodnie z typem lub poszczególnych obiektów w przypadku odwołania. Poniższy przykład powoduje utworzenie stylu, który ustawia kolor tła dla każdego elementu Button w oknie na wartość `Orange`:

XAML

```
<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>
```

Ponieważ ten styl jest przeznaczony dla wszystkich kontrolek Button, styl jest automatycznie stosowany do wszystkich przycisków w oknie, jak pokazano na poniższej ilustracji:

Aby uzyskać więcej informacji, zobacz Style i szablony.

Zasoby

Kontrolki w aplikacji powinny mieć taki sam wygląd, który może zawierać dowolne elementy od czcionek i kolorów tła do szablonów kontrolek, szablonów danych i stylów. Obsługa struktury WPF dla zasobów interfejsu użytkownika umożliwia hermetyzowanie tych zasobów w jednej lokalizacji w celu ponownego użycia.

W poniższym przykładzie zdefiniowano wspólny kolor tła, który jest współużytkowany przez elementy Button i Label:

XAML

```
<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>
```

Aby uzyskać więcej informacji, zobacz Jak zdefiniować zasób WPF i odwoływać się do niego.

Kontrolki niestandardowe

Mimo że struktura WPF zapewnia hosta obsługi dostosowywania, mogą wystąpić sytuacje, w których istniejące kontrolki WPF nie spełniają wymagań aplikacji lub jej użytkowników. Może się to zdarzyć w następujących przypadkach:

- Nie można utworzyć wymaganego interfejsu użytkownika przez dostosowanie wyglądu i sposobu działania istniejących implementacji WPF.
- Wymagane zachowanie nie jest obsługiwane (lub nie jest obsługiwane w łatwy sposób) przez istniejące implementacje WPF.

Jednak na tym etapie można skorzystać z jednego z trzech modeli WPF, aby utworzyć nową kontrolkę. Każdy model jest przeznaczony dla określonego scenariusza i wymaga niestandardowej kontrolki pochodzącej z określonej klasy bazowej WPF. Poniżej wymieniono t trzy modele:

- **Model kontrolki użytkownika**
  Kontrolka niestandardowa pochodzi od elementu UserControl i składa się z co najmniej jednej innej kontrolki.

- **Model kontrolki** Kontrolka niestandardowa pochodzi od elementu Control i służy do kompilowania implementacji, które oddzielają ich zachowanie od wyglądu przy użyciu szablonów, podobnie jak w przypadku większości kontrolek WPF. Pochodzenie od elementu Control pozwala na większą swobodę tworzenia niestandardowego interfejsu użytkownika niż w przypadku kontrolek użytkownika, ale może wymagać większego nakładu pracy.

- **Model elementu struktury**.
  Kontrolka niestandardowa pochodzi od elementu FrameworkElement, gdy jej wygląd jest definiowany przez niestandardową logikę renderowania (a nie szablony).

Aby uzyskać więcej informacji na temat kontrolek niestandardowych, zobacz Omówienie tworzenia kontrolek.

Zobacz też

- Samouczek: Tworzenie nowej aplikacji WPF
- Migrowanie aplikacji WPF do platformy .NET
- Omówienie okien WPF
- Omówienie powiązań danych
- Przegląd XAML

Samouczek: tworzenie nowej aplikacji WPF przy użyciu platformy .NET

Artykuł • 10.02.2023 • Czas czytania: 11 min

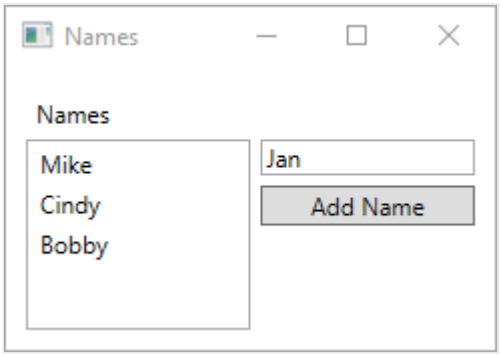
W tym krótkim samouczku dowiesz się, jak utworzyć nową aplikację Windows Presentation Foundation (WPF) przy użyciu programu Visual Studio. Po wygenerowaniu początkowej aplikacji dowiesz się, jak dodawać kontrolki i jak obsługiwać zdarzenia. Na koniec tego samouczka będziesz mieć prostą aplikację, która dodaje nazwy do pola listy.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Ten samouczek zawiera informacje na temat wykonywania następujących czynności:

- ✓ Tworzenie nowej aplikacji WPF
- ✓ Dodawanie kontrolek do formularza
- ✓ Obsługa zdarzeń sterowania w celu zapewnienia funkcjonalności aplikacji
- ✓ Uruchamianie aplikacji

Oto wersja zapoznawcza aplikacji, którą utworzysz, wykonując czynności opisane w tym samouczku:



Wymagania wstępne

- Visual Studio 2022 w wersji 17.4 lub nowszej
  - Wybieranie obciążenia programistycznego dla komputerów .NET
  - Wybierz pojedynczy składnik platformy .NET 7

> **Porada**

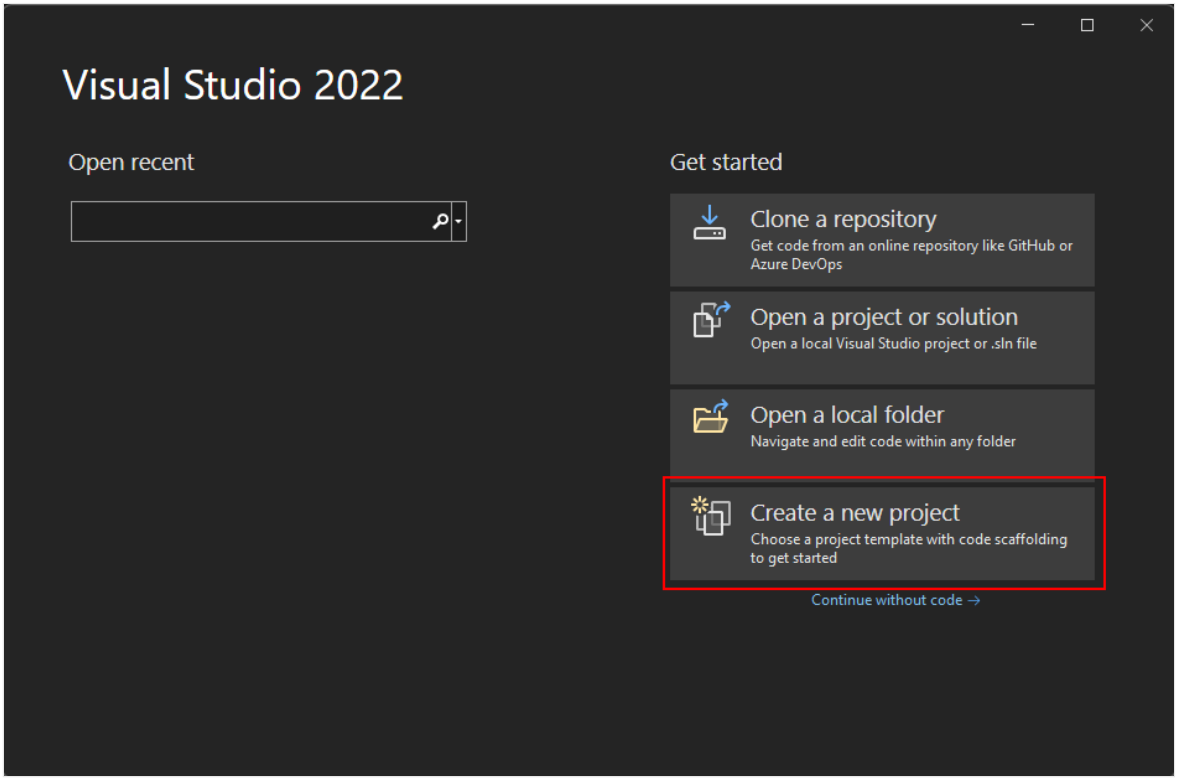
Użyj programu Visual Studio 2022 w wersji 17.4 lub nowszej i zainstaluj zarówno poszczególne składniki platformy .NET 7, jak i .NET 6. Dodano obsługę platformy .NET 7 w programie Visual Studio 2022 w wersji 17.4.

Tworzenie aplikacji WPF

Pierwszym krokiem do utworzenia nowej aplikacji jest otwarcie programu Visual Studio i wygenerowanie aplikacji na podstawie szablonu.

1. Otwórz program Visual Studio.

2. Wybierz pozycję **Utwórz nowy projekt**.



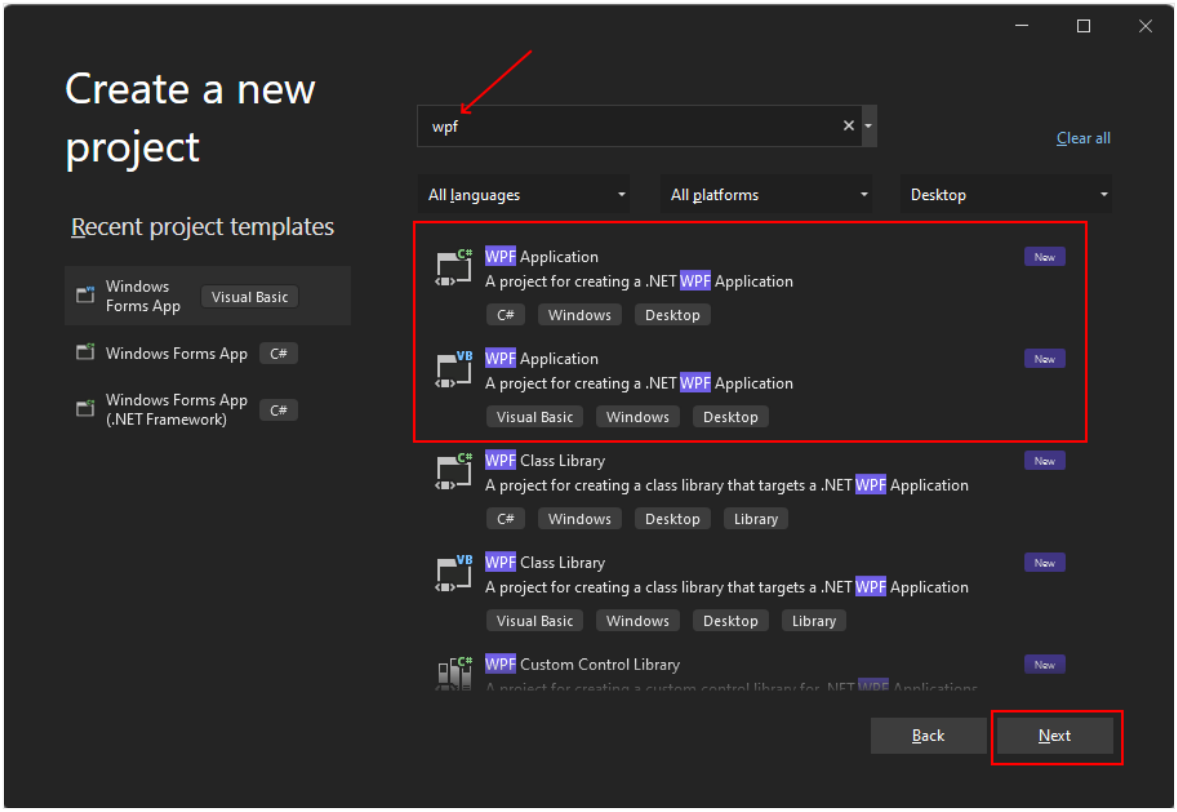
3. W polu **Wyszukaj szablony** wpisz *wpf*, a następnie naciśnij `klawisz Enter`.

4. Na liście rozwijanej **języka kodu** wybierz pozycję **C#** lub **Visual Basic**.

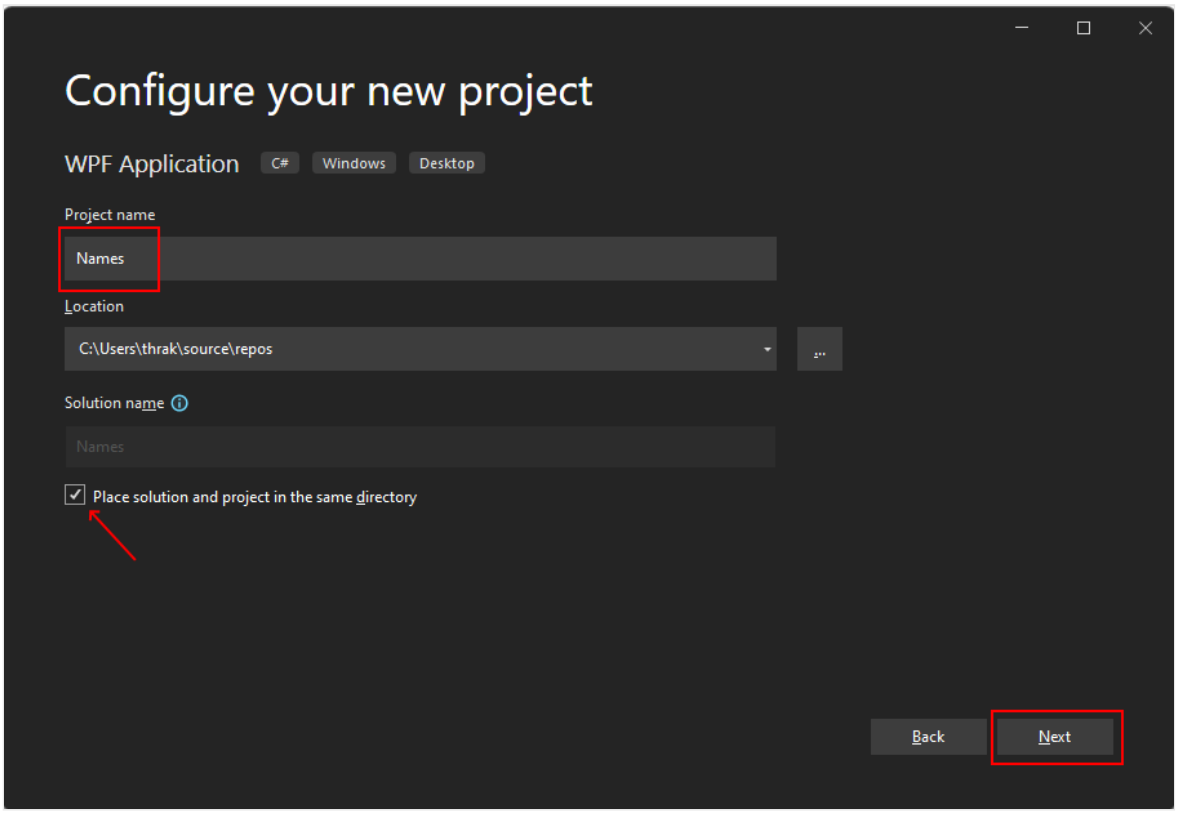
5. Na liście szablonów wybierz pozycję **Aplikacja WPF** , a następnie wybierz pozycję **Dalej**.

> ⓘ **Ważne**
>
> Nie wybieraj szablonu **Aplikacja WPF (.NET *Framework*)**.

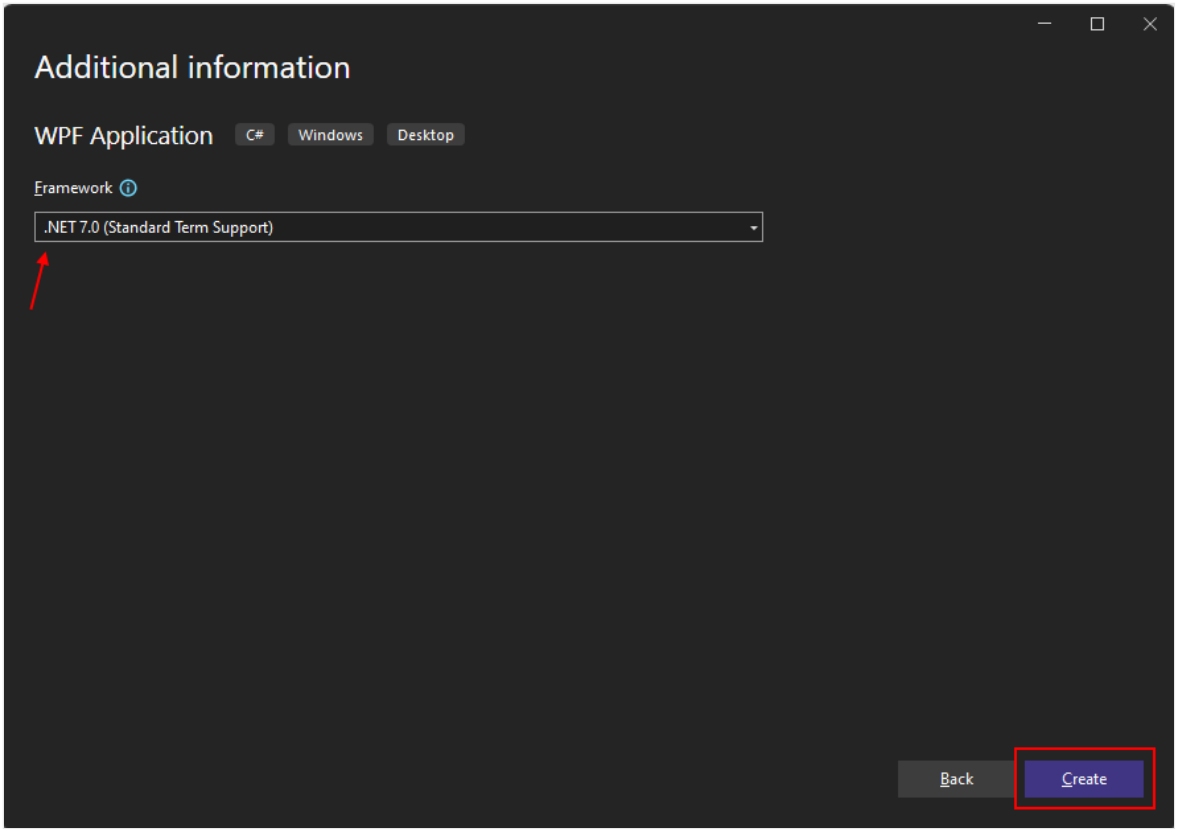
Na poniższej ilustracji przedstawiono zarówno szablony projektów C#, jak i Visual Basic .NET. Jeśli zastosowano filtr **języka kodu** , zobaczysz odpowiedni szablon.



6. W oknie **Konfigurowanie nowego projektu** wykonaj następujące czynności:
   a. W polu **Nazwa projektu** wprowadź *nazwy*.
   b. Zaznacz pole wyboru **Umieść rozwiązanie i projekt w tym samym katalogu** .
   c. Opcjonalnie wybierz inną **lokalizację** , aby zapisać kod.
   d. Kliknij przycisk **Next** (Dalej).



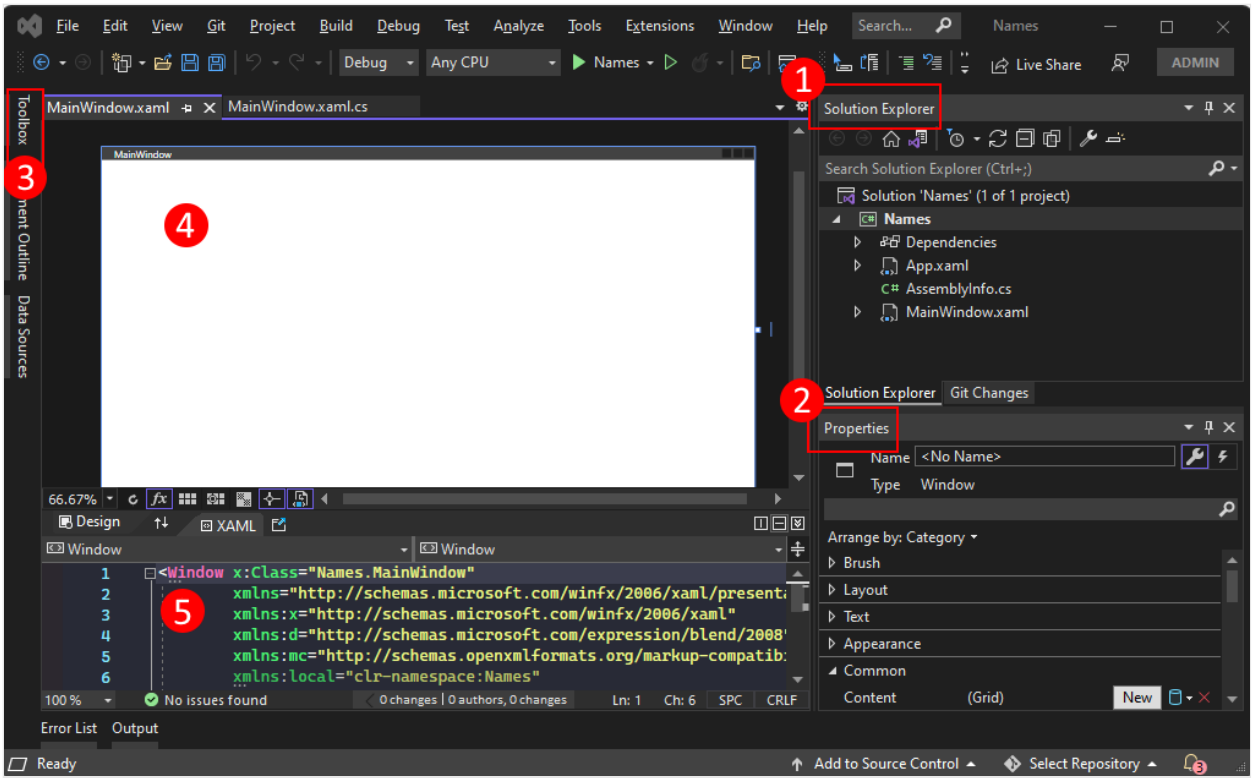
7. W oknie **Dodatkowe informacje** wybierz pozycję **.NET 7.0 (Standardowa obsługa terminów)** dla **programu Target Framework**. Wybierz przycisk **Utwórz**.



Po wygenerowaniu aplikacji program Visual Studio powinien otworzyć okienko projektanta XAML dla okna domyślnego *MainWindow*. Jeśli projektant nie jest widoczny, kliknij dwukrotnie plik *MainWindow.xaml* w okienku **Eksplorator rozwiązań**, aby otworzyć projektanta.

Ważne części programu Visual Studio

Obsługa WPF w programie Visual Studio ma pięć ważnych składników, z którymi będziesz korzystać podczas tworzenia aplikacji:



1. Eksplorator rozwiązań

   Wszystkie pliki projektu, kod, okna, zasoby zostaną wyświetlone w tym okienku.

2. Właściwości

   W tym okienku są wyświetlane ustawienia właściwości, które można skonfigurować na podstawie wybranego elementu. Jeśli na przykład wybierzesz element z **Eksplorator rozwiązań**, zobaczysz ustawienia właściwości powiązane z plikiem. Jeśli wybierzesz obiekt w **Projektancie**, zobaczysz ustawienia dla tego elementu.

3. Przybornik

   Przybornik zawiera wszystkie kontrolki, które można dodać do formularza. Aby dodać kontrolkę do bieżącego formularza, kliknij dwukrotnie kontrolkę lub przeciągnij i upuść kontrolkę.

4. Projektant XAML

   Jest to projektant dokumentu XAML. Jest interaktywna i można przeciągać i upuszczać obiekty z **przybornika**. Wybierając i przenosząc elementy w projektancie, możesz wizualnie utworzyć interfejs użytkownika dla aplikacji.

   Gdy zarówno projektant, jak i edytor są widoczne, zmiany w jednym są odzwierciedlane w drugiej. Po wybraniu elementów w projektancie w okienku **Właściwości** zostaną wyświetlone właściwości i atrybuty dotyczące tego obiektu.

5. Edytor kodu XAML

Jest to edytor kodu XAML dla dokumentu XAML. Edytor kodu XAML to sposób tworzenia interfejsu użytkownika ręcznie bez projektanta. Projektant może wnioskować wartości właściwości kontrolki po dodaniu kontrolki do projektanta. Edytor kodu XAML zapewnia o wiele większą kontrolę.

Gdy zarówno projektant, jak i edytor są widoczne, zmiany w jednym są odzwierciedlane w drugiej. Podczas nawigowania po daszku tekstu w edytorze kodu okienko **Właściwości** wyświetla właściwości i atrybuty dotyczące tego obiektu.

Badanie kodu XAML

Po utworzeniu projektu edytor kodu XAML jest widoczny z minimalną ilością kodu XAML do wyświetlenia okna. Jeśli edytor nie jest otwarty, kliknij dwukrotnie element *MainWindow.xaml* w **Eksplorator rozwiązań**. Powinien zostać wyświetlony kod XAML podobny do następującego przykładu:

XAML

```
<Window x:Class="Names.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:Names"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Przeanalizujmy ten kod XAML, aby lepiej go zrozumieć. XAML to po prostu XML, który może być przetwarzany przez kompilatory używane przez WPF. Opisuje on interfejs użytkownika WPF i wchodzi w interakcje z kodem platformy .NET. Aby zrozumieć język XAML, należy zapoznać się co najmniej z podstawami kodu XML.

Katalog główny `<Window>` dokumentu reprezentuje typ obiektu opisywanego przez plik XAML. Zadeklarowane są osiem atrybutów i zazwyczaj należą do trzech kategorii:

- Przestrzenie nazw

  Przestrzeń nazw XML zapewnia strukturę XML, określając, jaka zawartość XML można zadeklarować w pliku.

Atrybut główny `xmlns` importuje przestrzeń nazw XML dla całego pliku, a w tym przypadku mapuje na typy zadeklarowane przez WPF. Inne przestrzenie nazw XML deklarują prefiks i importują inne typy i obiekty dla pliku XAML. Na przykład `xmlns:local` przestrzeń nazw deklaruje prefiks i mapuje `local` na obiekty zadeklarowane przez projekt, te zadeklarowane w `Names` przestrzeni nazw kodu.

- Atrybut `x:Class`

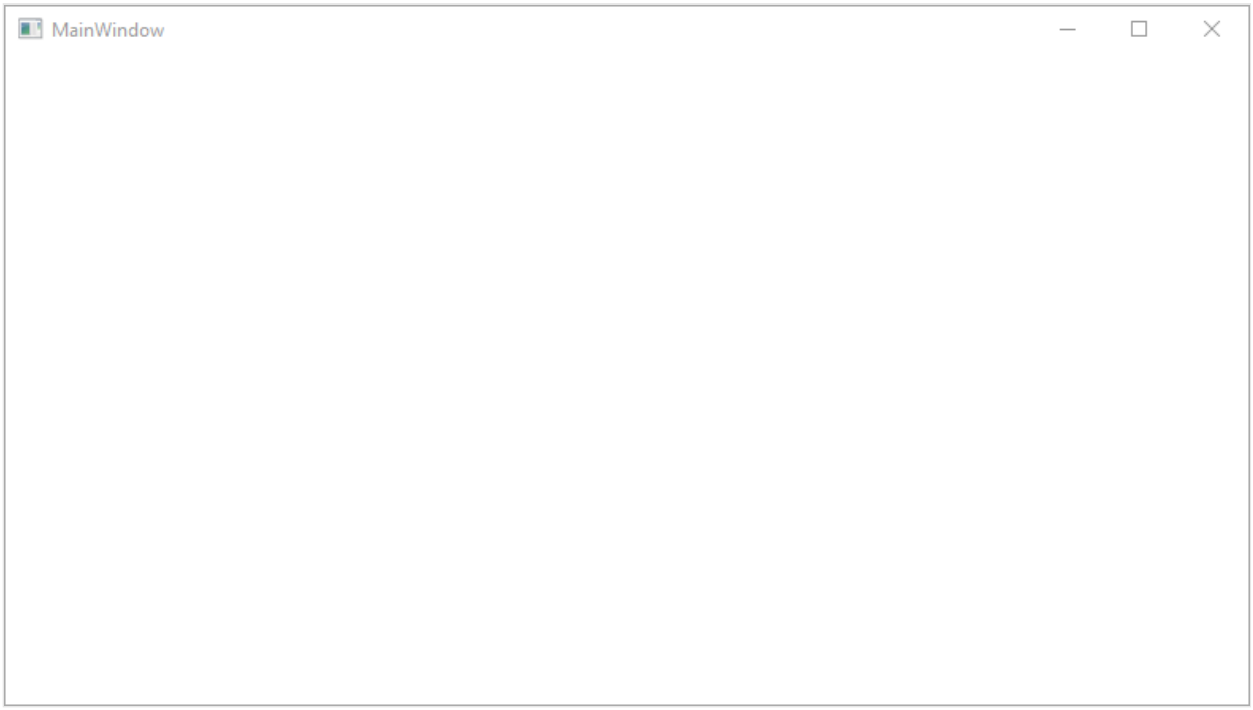
  Ten atrybut mapuje `<Window>` typ zdefiniowany przez kod: *MainWindow.xaml.cs* lub *MainWindow.xaml.vb* , który jest klasą `Names.MainWindow` .

- Atrybut `Title`

  Każdy atrybut normalny zadeklarowany w obiekcie XAML ustawia właściwość tego obiektu. W tym przypadku `Title` atrybut ustawia `Window.Title` właściwość .

Zmienianie okna

Najpierw uruchom projekt i wyświetl domyślne dane wyjściowe. Zostanie wyświetlone okno, które zostanie wyświetlone, bez żadnych kontrolek i tytuł **głównego systemu Windows**:



W naszej przykładowej aplikacji to okno jest zbyt duże, a pasek tytułu nie jest opisowy. Zmień tytuł i rozmiar okna, zmieniając odpowiednie atrybuty w języku XAML na następujące wartości:

```
XAML
```

```
<Window x:Class="Names.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:Names"
        mc:Ignorable="d"
        Title="Names" Height="180" Width="260">
    <Grid>

    </Grid>
</Window>
```

Przygotowywanie układu

WPF zapewnia zaawansowany system układów z wieloma różnymi kontrolkami układu. Kontrolki układu ułatwiają umieszczanie kontrolek podrzędnych i ich rozmiar, a nawet automatyczne wykonywanie tych czynności. Domyślna kontrolka układu podana w tym języku XAML to kontrolka `<Grid>` .

Kontrolka `Grid` umożliwia definiowanie wierszy i kolumn, podobnie jak tabela, oraz umieszczanie kontrolek w granicach określonej kombinacji wierszy i kolumn. Możesz mieć dowolną liczbę kontrolek podrzędnych lub innych kontrolek układu dodanych do elementu `Grid`. Można na przykład umieścić inną `Grid` kontrolkę w określonej kombinacji wierszy i kolumn, a następnie `Grid` zdefiniować więcej wierszy i kolumn oraz mieć własne elementy podrzędne.

Kontrolka `<Grid>` definiuje wiersze i kolumny, w których będą używane kontrolki. Siatka zawsze ma zadeklarowany pojedynczy wiersz i kolumnę, co oznacza, że siatka domyślnie jest pojedynczą komórką. To naprawdę nie zapewnia dużej elastyczności w umieszczaniu kontrolek.

Przed dodaniem nowych wierszy i kolumn dodaj nowy atrybut do `<Grid>` elementu : `Margin="10"`. To powoduje podniesienie siatki z okna i sprawia, że wygląda trochę ładniej.

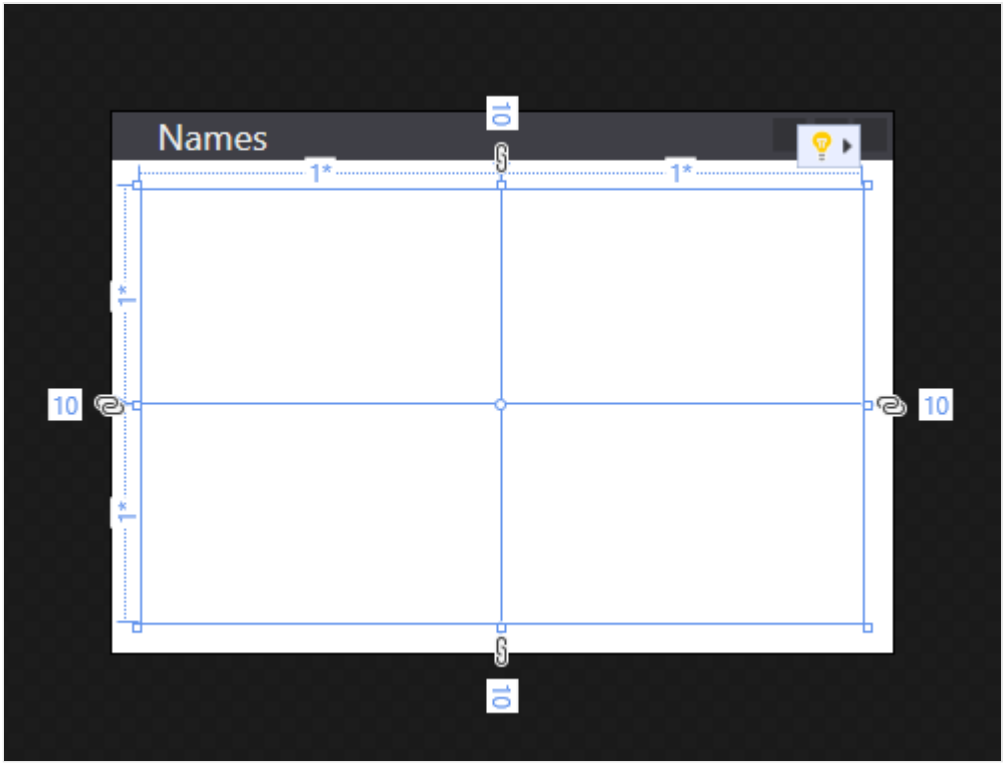
Następnie zdefiniuj dwa wiersze i dwie kolumny, dzieląc siatkę na cztery komórki:

XAML

```
<Window x:Class="Names.LayoutStep2"
        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:Names"
        mc:Ignorable="d"
        Title="Names" Height="180" Width="260">
    <Grid Margin="10">

        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

    </Grid>
</Window>
```

Wybierz siatkę w edytorze kodu XAML lub projektancie XAML, zobaczysz, że projektant XAML pokazuje każdy wiersz i kolumnę:



Dodawanie pierwszej kontrolki

Teraz, gdy siatka została utworzona, możemy rozpocząć dodawanie do niej kontrolek. Najpierw zacznij od kontrolki etykiety. Utwórz nowy `<Label>` element wewnątrz `<Grid>` elementu po definicjach wierszy i kolumn i nadaj mu wartość ciągu :`Names`

XAML

```
<Grid Margin="10">

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Label>Names</Label>

</Grid>
```

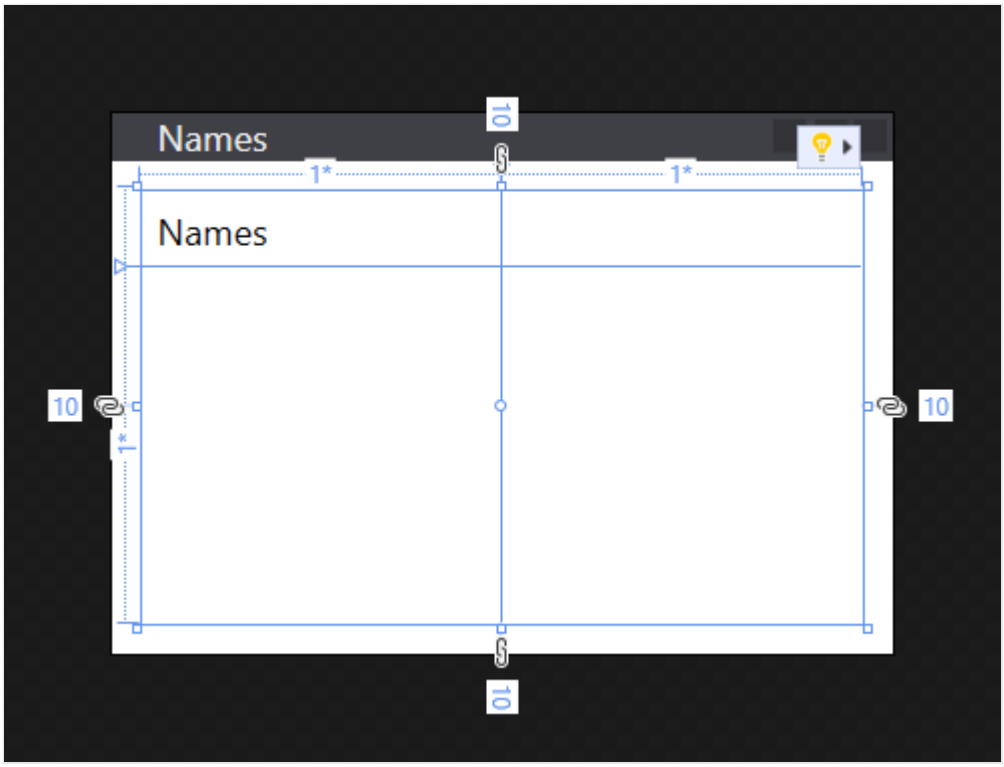
Element `<Label>Names</Label>` definiuje zawartość `Names`. Niektóre kontrolki rozumieją sposób obsługi zawartości, inne nie. Zawartość kontrolki mapuje na `Content` właściwość. Ustawienie zawartości za pomocą składni atrybutu XAML, należy użyć tego formatu: `<Label Content="Names" />`. Oba sposoby są takie same, ustawiając zawartość etykiety, aby wyświetlić tekst `Names`.

Mamy jednak problem, etykieta zajmuje połowę okna, ponieważ została automatycznie przypisana do pierwszego wiersza i kolumny siatki. W przypadku pierwszego wiersza nie potrzebujemy tego miejsca, ponieważ będziemy używać tylko tego wiersza dla etykiety. `Height` Zmień atrybut pierwszego `<RowDefinition>` elementu z `*` na `Auto`. Wartość `Auto` automatycznie zmienia rozmiar wiersza siatki na rozmiar jego zawartości, w tym przypadku kontrolkę etykiety.

XAML

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>
```

Zwróć uwagę, że projektant pokazuje teraz etykietę zajmującą niewielką ilość dostępnej wysokości. Jest teraz więcej miejsca na następny wiersz do zajmowania. Większość kontrolek definiuje jakiś rodzaj wartości wysokości i szerokości, które powinny zajmować, które najlepiej dla nich wyglądają. Na przykład kontrolka etykiety ma wartość wysokości, która gwarantuje, że można ją odczytać.



Umieszczanie kontrolek

Porozmawiajmy o umieszczaniu kontrolek. Etykieta utworzona w powyższej sekcji została automatycznie umieszczona w wierszu 0 i kolumnie 0 siatki. Numerowanie wierszy i kolumn rozpoczyna się od 0 i zwiększa się o 1 dla każdego nowego wiersza lub kolumny. Kontrolka nie wie nic o siatce, a kontrolka nie definiuje żadnych właściwości do kontrolowania jego umieszczania w siatce. Kontrolka mogła być nawet umieszczona w innej kontrolce układu, która ma własny zestaw reguł definiujących sposób umieszczania kontrolek.

Jak poinformować kontrolkę o użyciu innego wiersza lub kolumny, gdy kontrolka nie zna siatki? Dołączone właściwości! Siatka korzysta z zaawansowanego systemu właściwości dostarczonego przez WPF. Siatka definiuje nowe właściwości, które kontrolki podrzędne mogą deklarować i używać. Właściwości nie istnieją w samej kontrolce, są one dołączone przez siatkę po dodaniu kontrolki do siatki.

Siatka definiuje dwie właściwości, aby określić położenie wiersza i kolumny kontrolki podrzędnej: `Grid.Row` i `Grid.Column`. Jeśli te właściwości zostaną pominięte z kontrolki, oznacza to, że mają wartości domyślne 0, dlatego kontrolka jest umieszczana w wierszu `0` i kolumnie `0` siatki. Spróbuj zmienić położenie kontrolki `<Label>` , ustawiając `Grid.Column` atrybut na `1`:

XAML

```
<Label Grid.Column="1">Names</Label>
```

Zwróć uwagę, jak etykieta została teraz przeniesiona do drugiej kolumny. Możesz użyć `Grid.Row` właściwości i `Grid.Column` dołączonych, aby umieścić kolejne kontrolki, które utworzymy. Na razie jednak przywróć etykietę do wiersza 0.

Tworzenie pola listy nazw

Teraz, gdy siatka ma prawidłowy rozmiar i utworzoną etykietę, dodaj kontrolkę pola listy w wierszu poniżej etykiety. Pole listy będzie znajdować się w wierszu `1` i kolumnie `0`. Nadamy również tej kontrolce nazwę `lstNames`. Po nazwie kontrolki można odwoływać się do niej w kodzie. Nazwa jest przypisywana do kontrolki z atrybutem `x:Name` .

XAML

```
<Grid Margin="10">

    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Label>Names</Label>
    <ListBox Grid.Row="1" x:Name="lstNames" />

</Grid>
```

Dodawanie pozostałych kontrolek

Ostatnie dwie kontrolki, które dodamy, to pole tekstowe i przycisk, którego użytkownik użyje do wprowadzenia nazwy do dodania do pola listy. Jednak zamiast próbować utworzyć więcej wierszy i kolumn dla siatki, umieścimy te kontrolki w kontrolce `<StackPanel>` układu.

Panel stosu różni się od siatki w sposobie umieszczenia kontrolek. Gdy informujesz siatkę, w której kontrolki mają być z dołączonymi właściwościami `Grid.Row` i , `Grid.Column` panel stosu działa automatycznie, umieszczając pierwszą kontrolkę, a następnie umieszczając następną kontrolkę po niej, kontynuując do momentu umieszczenia wszystkich kontrolek. To "stosy" każdej kontrolki poniżej drugiej.

Utwórz kontrolkę `<StackPanel>` po polu listy i umieść ją w kolumnie `1` wiersza `1` siatki . Dodaj inny atrybut o nazwie `Margin` z wartością `5,0,0,0`:

XAML

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
</Grid.ColumnDefinitions>

<Label>Names</Label>
<ListBox Grid.Row="1" x:Name="lstNames" />

<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">

</StackPanel>
```

Atrybut `Margin` był wcześniej używany w siatce, ale umieszczamy tylko w jednej wartości . `10` Teraz użyliśmy wartości `5,0,0,0` na panelu stosu. Margines jest typem `Thickness` i może interpretować obie wartości. Grubość definiuje przestrzeń wokół każdej strony prostokątnej ramki, **lewej**, **górnej**, **prawej**, **dolnej**, odpowiednio. Jeśli wartość marginesu jest pojedynczą wartością, używa tej wartości dla wszystkich czterech stron.

Następnie utwórz kontrolkę `<TextBox>` i `<Button>` w pliku `<StackPanel>`.

XAML

```
<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
    <Button x:Name="btnAdd" Margin="0,5,0,0">Add Name</Button>
</StackPanel>
```

Układ okna został ukończony. Jednak nasza aplikacja nie ma w niej żadnej logiki, aby rzeczywiście była funkcjonalna. Następnie musimy podłączyć zdarzenia sterujące do kodu i pobrać aplikację, aby rzeczywiście coś zrobić.

Dodawanie kodu dla zdarzenia Click

Utworzony `<Button>` obiekt zawiera `Click` zdarzenie, które jest wywoływane po naciśnięciu przycisku przez użytkownika. Możesz subskrybować to zdarzenie i dodawać

kod, aby dodać nazwę do pola listy. Podobnie jak w przypadku ustawienia właściwości kontrolki przez dodanie atrybutu XAML, możesz użyć atrybutu XAML, aby zasubskrybować zdarzenie. Ustaw atrybut na `Click` `ButtonAddName_Click`

XAML

```
<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
    <Button x:Name="btnAdd" Margin="0,5,0,0" Click="ButtonAddName_Click">Add
Name</Button>
</StackPanel>
```

Teraz musisz wygenerować kod procedury obsługi. Kliknij prawym przyciskiem myszy `ButtonAddName_Click` i wybierz pozycję **Przejdź do definicji**. Ta akcja generuje metodę w kodzie, która jest zgodna z wprowadzoną nazwą programu obsługi.

C#

```
private void ButtonAddName_Click(object sender, RoutedEventArgs e)
{

}
```

Następnie dodaj następujący kod, aby wykonać te trzy kroki:

1. Upewnij się, że pole tekstowe zawiera nazwę.
2. Sprawdź, czy nazwa wprowadzona w polu tekstowym jeszcze nie istnieje.
3. Dodaj nazwę do pola listy.

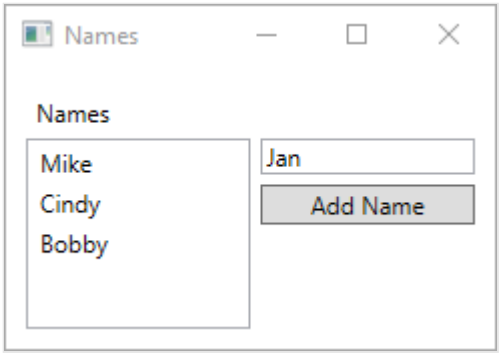
C#

```
private void ButtonAddName_Click(object sender, RoutedEventArgs e)
{
    if (!string.IsNullOrWhiteSpace(txtName.Text) &&
!lstNames.Items.Contains(txtName.Text))
    {
        lstNames.Items.Add(txtName.Text);
        txtName.Clear();
    }
}
```

Uruchamianie aplikacji

Teraz, gdy zdarzenie zostało zakodowane, możesz uruchomić aplikację, naciskając klawisz F5 lub wybierając pozycję **Debuguj**>**rozpocznij debugowanie** z menu. Zostanie

wyświetlone okno i możesz wprowadzić nazwę w polu tekstowym, a następnie dodać je, klikając przycisk.



Następne kroki

Dowiedz się więcej o Windows Presentation Foundation

Różnice w platformie .NET WPF

Artykuł • 18.02.2023 • Czas czytania: 2 min

W tym artykule opisano różnice między Windows Presentation Foundation (WPF) na platformie .NET i .NET Framework. WPF dla platformy .NET to rozwidlenie typu open source z oryginalnego WPF dla .NET Framework kodu źródłowego.

Istnieje kilka funkcji .NET Framework, których platforma .NET nie obsługuje. Aby uzyskać więcej informacji na temat nieobsługiwanych technologii, zobacz .NET Framework technologie niedostępne na platformie .NET.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

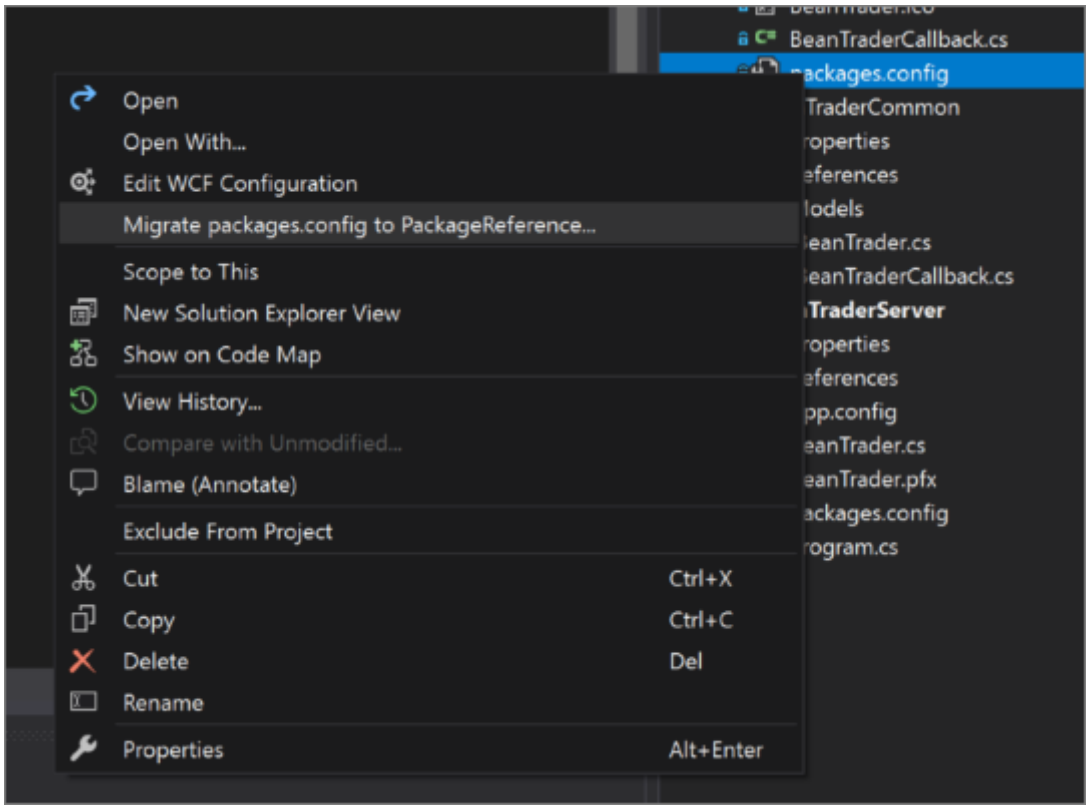
Projekty w stylu zestawu SDK

Platforma .NET używa plików projektu w stylu zestawu SDK. Te pliki projektu różnią się od tradycyjnych plików projektów .NET Framework zarządzanych przez program Visual Studio. Aby przeprowadzić migrację aplikacji .NET Framework WPF na platformę .NET, należy przekonwertować projekty. Aby uzyskać więcej informacji, zobacz Migrowanie aplikacji WPF do platformy .NET 5.

Odwołania do pakietu NuGet

Jeśli aplikacja .NET Framework wyświetla listę zależności NuGet w pliku *packages.config*, przeprowadź migrację <PackageReference> do formatu:

1. W programie Visual Studio otwórz okienko **Eksplorator rozwiązań**.
2. W projekcie WPF kliknij prawym przyciskiem myszy **packages.config**>**Migrowanie packages.config do pozycji PackageReference**.



Zostanie wyświetlone okno dialogowe przedstawiające obliczone zależności NuGet najwyższego poziomu i z pytaniem, które inne pakiety NuGet powinny zostać podniesione do najwyższego poziomu. Wybierz **przycisk OK** , a plik *packages.config* zostanie usunięty z projektu, a `<PackageReference>` elementy zostaną dodane do pliku projektu.

Gdy projekt używa `<PackageReference>`programu , pakiety nie są przechowywane lokalnie w folderze *Packages* , są przechowywane globalnie. Otwórz plik projektu i usuń wszystkie `<Analyzer>` elementy, które odniosły się do folderu *Packages* . Te analizatory są automatycznie dołączane do odwołań pakietów NuGet.

Zabezpieczenia dostępu kodu

Zabezpieczenia dostępu do kodu (CAS) nie są obsługiwane przez platformę .NET. Wszystkie funkcje związane z usługą CAS są traktowane zgodnie z założeniem pełnego zaufania. WPF dla platformy .NET usuwa kod związany z systemem CAS. Publiczna powierzchnia interfejsu API tych typów nadal istnieje, aby upewnić się, że wywołania do tych typów kończą się powodzeniem.

Publicznie zdefiniowane typy związane z systemem CAS zostały przeniesione z zestawów WPF i do podstawowych zestawów bibliotek .NET. Zestawy WPF mają zestawy typu przekazujące typy do nowej lokalizacji przeniesionych typów.

Zestaw źródłowy	Zestaw docelowy	Typ

Zestaw źródłowy	Zestaw docelowy	Typ
WindowsBase.dll	*System.Security.Permissions.dll*	MediaPermission MediaPermissionAttribute MediaPermissionAudio MediaPermissionImage MediaPermissionVideo WebBrowserPermission WebBrowserPermissionAttribute WebBrowserPermissionLevel
System.Xaml.dll	*System.Security.Permissions.dll*	XamlLoadPermission
System.Xaml.dll	*System.Windows.Extension.dll*	XamlAccessLevel

Uwaga

Aby zminimalizować tarcie przenoszenia, funkcje przechowywania i pobierania informacji związanych z następującymi właściwościami zostały zachowane w typie `XamlAccessLevel` .

- `PrivateAccessToTypeName`
- `AssemblyNameString`

Następne kroki

- Migrowanie aplikacji WPF do platformy .NET 5

Migrowanie aplikacji WPF do platformy .NET Core

Artykuł • 10.02.2023 • Czas czytania: 25 min

W tym artykule opisano kroki niezbędne do przeprowadzenia migracji aplikacji Windows Presentation Foundation (WPF) z .NET Framework do platformy .NET Core 3.0. Jeśli nie masz aplikacji WPF do portu, ale chcesz wypróbować ten proces, możesz użyć przykładowej aplikacji **Bean Trader** dostępnej w witrynie GitHub. Oryginalna aplikacja (przeznaczona dla .NET Framework 4.7.2) jest dostępna w folderze NetFx\BeanTraderClient. Najpierw wyjaśnimy kroki niezbędne do ogólnego przenoszenia aplikacji, a następnie omówimy konkretne zmiany dotyczące przykładu **Bean Trader** .

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wypróbowywanie asystenta uaktualniania

Asystent uaktualniania platformy .NET to narzędzie wiersza polecenia, które można uruchamiać w różnych rodzajach aplikacji platformy .NET Framework. Służy od do pomagania w uaktualnianiu aplikacji platformy .NET Framework do platformy .NET 5. Po uruchomieniu narzędzia w większości przypadków aplikacja będzie wymagać wykonania dodatkowych czynności w celu ukończenia migracji. Narzędzie obejmuje instalację analizatorów, które mogą pomóc w ukończeniu migracji.

Aby uzyskać więcej informacji, zobacz Uaktualnianie aplikacji Windows Forms do platformy .NET 5 przy użyciu Asystenta uaktualniania platformy .NET.

Wymagania wstępne

Aby przeprowadzić migrację do platformy .NET Core, musisz najpierw:

1. Omówienie i aktualizowanie zależności nuGet:
   a. Uaktualnij zależności NuGet, aby użyć `<PackageReference>` formatu .
   b. Przejrzyj zależności NuGet najwyższego poziomu pod kątem zgodności platformy .NET Core lub .NET Standard.
   c. Uaktualnij pakiety NuGet do nowszych wersji.

   d. Użyj analizatora portability platformy .NET , aby zrozumieć zależności platformy .NET.

2. Przeprowadź migrację pliku projektu do nowego formatu w stylu zestawu SDK:
   a. Wybierz, czy mają dotyczyć platformy .NET Core i .NET Framework, czy tylko platformy .NET Core.
   b. Skopiuj odpowiednie właściwości i elementy pliku projektu do nowego pliku projektu.

3. Rozwiązywanie problemów z kompilacją:
   a. Dodaj odwołanie do pakietu Microsoft.Windows.Compatibility .
   b. Znajdowanie i naprawianie różnic na poziomie interfejsu API.
   c. Usuń *sekcjeapp.config* inne niż `appSettings` lub `connectionStrings`.
   d. W razie potrzeby w razie potrzeby wygeneruj wygenerowany kod.

4. Testowanie środowiska uruchomieniowego:
   a. Upewnij się, że portowana aplikacja działa zgodnie z oczekiwaniami.
   b. Uważaj na wyjątki NotSupportedException .

Informacje o przykładzie

Ten artykuł odwołuje się do przykładowej aplikacji Bean Trader , ponieważ używa różnych zależności podobnych do rzeczywistych aplikacji WPF. Aplikacja nie jest duża, ale ma być krokiem od "Hello world" pod względem złożoności. Aplikacja demonstruje niektóre problemy, które użytkownicy mogą napotkać podczas przenoszenia rzeczywistych aplikacji. Aplikacja komunikuje się z usługą WCF, aby mogła działać prawidłowo, należy również uruchomić projekt BeanTraderServer (dostępny w tym samym repozytorium GitHub) i upewnić się, że konfiguracja BeanTraderClient wskazuje prawidłowy punkt końcowy. (Domyślnie w przykładzie założono, że serwer jest uruchomiony na tej samej maszynie w `http://localhost:8090`systemie , co będzie prawdziwe w przypadku lokalnego uruchomienia serwera BeanTraderServer).

Należy pamiętać, że ta przykładowa aplikacja ma na celu zademonstrowanie wyzwań i rozwiązań związanych z przenoszeniem platformy .NET Core. Nie ma na celu zademonstrowania najlepszych rozwiązań WPF. W rzeczywistości celowo zawiera pewne antywzór, aby upewnić się, że napotkasz co najmniej kilka interesujących wyzwań podczas przenoszenia.

Przygotowanie

Głównym wyzwaniem migracji aplikacji .NET Framework do platformy .NET Core jest to, że jej zależności mogą działać inaczej lub w ogóle nie. Migracja jest znacznie łatwiejsza niż kiedyś; wiele pakietów NuGet jest teraz przeznaczonych dla platformy .NET Standard. Począwszy od platformy .NET Core 2.0, obszary powierzchni .NET Framework i .NET Core stają się podobne. Mimo to pozostają pewne różnice (zarówno w obsłudze pakietów NuGet, jak i w dostępnych interfejsach API platformy .NET). Pierwszym krokiem migracji jest przejrzenie zależności aplikacji i upewnienie się, że odwołania są w formacie łatwo migrowanym do platformy .NET Core.

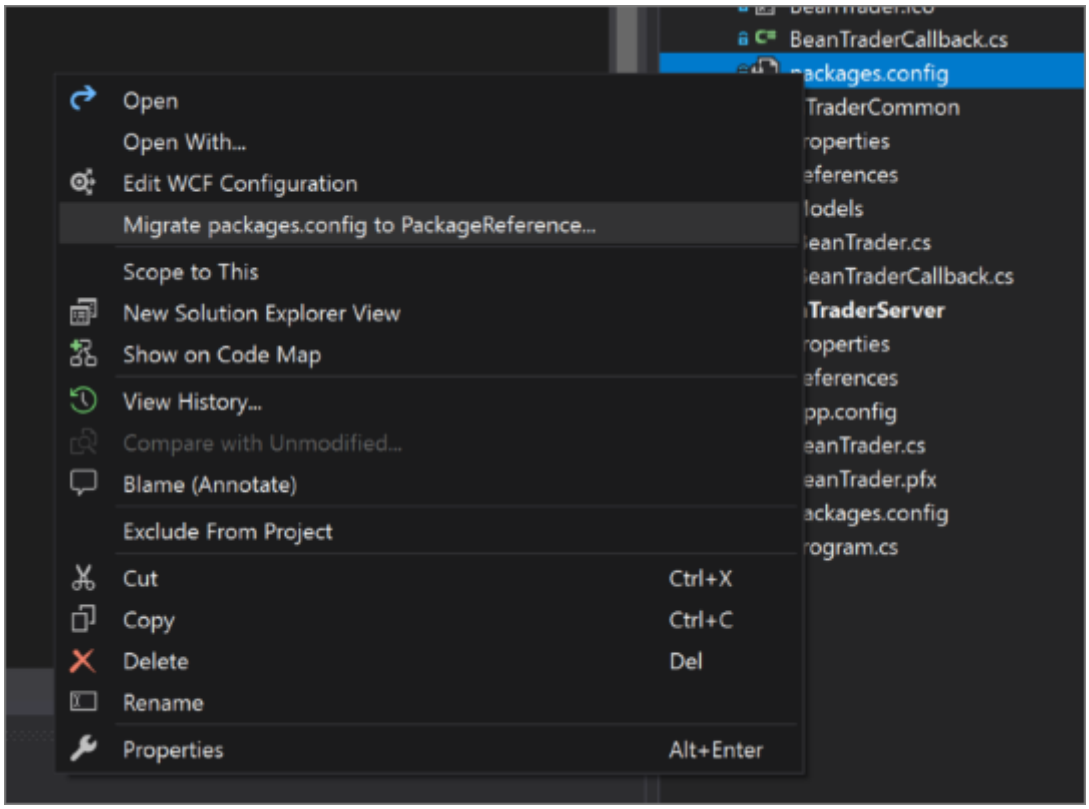
Uaktualnianie do odwołań nuGet `<PackageReference>`

Starsze projekty .NET Framework zwykle wyświetlają ich zależności NuGet w pliku *packages.config*. Nowy format pliku projektu w stylu zestawu SDK odwołuje się do pakietów NuGet jako <PackageReference> elementów w samym pliku csproj, a nie w oddzielnym pliku konfiguracji.

Podczas migracji istnieją dwie zalety korzystania z `<PackageReference>`odwołań do stylu :

- Jest to styl odwołania NuGet wymagany dla nowego pliku projektu platformy .NET Core. Jeśli już używasz programu `<PackageReference>`, te elementy pliku projektu można skopiować i wkleić bezpośrednio do nowego projektu.
- W przeciwieństwie do pliku `<PackageReference>` packages.config elementy odwołują się tylko do zależności najwyższego poziomu, od których zależy bezpośrednio projekt. Wszystkie inne przechodnie pakiety NuGet zostaną określone w czasie przywracania i zarejestrowane w automatycznie wygenerowanym pliku obj\project.assets.json. Dzięki temu znacznie łatwiej jest określić, jakie zależności ma projekt, co jest przydatne podczas określania, czy niezbędne zależności będą działać na platformie .NET Core, czy nie.

Pierwszym krokiem migracji aplikacji .NET Framework do platformy .NET Core jest zaktualizowanie jej do używania `<PackageReference>` odwołań NuGet. Program Visual Studio sprawia, że jest to proste. Po prostu kliknij prawym przyciskiem myszy plik *packages.config* projektu w **Eksplorator rozwiązań** programu Visual Studio, a następnie wybierz pozycję **Migruj packages.config do opcji PackageReference**.



Zostanie wyświetlone okno dialogowe przedstawiające obliczone zależności NuGet najwyższego poziomu i z pytaniem, które inne pakiety NuGet powinny zostać podniesione do najwyższego poziomu. Żaden z tych innych pakietów nie musi być najwyższym poziomem dla przykładu Bean Trader, więc można usunąć zaznaczenie wszystkich tych pól. Następnie kliknij **przycisk OK** , a plik *packages.config* zostanie usunięty, a `<PackageReference>` elementy zostaną dodane do pliku projektu.

`<PackageReference>`Odwołania -style nie przechowują pakietów NuGet lokalnie w folderze packages. Zamiast tego są one przechowywane globalnie jako optymalizacja. Po zakończeniu migracji zmodyfikuj plik csproj i usuń wszystkie `<Analyzer>` elementy odwołujące się do analizatorów, które wcześniej pochodziły z pliku . *packages* directory. Nie martw się; ponieważ nadal masz odwołania do pakietu NuGet, analizatory zostaną uwzględnione w projekcie. Wystarczy wyczyścić stare elementy w stylu `<Analyzer>` packages.config.

Przeglądanie pakietów NuGet

Teraz, gdy zobaczysz pakiety NuGet najwyższego poziomu, od których zależy projekt, możesz sprawdzić, czy te pakiety są dostępne na platformie .NET Core. Możesz określić, czy pakiet obsługuje platformę .NET Core, sprawdzając jego zależności od nuget.org. Witryna utworzona przez społeczność fuget.org wyświetla te informacje w górnej części strony informacji o pakiecie.

W przypadku określania wartości docelowej dla platformy .NET Core 3.0 wszystkie pakiety przeznaczone dla platformy .NET Core lub .NET Standard powinny działać (ponieważ platforma .NET Core implementuje obszar powierzchni platformy .NET Standard). W niektórych przypadkach określona wersja używanego pakietu nie będzie dotyczyć platformy .NET Core ani .NET Standard, ale nowsze wersje. W takim przypadku należy rozważyć uaktualnienie do najnowszej wersji pakietu.

Można również używać pakietów przeznaczonych dla .NET Framework, ale wprowadza to pewne ryzyko. Platforma .NET Core do .NET Framework zależności jest dozwolona, ponieważ platformy .NET Core i obszary powierzchni .NET Framework są na tyle podobne, że takie zależności *często* działają. Jeśli jednak pakiet spróbuje użyć interfejsu API platformy .NET, który nie jest obecny na platformie .NET Core, wystąpi wyjątek środowiska uruchomieniowego. W związku z tym należy odwoływać się tylko .NET Framework pakietów, gdy nie są dostępne żadne inne opcje i zrozumieć, że nakłada to obciążenie testowe.

Jeśli istnieją pakiety, do których nie odwołuje się platforma .NET Core lub .NET Standard, należy zastanowić się nad innymi alternatywami:

- Czy istnieją inne podobne pakiety, których można użyć zamiast tego? Czasami autorzy nuGet publikują oddzielnie ".. Wersje ich bibliotek core przeznaczone specjalnie dla platformy .NET Core. Pakiety biblioteki przedsiębiorstwa są przykładem publikowania społeczności ". Alternatywy NetCore. W innych przypadkach nowsze zestawy SDK dla określonej usługi (czasami z różnymi nazwami pakietów) są dostępne dla platformy .NET Standard. Jeśli nie są dostępne żadne alternatywy, możesz kontynuować korzystanie z pakietów docelowych .NET Framework, mając na uwadze, że należy je dokładnie przetestować po uruchomieniu na platformie .NET Core.

Przykład Bean Trader ma następujące zależności NuGet najwyższego poziomu:

- **Castle.Windsor, wersja 4.1.1**

  Ten pakiet jest przeznaczony dla platformy .NET Standard 1.6, więc działa na platformie .NET Core.

- **Microsoft.CodeAnalysis.FxCopAnalyzers, wersja 2.6.3**
  Jest to metapakiet, więc nie jest od razu oczywiste, które platformy obsługuje, ale dokumentacja wskazuje, że jego najnowsza wersja (2.9.2) będzie działać zarówno dla platform .NET Framework, jak i .NET Core.

- **Nito.AsyncEx, wersja 4.0.1**

Ten pakiet nie jest przeznaczony dla platformy .NET Core, ale nowsza wersja 5.0. Jest to powszechne w przypadku migracji, ponieważ wiele pakietów NuGet dodało ostatnio obsługę platformy .NET Standard, ale starsze wersje projektu będą dotyczyć tylko .NET Framework. Jeśli różnica między wersjami jest tylko różnicą wersji pomocniczej, często łatwo jest przeprowadzić uaktualnienie do nowszej wersji. Ponieważ jest to poważna zmiana wersji, należy zachować ostrożność podczas uaktualniania, ponieważ w pakiecie mogą występować istotne zmiany. Istnieje jednak droga naprzód, która jest dobra.

- MahApps.Metro, wersja 1.6.5

  Ten pakiet nie jest również przeznaczony dla platformy .NET Core, ale ma nowsze wersje wstępne (2.0-alpha), które to robi. Ponownie, musisz zwrócić uwagę na przełomowe zmiany, ale nowszy pakiet jest zachęcający.

Zależności NuGet przykładu Bean Trader są przeznaczone dla platformy .NET Standard/.NET Core lub mają nowsze wersje, więc w tym miejscu prawdopodobnie nie będą występować żadne problemy z blokowaniem.

Uaktualnianie pakietów NuGet

Jeśli to możliwe, warto uaktualnić wersje wszystkich pakietów przeznaczonych tylko dla platformy .NET Core lub .NET Standard z nowszymi wersjami w tym momencie (z projektem nadal przeznaczonym dla .NET Framework) w celu odnalezienia i rozwiązania wszelkich zmian powodujących niezgodność.

Jeśli wolisz nie wprowadzać żadnych istotnych zmian w istniejącej .NET Framework wersji aplikacji, może to poczekać, aż zostanie wyświetlony nowy plik projektu przeznaczony dla platformy .NET Core. Jednak uaktualnienie pakietów NuGet do wersji zgodnych z platformą .NET Core z wyprzedzeniem ułatwia proces migracji po utworzeniu nowego pliku projektu i zmniejsza liczbę różnic między .NET Framework a wersjami platformy .NET Core aplikacji.

Korzystając z przykładu Bean Trader, wszystkie niezbędne uaktualnienia można łatwo (przy użyciu menedżera pakietów NuGet programu Visual Studio) z jednym wyjątkiem: uaktualnianie z **MahApps.Metro 1.6.5** do **2.0** ujawnia istotne zmiany związane z interfejsami API zarządzania motywami i akcentami.

W idealnym przypadku aplikacja zostanie zaktualizowana tak, aby korzystała z nowszej wersji pakietu (ponieważ jest to bardziej prawdopodobne, że będzie działać na platformie .NET Core). Jednak w niektórych przypadkach może to nie być możliwe. W takich przypadkach nie uaktualnij **MahApps.Metro**, ponieważ niezbędne zmiany nie są proste, a ten samouczek koncentruje się na migracji do platformy .NET Core 3, a nie do

MahApps.Metro 2. Ponadto jest to zależność .NET Framework niskiego ryzyka, ponieważ aplikacja Bean Trader wykonuje tylko niewielką część **MahApps.Metro**. Będzie to oczywiście wymagać testowania, aby upewnić się, że wszystko działa po zakończeniu migracji. Gdyby był to rzeczywisty scenariusz, dobrze byłoby zgłosić problem, aby śledzić pracę, aby przejść do **MahApps.Metro** w wersji 2.0, ponieważ nie robi migracji teraz pozostawia jakiś dług techniczny.

Po zaktualizowaniu pakietów NuGet do najnowszych wersji `<PackageReference>` grupa elementów w pliku projektu przykładu Bean Trader powinna wyglądać następująco.

XML

```xml
<ItemGroup>
  <PackageReference Include="Castle.Windsor">
    <Version>4.1.1</Version>
  </PackageReference>
  <PackageReference Include="MahApps.Metro">
    <Version>1.6.5</Version>
  </PackageReference>
  <PackageReference Include="Microsoft.CodeAnalysis.FxCopAnalyzers">
    <Version>2.9.2</Version>
  </PackageReference>
  <PackageReference Include="Nito.AsyncEx">
    <Version>5.0.0</Version>
  </PackageReference>
</ItemGroup>
```

analiza przenośności .NET Framework

Po zrozumieniu stanu zależności NuGet projektu kolejną rzeczą, którą należy wziąć pod uwagę, jest .NET Framework zależności interfejsu API. Narzędzie Analizator portability analyzer platformy .NET jest przydatne do zrozumienia, które interfejsy API platformy .NET używane przez projekt są dostępne na innych platformach .NET.

Narzędzie jest dostarczane jako wtyczka programu Visual Studio, narzędzie wiersza polecenia lub opakowane w prosty graficzny interfejs użytkownika, który upraszcza jego opcje. Więcej informacji na temat korzystania z narzędzia .NET Portability Analyzer (port interfejsu API) przy użyciu graficznego interfejsu użytkownika można przeczytać we wpisie w blogu Przenoszenie aplikacji klasycznych do platformy .NET Core . Jeśli wolisz korzystać z wiersza polecenia, niezbędne kroki to:

1. Pobierz analizator portability platformy .NET , jeśli jeszcze go nie masz.
2. Upewnij się, że aplikacja .NET Framework do pomyślnego przeniesienia kompilacji (jest to dobry pomysł przed migracją niezależnie od tego).

3. Uruchom port interfejsu API z wierszem polecenia w następujący sposób.

Konsola

```
ApiPort.exe analyze -f <PathToBeanTraderBinaries> -r html -r excel -t
".NET Core"
```

Argument `-f` określa ścieżkę zawierającą pliki binarne do analizy. Argument `-r` określa żądany format pliku wyjściowego. Argument `-t` określa platformę .NET do analizowania użycia interfejsu API. W takim przypadku potrzebujesz platformy .NET Core.

Po otwarciu raportu HTML pierwsza sekcja będzie zawierać listę wszystkich analizowanych plików binarnych i procent używanych przez nie interfejsów API platformy .NET. Wartość procentowa nie jest sama w sobie zrozumiała. Bardziej przydatne jest sprawdzenie brakujących określonych interfejsów API. W tym celu wybierz nazwę zestawu lub przewiń w dół do raportów dla poszczególnych zestawów.

Skoncentruj się na zestawach, dla których jesteś właścicielem kodu źródłowego. Na przykład w raporcie Bean Trader ApiPort znajduje się wiele plików binarnych, ale większość z nich należy do pakietów NuGet. `Castle.Windsor` pokazuje, że zależy to od niektórych interfejsów API System.Web, których brakuje na platformie .NET Core. Nie jest to problem, ponieważ wcześniej zweryfikowano, że `Castle.Windsor` obsługuje platformę .NET Core. Pakiety NuGet często mają różne pliki binarne do użycia z różnymi platformami .NET, więc niezależnie od tego, czy wersja `Castle.Windsor` .NET Framework korzysta z interfejsów API System.Web, czy nie ma znaczenia, o ile pakiet jest również przeznaczony dla platformy .NET Standard lub .NET Core (co robi).

W przykładzie Bean Trader jedynym plikiem binarnym, który należy wziąć pod uwagę, jest **BeanTraderClient** , a raport pokazuje, że brakuje tylko dwóch interfejsów API platformy .NET: `System.ServiceModel.ClientBase<T>.Close` i `System.ServiceModel.ClientBase<T>.Open`.

BeanTraderClient, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null (.NETFramework,Version=v4.7.2)

Target type	.NET Core,Version=v3.0	Recommended changes
System.ServiceModel.ClientBase`1	✔	
Close	✘	
Open	✘	

Jest mało prawdopodobne, aby problemy z blokowaniem były spowodowane tym, że interfejsy API klienta WCF są (głównie) obsługiwane na platformie .NET Core, dlatego muszą istnieć alternatywy dla tych centralnych interfejsów API. W rzeczywistości, patrząc na `System.ServiceModel` obszar powierzchni platformy .NET Core (przy użyciu),

https://apisof.net widać, że istnieją alternatywy asynchroniczne na platformie .NET Core.

Na podstawie tego raportu i poprzedniej analizy zależności NuGet wygląda na to, że nie powinno występować żadne poważne problemy z migracją przykładu Bean Trader do platformy .NET Core. Możesz przystąpić do następnego kroku, w którym faktycznie rozpoczniesz migrację.

Migrowanie pliku projektu

Jeśli aplikacja nie korzysta z nowego formatu pliku projektu w stylu zestawu SDK, potrzebny będzie nowy plik projektu, który będzie przeznaczony dla platformy .NET Core. Możesz zastąpić istniejący plik csproj lub, jeśli wolisz zachować istniejący projekt nietknięty w bieżącym stanie, możesz dodać nowy plik csproj przeznaczony dla platformy .NET Core. Wersje aplikacji dla platformy .NET Framework i platformy .NET Core można tworzyć przy użyciu jednego pliku projektu w stylu zestawu SDK z wielowersyjnością (określając wiele `<TargetFrameworks>` obiektów docelowych).

Aby utworzyć nowy plik projektu, możesz utworzyć nowy projekt WPF w programie Visual Studio lub użyć `dotnet new wpf` polecenia w katalogu tymczasowym, aby wygenerować plik projektu, a następnie skopiować/zmienić jego nazwę na poprawną lokalizację. Istnieje również narzędzie utworzone przez społeczność , CsprojToVs2017 , które może zautomatyzować migrację niektórych plików projektu. Narzędzie jest przydatne, ale nadal potrzebuje człowieka do przejrzenia wyników, aby upewnić się, że wszystkie szczegóły migracji są poprawne. Jednym z konkretnych obszarów, które narzędzie nie obsługuje optymalnie, jest migrowanie pakietów NuGet z *plikówpackages.config* . Jeśli narzędzie zostanie uruchomione w pliku projektu, który nadal używa pliku *packages.config* do odwoływanie się do pakietów NuGet, zostanie automatycznie zmigrowane do `<PackageReference>` elementów, ale doda `<PackageReference>` elementy dla *wszystkich* pakietów zamiast tylko pakietów najwyższego poziomu. Jeśli przeprowadzono już migrację do `<PackageReference>` elementów za pomocą programu Visual Studio (tak jak zostało to zrobione w tym przykładzie), narzędzie może pomóc w pozostałej części konwersji. Podobnie jak Scott Hanselman zaleca w swoim wpisie na blogu migrowanie plików csproj , przenoszenie ręcznie jest edukacyjne i daje lepsze wyniki, jeśli masz tylko kilka projektów do portu. Jeśli jednak przenosisz dziesiątki lub setki plików projektu, to narzędzie takie jak CsprojToVs2017 może być pomocną pomocą.

Aby utworzyć nowy plik projektu dla przykładu Bean Trader, uruchom polecenie `dotnet new wpf` w katalogu tymczasowym i przenieś wygenerowany plik *csproj* do folderu *BeanTraderClient* i zmień jego nazwę **na BeanClient.Core.csproj**.

Ponieważ nowy format pliku projektu automatycznie zawiera pliki C#, pliki *resx* i pliki XAML, które znajduje w katalogu lub w jego katalogu, plik projektu jest już prawie gotowy! Aby zakończyć migrację, otwórz obok siebie stare i nowe pliki projektu i przejrzyj stare, aby sprawdzić, czy należy migrować jakiekolwiek informacje, które zawiera. W przykładowym przypadku Bean Trader należy skopiować następujące elementy do nowego projektu:

- Właściwości `<RootNamespace>`, `<AssemblyName>` i `<ApplicationIcon>` powinny zostać skopiowane.
- Musisz również dodać `<GenerateAssemblyInfo>false</GenerateAssemblyInfo>` właściwość do nowego pliku projektu, ponieważ przykład Bean Trader zawiera atrybuty na poziomie zestawu (na przykład `[AssemblyTitle]`) w pliku AssemblyInfo.cs. Domyślnie nowe projekty w stylu zestawu SDK automatycznie generują te atrybuty na podstawie właściwości w pliku csproj. Ponieważ nie chcesz, aby miało to miejsce w tym przypadku (automatycznie wygenerowane atrybuty powodują konflikt z atrybutami z pliku AssemblyInfo.cs), należy wyłączyć automatycznie wygenerowane atrybuty za pomocą `<GenerateAssemblyInfo>` polecenia .
- Mimo że pliki *resx* są automatycznie dołączane jako zasoby osadzone, inne `<Resource>` elementy, takie jak obrazy, nie są. Dlatego skopiuj `<Resource>` elementy do osadzania plików obrazów i ikon. Odwołania png do pojedynczego wiersza można uprościć przy użyciu obsługi nowego formatu pliku projektu dla wzorców symboli globbingowych: `<Resource Include="**\*.png" />`.
- `<None>` Podobnie elementy są dołączane automatycznie, ale domyślnie nie są kopiowane do katalogu wyjściowego. Ponieważ projekt Bean Trader zawiera `<None>` element skopiowany do katalogu wyjściowego (przy użyciu `PreserveNewest` zachowań), należy zaktualizować automatycznie wypełniony `<None>` element dla tego pliku, w ten sposób.

  XML

  ```xml
  <None Update="BeanTrader.pfx">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
  ```

- Przykład bean trader zawiera plik XAML (Default.Accent.xaml) jako `Content` (a nie jako `Page`) ponieważ motywy i akcenty zdefiniowane w tym pliku są ładowane z pliku XAML w czasie wykonywania, a nie osadzone w samej aplikacji. Nowy system projektu automatycznie dołącza ten plik jako `<Page>`, jednak ponieważ jest to plik

XAML. Dlatego należy usunąć plik XAML jako stronę (`<Page Remove="**\Default.Accent.xaml" />`) i dodać go jako zawartość.

```XML
<Content Include="Resources\Themes\Default.Accent.xaml">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</Content>
```

- Na koniec dodaj odwołania NuGet, kopiując element `<ItemGroup>` ze wszystkimi `<PackageReference>` elementami. Jeśli pakiety NuGet nie zostały wcześniej uaktualnione do wersji zgodnych z platformą .NET Core, możesz to zrobić teraz, gdy odwołania do pakietu znajdują się w projekcie specyficznym dla platformy .NET Core.

W tym momencie powinno być możliwe dodanie nowego projektu do rozwiązania BeanTrader i otwarcie go w programie Visual Studio. Projekt powinien wyglądać poprawnie w **Eksplorator rozwiązań** i `dotnet restore BeanTraderClient.Core.csproj` powinien pomyślnie przywrócić pakiety (z dwoma oczekiwanymi ostrzeżeniami dotyczącymi wersji MahApps.Metro, której używasz .NET Framework).

Chociaż można zachować oba pliki projektu obok siebie (i może być nawet pożądane, jeśli chcesz nadal tworzyć stary projekt dokładnie tak, jak to było), komplikuje proces migracji (dwa projekty będą próbować używać tego samego pojemnika i folderów obj) i zwykle nie jest konieczne. Jeśli chcesz skompilować obiekty docelowe platformy .NET Core i .NET Framework, możesz zastąpić `<TargetFramework>netcoreapp3.0</TargetFramework>` właściwość w nowym pliku `<TargetFrameworks>netcoreapp3.0;net472</TargetFrameworks>` projektu zamiast . W przypadku przykładu Bean Trader usuń stary plik projektu (BeanClient.csproj), ponieważ nie jest już potrzebny. Jeśli wolisz zachować oba pliki projektu, pamiętaj o utworzeniu ich do różnych ścieżek wyjściowych i pośrednich.

Rozwiązywanie problemów z kompilacją

Trzeci krok procesu przenoszenia polega na utworzeniu projektu. Niektóre aplikacje będą już kompilowane pomyślnie po przekonwertowaniu pliku projektu na projekt w stylu zestawu SDK. Jeśli tak jest w twojej aplikacji, gratulacje! Możesz przejść do kroku 4. Inne aplikacje będą potrzebować pewnych aktualizacji w celu utworzenia ich dla platformy .NET Core. Jeśli spróbujesz uruchomić `dotnet build` przykładowy projekt Bean Trader, na przykład (lub skompilujesz go w programie Visual Studio), wystąpi wiele błędów, ale szybko je naprawisz.

Odwołania System.ServiceModel i Microsoft.Windows.Compatibility

W typowym źródle błędów brakuje odwołań dla interfejsów API, które są dostępne dla platformy .NET Core, ale nie są automatycznie uwzględniane w metapakiecie aplikacji platformy .NET Core. Aby rozwiązać ten problem, należy odwołać się do `Microsoft.Windows.Compatibility` pakietu. Pakiet zgodności zawiera szeroki zestaw interfejsów API, które są wspólne w aplikacjach klasycznych systemu Windows, takich jak klient WCF, usługi katalogowe, rejestr, konfiguracja, interfejsy API list ACL i inne.

W przykładzie Bean Trader większość błędów kompilacji jest spowodowana brakującymi System.ServiceModel typami. Można je rozwiązać, odwołując się do niezbędnych pakietów NuGet WCF. Interfejsy API klienta programu WCF są jednak jednymi z tych, które znajdują się w `Microsoft.Windows.Compatibility` pakiecie, więc odwoływanie się do pakietu zgodności jest jeszcze lepszym rozwiązaniem (ponieważ rozwiązuje również wszelkie problemy związane z interfejsami API, a także rozwiązania problemów z programem WCF udostępnianych przez pakiet zgodności). Pakiet `Microsoft.Windows.Compatibility` pomaga w większości scenariuszy przenoszenia WPF i WinForms platformy .NET Core 3.0. Po dodaniu odwołania NuGet do `Microsoft.Windows.Compatibility`programu pozostaje tylko jeden błąd kompilacji.

Czyszczenie nieużywanych plików

Jeden typ problemu z migracją, który występuje często, dotyczy plików C# i XAML, które nie zostały wcześniej uwzględnione w kompilacji pobieranej przez nowe projekty w stylu zestawu SDK, które zawierają *wszystkie* źródła automatycznie.

Następny błąd kompilacji widoczny w przykładzie Bean Trader odnosi się do nieprawidłowej implementacji *interfejsu w pliku OldUnusedViewModel.cs*. Nazwa pliku to wskazówka, ale podczas inspekcji okaże się, że ten plik źródłowy jest niepoprawny. Wcześniej nie powodowała problemów, ponieważ nie została uwzględniona w oryginalnym projekcie .NET Framework. Pliki źródłowe, które były obecne na dysku, ale nieuwzględnione w starym *pliku csproj* , są teraz automatycznie dołączane.

W przypadku takich problemów jednorazowych można łatwo porównać plik z poprzednim *csproj* , aby potwierdzić, że plik nie jest potrzebny, a następnie `<Compile Remove="" />` albo, jeśli plik źródłowy nie jest już potrzebny, usuń go. W takim przypadku można bezpiecznie usunąć *plik OldUnusedViewModel.cs*.

Jeśli masz wiele plików źródłowych, które należy wykluczyć w ten sposób, możesz wyłączyć automatyczne dołączanie plików języka C#, ustawiając

`<EnableDefaultCompileItems>` właściwość na false w pliku projektu. Następnie możesz skopiować `<Compile Include>` elementy ze starego pliku projektu do nowego w celu uwzględnienia tylko źródeł kompilacji. `<EnableDefaultPageItems>` Podobnie można użyć do wyłączenia automatycznego dołączania stron XAML i `<EnableDefaultItems>` może kontrolować obie z jedną właściwością.

Krótka bok dla kompilatorów wieloprzepustowych

Po usunięciu pliku obraźliwego z przykładu Bean Trader możesz ponownie skompilować i uzyskać cztery błędy. Nie masz jeszcze jednego? Dlaczego liczba błędów wzrosła? Kompilator języka C# jest kompilatorem wieloprzepustowym. Oznacza to, że przechodzi przez każdy plik źródłowy dwa razy. Najpierw kompilator po prostu analizuje metadane i deklaracje w każdym pliku źródłowym i identyfikuje wszelkie problemy na poziomie deklaracji. Są to błędy, które zostały naprawione. Następnie ponownie przejdą przez kod, aby skompilować źródło języka C# w języku IL; są to drugi zestaw błędów, które widzisz teraz.

> ⓘ **Uwaga**
>
> Kompilator języka C# wykonuje **więcej niż dwa przebiegi**, ale wynikiem końcowym jest to, że błędy kompilatora dla dużych zmian kodu, takich jak te, zwykle występują w dwóch falach.

Poprawki zależności innych firm (Castle.Windsor)

Inną klasą problemu, która pojawia się w niektórych scenariuszach migracji, jest różnica interfejsu API między wersjami .NET Framework i .NET Core zależności. Nawet jeśli pakiet NuGet jest przeznaczony zarówno .NET Framework, jak i .NET Standard lub .NET Core, mogą istnieć różne biblioteki do użycia z różnymi obiektami docelowymi platformy .NET. Dzięki temu pakiety mogą obsługiwać wiele różnych platform .NET, które mogą wymagać różnych implementacji. Oznacza to również, że mogą istnieć niewielkie różnice interfejsu API w bibliotekach podczas określania wartości docelowej dla różnych platform .NET.

Następny zestaw błędów, które zostaną wyświetlone w przykładzie Bean Trader, są powiązane z `Castle.Windsor` interfejsami API. Projekt .NET Core Bean Trader używa tej samej wersji `Castle.Windsor` co projekt przeznaczony dla .NET Framework (4.1.1), ale implementacje dla tych dwóch platform są nieco inne.

W takim przypadku zostaną wyświetlone następujące problemy, które należy rozwiązać:

1. `Castle.MicroKernel.Registration.Classes.FromThisAssembly` nie jest dostępna na platformie .NET Core. Dostępny jest jednak podobny interfejs API `Classes.FromAssemblyContaining` , więc możemy zastąpić oba zastosowania `Classes.FromThisAssembly()` wywołaniami do `Classes.FromAssemblyContaining(t)`metody , gdzie `t` jest typem wywołującym.
2. Podobnie w pliku *Bootstrapper.cs*, `Castle.Windsor.Installer.FromAssembly`. Jest to niedostępne na platformie .NET Core. Zamiast tego wywołanie to można zastąpić ciągiem `FromAssembly.Containing(typeof(Bootstrapper))`.

Aktualizowanie użycia klienta WCF

Po naprawieniu `Castle.Windsor` różnic ostatni pozostały błąd kompilacji w projekcie .NET Core Bean Trader jest taki, że `BeanTraderServiceClient` (który pochodzi z `DuplexClientBase`) nie ma `Open` metody. Nie jest to zaskakujące, ponieważ jest to interfejs API wyróżniony przez analizator portability platformy .NET na początku tego procesu migracji. Patrząc na naszą uwagę na `BeanTraderServiceClient` większy problem, choć. Ten klient WCF został automatycznie wygenerowany przez narzędzie Svcutil.exe .

Klienci programu WCF wygenerowani przez program Svcutil są przeznaczone do użycia w .NET Framework.

Rozwiązania korzystające z klientów WCF wygenerowanych przez program svcutil będą musiały ponownie wygenerować klientów zgodnych z platformą .NET Standard do użycia z platformą .NET Core. Jednym z głównych powodów, dla których starzy klienci nie będą działać, jest to, że zależą one od konfiguracji aplikacji do definiowania powiązań i punktów końcowych programu WCF. Ponieważ interfejsy API programu .NET Standard WCF mogą działać międzyplatformowo (gdzie interfejsy API System.Configuration nie są dostępne), klienci WCF dla scenariuszy platformy .NET Core i .NET Standard muszą programowo definiować powiązania i punkty końcowe zamiast w konfiguracji.

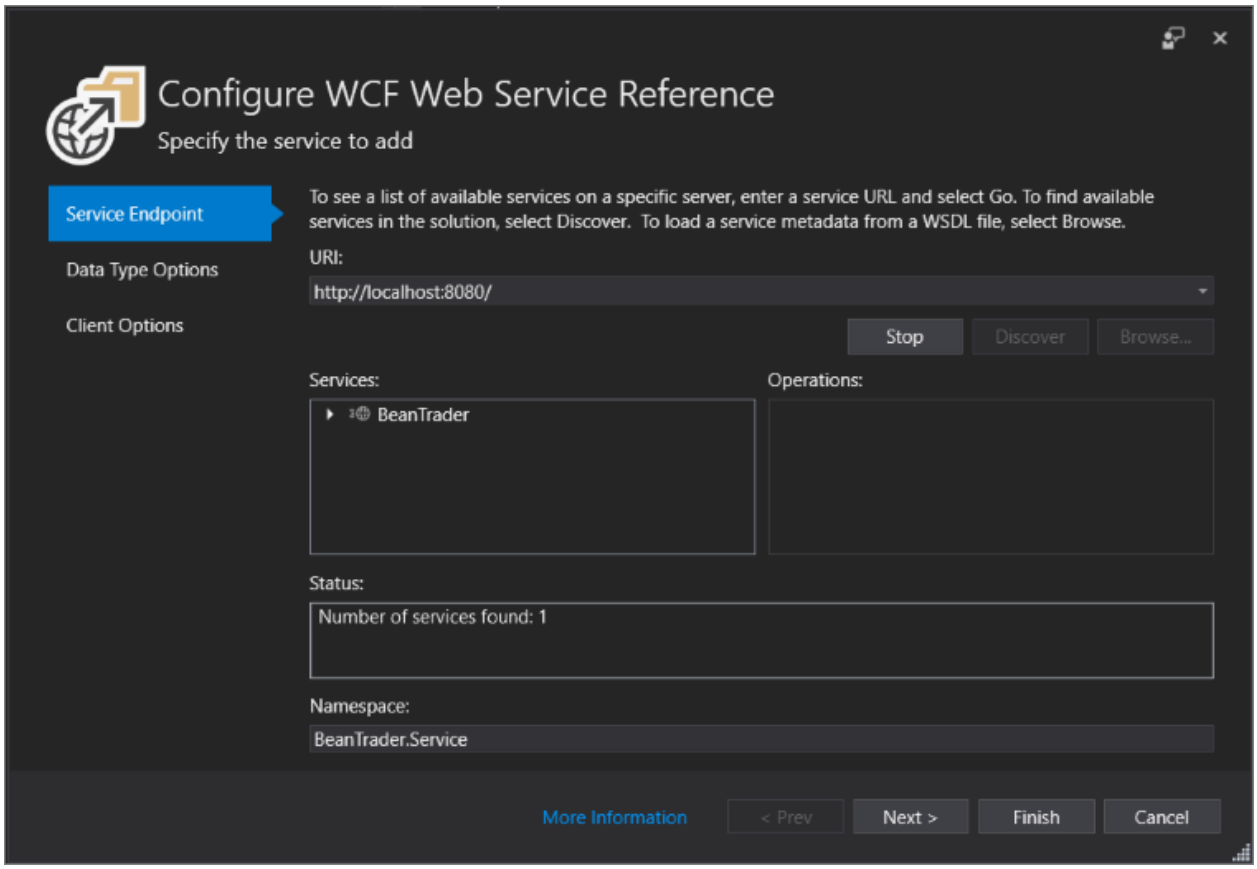
W rzeczywistości każde użycie klienta programu WCF, które zależy od sekcji app.config (utworzonej `<system.serviceModel>` przy użyciu narzędzia Svcutil lub ręcznie), należy zmienić w celu pracy na platformie .NET Core.

Istnieją dwa sposoby automatycznego generowania klientów WCF zgodnych z platformą .NET Standard:

- Narzędzie `dotnet-svcutil` to narzędzie platformy .NET, które generuje klientów WCF w sposób podobny do tego, w jaki sposób narzędzie Svcutil działało wcześniej.

- Program Visual Studio może generować klientów WCF przy użyciu opcji WCF Web Service Reference funkcji połączonych usług.

Jedną z metod działa dobrze. Oczywiście możesz samodzielnie napisać kod klienta WCF. W tym przykładzie wybrano opcję korzystania z funkcji usługi połączonej programu Visual Studio. W tym celu kliknij prawym przyciskiem myszy projekt *BeanClient.Core* w Eksploratorze rozwiązań programu Visual Studio i wybierz pozycję **Dodaj**>**połączoną usługę**. Następnie wybierz dostawcę WCF Web Service Reference. Spowoduje to wyświetlenie okna dialogowego, w którym można określić adres usługi internetowej Bean Trader zaplecza (`localhost:8080` jeśli korzystasz z serwera lokalnie) oraz przestrzeń nazw, której powinny używać wygenerowane typy (na przykład **BeanTrade.Service**).



Po wybraniu przycisku **Zakończ** nowy węzeł Połączone usługi zostanie dodany do projektu, a plik Reference.cs zostanie dodany w tym węźle zawierającym nowego klienta WCF platformy .NET Standard w celu uzyskania dostępu do usługi Bean Trader. Jeśli przyjrzysz się metodom `GetEndpointAddress` lub `GetBindingForEndpoint` w tym pliku, zobaczysz, że powiązania i punkty końcowe są generowane programowo (zamiast za pośrednictwem konfiguracji aplikacji). Funkcja "Dodaj połączone usługi" może również dodawać odwołania do niektórych pakietów System.ServiceModel w pliku projektu, które nie są potrzebne, ponieważ wszystkie niezbędne pakiety WCF są dołączone za pośrednictwem microsoft.Windows.Compatibility. Sprawdź plik csproj, aby sprawdzić, czy zostały dodane dodatkowe elementy System.ServiceModel `<PackageReference>` , a jeśli tak, usuń je.

Nasz projekt ma teraz nowe klasy klientów programu WCF (w pliku *Reference.cs*), ale nadal ma stare (w pliku BeanTrade.cs). W tym momencie dostępne są dwie opcje:

- Jeśli chcesz mieć możliwość skompilowania oryginalnego projektu .NET Framework (wraz z nowym docelowym programem .NET Core), możesz użyć `<Compile Remove="BeanTrader.cs" />` elementu w pliku csproj projektu platformy .NET Core, aby .NET Framework i .NET Core wersji aplikacji używały różnych klientów WCF. Ma to zaletę pozostawienia istniejącego projektu .NET Framework bez zmian, ale ma wadę, że kod korzystający z wygenerowanych klientów WCF może być nieco inny w przypadku platformy .NET Core niż w projekcie .NET Framework, więc prawdopodobnie będzie konieczne użycie `#if` dyrektywy do warunkowego skompilowania użycia klienta WCF (na przykład tworzenia klientów) w taki sposób, aby działały w taki sposób, gdy jest kompilowany dla platformy .NET Core i inny sposób podczas kompilowania na potrzeby .NET Framework.

- Jeśli z drugiej strony niektóre zmiany kodu w istniejącym projekcie .NET Framework są dopuszczalne, możesz usunąć plik *BeanTrade.cs* razem. Ponieważ nowy klient WCF jest kompilowany dla platformy .NET Standard, będzie działać zarówno w scenariuszach platformy .NET Core, jak i .NET Framework. Jeśli kompilujesz .NET Framework oprócz platformy .NET Core (przez wiele elementów docelowych lub dwóch plików csproj), możesz użyć tego nowego pliku *Reference.cs* dla obu obiektów docelowych. Takie podejście ma przewagę nad tym, że kod nie będzie musiał obsługiwać dwóch różnych klientów WCF; ten sam kod będzie używany wszędzie. Wadą jest to, że obejmuje zmianę (prawdopodobnie stabilnego) projektu .NET Framework.

W przypadku przykładu Bean Trader możesz wprowadzić niewielkie zmiany w oryginalnym projekcie, jeśli ułatwia to migrację, więc wykonaj następujące kroki, aby uzgodnić użycie klienta WCF:

1. Dodaj nowy plik Reference.cs do projektu .NET Framework *BeanClient.csproj* przy użyciu menu kontekstowego "Dodaj istniejący element" z eksploratora rozwiązań. Pamiętaj, aby dodać ciąg "jako link", aby ten sam plik był używany przez oba projekty (w przeciwieństwie do kopiowania pliku C#). Jeśli tworzysz zarówno dla platformy .NET Core, jak i .NET Framework za pomocą pojedynczego pliku csproj (używając wielowersyjnego), ten krok nie jest konieczny.

2. Usuń *plik BeanTrade.cs*.

3. Nowy klient WCF jest podobny do starego, ale wiele przestrzeni nazw w wygenerowanym kodzie różni się. W związku z tym należy zaktualizować projekt tak, aby typy klientów programu WCF były używane z klasy BeanTrade.Service (lub dowolnej wybranej nazwy przestrzeni nazw) zamiast BeanShell.Model lub bez

przestrzeni nazw. Kompilowanie *pliku BeanClient.Core.csproj* pomoże określić, gdzie należy wprowadzić te zmiany. Poprawki będą potrzebne zarówno w języku C#, jak i w plikach źródłowych XAML.

4. Na koniec dowiesz się, że w pliku *BeanClientFactory.cs* występuje błąd, ponieważ zmieniły się dostępne konstruktory typu `BeanTraderServiceClient` . Kiedyś można było podać `InstanceContext` argument (który został utworzony przy użyciu kontenera `CallbackHandler` `Castle.Windsor` IoC). Nowe konstruktory tworzą nowe `CallbackHandler`s. Istnieją jednak konstruktory w `BeanTraderServiceClient` typie podstawowym zgodnym z żądanym typem. Ponieważ automatycznie wygenerowany kod klienta programu WCF istnieje w klasach częściowych, można go łatwo rozszerzyć. W tym celu utwórz nowy plik o nazwie *BeanClient.csServiceClient.cs* , a następnie utwórz klasę częściową o tej samej nazwie (przy użyciu przestrzeni nazw BeanClient.Service). Następnie dodaj jeden konstruktor do typu częściowego, jak pokazano tutaj.

   C#

   ```
   public BeanTraderServiceClient(System.ServiceModel.InstanceContext
   callbackInstance) :
       base(callbackInstance,
   EndpointConfiguration.NetTcpBinding_BeanTraderService)
           { }
   ```

Po wprowadzeniu tych zmian przykład Bean Trader będzie teraz używać nowego klienta WCF zgodnego z platformą .NET Standard i można wprowadzić ostateczną poprawkę zmiany `Open` wywołania w *pliku TradingService.cs* , aby zamiast tego użyć `await` `OpenAsync` .

Po rozwiązaniu problemów z usługą WCF wersja platformy .NET Core przykładu Bean Trader jest teraz czysta!

Testowanie środowiska uruchomieniowego

Łatwo zapomnieć o tym, że praca migracji nie jest wykonywana zaraz po utworzeniu projektu w sposób czysty względem platformy .NET Core. Ważne jest również, aby pozostawić czas na testowanie przeniesionej aplikacji. Po pomyślnym utworzeniu aplikacji upewnij się, że aplikacja działa i działa zgodnie z oczekiwaniami, zwłaszcza jeśli używasz pakietów przeznaczonych dla .NET Framework.

Spróbujmy uruchomić portowaną aplikację Bean Trader i zobaczyć, co się stanie. Aplikacja nie jest daleko przed niepowodzeniem z powodu następującego wyjątku.

Dane wyjściowe

```
System.Configuration.ConfigurationErrorsException: 'Configuration system
failed to initialize'

Inner Exception
ConfigurationErrorsException: Unrecognized configuration section
system.serviceModel.
```

To ma sens, oczywiście. Pamiętaj, że program WCF nie używa już konfiguracji aplikacji, dlatego należy usunąć starą sekcję system.serviceModel pliku app.config. Zaktualizowany klient programu WCF zawiera wszystkie te same informacje w kodzie, więc sekcja konfiguracji nie jest już potrzebna. Jeśli chcesz, aby punkt końcowy programu WCF można było skonfigurować w app.config, możesz dodać go jako ustawienie aplikacji i zaktualizować kod klienta programu WCF, aby pobrać punkt końcowy usługi WCF z konfiguracji.

Po usunięciu sekcji system.serviceModel *wapp.config*aplikacja jest uruchamiana, ale kończy się niepowodzeniem z innym wyjątkiem, gdy użytkownik się zaloguje.

Dane wyjściowe

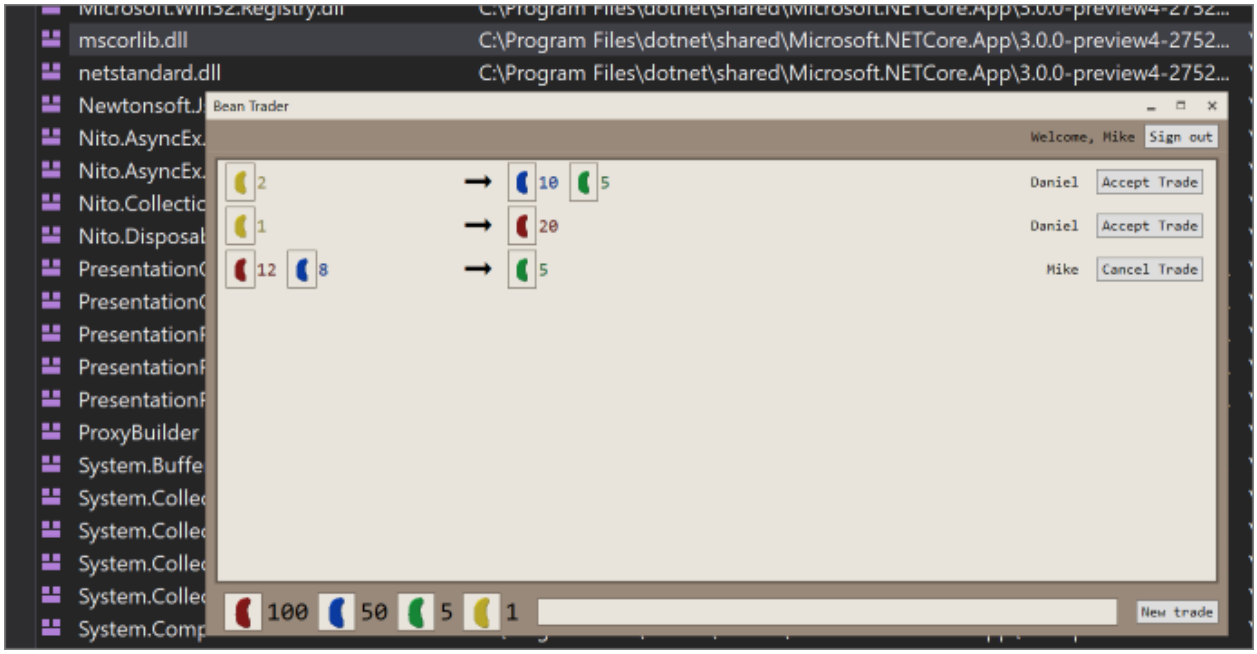
```
System.PlatformNotSupportedException: 'Operation is not supported on this
platform.'
```

Nieobsługiwany interfejs API to `Func<T>.BeginInvoke`. Jak wyjaśniono w temacie dotnet/corefx#5940, platforma .NET Core nie obsługuje `BeginInvoke` metod i `EndInvoke` dla typów delegatów z powodu podstawowych zależności komunikacji równorzędnej. Ten problem i jego poprawka zostały szczegółowo wyjaśnione w wpisie w blogu Migrating Delegate.BeginInvoke Calls for .NET Core, ale elementy i `EndInvoke` wywołania powinny zostać zastąpione `Task.Run` (lub asynchroniczne alternatywy, jeśli to `BeginInvoke` możliwe). Zastosowanie ogólnego rozwiązania w tym miejscu `BeginInvoke` może zostać zastąpione wywołaniem uruchomionym przez `Task.Run`usługę `Invoke` .

C#

```
Task.Run(() =>
{
    return userInfoRetriever.Invoke();
}).ContinueWith(result =>
{
    // BeginInvoke's callback is replaced with ContinueWith
    var task = result.ConfigureAwait(false);
    CurrentTrader = task.GetAwaiter().GetResult();
}, TaskScheduler.Default);
```

Po usunięciu `BeginInvoke` użycia aplikacja Bean Trader działa pomyślnie na platformie .NET Core!



Wszystkie aplikacje są różne, więc konkretne kroki wymagane do migracji własnych aplikacji do platformy .NET Core będą się różnić. Mam jednak nadzieję, że przykład Bean Trader pokazuje ogólny przepływ pracy i typy problemów, które można oczekiwać. I pomimo długości tego artykułu rzeczywiste zmiany potrzebne w przykładzie Bean Trader, aby praca nad platformą .NET Core była dość ograniczona. Wiele aplikacji przeprowadza migrację do platformy .NET Core w ten sam sposób; z ograniczonymi lub nawet żadnymi zmianami kodu nie są potrzebne.

Omówienie okien WPF (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 21 min

Użytkownicy korzystają z aplikacji Windows Presentation Foundation (WPF) za pośrednictwem okien. Głównym celem okna jest hostowanie zawartości, która wizualizuje dane i umożliwia użytkownikom interakcję z danymi. Aplikacje WPF udostępniają własne okna przy użyciu klasy Window. W tym artykule przedstawiono klasę Window przed omówieniem podstaw tworzenia okien i zarządzania nimi w aplikacjach.

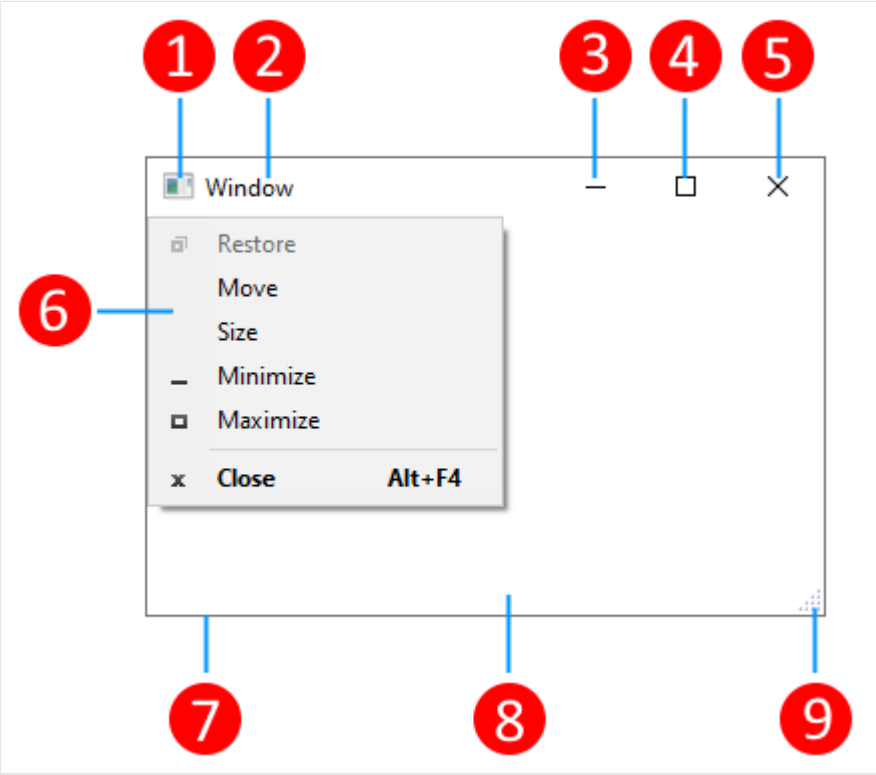
> ⓘ **Ważne**
>
> W tym artykule jest używany kod XAML wygenerowany na podstawie projektu języka **C#**. Jeśli używasz języka **Visual Basic**, kod XAML może wyglądać nieco inaczej. Te różnice zwykle występują w wartościach atrybutu `x:Class`. Język C# uwzględnia przestrzeń nazw katalogu głównego projektu, a język Visual Basic nie.
>
> Szablony projektów dla języka C# tworzą typ `App` zawarty w pliku *app.xaml*. W języku Visual Basic ten typ ma nazwę `Application`, a plik ma nazwę `Application.xaml`.

Klasa Window

W programie WPF okno jest hermetyzowane przez klasę Window używaną do wykonywania następujących czynności:

- Wyświetlanie okna.
- Konfigurowanie rozmiaru, położenia i wyglądu okna.
- Hostowanie zawartości specyficznej dla aplikacji.
- Zarządzanie okresem istnienia okna.

Na poniższej ilustracji przedstawiono części składowe okna:



Okno jest podzielone na dwa obszary: obszar nieklienckі i obszar klienta.

Obszar nieklienckі okna jest implementowany przez platformę WPF i obejmuje części okna, które są wspólne dla większości okien, w tym następujące elementy:

- Pasek tytułu (1–5).
- Ikona (1).
- Tytuł (2).
- Przyciski Minimalizuj (3), Maksymalizuj (4) i Zamknij (5).
- Menu systemowe (6) z elementami menu. Pojawia się po kliknięciu ikony (1).
- Obramowanie (7).

Obszar klienta okna to obszar w obrębie obszaru nieklienckiego okna i jest używany przez deweloperów do dodawania zawartości specyficznej dla aplikacji, takiej jak paski menu, paski narzędzi i kontrolki.

- Obszar klienta (8).
- Uchwyt zmieniania rozmiaru (9). Jest to kontrolka dodawana do obszaru klienta (8).

Implementowanie okna

Implementacja typowego okna obejmuje zarówno wygląd, jak i zachowanie, przy czym *wygląd* definiuje wygląd okna widoczny dla użytkowników, a *zachowanie* definiuje sposób działania okna, gdy użytkownicy z niego korzystają. Na platformie WPF można zaimplementować wygląd i zachowanie okna przy użyciu kodu lub znaczników XAML.

Jednak na ogół wygląd okna jest implementowany przy użyciu znaczników XAML, a jego zachowanie jest implementowane przy użyciu pliku codebehind, jak pokazano w poniższym przykładzie.

XAML

```
<Window x:Class="WindowsOverview.Window1"
        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:WindowsOverview"
        >

    <!-- Client area containing the content of the window -->

</Window>
```

Poniższy kod to plik codebehind dla języka XAML.

C#

```
using System.Windows;

namespace WindowsOverview
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

Aby umożliwić współdziałanie pliku znaczników XAML i pliku związanego z kodem, wymagane są następujące elementy:

- W znacznikach element `Window` musi zawierać atrybut `x:Class`. Gdy aplikacja zostanie skompilowana, istnienie atrybutu `x:Class` spowoduje, że aparat kompilacji firmy Microsoft (MSBuild) wygeneruje klasę `partial` pochodzącą z klasy Window o nazwie określonej przez atrybut `x:Class`. Wymaga to dodania deklaracji przestrzeni nazw XML dla schematu XAML (`xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`). Wygenerowana klasa `partial` implementuje metodę `InitializeComponent`, która jest wywoływana w

celu zarejestrowania zdarzeń i ustawienia właściwości implementowanych w znacznikach.

- W pliku codebehind klasa musi być klasą `partial` o tej samej nazwie, która jest określona przez atrybut `x:Class` w znacznikach, i musi pochodzić z klasy Window. Umożliwia to skojarzenie pliku związanego z kodem z klasą `partial`, która jest generowana dla pliku znaczników podczas kompilowania aplikacji. Aby uzyskać więcej informacji, zobacz Kompilowanie aplikacji WPF.
- W pliku codebehind klasa Window musi zaimplementować konstruktora, który wywołuje metodę `InitializeComponent`. Metoda `InitializeComponent` jest implementowana przez wygenerowaną klasę `partial` pliku znaczników w celu rejestrowania zdarzeń i ustawiania właściwości zdefiniowanych w znacznikach.

> **Uwaga**
>
> Po dodaniu nowej klasy **Window** do projektu przy użyciu programu Visual Studio klasa **Window** jest implementowana przy użyciu zarówno znaczników, jak i pliku codebehind, oraz zawiera niezbędną konfigurację do utworzenia skojarzenia między znacznikami i plikami związanymi z kodem zgodnie z opisem w tym miejscu.

Dzięki tej konfiguracji można skupić się na zdefiniowaniu wyglądu okna w znacznikach XAML i zaimplementowaniu jego zachowania w pliku codebehind. Poniższy przykład przedstawia okno z przyciskiem definiującym obsługę zdarzeń dla zdarzenia Click. Jest to implementowane w języku XAML, a procedura obsługi jest implementowana w pliku codebehind.

XAML

```
<Window x:Class="WindowsOverview.Window1"
        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:WindowsOverview"
        >

    <!-- Client area containing the content of the window -->

    <Button Click="Button_Click">Click This Button</Button>

</Window>
```

Poniższy kod to plik codebehind dla języka XAML.

C#

```
using System.Windows;

namespace WindowsOverview
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Button was clicked.");
        }
    }
}
```

Konfigurowanie okna dla programu MSBuild

Sposób implementacji okna określa, jak jest ono skonfigurowane dla programu MSBuild. W przypadku okna zdefiniowanego przy użyciu znaczników XAML i pliku codebehind:

- Pliki znaczników XAML są konfigurowane jako elementy `Page` programu MSBuild.
- Pliki związane z kodem są konfigurowane jako elementy `Compile` programu MSBuild.

Projekty zestawu SDK platformy .NET automatycznie importują poprawne elementy `Page` i `Compile` nie trzeba ich deklarować. Po skonfigurowaniu projektu dla platformy WPF pliki znaczników XAML są automatycznie importowane jako elementy `Page`, a odpowiedni plik związany z kodem jest importowany jako element `Compile`.

Projekty programu MSBuild nie będą automatycznie importować typów i należy je zadeklarować samodzielnie:

XML

```
<Project>
    ...
    <Page Include="MarkupAndCodeBehindWindow.xaml" />
    <Compile Include=" MarkupAndCodeBehindWindow.xaml.cs" />
    ...
</Project>
```

Aby uzyskać informacje na temat tworzenia aplikacji WPF, zobacz Kompilowanie aplikacji WPF.

Okres istnienia okna

Podobnie jak w przypadku każdej klasy okno ma okres istnienia, który rozpoczyna się, gdy po raz pierwszy zostanie utworzone jego wystąpienie, po czym jest ono otwierane, uaktywniane/dezaktywowane i ostatecznie zamykane.

Otwieranie okna

Aby otworzyć okno, należy najpierw utworzyć jego wystąpienie, co zostało pokazane w poniższym przykładzie:

C#

```
using System.Windows;

namespace WindowsOverview
{
    public partial class App : Application
    {
        private void Application_Startup(object sender, StartupEventArgs e)
        {
            // Create the window
            Window1 window = new Window1();

            // Open the window
            window.Show();
        }
    }
}
```

W tym przykładzie dla klasy `Window1` jest tworzone wystąpienie po uruchomieniu aplikacji, co następuje po wywołaniu zdarzenia Startup. Aby uzyskać więcej informacji na temat okna uruchamiania, zobacz Jak pobrać lub ustawić główne okno aplikacji.

Po utworzeniu wystąpienia okna odwołanie do niego jest automatycznie dodawane do listy okien zarządzanej przez obiekt Application. Pierwsze okno do utworzenia wystąpienia jest automatycznie ustawiane przez klasę Application jako główne okno aplikacji.

Okno jest ostatecznie otwierane przez wywołanie metody Show, jak pokazano na poniższej ilustracji:

Okno otwarte przez wywołanie metody Show to okno *niemodalne*, a aplikacja nie uniemożliwia użytkownikom interakcji z innymi oknami w aplikacji. Otwarcie okna przy użyciu metody ShowDialog powoduje otwarcie okna jako *modalnego* i ogranicza interakcję użytkownika do konkretnego okna. Aby uzyskać więcej informacji, zobacz Omówienie okien dialogowych.

Po wywołaniu metody Show okno wykonuje inicjowanie zanim zostanie wyświetlone, aby ustanowić infrastrukturę, która umożliwia mu odbieranie danych wejściowych użytkownika. Po zainicjowaniu okna zostanie wywołane zdarzenie SourceInitialized i zostanie wyświetlone okno.

Aby uzyskać więcej informacji, zobacz Jak otworzyć okno lub okno dialogowe.

Okno uruchamiania

W poprzednim przykładzie użyto zdarzenia `Startup` do uruchomienia kodu, który powodował wyświetlenie początkowego okna aplikacji. Zamiast tego, w ramach skrótu, użyj interfejsu StartupUri, aby określić ścieżkę do pliku XAML w aplikacji. Aplikacja automatycznie tworzy i wyświetla okno określone przez tę właściwość.

XAML

```
<Application x:Class="WindowsOverview.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:WindowsOverview"
             StartupUri="ClippedWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Własność okna

Okno otwierane przy użyciu metody Show nie ma niejawnej relacji z oknem, które je utworzyło. Użytkownicy mogą korzystać z obu okien niezależnie od siebie, co oznacza, że każde z okien może wykonywać następujące operacje:

- Zakrywanie drugiego (chyba że jedno z okien ma dla właściwości Topmost ustawioną wartość `true`).
- Być minimalizowane, maksymalizowane i przywracane bez wywierania wpływu na drugie.

Niektóre okna wymagają relacji z oknem, które je otwiera. Na przykład aplikacja zintegrowanego środowiska projektowego (IDE) może otwierać okna właściwości i okna narzędzi, a ich typowym zachowaniem jest zakrywanie okna, które je utworzyło. Ponadto takie okna powinny zawsze być zamykane, minimalizowane, maksymalizowane i przywracane w połączeniu z oknem, które je utworzyło. Taką relację można ustanowić, przez uczynienie jednego okna *właścicielem* innego okna. Osiągane jest to przez ustawienie właściwości Owner*okna posiadanego* z odwołaniem do *okna będącego właścicielem*. Jest to pokazane w następującym przykładzie.

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Create a window and make the current window its owner
    var ownedWindow = new ChildWindow1();
    ownedWindow.Owner = this;
    ownedWindow.Show();
}
```

Po ustanowieniu własności:

- Okno posiadane może odwoływać się do okna będącego jego właścicielem przez sprawdzenie wartości jego właściwości Owner.
- Okno będące właścicielem może odnajdować wszystkie okna, których jest właścicielem, przez sprawdzenie wartości jego właściwości OwnedWindows.

Aktywacja okna

Po otwarciu okna po raz pierwszy staje się ono oknem aktywnym. *Okno aktywne* to okno, które obecnie przechwytuje dane wejściowe użytkownika, takie jak naciśnięcia klawiszy i kliknięcia myszy. Gdy okno stanie się aktywne, wywołuje zdarzenie Activated.

> ⓘ **Uwaga**
>
> Po pierwszym otwarciu okna zdarzenia **Loaded** i **ContentRendered** są wywoływane dopiero po wywołaniu zdarzenia **Activated**. Mając to na uwadze, okno może być uznawane za otwarte, gdy zostanie wywołane zdarzenie **ContentRendered**.

Gdy okno stanie się aktywne, użytkownik może aktywować inne okno w tej samej aplikacji lub aktywować inną aplikację. W takim przypadku obecnie aktywne okno zostanie zdezaktywowane i wywoła zdarzenie Deactivated. Podobnie, gdy użytkownik wybierze obecnie dezaktywowane okno, okno stanie się ponownie aktywne i zostanie wywołane zdarzenie Activated.

Jednym z typowych powodów obsługi zdarzeń Activated i Deactivated jest włączenie i wyłączenie funkcji, która może być uruchamiana tylko wtedy, gdy okno jest aktywne. Na przykład niektóre okna wyświetlają interaktywną zawartość, która wymaga stałego wprowadzania danych przez użytkownika lub jego uwagi, takie jak gry i odtwarzacze wideo. Poniższy przykład to uproszczony odtwarzacz wideo, który pokazuje, jak obsługiwać zdarzenia Activated i Deactivated w celu zaimplementowania tego zachowania.

XAML

```
<Window x:Class="WindowsOverview.CustomMediaPlayerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Activated="Window_Activated"
        Deactivated="Window_Deactivated"
        Title="CustomMediaPlayerWindow" Height="450" Width="800">
    <Grid>
        <MediaElement x:Name="mediaElement" Stretch="Fill"
                      LoadedBehavior="Manual" Source="numbers.mp4" />
    </Grid>
</Window>
```

Poniższy kod to plik codebehind dla języka XAML.

C#

```
using System;
using System.Windows;

namespace WindowsOverview
{
    public partial class CustomMediaPlayerWindow : Window
    {
        public CustomMediaPlayerWindow() =>
            InitializeComponent();

        private void Window_Activated(object sender, EventArgs e)
        {
            // Continue playing media if window is activated
            mediaElement.Play();
        }

        private void Window_Deactivated(object sender, EventArgs e)
```

```
        {
            // Pause playing if media is being played and window is
deactivated
            mediaElement.Pause();
        }
    }
}
```

Inne typy aplikacji mogą nadal uruchamiać kod w tle po dezaktywacji okna. Na przykład klient poczty może kontynuować sondowanie serwera poczty, gdy użytkownik korzysta z innych aplikacji. Aplikacje takie jak te często udostępniają inne lub dodatkowe zachowanie, gdy okno główne jest dezaktywowane. W przypadku programu poczty może to oznaczać zarówno dodanie nowego elementu poczty do skrzynki odbiorczej, jak i dodanie ikony powiadomienia do zasobnika systemu. Ikona powiadomienia musi być wyświetlana tylko wtedy, gdy okno poczty nie jest aktywne, co jest ustalane przez sprawdzenie właściwości IsActive.

Jeśli zadanie w tle zakończy się, okno może wysłać do użytkownika bardziej pilne powiadomienie przez wywołanie metody Activate. Jeśli użytkownik wchodzi w interakcję z inną aplikacją uaktywnioną po wywołaniu zdarzenia Activate, przycisk paska zadań okna miga. Jeśli jednak użytkownik wchodzi w interakcję z bieżącą aplikacją, wywołanie zdarzenia Activate spowoduje przełączenie okna na pierwszy plan.

> ⓘ **Uwaga**
>
> Aktywację w zakresie aplikacji można obsługiwać przy użyciu zdarzeń **Application.Activated** i **Application.Deactivated**.

Zapobieganie aktywacji okna

Istnieją scenariusze, w których okna nie powinny być uaktywniane po ich wyświetleniu, takie jak okna konwersacji aplikacji czatu lub okna powiadomień aplikacji poczty e-mail.

Jeśli aplikacja ma okno, które nie powinno być uaktywniane po jego wyświetleniu, można ustawić dla jej właściwości ShowActivated wartość `false` przed wywołaniem metody Show po raz pierwszy. W konsekwencji:

- Okno nie jest uaktywnione.
- Zdarzenie okna Activated nie jest wywoływane.
- Obecnie uaktywnione okno pozostaje uaktywnione.

Okno zostanie uaktywnione, jednak dopiero gdy uaktywni je użytkownik przez kliknięcie obszaru klienta lub nieklienckiego. W takim przypadku:

- Okno jest uaktywnione.
- Zdarzenie okna Activated jest wywoływane.
- Okno uaktywnione wcześniej jest dezaktywowane.
- Zdarzenia Deactivated i Activated okna są następnie wywoływane zgodnie z oczekiwaniami w reakcji na akcje użytkownika.

Zamykanie okna

Okres życia okna zaczyna zbliżać się do końca po zamknięciu go przez użytkownika. Po zamknięciu okna nie można go ponownie otworzyć. Okno można zamknąć przy użyciu elementów w obszarze nieklienckim, takich jak poniższe:

- Element **Zamknij** menu **System**.
- Naciśnięcie klawiszy `ALT + F4`.
- Naciśnięcie przycisku **Zamknij**.
- Naciśnięcie klawisza `ESC`, gdy przycisk ma dla właściwości IsCancel ustawioną wartość `true` w oknie modalnym.

Możesz udostępnić więcej mechanizmów zamykania okna w obszarze klienta, z których najczęstsze są następujące:

- Element **Zakończ** w menu **Plik**, zwykle w przypadku głównych okien aplikacji.
- Element **Zamknij** w menu **Plik**, zwykle w pomocniczym oknie aplikacji.
- Przycisk **Anuluj**, zwykle w modalnym oknie dialogowym.
- Przycisk **Zamknij**, zwykle w niemodalnym oknie dialogowym.

Aby zamknąć okno w reakcji na jeden z tych mechanizmów niestandardowych, należy wywołać metodę Close. Poniższy przykład implementuje możliwość zamknięcia okna przez wybranie polecenia **Zakończ** z menu **Plik**.

XAML

```
<Window x:Class="WindowsOverview.ClosingWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ClosingWindow" Height="450" Width="800">
    <StackPanel>
        <Menu>
            <MenuItem Header="_File">
                <MenuItem Header="E_xit" Click="fileExitMenuItem_Click" />
            </MenuItem>
        </Menu>
    </StackPanel>
</Window>
```

Poniższy kod to plik codebehind dla języka XAML.

```C#
using System.Windows;

namespace WindowsOverview
{
    public partial class ClosingWindow : Window
    {
        public ClosingWindow() =>
            InitializeComponent();

        private void fileExitMenuItem_Click(object sender, RoutedEventArgs
e)
        {
            // Close the current window
            this.Close();
        }
    }
}
```

> ⓘ **Uwaga**
>
> Aplikację można skonfigurować tak, aby była automatycznie zamykana po zamknięciu głównego okna aplikacji (zobacz **MainWindow**) lub zamknięciu ostatniego okna. Aby uzyskać więcej informacji, zobacz **ShutdownMode**.

Chociaż okno można jawnie zamknąć za pomocą mechanizmów udostępnianych w obszarach klienta i nieklienckich, okno może również zostać niejawnie zamknięte w wyniku zachowania w innych częściach aplikacji lub systemu Windows, w tym następujących:

- Użytkownik wylogował się lub zamknął system Windows.
- Zamknięto element Owner okna.
- Zamknięto główne okno aplikacji i atrybut ShutdownMode ma wartość OnMainWindowClose.
- Wywołano metodę Shutdown.

> ⓘ **Ważne**
>
> Nie można ponownie otworzyć okna po jego zamknięciu.

Anulowanie zamknięcia okna

Gdy okno zostanie zamknięte, wywołuje dwa zdarzenia: Closing i Closed.

Zdarzenie Closing jest wywoływane przed zamknięciem okna i zapewnia mechanizm, za pomocą którego można zapobiec zamknięciu okna. Jedną z typowych przyczyn zapobiegania zamykaniu okna jest to, że zawartość okna zawiera zmodyfikowane dane. W takiej sytuacji można obsłużyć zdarzenie Closing w celu ustalenia, czy dane są zanieczyszczone, a jeśli tak, do zapytania użytkownika, czy kontynuować zamykanie okna bez zapisywania danych, czy anulować zamknięcie okna. W poniższym przykładzie przedstawiono kluczowe aspekty obsługi zdarzenia Closing.

XAML

```
<Window x:Class="WindowsOverview.DataWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataWindow" Height="450" Width="800"
        Closing="Window_Closing">
    <Grid>
        <TextBox x:Name="documentTextBox"
TextChanged="documentTextBox_TextChanged" />
    </Grid>
</Window>
```

Poniższy kod to plik codebehind dla języka XAML.

C#

```
using System.Windows;
using System.Windows.Controls;

namespace WindowsOverview
{
    public partial class DataWindow : Window
    {
        private bool _isDataDirty;

        public DataWindow() =>
            InitializeComponent();

        private void documentTextBox_TextChanged(object sender,
TextChangedEventArgs e) =>
            _isDataDirty = true;

        private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
        {
            // If data is dirty, prompt user and ask for a response
            if (_isDataDirty)
            {
                var result = MessageBox.Show("Document has changed. Close
```

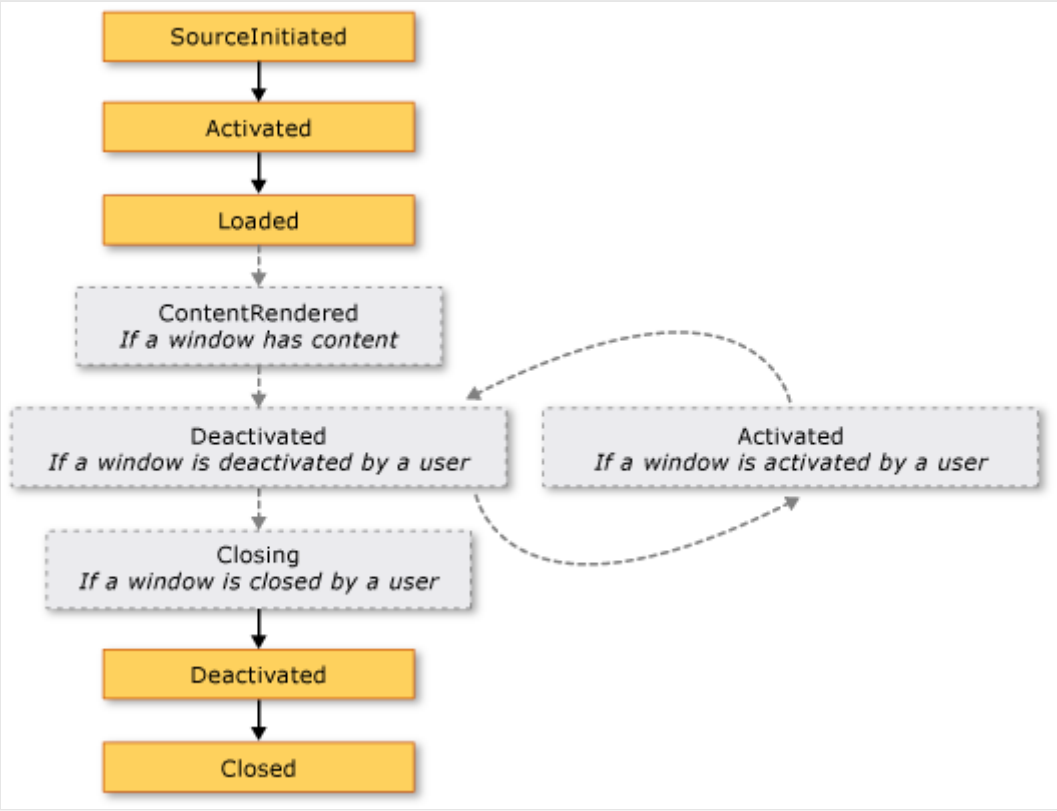
```
without saving?",
                                                "Question",
                                                MessageBoxButton.YesNo);

                // User doesn't want to close, cancel closure
                if (result == MessageBoxResult.No)
                    e.Cancel = true;
            }
        }
    }
}
```

Do obsługi zdarzeń Closing jest przekazywany element CancelEventArgs implementujący właściwość Cancel, dla której ustawiono wartość `true`, aby zapobiegać zamknięciu okna.

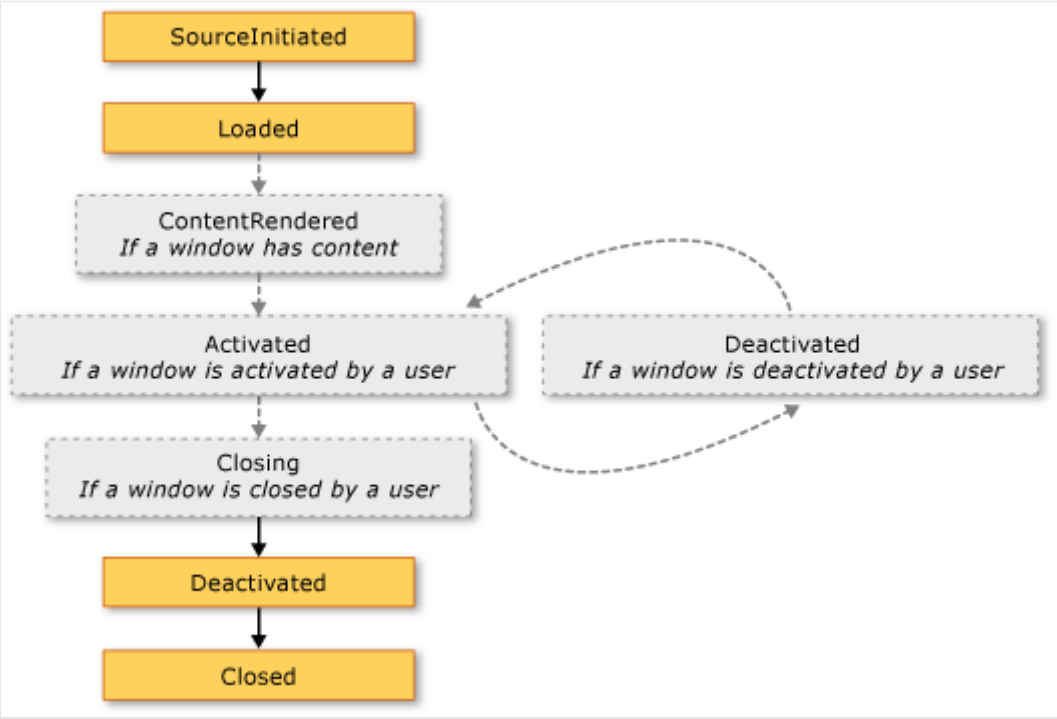
Jeśli zdarzenie Closing nie jest obsługiwane lub jest obsługiwane, ale nie zostało anulowane, okno zostanie zamknięte. Tuż przed zamknięciem okna zostanie wywołane zdarzenie Closed. W tym momencie nie można uniemożliwić zamknięcia okna.

Zdarzenia okresu istnienia okna

Poniższa ilustracja przedstawia sekwencję głównych zdarzeń w okresie istnienia okna:



Poniższa ilustracja przedstawia sekwencję głównych zdarzeń w okresie istnienia okna, które jest wyświetlane bez aktywacji (dla właściwości ShowActivated ustawiono wartość `false` przed wyświetleniem okna):



Lokalizacja okna

Gdy okno jest otwarte, ma lokalizację w wymiarach X i Y względem pulpitu. Tę lokalizację można ustalić, sprawdzając odpowiednio właściwości Left i Top. Ustaw te właściwości, aby zmienić lokalizację okna.

Możesz również określić początkową lokalizację obiektu Window po jego wyświetleniu po raz pierwszy, ustawiając dla właściwości WindowStartupLocation jedną z następujących wartości wyliczenia WindowStartupLocation:

- CenterOwner (domyślne)
- CenterScreen
- Manual

Jeśli lokalizacja uruchamiania jest określona jako Manual, a właściwości Left i Top nie zostały ustawione, obiekt Window zapyta system operacyjny o lokalizację, w której ma zostać wyświetlony.

Okna najwyżej w hierarchii i porządek osi Z

Oprócz lokalizacji w wymiarach X i Y okno ma również lokalizację w wymiarze Z, który określa jego położenie pionowe w odniesieniu do innych okien. Jest to nazywane porządkiem osi Z okna i istnieją jego dwa typy: **normalny** i **najwyżej w hierarchii**. Lokalizacja okna w *normalnym porządku osi Z* zależy od tego, czy jest ono obecnie aktywne, czy nie. Domyślnie okno znajduje się w normalnym porządku osi Z. Lokalizacja okna w *najwyższym w hierarchii porządku osi Z* także zależy od tego, czy jest ono

obecnie aktywne, czy nie. Ponadto okna w najwyższym w hierarchii porządku osi Z są zawsze umieszczone nad oknami w normalnym porządku wzdłuż osi Z. Okno jest umieszczane w najwyższym w hierarchii porządku osi Z po ustawieniu dla jego właściwości Topmost wartości `true`.

W każdym typie hierarchii porządku osi Z obecnie aktywne okno jest wyświetlane powyżej wszystkich innych okien w tym samym porządku osi Z.

Rozmiar okna

Oprócz lokalizacji na pulpicie okno ma rozmiar, który jest określany przez kilka właściwości, w tym różne właściwości szerokości i wysokości oraz SizeToContent.

Właściwości MinWidth, Width i MaxWidth są używane do zarządzania zakresem szerokości, które okno może mieć w swoim okresie istnienia.

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    MinWidth="300" Width="400" MaxWidth="500">
</Window>
```

Wysokość okna jest zarządzana przez właściwości MinHeight, Height i MaxHeight.

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    MinHeight="300" Height="400" MaxHeight="500">
</Window>
```

Ponieważ różne wartości szerokości i wysokości określają zakres, można określić szerokość i wysokość okna z możliwością zmiany rozmiaru w dowolnym miejscu w określonym zakresie dla odpowiedniego wymiaru. Aby wykryć bieżącą szerokość i wysokość, sprawdź odpowiednio właściwości ActualWidth i ActualHeight.

Jeśli chcesz, aby szerokość i wysokość okna miały rozmiar pasujący do rozmiaru zawartości okna, możesz użyć właściwości SizeToContent, która ma następujące wartości:

- SizeToContent.Manual
  Brak efektu (wartość domyślna).

- SizeToContent.Width
  Dopasuj do szerokości zawartości, co ma taki sam efekt jak ustawienie zarówno właściwości MinWidth, jak i MaxWidth na szerokość zawartości.
- SizeToContent.Height
  Dopasuj do wysokości zawartości, co ma taki sam efekt jak ustawienie zarówno właściwości MinHeight, jak i MaxHeight na wysokość zawartości.
- SizeToContent.WidthAndHeight
  Dopasuj do szerokości i wysokości zawartości, co ma taki sam efekt jak ustawienie zarówno właściwości MinHeight, jak i MaxHeight na wysokość zawartości, oraz ustawienie zarówno właściwości MinWidth, jak i MaxWidth na szerokość zawartości.

W poniższym przykładzie pokazano okno, które automatycznie dopasowuje rozmiar do zawartości, zarówno w pionie, jak i w poziomie, gdy jest wyświetlane po raz pierwszy.

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    SizeToContent="WidthAndHeight">
</Window>
```

W poniższym przykładzie pokazano, jak ustawić właściwość SizeToContent w kodzie, aby określić sposób zmieniania rozmiaru okna w celu dopasowania go do jego zawartości.

C#

```
// Manually alter window height and width
this.SizeToContent = SizeToContent.Manual;

// Automatically resize width relative to content
this.SizeToContent = SizeToContent.Width;

// Automatically resize height relative to content
this.SizeToContent = SizeToContent.Height;

// Automatically resize height and width relative to content
this.SizeToContent = SizeToContent.WidthAndHeight;
```

Kolejność pierwszeństwa dla właściwości określania rozmiaru

Zasadniczo różne właściwości rozmiarów okna łączą się w celu zdefiniowania zakresu szerokości i wysokości dla okna, którego rozmiar można zmieniać. Aby zapewnić

utrzymanie prawidłowego zakresu, obiekt Window oblicza wartości właściwości rozmiaru przy użyciu następujących kolejności pierwszeństwa.

W przypadku właściwości wysokości:

1. FrameworkElement.MinHeight
2. FrameworkElement.MaxHeight
3. SizeToContent.Height / SizeToContent.WidthAndHeight
4. FrameworkElement.Height

W przypadku właściwości szerokości:

1. FrameworkElement.MinWidth
2. FrameworkElement.MaxWidth
3. SizeToContent.Width / SizeToContent.WidthAndHeight
4. FrameworkElement.Width

Kolejność pierwszeństwa może również określać rozmiar okna, gdy jest ono zmaksymalizowane. Do zarządzania tym służy właściwość WindowState.

Stan okna

W okresie istnienia okno z możliwością zmiany rozmiaru może mieć trzy stany: normalne, zminimalizowane i zmaksymalizowane. Okno w stanie *normalnym* to domyślny stan okna. Okno o tym stanie umożliwia użytkownikowi przenoszenie i zmienianie jego rozmiaru za pomocą uchwytu zmiany rozmiaru lub obramowania, jeśli zmiana rozmiaru jest możliwa.

Okno w stanie *zminimalizowanym* jest zwijane do przycisku paska zadań, jeśli dla właściwości ShowInTaskbar ustawiono wartość `true`. W przeciwnym razie jest zwijane do najmniejszego możliwego rozmiaru, a następnie przenoszone do lewego dolnego rogu pulpitu. Nie można zmienić rozmiaru okna zminimalizowanego przy użyciu obramowania ani uchwytu zmiany rozmiaru, chociaż zminimalizowane okno, które nie jest wyświetlane na pasku zadań, można przeciągać po pulpicie.

Okno w stanie *zmaksymalizowanym* jest rozszerzone do maksymalnego rozmiaru, jaki może przyjąć, czyli będzie mieć maksymalnie takie rozmiary, na które zezwalają jego właściwości MaxWidth, MaxHeight i SizeToContent. Podobnie jak w przypadku okna zminimalizowanego, nie można zmieniać rozmiaru okna zmaksymalizowanego przy użyciu uchwytu zmiany rozmiaru ani przez przeciąganie obramowania.

> ⓘ **Uwaga**

Wartości właściwości **Top**, **Left**, **Width** i **Height** okna zawsze reprezentują wartości stanu normalnego, nawet jeśli okno jest obecnie zmaksymalizowane lub zminimalizowane.

Stan okna można skonfigurować, ustawiając jego właściwość WindowState, która może mieć jedną z następujących wartości wyliczenia WindowState:

- Normal (domyślne)
- Maximized
- Minimized

W poniższym przykładzie pokazano, jak utworzyć okno, które po otwarciu jest wyświetlane jako zmaksymalizowane.

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    WindowState="Maximized">
</Window>
```

Na ogół należy ustawić właściwość WindowState w celu skonfigurowania początkowego stanu okna. Gdy zostanie wyświetlone okno *z możliwością zmiany rozmiaru*, użytkownicy mogą naciskać przyciski minimalizuj, maksymalizuj i przywróć na pasku tytułu okna, aby zmienić stan okna.

Wygląd okna

Wygląd obszaru klienta okna można zmienić, dodając do niego zawartość specyficzną dla okna, taką jak przyciski, etykiety i pola tekstowe. W celu skonfigurowania obszaru nieklienckiego klasa Window udostępnia kilka właściwości, które obejmują Icon do ustawiania ikony okna i Title do ustawiania jego tytułu.

Możesz również zmienić wygląd i zachowanie obramowania obszaru nieklienckiego, konfigurując tryb zmiany rozmiaru okna, styl okna i to, czy jest ono wyświetlane jako przycisk na pasku zadań pulpitu.

Tryb zmiany rozmiaru

W zależności od właściwości WindowStyle możesz kontrolować, czy i jak użytkownicy mogą zmieniać rozmiar okna. Styl okna wpływa na następujące aspekty:

- Zezwalanie na zmianę rozmiaru lub nie zezwalanie na nią przez przeciąganie obramowania okna myszą.
- Czy przyciski **Minimalizuj**, **Maksymalizuj** i **Zamknij** są wyświetlane w obszarze nieklienckim.
- Czy są włączone przyciski **Minimalizuj**, **Maksymalizuj** i **Zamknij**.

Możesz skonfigurować sposób zmieniania rozmiaru okna, ustawiając jego właściwość ResizeMode, która może być jedną z następujących wartości wyliczenia ResizeMode:

- NoResize
- CanMinimize
- CanResize (domyślne)
- CanResizeWithGrip

Podobnie jak w przypadku elementu WindowStylemało prawdopodobna jest modyfikacja trybu zmiany rozmiaru okna w okresie jego istnienia, co oznacza, że najprawdopodobniej będzie on ustawiany na podstawie znaczników XAML.

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    ResizeMode="CanResizeWithGrip">
</Window>
```

Pamiętaj, że możesz wykryć, czy okno jest zmaksymalizowane, zminimalizowane lub przywrócone, sprawdzając właściwość WindowState.

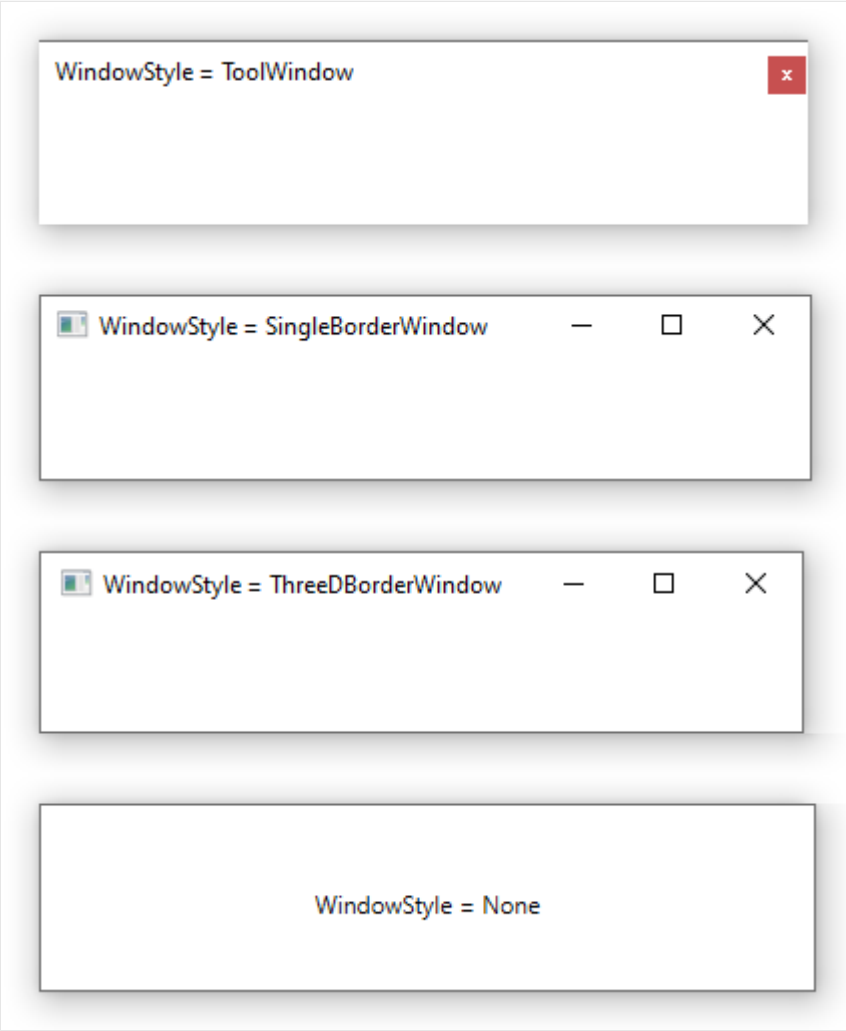
Styl okna

Obramowanie uwidocznione z obszaru nieklienckiego okna jest odpowiednie dla większości aplikacji. Istnieją jednak okoliczności, w których potrzebne są różne typy obramowań lub w ogóle nie są potrzebne żadne obramowania, w zależności od typu okna.

Aby kontrolować typ obramowania nadawany oknu, należy ustawić jego właściwość WindowStyle przy użyciu jednej z następujących wartości wyliczenia WindowStyle:

- None
- SingleBorderWindow (domyślne)
- ThreeDBorderWindow
- ToolWindow

Efekt zastosowania stylu okna przedstawiono na poniższej ilustracji:



Zwróć uwagę, że powyższy obraz nie pokazuje zauważalnej różnicy między właściwościami `SingleBorderWindow` i `ThreeDBorderWindow`. W systemie Windows XP właściwość `ThreeDBorderWindow` miała wpływ na sposób rysowania okna przez dodanie obramowania 3D do obszaru klienta. Począwszy od systemu Windows 7, różnice między dwoma stylami są minimalne.

Można ustawić właściwość WindowStyle przy użyciu znaczników XAML lub kodu. Ponieważ jest mało prawdopodobne, że ulegnie to zmianie w okresie istnienia okna, najprawdopodobniej skonfigurujesz to przy użyciu znaczników XAML.

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    WindowStyle="ToolWindow">
</Window>
```

Styl okna nieprostokątnego

Istnieją również sytuacje, gdy style obramowania, na które zezwala obiekt WindowStyle, są niewystarczające. Możesz na przykład utworzyć aplikację z obramowaniem nieprostokątnym, takie jak używane przez odtwarzacz Microsoft Windows Media Player.

Rozważmy na przykład okno dymka mowy pokazane na poniższej ilustracji:

Ten typ okna można utworzyć, ustawiając dla właściwości WindowStyle wartość None, i używając specjalnej obsługi, którą ma obiekt Window dla przezroczystości.

XAML

```
<Window x:Class="WindowsOverview.ClippedWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ClippedWindow" SizeToContent="WidthAndHeight"
        WindowStyle="None" AllowsTransparency="True"
Background="Transparent">
    <Grid Margin="20">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="20"/>
        </Grid.RowDefinitions>

        <Rectangle Stroke="#FF000000" RadiusX="10" RadiusY="10"/>
        <Path Fill="White" Stretch="Fill" Stroke="#FF000000"
HorizontalAlignment="Left" Margin="15,-5.597,0,-0.003" Width="30"
Grid.Row="1" Data="M22.166642,154.45381 L29.999666,187.66699
40.791059,154.54395"/>
        <Rectangle Fill="White" RadiusX="10" RadiusY="10" Margin="1"/>

        <TextBlock HorizontalAlignment="Left" VerticalAlignment="Center"
FontSize="25" Text="Greetings!" TextWrapping="Wrap" Margin="5,5,50,5"/>
        <Button HorizontalAlignment="Right" VerticalAlignment="Top"
Background="Transparent" BorderBrush="{x:Null}" Foreground="Red"
Content="❌" FontSize="15" />

        <Grid.Effect>
            <DropShadowEffect BlurRadius="10" ShadowDepth="3"
Color="LightBlue"/>
        </Grid.Effect>
    </Grid>
</Window>
```

Ta kombinacja wartości powoduje, że okno będzie renderowane jako przezroczyste. W tym stanie nie można używać przycisków zakończenia obszaru nieklienckiego okna i należy udostępnić własne.

Obecność na pasku zadań

Domyślny wygląd okna obejmuje przycisk paska zadań. Niektóre typy okien nie mają przycisku paska zadań, takie jak okna komunikatów, okna dialogowe lub okna, dla których dla właściwości WindowStyle ustawiono wartość ToolWindow. Możesz kontrolować, czy przycisk paska zadań dla okna jest wyświetlany, ustawiając właściwość ShowInTaskbar, która domyślnie ma wartość `true`.

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    ShowInTaskbar="False">
</Window>
```

Inne typy okien

NavigationWindow jest oknem przeznaczonym do hostowania zawartości z możliwością nawigacji.

Okna dialogowe to okna, które są często używane do zbierania informacji od użytkownika w celu ukończenia funkcji. Na przykład gdy użytkownik chce otworzyć plik, przez aplikację jest wyświetlane okno dialogowe **Otwieranie pliku** w celu uzyskania nazwy pliku od użytkownika. Aby uzyskać więcej informacji, zobacz Omówienie okien dialogowych.

Zobacz też

- Omówienie okien dialogowych
- Jak otworzyć okno lub okno dialogowe
- Jak otworzyć wspólne okno dialogowe
- Jak otworzyć okno komunikatu
- Jak zamknąć okno lub okno dialogowe
- System.Windows.Window
- System.Windows.MessageBox
- System.Windows.Navigation.NavigationWindow
- System.Windows.Application

Przegląd okien dialogowych (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 8 min

Windows Presentation Foundation (WPF) zapewnia sposoby projektowania własnych okien dialogowych. Okna dialogowe to okna, ale z określoną intencją i środowiskiem użytkownika. W tym artykule omówiono działanie okna dialogowego oraz typy okien dialogowych, które można tworzyć i używać. Okna dialogowe służą do:

- Wyświetlanie określonych informacji użytkownikom.
- Zbieranie informacji od użytkowników.
- Wyświetlaj i zbieraj informacje.
- Wyświetl monit systemu operacyjnego, taki jak okno drukowania.
- Wybierz plik lub folder.

Tego typu okna są nazywane *oknami dialogowymi*. Okno dialogowe można wyświetlić na dwa sposoby: modalne i moderacyjne.

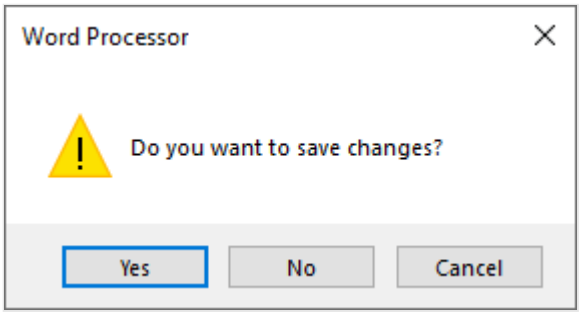
Wyświetlanie *modalnego* okna dialogowego dla użytkownika jest techniką, za pomocą której aplikacja przerywa działanie, dopóki użytkownik nie zamknie okna dialogowego. Zazwyczaj ma to postać monitu lub alertu. Inne okna w aplikacji nie mogą być wchodzić w interakcje, dopóki okno dialogowe nie zostanie zamknięte. Po zamknięciu *modalnego* okna dialogowego aplikacja będzie kontynuowana. Najbardziej typowe okna dialogowe służą do wyświetlania otwartego pliku lub monitu o zapisanie pliku, wyświetlenie okna dialogowego drukarki lub wyświetlenie komunikatu użytkownika o pewnym stanie.

Okno dialogowe *bez moderowania* nie uniemożliwia użytkownikowi aktywowania innych okien podczas jego otwierania. Jeśli na przykład użytkownik chce znaleźć wystąpienia określonego słowa w dokumencie, okno główne często otwiera okno dialogowe, aby zapytać użytkownika, jakiego słowa szuka. Ponieważ aplikacja nie chce uniemożliwiać użytkownikowi edytowania dokumentu, okno dialogowe nie musi być modalne. Modeless okno dialogowe co najmniej zapewnia przycisk **Zamknij** , aby zamknąć okno dialogowe. Inne przyciski mogą być udostępniane do uruchamiania określonych funkcji, takich jak przycisk **Znajdź dalej** , aby znaleźć następny wyraz w wyszukiwaniu wyrazów.

Za pomocą platformy WPF można utworzyć kilka typów okien dialogowych, takich jak okna dialogowe, typowe okna dialogowe i niestandardowe okna dialogowe. W tym artykule omówiono poszczególne elementy, a przykład okna dialogowego zawiera pasujące przykłady.

Pola komunikatów

Komunikat to okno dialogowe, które może służyć do wyświetlania informacji tekstowych i umożliwienia użytkownikom podejmowania decyzji za pomocą przycisków. Na poniższej ilustracji przedstawiono okno komunikatu, które zadaje pytanie i udostępnia użytkownikowi trzy przyciski umożliwiające udzielenie odpowiedzi na pytanie.



Aby utworzyć okno komunikatu, należy użyć MessageBox klasy . MessageBox Umożliwia skonfigurowanie tekstu pola komunikatu, tytułu, ikony i przycisków.

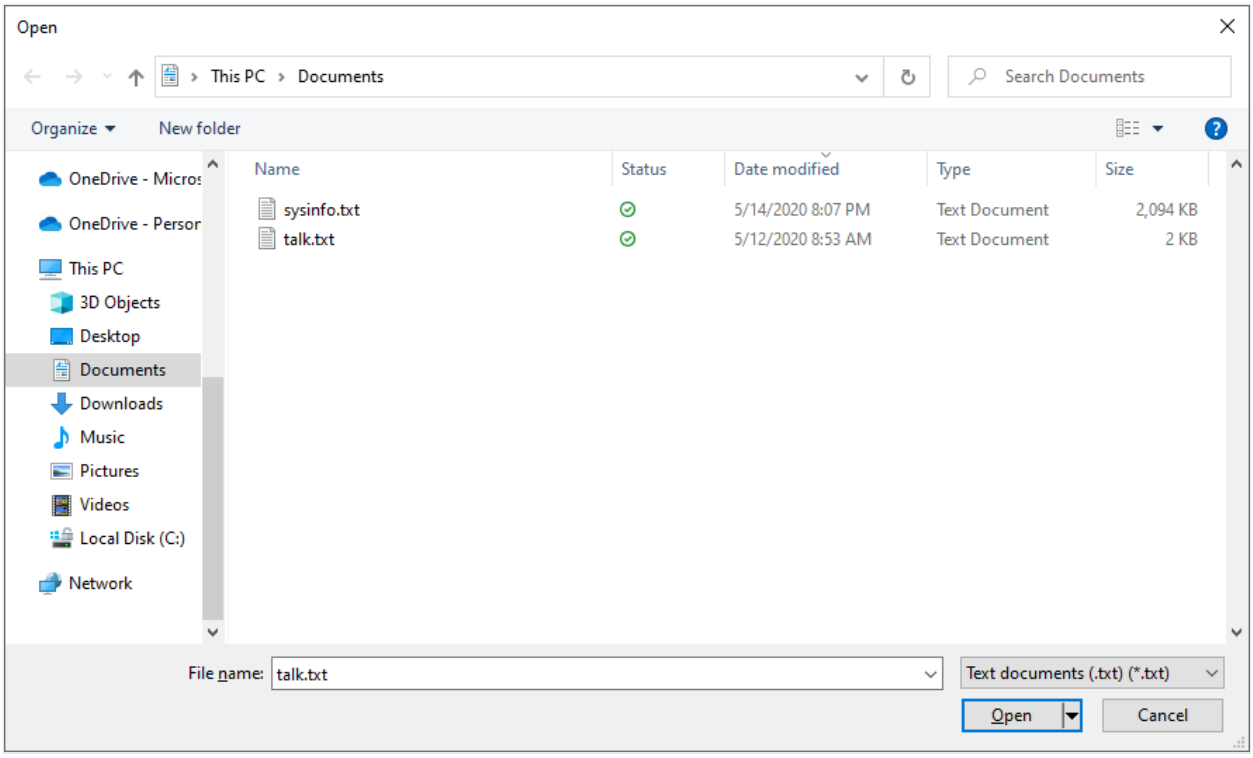
Aby uzyskać więcej informacji, zobacz Jak otworzyć okno komunikatu.

Typowe okna dialogowe

Windows implementuje różne rodzaje okien dialogowych wielokrotnego użytku, które są wspólne dla wszystkich aplikacji, w tym okna dialogowe do wybierania plików i drukowania.

Ponieważ te okna dialogowe są udostępniane przez system operacyjny, są współużytkowane przez wszystkie aplikacje działające w systemie operacyjnym. Te okna dialogowe zapewniają spójne środowisko użytkownika i są nazywane *typowymi oknami dialogowymi*. Ponieważ użytkownik używa wspólnego okna dialogowego w jednej aplikacji, nie musi nauczyć się używać tego okna dialogowego w innych aplikacjach.

WPF hermetyzuje otwarty plik, zapisz plik i wydrukuj wspólne okna dialogowe i uwidacznia je jako klasy zarządzane do użycia w aplikacjach autonomicznych.



Aby dowiedzieć się więcej na temat typowych okien dialogowych, zobacz następujące artykuły:

- Jak wyświetlić typowe okno dialogowe
- Pokaż okno dialogowe Otwieranie pliku
- Pokaż okno dialogowe Zapisywanie pliku
- Pokaż okno dialogowe Drukowanie

Niestandardowe okna dialogowe

Chociaż typowe okna dialogowe są przydatne i powinny być używane, gdy jest to możliwe, nie obsługują wymagań dotyczących okien dialogowych specyficznych dla domeny. W takich przypadkach należy utworzyć własne okna dialogowe. Jak zobaczymy, okno dialogowe jest oknem ze specjalnymi zachowaniami. Window implementuje te zachowania, a okno służy do tworzenia niestandardowych modalnych i moderacyjnych okien dialogowych.

Podczas tworzenia własnego okna dialogowego należy wziąć pod uwagę wiele zagadnień projektowych. Chociaż okno aplikacji i okno dialogowe zawierają podobieństwa, takie jak współużytkowanie tej samej klasy bazowej, okno dialogowe jest używane do określonego celu. Zazwyczaj okno dialogowe jest wymagane, gdy trzeba monitować użytkownika o jakieś informacje lub odpowiedź. Zazwyczaj aplikacja zostanie wstrzymana, gdy zostanie wyświetlone okno dialogowe (modalne), co ogranicza dostęp do pozostałej części aplikacji. Po zamknięciu okna dialogowego aplikacja będzie kontynuowana. Ograniczenie interakcji tylko do okna dialogowego nie jest wymagane.

Po zamknięciu okna WPF nie można go ponownie otworzyć. Niestandardowe okna dialogowe to okna WPF, a ta sama reguła ma zastosowanie. Aby dowiedzieć się, jak zamknąć okno, zobacz Jak zamknąć okno lub okno dialogowe.

Implementowanie okna dialogowego

Podczas projektowania okna dialogowego postępuj zgodnie z tymi sugestiami, aby utworzyć dobre środowisko użytkownika:

❌ NIE zaśmiecaj okna dialogowego. Środowisko dialogowe służy użytkownikowi do wprowadzania niektórych danych lub dokonywania wyboru.

✅ Podaj przycisk **OK** , aby zamknąć okno.

✅ Ustaw właściwość przycisku **OK**, aby `true` umożliwić użytkownikowi naciśnięcie klawisza IsDefault ENTER w celu zaakceptowania i zamknięcia okna.

✅ ROZWAŻ dodanie przycisku **Anuluj** , aby użytkownik mógł zamknąć okno i wskazać, że nie chce kontynuować.

✅ Ustaw właściwość przycisku **Anuluj**, aby `true` umożliwić użytkownikowi naciśnięcie klawisza IsCancel ESC w celu zamknięcia okna.

✅ Ustaw tytuł okna, aby dokładnie opisać, co reprezentuje okno dialogowe, lub co użytkownik powinien zrobić z oknem dialogowym.

✅ Ustaw minimalną szerokość i wartości wysokości okna, uniemożliwiając użytkownikowi zmianę rozmiaru okna za małą.

✅ ROZWAŻ wyłączenie możliwości zmiany rozmiaru okna, jeśli ShowInTaskbar jest ustawione na `false`. Zmiana rozmiaru można wyłączyć, ustawiając wartość ResizeMode na NoResize

Poniższy kod demonstruje tę konfigurację.

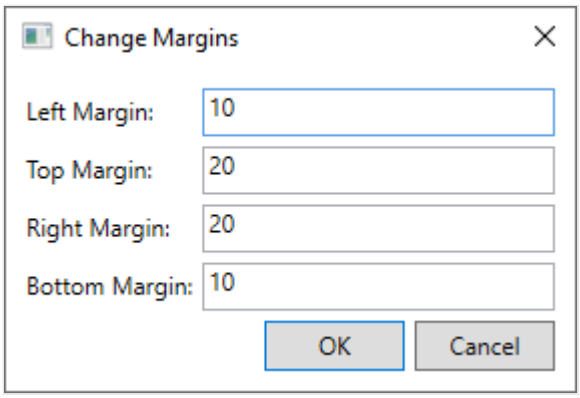
XAML

```
<Window x:Class="Dialogs.Margins"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Change Margins"
        Closing="Window_Closing"
        MinHeight="200"
        MinWidth="300"
        SizeToContent="WidthAndHeight"
        ResizeMode="NoResize"
        ShowInTaskbar="False"
```

```
        WindowStartupLocation="CenterOwner"
        FocusManager.FocusedElement="{Binding
ElementName=leftMarginTextBox}">
    <Grid Margin="10">
        <Grid.Resources>
            <!-- Default settings for controls -->
            <Style TargetType="{x:Type Label}">
                <Setter Property="Margin" Value="0,3,5,5" />
                <Setter Property="Padding" Value="0,0,0,5" />
            </Style>
            <Style TargetType="{x:Type TextBox}">
                <Setter Property="Margin" Value="0,0,0,5" />
            </Style>
            <Style TargetType="{x:Type Button}">
                <Setter Property="Width" Value="70" />
                <Setter Property="Height" Value="25" />
                <Setter Property="Margin" Value="5,0,0,0" />
            </Style>
        </Grid.Resources>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <!-- Left,Top,Right,Bottom margins-->
        <Label Grid.Column="0" Grid.Row="0">Left Margin:</Label>
        <TextBox Name="leftMarginTextBox" Grid.Column="1" Grid.Row="0" />

        <Label Grid.Column="0" Grid.Row="1">Top Margin:</Label>
        <TextBox Name="topMarginTextBox" Grid.Column="1" Grid.Row="1"/>

        <Label Grid.Column="0" Grid.Row="2">Right Margin:</Label>
        <TextBox Name="rightMarginTextBox" Grid.Column="1" Grid.Row="2" />

        <Label Grid.Column="0" Grid.Row="3">Bottom Margin:</Label>
        <TextBox Name="bottomMarginTextBox" Grid.Column="1" Grid.Row="3" />

        <!-- Accept or Cancel -->
        <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="4"
Orientation="Horizontal" HorizontalAlignment="Right">
            <Button Name="okButton" Click="okButton_Click"
IsDefault="True">OK</Button>
            <Button Name="cancelButton" IsCancel="True">Cancel</Button>
        </StackPanel>
```

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

Powyższy kod XAML tworzy okno podobne do poniższego obrazu:



Elementy interfejsu użytkownika otwierające okno dialogowe

Środowisko użytkownika okna dialogowego jest również rozszerzane na pasek menu lub przycisk okna, który go otwiera. Gdy element menu lub przycisk uruchamia funkcję, która wymaga interakcji użytkownika za pośrednictwem okna dialogowego, zanim funkcja będzie mogła kontynuować, kontrolka powinna używać wielokropka na końcu tekstu nagłówka:

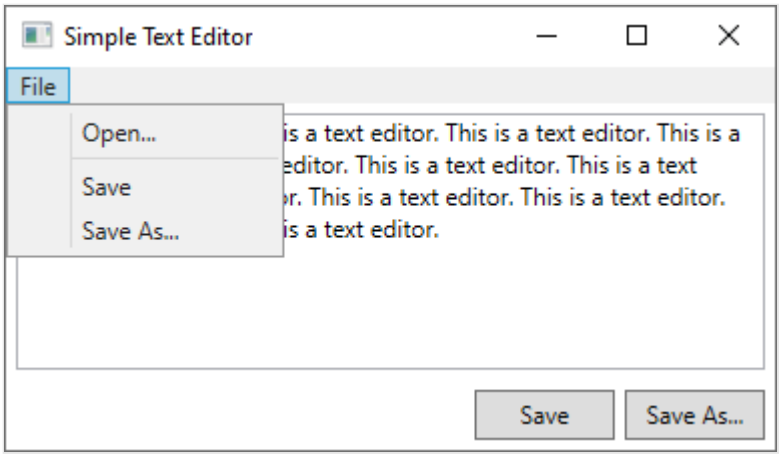
XAML

```
<MenuItem Header="_Margins..." Click="formatMarginsMenuItem_Click" />
<!-- or -->
<Button Content="_Margins..." Click="formatMarginsButton_Click" />
```

Gdy element menu lub przycisk uruchamia funkcję, która wyświetla okno dialogowe, które **nie** wymaga interakcji użytkownika, takie jak okno dialogowe *Informacje* , wielokropek nie jest wymagany.

Elementy menu

Elementy menu to typowy sposób udostępniania użytkownikom akcji aplikacji pogrupowanych w powiązane motywy. Prawdopodobnie znasz menu *Plik* w wielu różnych aplikacjach. W typowej aplikacji element menu *Plik* udostępnia sposoby zapisywania pliku, ładowania pliku i drukowania pliku. Jeśli akcja będzie wyświetlać modalne okno, nagłówek zwykle zawiera wielokropek, jak pokazano na poniższej ilustracji:

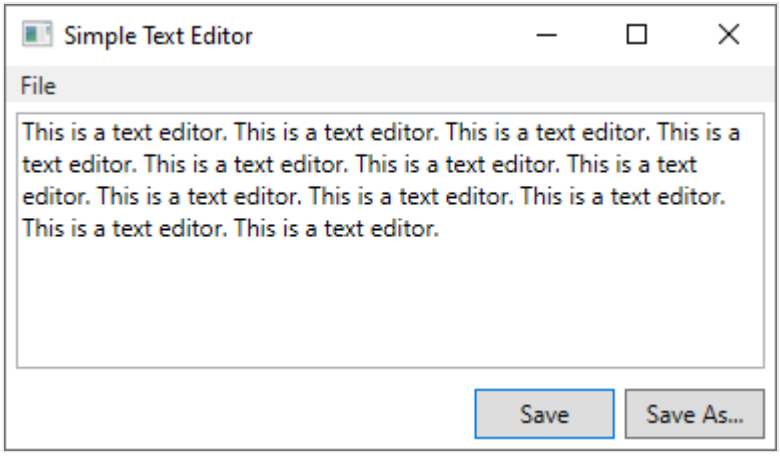


Dwa elementy menu mają wielokropek: `...`. Dzięki temu użytkownik może zidentyfikować, że po wybraniu tych elementów menu zostanie wyświetlone modalne okno, wstrzymując aplikację do momentu zamknięcia jej przez użytkownika.

Ta technika projektowania jest łatwym sposobem komunikowania się użytkownikom, czego powinni oczekiwać.

Przyciski

Możesz postępować zgodnie z tą samą zasadą opisaną w sekcji Elementy menu . Użyj wielokropka w tekście przycisku, aby wskazać, że gdy użytkownik naciśnie przycisk, zostanie wyświetlone modalne okno dialogowe. Na poniższej ilustracji znajdują się dwa przyciski i można łatwo zrozumieć, który przycisk wyświetla okno dialogowe:



Zwracanie wyniku

Otwarcie innego okna, zwłaszcza modalnego okna dialogowego, to doskonały sposób zwracania stanu i informacji do wywoływania kodu.

Modalne okna dialogowe

Gdy okno dialogowe jest wyświetlane przez wywołanie ShowDialog()metody , kod, który otworzył okno dialogowe, czeka na `ShowDialog` zwrócenie metody. Po powrocie metody kod, który go nazwał, musi zdecydować, czy kontynuować przetwarzanie, czy zatrzymać przetwarzanie. Użytkownik zazwyczaj wskazuje to, naciskając **przycisk OK** lub **Anuluj** w oknie dialogowym.

Po naciśnięciu przycisku **OK** należy go zaprojektować tak, `ShowDialog` aby zwracał `true`wartość , i przycisk **Anuluj** , aby zwrócić wartość `false`. Można to osiągnąć, ustawiając DialogResult właściwość po naciśnięciu przycisku.

C#

```
private void okButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = true;

private void cancelButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = false;
```

VB

```
Private Sub okButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = True
End Sub

Private Sub cancelButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = False
End Sub
```

Właściwość DialogResult można ustawić tylko wtedy, gdy okno dialogowe było wyświetlane za pomocą ShowDialog()polecenia . Po ustawieniu `DialogResult` właściwości okno dialogowe zostanie zamknięte.

Jeśli właściwość przycisku jest ustawiona IsCancel na `true`, a okno zostanie otwarte za pomocą ShowDialog()klawisza `ESC` , zamknie okno i ustawi wartość `DialogResult` `false`.

Aby uzyskać więcej informacji na temat zamykania okien dialogowych, zobacz Jak zamknąć okno lub okno dialogowe.

Przetwarzanie odpowiedzi

Funkcja ShowDialog() zwraca wartość logiczną, aby wskazać, czy użytkownik zaakceptował, czy anulował okno dialogowe. Jeśli powiadamiasz użytkownika o czymś, ale nie wymagasz podjęcia decyzji lub podania danych, możesz zignorować odpowiedź.

Odpowiedź można również sprawdzić, sprawdzając DialogResult właściwość . Poniższy kod pokazuje, jak przetworzyć odpowiedź:

C#

```
var dialog = new Margins();

// Display the dialog box and read the response
bool? result = dialog.ShowDialog();

if (result == true)
{
    // User accepted the dialog box
    MessageBox.Show("Your request will be processed.");
}
else
{
    // User cancelled the dialog box
    MessageBox.Show("Sorry it didn't work out, we'll try again later.");
}
```

VB

```
Dim marginsWindow As New Margins

Dim result As Boolean? = marginsWindow.ShowDialog()

If result = True Then
    ' User accepted the dialog box
    MessageBox.Show("Your request will be processed.")
Else
    ' User cancelled the dialog box
    MessageBox.Show("Sorry it didn't work out, we'll try again later.")
End If

marginsWindow.Show()
```

Okno dialogowe trybu bez moderowania

Aby wyświetlić okno dialogowe w trybie bez moderowania, wywołaj metodę Show(). Okno dialogowe powinno zawierać przynajmniej przycisk **Zamknij** . Inne przyciski i elementy interaktywne można podać, aby uruchomić określoną funkcję, taką jak przycisk **Znajdź dalej** , aby znaleźć następny wyraz w wyszukiwaniu wyrazów.

Ponieważ niemodalne okno dialogowe nie blokuje kontynuowania kodu wywołującego, musisz podać inny sposób zwracania wyniku. Możesz wykonać jedną z następujących czynności:

- Uwidacznia właściwość obiektu danych w oknie.
- Obsługa zdarzenia Window.Closed w kodzie wywołującym.
- Utwórz zdarzenia w oknie, które są wywoływane, gdy użytkownik wybierze obiekt lub naciśnie określony przycisk.

W poniższym przykładzie Window.Closed zdarzenie jest używane do wyświetlania użytkownikowi okna dialogowego po zamknięciu okna dialogowego. Wyświetlony komunikat odwołuje się do właściwości zamkniętego okna dialogowego. Aby uzyskać więcej informacji na temat zamykania okien dialogowych, zobacz Jak zamknąć okno lub okno dialogowe.

C#

```
var marginsWindow = new Margins();

marginsWindow.Closed += (sender, eventArgs) =>
{
    MessageBox.Show($"You closed the margins window! It had the title of
{marginsWindow.Title}");
};

marginsWindow.Show();
```

VB

```
Dim marginsWindow As New Margins

AddHandler marginsWindow.Closed, Sub(sender As Object, e As EventArgs)
                                     MessageBox.Show($"You closed the
margins window! It had the title of {marginsWindow.Title}")
                                 End Sub

marginsWindow.Show()
```

Zobacz też

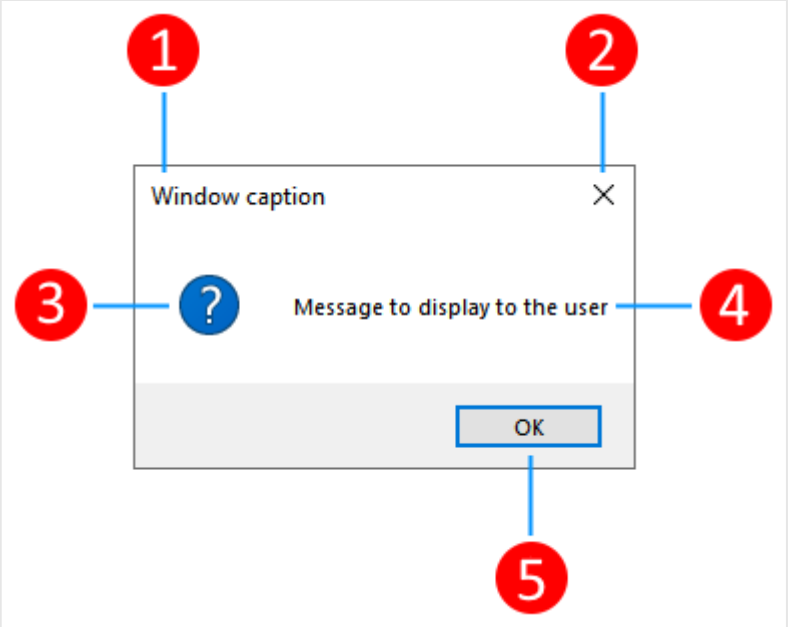
- Omówienie okien WPF
- Jak otworzyć okno lub okno dialogowe
- Jak otworzyć typowe okno dialogowe
- Jak otworzyć okno komunikatu
- Jak zamknąć okno lub okno dialogowe
- Przykładowe okno dialogowe
- System.Windows.Window
- System.Windows.MessageBox

Jak otworzyć pole komunikatu (WPF .NET)

Artykuł • 22.02.2023 • Czas czytania: 2 min

Komunikat to okno dialogowe, które służy do szybkiego wyświetlania informacji i opcjonalnie umożliwia użytkownikom podejmowanie decyzji. Dostęp do pola komunikatu jest udostępniany przez klasę MessageBox . Zostanie wyświetlone okno komunikatu *modalnie*. Kod wyświetlający pole komunikatu jest wstrzymany, dopóki użytkownik nie zamknie pola komunikatu za pomocą przycisku zamknij lub przycisku odpowiedzi.

Poniższa ilustracja przedstawia części pola komunikatu:



- Pasek tytułu z podpisem (1).
- Przycisk zamknięcia (2).
- Ikona (3).
- Komunikat wyświetlany użytkownikowi (4).
- Przyciski odpowiedzi (5).

W przypadku prezentowania lub zbierania złożonych danych okno dialogowe może być bardziej odpowiednie niż okno dialogowe. Aby uzyskać więcej informacji, zobacz Omówienie okien dialogowych.

Wyświetlanie pola komunikatu

Aby utworzyć pole komunikatu MessageBox , należy użyć klasy . Metoda MessageBox.Show umożliwia skonfigurowanie tekstu pola wiadomości, tytułu, ikony i przycisków pokazanych w następującym kodzie:

C#

```
string messageBoxText = "Do you want to save changes?";
string caption = "Word Processor";
MessageBoxButton button = MessageBoxButton.YesNoCancel;
MessageBoxImage icon = MessageBoxImage.Warning;
MessageBoxResult result;

result = MessageBox.Show(messageBoxText, caption, button, icon,
MessageBoxResult.Yes);
```

MessageBox.Show Przeciążenia metody zapewniają sposoby konfigurowania pola komunikatu. Dostępne są następujące opcje:

- **Podpis** paska tytułu
- **Ikona** wiadomości
- **Tekst** wiadomości
- **Przyciski** odpowiedzi

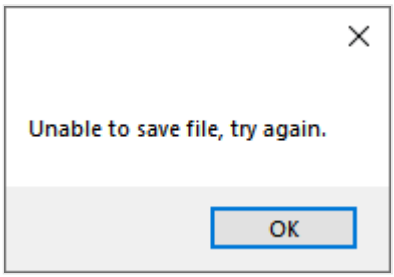
Poniżej przedstawiono kilka przykładów użycia pola komunikatu.

- Wyświetlanie alertu.

  C#

  ```
  MessageBox.Show("Unable to save file, try again.");
  ```

  Poprzedni kod wyświetla pole komunikatu, takie jak na poniższej ilustracji:

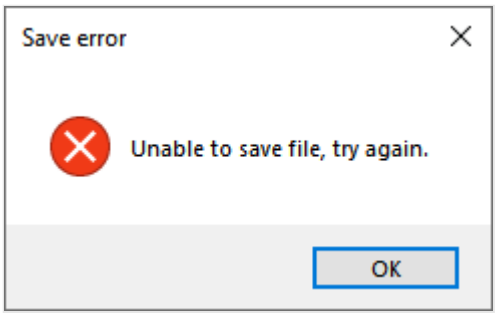
  

  Dobrym pomysłem jest użycie opcji udostępnianych przez klasę pola komunikatów. Używając tego samego alertu co wcześniej, ustaw więcej opcji, aby uczynić go bardziej atrakcyjnym wizualnie:

  C#

```
MessageBox.Show("Unable to save file, try again.", "Save error",
MessageBoxButton.OK, MessageBoxImage.Error);
```

Poprzedni kod wyświetla pole komunikatu, takie jak na poniższej ilustracji:

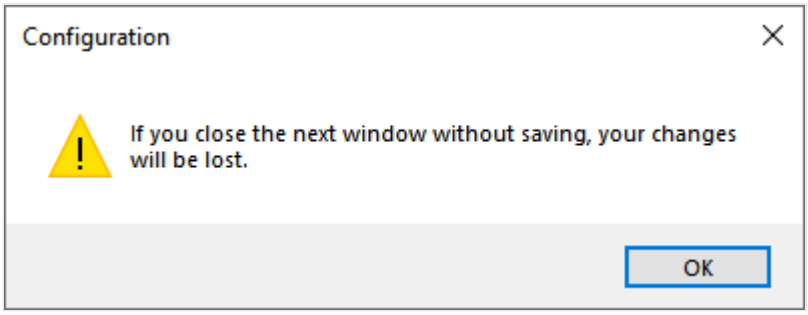


- Wyświetl ostrzeżenie.

C#

```
MessageBox.Show("If you close the next window without saving, your
changes will be lost.", "Configuration", MessageBoxButton.OK,
MessageBoxImage.Warning);
```

Poprzedni kod wyświetla pole komunikatu, takie jak na poniższej ilustracji:

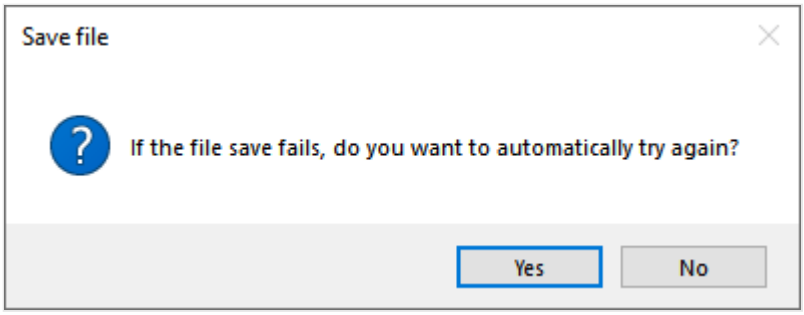


- Zadaj użytkownikowi pytanie.

C#

```
if (MessageBox.Show("If the file save fails, do you want to
automatically try again?",
                    "Save file",
                    MessageBoxButton.YesNo,
                    MessageBoxImage.Question) == MessageBoxResult.Yes)
{
    // Do something here
}
```

Poprzedni kod wyświetla pole komunikatu, takie jak na poniższej ilustracji:



Obsługa odpowiedzi pola komunikatu

Metoda MessageBox.Show wyświetla pole komunikatu i zwraca wynik. Wynik wskazuje, jak użytkownik zamknął pole komunikatu:

C#

```
result = MessageBox.Show(messageBoxText, caption, button, icon,
MessageBoxResult.Yes);

switch (result)
{
    case MessageBoxResult.Cancel:
        // User pressed Cancel
        break;
    case MessageBoxResult.Yes:
        // User pressed Yes
        break;
    case MessageBoxResult.No:
        // User pressed No
        break;
}
```

Gdy użytkownik naciska przyciski w dolnej części pola komunikatu, zostanie zwrócony odpowiedni MessageBoxResult element. Jeśli jednak użytkownik naciska klawisz ESC lub naciska przycisk **Zamknij** (#2 na ilustracji pola komunikatu), wynik okna komunikatu różni się w zależności od opcji przycisku:

Opcje przycisków	**Wynik przycisku** ESC **lub** Zamknij
`OK`	`OK`
`OKCancel`	`Cancel`
`YesNo`	Skrót klawiaturowy ESC i wyłączony przycisk **Zamknij** . Użytkownik musi nacisnąć **przycisk Tak** lub **Nie**.
`YesNoCancel`	`Cancel`

Aby uzyskać więcej informacji na temat używania pól komunikatów, zobacz MessageBox i przykład MessageBox .

Zobacz też

- Omówienie okien WPF
- Omówienie okien dialogowych
- Jak wyświetlić typowe okno dialogowe
- Przykład MessageBox
- System.Windows.MessageBox
- System.Windows.MessageBox.Show
- System.Windows.MessageBoxResult

Jak otworzyć okno lub okno dialogowe (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 2 min

Możesz tworzyć własne okna i wyświetlać je w Windows Presentation Foundation (WPF). W tym artykule dowiesz się, jak wyświetlać modalne i niemodalne okna i okna dialogowe.

Koncepcyjnie okno i okno dialogowe są takie same: są wyświetlane użytkownikowi w celu podania informacji lub interakcji. Są to obiekty "okien". Projekt okna i sposób jego obsługi to właśnie to, co tworzy okno dialogowe. Okno dialogowe ma zazwyczaj niewielki rozmiar i wymaga od użytkownika odpowiedzi. Aby uzyskać więcej informacji, zobacz Overview of WPF windows and Dialog boxes overview (Omówienie okien WPF i okien dialogowych).

Jeśli interesuje Cię otwieranie okien dialogowych systemu operacyjnego, zobacz Jak otworzyć typowe okno dialogowe.

Otwórz jako modalny

Po otwarciu modalnego okna zazwyczaj reprezentuje okno dialogowe. WPF ogranicza interakcję do okna modalnego, a kod, który otworzył okno, jest wstrzymywany do momentu zamknięcia okna. Ten mechanizm zapewnia prosty sposób monitowania użytkownika o dane i oczekiwania na odpowiedź.

Użyj metody ShowDialog , aby otworzyć okno. Poniższy kod iniekuje okno i otwiera je modalnie. Kod otwierający okno jest wstrzymywany i oczekuje na zamknięcie okna:

C#

```
var window = new Margins();

window.Owner = this;
window.ShowDialog();
```

Ważne

Po zamknięciu okna tego samego wystąpienia obiektu nie można użyć do ponownego otwarcia okna.

Aby uzyskać więcej informacji na temat sposobu obsługi odpowiedzi użytkownika na okno dialogowe, zobacz Omówienie okien dialogowych: Przetwarzanie odpowiedzi.

Otwórz jako bez moderów

Otwieranie okna bez trybu oznacza wyświetlanie go jako normalnego okna. Kod, który otwiera okno, będzie nadal działać, gdy okno stanie się widoczne. Możesz skupić się na wszystkich oknach bez trybów wyświetlanych przez aplikację i wchodzić z nimi w interakcje bez ograniczeń.

Użyj metody Show , aby otworzyć okno. Poniższy kod umożliwia otwarcie okna bez moderowania. Kod otwierający okno będzie nadal działać:

C#

```
var window = new Margins();

window.Owner = this;
window.Show();
```

> ⓘ **Ważne**
>
> Po zamknięciu okna tego samego wystąpienia obiektu nie można użyć do ponownego otwarcia okna.

Zobacz też

- Omówienie okien WPF
- Omówienie okien dialogowych
- Jak zamknąć okno lub okno dialogowe
- Jak otworzyć wspólne okno dialogowe
- Jak otworzyć okno komunikatu
- System.Windows.Window
- System.Windows.Window.DialogResult
- System.Windows.Window.Show()
- System.Windows.Window.ShowDialog()

Jak zamknąć okno lub okno dialogowe (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 3 min

W tym artykule poznasz różne sposoby zamykania okna lub okna dialogowego. Użytkownik może zamknąć okno przy użyciu elementów w obszarze innym niż klient, w tym następujących:

- Element **Zamknijw menu** System.
- Naciskając `klawisze ALT + F4`.
- Naciskając **przycisk Zamknij** .
- Naciśnięcie `klawisza ESC` , gdy przycisk ma właściwość IsCancel ustawioną na `true` w oknie modalnego.

Podczas projektowania okna należy zapewnić obszarowi klienta więcej mechanizmów zamykania okna. Oto niektóre typowe elementy projektu okna używane do jego zamknięcia:

- Element **Exit** (Zakończ) w menu **File** (Plik), zazwyczaj dla głównych okien aplikacji.
- Element **Zamknij** w menu **Plik** , zazwyczaj w oknie aplikacji pomocniczej.
- Przycisk **Anuluj** , zazwyczaj w modalnych oknach dialogowych.
- Przycisk **Zamknij** , zazwyczaj w nie moderowym oknie dialogowym.

> ⓘ **Ważne**
>
> Po zamknięciu okna tego samego wystąpienia obiektu nie można użyć do ponownego otwarcia okna.

Aby uzyskać więcej informacji o cyklu życia okna, zobacz Przegląd okien WPF: okres istnienia okna.

Zamykanie okna modalnego

Podczas zamykania okna otwartego ShowDialogDialogResult`true` za pomocą metody ustaw właściwość odpowiednio na wartość lub `false` , aby wskazać stan "zaakceptowano" lub "anulowano". Gdy tylko właściwość `DialogResult` zostanie ustawiona na wartość, okno zostanie zamknięte. Poniższy kod demonstruje ustawianie `DialogResult` właściwości :

C#

```
private void okButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = true;

private void cancelButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = false;
```

Możesz również wywołać metodę Close . Jeśli jest `Close` używana metoda , właściwość DialogResult jest ustawiona na `false`.

Po zamknięciu okna nie można otworzyć go ponownie z tym samym wystąpieniem obiektu. Jeśli spróbujesz wyświetlić to samo okno, zostanie zgłoszony InvalidOperationException wyjątek . Zamiast tego utwórz nowe wystąpienie okna i otwórz je.

Zamykanie okna bez moderowania

Podczas zamykania okna otwartego za pomocą Show metody użyj Close metody . Poniższy kod demonstruje zamknięcie okna bez modeless:

C#

```
private void closeButton_Click(object sender, RoutedEventArgs e) =>
    Close();
```

Close with IsCancel

Właściwość Button.IsCancel można ustawić na , aby `true` umożliwić automatyczne zamknięcie okna za pomocą klawisza ESC . Ta metoda działa tylko wtedy, gdy okno zostanie otwarte przy użyciu ShowDialog metody .

XAML

```
<Button Name="cancelButton" IsCancel="True">Cancel</Button>
```

Ukrywanie okna

Zamiast zamykać okno, można je ukryć za pomocą Hide metody . Ukryte okno można otworzyć ponownie, w przeciwieństwie do okna, które zostało zamknięte. Jeśli

zamierzasz ponownie użyć wystąpienia obiektu okna, ukryj okno zamiast go zamykać. Poniższy kod demonstruje ukrywanie okna:

```
C#

private void saveButton_Click(object sender, RoutedEventArgs e) =>
    Hide();
```

Anulowanie zamknięcia i ukrycia

Jeśli przyciski zostały zaprojektowane tak, aby ukrywały okno zamiast go zamykać, użytkownik nadal może pominąć to i zamknąć okno. Zamknij **element** menu systemowego i przycisk **Zamknij** obszaru okna, który nie jest klientem, spowoduje zamknięcie okna zamiast go ukryć. Rozważ ten scenariusz, gdy twoim celem jest ukrycie okna zamiast jego zamykania.

> ⊗ **Przestroga**
>
> Jeśli okno jest wyświetlane modalnie za pomocą **ShowDialog**, **DialogResult** właściwość zostanie ustawiona na wartość `null` , gdy okno jest ukryte. Musisz przekazać stan z powrotem do kodu wywołującego, dodając własną właściwość do okna.

Po zamknięciu okna jest wywoływane Closing zdarzenie. Procedura obsługi jest przekazywana do CancelEventArgsobiektu , który implementuje Cancel właściwość . Ustaw dla tej właściwości wartość , `true` aby zapobiec zamykaniu okna. Poniższy kod pokazuje, jak anulować zamknięcie, a zamiast tego ukryć okno:

```
C#

private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    // Cancel the closure
    e.Cancel = true;

    // Hide the window
    Hide();
}
```

Czasami nie chcesz ukrywać okna, ale w rzeczywistości uniemożliwiasz użytkownikowi jego zamknięcie. Aby uzyskać więcej informacji, zobacz Omówienie okien WPF: Anulowanie zamknięcia okna.

Zobacz też

- Omówienie okien WPF
- Omówienie okien dialogowych
- Jak otworzyć okno lub okno dialogowe
- System.Windows.Window.Close()
- System.Windows.Window.Closing
- System.Windows.Window.DialogResult
- System.Windows.Window.Hide()
- System.Windows.Window.Show()
- System.Windows.Window.ShowDialog()

Jak otworzyć typowe okno dialogowe (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 3 min

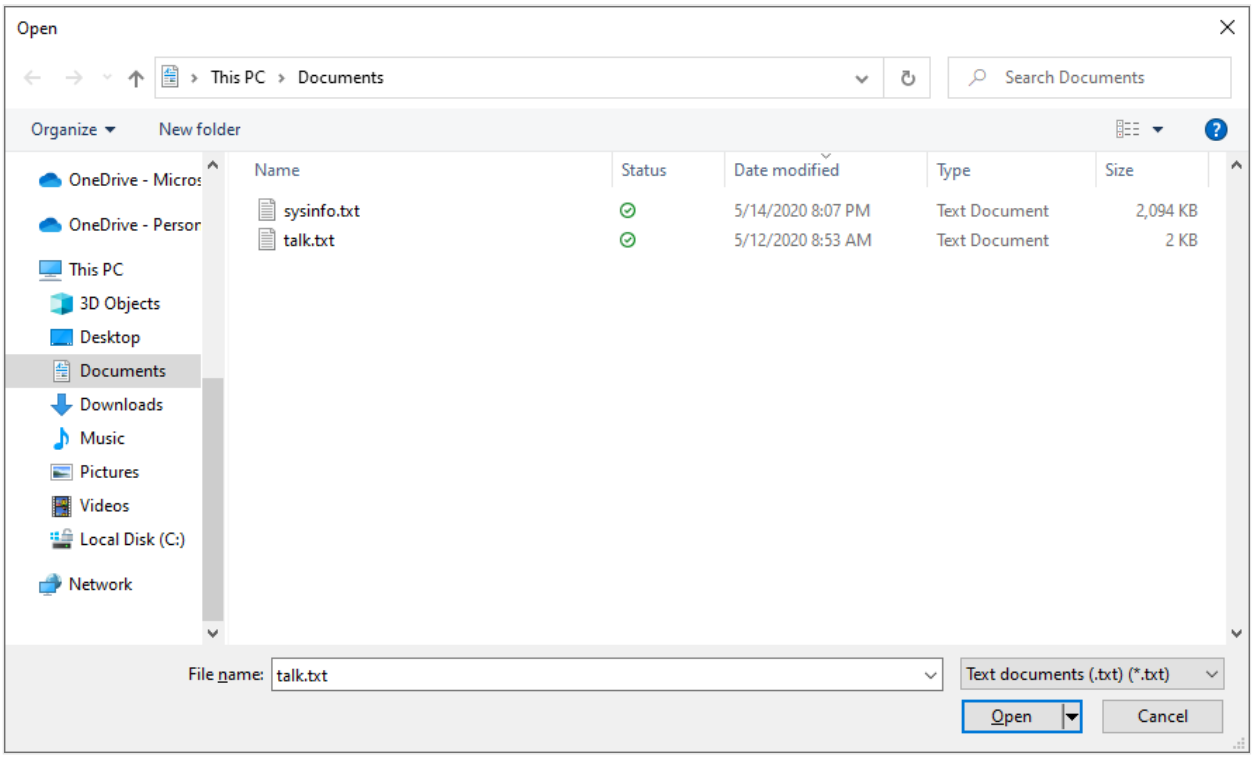
W tym artykule pokazano, jak można wyświetlić typowe okno dialogowe systemu w Windows Presentation Foundation (WPF). System Windows implementuje różne rodzaje okien dialogowych wielokrotnego użytku, które są wspólne dla wszystkich aplikacji, w tym okna dialogowe do wybierania plików i drukowania.

Ponieważ te okna dialogowe są udostępniane przez system operacyjny, są współużytkowane przez wszystkie aplikacje działające w systemie operacyjnym. Te okna dialogowe zapewniają spójne środowisko użytkownika i są nazywane *typowymi oknami dialogowymi*. Ponieważ użytkownik używa wspólnego okna dialogowego w jednej aplikacji, nie musi nauczyć się używać tego okna dialogowego w innych aplikacjach.

Okno komunikatu to inne typowe okno dialogowe. Aby uzyskać więcej informacji, zobacz Jak otworzyć okno komunikatu.

Okno dialogowe Otwieranie pliku

Otwarte okno dialogowe pliku jest używane przez funkcję otwierania pliku w celu pobrania nazwy pliku do otwarcia.



Typowe otwarte okno dialogowe pliku jest implementowane jako OpenFileDialog klasa i znajduje się w Microsoft.Win32 przestrzeni nazw. Poniższy kod pokazuje, jak utworzyć, skonfigurować i pokazać jeden oraz jak przetworzyć wynik.

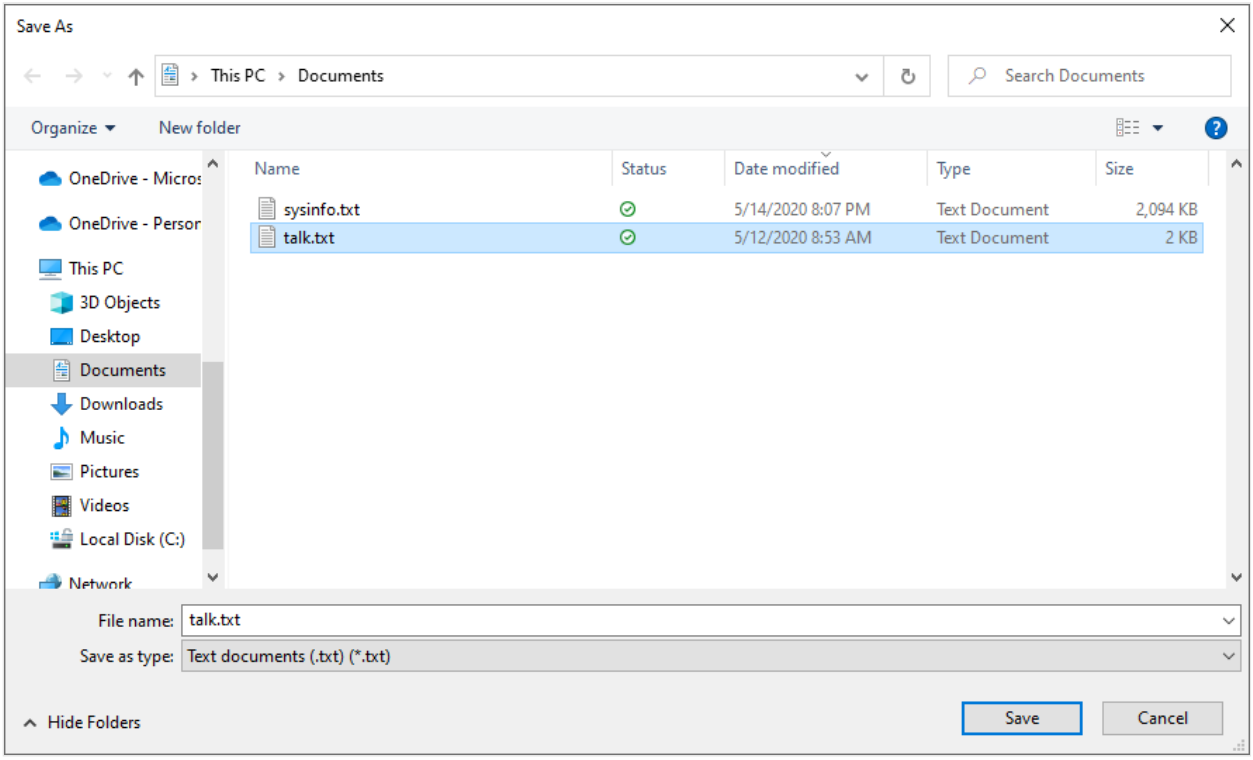
C#

```
// Configure open file dialog box
var dialog = new Microsoft.Win32.OpenFileDialog();
dialog.FileName = "Document"; // Default file name
dialog.DefaultExt = ".txt"; // Default file extension
dialog.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension

// Show open file dialog box
bool? result = dialog.ShowDialog();

// Process open file dialog box results
if (result == true)
{
    // Open document
    string filename = dialog.FileName;
}
```

Aby uzyskać więcej informacji na temat okna dialogowego otwierania pliku, zobacz Microsoft.Win32.OpenFileDialog.

Zapisz Plik — Okno dialogowe

Okno dialogowe zapisywania pliku jest używane przez funkcję zapisywania plików w celu pobrania nazwy pliku do zapisania.



Typowe okno dialogowe zapisywania pliku jest implementowane jako SaveFileDialog klasa i znajduje się w Microsoft.Win32 przestrzeni nazw. Poniższy kod pokazuje, jak utworzyć, skonfigurować i pokazać jeden oraz jak przetworzyć wynik.

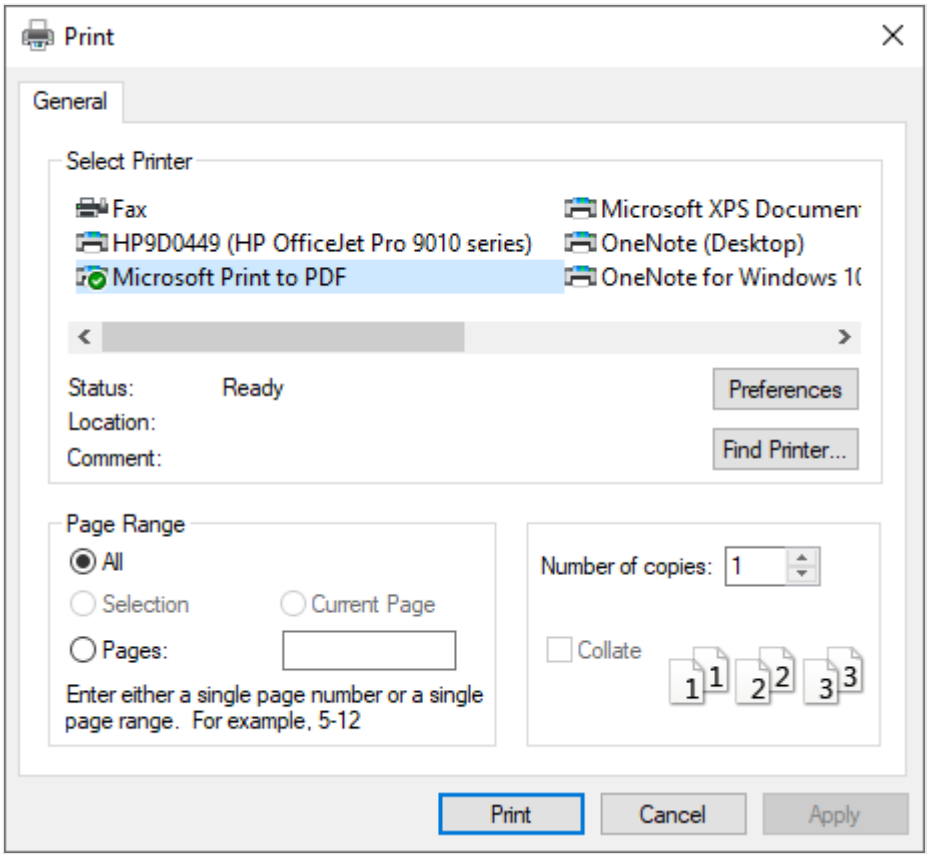
C#

```
// Configure save file dialog box
var dialog = new Microsoft.Win32.SaveFileDialog();
dialog.FileName = "Document"; // Default file name
dialog.DefaultExt = ".txt"; // Default file extension
dialog.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension

// Show save file dialog box
bool? result = dialog.ShowDialog();

// Process save file dialog box results
if (result == true)
{
    // Save document
    string filename = dialog.FileName;
}
```

Aby uzyskać więcej informacji na temat okna dialogowego zapisywania pliku, zobacz Microsoft.Win32.SaveFileDialog.

Drukuj — Okno dialogowe

Okno dialogowe drukowania służy do wybierania i konfigurowania drukarki, do której użytkownik chce drukować dane.



Wspólne okno dialogowe drukowania jest implementowane jako PrintDialog klasa i znajduje się w System.Windows.Controls przestrzeni nazw. Poniższy kod pokazuje, jak utworzyć, skonfigurować i pokazać go.

C#

```
// Configure printer dialog box
var dialog = new System.Windows.Controls.PrintDialog();
dialog.PageRangeSelection =
System.Windows.Controls.PageRangeSelection.AllPages;
dialog.UserPageRangeEnabled = true;

// Show save file dialog box
bool? result = dialog.ShowDialog();

// Process save file dialog box results
if (result == true)
{
    // Document was printed
}
```

Aby uzyskać więcej informacji na temat okna dialogowego drukowania, zobacz System.Windows.Controls.PrintDialog. Aby uzyskać szczegółowe omówienie drukowania w WPF, zobacz Omówienie drukowania.

Zobacz też

- Jak otworzyć okno komunikatu
- Przegląd okien dialogowych
- Omówienie okien WPF
- Microsoft.Win32.OpenFileDialog
- Microsoft.Win32.SaveFileDialog
- System.Windows.Controls.PrintDialog

Jak pobrać lub ustawić główne okno aplikacji (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 2 min

W tym artykule opisano sposób uzyskania lub ustawienia głównego okna aplikacji dla Windows Presentation Foundation (WPF). Pierwsze, Window które jest tworzone w aplikacji WPF, jest automatycznie ustawiane Application przez element jako główne okno aplikacji. Do okna głównego odwołuje się właściwość Application.MainWindow .

Przez większość czasu szablon projektu będzie ustawiany Application.StartupUri na plik XAML w aplikacji, taki jak _Window1.xaml_. Jest to pierwsze okno, które zostanie wyświetlone przez aplikację i zostanie wyświetlone w oknie głównym.

> **Porada**
>
> Domyślnym zachowaniem aplikacji jest zamknięcie po zamknięciu ostatniego okna. To zachowanie jest kontrolowane przez właściwość **Application.ShutdownMode** . Zamiast tego można skonfigurować aplikację do zamykania w przypadku **MainWindow** zamknięcia. Ustaw **Application.ShutdownMode** wartość na , **OnMainWindowClose** aby włączyć to zachowanie.

Ustawianie okna głównego w języku XAML

Szablony, które generują aplikację WPF, zazwyczaj ustawiają Application.StartupUri właściwość na plik XAML. Ta właściwość jest przydatna, ponieważ:

1. Można go łatwo zmienić na inny plik XAML w projekcie.
2. Automatycznie tworzy wystąpienia i wyświetla określone okno.
3. Określone okno staje się oknem Application.MainWindow.

XAML

```
<Application x:Class="MainApp.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:MainApp"
             StartupUri="Window1.xaml">

</Application>
```

Zamiast używać elementu Application.StartupUri, można ustawić wartość Application.MainWindow na okno zadeklarowane w języku XAML. Jednak określone tutaj okno nie będzie wyświetlane i należy ustawić jego widoczność.

XAML

```
<Application x:Class="MainApp.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:MainApp">

    <Application.MainWindow>
        <local:Window2 Visibility="Visible" />
    </Application.MainWindow>
</Application>
```

⊗ Przestroga

Jeśli ustawisz właściwości **Application.StartupUri** i **Application.MainWindow** , oba okna będą wyświetlane podczas uruchamiania aplikacji.

Ponadto możesz użyć zdarzenia Application.Startup , aby otworzyć okno. Aby uzyskać więcej informacji, zobacz Używanie zdarzenia uruchamiania do otwierania okna.

Ustawianie okna głównego w kodzie

Pierwsze okno tworzone przez aplikację automatycznie staje się głównym oknem i jest ustawiane na właściwość Application.MainWindow . Aby ustawić inne okno główne, zmień tę właściwość na okno:

C#

```
Application.Current.MainWindow = new Window2();

Application.Current.MainWindow.Show();
```

Jeśli aplikacja nigdy nie utworzyła wystąpienia okna, następujący kod jest funkcjonalnie równoważny z poprzednim kodem:

C#

```
var appWindow = new Window2();
```

```
appWindow.Show();
```

Po utworzeniu wystąpienia obiektu okna jest ono przypisywane do klasy Application.MainWindow.

Uzyskiwanie okna głównego

Aby uzyskać dostęp do okna wybranego jako okno główne, sprawdź właściwość Application.MainWindow . Poniższy kod wyświetla okno komunikatu z tytułem okna głównego po kliknięciu przycisku:

C#

```
private void Button_Click(object sender, RoutedEventArgs e) =>
    MessageBox.Show($"The main window's title is:
{Application.Current.MainWindow.Title}");
```

Zobacz też

- Omówienie okien WPF
- Otwieranie okna za pomocą zdarzenia uruchamiania
- Jak otworzyć okno lub okno dialogowe
- System.Windows.Application
- System.Windows.Application.MainWindow
- System.Windows.Application.StartupUri
- System.Windows.Application.ShutdownMode

Style i szablony (WPF .NET)

Artykuł • 04.03.2023 • Czas czytania: 10 min

Windows Presentation Foundation (WPF) style i tworzenie szablonów odnoszą się do zestawu funkcji, które pozwalają deweloperom i projektantom tworzyć wizualnie atrakcyjne efekty i spójny wygląd dla swojego produktu. Podczas dostosowywania wyglądu aplikacji potrzebujesz silnego modelu stylów i tworzenia szablonów, który umożliwia konserwację i udostępnianie wyglądu w aplikacjach i wśród nich. WPF zapewnia ten model.

Inną cechą modelu stylów WPF jest rozdzielenie prezentacji i logiki. Projektanci mogą pracować nad wyglądem aplikacji, używając jednocześnie tylko języka XAML, który deweloperzy pracują nad logiką programowania przy użyciu języka C# lub Visual Basic.

To omówienie koncentruje się na aspektach stylów i tworzenia szablonów aplikacji i nie omawia żadnych pojęć związanych z powiązaniem danych. Aby uzyskać informacje o powiązaniu danych, zobacz Omówienie powiązania danych.

Ważne jest, aby zrozumieć zasoby, które umożliwiają ponowne używanie stylów i szablonów. Aby uzyskać więcej informacji na temat zasobów, zobacz Omówienie zasobów XAML.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Przykład

Przykładowy kod przedstawiony w tym omówieniu jest oparty na prostej aplikacji do przeglądania zdjęć pokazanej na poniższej ilustracji.

Ten prosty przykład zdjęć używa stylów i tworzenia szablonów w celu utworzenia wizualnie atrakcyjnego środowiska użytkownika. Przykład zawiera dwa TextBlock elementy i kontrolkę ListBox powiązaną z listą obrazów.

Kompletny przykład można znaleźć w temacie Introduction to Styling and Templating Sample (Wprowadzenie do stylów i tworzenia szablonów).

Style

Można traktować Style jako wygodny sposób stosowania zestawu wartości właściwości do wielu elementów. Można użyć stylu dla dowolnego elementu, który pochodzi od FrameworkElement lub FrameworkContentElement , na przykład Window lub Button.

Najczęstszym sposobem deklarowania stylu jest zasób w `Resources` sekcji w pliku XAML. Ponieważ style są zasobami, przestrzegają tych samych reguł określania zakresu, które mają zastosowanie do wszystkich zasobów. Mówiąc prościej, gdzie deklarujesz styl, ma wpływ na to, gdzie można zastosować styl. Jeśli na przykład zadeklarujesz styl w elemecie głównym pliku XAML definicji aplikacji, styl może być używany w dowolnym miejscu w aplikacji.

Na przykład poniższy kod XAML deklaruje dwa style dla elementu `TextBlock`, jeden automatycznie zastosowany do wszystkich `TextBlock` elementów, a drugi, do którego należy jawnie odwoływać się.

XAML

```
<Window.Resources>
    <!-- .... other resources .... -->

    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>

    <!--A Style that extends the previous TextBlock Style with an x:Key of
TitleText-->
    <Style BasedOn="{StaticResource {x:Type TextBlock}}"
           TargetType="TextBlock"
           x:Key="TitleText">
        <Setter Property="FontSize" Value="26"/>
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0.0" Color="#90DDDD" />
```

```
                        <GradientStop Offset="1.0" Color="#5BFFFF" />
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
```

Oto przykład stylów zadeklarowanych powyżej, które są używane.

XAML

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My
Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

Aby uzyskać więcej informacji, zobacz Tworzenie stylu dla kontrolki.

ControlTemplates

W WPF kontrolka ControlTemplate definiuje wygląd kontrolki. Możesz zmienić strukturę i wygląd kontrolki, definiując nową ControlTemplate i przypisując ją do kontrolki. W wielu przypadkach szablony zapewniają wystarczającą elastyczność, dzięki czemu nie trzeba pisać własnych kontrolek niestandardowych.

Każda kontrolka ma szablon domyślny przypisany do właściwości Control.Template . Szablon łączy wizualną prezentację kontrolki z możliwościami kontrolki. Ponieważ definiujesz szablon w języku XAML, możesz zmienić wygląd kontrolki bez konieczności pisania kodu. Każdy szablon jest przeznaczony dla określonej kontrolki, takiej jak Button.

Często deklarujesz szablon jako zasób w `Resources` sekcji pliku XAML. Podobnie jak w przypadku wszystkich zasobów, mają zastosowanie reguły określania zakresu.

Szablony kontrolek są o wiele bardziej zaangażowane niż styl. Dzieje się tak, ponieważ szablon kontrolki ponownie zapisuje wygląd wizualizacji całej kontrolki, podczas gdy styl po prostu stosuje zmiany właściwości do istniejącej kontrolki. Jednak ponieważ szablon

kontrolki jest stosowany przez ustawienie właściwości Control.Template , można użyć stylu do zdefiniowania lub ustawienia szablonu.

Projektanci zazwyczaj umożliwiają utworzenie kopii istniejącego szablonu i zmodyfikowanie go. Na przykład w projektancie WPF programu Visual Studio wybierz kontrolkę, a następnie kliknij prawym przyciskiem `CheckBox` myszy i wybierz polecenie **Edytuj szablon**>**Utwórz kopię**. To polecenie generuje *styl, który definiuje szablon*.

XAML

```
<Style x:Key="CheckBoxStyle1" TargetType="{x:Type CheckBox}">
    <Setter Property="FocusVisualStyle" Value="{StaticResource
FocusVisual1}"/>
    <Setter Property="Background" Value="{StaticResource
OptionMark.Static.Background1}"/>
    <Setter Property="BorderBrush" Value="{StaticResource
OptionMark.Static.Border1}"/>
    <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}"/>
    <Setter Property="BorderThickness" Value="1"/>
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type CheckBox}">
                <Grid x:Name="templateRoot" Background="Transparent"
SnapsToDevicePixels="True">
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto"/>
                        <ColumnDefinition Width="*"/>
                    </Grid.ColumnDefinitions>
                    <Border x:Name="checkBoxBorder" Background="
{TemplateBinding Background}" BorderThickness="{TemplateBinding
BorderThickness}" BorderBrush="{TemplateBinding BorderBrush}"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
Margin="1" VerticalAlignment="{TemplateBinding VerticalContentAlignment}">
                        <Grid x:Name="markGrid">
                            <Path x:Name="optionMark" Data="F1 M
9.97498,1.22334L 4.6983,9.09834L 4.52164,9.09834L 0,5.19331L
1.27664,3.52165L 4.255,6.08833L 8.33331,1.52588e-005L 9.97498,1.22334 Z "
Fill="{StaticResource OptionMark.Static.Glyph1}" Margin="1" Opacity="0"
Stretch="None"/>
                            <Rectangle x:Name="indeterminateMark" Fill="
{StaticResource OptionMark.Static.Glyph1}" Margin="2" Opacity="0"/>
                        </Grid>
                    </Border>
                    <ContentPresenter x:Name="contentPresenter"
Grid.Column="1" Focusable="False" HorizontalAlignment="{TemplateBinding
HorizontalContentAlignment}" Margin="{TemplateBinding Padding}"
RecognizesAccessKey="True" SnapsToDevicePixels="{TemplateBinding
SnapsToDevicePixels}" VerticalAlignment="{TemplateBinding
VerticalContentAlignment}"/>
                </Grid>
                <ControlTemplate.Triggers>
```

```
                    <Trigger Property="HasContent" Value="true">
                        <Setter Property="FocusVisualStyle" Value="
{StaticResource OptionMarkFocusVisual1}"/>
                        <Setter Property="Padding" Value="4,-1,0,0"/>

... content removed to save space ...
```

Edytowanie kopii szablonu to doskonały sposób na poznanie sposobu działania szablonów. Zamiast tworzyć nowy pusty szablon, łatwiej jest edytować kopię i zmienić kilka aspektów prezentacji wizualnej.

Aby zapoznać się z przykładem, zobacz Tworzenie szablonu dla kontrolki.

Powiązanie szablonu

Być może zauważono, że zasób szablonu zdefiniowany w poprzedniej sekcji używa rozszerzenia znaczników TemplateBinding. A `TemplateBinding` to zoptymalizowana forma powiązania dla scenariuszy szablonów, analogicznie do powiązania skonstruowanego za pomocą `{Binding RelativeSource={RelativeSource TemplatedParent}}` elementu . `TemplateBinding` Jest przydatny w przypadku wiązania części szablonu z właściwościami kontrolki. Na przykład każda kontrolka BorderThickness ma właściwość . Użyj elementu , `TemplateBinding` aby zarządzać tym elementem w szablonie, którego dotyczy to ustawienie kontrolki.

ContentControl i ItemsControl

ContentPresenter Jeśli element jest zadeklarowany w obiekcie ContentControlControlTemplate , ContentPresenter element automatycznie powiąże się z właściwościami ContentTemplate i Content . Podobnie element ItemsPresenter , który znajduje się w ControlTemplate obiekcie ItemsControl , automatycznie powiąże się z właściwościami ItemTemplate i Items .

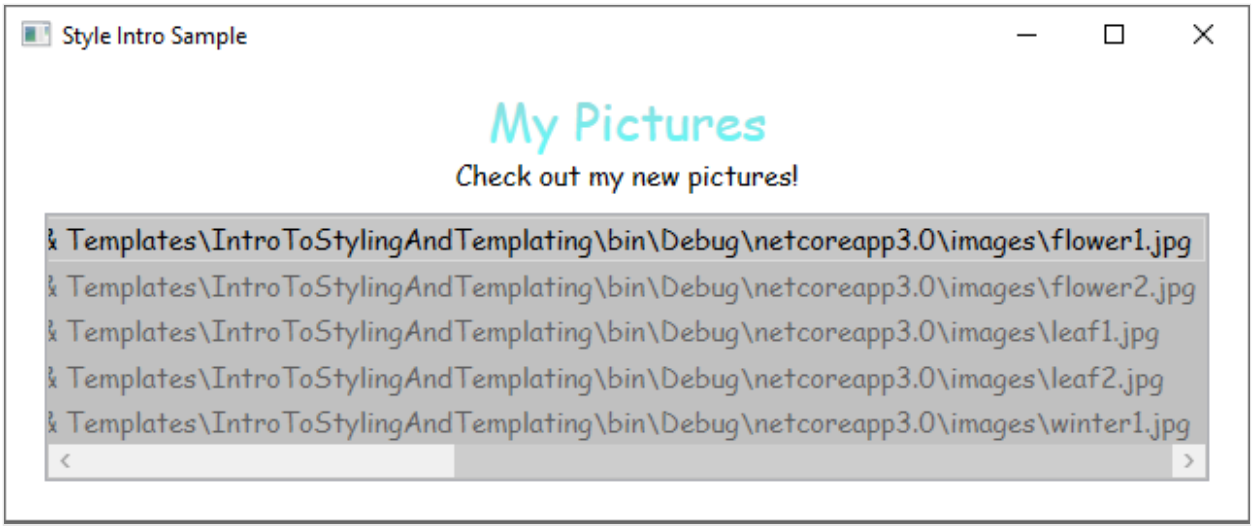
DataTemplates

W tej przykładowej aplikacji istnieje kontrolka ListBox powiązana z listą zdjęć.

XAML

```
<ListBox ItemsSource="{Binding Source={StaticResource MyPhotos}}"
         Background="Silver" Width="600" Margin="10" SelectedIndex="0"/>
```

Obecnie wygląda to ListBox następująco.



Większość kontrolek ma jakiś typ zawartości, a zawartość często pochodzi z danych, z którymi się wiążesz. W tym przykładzie dane są listą zdjęć. W WPF należy użyć elementu DataTemplate , aby zdefiniować wizualną reprezentację danych. Zasadniczo to, co należy umieścić w obiekcie DataTemplate , określa, jak wyglądają dane w renderowanej aplikacji.

W naszej przykładowej aplikacji każdy obiekt niestandardowy `Photo` ma `Source` właściwość typu string, która określa ścieżkę pliku obrazu. Obecnie obiekty zdjęć są wyświetlane jako ścieżki plików.

C#

```
public class Photo
{
    public Photo(string path)
    {
        Source = path;
    }

    public string Source { get; }

    public override string ToString() => Source;
}
```

Aby zdjęcia wyświetlane jako obrazy, należy utworzyć DataTemplate jako zasób.

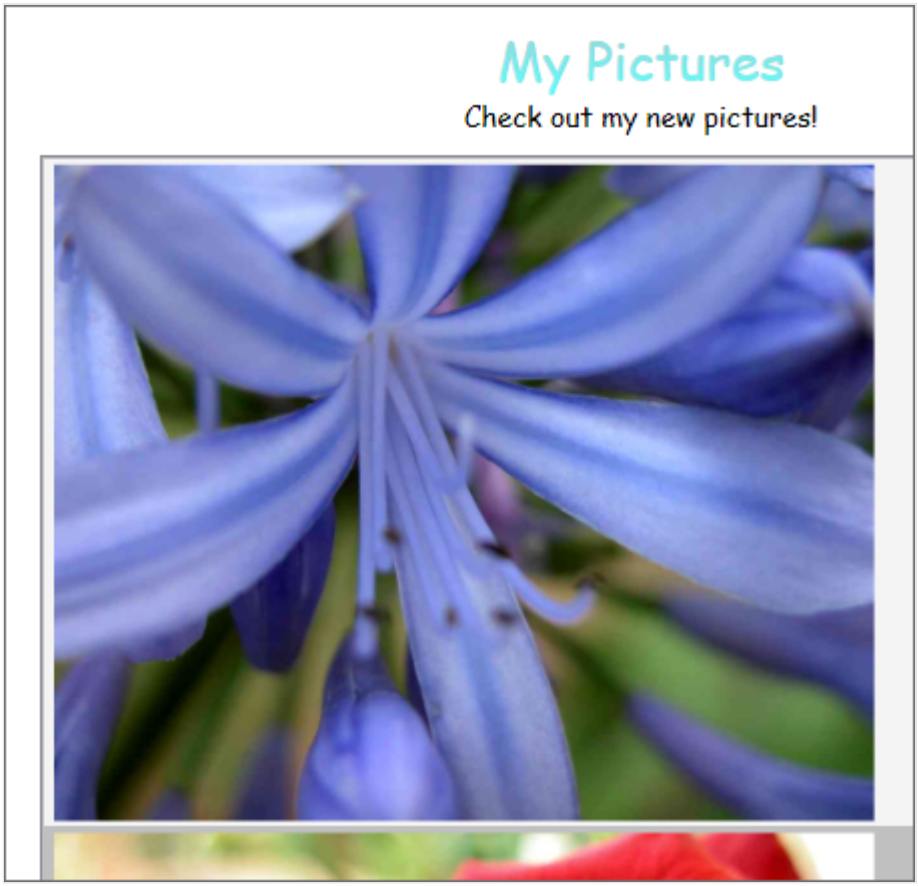
XAML

```
<Window.Resources>
    <!-- .... other resources .... -->

    <!--DataTemplate to display Photos as images
    instead of text strings of Paths-->
    <DataTemplate DataType="{x:Type local:Photo}">
        <Border Margin="3">
```

```
            <Image Source="{Binding Source}"/>
        </Border>
    </DataTemplate>
</Window.Resources>
```

Zwróć uwagę, że DataType właściwość jest podobna do TargetType właściwości Style. Jeśli element DataTemplate znajduje się w sekcji zasobów, podczas określania DataType właściwości typu i pomijania `x:Key` elementu , DataTemplate jest stosowany za każdym razem, gdy pojawi się ten typ. Zawsze masz możliwość przypisania obiektu DataTemplate za pomocą elementu , `x:Key` a następnie ustawiania go jako `StaticResource` właściwości, które przyjmują DataTemplate typy, takie jak ItemTemplate właściwość lub ContentTemplate właściwość.

Zasadniczo w DataTemplate powyższym przykładzie zdefiniowano, że za każdym razem, gdy istnieje `Photo` obiekt, powinien on być wyświetlany jako element Image w obiekcie Border. Dzięki temu DataTemplatenasza aplikacja wygląda teraz następująco.



Model tworzenia szablonów danych udostępnia inne funkcje. Jeśli na przykład wyświetlasz dane kolekcji zawierające inne kolekcje przy użyciu HeaderedItemsControl typu, takiego jak Menu lub TreeView, istnieje .HierarchicalDataTemplate Inną funkcją tworzenia szablonów danych jest DataTemplateSelectorelement , który umożliwia wybranie elementu DataTemplate do użycia na podstawie logiki niestandardowej. Aby uzyskać więcej informacji, zobacz Data Templating Overview (Omówienie tworzenia

szablonów danych), który zawiera bardziej szczegółowe omówienie różnych funkcji tworzenia szablonów danych.

Wyzwalacze

Wyzwalacz ustawia właściwości lub uruchamia akcje, takie jak animacja, gdy wartość właściwości zmienia się lub gdy jest zgłaszane zdarzenie. Style, ControlTemplatei DataTemplate wszystkie mają `Triggers` właściwość , która może zawierać zestaw wyzwalaczy. Istnieje kilka typów wyzwalaczy.

WłaściwościTriggers

Obiekt Trigger , który ustawia wartości właściwości lub uruchamia akcje na podstawie wartości właściwości, jest nazywany wyzwalaczem właściwości.

Aby zademonstrować sposób używania wyzwalaczy właściwości, można ustawić każdy ListBoxItem częściowo przezroczysty, chyba że zostanie wybrany. Poniższy styl ustawia Opacity wartość typu ListBoxItem na `0.5`wartość . IsSelected Gdy właściwość ma `true`wartość , właściwość jest jednak ustawiona Opacity na `1.0`wartość .

XAML

```
<Window.Resources>
    <!-- .... other resources .... -->

    <Style TargetType="ListBoxItem">
        <Setter Property="Opacity" Value="0.5" />
        <Setter Property="MaxHeight" Value="75" />
        <Style.Triggers>
            <Trigger Property="IsSelected" Value="True">
                <Trigger.Setters>
                    <Setter Property="Opacity" Value="1.0" />
                </Trigger.Setters>
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
```

W tym przykładzie użyto elementu , Trigger aby ustawić wartość właściwości, ale należy pamiętać, że Trigger klasa ma EnterActions również właściwości i ExitActions , które umożliwiają wyzwalaczowi wykonywanie akcji.

Zwróć uwagę, że MaxHeight właściwość obiektu ListBoxItem jest ustawiona na `75`wartość . Na poniższej ilustracji trzeci element jest wybranym elementem.

EventTriggers i Storyboards

Innym typem wyzwalacza jest EventTriggerelement , który uruchamia zestaw akcji na podstawie wystąpienia zdarzenia. Na przykład następujące EventTrigger obiekty określają, że po wprowadzeniu MaxHeightListBoxItemwskaźnika myszy właściwość animuje wartość `90` w ciągu drugiego `0.2` okresu. Gdy mysz odchodzi od elementu, właściwość powraca do oryginalnej wartości w okresie sekundy `1` . Zwróć uwagę, że nie trzeba określać To wartości animacji MouseLeave . Wynika to z tego, że animacja może śledzić oryginalną wartość.

XAML

```
<Style.Triggers>
    <Trigger Property="IsSelected" Value="True">
        <Trigger.Setters>
            <Setter Property="Opacity" Value="1.0" />
        </Trigger.Setters>
    </Trigger>
    <EventTrigger RoutedEvent="Mouse.MouseEnter">
        <EventTrigger.Actions>
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Duration="0:0:0.2"
                        Storyboard.TargetProperty="MaxHeight"
                        To="90"  />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Mouse.MouseLeave">
        <EventTrigger.Actions>
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Duration="0:0:1"
                        Storyboard.TargetProperty="MaxHeight"  />
                </Storyboard>
            </BeginStoryboard>
```

```
        </EventTrigger.Actions>
    </EventTrigger>
</Style.Triggers>
```

Aby uzyskać więcej informacji, zobacz Omówienie scenorysów.

Na poniższej ilustracji mysz wskazuje trzeci element.

MultiTriggers, DataTriggers i MultiDataTriggers

Oprócz Trigger funkcji i EventTriggeristnieją inne typy wyzwalaczy. MultiTrigger umożliwia ustawianie wartości właściwości na podstawie wielu warunków. Używasz DataTrigger właściwości i MultiDataTrigger gdy właściwość warunku jest powiązana z danymi.

Stany wizualne

Kontrolki są zawsze w określonym **stanie**. Na przykład gdy mysz przesuwa się nad powierzchnią kontrolki, kontrolka jest uważana za w typowym stanie `MouseOver`. Kontrolka bez określonego stanu jest uważana za stan wspólny `Normal` . Stany są podzielone na grupy, a wymienione wcześniej stany są częścią grupy `CommonStates`stanów . Większość kontrolek ma dwie grupy stanów: `CommonStates` i `FocusStates`. Dla każdej grupy stanów stosowanej do kontrolki kontrolka jest zawsze w jednym stanie każdej grupy, na przykład `CommonStates.MouseOver` i `FocusStates.Unfocused`. Jednak kontrolka nie może znajdować się w dwóch różnych stanach w tej samej grupie, takich jak `CommonStates.Normal` i `CommonStates.Disabled`. Oto tabela stanów, w których większość kontrolek rozpoznaje i używa.

Nazwa wizualizacji	Nazwa grupy VisualStateGroup	Opis
Normalne	CommonStates	Domyślny stan.

Nazwa wizualizacji	Nazwa grupy VisualStateGroup	Opis
Mouseover	CommonStates	Wskaźnik myszy jest umieszczony nad kontrolką.
Naciśnięte	CommonStates	Kontrolka jest naciśnięta.
Disabled	CommonStates	Kontrolka jest wyłączona.
Ustawiono fokus	FocusStates	Kontrolka ma fokus.
Unfocused	FocusStates	Kontrolka nie ma fokusu.

Definiując element System.Windows.VisualStateManager główny szablonu kontrolki, można wyzwalać animacje po wprowadzeniu określonego stanu kontrolki. Deklaruje `VisualStateManager` , które kombinacje VisualStateGroup i VisualState które mają być oglądane. Po wprowadzeniu stanu obserwowanego kontrolka zostanie uruchomiona animacja zdefiniowana przez element `VisualStateManager` .

Na przykład poniższy kod XAML obserwuje `CommonStates.MouseOver` stan animowania koloru wypełnienia elementu o nazwie `backgroundElement`. Gdy kontrolka powróci do `CommonStates.Normal` stanu, kolor wypełnienia elementu o nazwie `backgroundElement` zostanie przywrócony.

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="CommonStates">
                <VisualState Name="Normal">
                    <ColorAnimation
Storyboard.TargetName="backgroundElement"
                                    Storyboard.TargetProperty="(Shape.Fill).
(SolidColorBrush.Color)"
                                    To="{TemplateBinding Background}"
                                    Duration="0:0:0.3"/>
                </VisualState>
                <VisualState Name="MouseOver">
                    <ColorAnimation
Storyboard.TargetName="backgroundElement"
                                    Storyboard.TargetProperty="(Shape.Fill).
(SolidColorBrush.Color)"
                                    To="Yellow"
                                    Duration="0:0:0.3"/>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
```

```
        ...
```

Aby uzyskać więcej informacji na temat scenorysów, zobacz Storyboards Overview (Omówienie scenorysów).

Udostępnione zasoby i motywy

Typowa aplikacja WPF może mieć wiele zasobów interfejsu użytkownika, które są stosowane w całej aplikacji. Zbiorczo ten zestaw zasobów można uznać za motyw aplikacji. WPF zapewnia obsługę tworzenia pakietów zasobów interfejsu użytkownika jako motywu przy użyciu słownika zasobów, który jest hermetyzowany jako ResourceDictionary klasa.

Motywy WPF są definiowane przy użyciu mechanizmu stylów i tworzenia szablonów, który uwidacznia WPF na potrzeby dostosowywania wizualizacji dowolnego elementu.

Zasoby motywu WPF są przechowywane w słownikach zasobów osadzonych. Te słowniki zasobów muszą być osadzone w podpisanym zestawie i mogą być osadzone w tym samym zestawie co sam kod lub w zestawie równoległym. W przypadku PresentationFramework.dll zestaw, który zawiera kontrolki WPF, zasoby motywu znajdują się w serii zestawów równoległych.

Motyw staje się ostatnim miejscem do wyszukania podczas wyszukiwania stylu elementu. Zazwyczaj wyszukiwanie rozpocznie się od przejścia drzewa elementów w poszukiwaniu odpowiedniego zasobu, a następnie wyszukania w kolekcji zasobów aplikacji i na koniec wykonywania zapytań względem systemu. Dzięki temu deweloperzy aplikacji mogą ponownie zdefiniować styl dla dowolnego obiektu na poziomie drzewa lub aplikacji przed dotarciem do motywu.

Słowniki zasobów można zdefiniować jako poszczególne pliki, które umożliwiają ponowne użycie motywu w wielu aplikacjach. Motywy z możliwością zamiany można również tworzyć, definiując wiele słowników zasobów, które udostępniają te same typy zasobów, ale z różnymi wartościami. Przedefiniowanie tych stylów lub innych zasobów na poziomie aplikacji jest zalecanym podejściem do osłaniania aplikacji.

Aby udostępnić zestaw zasobów, w tym style i szablony, w aplikacjach, możesz utworzyć plik XAML i zdefiniować ResourceDictionary odwołanie do `shared.xaml` pliku.

XAML

```
<ResourceDictionary.MergedDictionaries>
  <ResourceDictionary Source="Shared.xaml" />
```

```
</ResourceDictionary.MergedDictionaries>
```

Jest to udostępnianie `shared.xaml` elementu , który definiuje ResourceDictionary element zawierający zestaw zasobów stylu i pędzla, który umożliwia kontrolkom w aplikacji spójny wygląd.

Aby uzyskać więcej informacji, zobacz Scalone słowniki zasobów.

Jeśli tworzysz motyw dla kontrolki niestandardowej, zobacz sekcję **Definiowanie zasobów na poziomie motywu** w przeglądzie Tworzenie kontrolek.

Zobacz też

- Pakuj URI w WPF
- Instrukcje: znajdowanie elementów generowanych przez element ControlTemplate
- Znajdowanie elementów DataTemplate-Generated

Jak utworzyć styl kontrolki (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 5 min

Za pomocą Windows Presentation Foundation (WPF) można dostosować wygląd istniejącej kontrolki przy użyciu własnego stylu wielokrotnego użytku. Style można stosować globalnie do aplikacji, okien i stron lub bezpośrednio do kontrolek.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Tworzenie stylu

Można traktować Style jako wygodny sposób stosowania zestawu wartości właściwości do jednego lub kilku elementów. Styl można użyć w dowolnym elemecie pochodzącym z FrameworkElement elementu lub FrameworkContentElement na przykład .WindowButton

Najbardziej typowym sposobem deklarowania stylu jest zasób w sekcji w `Resources` pliku XAML. Ponieważ style są zasobami, przestrzegają tych samych reguł określania zakresu, które mają zastosowanie do wszystkich zasobów. Mówiąc prościej, w przypadku deklarowania stylu wpływa na to, gdzie można zastosować styl. Jeśli na przykład zadeklarujesz styl w głównym elemecie pliku XAML definicji aplikacji, styl może być używany w dowolnym miejscu w aplikacji.

XAML

```
<Application x:Class="IntroToStylingAndTemplating.App"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:IntroToStylingAndTemplating"
             StartupUri="WindowExplicitStyle.xaml">
    <Application.Resources>
        <ResourceDictionary>

            <Style x:Key="Header1" TargetType="TextBlock">
                <Setter Property="FontSize" Value="15" />
                <Setter Property="FontWeight" Value="ExtraBold" />
            </Style>

        </ResourceDictionary>
```

```
    </Application.Resources>
</Application>
```

Jeśli zadeklarujesz styl w jednym z plików XAML aplikacji, styl może być używany tylko w tym pliku XAML. Aby uzyskać więcej informacji na temat reguł określania zakresu dla zasobów, zobacz Omówienie zasobów XAML.

XAML

```
<Window x:Class="IntroToStylingAndTemplating.WindowSingleResource"
        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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="WindowSingleResource" Height="450" Width="800">
    <Window.Resources>

        <Style x:Key="Header1" TargetType="TextBlock">
            <Setter Property="FontSize" Value="15" />
            <Setter Property="FontWeight" Value="ExtraBold" />
        </Style>

    </Window.Resources>
    <Grid />
</Window>
```

Styl składa się z elementów podrzędnych `<Setter>` , które ustawiają właściwości elementów, do których jest stosowany styl. W powyższym przykładzie zwróć uwagę, że styl ma być stosowany do `TextBlock` typów za pomocą atrybutu `TargetType` . Styl ustawi wartość FontSize`15` na i wartość FontWeight`ExtraBold`. Dodaj dla `<Setter>` każdej właściwości zmiany stylu.

Stosowanie stylu niejawnie

A Style to wygodny sposób stosowania zestawu wartości właściwości do więcej niż jednego elementu. Rozważmy na przykład następujące TextBlock elementy i ich domyślny wygląd w oknie.

XAML

```
<StackPanel>
    <TextBlock>My Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

Wygląd domyślny można zmienić, ustawiając właściwości, takie jak FontSize i FontFamily, bezpośrednio dla każdego TextBlock elementu. Jeśli jednak chcesz TextBlock , aby elementy współużytkować niektóre właściwości, możesz utworzyć obiekt Style w `Resources` sekcji pliku XAML, jak pokazano tutaj.

XAML

```
<Window.Resources>
    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>
</Window.Resources>
```

Po ustawieniu TargetType stylu na TextBlock typ i pominięcia `x:Key` atrybutu styl jest stosowany do wszystkich TextBlock elementów o określonym zakresie w stylu, który jest zazwyczaj samym plikiem XAML.

TextBlock Teraz elementy są wyświetlane w następujący sposób.

Jawne stosowanie stylu

Jeśli dodasz `x:Key` atrybut z wartością do stylu, styl nie jest już niejawnie stosowany do wszystkich elementów .TargetType Tylko elementy, które jawnie odwołują się do stylu, będą miały do nich zastosowany styl.

Oto styl z poprzedniej sekcji, ale zadeklarowany za pomocą atrybutu `x:Key` .

XAML

```
<Window.Resources>
    <Style x:Key="TitleText" TargetType="TextBlock">
```

```
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>
</Window.Resources>
```

Aby zastosować styl, ustaw Style właściwość elementu na `x:Key` wartość przy użyciu rozszerzenia znaczników StaticResource, jak pokazano tutaj.

XAML

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}">My Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

Zwróć uwagę, że pierwszy TextBlock element ma zastosowany styl, podczas gdy drugi element TextBlock pozostaje niezmieniony. Niejawny styl z poprzedniej sekcji został zmieniony na styl, który zadeklarował `x:Key` atrybut, co oznacza, że jedynym elementem, którego dotyczy styl, jest ten, który odwołuje się bezpośrednio do stylu.

Po zastosowaniu stylu jawnie lub niejawnie staje się zapieczętowany i nie można go zmienić. Jeśli chcesz zmienić styl, który został zastosowany, utwórz nowy styl, aby zastąpić istniejący. Aby uzyskać więcej informacji, zobacz IsSealed właściwość .

Możesz utworzyć obiekt, który wybiera styl do zastosowania na podstawie logiki niestandardowej. Aby zapoznać się z przykładem, zobacz przykład podany dla StyleSelector klasy.

Programowe stosowanie stylu

Aby przypisać nazwany styl do elementu programowo, pobierz styl z kolekcji zasobów i przypisz go do właściwości elementu Style . Elementy w kolekcji zasobów mają typ Object. W związku z tym należy rzutować pobrany styl na obiekt System.Windows.Style przed przypisaniem go do `Style` właściwości. Na przykład poniższy kod ustawia styl `TextBlock` nazwanego `textblock1` na zdefiniowany styl `TitleText`.

C#

```
textblock1.Style = (Style)Resources["TitleText"];
```

Rozszerzanie stylu

Być może chcesz, aby dwa TextBlock elementy współdzieliły niektóre wartości właściwości, takie jak FontFamily i wyśrodkowane HorizontalAlignment. Ale chcesz również, aby tekst **Moje obrazy** miał kilka dodatkowych właściwości. Możesz to zrobić, tworząc nowy styl oparty na pierwszym stylu, jak pokazano tutaj.

XAML

```
<Window.Resources>
    <!-- .... other resources .... -->

    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>

    <!--A Style that extends the previous TextBlock Style with an x:Key of
TitleText-->
    <Style BasedOn="{StaticResource {x:Type TextBlock}}"
           TargetType="TextBlock"
           x:Key="TitleText">
        <Setter Property="FontSize" Value="26"/>
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0.0" Color="#90DDDD" />
                        <GradientStop Offset="1.0" Color="#5BFFFF" />
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Window.Resources>
```

XAML

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My
Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

Ten `TextBlock` styl jest teraz wyśrodkowany, używa `Comic Sans MS` czcionki o rozmiarze `26`, a kolor pierwszego planu ustawiony na pokazany LinearGradientBrush w przykładzie. Zwróć uwagę, że zastępuje FontSize wartość stylu podstawowego. Jeśli istnieje więcej niż jedna Setter właściwość wskazująca tę samą właściwość w elemencie Style, `Setter` która jest zadeklarowana ostatnio ma pierwszeństwo.

Poniżej przedstawiono, jak TextBlock wyglądają teraz elementy:

Ten `TitleText` styl rozszerza styl utworzony dla TextBlock typu, do którego odwołuje się `BasedOn="{StaticResource {x:Type TextBlock}}"` element . Możesz również rozszerzyć styl, który ma `x:Key` element przy użyciu `x:Key` stylu. Jeśli na przykład istnieje styl o nazwie `Header1` i chcesz rozszerzyć ten styl, użyj polecenia `BasedOn="{StaticResource Header1}"`.

Relacja właściwości TargetType i atrybutu x:Key

Jak pokazano wcześniej, ustawienie TargetType właściwości na bez `TextBlock` przypisywania stylu `x:Key` powoduje zastosowanie stylu do wszystkich TextBlock elementów. W tym przypadku `x:Key` parametr jest niejawnie ustawiony na `{x:Type TextBlock}` wartość . Oznacza to, że jeśli jawnie ustawisz `x:Key` wartość inną niż `{x:Type TextBlock}`, Style parametr nie zostanie automatycznie zastosowany do wszystkich `TextBlock` elementów. Zamiast tego należy zastosować styl (używając `x:Key` wartości) do jawnie `TextBlock` elementów. Jeśli styl znajduje się w sekcji zasobów i nie ustawisz `TargetType` właściwości w swoim stylu, musisz ustawić `x:Key` atrybut.

Oprócz podania wartości domyślnej `x:Key` dla właściwości właściwość `TargetType` określa typ, do którego mają zastosowanie właściwości setter. Jeśli nie określisz `TargetType` elementu , musisz zakwalifikować właściwości w Setter obiektach z nazwą klasy przy użyciu składni `Property="ClassName.Property"`. Na przykład zamiast ustawiać `Property="FontSize"` wartość , musisz ustawić Property wartość `"TextBlock.FontSize"` lub `"Control.FontSize"`.

Należy również zauważyć, że wiele kontrolek WPF składa się z kombinacji innych kontrolek WPF. Jeśli tworzysz styl, który ma zastosowanie do wszystkich kontrolek typu, może zostać wyświetlony nieoczekiwany wynik. Jeśli na przykład utworzysz styl

przeznaczony TextBlock dla typu w Windowobiekcie , styl zostanie zastosowany do wszystkich `TextBlock` kontrolek w oknie, nawet jeśli `TextBlock` element jest częścią innej kontrolki, takiej jak ListBox.

Zobacz też

- Jak utworzyć szablon dla kontrolki
- Omówienie zasobów XAML
- Przegląd XAML

Jak utworzyć szablon dla kontrolki (WPF.NET)

Artykuł • 18.02.2023 • Czas czytania: 6 min

Za pomocą Windows Presentation Foundation (WPF) można dostosować strukturę wizualizacji i zachowanie istniejącej kontrolki przy użyciu własnego szablonu wielokrotnego użytku. Szablony można stosować globalnie do aplikacji, okien i stron lub bezpośrednio do kontrolek. Większość scenariuszy, które wymagają utworzenia nowej kontrolki, może być objęta zamiast tego tworzenie nowego szablonu dla istniejącej kontrolki.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

W tym artykule zapoznasz się z tworzeniem nowej ControlTemplate kontrolki Button .

Kiedy utworzyć kontrolkęTemplate

Kontrolki mają wiele właściwości, takich jak Background, Foregroundi FontFamily. Te właściwości kontrolują różne aspekty wyglądu kontrolki, ale zmiany, które można wprowadzić, ustawiając te właściwości, są ograniczone. Można na przykład ustawić Foreground właściwość na niebieską i FontStyle kursywę na .CheckBox Jeśli chcesz dostosować wygląd kontrolki poza ustawieniem innych właściwości kontrolki, możesz utworzyć element ControlTemplate.

W większości interfejsów użytkownika przycisk ma ten sam ogólny wygląd: prostokąt z tekstem. Jeśli chcesz utworzyć przycisk zaokrąglony, możesz utworzyć nową kontrolkę dziedziczą po przycisku lub odtworzyć funkcjonalność przycisku. Ponadto nowa kontrolka użytkownika zapewni wizualizację cykliczną.

Możesz uniknąć tworzenia nowych kontrolek, dostosowując układ wizualny istniejącej kontrolki. Za pomocą zaokrąglonego przycisku utworzysz ControlTemplate obiekt z żądanym układem wizualnym.

Z drugiej strony, jeśli potrzebujesz kontrolki z nowymi funkcjami, różnymi właściwościami i nowymi ustawieniami, utwórz nowy UserControlelement .

Wymagania wstępne

Utwórz nową aplikację WPF i w *pliku MainWindow.xaml* (lub innym wybranym oknie) ustaw następujące właściwości w **<elemecie Window>** :

Właściwość	**Wartość**
Title	`Template Intro Sample`
SizeToContent	`WidthAndHeight`
MinWidth	`250`

Ustaw zawartość **<elementu Window>** na następujący kod XAML:

XAML

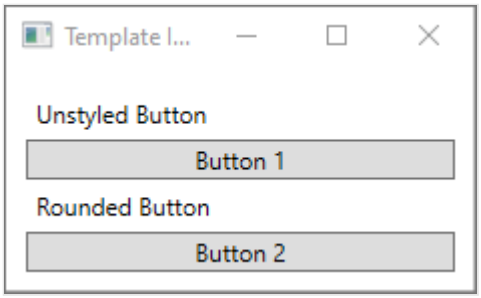
```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button>Button 2</Button>
</StackPanel>
```

W końcu plik *MainWindow.xaml* powinien wyglądać podobnie do następującego:

XAML

```
<Window x:Class="IntroToStylingAndTemplating.Window1"
        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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="Template Intro Sample" SizeToContent="WidthAndHeight"
MinWidth="250">
    <StackPanel Margin="10">
        <Label>Unstyled Button</Label>
        <Button>Button 1</Button>
        <Label>Rounded Button</Label>
        <Button>Button 2</Button>
    </StackPanel>
</Window>
```

Jeśli uruchomisz aplikację, wygląda następująco:



Tworzenie kontrolkiTemplate

Najczęstszym sposobem deklarowania elementu ControlTemplate jest zasób w sekcji w `Resources` pliku XAML. Ponieważ szablony są zasobami, są zgodne z tymi samymi regułami określania zakresu, które mają zastosowanie do wszystkich zasobów. Po prostu deklarowanie szablonu ma wpływ na miejsce zastosowania szablonu. Jeśli na przykład zadeklarujesz szablon w elemecie głównym pliku XAML definicji aplikacji, szablon może być używany w dowolnym miejscu w aplikacji. Jeśli zdefiniujesz szablon w oknie, tylko kontrolki w tym oknie mogą używać szablonu.

Aby rozpocząć od, dodaj `Window.Resources` element do pliku *MainWindow.xaml* :

XAML

```
<Window x:Class="IntroToStylingAndTemplating.Window2"
        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:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="Template Intro Sample" SizeToContent="WidthAndHeight"
MinWidth="250">
    <Window.Resources>

    </Window.Resources>
    <StackPanel Margin="10">
        <Label>Unstyled Button</Label>
        <Button>Button 1</Button>
        <Label>Rounded Button</Label>
        <Button>Button 2</Button>
    </StackPanel>
</Window>
```

Utwórz nowy **<element ControlTemplate>** z następującymi właściwościami:

Właściwość	Wartość

Właściwość	Wartość
x:Key	`roundbutton`
TargetType	`Button`

Ten szablon kontrolki będzie prosty:

- element główny kontrolki, a Grid
- , Ellipse aby narysować zaokrąglony wygląd przycisku
- a ContentPresenter , aby wyświetlić zawartość przycisku określonego przez użytkownika

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse Fill="{TemplateBinding Background}" Stroke="
{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

SzablonBinding

Podczas tworzenia nowego ControlTemplateelementu nadal możesz chcieć użyć właściwości publicznych, aby zmienić wygląd kontrolki. Rozszerzenie znaczników TemplateBinding wiąże właściwość elementu, który znajduje się w ControlTemplate właściwości publicznej zdefiniowanej przez kontrolkę. W przypadku korzystania z szablonu TemplateBinding można włączyć właściwości kontrolki, aby działały jako parametry szablonu. Oznacza to, że gdy właściwość kontrolki jest ustawiona, ta wartość jest przekazywana do elementu, który ma na nim właściwość TemplateBinding .

Elipsa

Zwróć uwagę, że **Fill** właściwości **<i elementu elipsy>** są powiązane z właściwościami i Background kontrolkiForeground.**Stroke**

Contentpresenter

Element <ContentPresenter> jest również dodawany do szablonu. Ponieważ ten szablon jest przeznaczony dla przycisku, należy wziąć pod uwagę, że przycisk dziedziczy z

ContentControlelementu . Przycisk przedstawia zawartość elementu. Możesz ustawić dowolny element wewnątrz przycisku, na przykład zwykły tekst, a nawet inną kontrolkę. Oba poniższe przyciski są prawidłowe:

XAML

```
<Button>My Text</Button>

<!-- and -->

<Button>
    <CheckBox>Checkbox in a button</CheckBox>
</Button>
```

W obu poprzednich przykładach tekst i pole wyboru są ustawione jako właściwość Button.Content . Niezależnie od tego, co jest ustawione jako zawartość, można przedstawić za pomocą **<elementu ContentPresenter>**, co robi szablon.

Jeśli element ControlTemplate jest stosowany do ContentControl typu, takiego jak `Button`, ContentPresenter element jest wyszukiwany w drzewie elementów. Jeśli element `ContentPresenter` zostanie znaleziony, szablon automatycznie powiąże właściwość kontrolki Content z właściwością `ContentPresenter`.

Korzystanie z szablonu

Znajdź przyciski zadeklarowane na początku tego artykułu.

XAML

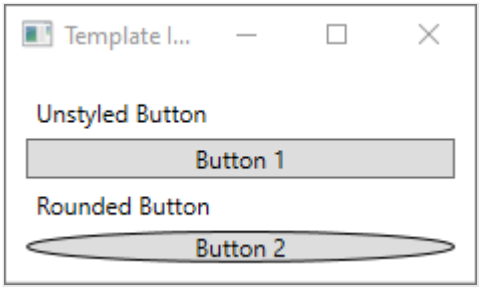
```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button>Button 2</Button>
</StackPanel>
```

Ustaw właściwość drugiego `roundbutton` przycisku Template na zasób:

XAML

```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}">Button 2</Button>
</StackPanel>
```

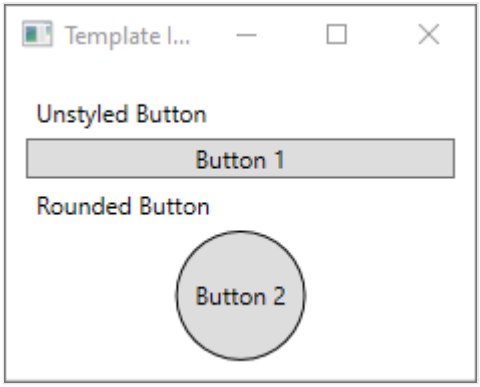
Jeśli uruchomisz projekt i przyjrzysz się wynikowi, zobaczysz, że przycisk ma zaokrąglone tło.



Być może zauważyłeś, że przycisk nie jest okręgiem, ale jest niesymetryczny. Ze względu na sposób **<działania elementu elipsy>** zawsze rozszerza się, aby wypełnić dostępne miejsce. Ustaw okrąg jako jednolity, zmieniając **width** właściwości przycisku i **height** na tę samą wartość:

XAML

```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}" Width="65"
Height="65">Button 2</Button>
</StackPanel>
```



Dodawanie wyzwalacza

Mimo że przycisk z zastosowanym szablonem wygląda inaczej, zachowuje się tak samo jak każdy inny przycisk. Po naciśnięciu przycisku Click zdarzenie zostanie wyzwolony. Jednak być może zauważysz, że po przeniesieniu myszy nad przyciskiem wizualizacje przycisku nie zmieniają się. Te interakcje wizualne są definiowane przez szablon.

Dzięki dynamicznym systemom zdarzeń i właściwości, które zapewnia WPF, można obserwować określoną właściwość dla wartości, a następnie w razie potrzeby ponownie

stylować szablon. W tym przykładzie zobaczysz właściwość przycisku IsMouseOver . Gdy mysz jest nad kontrolką, styl **<Ellipse>** z nowym kolorem. Ten typ wyzwalacza jest znany jako *WłaściwośćTrigger*.

Aby można było to zrobić, musisz dodać nazwę do **wielokropka>, do<** którego możesz się odwoływać. Nadaj mu nazwę **backgroundElement**.

XAML

```
<Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}"
Stroke="{TemplateBinding Foreground}" />
```

Następnie dodaj nowy Trigger element do kolekcji ControlTemplate.Triggers . Wyzwalacz będzie obserwować `IsMouseOver` zdarzenie dla wartości `true`.

XAML

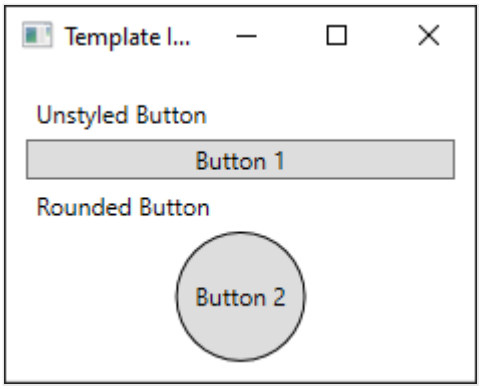
```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding
Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="IsMouseOver" Value="true">

        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

Następnie dodaj **<element Setter>** do **<wyzwalacza>**, który zmienia właściwość **<Fill wielokropka>** na nowy kolor.

XAML

```
<Trigger Property="IsMouseOver" Value="true">
    <Setter Property="Fill" TargetName="backgroundElement"
Value="AliceBlue"/>
</Trigger>
```

Uruchamianie projektu. Zwróć uwagę, że po przeniesieniu myszy nad przyciskiem kolor **<wielokropka>** zmienia się.



Korzystanie z elementu VisualState

Stany wizualne są definiowane i wyzwalane przez kontrolkę. Na przykład po przeniesieniu myszy na wierzchu kontrolki `CommonStates.MouseOver` stan jest wyzwalany. Zmiany właściwości można animować na podstawie bieżącego stanu kontrolki. W poprzedniej sekcji **<właściwośćTrigger>** została użyta do zmiany pierwszego planu przycisku na `AliceBlue` wartość , gdy `IsMouseOver` właściwość miała wartość `true`. Zamiast tego utwórz stan wizualny, który animuje zmianę tego koloru, zapewniając płynne przejście. Aby uzyskać więcej informacji na temat *visualStates*, zobacz Style i szablony w WPF.

Aby przekonwertować **element< PropertyTrigger>** na animowany stan wizualizacji, najpierw usuń **<element ControlTemplate.Triggers>** z szablonu.

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding
Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

Następnie w **<katalogu głównym siatki>** szablonu kontrolki dodaj **<element VisualStateManager.VisualStateGroups>** z **elementem< VisualStateGroup>** for `CommonStates`. Zdefiniuj dwa stany `Normal` i `MouseOver`.

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="CommonStates">
                <VisualState Name="Normal">
```

```
                </VisualState>
                <VisualState Name="MouseOver">
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding
Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

Wszystkie animacje zdefiniowane w **elemecie VisualState> są stosowane po wyzwoleniu**< tego stanu. Utwórz animacje dla każdego stanu. Animacje są umieszczane wewnątrz elementu **scenorysu**<>. Aby uzyskać więcej informacji na temat scenorysów, zobacz Storyboards Overview (Omówienie scenorysów).

- Normalne

  Ten stan animuje wypełnienie wielokropka, przywracając go do koloru kontrolki `Background` .

  XAML

  ```
  <Storyboard>
      <ColorAnimation Storyboard.TargetName="backgroundElement"
          Storyboard.TargetProperty="(Shape.Fill).
  (SolidColorBrush.Color)"
          To="{TemplateBinding Background}"
          Duration="0:0:0.3"/>
  </Storyboard>
  ```

- Mouseover

  Ten stan animuje kolor wielokropka `Background` na nowy kolor: `Yellow`.

  XAML

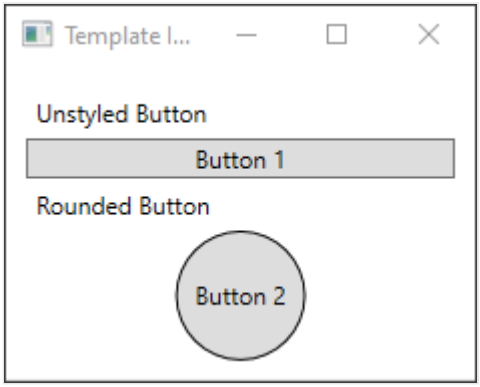
  ```
  <Storyboard>
      <ColorAnimation Storyboard.TargetName="backgroundElement"
          Storyboard.TargetProperty="(Shape.Fill).
  (SolidColorBrush.Color)"
          To="Yellow"
          Duration="0:0:0.3"/>
  </Storyboard>
  ```

KontrolkaTemplate<> powinna teraz wyglądać następująco.

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="CommonStates">
                <VisualState Name="Normal">
                    <Storyboard>
                        <ColorAnimation
Storyboard.TargetName="backgroundElement"
                            Storyboard.TargetProperty="(Shape.Fill).
(SolidColorBrush.Color)"
                            To="{TemplateBinding Background}"
                            Duration="0:0:0.3"/>
                    </Storyboard>
                </VisualState>
                <VisualState Name="MouseOver">
                    <Storyboard>
                        <ColorAnimation
Storyboard.TargetName="backgroundElement"
                            Storyboard.TargetProperty="(Shape.Fill).
(SolidColorBrush.Color)"
                            To="Yellow"
                            Duration="0:0:0.3"/>
                    </Storyboard>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <Ellipse Name="backgroundElement" Fill="{TemplateBinding
Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter x:Name="contentPresenter"
HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

Uruchamianie projektu. Zwróć uwagę, że po przeniesieniu myszy nad przyciskiem kolor **<wielokropka>** animuje.



Następne kroki

- Tworzenie stylu dla kontrolki
- Style i szablony
- Omówienie zasobów XAML

Omówienie dokumentów drukowania (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 10 min

Dzięki platformie Microsoft .NET deweloperzy aplikacji korzystający z platformy Windows Presentation Foundation (WPF) mają bogaty zestaw interfejsów API zarządzania systemami drukowania i drukowania. Podstawą tej funkcji jest format pliku XML Paper Specification (XPS) i ścieżka drukowania XPS.

Informacje o systemie XPS

XPS to elektroniczny format dokumentu, format pliku buforu i język opisu strony. Jest to otwarty format dokumentu, który używa konwencji XML, Open Packaging Conventions i innych standardów branżowych do tworzenia dokumentów międzyplatformowych. System XPS upraszcza proces tworzenia, udostępniania, drukowania, wyświetlania i archiwizowania dokumentów cyfrowych. Aby uzyskać więcej informacji na temat systemu XPS, zobacz Dokumenty XPS.

Ścieżka drukowania XPS

Ścieżka drukowania XPS to funkcja Windows, która definiuje sposób obsługi drukowania w aplikacjach Windows. Ścieżka drukowania XPS może zastąpić:

- Języki prezentacji dokumentów, takie jak format RTF lub przenośny format dokumentu.
- Formaty buforu wydruku, takie jak Windows Metafile lub Enhanced Metafile (EMF).
- Języki opisu strony, takie jak język poleceń drukarki lub PostScript.

W związku z tym ścieżka drukowania XPS utrzymuje format XPS z publikacji aplikacji w dół do końcowego przetwarzania w sterowniku drukarki lub urządzeniu.

Bufor wydruku dla dokumentów XPS obsługuje zarówno ścieżkę drukowania XPS, jak i ścieżkę drukowania GDI. Ścieżka drukowania XPS natywnie korzysta z pliku buforu XPS i wymaga sterownika drukarki XPS. Ścieżka drukowania XPS jest oparta na modelu sterownika drukarki XPS (XPSDrv).

Zalety ścieżki drukowania XPS obejmują:

- Obsługa drukowania WYSIWYG.

- Natywna obsługa zaawansowanych profilów kolorów, takich jak 32 bity na kanał, model kolorów CMYK, nazwane kolory, n-inks i przezroczystości i gradienty.
- Zwiększona wydajność drukowania — funkcje i ulepszenia XPS są dostępne tylko dla aplikacji przeznaczonych dla ścieżki drukowania XPS.
- Format XPS standardu branżowego.

W przypadku podstawowych scenariuszy drukowania prosty i intuicyjny interfejs API jest dostępny za pomocą standardowego interfejsu użytkownika na potrzeby konfiguracji drukowania i przesyłania zadań. W przypadku zaawansowanych scenariuszy interfejs API obsługuje dostosowywanie interfejsu użytkownika lub brak interfejsu użytkownika, synchroniczne lub asynchroniczne drukowanie oraz możliwości drukowania wsadowego. Zarówno proste, jak i zaawansowane opcje zapewniają obsługę drukowania w trybach pełnego lub częściowego zaufania.

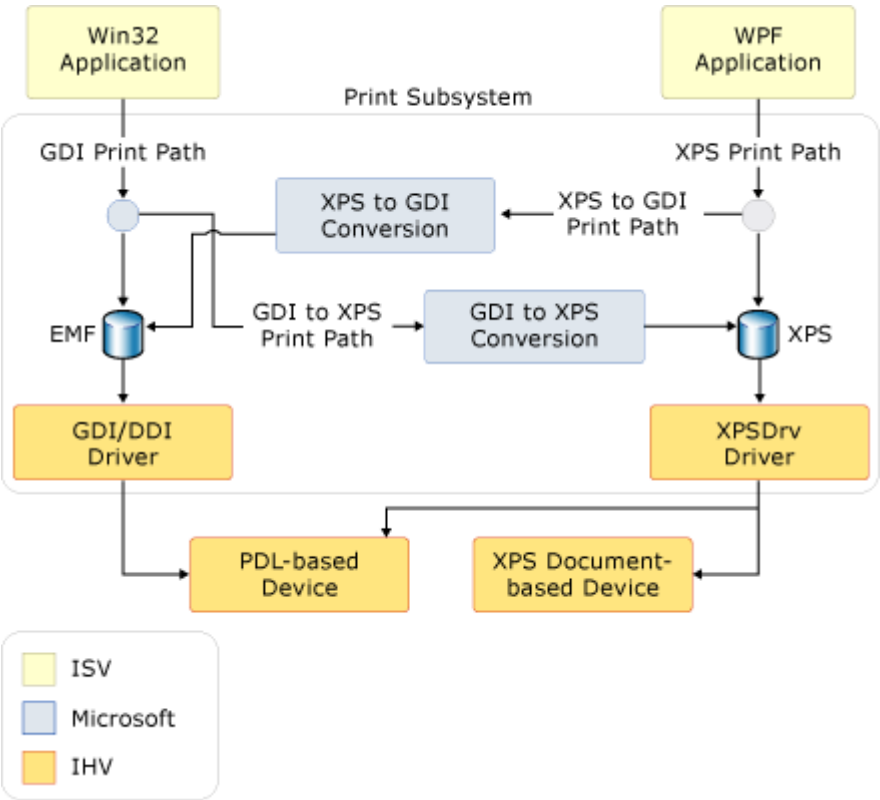
System XPS został zaprojektowany z myślą o rozszerzalności, więc funkcje i możliwości można dodać do systemu XPS w sposób modularny. Funkcje rozszerzalności obejmują:

- Schemat wydruku, który obsługuje szybkie rozszerzenie możliwości urządzenia. Publiczna część schematu jest regularnie aktualizowana w celu dodania żądanych możliwości urządzenia. Aby uzyskać więcej informacji, zobacz Rozszerzalna architektura.
- Rozszerzalny potok filtru używany przez sterowniki XPSDrv do obsługi zarówno bezpośredniego, jak i skalowalnego drukowania dokumentów XPS. Aby uzyskać więcej informacji, zobacz XPSDrv printer drivers (Sterowniki drukarek XPSDrv).

Architektura ścieżki wydruku

Aplikacje WPF natywnie obsługują ścieżkę drukowania XPS i mogą używać interfejsów API drukowania XPS do drukowania bezpośrednio do sterownika XPSDrv. Jeśli docelowa kolejka wydruku operacji zapisu nie ma sterownika XPSDrv, metody WriteXpsDocumentWriter i WriteAsync klasy będą automatycznie konwertować zawartość z XPS na format GDI dla ścieżki wydruku GDI.

Poniższa ilustracja przedstawia podsystem drukowania i definiuje części udostępniane przez firmę Microsoft oraz niezależnych dostawców oprogramowania i sprzętu.



Podstawowe drukowanie XPS

WPF ma interfejs API drukowania, który obsługuje zarówno podstawowe, jak i zaawansowane funkcje drukowania. W przypadku tych aplikacji, które nie wymagają rozbudowanego dostosowywania wydruku ani dostępu do kompletnego zestawu funkcji XPS, wystarczy podstawowa obsługa drukowania. Obsługa drukowania podstawowego jest zapewniana za pomocą kontrolki PrintDialog , która wymaga minimalnej konfiguracji, ma znany interfejs użytkownika i obsługuje wiele funkcji XPS.

PrintDialog

Kontrolka System.Windows.Controls.PrintDialog zapewnia pojedynczy punkt wejścia do interfejsu użytkownika, konfiguracji i przesyłania zadań XPS. Aby dowiedzieć się, jak utworzyć wystąpienie kontrolki i użyć jej, zobacz Jak wyświetlić okno dialogowe drukowania.

Zaawansowane drukowanie XPS

Aby uzyskać dostęp do pełnego zestawu funkcji XPS, użyj zaawansowanego interfejsu API drukowania. W tej sekcji opisano kilka odpowiednich interfejsów API, w tym PrintTicket, PrintCapabilities, PrintServer, PrintQueue i XpsDocumentWriter. Aby uzyskać pełną listę interfejsów API ścieżki wydruku XPS, zobacz System.Windows.Xps przestrzenie nazw i System.Printing .

PrintTicket i PrintCapabilities

Klasy PrintTicket i PrintCapabilities są podstawą zaawansowanych funkcji XPS. Oba obiekty zawierają struktury sformatowane XML cech zorientowanych na drukowanie, które są definiowane przez schemat wydruku. Funkcje obejmują dupleksowanie, sortowanie, drukowanie dwustronne i zszycie. A `PrintTicket` instruuje drukarkę, jak przetwarzać zadanie drukowania. Klasa `PrintCapabilities` definiuje możliwości drukarki. Wykonując zapytanie dotyczące możliwości drukarki, można utworzyć element, `PrintTicket` który w pełni wykorzystuje obsługiwane funkcje drukarki. Podobnie można uniknąć nieobsługiwanych funkcji.

Poniższy przykład wykonuje zapytanie względem `PrintCapabilities` drukarki i tworzy przy `PrintTicket` użyciu kodu.

C#

```
/// <summary>
/// Returns a print ticket, which is a set of instructions telling a printer
how
/// to set its various features, such as duplexing, collating, and stapling.
/// </summary>
/// <param name="printQueue">The print queue to print to.</param>
/// <returns>A print ticket.</returns>
public static PrintTicket GetPrintTicket(PrintQueue printQueue)
{
    PrintCapabilities printCapabilites = printQueue.GetPrintCapabilities();

    // Get a default print ticket from printer.
    PrintTicket printTicket = printQueue.DefaultPrintTicket;

    // Modify the print ticket.
    if (printCapabilites.CollationCapability.Contains(Collation.Collated))
        printTicket.Collation = Collation.Collated;
    if
(printCapabilites.DuplexingCapability.Contains(Duplexing.TwoSidedLongEdge))
        printTicket.Duplexing = Duplexing.TwoSidedLongEdge;
    if
(printCapabilites.StaplingCapability.Contains(Stapling.StapleDualLeft))
        printTicket.Stapling = Stapling.StapleDualLeft;

    // Returns a print ticket, which is a set of instructions telling a
printer how
    // to set its various features, such as duplexing, collating, and
stapling.
    return printTicket;
}
```

PrintServer i PrintQueue

Klasa PrintServer reprezentuje serwer wydruku sieciowego, a PrintQueue klasa reprezentuje drukarkę i kolejkę zadań wyjściowych skojarzonych z nim. Te interfejsy API obsługują zaawansowane zarządzanie zadaniami drukowania dla serwera. Klasa `PrintServer` lub jedna z jej klas pochodnych służy do zarządzania klasą `PrintQueue`.

Poniższy przykład tworzy obiekt LocalPrintServer i uzyskuje dostęp do komputera PrintQueueCollection lokalnego przy użyciu kodu.

C#

```
/// <summary>
/// Return a collection of print queues, which individually hold the
features or states
/// of a printer as well as common properties for all print queues.
/// </summary>
/// <returns>A collection of print queues.</returns>
public static PrintQueueCollection GetPrintQueues()
{
    // Create a LocalPrintServer instance, which represents
    // the print server for the local computer.
    LocalPrintServer localPrintServer = new();

    // Get the default print queue on the local computer.
    //PrintQueue printQueue = localPrintServer.DefaultPrintQueue;

    // Get all print queues on the local computer.
    PrintQueueCollection printQueueCollection =
localPrintServer.GetPrintQueues();

    // Return a collection of print queues, which individually hold the
features or states
    // of a printer as well as common properties for all print queues.
    return printQueueCollection;
}
```

XpsDocumentWriter

XpsDocumentWriter, z wieloma Write metodami i WriteAsync , służy do dodawania dokumentów XPS do klasy PrintQueue. Na przykład Write(FixedDocumentSequence, PrintTicket) metoda służy do synchronicznego dodawania dokumentu XPS z biletem wydruku do kolejki. Metoda WriteAsync(FixedDocumentSequence, PrintTicket) służy do asynchronicznego dodawania dokumentu XPS z biletem wydruku do kolejki.

Poniższy przykład tworzy `XpsDocumentWriter` i dodaje dokumenty XPS zarówno synchronicznie, jak i asynchronicznie do `PrintQueue` kodu using.

C#

```
/// <summary>
/// Asynchronously, add the XPS document together with a print ticket to the
print queue.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file.</param>
/// <param name="printQueue">The print queue to print to.</param>
/// <param name="printTicket">The print ticket for the selected print queue.
</param>
public static void PrintXpsDocumentAsync(string xpsFilePath, PrintQueue
printQueue, PrintTicket printTicket)
{
    // Create an XpsDocumentWriter object for the print queue.
    XpsDocumentWriter xpsDocumentWriter =
PrintQueue.CreateXpsDocumentWriter(printQueue);

    // Open the selected document.
    XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

    // Get a fixed document sequence for the selected document.
    FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();

    // Asynchronously, add the XPS document together with a print ticket to
the print queue.
    xpsDocumentWriter.WriteAsync(fixedDocSeq, printTicket);
}

/// <summary>
/// Synchronously, add the XPS document together with a print ticket to the
print queue.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file.</param>
/// <param name="printQueue">The print queue to print to.</param>
/// <param name="printTicket">The print ticket for the selected print queue.
</param>
public static void PrintXpsDocument(string xpsFilePath, PrintQueue
printQueue, PrintTicket printTicket)
{
    // Create an XpsDocumentWriter object for the print queue.
    XpsDocumentWriter xpsDocumentWriter =
PrintQueue.CreateXpsDocumentWriter(printQueue);

    // Open the selected document.
    XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

    // Get a fixed document sequence for the selected document.
    FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();

    // Synchronously, add the XPS document together with a print ticket to
the print queue.
    xpsDocumentWriter.Write(fixedDocSeq, printTicket);
}
```

Ścieżka drukowania interfejsu GDI

Mimo że aplikacje WPF natywnie obsługują ścieżkę drukowania XPS, mogą również dawać dane wyjściowe do ścieżki drukowania GDI, wywołując jedną z Write metod XpsDocumentWriter lub WriteAsync klasy i wybierając kolejkę wydruku dla drukarki innej niż XpsDrv.

W przypadku aplikacji, które nie wymagają funkcji ani obsługi systemu XPS, bieżąca ścieżka drukowania interfejsu GDI pozostaje niezmieniona. Aby uzyskać więcej informacji na temat ścieżki drukowania GDI i różnych opcji konwersji XPS, zobacz Microsoft XPS document converter (MXDC) i XPSDrv sterowników drukarek.

Model sterownika XPSDrv

Ścieżka drukowania XPS zwiększa wydajność buforu dzięki użyciu systemu XPS jako natywnego formatu buforu wydruku podczas drukowania na drukarce lub sterowniku z włączoną obsługą systemu XPS. W przeciwieństwie do pakietu EMF, który reprezentuje dane wyjściowe aplikacji jako serię wywołań do interfejsu GDI na potrzeby usług renderowania, format buforu XPS reprezentuje dokument. Dlatego gdy pliki buforu XPS są przekazywane do sterownika drukarki opartej na systemie XPS, nie wymagają dalszej interpretacji, ponieważ sterowniki działają bezpośrednio na danych w tym formacie. Ta funkcja eliminuje konwersje danych i przestrzeni kolorów wymagane dla plików EMF i sterowników wydruku opartych na GDI.

Uproszczony proces buforowania eliminuje konieczność wygenerowania pośredniego pliku buforu, takiego jak plik danych EMF, przed buforowaniem dokumentu. W przypadku mniejszych rozmiarów plików buforu ścieżka drukowania XPS może zmniejszyć ruch sieciowy i zwiększyć wydajność drukowania. W porównaniu z ich odpowiednikami EMF rozmiary plików buforu XPS są zwykle zmniejszane w przypadku korzystania ze ścieżki drukowania XPS. Zmniejszenie rozmiaru pliku buforu odbywa się za pomocą kilku mechanizmów:

- **Podsettingowanie czcionek**, które przechowuje tylko znaki używane w dokumencie w pliku XPS.
- **Zaawansowana obsługa grafiki**, która natywnie obsługuje przezroczystość i typy pierwotne gradientu, aby uniknąć rasteryzacji zawartości XPS.
- **Identyfikacja typowych zasobów**, takich jak obraz logo firmy używanego wiele razy w dokumencie. Typowe zasoby są traktowane jako zasoby udostępnione i są ładowane tylko raz.
- **Kompresja ZIP** używana we wszystkich dokumentach XPS.

Rozmiar pliku buforu XPS może nie zostać zmniejszony, jeśli grafika wektorowa jest wysoce złożona, wielowarstwowa lub niewydajnie napisana. W przeciwieństwie do plików buforu GDI pliki XPS osadzają czcionki urządzeń i czcionki oparte na komputerze na potrzeby wyświetlania ekranu, choć oba rodzaje czcionek są podzbiorne, a sterowniki drukarek mogą usuwać czcionki urządzenia przed przesłaniem pliku do drukarki.

> **Porada**
>
> Pliki XPS można również drukować przy użyciu **PrintQueue.AddJob** metod. Aby uzyskać więcej informacji, zobacz **Jak drukować pliki XPS**.

Zobacz też

- PrintDialog
- XpsDocumentWriter
- XpsDocument
- PrintTicket
- PrintCapabilities
- PrintServer
- PrintQueue
- Tematy z instrukcjami
- Dokumenty w WPF
- Dokumenty XPS
- Serializacja i magazyn dokumentów
- Konwerter dokumentów MICROSOFT XPS (MXDC)

Jak wyświetlić okno dialogowe drukowania (WPF .NET)

Artykuł • 11.02.2023 • Czas czytania: 7 min

Chcesz wydrukować z aplikacji? Klasy można użyć PrintDialog do otwarcia standardowego okna dialogowego drukowania systemu Microsoft Windows. Oto jak to zrobić.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

> ⓘ **Uwaga**
>
> Kontrolka **System.Windows.Controls.PrintDialog** używana dla WPF i omówiona tutaj nie powinna być mylona ze składnikiem **System.Windows.Forms.PrintDialog** Windows Forms.

Klasa PrintDialog zapewnia jedną kontrolkę do konfigurowania wydruku i przesyłania zadań drukowania. Kontrolka jest łatwa w użyciu i może zostać utworzone za pomocą znaczników XAML lub kodu. Poniższe przykłady tworzą i wyświetlają `PrintDialog` wystąpienie przy użyciu kodu.

Za pomocą okna dialogowego drukowania można skonfigurować opcje drukowania, takie jak:

- Drukuj tylko określony zakres stron.
- Wybierz z drukarek zainstalowanych na komputerze. Aby utworzyć następujące typy dokumentów, możesz użyć opcji **Składnik zapisywania dokumentów XPS firmy Microsoft** :
  - XML Paper Specification (XPS)
  - Specyfikacja open XML Paper (OpenXPS)

Drukuj cały dokument

Ten przykład wyświetla wszystkie strony dokumentu XPS. Domyślnie kod będzie:

1. Otwórz okno dialogowe drukowania, które monituje użytkownika o wybranie drukarki i uruchomienie zadania drukowania.
2. XpsDocument Utwórz wystąpienie obiektu z zawartością dokumentu XPS.
3. `XpsDocument` Użyj obiektu , aby wygenerować DocumentPaginator obiekt, który przechowuje wszystkie strony dokumentu XPS.
4. Wywołaj metodę PrintDocument , przekazując `DocumentPaginator` obiekt, aby wysłać wszystkie strony do określonej drukarki.

C#

```
/// <summary>
/// Print all pages of an XPS document.
/// Optionally, hide the print dialog window.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file</param>
/// <param name="hidePrintDialog">Whether to hide the print dialog window
(shown by default)</param>
/// <returns>Whether the document printed</returns>
public static bool PrintWholeDocument(string xpsFilePath, bool
hidePrintDialog = false)
{
    // Create the print dialog object and set options.
    PrintDialog printDialog = new();

    if (!hidePrintDialog)
    {
        // Display the dialog. This returns true if the user presses the
Print button.
        bool? isPrinted = printDialog.ShowDialog();
        if (isPrinted != true)
            return false;
    }

    // Print the whole document.
    try
    {
        // Open the selected document.
        XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

        // Get a fixed document sequence for the selected document.
        FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();

        // Create a paginator for all pages in the selected document.
        DocumentPaginator docPaginator = fixedDocSeq.DocumentPaginator;

        // Print to a new file.
        printDialog.PrintDocument(docPaginator, $"Printing
{Path.GetFileName(xpsFilePath)}");

        return true;
    }
```

```
    catch (Exception e)
    {
        MessageBox.Show(e.Message);

        return false;
    }
}
```

Drukowanie zakresu stron

Czasami chcesz wydrukować tylko określony zakres stron w dokumencie XPS. W tym celu rozszerzymy klasę abstrakcyjną DocumentPaginator , aby dodać obsługę zakresów stron. Domyślnie kod będzie:

1. Otwórz okno dialogowe drukowania, które monituje użytkownika o wybranie drukarki, określenie zakresu stron i uruchomienie zadania drukowania.
2. XpsDocument Utwórz wystąpienie obiektu z zawartością dokumentu XPS.
3. Użyj obiektu , `XpsDocument` aby wygenerować obiekt domyślny DocumentPaginator , który przechowuje wszystkie strony dokumentu XPS.
4. Utwórz wystąpienie klasy rozszerzonej `DocumentPaginator` obsługującej zakresy stron, przekazując obiekt domyślny `DocumentPaginator` i PageRange strukturę zwróconą przez PrintDialogobiekt .
5. Wywołaj metodę PrintDocument , przekazując wystąpienie klasy rozszerzonej `DocumentPaginator` , aby wysłać określony zakres stron do określonej drukarki.

C#

```
/// <summary>
/// Print a specific range of pages within an XPS document.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file</param>
/// <returns>Whether the document printed</returns>
public static bool PrintDocumentPageRange(string xpsFilePath)
{
    // Create the print dialog object and set options.
    PrintDialog printDialog = new()
    {
        UserPageRangeEnabled = true
    };

    // Display the dialog. This returns true if the user presses the Print
button.
    bool? isPrinted = printDialog.ShowDialog();
    if (isPrinted != true)
        return false;

    // Print a specific page range within the document.
```

```
    try
    {
        // Open the selected document.
        XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

        // Get a fixed document sequence for the selected document.
        FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();

        // Create a paginator for all pages in the selected document.
        DocumentPaginator docPaginator = fixedDocSeq.DocumentPaginator;

        // Check whether a page range was specified in the print dialog.
        if (printDialog.PageRangeSelection == PageRangeSelection.UserPages)
        {
            // Create a document paginator for the specified range of pages.
            docPaginator = new DocPaginator(fixedDocSeq.DocumentPaginator,
printDialog.PageRange);
        }

        // Print to a new file.
        printDialog.PrintDocument(docPaginator, $"Printing
{Path.GetFileName(xpsFilePath)}");

        return true;
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);

        return false;
    }
}

/// <summary>
/// Extend the abstract DocumentPaginator class to support page range
printing. This class is based on the following online resources:
///
/// https://www.thomasclaudiushuber.com/2009/11/24/wpf-printing-how-to-
print-a-pagerange-with-wpfs-printdialog-that-means-the-user-can-select-
specific-pages-and-only-these-pages-are-printed/
///
/// https://social.msdn.microsoft.com/Forums/vstudio/en-US/9180e260-0791-
4f2d-962d-abcb22ba8d09/how-to-print-multiple-page-ranges-with-wpf-
printdialog
///
/// https://social.msdn.microsoft.com/Forums/en-US/841e804b-9130-4476-8709-
0d2854c11582/exception-quotfixedpage-cannot-contain-another-fixedpagequot-
when-printing-to-the-xps-document?forum=wpf
/// </summary>
public class DocPaginator : DocumentPaginator
{
    private readonly DocumentPaginator _documentPaginator;
    private readonly int _startPageIndex;
    private readonly int _endPageIndex;
```

```
    private readonly int _pageCount;

    public DocPaginator(DocumentPaginator documentPaginator, PageRange
pageRange)
    {
        // Set document paginator.
        _documentPaginator = documentPaginator;

        // Set page indices.
        _startPageIndex = pageRange.PageFrom - 1;
        _endPageIndex = pageRange.PageTo - 1;

        // Validate and set page count.
        if (_startPageIndex >= 0 &&
            _endPageIndex >= 0 &&
            _startPageIndex <= _documentPaginator.PageCount - 1 &&
            _endPageIndex <= _documentPaginator.PageCount - 1 &&
            _startPageIndex <= _endPageIndex)
            _pageCount = _endPageIndex - _startPageIndex + 1;
    }

    public override bool IsPageCountValid => true;

    public override int PageCount => _pageCount;

    public override IDocumentPaginatorSource Source =>
_documentPaginator.Source;

    public override Size PageSize { get => _documentPaginator.PageSize; set
=> _documentPaginator.PageSize = value; }

    public override DocumentPage GetPage(int pageNumber)
    {
        DocumentPage documentPage =
_documentPaginator.GetPage(_startPageIndex + pageNumber);

        // Workaround for "FixedPageInPage" exception.
        if (documentPage.Visual is FixedPage fixedPage)
        {
            var containerVisual = new ContainerVisual();
            foreach (object child in fixedPage.Children)
            {
                var childClone =
(UIElement)child.GetType().GetMethod("MemberwiseClone",
BindingFlags.Instance | BindingFlags.NonPublic).Invoke(child, null);

                FieldInfo parentField =
childClone.GetType().GetField("_parent", BindingFlags.Instance |
BindingFlags.NonPublic);
                if (parentField != null)
                {
                    parentField.SetValue(childClone, null);
                    containerVisual.Children.Add(childClone);
                }
            }
```

```
            return new DocumentPage(containerVisual, documentPage.Size,
documentPage.BleedBox, documentPage.ContentBox);
        }

        return documentPage;
    }
}
```

Porada

Chociaż można użyć **PrintDocument** metody do drukowania bez otwierania okna dialogowego drukowania, ze względów wydajności lepiej użyć **AddJob** metody lub jednej z wielu **Write** metod i **WriteAsync** .**XpsDocumentWriter** Aby uzyskać więcej informacji na ten temat, zobacz **Jak wydrukować plik XPS** i **Omówienie dokumentów drukowania**.

Zobacz też

- PrintDialog
- Dokumenty w WPF
- Jak wydrukować plik XPS
- Drukowanie dokumentów — omówienie
- Moduł zapisywania dokumentów XPS firmy Microsoft

Jak wydrukować plik XPS (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 5 min

Czasami należy dodać nowe zadanie drukowania do kolejki wydruku bez otwierania okna dialogowego drukowania. W tym celu możesz użyć jednej z PrintQueue.AddJob metod. Oto jak to zrobić.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

W poniższym przykładzie używamy AddJob(String, String, Boolean) metody , jednego z kilku przeciążeń `AddJob` klasy , do:

- Dodaj nowe zadanie drukowania dla dokumentu specyfikacji papieru XML (XPS) do domyślnej kolejki wydruku.
- Nadaj nowe zadanie nazwę.
- Określ, czy dokument XPS ma zostać zweryfikowany (przy użyciu parametru `fastCopy`).

W przypadku korzystania z `AddJob(String, String, Boolean)` metody wartość parametru `fastCopy` jest kluczową kwestią:

- Jeśli ustawisz parametr na `fastCopy` `true` wartość , walidacja XPS zostanie pominięta, a zadanie drukowania będzie szybko buforować bez opinii o postępie strony po stronie.
- Jeśli ustawisz parametr na `false`, wątek, który wywołuje `AddJob` metodę, musi mieć stan apartamentu `fastCopy` jednowątkowego, w przeciwnym razie zostanie zgłoszony wyjątek. Aby uzyskać więcej informacji, zobacz sekcję **Uwagi** dotyczące AddJob(String, String, Boolean) programu .

Dodawanie nowych zadań drukowania do kolejki

W tym przykładzie do kolejki domyślnej jest dodawany co najmniej jeden dokument XPS. Kod:

1. Użyj polecenia Task.Run , aby uniknąć blokowania wątku interfejsu użytkownika — ponieważ nie ma asynchronicznego `AddJob` elementu .
2. Jeśli wartość parametru `fastCopy` to `false`, uruchom w AddJob(String, String, Boolean) wątku ze stanem apartamentu jednowątkowego.
3. Pobierz odwołanie do wartości domyślnej PrintQueue elementu LocalPrintServer.
4. Wywołaj `AddJob(String, String, Boolean)` odwołanie do kolejki wydruku, przekazując nazwę zadania, ścieżkę dokumentu XPS i `fastCopy` parametr.

Jeśli kolejka nie jest wstrzymana i drukarka działa, zadanie drukowania rozpocznie się automatycznie po osiągnięciu górnej części kolejki wydruku.

> **Porada**
>
> Aby uniknąć okna dialogowego **Zapisywanie pliku wyjściowego jako** podczas dodawania zadania drukowania do kolejki domyślnej, upewnij się, że domyślna drukarka nie jest składnikiem **zapisywania dokumentów w systemie Microsoft XPS**, **microsoft print to PDF** ani innymi opcjami drukowania do pliku.

C#

```
/// <summary>
/// Asyncronously, add a batch of XPS documents to the print queue using a
PrintQueue.AddJob method.
/// Handle the thread apartment state required by the PrintQueue.AddJob
method.
/// </summary>
/// <param name="xpsFilePaths">A collection of XPS documents.</param>
/// <param name="fastCopy">Whether to validate the XPS documents.</param>
/// <returns>Whether all documents were added to the print queue.</returns>
public static async Task<bool> BatchAddToPrintQueueAsync(IEnumerable<string>
xpsFilePaths, bool fastCopy = false)
{
    bool allAdded = true;

    // Queue some work to run on the ThreadPool.
    // Wait for completion without blocking the calling thread.
    await Task.Run(() =>
    {
        if (fastCopy)
            allAdded = BatchAddToPrintQueue(xpsFilePaths, fastCopy);
        else
        {
            // Create a thread to call the PrintQueue.AddJob method.
            Thread newThread = new(() =>
            {
                allAdded = BatchAddToPrintQueue(xpsFilePaths, fastCopy);
            });
```

```
            // Set the thread to single-threaded apartment state.
            newThread.SetApartmentState(ApartmentState.STA);

            // Start the thread.
            newThread.Start();

            // Wait for thread completion. Blocks the calling thread,
            // which is a ThreadPool thread.
            newThread.Join();
        }
    });

    return allAdded;
}

/// <summary>
/// Add a batch of XPS documents to the print queue using a
PrintQueue.AddJob method.
/// </summary>
/// <param name="xpsFilePaths">A collection of XPS documents.</param>
/// <param name="fastCopy">Whether to validate the XPS documents.</param>
/// <returns>Whether all documents were added to the print queue.</returns>
public static bool BatchAddToPrintQueue(IEnumerable<string> xpsFilePaths,
bool fastCopy)
{
    bool allAdded = true;

    // To print without getting the "Save Output File As" dialog, ensure
    // that your default printer is not the Microsoft XPS Document Writer,
    // Microsoft Print to PDF, or other print-to-file option.

    // Get a reference to the default print queue.
    PrintQueue defaultPrintQueue = LocalPrintServer.GetDefaultPrintQueue();

    // Iterate through the document collection.
    foreach (string xpsFilePath in xpsFilePaths)
    {
        // Get document name.
        string xpsFileName = Path.GetFileName(xpsFilePath);

        try
        {
            // The AddJob method adds a new print job for an XPS
            // document into the print queue, and assigns a job name.
            // Use fastCopy to skip XPS validation and progress
notifications.
            // If fastCopy is false, the thread that calls PrintQueue.AddJob
            // must have a single-threaded apartment state.
            PrintSystemJobInfo xpsPrintJob =
                    defaultPrintQueue.AddJob(jobName: xpsFileName,
documentPath: xpsFilePath, fastCopy);

            // If the queue is not paused and the printer is working, then
jobs will automatically begin printing.
            Debug.WriteLine($"Added {xpsFileName} to the print queue.");
```

```
        }
        catch (PrintJobException e)
        {
            allAdded = false;
            Debug.WriteLine($"Failed to add {xpsFileName} to the print
queue: {e.Message}\r\n{e.InnerException}");
        }
    }

    return allAdded;
}
```

Porada

Możesz również drukować pliki XPS przy użyciu:

- **PrintDialog.PrintDocument** lub **PrintDialog.PrintVisual** metody.
- **XpsDocumentWriter.Write** i **XpsDocumentWriter.WriteAsync** metody.

Aby uzyskać więcej informacji, zobacz **Jak wyświetlić okno dialogowe drukowania** i **Omówienie dokumentów drukowania**.

Zobacz też

- PrintQueue.AddJob
- Dokumenty w WPF
- Jak wyświetlić okno dialogowe drukowania
- Omówienie dokumentów drukowania

Omówienie powiązań danych (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 33 min

Powiązanie danych w programie Windows Presentation Foundation (WPF) zapewnia prosty i spójny sposób prezentowania i interakcji z danymi przez aplikacje. Elementy mogą być powiązane z danymi z różnych rodzajów źródeł danych w postaci obiektów platformy .NET i kodu XML. Wszystkie ContentControl, takie jak Button, i dowolne ItemsControl, takie jak ListBox i ListView, mają wbudowaną funkcjonalność umożliwiającą elastyczne stylizowanie pojedynczych elementów danych lub kolekcji elementów danych. Widoki sortowania, filtrowania i grupowania można wygenerować na podstawie danych.

Powiązanie danych w programie WPF ma kilka zalet w porównaniu z tradycyjnymi modelami, a w tym nieodłączną obsługę powiązania danych przez szeroką gamę właściwości, elastyczną reprezentację danych w interfejsie użytkownika i czystą separację logiki biznesowej od interfejsu użytkownika.

W tym artykule omówiono najpierw pojęcia podstawowe dotyczące powiązania danych WPF, a następnie użycie klasy Binding i innych funkcji powiązania danych.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

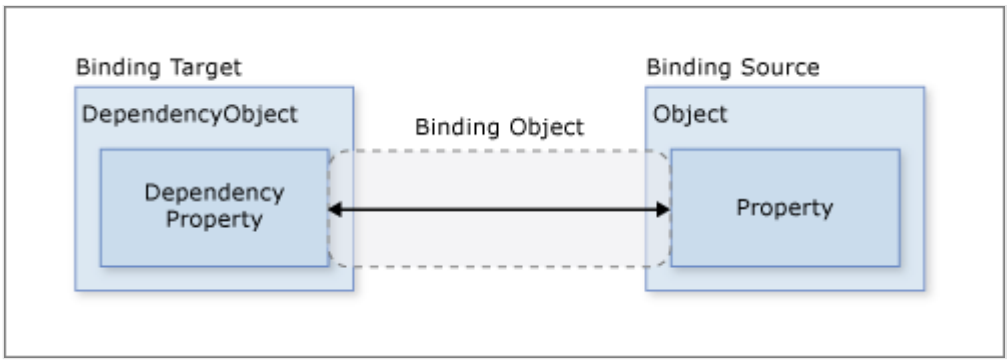
Co to jest powiązanie danych?

Powiązanie danych to proces, który ustanawia połączenie między interfejsem użytkownika aplikacji a wyświetlanymi danymi. Jeśli powiązanie ma poprawne ustawienia, a dane udostępniają odpowiednie powiadomienia, gdy zmieniają one jego wartość, elementy powiązane z danymi automatycznie odzwierciedlają zmiany. Powiązanie danych może również oznaczać, że jeśli zewnętrzna reprezentacja danych w elemencie ulegnie zmianie, dane bazowe można automatycznie zaktualizować, aby odzwierciedlić zmianę. Jeśli na przykład użytkownik edytuje wartość elementu `TextBox`, wartość danych bazowych zostanie automatycznie zaktualizowana w celu odzwierciedlenia tej zmiany.

Typowym zastosowaniem powiązania danych jest umieszczenie danych serwera lub lokalnej konfiguracji w formularzach lub innych kontrolkach interfejsu użytkownika. W programie WPF ta koncepcja została rozszerzona, aby uwzględnić powiązanie szerokiego zakresu właściwości z różnymi rodzajami źródeł danych. W programie WPF właściwości zależności elementów mogą być powiązane z obiektami platformy .NET (a w tym obiektami ADO.NET lub obiektami skojarzonymi z usługami i właściwościami internetowymi) oraz danymi XML.

Podstawowe pojęcia dotyczące powiązań danych

Niezależnie od tego, który element jest wiązany, i charakteru źródła danych, każde powiązanie zawsze jest zgodne z modelem przedstawionym na poniższej ilustracji.



Jak pokazano na rysunku, powiązanie danych jest zasadniczo mostem między obiektem docelowym powiązania a źródłem powiązania. Na rysunku przedstawiono następujące podstawowe pojęcia dotyczące powiązania danych WPF:

- Zazwyczaj każde powiązanie ma cztery składniki:
  - Obiekt docelowy powiązania.
  - Właściwość docelową.
  - Źródło powiązania.
  - Ścieżkę do wartości w źródle powiązania do użycia.

  Jeśli na przykład wiązana jest zawartość elementu `TextBox` z właściwością `Employee.Name`, należy skonfigurować powiązanie, takie jak poniższa tabela:

Ustawienie	Wartość
Cel	TextBox
Właściwość docelowa	Text
Obiekt źródłowy	`Employee`

Ustawienie	Wartość
Ścieżka wartości obiektu źródłowego	`Name`

- Właściwość docelowa musi być właściwością zależności.

  Większość właściwości UIElement to właściwości zależności, a większość właściwości zależności, z wyjątkiem właściwości tylko do odczytu, domyślnie obsługuje powiązanie danych. Właściwości zależności można definiować tylko w typach pochodnych DependencyObject. Wszystkie typy UIElement pochodzą z klasy `DependencyObject`.

- Źródła powiązań nie są ograniczone do niestandardowych obiektów .NET.

  Chociaż nie pokazano na rysunku, należy zauważyć, że obiekt źródłowy powiązania nie jest ograniczony do bycia niestandardowym obiektem platformy .NET. Powiązanie danych WPF obsługuje dane w postaci obiektów .NET, XML, a nawet obiektów elementów XAML. Aby podać kilka przykładów, źródło powiązania może być UIElement, dowolnym obiektem listy, obiektem ADO.NET lub usług internetowych albo węzłem XmlNode zawierającym dane XML. Aby uzyskać więcej informacji, zobacz Omówienie źródeł powiązań.

Ważne jest, aby pamiętać, że podczas ustanawiania powiązania należy wiązać element docelowy powiązania *ze* źródłem powiązania. Jeśli na przykład wyświetlasz niektóre bazowe dane XML w ListBox przy użyciu powiązania danych, wiążesz `ListBox` z danymi XML.

Aby ustanowić powiązanie, należy użyć obiektu Binding. W pozostałej części tego artykułu omówiono wiele pojęć związanych z obiektem `Binding` oraz niektóre właściwości i zastosowania tego obiektu.

Kontekst danych

Gdy powiązanie danych jest deklarowane w elementach XAML, rozwiązuje powiązanie danych, sprawdzając ich natychmiastową właściwość DataContext. Kontekst danych jest zazwyczaj **obiektem źródłowym powiązania** dla oceny **ścieżki wartości źródłowej powiązania**. To zachowanie można zastąpić w powiązaniu i ustawić określoną wartość **obiektu źródłowego powiązania**. Jeśli właściwość `DataContext` obiektu hostująca powiązanie nie jest ustawiona, właściwość elementu nadrzędnego `DataContext` jest zaznaczona i tak dalej, aż do katalogu głównego drzewa obiektów XAML. Krótko mówiąc, kontekst danych używany do rozpoznawania powiązania jest dziedziczony z elementu nadrzędnego, chyba że jest jawnie ustawiony na obiekcie.

Powiązania można skonfigurować do rozpoznawania za pomocą określonego obiektu w przeciwieństwie do używania kontekstu danych do rozpoznawania powiązań. Określenie obiektu źródłowego bezpośrednio jest używane, gdy na przykład kolor pierwszego planu obiektu jest powiązany z kolorem tła innego obiektu. Kontekst danych nie jest potrzebny, ponieważ powiązanie jest rozpoznawane między tymi dwoma obiektami. Odwrotnie powiązania, które nie są powiązane z określonymi obiektami źródłowymi, używają rozpoznawania kontekstu danych.

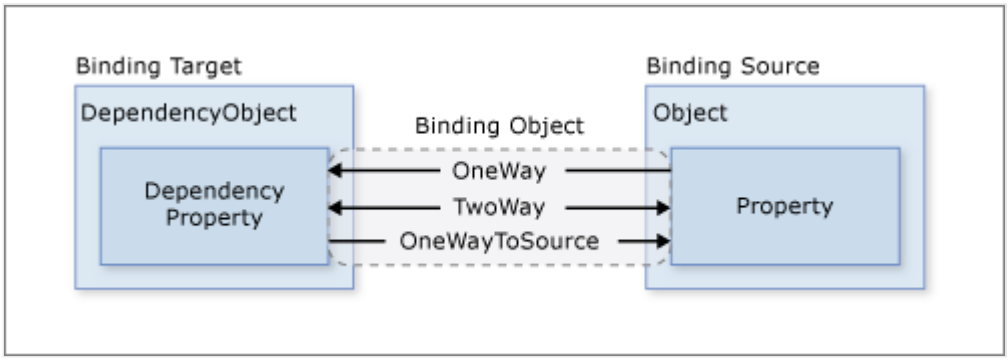
Po zmianie właściwości `DataContext` wszystkie powiązania, na które może mieć wpływ kontekst danych, są ponownie oceniane.

Kierunek przepływu danych

Jak wskazano przez strzałkę na poprzedniej ilustracji, przepływ danych powiązania może przechodzić z elementu docelowego powiązania do źródła powiązania (na przykład wartość źródłowa zmienia się, gdy użytkownik edytuje wartość elementu `TextBox`) i/lub ze źródła powiązania do obiektu docelowego powiązania (na przykład zawartość `TextBox` jest aktualizowana za pomocą zmian w źródle powiązania), jeśli źródło powiązania zapewnia właściwe powiadomienia.

Aplikacja może umożliwić użytkownikom zmianę danych i propagację ich z powrotem do obiektu źródłowego. Możesz też nie chcieć zezwolić użytkownikom na aktualizowanie danych źródłowych. Przepływ danych można kontrolować, ustawiając wartość Binding.Mode.

Na rysunku przedstawiono różne typy przepływu danych:



- Powiązanie OneWay powoduje, że zmiany właściwości źródłowej automatycznie aktualizują właściwość docelową, ale zmiany właściwości docelowej nie są propagowane z powrotem do właściwości źródłowej. Ten typ powiązania jest odpowiedni, jeśli wiązana kontrolka jest niejawnie tylko do odczytu. Na przykład można powiązać ze źródłem, takim jak znacznik akcji, lub być może właściwość docelowa nie ma interfejsu sterującego do wprowadzania zmian, takich jak kolor tła powiązanego z danymi tabeli. Jeśli nie ma potrzeby monitorowania zmian

właściwości docelowej, użycie trybu powiązania OneWay pozwala uniknąć narzutu pracy w trybie powiązania TwoWay.

- Powiązanie TwoWay powoduje zmiany właściwości źródłowej albo właściwości docelowej, aby automatycznie zaktualizować tę drugą. Ten typ powiązania jest odpowiedni dla formularzy edytowalnych lub innych w pełni interaktywnych scenariuszy interfejsu użytkownika. Większość właściwości jest domyślnie wiązana jako OneWay, ale niektóre właściwości zależności (zazwyczaj właściwości kontrolek edytowalnych przez użytkownika, takie jak TextBox.Text i CheckBox.IsChecked są domyślne powiązanie jako TwoWay. Programowy sposób określania, czy właściwość zależności wiąże się domyślnie jednokierunkowo, czy też dwukierunkowo, jest pobranie metadanych właściwości za pomocą polecenia DependencyProperty.GetMetadata, a następnie sprawdzenie wartości logicznej właściwości FrameworkPropertyMetadata.BindsTwoWayByDefault.

- OneWayToSource jest odwrotnym powiązaniem OneWay; aktualizuje właściwość źródłową, gdy właściwość docelowa ulegnie zmianie. Jednym z przykładowych scenariuszy jest to, że wystarczy ponownie ocenić wartość źródłową z interfejsu użytkownika.

- Nie pokazano na rysunku powiązania OneTime, co powoduje, że właściwość źródłowa inicjuje właściwość docelową, ale nie propaguje kolejnych zmian. Jeśli kontekst danych zmieni się lub obiekt w kontekście danych zmieni się, zmiana *nie* zostanie odzwierciedlona we właściwości docelowej. Ten typ powiązania jest odpowiedni, jeśli migawka bieżącego stanu jest odpowiednia albo dane są naprawdę statyczne. Ten typ powiązania jest również przydatny, jeśli chcesz zainicjować właściwość docelową pewną wartością z właściwości źródłowej, a kontekst danych nie jest wcześniej znany. Ten tryb jest zasadniczo prostszą formą powiązania OneWay, które zapewnia lepszą wydajność w przypadkach, gdy wartość źródłowa nie zmienia się.

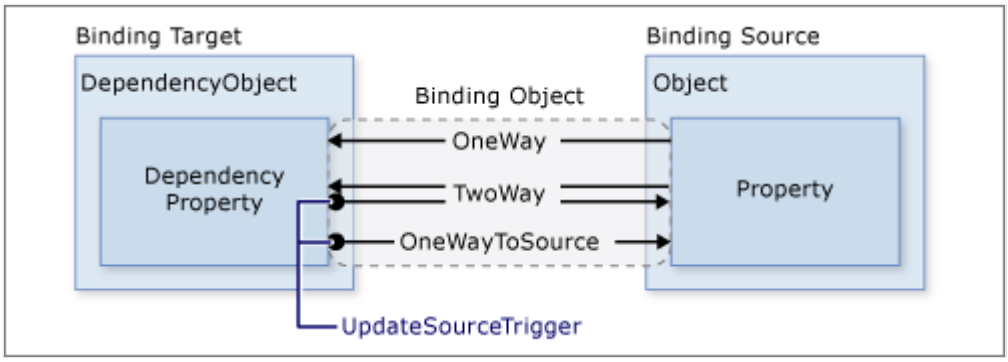
Aby wykryć zmiany źródła (dotyczy powiązań OneWay i TwoWay), źródło musi wdrożyć odpowiedni mechanizm powiadamiania o zmianie właściwości, taki jak INotifyPropertyChanged. Zobacz Instrukcje: implementowanie powiadomienia o zmianie właściwości (.NET Framework), aby zapoznać się z przykładem implementacji INotifyPropertyChanged.

Właściwość Binding.Mode zawiera więcej informacji na temat trybów powiązań i przykład sposobu określania kierunku powiązania.

Co wyzwala aktualizacje źródła

Powiązania, które są TwoWay lub OneWayToSource nasłuchują zmian we właściwości docelowej i propagują je z powrotem do źródła, co jest znane jako aktualizowanie źródła. Na przykład możesz edytować tekst pola tekstowego, aby zmienić podstawową wartość źródłową.

Czy jednak wartość źródłowa jest aktualizowana podczas edytowania tekstu, czy też po zakończeniu edytowania tekstu, a kontrolka traci fokus? Właściwość Binding.UpdateSourceTrigger określa, co wyzwala aktualizację źródła. Kropki strzałki w prawo na poniższej ilustracji ilustrują rolę właściwości Binding.UpdateSourceTrigger.



Jeśli wartość `UpdateSourceTrigger` to UpdateSourceTrigger.PropertyChanged, wartość wskazywana przez strzałkę w prawo elementu TwoWay lub powiązania OneWayToSource jest aktualizowana zaraz po zmianie właściwości docelowej. Jeśli jednak wartość `UpdateSourceTrigger` to LostFocus, ta wartość jest aktualizowana tylko przy użyciu nowej wartości, gdy właściwość docelowa traci fokus.

Podobnie jak w przypadku właściwości Mode, różne właściwości zależności mają różne wartości domyślne UpdateSourceTrigger. Wartość domyślna dla większości właściwości zależności to PropertyChanged, co powoduje natychmiastową zmianę wartości właściwości źródłowej po zmianie wartości właściwości docelowej. Natychmiastowe zmiany są odpowiednie dla CheckBox i innych prostych kontrolek. Jednak w przypadku pól tekstowych aktualizacja po każdym naciśnięciu klawiszy może zmniejszyć wydajność i uniemożliwić użytkownikowi zwykłą możliwość cofnięcia się i naprawienia błędów pisowni przed zatwierdzeniem nowej wartości. Na przykład właściwość `TextBox.Text` jest domyślnie ustawiona na wartość `UpdateSourceTrigger` równą LostFocus, co powoduje zmianę wartości źródłowej tylko, gdy element kontrolki traci fokus, a nie po zmianie właściwości `TextBox.Text`. Zobacz stronę właściwości UpdateSourceTrigger, aby uzyskać informacje na temat sposobu znajdowania wartości domyślnej właściwości zależności.

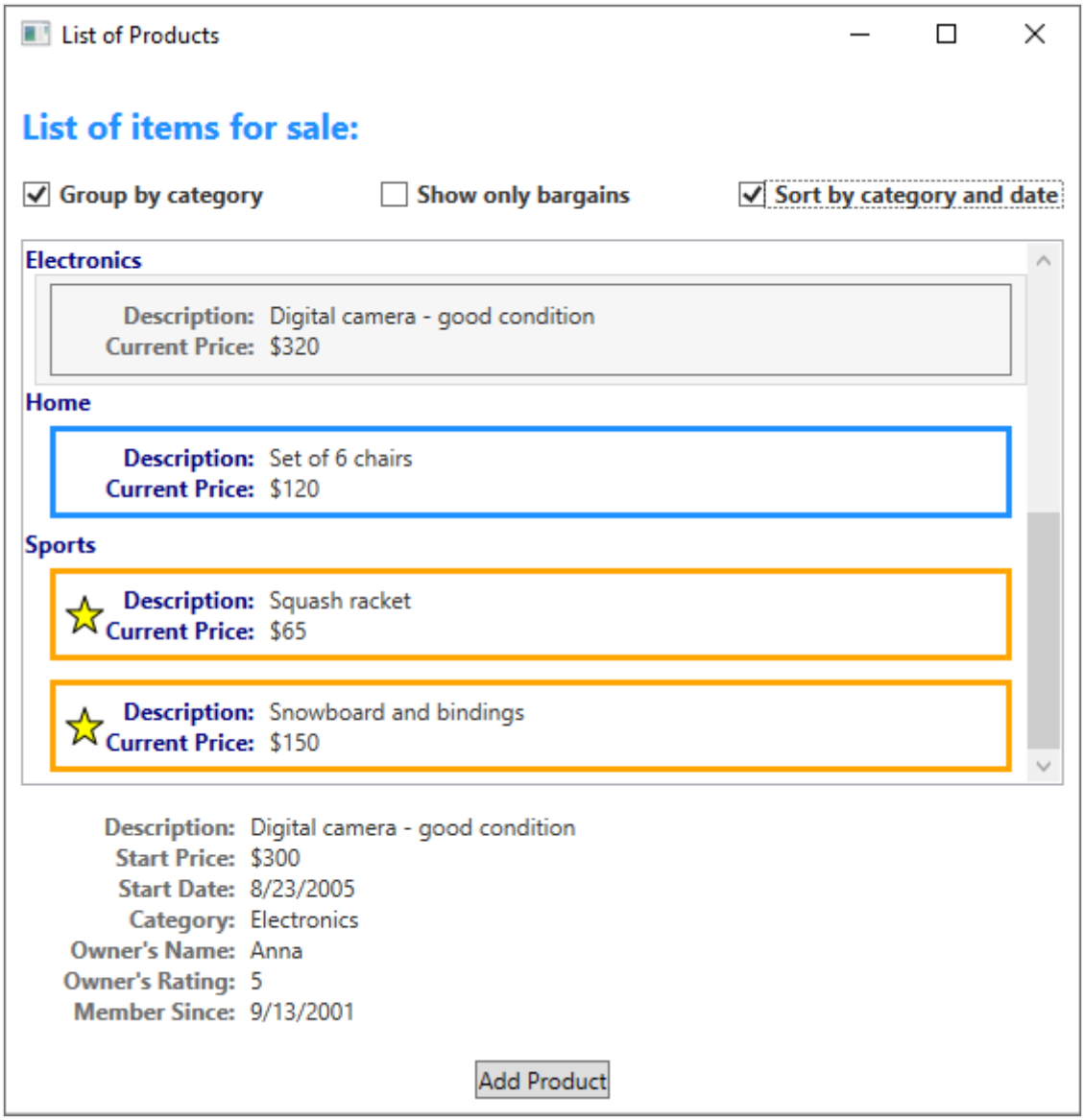
W poniższej tabeli przedstawiono przykładowy scenariusz dla każdej wartości UpdateSourceTrigger przy użyciu elementu TextBox jako przykładu.

Wartość UpdateSourceTrigger	Po zaktualizowaniu wartości źródłowej	Przykładowy scenariusz dla pola tekstowego
`LostFocus` (wartość domyślna dla TextBox.Text)	Gdy kontrolka TextBox traci fokus.	Pole tekstowe skojarzone z logiką weryfikacji (patrz Sprawdzanie poprawności danych poniżej).
`PropertyChanged`	Podczas wpisywania do elementu TextBox.	Kontrolki TextBox w oknie pokoju rozmów.
`Explicit`	Gdy aplikacja wywołuje metodę UpdateSource.	Kontrolki TextBox w formularzu edytowalnym (aktualizuje wartości źródłowe tylko, gdy użytkownik naciśnie przycisk przesyłania).

Na przykład zobacz Jak kontrolować, kiedy tekst pola tekstowego aktualizuje źródło (.NET Framework).

Przykład powiązania danych

Aby zapoznać się z przykładem powiązania danych, przyjrzyj się następującemu interfejsowi użytkownika aplikacji z Data Binding Demo, który wyświetla listę przedmiotów aukcyjnych.



Aplikacja demonstruje następujące funkcje powiązania danych:

- Zawartość obiektu ListBox jest powiązana z kolekcją obiektów *AuctionItem*. Obiekt *AuctionItem* ma właściwości, takie jak *Description*, *StartPrice*, *StartDate*, *Category* i *SpecialFeatures*.

- Dane (obiektów *AuctionItem*) wyświetlane w obiekcie `ListBox` są dopasowywane do szablonu, aby opis i bieżąca cena zostały wyświetlone dla każdego elementu. Szablon jest tworzony przy użyciu elementu DataTemplate. Ponadto wygląd każdego elementu zależy od wartości *SpecialFeatures* wyświetlanego elementu *AuctionItem*. Jeśli wartość *SpecialFeatures* elementu *AuctionItem* to *Color*, element ma niebieskie obramowanie. Jeśli wartość to *Highlight*, element ma pomarańczowe obramowanie i gwiazdkę. Sekcja Szablony danych zawiera informacje o tworzeniu szablonów danych.

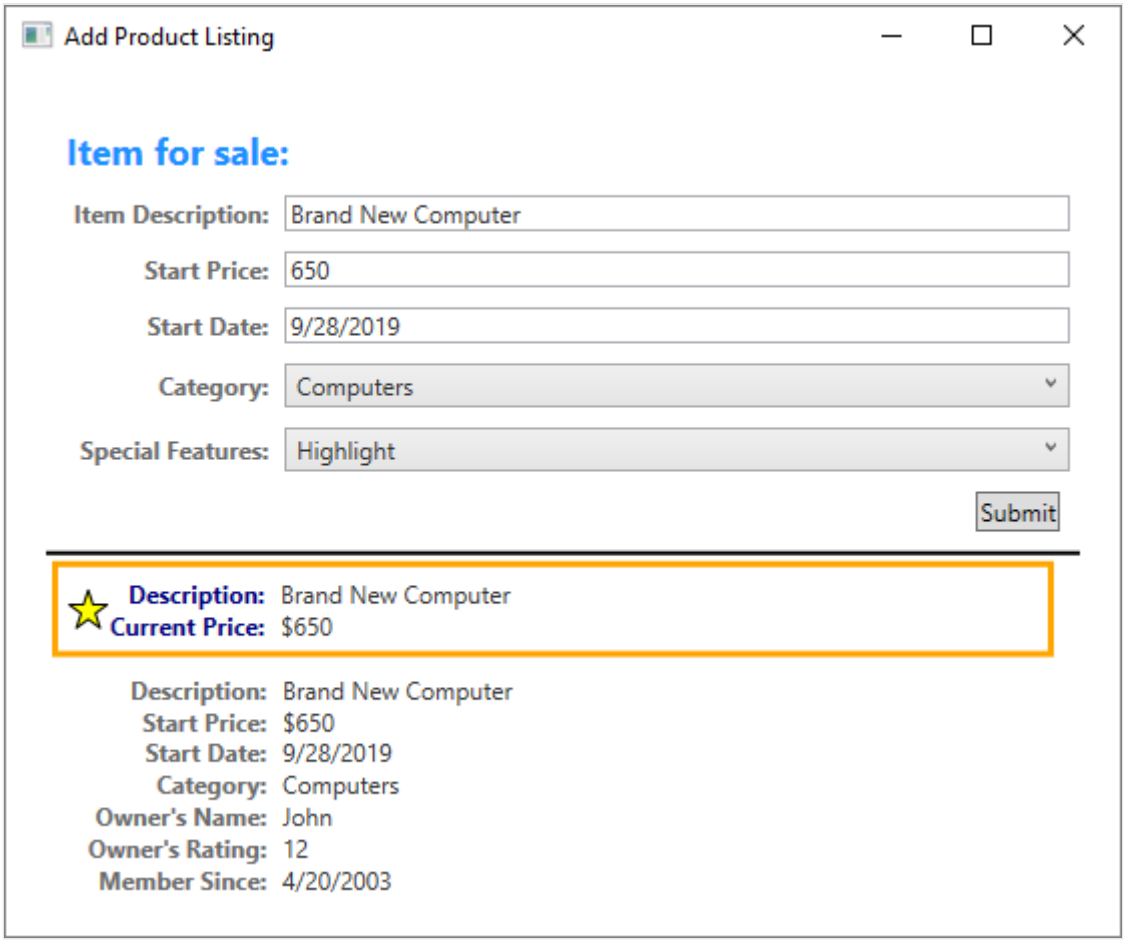
- Użytkownik może grupować, filtrować lub sortować dane przy użyciu podanego elementu `CheckBoxes`. Na powyższej ilustracji wybrano pozycję **Grupuj według kategorii** i **Sortuj według kategorii i daty** `CheckBoxes`. Być może zauważysz, że

dane są pogrupowane na podstawie kategorii produktu, a nazwa kategorii jest w kolejności alfabetycznej. Trudno zauważyć na obrazie, ale elementy są również sortowane według daty rozpoczęcia w ramach każdej kategorii. Sortowanie odbywa się przy użyciu *widoku kolekcji*. W sekcji Wiązanie z kolekcjami omówiono widoki kolekcji.

- Gdy użytkownik wybierze element, ContentControl wyświetli szczegóły wybranego elementu. To środowisko jest nazywane *scenariuszem główny-szczegółowy*. Sekcja scenariusza główny-szczegółowy zawiera informacje o tym typie powiązania.

- Typ właściwości *StartDate* to DateTime, co zwraca datę zawierającą godzinę do milisekund. W tej aplikacji użyto niestandardowego konwertera, aby był wyświetlany krótszy ciąg daty. Sekcja Konwersja danych zawiera informacje o konwerterach.

Gdy użytkownik wybierze przycisk *Dodaj produkt*, zostanie wyświetlony następujący formularz.



Użytkownik może edytować pola w formularzu, wyświetlić podgląd oferty produktów przy użyciu krótkich lub szczegółowych okienek podglądu i wybrać `Submit`, aby dodać nową listę produktów. Wszystkie istniejące ustawienia grupowania, filtrowania i sortowania będą stosowane do nowego wpisu. W tym konkretnym przypadku element

wprowadzony na powyższym obrazie będzie wyświetlany jako drugi element w kategorii *Komputer*.

Na tej ilustracji nie pokazano logiki weryfikacji podanej w *dacie rozpoczęcia*TextBox. Jeśli użytkownik wprowadzi nieprawidłową datę (nieprawidłowe formatowanie lub wcześniejszą datę), zostanie on powiadomiony za pomocą ToolTip i czerwonego wykrzyknika obok TextBox. W sekcji Walidacja danych omówiono sposób tworzenia logiki walidacji.

Przed przejściem do różnych funkcji powiązania danych opisanych powyżej najpierw omówimy podstawowe pojęcia, które mają kluczowe znaczenie dla zrozumienia powiązania danych WPF.

Tworzenie powiązania

Aby odtworzyć niektóre pojęcia omówione w poprzednich sekcjach, należy ustanowić powiązanie przy użyciu obiektu Binding, a każde powiązanie ma zwykle cztery składniki: obiekt docelowy powiązania, właściwość docelową, źródło powiązania i ścieżkę do wartości źródłowej do użycia. W tej sekcji omówiono sposób konfigurowania powiązania.

Źródła powiązań są powiązane z aktywnym DataContext dla elementu. Elementy automatycznie dziedziczą `DataContext`, jeśli nie zostały jawnie zdefiniowane.

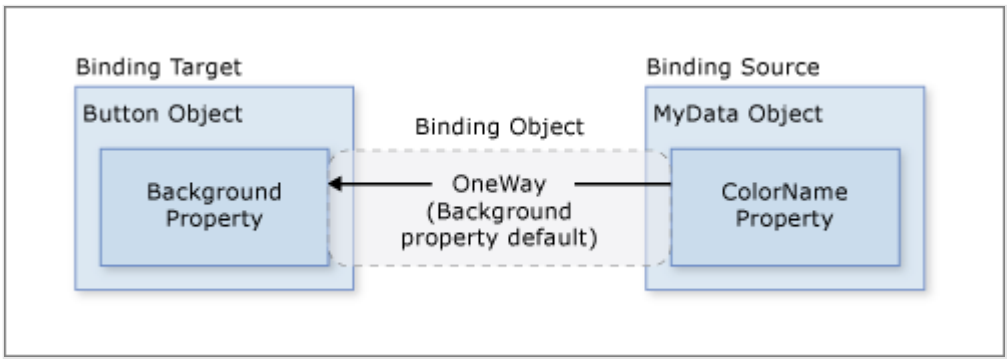
Rozważmy poniższy przykład, w którym obiekt źródłowy powiązania jest klasą o nazwie *MyData* zdefiniowaną w przestrzeni nazw *SDKSample*. W celach demonstracyjnych klasa *MyData* ma właściwość ciągu o nazwie *ColorName*, której wartość jest ustawiona na „Czerwony". W efekcie ten przykład generuje przycisk z czerwonym tłem.

XAML

```
<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:c="clr-namespace:SDKSample">
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSource"/>
    </DockPanel.Resources>
    <DockPanel.DataContext>
        <Binding Source="{StaticResource myDataSource}"/>
    </DockPanel.DataContext>
    <Button Background="{Binding Path=ColorName}"
            Width="150" Height="30">
        I am bound to be RED!
    </Button>
</DockPanel>
```

Aby uzyskać więcej informacji na temat składni deklaracji powiązania i przykładów sposobu konfigurowania powiązania w kodzie, zobacz Omówienie deklaracji powiązań.

Jeśli zastosujemy ten przykład do naszego podstawowego diagramu, wynikowy rysunek będzie wyglądać podobnie do poniższego. Na tym rysunku opisano powiązanie OneWay, ponieważ właściwość Background domyślnie obsługuje powiązanie OneWay.



Możesz się zastanawiać, dlaczego to powiązanie działa, mimo że właściwość *ColorName* jest typu ciąg, podczas gdy właściwość Background jest typu Brush. To powiązanie używa domyślnej konwersji typów, która została omówiona w sekcji Konwersja danych.

Określanie źródła powiązania

Zwróć uwagę, że w poprzednim przykładzie źródło powiązania jest określane przez ustawienie właściwości DockPanel.DataContext. Następnie Button dziedziczy wartość DataContext z DockPanel elementu, który jest jego elementem nadrzędnym. Aby powtórzyć, obiekt źródłowy powiązania jest jednym z czterech niezbędnych składników powiązania. Dlatego bez określonego obiektu źródłowego powiązania powiązanie nie zrobi nic.

Istnieje kilka sposobów określania obiektu źródłowego powiązania. Użycie właściwości DataContext do elementu nadrzędnego jest przydatne w przypadku wiązania wielu właściwości z tym samym źródłem. Czasami jednak bardziej odpowiednie może być określenie źródła powiązania dla poszczególnych deklaracji powiązań. W poprzednim przykładzie zamiast używać właściwości DataContext, można określić źródło powiązania, ustawiając właściwość Binding.Source bezpośrednio w deklaracji powiązania przycisku, jak w poniższym przykładzie.

XAML

```
<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
           xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
           xmlns:c="clr-namespace:SDKSample">
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSource"/>
    </DockPanel.Resources>
```

```
    <Button Background="{Binding Source={StaticResource myDataSource},
Path=ColorName}"
            Width="150" Height="30">
        I am bound to be RED!
    </Button>
</DockPanel>
```

Poza ustawieniem właściwości DataContext elementu bezpośrednio dziedziczenie wartości DataContext z obiektu przodka (na przykład przycisku w pierwszym przykładzie) i jawne określenie źródła powiązania przez ustawienie właściwości Binding.Source powiązania (na przykład przycisku ostatniego przykładu), możesz również użyć właściwości Binding.ElementName lub właściwości Binding.RelativeSource w celu określenia źródła powiązania. Właściwość ElementName jest przydatna podczas tworzenia powiązań z innymi elementami w aplikacji, takimi jak użycie suwaka w celu dostosowania szerokości przycisku. Właściwość RelativeSource jest przydatna, gdy powiązanie jest określone w obiekcie ControlTemplate lub Style. Aby uzyskać więcej informacji, zobacz Omówienie źródeł powiązań.

Określanie ścieżki do wartości

Jeśli źródło powiązania jest obiektem, należy użyć właściwości Binding.Path, aby określić wartość do użycia dla powiązania. Jeśli tworzysz powiązanie z danymi XML, użyj właściwości Binding.XPath, aby określić wartość. W niektórych przypadkach może to mieć zastosowanie do używania właściwości Path nawet, gdy dane są danymi XML. Jeśli na przykład chcesz uzyskać dostęp do właściwości Name zwróconego elementu XmlNode (w wyniku zapytania XPath), należy użyć właściwości Path oprócz właściwości XPath.

Aby uzyskać więcej informacji, zobacz właściwości Path i XPath.

Mimo że podkreśliliśmy, że wartość Path do użycia jest jednym z czterech niezbędnych składników powiązania, w scenariuszach, które chcesz powiązać z całym obiektem, wartość do użycia będzie taka sama jak obiekt źródłowy powiązania. W takich przypadkach ma zastosowanie do braku określenia elementu Path. Rozważmy następujący przykład.

XAML

```
<ListBox ItemsSource="{Binding}"
         IsSynchronizedWithCurrentItem="true"/>
```

W powyższym przykładzie użyto pustej składni powiązania: {Binding}. W tym przypadku element ListBox dziedziczy DataContext z nadrzędnego elementu DockPanel (nie

pokazano w tym przykładzie). Jeśli ścieżka nie zostanie określona, wartością domyślną jest powiązanie z całym obiektem. Innymi słowy, w tym przykładzie ścieżka została pominięta, ponieważ wiążemy właściwość ItemsSource z całym obiektem. (Zobacz sekcję Wiązanie z kolekcjami, aby zapoznać się ze szczegółową dyskusją).

Poza powiązaniem z kolekcją ten scenariusz jest również przydatny, gdy chcesz powiązać z całym obiektem zamiast, a nie tylko z jedną właściwością obiektu. Jeśli na przykład obiekt źródłowy jest typu String, możesz po prostu chcieć powiązać z samym ciągiem. Innym typowym scenariuszem jest powiązanie elementu z obiektem z kilkoma właściwościami.

Może być konieczne zastosowanie logiki niestandardowej, aby dane miały znaczenie dla powiązanej właściwości docelowej. Logika niestandardowa może mieć postać konwertera niestandardowego, jeśli domyślna konwersja typu nie istnieje. Zobacz Konwersja danych, aby uzyskać informacje o konwerterach.

Wiązanie i BindingExpression

Przed rozpoczęciem pracy z innymi funkcjami i użyciem powiązania danych warto wprowadzić klasę BindingExpression. Jak pokazano w poprzednich sekcjach, klasa Binding jest klasą wysokiego poziomu dla deklaracji powiązania. Zawiera ona wiele właściwości, które umożliwiają określenie cech powiązania. Powiązana klasa, BindingExpression, jest obiektem bazowym, który utrzymuje połączenie między źródłem a obiektem docelowym. Powiązanie zawiera wszystkie informacje, które mogą być współużytkowane przez kilka wyrażeń powiązań. BindingExpression jest wyrażeniem wystąpienia, którego nie można udostępnić i zawiera wszystkie informacje o wystąpieniu obiektu Binding.

Rozważmy następujący przykład, gdzie `myDataObject` jest wystąpieniem klasy `MyData`, `myBinding` jest obiektem źródłowym Binding i `MyData` jest zdefiniowaną klasą zawierającą właściwość ciągu o nazwie `ColorName`. W tym przykładzie zawartość tekstowa klasy `myText`, wystąpienie klasy TextBlock są powiązane z elementem `ColorName`.

C#

```
// Make a new source
var myDataObject = new MyData();
var myBinding = new Binding("ColorName")
{
    Source = myDataObject
};
```

```
// Bind the data source to the TextBox control's Text dependency property
myText.SetBinding(TextBlock.TextProperty, myBinding);
```

Do utworzenia innych powiązań można użyć tego samego obiektu *myBinding*. Na przykład można użyć obiektu *myBinding*, aby powiązać zawartość tekstową pola wyboru z właściwością *colorName*. W tym scenariuszu będą istnieć dwa wystąpienia BindingExpression współużytkujące obiekt *myBinding*.

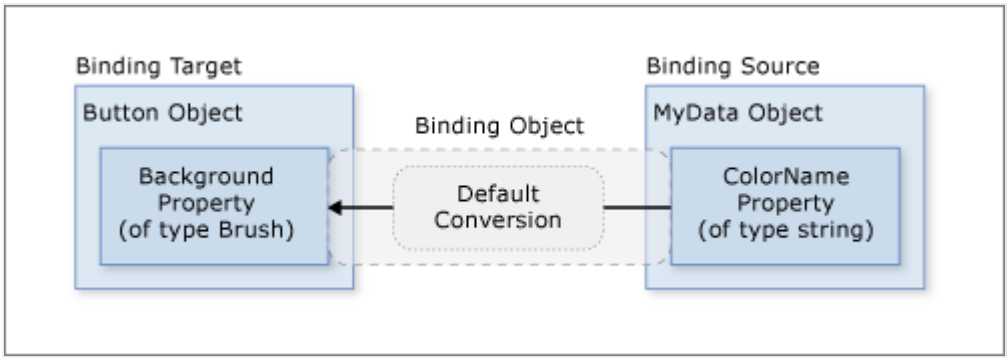
Obiekt BindingExpression jest zwracany przez wywołanie GetBindingExpression obiektu powiązanego z danymi. W poniższych artykułach przedstawiono niektóre użycia klasy BindingExpression:

- Pobieranie obiektu wiążącego z powiązanej własności docelowej (.NET Framework)
- Kontrolowanie momentu aktualizowania źródła przez tekst kontrolki TextBox (.NET Framework)

Konwersja danych

W sekcji Tworzenie powiązania przycisk jest czerwony, ponieważ jego właściwość Background jest powiązana z właściwością ciągu o wartości „Czerwony". Ta wartość ciągu działa, ponieważ konwerter typów jest obecny w typie Brush, aby przekonwertować wartość ciągu na wartość Brush.

Dodanie tych informacji do rysunku w sekcji Tworzenie powiązania wygląda następująco.



Co jednak zrobić, jeśli zamiast właściwości typu ciąg obiekt źródłowy powiązania ma właściwość *Color* typu Color? W takim przypadku, aby powiązanie działało, należy najpierw przekształcić wartość właściwości *Color* w coś, co akceptuje właściwość Background. Należy utworzyć konwerter niestandardowy, implementując interfejs IValueConverter, jak w poniższym przykładzie.

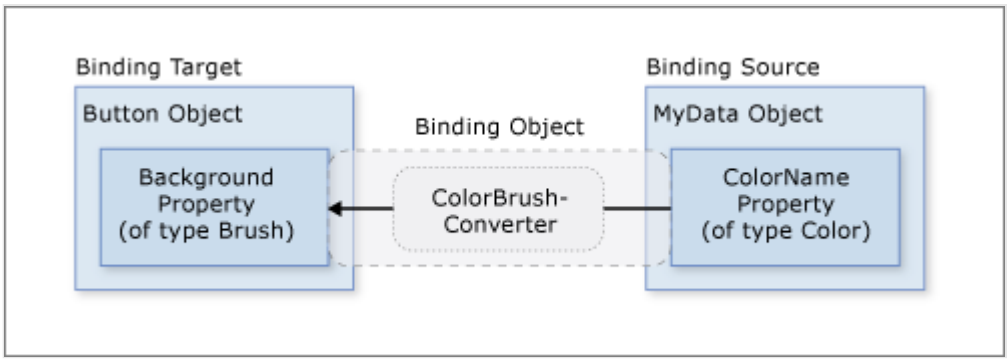
C#

```
[ValueConversion(typeof(Color), typeof(SolidColorBrush))]
public class ColorBrushConverter : IValueConverter
```

```
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        Color color = (Color)value;
        return new SolidColorBrush(color);
    }

    public object ConvertBack(object value, Type targetType, object
parameter, System.Globalization.CultureInfo culture)
    {
        return null;
    }
}
```

Aby uzyskać więcej informacji, zobacz IValueConverter.

Teraz niestandardowy konwerter jest używany zamiast konwersji domyślnej, a nasz diagram wygląda następująco.



Aby powtórzyć, konwersje domyślne mogą być dostępne ze względu na konwertery typów, które są obecne w typie wiązanym. To zachowanie będzie zależeć od tego, które konwertery typów są dostępne w obiekcie docelowym. W razie wątpliwości utwórz własny konwerter.

Poniżej przedstawiono kilka typowych scenariuszy, w których warto zaimplementować konwerter danych:

- Dane powinny być wyświetlane inaczej w zależności od kultury. Na przykład można zaimplementować konwerter walut lub konwerter daty/godziny kalendarza na podstawie konwencji używanych w określonej kulturze.

- Używane dane niekoniecznie są przeznaczone do zmiany wartości tekstowej właściwości, ale zamiast tego mają na celu zmianę innej wartości, takiej jak źródło obrazu, kolor lub styl tekstu wyświetlanego. Konwertery mogą być używane w tym wystąpieniu poprzez przekonwertowanie powiązania właściwości, która może nie być odpowiednia, na przykład powiązanie pola tekstowego z właściwością Background komórki tabeli.

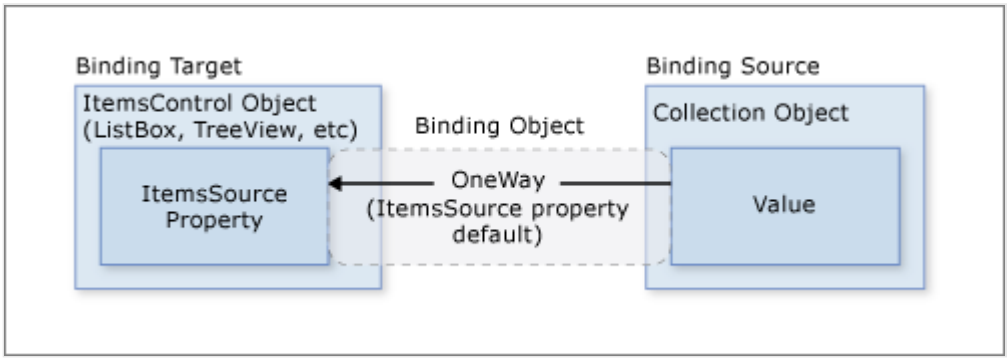
- Do tych samych danych jest powiązanych więcej niż jedna kontrolka lub wiele właściwości kontrolek. W takim przypadku powiązanie podstawowe może po prostu wyświetlać tekst, podczas gdy inne powiązania obsługują określone problemy z wyświetlaniem, ale nadal używają tego samego powiązania, co informacje źródłowe.

- Właściwość docelowa zawiera kolekcję powiązań, która jest określana jako MultiBinding. W przypadku elementu MultiBinding należy użyć niestandardowego elementu IMultiValueConverter, aby wygenerować ostateczną wartość z wartości powiązań. Na przykład kolor może być obliczany z czerwonych, niebieskich i zielonych wartości, które mogą być wartościami z tych samych lub różnych obiektów źródłowych powiązania. Zobacz MultiBinding, aby poznać przykłady i informacje.

Powiązanie z kolekcjami

Obiekt źródłowy powiązania może być traktowany jako pojedynczy obiekt, którego właściwości zawierają dane, lub jako kolekcja danych obiektów polimorficznych, które są często grupowane razem (na przykład wynik zapytania do bazy danych). Do tej pory omówiliśmy tylko powiązanie z pojedynczymi obiektami. Jednak powiązanie z kolekcją danych jest typowym scenariuszem. Na przykład typowym scenariuszem jest użycie elementu ItemsControl, takiego jak ListBox, ListView lub TreeView w celu wyświetlenia kolekcji danych, na przykład w aplikacji pokazanej w sekcji Co to jest powiązanie danych.

Na szczęście nasz podstawowy diagram nadal ma zastosowanie. Jeśli tworzysz powiązanie ItemsControl z kolekcją, diagram wygląda następująco.



Jak pokazano na tym diagramie, aby powiązać element ItemsControl z obiektem kolekcji, właściwość ItemsControl.ItemsSource jest właściwością do użycia. Możesz traktować `ItemsSource` jako zawartość elementu ItemsControl. Powiązanie to OneWay, ponieważ właściwość `ItemsSource` domyślnie obsługuje powiązanie `OneWay`.

Jak zaimplementować kolekcje

Możesz wyliczyć dowolną kolekcję, która implementuje interfejs IEnumerable. Aby jednak skonfigurować powiązania dynamiczne tak, aby wstawienie lub usunięcie w kolekcji automatycznie zaktualizowało interfejs użytkownika, kolekcja musi zaimplementować interfejs INotifyCollectionChanged. Ten interfejs uwidacznia zdarzenie, które powinno być zgłaszane za każdym razem, gdy podstawowa kolekcja ulegnie zmianie.

WPF udostępnia klasę ObservableCollection<T>, która jest wbudowaną implementacją kolekcji danych, która uwidacznia interfejs INotifyCollectionChanged. Aby w pełni obsługiwać przesyłanie wartości danych z obiektów źródłowych do obiektów docelowych, każdy obiekt w kolekcji obsługujący właściwości możliwe do powiązania musi również implementować interfejs INotifyPropertyChanged. Aby uzyskać więcej informacji, zobacz Omówienie źródeł powiązań.

Przed wdrożeniem własnej kolekcji rozważ użycie ObservableCollection<T> lub jednej z istniejących klas kolekcji, takich jak, między innymi, List<T>, Collection<T> i BindingList<T>. Jeśli masz zaawansowany scenariusz i chcesz zaimplementować własną kolekcję, rozważ użycie metody IList, która udostępnia inną niż ogólna kolekcję obiektów, do których można uzyskać dostęp indywidualnie przez indeks, a tym samym zapewnia najlepszą wydajność.

Widoki kolekcji

Po powiązaniu ItemsControl z kolekcją danych możesz sortować, filtrować lub grupować dane. W tym celu należy użyć widoków kolekcji, czyli klas implementujących interfejs ICollectionView.

Co to są widoki kolekcji?

Widok kolekcji jest warstwą powyżej kolekcji źródłowej powiązania, która umożliwia nawigowanie i wyświetlanie kolekcji źródłowej na podstawie zapytań sortowania, filtrowania i grupowania bez konieczności zmieniania samej bazowej kolekcji źródłowej. Widok kolekcji zachowuje również wskaźnik do bieżącego elementu w kolekcji. Jeśli kolekcja źródłowa implementuje interfejs INotifyCollectionChanged, zmiany zgłoszone przez zdarzenie CollectionChanged są propagowane do widoków.

Ponieważ widoki nie zmieniają bazowych kolekcji źródłowych, każda kolekcja źródłowa może mieć skojarzone wiele widoków. Na przykład może istnieć kolekcja obiektów *Task*. Za pomocą widoków można wyświetlać te same dane na różne sposoby. Na przykład po lewej stronie możesz wyświetlić zadania posortowane według priorytetu, a po prawej stronie — pogrupowane według obszaru.

Jak utworzyć widok

Jednym ze sposobów tworzenia i używania widoku jest utworzenie wystąpienia obiektu widoku bezpośrednio, a następnie użycie go jako źródła powiązania. Rozważmy na przykład Pokaz powiązania danych aplikację wyświetlaną w sekcji Co to jest powiązanie danych. Aplikacja jest implementowana tak, aby ListBox wiązała z widokiem kolekcji danych zamiast bezpośrednio z kolekcją danych. Poniższy przykład został wyodrębniony z aplikacji Pokaz powiązania danych. Klasa CollectionViewSource jest serwerem proxy XAML klasy dziedziczonej z klasy CollectionView. W tym konkretnym przykładzie element Source widoku jest powiązany z kolekcją *AuctionItems* (typu ObservableCollection<T>) bieżącego obiektu aplikacji.

XAML

```
<Window.Resources>
    <CollectionViewSource
      Source="{Binding Source={x:Static Application.Current},
Path=AuctionItems}"
      x:Key="listingDataView" />
</Window.Resources>
```

Zasób *listingDataView* następnie służy jako źródło powiązania dla elementów w aplikacji, takich jak ListBox.

XAML

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
         ItemsSource="{Binding Source={StaticResource listingDataView}}" />
```

Aby utworzyć inny widok dla tej samej kolekcji, możesz utworzyć inne wystąpienie CollectionViewSource i nadać mu inną nazwę `x:Key`.

W poniższej tabeli przedstawiono typy danych widoku tworzone jako domyślny widok kolekcji lub przez CollectionViewSource na podstawie typu kolekcji źródłowej.

Typ kolekcji źródłowej	Typ widoku kolekcji	Uwagi
IEnumerable	Typ wewnętrzny oparty na CollectionView	Nie można zgrupować elementów.
IList	ListCollectionView	Najszybszy.
IBindingList	BindingListCollectionView	

Korzystanie z widoku domyślnego

Określenie widoku kolekcji jako źródła powiązania jest jednym ze sposobów tworzenia i używania widoku kolekcji. WPF tworzy również domyślny widok kolekcji dla każdej kolekcji używanej jako źródło powiązania. Jeśli powiążesz bezpośrednio z kolekcją, WPF wiąże się z jej widokiem domyślnym. Ten widok domyślny jest współużytkowany przez wszystkie powiązania z tą samą kolekcją, więc zmiany wprowadzone w widoku domyślnym przez jedną powiązaną kontrolkę lub kod (takie jak sortowanie lub zmiana w bieżącym wskaźniku do elementu, które zostaną omówionego później) zostaną odzwierciedlone we wszystkich innych powiązaniach z tą samą kolekcją.

Aby uzyskać widok domyślny, użyj metody GetDefaultView. Aby zapoznać się z przykładem, zobacz Pobieranie domyślnego widoku kolekcji danych (.NET Framework).

Widoki kolekcji z tabelami ADO.NET DataTables

Aby zwiększyć wydajność, widoki kolekcji dla ADO.NET obiekty DataTable lub DataView delegują sortowanie i filtrowanie do obiektu DataView, co powoduje udostępnianie sortowania i filtrowania we wszystkich widokach kolekcji źródła danych. Aby umożliwić każdemu widokowi kolekcji sortowanie i filtrowanie niezależnie, zainicjuj każdy widok kolekcji własnym obiektem DataView.

Sortowanie

Jak wspomniano wcześniej, widoki mogą stosować kolejność sortowania do kolekcji. Ponieważ istnieje to w kolekcji bazowej, dane mogą mieć odpowiedni, nieodłączny porządek lub mogą go nie mieć. Widok nad kolekcją umożliwia narzucenie kolejności lub zmianę kolejności domyślnej na podstawie kryteriów porównania, które podajesz. Ponieważ jest to widok danych oparty na kliencie, typowym scenariuszem jest to, że użytkownik może chcieć sortować kolumny danych tabelarycznych zgodnie z wartością odpowiadającą kolumnie. Za pomocą widoków można ponownie zastosować sortowanie oparte na użytkownikach bez wprowadzania żadnych zmian w kolekcji źródłowej, a nawet konieczności ponownego wprowadzania zapytania dla zawartości kolekcji. Aby zapoznać się z przykładem, zobacz Sortowanie kolumny GridView po kliknięciu nagłówka (.NET Framework).

Poniższy przykład przedstawia logikę sortowania „Sortuj według kategorii i daty” CheckBox interfejsu użytkownika aplikacji w sekcji Co to jest powiązanie danych.

```
C#
```

```
private void AddSortCheckBox_Checked(object sender, RoutedEventArgs e)
{
    // Sort the items first by Category and then by StartDate
    listingDataView.SortDescriptions.Add(new SortDescription("Category",
ListSortDirection.Ascending));
    listingDataView.SortDescriptions.Add(new SortDescription("StartDate",
ListSortDirection.Ascending));
}
```

Filtrowanie

Widoki mogą również stosować filtr do kolekcji, tak aby widok pokazywał tylko określony podzbiór pełnej kolekcji. Możesz filtrować według warunku w danych. Na przykład, podobnie jak to jest wykonywane przez aplikację w sekcji Co to jest powiązanie danych, „Pokaż tylko okazje" CheckBox zawiera logikę filtrowania elementów, które kosztują 25 USD lub więcej. Poniższy kod jest wykonywany, aby ustawić właściwość *ShowOnlyBargainsFilter* jako procedurę obsługi zdarzeń Filter po wybraniu opcji CheckBox.

C#

```
private void AddFilteringCheckBox_Checked(object sender, RoutedEventArgs e)
{
    if (((CheckBox)sender).IsChecked == true)
        listingDataView.Filter += ListingDataView_Filter;
    else
        listingDataView.Filter -= ListingDataView_Filter;
}
```

Procedura obsługi zdarzeń *ShowOnlyBargainsFilter* ma następującą implementację.

C#

```
private void ListingDataView_Filter(object sender, FilterEventArgs e)
{
    // Start with everything excluded
    e.Accepted = false;

    // Only inlcude items with a price less than 25
    if (e.Item is AuctionItem product && product.CurrentPrice < 25)
        e.Accepted = true;
}
```

Jeśli używasz jednej z klas CollectionView bezpośrednio zamiast CollectionViewSource, użyj właściwości Filter, aby określić wywołanie zwrotne. Aby zapoznać się z przykładem,

zobacz Filtrowanie danych w widoku (.NET Framework).

Grupowanie

Z wyjątkiem klasy wewnętrznej, która wyświetla kolekcję IEnumerable, wszystkie widoki kolekcji obsługują *grupowanie*, co umożliwia użytkownikowi partycjonowanie kolekcji w widoku kolekcji na grupy logiczne. Grupy mogą być jawne, gdy użytkownik dostarcza listę grup, lub niejawne, gdy grupy są generowane dynamicznie w zależności od danych.

W poniższym przykładzie przedstawiono logikę „Grupuj według kategorii” CheckBox.

C#

```
// This groups the items in the view by the property "Category"
var groupDescription = new PropertyGroupDescription();
groupDescription.PropertyName = "Category";
listingDataView.GroupDescriptions.Add(groupDescription);
```

Aby uzyskać inny przykład grupowania, zobacz Grupowanie elementów w widoku listy, który implementuje widok siatki (.NET Framework).

Bieżące wskaźniki elementów

Widoki obsługują również pojęcie bieżącego elementu. Możesz nawigować po obiektach w widoku kolekcji. Podczas nawigowania przenosisz wskaźnik elementu, który umożliwia pobranie obiektu, który istnieje w tej określonej lokalizacji w kolekcji. Aby zapoznać się z przykładem, zobacz Nawigowanie po obiektach w widoku kolekcji danych (.NET Framework).

Ponieważ WPF wiąże się z kolekcją tylko przy użyciu widoku (określonego widoku albo widoku domyślnego kolekcji), wszystkie powiązania z kolekcjami mają bieżący wskaźnik elementu. W przypadku powiązania z widokiem ukośnik („/”) w wartości `Path` wyznacza bieżący element widoku. W poniższym przykładzie kontekst danych jest widokiem kolekcji. Pierwszy wiersz jest powiązany z kolekcją. Drugi wiersz wiąże się z bieżącym elementem w kolekcji. Trzeci wiersz wiąże się z właściwością `Description` bieżącego elementu w kolekcji.

XAML

```
<Button Content="{Binding }" />
<Button Content="{Binding Path=/}" />
<Button Content="{Binding Path=/Description}" />
```

Składnia ukośnika i właściwości może być również układana w stos w celu przechodzenia przez hierarchię kolekcji. Poniższy przykład wiąże się z bieżącym elementem kolekcji o nazwie `Offices`, który jest właściwością bieżącego elementu kolekcji źródłowej.

XAML

```
<Button Content="{Binding /Offices/}" />
```

Na bieżący wskaźnik elementu może mieć wpływ dowolne sortowanie lub filtrowanie stosowane do kolekcji. Sortowanie zachowuje bieżący wskaźnik elementu na wybranym ostatnim elemencie, ale struktura widoku kolekcji jest teraz ustawiana wokół niego. (Być może wybrany element był na początku listy wcześniej, ale teraz wybrany element może być gdzieś w środku). Filtrowanie zachowuje wybrany element, jeśli ten wybór pozostaje w widoku po filtrowaniu. W przeciwnym razie bieżący wskaźnik elementu jest ustawiony na pierwszy element przefiltrowanego widoku kolekcji.

Scenariusz powiązania główny-szczegółowy

Pojęcie bieżącego elementu jest przydatne nie tylko w przypadku nawigacji po elementach w kolekcji, ale także dla scenariusza powiązania główny-szczegółowy. Ponownie rozważ interfejs użytkownika aplikacji w sekcji Co to jest powiązanie danych. W tej aplikacji wybór w elemencie ListBox określa zawartość wyświetlaną w elemencie ContentControl. Mówiąc inaczej, po wybraniu elementu ListBox element ContentControl wyświetla szczegóły wybranego elementu.

Scenariusz główny-szczegółowy można zaimplementować po prostu używając co najmniej dwóch kontrolek powiązanych z tym samym widokiem. Poniższy przykład z Pokaz powiązania danych pokazuje znaczniki elementu ListBox oraz elementu ContentControl widoczne w interfejsie użytkownika aplikacji w sekcji Co to jest powiązanie danych.

XAML

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
         ItemsSource="{Binding Source={StaticResource listingDataView}}" />
<ContentControl Name="Detail" Grid.Row="3" Grid.ColumnSpan="3"
                Content="{Binding Source={StaticResource listingDataView}}"
                ContentTemplate="{StaticResource
detailsProductListingTemplate}"
                Margin="9,0,0,0"/>
```

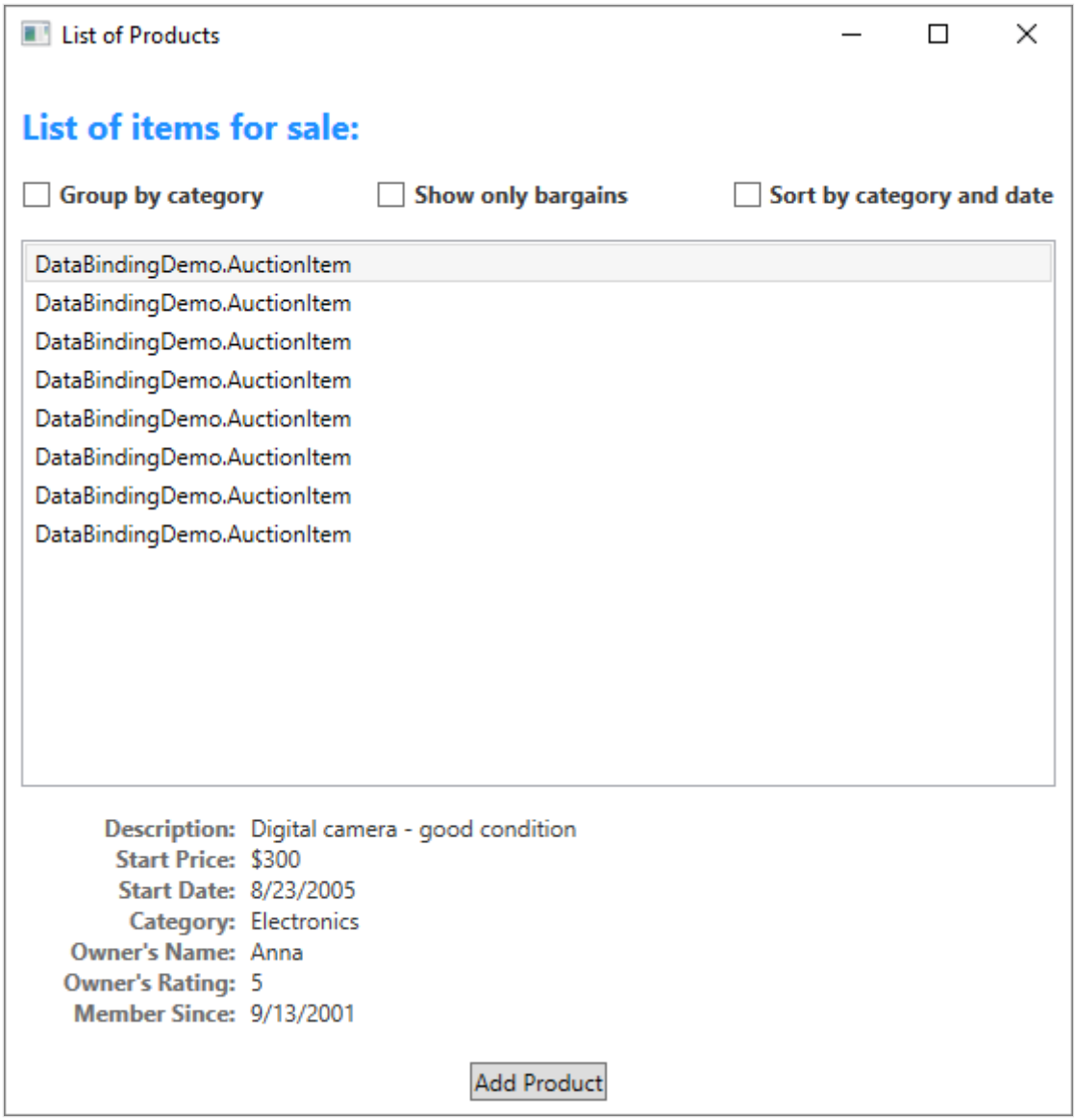
Zwróć uwagę, że obie kontrolki są powiązane z tym samym źródłem — zasobem statycznym *listDataView* (zobacz definicję tego zasobu w sekcji Jak utworzyć widok). To powiązanie działa, ponieważ, gdy pojedynczy obiekt (w tym przypadku ContentControl) jest powiązany z widokiem kolekcji, automatycznie ono wiąże się z CurrentItem widoku. Obiekty CollectionViewSource są automatycznie synchronizowane z walutą i zaznaczeniem. Jeśli kontrolka listy nie jest powiązana z obiektem CollectionViewSource, jak w tym przykładzie, należy ustawić jej właściwość IsSynchronizedWithCurrentItem na wartość `true`, aby to działało.

Aby zapoznać się z innymi przykładami, zobacz Powiązanie z kolekcją i wyświetlanie informacji na podstawie zaznaczenia (.NET Framework) i Używanie wzorca główny-szczegółowy z danymi hierarchicznymi (.NET Framework).

Można zauważyć, że w powyższym przykładzie użyto szablonu. W rzeczywistości dane nie będą wyświetlane w sposób, w jaki chcemy, bez użycia szablonów (ten jest jawnie używany przez element ContentControl i używany niejawnie przez element ListBox). Teraz przejdźmy do tworzenia szablonów danych w następnej sekcji.

Tworzenie szablonów danych

Bez użycia szablonów danych nasz interfejs użytkownika aplikacji w sekcji Przykład powiązania danych wyglądałby następująco:



Jak pokazano w przykładzie w poprzedniej sekcji, zarówno kontrolka ListBox, jak i obiekt ContentControl są powiązane z całym obiektem kolekcji (lub dokładniej widokiem na obiekt kolekcji) elementu *AuctionItem*. Bez konkretnych instrukcji wyświetlania kolekcji danych ListBox wyświetla reprezentację ciągu każdego obiektu w kolekcji bazowej, a ContentControl wyświetla reprezentację ciągu obiektu, z którym jest powiązany.

Aby rozwiązać ten problem, aplikacja definiuje elementy DataTemplates. Jak pokazano w przykładzie w poprzedniej sekcji, ContentControl jawnie używa szablonu danych *DetailsProductListingTemplate*. Kontrolka ListBox niejawnie używa następującego szablonu danych podczas wyświetlania obiektów *AuctionItem* w kolekcji.

XAML

```
<DataTemplate DataType="{x:Type src:AuctionItem}">
    <Border BorderThickness="1" BorderBrush="Gray"
            Padding="7" Name="border" Margin="3" Width="500">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
```

```
                <RowDefinition/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="20"/>
                <ColumnDefinition Width="86"/>
                <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>

            <Polygon Grid.Row="0" Grid.Column="0" Grid.RowSpan="4"
                     Fill="Yellow" Stroke="Black" StrokeThickness="1"
                     StrokeLineJoin="Round" Width="20" Height="20"
                     Stretch="Fill"
                     Points="9,2 11,7 17,7 12,10 14,15 9,12 4,15 6,10 1,7
7,7"
                     Visibility="Hidden" Name="star"/>

            <TextBlock Grid.Row="0" Grid.Column="1" Margin="0,0,8,0"
                       Name="descriptionTitle"
                       Style="{StaticResource smallTitleStyle}">Description:
</TextBlock>

            <TextBlock Name="DescriptionDTDataType" Grid.Row="0"
Grid.Column="2"
                       Text="{Binding Path=Description}"
                       Style="{StaticResource textStyleTextBlock}"/>

            <TextBlock Grid.Row="1" Grid.Column="1" Margin="0,0,8,0"
                       Name="currentPriceTitle"
                       Style="{StaticResource smallTitleStyle}">Current
Price:</TextBlock>

            <StackPanel Grid.Row="1" Grid.Column="2"
Orientation="Horizontal">
                <TextBlock Text="$" Style="{StaticResource
textStyleTextBlock}"/>
                <TextBlock Name="CurrentPriceDTDataType"
                           Text="{Binding Path=CurrentPrice}"
                           Style="{StaticResource textStyleTextBlock}"/>
            </StackPanel>
        </Grid>
    </Border>
    <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding Path=SpecialFeatures}">
            <DataTrigger.Value>
                <src:SpecialFeatures>Color</src:SpecialFeatures>
            </DataTrigger.Value>
            <DataTrigger.Setters>
                <Setter Property="BorderBrush" Value="DodgerBlue"
TargetName="border" />
                <Setter Property="Foreground" Value="Navy"
TargetName="descriptionTitle" />
                <Setter Property="Foreground" Value="Navy"
TargetName="currentPriceTitle" />
                <Setter Property="BorderThickness" Value="3"
```

```
TargetName="border" />
                <Setter Property="Padding" Value="5" TargetName="border" />
            </DataTrigger.Setters>
        </DataTrigger>
        <DataTrigger Binding="{Binding Path=SpecialFeatures}">
            <DataTrigger.Value>
                <src:SpecialFeatures>Highlight</src:SpecialFeatures>
            </DataTrigger.Value>
            <Setter Property="BorderBrush" Value="Orange"
TargetName="border" />
            <Setter Property="Foreground" Value="Navy"
TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy"
TargetName="currentPriceTitle" />
            <Setter Property="Visibility" Value="Visible" TargetName="star"
/>
            <Setter Property="BorderThickness" Value="3" TargetName="border"
/>
            <Setter Property="Padding" Value="5" TargetName="border" />
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>
```

Korzystając z tych dwóch elementów DataTemplates, wynikowy interfejs użytkownika wygląda tak, jak wyświetlany w sekcji Co to jest powiązanie danych. Jak widać na tym zrzucie ekranu, oprócz umożliwienia umieszczania danych w kontrolkach, element DataTemplates umożliwia definiowanie atrakcyjnych wizualizacji dla danych. Na przykład elementy DataTrigger są używane w powyższym elemencie DataTemplate, aby elementy *AuctionItem* o wartości *SpecialFeatures* równej *HighLight* były wyświetlane z pomarańczowym obramowaniem i gwiazdką.

Aby uzyskać więcej informacji na temat szablonów danych, zobacz Omówienie tworzenia szablonów danych (.NET Framework).

Walidacja danych

Większość aplikacji, które przyjmują dane wejściowe użytkownika, musi mieć logikę walidacji, aby upewnić się, że użytkownik wprowadził oczekiwane informacje. Sprawdzanie poprawności może być oparte na typie, zakresie, formacie lub innych wymaganiach specyficznych dla aplikacji. W tej sekcji omówiono sposób działania walidacji danych w programie WPF.

Kojarzenie reguł walidacji z powiązaniem

Model powiązania danych WPF umożliwia skojarzenie ValidationRules z obiektem Binding. Na przykład poniższy przykład wiąże obiekt TextBox z właściwością o nazwie

`StartPrice` i dodaje obiekt ExceptionValidationRule do właściwości Binding.ValidationRules.

XAML

```
<TextBox Name="StartPriceEntryForm" Grid.Row="2"
         Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5"
Grid.ColumnSpan="2">
    <TextBox.Text>
        <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">
            <Binding.ValidationRules>
                <ExceptionValidationRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Obiekt ValidationRule sprawdza, czy wartość właściwości jest prawidłowa. WPF ma dwa typy wbudowanych obiektów ValidationRule:

- Obiekt ExceptionValidationRule sprawdza wyjątki zgłoszone podczas aktualizacji właściwości źródła powiązania. W poprzednim przykładzie `StartPrice` jest typu liczby całkowitej. Gdy użytkownik wprowadzi wartość, której nie można przekonwertować na liczbę całkowitą, zgłaszany jest wyjątek, powodując oznaczenie powiązania jako nieprawidłowego. Alternatywna składnia jawnego ustawiania ExceptionValidationRule polega na ustawieniu właściwości ValidatesOnExceptions na wartość `true` w obiekcie Binding lub MultiBinding.

- Obiekt DataErrorValidationRule sprawdza błędy zgłaszane przez obiekty implementujące interfejs IDataErrorInfo. Aby uzyskać więcej informacji na temat używania tej reguły walidacji, zobacz DataErrorValidationRule. Alternatywna składnia jawnego ustawiania DataErrorValidationRule polega na ustawieniu właściwości ValidatesOnDataErrors na wartość `true` w obiekcie Binding lub MultiBinding.

Możesz również utworzyć własną regułę walidacji, korzystając z klasy ValidationRule i implementując metodę Validate. W poniższym przykładzie przedstawiono regułę używaną przez obiekt *Dodawanie listy produktów* jako „Data rozpoczęcia" TextBox z sekcji Co to jest powiązanie danych.

C#

```
public class FutureDateRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo
cultureInfo)
```

```
    {
        // Test if date is valid
        if (DateTime.TryParse(value.ToString(), out DateTime date))
        {
            // Date is not in the future, fail
            if (DateTime.Now > date)
                return new ValidationResult(false, "Please enter a date in
the future.");
        }
        else
        {
            // Date is not a valid date, fail
            return new ValidationResult(false, "Value is not a valid
date.");
        }

        // Date is valid and in the future, pass
        return ValidationResult.ValidResult;
    }
}
```

Element *StartDateEntryForm*TextBox używa tego elementu *FutureDateRule*, jak pokazano w poniższym przykładzie.

XAML

```
<TextBox Name="StartDateEntryForm" Grid.Row="3"
         Validation.ErrorTemplate="{StaticResource validationTemplate}"
         Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5"
Grid.ColumnSpan="2">
    <TextBox.Text>
        <Binding Path="StartDate" UpdateSourceTrigger="PropertyChanged"
                 Converter="{StaticResource dateConverter}" >
            <Binding.ValidationRules>
                <src:FutureDateRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

Ponieważ wartość UpdateSourceTrigger to PropertyChanged, aparat powiązania aktualizuje wartość źródłową przy każdym naciśnięciu klawiszy, co oznacza, że sprawdza również każdą regułę w kolekcji ValidationRules przy każdym naciśnięciu klawisza. Omówimy to dokładniej w sekcji Proces walidacji.

Przekazywanie opinii wizualnej

Jeśli użytkownik wprowadzi nieprawidłową wartość, możesz przekazać opinię na temat błędu w interfejsie użytkownika aplikacji. Jednym ze sposobów przekazywania takiej

opinii jest ustawienie dołączonej właściwości Validation.ErrorTemplate na niestandardową ControlTemplate. Jak pokazano w poprzedniej podsekcji, element *StartDateEntryFormTextBox* używa elementu ErrorTemplate o nazwie *validationTemplate*. W poniższym przykładzie przedstawiono definicję elementu *validationTemplate*.

XAML

```
<ControlTemplate x:Key="validationTemplate">
    <DockPanel>
        <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
        <AdornedElementPlaceholder/>
    </DockPanel>
</ControlTemplate>
```

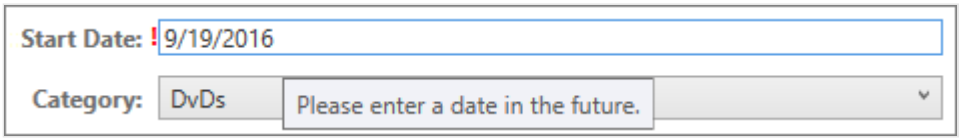
Element AdornedElementPlaceholder określa, gdzie należy umieścić zdobioną kontrolkę.

Ponadto można również użyć elementu ToolTip, aby wyświetlić komunikat o błędzie. Zarówno element *StartDateEntryForm*, jak i *StartPriceEntryFormTextBox* używają stylu *textStyleTextBox*, który tworzy ToolTip wyświetlający komunikat o błędzie. W poniższym przykładzie przedstawiono definicję stylu *textStyleTextBox*. Dołączona właściwość Validation.HasError to wtedy `true`, gdy co najmniej jedno powiązanie we właściwościach powiązanego elementu jest błędne.

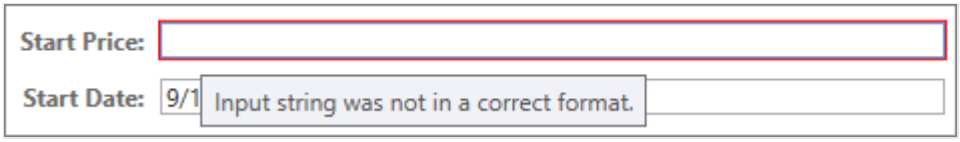
XAML

```
<Style x:Key="textStyleTextBox" TargetType="TextBox">
    <Setter Property="Foreground" Value="#333333" />
    <Setter Property="MaxLength" Value="40" />
    <Setter Property="Width" Value="392" />
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="ToolTip"
                    Value="{Binding
(Validation.Errors).CurrentItem.ErrorContent, RelativeSource={RelativeSource
Self}}" />
        </Trigger>
    </Style.Triggers>
</Style>
```

W przypadku elementu niestandardowego ErrorTemplate i ToolTip element *StartDateEntryFormTextBox* wygląda następująco, gdy wystąpi błąd walidacji.



Jeśli Binding ma skojarzone reguły walidacji, ale nie zostało określone ErrorTemplate w powiązanej kontrolce, zostanie użyta wartość domyślna ErrorTemplate do powiadamiania użytkowników o błędzie walidacji. Wartość domyślna ErrorTemplate to szablon kontrolki, który definiuje czerwone obramowanie w warstwie modułu definiowania układu kodu. W przypadku wartości domyślnej ErrorTemplate i ToolTip interfejs użytkownika elementu *StartPriceEntryForm*TextBox wygląda następująco, gdy wystąpi błąd walidacji.



Aby zapoznać się z przykładem sposobu zapewnienia logiki w celu sprawdzenia poprawności wszystkich kontrolek w oknie dialogowym, zobacz sekcję Niestandardowe okna dialogowe w omówieniu okien dialogowych.

Proces walidacji

Walidacja zwykle występuje, gdy wartość obiektu docelowego jest przenoszona do właściwości źródła powiązania. Ten transfer odbywa się w przypadku powiązań TwoWay i OneWayToSource. Aby powtórzyć to, co powoduje aktualizację źródła, zależy od wartości właściwości UpdateSourceTrigger, zgodnie z opisem w sekcji Co wyzwala aktualizacje źródłowe.

Poniższe elementy opisują proces *walidacji*. Jeśli błąd weryfikacji lub inny typ błędu wystąpi w dowolnym momencie tego procesu, proces zostanie zatrzymany:

1. Aparat powiązań sprawdza, czy istnieją jakiekolwiek zdefiniowane obiekty niestandardowe ValidationRule, których wartość ValidationStep jest ustawiona na RawProposedValue dla Binding, w którym to przypadku wywołuje to metodę Validate dla każdego ValidationRule, dopóki jeden z nich nie napotka błędu lub do momentu zaliczenia wszystkich z nich.

2. Następnie aparat powiązania wywołuje konwerter, jeśli istnieje.

3. Jeśli konwerter zadziała poprawnie, aparat powiązań sprawdza, czy istnieją jakiekolwiek zdefiniowane obiekty niestandardowe ValidationRule, których wartość ValidationStep jest ustawiona na ConvertedProposedValue dla Binding, w którym to przypadku wywołuje to metodę Validate dla każdego ValidationRule, który ma ValidationStep ustawione na ConvertedProposedValue, dopóki jeden z nich nie napotka błędu lub do momentu zaliczenia wszystkich z nich.

4. Aparat powiązań ustawia właściwość źródłową.

5. Aparat powiązań sprawdza, czy istnieją jakiekolwiek zdefiniowane obiekty niestandardowe ValidationRule, których wartość ValidationStep jest ustawiona na UpdatedValue dla Binding, w którym to przypadku wywołuje to metodę Validate dla każdego ValidationRule, który ma ValidationStep ustawione na UpdatedValue, dopóki jeden z nich nie napotka błędu lub do momentu zaliczenia wszystkich z nich. Jeśli element DataErrorValidationRule jest skojarzony z powiązaniem i jego ValidationStep jest ustawione na wartość domyślną, UpdatedValue, element DataErrorValidationRule jest sprawdzany w tym momencie. W tym momencie sprawdzane jest każde powiązanie, które ma ValidatesOnDataErrors ustawione na wartość `true`.

6. Aparat powiązań sprawdza, czy istnieją jakiekolwiek zdefiniowane obiekty niestandardowe ValidationRule, których wartość ValidationStep jest ustawiona na CommittedValue dla Binding, w którym to przypadku wywołuje to metodę Validate dla każdego ValidationRule, który ma ValidationStep ustawione na CommittedValue, dopóki jeden z nich nie napotka błędu lub do momentu zaliczenia wszystkich z nich.

Jeśli obiekt ValidationRule nie zaliczy w dowolnym momencie tego procesu, aparat powiązania tworzy obiekt ValidationError i dodaje go do kolekcji Validation.Errors powiązanego elementu. Zanim aparat powiązań uruchomi obiekty ValidationRule w dowolnym kroku usuwa wszystkie ValidationError dodane do dołączonej właściwości Validation.Errors elementu powiązanego w tym kroku. Jeśli na przykład element ValidationRule, którego ValidationStep jest ustawione na UpdatedValue, nie zaliczył, następnym razem, gdy wystąpi proces walidacji, aparat powiązania usuwa ten ValidationError bezpośrednio przed wywołaniem dowolnego ValidationRule, którego ValidationStep jest ustawione na UpdatedValue.

Gdy właściwość Validation.Errors nie jest pusta, dołączona właściwość Validation.HasError elementu jest ustawiona na wartość `true`. Ponadto jeśli właściwość NotifyOnValidationError obiektu Binding jest ustawiona na `true`, aparat powiązania zgłasza dołączone zdarzenie Validation.Error w elemencie.

Należy również zauważyć, że prawidłowy transfer wartości w obu kierunkach (element docelowy do źródła lub źródło do elementu docelowego) czyści dołączoną właściwość Validation.Errors.

Jeśli powiązanie ma ExceptionValidationRule skojarzone z nim lub właściwość ValidatesOnExceptions była ustawiona na `true` i wyjątek jest zgłaszany, gdy aparat powiązania ustawia źródło, aparat powiązania sprawdza, czy istnieje UpdateSourceExceptionFilter. Możesz użyć wywołania zwrotnego UpdateSourceExceptionFilter, aby zapewnić niestandardową procedurę obsługi

wyjątków. Jeśli parametr UpdateSourceExceptionFilter nie został określony w elemencie Binding, aparat powiązania tworzy obiekt ValidationError z wyjątkiem i dodaje go do kolekcji Validation.Errors powiązanego elementu.

Mechanizm debugowania

Możesz ustawić dołączoną właściwość PresentationTraceSources.TraceLevel dla obiektu związanego z powiązaniem, aby otrzymywać informacje o stanie określonego powiązania.

Zobacz też

- Pokaz powiązania danych
- Omówienie deklaracji powiązań
- Omówienie źródeł powiązań
- DataErrorValidationRule

Przegląd deklaracji powiązań (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 7 min

Zazwyczaj deweloperzy deklarują powiązania bezpośrednio w znacznikach XAML elementów interfejsu użytkownika, z których chcą powiązać dane. Można jednak również zadeklarować powiązania w kodzie. W tym artykule opisano sposób deklarowania powiązań zarówno w języku XAML, jak i w kodzie.

Wymagania wstępne

Przed przeczytaniem tego artykułu należy zapoznać się z koncepcją i użyciem rozszerzeń znaczników. Aby uzyskać więcej informacji na temat rozszerzeń znaczników, zobacz Rozszerzenia znaczników i WPF XAML.

Ten artykuł nie obejmuje pojęć związanych z powiązaniem danych. Aby uzyskać omówienie pojęć związanych z powiązaniem danych, zobacz Omówienie powiązania danych.

Deklarowanie powiązania w XAML

Binding jest rozszerzeniem znaczników. Gdy używasz rozszerzenia powiązania do deklarowania powiązania, `Binding` deklaracja składa się z serii klauzul następujących po słocie kluczowym i rozdzielonych przecinkami (,). Klauzule w deklaracji powiązania mogą być w dowolnej kolejności i istnieje wiele możliwych kombinacji. Klauzule są *parami NameValue*=, gdzie *Name*Binding jest nazwą właściwości, a *Value* jest wartością ustawianą dla właściwości.

Podczas tworzenia ciągów deklaracji powiązania w znacznikach muszą one być dołączone do określonej właściwości zależności obiektu docelowego. W poniższym przykładzie pokazano, jak powiązać właściwość TextBox.Text przy użyciu rozszerzenia powiązania, określając właściwości Source i Path .

XAML

```
<TextBlock Text="{Binding Source={StaticResource myDataSource},
Path=Name}"/>
```

Większość właściwości klasy można określić w Binding ten sposób. Aby uzyskać więcej informacji na Binding temat rozszerzenia powiązania i listy właściwości, których nie można ustawić przy użyciu rozszerzenia powiązania, zobacz Omówienie rozszerzenia znaczników powiązania (.NET Framework).

Składnia elementu obiektu

Składnia elementu obiektu jest alternatywą dla tworzenia deklaracji powiązania. W większości przypadków użycie rozszerzenia znaczników lub składni elementu obiektu nie ma żadnej konkretnej korzyści. Jeśli jednak rozszerzenie znaczników nie obsługuje scenariusza, na przykład gdy wartość właściwości ma typ inny niż ciąg, dla którego nie istnieje konwersja typu, należy użyć składni elementu obiektu.

W poprzedniej sekcji pokazano, jak powiązać z rozszerzeniem XAML. W poniższym przykładzie pokazano wykonanie tego samego powiązania, ale użyto składni elementu obiektu:

XAML

```
<TextBlock>
    <TextBlock.Text>
        <Binding Source="{StaticResource myDataSource}" Path="Name"/>
    </TextBlock.Text>
</TextBlock>
```

Aby uzyskać więcej informacji na temat różnych terminów, zobacz Szczegóły składni języka XAML (.NET Framework).

MultiBinding i PriorityBinding

MultiBinding i PriorityBinding nie obsługują składni rozszerzenia XAML. Dlatego należy użyć składni elementu obiektu, jeśli MultiBinding deklaruje się element lub w PriorityBinding języku XAML.

Tworzenie powiązania w kodzie

Innym sposobem określenia powiązania jest ustawienie właściwości bezpośrednio na obiekcie w kodzie, a Binding następnie przypisanie powiązania do właściwości. W poniższym przykładzie pokazano, jak utworzyć obiekt Binding w kodzie.

C#

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Make a new data source object
    var personDetails = new Person()
    {
        Name = "John",
        Birthdate = DateTime.Parse("2001-02-03")
    };

    // New binding object using the path of 'Name' for whatever source
object is used
    var nameBindingObject = new Binding("Name");

    // Configure the binding
    nameBindingObject.Mode = BindingMode.OneWay;
    nameBindingObject.Source = personDetails;
    nameBindingObject.Converter = NameConverter.Instance;
    nameBindingObject.ConverterCulture = new CultureInfo("en-US");

    // Set the binding to a target object. The TextBlock.Name property on
the NameBlock UI element
    BindingOperations.SetBinding(NameBlock, TextBlock.TextProperty,
nameBindingObject);
}
```

Poprzedni kod ustawia następujący kod w powiązaniu:

- Ścieżka właściwości obiektu źródła danych.
- Tryb powiązania.
- Źródło danych, w tym przypadku proste wystąpienie obiektu reprezentujące osobę.
- Opcjonalny konwerter, który przetwarza wartość przychodzącą z obiektu źródła danych, zanim zostanie przypisana do właściwości target.

Gdy obiektem, który jest powiązaniem, jest obiekt lub FrameworkElementFrameworkContentElement, `SetBinding` możesz wywołać metodę bezpośrednio w obiekcie zamiast używać metody BindingOperations.SetBinding. Aby uzyskać przykład, zobacz How to: Create a Binding in Code (Jak utworzyć powiązanie w kodzie).

W poprzednim przykładzie użyto prostego typu obiektu danych .`Person` Poniżej przedstawiono kod dla tego obiektu:

C#

```
class Person
{
    public string Name { get; set; }
```

```
    public DateTime Birthdate { get; set; }
}
```

Składnia ścieżki powiązania

Użyj właściwości Path , aby określić wartość źródłową, z którą chcesz powiązać:

- W najprostszym przypadku wartość Path właściwości jest nazwą właściwości obiektu źródłowego do użycia dla powiązania, na przykład `Path=PropertyName`.
- Właściwości podrzędne właściwości można określić za pomocą podobnej składni, jak w języku C#. Na przykład klauzula ustawia `Path=ShoppingCart.Order` powiązanie na podwłaściwość `Order` obiektu lub właściwości `ShoppingCart`.
- Aby powiązać element z dołączona właściwością, umieść nawiasy wokół dołączonej właściwości. Aby na przykład powiązać z dołączona właściwością DockPanel.Dock, składnia to `Path=(DockPanel.Dock)`.
- Indeksatory właściwości można określić w nawiasach kwadratowych po nazwie właściwości, w której zastosowano indeksator. Na przykład klauzula ustawia `Path=ShoppingCart[0]` powiązanie na indeks, który odpowiada sposobu obsługi ciągu literału "0" przez indeksowanie wewnętrzne właściwości. Indeksatory zagnieżdżone są również obsługiwane.
- Indeksery i właściwości podrzędne mogą być mieszane w klauzuli `Path` , na przykład `Path=ShoppingCart.ShippingInfo[MailingAddress,Street].`
- Wewnątrz indeksatorów. Można mieć wiele parametrów indeksatora rozdzielonych przecinkami (`,`). Typ każdego parametru można określić za pomocą nawiasów. Na przykład można użyć elementu `Path="[(sys:Int32)42,(sys:Int32)24]"`, gdzie `sys` jest mapowane na przestrzeń `System` nazw .
- Gdy źródło jest widokiem kolekcji, bieżący element można określić ukośnikiem (`/`). Na przykład klauzula ustawia `Path=/` powiązanie na bieżący element w widoku. Gdy źródło jest kolekcją, ta składnia określa bieżący element domyślnego widoku kolekcji.
- Nazwy właściwości i ukośniki można łączyć w celu przechodzenia przez właściwości, które są kolekcjami. Na przykład określa `Path=/Offices/ManagerName` bieżący element kolekcji źródłowej, który zawiera właściwość `Offices` , która jest również kolekcją. Jego bieżącym elementem jest obiekt, który zawiera właściwość `ManagerName` .

- Opcjonalnie ścieżka okresu (`.`) może służyć do powiązania z bieżącym źródłem. Na przykład instrukcja `Text="{Binding}"` jest równoważna instrukcji `Text="{Binding Path=.}"`.

Mechanizm ucieczki

- Wewnątrz indeksatorów (`[ ]`) znak caret (`^`) unika następnego znaku.
- W przypadku ustawienia w Path języku XAML należy również użyć ucieczki (przy użyciu jednostek XML) niektórych znaków, które są specjalne dla definicji języka XML:
  - Użyj , `&` aby przed znakiem ucieczki "`&`".
  - Użyj , `>` aby uniknąć tagu końcowego "`>`".
- Ponadto jeśli opiszesz całe powiązanie w atlicie przy użyciu składni rozszerzenia znaczników, musisz użyć znaku ucieczki (`\`przy użyciu ukośnika odwrotnego), które są specjalne dla parsera rozszerzenia znaczników WPF:
  - Ukośnik odwrotny (`\`) to sam znak ucieczki.
  - Znak równości (`=`) oddziela nazwę właściwości od wartości właściwości.
  - Przecinek (`,`) oddziela właściwości.
  - Prawy nawias klamrowy (`}`) to koniec rozszerzenia znaczników.

Kierunek powiązania

Użyj właściwości Binding.Mode , aby określić kierunek powiązania. Dostępne są następujące tryby aktualizacji powiązań:

Tryb powiązania	Opis
BindingMode.TwoWay	Aktualizuje właściwość docelową lub właściwość za każdym razem, gdy właściwość docelowa lub właściwość źródłową się zmieni.
BindingMode.OneWay	Aktualizuje właściwość docelową tylko wtedy, gdy właściwość źródłową się zmienia.
BindingMode.OneTime	Aktualizuje właściwość docelową tylko wtedy, gdy aplikacja jest uruchamiana lub DataContext gdy element jest zmieniany.

Tryb powiązania	Opis
BindingMode.OneWayToSource	Aktualizuje właściwość źródłową po zmianie właściwości docelowej.
BindingMode.Default	Powoduje, że Mode wartość domyślna właściwości docelowej ma być używana.

Aby uzyskać więcej informacji, zobacz wyliczenie BindingMode .

W poniższym przykładzie pokazano, jak ustawić właściwość Mode :

XAML

```
<TextBlock Name="IncomeText" Text="{Binding Path=TotalIncome, Mode=OneTime}"
/>
```

Aby wykryć zmiany źródła (dotyczy powiązań OneWayTwoWay i), źródło musi zaimplementować odpowiedni mechanizm powiadamiania o zmianie właściwości, taki jak INotifyPropertyChanged. Aby uzyskać więcej informacji, zobacz Dostarczanie powiadomień o zmianach.

W TwoWay przypadku OneWayToSource powiązań lub można kontrolować czas aktualizacji źródła, ustawiając właściwość UpdateSourceTrigger . Aby uzyskać więcej informacji, zobacz UpdateSourceTrigger.

Domyślne zachowania

Zachowanie domyślne jest następujące, jeśli nie zostanie określone w deklaracji:

- Zostanie utworzony domyślny konwerter, który próbuje wykonać konwersję typu między wartością źródłową powiązania a docelową wartością powiązania. Jeśli nie można uzyskać konwersji, konwerter domyślny zwraca wartość `null`.
- Jeśli nie ustawisz wartości ConverterCulture, aparat powiązań użyje `Language` właściwości obiektu docelowego powiązania. W języku XAML wartość domyślna `en-US` to lub dziedziczy wartość z elementu głównego (lub dowolnego elementu) strony, jeśli została jawnie ustawiona.
- Jeśli powiązanie ma już kontekst danych (na przykład dziedziczony kontekst danych z elementu nadrzędnego) i dowolny element lub kolekcja zwracana przez ten kontekst jest odpowiednia do powiązania bez konieczności dalszej modyfikacji ścieżki, deklaracja powiązania nie może mieć żadnych klauzul: `{Binding}`. Jest to często sposób, w jaki powiązanie jest określane dla stylów danych, gdzie

powiązanie działa na kolekcję. Aby uzyskać więcej informacji, zobacz Using Entire Objects as a Binding Source (Używanie całych obiektów jako źródła powiązania).

- Wartość domyślna Mode różni się w zależności od powiązanej właściwości zależności. Zawsze możesz jawnie zadeklarować tryb powiązania, aby upewnić się, że powiązanie ma żądane zachowanie. Ogólnie rzecz biorąc, edytowalne przez użytkownika właściwości kontrolek, TextBox.Text takie jak i RangeBase.Value, są domyślnie powiązaniami dwukierunkowymi, ale większość innych właściwości jest domyślnie łączona z powiązaniami jednokierunkowymi.

- Wartość domyślna UpdateSourceTrigger różni się również LostFocusPropertyChanged w zależności od powiązanej właściwości zależności. Wartość domyślna większości właściwości zależności to PropertyChanged, a właściwość TextBox.Text ma wartość domyślną LostFocus.

Zobacz też

- Omówienie powiązań danych
- Omówienie powiązań źródeł
- Składnia XAML PropertyPath (.NET Framework)

Omówienie źródeł powiązań (WPF .NET)

Artykuł • 18.02.2023 • Czas czytania: 5 min

W powiązaniu danych obiekt źródłowy powiązania odwołuje się do obiektu, z którego uzyskujesz dane. W tym artykule omówiono typy obiektów, których można używać jako źródła powiązania, takich jak obiekty CLR platformy .NET, XML i DependencyObject obiekty.

Powiązania typów źródłowych

Powiązanie danych Windows Presentation Foundation (WPF) obsługuje następujące typy źródeł powiązań:

- **Obiekty środowiska uruchomieniowego języka wspólnego platformy .NET (CLR)**

  Można powiązać z właściwościami publicznymi, właściwościami podrzędni i indeksatorami dowolnego obiektu środowiska uruchomieniowego języka wspólnego (CLR). Aparat powiązań używa odbicia CLR, aby uzyskać wartości właściwości. Obiekty implementujące ICustomTypeDescriptor lub zarejestrowane TypeDescriptionProvider również współpracują z aparatem powiązania.

  Aby uzyskać więcej informacji na temat implementowania klasy, która może służyć jako źródło powiązania, zobacz Implementowanie źródła powiązania dla obiektów w dalszej części tego artykułu.

- **Obiekty dynamiczne**

  Można powiązać z dostępnymi właściwościami i indeksatorami obiektu, który implementuje IDynamicMetaObjectProvider interfejs. Jeśli możesz uzyskać dostęp do elementu członkowskiego w kodzie, możesz powiązać go z nim. Jeśli na przykład obiekt dynamiczny umożliwia dostęp do elementu członkowskiego w kodzie za pośrednictwem metody `someObjet.AProperty`, możesz powiązać z nim, ustawiając ścieżkę powiązania na `AProperty`.

- **obiekty ADO.NET**

  Można powiązać z obiektami ADO.NET, takimi jak DataTable. ADO.NET DataView implementuje IBindingList interfejs, który zapewnia powiadomienia o zmianach, dla których nasłuchuje aparat powiązania.

- **Obiekty XML**

Zapytania można powiązać i uruchamiać `XPath` na obiekcie XmlNode, XmlDocumentlub XmlElement. Wygodnym sposobem uzyskiwania dostępu do danych XML, które jest źródłem powiązania w adiustacji, jest użycie XmlDataProvider obiektu. Aby uzyskać więcej informacji, zobacz Powiązanie z danymi XML przy użyciu elementu XMLDataProvider i zapytań XPath (.NET Framework).

Można również powiązać z obiektem XElement lub lub XDocumentpowiązać wyniki zapytań uruchamianych na obiektach tych typów przy użyciu LINQ to XML. Wygodnym sposobem używania LINQ to XML do uzyskiwania dostępu do danych XML, które jest źródłem powiązania w znacznikach, jest użycie ObjectDataProvider obiektu. Aby uzyskać więcej informacji, zobacz Bind to XDocument, XElement lub LINQ for XML Query Results (.NET Framework).

- **DependencyObject Obiektów**

  Można powiązać z właściwościami zależności dowolnego DependencyObjectobiektu . Aby zapoznać się z przykładem, zobacz Powiązanie właściwości dwóch kontrolek (.NET Framework).

Implementowanie źródła powiązania na obiektach

Obiekty CLR mogą stać się źródłami powiązań. Istnieje kilka kwestii, o których należy pamiętać podczas implementowania klasy, która będzie służyć jako źródło powiązania.

Podawanie powiadomień o zmianie

Jeśli używasz powiązania OneWay lub TwoWay , zaimplementuj odpowiedni mechanizm powiadamiania o zmianie właściwości. Zalecanym mechanizmem jest zaimplementowanie interfejsu przez clR lub klasę dynamiczną INotifyPropertyChanged . Aby uzyskać więcej informacji, zobacz Instrukcje: implementowanie powiadomienia o zmianie właściwości (.NET Framework).

Istnieją dwa sposoby powiadamiania subskrybenta o zmianie właściwości:

1. Zaimplementuj interfejs INotifyPropertyChanged.

   Jest to zalecany mechanizm powiadomień. Element INotifyPropertyChanged dostarcza PropertyChanged zdarzenie, które jest zgodne z systemem powiązań. Podnosząc to zdarzenie i podając nazwę zmienionej właściwości, powiadomisz element docelowy powiązania zmiany.

2. Zaimplementuj `PropertyChanged` wzorzec.

   Każda właściwość, która musi powiadomić obiekt docelowy powiązania o zmianie, ma odpowiednie `PropertyNameChanged` zdarzenie, gdzie `PropertyName` jest nazwą właściwości. Zdarzenie jest wywoływane za każdym razem, gdy właściwość się zmienia.

Jeśli źródło powiązania implementuje jeden z tych mechanizmów powiadomień, aktualizacje docelowe są wykonywane automatycznie. Jeśli z jakiegokolwiek powodu źródło powiązania nie udostępnia odpowiednich powiadomień o zmianie właściwości, możesz użyć UpdateTarget metody , aby jawnie zaktualizować właściwość docelową.

Inne cechy

Poniższa lista zawiera inne ważne kwestie do zapamiętania:

- Obiekty danych, które służą jako źródła powiązań, można zadeklarować w języku XAML jako zasoby, pod warunkiem że mają **konstruktor bez parametrów**. W przeciwnym razie należy utworzyć obiekt danych w kodzie i przypisać go bezpośrednio do kontekstu danych drzewa obiektów XAML lub jako źródło powiązania powiązania.

- Właściwości używane jako właściwości źródła powiązania muszą być właściwościami publicznymi klasy. Jawnie zdefiniowane właściwości interfejsu nie mogą być dostępne do celów powiązań ani nie mogą być chronione, prywatne, wewnętrzne lub wirtualne właściwości, które nie mają implementacji podstawowej.

- Nie można powiązać z polami publicznymi.

- Typ właściwości zadeklarowanej w klasie jest typem przekazywanym do powiązania. Jednak typ ostatecznie używany przez powiązanie zależy od typu właściwości docelowej powiązania, a nie właściwości źródłowej powiązania. Jeśli istnieje różnica w typie, możesz napisać konwerter, aby obsłużyć sposób, w jaki właściwość niestandardowa jest początkowo przekazywana do powiązania. Aby uzyskać więcej informacji, zobacz IValueConverter.

Całe obiekty jako źródło powiązania

Można użyć całego obiektu jako źródła powiązania. Określ źródło powiązania przy użyciu Source właściwości lub DataContext , a następnie podaj pustą deklarację powiązania: `{Binding}`. Scenariusze, w których jest to przydatne, obejmują powiązanie z obiektami, które są ciągiem typu, powiązaniem z obiektami z wieloma właściwościami,

które cię interesują, lub powiązaniem z obiektami kolekcji. Przykład powiązania z całym obiektem kolekcji można znaleźć w temacie How to Use the Master-Detail Pattern with Hierarchical Data (.NET Framework) (Jak używać wzorca Master-Detail z danymi hierarchicznymi (.NET Framework).

Może być konieczne zastosowanie logiki niestandardowej, aby dane miały znaczenie dla powiązanej właściwości docelowej. Logika niestandardowa może być w postaci konwertera niestandardowego lub DataTemplate. Aby uzyskać więcej informacji na temat konwerterów, zobacz Konwersja danych. Aby uzyskać więcej informacji na temat szablonów danych, zobacz Omówienie szablonów danych (.NET Framework).

Obiekty kolekcji jako źródło powiązania

Często obiekt, którego chcesz użyć jako źródła powiązania, jest kolekcją obiektów niestandardowych. Każdy obiekt służy jako źródło dla jednego wystąpienia powtarzającego się powiązania. Na przykład może istnieć `CustomerOrders` kolekcja składająca się z `CustomerOrder` obiektów, w której aplikacja iteruje kolekcję w celu określenia liczby zamówień i danych zawartych w każdej kolejności.

Możesz wyliczyć dowolną kolekcję, która implementuje interfejs IEnumerable. Aby jednak skonfigurować powiązania dynamiczne tak, aby wstawienie lub usunięcie w kolekcji automatycznie zaktualizowało interfejs użytkownika, kolekcja musi zaimplementować interfejs INotifyCollectionChanged. Ten interfejs uwidacznia zdarzenie, które musi zostać zgłoszone za każdym razem, gdy zmienia się podstawowa kolekcja.

Klasa ObservableCollection<T> jest wbudowaną implementacją kolekcji danych, która uwidacznia INotifyCollectionChanged interfejs. Poszczególne obiekty danych w kolekcji muszą spełniać wymagania opisane w poprzednich sekcjach. Aby zapoznać się z przykładem, zobacz How to Create and Bind to an ObservableCollection (.NET Framework)(Jak utworzyć i powiązać z obiektem ObservableCollection (.NET Framework). Przed wdrożeniem własnej kolekcji rozważ użycie ObservableCollection<T> lub jedną z istniejących klas kolekcji, takich jak List<T>, Collection<T>i BindingList<T>, między innymi.

Po określeniu kolekcji jako źródła powiązania platforma WPF nie wiąże się bezpośrednio z kolekcją. Zamiast tego WPF faktycznie wiąże się z domyślnym widokiem kolekcji. Aby uzyskać informacje o widokach domyślnych, zobacz Używanie widoku domyślnego.

Jeśli masz zaawansowany scenariusz i chcesz zaimplementować własną kolekcję, rozważ użycie interfejsu IList . Ten interfejs udostępnia niegeneryjną kolekcję obiektów, do

których można uzyskać dostęp indywidualnie według indeksu, co może zwiększyć wydajność.

Wymagania dotyczące uprawnień w powiązaniu danych

W przeciwieństwie do .NET Framework platforma .NET działa z zabezpieczeniami pełnego zaufania. Wszystkie powiązania danych są uruchamiane z tym samym dostępem co użytkownik, który uruchamia aplikację.

Zobacz też

- ObjectDataProvider
- XmlDataProvider
- Omówienie powiązań danych
- Omówienie źródeł powiązań
- Omówienie powiązania danych WPF z LINQ to XML (.NET Framework)
- Optymalizacja wydajności: powiązanie danych (.NET Framework)

Jak powiązać z wyliczeniem (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 2 min

W tym przykładzie pokazano, jak powiązać z wyliczeniem. Niestety nie ma bezpośredniego sposobu użycia wyliczenia jako źródła powiązania danych. Enum.GetValues(Type) Jednak metoda zwraca kolekcję wartości. Te wartości mogą być opakowane w obiekt ObjectDataProvider i używane jako źródło danych.

Typ ObjectDataProvider zapewnia wygodny sposób tworzenia obiektu w języku XAML i używania go jako źródła danych.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Odwołanie do wyliczenia

ObjectDataProvider Użyj typu, aby opakowować tablicę wartości wyliczenia dostarczonych przez sam typ wyliczenia.

1. Utwórz nowy `ObjectDataProvider` zasób XAML w aplikacji XAML lub XAML obiektu, z którym pracujesz. W tym przykładzie użyto okna i utworzono `ObjectDataProvider` element z kluczem zasobu .`EnumDataSource`

   XAML

   ```
   <Window.Resources>
       <ObjectDataProvider x:Key="EnumDataSource"
                           ObjectType="{x:Type sys:Enum}"
                           MethodName="GetValues">
           <ObjectDataProvider.MethodParameters>
               <x:Type TypeName="HorizontalAlignment" />
           </ObjectDataProvider.MethodParameters>
       </ObjectDataProvider>
   </Window.Resources>
   ```

   W tym przykładzie użyto `ObjectDataProvider` trzech właściwości do pobrania wyliczenia:

Właściwość	Opis

Właściwość	Opis
`ObjectType`	Typ obiektu, który ma zostać zwrócony przez dostawcę danych. W tym przykładzie System.Enum. `sys:` Przestrzeń nazw XAML jest mapowana na `System`.
`MethodName`	Nazwa metody do uruchomienia w typie `System.Enum` . W tym przykładzie Enum.GetValues.
`MethodParameters`	Kolekcja wartości, które mają być podane w metodzie `MethodName` . W tym przykładzie metoda przyjmuje `System.Type` wyliczenie.

Skutecznie kod XAML rozkłada wywołanie metody, nazwę metody, parametry i typ zwracany. Skonfigurowany `ObjectDataProvider` w poprzednim przykładzie jest odpowiednikiem następującego kodu:

C#

```
var enumDataSource =
System.Enum.GetValues(typeof(System.Windows.HorizontalAlignment));
```

2. Odwołanie do `ObjectDataProvider` zasobu. Poniższy kod XAML zawiera listę wartości wyliczenia w kontrolce ListBox :

XAML

```
<ListBox Name="myComboBox" SelectedIndex="0"
         ItemsSource="{Binding Source={StaticResource
EnumDataSource}}"/>
```

Pełny kod XAML

Poniższy kod XAML reprezentuje proste okno, które wykonuje następujące czynności:

1. Opakowuje HorizontalAlignment wyliczenie w ObjectDataProvider źródle danych jako zasób.
2. Udostępnia kontrolkę do wyświetlania ListBox listy wszystkich wartości wyliczenia.
3. Wiąże Button właściwość kontrolki HorizontalAlignment z wybranym elementem w elemencie `ListBox`.

XAML

```
<Window x:Class="ArticleExample.BindEnumFull"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        SizeToContent="WidthAndHeight"
        Title="Enum binding">
    <Window.Resources>
        <ObjectDataProvider x:Key="EnumDataSource"
                            ObjectType="{x:Type sys:Enum}"
                            MethodName="GetValues">
            <ObjectDataProvider.MethodParameters>
                <x:Type TypeName="HorizontalAlignment" />
            </ObjectDataProvider.MethodParameters>
        </ObjectDataProvider>
    </Window.Resources>

    <StackPanel Width="300" Margin="10">
        <TextBlock>Choose the HorizontalAlignment value of the Button:
</TextBlock>

        <ListBox Name="myComboBox" SelectedIndex="0"
                 ItemsSource="{Binding Source={StaticResource
EnumDataSource}}"/>

        <Button Content="I'm a button"
                HorizontalAlignment="{Binding ElementName=myComboBox,
Path=SelectedItem}" />
    </StackPanel>
</Window>
```

Zobacz też

- Omówienie powiązań danych
- Omówienie źródeł powiązań
- StaticResource — Rozszerzenie znaczników
- Alternatywny sposób powiązania z wyliczeniem

Przegląd zdarzeń trasowanych (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 20 min

Windows Presentation Foundation (WPF) deweloperzy aplikacji i autorzy składników mogą używać zdarzeń kierowanych do propagacji zdarzeń za pomocą drzewa elementów i wywoływać programy obsługi zdarzeń na wielu odbiornikach w drzewie. Te funkcje nie są znajdowane w zdarzeniach środowiska uruchomieniowego języka wspólnego (CLR). Kilka zdarzeń WPF to zdarzenia kierowane, takie jak ButtonBase.Click. W tym artykule omówiono podstawowe pojęcia dotyczące zdarzeń kierowanych i oferuje wskazówki dotyczące tego, kiedy i jak reagować na kierowane zdarzenia.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule przyjęto założenie, że podstawowa wiedza na temat środowiska uruchomieniowego języka wspólnego (CLR), programowania obiektowego i sposobu, w jaki układ elementów WPF może być koncepcyjny jako drzewo. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

Co to jest zdarzenie kierowane?

Możesz rozważyć kierowanie zdarzeń z perspektywy funkcjonalnej lub implementacji:

- Z perspektywy **funkcjonalnej** zdarzenie kierowane jest typem zdarzenia, które może wywoływać programy obsługi na wielu odbiornikach w drzewie elementów, a nie tylko w źródle zdarzeń. Odbiornik zdarzeń to element, w którym jest dołączony i wywoływany program obsługi zdarzeń. Źródłem zdarzeń jest element lub obiekt, który pierwotnie zgłosił zdarzenie.
- Z perspektywy **implementacji** zdarzenie kierowane jest zdarzeniem zarejestrowanym w systemie zdarzeń WPF, wspieranym przez wystąpienie RoutedEvent klasy i przetwarzanym przez system zdarzeń WPF. Zazwyczaj

zdarzenie kierowane jest implementowane przy użyciu zdarzenia CLR "otoka", aby umożliwić dołączanie programów obsługi w języku XAML i w kodzie za pomocą zdarzenia CLR.

Aplikacje WPF zwykle zawierają wiele elementów, które zostały zadeklarowane w języku XAML lub utworzone w kodzie. Elementy aplikacji istnieją w drzewie elementów. W zależności od sposobu definiowania zdarzenia kierowanego, gdy zdarzenie jest wywoływane w elemecie źródłowym:

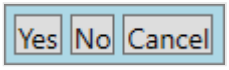
- Bąbelki w górę przez drzewo elementów z elementu źródłowego do elementu głównego, który jest zazwyczaj stroną lub oknem.
- Tunele w dół przez drzewo elementów z elementu głównego do elementu źródłowego.
- Nie przechodzi przez drzewo elementów i występuje tylko w elemecie źródłowym.

Rozważ następujące drzewo częściowe:

XAML

```
<Border Height="30" Width="200" BorderBrush="Gray" BorderThickness="1">
    <StackPanel Background="LightBlue" Orientation="Horizontal"
Button.Click="YesNoCancelButton_Click">
        <Button Name="YesButton">Yes</Button>
        <Button Name="NoButton">No</Button>
        <Button Name="CancelButton">Cancel</Button>
    </StackPanel>
</Border>
```

Drzewo elementów jest renderowane w sposób pokazany:



Każdy z trzech przycisków jest potencjalnym Click źródłem zdarzeń. Po kliknięciu jednego z przycisków zostanie wyświetlone `Click` zdarzenie, które bąbelkuje od przycisku do elementu głównego. Elementy Button i Border nie mają dołączonych procedur obsługi zdarzeń, ale StackPanel nie. Prawdopodobnie inne elementy wyższe w drzewie, które nie są wyświetlane, również mają `Click` dołączone programy obsługi zdarzeń. Gdy `Click` zdarzenie osiągnie `StackPanel` element, system zdarzeń WPF wywołuje `YesNoCancelButton_Click` program obsługi, który jest do niego dołączony. Trasa zdarzenia dla `Click` zdarzenia w przykładzie to: `Button` -`StackPanel`> - - >`Border`> kolejne elementy nadrzędne.

Uwaga

Element, który pierwotnie zgłosił zdarzenie kierowane, jest identyfikowany jako **RoutedEventArgs.Source** w parametrach programu obsługi zdarzeń. Odbiornik zdarzeń jest elementem, w którym program obsługi zdarzeń jest dołączony i wywoływany, i jest identyfikowany jako **nadawca** w parametrach programu obsługi zdarzeń.

Scenariusze najwyższego poziomu dotyczące zdarzeń kierowanych

Poniżej przedstawiono niektóre scenariusze, które motywowały koncepcję zdarzenia kierowanego i odróżniały je od typowego zdarzenia CLR:

- **Kompozycja sterowania i hermetyzacja: różne kontrolki** w WPF mają bogaty model zawartości. Na przykład można umieścić obraz w obiekcie Button, który skutecznie rozszerza drzewo wizualne przycisku. Dodany obraz nie może jednak przerwać działania przycisku hit-test, który musi odpowiadać, gdy użytkownik kliknie piksel obrazu.

- **Pojedyncze punkty załączników programu obsługi**: można zarejestrować procedurę obsługi dla zdarzenia każdego przycisku `Click` , ale z zdarzeniami kierowanymi można dołączyć pojedynczą procedurę obsługi, jak pokazano w poprzednim przykładzie XAML. Umożliwia to zmianę drzewa elementów w ramach pojedynczej procedury obsługi, takiej jak dodawanie lub usuwanie większej liczby przycisków bez konieczności rejestrowania `Click` zdarzenia każdego przycisku. Gdy zdarzenie zostanie podniesione, logika `Click` obsługi może określić miejsce, z którego pochodzi zdarzenie. Poniższa procedura obsługi określona w wcześniej wyświetlonym drzewie elementów XAML zawiera logikę:

C#

```
private void YesNoCancelButton_Click(object sender, RoutedEventArgs e)
{
    FrameworkElement sourceFrameworkElement = e.Source as
FrameworkElement;
    switch (sourceFrameworkElement.Name)
    {
        case "YesButton":
            // YesButton logic.
            break;
        case "NoButton":
            // NoButton logic.
            break;
        case "CancelButton":
            // CancelButton logic.
```

```
                break;
        }
        e.Handled = true;
    }
```

- **Obsługa klas**: zdarzenia kierowane obsługują procedurę obsługi zdarzeń klasy definiowaną w klasie. Programy obsługi klas obsługują zdarzenie przed wszystkimi procedurami obsługi wystąpień dla tego samego zdarzenia w dowolnym wystąpieniu klasy.

- **Odwoływanie się do zdarzenia bez odbicia**: każde kierowane zdarzenie tworzy RoutedEvent identyfikator pola, aby zapewnić niezawodną technikę identyfikacji zdarzeń, która nie wymaga odbicia statycznego ani czasu wykonywania w celu zidentyfikowania zdarzenia.

Jak są implementowane zdarzenia kierowane

Zdarzenie kierowane to zdarzenie zarejestrowane w systemie zdarzeń WPF, wspierane przez wystąpienie RoutedEvent klasy i przetwarzane przez system zdarzeń WPF. Wystąpienie `RoutedEvent` uzyskane z rejestracji jest zwykle przechowywane jako `public static readonly` element członkowski klasy, która ją zarejestrowała. Ta klasa jest nazywana klasą "właściciel" zdarzenia. Zazwyczaj zdarzenie kierowane implementuje identycznie nazwane zdarzenie CLR "otoka". Otoka zdarzeń CLR zawiera `add` i `remove` metody dostępu umożliwiające dołączanie procedur obsługi w języku XAML i w kodzie za pomocą składni zdarzeń specyficznych dla języka. Metody `add` i `remove` zastępują implementację środowiska CLR i wywołają kierowane zdarzenia AddHandler i RemoveHandler metody. Mechanizm tworzenia kopii zapasowych zdarzeń kierowanych i połączenia jest koncepcyjnie podobny do sposobu, w jaki właściwość zależności jest właściwością CLR, która jest wspierana przez DependencyProperty klasę i zarejestrowana w systemie właściwości WPF.

Poniższy przykład rejestruje `Tap` zdarzenie kierowane, przechowuje zwrócone `RoutedEvent` wystąpienie i implementuje otokę zdarzeń CLR.

C#

```
// Register a custom routed event using the Bubble routing strategy.
public static readonly RoutedEvent TapEvent =
EventManager.RegisterRoutedEvent(
    name: "Tap",
    routingStrategy: RoutingStrategy.Bubble,
    handlerType: typeof(RoutedEventHandler),
    ownerType: typeof(CustomButton));
```

```
// Provide CLR accessors for adding and removing an event handler.
public event RoutedEventHandler Tap
{
    add { AddHandler(TapEvent, value); }
    remove { RemoveHandler(TapEvent, value); }
}
```

Strategie routingu

Zdarzenia kierowane używają jednej z trzech strategii routingu:

- **Bubbling**: Początkowo wywoływane są programy obsługi zdarzeń w źródle zdarzeń. Zdarzenie kierowane następnie kieruje do kolejnych elementów nadrzędnych, wywołując ich programy obsługi zdarzeń z kolei, aż dotrze do katalogu głównego drzewa elementów. Większość zdarzeń kierowanych korzysta ze strategii routingu. Zdarzenia kierowane przez bubbling są zwykle używane do zgłaszania zmian danych wejściowych lub stanu z kontrolek złożonych lub innych elementów interfejsu użytkownika.

- **Tunelowanie**: początkowo wywoływane są programy obsługi zdarzeń w katalogu głównym drzewa elementów. Zdarzenie kierowane następnie kieruje do kolejnych elementów podrzędnych, wywołując ich programy obsługi zdarzeń, aż do momentu dotarcia do źródła zdarzeń. Zdarzenia, które są zgodne z trasą tunelowania, są również nazywane zdarzeniami w wersji zapoznawczej . Zdarzenia wejściowe WPF są zazwyczaj implementowane jako pary podglądu i bubbling.

- **Bezpośredni**: wywoływane są tylko programy obsługi zdarzeń w źródle zdarzeń. Ta strategia bez routingu jest analogiczna do zdarzeń platformy interfejsu użytkownika Windows Forms, które są standardowymi zdarzeniami CLR. W przeciwieństwie do zdarzeń CLR zdarzenia kierowane obsługują obsługę klas i mogą być używane przez klasy EventSetters i EventTriggers.

Dlaczego warto używać zdarzeń trasowanych?

Jako deweloper aplikacji nie zawsze musisz wiedzieć lub zadbać o to, aby zdarzenie, które obsługujesz, zostało zaimplementowane jako zdarzenie kierowane. Zdarzenia kierowane mają specjalne zachowanie, ale takie zachowanie jest w dużej mierze niewidoczne, jeśli obsługujesz zdarzenie w elemecie, który go podniósł. Jednak zdarzenia kierowane są istotne, gdy chcesz dołączyć procedurę obsługi zdarzeń do elementu nadrzędnego w celu obsługi zdarzeń zgłaszanych przez elementy podrzędne, na przykład w ramach kontrolki złożonej.

Odbiorniki zdarzeń kierowanych nie potrzebują zdarzeń kierowanych, które obsługują, aby były członkami swojej klasy. Dowolny UIElement lub ContentElement może być odbiornikiem zdarzeń dla dowolnego zdarzenia kierowanego. Ponieważ elementy wizualne pochodzą z `UIElement` elementu lub `ContentElement`, można użyć zdarzeń kierowanych jako koncepcyjnego "interfejsu", który obsługuje wymianę informacji o zdarzeniach między różnymi elementami w aplikacji. Koncepcja "interfejsu" dla zdarzeń kierowanych jest szczególnie stosowana do zdarzeń wejściowych.

Zdarzenia kierowane obsługują wymianę informacji o zdarzeniach między elementami wzdłuż trasy zdarzenia, ponieważ każdy odbiornik ma dostęp do tego samego wystąpienia danych zdarzenia. Jeśli jeden element zmieni coś w danych zdarzenia, ta zmiana jest widoczna dla kolejnych elementów w trasie zdarzeń.

Oprócz aspektu routingu można wybrać zaimplementowanie zdarzenia kierowanego zamiast standardowego zdarzenia CLR z następujących powodów:

- Niektóre funkcje stylów I szablonów WPF, takie jak EventSetters i EventTriggers, wymagają zdarzenia, do których odwołuje się zdarzenie, aby było zdarzeniem kierowanym.
- Zdarzenia trasowane obsługują programy obsługi zdarzeń klasy , które obsługują zdarzenie przed wszystkimi procedurami obsługi wystąpień dla tego samego zdarzenia w dowolnym wystąpieniu klasy odbiornika. Ta funkcja jest przydatna w projekcie sterowania, ponieważ program obsługi klas może wymuszać zachowania klas sterowane zdarzeniami, których nie można przypadkowo pominąć przez program obsługi wystąpień.

Dołączanie i implementowanie procedury obsługi zdarzeń kierowanych

W języku XAML program obsługi zdarzeń jest dołączany do elementu, deklarując nazwę zdarzenia jako atrybut elementu odbiornika zdarzeń. Wartość atrybutu to nazwa metody procedury obsługi. Metoda obsługi musi zostać zaimplementowana w klasie częściowej kodu dla strony XAML. Odbiornik zdarzeń jest elementem, w którym jest dołączona i wywoływana procedura obsługi zdarzeń.

W przypadku zdarzenia, które jest elementem członkowskim (dziedziczone lub w inny sposób) klasy odbiornika, można dołączyć procedurę obsługi w następujący sposób:

XAML

```
<Button Name="Button1" Click="Button_Click">Click me</Button>
```

Jeśli zdarzenie nie jest członkiem klasy odbiornika, należy użyć kwalifikowanej nazwy zdarzenia w postaci `<owner type>.<event name>`. Na przykład, ponieważ StackPanel klasa nie implementuje Click zdarzenia, aby dołączyć procedurę obsługi do `StackPanel` `Click` zdarzenia, które bąbelkuje do tego elementu, należy użyć kwalifikowanej składni nazwy zdarzenia:

XAML

```
<StackPanel Name="StackPanel1" Button.Click="Button_Click">
    <Button>Click me</Button>
</StackPanel>
```

Podpis metody obsługi zdarzeń w kodzie za pomocą kodu musi być zgodny z typem delegata dla zdarzenia kierowanego. `sender` Parametr RoutedEventHandler delegata Click dla zdarzenia określa element, do którego jest dołączony program obsługi zdarzeń. `args` Parametr delegata `RoutedEventHandler` zawiera dane zdarzenia. Zgodna implementacja kodu dla procedury obsługi zdarzeń `Button_Click` może być:

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Click event logic.
}
```

Chociaż RoutedEventHandler jest podstawowym delegatem procedury obsługi zdarzeń kierowanych, niektóre kontrolki lub scenariusze implementacji wymagają różnych delegatów, które obsługują bardziej wyspecjalizowane dane zdarzeń. Na przykład w przypadku zdarzenia kierowanego DragEnter program obsługi powinien zaimplementować delegata DragEventHandler . Dzięki temu kod obsługi może uzyskać dostęp do DragEventArgs.Data właściwości w danych zdarzeń, która zawiera ładunek schowka z operacji przeciągania.

Składnia XAML do dodawania procedur obsługi zdarzeń trasowanych jest taka sama jak w przypadku standardowych procedur obsługi zdarzeń CLR. Aby uzyskać więcej informacji na temat dodawania programów obsługi zdarzeń w języku XAML, zobacz XAML w WPF. Pełny przykład dołączania procedury obsługi zdarzeń do elementu przy użyciu języka XAML można znaleźć w temacie How to handle a routed event (Jak obsługiwać zdarzenie kierowane).

Aby dołączyć program obsługi zdarzeń dla zdarzenia kierowanego do elementu przy użyciu kodu, zazwyczaj dostępne są dwie opcje:

- Bezpośrednio wywołaj metodę AddHandler . Programy obsługi zdarzeń trasowanych można zawsze dołączać w ten sposób. W tym przykładzie program obsługi zdarzeń jest dołączany `Click` do przycisku przy użyciu `AddHandler` metody :

  ```
  C#

  Button1.AddHandler(ButtonBase.ClickEvent, new
  RoutedEventHandler(Button_Click));
  ```

  Aby dołączyć program obsługi zdarzenia przycisku `Click` do innego elementu w trasie zdarzenia, na przykład StackPanel o nazwie `StackPanel1`:

  ```
  C#

  StackPanel1.AddHandler(ButtonBase.ClickEvent, new
  RoutedEventHandler(Button_Click));
  ```

- Jeśli kierowane zdarzenie implementuje otokę zdarzeń CLR, użyj składni zdarzeń specyficznych dla języka, aby dodać programy obsługi zdarzeń tak samo jak w przypadku standardowego zdarzenia CLR. Większość istniejących zdarzeń kierowanych WPF implementuje otokę CLR, włączając w ten sposób składnię zdarzeń specyficznych dla języka. Ten przykład dołącza procedurę `Click` obsługi zdarzeń do przycisku przy użyciu składni specyficznej dla języka:

  ```
  C#

  Button1.Click += Button_Click;
  ```

Aby zapoznać się z przykładem dołączania procedury obsługi zdarzeń w kodzie, zobacz How to add an event handler using code (Jak dodać procedurę obsługi zdarzeń przy użyciu kodu). Jeśli kodujesz w języku Visual Basic, możesz również użyć słowa kluczowego `Handles` , aby dodać programy obsługi w ramach deklaracji programu obsługi. Aby uzyskać więcej informacji, zobacz Obsługa zdarzeń w języku Visual Basic i WPF.

Koncepcja obsługi

Wszystkie zdarzenia kierowane mają wspólną klasę bazową dla danych zdarzeń, czyli klasę RoutedEventArgs . Klasa `RoutedEventArgs` definiuje właściwość logiczną Handled . Celem `Handled` właściwości jest umożliwianie programowi obsługi zdarzeń wzdłuż trasy

zdarzeń oznaczenie zdarzenia kierowanego jako *obsługiwanego*. Aby oznaczyć zdarzenie jako obsługiwane, ustaw wartość `Handled` na `true` w kodzie procedury obsługi zdarzeń.

Wartość elementu `Handled` wpływa na sposób przetwarzania zdarzenia kierowanego podczas podróży wzdłuż trasy zdarzeń. Jeśli `Handled` są `true` w udostępnionych danych zdarzeń zdarzenia kierowanego, programy obsługi dołączone do innych elementów dalej wzdłuż trasy zdarzeń zwykle nie będą wywoływane dla tego konkretnego wystąpienia zdarzenia. W przypadku najbardziej typowych scenariuszy obsługi oznaczanie zdarzenia jako obsługiwane skutecznie zatrzymuje kolejne programy obsługi wzdłuż trasy zdarzeń, niezależnie od tego, czy programy obsługi wystąpień lub klas, od reagowania na to konkretne wystąpienie zdarzenia. Jednak w rzadkich przypadkach, gdy potrzebujesz programu obsługi zdarzeń, aby reagować na kierowane zdarzenia, które zostały oznaczone jako obsługiwane, można:

- Dołącz procedurę obsługi w kodzie przy użyciu AddHandler(RoutedEvent, Delegate, Boolean) przeciążenia z parametrem ustawionym `handledEventsToo` na wartość `true`.
- Ustaw atrybut w elemecie HandledEventsToo`EventSetter` na `true`.

Koncepcja `Handled` może mieć wpływ na sposób projektowania aplikacji i kodowania procedur obsługi zdarzeń. Można koncepcyjnie `Handled` utworzyć prosty protokół do przetwarzania kierowanych zdarzeń. Sposób użycia tego protokołu jest do Ciebie, ale oczekiwane użycie parametru `Handled` to:

- Jeśli zdarzenie kierowane jest oznaczone jako obsługiwane, nie musi być ponownie obsługiwane przez inne elementy wzdłuż trasy.
- Jeśli zdarzenie kierowane nie jest oznaczone jako obsługiwane, odbiorniki wcześniej w trasie zdarzeń nie mają procedury obsługi zdarzenia lub żaden z zarejestrowanych programów obsługi nie odpowiedział na zdarzenie w sposób, który uzasadnia oznaczenie zdarzenia jako obsługiwane. Programy obsługi w bieżącym odbiorniku mają trzy możliwe kursy akcji:
  - Nie podejmuj żadnych działań. Zdarzenie pozostaje nieobsługiwane i kieruje je do następnego odbiornika w drzewie.
  - Uruchom kod w odpowiedzi na zdarzenie, ale nie w pewnym stopniu, który uzasadnia oznaczenie zdarzenia jako obsługiwanego. Zdarzenie pozostaje nieobsługiwane i kieruje je do następnego odbiornika w drzewie.
  - Uruchom kod w odpowiedzi na zdarzenie w zakresie, który uzasadnia oznaczenie zdarzenia jako obsługiwanego. Oznacz zdarzenie jako obsługiwane

w danych zdarzenia. Zdarzenie nadal kieruje do następnego odbiornika w drzewie, ale większość odbiorników nie będzie wywoływać dalszych procedur obsługi. Wyjątek to odbiorniki z procedurami obsługi, które zostały specjalnie zarejestrowane z ustawioną wartością `handledEventsToo` `true`.

Aby uzyskać więcej informacji na temat obsługi zdarzeń kierowanych, zobacz Oznaczanie zdarzeń trasowanych jako obsługiwane i obsługa klas.

Mimo że deweloperzy, którzy obsługują tylko rozsyłane zdarzenie na obiekcie, który go podniósł, mogą nie być zaniepokojeni innymi odbiornikami, dobrym rozwiązaniem jest oznaczenie zdarzenia jako obsłużonego mimo to. Uniemożliwia to nieprzewidziane skutki uboczne, jeśli element dalej wzdłuż trasy zdarzeń ma procedurę obsługi dla tego samego zdarzenia kierowanego.

Programy obsługi klas

Programy obsługi zdarzeń trasowanych mogą być procedurami obsługi *wystąpień* lub procedurami obsługi *klas* . Procedury obsługi klas dla danej klasy są wywoływane przed każdym procedurą obsługi wystąpień odpowiadającą na to samo zdarzenie w dowolnym wystąpieniu tej klasy. Ze względu na to zachowanie, gdy zdarzenia kierowane są oznaczone jako obsługiwane, są one często oznaczone jako takie w programach obsługi klas. Istnieją dwa typy programów obsługi klas:

- Programy obsługi zdarzeń klas statycznych, które są rejestrowane przez wywołanie RegisterClassHandler metody w konstruktorze klasy statycznej.
- Zastąpij programy obsługi zdarzeń klasy, które są rejestrowane przez zastępowanie metod zdarzeń wirtualnych klasy bazowej. Metody zdarzeń wirtualnych klasy bazowej istnieją przede wszystkim dla zdarzeń wejściowych i mają nazwy rozpoczynające się od **<w polu Nazwa> zdarzenia** i **Nazwa> zdarzenia OnPreview**<.

Niektóre kontrolki WPF mają nieodłączną obsługę klas dla niektórych zdarzeń trasowanych. Obsługa klas może dać wygląd na zewnątrz, że zdarzenie kierowane nie jest nigdy zgłaszane, ale w rzeczywistości jest oznaczone jako obsługiwane przez program obsługi klas. Jeśli potrzebujesz programu obsługi zdarzeń, aby odpowiedzieć na obsługiwane zdarzenie, możesz zarejestrować program obsługi z ustawioną wartością `handledEventsToo` `true`. Aby uzyskać więcej informacji na temat implementowania własnych procedur obsługi klas lub pracy wokół niepożądanej obsługi klas, zobacz Oznaczanie kierowanych zdarzeń jako obsłużonych i obsługa klas.

Zdarzenia dołączone w WPF

Język XAML definiuje również specjalny typ zdarzenia nazywany *dołączonym zdarzeniem*. Dołączone zdarzenia mogą służyć do definiowania nowego zdarzenia kierowanego w klasie innej niż element i zgłaszać to zdarzenie na dowolnym elemenie w drzewie. W tym celu należy zarejestrować dołączone zdarzenie jako zdarzenie kierowane i podać określony kod zapasowy , który obsługuje dołączone funkcje zdarzeń. Ponieważ dołączone zdarzenia są rejestrowane jako zdarzenia kierowane, po wywołaniu elementu są propagowane przez drzewo elementów.

W składni XAML dołączone zdarzenie jest określane przez jego nazwę zdarzenia *i* typ właściciela w postaci `<owner type>.<event name>`. Ponieważ nazwa zdarzenia jest kwalifikowana przy użyciu nazwy jego typu właściciela, składnia umożliwia dołączanie zdarzenia do dowolnego elementu, który można utworzyć wystąpienie. Ta składnia ma również zastosowanie do procedur obsługi zwykłych zdarzeń kierowanych, które dołączają do dowolnego elementu wzdłuż trasy zdarzeń. Można również dołączać programy obsługi dla dołączonych zdarzeń w kodzie, wywołując metodę AddHandler w obiekcie, do którego powinien zostać dołączony program obsługi.

System wejściowy WPF intensywnie używa dołączonych zdarzeń. Jednak prawie wszystkie dołączone zdarzenia są udostępniane jako równoważne nieprzyłączane zdarzenia kierowane przez elementy podstawowe. Rzadko będziesz używać dołączonych zdarzeń lub obsługiwać je bezpośrednio. Na przykład łatwiej jest obsłużyć bazowe dołączone Mouse.MouseDown zdarzenie na UIElement obiekcie za pośrednictwem równoważnego UIElement.MouseDown zdarzenia kierowanego niż za pomocą dołączonej składni zdarzeń w języku XAML lub kod-behind.

Aby uzyskać więcej informacji na temat dołączonych zdarzeń w WPF, zobacz Omówienie dołączonych zdarzeń.

Kwalifikowane nazwy zdarzeń w języku XAML

Składnia `<owner type>.<event name>` kwalifikuje nazwę zdarzenia z nazwą jego typu właściciela. Ta składnia umożliwia dołączanie zdarzenia do dowolnego elementu, a nie tylko elementów, które implementują zdarzenie jako element członkowski ich klasy. Składnia ma zastosowanie podczas dołączania programów obsługi w języku XAML dla dołączonych zdarzeń lub kierowanych zdarzeń na dowolnych elementach wzdłuż trasy zdarzeń. Rozważmy scenariusz, w którym chcesz dołączyć procedurę obsługi do elementu nadrzędnego w celu obsługi zdarzeń kierowanych zgłoszonych w elementach podrzędnych. Jeśli element nadrzędny nie ma zdarzenia kierowanego jako element członkowski, należy użyć składni kwalifikowanej nazwy zdarzenia. Przykład:

```
XAML
```

```
<StackPanel Name="StackPanel1" Button.Click="Button_Click">
    <Button>Click me</Button>
</StackPanel>
```

W tym przykładzie odbiornik elementu nadrzędnego, do którego dodano procedurę StackPanelobsługi zdarzeń, to . Jednak Click zdarzenie kierowane jest implementowane i wywoływane w ButtonBase klasie i dostępne dla Button klasy poprzez dziedziczenie. Button Mimo że klasa "jest właścicielem" `Click` zdarzenia, kierowany system zdarzeń zezwala na programy obsługi dla dowolnego kierowanego zdarzenia do dołączenia do dowolnego UIElement odbiornika lub ContentElement wystąpienia, które w przeciwnym razie mogą mieć programy obsługi dla zdarzenia CLR. Domyślną przestrzenią nazw tych kwalifikowanych nazw atrybutów zdarzeń jest zazwyczaj domyślna `xmlns` przestrzeń nazw WPF `xmlns` , ale można również określić prefiksowane przestrzenie nazw dla niestandardowych zdarzeń trasowanych. Aby uzyskać więcej informacji na temat `xmlns`programu , zobacz Przestrzenie nazw XAML i mapowanie przestrzeni nazw dla języka WPF XAML.

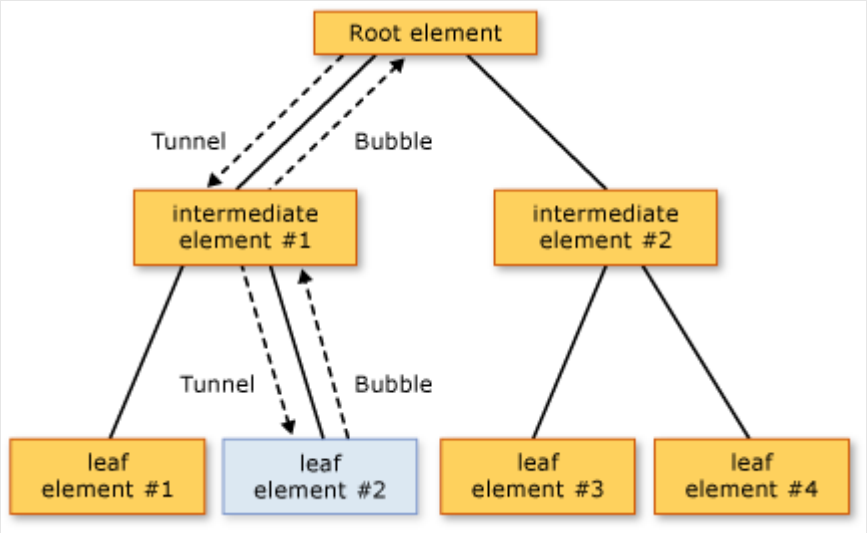
Zdarzenia wejściowe WPF

Jednym z częstych zastosowań kierowanych zdarzeń na platformie WPF są zdarzenia wejściowe. Zgodnie z konwencją zdarzenia kierowane przez platformę WPF, które są zgodne z trasą tunelowania, mają nazwę poprzedzoną prefiksem "Wersja zapoznawcza". Prefiks podglądu oznacza, że zdarzenie podglądu zostanie zakończone przed rozpoczęciem sparowanego zdarzenia bubbling. Zdarzenia wejściowe często występują w parach, z jednym zdarzeniem podglądu, a drugim zdarzeniem kierowanym przez bubbling. Na przykład PreviewKeyDown i KeyDown. Pary zdarzeń współużytkuje to samo wystąpienie danych zdarzenia, które dla `PreviewKeyDown` i `KeyDown` mają typ KeyEventArgs. Czasami zdarzenia wejściowe mają tylko wersję bubbling lub tylko wersję kierowaną bezpośrednio. W dokumentacji interfejsu API przekierowane tematy zdarzeń odwołujące się do wielu par zdarzeń i wyjaśnienie strategii routingu dla każdego kierowanego zdarzenia.

Zdarzenia wejściowe WPF, które występują w parach, są implementowane tak, aby pojedyncza akcja użytkownika z urządzenia wejściowego, taka jak naciśnięcie przycisku myszy, podnosi podgląd i bubbling kierowane zdarzenia w sekwencji. Najpierw zdarzenie w wersji zapoznawczej jest wywoływane i kończy swoją trasę. Po zakończeniu zdarzenia w wersji zapoznawczej zdarzenie bubbling jest wywoływane i kończy swoją trasę. Wywołanie RaiseEvent metody w klasie implementowania, która zgłasza zdarzenie bubbling ponownie używa danych zdarzenia ze zdarzenia podglądu dla zdarzenia bubbling.

Zdarzenie wejściowe podglądu oznaczone jako obsługiwane nie będzie wywoływać żadnych normalnie zarejestrowanych programów obsługi zdarzeń dla pozostałej części trasy podglądu, a sparowane zdarzenie bubbling nie zostanie podniesione. To zachowanie obsługi jest przydatne w przypadku projektantów kontrolek złożonych, którzy chcą, aby zdarzenia wejściowe oparte na teście trafień lub zdarzenia wejściowe oparte na fokusie były zgłaszane na najwyższym poziomie kontroli. Elementy najwyższego poziomu kontrolki mają możliwość obsługi zdarzeń podglądu klasy z podskładników sterowania, aby "zastąpić" je zdarzeniem specyficznym dla kontroli najwyższego poziomu.

Aby zilustrować działanie przetwarzania zdarzeń wejściowych, rozważmy następujący przykład zdarzenia wejściowego. Na poniższej ilustracji `leaf element #2` drzewa jest źródłem zdarzeń i `PreviewMouseDown` `MouseDown` sparowanych:



Kolejność przetwarzania zdarzeń po akcji myszy w dół elementu liścia #2 to:

1. `PreviewMouseDown` zdarzenie tunelowania w elemedycie głównym.
2. `PreviewMouseDown` zdarzenie tunelowania dla elementu pośredniego #1.
3. `PreviewMouseDown` zdarzenie tunelowania elementu liścia #2, który jest elementem źródłowym.
4. `MouseDown` Zdarzenie bubbling elementu liścia #2, który jest elementem źródłowym.
5. `MouseDown` Zdarzenie bubbling w elemecie pośrednim #1.
6. `MouseDown` Bubbling zdarzenie na elemecie głównym.

Delegat procedury obsługi zdarzeń kierowanych zawiera odwołania zarówno do obiektu, który wywołał zdarzenie, jak i obiektu, w którym wywoływano procedurę obsługi. Obiekt, który pierwotnie zgłosił zdarzenie, jest zgłaszany przez Source właściwość w danych zdarzenia. Obiekt, w którym wywoływano procedurę obsługi, jest zgłaszany przez parametr nadawcy . W przypadku każdego wystąpienia zdarzenia kierowanego obiekt,

który zgłosił zdarzenie, nie zmienia się, ponieważ zdarzenie przechodzi przez drzewo elementów, ale robi to `sender` . W krokach 3 i 4 powyższego diagramu `Source` obiekty i `sender` są tym samym obiektem.

Jeśli program obsługi zdarzeń wejściowych ukończy logikę specyficzną dla aplikacji wymaganą do rozwiązania zdarzenia, należy oznaczyć zdarzenie wejściowe jako obsługiwane. Zazwyczaj po oznaczeniu Handledzdarzenia wejściowego programy obsługi dalej wzdłuż trasy zdarzenia nie są wywoływane. Jednak programy obsługi zdarzeń wejściowych zarejestrowane za `handledEventsToo` pomocą zestawu `true` parametrów będą wywoływane nawet wtedy, gdy zdarzenie jest oznaczone jako obsługiwane. Aby uzyskać więcej informacji, zobacz Podgląd zdarzeń i Oznaczanie kierowanych zdarzeń jako obsługiwanych i obsługi klas.

Koncepcja par zdarzeń w wersji zapoznawczej i bubbling, z udostępnionymi danymi zdarzeń i sekwencyjnym podnoszeniem zdarzenia podglądu, a następnie zdarzeniem bubbling, ma zastosowanie tylko do niektórych zdarzeń wejściowych WPF, a nie do wszystkich kierowanych zdarzeń. Jeśli zaimplementujesz własne zdarzenie wejściowe w celu rozwiązania zaawansowanego scenariusza, rozważ zastosowanie podejścia pary zdarzeń wejściowych WPF.

Jeśli wdrażasz własną złożoną kontrolkę, która reaguje na zdarzenia wejściowe, rozważ użycie zdarzeń podglądu w celu pomijania i zastępowania zdarzeń wejściowych zgłoszonych w podskładników zdarzeniem najwyższego poziomu, które reprezentuje pełną kontrolę. Aby uzyskać więcej informacji, zobacz Oznaczanie trasowanych zdarzeń jako obsłużonych i obsługa klas.

Aby uzyskać więcej informacji na temat systemu wejściowego WPF oraz sposobu interakcji danych wejściowych i zdarzeń w typowych scenariuszach aplikacji, zobacz Omówienie danych wejściowych.

EventSetters i EventTriggers

W stylach znaczników można uwzględnić wstępnie zadeklarowaną składnię obsługi zdarzeń XAML przy użyciu elementu EventSetter. Po przetworzeniu kodu XAML program obsługi przywoływany jest dodawany do wystąpienia stylizowanego. Można zadeklarować `EventSetter` tylko dla zdarzenia kierowanego. W poniższym przykładzie metoda obsługi zdarzeń, do których odwołuje się odwołanie `ApplyButtonStyle` , jest implementowana w kodzie za pomocą kodu.

```
XAML
```

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="{x:Type Button}">
            <EventSetter Event="Click" Handler="ApplyButtonStyle"/>
        </Style>
    </StackPanel.Resources>
    <Button>Click me</Button>
    <Button Click="Button_Click">Click me</Button>
</StackPanel>
```

`Style` Prawdopodobnie węzeł zawiera już inne informacje o stylu, które odnoszą się do kontrolek określonego typu, i że EventSetter element jest częścią tych stylów, promuje ponowne użycie kodu nawet na poziomie znaczników. `EventSetter` Ponadto tworzy abstrakcję nazw metod obsługi z dala od ogólnej aplikacji i znaczników strony.

Inną wyspecjalizowaną składnią, która łączy trasowane zdarzenia i funkcje animacji WPF jest EventTrigger. Podobnie jak w przypadku `EventSetter` elementu , można zadeklarować `EventTrigger` tylko dla zdarzenia kierowanego. `EventTrigger` Zazwyczaj element jest zadeklarowany jako część stylu, ale `EventTrigger` można go zadeklarować w elementach na poziomie strony w ramach Triggers kolekcji lub w obiekcie ControlTemplate. Element `EventTrigger` umożliwia określenie Storyboard , które jest uruchamiane za każdym razem, gdy zdarzenie kierowane dociera do elementu w jego trasie, który deklaruje `EventTrigger` dla tego zdarzenia. Zaletą `EventTrigger` nieco obsługi zdarzenia i wywoływania go do uruchomienia istniejącej scenorysu jest to, że zapewnia lepszą `EventTrigger` kontrolę nad scenorysem i jego zachowaniem w czasie wykonywania. Aby uzyskać więcej informacji, zobacz Używanie wyzwalaczy zdarzeń do kontrolowania scenorysu po jego uruchomieniu.

Więcej informacji o zdarzeniach kierowanych

W tym artykule możesz użyć pojęć i wskazówek jako punktu wyjścia podczas tworzenia niestandardowych zdarzeń kierowanych we własnych klasach. Zdarzenia niestandardowe można również obsługiwać za pomocą wyspecjalizowanych klas danych zdarzeń i delegatów. Właściciel zdarzenia kierowanego może być dowolną klasą, ale zdarzenia kierowane muszą być wywoływane przez klasy i obsługiwane przez UIElement klasy lub ContentElement pochodne, aby były przydatne. Aby uzyskać więcej informacji na temat zdarzeń niestandardowych, zobacz Tworzenie niestandardowego zdarzenia kierowanego.

Zobacz też

- EventManager
- RoutedEvent
- RoutedEventArgs
- Oznaczanie zdarzeń trasowanych jako obsłużonych oraz obsługa klas
- Omówienie danych wejściowych
- Omówienie poleceń
- Niestandardowe właściwości zależności
- Drzewa w WPF
- Słabe wzorce zdarzeń

Omówienie dołączonych zdarzeń (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 6 min

Rozszerzalny język znaczników aplikacji (XAML) definiuje składnik języka i typ *zdarzenia nazywany dołączonym zdarzeniem*. Dołączone zdarzenia mogą służyć do definiowania nowego zdarzenia kierowanego w klasie innej niż element i zgłaszania tego zdarzenia na dowolnym elemenie w drzewie. W tym celu należy zarejestrować dołączone zdarzenie jako zdarzenie kierowane i podać konkretny kod kopii zapasowej , który obsługuje dołączone funkcje zdarzeń. Ponieważ dołączone zdarzenia są rejestrowane jako zdarzenia kierowane, gdy są wywoływane na elemecie, są propagowane przez drzewo elementów.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule przyjęto założenie, że podstawowa wiedza na temat zdarzeń kierowanych Windows Presentation Foundation (WPF) i że przeczytano omówienie zdarzeń trasowanych i XAML w WPF. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem XAML i wiedzieć, jak pisać aplikacje WPF.

Dołączona składnia zdarzeń

W składni XAML dołączone zdarzenie jest określane przez nazwę zdarzenia *i* jego typ właściciela w postaci `<owner type>.<event name>`. Ponieważ nazwa zdarzenia jest kwalifikowana przy użyciu nazwy jego typu właściciela, składnia umożliwia dołączanie zdarzenia do dowolnego elementu, który można utworzyć. Ta składnia ma również zastosowanie do procedur obsługi zwykłych zdarzeń kierowanych, które dołączają do dowolnego elementu wzdłuż trasy zdarzenia.

Następująca składnia atrybutów XAML dołącza procedurę `AquariumFilter_Clean` obsługi dołączonego `AquariumFilter.Clean` zdarzenia do `aquarium1` elementu:

```
XAML
```

```
<aqua:Aquarium x:Name="aquarium1" Height="300" Width="400"
aqua:AquariumFilter.Clean="AquariumFilter_Clean"/>
```

W tym przykładzie prefiks jest niezbędny, `aqua:` ponieważ `AquariumFilter` klasy i `Aquarium` istnieją w innej przestrzeni nazw środowiska uruchomieniowego języka wspólnego (CLR) i zestawie.

Można również dołączyć programy obsługi dla dołączonych zdarzeń w kodzie za nim. W tym celu wywołaj metodę AddHandler w obiekcie, do którego program obsługi powinien dołączyć i przekazać identyfikator zdarzenia i program obsługi jako parametry do metody.

Jak WPF implementuje dołączone zdarzenia

Zdarzenia dołączone do platformy WPF są implementowane jako zdarzenia kierowane wspierane przez RoutedEvent pole. W rezultacie dołączone zdarzenia są propagowane przez drzewo elementów po ich podniesieniu. Ogólnie rzecz biorąc, obiekt, który zgłasza dołączone zdarzenie, znane jako źródło zdarzeń, jest systemem lub źródłem usługi. Źródła systemu lub usługi nie są bezpośrednią częścią drzewa elementów. W przypadku innych dołączonych zdarzeń źródło zdarzeń może być elementem w drzewie, takim jak składnik w kontrolce złożonej.

Dołączone scenariusze zdarzeń

W środowisku WPF dołączone zdarzenia są używane w niektórych obszarach funkcji, w których istnieje abstrakcja na poziomie usługi. Na przykład WPF wykorzystuje dołączone zdarzenia włączone przez statyczne Mouse lub Validation klasy. Klasy, które wchodzą w interakcję z usługą lub korzystają z niej, mogą wchodzić w interakcje ze zdarzeniem przy użyciu dołączonej składni zdarzeń lub udostępniać dołączone zdarzenie jako zdarzenie kierowane. Druga opcja jest częścią tego, jak klasa może zintegrować możliwości usługi.

System wejściowy WPF intensywnie używa dołączonych zdarzeń. Jednak prawie wszystkie dołączone zdarzenia są wyświetlane jako równoważne nieprzyłączone zdarzenia kierowane przez elementy podstawowe. Każde zdarzenie wejściowe kierowane jest elementem członkowskim klasy podstawowego elementu i jest obsługiwane za pomocą zdarzenia CLR "otoka". Rzadko będziesz używać dołączonych zdarzeń lub obsługiwać je bezpośrednio. Na przykład łatwiej jest obsłużyć bazowe dołączone Mouse.MouseDown zdarzenie za UIElement pośrednictwem równoważnego UIElement.MouseDown zdarzenia kierowanego niż przy użyciu dołączonej składni zdarzeń w języku XAML lub kod-behind.

Dołączone zdarzenia służą do celów architektury, umożliwiając przyszłe rozszerzanie urządzeń wejściowych. Na przykład nowe urządzenie wejściowe będzie musiało podnieść `Mouse.MouseDown` tylko w celu symulowania danych wejściowych myszy i nie trzeba z niego korzystać `Mouse` . Ten scenariusz obejmuje obsługę kodu zdarzenia, ponieważ obsługa kodu XAML dołączonego zdarzenia nie byłaby odpowiednia.

Obsługa dołączonego zdarzenia

Proces kodowania i obsługi dołączonego zdarzenia jest zasadniczo taki sam jak w przypadku zdarzenia nieprzyłączonego kierowanego.

Jak wspomniano wcześniej, istniejące zdarzenia dołączone do WPF zwykle nie mają być obsługiwane bezpośrednio w WPF. Częściej celem dołączonego zdarzenia jest włączenie elementu w kontrolce złożonej w celu zgłoszenia stanu do elementu nadrzędnego w kontrolce. W tym scenariuszu zdarzenie jest wywoływane w kodzie i opiera się na obsłudze klas w odpowiedniej klasie nadrzędnej. Na przykład oczekuje się, że elementy w obiekcie Selector będą zgłaszać Selected dołączone zdarzenie, które jest następnie klasą obsługiwaną przez klasę `Selector` . Klasa `Selector` potencjalnie konwertuje `Selected` zdarzenie na SelectionChanged zdarzenie kierowane. Aby uzyskać więcej informacji o zdarzeniach kierowanych i obsłudze klas, zobacz Oznaczanie zdarzeń trasowanych jako obsługiwanych i obsługa klas.

Definiowanie niestandardowego dołączonego zdarzenia

Jeśli pochodzisz z typowych klas bazowych WPF, możesz zaimplementować niestandardowe dołączone zdarzenie, włączając dwie metody dostępu w klasie. Te metody to:

- **Metoda obsługi dodaj<nazwę>zdarzenia** z pierwszym parametrem, który jest elementem, na którym jest dołączony program obsługi zdarzeń, a drugi parametr, który jest programem obsługi zdarzeń do dodania. Metoda musi być `public` i `static`, bez wartości zwracanej. Metoda wywołuje metodę klasy bazowej AddHandler , przekazując zdarzenie kierowane i procedurę obsługi jako argumenty. Ta metoda obsługuje składnię atrybutów XAML do dołączania procedury obsługi zdarzeń do elementu. Ta metoda umożliwia również dostęp kodu do magazynu obsługi zdarzeń dla dołączonego zdarzenia.

- Metoda **obsługi usuń<nazwę>zdarzenia** z pierwszym parametrem, który jest elementem, na którym jest dołączony program obsługi zdarzeń, a drugi parametr,

który jest procedurą obsługi zdarzeń do usunięcia. Metoda musi być `public` i `static`, bez wartości zwracanej. Metoda wywołuje metodę klasy bazowej RemoveHandler , przekazując zdarzenie kierowane i procedurę obsługi jako argumenty. Ta metoda umożliwia dostęp kodu do magazynu programu obsługi zdarzeń dla dołączonego zdarzenia.

WPF implementuje dołączone zdarzenia jako zdarzenia kierowane, ponieważ identyfikator elementu RoutedEvent jest definiowany przez system zdarzeń WPF. Ponadto routing zdarzenia jest naturalnym rozszerzeniem koncepcji na poziomie języka XAML dołączonego zdarzenia. Ta strategia implementacji ogranicza obsługę dołączonych zdarzeń do UIElement klas pochodnych lub ContentElement klas pochodnych, ponieważ tylko te klasy mają AddHandler implementacje.

Na przykład poniższy kod definiuje `Clean` dołączone zdarzenie w `AquariumFilter` klasie właściciela, która nie jest klasą elementu. Kod definiuje dołączone zdarzenie jako zdarzenie kierowane i implementuje wymagane metody dostępu.

C#

```
public class AquariumFilter
{
    // Register a custom routed event using the bubble routing strategy.
    public static readonly RoutedEvent CleanEvent =
EventManager.RegisterRoutedEvent(
        "Clean", RoutingStrategy.Bubble, typeof(RoutedEventHandler),
typeof(AquariumFilter));

    // Provide an add handler accessor method for the Clean event.
    public static void AddCleanHandler(DependencyObject dependencyObject,
RoutedEventHandler handler)
    {
        if (dependencyObject is not UIElement uiElement)
            return;

        uiElement.AddHandler(CleanEvent, handler);
    }

    // Provide a remove handler accessor method for the Clean event.
    public static void RemoveCleanHandler(DependencyObject dependencyObject,
RoutedEventHandler handler)
    {
        if (dependencyObject is not UIElement uiElement)
            return;

        uiElement.RemoveHandler(CleanEvent, handler);
    }
}
```

Metoda RegisterRoutedEvent zwracająca dołączony identyfikator zdarzenia jest tą samą metodą używaną do rejestrowania nieprzyłączonych zdarzeń trasowanych. Zarówno dołączone, jak i nieprzyłączone zdarzenia są rejestrowane w scentralizowanym magazynie wewnętrznym. Ta implementacja magazynu zdarzeń umożliwia pojęcie "zdarzenia jako interfejs" omówione w temacie Przegląd zdarzeń trasowanych.

W przeciwieństwie do zdarzenia CLR "otoka" używanego do tworzenia kopii zapasowych zdarzeń nieprzyłączanych, dołączone metody dostępu do zdarzeń można zaimplementować w klasach, które nie pochodzą z UIElement lub ContentElement. Jest to możliwe, ponieważ dołączony kod kopii zapasowej zdarzenia wywołuje UIElement.AddHandler metody i UIElement.RemoveHandler przekazane w `UIElement` wystąpieniu. Z kolei otoka CLR dla nieprzyłączanych zdarzeń trasowanych wywołuje te metody bezpośrednio w klasie będącej właścicielem, aby klasa musiała pochodzić z `UIElement` klasy .

Wywoływanie zdarzenia dołączonego do platformy WPF

Proces zgłaszania dołączonego zdarzenia jest zasadniczo taki sam jak w przypadku zdarzenia nieprzyłączonego kierowanego.

Zazwyczaj kod nie będzie musiał zgłaszać żadnych istniejących zdarzeń dołączonych do platformy WPF, ponieważ te zdarzenia są zgodne z ogólnym modelem koncepcyjnym "usługi". W tym modelu klasy usług, takie jak InputManager, są odpowiedzialne za podnoszenie zdarzeń dołączonych WPF.

Podczas definiowania niestandardowego dołączonego zdarzenia przy użyciu modelu WPF opartego na dołączonych zdarzeniach trasowanych użyj UIElement.RaiseEvent metody , aby zgłosić dołączone zdarzenie na dowolnym UIElement lub ContentElement. Podczas podnoszenia zdarzenia kierowanego, niezależnie od tego, czy jest on dołączony, czy nie, musisz wyznaczyć element w drzewie elementów jako źródło zdarzeń. To źródło jest następnie zgłaszane jako obiekt wywołujący `RaiseEvent` . Aby na przykład zgłosić `AquariumFilter.Clean` dołączone zdarzenie kierowane w usłudze `aquarium1`:

```
C#

aquarium1.RaiseEvent(new RoutedEventArgs(AquariumFilter.CleanEvent));
```

W poprzednim przykładzie `aquarium1` jest źródłem zdarzeń.

Zobacz też

- Omówienie zdarzeń trasowanych
- Szczegółowo opisano składnię XAML
- Klasy XAML i niestandardowe dla WPF

Zdarzenia okresu istnienia obiektu (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 7 min

W okresie ich istnienia wszystkie obiekty w kodzie zarządzanym platformy Microsoft .NET przechodzą przez *etapy tworzenia*, *używania* i *niszczenia* . Windows Presentation Foundation (WPF) dostarcza powiadomienia o tych etapach, ponieważ występują one w obiekcie, wywołując zdarzenia okresu istnienia. W przypadku elementów na poziomie struktury WPF (obiektów wizualnych) WPF implementuje Initializedzdarzenia , Loadedi Unloaded okres istnienia. Deweloperzy mogą używać tych zdarzeń okresu istnienia jako punktów zaczepienia do operacji związanych z kodem, które obejmują elementy. W tym artykule opisano zdarzenia okresu istnienia obiektów wizualizacji, a następnie wprowadzono inne zdarzenia okresu istnienia, które mają zastosowanie do elementów okna, hostów nawigacji lub obiektów aplikacji.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat sposobu, w jaki układ elementów WPF może być koncepcyjny jako drzewo i że przeczytano omówienie zdarzeń trasowanych. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem XAML (Extensible Application Markup Language) i wiedzieć, jak pisać aplikacje WPF.

Zdarzenia okresu istnienia dla obiektów wizualizacji

Elementy na poziomie struktury WPF pochodzą z FrameworkElement elementu lub FrameworkContentElement. Zdarzenia Initialized, Loadedi Unloaded okresu istnienia są wspólne dla wszystkich elementów na poziomie platformy WPF. W poniższym przykładzie pokazano drzewo elementów, które jest implementowane głównie w języku XAML. Język XAML definiuje element nadrzędny Canvas , który zawiera zagnieżdżone

elementy, z których każda używa składni atrybutu XAML do dołączania `Initialized` programów obsługi zdarzeń , `Loaded` i `Unloaded` okresu istnienia.

XAML

```
<Canvas x:Name="canvas">
    <StackPanel x:Name="outerStackPanel" Initialized="InitHandler"
Loaded="LoadHandler" Unloaded="UnloadHandler">
        <custom:ComponentWrapper x:Name="componentWrapper"
Initialized="InitHandler" Loaded="LoadHandler" Unloaded="UnloadHandler">
            <TextBox Name="textBox1" Initialized="InitHandler"
Loaded="LoadHandler" Unloaded="UnloadHandler" />
            <TextBox Name="textBox2" Initialized="InitHandler"
Loaded="LoadHandler" Unloaded="UnloadHandler" />
        </custom:ComponentWrapper>
    </StackPanel>
    <Button Content="Remove canvas child elements" Click="Button_Click"/>
</Canvas>
```

Jednym z elementów XAML jest kontrolka niestandardowa, która pochodzi z klasy bazowej, która przypisuje programy obsługi zdarzeń okresu istnienia w kodzie.

C#

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

    // Handler for the Initialized lifetime event (attached in XAML).
    private void InitHandler(object sender, System.EventArgs e) =>
        Debug.WriteLine($"Initialized event on
{((FrameworkElement)sender).Name}.");

    // Handler for the Loaded lifetime event (attached in XAML).
    private void LoadHandler(object sender, RoutedEventArgs e) =>
        Debug.WriteLine($"Loaded event on
{((FrameworkElement)sender).Name}.");

    // Handler for the Unloaded lifetime event (attached in XAML).
    private void UnloadHandler(object sender, RoutedEventArgs e) =>
        Debug.WriteLine($"Unloaded event on
{((FrameworkElement)sender).Name}.");

    // Remove nested controls.
    private void Button_Click(object sender, RoutedEventArgs e) =>
        canvas.Children.Clear();
}

// Custom control.
public class ComponentWrapper : ComponentWrapperBase { }

// Custom base control.
```

```
public class ComponentWrapperBase : StackPanel
{
    public ComponentWrapperBase()
    {
        // Assign handler for the Initialized lifetime event (attached in
code-behind).
        Initialized += (object sender, System.EventArgs e) =>
            Debug.WriteLine($"Initialized event on componentWrapperBase.");

        // Assign handler for the Loaded lifetime event (attached in code-
behind).
        Loaded += (object sender, RoutedEventArgs e) =>
            Debug.WriteLine($"Loaded event on componentWrapperBase.");

        // Assign handler for the Unloaded lifetime event (attached in code-
behind).
        Unloaded += (object sender, RoutedEventArgs e) =>
            Debug.WriteLine($"Unloaded event on componentWrapperBase.");
    }
}

/* Output:
Initialized event on textBox1.
Initialized event on textBox2.
Initialized event on componentWrapperBase.
Initialized event on componentWrapper.
Initialized event on outerStackPanel.

Loaded event on outerStackPanel.
Loaded event on componentWrapperBase.
Loaded event on componentWrapper.
Loaded event on textBox1.
Loaded event on textBox2.

Unloaded event on outerStackPanel.
Unloaded event on componentWrapperBase.
Unloaded event on componentWrapper.
Unloaded event on textBox1.
Unloaded event on textBox2.
*/
```

Dane wyjściowe programu pokazują kolejność wywołań `Initialized` zdarzeń , `Loaded` i `Unloaded` okresu istnienia dla każdego obiektu drzewa. Te zdarzenia są opisane w poniższych sekcjach w kolejności, w której są wywoływane na każdym obiekcie drzewa.

Zdarzenie zainicjowanego okresu istnienia

System zdarzeń WPF zgłasza Initialized zdarzenie dla elementu:

- Po ustawieniu właściwości elementu.

- Mniej więcej w tym samym czasie, gdy obiekt jest inicjowany przez wywołanie jego konstruktora.

Niektóre właściwości elementów, takie jak Panel.Children, mogą zawierać elementy podrzędne. Elementy nadrzędne nie mogą zgłaszać inicjowania, dopóki ich elementy podrzędne nie zostaną zainicjowane. Dlatego wartości właściwości są ustawiane od najbardziej głęboko zagnieżdżonych elementów w drzewie elementów, a następnie kolejnych elementów nadrzędnych do katalogu głównego aplikacji. `Initialized` Ponieważ zdarzenie występuje, gdy właściwości elementu są ustawione, to zdarzenie jest najpierw wywoływane na najbardziej głęboko zagnieżdżonych elementach zgodnie z definicją w znacznikach, a następnie kolejne elementy nadrzędne do katalogu głównego aplikacji. Gdy obiekty są dynamicznie tworzone w kodzie, ich inicjowanie może nie być sekwencyjne.

System zdarzeń WPF nie czeka na zainicjowanie wszystkich elementów w drzewie elementów przed wywołaniem `Initialized` zdarzenia na elemecie. Dlatego podczas pisania `Initialized` procedury obsługi zdarzeń dla dowolnego elementu należy pamiętać, że otaczające elementy w drzewie logicznym lub wizualnym, szczególnie elementy nadrzędne, mogły nie zostać utworzone. Lub ich zmienne składowe i powiązania danych mogą być niezainicjowane.

> ⓘ **Uwaga**
>
> Gdy `Initialized` zdarzenie jest wywoływane na elemecie, użycie wyrażeń elementu, takie jak zasoby dynamiczne lub powiązanie, będzie nieszacowane.

Załadowane zdarzenie okresu istnienia

System zdarzeń WPF zgłasza Loaded zdarzenie dla elementu:

- Gdy drzewo logiczne zawierające element jest ukończone i połączone ze źródłem prezentacji. Źródło prezentacji zapewnia dojście do okna (HWND) i powierzchni renderowania.
- Gdy powiązanie danych ze źródłami lokalnymi, takie jak inne właściwości lub bezpośrednio zdefiniowane źródła danych, jest kompletne.
- Po obliczeniu wszystkich wartości niezbędnych do renderowania przez system układu.
- Przed ostatecznym renderowaniem.

Zdarzenie `Loaded` nie jest wywoływane dla żadnego elementu w drzewie elementów, dopóki *wszystkie* elementy w drzewie logicznym nie zostaną załadowane. System

zdarzeń WPF najpierw podnosi `Loaded` zdarzenie na głównym elemenie drzewa elementów, a następnie na każdym kolejnym elemecie podrzędnym w dół do najbardziej głęboko zagnieżdżonych elementów. Chociaż to zdarzenie może przypominać zdarzenie kierowane przez tunelowanie , `Loaded` zdarzenie nie przenosi danych zdarzeń z jednego elementu do drugiego, dlatego oznaczanie zdarzenia jako obsłużonego nie ma wpływu.

> ⓘ **Uwaga**
>
> System zdarzeń WPF nie może zagwarantować, że powiązania danych asynchronicznych zostały ukończone przed zdarzeniem `Loaded` . Powiązania danych asynchronicznych wiążą się ze źródłami zewnętrznymi lub dynamicznymi.

Zdarzenie rozładowanego okresu istnienia

System zdarzeń WPF zgłasza Unloaded zdarzenie dla elementu:

- Po usunięciu źródła prezentacji lub
- Po usunięciu jej elementu nadrzędnego wizualizacji.

System zdarzeń WPF najpierw podnosi `Unloaded` zdarzenie na głównym elemenie drzewa elementów, a następnie na każdym kolejnym elemecie podrzędnym w dół do najbardziej głęboko zagnieżdżonych elementów. Mimo że to zdarzenie może przypominać zdarzenie kierowane przez tunelowanie , `Unloaded` zdarzenie nie propaguje danych zdarzeń z elementu do elementu, dlatego oznaczanie zdarzenia jako obsługiwanego nie ma wpływu.

`Unloaded` Gdy zdarzenie jest wywoływane na elemecie, jest elementem nadrzędnym lub dowolnym elementem wyższym w drzewie logicznym lub wizualnym, być może nie został już *zdenerwowany*. Nie ustawiono oznacza, że powiązania danych, odwołania do zasobów i style elementu nie są już ustawione na normalną lub ostatnią znaną wartość czasu wykonywania.

Inne zdarzenia okresu istnienia

Z perspektywy zdarzeń okresu istnienia istnieją cztery główne typy obiektów WPF: elementy ogólne, elementy okna, hosty nawigacji i obiekty aplikacji. Zdarzenia Initialized, Loadedi Unloaded okresu istnienia mają zastosowanie do wszystkich elementów na poziomie platformy. Inne zdarzenia okresu istnienia dotyczą konkretnie

elementów okna, hostów nawigacji lub obiektów aplikacji. Aby uzyskać informacje o tych innych zdarzeniach istnienia, zobacz:

- Omówienie zarządzania aplikacjami dla Application obiektów.
- Omówienie okien WPF dla Window elementów.
- Omówienie nawigacji dla Pageelementów , NavigationWindowi Frame .

Zobacz też

- Initialized
- Loaded
- Unloaded
- Obsługa załadowanego zdarzenia
- Załadowane zdarzenie i zdarzenie zainicjowane
- Drzewa w WPF
- Omówienie zdarzeń trasowanych

Oznaczanie kierowanych zdarzeń jako obsługiwanych i obsługi klas (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 12 min

Chociaż nie ma bezwzględnej reguły, kiedy oznaczyć zdarzenie kierowane jako obsługiwane, rozważ oznaczenie zdarzenia jako obsługiwanego, jeśli kod reaguje na zdarzenie w znaczący sposób. Zdarzenie kierowane oznaczone jako obsługiwane będzie kontynuowane wzdłuż trasy, ale wywoływane są tylko programy obsługi skonfigurowane do reagowania na obsługiwane zdarzenia. Zasadniczo oznaczanie kierowanego zdarzenia jako obsłużonego ogranicza widoczność odbiorników wzdłuż trasy zdarzeń.

Programy obsługi zdarzeń trasowanych mogą być procedurami obsługi wystąpień lub procedurami obsługi klas. Programy obsługi wystąpień obsługują kierowane zdarzenia na obiektach lub elementach XAML. Programy obsługi klas obsługują zdarzenie kierowane na poziomie klasy i są wywoływane przed każdym procedurą obsługi wystąpień odpowiadającą na to samo zdarzenie w dowolnym wystąpieniu klasy. Gdy zdarzenia kierowane są oznaczone jako obsługiwane, są one często oznaczone jako takie w programach obsługi klas. W tym artykule omówiono korzyści i potencjalne pułapki oznaczania trasowanych zdarzeń zgodnie z procedurą obsługi, różne typy kierowanych zdarzeń i kierowane programy obsługi zdarzeń oraz pomijanie zdarzeń w kontrolkach złożonych.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat kierowanych zdarzeń i przeczytano omówienie zdarzeń trasowanych. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje Windows Presentation Foundation (WPF).

Kiedy oznaczyć zdarzenia kierowane jako obsługiwane

Zazwyczaj tylko jedna procedura obsługi powinna zapewnić znaczącą odpowiedź dla każdego kierowanego zdarzenia. Unikaj używania systemu zdarzeń kierowanych, aby zapewnić znaczącą odpowiedź w wielu programach obsługi. Definicja tego, co stanowi znaczącą reakcję, jest subiektywna i zależy od twojej aplikacji. Ogólne wskazówki:

- Istotne odpowiedzi obejmują ustawianie fokusu, modyfikowanie stanu publicznego, ustawianie właściwości wpływających na reprezentację wizualną, podnoszenie nowych zdarzeń i całkowite obsługiwanie zdarzenia.
- Nieistotne odpowiedzi obejmują modyfikowanie stanu prywatnego bez wpływu wizualnego lub programowego, rejestrowanie zdarzeń i badanie danych zdarzeń bez reagowania na zdarzenie.

Niektóre kontrolki WPF pomijają zdarzenia na poziomie składników, które nie wymagają dalszej obsługi, oznaczając je jako obsługiwane. Jeśli chcesz obsłużyć zdarzenie oznaczone jako obsługiwane przez kontrolkę, zobacz Praca nad pomijaniem zdarzeń przez kontrolki.

Aby oznaczyć zdarzenie jako *obsługiwane*, ustaw Handled wartość właściwości w danych zdarzenia na `true`wartość . Chociaż możliwe jest przywrócenie tej wartości do `false`wartości , należy to zrobić rzadko.

Podgląd i bubbling trasowane pary zdarzeń

Pary zdarzeń kierowanych w wersji zapoznawczej i bubbling są specyficzne dla zdarzeń wejściowych. Kilka zdarzeń wejściowych implementuje tunelowanie i bubbling kierowane pary zdarzeń, takich jak PreviewKeyDown i KeyDown. Prefiks `Preview` oznacza, że zdarzenie bubbling rozpoczyna się po zakończeniu zdarzenia podglądu. Każda para zdarzeń w wersji zapoznawczej i bubbling współudzieli to samo wystąpienie danych zdarzenia.

Programy obsługi zdarzeń trasowanych są wywoływane w kolejności odpowiadającej strategii routingu zdarzenia:

1. Zdarzenie podglądu jest przesyłane z elementu głównego aplikacji w dół do elementu, który wywołał zdarzenie kierowane. Podgląd programów obsługi zdarzeń dołączonych do elementu głównego aplikacji są wywoływane jako pierwsze, a następnie programy obsługi dołączone do kolejnych zagnieżdżonych elementów.
2. Po zakończeniu zdarzenia podglądu sparowane zdarzenie bubbling jest przesyłane z elementu, który podniósł zdarzenie kierowane do elementu głównego aplikacji. Programy obsługi zdarzeń bubbling dołączone do tego samego elementu, który

wywołał zdarzenie kierowane, są wywoływane jako pierwsze, a następnie programy obsługi dołączone do kolejnych elementów nadrzędnych.

Sparowane zdarzenia podglądu i bubbling są częścią wewnętrznej implementacji kilku klas WPF, które deklarują i zgłaszają własne kierowane zdarzenia. Bez tej wewnętrznej implementacji na poziomie klasy zdarzenia kierowane w wersji zapoznawczej i bubbling są całkowicie oddzielone i nie będą udostępniać danych zdarzeń — niezależnie od nazewnictwa zdarzeń. Aby uzyskać informacje na temat implementowania zdarzeń kierowanych przez elementy wejściowe lub tunelowania w klasie niestandardowej, zobacz Tworzenie niestandardowego zdarzenia kierowanego.

Ponieważ każda para zdarzeń w wersji zapoznawczej i bubbling współudzieli to samo wystąpienie danych zdarzeń, jeśli zdarzenie kierowane w wersji zapoznawczej jest oznaczone jako obsługiwane, zostanie również obsłużone sparowane zdarzenie bubbling. Jeśli zdarzenie kierowane przez bubbling jest oznaczone jako obsługiwane, nie będzie miało to wpływu na sparowane zdarzenie podglądu, ponieważ zdarzenie podglądu zostało ukończone. Podczas oznaczania podglądu i bubbling wejściowych par zdarzeń zgodnie z obsługą. Obsługiwane zdarzenie wejściowe podglądu nie będzie wywoływać żadnych normalnie zarejestrowanych programów obsługi zdarzeń dla pozostałej części trasy tunelowania, a sparowane zdarzenie bubbling nie zostanie podniesione. Obsługiwane zdarzenie wejściowe bubbling nie będzie wywoływać żadnych normalnie zarejestrowanych programów obsługi zdarzeń dla pozostałej części trasy bubbling.

Procedury obsługi zdarzeń trasowanych wystąpień i klas

Programy obsługi zdarzeń trasowanych mogą być procedurami obsługi *wystąpień* lub procedurami obsługi *klas* . Procedury obsługi klas dla danej klasy są wywoływane przed każdym procedurą obsługi wystąpień odpowiadającą na to samo zdarzenie w dowolnym wystąpieniu tej klasy. Ze względu na to zachowanie, gdy zdarzenia kierowane są oznaczone jako obsługiwane, są one często oznaczone jako takie w programach obsługi klas. Istnieją dwa typy programów obsługi klas:

- Programy obsługi zdarzeń klas statycznych, które są rejestrowane przez wywołanie RegisterClassHandler metody w konstruktorze klasy statycznej.
- Zastąpij programy obsługi zdarzeń klasy, które są rejestrowane przez zastępowanie metod zdarzeń wirtualnych klasy bazowej. Metody zdarzeń wirtualnych klasy bazowej istnieją przede wszystkim dla zdarzeń wejściowych i mają nazwy rozpoczynające się od **<w polu Nazwa> zdarzenia** i **Nazwa> zdarzenia OnPreview<**.

Programy obsługi zdarzeń wystąpień

Programy obsługi wystąpień można dołączać do obiektów lub elementów XAML, wywołując metodę AddHandler bezpośrednio. Zdarzenia kierowane w programie WPF implementują otokę zdarzeń środowiska uruchomieniowego języka wspólnego (CLR), która używa `AddHandler` metody do dołączania programów obsługi zdarzeń. Ponieważ składnia atrybutów XAML do dołączania programów obsługi zdarzeń powoduje wywołanie otoki zdarzeń CLR, nawet dołączanie programów obsługi w języku XAML jest rozpoznawane jako `AddHandler` wywołanie. W przypadku zdarzeń obsługiwanych:

- Programy obsługi dołączone przy użyciu składni atrybutów XAML lub wspólnego podpisu `AddHandler` nie są wywoływane.
- Wywoływane są programy obsługi dołączone przy użyciu AddHandler(RoutedEvent, Delegate, Boolean) przeciążenia z ustawionym parametrem `handledEventsToo` `true` . To przeciążenie jest dostępne w rzadkich przypadkach, gdy konieczne jest reagowanie na obsługiwane zdarzenia. Na przykład niektóre elementy w drzewie elementów oznaczyły zdarzenie jako obsługiwane, ale inne elementy dalej wzdłuż trasy zdarzenia muszą odpowiadać na obsłużone zdarzenie.

Poniższy przykładowy kod XAML dodaje niestandardową kontrolkę o nazwie `componentWrapper`, która opakowuje TextBox nazwę `componentTextBox` do StackPanel nazwy `outerStackPanel`. Procedura obsługi zdarzeń wystąpienia dla PreviewKeyDown zdarzenia jest dołączana do składni atrybutu `componentWrapper` XAML. W związku z tym program obsługi wystąpień będzie reagować tylko na nieobsługiwane `PreviewKeyDown` zdarzenia tunelowania zgłaszane przez program `componentTextBox`.

XAML

```
<StackPanel Name="outerStackPanel" VerticalAlignment="Center">
    <custom:ComponentWrapper
        x:Name="componentWrapper"
        TextBox.PreviewKeyDown="HandlerInstanceEventInfo"
        HorizontalAlignment="Center">
        <TextBox Name="componentTextBox" Width="200" />
    </custom:ComponentWrapper>
</StackPanel>
```

Konstruktor `MainWindow` dołącza program obsługi wystąpienia dla `KeyDown` zdarzenia bubbling do `componentWrapper` przy użyciu UIElement.AddHandler(RoutedEvent, Delegate, Boolean) przeciążenia, z parametrem ustawionym `handledEventsToo` na `true`. W rezultacie program obsługi zdarzeń wystąpienia będzie reagować zarówno na nieobsługiwane, jak i obsługiwane zdarzenia.

C#

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

        // Attach an instance handler on componentWrapper that will be
invoked by handled KeyDown events.
        componentWrapper.AddHandler(KeyDownEvent, new
RoutedEventHandler(Handler.InstanceEventInfo),
            handledEventsToo: true);
    }

    // The handler attached to componentWrapper in XAML.
    public void HandlerInstanceEventInfo(object sender, KeyEventArgs e) =>
        Handler.InstanceEventInfo(sender, e);
}
```

Implementacja kodu w pliku jest wyświetlana `ComponentWrapper` w następnej sekcji.

Programy obsługi zdarzeń klas statycznych

Programy obsługi zdarzeń klas statycznych można dołączać, wywołując metodę RegisterClassHandler w konstruktorze statycznym klasy. Każda klasa w hierarchii klas może zarejestrować własną program obsługi klas statycznych dla każdego kierowanego zdarzenia. W związku z tym może istnieć wiele statycznych procedur obsługi klas wywoływanych dla tego samego zdarzenia w dowolnym węźle w trasie zdarzeń. Po utworzeniu trasy zdarzeń dla zdarzenia wszystkie programy obsługi klas statycznych dla każdego węzła są dodawane do trasy zdarzeń. Kolejność wywołań procedur obsługi klas statycznych w węźle rozpoczyna się od najbardziej pochodnej procedury obsługi klas statycznych, a następnie statycznych procedur obsługi klas z każdej kolejnej klasy bazowej.

Programy obsługi zdarzeń klasy statycznej zarejestrowane przy użyciu RegisterClassHandler(Type, RoutedEvent, Delegate, Boolean) przeciążenia z zestawem `handledEventsToo` parametrów będą reagować `true` zarówno na nieobsługiwane, jak i obsługiwane zdarzenia kierowane.

Programy obsługi klas statycznych są zwykle rejestrowane w celu reagowania tylko na nieobsługiwane zdarzenia. W takim przypadku, jeśli program obsługi klas pochodnych w węźle oznacza zdarzenie jako obsługiwane, programy obsługi klas bazowych dla tego zdarzenia nie będą wywoływane. W tym scenariuszu program obsługi klas bazowych jest skutecznie zastępowany przez program obsługi klas pochodnych. Programy obsługi klas

bazowych często przyczyniają się do sterowania projektowaniem w obszarach, takich jak wygląd wizualizacji, logika stanu, obsługa danych wejściowych i obsługa poleceń, dlatego należy zachować ostrożność podczas ich zastępowania. Programy obsługi klas pochodnych, które nie oznaczają zdarzenia jako obsługiwanego, uzupełniają programy obsługi klas bazowych zamiast ich zastępowania.

Poniższy przykładowy kod przedstawia hierarchię klas dla kontrolki niestandardowej `ComponentWrapper` , do którego odwołuje się poprzedni kod XAML. Klasa `ComponentWrapper` pochodzi z `ComponentWrapperBase` klasy, która z kolei pochodzi z StackPanel klasy . Metoda `RegisterClassHandler` używana w konstruktorze `ComponentWrapper` statycznym klas i `ComponentWrapperBase` rejestruje program obsługi zdarzeń klasy statycznej dla każdej z tych klas. System zdarzeń WPF wywołuje `ComponentWrapper` program obsługi klas statycznych przed `ComponentWrapperBase` programem obsługi klas statycznych.

C#

```
public class ComponentWrapper : ComponentWrapperBase
{
    static ComponentWrapper()
    {
        // Class event handler implemented in the static constructor.
        EventManager.RegisterClassHandler(typeof(ComponentWrapper),
KeyDownEvent,
            new RoutedEventHandler(Handler.ClassEventInfo_Static));
    }

    // Class event handler that overrides a base class virtual method.
    protected override void OnKeyDown(KeyEventArgs e)
    {
        Handler.ClassEventInfo_Override(this, e);

        // Call the base OnKeyDown implementation on ComponentWrapperBase.
        base.OnKeyDown(e);
    }
}

public class ComponentWrapperBase : StackPanel
{
    // Class event handler implemented in the static constructor.
    static ComponentWrapperBase()
    {
        EventManager.RegisterClassHandler(typeof(ComponentWrapperBase),
KeyDownEvent,
            new RoutedEventHandler(Handler.ClassEventInfoBase_Static));
    }

    // Class event handler that overrides a base class virtual method.
    protected override void OnKeyDown(KeyEventArgs e)
```

```
    {
        Handler.ClassEventInfoBase_Override(this, e);

        e.Handled = true;
        Debug.WriteLine("The KeyDown routed event is marked as handled.");

        // Call the base OnKeyDown implementation on StackPanel.
        base.OnKeyDown(e);
    }
}
```

Implementacja obsługi zdarzeń klasy zastępowania w tym przykładzie kodu jest omówiona w następnej sekcji.

Zastąpij programy obsługi zdarzeń klasy

Niektóre klasy bazowe elementów wizualizacji uwidaczniają puste wartości **W<nazwach> zdarzeń** i **OnPreview<nazwy> zdarzeń** wirtualnych dla każdego z ich publicznych kierowanych zdarzeń wejściowych. Na przykład UIElement implementuje OnKeyDown programy obsługi zdarzeń wirtualnych i OnPreviewKeyDown oraz wiele innych. Można zastąpić programy obsługi zdarzeń wirtualnych klasy bazowej, aby zaimplementować programy obsługi zdarzeń klasy zastępowania dla klas pochodnych. Można na przykład dodać program obsługi klasy zastąpienia dla DragEnter zdarzenia w dowolnej `UIElement` klasie pochodnej, przesłaniając metodę wirtualną OnDragEnter . Zastępowanie metod wirtualnych klasy bazowej to prostszy sposób implementowania procedur obsługi klas niż rejestrowanie procedur obsługi klas w konstruktorze statycznym. W ramach przesłonięcia można zgłaszać zdarzenia, inicjować logikę specyficzną dla klasy, aby zmienić właściwości elementu w wystąpieniach, oznaczyć zdarzenie jako obsługiwane lub wykonać inną logikę obsługi zdarzeń.

W przeciwieństwie do programów obsługi zdarzeń klas statycznych system zdarzeń WPF wywołuje tylko programy obsługi zdarzeń klasy zastępowania dla najbardziej pochodnej klasy w hierarchii klas. Najbardziej pochodna klasa w hierarchii klas może następnie użyć podstawowego słowa kluczowego do wywołania podstawowej implementacji metody wirtualnej. W większości przypadków należy wywołać implementację podstawową, niezależnie od tego, czy zdarzenie jest oznaczone jako obsługiwane. Należy pominąć wywoływanie implementacji podstawowej tylko wtedy, gdy klasa ma wymaganie zastąpienia podstawowej logiki implementacji, jeśli istnieje. Niezależnie od tego, czy implementacja podstawowa jest wywoływana przed zastąpieniem, czy po nim, zależy od charakteru implementacji.

W poprzednim przykładzie kodu metoda wirtualna klasy `OnKeyDown` bazowej jest zastępowana zarówno w klasach , jak `ComponentWrapper` i `ComponentWrapperBase` .

Ponieważ system zdarzeń WPF wywołuje tylko program obsługi zdarzeń klasy zastąpienia, ta procedura obsługi używa `base.OnKeyDown(e)` do wywoływania `StackPanel.OnKeyDown` `ComponentWrapperBase.OnKeyDown` programu obsługi zdarzeń klasy przesłonięcia `ComponentWrapper.OnKeyDown`, który z kolei używa `base.OnKeyDown(e)` metody wirtualnej. Kolejność zdarzeń w poprzednim przykładzie kodu to:

1. Program obsługi wystąpień dołączony do `componentWrapper` jest wyzwalany przez `PreviewKeyDown` zdarzenie kierowane.
2. Statyczny program obsługi klas dołączony do `componentWrapper` jest wyzwalany przez `KeyDown` kierowane zdarzenie.
3. Statyczny program obsługi klas dołączony do `componentWrapperBase` jest wyzwalany przez `KeyDown` kierowane zdarzenie.
4. Przesłonięć program obsługi klas dołączony do `componentWrapper` jest wyzwalany przez `KeyDown` zdarzenie kierowane.
5. Przesłonięć program obsługi klas dołączony do `componentWrapperBase` jest wyzwalany przez `KeyDown` zdarzenie kierowane.
6. Zdarzenie `KeyDown` kierowane jest oznaczone jako obsługiwane.
7. Program obsługi wystąpień dołączony do `componentWrapper` jest wyzwalany przez `KeyDown` zdarzenie kierowane. Procedura obsługi została zarejestrowana przy użyciu parametru ustawionego `handledEventsToo` na wartość `true`.

Pomijanie zdarzeń wejściowych w kontrolkach złożonych

Niektóre kontrolki złożone pomijają zdarzenia wejściowe na poziomie składnika, aby zastąpić je niestandardowym zdarzeniem wysokiego poziomu, które niesie więcej informacji lub implikuje bardziej szczegółowe zachowanie. Kontrolka złożona składa się z wielu praktycznych kontrolek lub klas bazowych kontrolek. Klasycznym przykładem jest kontrolka Button , która przekształca różne zdarzenia myszy w Click zdarzenie kierowane. Klasa `Button` bazowa to ButtonBase, która pośrednio pochodzi z klasy UIElement. Większość infrastruktury zdarzeń potrzebnej do sterowania przetwarzaniem wejściowym jest dostępna na `UIElement` poziomie. `UIElement` uwidacznia kilka Mouse zdarzeń, takich jak MouseLeftButtonDown i MouseRightButtonDown. `UIElement` Implementuje również puste metody OnMouseLeftButtonDown wirtualne i OnMouseRightButtonDown jako wstępniezarejestrowane programy obsługi klas. `ButtonBase` zastępuje te programy obsługi klas, a w ramach programu obsługi zastąpienia ustawia Handled właściwość na `true` i zgłasza `Click` zdarzenie. Wynikiem

końcowym dla większości odbiorników jest to, że `MouseLeftButtonDown` zdarzenia i `MouseRightButtonDown` są ukryte, a zdarzenie wysokiego poziomu `Click` jest widoczne.

Praca nad pomijaniem zdarzeń wejściowych

Czasami pomijanie zdarzeń w ramach poszczególnych kontrolek może zakłócać logikę obsługi zdarzeń w aplikacji. Jeśli na przykład aplikacja użyła składni atrybutu XAML w celu dołączenia procedury obsługi zdarzenia MouseLeftButtonDown w elemecie głównym XAML, ta procedura obsługi nie zostanie wywołana, ponieważ Button kontrolka oznacza `MouseLeftButtonDown` zdarzenie jako obsługiwane. Jeśli chcesz, aby elementy w katalogu głównym aplikacji mogły być wywoływane dla obsłużonego zdarzenia kierowanego, możesz wykonać następujące czynności:

- Dołączanie procedur obsługi przez wywołanie UIElement.AddHandler(RoutedEvent, Delegate, Boolean) metody z parametrem ustawionym `handledEventsToo` na `true`. Takie podejście wymaga dołączenia procedury obsługi zdarzeń w kodzie po uzyskaniu odwołania do obiektu dla elementu, do którego zostanie dołączony.

- Jeśli zdarzenie oznaczone jako obsługiwane jest zdarzenie wejściowe bubbling, dołącz programy obsługi dla sparowanego zdarzenia podglądu, jeśli jest dostępne. Jeśli na przykład kontrolka pomija `MouseLeftButtonDown` zdarzenie, można zamiast tego dołączyć procedurę obsługi dla PreviewMouseLeftButtonDown zdarzenia. To podejście działa tylko w przypadku par zdarzeń wejściowych w wersji zapoznawczej i bubbling, które współużytkuje dane zdarzenia. Należy zachować ostrożność, aby nie oznaczyć `PreviewMouseLeftButtonDown` go jako obsłużonego, ponieważ spowoduje to całkowite pominięcie Click zdarzenia.

Aby zapoznać się z przykładem obejścia pomijania zdarzeń wejściowych, zobacz Praca nad pomijaniem zdarzeń przez kontrolki.

Zobacz też

- EventManager
- Podgląd zdarzeń
- Tworzenie niestandardowego zdarzenia trasowanego
- Omówienie zdarzeń trasowanych

Zdarzenia w wersji zapoznawczej (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 7 min

Zdarzenia w wersji zapoznawczej, nazywane również zdarzeniami tunelowania, są kierowane zdarzenia przechodzące w dół przez drzewo elementów z elementu głównego aplikacji do elementu, który podniósł zdarzenie. Element, który zgłasza zdarzenie, jest zgłaszany jako Source dane zdarzenia. Nie wszystkie scenariusze zdarzeń obsługują lub wymagają zdarzeń w wersji zapoznawczej. W tym artykule opisano, gdzie istnieją zdarzenia w wersji zapoznawczej oraz jak aplikacje lub składniki mogą z nimi współdziałać. Aby uzyskać informacje na temat tworzenia zdarzenia w wersji zapoznawczej, zobacz Jak utworzyć niestandardowe zdarzenie kierowane.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat zdarzeń kierowanych i zapoznasz się z omówieniem zdarzeń trasowanych. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje Windows Presentation Foundation (WPF).

Podgląd zdarzeń oznaczonych jako obsługiwane

Należy zachować ostrożność podczas oznaczania zdarzeń w wersji zapoznawczej, które są obsługiwane w danych zdarzeń. Oznaczanie zdarzenia podglądu jako obsługiwanego w elemecie innym niż element, który go podniósł, może uniemożliwić element, który podniósł go do obsługi zdarzenia. Czasami oznaczanie zdarzeń w wersji zapoznawczej zgodnie z procedurą obsługi jest zamierzone. Na przykład kontrolka złożona może pomijać zdarzenia zgłaszane przez poszczególne składniki i zastępować je zdarzeniami zgłaszanymi przez pełną kontrolkę. Zdarzenia niestandardowe dla kontrolki mogą dostarczać dostosowane dane zdarzeń i wyzwalacz na podstawie relacji stanu składnika.

W przypadku zdarzeń wejściowych dane zdarzenia są współużytkowane zarówno przez odpowiedniki podglądu, jak i bez podglądu (bubbling) poszczególnych zdarzeń. Jeśli używasz procedury obsługi klasy zdarzeń w wersji zapoznawczej do oznaczania zdarzenia wejściowego jako obsługiwanego, programy obsługi klas dla zdarzenia wejściowego bubbling zwykle nie będą wywoływane. Jeśli używasz programu obsługi zdarzeń w wersji zapoznawczej do oznaczania zdarzenia jako obsługiwanego, programy obsługi wystąpień dla zdarzenia wejściowego bubbling zwykle nie będą wywoływane. Mimo że można skonfigurować programy obsługi klas i wystąpień, które mają być wywoływane, nawet jeśli zdarzenie jest oznaczone jako obsługiwane, ta konfiguracja obsługi nie jest powszechna. Aby uzyskać więcej informacji na temat obsługi klas i sposobu jej powiązania z zdarzeniami w wersji zapoznawczej, zobacz Oznaczanie zdarzeń kierowanych jako obsługiwanych i obsługi klas.

> ⓘ **Uwaga**
>
> Nie wszystkie zdarzenia w wersji zapoznawczej to zdarzenia **tunelowania** . Na przykład **PreviewMouseLeftButtonDown** zdarzenie wejściowe następuje po trasie w dół przez drzewo elementów, ale jest zdarzeniem kierowanym **bezpośrednio** , które zostało podniesione i odzyskane przez każdą z nich **UIElement** w trasie.

Obejście pomijania zdarzeń przez kontrolki

Niektóre kontrolki złożone pomijają zdarzenia wejściowe na poziomie składnika, aby zastąpić je dostosowanym zdarzeniem wysokiego poziomu. Na przykład WPF ButtonBase oznacza MouseLeftButtonDown zdarzenie wejściowe bubbling jako obsługiwane w metodzie OnMouseLeftButtonDown i zgłasza Click zdarzenie. `MouseLeftButtonDown` Zdarzenie i jego dane zdarzenia nadal są kontynuowane wzdłuż trasy drzewa elementów, ale ponieważ zdarzenie jest oznaczone jako Handled dane zdarzenia, wywoływane są tylko programy obsługi skonfigurowane do reagowania na obsługiwane zdarzenia.

Jeśli chcesz, aby inne elementy w katalogu głównym aplikacji obsługiwały zdarzenie kierowane oznaczone jako obsługiwane, możesz wykonać następujące czynności:

- Dołącz programy obsługi, wywołując metodę UIElement.AddHandler(RoutedEvent, Delegate, Boolean) i ustawiając parametr `handledEventsToo` na `true`. Takie podejście wymaga dołączenia procedury obsługi zdarzeń w kodzie za pomocą kodu po uzyskaniu odwołania do obiektu do elementu, do którego zostanie dołączony.

- Jeśli zdarzenie oznaczone jako obsługiwane jest zdarzenie bubbling, dołącz programy obsługi dla równoważnego zdarzenia w wersji zapoznawczej, jeśli jest dostępne. Na przykład jeśli kontrolka pomija MouseLeftButtonDown zdarzenie, możesz zamiast tego dołączyć procedurę obsługi dla PreviewMouseLeftButtonDown zdarzenia. Takie podejście działa tylko w przypadku zdarzeń wejściowych elementu podstawowego, które implementują strategie routingu tunelowania i rozbijania oraz udostępniają dane zdarzeń.

Poniższy przykład implementuje podstawową kontrolkę niestandardową TextBoxo nazwie `componentWrapper` , która zawiera element . Kontrolka jest dodawana do StackPanel nazwy `outerStackPanel`.

XAML

```
<StackPanel Name="outerStackPanel"
    VerticalAlignment="Center"
    custom:ComponentWrapper.CustomKey="Handler_PrintEventInfo"
    TextBox.KeyDown="Handler_PrintEventInfo"
    TextBox.PreviewKeyDown="Handler_PrintEventInfo" >
    <custom:ComponentWrapper
        x:Name="componentWrapper"
        TextBox.KeyDown="ComponentWrapper_KeyDown"
        custom:ComponentWrapper.CustomKey="Handler_PrintEventInfo"
        HorizontalAlignment="Center">
        <TextBox Name="componentTextBox" Width="200"
KeyDown="Handler_PrintEventInfo" />
    </custom:ComponentWrapper>
</StackPanel>
```

Kontrolka `componentWrapper` nasłuchuje KeyDown zdarzenia bubbling podniesione przez jego `TextBox` składnik za każdym razem, gdy wystąpi naciśnięcie klawisza. W tym wystąpieniu kontrolka `componentWrapper` :

1. `KeyDown` Oznacza rozsyłane zdarzenie kierowane zgodnie z obsługą, aby go pominąć. W związku z tym wyzwalana jest tylko `outerStackPanel` procedura obsługi skonfigurowana w kodzie w celu reagowania na *obsługiwane* `KeyDown` zdarzenia. Procedura `outerStackPanel` obsługi dołączona w języku XAML dla `KeyDown` zdarzeń nie jest wywoływana.

2. Zgłasza niestandardowe zdarzenie kierowane trasowane o nazwie `CustomKey`, które wyzwala procedurę `outerStackPanel` obsługi dla `CustomKey` zdarzenia.

C#

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

        // Attach a handler on outerStackPanel that will be invoked by
handled KeyDown events.
        outerStackPanel.AddHandler(KeyDownEvent, new
RoutedEventHandler(Handler_PrintEventInfo),
            handledEventsToo: true);
    }

    private void ComponentWrapper_KeyDown(object sender,
System.Windows.Input.KeyEventArgs e)
    {
        Handler_PrintEventInfo(sender, e);

        Debug.WriteLine("KeyDown event marked as handled on
componentWrapper.\r\n" +
            "CustomKey event raised on componentWrapper.");

        // Mark the event as handled.
        e.Handled = true;

        // Raise the custom click event.
        componentWrapper.RaiseCustomRoutedEvent();
    }

    private void Handler_PrintEventInfo(object sender,
System.Windows.Input.KeyEventArgs e)
    {
        string senderName = ((FrameworkElement)sender).Name;
        string sourceName = ((FrameworkElement)e.Source).Name;
        string eventName = e.RoutedEvent.Name;
        string handledEventsToo = e.Handled ? " Parameter handledEventsToo
set to true." : "";

        Debug.WriteLine($"Handler attached to {senderName} " +
            $"triggered by {eventName} event raised on {sourceName}.
{handledEventsToo}");
    }

    private void Handler_PrintEventInfo(object sender, RoutedEventArgs e)
    {
        string senderName = ((FrameworkElement)sender).Name;
        string sourceName = ((FrameworkElement)e.Source).Name;
        string eventName = e.RoutedEvent.Name;
        string handledEventsToo = e.Handled ? " Parameter handledEventsToo
set to true." : "";

        Debug.WriteLine($"Handler attached to {senderName} " +
            $"triggered by {eventName} event raised on {sourceName}.
{handledEventsToo}");
```

```
    }

    // Debug output:
    //
    // Handler attached to outerStackPanel triggered by PreviewKeyDown event
raised on componentTextBox.
    // Handler attached to componentTextBox triggered by KeyDown event
raised on componentTextBox.
    // Handler attached to componentWrapper triggered by KeyDown event
raised on componentTextBox.
    // KeyDown event marked as handled on componentWrapper.
    // CustomKey event raised on componentWrapper.
    // Handler attached to componentWrapper triggered by CustomKey event
raised on componentWrapper.
    // Handler attached to outerStackPanel triggered by CustomKey event
raised on componentWrapper.
    // Handler attached to outerStackPanel triggered by KeyDown event raised
on componentTextBox. Parameter handledEventsToo set to true.
}

public class ComponentWrapper : StackPanel
{
    // Register a custom routed event using the Bubble routing strategy.
    public static readonly RoutedEvent CustomKeyEvent =
        EventManager.RegisterRoutedEvent(
            name: "CustomKey",
            routingStrategy: RoutingStrategy.Bubble,
            handlerType: typeof(RoutedEventHandler),
            ownerType: typeof(ComponentWrapper));

    // Provide CLR accessors for assigning an event handler.
    public event RoutedEventHandler CustomKey
    {
        add { AddHandler(CustomKeyEvent, value); }
        remove { RemoveHandler(CustomKeyEvent, value); }
    }

    public void RaiseCustomRoutedEvent()
    {
        // Create a RoutedEventArgs instance.
        RoutedEventArgs routedEventArgs = new(routedEvent: CustomKeyEvent);

        // Raise the event, which will bubble up through the element tree.
        RaiseEvent(routedEventArgs);
    }
}
```

W przykładzie przedstawiono dwa obejścia dotyczące pobierania pominiętego `KeyDown` zdarzenia kierowanego w celu wywołania procedury obsługi zdarzeń dołączonej do elementu `outerStackPanel`:

- Dołącz procedurę obsługi zdarzeń PreviewKeyDown do elementu `outerStackPanel`. Ponieważ zdarzenie kierowane w wersji zapoznawczej poprzedza równoważne zdarzenie przekierowane, procedura obsługi w przykładzie jest uruchamiana przed `KeyDown` procedurą obsługi, `PreviewKeyDown` która pomija zarówno zdarzenia w wersji zapoznawczej, jak i bubbling za pośrednictwem udostępnionych danych zdarzeń.

- Dołącz procedurę `KeyDown` obsługi zdarzeń do `outerStackPanel` elementu przy użyciu UIElement.AddHandler(RoutedEvent, Delegate, Boolean) metody w kodzie za pomocą parametru ustawionego `handledEventsToo` na `true`.

> **Uwaga**
>
> Oznaczanie podglądu lub odpowiedników zdarzeń wejściowych, które są obsługiwane, to obie strategie pomijania zdarzeń zgłaszanych przez składniki kontrolki. Używane podejście zależy od wymagań aplikacji.

Zobacz też

- Oznaczanie zdarzeń trasowanych jako obsłużonych oraz obsługa klas
- Omówienie zdarzeń trasowanych
- Jak utworzyć niestandardowe zdarzenie kierowane

Zdarzenia zmiany właściwości (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 4 min

Windows Presentation Foundation (WPF) definiuje kilka zdarzeń, które są zgłaszane w odpowiedzi na zmianę wartości właściwości. Często właściwość jest właściwością zależności. Samo zdarzenie może być zdarzeniem kierowanym lub standardowym zdarzeniem środowiska uruchomieniowego języka wspólnego (CLR), w zależności od tego, czy zdarzenie powinno być kierowane przez drzewo elementów, czy występuje tylko w obiekcie, w którym zmieniono właściwość. Ten ostatni scenariusz ma zastosowanie, gdy zmiana właściwości ma zastosowanie tylko do obiektu, w którym zmieniono wartość właściwości.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat właściwości zależności i przeczytano omówienie zdarzeń trasowanych.

Identyfikowanie zdarzenia zmiany właściwości

Nie wszystkie zdarzenia, które zgłaszają zmianę właściwości, są jawnie identyfikowane jako zdarzenie zmienione przez ich sygnaturę lub wzorzec nazewnictwa. Dokumentacja zestawu SDK zawiera odwołania krzyżowe właściwości z zdarzeniami i wskazuje, czy zdarzenie jest bezpośrednio powiązane ze zmianą wartości właściwości.

Niektóre zdarzenia używają typu danych zdarzenia i delegata, które są specyficzne dla zdarzeń zmienionych właściwości. Na przykład zdarzenia RoutedPropertyChanged i DependencyPropertyChanged mają określone podpisy. Te typy zdarzeń zostały omówione w poniższych sekcjach.

Zdarzenia RoutedPropertyChanged

Zdarzenia RoutedPropertyChanged mają RoutedPropertyChangedEventArgs<T> dane zdarzenia i delegata RoutedPropertyChangedEventHandler<T> . Zarówno dane zdarzenia, jak i delegat mają ogólny parametr `T` typu . Podczas definiowania procedury obsługi należy określić rzeczywisty typ zmienionej właściwości. Dane zdarzenia zawierają OldValue właściwości i NewValue , których typ środowiska uruchomieniowego jest taki sam jak zmieniona właściwość.

Część "Routed" nazwy wskazuje, że zmienione zdarzenie właściwości jest rejestrowane jako zdarzenie kierowane. Zaletą zmienionego zdarzenia *kierowanego* właściwości jest to, że elementy nadrzędne są powiadamiane za każdym razem, gdy zmieniają się właściwości elementu podrzędnego. Oznacza to, że element najwyższego poziomu kontrolki odbiera zdarzenia zmiany właściwości, gdy wartość którejkolwiek z jego złożonych części ulega zmianie. Załóżmy na przykład, że tworzysz kontrolkę zawierającą kontrolkę RangeBase , taką jak Slider. Value Jeśli właściwość zmieni się na części suwaka, możesz obsłużyć zmianę w kontrolce nadrzędnej, a nie w części.

Unikaj używania procedury obsługi zdarzeń zmienionej właściwości, aby zweryfikować wartość właściwości, ponieważ nie jest to intencja projektu dla większości zdarzeń zmienionych właściwości. Ogólnie rzecz biorąc, zdarzenie zmiany właściwości jest udostępniane, aby można było reagować na zmianę wartości w innych obszarach logiki kodu. Zmiana wartości właściwości ponownie z wewnątrz właściwości zmienionej procedury obsługi zdarzeń nie jest zalecana i może spowodować niezamierzoną rekursję w zależności od implementacji programu obsługi.

Jeśli właściwość jest niestandardową właściwością zależności lub jeśli pracujesz z klasą pochodną, w której zdefiniowano kod tworzenia wystąpienia, system właściwości WPF ma lepszy sposób śledzenia zmian właściwości. W ten sposób należy użyć wbudowanych CoerceValueCallback wywołań zwrotnych systemu właściwości i PropertyChangedCallback właściwości. Aby uzyskać więcej informacji na temat używania systemu właściwości WPF do walidacji i przymusu, zobacz wywołania zwrotne właściwości zależności i walidacji i właściwości zależności niestandardowych.

Zdarzenia DependencyPropertyChanged

Zdarzenia DependencyPropertyChanged mają DependencyPropertyChangedEventArgs dane zdarzenia i delegata DependencyPropertyChangedEventHandler . Te zdarzenia to standardowe zdarzenia CLR, a nie zdarzenia kierowane. `DependencyPropertyChangedEventArgs` jest nietypowym typem raportowania danych zdarzeń, ponieważ nie pochodzi od EventArgs i jest strukturą, a nie klasą.

`DependencyPropertyChanged` Przykładem zdarzenia jest IsMouseCapturedChanged. `DependencyPropertyChanged` zdarzenia są nieco bardziej typowe niż

`RoutedPropertyChanged` zdarzenia.

Podobnie jak w przypadku danych zdarzenia RoutedPropertyChanged, dane zdarzenia DependencyPropertyChanged zawierają OldValue właściwości i NewValue . Ze względów wymienionych wcześniej należy unikać używania procedury obsługi zdarzeń zmienionej właściwości, aby ponownie zmienić wartość właściwości.

Wyzwalacze właściwości

Ściśle powiązaną koncepcją ze zdarzeniem zmiany właściwości jest wyzwalacz właściwości. Wyzwalacz właściwości jest tworzony w stylu lub szablonie. Wyzwalacz właściwości umożliwia utworzenie zachowania warunkowego na podstawie wartości właściwości, do której jest przypisany wyzwalacz.

Właściwość, na której działa wyzwalacz właściwości, musi być właściwością zależności. Może to być i często jest właściwością zależności tylko do odczytu. Jeśli właściwość zależności uwidoczniona przez kontrolkę ma nazwę rozpoczynającą się od "Is", jest to dobry wskaźnik, że właściwość została co najmniej częściowo zaprojektowana jako wyzwalacz właściwości. Właściwości z tym nazewnictwem są często właściwościami zależności tylko Boolean do odczytu, gdzie podstawowym scenariuszem dla właściwości jest raportowanie stanu kontroli. Jeśli stan kontroli wpływa na interfejs użytkownika w czasie rzeczywistym, właściwość zależności jest kandydatem wyzwalacza właściwości.

Niektóre właściwości zależności mają zdarzenie zmienionej właściwości dedykowanej. Na przykład IsMouseCaptured ma IsMouseCapturedChanged zmienione zdarzenie właściwości. Właściwość `IsMouseCaptured` jest tylko do odczytu, a jej wartość jest modyfikowana przez system wejściowy. System wejściowy zgłasza `IsMouseCapturedChanged` zdarzenie przy każdej zmianie w czasie rzeczywistym.

Ograniczenia wyzwalacza właściwości

W porównaniu do zdarzenia zmienionej właściwości true wyzwalacze właściwości mają pewne ograniczenia.

Wyzwalacze właściwości działają za pomocą dokładnej logiki dopasowania, w której określasz nazwę właściwości i określoną wartość, która aktywuje wyzwalacz. Może to być na przykład `<Setter Property="IsMouseCaptured" Value="true"> ... </Setter>`. Składnia wyzwalacza właściwości ogranicza większość użycia właściwości do Boolean właściwości lub właściwości, które przyjmują dedykowaną wartość wyliczenia. Zakres możliwych wartości musi być zarządzany, aby można było zdefiniować wyzwalacz dla każdego przypadku. Czasami wyzwalacze właściwości istnieją tylko dla wartości specjalnych,

takich jak gdy liczba elementów osiągnie zero. Nie można ustawić pojedynczego wyzwalacza w celu aktywowania, gdy wartość właściwości zmienia się z dala od określonej wartości, takiej jak zero. Zamiast używać wielu wyzwalaczy dla wszystkich przypadków niezerowych, rozważ zaimplementowanie procedury obsługi zdarzeń kodu lub domyślne zachowanie, które przełącza się z powrotem ze stanu wyzwalacza, gdy wartość jest niezerowa.

Składnia wyzwalacza właściwości jest analogiczna do instrukcji "if" w programowaniu. Jeśli warunek wyzwalacza ma wartość true, "treść" wyzwalacza właściwości to "run". "Treść" wyzwalacza właściwości nie jest kodem, jego znacznikami. Ta adiustacja jest ograniczona do używania co najmniej jednego Setter elementu do ustawiania innych właściwości obiektu, w którym zastosowano styl lub szablon.

Jeśli warunek "if" wyzwalacza właściwości ma szeroką gamę możliwych wartości, zaleca się ustawienie tej samej wartości właściwości na wartość domyślną przy użyciu `Setter` wyzwalacza zewnętrznego. W ten sposób element ustawiający w wyzwalaczu będzie miał pierwszeństwo, gdy warunek wyzwalacza to `true`, w przeciwnym razie `Setter` poza wyzwalaczem będzie pierwszeństwo.

Wyzwalacze właściwości są przydatne w scenariuszach, w których co najmniej jedna właściwość wyglądu powinna ulec zmianie w zależności od stanu innej właściwości w tym samym elemecie.

Aby dowiedzieć się więcej na temat wyzwalaczy właściwości, zobacz Styling and Templating (Tworzenie stylów i tworzenie szablonów).

Zobacz też

- Omówienie zdarzeń trasowanych
- Omówienie właściwości zależności

Obsługa zdarzeń Visual Basic i WPF (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 4 min

Jeśli kodujesz na platformie .NET w języku Visual Basic, możesz użyć słowa kluczowego Handles specyficznego dla języka, aby dołączyć procedurę obsługi zdarzeń do obiektu. Obiekt może być wystąpieniem w kodzie lub elementem w języku Extensible Application Markup Language (XAML). `Handles` można użyć do przypisywania procedur obsługi zdarzeń dla zdarzeń środowiska uruchomieniowego języka wspólnego (CLR) lub zdarzeń kierowanych Windows Presentation Foundation (WPF). Jednak ma pewne ograniczenia użycia, `Handles` gdy są używane do dołączania procedur obsługi zdarzeń dla zdarzeń kierowanych.

> **ⓘ Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat zdarzeń kierowanych i zapoznasz się z omówieniem zdarzeń trasowanych. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje Windows Presentation Foundation (WPF).

Składnia

Składnia deklaracji używającej słowa kluczowego `Sub` Handles to: `Sub <procedure name> Handles <object name>.<event name>`. Ta składnia wyznacza procedurę jako procedurę obsługi zdarzeń, która będzie uruchamiana, gdy zdarzenie określone przez `<event name>` jest wywoływane na obiekcie określonym przez `<object name>`. Zdarzenie musi być elementem członkowskim klasy obiektu lub klasy bazowej. W poniższym przykładzie pokazano, jak dołączyć procedurę obsługi zdarzeń do elementu XAML przy użyciu polecenia `Handles`.

VB

```
' Click event handler attached to XamlButton using Handles.
Private Sub XamlButton_Click(sender As Object, e As RoutedEventArgs) Handles
XamlButton.Click

    ' Handler logic.
    Debug.WriteLine($"Click event handler attached to XamlButton using
Handles.")

End Sub
```

Aby użyć `Handles` obiektu zdefiniowanego w kodzie, zazwyczaj deklarujesz obiekt przy użyciu słowa kluczowego WithEvents . Aby uzyskać więcej informacji na `WithEvents` temat użycia, zobacz te przykłady. WPF automatycznie deklaruje wszystkie elementy XAML przy użyciu polecenia `Friend WithEvents`. W poniższym przykładzie pokazano, jak zadeklarować obiekt zdefiniowany w kodzie za pomocą polecenia `WithEvents`.

VB

```
' Declare a new button using WithEvents.
Dim WithEvents CodeButton As New Button With {
    .Content = "New button",
    .Background = Brushes.Yellow
}

' Click event handler attached to CodeButton using Handles.
Private Sub CodeButton_Click(sender As Object, e As RoutedEventArgs) Handles
CodeButton.Click

    ' Handler logic.
    Debug.WriteLine($"Click event handler attached to CodeButton using
Handles.")

End Sub
```

Aby użyć tej samej procedury obsługi dla wielu zdarzeń, rozdzielaj `<object name>.<event name>` przecinkami zdarzenia. Na przykład `Sub Button_Click(sender As Object, e As RoutedEventArgs) Handles Button1.Click, Button2.Click`. Kolejność zdarzeń rozdzielanych przecinkami jest niematerialna.

Możesz przypisać różne programy obsługi dla tego samego zdarzenia z wieloma `Handles` instrukcjami. Kolejność instrukcji `Handles` nie określa kolejności wywoływanej procedury obsługi po wystąpieniu zdarzenia.

Porada

Aby usunąć procedurę obsługi, która została dodana za pomocą `Handles` metody , wywołaj metodę **RemoveHandler**. Na przykład `RemoveHandler Button1.Click, AddressOf Button1_Click`.

Używanie funkcji "Handles" w aplikacji WPF

W przypadku obiektu zdefiniowanego w języku XAML składnia `<object name>.<event name>` zdarzeń Handles wymaga elementu XAML, który reprezentuje obiekt, aby mieć Name właściwość lub x:Name. Jednak właściwość name nie jest wymagana dla elementu głównego strony XAML, dla którego można użyć nazwy `Me`. W poniższym przykładzie pokazano, jak dołączyć procedurę obsługi zdarzeń do katalogu głównego strony XAML przy użyciu polecenia `Handles`.

VB

```
' Loaded event handler attached to the XAML page root using Handles.
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded

    ' Handler logic.
    Debug.WriteLine($"Loaded event handler attached to Window using
Handles.")

End Sub
```

Po skompilowaniu strony XAML każdy element XAML z parametrem `Name` or `x:Name` jest zadeklarowany jako `Friend WithEvents`. W związku z tym można użyć dowolnego elementu XAML z elementem `Handles`.

Porada

Funkcja IntelliSense programu Visual Studio pokazuje obiekty, których można używać z `Handles` programem .

Niezależnie od tego, czy program obsługi zdarzeń jest dołączany przy użyciu `Handles` składni atrybutu XAML, instrukcji AddHandler lub AddHandler metody, zachowanie systemu zdarzeń jest takie samo.

Uwaga

Nie używaj zarówno atrybutów XAML, jak i `Handles` do dołączania tej samej procedury obsługi zdarzeń do tego samego zdarzenia. W przeciwnym razie program obsługi zdarzeń zostanie wywołany dwa razy dla każdego zdarzenia.

Ograniczenia

Słowo kluczowe Handles ma następujące ograniczenia użycia:

- Program obsługi zdarzeń można dołączyć do obiektu tylko `Handles` wtedy, gdy zdarzenie jest elementem członkowskim klasy obiektu lub klasy bazowej. Można na przykład użyć `Handles` polecenia , aby dołączyć procedurę Click obsługi zdarzeń do przycisku, którego klasa ButtonBase bazowa zgłasza `Click` zdarzenie kierowane. Jednak jedną z funkcji zdarzeń kierowanych jest to, że przechodzą przez drzewo elementów, co umożliwia nasłuchiwanie i obsługę `Click` zdarzenia na wyższym poziomie niż element, który go podniósł. Zdarzenie kierowane, dla którego element nadrzędny nasłuchuje i obsługuje, jest nazywane *dołączonym zdarzeniem*. `Handles` Nie można używać dla dołączonych zdarzeń, ponieważ jego składnia nie obsługuje określania innego odbiornika w drzewie elementów XAML niż element, który wzbudził zdarzenie. Aby przypisać programy obsługi zdarzeń dla dołączonych zdarzeń, należy użyć składni atrybutu AddHandler XAML lub metody. Aby uzyskać więcej informacji na temat dołączonych zdarzeń, zobacz Omówienie dołączonych zdarzeń i Dołączone zdarzenia w WPF.

- `Handles` składnia nie obsługuje wywołania programu obsługi zdarzeń dla Handled zdarzeń. Aby umożliwić wywoływanie programu obsługi zdarzeń dla `Handled` zdarzeń, dołącz procedurę obsługi zdarzeń przy użyciu AddHandler(RoutedEvent, Delegate, Boolean) metody i ustaw jej `handledEventsToo` parametr na `true`.

Zobacz też

- AddHandler
- Oznaczanie zdarzeń trasowanych jako obsłużonych oraz obsługa klas
- Omówienie zdarzeń trasowanych
- Omówienie zdarzeń dołączonych
- XAML w WPF

Słabe wzorce zdarzeń (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 7 min

W aplikacjach istnieje możliwość, że programy obsługi dołączone do źródeł zdarzeń nie zostaną zniszczone we współpracy z obiektem odbiornika, który dołączył procedurę obsługi do źródła. Taka sytuacja może prowadzić do przecieków pamięci. Windows Presentation Foundation (WPF) wprowadza wzorzec projektowy, którego można użyć do rozwiązania tego problemu. Wzorzec projektu zapewnia dedykowaną klasę menedżera dla określonych zdarzeń i implementuje interfejs na odbiornikach dla tego zdarzenia. Ten wzorzec projektowy jest nazywany *słabym wzorcem zdarzeń*.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat kierowanych zdarzeń i przeczytano omówienie zdarzeń trasowanych. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje Windows Presentation Foundation (WPF).

Dlaczego warto zaimplementować słaby wzorzec zdarzeń?

Nasłuchiwanie zdarzeń może prowadzić do przecieków pamięci. Zwykłą techniką nasłuchiwania zdarzenia jest użycie składni specyficznej dla języka w celu dołączenia programu obsługi do zdarzenia w źródle. Na przykład instrukcja `source.SomeEvent += new SomeEventHandler(MyEventHandler)` języka C# lub instrukcja `AddHandler source.SomeEvent, AddressOf MyEventHandler`VB . Jednak ta technika tworzy silne odwołanie ze źródła zdarzeń do odbiornika zdarzeń. Jeśli program obsługi zdarzeń nie zostanie jawnie wyrejestrowany, okres istnienia obiektu odbiornika będzie mieć wpływ na okres istnienia obiektu źródła. W pewnych okolicznościach można chcieć, aby okres istnienia obiektu odbiornika był kontrolowany przez inne czynniki, takie jak to, czy obecnie należy do drzewa wizualnego aplikacji. Za każdym razem, gdy okres istnienia obiektu źródła wykracza poza okres istnienia użytecznego obiektu odbiornika, odbiornik

jest utrzymywany przy życiu dłużej niż jest to konieczne. W takim przypadku nieprzydzielona ilość pamięci jest nieprzydzielona do przecieku pamięci.

Słaby wzorzec zdarzeń został zaprojektowany w celu rozwiązania problemu z przeciekiem pamięci. Słaby wzorzec zdarzenia może być używany, gdy odbiornik musi zarejestrować się na potrzeby zdarzenia, ale odbiornik nie wie jawnie, kiedy wyrejestrować. Słaby wzorzec zdarzenia może być również używany, gdy okres istnienia obiektu źródła przekracza użyteczny okres istnienia obiektu odbiornika. W takim przypadku *przydatne* jest określone przez Ciebie. Wzorzec słabego zdarzenia umożliwia odbiornikowi rejestrowanie i odbieranie zdarzenia bez wpływu na charakterystykę okresu istnienia obiektu odbiornika w jakikolwiek sposób. W efekcie domniemane odwołanie ze źródła nie określa, czy odbiornik kwalifikuje się do odzyskiwania pamięci. Odwołanie jest słabym odwołaniem, w związku z czym nazewnictwo słabego wzorca zdarzeń i powiązanych interfejsów API. Odbiornik może być odśmiecany lub zniszczony w inny sposób, a źródło może kontynuować bez zachowywania odwołań niezwiązanych z procedurą obsługi do teraz zniszczonego obiektu.

Kto powinien implementować słaby wzorzec zdarzeń?

Słaby wzorzec zdarzenia jest głównie istotny dla autorów kontrolek. Jako autor kontrolki jesteś w dużej mierze odpowiedzialny za zachowanie i zawieranie kontrolki oraz wpływ na aplikacje, w których jest wstawiony. Obejmuje to zachowanie okresu istnienia obiektu kontrolki, w szczególności obsługę opisanego problemu z wyciekiem pamięci.

Niektóre scenariusze z natury nadają się do stosowania słabego wzorca zdarzeń. Jednym z takich scenariuszy jest powiązanie danych. W powiązaniu danych często obiekt źródłowy jest niezależny od obiektu odbiornika, który jest obiektem docelowym powiązania. Wiele aspektów powiązania danych WPF ma już zastosowany słaby wzorzec zdarzeń w sposób implementowania zdarzeń.

How to Implement the weak event pattern (Jak zaimplementować słaby wzorzec zdarzeń)

Istnieją cztery sposoby implementowania słabego wzorca zdarzeń, a każde podejście używa innego menedżera zdarzeń. Wybierz menedżera zdarzeń, który najlepiej odpowiada Twojemu scenariuszowi.

- Istniejący słaby menedżer zdarzeń:

Użyj istniejącej słabej klasy menedżera zdarzeń, gdy zdarzenie, które chcesz zasubskrybować, ma odpowiadający element WeakEventManager. Aby uzyskać listę słabych menedżerów zdarzeń dołączonych do platformy WPF, zobacz hierarchię `WeakEventManager` dziedziczenia w klasie. Ponieważ dołączone słabe menedżery zdarzeń są ograniczone, prawdopodobnie trzeba będzie wybrać jedno z innych podejść.

- Ogólny słaby menedżer zdarzeń:

  Użyj funkcji ogólnej WeakEventManager<TEventSource,TEventArgs> , gdy istniejący WeakEventManager nie jest dostępny i szukasz najprostszego sposobu implementowania słabych zdarzeń. Jednak ogólny `WeakEventManager<TEventSource,TEventArgs>` jest mniej wydajny niż istniejący lub niestandardowy słaby menedżer zdarzeń, ponieważ używa odbicia w celu odnalezienia zdarzenia z jego nazwy. Ponadto kod potrzebny do zarejestrowania zdarzenia przy użyciu funkcji ogólnej `WeakEventManager<TEventSource,TEventArgs>` jest bardziej szczegółowy niż użycie istniejącego lub niestandardowego `WeakEventManager`.

- Niestandardowy słaby menedżer zdarzeń:

  Tworzenie niestandardowego WeakEventManager , gdy istniejący `WeakEventManager` element nie jest dostępny, a wydajność ma kluczowe znaczenie. Chociaż bardziej wydajny niż ogólny `WeakEventManager`, niestandardowy `WeakEventManager` wymaga pisania większej liczby kodów z góry.

- Słaby menedżer zdarzeń innych firm :

  Użyj słabego menedżera zdarzeń innej firmy, jeśli potrzebujesz funkcji, które nie są udostępniane przez inne podejścia. NuGet ma kilka słabych menedżerów zdarzeń . Wiele struktur WPF obsługuje również wzorzec. Na przykład zapoznaj się z dokumentacją prism dotyczącą luźno powiązanej subskrypcji zdarzeń .

W poniższych sekcjach opisano sposób implementowania słabego wzorca zdarzeń za pomocą różnych typów menedżera zdarzeń. W przypadku ogólnych i niestandardowych słabych przykładów menedżera zdarzeń zdarzenie do zasubskrybowania ma następujące cechy.

- Nazwa zdarzenia to `SomeEvent`.
- Zdarzenie jest wywoływane przez klasę `SomeEventSource` .
- Procedura obsługi zdarzeń ma typ `EventHandler<SomeEventArgs>`.
- Zdarzenie przekazuje parametr typu `SomeEventArgs` do procedur obsługi zdarzeń.

Używanie istniejącej słabej klasy menedżera zdarzeń

1. Znajdź istniejącego słabego menedżera zdarzeń. Aby uzyskać listę słabych menedżerów zdarzeń zawartych w WPF, zobacz hierarchię WeakEventManager dziedziczenia klasy.

2. Użyj nowego słabego menedżera zdarzeń zamiast normalnego podpinania zdarzeń.

   Jeśli na przykład kod używa następującego wzorca do subskrybowania zdarzenia:

   C#
   ```
   source.LostFocus += new RoutedEventHandler(Source_LostFocus);
   ```

   Zmień go na następujący wzorzec:

   C#
   ```
   LostFocusEventManager.AddHandler(source, Source_LostFocus);
   ```

   Podobnie, jeśli kod używa następującego wzorca do anulowania subskrypcji zdarzenia:

   C#
   ```
   source.LostFocus -= new RoutedEventHandler(Source_LostFocus);
   ```

   Zmień go na następujący wzorzec:

   C#
   ```
   LostFocusEventManager.RemoveHandler(source, Source_LostFocus);
   ```

Używanie ogólnej słabej klasy menedżera zdarzeń

Użyj klasy ogólnej WeakEventManager<TEventSource,TEventArgs> zamiast normalnego podpinania zdarzeń.

Gdy używasz `WeakEventManager<TEventSource,TEventArgs>` polecenia do rejestrowania odbiorników zdarzeń, należy podać źródło zdarzenia i EventArgs wpisać jako parametry typu do klasy. Wywołaj metodę AddHandler , jak pokazano w poniższym kodzie:

C#

```
WeakEventManager<SomeEventSource, SomeEventArgs>.AddHandler(source,
"SomeEvent", Source_SomeEvent);
```

Tworzenie niestandardowej słabej klasy menedżera zdarzeń

1. Skopiuj następujący szablon klasy do projektu. Następująca klasa dziedziczy z WeakEventManager klasy :

C#

```
class SomeEventWeakEventManager : WeakEventManager
{
    private SomeEventWeakEventManager()
    {
    }

    /// <summary>
    /// Add a handler for the given source's event.
    /// </summary>
    public static void AddHandler(SomeEventSource source,
                                  EventHandler<SomeEventArgs> handler)
    {
        if (source == null)
            throw new ArgumentNullException(nameof(source));
        if (handler == null)
            throw new ArgumentNullException(nameof(handler));

        CurrentManager.ProtectedAddHandler(source, handler);
    }

    /// <summary>
    /// Remove a handler for the given source's event.
    /// </summary>
    public static void RemoveHandler(SomeEventSource source,
                                     EventHandler<SomeEventArgs>
handler)
    {
        if (source == null)
            throw new ArgumentNullException(nameof(source));
        if (handler == null)
            throw new ArgumentNullException(nameof(handler));

        CurrentManager.ProtectedRemoveHandler(source, handler);
    }

    /// <summary>
    /// Get the event manager for the current thread.
    /// </summary>
```

```
    private static SomeEventWeakEventManager CurrentManager
    {
        get
        {
            Type managerType = typeof(SomeEventWeakEventManager);
            SomeEventWeakEventManager manager =

(SomeEventWeakEventManager)GetCurrentManager(managerType);

            // at first use, create and register a new manager
            if (manager == null)
            {
                manager = new SomeEventWeakEventManager();
                SetCurrentManager(managerType, manager);
            }

            return manager;
        }
    }

    /// <summary>
    /// Return a new list to hold listeners to the event.
    /// </summary>
    protected override ListenerList NewListenerList()
    {
        return new ListenerList<SomeEventArgs>();
    }

    /// <summary>
    /// Listen to the given source for the event.
    /// </summary>
    protected override void StartListening(object source)
    {
        SomeEventSource typedSource = (SomeEventSource)source;
        typedSource.SomeEvent += new EventHandler<SomeEventArgs>
(OnSomeEvent);
    }

    /// <summary>
    /// Stop listening to the given source for the event.
    /// </summary>
    protected override void StopListening(object source)
    {
        SomeEventSource typedSource = (SomeEventSource)source;
        typedSource.SomeEvent -= new EventHandler<SomeEventArgs>
(OnSomeEvent);
    }

    /// <summary>
    /// Event handler for the SomeEvent event.
    /// </summary>
    void OnSomeEvent(object sender, SomeEventArgs e)
    {
        DeliverEvent(sender, e);
```

```
    }
}
```

2. Zmień nazwę `SomeEventWeakEventManager`, `SomeEvent`, `SomeEventSource` i `SomeEventArgs` na zgodną z nazwą zdarzenia.

3. Ustaw modyfikatory dostępu dla słabej klasy menedżera zdarzeń, aby pasować do ułatwień dostępu zarządzanego zdarzenia.

4. Użyj nowego słabego menedżera zdarzeń zamiast normalnego podpinania zdarzeń.

   Jeśli na przykład kod używa następującego wzorca do subskrybowania zdarzenia:

   ```C#
   source.SomeEvent += new EventHandler<SomeEventArgs>(Source_SomeEvent);
   ```

   Zmień go na następujący wzorzec:

   ```C#
   SomeEventWeakEventManager.AddHandler(source, Source_SomeEvent);
   ```

   Podobnie, jeśli kod używa następującego wzorca do anulowania subskrypcji zdarzenia:

   ```C#
   source.SomeEvent -= new EventHandler<SomeEventArgs>(Source_SomeEvent);
   ```

   Zmień go na następujący wzorzec:

   ```C#
   SomeEventWeakEventManager.RemoveHandler(source, Source_SomeEvent);
   ```

Zobacz też

- WeakEventManager
- IWeakEventListener
- Omówienie zdarzeń trasowanych

- Omówienie powiązań danych

Jak dodać program obsługi zdarzeń przy użyciu kodu (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 4 min

Program obsługi zdarzeń można przypisać do elementu w Windows Presentation Foundation (WPF) przy użyciu znaczników lub kodu. Chociaż jest on niestandardowy do przypisywania programu obsługi zdarzeń w języku Extensible Application Markup Language (XAML), czasami może być konieczne przypisanie procedury obsługi zdarzeń w kodzie. Na przykład użyj kodu, gdy:

- Program obsługi zdarzeń jest przypisywany do elementu po stronie znaczników zawierającej łady elementu.
- Dodasz element i przypiszesz jego procedurę obsługi zdarzeń po stronie znaczników, która będzie zawierać ładowanie elementów.
- Drzewo elementów dla aplikacji definiuje się całkowicie w kodzie.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat kierowanych zdarzeń i przeczytano omówienie zdarzeń trasowanych. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje Windows Presentation Foundation (WPF).

Składnia przypisania programu obsługi zdarzeń

Język C# obsługuje przypisywanie programu obsługi zdarzeń przy użyciu:

- Operator `+=` , który jest również używany w modelu obsługi zdarzeń środowiska uruchomieniowego języka wspólnego (CLR).
- Metoda UIElement.AddHandler .

Język VB obsługuje przypisywanie programu obsługi zdarzeń przy użyciu:

- Instrukcja AddHandler z operatorem AddressOf , który jest również używany w modelu obsługi zdarzeń CLR.
- Słowo kluczowe Handles w definicji programu obsługi zdarzeń. Aby uzyskać więcej informacji, zobacz Obsługa zdarzeń w języku Visual Basic i WPF.
- Metoda UIElement.AddHandler wraz z operatorem `AddressOf` w celu odwołania się do programu obsługi zdarzeń.

Przykład

W poniższym przykładzie użyto języka XAML do zdefiniowania Button nazwy `ButtonCreatedByXaml` i przypisania `ButtonCreatedByXaml_Click` metody jako procedury Click obsługi zdarzeń. `Click` to wbudowane zdarzenie kierowane dla przycisków, które pochodzą z .ButtonBase

XAML

```
<StackPanel Name="StackPanel1">
    <Button
        Name="ButtonCreatedByXaml"
        Click="ButtonCreatedByXaml_Click"
        Content="Create a new button with an event handler"
        Background="LightGray">
    </Button>
</StackPanel>
```

W przykładzie użyto kodu do zaimplementowania `ButtonCreatedByXaml_Click` procedur obsługi i `ButtonCreatedByCode_Click` oraz przypisania `ButtonCreatedByCode_Click` procedury obsługi do `ButtonCreatedByCode` elementów i `StackPanel1` . Metody obsługi zdarzeń można zaimplementować tylko w kodzie.

C#

```
// The click event handler for the existing button 'ButtonCreatedByXaml'.
private void ButtonCreatedByXaml_Click(object sender, RoutedEventArgs e)
{
    // Create a new button.
    Button ButtonCreatedByCode = new();

    // Specify button properties.
    ButtonCreatedByCode.Name = "ButtonCreatedByCode";
    ButtonCreatedByCode.Content = "New button and event handler created in
code";
    ButtonCreatedByCode.Background = Brushes.Yellow;

    // Add the new button to the StackPanel.
    StackPanel1.Children.Add(ButtonCreatedByCode);
```

```
    // Assign an event handler to the new button using the '+=' operator.
    ButtonCreatedByCode.Click += new
RoutedEventHandler(ButtonCreatedByCode_Click);

    // Assign an event handler to the new button using the AddHandler
method.
    // AddHandler(ButtonBase.ClickEvent, new
RoutedEventHandler(ButtonCreatedByCode_Click);

    // Assign an event handler to the StackPanel using the AddHandler
method.
    StackPanel1.AddHandler(ButtonBase.ClickEvent, new
RoutedEventHandler(ButtonCreatedByCode_Click));
}

// The Click event handler for the new button 'ButtonCreatedByCode'.
private void ButtonCreatedByCode_Click(object sender, RoutedEventArgs e)
{
    string sourceName = ((FrameworkElement)e.Source).Name;
    string senderName = ((FrameworkElement)sender).Name;

    Debug.WriteLine($"Routed event handler attached to {senderName}, " +
        $"triggered by the Click routed event raised by {sourceName}.");
}
```

Po `ButtonCreatedByXaml` kliknięciu i uruchomieniu `ButtonCreatedByXaml_Click` programu obsługi zdarzeń programowo:

1. Dodaje nowy przycisk o nazwie `ButtonCreatedByCode` do już skonstruowanego drzewa elementów XAML.
2. Określa właściwości nowego przycisku, takie jak nazwa, zawartość i kolor tła.
3. Przypisuje program obsługi zdarzeń `ButtonCreatedByCode_Click` do elementu `ButtonCreatedByCode`.
4. Przypisuje tę samą `ButtonCreatedByCode_Click` procedurę obsługi zdarzeń do programu `StackPanel1`.

Po `ButtonCreatedByCode` kliknięciu:

1. Zdarzenie Click kierowane jest wywoływane w dniu `ButtonCreatedByCode`.
2. Wyzwalana `ButtonCreatedByCode_Click` jest procedura obsługi zdarzeń przypisana do `ButtonCreatedByCode` elementu .
3. Przekierowane `Click` zdarzenie przechodzi w górę drzewa elementów do .`StackPanel1`
4. Wyzwalana `ButtonCreatedByCode_Click` jest procedura obsługi zdarzeń przypisana do `StackPanel1` elementu .

5. Zdarzenie `Click` kierowane kontynuuje drzewo elementów potencjalnie wyzwalając inne `Click` programy obsługi zdarzeń przypisane do innych elementów przechodzących.

Procedura `ButtonCreatedByCode_Click` obsługi zdarzeń uzyskuje następujące informacje o zdarzeniu, które go wyzwoliło:

- Obiekt nadawcy , który jest elementem przypisanym do programu obsługi zdarzeń. Element `sender` będzie `ButtonCreatedByCode` pierwszym uruchomieniem programu obsługi, a `StackPanel1` po raz drugi.
- Obiekt RoutedEventArgs.Source , który jest elementem, który pierwotnie wywołał zdarzenie. W tym przykładzie parametr `Source` ma zawsze `ButtonCreatedByCode`wartość .

> **Uwaga**
>
> Kluczową różnicą między zdarzeniem kierowanym a zdarzeniem CLR jest to, że kierowane zdarzenie przechodzi przez drzewo elementów, szukając procedur obsługi, podczas gdy zdarzenie CLR nie przechodzi przez drzewo elementów, a programy obsługi mogą dołączać tylko do obiektu źródłowego, który wywołał zdarzenie. W rezultacie zdarzenie `sender` kierowane może być dowolnym elementem przechodzącym w drzewie elementów.

Aby uzyskać więcej informacji na temat tworzenia i obsługi zdarzeń kierowanych, zobacz Jak utworzyć niestandardowe zdarzenie kierowane i obsługiwać zdarzenie kierowane.

Zobacz też

- Omówienie zdarzeń trasowanych
- XAML w WPF

Jak utworzyć niestandardowe zdarzenie kierowane (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 5 min

Windows Presentation Foundation (WPF) deweloperzy aplikacji i autorzy składników mogą tworzyć niestandardowe zdarzenia kierowane w celu rozszerzenia funkcjonalności zdarzeń środowiska uruchomieniowego języka wspólnego (CLR). Aby uzyskać informacje na temat możliwości zdarzenia kierowanego, zobacz Dlaczego warto używać zdarzeń trasowanych. W tym artykule opisano podstawy tworzenia niestandardowego zdarzenia kierowanego.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat kierowanych zdarzeń i przeczytano omówienie zdarzeń trasowanych. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje Windows Presentation Foundation (WPF).

Kroki zdarzeń kierowanych

Podstawowe kroki tworzenia zdarzenia kierowanego to:

1. Zarejestruj przy RoutedEvent użyciu RegisterRoutedEvent metody .
2. Wywołanie rejestracji zwraca wystąpienie znane jako identyfikator zdarzenia kierowanego, który zawiera zarejestrowaną `RoutedEvent` nazwę zdarzenia, strategię routingu i inne szczegóły zdarzenia. Przypisz identyfikator do statycznego pola tylko do odczytu. Zgodnie z konwencją:
    - Identyfikator zdarzenia kierowanego ze strategią bubbling nosi nazwę `<event name>Event`. Jeśli na przykład nazwa zdarzenia to `Tap` identyfikator powinien mieć nazwę `TapEvent`.

- Identyfikator zdarzenia kierowanego ze strategią tunelowania nosi nazwę `Preview<event name>Event`. Jeśli na przykład nazwa zdarzenia to `Tap` identyfikator powinien mieć nazwę `PreviewTapEvent`.

3. Zdefiniuj metody dostępu do zdarzeń CLR i usuń je. Bez metod dostępu do zdarzeń CLR można dodawać lub usuwać programy obsługi zdarzeń tylko za pomocą wywołań bezpośrednich do UIElement.AddHandler metod i UIElement.RemoveHandler . Dzięki akcesorom zdarzeń CLR uzyskasz następujące mechanizmy przypisywania programu obsługi zdarzeń:

   - W przypadku języka XAML (Extensible Application Markup Language) można użyć składni atrybutów w celu dodania procedur obsługi zdarzeń.
   - W języku C# można użyć `+=` operatorów i `-=` do dodawania lub usuwania procedur obsługi zdarzeń.
   - W przypadku języka VB można użyć instrukcji AddHandler i RemoveHandler , aby dodać lub usunąć programy obsługi zdarzeń.

4. Dodaj logikę niestandardową do wyzwalania zdarzenia kierowanego. Na przykład logika może wyzwolić zdarzenie na podstawie danych wejściowych użytkownika i stanu aplikacji.

Przykład

Poniższy przykład implementuje klasę `CustomButton` w niestandardowej bibliotece kontrolek. Klasa `CustomButton` , która pochodzi z klasy Button:

1. Rejestruje RoutedEvent nazwę przy `ConditionalClick` użyciu RegisterRoutedEvent metody i określa strategię bubbling podczas rejestracji.
2. Przypisuje wystąpienie zwrócone `RoutedEvent` z wywołania rejestracji do statycznego pola readonly o nazwie `ConditionalClickEvent`.
3. Definiuje clR dodawanie i usuwanie akcesorów zdarzeń.
4. Dodaje logikę niestandardową w celu podniesienia niestandardowego zdarzenia kierowanego po kliknięciu `CustomButton` elementu i zastosowania warunku zewnętrznego. Mimo że przykładowy kod zgłasza `ConditionalClick` zdarzenie kierowane z metody wirtualnej przesłoniętej `OnClick` , możesz zgłosić zdarzenie w dowolny sposób.

C#

```
public class CustomButton : Button
{
    // Register a custom routed event using the Bubble routing strategy.
```

```
    public static readonly RoutedEvent ConditionalClickEvent =
EventManager.RegisterRoutedEvent(
        name: "ConditionalClick",
        routingStrategy: RoutingStrategy.Bubble,
        handlerType: typeof(RoutedEventHandler),
        ownerType: typeof(CustomButton));

    // Provide CLR accessors for assigning an event handler.
    public event RoutedEventHandler ConditionalClick
    {
        add { AddHandler(ConditionalClickEvent, value); }
        remove { RemoveHandler(ConditionalClickEvent, value); }
    }

    void RaiseCustomRoutedEvent()
    {
        // Create a RoutedEventArgs instance.
        RoutedEventArgs routedEventArgs = new(routedEvent:
ConditionalClickEvent);

        // Raise the event, which will bubble up through the element tree.
        RaiseEvent(routedEventArgs);
    }

    // For demo purposes, we use the Click event as a trigger.
    protected override void OnClick()
    {
        // Some condition combined with the Click event will trigger the
ConditionalClick event.
        if (DateTime.Now > new DateTime())
            RaiseCustomRoutedEvent();

        // Call the base class OnClick() method so Click event subscribers
are notified.
        base.OnClick();
    }
}
```

Przykład obejmuje oddzielną aplikację WPF, która używa znaczników XAML w celu dodania wystąpienia elementu do StackPanelklasy , oraz przypisania `Handler_ConditionalClick` metody jako `ConditionalClick` procedury obsługi zdarzeń dla `CustomButton` elementów i `StackPanel1` .`CustomButton`

XAML

```
<Window x:Class="CodeSample.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:WpfControl;assembly=WpfControlLibrary"
        Title="How to create a custom routed event" Height="100"
Width="300">
```

```
    <StackPanel Name="StackPanel1"
custom:CustomButton.ConditionalClick="Handler_ConditionalClick">
        <custom:CustomButton
            Name="customButton"
            ConditionalClick="Handler_ConditionalClick"
            Content="Click to trigger a custom routed event"
            Background="LightGray">
        </custom:CustomButton>
    </StackPanel>
</Window>
```

W kodzie aplikacja WPF definiuje metodę `Handler_ConditionalClick` obsługi zdarzeń. Metody obsługi zdarzeń można zaimplementować tylko w kodzie.

C#

```
// The ConditionalClick event handler.
private void Handler_ConditionalClick(object sender, RoutedEventArgs e)
{
    string senderName = ((FrameworkElement)sender).Name;
    string sourceName = ((FrameworkElement)e.Source).Name;

    Debug.WriteLine($"Routed event handler attached to {senderName}, " +
        $"triggered by the ConditionalClick routed event raised on
{sourceName}.");
}

// Debug output when CustomButton is clicked:
// Routed event handler attached to CustomButton,
//     triggered by the ConditionalClick routed event raised on
CustomButton.
// Routed event handler attached to StackPanel1,
//     triggered by the ConditionalClick routed event raised on
CustomButton.
```

Po `CustomButton` kliknięciu:

1. Zdarzenie `ConditionalClick` kierowane jest wywoływane w dniu `CustomButton`.
2. Wyzwalana `Handler_ConditionalClick` jest procedura obsługi zdarzeń dołączona do `CustomButton` .
3. Przekierowane `ConditionalClick` zdarzenie przechodzi w górę drzewa elementów do .`StackPanel1`
4. Wyzwalana `Handler_ConditionalClick` jest procedura obsługi zdarzeń dołączona do `StackPanel1` .
5. Zdarzenie `ConditionalClick` kierowane kontynuuje drzewo elementów potencjalnie wyzwalając inne `ConditionalClick` programy obsługi zdarzeń dołączone do innych elementów przechodzących.

Procedura `Handler_ConditionalClick` obsługi zdarzeń uzyskuje następujące informacje o zdarzeniu, które go wyzwoliło:

- Obiekt nadawcy , który jest elementem dołączonym do programu obsługi zdarzeń. Element `sender` będzie `CustomButton` pierwszym uruchomieniem programu obsługi, a `StackPanel1` po raz drugi.
- Obiekt RoutedEventArgs.Source , który jest elementem, który pierwotnie wywołał zdarzenie. W tym przykładzie parametr `Source` ma zawsze `CustomButton`wartość .

> **Uwaga**
>
> Kluczową różnicą między zdarzeniem kierowanym a zdarzeniem CLR jest to, że kierowane zdarzenie przechodzi przez drzewo elementów, szukając procedur obsługi, podczas gdy zdarzenie CLR nie przechodzi przez drzewo elementów, a programy obsługi mogą dołączać tylko do obiektu źródłowego, który wywołał zdarzenie. W rezultacie zdarzenie `sender` kierowane może być dowolnym elementem przechodzącym w drzewie elementów.

Zdarzenie tunelowania można utworzyć w taki sam sposób jak zdarzenie bubbling, z wyjątkiem ustawienia strategii routingu w wywołaniu rejestracji zdarzeń na Tunnel. Aby uzyskać więcej informacji na temat zdarzeń tunelowania, zobacz Zdarzenia wejściowe WPF.

Zobacz też

- Omówienie zdarzeń trasowanych
- Omówienie danych wejściowych
- Omówienie tworzenia kontrolek
- Obsługa zdarzenia kierowanego.

Omówienie właściwości zależności (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 11 min

Windows Presentation Foundation (WPF) udostępnia zestaw usług, których można użyć do rozszerzenia funkcjonalności właściwości typu. Zbiorczo te usługi są określane jako system właściwości WPF. Właściwość wspierana przez system właściwości WPF jest nazywana właściwością zależności. W tym omówieniu opisano system właściwości WPF i możliwości właściwości zależności, w tym sposób używania istniejących właściwości zależności w języku XAML i w kodzie. W tym omówieniu przedstawiono również wyspecjalizowane aspekty właściwości zależności, takie jak metadane właściwości zależności i sposób tworzenia własnej właściwości zależności w klasie niestandardowej.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że podstawowa wiedza na temat systemu typów .NET i programowania obiektowego. Aby postępować zgodnie z przykładami w tym artykule, pomaga zrozumieć język XAML i wiedzieć, jak pisać aplikacje WPF. Aby uzyskać więcej informacji, zobacz Samouczek: tworzenie nowej aplikacji WPF przy użyciu platformy .NET.

Właściwości zależności i właściwości środowiska CLR

Właściwości platformy WPF są zwykle uwidocznione jako standardowe właściwości platformy .NET. Możesz wchodzić w interakcje z tymi właściwościami na poziomie podstawowym i nigdy nie wiedzieć, że są one implementowane jako właściwość zależności. Jednak znajomość niektórych lub wszystkich funkcji systemu właściwości WPF pomoże Ci skorzystać z tych funkcji.

Celem właściwości zależności jest zapewnienie sposobu obliczenia wartości właściwości na podstawie wartości innych danych wejściowych, takich jak:

- Właściwości systemowe, takie jak motywy i preferencje użytkownika.
- Mechanizmy określania właściwości just in time, takie jak powiązanie danych i animacje/scenorysy.
- Szablony wielokrotnego użytku, takie jak zasoby i style.
- Wartości znane za pośrednictwem relacji nadrzędny-podrzędny z innymi elementami w drzewie elementów.

Ponadto właściwość zależności może zapewnić:

- Samodzielna walidacja.
- Wartości domyślne.
- Wywołania zwrotne monitorujące zmiany innych właściwości.
- System, który może przekształcać wartości właściwości na podstawie informacji o środowisku uruchomieniowym.

Klasy pochodne mogą zmieniać niektóre cechy istniejącej właściwości, przesłaniając metadane właściwości zależności, zamiast zastępować rzeczywistą implementację istniejących właściwości lub tworzenie nowych właściwości.

W dokumentacji zestawu SDK można zidentyfikować właściwość zależności przez obecność sekcji Informacje o właściwości zależności na stronie dokumentacji zarządzanej dla tej właściwości. Sekcja Informacje o właściwości zależności zawiera link do pola identyfikatora DependencyProperty dla tej właściwości zależności. Zawiera również listę opcji metadanych dla tej właściwości, informacje o zastąpieniu poszczególnych klas i inne szczegóły.

Właściwości zależności z powrotem właściwości środowiska CLR

Właściwości zależności i system właściwości WPF rozszerzają funkcjonalność właściwości, udostępniając typ, który tworzy kopię zapasową właściwości, jako alternatywę dla standardowego wzorca tworzenia kopii zapasowej właściwości z polem prywatnym. Nazwa tego typu to DependencyProperty. Innym ważnym typem definiującym system właściwości WPF jest DependencyObject, który definiuje klasę bazową, która może rejestrować i być właścicielem właściwości zależności.

Oto kilka często używanych terminologii:

- **Właściwość zależności**, która jest właściwością wspieraną przez element DependencyProperty.

- **Identyfikator właściwości zależności**, który jest wystąpieniem `DependencyProperty` uzyskanym jako wartość zwracana podczas rejestrowania właściwości zależności, a następnie przechowywany jako statyczny element członkowski klasy. Wiele interfejsów API korzystających z systemu właściwości WPF używa identyfikatora właściwości zależności jako parametru.

- **CLR "otoka"**, czyli `get` implementacje właściwości i `set` . Te implementacje zawierają identyfikator właściwości zależności, używając go w wywołaniach GetValue i SetValue . W ten sposób system właściwości WPF zapewnia obsługę kopii zapasowej właściwości.

W poniższym przykładzie zdefiniowano `IsSpinning` właściwość zależności, aby pokazać relację identyfikatora `DependencyProperty` z właściwością, która jest wspierana.

C#

```
public static readonly DependencyProperty IsSpinningProperty =
DependencyProperty.Register(
    "IsSpinning", typeof(bool),
    typeof(MainWindow)
    );

public bool IsSpinning
{
    get => (bool)GetValue(IsSpinningProperty);
    set => SetValue(IsSpinningProperty, value);
}
```

Ważna jest konwencja nazewnictwa właściwości i jej pola zapasowego DependencyProperty . Nazwa pola jest zawsze nazwą właściwości z dołączonym sufiksem `Property` . Aby uzyskać więcej informacji na temat tej konwencji i jej przyczyn, zobacz Właściwości zależności niestandardowych.

Ustawianie wartości właściwości

Właściwości można ustawić w kodzie lub W języku XAML.

Ustawianie wartości właściwości w języku XAML

Poniższy przykład XAML ustawia kolor tła przycisku na czerwony. Wartość ciągu atrybutu XAML jest konwertowana przez analizator XAML WPF na typ WPF. W wygenerowanym kodzie typ WPF jest Colormetodą .SolidColorBrush

XAML

```
<Button Content="I am red" Background="Red"/>
```

Język XAML obsługuje kilka formularzy składni dla właściwości ustawień. Składnia używana dla określonej właściwości zależy od typu wartości używanego przez właściwość i innych czynników, takich jak obecność konwertera typów. Aby uzyskać więcej informacji na temat składni XAML dla właściwości ustawień, zobacz XAML w składni WPF i XAML .

Poniższy przykład XAML przedstawia inne tło przycisku, które używa składni elementu właściwości zamiast składni atrybutu. Zamiast ustawiać prosty pełny kolor, język XAML ustawia właściwość przycisku `Background` na obraz. Element reprezentuje ten obraz, a atrybut zagnieżdżonego elementu określa źródło obrazu.

XAML

```
<Button Content="I have an image background">
    <Button.Background>
        <ImageBrush ImageSource="stripes.jpg"/>
    </Button.Background>
</Button>
```

Ustawianie właściwości w kodzie

Ustawienie wartości właściwości zależności w kodzie jest zwykle tylko wywołaniem `set` implementacji uwidocznionej przez klasę "otoki":

C#

```
Button myButton = new();
myButton.Width = 200.0;
```

Uzyskanie wartości właściwości jest zasadniczo wywołaniem `get` implementacji "otoki":

C#

```
double whatWidth = myButton.Width;
```

Możesz również wywołać interfejsy GetValue API systemu właściwości i SetValue bezpośrednio. Bezpośrednie wywoływanie interfejsów API jest odpowiednie dla niektórych scenariuszy, ale zwykle nie w przypadku używania istniejących właściwości. Zazwyczaj otoki są wygodniejsze i zapewniają lepszą ekspozycję właściwości dla narzędzi deweloperskich.

Właściwości można również ustawić w języku XAML, a następnie uzyskiwać do ich dostępu w dalszej części kodu za pomocą kodu. Aby uzyskać szczegółowe informacje, zobacz Code-behind and XAML in WPF (Kod za pomocą kodu i XAML w WPF).

Funkcje właściwości udostępniane przez właściwość zależności

W przeciwieństwie do właściwości, która jest wspierana przez pole, właściwość zależności rozszerza funkcjonalność właściwości. Często dodatkowa funkcjonalność reprezentuje lub obsługuje jedną z następujących funkcji:

- Zasoby
- Powiązanie danych
- Style
- Animacje
- Przesłonięcia metadanych
- Dziedziczenie wartości właściwości
- Integracja z projektantem WPF

Zasoby

Wartość właściwości zależności można ustawić, odwołując się do zasobu. Zasoby są zwykle określane jako `Resources` wartość właściwości elementu głównego strony lub aplikacji, ponieważ te lokalizacje oferują wygodny dostęp do zasobu. W tym przykładzie SolidColorBrush zdefiniujemy zasób:

XAML

```
<StackPanel.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
</StackPanel.Resources>
```

Teraz, gdy zasób jest zdefiniowany, możemy odwołać się do zasobu, aby podać wartość właściwości `Background` :

XAML

```
<Button Background="{DynamicResource MyBrush}" Content="I am gold" />
```

W języku WPF XAML można użyć odwołania do zasobów statycznych lub dynamicznych. Ten konkretny zasób jest przywoływał jako element DynamicResource. Dynamiczne odwołanie do zasobów może służyć tylko do ustawiania właściwości zależności, dlatego jest to użycie odwołania do zasobów dynamicznych, które jest włączone przez system właściwości WPF. Aby uzyskać więcej informacji, zobacz Zasoby XAML.

> ⓘ **Uwaga**
>
> Zasoby są traktowane jako wartość lokalna, co oznacza, że jeśli ustawisz inną wartość lokalną, usuniesz odwołanie do zasobu. Aby uzyskać więcej informacji, zobacz **Pierwszeństwo wartości właściwości zależności**.

Powiązanie danych

Właściwość zależności może odwoływać się do wartości za pomocą powiązania danych. Powiązanie danych działa za pomocą określonej składni rozszerzenia znaczników w języku XAML lub Binding obiektu w kodzie. W przypadku powiązania danych określenie ostatecznej wartości właściwości jest odroczone do czasu wykonywania, w którym czasie wartość jest uzyskiwana ze źródła danych.

Poniższy przykład ustawia Content właściwość dla Buttonklasy przy użyciu powiązania zadeklarowanego w języku XAML. Powiązanie używa dziedziczonego kontekstu danych i XmlDataProvider źródła danych (nie pokazano). Samo powiązanie określa właściwość źródłową w źródle danych przez .XPath

XAML

```
<Button Content="{Binding Source={StaticResource TestData},
XPath=test[1]/@text}"/>
```

> ⓘ **Uwaga**
>
> Powiązania są traktowane jako wartość lokalna, co oznacza, że jeśli ustawisz inną wartość lokalną, usuniesz powiązanie. Aby uzyskać szczegółowe informacje, zobacz **Pierwszeństwo wartości właściwości zależności**.

Właściwości zależności lub DependencyObject klasa nie obsługują INotifyPropertyChanged natywnie powiadomień o zmianach wartości `DependencyObject` właściwości źródłowej dla operacji powiązania danych. Aby uzyskać więcej informacji na

temat tworzenia właściwości do użycia w powiązaniu danych, które mogą zgłaszać zmiany w obiekcie docelowym powiązania danych, zobacz Omówienie powiązań danych.

Style

Style i szablony są atrakcyjnymi przyczynami używania właściwości zależności. Style są szczególnie przydatne w przypadku ustawiania właściwości definiujących interfejs użytkownika aplikacji. Style są zwykle definiowane jako zasoby w języku XAML. Style wchodzą w interakcję z systemem właściwości, ponieważ zwykle zawierają "metody ustawiające" dla określonych właściwości i "wyzwalacze", które zmieniają wartość właściwości na podstawie wartości środowiska uruchomieniowego dla innej właściwości.

Poniższy przykład tworzy prosty styl, który zostanie zdefiniowany wewnątrz słownika Resources (nie pokazano). Następnie styl jest stosowany bezpośrednio do Style właściwości obiektu Button. Ustawianie w obrębie stylu ustawia Background właściwość dla stylu `Button` na zielony.

XAML

```
<Style x:Key="GreenButtonStyle">
    <Setter Property="Control.Background" Value="Green"/>
</Style>
```

XAML

```
<Button Style="{StaticResource GreenButtonStyle}" Content="I am green"/>
```

Aby uzyskać więcej informacji, zobacz Styling and templating (Tworzenie stylów i tworzenie szablonów).

Animacje

Właściwości zależności można animować. Po uruchomieniu zastosowanej animacji animowana wartość ma wyższy priorytet niż jakakolwiek inna wartość właściwości, w tym wartość lokalna.

Poniższy przykład animuje Background właściwość obiektu Button. Technicznie składnia elementu właściwości ustawia wartość pustą SolidColorBrush jako `Background`, a Color właściwość obiektu `SolidColorBrush` jest animowana.

XAML

```
<Button Content="I am animated">
    <Button.Background>
        <SolidColorBrush x:Name="AnimBrush"/>
    </Button.Background>
    <Button.Triggers>
        <EventTrigger RoutedEvent="FrameworkElement.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <ColorAnimation
                        Storyboard.TargetName="AnimBrush"
                        Storyboard.TargetProperty="(SolidColorBrush.Color)"
                        From="Blue" To="White" Duration="0:0:1"
                        AutoReverse="True" RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Button.Triggers>
</Button>
```

Aby uzyskać więcej informacji na temat animowania właściwości, zobacz Omówienie animacji i Omówienie scenorysów.

Przesłonięcia metadanych

Określone zachowania właściwości zależności można zmienić, przesłaniając jej metadane, gdy pochodzisz z klasy, która pierwotnie zarejestrowała właściwość zależności. Zastępowanie metadanych opiera się na identyfikatorze DependencyProperty i nie wymaga ponownego implementowania właściwości. Zmiana metadanych jest obsługiwana natywnie przez system właściwości. Każda klasa potencjalnie przechowuje poszczególne metadane dla wszystkich właściwości dziedziczonej z klas bazowych dla poszczególnych typów.

Poniższy przykład zastępuje metadane właściwości DefaultStyleKey zależności. Zastępowanie metadanych dla tej konkretnej właściwości zależności jest częścią wzorca implementacji do tworzenia kontrolek, które mogą używać domyślnych stylów z motywów.

C#

```
public class SpinnerControl : ItemsControl
{
    static SpinnerControl() => DefaultStyleKeyProperty.OverrideMetadata(
            typeof(SpinnerControl),
            new FrameworkPropertyMetadata(typeof(SpinnerControl))
        );
}
```

Aby uzyskać więcej informacji na temat zastępowania lub uzyskiwania dostępu do metadanych dla właściwości zależności, zobacz Zastępowanie metadanych dla właściwości zależności.

Dziedziczenie wartości właściwości

Element może dziedziczyć wartość właściwości zależności od jej obiektu nadrzędnego w drzewie obiektów.

> ⓘ **Uwaga**
>
> Zachowanie dziedziczenia wartości właściwości nie jest globalnie włączone dla wszystkich właściwości zależności, ponieważ czas obliczania dziedziczenia wpływa na wydajność. Dziedziczenie wartości właściwości jest zwykle włączone tylko w scenariuszach, które sugerują możliwość stosowania. Możesz sprawdzić, czy właściwość zależności dziedziczy, przeglądając sekcję **Informacje o właściwości zależności** dla tej właściwości zależności w dokumentacji zestawu SDK.

W poniższym przykładzie pokazano powiązanie zawierające właściwość określającą DataContext źródło powiązania. Dlatego powiązania w obiektach podrzędnych nie muszą określać źródła i mogą używać dziedziczonej wartości z `DataContext` obiektu nadrzędnego StackPanel . Lub obiekt podrzędny może bezpośrednio określić własne `DataContext` lub w SourceBindingobiekcie , a nie używać dziedziczonej wartości.

XAML

```
<StackPanel Canvas.Top="50" DataContext="{Binding Source={StaticResource
TestData}}">
    <Button Content="{Binding XPath=test[2]/@text}"/>
</StackPanel>
```

Aby uzyskać więcej informacji, zobacz Dziedziczenie wartości właściwości.

Integracja projektanta WPF

Kontrolki niestandardowe z właściwościami zaimplementowanymi jako właściwości zależności dobrze integrują się z projektantem WPF dla programu Visual Studio. Przykładem jest możliwość edytowania właściwości zależności bezpośrednich i dołączonych w oknie **Właściwości** . Aby uzyskać więcej informacji, zobacz Omówienie tworzenia kontrolek.

Następstwo wartości właściwości zależności

Każdy z danych wejściowych opartych na właściwościach w systemie właściwości WPF może ustawić wartość właściwości zależności. Pierwszeństwo wartości właściwości zależności istnieje tak, aby różne scenariusze dotyczące sposobu, w jaki właściwości uzyskują swoje wartości, współdziałają w przewidywalny sposób.

> **Uwaga**
>
> Dokumentacja zestawu SDK czasami używa terminu "wartość lokalna" lub "lokalnie ustawiona wartość" podczas omawiania właściwości zależności. Lokalnie ustawiona wartość to wartość właściwości ustawiana bezpośrednio w wystąpieniu obiektu w kodzie lub jako atrybut elementu w języku XAML.

Następny przykład zawiera styl, który ma zastosowanie do Background właściwości dowolnego przycisku, ale określa jeden przycisk z lokalnie ustawioną `Background` właściwością. Technicznie ten przycisk ma ustawioną `Background` dwukrotnie właściwość , chociaż ma zastosowanie tylko jedna wartość — wartość o najwyższym prioryencie. Wartość ustawiona lokalnie ma najwyższy priorytet, z wyjątkiem uruchomionej animacji, która nie istnieje w tym miejscu. Dlatego drugi przycisk używa lokalnie ustawionej wartości właściwości `Background` zamiast wartości ustawiającej styl. Pierwszy przycisk nie ma wartości lokalnej ani innej wartości o wyższym priorytecie niż zestaw stylów, dlatego używa wartości ustawiającej styl dla `Background` właściwości.

XAML

```
<StackPanel>
    <StackPanel.Resources>
        <Style x:Key="{x:Type Button}" TargetType="{x:Type Button}">
            <Setter Property="Background" Value="Orange"/>
        </Style>
    </StackPanel.Resources>
    <Button>I am styled orange</Button>
    <Button Background="Pink">I am locally set to pink (not styled orange)
</Button>
</StackPanel>
```

Dlaczego pierwszeństwo właściwości zależności istnieje?

Lokalnie ustawione wartości mają pierwszeństwo przed wartościami ustawiania stylów, które obsługują lokalną kontrolę właściwości elementu. Aby uzyskać szczegółowe informacje, zobacz Pierwszeństwo wartości właściwości zależności.

> **Uwaga**
>
> Wiele właściwości zdefiniowanych w elementach WPF nie jest właściwościami zależności, ponieważ właściwości zależności były zwykle implementowane tylko wtedy, gdy wymagana była funkcja systemu właściwości WPF. Funkcje obejmują powiązanie danych, styl, animację, obsługę wartości domyślnych, dziedziczenie, dołączone właściwości i unieważnienie.

Dowiedz się więcej o właściwościach zależności

- Deweloperzy składników lub deweloperzy aplikacji mogą chcieć utworzyć własną właściwość zależności, aby dodać możliwości, takie jak obsługa powiązania danych lub stylów, lub unieważnienie i obsługa przymusu wartości. Aby uzyskać więcej informacji, zobacz Właściwości zależności niestandardowych.

- Należy wziąć pod uwagę właściwości zależności jako właściwości publiczne, dostępne lub wykrywalne przez dowolny obiekt wywołujący z dostępem do wystąpienia. Aby uzyskać więcej informacji, zobacz Zabezpieczenia właściwości zależności.

- Dołączona właściwość jest typem właściwości, która obsługuje wyspecjalizowaną składnię w języku XAML. Dołączona właściwość często nie ma korespondencji 1:1 z właściwością środowiska uruchomieniowego języka wspólnego i nie musi być właściwością zależności. Głównym celem dołączonej właściwości jest zezwolenie elementom podrzędnym na zgłaszanie wartości właściwości do elementu nadrzędnego, nawet jeśli element nadrzędny i element podrzędny nie zawierają tej właściwości w ramach list składowych klasy. Jednym z podstawowych scenariuszy jest włączenie elementu podrzędnego w celu poinformowania elementów nadrzędnych, jak je przedstawić w interfejsie użytkownika. Przykłady można znaleźć w temacie Dock i Left. Aby uzyskać więcej informacji, zobacz Omówienie dołączonych właściwości.

Zobacz też

- Niestandardowe właściwości zależności
- Właściwości zależności tylko do odczytu
- XAML w WPF
- Architektura WPF

Pierwszeństwo wartości właściwości zależności (WPF .NET)

Artykuł • 13.02.2023 • Czas czytania: 11 min

Działanie systemu właściwości Windows Presentation Foundation (WPF) wpływa na wartość właściwości zależności. W tym artykule wyjaśniono, jak pierwszeństwo różnych danych wejściowych opartych na właściwościach w systemie właściwości WPF określa efektywną wartość właściwości zależności.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

System właściwości WPF

System właściwości WPF używa różnych czynników do określenia wartości właściwości zależności, takich jak walidacja właściwości w czasie rzeczywistym, późne powiązanie i powiadomienia o zmianie właściwości dla powiązanych właściwości. Mimo że kolejność i logika używana do określania wartości właściwości zależności jest złożona, uczenie się może pomóc uniknąć niepotrzebnych ustawień właściwości, a także dowiedzieć się, dlaczego próba ustawienia właściwości zależności nie spowodowało oczekiwanej wartości.

Właściwości zależności ustawione w wielu miejscach

Poniższy przykład XAML pokazuje, jak trzy różne operacje "set" we właściwości przycisku Background mogą mieć wpływ na jego wartość.

```
XAML
```

```
<StackPanel>
    <StackPanel.Resources>
        <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type
Button}">
            <Border Background="{TemplateBinding Background}"
BorderThickness="{TemplateBinding BorderThickness}"
                    BorderBrush="{TemplateBinding BorderBrush}">
                <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
            </Border>
        </ControlTemplate>
    </StackPanel.Resources>

    <Button Template="{StaticResource ButtonTemplate}" Background="Red">
        <Button.Style>
            <Style TargetType="{x:Type Button}">
                <Setter Property="Background" Value="Blue"/>
                <Style.Triggers>
                    <Trigger Property="IsMouseOver" Value="True">
                        <Setter Property="Background" Value="Yellow" />
                    </Trigger>
                </Style.Triggers>
            </Style>
        </Button.Style>
        Which color do you expect?
    </Button>
</StackPanel>
```

W tym przykładzie `Background` właściwość jest lokalnie ustawiona na `Red` wartość . Jednak niejawny styl zadeklarowany w zakresie przycisku próbuje ustawić `Background` właściwość na `Blue`. Gdy mysz jest nad przyciskiem, wyzwalacz w niejawny stylu próbuje ustawić `Background` właściwość na `Yellow`. Z wyjątkiem przymusu i animacji wartość właściwości ustawionej lokalnie ma najwyższy priorytet, więc przycisk będzie czerwony — nawet w przypadku myszy. Jeśli jednak usuniesz lokalnie ustawioną wartość z przycisku, zostanie ona pobrana `Background` z stylu. W stylu wyzwalacze mają pierwszeństwo, więc przycisk będzie żółty na myszy i w przeciwnym razie niebieski. W przykładzie jest zastępowana wartość domyślna ControlTemplate przycisku, ponieważ szablon domyślny ma trwale zakodowaną wartość myszy `Background` .

Lista pierwszeństwa właściwości zależności

Poniższa lista to ostateczna kolejność pierwszeństwa używana przez system właściwości podczas przypisywania wartości środowiska uruchomieniowego do właściwości zależności. Najwyższy priorytet jest wymieniony jako pierwszy.

1. **Przymus systemu właściwości**. Aby uzyskać więcej informacji na temat przymusu, zobacz Coercion i animacje.

2. **Aktywne animacje lub animacje z zachowaniem blokady**. Aby mieć praktyczny efekt, wartość animacji musi mieć pierwszeństwo przed wartością podstawową (bezanimowaną), nawet jeśli wartość podstawowa została ustawiona lokalnie. Aby uzyskać więcej informacji, zobacz Coercion i animacje.

3. **Wartości lokalne**. Wartość lokalną można ustawić za pomocą właściwości "otoki", która jest równa ustawieniu atrybutu lub elementu właściwości w języku XAML lub wywołaniu interfejsu SetValue API przy użyciu właściwości określonego wystąpienia. Wartość lokalna ustawiona za pośrednictwem powiązania lub zasobu będzie mieć taki sam pierwszeństwo jak wartość, która jest ustawiana bezpośrednio.

4. **TemplatedParent wartości właściwości szablonu**. Element ma TemplatedParent wartość , jeśli została utworzona przez szablon (ControlTemplate lub DataTemplate). Aby uzyskać więcej informacji, zobacz TemplatedParent. W szablonie `TemplatedParent` określonym przez element kolejność pierwszeństwa to:

   a. Wyzwalaczy.

   b. Zestawy właściwości, zazwyczaj za pomocą atrybutów XAML.

5. **Niejawne style**. Dotyczy tylko Style właściwości . Wartość `Style` to dowolny zasób stylu z wartością zgodną TargetType z typem elementu. Zasób stylu musi istnieć w obrębie strony lub aplikacji. Wyszukiwanie niejawnego zasobu stylu nie rozszerza się na zasoby stylów w motywach.

6. **Wyzwalacze stylu**. Wyzwalacz stylu jest wyzwalaczem w jawnym lub niejawnym stylu. Styl musi istnieć na stronie lub w aplikacji. Wyzwalacze w stylach domyślnych mają niższy priorytet.

7. **Wyzwalacze szablonu**. Wyzwalacz szablonu jest wyzwalaczem bezpośrednio zastosowanym szablonem lub szablonem w stylu. Styl musi istnieć na stronie lub w aplikacji.

8. **Wartości ustawiania stylów**. Wartość ustawiania stylu jest wartością stosowana przez Setter element w stylu. Styl musi istnieć na stronie lub w aplikacji.

9. **Style domyślne**, nazywane również **stylami motywu**. Aby uzyskać więcej informacji, zobacz Domyślne style (motyw). W domyślnym stylu kolejność pierwszeństwa to:

   a. Aktywne wyzwalacze.

b. Ustawiające.

10. **Dziedziczenie**. Niektóre właściwości zależności elementu podrzędnego dziedziczą ich wartość z elementu nadrzędnego. Dlatego może nie być konieczne ustawienie wartości właściwości dla każdego elementu w całej aplikacji. Aby uzyskać więcej informacji, zobacz Dziedziczenie wartości właściwości.

11. **Wartość domyślna z metadanych właściwości zależności** Właściwość zależności może mieć wartość domyślną ustawioną podczas rejestracji systemu właściwości tej właściwości. Klasy pochodne dziedziczące właściwość zależności mogą zastępować metadane właściwości zależności (w tym wartość domyślną) na podstawie poszczególnych typów. Aby uzyskać więcej informacji, zobacz Metadane właściwości zależności. W przypadku właściwości dziedziczonej wartość domyślna elementu nadrzędnego ma pierwszeństwo przed wartością domyślną elementu podrzędnego. Jeśli więc właściwość dziedziczona nie jest ustawiona, zostanie użyta domyślna wartość elementu głównego lub nadrzędnego zamiast domyślnej wartości elementu podrzędnego.

SzablondParent

TemplatedParent pierwszeństwo nie ma zastosowania do właściwości elementów zadeklarowanych bezpośrednio w standardowym adiustacji aplikacji. Koncepcja `TemplatedParent` istnieje tylko dla elementów podrzędnych w drzewie wizualnym, które istnieją za pośrednictwem zastosowania szablonu. Gdy system właściwości przeszukuje szablon określony przez `TemplatedParent` wartości właściwości elementu, wyszukuje szablon, który utworzył element. Wartości właściwości szablonu `TemplatedParent` zwykle działają tak, jakby były lokalnie ustawiane wartości w elemencie, ale z mniejszym pierwszeństwem niż rzeczywiste wartości lokalne, ponieważ szablony są potencjalnie współużytkowane. Aby uzyskać więcej informacji, zobacz TemplatedParent.

Właściwość Style

Ta sama kolejność pierwszeństwa ma zastosowanie do wszystkich właściwości zależności, z wyjątkiem Style właściwości . Właściwość jest unikatowa `Style` w tym, że nie może być stylizowana. Wymuszanie lub animowanie `Style` właściwości nie jest zalecane (a animowanie `Style` właściwości wymaga niestandardowej klasy animacji). W związku z tym nie wszystkie elementy pierwszeństwa mają zastosowanie. Istnieją tylko trzy sposoby ustawiania `Style` właściwości:

- **Styl jawny.** Właściwość `Style` elementu jest ustawiana bezpośrednio. Wartość `Style` właściwości działa tak, jakby była to wartość lokalna i ma taki sam pierwszeństwo jak element 3 na liście pierwszeństwa. W większości scenariuszy jawne style nie są zdefiniowane w tekście i zamiast tego są jawnie przywołyne jako zasób, na przykład `Style="{StaticResource myResourceKey}"`.

- **Niejawny styl.** Właściwość `Style` elementu nie jest ustawiana bezpośrednio. Zamiast tego styl jest stosowany, gdy istnieje na pewnym poziomie na stronie lub aplikacji i ma klucz zasobu zgodny z typem elementu, do którego ma zastosowanie styl, na przykład `<Style TargetType="x:Type Button">`. Typ musi być dokładnie zgodny, na przykład `<Style TargetType="x:Type Button">` nie zostanie zastosowany do `MyButton` typu, nawet jeśli `MyButton` pochodzi z `Button`klasy . Wartość `Style` właściwości ma taki sam pierwszeństwo jak element 5 na liście pierwszeństwa. Można wykryć niejawną wartość stylu, wywołując metodę DependencyPropertyHelper.GetValueSource , przekazując `Style` właściwość i sprawdzając `ImplicitStyleReference` wyniki.

- **Styl domyślny**, znany również jako **styl motywu**. Właściwość `Style` elementu nie jest ustawiana bezpośrednio. Zamiast tego pochodzi z oceny motywu środowiska uruchomieniowego przez aparat prezentacji WPF. Przed środowiskiem uruchomieniowym `Style` wartość właściwości to `null`. Wartość `Style` właściwości ma taki sam pierwszeństwo jak element 9 na liście pierwszeństwa.

Style domyślne (motyw)

Każda kontrolka dostarczana z WPF ma domyślny styl, który może się różnić w zależności od motywu, dlatego styl domyślny jest czasami określany jako *styl motywu*.

Element ControlTemplate jest ważnym elementem w domyślnym stylu kontrolki. `ControlTemplate` jest wartością ustawianą dla właściwości stylu Template . Jeśli style domyślne nie zawierają szablonu, kontrolka bez szablonu niestandardowego w ramach stylu niestandardowego nie będzie miała wyglądu wizualnego. Szablon nie tylko definiuje wygląd wizualizacji kontrolki, ale także definiuje połączenia między właściwościami w drzewie wizualnym szablonu i odpowiednią klasą kontrolki. Każda kontrolka uwidacznia zestaw właściwości, które mogą wpływać na wygląd wizualizacji kontrolki bez zastępowania szablonu. Rozważmy na przykład domyślny wygląd kontrolki Thumb , która jest składnikiem ScrollBar .

Kontrolka Thumb ma pewne dostosowywalne właściwości. Domyślny szablon kontrolki `Thumb` tworzy podstawową strukturę lub drzewo wizualne z kilkoma zagnieżdżonymi Border składnikami w celu utworzenia wyglądu skośnego. W szablonie właściwości,

które mają być dostosowywane przez klasę `Thumb` , są udostępniane za pośrednictwem szablonuBinding. Domyślny szablon kontrolki `Thumb` ma różne właściwości obramowania, które współużytkuje powiązanie szablonu z właściwościami, takimi jak Background lub BorderThickness. Jednak jeśli wartości właściwości lub układów wizualnych są trwale zakodowane w szablonie lub są powiązane z wartościami pochodzącymi bezpośrednio z motywu, można zmienić te wartości tylko przez zastąpienie całego szablonu. Ogólnie rzecz biorąc, jeśli właściwość pochodzi z szablonowego elementu nadrzędnego i nie jest uwidaczniona przez `TemplateBinding`element , wartość właściwości nie może zostać zmieniona przez style, ponieważ nie ma wygodnego sposobu, aby go kierować. Jednak ta właściwość może nadal mieć wpływ na dziedziczenie wartości właściwości w zastosowanym szablonie lub przez wartość domyślną.

Style domyślne określają TargetType element w swoich definicjach. Ocena motywu środowiska uruchomieniowego jest zgodna z `TargetType` domyślnym stylem do DefaultStyleKey właściwości kontrolki. Z kolei zachowanie wyszukiwania dla stylów niejawnych używa rzeczywistego typu kontrolki. Wartość `DefaultStyleKey` jest dziedziczona przez klasy pochodne, więc elementy pochodne, które w przeciwnym razie nie mają skojarzonego stylu, uzyskać domyślny wygląd wizualizacji. Jeśli na przykład pochodzisz `MyButton` z Buttonklasy , `MyButton` dziedziczy domyślny szablon .`Button` Klasy pochodne mogą zastąpić wartość `DefaultStyleKey` domyślną w metadanych właściwości zależności. W związku z tym, jeśli chcesz innej reprezentacji wizualnej dla `MyButton`elementu , możesz zastąpić metadane właściwości zależności dla `DefaultStyleKey` elementu w `MyButton`systemie , a następnie zdefiniować odpowiedni styl domyślny, w tym szablon, który należy spakować za pomocą `MyButton` kontrolki. Aby uzyskać więcej informacji, zobacz Omówienie tworzenia kontrolek.

Zasób dynamiczny

Dynamiczne odwołania do zasobów i operacje powiązań mają pierwszeństwo przed lokalizacją, w której są ustawione. Na przykład zasób dynamiczny zastosowany do wartości lokalnej ma taki sam priorytet jak element 3 na liście pierwszeństwa. W innym przykładzie dynamiczne powiązanie zasobów zastosowane do ustawiania właściwości w stylu domyślnym ma taki sam priorytet jak element 9 na liście pierwszeństwa. Ponieważ dynamiczne odwołania do zasobów i powiązanie muszą pobierać wartości ze stanu środowiska uruchomieniowego aplikacji, proces określania pierwszeństwa wartości właściwości dla każdej danej właściwości rozciąga się na środowisko uruchomieniowe.

Odwołania do zasobów dynamicznych nie są technicznie częścią systemu właściwości i mają własną kolejność wyszukiwania, która współdziała z listą pierwszeństwa.

Zasadniczo pierwszeństwo dynamicznych odwołań do zasobów to: element do katalogu głównego strony, aplikacja, motyw, a następnie system. Aby uzyskać więcej informacji, zobacz Zasoby XAML.

Mimo że odwołania do zasobów dynamicznych i powiązania mają pierwszeństwo przed lokalizacją, w której są ustawione, wartość jest odroczona. Jedną z konsekwencji jest to, że jeśli ustawisz zasób dynamiczny lub powiązanie z wartością lokalną, każda zmiana wartości lokalnej całkowicie zastąpi zasób dynamiczny lub powiązanie. Nawet jeśli wywołasz metodę ClearValue w celu wyczyszczenia wartości ustawionej lokalnie, zasób dynamiczny lub powiązanie nie zostaną przywrócone. W rzeczywistości, jeśli wywołasz `ClearValue` właściwość, która ma dynamiczny zasób lub powiązanie (bez wartości lokalnej literału), zasób dynamiczny lub powiązanie zostanie wyczyszczone.

SetCurrentValue

Metoda SetCurrentValue jest innym sposobem ustawiania właściwości, ale nie znajduje się na liście pierwszeństwa. `SetCurrentValue` Umożliwia zmianę wartości właściwości bez zastępowania źródła poprzedniej wartości. Jeśli na przykład właściwość jest ustawiana przez wyzwalacz, a następnie przypisujesz inną wartość przy użyciu `SetCurrentValue`metody , następna akcja wyzwalacza ustawi właściwość z powrotem na wartość wyzwalacza. Można użyć `SetCurrentValue` zawsze, gdy chcesz ustawić wartość właściwości bez podawania tej wartości poziomu pierwszeństwa wartości lokalnej. Podobnie można użyć `SetCurrentValue` polecenia , aby zmienić wartość właściwości bez zastępowania powiązania.

Przymus i animacja

Wymuszanie i animacja działają w *oparciu o wartość podstawową*. *Wartość bazowa to wartość* właściwości zależności o najwyższym prioryence, określona przez ocenę w górę przez listę pierwszeństwa do momentu osiągnięcia elementu 2.

Jeśli animacja nie określa zarówno wartości właściwości, jak From i To dla niektórych zachowań lub jeśli animacja celowo powróci do wartości podstawowej po zakończeniu, wartość podstawowa może mieć wpływ na animowaną wartość. Aby to zobaczyć w praktyce, uruchom przykładową aplikację Wartości docelowe . W przykładzie dla wysokości prostokąta spróbuj ustawić początkowe wartości lokalne, które różnią się od dowolnej `From` wartości. Przykładowe animacje zaczynają się od razu przy użyciu `From` wartości zamiast wartości podstawowej. Stop Określając jako wartość , po zakończeniu FillBehavioranimacja spowoduje zresetowanie wartości właściwości do jej wartości

podstawowej. Pierwszeństwo normalne jest używane do określania wartości bazowej po zakończeniu animacji.

Do jednej właściwości można zastosować wiele animacji, a każda animacja ma inny priorytet. Zamiast stosować animację o najwyższym prioryence, aparat prezentacji WPF może składać wartości animacji, w zależności od sposobu definiowania animacji i typu animowanych wartości. Aby uzyskać więcej informacji, zobacz Omówienie animacji.

Przymus znajduje się w górnej części listy pierwszeństwa. Nawet uruchomiona animacja podlega przymusowi wartości. Niektóre istniejące właściwości zależności w WPF mają wbudowane wymuszanie. W przypadku niestandardowych właściwości zależności można zdefiniować zachowanie przymusu, pisząc CoerceValueCallback element przekazywany jako część metadanych podczas tworzenia właściwości. Można również zastąpić zachowanie wymuszania istniejących właściwości, przesłaniając metadane dla tej właściwości w klasie pochodnej. Przymus wchodzi w interakcję z wartością podstawową w taki sposób, że ograniczenia przymusu są stosowane w miarę ich istnienia w czasie, ale wartość podstawowa jest nadal zachowywana. W związku z tym, jeśli ograniczenia przymusu zostaną później zniesione, przymus zwróci najbliższą wartość możliwą do wartości podstawowej, a potencjalnie wpływ przymusu na właściwość przestanie działać, gdy tylko wszystkie ograniczenia zostaną zniesione. Aby uzyskać więcej informacji na temat zachowania wymuszania, zobacz wywołania zwrotne i walidacja właściwości zależności.

Zachowania wyzwalacza

Kontrolki często definiują zachowania wyzwalacza w ramach ich stylu domyślnego. Ustawienie właściwości lokalnych kontrolek może potencjalnie powodować konflikt z tymi wyzwalaczami, uniemożliwiając wyzwalaczom reagowanie (wizualnie lub behawioralnie) na zdarzenia sterowane przez użytkownika. Typowym zastosowaniem wyzwalacza właściwości jest kontrolowanie właściwości stanu, takich jak IsSelected lub IsEnabled. Na przykład, gdy element Button jest wyłączony, wyzwalacz stylu motywu (`IsEnabled` to `false`) ustawia Foreground wartość, aby `Button` wygląd był wyszarzony. Jeśli ustawiono wartość lokalną `Foreground` , wyższa wartość właściwości lokalnej pierwszeństwa spowoduje zastąpienie wartości stylu `Foreground` motywu, nawet jeśli wartość jest wyłączona `Button` . Podczas ustawiania wartości właściwości, które przesłaniają zachowania wyzwalacza na poziomie motywu dla kontrolki, należy zachować ostrożność, aby nie zakłócać zamierzonego środowiska użytkownika dla tej kontrolki.

Clearvalue

Metoda ClearValue czyści dowolną lokalnie zastosowaną wartość właściwości zależności dla elementu. Jednak wywołanie `ClearValue` nie gwarantuje, że wartość domyślna ustanowiona w metadanych podczas rejestracji właściwości jest nową obowiązującą wartością. Wszyscy inni uczestnicy listy pierwszeństwa są nadal aktywni i usuwana jest tylko lokalnie ustawiona wartość. Jeśli na przykład wywołasz `ClearValue` właściwość, która ma styl motywu, wartość stylu motywu zostanie zastosowana jako nowa wartość, a nie domyślna oparta na metadanych. Jeśli chcesz ustawić wartość właściwości na wartość domyślną zarejestrowanych metadanych, pobierz domyślną wartość metadanych, wykonując zapytanie dotyczące metadanych właściwości zależności, a następnie lokalnie ustaw wartość właściwości za pomocą wywołania na SetValue.

Zobacz też

- DependencyObject
- DependencyProperty
- Omówienie właściwości zależności
- Niestandardowe właściwości zależności
- Wywołania zwrotne właściwości zależności i walidacja

Omówienie dołączonych właściwości (WPF .NET)

Artykuł • 11.02.2023 • Czas czytania: 8 min

Dołączona właściwość to koncepcja języka XAML (Extensible Application Markup Language). Dołączone właściwości umożliwiają ustawienie dodatkowych par właściwości/wartości dla dowolnego elementu XAML pochodzącego z DependencyObjectklasy , mimo że element nie definiuje tych dodatkowych właściwości w modelu obiektów. Dodatkowe właściwości są globalnie dostępne. Dołączone właściwości są zwykle definiowane jako wyspecjalizowana forma właściwości zależności, która nie ma konwencjonalnej otoki właściwości.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem XAML i wiedzieć, jak pisać aplikacje Windows Presentation Foundation (WPF).

Dlaczego warto używać dołączonych właściwości

Dołączona właściwość umożliwia elementowi podrzędnym określenie unikatowej wartości właściwości zdefiniowanej w elemecie nadrzędnym. Typowy scenariusz to element podrzędny określający sposób renderowania go w interfejsie użytkownika przez jego element nadrzędny. Na przykład jest dołączoną właściwością, DockPanel.Dock ponieważ jest ona ustawiana na elementach podrzędnych elementu DockPanel, a nie samego `DockPanel` . Klasa `DockPanel` definiuje pole statyczne DependencyProperty o nazwie DockProperty, a następnie udostępnia GetDock metody i SetDock jako publiczne metody dostępu dla dołączonej właściwości.

Dołączone właściwości w języku XAML

W języku XAML ustawiasz dołączone właściwości przy użyciu składni `<attached property provider type>.<property name>`, gdzie dołączony dostawca właściwości jest klasą definiującą dołączoną właściwość. W poniższym przykładzie pokazano, jak element podrzędny DockPanel może ustawić DockPanel.Dock wartość właściwości.

XAML

```
<DockPanel>
    <TextBox DockPanel.Dock="Top">Enter text</TextBox>
</DockPanel>
```

Użycie jest podobne do właściwości statycznej, w której odwołuje się typ, który jest właścicielem i rejestruje dołączoną właściwość (na przykład DockPanel), a nie nazwę wystąpienia.

Po określeniu dołączonej właściwości przy użyciu atrybutu XAML ma zastosowanie tylko akcja ustawiona. Nie można bezpośrednio uzyskać wartości właściwości za pomocą języka XAML, chociaż istnieją pewne mechanizmy pośrednie do porównywania wartości, takich jak wyzwalacze w stylach.

Dołączone właściwości w WPF

Dołączone właściwości są koncepcją XAML, właściwości zależności są koncepcją WPF. W WPF większość właściwości dołączonych interfejsu użytkownika w typach WPF jest implementowana jako właściwości zależności. Właściwości dołączone WPF, które są implementowane jako właściwości zależności obsługują pojęcia właściwości zależności, takie jak metadane właściwości, w tym wartości domyślne z metadanych.

Dołączone modele użycia właściwości

Chociaż dowolny obiekt może ustawić dołączoną wartość właściwości, nie oznacza to ustawienia wartości spowoduje wygenerowanie namacalnego wyniku lub wartość zostanie użyta przez inny obiekt. Głównym celem dołączonych właściwości jest zapewnienie sposobu dla obiektów z szerokiej gamy hierarchii klas i relacji logicznych w celu zgłaszania typowych informacji do typu definiującego dołączoną właściwość. Dołączone użycie właściwości zwykle jest zgodne z jednym z następujących modeli:

- Typ definiujący dołączoną właściwość jest elementem nadrzędnym elementów, które ustawiają wartości dla dołączonej właściwości. Typ nadrzędny iteruje obiekty

podrzędne za pomocą wewnętrznej logiki, która działa na strukturze drzewa obiektów, uzyskuje wartości i działa na tych wartościach w jakiś sposób.
- Typ definiujący dołączoną właściwość jest używany jako element podrzędny dla różnych możliwych elementów nadrzędnych i modeli zawartości.
- Typ definiujący dołączoną właściwość reprezentuje usługę. Inne typy ustawiają wartości dla dołączonej właściwości. Następnie, gdy element, który ustawi właściwość jest oceniany w kontekście usługi, dołączone wartości właściwości są uzyskiwane za pomocą wewnętrznej logiki klasy usługi.

Przykład właściwości dołączonej zdefiniowanej przez obiekt nadrzędny

Typowy scenariusz, w którym WPF definiuje dołączoną właściwość, jest wtedy, gdy element nadrzędny obsługuje kolekcję elementów podrzędnych, a element nadrzędny implementuje zachowanie na podstawie danych zgłaszanych przez poszczególne elementy podrzędne.

DockPanel definiuje dołączoną DockPanel.Dock właściwość. `DockPanel` ma kod na poziomie klasy, w szczególności MeasureOverride i ArrangeOverride, który jest częścią logiki renderowania. Wystąpienie `DockPanel` sprawdza, czy dowolne z jego natychmiastowych elementów podrzędnych ustawiło wartość .`DockPanel.Dock` Jeśli tak, te wartości stają się danymi wejściowymi logiki renderowania zastosowanej do każdego elementu podrzędnego. Chociaż teoretycznie możliwe jest, aby dołączone właściwości wpływały na elementy poza bezpośrednim elementem nadrzędnym, zdefiniowane zachowanie dla wystąpienia zagnieżdżonego `DockPanel` polega na interakcji tylko z jego natychmiastową kolekcją elementów podrzędnych. Dlatego jeśli ustawisz `DockPanel.Dock` element, który nie `DockPanel` ma elementu nadrzędnego, zostanie zgłoszony żaden błąd lub wyjątek i utworzono wartość właściwości globalnej, która nie będzie wykorzystywana przez żaden `DockPanel`element .

Dołączone właściwości w kodzie

Dołączone właściwości w WPF nie mają typowych metod CLR `get` i `set` otoki, ponieważ właściwości mogą być ustawione spoza przestrzeni nazw CLR. Aby umożliwić procesorowi XAML ustawienie tych wartości podczas analizowania kodu XAML, klasa definiująca dołączoną właściwość musi implementować dedykowane metody dostępu w postaci `Get<property name>` i `Set<property name>`.

Możesz również użyć dedykowanych metod dostępu, aby pobrać i ustawić dołączoną właściwość w kodzie, jak pokazano w poniższym przykładzie. W tym przykładzie

`myTextBox` jest wystąpieniem TextBox klasy.

```
C#

DockPanel myDockPanel = new();
TextBox myTextBox = new();
myTextBox.Text = "Enter text";

// Add child element to the DockPanel.
myDockPanel.Children.Add(myTextBox);

// Set the attached property value.
DockPanel.SetDock(myTextBox, Dock.Top);
```

Jeśli nie dodasz `myTextBox` jako element podrzędny elementu , wywołanie `SetDock` nie zgłosi wyjątku `myDockPanel` ani nie będzie miało żadnego wpływu. DockPanel.Dock Tylko wartość ustawiona na element podrzędny `DockPanel` elementu może mieć wpływ na renderowanie, a renderowanie będzie takie samo, czy ustawiono wartość przed dodaniem elementu podrzędnego do `DockPanel` elementu .

Z perspektywy kodu dołączona właściwość jest jak pole zaplecza, które ma metody dostępu zamiast metod dostępu do właściwości, i można je ustawić na dowolnym obiekcie bez uprzedniego zdefiniowania na tych obiektach.

Dołączone metadane właściwości

Metadane dołączonej właściwości zazwyczaj nie różnią się od właściwości zależności. Podczas rejestrowania dołączonej właściwości użyj polecenia FrameworkPropertyMetadata , aby określić cechy właściwości, takie jak to, czy właściwość ma wpływ na renderowanie lub pomiar. Po określeniu wartości domyślnej przez zastąpienie dołączonych metadanych właściwości ta wartość staje się wartością domyślną dla niejawnej dołączonej właściwości w wystąpieniach klasy zastępowania. Jeśli dołączona wartość właściwości nie jest ustawiona w inny sposób, wartość domyślna jest zgłaszana, gdy właściwość jest odpytywane przy użyciu `Get<property name>` metody dostępu z wystąpieniem klasy, w którym określono metadane.

Aby włączyć dziedziczenie wartości właściwości dla właściwości, użyj dołączonych właściwości zamiast nieprzyłączonych właściwości zależności. Aby uzyskać więcej informacji, zobacz Dziedziczenie wartości właściwości.

Niestandardowe dołączone właściwości

Kiedy utworzyć dołączoną właściwość

Tworzenie dołączonej właściwości jest przydatne, gdy:

- Potrzebny jest mechanizm ustawiania właściwości dostępny dla klas innych niż klasa definiująca. Typowy scenariusz dotyczy układu interfejsu użytkownika, na przykład DockPanel.Dock, Panel.ZIndexi Canvas.Top są przykładami istniejących właściwości układu. W scenariuszu układu elementy podrzędne elementu sterującego układem są w stanie wyrazić wymagania układu do ich układu nadrzędnego i ustawić wartość dołączonej właściwości zdefiniowanej przez element nadrzędny.

- Jedna z klas reprezentuje usługę i chcesz, aby inne klasy integrować usługę bardziej przezroczystie.

- Potrzebujesz obsługi projektanta WPF programu Visual Studio, takiego jak możliwość edytowania właściwości w oknie **Właściwości** . Aby uzyskać więcej informacji, zobacz Omówienie tworzenia kontrolek.

- Chcesz użyć dziedziczenia wartości właściwości.

Jak utworzyć dołączoną właściwość

Jeśli klasa definiuje dołączoną właściwość wyłącznie do użytku przez inne typy, klasa nie musi pochodzić z DependencyObjectklasy . W przeciwnym razie postępuj zgodnie z modelem WPF, aby mieć dołączoną właściwość również być właściwością zależności, wyprowadzając klasę z `DependencyObject`klasy .

Zdefiniuj dołączoną właściwość jako zależność w klasie definiującej, deklarując `public static readonly` pole typu DependencyProperty. Następnie przypisz wartość zwracaną RegisterAttached metody do pola, które jest również nazywane *identyfikatorem właściwości zależności*. Postępuj zgodnie z konwencją nazewnictwa właściwości WPF, która odróżnia pola od właściwości, które reprezentują, nazywając pole `<property name>Property`identyfikatora . Ponadto podaj metody statyczne `Get<property name>` i `Set<property name>` metody dostępu, które umożliwiają systemowi właściwości uzyskiwanie dostępu do dołączonej właściwości.

W poniższym przykładzie pokazano, jak zarejestrować właściwość zależności przy użyciu RegisterAttached metody i jak zdefiniować metody dostępu. W przykładzie nazwa dołączonej właściwości to `HasFish`, więc pole identyfikatora ma nazwę , a metody dostępu mają nazwę `HasFishProperty` `GetHasFish` i `SetHasFish`.

C#

```
public class Aquarium : UIElement
{
    // Register an attached dependency property with the specified
    // property name, property type, owner type, and property metadata.
    public static readonly DependencyProperty HasFishProperty =
        DependencyProperty.RegisterAttached(
      "HasFish",
      typeof(bool),
      typeof(Aquarium),
      new FrameworkPropertyMetadata(defaultValue: false,
          flags: FrameworkPropertyMetadataOptions.AffectsRender)
    );

    // Declare a get accessor method.
    public static bool GetHasFish(UIElement target) =>
        (bool)target.GetValue(HasFishProperty);

    // Declare a set accessor method.
    public static void SetHasFish(UIElement target, bool value) =>
        target.SetValue(HasFishProperty, value);
}
```

Uzyskiwanie dostępu

Sygnatura `get` metody dostępu to `public static object Get<property name>(DependencyObject target)`, gdzie:

- `target` DependencyObject to właściwość, z której jest odczytywana dołączona właściwość. Typ `target` może być bardziej szczegółowy niż `DependencyObject`. Na przykład DockPanel.GetDock metoda dostępu wpisze `target` wartość jako, UIElement ponieważ dołączona właściwość ma zostać ustawiona na `UIElement` wystąpieniach. `UiElement` pośrednio pochodzi z `DependencyObject` elementu .
- Typ zwracany może być bardziej szczegółowy niż `object`. Na przykład GetDock metoda wpisze zwróconą wartość jako Dock , ponieważ wartość zwracana powinna być wyliczeniem `Dock` .

> ⓘ **Uwaga**
>
> Akcesorium `get` dla dołączonej właściwości jest wymagane do obsługi powiązań danych w narzędziach projektowych, takich jak Visual Studio lub Blend for Visual Studio.

Zestaw dostępu

Sygnatura `set` metody dostępu to `public static void Set<property name> (DependencyObject target, object value)`, gdzie:

- `target` jest elementem DependencyObject , na którym zapisano dołączoną właściwość. Typ `target` może być bardziej szczegółowy niż `DependencyObject`. Na przykład metoda wpisze `target` wartość jako, UIElementSetDock ponieważ dołączona właściwość ma zostać ustawiona na UIElement wystąpieniach. `UiElement` pośrednio pochodzi z `DependencyObject`elementu .
- Typ `value` może być bardziej szczegółowy niż `object`. Na przykład SetDock metoda wymaga Dock wartości. Moduł ładujący XAML musi być w stanie wygenerować `value` typ z ciągu znaczników, który reprezentuje dołączoną wartość właściwości. Dlatego musi istnieć konwersja typu, serializator wartości lub obsługa rozszerzenia znaczników dla używanego typu.

Dołączone atrybuty właściwości

WPF definiuje kilka atrybutów platformy .NET, które zawierają informacje o dołączonych właściwościach do procesów odbicia, a także dla konsumentów informacji o odbiciem i właściwościach, takich jak projektanci. Projektanci używają atrybutów platformy .NET zdefiniowanych w programie WPF, aby ograniczyć właściwości wyświetlane w oknie właściwości, aby uniknąć przeciążenia użytkowników z globalną listą wszystkich dołączonych właściwości. Możesz rozważyć zastosowanie tych atrybutów do własnych niestandardowych dołączonych właściwości. Przeznaczenie i składnia atrybutów platformy .NET są opisane na tych stronach referencyjnych:

- AttachedPropertyBrowsableAttribute
- AttachedPropertyBrowsableForChildrenAttribute
- AttachedPropertyBrowsableForTypeAttribute
- AttachedPropertyBrowsableWhenAttributePresentAttribute

Więcej tutaj

- Aby uzyskać więcej informacji na temat tworzenia dołączonej właściwości, zobacz Rejestrowanie dołączonej właściwości.
- Aby uzyskać bardziej zaawansowane scenariusze użycia właściwości zależności i dołączonych właściwości, zobacz Niestandardowe właściwości zależności.
- Właściwość można zarejestrować zarówno jako dołączoną właściwość, jak i właściwość zależności, i dołączyć konwencjonalne otoki właściwości. W ten sposób właściwość można ustawić na elemecie za pomocą otoki właściwości, a także dla

dowolnego innego elementu przy użyciu składni właściwości dołączonych XAML. Aby zapoznać się z przykładem, zobacz FrameworkElement.FlowDirection.

Zobacz też

- DependencyProperty
- Omówienie właściwości zależności
- Niestandardowe właściwości zależności
- XAML w WPF
- Instrukcje: rejestrowanie dołączonej właściwości

Właściwości zależności typu kolekcji (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 5 min

Ten artykuł zawiera wskazówki i sugerowane wzorce implementacji właściwości zależności, która jest typem kolekcji.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

Implementowanie właściwości zależności typu kolekcji

Ogólnie rzecz biorąc, wzorzec implementacji właściwości zależności jest otoką właściwości CLR wspieraną przez DependencyProperty identyfikator zamiast pola lub innej konstrukcji. Ten sam wzorzec można wykonać podczas implementowania właściwości zależności typu kolekcji. Wzorzec jest bardziej złożony, jeśli typ elementu kolekcji jest klasą DependencyObject pochodną Freezable lub .

Inicjowanie kolekcji

Podczas tworzenia właściwości zależności zazwyczaj określasz wartość domyślną za pomocą metadanych właściwości zależności zamiast określania wartości właściwości początkowej. Jeśli jednak wartość właściwości jest typem referencyjnym, wartość domyślna powinna być ustawiona w konstruktorze klasy, która rejestruje właściwość zależności. Metadane właściwości zależności nie powinny zawierać domyślnej wartości typu odwołania, ponieważ ta wartość zostanie przypisana do wszystkich wystąpień klasy, tworząc pojedynczą klasę.

Poniższy przykład deklaruje klasę `Aquarium` zawierającą kolekcję FrameworkElement elementów w ogólnym List<T>obiekcie . Domyślna wartość kolekcji nie jest uwzględniana w przekazanej PropertyMetadataRegisterReadOnly(String, Type, Type, PropertyMetadata) metodzie, a zamiast tego konstruktor klasy służy do ustawiania domyślnej wartości kolekcji na nową ogólną `List`wartość .

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, owner type, and property metadata.
    private static readonly DependencyPropertyKey
s_aquariumContentsPropertyKey =
        DependencyProperty.RegisterReadOnly(
          name: "AquariumContents",
          propertyType: typeof(List<FrameworkElement>),
          ownerType: typeof(Aquarium),
          typeMetadata: new FrameworkPropertyMetadata()
          //typeMetadata: new FrameworkPropertyMetadata(new
List<FrameworkElement>())
        );

    // Set the default collection value in a class constructor.
    public Aquarium() => SetValue(s_aquariumContentsPropertyKey, new
List<FrameworkElement>());

    // Declare a public get accessor.
    public List<FrameworkElement> AquariumContents =>

(List<FrameworkElement>)GetValue(s_aquariumContentsPropertyKey.DependencyPro
perty);
}

public class Fish : FrameworkElement { }
```

Poniższy kod testowy tworzy wystąpienie dwóch oddzielnych `Aquarium` wystąpień i dodaje inny `Fish` element do każdej kolekcji. Jeśli uruchomisz kod, zobaczysz, że każde `Aquarium` wystąpienie ma jeden element kolekcji zgodnie z oczekiwaniami.

C#

```
private void InitializeAquariums(object sender, RoutedEventArgs e)
{
    Aquarium aquarium1 = new();
    Aquarium aquarium2 = new();
    aquarium1.AquariumContents.Add(new Fish());
    aquarium2.AquariumContents.Add(new Fish());
    MessageBox.Show(
        $"aquarium1 contains {aquarium1.AquariumContents.Count} fish\r\n" +
```

```
        $"aquarium2 contains {aquarium2.AquariumContents.Count} fish");
}
```

Jeśli jednak oznaczysz jako komentarz konstruktor klasy i przekażesz domyślną wartość kolekcji jako PropertyMetadata metodę RegisterReadOnly(String, Type, Type, PropertyMetadata) , zobaczysz, że każde `Aquarium` wystąpienie otrzymuje dwa elementy kolekcji! Dzieje się tak, ponieważ oba `Fish` wystąpienia są dodawane do tej samej listy, która jest współużytkowana przez wszystkie wystąpienia klasy Aquarium. Dlatego, gdy intencja jest przeznaczona dla każdego wystąpienia obiektu, aby miało własną listę, wartość domyślna powinna być ustawiona w konstruktorze klasy.

Inicjowanie kolekcji odczytu i zapisu

Poniższy przykład deklaruje właściwość zależności typu kolekcji odczytu i zapisu w `Aquarium` klasie przy użyciu metod Register(String, Type, Type) sygnatur innych niż klucz i SetValue(DependencyProperty, Object).

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, and owner type. Store the dependency property
    // identifier as a public static readonly member of the class.
    public static readonly DependencyProperty AquariumContentsProperty =
        DependencyProperty.Register(
          name: "AquariumContents",
          propertyType: typeof(List<FrameworkElement>),
          ownerType: typeof(Aquarium)
        );

    // Set the default collection value in a class constructor.
    public Aquarium() => SetValue(AquariumContentsProperty, new
List<FrameworkElement>());

    // Declare public get and set accessors.
    public List<FrameworkElement> AquariumContents
    {
        get => (List<FrameworkElement>)GetValue(AquariumContentsProperty);
        set => SetValue(AquariumContentsProperty, value);
    }
}
```

Właściwości zależności FreezableCollection

Właściwość zależności typu kolekcji nie zgłasza automatycznie zmian w jej podwłaściwości. W związku z tym w przypadku powiązania z kolekcją powiązanie może nie zgłaszać zmian, unieważniając niektóre scenariusze powiązań danych. Jeśli jednak używasz FreezableCollection<T> dla typu właściwości zależności, zmiany właściwości elementów kolekcji są prawidłowo zgłaszane, a powiązanie działa zgodnie z oczekiwaniami.

Aby włączyć powiązanie podwłaściwości w kolekcji obiektów zależności, użyj typu kolekcji , z ograniczeniem typu `FreezableCollection`typu dowolnej DependencyObject klasy pochodnej.

Poniższy przykład deklaruje klasę `Aquarium` zawierającą `FreezableCollection` ograniczenie FrameworkElementtypu . Domyślna wartość kolekcji nie jest uwzględniana RegisterReadOnly(String, Type, Type, PropertyMetadata) w przekazanej PropertyMetadata metodzie, a zamiast tego konstruktor klasy służy do ustawiania domyślnej wartości kolekcji na nową `FreezableCollection`wartość .

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, and owner type.
    private static readonly DependencyPropertyKey
s_aquariumContentsPropertyKey =
        DependencyProperty.RegisterReadOnly(
          name: "AquariumContents",
          propertyType: typeof(FreezableCollection<FrameworkElement>),
          ownerType: typeof(Aquarium),
          typeMetadata: new FrameworkPropertyMetadata()
        );

    // Store the dependency property identifier as a static member of the
class.
    public static readonly DependencyProperty AquariumContentsProperty =
        s_aquariumContentsPropertyKey.DependencyProperty;

    // Set the default collection value in a class constructor.
    public Aquarium() => SetValue(s_aquariumContentsPropertyKey, new
FreezableCollection<FrameworkElement>());

    // Declare a public get accessor.
    public FreezableCollection<FrameworkElement> AquariumContents =>

(FreezableCollection<FrameworkElement>)GetValue(AquariumContentsProperty);
}
```

Zobacz też

- FreezableCollection<T>
- Klasy XAML i niestandardowe dla WPF
- Powiązanie danych — omówienie
- Przegląd Właściwości zależności
- Właściwości zależności niestandardowych
- Metadane właściwości zależności

Niestandardowe właściwości zależności (WPF .NET)

Artykuł • 11.02.2023 • Czas czytania: 12 min

Windows Presentation Foundation (WPF) deweloperzy aplikacji i autorzy składników mogą tworzyć niestandardowe właściwości zależności w celu rozszerzenia funkcjonalności ich właściwości. W przeciwieństwie do właściwości środowiska uruchomieniowego języka wspólnego (CLR) właściwość zależności dodaje obsługę stylów, powiązań danych, dziedziczenia, animacji i wartości domyślnych. Background, Widthi Text to przykłady istniejących właściwości zależności w klasach WPF. W tym artykule opisano sposób implementowania niestandardowych właściwości zależności oraz przedstawiono opcje poprawy wydajności, użyteczności i wszechstronności.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat właściwości zależności i zapoznasz się z omówieniem właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem XAML (Extensible Application Markup Language) i wiedzieć, jak pisać aplikacje WPF.

Identyfikator właściwości zależności

Właściwości zależności to właściwości, które są zarejestrowane w systemie właściwości WPF za pośrednictwem Register lub RegisterReadOnly wywołań. Metoda `Register` zwraca wystąpienie, które zawiera zarejestrowaną DependencyProperty nazwę i cechy właściwości zależności. Wystąpienie zostanie przypisane `DependencyProperty` do statycznego pola tylko do odczytu, *nazywanego identyfikatorem właściwości zależności*, które zgodnie z konwencją nosi nazwę `<property name>Property`. Na przykład pole identyfikatora Background właściwości to zawsze BackgroundProperty.

Identyfikator właściwości zależności jest używany jako pole zapasowe do pobierania lub ustawiania wartości właściwości zamiast standardowego wzorca tworzenia kopii zapasowej właściwości z polem prywatnym. System właściwości nie tylko używa

identyfikatora, procesory XAML mogą go używać, a kod (i prawdopodobnie kod zewnętrzny) może uzyskiwać dostęp do właściwości zależności za pośrednictwem ich identyfikatorów.

Właściwości zależności można stosować tylko do klas, które pochodzą z DependencyObject typów. Większość klas WPF obsługuje właściwości zależności, ponieważ `DependencyObject` znajduje się w pobliżu katalogu głównego hierarchii klas WPF. Aby uzyskać więcej informacji na temat właściwości zależności oraz terminologii i konwencji używanych do ich opisywania, zobacz Omówienie właściwości zależności.

Otoki właściwości zależności

Właściwości zależności WPF, które nie są dołączone, są uwidaczniane przez otokę CLR, która implementuje `get` i `set` metody dostępu. Korzystając z otoki właściwości, odbiorcy właściwości zależności mogą pobierać lub ustawiać wartości właściwości zależności, tak samo jak każda inna właściwość CLR. Metody `get` i `set` współdziałają z podstawowym systemem właściwości za pośrednictwem metody DependencyObject.GetValue i DependencyObject.SetValue wywołuje metodę , przekazując identyfikator właściwości zależności jako parametr. Konsumenci właściwości zależności zwykle nie są wywoływani `GetValue` ani `SetValue` bezpośrednio, ale jeśli implementujesz niestandardową właściwość zależności, użyjesz tych metod w otoce.

Kiedy należy zaimplementować właściwość zależności

Podczas implementowania właściwości w klasie, która pochodzi z DependencyObjectklasy , należy ustawić ją jako właściwość zależności, tworząc kopię zapasową właściwości za pomocą identyfikatora DependencyProperty . Niezależnie od tego, czy korzystne jest utworzenie właściwości zależności, zależy od danego scenariusza. Mimo że tworzenie kopii zapasowej właściwości z polem prywatnym jest odpowiednie dla niektórych scenariuszy, rozważ zaimplementowanie właściwości zależności, jeśli chcesz, aby właściwość obsługiwała co najmniej jedną z następujących funkcji WPF:

- Właściwości, które można ustawić w stylu. Aby uzyskać więcej informacji, zobacz Style i szablony.
- Właściwości obsługujące powiązanie danych. Aby uzyskać więcej informacji na temat właściwości zależności powiązania danych, zobacz Wiązanie właściwości dwóch kontrolek.

- Właściwości, które można ustawić za pomocą dynamicznych odwołań do zasobów. Aby uzyskać więcej informacji, zobacz Zasoby XAML.
- Właściwości, które automatycznie dziedziczą ich wartość z elementu nadrzędnego w drzewie elementów. W tym celu należy zarejestrować się przy użyciu polecenia RegisterAttached, nawet jeśli utworzysz również otokę właściwości na potrzeby dostępu do środowiska CLR. Aby uzyskać więcej informacji, zobacz Dziedziczenie wartości właściwości.
- Właściwości, które są animatowalne. Aby uzyskać więcej informacji, zobacz Omówienie animacji.
- Powiadomienie przez system właściwości WPF po zmianie wartości właściwości. Zmiany mogą być spowodowane akcjami systemu właściwości, środowiska, użytkownika lub stylów. Właściwość może określić metodę wywołania zwrotnego w metadanych właściwości, które będą wywoływane za każdym razem, gdy system właściwości określi, że wartość właściwości została zmieniona. Powiązana koncepcja to przymus wartości właściwości. Aby uzyskać więcej informacji, zobacz Wywołania zwrotne właściwości zależności i walidacja.
- Dostęp do metadanych właściwości zależności odczytywanych przez procesy WPF. Na przykład można użyć metadanych właściwości do:
  - Określ, czy zmieniona wartość właściwości zależności powinna sprawić, że system układu ponownie komponuje wizualizacje dla elementu.
  - Ustaw wartość domyślną właściwości zależności, przesłaniając metadane dla klas pochodnych.
- Obsługa projektanta WPF programu Visual Studio, na przykład edytowanie właściwości kontrolki niestandardowej w oknie **Właściwości** . Aby uzyskać więcej informacji, zobacz Omówienie tworzenia kontrolek.

W niektórych scenariuszach zastąpienie metadanych istniejącej właściwości zależności jest lepszą opcją niż zaimplementowanie nowej właściwości zależności. To, czy zastąpienie metadanych jest praktyczne, zależy od danego scenariusza i tego, jak bardzo ten scenariusz przypomina implementację istniejących właściwości i klas zależności WPF. Aby uzyskać więcej informacji na temat zastępowania metadanych na istniejących właściwościach zależności, zobacz Metadane właściwości zależności.

Lista kontrolna tworzenia właściwości zależności

Wykonaj następujące kroki, aby utworzyć właściwość zależności. Niektóre kroki można połączyć i zaimplementować w jednym wierszu kodu.

1. (Opcjonalnie) Tworzenie metadanych właściwości zależności.

2. Zarejestruj właściwość zależności w systemie właściwości, określając nazwę właściwości, typ właściciela, typ wartości właściwości i opcjonalnie metadane właściwości.

3. Zdefiniuj DependencyProperty identyfikator jako `public static readonly` pole typu właściciela. Nazwa pola identyfikatora to nazwa właściwości z dołączonym sufiksem `Property` .

4. Zdefiniuj właściwość otoki CLR o takiej samej nazwie jak nazwa właściwości zależności. W otoce CLR zaimplementuj `get` metody i `set` metody dostępu łączące się z właściwością zależności, która wspiera otokę.

Rejestrowanie właściwości

Aby właściwość była właściwością zależności, należy zarejestrować ją w systemie właściwości. Aby zarejestrować swoją właściwość, wywołaj metodę Register z wewnątrz treści klasy, ale poza definicjami składowymi. Metoda `Register` zwraca unikatowy identyfikator właściwości zależności, który będzie używany podczas wywoływania interfejsu API systemu właściwości. Powodem, dla którego `Register` wywołanie jest wykonywane poza definicjami elementów członkowskich, jest to, że przypisujesz wartość zwracaną `public static readonly` do pola typu DependencyProperty. To pole, które utworzysz w klasie, jest identyfikatorem właściwości zależności. W poniższym przykładzie pierwszy argument `Register` nazwy właściwości `AquariumGraphic` zależności .

C#

```
// Register a dependency property with the specified property name,
// property type, owner type, and property metadata. Store the dependency
// property identifier as a public static readonly member of the class.
public static readonly DependencyProperty AquariumGraphicProperty =
    DependencyProperty.Register(
      name: "AquariumGraphic",
      propertyType: typeof(Uri),
      ownerType: typeof(Aquarium),
      typeMetadata: new FrameworkPropertyMetadata(
          defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
          flags: FrameworkPropertyMetadataOptions.AffectsRender,
          propertyChangedCallback: new
PropertyChangedCallback(OnUriChanged))
    );
```

> **Uwaga**
>
> Definiowanie właściwości zależności w treści klasy jest typową implementacją, ale istnieje również możliwość zdefiniowania właściwości zależności w konstruktorze statycznym klasy. Takie podejście może mieć sens, jeśli potrzebujesz więcej niż jednego wiersza kodu do zainicjowania właściwości zależności.

Nazewnictwo właściwości zależności

Ustanowiona konwencja nazewnictwa właściwości zależności jest obowiązkowa dla normalnego zachowania systemu właściwości. Nazwa tworzonego pola identyfikatora musi być zarejestrowaną nazwą właściwości z sufiksem `Property`.

Nazwa właściwości zależności musi być unikatowa w klasie rejestrującej. Właściwości zależności dziedziczone za pośrednictwem typu podstawowego zostały już zarejestrowane i nie mogą być zarejestrowane przez typ pochodny. Można jednak użyć właściwości zależności, która została zarejestrowana przez inny typ, nawet typ, z którego klasa nie dziedziczy, dodając klasę jako właściciela właściwości zależności. Aby uzyskać więcej informacji na temat dodawania klasy jako właściciela, zobacz Metadane właściwości zależności.

Implementowanie otoki właściwości

Zgodnie z konwencją nazwa właściwości otoki musi być taka sama jak pierwszy parametr Register wywołania, który jest nazwą właściwości zależności. Implementacja otoki wywoła GetValue metodę `get` dostępu i SetValue w metodzie `set` dostępu (dla właściwości odczytu i zapisu). Poniższy przykład przedstawia otokę — po wywołaniu rejestracji i deklaracji pola identyfikatora. Wszystkie właściwości zależności publicznych w klasach WPF używają podobnego modelu otoki.

C#

```
// Register a dependency property with the specified property name,
// property type, owner type, and property metadata. Store the dependency
// property identifier as a public static readonly member of the class.
public static readonly DependencyProperty AquariumGraphicProperty =
    DependencyProperty.Register(
      name: "AquariumGraphic",
      propertyType: typeof(Uri),
      ownerType: typeof(Aquarium),
      typeMetadata: new FrameworkPropertyMetadata(
```

```
            defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
            flags: FrameworkPropertyMetadataOptions.AffectsRender,
            propertyChangedCallback: new
PropertyChangedCallback(OnUriChanged))
    );

// Declare a read-write property wrapper.
public Uri AquariumGraphic
{
    get => (Uri)GetValue(AquariumGraphicProperty);
    set => SetValue(AquariumGraphicProperty, value);
}
```

Z wyjątkiem rzadkich przypadków implementacja otoki powinna zawierać GetValue tylko kod i SetValue . Ze względu na to zobacz Implikacje dla niestandardowych właściwości zależności.

Jeśli twoja właściwość nie jest zgodna z ustalonymi konwencjami nazewnictwa, mogą wystąpić następujące problemy:

- Niektóre aspekty stylów i szablonów nie będą działać.
- Większość narzędzi i projektantów opiera się na konwencjach nazewnictwa, aby prawidłowo serializować kod XAML i zapewnić pomoc w środowisku projektanta na poziomie poszczególnych właściwości.
- Bieżąca implementacja modułu ładującego WPF XAML całkowicie pomija otoki i opiera się na konwencji nazewnictwa do przetwarzania wartości atrybutów. Aby uzyskać więcej informacji, zobacz Właściwości ładowania i zależności XAML.

Metadane zależności właściwości

Podczas rejestrowania właściwości zależności system właściwości tworzy obiekt metadanych do przechowywania właściwości. Przeciążenia Register metody umożliwiają określenie metadanych właściwości podczas rejestracji, na przykład Register(String, Type, Type, PropertyMetadata). Typowym zastosowaniem metadanych właściwości jest zastosowanie niestandardowej wartości domyślnej dla nowych wystąpień korzystających z właściwości zależności. Jeśli nie podasz metadanych właściwości, system właściwości przypisze wartości domyślne do wielu właściwości zależności.

Jeśli tworzysz właściwość zależności na klasie pochodzącej z FrameworkElementklasy , możesz użyć bardziej wyspecjalizowanej klasy FrameworkPropertyMetadata metadanych, a nie jej klasy PropertyMetadatabazowej . Kilka FrameworkPropertyMetadata podpisów konstruktora umożliwia określenie różnych

kombinacji właściwości metadanych. Jeśli chcesz po prostu określić wartość domyślną, użyj FrameworkPropertyMetadata(Object) i przekaż wartość domyślną do parametru `Object` . Upewnij się, że typ wartości jest zgodny z `propertyType` określonym w wywołaniu `Register` .

Niektóre FrameworkPropertyMetadata przeciążenia umożliwiają określenie flag opcji metadanych dla właściwości. System właściwości konwertuje te flagi na właściwości dyskretne, a wartości flagi są używane przez procesy WPF, takie jak aparat układu.

Ustawianie flag metadanych

Podczas ustawiania flag metadanych należy wziąć pod uwagę następujące kwestie:

- Jeśli wartość właściwości (lub jej zmiany) wpływa na sposób renderowania elementu interfejsu użytkownika przez system układu, ustaw co najmniej jedną z następujących flag:

  - AffectsMeasure, co wskazuje, że zmiana wartości właściwości wymaga zmiany renderowania interfejsu użytkownika, a w szczególności miejsca zajmowanego przez obiekt nadrzędny. Na przykład ustaw tę flagę `Width` metadanych dla właściwości.

  - AffectsArrange, co wskazuje, że zmiana wartości właściwości wymaga zmiany renderowania interfejsu użytkownika, w szczególności pozycji obiektu w jego obiekcie nadrzędnym. Zazwyczaj obiekt nie zmienia również rozmiaru. Na przykład ustaw tę flagę `Alignment` metadanych dla właściwości.

  - AffectsRender, co wskazuje, że nastąpiła zmiana, która nie ma wpływu na układ i miarę, ale nadal wymaga innego renderowania. Na przykład ustaw tę flagę `Background` dla właściwości lub inną właściwość, która ma wpływ na kolor elementu.

  Te flagi można również używać jako danych wejściowych do przesłonięć implementacji wywołań zwrotnych systemu właściwości (lub układu). Możesz na przykład użyć OnPropertyChanged wywołania zwrotnego, aby wywołać wywołanie InvalidateArrange , gdy właściwość wystąpienia zgłasza zmianę wartości i ma AffectsArrange ustawioną wartość w metadanych.

- Niektóre właściwości wpływają na charakterystykę renderowania elementu nadrzędnego na inne sposoby. Na przykład zmiany MinOrphanLines właściwości mogą zmienić ogólny renderowanie dokumentu przepływu. Użyj AffectsParentArrange polecenia lub AffectsParentMeasure zasygnalizuj akcje nadrzędne we własnych właściwościach.

- Domyślnie właściwości zależności obsługują powiązanie danych. Można jednak użyć IsDataBindingAllowed polecenia , aby wyłączyć powiązanie danych, gdy nie ma dla niego realistycznego scenariusza lub gdy wydajność powiązania danych jest problematyczna, na przykład na dużych obiektach.

- Mimo że domyślny tryb powiązania danych dla właściwości zależności to OneWay, można zmienić tryb powiązania określonego powiązania na TwoWay. Aby uzyskać więcej informacji, zobacz Kierunek powiązania. Jako autor właściwości zależności można nawet utworzyć powiązanie dwukierunkowe w trybie domyślnym. Przykładem istniejącej właściwości zależności, która używa powiązania danych dwukierunkowego, jest MenuItem.IsSubmenuOpen, która ma stan oparty na innych właściwościach i wywołaniach metody. W tym `IsSubmenuOpen` scenariuszu jest to, że jego logika ustawienia i komposiowanie MenuItemelementu , współdziałają z domyślnym stylem motywu. TextBox.Text to inna właściwość zależności WPF, która domyślnie używa powiązania dwukierunkowego.

- Dziedziczenie właściwości dla właściwości zależności można włączyć, ustawiając flagę Inherits . Dziedziczenie właściwości jest przydatne w scenariuszach, w których elementy nadrzędne i podrzędne mają wspólną właściwość i ma sens, aby element podrzędny dziedziczył wartość nadrzędną dla właściwości wspólnej. Przykładem właściwości dziedziczonej jest DataContext, która obsługuje operacje powiązania korzystające ze scenariusza szczegółów wzorca na potrzeby prezentacji danych. Dziedziczenie wartości właściwości umożliwia określenie kontekstu danych na stronie lub katalogu głównym aplikacji, co pozwala zapisywać konieczność określenia go dla powiązań elementów podrzędnych. Mimo że dziedziczona wartość właściwości zastępuje wartość domyślną, wartości właściwości można ustawić lokalnie na dowolnym elemenie podrzędnym. Użyj dziedziczenia wartości właściwości oszczędnie, ponieważ ma koszt wydajności. Aby uzyskać więcej informacji, zobacz Dziedziczenie wartości właściwości.

- Ustaw flagę Journal , aby wskazać, że właściwość zależności powinna zostać wykryta lub użyta przez usługi dziennika nawigacji. Na przykład właściwość ustawia flagę `Journal`, SelectedIndex aby zalecać, aby aplikacje przechowywały wybraną historię dziennika elementów.

Właściwości zależności tylko do odczytu

Można zdefiniować właściwość zależności, która jest tylko do odczytu. Typowy scenariusz to właściwość zależności, która przechowuje stan wewnętrzny. Na przykład jest tylko do odczytu, IsMouseOver ponieważ jego stan powinien być określany tylko

przez dane wejściowe myszy. Aby uzyskać więcej informacji, zobacz Właściwości zależności tylko do odczytu.

Właściwości zależności typu kolekcji

Właściwości zależności typu kolekcji mają dodatkowe problemy z implementacją, takie jak ustawienie wartości domyślnej dla typów referencyjnych i obsługi powiązań danych dla elementów kolekcji. Aby uzyskać więcej informacji, zobacz Właściwości zależności typu kolekcji.

Zabezpieczenia właściwości zależności

Zazwyczaj właściwości zależności będą deklarowane jako właściwości publiczne i DependencyProperty pola identyfikatora jako `public static readonly` pola. Jeśli określisz bardziej restrykcyjny poziom dostępu, taki jak `protected`, właściwość zależności będzie nadal dostępna za pośrednictwem jego identyfikatora w połączeniu z interfejsami API systemu właściwości. Nawet pole identyfikatora chronionego jest potencjalnie dostępne za pośrednictwem interfejsów API raportowania metadanych lub określania wartości WPF, takich jak LocalValueEnumerator. Aby uzyskać więcej informacji, zobacz Zabezpieczenia właściwości zależności.

W przypadku właściwości zależności tylko do odczytu wartość zwracana z RegisterReadOnly klasy to DependencyPropertyKey, a zazwyczaj nie `DependencyPropertyKey` `public` będzie elementem członkowskim klasy. Ponieważ system właściwości WPF nie propaguje `DependencyPropertyKey` poza kodem, właściwość zależności tylko do odczytu ma lepsze `set` zabezpieczenia niż właściwość zależności read-write.

Właściwości zależności i konstruktory klas

Istnieje ogólna zasada programowania kodu zarządzanego, często wymuszana przez narzędzia do analizy kodu, że konstruktory klas nie powinny wywoływać metod wirtualnych. Wynika to z faktu, że konstruktory podstawowe mogą być wywoływane podczas inicjowania konstruktora klasy pochodnej, a metoda wirtualna wywoływana przez konstruktor podstawowy może działać przed ukończeniem inicjowania klasy pochodnej. Gdy pochodzisz z klasy, która już pochodzi z DependencyObjectklasy , sam system właściwości wywołuje i uwidacznia metody wirtualne wewnętrznie. Te metody wirtualne są częścią usług systemu właściwości WPF. Zastąpienie metod umożliwia klasom pochodnym uczestnictwo w określaniu wartości. Aby uniknąć potencjalnych

problemów z inicjowaniem środowiska uruchomieniowego, nie należy ustawiać wartości właściwości zależności w konstruktorach klas, chyba że jest używany określony wzorzec konstruktora. Aby uzyskać więcej informacji, zobacz Bezpieczne wzorce konstruktorów dla obiektów DependencyObjects.

Zobacz też

- Omówienie właściwości zależności
- Metadane zależności właściwości
- Omówienie tworzenia kontrolek
- Właściwości zależności typu kolekcji
- Zabezpieczenia właściwości zależności
- Ładowanie kodu XAML i właściwości zależności
- Bezpieczne wzorce konstruktora dla obiektów DependencyObject

Metadane właściwości zależności (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 7 min

System właściwości Windows Presentation Foundation (WPF) zawiera system raportowania metadanych właściwości zależności. Informacje dostępne w systemie raportowania metadanych przekraczają to, co jest dostępne za pośrednictwem odbicia lub ogólnych cech środowiska uruchomieniowego języka wspólnego (CLR). Podczas rejestrowania właściwości zależności możesz utworzyć i przypisać do niej metadane. Jeśli pochodzisz z klasy, która definiuje właściwość zależności, możesz zastąpić metadane dla dziedziczonej właściwości zależności. Jeśli dodasz klasę jako właściciela właściwości zależności, możesz zastąpić metadane odziedziczonej właściwości zależności.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Wymagania wstępne

W tym artykule przyjęto założenie, że masz podstawową wiedzę na temat właściwości zależności i zapoznasz się z omówieniem właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem XAML (Extensible Application Markup Language) i wiedzieć, jak pisać aplikacje WPF.

Jak są używane metadane

Możesz wykonać zapytanie dotyczące metadanych właściwości zależności, aby zbadać charakterystykę właściwości zależności. Gdy system właściwości przetwarza właściwość zależności, uzyskuje dostęp do metadanych. Obiekt metadanych dla właściwości zależności zawiera następujące typy informacji:

- Wartość domyślna właściwości zależności, która jest ustawiana przez system właściwości, gdy nie ma zastosowania żadna inna wartość, na przykład lokalna, styl lub wartość dziedziczenia. Aby uzyskać więcej informacji na temat pierwszeństwa wartości wartości właściwości zależności w czasie wykonywania, zobacz Pierwszeństwo wartości właściwości zależności.

- Odwołania do wywołań zwrotnych wartości przymusu i wywołań zwrotnych właściwości w typie właściciela. Można uzyskać odwołania tylko do wywołań zwrotnych, które mają `public` modyfikator dostępu lub znajdują się w dozwolonym zakresie dostępu. Aby uzyskać więcej informacji na temat wywołań zwrotnych właściwości zależności, zobacz Wywołania zwrotne właściwości zależności i walidacja.

- Właściwości zależności na poziomie struktury WPF (jeśli właściwość zależności jest właściwością struktury WPF). Procesy WPF, takie jak aparat układu struktury i logika dziedziczenia właściwości, wysyłają zapytania do metadanych na poziomie struktury WPF. Aby uzyskać więcej informacji, zobacz Metadane właściwości struktury.

Interfejsy API metadanych

Klasa PropertyMetadata przechowuje większość metadanych używanych przez system właściwości. Wystąpienia metadanych można tworzyć i przypisywać za pomocą:

- Typy rejestrujące właściwości zależności w systemie właściwości.

- Typy dziedziczone z klasy definiującej właściwość zależności.

- Typy, które dodają się jako właściciel właściwości zależności.

Jeśli typ rejestruje właściwość zależności bez określania metadanych, system właściwości przypisuje `PropertyMetadata` obiekt z wartościami domyślnymi dla tego typu do właściwości zależności.

Aby pobrać metadane dla właściwości zależności, wywołaj jedno z GetMetadata przeciążeń identyfikatora DependencyProperty . Metadane są zwracane jako `PropertyMetadata` obiekt.

Bardziej szczegółowe klasy metadanych, pochodzące z `PropertyMetadata`klasy , istnieją dla różnych obszarów architektury. Na przykład UIPropertyMetadata obsługuje raportowanie animacji i FrameworkPropertyMetadata obsługuje właściwości struktury WPF. Właściwości zależności można również zarejestrować w `PropertyMetadata` klasach pochodnych. Mimo że `GetMetadata` zwraca `PropertyMetadata` obiekt, jeśli ma to zastosowanie, można rzutować do typu pochodnego w celu zbadania właściwości specyficznych dla typu.

Cechy właściwości, które są widoczne przez `FrameworkPropertyMetadata` element, są czasami określane jako *flagi*. Podczas tworzenia `FrameworkPropertyMetadata` wystąpienia można przekazać wystąpienie typu FrameworkPropertyMetadataOptions wyliczenia do

konstruktora `FrameworkPropertyMetadata` . `FrameworkPropertyMetadataOptions` Umożliwia określenie flag metadanych w kombinacji bitowej. `FrameworkPropertyMetadataOptions` Używa `FrameworkPropertyMetadata`, aby zachować długość podpisu konstruktora rozsądnie. W przypadku rejestracji właściwości zależności flagi metadanych, na których ustawiono `FrameworkPropertyMetadataOptions` , są widoczne w ramach `FrameworkPropertyMetadata` właściwości, `Boolean` a nie bitowej kombinacji flag, aby bardziej intuicyjne było wykonywanie zapytań dotyczących właściwości metadanych.

Czy zastąpić lub utworzyć nowe metadane?

Gdy dziedziczysz właściwość zależności, możesz zmienić właściwości właściwości zależności, przesłaniając jej metadane. Jednak może nie zawsze być w stanie osiągnąć scenariusz właściwości zależności przez zastępowanie metadanych, a czasami konieczne jest zdefiniowanie niestandardowej właściwości zależności w klasie z nowymi metadanymi. Właściwości zależności niestandardowych mają takie same możliwości jak właściwości zależności zdefiniowane przez typy WPF. Aby uzyskać więcej informacji, zobacz Właściwości zależności niestandardowych.

Jedną z cech właściwości zależności, której nie można zastąpić, jest jej typ wartości. Jeśli dziedziczona właściwość zależności ma przybliżone zachowanie, ale scenariusz wymaga innego typu wartości, rozważ zaimplementowanie niestandardowej właściwości zależności. Możesz połączyć wartości właściwości za pomocą konwersji typów lub innego wdrożenia w klasie pochodnej.

Scenariusze zastępowania metadanych

Przykładowe scenariusze zastępowania istniejących metadanych właściwości zależności to:

- Zmiana wartości domyślnej, która jest typowym scenariuszem.
- Zmiana lub dodanie wywołań zwrotnych zmiany właściwości, co może być konieczne, jeśli dziedziczona właściwość zależności współdziała z innymi właściwościami zależności inaczej niż jej implementacja podstawowa. Jedną z cech modelu programowania, który obsługuje zarówno kod, jak i znaczniki, jest to, że wartości właściwości mogą być ustawiane w dowolnej kolejności. Ten czynnik może mieć wpływ na sposób implementowania wywołań zwrotnych zmiany właściwości. Aby uzyskać więcej informacji, zobacz Wywołania zwrotne właściwości zależności i walidacja.

- Zmiana opcji metadanych właściwości struktury WPF. Zazwyczaj opcje metadanych są ustawiane podczas rejestracji nowej właściwości zależności, ale można je ponownie określić w OverrideMetadata wywołaniach lub AddOwner . Aby uzyskać więcej informacji na temat zastępowania metadanych właściwości struktury, zobacz Określanie strukturyPropertyMetadata. Aby dowiedzieć się, jak ustawić opcje metadanych właściwości struktury podczas rejestrowania właściwości zależności, zobacz Właściwości zależności niestandardowych.

> ⓘ **Uwaga**
>
> Ponieważ wywołania zwrotne weryfikacji nie są częścią metadanych, nie można ich zmienić przez zastępowanie metadanych. Aby uzyskać więcej informacji, zobacz **Wywołania zwrotne wartości walidacji**.

Zastępowanie metadanych

Podczas implementowania nowej właściwości zależności można ustawić jej metadane przy użyciu przeciążeń Register metody . Jeśli klasa dziedziczy właściwość zależności, możesz zastąpić dziedziczone wartości metadanych przy użyciu OverrideMetadata metody . Na przykład można użyć `OverrideMetadata` polecenia , aby ustawić wartości specyficzne dla typu. Aby uzyskać więcej informacji i przykładów kodu, zobacz Zastępowanie metadanych dla właściwości zależności.

Przykładem właściwości zależności WPF jest Focusable. Klasa FrameworkElement rejestruje element `Focusable`. Klasa Control pochodzi z `FrameworkElement` klasy , dziedziczy `Focusable` właściwość zależności i zastępuje dziedziczone metadane właściwości. Zastąpienie zmienia domyślną wartość właściwości z `false` na `true`, ale zachowuje inne dziedziczone wartości metadanych.

Ponieważ większość istniejących właściwości zależności nie jest właściwościami wirtualnymi, ich dziedziczona implementacja w tle istniejącego elementu członkowskiego. Po zastąpieniu charakterystyki metadanych nowa wartość metadanych zastępuje oryginalną wartość lub są one scalane:

- W przypadku elementu DefaultValuenowa wartość zastąpi istniejącą wartość domyślną. Jeśli nie określisz `DefaultValue` elementu w metadanych przesłonięcia, wartość pochodzi z najbliższego przodka określonego `DefaultValue` w metadanych.

- W przypadku elementu PropertyChangedCallbackdomyślna logika scalania przechowuje wszystkie `PropertyChangedCallback` wartości w tabeli, a wszystkie są

wywoływane po zmianie właściwości. Kolejność wywołania zwrotnego jest określana przez głębokość klasy, gdzie wywołanie zwrotne zarejestrowane przez klasę bazową w hierarchii będzie uruchamiane jako pierwsze.

- W przypadku elementu CoerceValueCallbacknowa wartość zastąpi istniejącą `CoerceValueCallback` wartość. Jeśli nie określisz `CoerceValueCallback` elementu w metadanych przesłonięcia, wartość pochodzi z najbliższego przodka określonego `CoerceValueCallback` w metadanych.

> ⓘ **Uwaga**
>
> Domyślna logika scalania jest implementowana przez metodę **Merge** . Niestandardową logikę scalania można określić w klasie pochodnej, która dziedziczy właściwość zależności, przez zastąpienie `Merge` w tej klasie.

Dodawanie klasy jako właściciela

Aby "dziedziczyć" właściwość zależności zarejestrowaną w innej hierarchii klas, użyj AddOwner metody . Ta metoda jest zwykle używana, gdy klasa dodawania nie pochodzi z typu, który zarejestrował właściwość zależności. W wywołaniu `AddOwner` dodanie klasy może tworzyć i przypisywać metadane specyficzne dla typu dla właściwości zależności dziedziczonej. Aby być pełnym uczestnikiem systemu właściwości, za pomocą kodu i znaczników, klasa dodawania powinna implementować następujące publiczne składowe:

- Pole identyfikatora właściwości zależności. Wartość identyfikatora właściwości zależności jest wartością zwracaną wywołania `AddOwner` . To pole powinno być polem `public static readonly` typu DependencyProperty.

- Otoka CLR, która implementuje `get` metody i `set` metody dostępu. Korzystając z otoki właściwości, odbiorcy właściwości zależności mogą pobierać lub ustawiać wartości właściwości zależności, tak samo jak każda inna właściwość CLR. Metody `get` i `set` współdziałają z podstawowym systemem właściwości za pośrednictwem metody DependencyObject.GetValue i DependencyObject.SetValue wywołuje metodę , przekazując identyfikator właściwości zależności jako parametr. Zaimplementuj otokę w taki sam sposób, jak podczas rejestrowania niestandardowej właściwości zależności. Aby uzyskać więcej informacji, zobacz Właściwości zależności niestandardowych

Klasa, która wywołuje `AddOwner` , ma te same wymagania dotyczące uwidaczniania modelu obiektów właściwości zależności dziedziczonej jako klasy definiującej nową

niestandardową właściwość zależności. Aby uzyskać więcej informacji, zobacz Dodawanie typu właściciela dla właściwości zależności.

Metadane dołączonej właściwości

W WPF większość właściwości dołączonych interfejsu użytkownika w typach WPF jest implementowana jako właściwości zależności. Dołączone właściwości zaimplementowane jako właściwości zależności obsługują koncepcje właściwości zależności, takie jak metadane, które klasy pochodne mogą zastąpić. Metadane dołączonej właściwości zwykle nie różnią się od właściwości zależności. Można zastąpić wartość domyślną, właściwości zmiany wywołań zwrotnych i właściwości struktury WPF dla dziedziczonej właściwości dołączonej w wystąpieniach klasy zastępowania. Aby uzyskać więcej informacji, zobacz Dołączone metadane właściwości

> ⓘ **Uwaga**
>
> Zawsze używaj **RegisterAttached** polecenia , aby zarejestrować właściwości określone **Inherits** w metadanych. Mimo że dziedziczenie wartości właściwości może wydawać się działać dla nieprzyłączone właściwości zależności, zachowanie dziedziczenia wartości dla właściwości nieprzyłączonej za pośrednictwem niektórych podziałów obiektów w drzewie środowiska uruchomieniowego jest niezdefiniowane. Właściwość `Inherits` nie jest odpowiednia dla nieprzyłączone właściwości. Aby uzyskać więcej informacji, zobacz **RegisterAttached(String, Type, Type, PropertyMetadata)**sekcję i uwagi w temacie **Inherits**.

Dodawanie klasy jako właściciela dołączonej właściwości

Aby dziedziczyć dołączoną właściwość z innej klasy, ale uwidaczniać ją jako właściwość zależności nieprzyłączoną do klasy:

- Wywołaj metodę AddOwner , aby dodać klasę jako właściciela dołączonej właściwości zależności.
- Przypisz wartość zwracaną `AddOwner` wywołania do `public static readonly` pola do użycia jako identyfikator właściwości zależności.
- Zdefiniuj otokę CLR, która dodaje właściwość jako składową klasy i obsługuje nieprzyłączone użycie właściwości.

Zobacz też

- PropertyMetadata
- DependencyObject
- DependencyProperty
- GetMetadata
- AddOwner
- Omówienie właściwości zależności
- Metadane właściwości struktury

Wywołania zwrotne i walidacja właściwości zależności (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 12 min

W tym artykule opisano sposób definiowania właściwości zależności i implementowania wywołań zwrotnych właściwości zależności. Wywołania zwrotne obsługują walidację wartości, wymuszanie wartości i inną logikę, która jest potrzebna w przypadku zmiany wartości właściwości.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

Wywołania zwrotne weryfikujące wartość

Wywołania zwrotne z walidacją wartości umożliwiają sprawdzenie, czy nowa wartość właściwości zależności jest prawidłowa, zanim zostanie zastosowana przez system właściwości. To wywołanie zwrotne zgłasza wyjątek, jeśli wartość nie spełnia kryteriów weryfikacji.

Wywołania zwrotne wartości weryfikowania wartości można przypisać tylko do właściwości zależności raz podczas rejestracji właściwości. Podczas rejestrowania właściwości zależności możesz przekazać ValidateValueCallback odwołanie do Register(String, Type, Type, PropertyMetadata, ValidateValueCallback) metody . Wywołania zwrotne wartości weryfikacji nie są częścią metadanych właściwości i nie można ich zastąpić.

Efektywną wartością właściwości zależności jest jej zastosowana wartość. Efektywna wartość jest określana za pomocą pierwszeństwa wartości właściwości , gdy istnieje wiele danych wejściowych opartych na właściwościach. Jeśli wywołanie zwrotne wartości

weryfikacji jest zarejestrowane dla właściwości zależności, system właściwości wywoła wywołanie zwrotne wartości weryfikacji wartości po zmianie wartości, przekazując nową wartość jako obiekt. W wywołaniu zwrotnym można rzutować obiekt wartości z powrotem do typu zarejestrowanego w systemie właściwości, a następnie uruchomić na nim logikę walidacji. Wywołanie zwrotne zwraca wartość `true` , jeśli wartość jest prawidłowa dla właściwości, w przeciwnym razie `false`.

Jeśli wywołanie zwrotne weryfikuj-wartość zwraca `false` wartość , zgłaszany jest wyjątek, a nowa wartość nie jest stosowana. Autorzy aplikacji muszą być przygotowani do obsługi tych wyjątków. Typowym zastosowaniem wywołań zwrotnych wartości weryfikacji jest walidacja wartości wyliczenia lub ograniczenie wartości liczbowych, gdy reprezentują one miary, które mają limity. Wywołania zwrotne wartości weryfikacji są wywoływane przez system właściwości w różnych scenariuszach, w tym:

- Inicjowanie obiektu, które stosuje wartość domyślną w czasie tworzenia.
- Programowe wywołania do SetValueprogramu .
- Przesłonięcia metadanych, które określają nową wartość domyślną.

Wywołania zwrotne weryfikują wartość nie mają parametru DependencyObject , który określa wystąpienie, na którym ustawiono nową wartość. Wszystkie wystąpienia `DependencyObject` udziału tego samego wywołania zwrotnego wartości weryfikacji wartości, dzięki czemu nie mogą być używane do sprawdzania poprawności scenariuszy specyficznych dla wystąpienia. Aby uzyskać więcej informacji, zobacz ValidateValueCallback.

W poniższym przykładzie pokazano, jak zapobiec właściwości, wpisana jako , ustawiona na DoublePositiveInfinity lub NegativeInfinity.

C#

```
public class Gauge1 : Control
{
    public Gauge1() : base() { }

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty CurrentReadingProperty =
        DependencyProperty.Register(
            name: "CurrentReading",
            propertyType: typeof(double),
            ownerType: typeof(Gauge1),
            typeMetadata: new FrameworkPropertyMetadata(
                defaultValue: double.NaN,
                flags: FrameworkPropertyMetadataOptions.AffectsMeasure),
            validateValueCallback: new
ValidateValueCallback(IsValidReading));
```

```
    // CLR wrapper with get/set accessors.
    public double CurrentReading
    {
        get => (double)GetValue(CurrentReadingProperty);
        set => SetValue(CurrentReadingProperty, value);
    }

    // Validate-value callback.
    public static bool IsValidReading(object value)
    {
        double val = (double)value;
        return !val.Equals(double.NegativeInfinity) &&
            !val.Equals(double.PositiveInfinity);
    }
}
```

C#

```
public static void TestValidationBehavior()
{
    Gauge1 gauge = new();

    Debug.WriteLine($"Test value validation scenario:");

    // Set allowed value.
    gauge.CurrentReading = 5;
    Debug.WriteLine($"Current reading: {gauge.CurrentReading}");

    try
    {
        // Set disallowed value.
        gauge.CurrentReading = double.PositiveInfinity;
    }
    catch (ArgumentException e)
    {
        Debug.WriteLine($"Exception thrown by ValidateValueCallback:
{e.Message}");
    }

    Debug.WriteLine($"Current reading: {gauge.CurrentReading}");

    // Current reading: 5
    // Exception thrown by ValidateValueCallback: '∞' is not a valid value
for property 'CurrentReading'.
    // Current reading: 5
}
```

Wywołania zwrotne zmienione przez właściwość

Wywołania zwrotne zmienione przez właściwość powiadamiają o zmianie obowiązującej wartości właściwości zależności.

Wywołania zwrotne zmienione przez właściwość są częścią metadanych właściwości zależności. Jeśli pochodzisz z klasy definiującej właściwość zależności lub dodaj klasę jako właściciela właściwości zależności, możesz zastąpić metadane. Podczas zastępowania metadanych możesz podać nowe PropertyChangedCallback odwołanie. Użyj wywołania zwrotnego zmienionego przez właściwość, aby uruchomić logikę wymaganą w przypadku zmiany wartości właściwości.

W przeciwieństwie do wywołań zwrotnych validate-value, wywołania zwrotne zmienione przez właściwość mają parametr określający DependencyObject wystąpienie, na którym ustawiono nową wartość. W następnym przykładzie pokazano, jak wywołanie zwrotne zmienione przez właściwość może używać `DependencyObject` odwołania wystąpienia do wyzwalania wywołań zwrotnych coerce-value.

Wywołania zwrotne coerce-value

Wywołania zwrotne coerce-value zapewniają możliwość otrzymywania powiadomień, gdy obowiązująca wartość właściwości zależności ma ulec zmianie, aby można było dostosować nową wartość przed jej zastosowaniem. Oprócz wyzwalania przez system właściwości można wywołać wywołania zwrotne coerce-value z kodu.

Wywołania zwrotne wartości coerce są częścią metadanych właściwości zależności. Jeśli pochodzisz z klasy definiującej właściwość zależności lub dodaj klasę jako właściciela właściwości zależności, możesz zastąpić metadane. Podczas zastępowania metadanych możesz podać odwołanie do nowego CoerceValueCallbackelementu . Użyj wywołania zwrotnego coerce-value, aby ocenić nowe wartości i w razie potrzeby je wymuś. Wywołanie zwrotne zwraca wartość coerced, jeśli doszło do przymusu, w przeciwnym razie zwraca niezniechęconą nową wartość.

Podobnie jak wywołania zwrotne zmienione przez właściwość, wywołania zwrotne coerce-value mają parametr określający DependencyObject wystąpienie, na którym ustawiono nową wartość. W następnym przykładzie pokazano, jak wywołanie zwrotne wartości coerce-value może użyć `DependencyObject` odwołania wystąpienia do wartości właściwości współwłaścicieli.

> ⓘ **Uwaga**
>
> Wartości właściwości domyślnych nie mogą być coerced. Właściwość zależności ma wartość domyślną ustawioną na inicjowanie obiektu lub w przypadku wyczyszczenia innych wartości przy użyciu polecenia **ClearValue**.

Coerce-value i właściwości zmienione wywołania zwrotne w kombinacji

Zależności między właściwościami elementu można tworzyć przy użyciu wywołań zwrotnych coerce-value i wywołań zwrotnych zmienionych przez właściwość w kombinacji. Na przykład zmiany w jednej właściwości wymuszają przymus lub ponowną ocenę w innej właściwości zależności. W następnym przykładzie przedstawiono typowy scenariusz: trzy właściwości zależności, które odpowiednio przechowują bieżącą wartość, wartość minimalną i maksymalną wartość elementu interfejsu użytkownika. Jeśli wartość maksymalna zmieni się tak, aby była mniejsza niż bieżąca wartość, bieżąca wartość zostanie ustawiona na nową wartość maksymalną. Jeśli wartość minimalna zmieni się tak, aby była większa niż bieżąca wartość, bieżąca wartość zostanie ustawiona na nową wartość minimalną. W tym przykładzie PropertyChangedCallback wartość dla bieżącej wartości jawnie wywołuje CoerceValueCallback wartości minimalne i maksymalne.

C#

```
public class Gauge2 : Control
{
    public Gauge2() : base() { }

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty CurrentReadingProperty =
        DependencyProperty.Register(
            name: "CurrentReading",
            propertyType: typeof(double),
            ownerType: typeof(Gauge2),
            typeMetadata: new FrameworkPropertyMetadata(
                defaultValue: double.NaN,
                flags: FrameworkPropertyMetadataOptions.AffectsMeasure,
                propertyChangedCallback: new
PropertyChangedCallback(OnCurrentReadingChanged),
                coerceValueCallback: new
CoerceValueCallback(CoerceCurrentReading)
            ),
            validateValueCallback: new ValidateValueCallback(IsValidReading)
        );

    // CLR wrapper with get/set accessors.
    public double CurrentReading
    {
        get => (double)GetValue(CurrentReadingProperty);
        set => SetValue(CurrentReadingProperty, value);
    }

    // Validate-value callback.
```

```
    public static bool IsValidReading(object value)
    {
        double val = (double)value;
        return !val.Equals(double.NegativeInfinity) &&
!val.Equals(double.PositiveInfinity);
    }

    // Property-changed callback.
    private static void OnCurrentReadingChanged(DependencyObject depObj,
DependencyPropertyChangedEventArgs e)
    {
        depObj.CoerceValue(MinReadingProperty);
        depObj.CoerceValue(MaxReadingProperty);
    }

    // Coerce-value callback.
    private static object CoerceCurrentReading(DependencyObject depObj,
object value)
    {
        Gauge2 gauge = (Gauge2)depObj;
        double currentVal = (double)value;
        currentVal = currentVal < gauge.MinReading ? gauge.MinReading :
currentVal;
        currentVal = currentVal > gauge.MaxReading ? gauge.MaxReading :
currentVal;
        return currentVal;
    }

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty MaxReadingProperty =
DependencyProperty.Register(
        name: "MaxReading",
        propertyType: typeof(double),
        ownerType: typeof(Gauge2),
        typeMetadata: new FrameworkPropertyMetadata(
            defaultValue: double.NaN,
            flags: FrameworkPropertyMetadataOptions.AffectsMeasure,
            propertyChangedCallback: new
PropertyChangedCallback(OnMaxReadingChanged),
            coerceValueCallback: new CoerceValueCallback(CoerceMaxReading)
        ),
        validateValueCallback: new ValidateValueCallback(IsValidReading)
    );

    // CLR wrapper with get/set accessors.
    public double MaxReading
    {
        get => (double)GetValue(MaxReadingProperty);
        set => SetValue(MaxReadingProperty, value);
    }

    // Property-changed callback.
    private static void OnMaxReadingChanged(DependencyObject depObj,
DependencyPropertyChangedEventArgs e)
```

```
    {
        depObj.CoerceValue(MinReadingProperty);
        depObj.CoerceValue(CurrentReadingProperty);
    }

    // Coerce-value callback.
    private static object CoerceMaxReading(DependencyObject depObj, object
value)
    {
        Gauge2 gauge = (Gauge2)depObj;
        double maxVal = (double)value;
        return maxVal < gauge.MinReading ? gauge.MinReading : maxVal;
    }

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty MinReadingProperty =
DependencyProperty.Register(
    name: "MinReading",
    propertyType: typeof(double),
    ownerType: typeof(Gauge2),
    typeMetadata: new FrameworkPropertyMetadata(
        defaultValue: double.NaN,
        flags: FrameworkPropertyMetadataOptions.AffectsMeasure,
        propertyChangedCallback: new
PropertyChangedCallback(OnMinReadingChanged),
        coerceValueCallback: new CoerceValueCallback(CoerceMinReading)
    ),
    validateValueCallback: new ValidateValueCallback(IsValidReading));

    // CLR wrapper with get/set accessors.
    public double MinReading
    {
        get => (double)GetValue(MinReadingProperty);
        set => SetValue(MinReadingProperty, value);
    }

    // Property-changed callback.
    private static void OnMinReadingChanged(DependencyObject depObj,
DependencyPropertyChangedEventArgs e)
    {
        depObj.CoerceValue(MaxReadingProperty);
        depObj.CoerceValue(CurrentReadingProperty);
    }

    // Coerce-value callback.
    private static object CoerceMinReading(DependencyObject depObj, object
value)
    {
        Gauge2 gauge = (Gauge2)depObj;
        double minVal = (double)value;
        return minVal > gauge.MaxReading ? gauge.MaxReading : minVal;
    }
}
```

Zaawansowane scenariusze wywołania zwrotnego

Ograniczenia i żądane wartości

Jeśli lokalnie ustawiona wartość właściwości zależności zostanie zmieniona przez przymus, niezmieniona lokalnie ustawiona wartość zostanie zachowana jako *żądana wartość*. Jeśli przymus jest oparty na innych wartościach właściwości, system właściwości będzie dynamicznie ponownie wycenić przymus za każdym razem, gdy te inne wartości się zmieniają. W ramach ograniczeń przymusu system właściwości zastosuje wartość znajdującą się najbliżej żądanej wartości. Jeśli warunek przymusu nie będzie już stosowany, system właściwości przywróci żądaną wartość — przy założeniu, że nie jest aktywna wyższa wartość pierwszeństwa . Poniższy przykład testuje przymus w bieżącym scenariuszu wartości, wartości minimalnej i maksymalnej wartości.

C#

```
public static void TestCoercionBehavior()
{
    Gauge2 gauge = new()
    {
        // Set initial values.
        MinReading = 0,
        MaxReading = 10,
        CurrentReading = 5
    };

    Debug.WriteLine($"Test current/min/max values scenario:");

    // Current reading is not coerced.
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading is coerced to max value.
    gauge.MaxReading = 3;
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading is coerced, but tracking back to the desired value.
    gauge.MaxReading = 4;
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading reverts to the desired value.
    gauge.MaxReading = 10;
```

```
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading remains at the desired value.
    gauge.MinReading = 5;
    gauge.MaxReading = 5;
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading: 5 (min=0, max=10)
    // Current reading: 3 (min=0, max=3)
    // Current reading: 4 (min=0, max=4)
    // Current reading: 5 (min=0, max=10)
    // Current reading: 5 (min=5, max=5)
}
```

Dość złożone scenariusze zależności mogą wystąpić, gdy istnieje wiele właściwości zależnych od siebie w sposób cykliczny. Technicznie nie ma nic złego w przypadku złożonych zależności — z tą różnicą, że duża liczba ponownych ocen może zmniejszyć wydajność. Ponadto złożone zależności, które są widoczne w interfejsie użytkownika, mogą mylić użytkowników. Traktuj PropertyChangedCallback i CoerceValueCallback tak jednoznacznie, jak to możliwe, i nie ograniczaj nadmiernie.

Anuluj zmiany wartości

Wracając UnsetValue z elementu CoerceValueCallback, można odrzucić zmianę wartości właściwości. Ten mechanizm jest przydatny, gdy zmiana wartości właściwości jest inicjowana asynchronicznie, ale po zastosowaniu nie jest już prawidłowa dla bieżącego stanu obiektu. Innym scenariuszem może być selektywne pomijanie zmiany wartości w zależności od tego, skąd pochodzi. W poniższym przykładzie `CoerceValueCallback` metoda wywołuje GetValueSource metodę ValueSource , która zwraca strukturę z wyliczeniem BaseValueSource identyfikującym źródło nowej wartości.

C#

```
// Coerce-value callback.
private static object CoerceCurrentReading(DependencyObject depObj, object
value)
{
    // Get value source.
    ValueSource valueSource =
        DependencyPropertyHelper.GetValueSource(depObj,
CurrentReadingProperty);

    // Reject any property value change that's a locally set value.
    return valueSource.BaseValueSource == BaseValueSource.Local ?
```

```
        DependencyProperty.UnsetValue : value;
}
```

Zobacz też

- ValidateValueCallback
- PropertyChangedCallback
- CoerceValueCallback
- Omówienie właściwości zależności
- Metadane zależności właściwości
- Niestandardowe właściwości zależności

Właściwości zależności tylko do odczytu (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 4 min

Możesz użyć właściwości zależności tylko do odczytu, aby zapobiec ustawianiu wartości właściwości spoza kodu. W tym artykule omówiono istniejące właściwości zależności tylko do odczytu oraz scenariusze i techniki tworzenia niestandardowej właściwości zależności tylko do odczytu.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

Istniejące właściwości zależności tylko do odczytu

Właściwości zależności tylko do odczytu zwykle zgłaszają stan i nie powinny być modyfikowalne za pośrednictwem `public` metody dostępu. Na przykład struktura Windows Presentation Foundation (WPF) implementuje IsMouseOver właściwość jako tylko do odczytu, ponieważ jej wartość powinna być określana tylko przez dane wejściowe myszy. Jeśli `IsMouseOver` dozwolone są inne dane wejściowe, jego wartość może stać się niespójna z danymi wejściowymi myszy. Chociaż nie można ustawić za pomocą `public` metody dostępu, wiele istniejących właściwości zależności tylko do odczytu ma wartości określone przez wiele danych wejściowych.

Użycie właściwości zależności tylko do odczytu

Właściwości zależności tylko do odczytu nie mają zastosowania w kilku scenariuszach, w których właściwości zależności zwykle oferują rozwiązanie. Scenariusze, w których nie dotyczy, obejmują powiązanie danych, zastosowanie stylu do wartości, walidacji, animacji i dziedziczenia. Jednak właściwość zależności tylko do odczytu może być używana jako wyzwalacz właściwości w stylu. Na przykład IsMouseOver jest często używany do wyzwalania zmian w tle, na pierwszym planie lub innej widocznej właściwości kontrolki, gdy nad nią znajduje się mysz. System właściwości WPF wykrywa i zgłasza zmiany we właściwościach zależności tylko do odczytu, co powoduje obsługę funkcji wyzwalacza właściwości. Właściwości zależności tylko do odczytu są również przydatne podczas implementowania właściwości zależności typu kolekcji, gdzie tylko elementy kolekcji muszą być zapisywalne, a nie sam obiekt kolekcji. Aby uzyskać więcej informacji, zobacz Właściwości zależności typu kolekcji.

> **Uwaga**
>
> Tylko właściwości zależności, a nie zwykłe właściwości środowiska uruchomieniowego języka wspólnego, mogą być używane jako wyzwalacze właściwości w stylu.

Tworzenie niestandardowych właściwości zależności tylko do odczytu

Przed utworzeniem właściwości zależności, która jest tylko do odczytu, sprawdź scenariusze, które nie mają zastosowania.

Proces tworzenia właściwości zależności tylko do odczytu jest na wiele sposobów podobny do tworzenia właściwości zależności odczytu i zapisu z następującymi różnicami:

- Podczas rejestrowania właściwości tylko do odczytu wywołaj metodę RegisterReadOnly zamiast Register.
- Podczas implementowania otoki właściwości CLR upewnij się, że nie ma ona dostępu publicznego `set` .
- `RegisterReadOnly` zwraca wartość DependencyPropertyKey zamiast DependencyProperty. Zapisz element `DependencyPropertyKey` w niepublicowym elemencie członkowskim klasy.

Możesz określić wartość właściwości zależności tylko do odczytu przy użyciu dowolnej wybranej logiki. Zalecanym sposobem ustawienia wartości właściwości , początkowo lub

w ramach logiki środowiska uruchomieniowego, jest użycie przeciążenia SetValue , które akceptuje parametr typu `DependencyPropertyKey`. Preferowane `SetValue` jest obejście systemu właściwości i bezpośrednie ustawienie pola zaplecza.

Jak i gdzie ustawisz wartość właściwości zależności tylko do odczytu w aplikacji wpłynie na poziom dostępu przypisany do składowej klasy, która przechowuje `DependencyPropertyKey` element . Jeśli ustawisz tylko wartość właściwości z klasy, która rejestruje właściwość zależności, możesz użyć `private` modyfikatora dostępu. W scenariuszach, w których wartości właściwości zależności wpływają na siebie, można użyć parowanych PropertyChangedCallback i CoerceValueCallback wywołań zwrotnych w celu wyzwolenia zmian wartości. Aby uzyskać więcej informacji, zobacz Metadane właściwości zależności.

Jeśli musisz zmienić wartość właściwości zależności tylko do odczytu spoza klasy, która ją rejestruje, możesz użyć `internal` modyfikatora dostępu dla elementu `DependencyPropertyKey`. Można na przykład wywołać procedurę `SetValue` obsługi zdarzeń w tym samym zestawie. W poniższym przykładzie zdefiniowano klasę Akwarium, która wywołuje metodę `RegisterReadOnly` tworzenia właściwości `FishCount` zależności tylko do odczytu . Element `DependencyPropertyKey` jest przypisany do `internal static readonly` pola, aby kod w tym samym zestawie mógł zmienić wartość właściwości zależności tylko do odczytu.

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, owner type, and property metadata.
    // Assign DependencyPropertyKey to a nonpublic field.
    internal static readonly DependencyPropertyKey FishCountPropertyKey =
        DependencyProperty.RegisterReadOnly(
          name: "FishCount",
          propertyType: typeof(int),
          ownerType: typeof(Aquarium),
          typeMetadata: new FrameworkPropertyMetadata());

    // Declare a public get accessor.
    public int FishCount =>
        (int)GetValue(FishCountPropertyKey.DependencyProperty);
}
```

Ponieważ system właściwości WPF nie propaguje DependencyPropertyKey poza kodem, właściwości zależności tylko do odczytu mają lepsze zabezpieczenia zapisu niż właściwości zależności odczytu i zapisu. Użyj właściwości zależności tylko do odczytu,

jeśli chcesz ograniczyć dostęp do zapisu do tych, którzy mają odwołanie do elementu `DependencyPropertyKey`.

Natomiast identyfikator właściwości zależności dla właściwości zależności odczytu i zapisu jest dostępny za pośrednictwem systemu właściwości, niezależnie od tego, jaki modyfikator dostępu go przypiszesz. Aby uzyskać więcej informacji, zobacz Zabezpieczenia właściwości zależności.

Zobacz też

- Omówienie właściwości zależności
- Implementowanie właściwości Zależności
- Niestandardowe właściwości zależności
- Właściwości zależności typu kolekcji
- Zabezpieczenia właściwości zależności
- Style i szablony w WPF

Metadane właściwości struktury (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 4 min

Można ustawić opcje metadanych właściwości struktury dla właściwości zależności na poziomie platformy Windows Presentation Foundation (WPF). Oznaczenie poziomu platformy WPF ma zastosowanie, gdy interfejsy API prezentacji WPF i pliki wykonywalne obsługują renderowanie i powiązanie danych. Interfejsy API prezentacji i pliki wykonywalne wysyłają zapytania dotyczące FrameworkPropertyMetadata właściwości zależności.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

Kategorie metadanych właściwości platformy

FrameworkPropertyMetadata należy do następujących kategorii:

- Metadane wpływające na układ elementu, w szczególności AffectsArrangeflagi , AffectsMeasurei AffectsRender metadanych. Możesz ustawić te flagi, jeśli implementacja właściwości zależności ma wpływ na aspekt wizualny i implementujesz MeasureOverride lub ArrangeOverride w klasie. Metody `MeasureOverride` i `ArrangeOverride` zapewniają zachowanie specyficzne dla implementacji i renderowanie informacji do systemu układu. Gdy `AffectsArrange`właściwość , `AffectsMeasure`lub `AffectsRender` jest ustawiona na `true` wartość w metadanych właściwości zależności i jej obowiązujące zmiany wartości, system właściwości WPF zainicjuje żądanie unieważnienia wizualizacji elementu w celu wyzwolenia ponownego rysowania.

- Metadane wpływające na układ elementu nadrzędnego elementu, w szczególności AffectsParentArrange flagi i AffectsParentMeasure metadanych. Przykłady właściwości zależności WPF, które ustawiają te flagi, to FixedPage.Left i Paragraph.KeepWithNext.

- Metadane dziedziczenia wartości właściwości, w szczególności Inherits flagi i OverridesInheritanceBehavior metadanych. Domyślnie właściwości zależności nie dziedziczą wartości. OverridesInheritanceBehavior umożliwia również przejście dziedziczenia do drzewa wizualnego, co jest niezbędne w przypadku niektórych scenariuszy komposiowania sterowania. Aby uzyskać więcej informacji, zobacz Dziedziczenie wartości właściwości.

  > **Uwaga**
  >
  > Termin "dziedziczy" w kontekście wartości właściwości jest specyficzny dla właściwości zależności i nie odnosi się bezpośrednio do zarządzanych typów kodu i dziedziczenia składowych za pośrednictwem typów pochodnych. W kontekście właściwości zależności oznacza to, że elementy podrzędne mogą dziedziczyć wartości właściwości zależności od elementów nadrzędnych.

- Metadane powiązania danych, w szczególności BindsTwoWayByDefault flagi i IsNotDataBindable metadanych. Domyślnie właściwości zależności w strukturze WPF obsługują powiązanie jednokierunkowe. Rozważ ustawienie powiązania dwukierunkowego jako domyślnego ustawienia właściwości, które zgłaszają stan *i* są modyfikowalne przez akcję użytkownika, na przykład IsSelected. Należy również rozważyć ustawienie powiązania dwukierunkowego jako domyślnego, gdy użytkownicy kontrolki oczekują zaimplementowania jej właściwości, na przykład TextBox.Text. `BindsTwoWayByDefault` wpływa tylko na domyślny tryb powiązania. Aby edytować kierunek przepływu danych powiązania, ustaw wartość Binding.Mode. Możesz użyć `IsNotDataBindable` polecenia , aby wyłączyć powiązanie danych, gdy nie ma dla niego przypadku użycia. Aby uzyskać więcej informacji na temat powiązań danych, zobacz Omówienie powiązania danych.

- Rejestrowanie metadanych, w szczególności flaga Journal metadanych. Wartość domyślna flagi `Journal` dotyczy tylko `true` niektórych właściwości zależności, takich jak SelectedIndex. Kontrolki danych wejściowych użytkownika powinny ustawiać flagę `Journal` właściwości, których wartości przechowują wybory użytkowników, które muszą być przechowywane. Flaga `Journal` jest odczytywana przez aplikacje lub usługi, które obsługują dzienniki, w tym usługi dziennika WPF. Aby uzyskać informacje na temat przechowywania kroków nawigacji, zobacz Omówienie nawigacji.

FrameworkPropertyMetadata pochodzi bezpośrednio z UIPropertyMetadataelementu i implementuje flagi omówione tutaj. Jeśli nie ustawiono w szczególności, `FrameworkPropertyMetadata` flagi mają wartość `false`domyślną .

Reading FrameworkPropertyMetadata

Aby pobrać metadane dla właściwości zależności, wywołaj GetMetadata identyfikator DependencyProperty . Wywołanie `GetMetadata` zwraca `PropertyMetadata` obiekt. Jeśli musisz wysłać zapytanie o wartości `PropertyMetadata` metadanych platformy do FrameworkPropertyMetadataelementu .

Określanie strukturyPropertyMetadata

Podczas rejestrowania właściwości zależności możesz utworzyć i przypisać do niej metadane. Obiekt metadanych, który przypisujesz, może być PropertyMetadata lub jedną z jego klas pochodnych, na przykład FrameworkPropertyMetadata. Wybierz `FrameworkPropertyMetadata` właściwości zależności, które opierają się na interfejsach API prezentacji WPF i plikach wykonywalnych do renderowania i powiązania danych. Bardziej zaawansowaną opcją jest utworzenie `FrameworkPropertyMetadata` niestandardowej klasy raportowania metadanych z większą pulą flag. Możesz też użyć UIPropertyMetadata właściwości innych niż struktura, które wpływają na renderowanie interfejsu użytkownika.

Chociaż opcje metadanych są zwykle ustawiane podczas rejestracji nowej właściwości zależności, można je ponownie zmienić w OverrideMetadata wywołaniach lub AddOwner . Podczas zastępowania metadanych zawsze przesłonięć ten sam typ metadanych, który był używany podczas rejestracji właściwości.

Cechy właściwości, które są uwidocznione przez `FrameworkPropertyMetadata` element, są czasami określane jako *flagi*. Jeśli tworzysz `FrameworkPropertyMetadata` wystąpienie, istnieją dwa sposoby wypełniania wartości flag:

1. Ustaw flagi na wystąpieniu FrameworkPropertyMetadataOptions typu wyliczenia. `FrameworkPropertyMetadataOptions` umożliwia określenie flag metadanych w kombinacji bitowej OR. Następnie utwórz `FrameworkPropertyMetadata` wystąpienie przy użyciu konstruktora, który ma `FrameworkPropertyMetadataOptions` parametr i przekaż wystąpienie `FrameworkPropertyMetadataOptions` . Aby zmienić flagi metadanych po przekazaniu `FrameworkPropertyMetadataOptions` do konstruktora FrameworkPropertyMetadata , zmień odpowiednią właściwość na nowym `FrameworkPropertyMetadata` wystąpieniu. Jeśli na przykład ustawisz flagę

FrameworkPropertyMetadataOptions.NotDataBindable , możesz cofnąć ją, ustawiając wartość FrameworkPropertyMetadata.IsNotDataBindable`false`.

2. `FrameworkPropertyMetadata` Utwórz wystąpienie przy użyciu konstruktora, który nie ma `FrameworkPropertyMetadataOptions` parametru, a następnie ustaw odpowiednie Boolean flagi na .`FrameworkPropertyMetadata` Ustaw wartości flagi przed skojarzeniem `FrameworkPropertyMetadata` wystąpienia z właściwością zależności, w przeciwnym razie uzyskasz wartość InvalidOperationException.

Zachowanie zastępowania metadanych

Podczas zastępowania metadanych właściwości platformy zmieniono wartości metadanych zamień lub zostały scalone z oryginalnymi wartościami:

- W przypadku elementu PropertyChangedCallbackdomyślna logika scalania zachowuje poprzednie `PropertyChangedCallback` wartości w tabeli, a wszystkie są wywoływane po zmianie właściwości. Kolejność wywołania zwrotnego jest określana przez głębokość klasy, w której wywołanie zwrotne zarejestrowane przez klasę bazową w hierarchii będzie uruchamiane najpierw. Dziedziczone wywołania zwrotne są uruchamiane tylko raz i są własnością klasy, która dodała je do metadanych.

- W przypadku elementu DefaultValuenowa wartość zastąpi istniejącą wartość domyślną. Jeśli nie określisz `DefaultValue` elementu w metadanych przesłonięcia i jeśli istniejąca FrameworkPropertyMetadata ma `Inherits` ustawioną flagę, wartość domyślna pochodzi z najbliższego modułu nadrzędnego określonego `DefaultValue` w metadanych.

- W przypadku elementu CoerceValueCallbacknowa wartość zastąpi istniejącą `CoerceValueCallback` wartość. Jeśli nie określisz `CoerceValueCallback` wartości w metadanych zastąpienia, wartość pochodzi z najbliższego modułu nadrzędnego w łańcuchu dziedziczenia, który określił `CoerceValueCallback`wartość .

- W przypadku `FrameworkPropertyMetadata` flag innych niż dziedziczone można zastąpić wartość domyślną `false` wartością `true` . Można jednak zastąpić tylko `true` wartość wartością `false` , Inherits, JournalOverridesInheritanceBehaviori SubPropertiesDoNotAffectRender.

ⓘ **Uwaga**

Domyślna logika scalania jest implementowana przez metodę **Merge** . Można określić niestandardową logikę scalania w klasie pochodnej, która dziedziczy właściwość zależności, przesłaniając `Merge` ją w tej klasie.

Zobacz też

- PropertyMetadata
- GetMetadata
- OverrideMetadata
- AddOwner
- Metadane właściwości zależności
- Przegląd Właściwości zależności
- Właściwości zależności niestandardowych

Zabezpieczenia właściwości zależności (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 5 min

Ułatwienia dostępu właściwości zależności odczytu i zapisu za pośrednictwem systemu właściwości Windows Presentation Foundation (WPF) skutecznie sprawiają, że są one właściwościami publicznymi. W związku z tym nie można zagwarantować zabezpieczeń wartości właściwości zależności read-write. System właściwości WPF zapewnia więcej zabezpieczeń dla właściwości zależności tylko do odczytu, dzięki czemu można ograniczyć dostęp do zapisu.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Dostęp i zabezpieczenia otoki właściwości

Otoka właściwości środowiska uruchomieniowego języka wspólnego (CLR) jest zwykle uwzględniana w implementacjach właściwości zależności odczytu i zapisu w celu uproszczenia uzyskiwania lub ustawiania wartości właściwości. W przypadku dołączeń otoka właściwości CLR to metoda wygody, która implementuje GetValue wywołania statyczne i SetValue , które współdziałają z podstawową właściwością zależności. Zasadniczo otoka właściwości CLR uwidacznia właściwość zależności jako właściwość CLR wspieraną przez właściwość zależności, a nie pole prywatne.

Zastosowanie mechanizmów zabezpieczeń i ograniczenie dostępu do otoki właściwości CLR może uniemożliwić użycie metody wygody, ale te techniki nie uniemożliwią bezpośrednich wywołań do `GetValue` lub `SetValue`. Innymi słowy, właściwość zależności read-write jest zawsze dostępna za pośrednictwem systemu właściwości WPF. Jeśli implementujesz właściwość zależności read-write, unikaj ograniczania dostępu do otoki właściwości CLR. Zamiast tego zadeklaruj otokę właściwości CLR jako element członkowski publiczny, aby osoby wywołujące wiedziały o prawdziwym poziomie dostępu właściwości zależności.

Ekspozycja systemu właściwości właściwości zależności

System właściwości WPF zapewnia dostęp do właściwości zależności read-write za pomocą identyfikatora DependencyProperty . Identyfikator jest używany do użycia w GetValue wywołaniach i SetValue . Nawet jeśli pole identyfikatora statycznego jest niepubliczne, kilka aspektów systemu właściwości zwróci `DependencyProperty` wartość tak, jak istnieje w wystąpieniu klasy lub klasy pochodnej. Na przykład GetLocalValueEnumerator metoda zwraca identyfikatory wystąpień właściwości zależności z lokalnie ustawioną wartością. Ponadto można zastąpić metodę wirtualną OnPropertyChanged w celu odbierania danych zdarzeń, które będą zgłaszać `DependencyProperty` identyfikator właściwości zależności, które uległy zmianie. Aby osoby wywołujące wiedziały o prawdziwym poziomie dostępu właściwości zależności read-write, zadeklaruj pole identyfikatora jako element członkowski publiczny.

> ⓘ **Uwaga**
>
> Mimo że deklarowanie pola `private` identyfikatora właściwości zależności zmniejsza liczbę sposobów dostępności właściwości zależności read-write, właściwość nie będzie **prywatna** zgodnie z definicją języka CLR.

Zabezpieczenia walidacji

Zastosowanie elementu Demand do elementu i oczekuje, że walidacja zakończy się niepowodzeniem ValidateValueCallback po `Demand` awarii, nie jest odpowiednim mechanizmem zabezpieczeń ograniczania zmian wartości właściwości. Ponadto nowe unieważnienie wartości wymuszane za pośrednictwem `ValidateValueCallback` może zostać pominięte przez złośliwych osób wywołujących, jeśli te osoby wywołujące działają w domenie aplikacji.

Dostęp do właściwości zależności tylko do odczytu

Aby ograniczyć dostęp, zarejestruj swoją właściwość jako właściwość zależności tylko do odczytu, wywołując metodę RegisterReadOnly . Metoda `RegisterReadOnly` zwraca wartość DependencyPropertyKey, którą można przypisać do pola klasy innej niż publiczna. W przypadku właściwości zależności tylko do odczytu system właściwości WPF zapewni dostęp do zapisu tylko tym, którzy mają odwołanie do `DependencyPropertyKey`elementu . Aby zilustrować to zachowanie, następujący kod testowy:

- Tworzy wystąpienie klasy, która implementuje właściwości zależności tylko do odczytu i zapisu i tylko do odczytu.
- `private` Przypisuje modyfikator dostępu do każdego identyfikatora.
- Implementuje `get` tylko metody dostępu.
- GetLocalValueEnumerator Używa metody w celu uzyskania dostępu do podstawowych właściwości zależności za pośrednictwem systemu właściwości WPF.
- Wywołania GetValue i SetValue przetestowanie dostępu do każdej wartości właściwości zależności.

C#

```
    /// <summary>
    ///  Test get/set access to dependency properties exposed through the
WPF property system.
    /// </summary>
    public static void DependencyPropertyAccessTests()
    {
        // Instantiate a class that implements read-write and read-only
dependency properties.
        Aquarium _aquarium = new();
        // Access each dependency property using the LocalValueEnumerator
method.
        LocalValueEnumerator localValueEnumerator =
_aquarium.GetLocalValueEnumerator();
        while (localValueEnumerator.MoveNext())
        {
            DependencyProperty dp = localValueEnumerator.Current.Property;
            string dpType = dp.ReadOnly ? "read-only" : "read-write";
            // Test read access.
            Debug.WriteLine($"Attempting to get a {dpType} dependency
property value...");
            Debug.WriteLine($"Value ({dpType}):
{(int)_aquarium.GetValue(dp)}");
            // Test write access.
            try
            {
                Debug.WriteLine($"Attempting to set a {dpType} dependency
property value to 2...");
                _aquarium.SetValue(dp, 2);
            }
            catch (InvalidOperationException e)
            {
                Debug.WriteLine(e.Message);
            }
            finally
            {
                Debug.WriteLine($"Value ({dpType}):
{(int)_aquarium.GetValue(dp)}");
            }
        }
```

```
        // Test output:

        // Attempting to get a read-write dependency property value...
        // Value (read-write): 1
        // Attempting to set a read-write dependency property value to 2...
        // Value (read-write): 2

        // Attempting to get a read-only dependency property value...
        // Value (read-only): 1
        // Attempting to set a read-only dependency property value to 2...
        // 'FishCountReadOnly' property was registered as read-only
        // and cannot be modified without an authorization key.
        // Value (read-only): 1
    }
}

public class Aquarium : DependencyObject
{
    public Aquarium()
    {
        // Assign locally-set values.
        SetValue(FishCountProperty, 1);
        SetValue(FishCountReadOnlyPropertyKey, 1);
    }

    // Failed attempt to restrict write-access by assigning the
    // DependencyProperty identifier to a non-public field.
    private static readonly DependencyProperty FishCountProperty =
        DependencyProperty.Register(
          name: "FishCount",
          propertyType: typeof(int),
          ownerType: typeof(Aquarium),
          typeMetadata: new PropertyMetadata());

    // Successful attempt to restrict write-access by assigning the
    // DependencyPropertyKey to a non-public field.
    private static readonly DependencyPropertyKey
FishCountReadOnlyPropertyKey =
        DependencyProperty.RegisterReadOnly(
          name: "FishCountReadOnly",
          propertyType: typeof(int),
          ownerType: typeof(Aquarium),
          typeMetadata: new PropertyMetadata());

    // Declare public get accessors.
    public int FishCount => (int)GetValue(FishCountProperty);
    public int FishCountReadOnly =>
(int)GetValue(FishCountReadOnlyPropertyKey.DependencyProperty);
}
```

Zobacz też

- DependencyProperty
- GetLocalValueEnumerator
- OnPropertyChanged
- Niestandardowe właściwości zależności
- Implementowanie właściwości Zależności

Bezpieczne wzorce konstruktorów dla obiektów DependencyObjects (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 9 min

Istnieje ogólna zasada programowania kodu zarządzanego, często wymuszana przez narzędzia do analizy kodu, że konstruktory klas nie powinny wywoływać metod zastępowalnych. Jeśli metoda zastępowalna jest wywoływana przez konstruktor klasy bazowej, a klasa pochodna zastępuje tę metodę, metoda zastąpienia w klasie pochodnej może zostać uruchomiona przed konstruktorem klasy pochodnej. Jeśli konstruktor klasy pochodnej wykonuje inicjowanie klasy, metoda klasy pochodnej może uzyskać dostęp do niezainicjowanych składowych klasy. Klasy właściwości zależności powinny unikać ustawiania wartości właściwości zależności w konstruktorze klasy, aby uniknąć problemów z inicjowaniem środowiska uruchomieniowego. W tym artykule opisano sposób implementowania DependencyObject konstruktorów w sposób, który pozwala uniknąć tych problemów.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Metody wirtualne i wywołania zwrotne systemu właściwości

Metody wirtualne właściwości zależności i wywołania zwrotne są częścią systemu właściwości Windows Presentation Foundation (WPF) i rozszerzają wszechstronność właściwości zależności.

Podstawowa operacja podobna do ustawiania wartości właściwości zależności przy użyciu SetValue wywołania OnPropertyChanged zdarzenia i potencjalnie kilku wywołań zwrotnych systemu właściwości WPF.

`OnPropertyChanged` jest przykładem metody wirtualnej systemu właściwości WPF, która może zostać zastąpiona przez klasy, które mają DependencyObject w hierarchii dziedziczenia. Jeśli ustawisz wartość właściwości zależności w konstruktorze, który jest wywoływany podczas tworzenia wystąpienia niestandardowej klasy właściwości

zależności, a klasa utworzona z niej zastępuje `OnPropertyChanged` metodę wirtualną, metoda klasy pochodnej `OnPropertyChanged` zostanie uruchomiona przed dowolnym konstruktorem klasy pochodnej.

PropertyChangedCallback i CoerceValueCallback są przykładami wywołań zwrotnych systemu właściwości WPF, które mogą być rejestrowane przez klasy właściwości zależności i przesłaniane przez klasy, które pochodzą z nich. Jeśli ustawisz wartość właściwości zależności w konstruktorze niestandardowej klasy właściwości zależności, a klasa pochodząca z niej zastępuje jedną z tych wywołań zwrotnych w metadanych właściwości, wywołanie zwrotne klasy pochodnej zostanie uruchomione przed dowolnym konstruktorem klasy pochodnej. Ten problem nie jest istotny, ValidateValueCallback ponieważ nie jest częścią metadanych właściwości i można go określić tylko przez klasę rejestrującą.

Aby uzyskać więcej informacji na temat wywołań zwrotnych właściwości zależności, zobacz Wywołania zwrotne właściwości zależności i walidacja.

Analizatory .NET

Analizatory platformy kompilatora .NET sprawdzają kod C# lub Visual Basic pod kątem problemów z jakością kodu i stylem. Jeśli wywołasz metody zastępowalne w konstruktorze, gdy reguła analizatora CA2214 jest aktywna, zostanie wyświetlone ostrzeżenie `CA2214: Don't call overridable methods in constructors`. Reguła nie będzie jednak oznaczać flagi metod wirtualnych i wywołań zwrotnych wywoływanych przez bazowy system właściwości WPF, gdy wartość właściwości zależności jest ustawiona w konstruktorze.

Problemy spowodowane przez klasy pochodne

Jeśli przypieczętujesz niestandardową klasę właściwości zależności lub wiesz, że klasa nie będzie pochodzić z klasy, problemy z inicjowaniem środowiska uruchomieniowego klasy pochodnej nie mają zastosowania do tej klasy. Jeśli jednak tworzysz klasę właściwości zależności, która jest dziedziczona, na przykład jeśli tworzysz szablony lub rozszerzalny zestaw bibliotek kontrolek, unikaj wywoływania metod zastępowalnych lub ustawiania wartości właściwości zależności z konstruktora.

Poniższy kod testowy przedstawia niebezpieczny wzorzec konstruktora, w którym konstruktor klasy bazowej ustawia wartość właściwości zależności, wyzwalając wywołania metod wirtualnych i wywołań zwrotnych.

```
C#
```

```
    private static void TestUnsafeConstructorPattern()
    {
        //Aquarium aquarium = new();
        //Debug.WriteLine($"Aquarium temperature (C):
{aquarium.TempCelcius}");

        // Instantiate and set tropical aquarium temperature.
        TropicalAquarium tropicalAquarium = new(tempCelcius: 25);
        Debug.WriteLine($"Tropical aquarium temperature (C): " +
            $"{tropicalAquarium.TempCelcius}");

        /* Test output:
        Derived class static constructor running.
        Base class ValidateValueCallback running.
        Base class ValidateValueCallback running.
        Base class ValidateValueCallback running.
        Base class parameterless constructor running.
        Base class ValidateValueCallback running.
        Derived class CoerceValueCallback running.
        Derived class CoerceValueCallback: null reference exception.
        Derived class OnPropertyChanged event running.
        Derived class OnPropertyChanged event: null reference exception.
        Derived class PropertyChangedCallback running.
        Derived class PropertyChangedCallback: null reference exception.
        Aquarium temperature (C): 20
        Derived class parameterless constructor running.
        Derived class parameter constructor running.
        Base class ValidateValueCallback running.
        Derived class CoerceValueCallback running.
        Derived class OnPropertyChanged event running.
        Derived class PropertyChangedCallback running.
        Tropical aquarium temperature (C): 25
        */
    }
}

public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata with default value,
    // and validate-value callback.
    public static readonly DependencyProperty TempCelciusProperty =
        DependencyProperty.Register(
            name: "TempCelcius",
            propertyType: typeof(int),
            ownerType: typeof(Aquarium),
            typeMetadata: new PropertyMetadata(defaultValue: 0),
            validateValueCallback:
                new ValidateValueCallback(ValidateValueCallback));

    // Parameterless constructor.
    public Aquarium()
    {
        Debug.WriteLine("Base class parameterless constructor running.");
```

```
        // Set typical aquarium temperature.
        TempCelcius = 20;

        Debug.WriteLine($"Aquarium temperature (C): {TempCelcius}");
    }

    // Declare public read-write accessors.
    public int TempCelcius
    {
        get => (int)GetValue(TempCelciusProperty);
        set => SetValue(TempCelciusProperty, value);
    }

    // Validate-value callback.
    public static bool ValidateValueCallback(object value)
    {
        Debug.WriteLine("Base class ValidateValueCallback running.");
        double val = (int)value;
        return val >= 0;
    }
}

public class TropicalAquarium : Aquarium
{
    // Class field.
    private static List<int> s_temperatureLog;

    // Static constructor.
    static TropicalAquarium()
    {
        Debug.WriteLine("Derived class static constructor running.");

        // Create a new metadata instance with callbacks specified.
        PropertyMetadata newPropertyMetadata = new(
            defaultValue: 0,
            propertyChangedCallback: new
PropertyChangedCallback(PropertyChangedCallback),
            coerceValueCallback: new
CoerceValueCallback(CoerceValueCallback));

        // Call OverrideMetadata on the dependency property identifier.
        TempCelciusProperty.OverrideMetadata(
            forType: typeof(TropicalAquarium),
            typeMetadata: newPropertyMetadata);
    }

    // Parameterless constructor.
    public TropicalAquarium()
    {
        Debug.WriteLine("Derived class parameterless constructor running.");
        s_temperatureLog = new List<int>();
    }

    // Parameter constructor.
```

```
    public TropicalAquarium(int tempCelcius) : this()
    {
        Debug.WriteLine("Derived class parameter constructor running.");
        TempCelcius = tempCelcius;
        s_temperatureLog.Add(tempCelcius);
    }

    // Property-changed callback.
    private static void PropertyChangedCallback(DependencyObject depObj,
        DependencyPropertyChangedEventArgs e)
    {
        Debug.WriteLine("Derived class PropertyChangedCallback running.");
        try
        {
            s_temperatureLog.Add((int)e.NewValue);
        }
        catch (NullReferenceException)
        {
            Debug.WriteLine("Derived class PropertyChangedCallback: null
reference exception.");
        }
    }

    // Coerce-value callback.
    private static object CoerceValueCallback(DependencyObject depObj,
object value)
    {
        Debug.WriteLine("Derived class CoerceValueCallback running.");
        try
        {
            s_temperatureLog.Add((int)value);
        }
        catch (NullReferenceException)
        {
            Debug.WriteLine("Derived class CoerceValueCallback: null
reference exception.");
        }
        return value;
    }

    // OnPropertyChanged event.
    protected override void
OnPropertyChanged(DependencyPropertyChangedEventArgs e)
    {
        Debug.WriteLine("Derived class OnPropertyChanged event running.");
        try
        {
            s_temperatureLog.Add((int)e.NewValue);
        }
        catch (NullReferenceException)
        {
            Debug.WriteLine("Derived class OnPropertyChanged event: null
reference exception.");
        }
```

```
        // Mandatory call to base implementation.
        base.OnPropertyChanged(e);
    }
}
```

Kolejność wywoływana metod w niebezpiecznym teście wzorca konstruktora to:

1. Konstruktor statyczny klasy pochodnej, który zastępuje metadane właściwości zależności do rejestrowania `Aquarium`PropertyChangedCallback i CoerceValueCallback.

2. Konstruktor klasy bazowej, który ustawia nową wartość właściwości zależności, co powoduje wywołanie SetValue metody. Wywołanie `SetValue` wyzwala wywołania zwrotne i zdarzenia w następującej kolejności:

   a. ValidateValueCallback, który jest implementowany w klasie bazowej. To wywołanie zwrotne nie jest częścią metadanych właściwości zależności i nie można zaimplementować w klasie pochodnej przez zastąpienie metadanych.

   b. `PropertyChangedCallback`, który jest implementowany w klasie pochodnej przez zastępowanie metadanych właściwości zależności. To wywołanie zwrotne powoduje wyjątek odwołania o wartości null, gdy wywołuje metodę w niezainicjowanym polu `s_temperatureLog`klasy .

   c. `CoerceValueCallback`, który jest implementowany w klasie pochodnej przez zastępowanie metadanych właściwości zależności. To wywołanie zwrotne powoduje wyjątek odwołania o wartości null, gdy wywołuje metodę w niezainicjowanym polu `s_temperatureLog`klasy .

   d. OnPropertyChanged zdarzenie, które jest implementowane w klasie pochodnej przez zastąpienie metody wirtualnej. To zdarzenie powoduje wyjątek odwołania o wartości null, gdy wywołuje metodę w niezainicjowanym polu `s_temperatureLog`klasy .

3. Konstruktor bez parametrów klasy pochodnej, który inicjuje `s_temperatureLog`.

4. Konstruktor parametru klasy pochodnej, który ustawia nową wartość właściwości zależności, co powoduje kolejne wywołanie `SetValue` metody. Ponieważ `s_temperatureLog` jest teraz inicjowane, wywołania zwrotne i zdarzenia będą uruchamiane bez powodowania wyjątków odwołania o wartości null.

Te problemy z inicjowaniem można uniknąć dzięki użyciu bezpiecznych wzorców konstruktorów.

Bezpieczne wzorce konstruktorów

Problemy z inicjowaniem klasy pochodnej pokazane w kodzie testowym można rozwiązać na różne sposoby, w tym:

- Unikaj ustawiania wartości właściwości zależności w konstruktorze niestandardowej klasy właściwości zależności, jeśli klasa może być używana jako klasa bazowa. Jeśli musisz zainicjować wartość właściwości zależności, rozważ ustawienie wymaganej wartości jako wartości domyślnej w metadanych właściwości podczas rejestracji właściwości zależności lub podczas zastępowania metadanych.

- Zainicjuj pola klasy pochodnej przed ich użyciem. Na przykład przy użyciu dowolnego z następujących podejść:

  - Utwórz wystąpienie i przypisz pola wystąpienia w pojedynczej instrukcji. W poprzednim przykładzie instrukcja `List<int> s_temperatureLog = new();` unikałaby późnego przypisania.

  - Wykonaj przypisanie w konstruktorze statycznym klasy pochodnej, który jest uruchamiany przed dowolnym konstruktorem klasy bazowej. W poprzednim przykładzie umieszczenie instrukcji `s_temperatureLog = new List<int>();` przypisania w konstruktorze statycznym klasy pochodnej pozwoli uniknąć późnego przypisania.

  - Użyj inicjowania i tworzenia wystąpienia z opóźnieniem, co inicjuje obiekty jako i wtedy, gdy są potrzebne. W poprzednim przykładzie utworzenie wystąpienia i przypisanie przy użyciu leniwej inicjalizacji i utworzenia wystąpienia pozwoli uniknąć późnego przypisania `s_temperatureLog` . Aby uzyskać więcej informacji, zobacz Inicjowanie z opóźnieniem.

- Unikaj używania niezainicjowanych zmiennych klasy w wywołaniach zwrotnych i zdarzeniach systemu właściwości WPF.

Zobacz też

- OnPropertyChanged
- PropertyChangedCallback
- CoerceValueCallback
- ValidateValueCallback
- Wywołania zwrotne i walidacja właściwości zależności
- Właściwości zależności niestandardowych
- Przegląd Właściwości zależności

- Zabezpieczenia właściwości zależności

Dziedziczenie wartości właściwości (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 6 min

Dziedziczenie wartości właściwości jest funkcją systemu właściwości Windows Presentation Foundation (WPF) i ma zastosowanie do właściwości zależności. Dziedziczenie wartości właściwości umożliwia elementom podrzędnym w drzewie elementów uzyskanie wartości określonej właściwości z najbliższego elementu nadrzędnego. Ponieważ element nadrzędny mógł również uzyskać jego wartość właściwości za pośrednictwem dziedziczenia wartości właściwości, system potencjalnie powtarza się z powrotem do katalogu głównego strony.

System właściwości WPF domyślnie nie włącza dziedziczenia wartości właściwości, a dziedziczenie wartości jest nieaktywne, chyba że jest to specjalnie włączone w metadanych właściwości zależności. Nawet w przypadku włączenia dziedziczenia wartości właściwości element podrzędny będzie dziedziczyć tylko wartość właściwości w przypadku braku wyższej wartości pierwszeństwa .

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

Dziedziczenie przez drzewo elementów

Dziedziczenie wartości właściwości nie jest takie samo, jak dziedziczenie klas w programowaniu zorientowanym na obiekty, gdzie klasy pochodne dziedziczą składowe klasy bazowej. Tego rodzaju dziedziczenie jest również aktywne w WPF, chociaż w języku XAML dziedziczone właściwości klasy bazowej są widoczne jako atrybuty elementów XAML reprezentujących klasy pochodne.

Dziedziczenie wartości właściwości to mechanizm, za pomocą którego wartość właściwości zależności jest propagowana z elementów nadrzędnych do elementów podrzędnych w drzewie elementów zawierających właściwość. W znaczników XAML drzewo elementów jest widoczne jako elementy zagnieżdżone.

Poniższy przykład przedstawia zagnieżdżone elementy w języku XAML. WPF rejestruje AllowDrop właściwość zależności w UIElement klasie z metadanymi właściwości, które umożliwiają dziedziczenie wartości właściwości i ustawia wartość domyślną na `false`. Właściwość `AllowDrop` zależności istnieje w elementach Canvas, StackPaneli Label , ponieważ wszystkie pochodzą z `UIElement` elementu . `AllowDrop` Ponieważ właściwość zależności jest `canvas1` ustawiona na `true` wartość , element podrzędny `stackPanel1` i `label1` elementy dziedziczą `true` jako ich `AllowDrop` wartość.

XAML

```
<Canvas x:Name="canvas1" Grid.Column="0" Margin="20" Background="Orange"
AllowDrop="True">
    <StackPanel Name="stackPanel1" Margin="20" Background="Green">
        <Label Name="label1" Margin="20" Height="40" Width="40"
Background="Blue"/>
    </StackPanel>
</Canvas>
```

Drzewo elementów można również utworzyć programowo, dodając obiekty elementów do kolekcji elementów podrzędnych innego obiektu elementu. W czasie wykonywania dziedziczenie wartości właściwości działa na wynikowym drzewie obiektów. W poniższym przykładzie `stackPanel2` element jest dodawany do kolekcji podrzędnej elementu `canvas2`. `label2` Podobnie jest dodawany do kolekcji podrzędnej elementu `stackPanel2`. AllowDrop Ponieważ właściwość zależności jest `canvas2` ustawiona na `true` wartość , element podrzędny `stackPanel2` i `label2` elementy dziedziczą `true` jako ich `AllowDrop` wartość.

C#

```
Canvas canvas2 = new()
{
    AllowDrop = true
};
StackPanel stackPanel2 = new();
Label label2 = new();
canvas2.Children.Add(stackPanel2);
stackPanel2.Children.Add(label2);
```

Praktyczne zastosowania dziedziczenia wartości właściwości

Określone właściwości zależności WPF mają domyślnie włączone dziedziczenie wartości, takie jak AllowDrop i FlowDirection. Zazwyczaj właściwości z włączonym dziedziczeniem wartości są domyślnie implementowane w klasach podstawowych elementów interfejsu użytkownika, więc istnieją w klasach pochodnych. Na przykład, ponieważ `AllowDrop` jest implementowana w klasie bazowej UIElement , ta właściwość zależności istnieje również na każdej kontrolce pochodzącej z `UIElement` klasy . WPF umożliwia dziedziczenie wartości we właściwościach zależności, dla których użytkownik jest wygodny, aby ustawić wartość właściwości raz w elemecie nadrzędnym i mieć tę wartość właściwości propagowaną do elementów podrzędnych w drzewie elementów podrzędnych.

Model dziedziczenia wartości właściwości przypisuje wartości właściwości, zarówno dziedziczone, jak i niezabezpieczone, zgodnie z pierwszeństwem wartości właściwości zależności. Dlatego wartość właściwości elementu nadrzędnego zostanie zastosowana tylko do elementu podrzędnego, jeśli właściwość elementu podrzędnego nie ma wyższej wartości pierwszeństwa, takiej jak wartość ustawiona lokalnie lub wartość uzyskana za pomocą stylów, szablonów lub powiązania danych.

Właściwość FlowDirection zależności ustawia kierunek układu elementów tekstu i podrzędnego interfejsu użytkownika w elemecie nadrzędnym. Zazwyczaj oczekuje się, że kierunek przepływu elementów tekstu i interfejsu użytkownika na stronie będzie spójny. Ponieważ dziedziczenie wartości jest włączone w metadanych właściwości klasy `FlowDirection`, wartość musi być ustawiana tylko raz w górnej części drzewa elementów dla strony. W rzadkich przypadkach, gdy dla strony jest przeznaczona kombinacja wskazówek przepływu, można ustawić inny kierunek przepływu na elemenie w drzewie, przypisując lokalnie ustawioną wartość. Nowy kierunek przepływu będzie następnie propagowany do elementów podrzędnych poniżej tego poziomu.

Tworzenie właściwości niestandardowej dziedziczonej

Właściwość zależności niestandardowej można dziedziczyć, włączając Inherits właściwość w wystąpieniu FrameworkPropertyMetadataklasy , a następnie rejestrując niestandardową właściwość zależności przy użyciu tego wystąpienia metadanych. Domyślnie `Inherits` wartość jest ustawiona na `false` wartość .`FrameworkPropertyMetadata` Dziedziczenie wartości właściwości wpływa na wydajność, więc ustawienie jest ustawione `Inherits` `true` tylko wtedy, gdy ta funkcja jest potrzebna.

Podczas rejestrowania właściwości zależności z włączoną funkcją w metadanych użyj RegisterAttached metody zgodnie z `Inherits` opisem w temacie Rejestrowanie dołączonej właściwości. Ponadto przypisz wartość domyślną do właściwości, aby istniała wartość dziedziczona. Możesz również utworzyć otokę właściwości z typem `get` właściciela i `set` metody dostępu, tak jak w przypadku właściwości zależności nieprzyłączonej. W ten sposób można ustawić wartość właściwości przy użyciu otoki właściwości dla właściciela lub typu pochodnego. Poniższy przykład tworzy właściwość zależności o nazwie `IsTransparent`, z `Inherits` włączoną i wartością domyślną .`false` Przykład zawiera również otokę właściwości z elementami `get` i `set` akcesoriami.

C#

```
public class Canvas_IsTransparentInheritEnabled : Canvas
{
    // Register an attached dependency property with the specified
    // property name, property type, owner type, and property metadata
    // (default value is 'false' and property value inheritance is enabled).
    public static readonly DependencyProperty IsTransparentProperty =
        DependencyProperty.RegisterAttached(
            name: "IsTransparent",
            propertyType: typeof(bool),
            ownerType: typeof(Canvas_IsTransparentInheritEnabled),
            defaultMetadata: new FrameworkPropertyMetadata(
                defaultValue: false,
                flags: FrameworkPropertyMetadataOptions.Inherits));

    // Declare a get accessor method.
    public static bool GetIsTransparent(Canvas element)
    {
        return (bool)element.GetValue(IsTransparentProperty);
    }

    // Declare a set accessor method.
    public static void SetIsTransparent(Canvas element, bool value)
    {
        element.SetValue(IsTransparentProperty, value);
    }

    // For convenience, declare a property wrapper with get/set accessors.
    public bool IsTransparent
    {
        get => (bool)GetValue(IsTransparentProperty);
        set => SetValue(IsTransparentProperty, value);
    }
}
```

Dołączone właściwości są koncepcyjnie podobne do właściwości globalnych. Możesz sprawdzić ich wartość na dowolnym DependencyObject i uzyskać prawidłowy wynik. Typowy scenariusz dla dołączonych właściwości polega na ustawieniu wartości

właściwości w elementach podrzędnych, a ten scenariusz jest bardziej skuteczny, jeśli właściwość, której dotyczy, jest niejawnie obecna jako dołączona właściwość dla każdego DependencyObject elementu w drzewie.

Dziedziczenie wartości właściwości w granicach drzewa

Dziedziczenie właściwości działa przez przechodzenie drzewa elementów. To drzewo jest często równoległe do drzewa logicznego. Jednak za każdym razem, gdy uwzględnisz obiekt podstawowego poziomu WPF, taki jak Brush, w znaczniku definiującym drzewo elementów, utworzono nieciągłe drzewo logiczne. Prawdziwe drzewo logiczne nie rozciąga się koncepcyjnie przez `Brush` element , ponieważ drzewo logiczne jest koncepcją na poziomie struktury WPF. Możesz użyć metod LogicalTreeHelper pomocnika do analizowania i wyświetlania zakresu drzewa logicznego. Dziedziczenie wartości właściwości jest w stanie przekazać dziedziczone wartości za pośrednictwem nieciągliwego drzewa logicznego, ale tylko wtedy, gdy właściwość dziedziczona została zarejestrowana jako dołączona właściwość i nie istnieje celowa granica blokująca dziedziczenie, na przykład Frame.

> ⓘ **Uwaga**
>
> Mimo że dziedziczenie wartości właściwości może działać dla właściwości zależności nieprzywiązanych, zachowanie dziedziczenia dla właściwości nieprzyłączonej przez niektóre granice elementów w drzewie środowiska uruchomieniowego jest niezdefiniowane. Za każdym razem, gdy określisz **Inherits** metadane właściwości, zarejestruj właściwości przy użyciu polecenia **RegisterAttached**.

Zobacz też

- Metadane zależności właściwości
- Omówienie dołączonych właściwości
- Rejestrowanie właściwości dołączonej
- Niestandardowe właściwości zależności
- Następstwo wartości właściwości zależności
- Metadane właściwości struktury

Właściwości ładowania i zależności XAML (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 3 min

Implementacja Windows Presentation Foundation (WPF) procesora XAML (Extensible Application Markup Language) jest z natury świadoma właściwości zależności. W związku z tym procesor XAML używa metod systemu właściwości WPF do ładowania atrybutów właściwości XAML i procesu, a całkowicie pomija otoki właściwości zależności przy użyciu metod systemu właściwości WPF, takich jak GetValue i SetValue. Dlatego w przypadku dodania logiki niestandardowej do otoki właściwości niestandardowej właściwości zależności nie będzie wywoływana przez procesor XAML, gdy wartość właściwości jest ustawiona w języku XAML.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Wymagania wstępne

W tym artykule założono podstawową wiedzę na temat właściwości zależności i omówiono właściwości zależności. Aby postępować zgodnie z przykładami w tym artykule, warto zapoznać się z językiem Extensible Application Markup Language (XAML) i wiedzieć, jak pisać aplikacje WPF.

Wydajność modułu ładującego XAML WPF

Procesor XAML WPF jest obliczanie mniej kosztowny, aby bezpośrednio wywołać metodę SetValue ustawiania wartości właściwości zależności, a nie używać otoki właściwości zależności.

Jeśli procesor XAML używał otoki właściwości, wymagałoby wywnioskowania całego modelu obiektu kodu kopii zapasowej tylko na podstawie relacji typu i składowych wskazanych w adiustacji. Chociaż typ można zidentyfikować na podstawie znaczników przy użyciu kombinacji atrybutów zestawu i identyfikując elementy członkowskie, określając, które elementy członkowskie można ustawić jako atrybut, i rozpoznawania obsługiwanych `xmlns` typów wartości właściwości, wymagałoby obszernego odbicia przy użyciu metody PropertyInfo.

System właściwości WPF utrzymuje tabelę magazynu właściwości zależności zaimplementowanych na danym DependencyObject typie pochodnym. Procesor XAML używa tej tabeli do wnioskowania identyfikatora właściwości zależności dla właściwości zależności. Na przykład zgodnie z konwencją identyfikator właściwości zależności dla właściwości zależności o nazwie `ABC` to `ABCProperty`. Procesor XAML może efektywnie ustawić wartość dowolnej właściwości zależności, wywołując metodę `SetValue` na typie zawierającym przy użyciu identyfikatora właściwości zależności.

Aby uzyskać więcej informacji na temat otoek właściwości zależności, zobacz Niestandardowe właściwości zależności.

Implikacje dotyczące niestandardowych właściwości zależności

Procesor WPF XAML pomija otoki właściwości i bezpośrednio wywołuje SetValue , aby ustawić wartość właściwości zależności. Dlatego należy unikać umieszczania dodatkowej logiki w `set` metodzie dostępu do niestandardowej właściwości zależności, ponieważ ta logika nie zostanie uruchomiona, gdy wartość właściwości jest ustawiona w języku XAML. Akcesorium `set` powinno zawierać tylko wywołanie `SetValue` .

Podobnie aspekty procesora WPF XAML, które pobierają wartości właściwości pomijają otokę właściwości i bezpośrednio wywołają metodę GetValue. Dlatego należy unikać umieszczania dodatkowej logiki we `get` właściwości zależności niestandardowej, ponieważ ta logika nie będzie uruchamiana, gdy wartość właściwości jest odczytywana w języku XAML. Akcesorium `get` powinno zawierać tylko wywołanie `GetValue` .

Przykład właściwości zależności z otokami

W poniższym przykładzie przedstawiono zalecaną definicję właściwości zależności z otokami właściwości. Identyfikator właściwości zależności jest przechowywany jako `public static readonly` pole, a `get` metody i `set` nie zawierają kodu poza wymaganymi metodami systemu właściwości WPF, które są z powrotem wartości właściwości zależności. Jeśli masz kod, który musi zostać uruchomiony, gdy wartość właściwości zależności ulegnie zmianie, rozważ umieszczenie tego kodu we PropertyChangedCallback właściwości zależności. Aby uzyskać więcej informacji, zobacz Wywołania zwrotne zmienione przez właściwość.

C#

```
// Register a dependency property with the specified property name,
// property type, owner type, and property metadata. Store the dependency
```

```
// property identifier as a public static readonly member of the class.
public static readonly DependencyProperty AquariumGraphicProperty =
    DependencyProperty.Register(
      name: "AquariumGraphic",
      propertyType: typeof(Uri),
      ownerType: typeof(Aquarium),
      typeMetadata: new FrameworkPropertyMetadata(
          defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
          flags: FrameworkPropertyMetadataOptions.AffectsRender,
          propertyChangedCallback: new
PropertyChangedCallback(OnUriChanged))
    );

// Property wrapper with get & set accessors.
public Uri AquariumGraphic
{
    get => (Uri)GetValue(AquariumGraphicProperty);
    set => SetValue(AquariumGraphicProperty, value);
}

// Property-changed callback.
private static void OnUriChanged(DependencyObject dependencyObject,
    DependencyPropertyChangedEventArgs e)
{
    // Some custom logic that runs on effective property value change.
    Uri newValue = (Uri)dependencyObject.GetValue(AquariumGraphicProperty);
    Debug.WriteLine($"OnUriChanged: {newValue}");
}
```

Zobacz też

- Omówienie właściwości zależności
- XAML w WPF
- Niestandardowe właściwości zależności
- Metadane zależności właściwości
- Właściwości zależności typu kolekcji
- Zabezpieczenia właściwości zależności
- Bezpieczne wzorce konstruktora dla obiektów DependencyObject

Jak zaimplementować właściwość zależności (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 2 min

W tym artykule opisano sposób implementowania właściwości zależności przy użyciu DependencyProperty pola do tworzenia kopii zapasowej właściwości środowiska uruchomieniowego języka wspólnego (CLR). Właściwości zależności obsługują kilka zaawansowanych funkcji systemu właściwości Windows Presentation Foundation (WPF). Te funkcje obejmują style, powiązanie danych, dziedziczenie, animację i wartości domyślne. Jeśli chcesz, aby właściwości zdefiniowane do obsługi tych funkcji były obsługiwane, zaimplementuj właściwości jako właściwość zależności.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Przykład

W poniższym przykładzie pokazano, jak zarejestrować właściwość zależności, wywołując metodę Register . Metoda `Register` zwraca DependencyProperty wystąpienie o nazwie *identyfikator właściwości zależności*. Identyfikator jest przechowywany w `static readonly` polu i zawiera nazwę i cechy właściwości zależności.

Pole identyfikatora musi być zgodne z konwencją `<property name>Property` nazewnictwa . Jeśli na przykład zarejestrujesz właściwość zależności o nazwie `Location`, pole identyfikatora powinno mieć nazwę `LocationProperty`. Jeśli nie będziesz przestrzegać tego wzorca nazewnictwa, projektanci WPF mogą nie zgłaszać właściwości poprawnie, a aspekty aplikacji stylu systemu właściwości mogą nie zachowywać się zgodnie z oczekiwaniami.

W poniższym przykładzie nazwa właściwości zależności i jej metody dostępu CLR to `HasFish`, więc pole identyfikatora ma nazwę `HasFishProperty`. Typ właściwości zależności to Boolean i typ właściciela rejestrujący właściwość zależności to `Aquarium`.

Możesz określić domyślne metadane dla właściwości zależności. W tym przykładzie ustawiono wartość `false` domyślną dla `HasFish` właściwości zależności.

```
C#
```

```
public class Aquarium : DependencyObject
{
    public static readonly DependencyProperty HasFishProperty =
        DependencyProperty.Register(
            name: "HasFish",
            propertyType: typeof(bool),
            ownerType: typeof(Aquarium),
            typeMetadata: new FrameworkPropertyMetadata(defaultValue:
false));

    public bool HasFish
    {
        get => (bool)GetValue(HasFishProperty);
        set => SetValue(HasFishProperty, value);
    }
}
```

Aby uzyskać więcej informacji na temat sposobu i dlaczego zaimplementować właściwość zależności, a nie tylko tworzenie kopii zapasowej właściwości CLR za pomocą pola prywatnego, zobacz Omówienie właściwości zależności.

Zobacz też

- Omówienie właściwości zależności
- Tematy z instrukcjami

Jak zarejestrować dołączoną właściwość (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 2 min

W tym artykule opisano sposób rejestrowania dołączonej właściwości i udostępniania publicznych metod dostępu, które umożliwiają dostęp do dołączonej właściwości za pośrednictwem języka XAML (Extensible Application Markup Language) i kodu. Dołączone właściwości umożliwiają ustawienie dodatkowych par właściwości/wartości dla dowolnego elementu XAML, mimo że element nie definiuje tych dodatkowych właściwości w modelu obiektów. Dodatkowe właściwości są globalnie dostępne. Dołączone właściwości są zwykle definiowane jako wyspecjalizowana forma właściwości zależności, która nie ma konwencjonalnej otoki właściwości. Większość dołączonych właściwości dla typów Windows Presentation Foundation (WPF) jest również implementowana jako właściwości zależności. Właściwości zależności można tworzyć na dowolnym DependencyObject typie pochodnym.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Przykład

W poniższym przykładzie pokazano, jak zarejestrować dołączoną właściwość jako właściwość zależności przy użyciu RegisterAttached metody . Klasa dostawcy ma opcję określenia wartości domyślnej w metadanych właściwości. Aby uzyskać więcej informacji na temat metadanych właściwości, zobacz Metadane właściwości zależności. W tym przykładzie `HasFish` właściwość ma typ wartości z wartością domyślną ustawioną Boolean na `false`.

Klasa dostawcy dla dołączonej właściwości musi zapewniać statyczne metody dostępu get/set, które są zgodne z konwencją `Get<property name>` nazewnictwa i `Set<property name>`. Czytnik XAML używa metod dostępu do rozpoznawania atrybutu XAML dla dołączonej właściwości i rozpoznawania jej wartości do odpowiedniego typu. Te metody dostępu są niezbędne, nawet jeśli dołączona właściwość nie jest zarejestrowana jako właściwość zależności.

C#

```
public class Aquarium : UIElement
{
    // Register an attached dependency property with the specified
    // property name, property type, owner type, and property metadata.
    public static readonly DependencyProperty HasFishProperty =
        DependencyProperty.RegisterAttached(
          "HasFish",
          typeof(bool),
          typeof(Aquarium),
          new FrameworkPropertyMetadata(defaultValue: false,
              flags: FrameworkPropertyMetadataOptions.AffectsRender)
        );

    // Declare a get accessor method.
    public static bool GetHasFish(UIElement target) =>
        (bool)target.GetValue(HasFishProperty);

    // Declare a set accessor method.
    public static void SetHasFish(UIElement target, bool value) =>
        target.SetValue(HasFishProperty, value);
}
```

Zobacz też

- DependencyProperty
- Omówienie dołączonych właściwości
- Omówienie właściwości zależności
- Niestandardowe właściwości zależności
- Tematy z instrukcjami

Jak zastąpić metadane dla właściwości zależności (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 2 min

Gdy pochodzisz z klasy definiującej właściwość zależności, dziedziczysz właściwość zależności i jej metadane. W tym artykule opisano, jak można zastąpić metadane dziedziczonej właściwości zależności przez wywołanie OverrideMetadata metody . Zastępowanie metadanych umożliwia modyfikowanie właściwości dziedziczonej zależności w celu dopasowania ich do wymagań specyficznych dla podklasy.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Tło

Klasa, która definiuje właściwość zależności, może określać jej właściwości w PropertyMetadata typie pochodnym lub jednym z typów pochodnych, takich jak FrameworkPropertyMetadata. Jedną z tych cech jest domyślna wartość właściwości zależności. Wiele klas definiujących właściwości zależności, określa metadane właściwości podczas rejestracji właściwości zależności. Jeśli podczas rejestracji nie określono metadanych, system właściwości WPF przypisuje `PropertyMetadata` obiekt z wartościami domyślnymi. Klasy pochodne dziedziczące właściwości zależności za pośrednictwem dziedziczenia klas mają możliwość zastąpienia oryginalnych metadanych dowolnej właściwości zależności. W ten sposób klasy pochodne mogą selektywnie modyfikować właściwości zależności w celu spełnienia wymagań klasy. Podczas wywoływania OverrideMetadata(Type, PropertyMetadata)klasy pochodnej określa swój własny typ jako pierwszy parametr i wystąpienie metadanych jako drugi parametr.

Klasa pochodna, która zastępuje metadane we właściwości zależności, musi to zrobić, zanim właściwość zostanie umieszczona w użyciu przez system właściwości. Właściwość zależności jest umieszczana w użyciu, gdy wystąpienie dowolnej klasy, która rejestruje właściwość, jest tworzone. Aby spełnić to wymaganie, klasa pochodna powinna wywołać OverrideMetadata w konstruktorze statycznym. Zastąpienie metadanych właściwości zależności po utworzeniu wystąpienia typu właściciela nie spowoduje wystąpienia wyjątków, ale spowoduje niespójne zachowania w systemie właściwości. Ponadto typ

pochodny nie może zastąpić metadanych właściwości zależności więcej niż raz, a próby wykonania tej czynności spowodują wystąpienie wyjątku.

Przykład

W poniższym przykładzie klasa `TropicalAquarium` pochodna zastępuje metadane właściwości zależności dziedziczonej z klasy `Aquarium`bazowej . Typ metadanych to FrameworkPropertyMetadata, który obsługuje charakterystyki struktury WPF związane z interfejsem użytkownika, takie jak AffectsRender. Klasa pochodna nie zastępuje dziedziczonej `AffectsRender` flagi, ale aktualizuje wartość `AquariumGraphic` domyślną dla wystąpień klasy pochodnej.

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, owner type, and property metadata.
    public static readonly DependencyProperty AquariumGraphicProperty =
        DependencyProperty.Register(
          name: "AquariumGraphic",
          propertyType: typeof(Uri),
          ownerType: typeof(Aquarium),
          typeMetadata: new FrameworkPropertyMetadata(
              defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
              flags: FrameworkPropertyMetadataOptions.AffectsRender)
        );

    // Declare a read-write CLR wrapper with get/set accessors.
    public Uri AquariumGraphic
    {
        get => (Uri)GetValue(AquariumGraphicProperty);
        set => SetValue(AquariumGraphicProperty, value);
    }
}
```

C#

```
public class TropicalAquarium : Aquarium
{
    // Static constructor.
    static TropicalAquarium()
    {
        // Create a new metadata instance with a modified default value.
        FrameworkPropertyMetadata newPropertyMetadata = new(
            defaultValue: new Uri("http://www.contoso.com/tropical-aquarium-
graphic.jpg"));
```

```
        // Call OverrideMetadata on the dependency property identifier.
        // Pass in the type for which the new metadata will be applied
        // and the new metadata instance.
        AquariumGraphicProperty.OverrideMetadata(
            forType: typeof(TropicalAquarium),
            typeMetadata: newPropertyMetadata);
    }
}
```

Zobacz też

- DependencyProperty
- Metadane zależności właściwości
- Omówienie właściwości zależności
- Niestandardowe właściwości zależności

Omówienie zasobów XAML (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 12 min

Zasób to obiekt, który może być ponownie używany w różnych miejscach w aplikacji. Przykłady zasobów obejmują pędzle i style. W tym omówieniu opisano sposób używania zasobów w języku XAML (Extensible Application Markup Language). Zasoby można również tworzyć i uzyskiwać do ich dostępu przy użyciu kodu.

> **Uwaga**
>
> Zasoby XAML opisane w tym artykule różnią się od *zasobów aplikacji*, które zazwyczaj są plikami dodawanymi do aplikacji, takimi jak zawartość, dane lub pliki osadzone.

> **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Korzystanie z zasobów w języku XAML

Poniższy przykład definiuje SolidColorBrush element jako zasób w elemecie głównym strony. W przykładzie odwołuje się on do zasobu i używa go do ustawiania właściwości kilku elementów podrzędnych, w tym Ellipseelementów , a TextBlocki Button.

XAML

```
<Window x:Class="resources.ResExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ResExample" Height="400" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
        <Style TargetType="Border">
            <Setter Property="Background" Value="#4E1A3D" />
            <Setter Property="BorderThickness" Value="5" />
            <Setter Property="BorderBrush">
                <Setter.Value>
                    <LinearGradientBrush>
                        <GradientStop Offset="0.0" Color="#4E1A3D"/>
                        <GradientStop Offset="1.0" Color="Salmon"/>
                    </LinearGradientBrush>
                </Setter.Value>
```

```
            </Setter>
        </Style>
        <Style TargetType="TextBlock" x:Key="TitleText">
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="Foreground" Value="#4E87D4"/>
            <Setter Property="FontFamily" Value="Trebuchet MS"/>
            <Setter Property="Margin" Value="0,10,10,10"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="Label">
            <Setter Property="HorizontalAlignment" Value="Right"/>
            <Setter Property="FontSize" Value="13"/>
            <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
            <Setter Property="FontFamily" Value="Arial"/>
            <Setter Property="FontWeight" Value="Bold"/>
            <Setter Property="Margin" Value="0,3,10,0"/>
        </Style>
    </Window.Resources>

    <Border>
        <StackPanel>
            <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
            <TextBlock Style="{StaticResource Label}">Label</TextBlock>
            <TextBlock HorizontalAlignment="Right" FontSize="36"
Foreground="{StaticResource MyBrush}" Text="Text" Margin="20" />
            <Button HorizontalAlignment="Left" Height="30" Background="
{StaticResource MyBrush}" Margin="40">Button</Button>
            <Ellipse HorizontalAlignment="Center" Width="100" Height="100"
Fill="{StaticResource MyBrush}" Margin="10" />
        </StackPanel>
    </Border>

</Window>
```

Każdy element na poziomie platformy (FrameworkElement lub) ma Resources właściwość , która jest typem zawierającym ResourceDictionary zdefiniowane FrameworkContentElementzasoby. Zasoby można zdefiniować dla dowolnego elementu, takiego jak Button. Jednak zasoby są najczęściej definiowane w elemecie głównym, który znajduje się Window w przykładzie.

Każdy zasób w słowniku zasobów musi mieć unikatowy klucz. Podczas definiowania zasobów w znacznikach należy przypisać unikatowy klucz za pomocą dyrektywy x:Key. Zazwyczaj klucz jest ciągiem; Można go jednak również ustawić na inne typy obiektów przy użyciu odpowiednich rozszerzeń znaczników. Klucze inne niż ciągi dla zasobów są używane przez niektóre obszary funkcji w WPF, zwłaszcza w przypadku stylów, zasobów składników i stylów danych.

Można użyć zdefiniowanego zasobu ze składnią rozszerzenia znaczników zasobów, która określa nazwę klucza zasobu. Na przykład użyj zasobu jako wartości właściwości w innym elemecie.

```XAML
<Button Background="{StaticResource MyBrush}"/>
<Ellipse Fill="{StaticResource MyBrush}"/>
```

W poprzednim przykładzie, gdy moduł ładujący XAML przetwarza wartość `{StaticResource MyBrush}` właściwości Background na Button, logika wyszukiwania zasobów najpierw sprawdza słownik zasobów dla `Button` elementu. Jeśli `Button` nie ma definicji klucza `MyBrush` zasobu (w tym przykładzie nie, jego kolekcja zasobów jest pusta), wyszukiwanie następnie sprawdza element nadrzędny elementu `Button`. Jeśli zasób nie jest zdefiniowany w obiekcie nadrzędnym, kontynuuje sprawdzanie drzewa logicznego obiektu w górę, dopóki nie zostanie znalezione.

Jeśli zdefiniujesz zasoby w elemecie głównym, wszystkie elementy w drzewie logicznym, takie jak lub WindowPage, będą mogły uzyskiwać do niego dostęp. Możesz użyć tego samego zasobu, aby ustawić wartość dowolnej właściwości, która akceptuje ten sam typ, który reprezentuje zasób. W poprzednim przykładzie ten sam `MyBrush` zasób ustawia dwie różne właściwości: Button.Background i Rectangle.Fill.

Zasoby statyczne i dynamiczne

Do zasobu można odwoływać się jako statyczny lub dynamiczny. Odwołania są tworzone przy użyciu rozszerzenia znaczników StaticResource lub rozszerzenia dynamicResource Markup. Rozszerzenie znaczników to funkcja XAML, która umożliwia określenie odwołania do obiektu przez przetworzenie ciągu atrybutu rozszerzenia znaczników i zwrócenie obiektu do modułu ładującego XAML. Aby uzyskać więcej informacji na temat zachowania rozszerzenia znaczników, zobacz Markup Extensions and WPF XAML (Rozszerzenia znaczników i WPF XAML).

W przypadku używania rozszerzenia znaczników zazwyczaj podaje się jeden lub więcej parametrów w postaci ciągu, które są przetwarzane przez to rozszerzenie znaczników. Rozszerzenie znaczników StaticResource przetwarza klucz, wyszukując wartość tego klucza we wszystkich dostępnych słownikach zasobów. Przetwarzanie odbywa się podczas ładowania, czyli wtedy, gdy proces ładowania musi przypisać wartość właściwości. Rozszerzenie znaczników DynamicResource zamiast tego przetwarza klucz przez utworzenie wyrażenia, a to wyrażenie pozostaje nieoceniona do momentu uruchomienia aplikacji, w którym to czasie wyrażenie jest oceniane w celu podania wartości.

Podczas odwoływanie się do zasobu następujące kwestie mogą mieć wpływ na to, czy używasz odwołania do zasobów statycznych, czy dynamicznego odwołania do zasobów:

- Podczas określania ogólnego projektu sposobu tworzenia zasobów dla aplikacji (na stronę w aplikacji, w luźnym języku XAML lub w zestawie tylko do zasobów) należy wziąć pod uwagę następujące kwestie:
- Funkcjonalność aplikacji. Czy aktualizowanie zasobów w czasie rzeczywistym jest częścią wymagań aplikacji?
- Odpowiednie zachowanie odnośnika tego typu odwołania do zasobu.
- Określona właściwość lub typ zasobu oraz natywne zachowanie tych typów.

Zasoby statyczne

Odwołania do zasobów statycznych działają najlepiej w następujących sytuacjach:

- Projekt aplikacji koncentruje większość swoich zasobów na słownikach zasobów na poziomie strony lub aplikacji.

  Statyczne odwołania do zasobów nie są ponownie oceniane na podstawie zachowań środowiska uruchomieniowego, takich jak ponowne ładowanie strony. W związku z tym może istnieć pewna korzyść z wydajności, aby uniknąć dużej liczby odwołań do zasobów dynamicznych, gdy nie są one niezbędne na podstawie projektu zasobów i aplikacji.

- Ustawiasz wartość właściwości, która nie znajduje się na DependencyObject obiekcie lub Freezable.
- Tworzysz słownik zasobów skompilowany w bibliotekę DLL współdzieloną między aplikacjami.
- Tworzysz motyw dla kontrolki niestandardowej i definiujesz zasoby używane w motywach.

  W tym przypadku zwykle nie jest pożądane zachowanie odnośnika odwołania do zasobów dynamicznych. Zamiast tego należy użyć zachowania odwołania do zasobów statycznych, aby wyszukiwanie było przewidywalne i samodzielne dla motywu. W przypadku odwołania do zasobów dynamicznych nawet odwołanie w obrębie motywu pozostaje niezawarte do czasu wykonywania. Istnieje prawdopodobieństwo, że po zastosowaniu motywu niektóre elementy lokalne ponownie zdefiniowają klucz, do którego motyw próbuje się odwołać, a element lokalny spadnie przed samym motywem w wyszukiwaniu. W takim przypadku motyw nie będzie zachowywał się zgodnie z oczekiwaniami.

- Używasz zasobów do ustawiania dużej liczby właściwości zależności. Właściwości zależności mają efektywne buforowanie wartości włączone przez system właściwości, więc jeśli podasz wartość właściwości zależności, która może być oceniana w czasie ładowania, właściwość zależności nie musi sprawdzać ponownego oszacowania wyrażenia i może zwrócić ostatnią obowiązującą wartość. Ta technika może być korzyścią dla wydajności.

- Chcesz zmienić zasób bazowy dla wszystkich użytkowników lub zachować oddzielne wystąpienia z możliwością zapisu dla każdego konsumenta przy użyciu atrybutu x:Shared.

Zachowanie wyszukiwania zasobów statycznych

Poniżej opisano proces wyszukiwania, który odbywa się automatycznie, gdy zasób statyczny jest przywołyny przez właściwość lub element:

1. Proces wyszukiwania sprawdza żądany klucz w słowniku zasobów zdefiniowanym przez element, który ustawia właściwość .

2. Następnie proces wyszukiwania przechodzi drzewo logiczne w górę do elementu nadrzędnego i jego słownika zasobów. Ten proces będzie kontynuowany do momentu osiągnięcia elementu głównego.

3. Zasoby aplikacji są sprawdzane. Zasoby aplikacji to te zasoby w słowniku zasobów zdefiniowane przez Application obiekt dla aplikacji WPF.

Odwołania do zasobów statycznych z poziomu słownika zasobów muszą odwoływać się do zasobu, który został już zdefiniowany leksykalnie przed odwołaniem do zasobu. Odwołania do przodu nie mogą być rozpoznawane przez odwołanie do zasobów statycznych. Z tego powodu zaprojektuj strukturę słownika zasobów, tak aby zasoby zostały zdefiniowane na początku każdego odpowiedniego słownika zasobów lub w jego pobliżu.

Wyszukiwanie zasobów statycznych może rozszerzyć na motywy lub zasoby systemowe, ale to wyszukiwanie jest obsługiwane tylko dlatego, że moduł ładujący XAML odchyla żądanie. Odroczenie jest konieczne, aby motyw środowiska uruchomieniowego w czasie ładowania strony był prawidłowo stosowany do aplikacji. Jednak odwołania do zasobów statycznych do kluczy, które są znane tylko w motywach lub jako zasoby systemowe nie są zalecane, ponieważ takie odwołania nie są ponownie oceniane, jeśli motyw zostanie zmieniony przez użytkownika w czasie rzeczywistym. Dynamiczna dokumentacja zasobów jest bardziej niezawodna w przypadku żądania motywu lub zasobów systemowych. Wyjątek występuje, gdy sam element motywu żąda innego zasobu.

Odwołania te powinny być odwołaniami do zasobów statycznych ze względów wymienionych wcześniej.

Zachowanie wyjątku, jeśli nie znaleziono odwołania do zasobów statycznych, różni się. Jeśli zasób został odroczony, wyjątek występuje w czasie wykonywania. Jeśli zasób nie został odroczony, wyjątek występuje w czasie ładowania.

Zasoby dynamiczne

Zasoby dynamiczne działają najlepiej, gdy:

- Wartość zasobu, w tym zasobów systemowych lub zasobów, które w przeciwnym razie można ustawić przez użytkownika, zależy od warunków, które nie są znane dopiero w czasie wykonywania. Można na przykład utworzyć wartości ustawiające, które odwołują się do właściwości systemu uwidocznionych przez SystemColors, SystemFontslub SystemParameters. Te wartości są naprawdę dynamiczne, ponieważ ostatecznie pochodzą ze środowiska uruchomieniowego użytkownika i systemu operacyjnego. Możesz również mieć motywy na poziomie aplikacji, które mogą ulec zmianie, gdzie dostęp do zasobów na poziomie strony musi również przechwytywać zmianę.
- Tworzysz lub odwołujesz się do stylów motywu dla kontrolki niestandardowej.
- Zamierzasz dostosować zawartość obiektu ResourceDictionary w okresie istnienia aplikacji.
- Masz złożoną strukturę zasobów, która ma współzależności, gdzie może być wymagane odwołanie do przodu. Odwołania do zasobów statycznych nie obsługują odwołań do przodu, ale odwołania do zasobów dynamicznych obsługują je, ponieważ zasób nie musi być oceniany do czasu wykonania, a odwołania do przodu nie są zatem istotne.
- Odwołujesz się do zasobu, który jest duży z perspektywy kompilowania lub zestawu roboczego, a zasób może nie być używany natychmiast po załadowaniu strony. Odwołania do zasobów statycznych są zawsze ładowane z kodu XAML podczas ładowania strony. Jednak dynamiczne odwołanie do zasobów nie jest ładowane, dopóki nie zostanie użyte.
- Tworzysz styl, w którym wartości ustawiające mogą pochodzić z innych wartości, które mają wpływ na motywy lub inne ustawienia użytkownika.
- Stosujesz zasoby do elementów, które mogą być ponownie używane w drzewie logicznym w okresie istnienia aplikacji. Zmiana elementu nadrzędnego również

potencjalnie zmienia zakres wyszukiwania zasobów, więc jeśli chcesz, aby zasób ponownie przewartego elementu został przeszacowany na podstawie nowego zakresu, zawsze używaj odwołania do zasobu dynamicznego.

Zachowanie dynamicznego wyszukiwania zasobów

Zachowanie wyszukiwania zasobów dla dynamicznej dokumentacji zasobu jest równoległe zachowanie wyszukiwania w kodzie, jeśli wywołasz FindResource metodę lub SetResourceReference:

1. Wyszukiwanie sprawdza żądany klucz w słowniku zasobów zdefiniowanym przez element, który ustawia właściwość :

    - Jeśli element definiuje Style właściwość, System.Windows.FrameworkElement.Style element ma zaznaczony jego Resources słownik.

    - Jeśli element definiuje Template właściwość, System.Windows.FrameworkTemplate.Resources jest sprawdzany słownik elementu.

2. Odnośnik przechodzi przez drzewo logiczne w górę do elementu nadrzędnego i jego słownika zasobów. Ten proces będzie kontynuowany do momentu osiągnięcia elementu głównego.

3. Zasoby aplikacji są sprawdzane. Zasoby aplikacji to te zasoby w słowniku zasobów, które są definiowane przez Application obiekt dla aplikacji WPF.

4. Słownik zasobów motywu jest sprawdzany pod kątem aktualnie aktywnego motywu. Jeśli motyw zmieni się w czasie wykonywania, wartość jest ponownie wyceniana.

5. Zasoby systemowe są sprawdzane.

Zachowanie wyjątku (jeśli istnieje) różni się:

- Jeśli zasób został żądany przez FindResource wywołanie i nie został znaleziony, zgłaszany jest wyjątek.

- Jeśli zasób został żądany przez TryFindResource wywołanie i nie został znaleziony, nie zostanie zgłoszony wyjątek, a zwrócona wartość to `null`. Jeśli ustawiona właściwość nie akceptuje `null`wartości , nadal istnieje możliwość, że zostanie zgłoszony głębszy wyjątek, w zależności od ustawionej właściwości indywidualnej.

- Jeśli zasób został żądany przez odwołanie do zasobu dynamicznego w języku XAML i nie został znaleziony, zachowanie zależy od ogólnego systemu właściwości. Ogólne zachowanie jest takie, jakby nie wystąpiła żadna operacja ustawienia właściwości na poziomie, na którym istnieje zasób. Jeśli na przykład próbujesz ustawić tło na pojedynczym elemencie przycisku przy użyciu zasobu, którego nie można ocenić, nie ma żadnych wyników zestawu wartości, ale efektywna wartość nadal może pochodzić od innych uczestników systemu właściwości i pierwszeństwa wartości. Na przykład wartość tła może nadal pochodzić ze stylu przycisku zdefiniowanego lokalnie lub stylu motywu. W przypadku właściwości, które nie są zdefiniowane przez style motywu, efektywna wartość po nieudanej ocenie zasobu może pochodzić z wartości domyślnej w metadanych właściwości.

Ograniczenia

Odwołania do zasobów dynamicznych mają pewne istotne ograniczenia. Co najmniej jeden z następujących warunków musi być spełniony:

- Właściwość ustawiana musi być właściwością na obiekcie FrameworkElement lub FrameworkContentElement. Ta właściwość musi być wspierana przez DependencyPropertyelement .
- Odwołanie dotyczy wartości w elemencie `StyleSetter`.
- Ustawiana właściwość musi być właściwością podaną Freezable jako wartość FrameworkElement właściwości lub FrameworkContentElement lub Setter wartości.

Ponieważ ustawiana właściwość musi być właściwością DependencyProperty lub Freezable , większość zmian właściwości może być propagowana do interfejsu użytkownika, ponieważ zmiana właściwości (zmieniona wartość zasobu dynamicznego) jest potwierdzana przez system właściwości. Większość kontrolek obejmuje logikę, która wymusza inny układ kontrolki, jeśli DependencyProperty zmiany i ta właściwość może mieć wpływ na układ. Jednak nie wszystkie właściwości, które mają rozszerzenie znaczników DynamicResource , ponieważ ich wartość jest gwarantowana w celu zapewnienia aktualizacji w czasie rzeczywistym w interfejsie użytkownika. Ta funkcja nadal może się różnić w zależności od właściwości i w zależności od typu, który jest właścicielem właściwości, a nawet logicznej struktury aplikacji.

Style, DataTemplates i niejawne klucze

Chociaż wszystkie elementy w elemecie ResourceDictionary muszą mieć klucz, nie oznacza to, że wszystkie zasoby muszą mieć jawny `x:Key`element . Kilka typów obiektów

obsługuje niejawny klucz zdefiniowany jako zasób, gdzie wartość klucza jest powiązana z wartością innej właściwości. Ten typ klucza jest znany jako klucz niejawny, a `x:Key` atrybut jest jawnym kluczem. Możesz zastąpić dowolny niejawny klucz, określając jawny klucz.

Jednym z ważnych scenariuszy dla zasobów jest zdefiniowanie .Style W rzeczywistości element Style jest prawie zawsze zdefiniowany jako wpis w słowniku zasobów, ponieważ style są z natury przeznaczone do ponownego użycia. Aby uzyskać więcej informacji na temat stylów, zobacz Style i szablony (WPF .NET).

Style kontrolek można tworzyć przy użyciu klucza niejawnego i przywoływały do niejawnego klucza. Style motywu definiujące domyślny wygląd kontrolki opierają się na tym niejawnym kluczu. Z punktu widzenia żądania niejawnego klucza jest Type sam formant. Z punktu widzenia definiowania zasobów niejawny klucz jest TargetType stylem. W związku z tym, jeśli tworzysz motywy dla kontrolek niestandardowych lub tworzysz style, które współdziałają z istniejącymi stylami motywu, nie musisz określać dyrektywy x:Key dla tego Styleelementu . A jeśli chcesz używać stylów motywów, nie musisz w ogóle określać żadnego stylu. Na przykład poniższa definicja stylu działa, mimo że Style zasób nie wydaje się mieć klucza:

XAML

```
<Style TargetType="Button">
    <Setter Property="Background" Value="#4E1A3D" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="BorderThickness" Value="5" />
    <Setter Property="BorderBrush">
        <Setter.Value>
            <LinearGradientBrush>
                <GradientStop Offset="0.0" Color="#4E1A3D"/>
                <GradientStop Offset="1.0" Color="Salmon"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

Ten styl naprawdę ma klucz: niejawny klucz: `System.Windows.Controls.Button` typ. W znacznikach można określić TargetType bezpośrednio nazwę typu (lub opcjonalnie użyć elementu {x:Type...} , aby zwrócić Typeelement .

Za pomocą domyślnych mechanizmów stylu motywu używanych przez WPF ten styl jest stosowany jako styl Button środowiska uruchomieniowego na stronie, mimo że Button sam nie próbuje określić jego Style właściwości ani określonego odwołania do zasobu do stylu. Styl zdefiniowany na stronie znajduje się wcześniej w sekwencji odnośników niż styl słownika motywu, używając tego samego klucza, który ma styl słownika motywu.

Możesz po prostu określić `<Button>Hello</Button>` dowolne miejsce na stronie, a styl zdefiniowany za pomocą TargetType`Button` polecenia będzie miał zastosowanie do tego przycisku. Jeśli chcesz, możesz nadal jawnie klucz stylu z taką samą wartością typu, jak TargetType w przypadku jasności w adiustacji, ale jest to opcjonalne.

Niejawne klucze stylów nie mają zastosowania do kontrolki , jeśli OverridesDefaultStyle jest to `true`. (Należy również pamiętać, że OverridesDefaultStyle można ustawić jako część natywnego zachowania klasy sterującej, a nie jawnie w wystąpieniu kontrolki). Ponadto, aby obsługiwać niejawne klucze dla scenariuszy klasy pochodnej, kontrolka musi zastąpić DefaultStyleKey (wszystkie istniejące kontrolki dostarczone w ramach WPF obejmują to zastąpienie). Aby uzyskać więcej informacji na temat stylów, motywów i projektowania kontrolek, zobacz Wytyczne dotyczące projektowania kontrolek stylowalnych.

DataTemplate ma również klucz niejawny. Klucz niejawny dla elementu DataTemplate to DataType wartość właściwości. DataType można również określić jako nazwę typu, a nie jawnie przy użyciu elementu {x:Type...}. Aby uzyskać szczegółowe informacje, zobacz Omówienie tworzenia szablonów danych.

Zobacz też

- Zasoby w kodzie
- Scalone słowniki zasobów
- Jak zdefiniować i odwołać się do zasobu WPF
- Jak używać zasobów systemowych
- Jak używać zasobów aplikacji
- x:Type markup extension (Rozszerzenie znaczników x:Type)
- ResourceDictionary
- Zasoby aplikacji
- Definiowanie i odwołowanie się do zasobu
- Omówienie zarządzania aplikacjami
- Rozszerzenie znaczników StaticResource
- Rozszerzenie znaczników DynamicResource

Scalone słowniki zasobów (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 4 min

zasoby Windows Presentation Foundation (WPF) obsługują funkcję scalonego słownika zasobów. Ta funkcja umożliwia zdefiniowanie części zasobów aplikacji WPF poza skompilowanymi aplikacjami XAML. Zasoby można następnie udostępniać między aplikacjami i są również wygodniej odizolowane pod kątem lokalizacji.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Tworzenie scalonego słownika

W adiustacji użyj następującej składni, aby wprowadzić scalony słownik zasobów do strony:

XAML

```
<Page.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="myresourcedictionary.xaml"/>
      <ResourceDictionary Source="myresourcedictionary2.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Page.Resources>
```

Element ResourceDictionary nie ma dyrektywy x:Key, która jest zwykle wymagana dla wszystkich elementów w kolekcji zasobów. Jednak inne `ResourceDictionary` odwołanie w MergedDictionaries kolekcji jest specjalnym przypadkiem zarezerwowanym dla tego scalonego scenariusza słownika zasobów. Ponadto wprowadzenie `ResourceDictionary` scalonego słownika zasobów nie może mieć dyrektywy x:Key.

Zazwyczaj każdy ResourceDictionary element w MergedDictionaries kolekcji określa Source atrybut. Wartość `Source` powinna być jednolitym identyfikatorem zasobu (URI), który jest rozpoznawany jako lokalizacja pliku zasobów do scalenia. Miejsce docelowe tego identyfikatora URI musi być innym plikiem XAML z elementem `ResourceDictionary` głównym.

> ⓘ **Uwaga**
>
> Zgodnie z prawem można zdefiniować zasoby w obrębie **ResourceDictionary** określonego jako scalony słownik, albo jako alternatywę dla określenia **Source**, lub oprócz zasobów uwzględnionych w określonym źródle. Nie jest to jednak typowy scenariusz. Głównym scenariuszem scalania słowników jest scalanie zasobów z lokalizacji plików zewnętrznych. Jeśli chcesz określić zasoby w znacznikach dla strony, zdefiniuj je w głównym `ResourceDictionary` , a nie w scalonych słownikach.

Zachowanie scalonego słownika

Zasoby w scalanym słowniku zajmują lokalizację w zakresie wyszukiwania zasobów, który znajduje się tuż po zakresie głównego słownika zasobów, z którymi są scalane. Mimo że klucz zasobu musi być unikatowy w dowolnym słowniku, klucz może istnieć wiele razy w zestawie scalonych słowników. W takim przypadku zwrócony zasób będzie pochodzić z ostatniego słownika znalezionego sekwencyjnie w MergedDictionaries kolekcji. MergedDictionaries Jeśli kolekcja została zdefiniowana w języku XAML, kolejność scalonych słowników w kolekcji jest kolejnością elementów podanych w znacznikach. Jeśli klucz jest zdefiniowany w słowniku podstawowym, a także w słowniku, który został scalony, zasób zwrócony będzie pochodzić ze słownika podstawowego. Te reguły określania zakresu mają jednakowe zastosowanie zarówno w przypadku odwołań do zasobów statycznych, jak i odwołań do zasobów dynamicznych.

Scalone słowniki i kod

Scalane słowniki można dodawać do słownika `Resources` za pomocą kodu. Wartość domyślna, początkowo pusta ResourceDictionary , która istnieje dla dowolnej `Resources` właściwości, ma również domyślną, początkowo pustą MergedDictionaries właściwość kolekcji. Aby dodać scalony słownik za pomocą kodu, należy uzyskać odwołanie do żądanego podstawowego `ResourceDictionary`elementu , uzyskać jego `MergedDictionaries` wartość właściwości i wywołać `Add` metodę ogólną `Collection` zawartą w `MergedDictionaries`elemencie . Dodany obiekt musi być nowym `ResourceDictionary`obiektem .

W kodzie nie ustawiasz Source właściwości . Zamiast tego należy uzyskać `ResourceDictionary` obiekt, tworząc go lub ładując go. Jednym ze sposobów załadowania istniejącego `ResourceDictionary` elementu w celu wywołania XamlReader.Load istniejącego strumienia plików XAML, który ma katalog główny, a

następnie rzutowanie wartości zwracanej `ResourceDictionary` na `ResourceDictionary`wartość .

Scalone identyfikatory URI słownika

Istnieje kilka technik dołączania scalonego słownika zasobów, które są wskazywane przez używany format identyfikatora URI. Ogólnie rzecz biorąc, te techniki można podzielić na dwie kategorie: zasoby kompilowane w ramach projektu i zasoby, które nie są kompilowane w ramach projektu.

W przypadku zasobów kompilowanych w ramach projektu można użyć ścieżki względnej odwołującej się do lokalizacji zasobu. Ścieżka względna jest oceniana podczas kompilacji. Zasób musi być zdefiniowany jako część projektu jako akcja kompilacji **zasobu** . Jeśli w projekcie dołączysz plik *xaml* zasobu jako **zasób**, nie musisz kopiować pliku zasobu do katalogu wyjściowego, zasób jest już uwzględniony w skompilowanej aplikacji. Można również użyć akcji Kompilacja **zawartości** , ale należy skopiować pliki do katalogu wyjściowego, a także wdrożyć pliki zasobów w tej samej relacji ścieżki do pliku wykonywalnego.

> ⓘ **Uwaga**
>
> Nie używaj akcji kompilacji **zasób osadzony** . Sama akcja kompilacji jest obsługiwana w przypadku aplikacji WPF, ale rozdzielczość **Source** nie obejmuje **ResourceManager**elementu , a tym samym nie może oddzielić pojedynczego zasobu od strumienia. Nadal można użyć **zasobu osadzonego** do innych celów, o ile również użyto **ResourceManager** dostępu do zasobów.

Powiązaną techniką jest użycie **identyfikatora URI pakietu** do pliku XAML i odwoływanie się do niego jako **źródło**. **Identyfikator URI pakietu** umożliwia odwoływanie się do składników zestawów z odwołaniami i innych technik. Aby uzyskać więcej informacji na temat **identyfikatorów URI pakietów**, zobacz Zasoby aplikacji WPF, Zawartość i Pliki danych.

W przypadku zasobów, które nie są kompilowane w ramach projektu, identyfikator URI jest oceniany w czasie wykonywania. Aby odwołać się do pliku zasobu, możesz użyć typowego transportu identyfikatora URI, takiego jak *plik:* lub *http:* . Wadą korzystania z niekompilowanego podejścia do zasobów jest to, że *plik:* dostęp wymaga dodatkowych kroków wdrażania, a *http:* dostęp oznacza strefę zabezpieczeń Internetu.

Ponowne uzyskiwanie scalonych słowników

Można ponownie użyć lub udostępnić scalone słowniki zasobów między aplikacjami, ponieważ słownik zasobów do scalenia może być przywoływał za pomocą dowolnego prawidłowego identyfikatora URI. Dokładnie to, jak to zrobić, zależy od strategii wdrażania aplikacji i modelu aplikacji, który obserwujesz. Wcześniej wymieniona strategia **identyfikatora URI pakietu** umożliwia powszechnie źródło scalonego zasobu w wielu projektach podczas opracowywania przez udostępnienie odwołania do zestawu. W tym scenariuszu zasoby są nadal dystrybuowane przez klienta, a co najmniej jedna z aplikacji musi wdrożyć przywoływanego zestawu. Można również odwoływać się do scalonych zasobów za pomocą rozproszonego identyfikatora URI, który używa protokołu *http:* .

Pisanie scalonych słowników jako plików aplikacji lokalnych lub lokalnego magazynu udostępnionego jest innym możliwym scenariuszem scalania słownika i wdrażania aplikacji.

Lokalizacja

Jeśli zasoby, które muszą być zlokalizowane, są izolowane do słowników, które są scalane z podstawowymi słownikami i przechowywane jako luźne XAML, te pliki mogą być zlokalizowane oddzielnie. Ta technika jest uproszczoną alternatywą dla lokalizowania zestawów zasobów satelitarnych. Aby uzyskać szczegółowe informacje, zobacz Omówienie globalizacji i lokalizacji WPF.

Zobacz też

- ResourceDictionary
- Omówienie zasobów XAML
- Zasoby w kodzie
- Zasoby aplikacji WPF, zawartość, pliki danych
- Przegląd Lokalizacja i globalizacja WPF

Zasoby w kodzie (WPF .NET)

Artykuł • 08.02.2023 • Czas czytania: 3 min

To omówienie koncentruje się na tym, jak można Windows Presentation Foundation zasobów (WPF) lub tworzyć je przy użyciu kodu, a nie składni XAML. Aby uzyskać więcej informacji na temat ogólnego użycia zasobów i zasobów z perspektywy składni JĘZYKA XAML, zobacz Overview of XAML resources (Omówienie zasobów XAML).

Uzyskiwanie dostępu do zasobów z kodu

Klucze identyfikujące zasoby zdefiniowane w języku XAML są również używane do pobierania określonych zasobów w przypadku żądania zasobu w kodzie. Najprostszym sposobem pobrania zasobu z kodu FindResource jest wywołanie metody lub TryFindResource z obiektów na poziomie struktury w aplikacji. Różnica behawioralna między tymi metodami polega na tym, co się dzieje, jeśli żądany klucz nie zostanie znaleziony. FindResource zgłasza wyjątek. TryFindResource Funkcja nie zgłasza wyjątku, ale zwraca wartość `null`. Każda metoda przyjmuje klucz zasobu jako parametr wejściowy i zwraca obiekt o luźnym typie.

Zazwyczaj klucz zasobu jest ciągiem, ale czasami nie są używane ciągi. Logika wyszukiwania do rozpoznawania zasobów kodu jest taka sama jak przypadek XAML odwołania do zasobu dynamicznego. Wyszukiwanie zasobów rozpoczyna się od elementu wywołującego, a następnie jest kontynuowane przez elementy nadrzędne w drzewie logicznym. W razie potrzeby odnośnik będzie kontynuowany do zasobów aplikacji, motywów i zasobów systemowych. Żądanie kodu dla zasobu prawidłowo uwzględnia zmiany w tych zasobach, które miały miejsce w czasie wykonywania.

Poniższy przykład kodu przedstawia program obsługi Click zdarzeń, który znajduje zasób według klucza i używa zwróconej wartości do ustawienia właściwości.

C#

```
private void myButton_Click(object sender, RoutedEventArgs e)
{
    Button button = (Button)sender;
    button.Background = (Brush)this.FindResource("RainbowBrush");
}
```

Alternatywną metodą przypisywania odwołania do zasobu jest SetResourceReference. Ta metoda przyjmuje dwa parametry: klucz zasobu i identyfikator określonej właściwości zależności, która znajduje się w wystąpieniu elementu, do którego należy przypisać

wartość zasobu. Funkcjonalnie ta metoda jest taka sama i ma tę zaletę, że nie wymaga rzutowania wartości zwracanych.

Innym sposobem programowego uzyskiwania dostępu do zasobów jest uzyskanie dostępu do zawartości Resources właściwości jako słownika. Słowniki zasobów służą do dodawania nowych zasobów do istniejących kolekcji, sprawdzania, czy nazwa danego klucza jest już używana przez kolekcję i innych operacji. Jeśli piszesz aplikację WPF w całości w kodzie, możesz również utworzyć całą kolekcję w kodzie i przypisać do niego zasoby. Kolekcję można następnie przypisać do Resources właściwości elementu. Opisano to w następnej sekcji.

Indeks można indeksować w ramach dowolnej kolekcji Resources przy użyciu określonego klucza jako indeksu. Zasoby, do których uzyskuje się dostęp w ten sposób, nie są zgodne ze zwykłymi regułami środowiska uruchomieniowego rozpoznawania zasobów. Uzyskujesz dostęp tylko do tej konkretnej kolekcji. Wyszukiwania zasobów nie przechodzi przez zakres zasobów do katalogu głównego lub aplikacji, jeśli nie znaleziono prawidłowego obiektu w żądanym kluczu. Jednak takie podejście może mieć w niektórych przypadkach zalety wydajności, ponieważ zakres wyszukiwania klucza jest bardziej ograniczony. Aby uzyskać więcej informacji na temat pracy bezpośrednio ze słownikiem zasobów, zobacz ResourceDictionary klasę .

Tworzenie zasobów za pomocą kodu

Jeśli chcesz utworzyć całą aplikację WPF w kodzie, możesz również utworzyć w kodzie wszystkie zasoby w tej aplikacji. Aby to osiągnąć, utwórz nowe wystąpienieResourceDictionary, a następnie dodaj wszystkie zasoby do słownika przy użyciu kolejnych wywołań do .ResourceDictionary.Add Następnie przypisz utworzony element , ResourceDictionary aby Resources ustawić właściwość dla elementu, który znajduje się w zakresie strony, lub elementu Application.Resources. Można również zachować jako obiekt ResourceDictionary autonomiczny bez dodawania go do elementu. Jeśli jednak to zrobisz, musisz uzyskać dostęp do zasobów w nim według klucza elementu, tak jakby był to słownik ogólny. Element ResourceDictionary , który nie jest dołączony do właściwości elementu `Resources` , nie istnieje jako część drzewa elementów FindResource i nie ma zakresu w sekwencji wyszukiwania, która może być używana przez metody i powiązane.

Używanie obiektów jako kluczy

Większość użycia zasobów ustawi klucz zasobu jako ciąg. Jednak różne funkcje WPF celowo używają typu obiektu jako klucza zamiast ciągu. Możliwość obsługi klucza

zasobu przez typ obiektu jest używana przez styl WPF i obsługę ich obsługi. Style i motywy, które staną Type się domyślnymi elementami kontrolek bez stylu, są kluczami kontrolki, do których mają zastosowanie.

Kluczowanie według typu zapewnia niezawodny mechanizm wyszukiwania, który działa w przypadku domyślnych wystąpień każdego typu formantu. Typ może być wykrywany przez odbicie i używany do klas pochodnych stylów, mimo że typ pochodny w przeciwnym razie nie ma stylu domyślnego. Klucz dla zasobu zdefiniowanego Type w języku XAML można określić przy użyciu rozszerzenia znaczników x:Type. Podobne rozszerzenia istnieją dla innych nieciągłego użycia kluczy, które obsługują funkcje WPF, takie jak ComponentResourceKey, rozszerzenie znaczników.

Aby uzyskać więcej informacji, zobacz Style, DataTemplates i klucze niejawne.

Zobacz też

- Omówienie zasobów XAML
- How to define and reference a WPF resource (Jak zdefiniować zasób WPF i odwoływać się do tego zasobu)
- Jak używać zasobów systemowych
- Jak używać zasobów aplikacji

Jak zdefiniować i odwołać się do zasobu WPF (WPF .NET)

Artykuł • 10.02.2023 • Czas czytania: 4 min

W tym przykładzie pokazano, jak zdefiniować zasób i odwołać się do niego. Zasób można odwoływać się za pomocą języka XAML lub kodu.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Przykład XAML

W poniższym przykładzie zdefiniowano dwa typy zasobów: SolidColorBrush zasób i kilka Style zasobów.

XAML

```
<Window.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
    <Style TargetType="Border">
        <Setter Property="Background" Value="#4E1A3D" />
        <Setter Property="BorderThickness" Value="5" />
        <Setter Property="BorderBrush">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0.0" Color="#4E1A3D"/>
                    <GradientStop Offset="1.0" Color="Salmon"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
    <Style TargetType="TextBlock" x:Key="TitleText">
        <Setter Property="FontSize" Value="18"/>
        <Setter Property="Foreground" Value="#4E87D4"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="Margin" Value="0,10,10,10"/>
    </Style>
    <Style TargetType="TextBlock" x:Key="Label">
        <Setter Property="HorizontalAlignment" Value="Right"/>
        <Setter Property="FontSize" Value="13"/>
        <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
        <Setter Property="FontFamily" Value="Arial"/>
        <Setter Property="FontWeight" Value="Bold"/>
```

```
        <Setter Property="Margin" Value="0,3,10,0"/>
    </Style>
</Window.Resources>
```

Zasoby

Zasób SolidColorBrush`MyBrush` służy do podania wartości kilku właściwości, które każdy z nich ma Brush wartość typu. Ten zasób jest przywołyyny za `x:Key` pomocą wartości.

XAML

```
<Border>
    <StackPanel>
        <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
        <TextBlock Style="{StaticResource Label}">Label</TextBlock>
        <TextBlock HorizontalAlignment="Right" FontSize="36" Foreground="
{StaticResource MyBrush}" Text="Text" Margin="20" />
        <Button HorizontalAlignment="Left" Height="30" Background="
{StaticResource MyBrush}" Margin="40">Button</Button>
        <Ellipse HorizontalAlignment="Center" Width="100" Height="100"
Fill="{StaticResource MyBrush}" Margin="10" />
    </StackPanel>
</Border>
```

W poprzednim przykładzie `MyBrush` zasób jest uzyskiwany za pomocą rozszerzenia StaticResource Markup. Zasób jest przypisywany do właściwości, która może akceptować typ zdefiniowanego zasobu. W tym przypadku Backgroundwłaściwości , Foregroundi Fill .

Wszystkie zasoby w słowniku zasobów muszą podać klucz. Gdy style są jednak zdefiniowane, mogą pominąć klucz, jak wyjaśniono w następnej sekcji.

Zasoby są również żądane przez zamówienie znalezione w słowniku, jeśli używasz rozszerzenia StaticResource Markup , aby odwołać się do nich z innego zasobu. Upewnij się, że dowolny zasób, do którego odwołujesz się, jest zdefiniowany we wcześniejszej kolekcji niż żądany zasób. Aby uzyskać więcej informacji, zobacz Zasoby statyczne.

W razie potrzeby można obejść ścisłą kolejność tworzenia odwołań do zasobów przy użyciu rozszerzenia DynamicResource Markup , aby odwołać się do zasobu w czasie wykonywania, ale należy pamiętać, że ta `DynamicResource` technika ma konsekwencje dotyczące wydajności. Aby uzyskać więcej informacji, zobacz Zasoby dynamiczne.

Zasoby stylu

Poniższy przykład odwołuje się do stylów niejawnie i jawnie:

XAML

```
<Border>
    <StackPanel>
        <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
        <TextBlock Style="{StaticResource Label}">Label</TextBlock>
        <TextBlock HorizontalAlignment="Right" FontSize="36" Foreground="
{StaticResource MyBrush}" Text="Text" Margin="20" />
        <Button HorizontalAlignment="Left" Height="30" Background="
{StaticResource MyBrush}" Margin="40">Button</Button>
        <Ellipse HorizontalAlignment="Center" Width="100" Height="100"
Fill="{StaticResource MyBrush}" Margin="10" />
    </StackPanel>
</Border>
```

W poprzednim przykładzie Style kodu zasoby `TitleText` i `Label`, każdy element docelowy określonego typu kontrolki. W tym przypadku oba są przeznaczone dla elementu TextBlock. Style ustawiają różne właściwości kontrolek docelowych, gdy zasób tego stylu jest przywołyny przez jego klucz zasobu dla Style właściwości.

Styl jednak przeznaczony dla kontrolki Border nie definiuje klucza. Po pominięciu klucza typ obiektu objętego TargetType właściwością jest niejawnie używany jako klucz stylu. Gdy styl jest kluczem do typu, staje się domyślnym stylem dla wszystkich kontrolek tego typu, o ile te kontrolki znajdują się w zakresie stylu. Aby uzyskać więcej informacji, zobacz Style, DataTemplates i niejawne klucze.

Przykłady kodu

Poniższe fragmenty kodu przedstawiają tworzenie i ustawianie zasobów za pomocą kodu

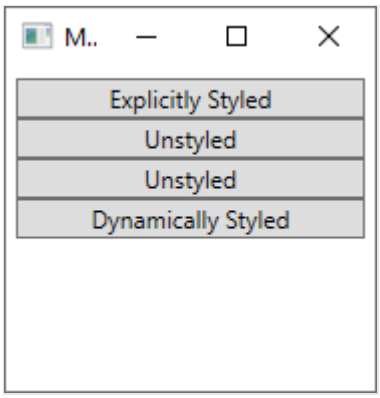
Tworzenie zasobu stylu

Tworzenie zasobu i przypisywanie go do słownika zasobów może nastąpić w dowolnym momencie. Jednak tylko elementy XAML korzystające ze składni DynamicResource zostaną automatycznie zaktualizowane przy użyciu zasobu po jego utworzeniu.

Weźmy na przykład następujące okno. Ma cztery przyciski. Czwarty przycisk używa elementu DynamicResource do stylu. Jednak ten zasób jeszcze nie istnieje, więc wygląda jak normalny przycisk:

XAML

```
<StackPanel Margin="5">
    <Button Click="Button_Click">Explicitly Styled</Button>
    <Button>Unstyled</Button>
    <Button>Unstyled</Button>
    <Button Style="{DynamicResource ResourceKey=buttonStyle1}">Dynamically
Styled</Button>
</StackPanel>
```



Następujący kod jest wywoływany po kliknięciu pierwszego przycisku i wykonaniu następujących zadań:

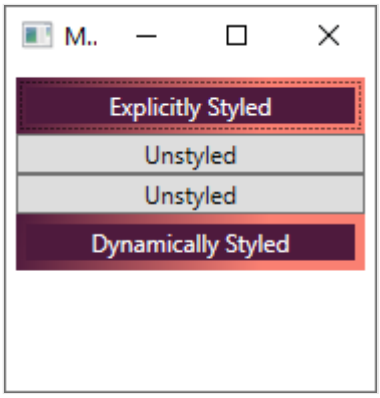
- Tworzy kilka kolorów w celu łatwego odwołania.
- Tworzy nowy styl.
- Przypisuje zestawery do stylu.
- Dodaje styl jako zasób o nazwie `buttonStyle1` do słownika zasobów okna.
- Przypisuje styl bezpośrednio do przycisku wywołującego `Click` zdarzenie.

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Create colors
    Color purple = (Color)ColorConverter.ConvertFromString("#4E1A3D");
    Color white = Colors.White;
    Color salmon = Colors.Salmon;

    // Create a new style for a button
    var buttonStyle = new Style(typeof(Button));

    // Set the properties of the style
    buttonStyle.Setters.Add(new Setter(Control.BackgroundProperty, new
SolidColorBrush(purple)));
    buttonStyle.Setters.Add(new Setter(Control.ForegroundProperty, new
SolidColorBrush(white)));
    buttonStyle.Setters.Add(new Setter(Control.BorderBrushProperty, new
LinearGradientBrush(purple, salmon, 45d)));
    buttonStyle.Setters.Add(new Setter(Control.BorderThicknessProperty, new
Thickness(5)));
```

```
    // Set this style as a resource. Any DynamicResource tied to this key
will be updated.
    this.Resources["buttonStyle1"] = buttonStyle;

    // Set this style directly to a button
    ((Button)sender).Style = buttonStyle;
}
```

Po uruchomieniu kodu zostanie zaktualizowane okno:



Zwróć uwagę, że styl przycisku na początku został zaktualizowany. Styl został zastosowany automatycznie, ponieważ przycisk użył rozszerzenia DynamicResource Markup , aby odwołać się do stylu, który jeszcze nie istnieje. Po utworzeniu stylu i dodaniu do zasobów okna zastosowano go do przycisku. Aby uzyskać więcej informacji, zobacz Zasoby dynamiczne.

Znajdowanie zasobu

Poniższy kod przechodzi przez drzewo logiczne obiektu XAML, w którym jest uruchamiany, aby znaleźć określony zasób. Zasób może być zdefiniowany na samym obiekcie, jest nadrzędny, przez cały sposób do katalogu głównego, samej aplikacji. Poniższy kod wyszukuje zasób, zaczynając od samego przycisku:

C#

```
myButton.Style = myButton.TryFindResource("buttonStyle1") as Style;
```

Jawne odwołanie do zasobu

Jeśli masz odwołanie do zasobu, wyszukując go lub tworząc go, można go przypisać bezpośrednio do właściwości:

C#

```
// Set this style as a resource. Any DynamicResource tied to this key will
be updated.
this.Resources["buttonStyle1"] = buttonStyle;
```

Zobacz też

- Omówienie zasobów XAML
- Style i szablony
- Jak używać zasobów systemowych
- Jak używać zasobów aplikacji

Jak używać zasobów aplikacji (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 2 min

W tym przykładzie pokazano, jak używać zasobów zdefiniowanych przez aplikację. Zasoby można definiować na poziomie aplikacji, zazwyczaj za pomocą pliku *App.xaml* lub *Application.xaml* , niezależnie od tego, którego używa projekt. Zasoby zdefiniowane przez aplikację są globalnie ograniczone i dostępne we wszystkich częściach aplikacji.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Przykład

W poniższym przykładzie pokazano plik definicji aplikacji. Plik definicji aplikacji definiuje sekcję zasobu (wartość właściwości Resources). Zasoby zdefiniowane na poziomie aplikacji mogą być dostępne dla wszystkich innych stron, które są częścią aplikacji. W tym przypadku zasób jest zadeklarowanym stylem. Ponieważ pełny styl zawierający szablon kontrolki może być długi, w tym przykładzie pominięto szablon kontrolki zdefiniowany w ContentTemplate zestawie właściwości stylu.

XAML

```
<Application.Resources>
    <Style TargetType="Border" x:Key="FancyBorder">
        <Setter Property="Background" Value="#4E1A3D" />
        <Setter Property="BorderThickness" Value="5" />
        <Setter Property="BorderBrush">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0.0" Color="#4E1A3D"/>
                    <GradientStop Offset="1.0" Color="Salmon"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Application.Resources>
```

Poniższy przykład przedstawia stronę XAML, która odwołuje się do zasobu na poziomie aplikacji z poprzedniego przykładu. Do zasobu odwołuje się rozszerzenie znaczników

StaticResource , które określa unikatowy klucz zasobu. Zasób "FancyBorder" nie znajduje się w zakresie bieżącego obiektu i okna, więc wyszukiwanie zasobów jest kontynuowane poza bieżącą stroną i do zasobów na poziomie aplikacji.

XAML

```
<Border Style="{StaticResource FancyBorder}">
    <StackPanel Margin="5">
        <Button>Button 1</Button>
        <Button>Button 2</Button>
        <Button>Button 3</Button>
        <Button>Button 4</Button>
    </StackPanel>
</Border>
```

Zobacz też

- Omówienie zasobów XAML
- Zasoby w kodzie
- How to define and reference a WPF resource (Jak definiować i odwoływać się do zasobu WPF)
- Jak używać zasobów systemowych

Jak używać zasobów systemowych (WPF .NET)

Artykuł • 15.02.2023 • Czas czytania: 3 min

W tym przykładzie pokazano, jak używać zasobów zdefiniowanych przez system. Zasoby systemowe są udostępniane przez WPF i umożliwiają dostęp do zasobów systemu operacyjnego, takich jak czcionki, kolory i ikony. Zasoby systemowe uwidaczniają kilka wartości zdefiniowanych przez system jako zarówno zasobów, jak i właściwości, aby ułatwić tworzenie wizualizacji spójnych z systemem Windows.

> **Ważne**
>
> Dokumentacja przewodnika po pulpicie dla platform .NET 7 i .NET 6 jest w trakcie budowy.

Czcionki

Użyj klasy , SystemFonts aby odwołać się do czcionek używanych przez system operacyjny. Ta klasa zawiera wartości czcionek systemowych jako właściwości statyczne i właściwości odwołujące się do kluczy zasobów, które mogą być używane do uzyskiwania dostępu do tych wartości dynamicznie w czasie wykonywania. Na przykład CaptionFontFamily jest wartością SystemFonts i CaptionFontFamilyKey jest odpowiednim kluczem zasobu.

W poniższym przykładzie pokazano, jak uzyskać dostęp do właściwości SystemFonts jako wartości statycznych i używać ich do stylu lub dostosować blok tekstowy:

XAML

```
<TextBlock FontSize="{x:Static SystemFonts.SmallCaptionFontSize}"
           FontWeight="{x:Static SystemFonts.SmallCaptionFontWeight}"
           FontFamily="{x:Static SystemFonts.SmallCaptionFontFamily}"
           Text="Small Caption Font">
</TextBlock>
```

Aby użyć wartości SystemFonts w kodzie, nie musisz używać wartości statycznej ani odwołania do zasobu dynamicznego. Zamiast tego należy użyć właściwości innych niż klucz klasy SystemFonts . Chociaż właściwości inne niż klucz są najwyraźniej zdefiniowane jako właściwości statyczne, zachowanie środowiska WPF jako hostowanego przez system będzie ponownie oceniać właściwości w czasie rzeczywistym

i będzie prawidłowo uwzględniać zmiany oparte na użytkownikach na wartościach systemowych. W poniższym przykładzie pokazano, jak określić ustawienia czcionki przycisku:

C#

```
var myButton = new Button()
{
    Content = "SystemFonts",
    Background = SystemColors.ControlDarkDarkBrush,
    FontSize = SystemFonts.IconFontSize,
    FontWeight = SystemFonts.MessageFontWeight,
    FontFamily = SystemFonts.CaptionFontFamily
};

mainStackPanel.Children.Add(myButton);
```

Czcionki dynamiczne w języku XAML

Metryki czcionek systemowych mogą być używane jako zasoby statyczne lub dynamiczne. Użyj zasobu dynamicznego, jeśli chcesz, aby metryka czcionki aktualizowała się automatycznie podczas uruchamiania aplikacji; w przeciwnym razie użyj zasobu statycznego.

> ⓘ **Uwaga**
>
> Zasoby dynamiczne mają słowo kluczowe `Key` dołączone do nazwy właściwości.

W poniższym przykładzie pokazano, jak uzyskać dostęp do zasobów dynamicznych czcionek systemowych i używać ich do stylu lub dostosować blok tekstowy:

XAML

```
<TextBlock FontSize="{DynamicResource {x:Static
SystemFonts.SmallCaptionFontSize}}"
           FontWeight="{DynamicResource {x:Static
SystemFonts.SmallCaptionFontWeight}}"
           FontFamily="{DynamicResource {x:Static
SystemFonts.SmallCaptionFontFamily}}"
           Text="Small Caption Font">
</TextBlock>
```

Parametry

Użyj klasy , SystemParameters aby odwołać się do właściwości na poziomie systemu, takich jak rozmiar ekranu podstawowego. Ta klasa zawiera zarówno właściwości wartości parametru systemowego, jak i klucze zasobów powiązane z wartościami. Na przykład FullPrimaryScreenHeight jest wartością SystemParameters właściwości i FullPrimaryScreenHeightKey jest odpowiednim kluczem zasobu.

W poniższym przykładzie pokazano, jak uzyskiwać dostęp do wartości statycznych SystemParameters i używać ich do stylu lub dostosować przycisk. Ten przykład znaczników określa rozmiar przycisku, stosując SystemParameters wartości do przycisku:

XAML

```
<Button FontSize="8"
        Height="{x:Static SystemParameters.CaptionHeight}"
        Width="{x:Static SystemParameters.IconGridWidth}"
        Content="System Parameters">
</Button>
```

Aby użyć wartości SystemParameters w kodzie, nie musisz używać odwołań statycznych ani odwołań do zasobów dynamicznych. Zamiast tego użyj wartości SystemParameters klasy . Chociaż właściwości inne niż klucz są najwyraźniej zdefiniowane jako właściwości statyczne, zachowanie środowiska WPF jako hostowanego przez system będzie ponownie oceniać właściwości w czasie rzeczywistym i będzie prawidłowo uwzględniać zmiany oparte na użytkownikach na wartościach systemowych. W poniższym przykładzie pokazano, jak ustawić szerokość i wysokość przycisku przy użyciu SystemParameters wartości:

C#

```
var myButton = new Button()
{
    Content = "SystemParameters",
    FontSize = 8,
    Background = SystemColors.ControlDarkDarkBrush,
    Height = SystemParameters.CaptionHeight,
    Width = SystemParameters.CaptionWidth,
};

mainStackPanel.Children.Add(myButton);
```

Parametry dynamiczne w języku XAML

Metryki parametrów systemowych mogą być używane jako zasoby statyczne lub dynamiczne. Użyj zasobu dynamicznego, jeśli chcesz, aby metryka parametru

aktualizowała się automatycznie podczas uruchamiania aplikacji; w przeciwnym razie użyj zasobu statycznego.

> ⓘ **Uwaga**
>
> Zasoby dynamiczne mają słowo kluczowe `Key` dołączone do nazwy właściwości.

W poniższym przykładzie pokazano, jak uzyskać dostęp do zasobów dynamicznych parametrów systemowych i używać ich do stylu lub dostosowywania przycisku. Ten przykład XAML rozmiaruje przycisk, przypisując SystemParameters wartości do szerokości i wysokości przycisku.

XAML

```
<Button FontSize="8"
        Height="{DynamicResource {x:Static
SystemParameters.CaptionHeightKey}}"
        Width="{DynamicResource {x:Static
SystemParameters.IconGridWidthKey}}"
        Content="System Parameters">
</Button>
```

Zobacz też

- Omówienie zasobów XAML
- Zasoby w kodzie
- Jak zdefiniować i odwołać się do zasobu WPF
- Jak używać zasobów aplikacji

Omówienie języka XAML (WPF .NET)

Artykuł • 16.03.2023 • Czas czytania: 24 min

W tym artykule opisano funkcje języka XAML i pokazano, jak można użyć języka XAML do pisania aplikacji platformy Windows Presentation Foundation (WPF). W tym artykule opisano w szczególności sposób implementacji języka XAML na platformie WPF. Sama koncepcja języka XAML wykracza poza platformę WPF.

> ⓘ **Ważne**
>
> Dokumentacja przewodnika dla komputerów stacjonarnych dla platform .NET 7 i .NET 6 jest w budowie.

Co to jest XAML

XAML to deklaratywny język znaczników. Po zastosowaniu do modelu programowania .NET język XAML upraszcza tworzenie interfejsu użytkownika dla aplikacji .NET. Można utworzyć widoczne elementy interfejsu użytkownika przy użyciu deklaratywnych znaczników XAML, a następnie oddzielić definicję interfejsu użytkownika od logiki czasu wykonywania za pomocą plików codebehind dołączonych do znaczników za pośrednictwem definicji klas częściowych. W języku XAML bezpośrednio reprezentowane jest tworzenie wystąpień obiektów w określonym zbiorze typów zapasowych zdefiniowanych w zestawach. To wyróżnia go spośród większości innych języków znaczników, które stanowią zazwyczaj języki interpretowane bez takiego bezpośredniego powiązania z systemem typów zapasowych. Język XAML umożliwia utworzenie przepływu pracy, w którym różne osoby mogą oddzielnie pracować nad interfejsem użytkownika i logiką aplikacji, potencjalnie przy użyciu różnych narzędzi.

W postaci tekstowej pliki języka XAML są plikami XML, na ogół mającymi rozszerzenie `.xaml`. Te pliki mogą być kodowane przy użyciu dowolnego kodowania XML, ale typowe jest kodowanie UTF-8.

W poniższym przykładzie pokazano, jak można utworzyć przycisk w ramach interfejsu użytkownika. Ten przykład ma na celu przedstawienie sposobu, w jaki typowe metafory programowania interfejsu użytkownika są reprezentowane w języku XAML (nie jest to kompletny kod przykładowy).

```
XAML
```

```
<StackPanel>
    <Button Content="Click Me"/>
</StackPanel>
```

Składnia języka XAML w skrócie

W poniższych sekcjach opisano podstawowe formy składni języka XAML i pokazano krótki przykład użycia znaczników. Ich celem nie jest przekazanie pełnych informacji na temat poszczególnych form składni, na przykład tego, jak są one reprezentowane w systemie typów zapasowych. Aby uzyskać więcej informacji na temat specyfiki składni języka XAML, zobacz Szczegółowe informacje o składni języka XAML.

Jeśli masz już doświadczenie z językiem XML, znaczna część materiału w kilku kolejnych sekcjach wyda Ci się znajoma. Wynika to z jednej z podstawowych zasad projektu języka XAML. Język XAML definiuje własne pojęcia, ale działają one w ramach języka XML i formatu znaczników.

Elementy obiektów XAML

Element obiektu zazwyczaj deklaruje wystąpienie typu. Ten typ jest definiowany w zestawach, do których odwołuje się technologia wykorzystująca XAML jako język.

Składnia elementu obiektu zawsze zaczyna się od otwierającego nawiasu ostrego (`<`). Następuje po nim nazwa typu, którego wystąpienie chcesz utworzyć. Nazwa może zawierać prefiks (ta koncepcja zostanie wyjaśniona później). Następnie można opcjonalnie zadeklarować atrybuty elementu obiektu. Tag elementu obiektu kończy się zamykającym nawiasem ostrym (`>`). Zamiast tego można też użyć formatu samozamykającego bez zawartości, kończąc tag ukośnikiem i zamykającym nawiasem ostrym (`/>`). Przyjrzyj się jeszcze raz przykładowemu fragmentowi kodu powyżej.

XAML

```
<StackPanel>
    <Button Content="Click Me"/>
</StackPanel>
```

Określono w nim dwa elementy obiektów: `<StackPanel>` (z zawartością i tagiem zamykającym w dalszej części) oraz `<Button .../>` (w formacie samozamykającym, z kilkoma atrybutami). Elementy obiektów `StackPanel` i `Button` są mapowane na nazwę klasy zdefiniowaną przez platformę WPF i należącą do zestawów platformy WPF. Określając tag elementu obiektu, tworzysz instrukcję przetwarzania języka XAML w celu

utworzenia nowego wystąpienia typu bazowego. Każde wystąpienie tworzone jest przez wywołanie konstruktora bez parametrów dla danego typu podczas analizowania i ładowania kodu XAML.

Składnia atrybutów (właściwości)

Właściwości obiektu można często wyrazić jako atrybuty elementu obiektu. Składnia atrybutów obejmuje nazwę ustawianej właściwości obiektu i operator przypisania (=). Wartość atrybutu zawsze określa się jako ciąg zawarty w cudzysłowie.

Składnia atrybutów to najbardziej zoptymalizowana składnia ustawiania właściwości, a także najbardziej intuicyjna składnia dla deweloperów, którzy wcześniej używali już języków znaczników. Na przykład poniższy kod tworzy przycisk z czerwonym tekstem na niebieskim tle, na którym jest wyświetlany tekst z właściwości `Content`.

XAML

```
<Button Background="Blue" Foreground="Red" Content="This is a button"/>
```

Składnia elementów właściwości

W przypadku niektórych właściwości elementu obiektu nie można zastosować składni atrybutów, ponieważ obiektów lub informacji wymaganych do określenia wartości właściwości nie można odpowiednio wyrazić z zachowaniem ograniczeń dotyczących ciągów i cudzysłowów mających zastosowanie w składni atrybutów. W takich przypadkach można użyć innej składni, nazywanej składnią elementów właściwości.

Składnia tagu początkowego elementu właściwości to `<TypeName.PropertyName>`. Zawartość takiego tagu to na ogół element obiektu o typie, który dana właściwość przyjmuje jako wartość. Po określeniu zawartości należy zamknąć element właściwości tagiem końcowym. Składnia tagu końcowego to `</TypeName.PropertyName>`.

Jeśli zastosowanie składni atrybutów jest możliwe, jest to zazwyczaj wygodniejsze i umożliwia pisanie bardziej kompaktowego kodu, ale często jest to tylko kwestia stylu, a nie ograniczenie techniczne. W poniższym przykładzie pokazano te same właściwości, które zostały ustawione w poprzednim przykładzie składni atrybutów, ale tym razem dla wszystkich właściwości obiektu `Button` zastosowano składnię elementów właściwości.

XAML

```
<Button>
    <Button.Background>
```

```
        <SolidColorBrush Color="Blue"/>
    </Button.Background>
    <Button.Foreground>
        <SolidColorBrush Color="Red"/>
    </Button.Foreground>
    <Button.Content>
        This is a button
    </Button.Content>
</Button>
```

Składnia kolekcji

Język XAML zawiera pewne optymalizacje, umożliwiające generowanie znaczników łatwiejszych do odczytania przez człowieka. Jedna z takich optymalizacji polega na tym, że jeśli określona właściwość przyjmuje typ kolekcji, elementy zadeklarowane w kodzie jako elementy podrzędne tej właściwości stają się częścią kolekcji. W takim przypadku wartością ustawioną dla właściwości kolekcji jest kolekcja elementów obiektów podrzędnych.

W poniższym przykładzie pokazano składnię kolekcji używaną do ustawienia wartości właściwości GradientStops.

XAML

```
<LinearGradientBrush>
    <LinearGradientBrush.GradientStops>
        <!-- no explicit new GradientStopCollection, parser knows how to
find or create -->
        <GradientStop Offset="0.0" Color="Red" />
        <GradientStop Offset="1.0" Color="Blue" />
    </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

Właściwości zawartości XAML

Język XAML ma funkcję, dzięki której klasa może określać dokładnie jedną ze swoich właściwości jako właściwość *zawartości* języka XAML. Elementy podrzędne tego elementu obiektu służą do ustawiania wartości, jaką ma ta właściwość zawartości. Innymi słowy, wyłącznie w przypadku właściwości zawartości można pominąć element właściwości podczas ustawiania tej właściwości w kodzie XAML, aby metafora obiektu nadrzędnego i podrzędnego w kodzie była bardziej widoczna.

Na przykład klasa Border jako właściwość *zawartości* określa właściwość Child. Dwa następujące elementy Border są traktowane identycznie. Pierwszy z nich korzysta ze

składni właściwości zawartości i pomija element właściwości `Border.Child`. W drugim jawnie pokazano właściwość `Border.Child`.

XAML

```
<Border>
    <TextBox Width="300"/>
</Border>
<!--explicit equivalent-->
<Border>
    <Border.Child>
        <TextBox Width="300"/>
    </Border.Child>
</Border>
```

Zgodnie z regułami języka XAML wartość właściwości zawartości XAML należy podać w całości przed wszystkimi innymi elementami właściwości w tym elemencie obiektu lub w całości po nich. Na przykład następujący kod nie da się skompilować.

XAML

```
<Button>I am a
  <Button.Background>Blue</Button.Background>
  blue button</Button>
```

Aby uzyskać więcej informacji na temat specyfiki składni języka XAML, zobacz Szczegółowe informacje o składni języka XAML.

Zawartość tekstowa

W kilku elementach XAML można bezpośrednio przetwarzać tekst jako zawartość. Aby było to możliwe, musi być spełniony jeden z następujących warunków:

- Klasa musi deklarować właściwość zawartości o typie, który można przypisać do ciągu (może to być typ Object). Na przykład dowolny element ContentControl używa właściwości zawartości Content, której typ to Object, co umożliwia następujące zastosowanie w przypadku elementu ContentControl takiego jak Button: `<Button>Hello</Button>`.

- Ten typ musi deklarować konwerter typów, a w takim przypadku zawartość tekstowa jest używana jako tekst inicjowania dla tego konwertera typów. Na przykład kod `<Brush>Blue</Brush>` konwertuje wartość zawartości `Blue` na pędzel. Takie przypadki są mniej powszechne w praktyce.

- Typ musi być znanym typem pierwotnym języka XAML.

Połączenie składni kolekcji i właściwości zawartości

Rozważ następujący przykład.

XAML

```
<StackPanel>
    <Button>First Button</Button>
    <Button>Second Button</Button>
</StackPanel>
```

Tutaj każdy element Button jest elementem podrzędnym elementu StackPanel. Jest to zoptymalizowany i intuicyjny kod, w którym pominięto dwa tagi z dwóch różnych przyczyn.

- **Pominięto element właściwości StackPanel.Children:** element StackPanel pochodzi od elementu Panel. Element Panel definiuje właściwość Panel.Children jako swoją właściwość zawartości XAML.

- **Pominięto element obiektu UIElementCollection:** właściwość Panel.Children przyjmuje typ UIElementCollection, który implementuje kolekcję IList. Tag elementu kolekcji można pominąć ze względu na reguły przetwarzania kolekcji XAML, takich jak IList. (W tym przypadku nie można tak naprawdę utworzyć wystąpienia kolekcji UIElementCollection, ponieważ nie uwidacznia ona konstruktora bez parametrów, dlatego element obiektu UIElementCollection jest w tym przykładzie oznaczony jako komentarz).

XAML

```
<StackPanel>
    <StackPanel.Children>
        <!--<UIElementCollection>-->
        <Button>First Button</Button>
        <Button>Second Button</Button>
        <!--</UIElementCollection>-->
    </StackPanel.Children>
</StackPanel>
```

Składnia atrybutów (zdarzenia)

Składni atrybutów można również używać w przypadku składowych, które są zdarzeniami, a nie właściwościami. W takim przypadku nazwa atrybutu jest nazwą zdarzenia. W implementacji zdarzeń XAML na platformie WPF wartość atrybutu jest nazwą procedury obsługi, która implementuje delegata tego zdarzenia. Na przykład w

poniższym kodzie przypisano procedurę obsługi zdarzenia Click do obiektu Button utworzonego przy użyciu znacznika:

```
XAML

<Button Click="Button_Click" >Click Me!</Button>
```

Zdarzenia i kod XAML na platformie WPF to zagadnienie znacznie szersze niż ten przykład składni atrybutu. Być może zastanawiasz się na przykład, co reprezentuje i jak jest definiowany element `ClickHandler` wymieniony tutaj. Zostanie to wyjaśnione w sekcji Zdarzenia i pliki codebehind języka XAML w dalszej części tego artykułu.

Wielkość liter i białe znaki w języku XAML

Język XAML z zasady uwzględnia wielkość liter. Na potrzeby rozpoznawania typów zapasowych w języku XAML na platformie WPF wielkość liter jest uwzględniana na tych samych zasadach, co w środowisku uruchomieniowym CLR. Elementy obiektów, elementy właściwości i nazwy atrybutów należy określać z uwzględnieniem wielkości liter, gdy ich nazwy są porównywane z bazowym typem w zestawie lub ze składową typu. Wielkość liter jest też uwzględniana w słowach kluczowych i typach pierwotnych języka XAML. W wartościach nie zawsze jest uwzględniana wielkość liter. Uwzględnianie wielkości liter w wartościach zależy od działania konwertera typów skojarzonego z właściwością przyjmującą daną wartość lub z typem wartości tej właściwości. Na przykład wartości przyjmujące typ Boolean mogą przyjmować wartości `true` i `True` jako równoważne, ale tylko dlatego, że natywna konwersja typów analizatora języka XAML platformy WPF, z ciągu na typ Boolean, dopuszcza je jako równoważne.

Procesory i serializatory języka XAML dla platformy WPF będą ignorować lub usuwać wszystkie nieistotne białe znaki i normalizować wszelkie istotne białe znaki. Jest to zgodne z zaleceniami dotyczącymi domyślnego działania białych znaków w specyfikacji języka XAML. To zachowanie ma znaczenie tylko w przypadku określania ciągów we właściwościach zawartości XAML. Mówiąc najprościej, język XAML konwertuje znaki spacji, nowego wiersza i tabulatora na spacje, a następnie zachowuje jedną spację, jeśli zostanie znaleziona na początku lub na końcu nieprzerwanego ciągu. Ten artykuł nie zawiera pełnego wyjaśnienia obsługi białych znaków w języku XAML. Aby uzyskać więcej informacji, zobacz Przetwarzanie białych znaków w języku XAML.

Rozszerzenia struktury znaczników

Rozszerzenia struktury znaczników to koncepcja języka XAML. Użycie nawiasów klamrowych (`{` i `}`) podczas podawania wartości w składni atrybutów wskazuje na zastosowanie rozszerzenia struktury znaczników. To użycie powoduje, że przetwarzanie kodu XAML ucieka od ogólnego traktowania wartości atrybutów albo jako ciągów literału albo jako wartości konwertowalnych na ciąg.

Najczęściej używane rozszerzenia struktury znaczników w przypadku tworzenia aplikacji WPF to Binding, używane w wyrażeniach powiązań danych, oraz odwołania do zasobów StaticResource i DynamicResource. Korzystając z rozszerzeń struktury znaczników, można wprowadzać wartości właściwości przy użyciu składni atrybutów, nawet jeśli dana właściwość generalnie nie obsługuje składni atrybutów. Rozszerzenia struktury znaczników często korzystają z typów wyrażeń pośrednich do włączania funkcji takich jak opóźnianie wartości lub odwoływanie się do innych obiektów, które są obecne tylko w czasie wykonywania.

Na przykład następujący kod ustawia wartość właściwości Style przy użyciu składni atrybutów. Właściwość Style przyjmuje wystąpienie klasy Style, którego domyślnie nie można utworzyć przy użyciu ciągu składni atrybutów. Jednak w tym przypadku ten atrybut odwołuje się do określonego rozszerzenia struktury znaczników, `StaticResource`. Gdy to rozszerzenie struktury znaczników jest przetwarzane, zwraca odwołanie do stylu, którego wystąpienie utworzono wcześniej jako zasób z kluczem w słowniku zasobów.

XAML

```
<Window x:Class="index.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Window1" Height="100" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
        <Style TargetType="Border" x:Key="PageBackground">
            <Setter Property="BorderBrush" Value="Blue"/>
            <Setter Property="BorderThickness" Value="5" />
        </Style>
    </Window.Resources>
    <Border Style="{StaticResource PageBackground}">
        <StackPanel>
            <TextBlock Text="Hello" />
        </StackPanel>
    </Border>
</Window>
```

Aby zapoznać się z listą referencyjną wszystkich rozszerzeń struktury znaczników dla języka XAML zaimplementowanych na platformie WPF, zobacz Rozszerzenia języka XAML na platformie WPF. Aby zapoznać się z listą referencyjną rozszerzeń znaczników zdefiniowanych przez system.Xaml i są szerzej dostępne dla implementacji XAML

platformy .NET, zobacz XAML Namespace (x:) Funkcje językowe. Aby uzyskać więcej informacji na temat pojęć dotyczących rozszerzeń struktury znaczników, zobacz Rozszerzenia struktury znaczników a język XAML na platformie WPF.

Konwertery typów

W sekcji Składnia języka XAML w skrócie podano, że wartość atrybutu musi być możliwa do ustawienia przy użyciu ciągu. Podstawowy, natywny sposób obsługi konwersji ciągów na inne typy obiektów lub wartości pierwotne opiera się na samym typie String oraz na natywnym przetwarzaniu pewnych typów takich jak DateTime i Uri. Jednak wiele typów platformy WPF lub ich składowych rozszerza podstawowe działanie przetwarzania atrybutów w postaci ciągu, dzięki czemu wystąpienia bardziej złożonych typów obiektów również można określić jako ciągi i atrybuty.

Struktura Thickness to przykład typu z konwersją typów na potrzeby użycia w kodzie XAML. Struktura Thickness wskazuje wymiary w zagnieżdżonym prostokącie i jest używana jako wartość właściwości takich jak Margin. Dzięki umieszczeniu konwertera typu w strukturze Thickness wszystkie właściwości używające wartości Thickness są łatwiejsze do określenia w języku XAML, ponieważ można je określić jako atrybuty. W poniższym przykładzie użyto konwersji typu i składni atrybutu, aby podać wartość właściwości Margin:

XAML

```
<Button Margin="10,20,10,30" Content="Click me"/>
```

Powyższy przykład składni atrybutu jest odpowiednikiem poniższej, bardziej rozbudowanej składni przykładowej, gdzie właściwość Margin jest zamiast tego ustawiana za pomocą składni elementu właściwości zawierającej element obiektu Thickness. Cztery kluczowe właściwości obiektu Thickness są ustawione jako atrybuty nowego wystąpienia:

XAML

```
<Button Content="Click me">
    <Button.Margin>
        <Thickness Left="10" Top="20" Right="10" Bottom="30"/>
    </Button.Margin>
</Button>
```

Uwaga

Istnieje również ograniczona liczba obiektów, w których konwersja typu jest jedynym publicznym sposobem ustawienia właściwości na dany typ bez użycia podklasy, ponieważ sam typ nie ma konstruktora bez parametrów. Może to być na przykład **Cursor**.

Aby uzyskać więcej informacji na temat konwersji typów, zobacz Konwertery typów i język XAML.

Elementy główne i przestrzenie nazw

Plik XAML musi mieć tylko jeden element główny, aby był zarówno poprawnie sformułowanym plikiem XML, jak i prawidłowym plikiem XAML. W typowych scenariuszach użycia platformy WPF używa się elementu głównego mającego wyraźne znaczenie w modelu aplikacji WPF (na przykład Window lub Page w przypadku strony, ResourceDictionary w przypadku słownika zewnętrznego lub Application w przypadku definicji aplikacji). Poniższy przykład przedstawia element główny typowego pliku XAML dla strony WPF z elementem głównym Page.

XAML

```
<Page x:Class="index.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Page1">

</Page>
```

Element główny zawiera również atrybuty `xmlns` i `xmlns:x`. Te atrybuty wskazują procesorowi języka XAML, które przestrzenie nazw XAML zawierają definicje typów dla typów zapasowych, do których kod będzie się odwoływać jako do elementów. Atrybut `xmlns` wskazuje konkretnie domyślną przestrzeń nazw XAML. W domyślnej przestrzeni nazw XAML elementy obiektów w kodzie można określać bez prefiksu. W większości scenariuszy dotyczących aplikacji WPF i w prawie wszystkich przykładach podanych w sekcjach zestawu SDK dla platformy WPF domyślna przestrzeń nazw XAML jest mapowana na przestrzeń nazw `http://schemas.microsoft.com/winfx/2006/xaml/presentation` platformy WPF. Atrybut `xmlns:x` wskazuje dodatkową przestrzeń nazw XAML, która mapuje przestrzeń nazw `http://schemas.microsoft.com/winfx/2006/xaml` języka XAML.

To użycie atrybutu `xmlns` w celu zdefiniowania użycia i mapowania zakresu nazw jest zgodne ze specyfikacją języka XML w wersji 1.0. Zakresy nazw XAML różnią się od

zakresów nazw XML tylko tym, że zakres nazw XAML ma też znaczenie dla sposobu, w jaki jego elementy powiązane są z typami podczas rozpoznawania typów i analizowania kodu XAML.

Atrybuty `xmlns` są ściśle niezbędne tylko w przypadku elementu głównego każdego pliku XAML. Definicje atrybutów `xmlns` będą stosowane do wszystkich elementów podrzędnych elementu głównego (to działanie również jest zgodne ze specyfikacją języka XML w wersji 1.0 dla atrybutu `xmlns`). Atrybuty `xmlns` są też dozwolone w przypadku innych elementów znajdujących się poniżej elementu głównego i mają zastosowanie do wszelkich elementów podrzędnych definiującego elementu. Jednak częste definiowanie i ponowne definiowanie przestrzeni nazw XAML może spowodować, że styl kodu XAML będzie mało czytelny.

Implementacja procesora XAML na platformie WPF obejmuje infrastrukturę uwzględniającą podstawowe zestawy platformy WPF. Podstawowe zestawy platformy WPF zawierają typy obsługujące mapowania platformy WPF na domyślną przestrzeń nazw XAML. Jest to możliwe dzięki konfiguracji stanowiącej część pliku kompilacji projektu oraz systemów kompilacji i projektów platformy WPF. Zatem zadeklarowanie domyślnej przestrzeni nazw XAML jako domyślnej wartości atrybutu `xmlns` wystarczy do odwoływania się do elementów języka XAML pochodzących z zestawów platformy WPF.

Prefiks x:

W poprzednim przykładzie elementu głównego użyto prefiksu `x:` do mapowania przestrzeni nazw XAML `http://schemas.microsoft.com/winfx/2006/xaml`, czyli dedykowanej przestrzeni nazw XAML obsługującej konstrukcje języka XAML. Ten prefiks `x:` służy do mapowania tej przestrzeni nazw XAML w szablonach projektów, w przykładach i w dokumentacji w całym tym zestawie SDK. Przestrzeń nazw XAML dla języka XAML zawiera kilka konstrukcji programowania, z których będziesz często korzystać we własnym kodzie XAML. Poniżej znajduje się lista najczęściej używanych konstrukcji programowania z prefiksem `x:`:

- x:Key: ustawia unikatowy klucz dla każdego zasobu w elemencie ResourceDictionary (lub w podobnych koncepcjach słownika w innych strukturach). Konstrukcja `x:Key` prawdopodobnie będzie odpowiadać za 90% użycia prefiksu `x:` w kodzie typowej aplikacji WPF.

- x:Class: określa przestrzeń nazw środowiska uruchomieniowego CLR i nazwę klasy udostępniającej plik codebehind strony XAML. Zgodnie z modelem programowania platformy WPF wymagana jest klasa obsługująca plik codebehind, dlatego niemal zawsze prefiks `x:` będzie mapowany, nawet jeśli nie ma zasobów.

- x:Name: określa nazwę obiektu czasu wykonywania dla wystąpienia istniejącego w kodzie czasu wykonywania po przetworzeniu elementu obiektu. Na ogół będziesz często używać zdefiniowanej na platformie WPF właściwości równoważnej wobec konstrukcji x:Name. Takie właściwości są mapowane konkretnie na właściwość zapasową środowiska uruchomieniowego CLR i dlatego są wygodniejsze w przypadku programowania aplikacji, gdy często używa się kodu czasu wykonywania do znalezienia nazwanych elementów z zainicjowanego kodu XAML. Najbardziej typową właściwością tego rodzaju jest właściwość FrameworkElement.Name. Nadal można używać konstrukcji x:Name, gdy równoważna właściwość WPF Name na poziomie struktury nie jest obsługiwana w określonym typie. Jest tak w niektórych scenariuszach animacji.

- x:Static: umożliwia użycie odwołania zwracającego wartość statyczną, która w innym przypadku nie jest właściwością zgodną z językiem XAML.

- x:Type: tworzy odwołanie Type na podstawie nazwy typu. Służy do określania atrybutów przyjmujących wartość Type, takich jak Style.TargetType, chociaż często właściwości mają natywną konwersję z ciągu na wartość Type, w związku z czym zastosowanie rozszerzenia struktury znaczników x:Type jest opcjonalne.

Istnieją dodatkowe konstrukcje programowania korzystające z prefiksu `x:`/przestrzeni nazw XAML, które nie są tak powszechne. Aby uzyskać szczegółowe informacje, zobacz Przestrzeń nazw XAML (x:) — funkcje języka.

Prefiksy i typy niestandardowe

W przypadku własnych zestawów niestandardowych lub zestawów innych niż podstawowe zestawy **PresentationCore**, **PresentationFramework** i **WindowsBase** platformy WPF można określić zestaw w ramach niestandardowego mapowania `xmlns`. Następnie można odwoływać się do typów z tego zestawu we własnym kodzie XAML, o ile dany typ jest poprawnie zaimplementowany na potrzeby obsługi pożądanych zastosowań języka XAML.

Poniżej przedstawiono podstawowy przykład działania niestandardowych prefiksów w kodzie XAML. Prefiks `custom` jest zdefiniowany w tagu elementu głównego i zamapowany na określony zestaw, który jest spakowany z aplikacją i dostępny dla niej. Ten zestaw zawiera typ `NumericUpDown`, zaimplementowany w celu obsługi ogólnego użycia języka XAML, z użyciem dziedziczenia klasy, które umożliwia wstawienie go w tym konkretnym punkcie w modelu zawartości XAML na platformie WPF. Wystąpienie tej kontrolki `NumericUpDown` jest deklarowane jako element obiektu przy użyciu prefiksu, tak

aby analizator XAML wiedział, która przestrzeń nazw XAML zawiera odpowiedni typ i gdzie znajduje się zestaw zapasowy zawierający definicję typu.

XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-
namespace:NumericUpDownCustomControl;assembly=CustomLibrary"
    >
  <StackPanel Name="LayoutRoot">
    <custom:NumericUpDown Name="numericCtrl1" Width="100" Height="60"/>
...
  </StackPanel>
</Page>
```

Aby uzyskać więcej informacji na temat typów niestandardowych w języku XAML, zobacz XAML i klasy niestandardowe na platformie WPF.

Aby uzyskać więcej informacji na temat związku między przestrzeniami nazw XML a przestrzeniami nazw kodu w zestawach, zobacz Przestrzenie nazw XAML i mapowanie przestrzeni nazw w przypadku języka XAML na platformie WPF.

Zdarzenia i pliki codebehind języka XAML

Większość aplikacji WPF zawiera zarówno kod XAML, jak i pliki codebehind. W projekcie kod XAML jest zapisywany jako plik `.xaml`, a do napisania pliku codebehind używa się języka środowiska uruchomieniowego CLR, takiego jak Microsoft Visual Basic lub C#. Podczas kompilowania znaczników z pliku XAML w ramach modeli programowania i aplikacji WPF lokalizację pliku codebehind języka XAML dla pliku XAML wskazuje się przez określenie przestrzeni nazw i klasy jako atrybutu `x:Class` elementu głównego pliku XAML.

W przykładach podanych do tej pory pokazano kilka przycisków, ale żaden z nich nie miał jeszcze skojarzonego żadnego działania logicznego. Podstawowym mechanizmem dodawania działania do elementu obiektu na poziomie aplikacji jest użycie istniejącego zdarzenia klasy elementu i napisanie specyficznej procedury obsługi tego zdarzenia, wywoływanej w przypadku wystąpienia zdarzenia w czasie wykonywania. Nazwę zdarzenia i nazwę procedury obsługi do użycia określa się w kodzie XAML, a kod wdrażający procedurę obsługi definiuje się w pliku codebehind.

XAML

```
<Page x:Class="ExampleNamespace.ExamplePage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel>
        <Button Click="Button_Click">Click me</Button>
    </StackPanel>
</Page>
```

C#

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace ExampleNamespace;

public partial class ExamplePage : Page
{
    public ExamplePage() =>
        InitializeComponent();

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        var buttonControl = (Button)e.Source;
        buttonControl.Foreground = Brushes.Red;
    }
}
```

Zwróć uwagę, że plik związany z kodem używa przestrzeni `ExampleNamespace` nazw CLR (przestrzeń nazw nie jest widoczna w języku Visual Basic) i deklaruje `ExamplePage` jako częściową klasę w tej przestrzeni nazw. To odpowiada atrybutu `x:Class` określonej jako `ExampleNamespace.ExamplePage` w elemencie głównym znaczników. Kompilator języka znaczników platformy WPF utworzy klasę częściową dla każdego kompilowanego pliku XAML, wyprowadzając klasę z typu elementu głównego. Gdy dostarczysz plik codebehind, który również definiuje tę samą klasę częściową, wynikowy kod jest łączony z tą samą przestrzenią nazw i klasą, co w kompilowanej aplikacji.

> **ⓘ Ważne**
>
> W języku Visual Basic główna przestrzeń nazw jest domniemana zarówno dla kodu XAML, jak i kodu. Widoczne są tylko zagnieżdżone przestrzenie nazw. W tym artykule przedstawiono kod XAML projektu języka C#.

Aby uzyskać więcej informacji na temat wymagań dotyczących programowania z użyciem plików codebehind na platformie WPF, zobacz Wymagania dotyczące plików

codebehind, obsługi zdarzeń i klas częściowych na platformie WPF.

Jeśli nie chcesz tworzyć oddzielnego pliku codebehind, możesz również umieścić kod śródwierszowy w pliku XAML. Jednak użycie kodu śródwierszowego to mniej wszechstronna technika, która ma znaczne ograniczenia. Aby uzyskać więcej informacji, zobacz Pliki codebehind i język XAML na platformie WPF.

Zdarzenia trasowane

Szczególną funkcją zdarzeń o podstawowym znaczeniu dla platformy WPF jest zdarzenie trasowane. Zdarzenia trasowane umożliwiają elementowi obsługę zdarzenia, które zostało zgłoszone przez inny element, o ile te elementy są połączone za pośrednictwem relacji drzewa. W przypadku określenia obsługi zdarzeń przy użyciu atrybutu XAML można nasłuchiwać zdarzenia trasowanego i obsługiwać je w dowolnym elemencie, w tym w elementach, które nie mają tego konkretnego zdarzenia w tabeli składowych klasy. W tym celu atrybut nazwy zdarzenia jest kwalifikowany przy użyciu nazwy klasy, do której należy. Na przykład element nadrzędny `StackPanel` w bieżącym przykładzie dotyczącym obiektów `StackPanel` / `Button` może zarejestrować procedurę obsługi dla zdarzenia Click przycisku elementu podrzędnego, określając atrybut `Button.Click` w elemencie obiektu `StackPanel`, z nazwą procedury obsługi jako wartością atrybutu. Aby uzyskać więcej informacji, zobacz Omówienie zdarzeń trasowanych.

Elementy nazwane

Domyślnie wystąpienie obiektu utworzone na wykresie obiektów w wyniku przetworzenia elementu obiektu XAML nie ma żadnego unikatowego identyfikatora ani odwołania do obiektu. Z kolei w przypadku wywołania konstruktora w kodzie niemal zawsze używa się wyniku konstruktora do ustawienia zmiennej w skonstruowanym wystąpieniu, co umożliwia odwoływanie się do tego wystąpienia w dalszej części kodu. Aby zapewnić ustandaryzowany dostęp do obiektów tworzonych za pośrednictwem definicji znacznika, język XAML definiuje atrybut x:Name. Można ustawić wartość atrybutu `x:Name` w dowolnym elemencie obiektu. W pliku codebehind wybrany identyfikator jest odpowiednikiem zmiennej wystąpienia odwołującej się do skonstruowanego wystąpienia. Pod każdym względem nazwane elementy działają tak, jakby były wystąpieniami obiektów (nazwa odwołuje się do danego wystąpienia), a plik codebehind może odwoływać się do nazwanych elementów w celu obsługi interakcji w czasie wykonywania w aplikacji. To powiązanie między wystąpieniami a zmiennymi zapewnia kompilator znaczników XAML platformy WPF, a mówiąc dokładniej, opiera się ono na funkcjach i wzorcach takich jak InitializeComponent, które nie będą szczegółowo omawiane w tym artykule.

Elementy XAML na poziomie struktury na platformie WPF dziedziczą właściwość Name, która stanowi odpowiednik zdefiniowanego w języku XAML atrybutu `x:Name`. Niektóre inne klasy również zapewniają odpowiedniki na poziomie właściwości dla atrybutu `x:Name`, który definiuje się też zazwyczaj jako właściwość `Name`. Mówiąc ogólnie, jeśli nie można znaleźć właściwości `Name` w tabeli składowych wybranego elementu lub typu, należy użyć zamiast niej atrybutu `x:Name`. Wartości atrybutu `x:Name` tworzą identyfikator elementu XAML, który może być używany w czasie wykonywania albo przez określone podsystemy, albo przez metody narzędzi, takie jak FindName.

W poniższym przykładzie ustawiono właściwość Name dla elementu StackPanel. Następnie procedura obsługi obiektu Button w ramach tego elementu StackPanel odwołuje się do elementu StackPanel za pośrednictwem odwołania do wystąpienia `buttonContainer` ustawionego za pomocą właściwości Name.

XAML

```
<StackPanel Name="buttonContainer">
    <Button Click="RemoveThis_Click">Click to remove this button</Button>
</StackPanel>
```

C#

```
private void RemoveThis_Click(object sender, RoutedEventArgs e)
{
    var element = (FrameworkElement)e.Source;

    if (buttonContainer.Children.Contains(element))
        buttonContainer.Children.Remove(element);
}
```

Podobnie jak zmienne, nazwy wystąpień XAML podlegają koncepcji zakresu, a więc można wymuszać stosowanie unikatowych nazw w określonym, przewidywalnym zakresie. Podstawowe znaczniki definiujące stronę określają jeden unikatowy zakres nazw XAML, a granicą zakresu nazw XAML jest element główny tej strony. Jednak inne źródła znaczników, na przykład style lub szablony w ramach stylów, mogą wchodzić w interakcje ze stroną w czasie wykonywania, a takie źródła znaczników często mają własne zakresy nazw XAML, które niekoniecznie są zgodne z zakresem nazw XAML strony. Aby uzyskać więcej informacji na temat atrybutu `x:Name` i zakresów nazw XAML, zobacz Name, Dyrektywa x:Name lub Zakresy nazw XAML na platformie WPF.

Właściwości dołączone i zdarzenia dołączone

Język XAML ma funkcję umożliwiającą określanie pewnych właściwości lub zdarzeń w dowolnym elemencie, nawet jeśli ta właściwość lub to zdarzenie nie istnieje w definicjach typu dla danego elementu. W odniesieniu do właściwości nazywa się to właściwością dołączoną, a w odniesieniu do zdarzeń — zdarzeniem dołączonym. Można wyobrazić sobie właściwości dołączone i zdarzenia dołączone jako globalne składowe, które można ustawić w dowolnym elemencie/wystąpieniu obiektu XAML. Jednak takie elementy/klasy lub ogólna infrastruktura muszą obsługiwać magazyn właściwości zapasowych dla wartości dołączonych.

Właściwości dołączone w języku XAML są zwykle stosowane przy użyciu składni atrybutów. W składni atrybutów właściwość dołączoną określa się w następującej formie: `ownerType.propertyName`.

Na pierwszy rzut oka przypomina to użycie elementu właściwości, ale w tym przypadku określony typ `ownerType` jest zawsze inny niż typ elementu obiektu, w którym ustawiana jest właściwość dołączona. Typ `ownerType` udostępnia metody dostępu wymagane przez procesor XAML w celu uzyskania lub ustawienia wartości właściwości dołączonej.

Najczęstszym scenariuszem użycia właściwości dołączonych jest umożliwienie elementom podrzędnym raportowania wartości właściwości do elementu nadrzędnego.

Poniższy przykład przedstawia użycie właściwości dołączonej DockPanel.Dock. Klasa DockPanel definiuje metody dostępu do właściwości dołączonej DockPanel.Dock i jest jej właścicielem. Klasa DockPanel zawiera również logikę wykonującą iterację przez jej elementy podrzędne i sprawdza każdy element pod kątem ustawionej wartości właściwości DockPanel.Dock. W przypadku znalezienia wartości jest ona używana do określenia położenia elementów podrzędnych w układzie. Użycie właściwości dołączonej DockPanel.Dock i tej funkcji określania położenia to właśnie główne uzasadnienie wprowadzenia klasy DockPanel.

XAML

```
<DockPanel>
    <Button DockPanel.Dock="Left" Width="100" Height="20">I am on the
left</Button>
    <Button DockPanel.Dock="Right" Width="100" Height="20">I am on the
right</Button>
</DockPanel>
```

Na platformie WPF większość właściwości dołączonych, jeśli nie wszystkie, implementuje się także jako właściwości zależności. Aby uzyskać więcej informacji, zobacz Omówienie właściwości dołączonych.

Zdarzenia dołączone wykorzystują podobną formę składni atrybutów: `ownerType.eventName`. Podobnie jak w przypadku zdarzeń niedołączonych wartość atrybutu zdarzenia dołączonego w kodzie XAML określa nazwę metody procedury obsługi, która jest wywoływana w celu obsługi zdarzenia w tym elemencie. Użycie zdarzeń dołączonych w aplikacjach XAML na platformie WPF jest rzadsze. Aby uzyskać więcej informacji, zobacz Omówienie zdarzeń dołączonych.

Typy podstawowe

Bazowy kod XAML na platformie WPF i jego przestrzeń nazw XAML to kolekcja typów odpowiadających obiektom środowiska uruchomieniowego CLR i elementom znaczników XAML. Jednak nie wszystkie klasy można mapować na elementy. Klasy abstrakcyjne, takie jak ButtonBase, i niektóre nieabstrakcyjne klasy bazowe są używane na potrzeby dziedziczenia w modelu obiektów środowiska uruchomieniowego CLR. Klasy bazowe, w tym abstrakcyjne, pozostają ważne podczas programowania w języku XAML, ponieważ każdy konkretny element XAML dziedziczy składowe z jakiejś klasy bazowej w hierarchii. Tymi składowymi często są właściwości, które można ustawić jako atrybuty w elemencie, lub zdarzenia, które można obsługiwać. FrameworkElement to konkretna klasa bazowa interfejsu użytkownika platformy WPF na poziomie struktury. Projektując interfejs użytkownika, będziesz korzystać z różnych klas kształtów, paneli, dekoratorów lub kontrolek, a wszystkie one pochodzą z klasy FrameworkElement. Powiązana klasa bazowa FrameworkContentElement obsługuje elementy zorientowane na dokumenty, które sprawdzają się w przypadku prezentacji układu przepływu i korzystają z interfejsów API celowo dublujących interfejsy API w klasie FrameworkElement. Kombinacja atrybutów na poziomie elementu i modelu obiektów CLR zapewnia zestaw typowych właściwości, które można ustawić w większości konkretnych elementów XAML, niezależnie od tego, jaki to element XAML i jaki jest jego typ bazowy.

Zabezpieczenia

XAML to język znaczników, w którym bezpośrednio reprezentowane jest tworzenie wystąpień obiektów i ich wykonywanie. Dlatego elementy utworzone w kodzie XAML mają taką samą możliwość interakcji z zasobami systemowymi (na przykład dostępem do sieci czy operacjami we/wy systemu plików), jaką ma kod aplikacji. Kod XAML ma również taki sam dostęp do zasobów systemowych, jak hostująca aplikacja.

Zabezpieczenia dostępu kodu (CAS) na platformie WPF

W odróżnieniu od programu .NET Framework platforma WPF dla platformy .NET nie obsługuje funkcji CAS. Aby uzyskać więcej informacji, zobacz Różnice w zabezpieczeniach dostępu kodu.

Ładowanie znaczników XAML z kodu

Za pomocą języka XAML można zdefiniować cały interfejs użytkownika, ale czasami wskazane jest zdefiniowanie w nim tylko fragmentu interfejsu użytkownika. Za pomocą tej funkcji możliwe jest:

- Częściowe dostosowywanie.
- Lokalne przechowywanie informacji o interfejsie użytkownika.
- Modelowanie obiektów biznesowych.

Kluczem do tych scenariuszy jest klasa XamlReader i zawarta w niej metoda Load. Jako dane wejściowe wprowadza się plik XAML, a wynikiem jest obiekt reprezentujący całe drzewo obiektów czasu wykonywania utworzonych na podstawie znaczników. Ten obiekt można następnie wstawić jako właściwość innego obiektu, który już istnieje w aplikacji. O ile dana właściwość znajduje się w modelu zawartości i ma funkcje wyświetlania, które powiadomią aparat wykonywania o dodaniu nowej zawartości do aplikacji, można łatwo modyfikować zawartość uruchomionej aplikacji dzięki dynamicznemu ładowaniu kodu XAML.

Zobacz też

- Szczegóły składni XAML
- Klasy XAML i niestandardowe dla WPF
- Przestrzeń nazw XAML (x:) Funkcje językowe
- Rozszerzenia WPF XAML
- Przegląd Elementy bazy
- Drzewa w WPF

Usługi XAML

Artykuł • 16.11.2022 • Czas czytania: 7 min

W tym temacie opisano możliwości zestawu technologii nazywanego usługami XAML platformy .NET. Większość opisanych usług i interfejsów API znajduje się w zestawie `System.Xaml`. Usługi obejmują składniki odczytywania i zapisywania, klasy schematów i fabryki obsługi schematów, przypisywanie klas, wewnętrzną obsługę języka XAML i inne funkcje języka XAML.

O tej dokumentacji

Dokumentacja koncepcyjna dla usług XAML platformy .NET zakłada, że masz doświadczenie z językiem XAML i sposobem, w jaki może on być stosowany do określonej platformy, na przykład Windows Presentation Foundation (WPF) lub Windows Workflow Foundation lub określonego obszaru funkcji technologii, na przykład funkcji dostosowywania kompilacji w elemencie Microsoft.Build.Framework.XamlTypes. W tej dokumentacji nie jest podejmowana próba wyjaśnienia podstaw języka XAML jako języka znaczników, terminologii składni XAML ani innego materiału wprowadzającego. Zamiast tego w tej dokumentacji skupiono się konkretnie na korzystaniu z usług XAML platformy .NET, które są włączone w bibliotece zestawów System.Xaml. Większość z tych interfejsów API jest przeznaczona dla scenariuszy rozszerzalności i integracji języka XAML. Może to obejmować dowolny z następujących scenariuszy:

- Rozszerzanie możliwości podstawowych składników odczytywania lub zapisywania XAML (bezpośrednie przetwarzanie strumienia węzłów XAML, wyprowadzanie własnego składnika odczytywania lub zapisywania XAML).
- Definiowanie typów niestandardowych z możliwością używania w języku XAML, które nie mają określonych zależności struktury, oraz przypisywanie typów w celu przekazania ich cech systemowych typu XAML do usług XAML platformy .NET.
- Hostowanie składników odczytywania lub zapisywania XAML jako składnika aplikacji, takiego jak projektant wizualny lub interakcyjny edytor dla źródeł znaczników XAML.
- Pisanie konwerterów wartości XAML (rozszerzenia znaczników, konwertery typów dla typów niestandardowych).
- Definiowanie niestandardowego kontekstu schematu XAML (przy użyciu alternatywnych technik ładowania zestawów dla źródeł typów zapasowych, przy użyciu technik wyszukiwania znanych typów zamiast zawsze odzwierciedlania

zestawów, przy użyciu załadowanych koncepcji zestawów, które nie używają środowiska uruchomieniowego języka wspólnego (CLR) `AppDomain` ani skojarzonego z nim modelu zabezpieczeń).

- Rozszerzanie podstawowego systemu typów XAML.
- Używanie technik `Lookup` lub `Invoker` do wpływania na system typów XAML i sposobu oceniania kopii zapasowych typów.

Jeśli szukasz materiałów wprowadzających dotyczących XAML jako języka, możesz skorzystać z artykułu Omówienie języka XAML (WPF .NET). W tym temacie omówiono język XAML dla odbiorców, którzy są nowymi użytkownikami zarówno platformy Windows Presentation Foundation (WPF), jak i w zakresie używania znaczników XAML i funkcji języka XAML. Inny przydatny dokument to materiał wprowadzający w artykule Specyfikacja języka XAML.

Usługi XAML platformy .NET i zestaw `System.Xaml` w architekturze platformy .NET

Usługi XAML platformy .NET i zestaw `System.Xaml` definiują wiele elementów potrzebnych do obsługi funkcji języka XAML. Obejmuje to klasy bazowe dla składników odczytywania i zapisywania XAML. Najważniejszą funkcją dodaną do usług XAML platformy .NET, która nie była obecna w żadnej implementacji języka XAML specyficznej dla platformy, jest reprezentacja systemu typów dla języka XAML. Reprezentacja systemu typów przedstawia kod XAML w sposób zorientowany obiektowo, który koncentruje się na możliwościach języka XAML bez uwzględniania zależności od określonych możliwości platform.

System typów XAML nie jest ograniczony przez formularz znaczników ani specyfikę czasu wykonywania pochodzenia XAML, nie jest on też ograniczony przez żaden konkretny system typu zapasowego. System typów XAML zawiera reprezentacje obiektów dla typów, składowych, kontekstów schematu XAML, koncepcji na poziomie XML i innych koncepcji języka XAML lub elementów wewnętrznych XAML. Używanie lub rozszerzanie systemu typów XAML umożliwia pozyskiwanie od klas, takich jak składniki odczytywania i zapisywania XAML, oraz rozszerzanie funkcjonalności reprezentacji XAML do określonych funkcji włączonych przez platformę, technologię lub aplikację, która używa kodu XAML lub go emituje. Koncepcja kontekstu schematu XAML umożliwia praktyczne operacje zapisywania wykresu obiektów z połączenia implementacji składnika zapisywania obiektów XAML, systemu typów zapasowych technologii przekazywanego za pośrednictwem informacji o zestawie w kontekście, oraz źródła

węzła XAML. Aby uzyskać więcej informacji na temat koncepcji schematu XAML, zobacz Domyślny kontekst schematu XAML i kontekst schematu XAML platformy WPF.

Strumienie węzłów XAML, składniki odczytywania i zapisywania XAML

Aby zrozumieć rolę odgrywaną przez usługi XAML platformy .NET w relacji między językiem XAML a określonymi technologiami, które używają XAML jako języka, warto zrozumieć koncepcję strumienia węzłów XAML oraz sposób, w jaki ta koncepcja kształtuje interfejs API i terminologię. Strumień węzłów XAML to koncepcyjny pośrednik między reprezentacją języka XAML a wykresem obiektów reprezentowanym lub definiowanym przez kod XAML.

- Składnik odczytywania XAML to jednostka, która przetwarza kod XAML w jakiejś formie i tworzy strumień węzłów XAML. W interfejsie API składnik odczytywania XAML jest reprezentowany przez klasę bazową XamlReader.

- Składnik zapisywania XAML to jednostka, która przetwarza strumień węzłów XAML i tworzy coś innego. W interfejsie API składnik zapisywania XAML jest reprezentowany przez klasę bazową XamlWriter.

  Dwa najbardziej typowe scenariusze związane z XAML to ładowanie kodu XAML w celu utworzenia wystąpienia wykresu obiektów i zapisywanie wykresu obiektów z aplikacji lub narzędzia oraz utworzenie reprezentacji XAML (zazwyczaj w formie znaczników zapisanych jako plik tekstowy). Ładowanie kodu XAML i tworzenie wykresu obiektów jest często określane w tej dokumentacji jako ścieżka ładowania. Zapisywanie lub serializowanie istniejącego wykresu obiektów w języku XAML jest często określane w tej dokumentacji jako ścieżka zapisu.

  Najczęstszy typ ścieżki ładowania można opisać w następujący sposób:

- Zacznij od reprezentacji XAML w formacie XML z kodowaniem UTF i zapisanej jako plik tekstowy.

- Załaduj ten kod XAML do podklasy XamlXmlReader. XamlXmlReader jest podklasą XamlReader.

- Wynikiem jest strumień węzłów XAML. Dostęp do poszczególnych węzłów strumienia węzłów XAML można uzyskać przy użyciu interfejsu API XamlXmlReader / XamlReader. Tutaj najbardziej typową operacją jest przechodzenie przez strumień węzłów XAML i przetwarzanie każdego węzła przy użyciu metafory „bieżący rekord”.

- Przekaż wynikowe węzły ze strumienia węzłów XAML do interfejsu API XamlObjectWriter. XamlObjectWriter jest podklasą XamlWriter.
- Podklasa XamlObjectWriter zapisuje wykres obiektów, po jednym obiekcie naraz, zgodnie z postępem za pośrednictwem źródłowego strumienia węzłów XAML. Zapisywanie obiektów odbywa się z pomocą kontekstu schematu XAML i implementacji, która może uzyskiwać dostęp do zestawów oraz typów systemu typu zapasowego i struktury.
- Wywołaj element Result na końcu strumienia węzłów XAML, aby uzyskać obiekt główny wykresu obiektów.

  Najczęstszy typ ścieżki zapisu można opisać w następujący sposób:

- Zacznij od wykresu obiektów całego czasu wykonywania aplikacji, zawartości interfejsu użytkownika i stanu czasu wykonywania lub mniejszego segmentu ogólnej reprezentacji obiektu aplikacji w czasie wykonywania.
- Z logicznego obiektu początkowego, takiego jak katalog główny aplikacji lub katalog główny dokumentu, załaduj obiekty do podklasy XamlObjectReader. XamlObjectReader jest podklasą XamlReader.
- Wynikiem jest strumień węzłów XAML. Dostęp do poszczególnych węzłów strumienia węzłów XAML można uzyskać przy użyciu interfejsu API XamlObjectReader i XamlReader. Tutaj najbardziej typową operacją jest przechodzenie przez strumień węzłów XAML i przetwarzanie każdego węzła przy użyciu metafory „bieżący rekord”.
- Przekaż wynikowe węzły ze strumienia węzłów XAML do interfejsu API XamlXmlWriter. XamlXmlWriter jest podklasą XamlWriter.
- Składnik XamlXmlWriter zapisuje kod XAML w kodowaniu UTF XML. Można to zapisać jako plik tekstowy, strumień lub w innych formach.
- Wywołaj metodę Flush, aby uzyskać końcowe dane wyjściowe.

Aby uzyskać więcej informacji na temat koncepcji dotyczących strumienia węzłów XAML, zobacz Opis koncepcji i struktur strumienia węzłów XAML.

Klasa XamlServices

Nie zawsze jest konieczne zajmowanie się strumieniem węzłów XAML. Jeśli potrzebujesz podstawowej ścieżki ładowania lub podstawowej ścieżki zapisu, możesz użyć interfejsów API w klasie XamlServices.

- Różne sygnatury metody Load implementują ścieżkę ładowania. Możesz załadować plik lub strumień albo załadować element XmlReader, TextReader lub XamlReader, który opakowuje dane wejściowe XAML, ładując je za pomocą interfejsu API tego składnika odczytywania.
- Różne sygnatury elementu Save zapisują wykres obiektów i generują dane wyjściowe jako strumień, plik lub wystąpienie XmlWriter/TextWriter.
- Transform konwertuje kod XAML, łącząc ścieżkę ładowania i ścieżkę zapisu jako pojedynczą operację. Można użyć innego kontekstu schematu lub innego systemu typów zapasowych dla elementów XamlReader i XamlWriter, co wpływa na sposób przekształcania wynikowego kodu XAML.

Aby uzyskać więcej informacji na temat używania klasy XamlServices, zobacz Klasa XAMLServices i podstawowe odczytywanie lub zapisywanie w języku XAML.

System typów XAML

System typów XAML udostępnia interfejsy API, które są wymagane do pracy z danym pojedynczym węzłem strumienia węzłów XAML.

XamlType to reprezentacja obiektu — tego, co przetwarzasz między początkowym a końcowym węzłem obiektu.

XamlMember to reprezentacja składowej obiektu — tego, co przetwarzasz między początkowym a końcowym węzłem składowej.

Interfejsy API, takie jak GetAllMembers, GetMember i DeclaringType, raportują relacje między elementami XamlType i XamlMember.

Domyślne zachowanie systemu typów XAML implementowane przez usługi XAML platformy .NET jest oparte na środowisku uruchomieniowym języka wspólnego (CLR) i analizie statycznej typów CLR w zestawach przy użyciu odbicia. W związku z tym w przypadku określonego typu CLR domyślna implementacja systemu typów XAML może uwidocznić schemat XAML tego typu i jego składowych oraz raportować go w pojęciach systemu typów XAML. W domyślnym systemie typów XAML koncepcja przypisywania typów jest mapowana na dziedziczenie CLR, a koncepcje wystąpień, typów wartości itd., są również mapowane na obsługujące je zachowania i funkcje środowiska CLR.

Dokumentacja funkcji języka XAML

Aby obsługiwać język XAML, usługi XAML platformy .NET zapewniają konkretną implementację koncepcji języka XAML zgodnie z definicją dla przestrzeni nazw XAML języka XAML. Są one udokumentowane jako konkretne strony referencyjne. Funkcje języka są udokumentowane z perspektywy zachowania tych funkcji języka podczas ich przetwarzania przez czytnik XAML lub składnik zapisywania XAML zdefiniowany przez usługi XAML platformy .NET. Aby uzyskać więcej informacji, zobacz Funkcje języka przestrzeni nazw XAML (x:).