WPF Control API Reference

Build your own WPF control as a .NET DLL and place it on a WPF display through the WPF Control element.

A custom WPF control is a WPF `UserControl` that you compile into a DLL. On a display, the WPF Control element loads that DLL, lists the control's public properties, and binds them to tags.

This page covers **building** the control: the project, the classes, the rules the Designer applies, and a template you can download and adapt.

> To **place and configure** the finished control on a display (selecting the DLL, the Properties table, binding modes), see WPF Control Reference.

Download the template
Requirements
Rules
Build your own control from the template
1. Open and build the template
2. Rename it
3. Add your properties
4. Design the visual
5. Place it on a display
6. Bind it to tags
The IPortableControl members
Troubleshooting
Source code
Related
In this section...

Download the template

Download WPFControl-TCustom.zip. It contains a complete, minimal control named `TCustom`: it reads two values, shows each one in a text box, and shows both joined as A_B. Build it, place it on a display, then change it into your own control.

File	Role
`TCustom.cs`	The control. The only public class in the assembly. Properties, visual construction, tag linking, save and clipboard support.
`TCustomControl.xaml` / `.xaml.cs`	The visual. Internal, created by `TCustom`.
`TCustomConfig.xaml` / `.xaml.cs`	A configuration dialog. Internal, returned by `TCustom.GetConfigControl()`. Not used by the WPF Control element (see *The IPortableControl members*).
`TCustom.png`	Icon, embedded as a resource.
`TCustom.csproj` / `TCustom.sln`	The project. References the FrameworX assemblies from your installation and does not include them.

Validated with FrameworX 10.1.5.

Requirements

- Windows and Visual Studio with the **.NET desktop development** workload.
- A **.NET Framework 4.8** class library, the same target as the FrameworX built-in WPF controls.
- A FrameworX installation. The project references `T.Library.dll`, `T.Library.Wpf.dll`, `T.Toolkit.dll` and `T.Toolkit.Wpf.dll` from the installation's `fx-10` folder, by default `C:\Program Files\Tatsoft\FrameworX\fx-10\`.
- The control runs only on WPF clients: Windows, RichClient and SmartClient.
- The display must use the **WPF** engine. The WPF Control element is not available on Portable displays.

Rules

Follow these rules for every custom WPF control. Breaking one does not always produce an error: the control can be missing from a list, show the wrong class, or render blank.

1. **The control class is public.** It must be public, not abstract, derive from `System.Windows.Controls.UserControl`, and have a public parameterless constructor. Only such types are listed in Select Component.
2. **Every other UserControl in the assembly is internal.** The visual, the configuration dialog and any helper control must be internal, or they are also listed in Select Component and a user can pick the wrong one. For a XAML UserControl, set `x:ClassModifier="internal"` on the root element and declare the code-behind as `internal partial class`.
3. **The assembly name equals the control's full type name.** Namespace plus class: the template's control `T.Portable.Controls.TCustom` compiles to `T.Portable.Controls.TCustom.dll`. The WPF Control settings reject a DLL whose file name does not match the selected control.

4. **Every bindable value is a public property that raises PropertyChanged.** Implement `INotifyPropertyChanged`. A property computed from others (the template's `ConcatenatedText`) must also be notified when one of its inputs changes, or the display does not refresh it.
5. **The visual is built in the constructor, once.** Do not rely on a later lifecycle call to create it, and never add a child element that already has a parent. Either mistake leaves the control blank on the display.
6. **FrameworX assemblies are referenced, not copied.** Set `Private` to `False` (Copy Local off) on every FrameworX reference. FrameworX loads its own copies at runtime.
7. **Close the Designer and any running client before rebuilding.** They keep the DLL loaded, and the build cannot overwrite it.

Build your own control from the template

1. Open and build the template

Open `TCustom.sln` and build. The output is `bin\Release\T.Portable.Controls.TCustom.dll` (or `bin\Debug\`). If FrameworX is not installed in `C:\Program Files\Tatsoft\FrameworX`, change `FrameworXPath` in `TCustom.csproj` to your installation's `fx-10` folder, keeping the trailing backslash.

TCustom.csproj: assembly name, target and FrameworX path

```
<!-- The assembly name MUST equal the full type name of the control class (namespace + class). -->
<AssemblyName>T.Portable.Controls.TCustom</AssemblyName>
<TargetFrameworkVersion>v4.8</TargetFrameworkVersion>
<FileAlignment>512</FileAlignment>
<Deterministic>true</Deterministic>
<!-- Folder that holds the FrameworX reference assemblies. Override on the command line with
     /p:FrameworXPath=<your folder>\ if FrameworX is installed somewhere else. Keep the trailing backslash.
-->
<FrameworXPath Condition=" '$(FrameworXPath)' == '' ">C:\Program Files\Tatsoft\FrameworX\fx-10\<
/FrameworXPath>
```

TCustom.csproj: a FrameworX reference, not copied to the output

```
<Reference Include="T.Library">
  <HintPath>$(FrameworXPath)T.Library.dll</HintPath>
  <Private>False</Private>
</Reference>
```

2. Rename it

Choose your namespace and class name, then change every place below. Keep rule 3: `AssemblyName` must equal namespace plus class.

File	What to change
`TCustom.csproj`	`AssemblyName` and `RootNamespace`.
`TCustom.cs`	The namespace, the class name, the constructor name, and the assembly and file name in the `pack://` URI inside `GetIconImage()`.
`TCustomControl.xaml`, `TCustomConfig.xaml`	`x:Class`, the `clr-namespace`, and the control type in `{x:Type local:TCustom}`.
`TCustomControl.xaml.cs`, `TCustomConfig.xaml.cs`	The namespace, the class names, and every reference to `TCustom`.

3. Add your properties

Each value the display can bind to is a public property. The setter returns early when the value is unchanged, stores it, and raises `PropertyChanged` for the property itself and for every property computed from it.

TCustom.cs: a bindable property and a computed one

```
private string _firstTextValue = string.Empty;

/// <summary>
/// Resolved runtime value of the first token/object.
/// </summary>
public string FirstTextValueStr
{
    get => _firstTextValue;
    set
    {
        if (_firstTextValue == value)
            return;

        _firstTextValue = value;
        _firstTextObjectRef?.SetValue(value);

        OnPropertyChanged();
        OnPropertyChanged(nameof(ConcatenatedText));
    }
}

/// <summary>
/// Combined text shown by the control.
/// </summary>
public string ConcatenatedText => $"{FirstTextValueStr}_{SecondTextValueStr}";
```

The line `_firstTextObjectRef?.SetValue(value)` belongs to the template's own tag linking (see *The IPortableControl members*). It does nothing when that link is not set, so the property works the same with the WPF Control element.

4. Design the visual

The visual is a separate internal UserControl. Its bindings name the control class as their source through `RelativeSource`, so they reach your properties whatever the `DataContext` is. `UpdateSourceTrigger=PropertyChanged` pushes each keystroke to the property instead of waiting for the text box to lose focus.

TCustomControl.xaml

```
<UserControl x:Class="T.Portable.Controls.TCustomControl"
             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:T.Portable.Controls"
             x:ClassModifier="internal"
             mc:Ignorable="d">

    <StackPanel Margin="10" Width="360">
        <TextBlock Margin="0,0,0,6" Text="CustomControl (render)"/>
        <TextBlock FontWeight="Bold"><Run Text="StringA"/><Run Text=":"/></TextBlock>
        <TextBox Text="{Binding Path=FirstTextValueStr, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged,
RelativeSource={RelativeSource AncestorType={x:Type local:TCustom}}}"/>
        <TextBlock FontWeight="Bold" Text="StringB:"/>
        <TextBox Text="{Binding Path=SecondTextValueStr, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged,
RelativeSource={RelativeSource AncestorType={x:Type local:TCustom}}}"/>
        <Border Padding="8" BorderBrush="Gray" BorderThickness="1" CornerRadius="4">
            <StackPanel>
                <TextBlock FontWeight="Bold" Text="StringA_StringB:"/>
                <TextBlock Margin="0,6,0,0"
                           Text="{Binding Path=ConcatenatedText, RelativeSource={RelativeSource AncestorType=
{x:Type local:TCustom}}}"
                           Background="White"
                           />
            </StackPanel>
        </Border>
    </StackPanel>
</UserControl>
```

The control class builds the visual in its constructor through one method that is safe to call any number of times. It creates the grid only when there is none, and adds the child only when it has not been added. `Loaded` and `StartRuntime()` call the same method.

TCustom.cs: constructor and visual construction

```
public TCustom()
{
    MinWidth = 10;
    MinHeight = 10;

    // Build the visual up front so the control is never empty,
    // whichever lifecycle hook the host calls first.
    EnsureVisual();

    Loaded += OnLoaded;
}

private void EnsureVisual()
{
    if (_mainGrid == null)
    {
        _mainGrid = new Grid();
        Content = _mainGrid;
    }

    if (_control == null)
    {
        _control = new TCustomControl(this);
        _mainGrid.Children.Add(_control);
    }

    _control.SetDataContextToParent();
    _mainGrid.Background = Background;
}
```

5. Place it on a display

Close the Designer if it has the previous build loaded (rule 7), build, and then place the control with the WPF Control element as described in WPF Control Reference: select your DLL, then pick your control class in Select Component. Only the public control class is listed there.

6. Bind it to tags

Double-click the control on the display to open its settings. The Properties table lists the control's public properties. For each value, enter the tag in **ObjectLink** and choose the **Binding** mode. For the template:

ControlProperty	Binding	ObjectLink
`FirstTextValueStr`	TwoWay	`Tag.Integer1`
`SecondTextValueStr`	TwoWay	`Tag.Integer2`
`ConcatenatedText`	OneWay	Leave empty. It is read-only and is shown by the control.

With these links, the control shows each tag's value and the two values joined. The binding modes are described in WPF Control Reference.

The IPortableControl members

The WPF Control element only needs a public UserControl. The template also implements the `IPortableControl` interface (namespace `T.Toolkit`), which carries the members FrameworX uses for controls registered as extension controls. On the WPF Control element, tags are bound through the Properties table and the dialog returned by `GetConfigControl()` is not opened. You can keep these members as they are.

Member	What the template does
`EndInit()`	Calls `base.EndInit()` and `WK.EndInit(this)`, once.
`StartRuntime()`	Builds the visual if needed, then parses `FirstTokenLinkStr` and `SecondTokenLinkStr` into `ObjectReference` objects and subscribes to their changes.
`Dispose()`	Releases the `ObjectReference` subscriptions.
`GetTokens`, `ApplyTokens`	Expose the link expressions to symbol token replacement.
`GetLabels`, `ResolveLabels`	Resolve labels in the link expressions.
`GetStrings`, `ApplyStrings`	Localization hooks. Empty in the template.
`OnStartClipboardCopy`, `OnFinishClipboardCopy`, `OnFinishClipboardPaste`	Keep the link expressions valid through copy and paste.
`OnSave(int displayId, object el)`	Saves the link expressions with the display.
`GetConfigControl()`	Returns the configuration dialog `TCustomConfig`, whose field editors use the `EditInfo` exposed by `NumberOrObjectField`.
`GetIconImage()`	Static method, not part of the interface. Returns the icon embedded as `TCustom.png`.

Troubleshooting

Symptom	Cause	Fix
Select Component lists more classes than your control.	Other UserControls in the assembly are public.	Make them internal (rule 2).
Warning: "DLL filename must be same of the control".	The selected class's full name does not match the DLL file name.	Select the control class, or set `AssemblyName` to namespace plus class (rule 3).
The control is blank on the display.	The visual is created only in a later lifecycle call, or a child element is added a second time.	Build the visual in the constructor, once (rule 5, step 4).
A value does not refresh on the display.	The property, or a property it feeds, does not raise `PropertyChanged`.	Raise it in the setter for the property and for every computed property (rule 4).
Build fails: "The process cannot access the file ... because it is being used by another process".	The Designer or a running client has the DLL loaded.	Close them and build again (rule 7).

WPF Control is not offered in the Viewer group.	The display uses the Portable engine.	Place the control on a display whose engine is WPF.
Your control's own configuration dialog does not open.	The WPF Control element uses its own Properties table.	Expected. Bind the properties in that table (step 6).

Source code

The complete source of this template is in `WPFControl-TCustom.zip`, available under **Download the template** at the top of this page. That zip is the single, authoritative copy of the code. The code on this page is excerpts only, to explain how the template works.

Related

- WPF Control Reference: place and configure the WPF Control element on a display.
- WPF Client overlay limitations (HWND-hosted controls): why a control that hosts native Win32 content paints above overlapping WPF visuals.
- Custom Control API Reference: choose between WPF, Blazor and Portable controls.
- Blazor Control API Reference: a browser-only control built with Razor components.
- Portable API Reference: one control for both the WPF clients and the browser.

In this section...