

Hand­s-on lab

Lab 2: Binding to Data

May 2015

Contents

[Overview 3](#_Toc420068811)

[Exercise 1: Classic and Compiled Bindings 4](#_Toc420068812)

[Task 1 – Open the starter project 4](#_Toc420068813)

[Task 2 – Create the model 5](#_Toc420068814)

[Task 3 – Create the view 7](#_Toc420068815)

[Task 4 – Add classic and compiled bindings 8](#_Toc420068816)

[Exercise 2: Binding to the Property of an Element 11](#_Toc420068817)

[Task 1 – Open the starter project 11](#_Toc420068818)

[Task 2 – Create the ElementName binding 12](#_Toc420068819)

[Summary 17](#_Toc420068820)

Overview

The purpose of this lab is to introduce the concept of data binding and demonstrate how it can be implemented. In summary, data binding is the process that establishes a connection between the application UI and business logic. If the binding has the correct settings and the data provides the proper notifications, then, when the data changes its value, the elements that are bound to the data reflect changes automatically. Data binding can also mean that if an outer representation of the data in an element changes, then the underlying data can be automatically updated to reflect the change. For example, if the user edits the value in a **TextBox** element, the underlying data value is automatically updated to reflect that change.

As this lab is being written, the MSDN documentation for Windows 10 is not yet fully available, however the [Data binding overview](https://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh758320.aspx) for Windows 8.1 is still relevant to what we are referring to as Classic Binding. Windows 10 introduces a new style of binding that address a number of issues relating to performance in Classic Binding – we refer to this new binding capability as Compiled Binding. More details regarding the capabilities of Compiled Binding can be viewed in this session from Build 2015: [Data Binding: Boost Your Apps' Performance Through New Enhancements to XAML Data Binding](http://channel9.msdn.com/Events/Build/2015/3-635).

# Objectives

* 1. This lab will show you how to:
  + Bind to data for display in your UI
  + Leverage both classic and compiled bindings
  + Bind to another element

# System requirements

* 1. You must have the following to complete this lab:
  + Microsoft Windows 10
  + Microsoft Visual Studio 2015

# Setup

* 1. You must perform the following steps to prepare your computer for this lab:
  2. Install Microsoft Windows 10.
  3. Install Microsoft Visual Studio 2015.
  4. Install the Universal Windows Platform SDK.

# Exercises

* 1. This Hands-on lab includes the following exercises:
  2. Classic and Compiled Bindings
  3. Binding to the Property of An Element
  4. Estimated time to complete this lab:  **45 to 60 minutes**.

Exercise 1: Classic and Compiled Bindings

In this exercise you will create a simple app that demonstrates how to bind to a list of data. You will leverage both classic and compiled binding to connect the UI to the ViewModel.

1. Task 1 – Open the starter project
   1. The first task is to open the starter project we have created for you.
2. Open the file location where you installed the hands-on labs. Navigate to the **\Lab 2\Solution\Binding** folder and open **Binding.sln** in Visual Studio 2015.
3. To prepare to build and run your app, use the Solution Configurations dropdown to choose the Debug configuration and use the Solution Platforms dropdown to target x86 (Figure 8). To run the app on the local machine, select Local Machine from the drop-down list next to the Start Debugging button on the debugger Standard toolbar.
4. Build and run the app.

**

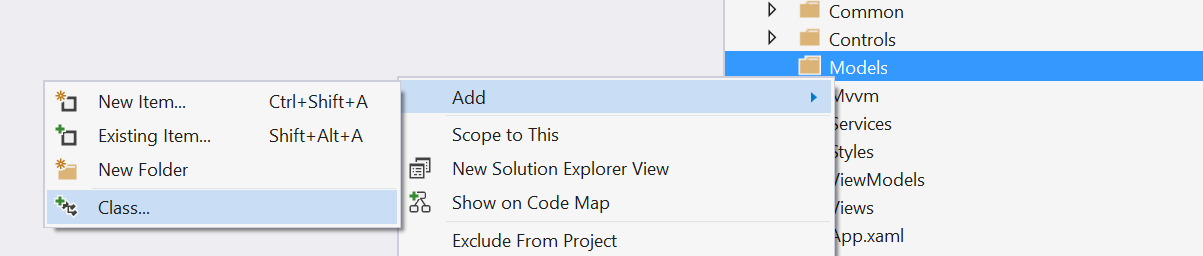
**Figure 1**

*The blank starter app.*

1. Return to Visual Studio and stop debugging.
2. Task 2 – Create the model

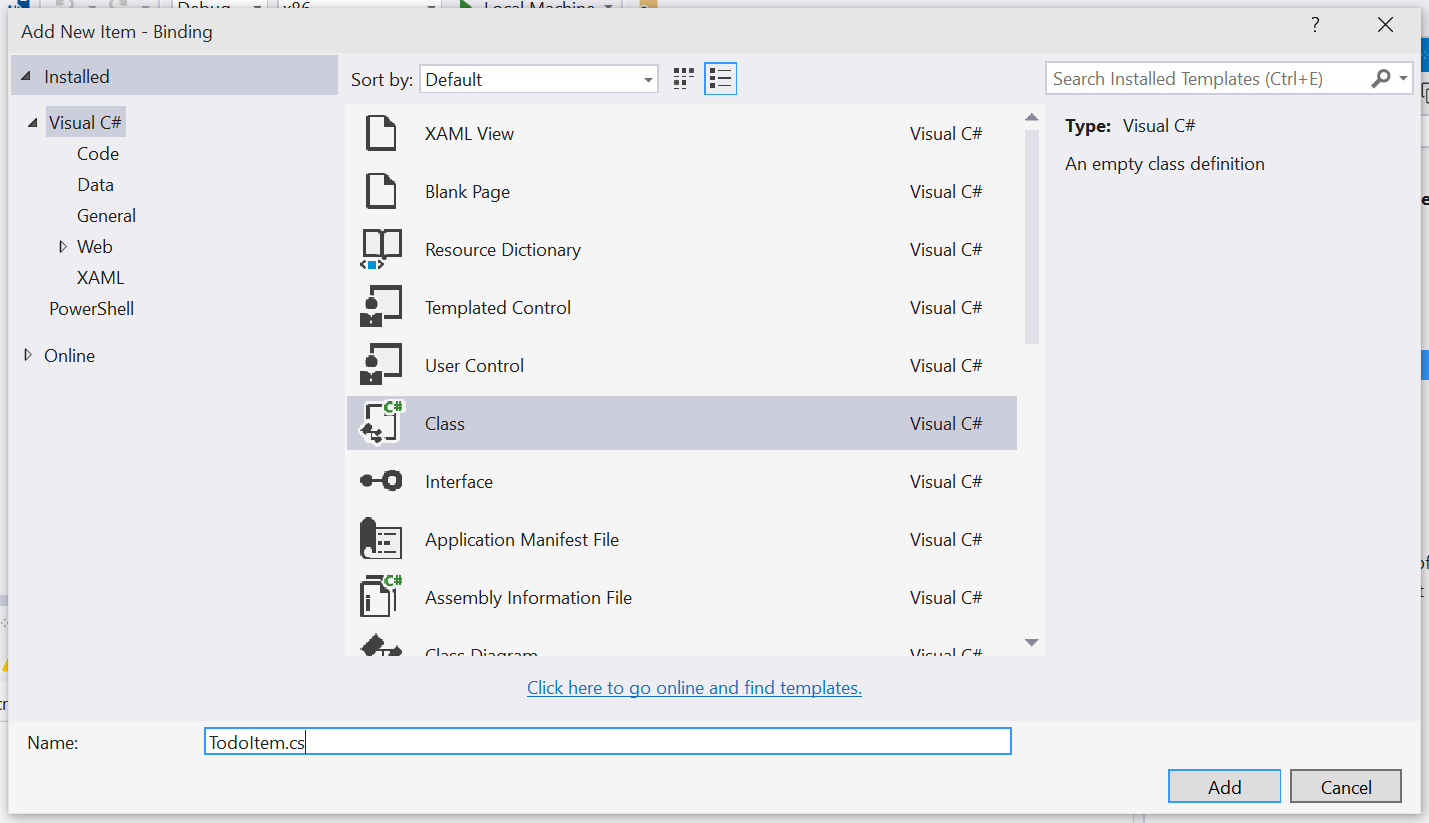
During this task, we will create the model for our app and extend the capabilities of our MainPageViewModel.

1. Right-click on the Models directory in the Solutions Explorer and choose **Add > Class.** Name the class TodoItem.



**Figure 2**

*Add a new class.*



**Figure 3**

*Add the TodoItem class.*

1. Open TodoItem.cs and make the class public. Add a property for the title field.
   * 1. C#

public class TodoItem

{

public string Title { get; set; }

}

1. Open ViewModels > MainPageViewModel.cs. Within the class definition, create an observable collection property that will contain our list of items. Make sure to reference the System.Collections.ObjectModel namespace.
   1. **Note:** We can assign a new instance of the collection to our property as part of the declaration. This is new to C# 6. More details concerning the new features of C# 6 can be seen [here](http://blogs.msdn.com/b/csharpfaq/archive/2014/11/20/new-features-in-c-6.aspx).
      1. C#

using System.Collections.ObjectModel;

namespace Binding.ViewModels

{

public class MainPageViewModel : Mvvm.ViewModelBase

{

public ObservableCollection<Models.TodoItem> Items { get; private set; } = new ObservableCollection<Models.TodoItem>();

}

}

1. Create the constructor and use it to generate some TodoItems. Add those items to the **Items** observable collection.
   * 1. C#

public MainPageViewModel()

{

for (int i = 0; i < 100; i++)

{

var item = new Models.TodoItem

{

Title = string.Format("Task Title {0}", i)

};

this.Items.Add(item);

}

}

1. Task 3 – Create the view

During this task we will create the main view for our app.

1. Open **MainPage.xaml** and reference your project’s ViewModels namespace.
   * 1. XAML
   1. xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   2. xmlns:vm="using:Binding.ViewModels"
   3. mc:Ignorable="d">
   4. **Note:** The **MainPage.xaml** starter file in this exercise derives from the **Blank Page** item type in the **Add New Item** dialog. Documentation for the Blank Page item template is at <http://go.microsoft.com/fwlink/?LinkId=234238>.
2. We will create an instance of our view model – **MainPageViewModel** – and assign it to the page data context using the XAML below.
   * 1. XAML

<Page.DataContext>

<vm:MainPageViewModel />

</Page.DataContext>

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

1. Now let’s add column definitions to the grid.
   * 1. XAML

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

</Grid>

1. Task 4 – Add classic and compiled bindings

During this task we will add two lists to our app and bind them to the data in our **MainPageViewModel**. We will use both classic and compiled bindings.

1. Add a **ListView** to the grid using the items generated in the **MainPageViewModel** constructor. This **ListView** uses classic bindings to bind to the observable collection of TodoItems and their respective titles.
   * 1. XAML

</Grid.ColumnDefinitions>

<ListView ItemsSource="{Binding Items}" Header="Classic" Grid.Column="0">

<ListView.ItemTemplate>

<DataTemplate>

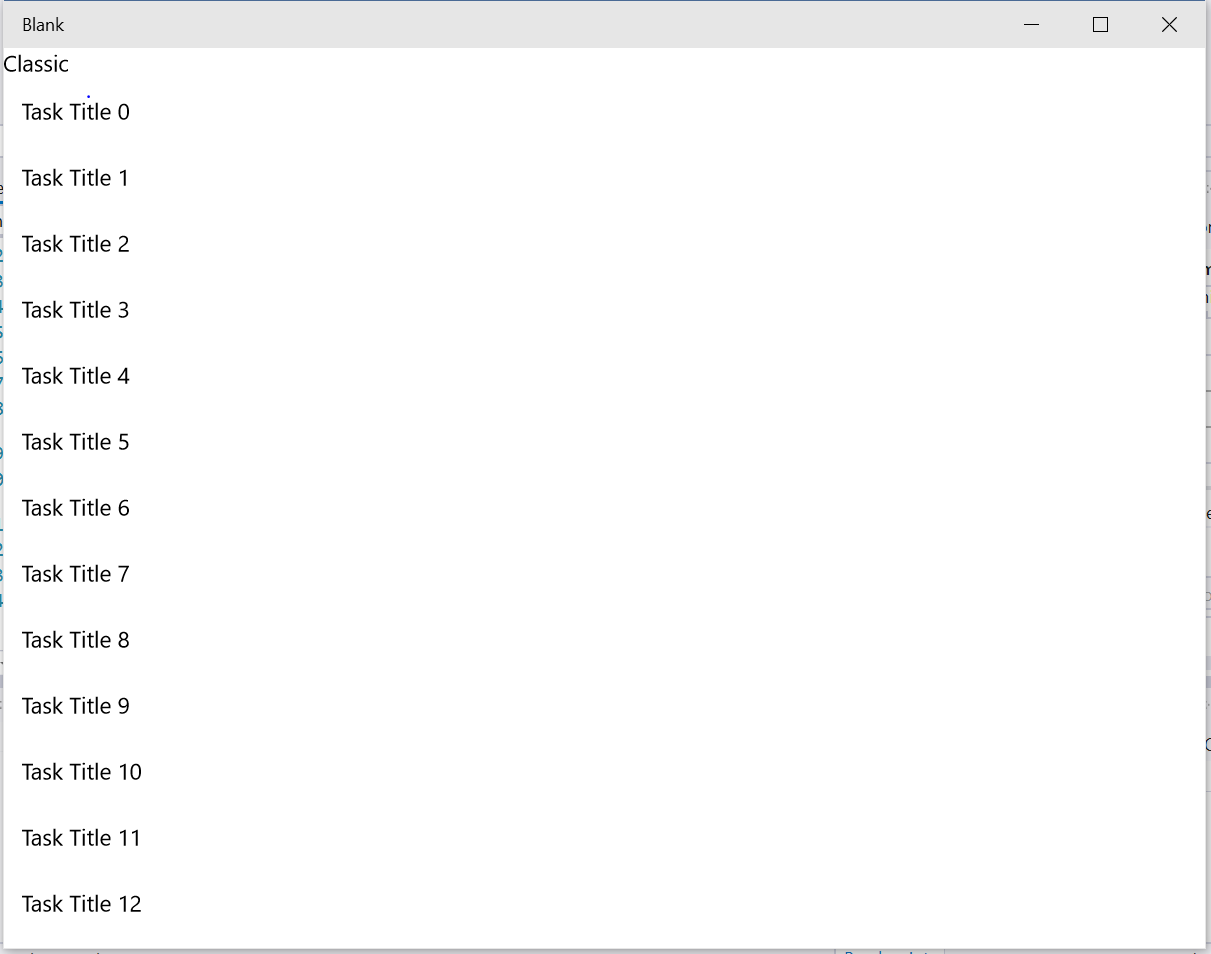
<TextBlock Text="{Binding Title}" />

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

1. Build and run your project. You will see a Header title of “Classic” and the list of 100 TodoItems generated by the view model.

**

**Figure 4**

*The basic view: a list of TodoItems bound with classic bindings.*

1. Stop debugging and return to MainPage.xaml. Copy the current ListView to create a second ListView in the next column. Change the **ItemsSource** binding to a compiled binding.
   1. **Note:** Compiled bindings use the new **x:Bind** syntax.
      1. XAML

</ListView>

<ListView ItemsSource="{x:Bind Items}" Header="Compiled" Grid.Column="1">

<ListView.ItemTemplate>

<DataTemplate>

<TextBlock Text="{Binding Title}" />

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

1. Open MainPage.xaml.cs. We will use a lambda function to assign the ViewModel to the DataContext when the DataContext is changed.
   1. **Note:** Compiled binding references a strongly-typed property of the Page and does not use the DataContext.
      1. C#

public MainPage()

{

this.InitializeComp onent();

this.DataContextChanged += (s, e) => { ViewModel = DataContext as ViewModels.MainPageViewModel; };

}

public ViewModels.MainPageViewModel ViewModel { get; set; }

1. Return to MainPage.xaml. Update the binding for the compiled ListView to reflect the ViewModel property from the code-behind.
   * 1. XAML

<ListView ItemsSource="{x:Bind ViewModel.Items}" Header="Compiled" Grid.Column="1">

<ListView.ItemTemplate>

<DataTemplate>

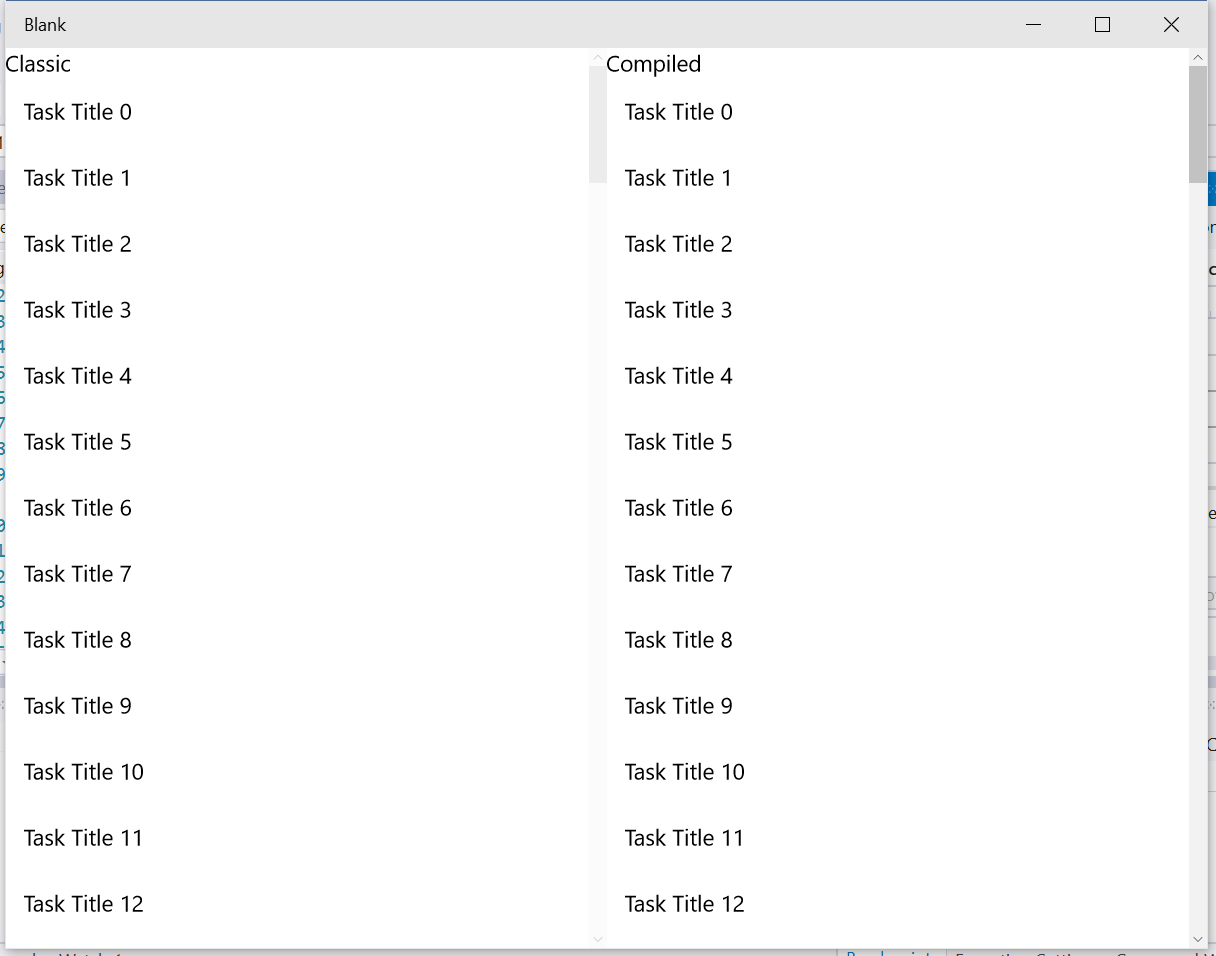
<TextBlock Text="{Binding Title}" />

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

1. Build and run your app. You will now see two columns, one Classic and one Compiled, each containing the list of TodoItems.

**

**Figure 5**

*Classic and compiled ListView bindings.*

1. So far, we have changed the list to use a compiled binding, but not the DataTemplate. Change the title binding within the compiled list to a compiled binding. Add the Binding.Models namespace to the ListView, and define the DataType in the DataTemplate.
   1. **Note:** As you can see from above, you can mix and match classic and compiled binding as necessary.
      1. XAML

<ListView ItemsSource="{x:Bind ViewModel.Items}" xmlns:m="using:Binding.Models" Header="Compiled" Grid.Column="1">

<ListView.ItemTemplate>

<DataTemplate x:DataType="m:TodoItem">

<TextBlock Text="{x:Bind Title}" />

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

1. Build and run your app. Both lists should appear as before, but the Compiled list now has compiled title bindings.

Exercise 2: Binding to the Property of an Element

1. In this exercise, you will use the ElementName property to bind to the property of another element in your application. Specifically, you will bind the height and width of a rectangle to the value of a slider. Moving the slider will change the size of the rectangle.
   1. **Note:** The ElementName property is one of the ways you can explicitly set the source of a Binding and override the inherited data context. For more information, see <https://msdn.microsoft.com/en-us/library/ms746695(v=vs.110).aspx>.
2. Task 1 – Open the starter project
   1. The first task is to open the starter project we have created for you.
3. Open the file location where you installed the hands-on labs. Navigate to the **\Lab 2\Solution\ElementName** folder and open **ElementName.sln** in Visual Studio 2015.
4. To prepare to build and run your app, use the Solution Configurations dropdown to choose the Debug configuration and use the Solution Platforms dropdown to target x86 (Figure 8). To run the app on the local machine, select Local Machine from the drop-down list next to the Start Debugging button on the debugger Standard toolbar.
5. Build and run the app.

**

**Figure 6**

*The starter app.*

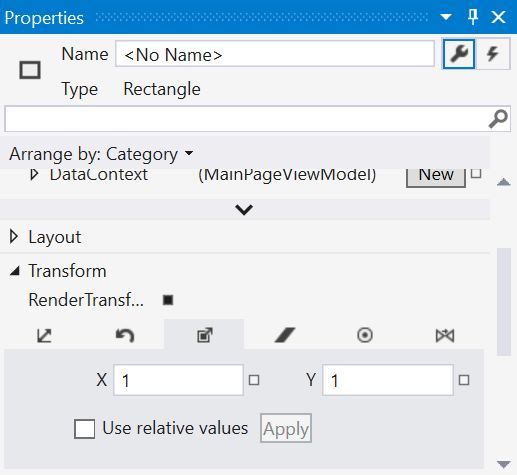
1. Return to Visual Studio and stop debugging.
2. Task 2 – Create the ElementName binding
   1. In this task, you will create a rectangle and a slider. You will then bind the ScaleX and ScaleY properties of the rectangle to the value of the slider.
3. Open **MainPage.xaml**. Add a rectangle within the **StackPanel**.
   * 1. **XAML**
   1. <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
   3. <Rectangle Margin="50" Height="100" Width="100" Fill="SteelBlue" />
   5. </StackPanel>
4. Add a slider below the rectangle and assign it an **x:Name** of **MySlider**­.
   * 1. **XAML**
   1. <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
   3. <Rectangle Margin="50" Height="100" Width="100" Fill="SteelBlue" />
   5. <Slider Minimum=".5" Maximum="2.0" StepFrequency=".1" x:Name="MySlider" />
   7. </StackPanel>
5. In your XAML code, highlight the Rectangle element to open its Properties window.



**Figure 7**

*Highlight the rectangle to open its Properties window.*

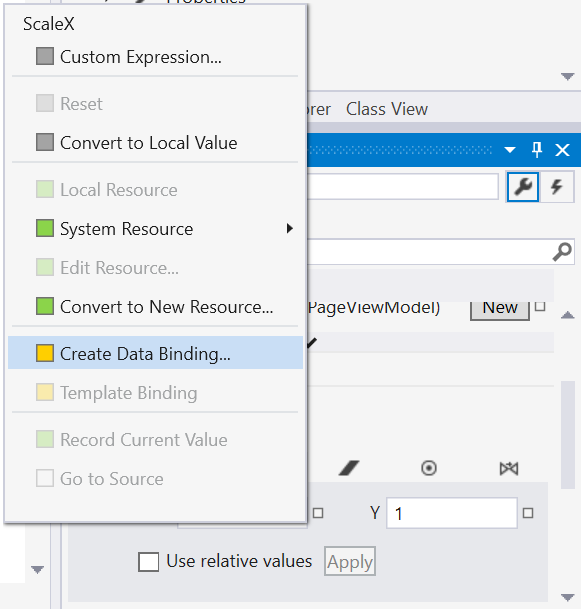
1. Expand the **Transform** section in the Properties window and select the **Scale** tab (see **Figure 8**).
   1. Note: The X and Y properties shown in the Scale tab are in fact ScaleX and ScaleY.



**Figure 8**

*Scale properties.*

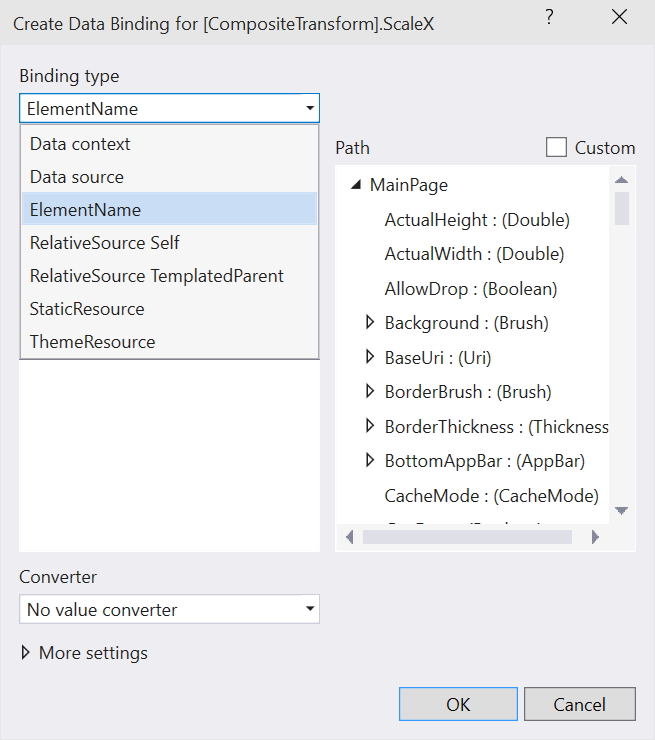
1. Select the small box next to the **ScaleX** property to open its context menu. Confirm that the **ScaleX** property name displays at the top of the menu and select the **Create Data Binding** option. The Create Data Binding dialog will open.



**Figure 9**

*The context menu for the ScaleX property. Note the property name at the top of the menu.*

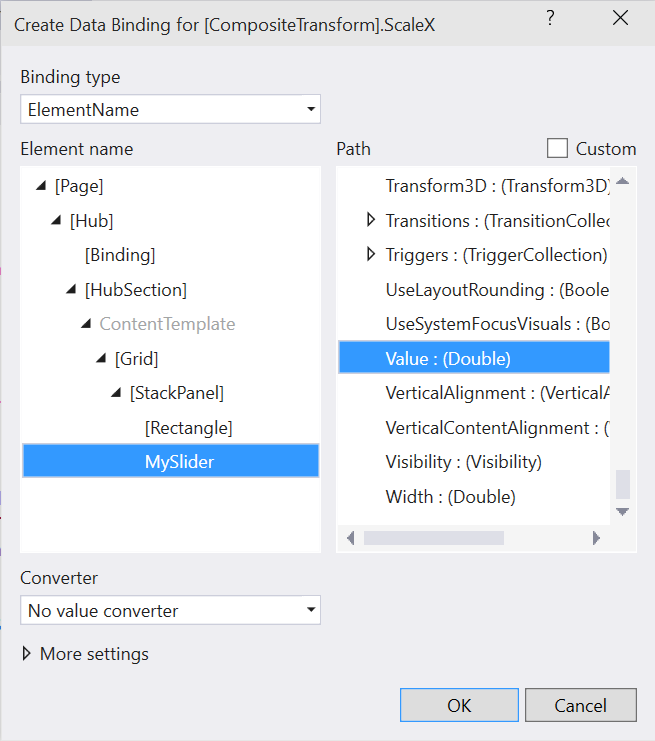
1. In the Create Data Binding dialog, use the **Binding type** dropdown menu to change the binding type to **ElementName**.

**

**Figure 10**

*Change the binding type to ElementName.*

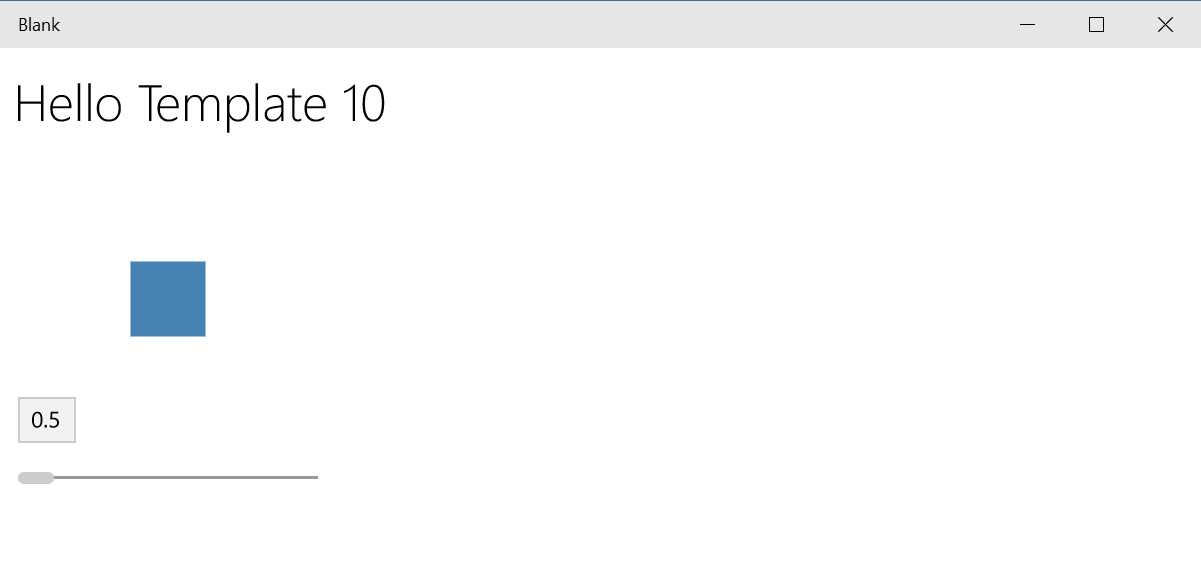
1. In the Element name panel, expand the hierarchy of your page until you reach the elements in the **StackPanel**. Select **MySlider**. In the Path panel, select **Value : (Double)** (see Figure 6). Leave all other fields the same and click **OK** to confirm. The code for the binding will appear in the XAML for your rectangle.

**

**Figure 11**

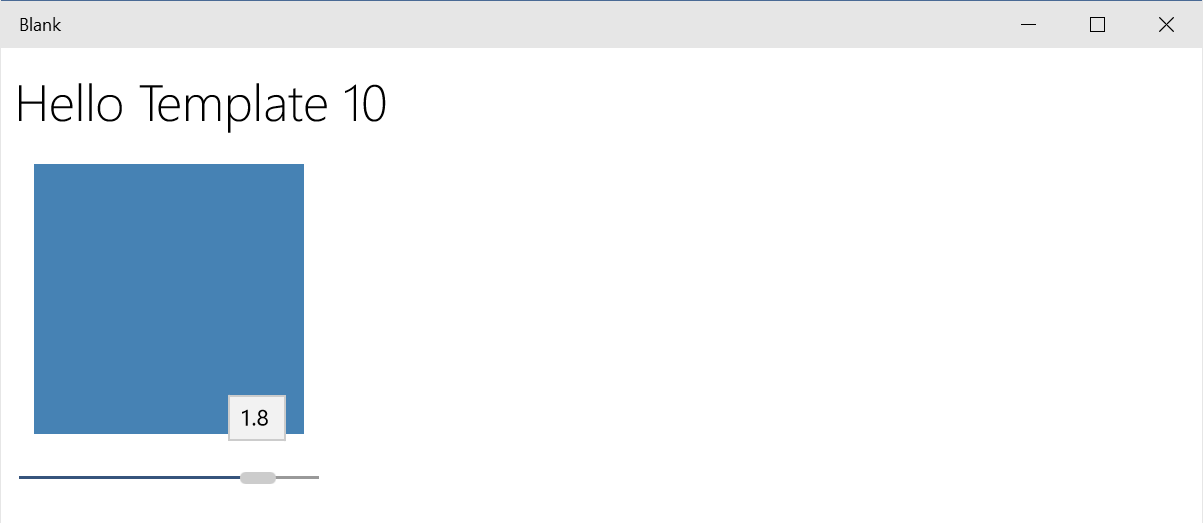
*Bind the ScaleX property of the rectangle to the value of MySlider.*

1. Copy the code for the ScaleX binding to create an identical binding for the ScaleY property.
   * 1. **XAML**
   1. <Rectangle Margin="50" Height="100" Width="100" Fill="SteelBlue" RenderTransformOrigin="0.5,0.5" >
   2. <Rectangle.RenderTransform>
   3. <CompositeTransform ScaleX="{Binding Value, ElementName=MySlider}" ScaleY="{Binding Value, ElementName=MySlider}" />
   4. </Rectangle.RenderTransform>
   5. </Rectangle>
2. Build and run your project. Use the slider to change the size of the rectangle.

**

**Figure 12**

*The rectangle at 0.5 scale.*

**

**Figure 13**

*The rectangle at 1.8 scale.*

1. Stop debugging and return to Visual Studio.

Summary

* 1. During this lab we have learned how we can bind data from our model via a view model to our UI (or view). We have used examples of classic and compiled bindings and seen that classic bindings use the datacontext, whereas compiled bindings must access a strongly-typed property on the page. We have also shown how we can bind to other elements on our page.