MSDN Hands-On Lab

Building Line of Business Applications with Windows Presentation Foundation

Ruihua Jin

Ronnie Saurenmann
(ronnies@microsoft.com)

Microsoft Switzerland

Document version : 1.0

Last update : May 2007

Table of Contents

About this Hands-On Lab

What You Will Learn About

Windows Presentation Foundation (WPF) provides developers with a unified programming model for building rich Windows smart client user experiences that incorporate UI, media, and documents. In this Hands-On Lab you will learn about building a WPF application by using Microsoft Expression Blend. The finished project looks like the following, which is a Microsoft Outlook 2007 UI replica:

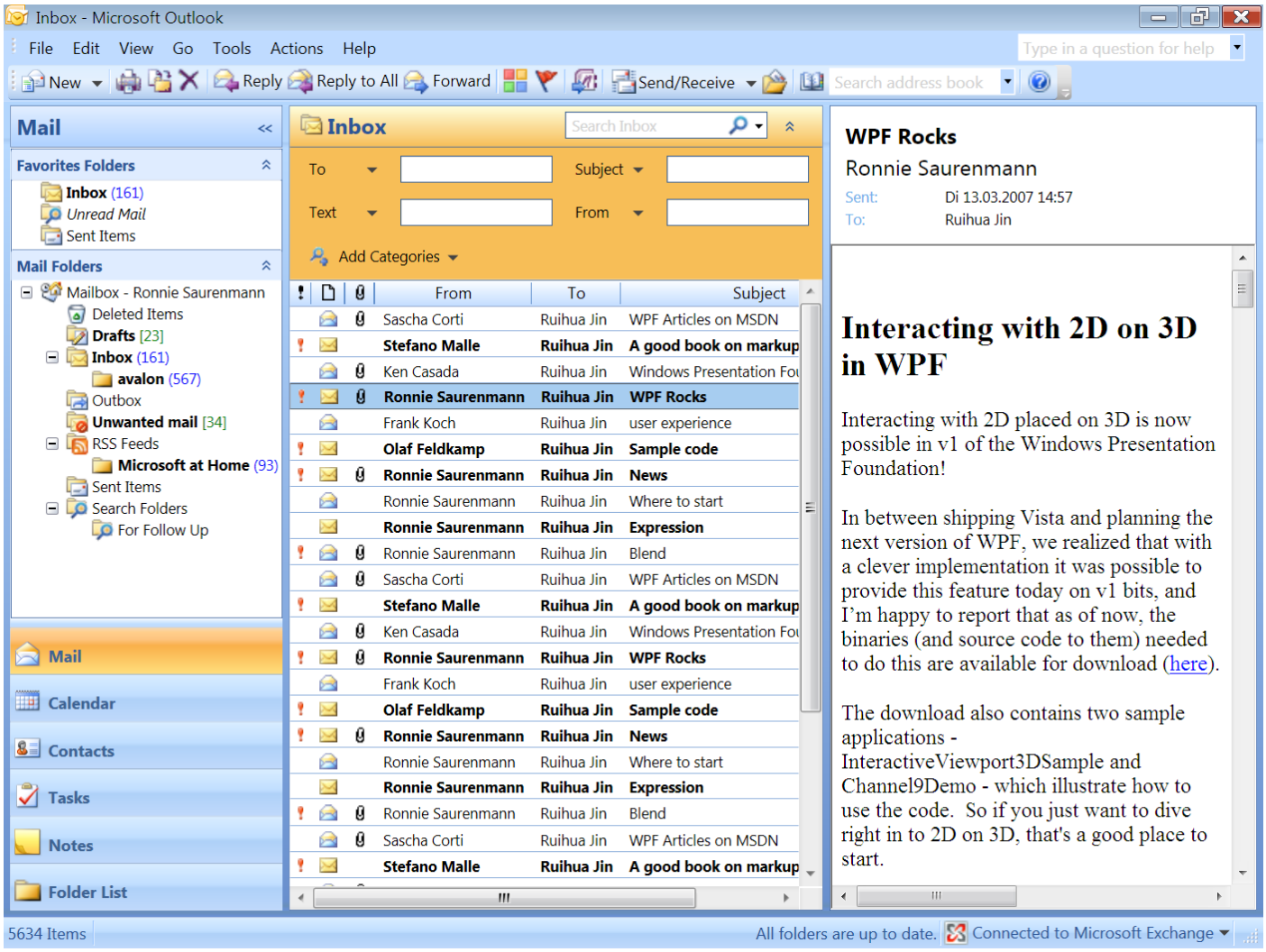


Figure 1 The finished project.

The following WPF features are involved:

- Controls for layout and design like **Grid**, **DockPanel**, **StackPanel**, etc.
- Common user interface controls like **Button, ToggleButton, Border**, **TextBlock**, **ListView**, etc.
- Drawing paths
- Transforming objects
- Solid brushes, linear gradient brushes
- Data binding, data converters
- Styles, control templates, template binding
- Data templates
- Property triggers, data triggers
- Event handling

Organization of This Hands-On Lab

The Hands-On Lab is divided into three courses: one basic course and two advanced courses.

Part I – Building Components of Outlook UI (Basic)

In the first part of the Hands-On Lab basic knowledge of WPF is imparted. All the important components of the Outlook UI are built during this course.

Part II – Customizing Buttons (Advanced)

In the second part you will learn about how to customize **Button** controls after your own taste.

Part III – Customizing ListView (Advanced)

In the third part you will learn about how to customize a **ListView** control after your own taste.

Conventions Used in This Documentation


Important point to learn by heart


Tip, note


Exercise


Step-by-step introduction in MS Expression Blend


Code snippet (to save you a lot of typing, we have prepared code snippets which can be copied and then pasted. The files for the code snippets are located under the directory ***project folder*\OutlookUI HOL\code snippets**, where ***project folder*** is the directory into which you have copied the project).

Where to Start

It is important for us that you become acquainted with as many WPF features as possible during the Hands-On Lab, so we have prepared several projects which you can start with under ***project folder*\OutlookUI HOL\Initial Projects**, feel free to choose your own starting point. The following table gives you the overview:

Starting point	**Project name**
Basic, step 1	OutlookUI HOL Basic - Initial
Basic, step 6	OutlookUI HOL Basic6 - Initial
Basic, step 8	OutlookUI HOL Basic8 - Initial
Basic, step 10	OutlookUI HOL Basic10 - Initial
Advanced (Sidebar)	OutlookUI HOL Basic - Final
Advanced (ListView)	OutlookUI HOL Basic - Final

Building a MS Outlook 2007 User Interface Replica (Basic)

In this basic course, we will build a WPF application with all the important components you see in MS Outlook 2007. After you finish the basic course, your application will look like the following:

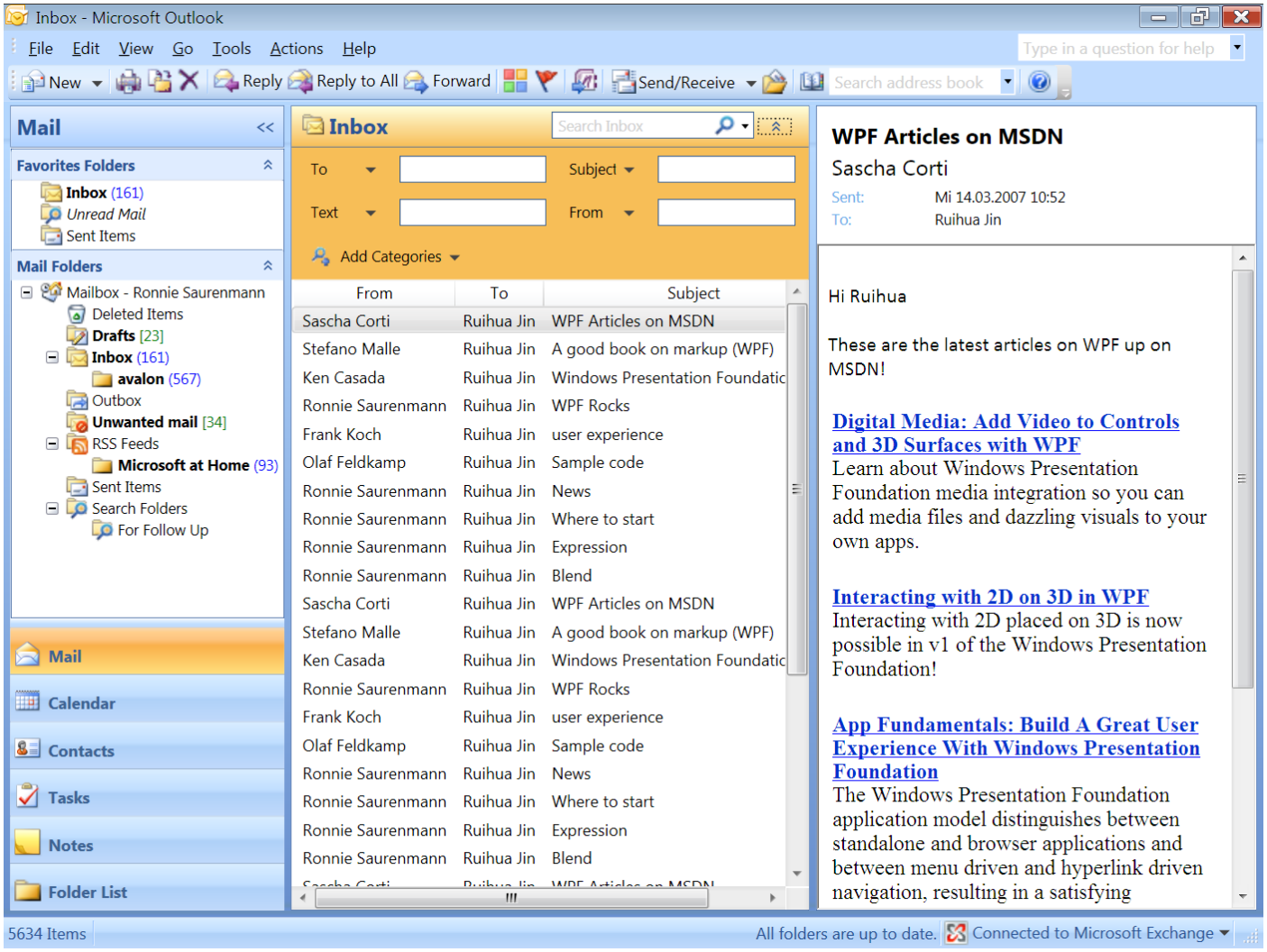


Figure 2 The finished project of the basic course.

You will learn about the following features in this basic course:

- Expression Blend user interface
- Controls for layout and design like **Grid**, **DockPanel**, **StackPanel**, etc.
- Common user interface controls like **Button, ToggleButton, Border**, **TextBlock**, **ListView**, etc.
- Solid brushes, linear gradient brushes
- Drawing paths
- Transforming objects
- Data binding an element's property to another element's property
- Data binding an element's data context to an XML data source and data binding an element's property to an XML element
- Data converters
- Styles
- Control Templates
- Property triggers
- Event handling

Task 1: Opening the Project in Expression Blend

The following procedure describes how to open the **OutlookUI HOL Basic - Initial** project:

1. In Expression Blend, click **File**, and then click **Open Project…**. The **Open Project** dialog box opens.
2. Go to the directory ***project folder*\OutlookUI HOL\Initial Projects\OutlookUI HOL Basic - Initial**, where *project folder* is the folder into which you copied the project. Select **OutlookUI HOL.sln**, and then click **Open**.

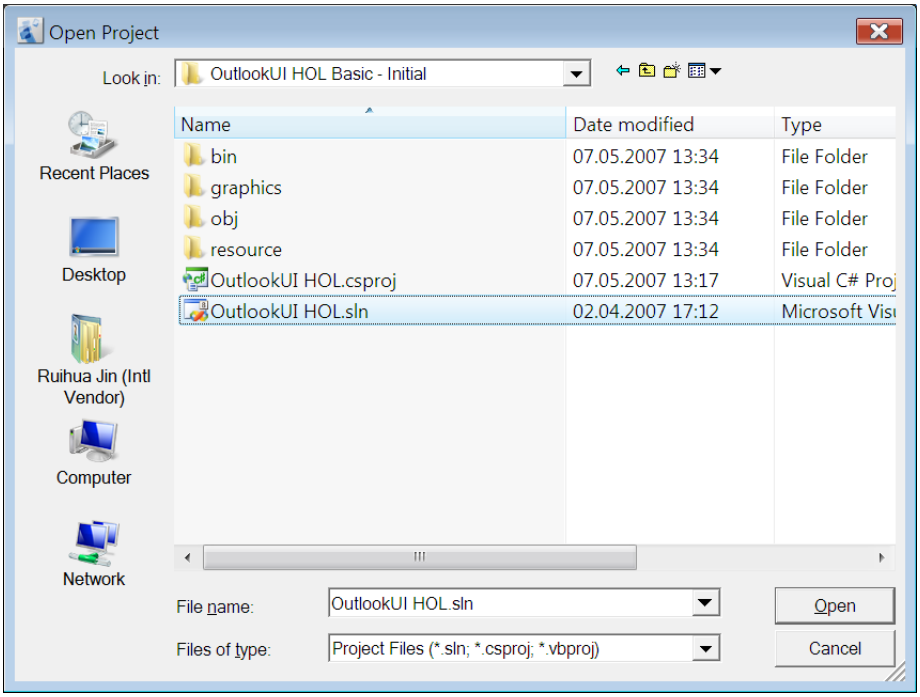


Figure 3 ***Open Project*** **dialog box for opening the OutlookUI HOL Basic - Initial project.**

Now you have a project loaded into memory and let's start building an impressive user interface.

Task 2: Getting Familiar with Expression Blend

The user interface of Expression Blend is divided into 6 major areas:

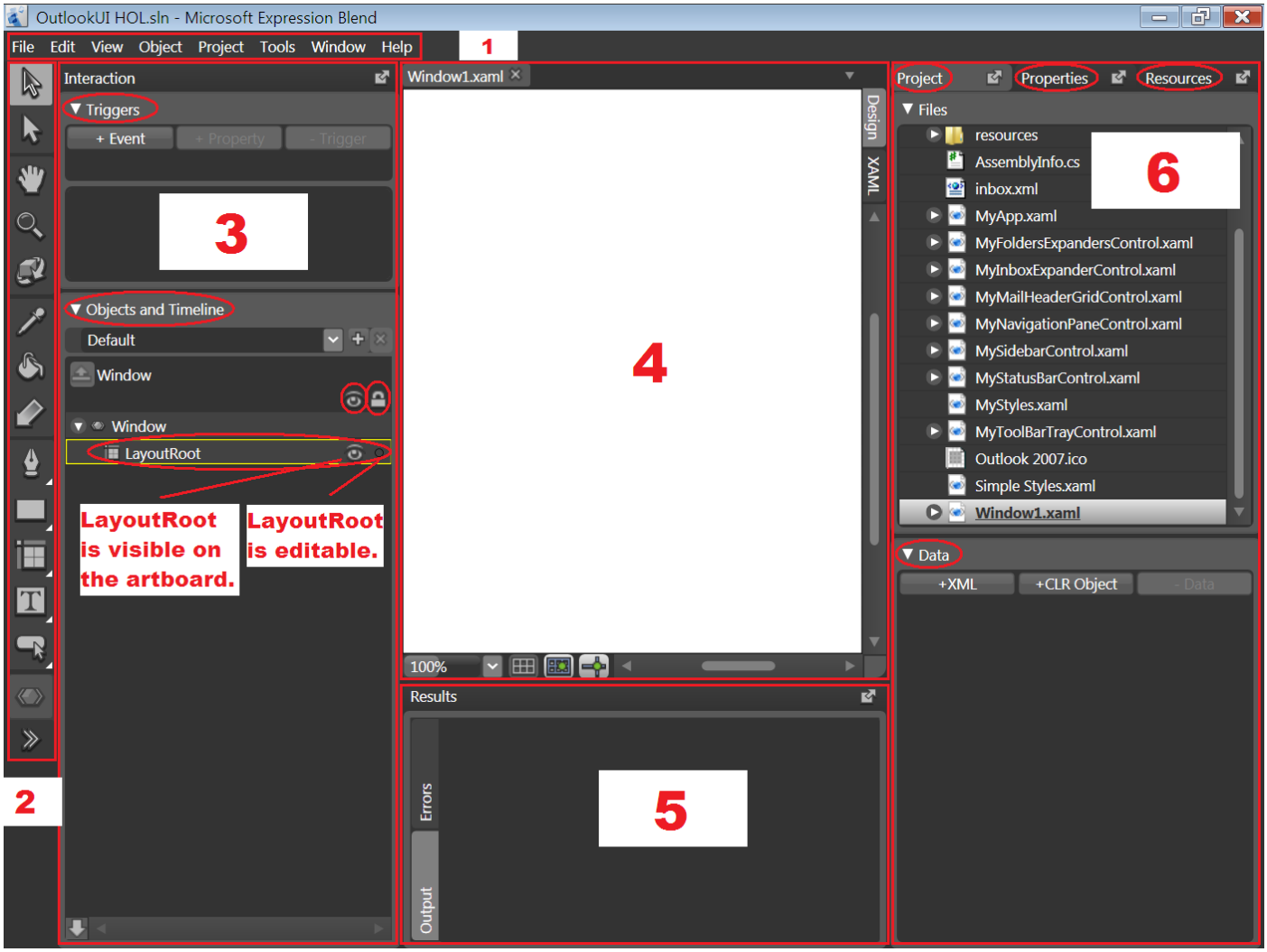


Figure 4 The default user interface of Expression Blend.

1. The **Menu**.
2. The **ToolBox**. The **Toolbox** provides you various assets which you can draw on the artboard. Pause the pointer over the tools to see their names. Some of the tools include a small triangle which you can right-click to display a list of other available assets. To get a complete list of assets click the **Asset Library** button .
3. The **Interaction** panel.
    - The **Triggers** section. Under **Triggers** in the **Interaction** panel you can define property triggers or event triggers for a certain control.
    - The **Objects and Timeline** section. If you have an XAML file like **Window1.xaml** open on the artboard in **Design** view, the hierarchical structure of the XAML document is displayed under **Objects and Timeline**. An arrow is shown beside an element if this element has child elements, click the arrow to expand or collapse the element. For a WPF application, the root element is either **Window** or **Page**, and you can draw various controls of your own choice under the **Window** resp. **Page** element.

During this HOL you may encounter the following problem: the artboard suddenly turns white and you believe you have lost all your work. Don't panic, just check that is displayed beside all your controls in the **Objects and Timeline** section, like the one beside **LayoutRoot** shown in Figure 4. The icon indicates that the related control and its child elements are visible on the artboard. If you see instead of , then it's likely that you have clicked unconsciously, just click to get your controls back on the artboard.

Similarly, the icon  tells us whether the related control and its child elements are editable.  stands for "locked", and  stands for "editable" which is the default. Always remember to check the editability of a control when you are wondering why you cannot select resp. edit the control.

4. The **artboard**. The **artboard** provides you two views: the **Design** view and the **XAML** view. The **Design** view shows you the look of your application. The **XAML** view displays the whole XAML document. Click the **Design** tab and the **XAML** tab to switch between these two views.

Expression Blend, XAML, Windows Presentation Foundation and .NET Framework 3.0

Microsoft® Expression Blend™ is an all new, full-featured professional design tool for creating engaging, sophisticated, and Web-connected user interfaces for Windows®-based applications.

Extensible Application Markup Language, or **XAML** (pronounced "zammel"), is an XML-based markup language developed by Microsoft. XAML is the language behind the visual presentation of an application that you develop in Microsoft® Expression Blend™, just as HTML is the language behind the visual presentation of a Web page. Creating an application in Expression Blend means writing XAML code, either by hand or visually by working in the **Design** view of Expression Blend.

XAML is part of the **Microsoft® Windows Presentation Foundation** (**WPF**). WPF is the category of features in the **Microsoft® .NET Framework 3.0** that deal with the visual presentation of Windows-based applications and Web browser-based client applications.

The XAML for any given document in Expression Blend is stored in a **.xaml** file. If there is underlying code for your XAML document, that code is stored in a file of the same name, with the additional extension of **.cs** or **.vb**. For example, if your XAML document is named **Window1.xaml**, the code-behind file will be called **Window1.xaml.cs** if the programming language of the code is C#.

To get the complete view of the **artboard** choose **Hide Panels** (**F4** or **TAB**) from the **Window** menu.

5. The **Results** panel. The **Results** panel displays information like warnings, errors which originate from an erroneous XAML document, or information output during a project build.

WPF parser

When you build your project, the WPF parser reads the **.xaml** files for that project and reports any resulting errors. Likewise, when you open an existing project in Expression Blend, the parser reads the **.xaml** files that are included in your project folder and attempts to parse the elements and display the documents on the artboard in **Design** view. In both cases, if the parser encounters errors, the artboard is disabled, and Expression Blend displays an error message with a link to open XAML view so that you can resolve the errors. The parsing errors are also reported on the **Errors** tab in the **Results** panel.

6. The **Project/Properties/Resources** panels.
   - The **Project** panel.
     - The **Files** section. You get a list of all the files involved in the current project under **Files** in the **Project** panel. This is the location where you open files, add files, assemblies or other related resources like images, etc. to your project.

Switch to the **Project** panel and have a look at the file structure of our project.

     - The **Data** section. Under **Data** in the **Project** panel you can add XML or CLR object data sources to the project. You will learn how to add an XML data source and data bind controls to it in this lab.
   - The **Properties** panel. The **Properties** panel shows all the properties and events of a *selected* control.

Control selection for configuring properties

To ***select*** a control, click the control once under **Objects and Timeline**, after that the element name is highlighted in gray in the background and the corresponding element on the artboard is highlighted by a blue bounding outline (see Figure 5).

Select the **LayoutRoot** element and have a look at its properties.

Note that the properties are categorized by type such as **Brushes**, **Appearance**, etc., which allows you to easily locate a specific property. If you know the name of a property or part of it, you can also type it into the **Search** box to get only the properties shown which you want. Remember to clear the **Search** box when you don't want the properties to be filtered any more.

To view all the properties of a certain category, click the down-arrow if existing.

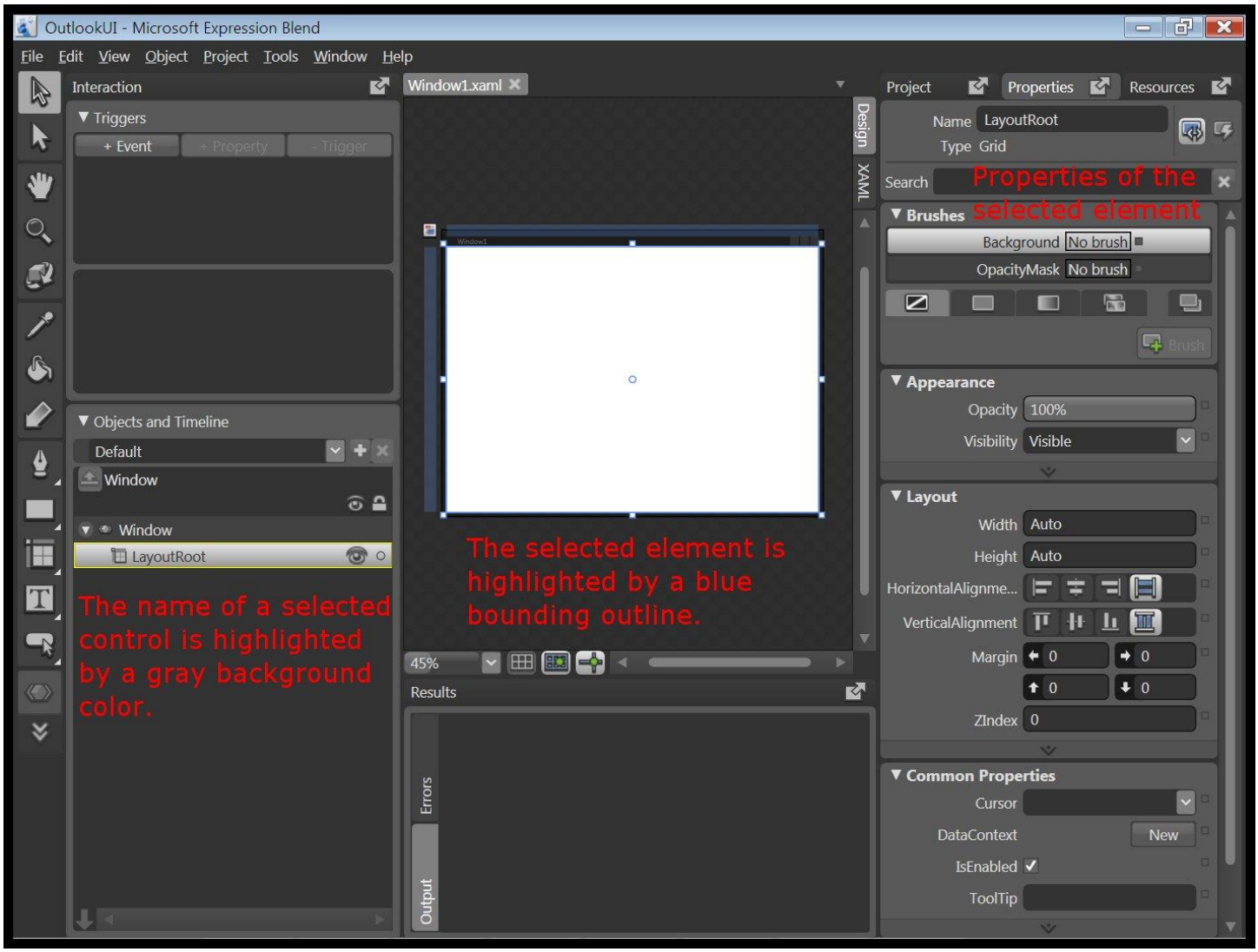


Figure 5 LayoutRoot is the selected element.

- The **Resources** panel. The **Resources** panel shows you all the resources defined for the current project. Examples of resources are brushes, styles and control templates.

Switch to the **Resources** panel and click **MyStyles.xaml**, have a look at brushes which will contribute to our **OutlookUI**.

Task 3: Adding a DockPanel Control to LayoutRoot

Before you begin adding controls to your WPF application, you should first make a decision on the layout panel your controls will be contained in. There are three commonly used layout panels: **Grid**, **DockPanel** and **StackPanel**.

- A **Grid** control arranges child elements in a layout of rows and columns that form a grid. The **LayoutRoot** element provided by Expression Blend by default when you start with a new project is a **Grid** control.
- A **DockPanel** control arranges child elements so that they stay to one edge of the panel: **Left**, **Top**, **Right** or **Bottom**.
- A **StackPanel** control arranges child elements into a single line that can be oriented horizontally or vertically.

For our project we choose a **DockPanel** control as the root container. The following procedure shows you how to add a **DockPanel** control to **LayoutRoot**.

1. Under **Objects and Timeline,** double-click **LayoutRoot** to activate it.

Control activation for adding child elements

You need to ***activate*** the element before you add child elements to it. Double-click the parent element under **Objects and Timeline** in the **Interaction** panel, after that a yellow bounding box around the element name appears, which indicates that the element is active and you can now add child elements to it (see Figure 6).

Control selection vs. control activation

To edit the properties of an element, you only need to ***select*** the element: click the element once under **Objects and Timeline**. You can select an element without activating it.

You can also use the **Selection** tool  from the **Toolbox** to select resp. activate an element by clicking resp. double-clicking it on the artboard.

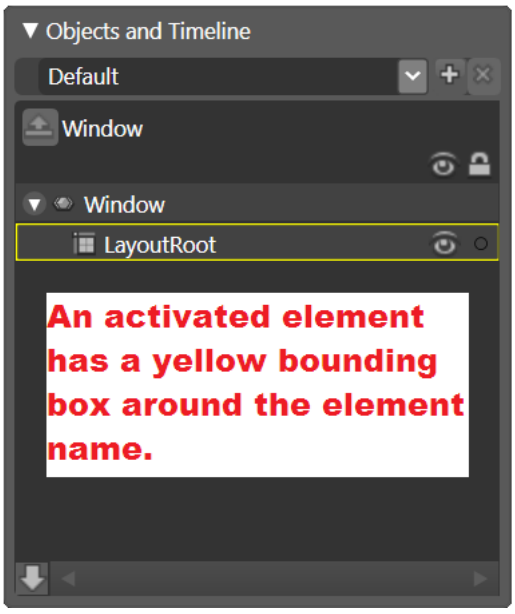


Figure 6 LayoutRoot is the active element.

2. Add the **DockPanel** tool to the **ToolBox**. 

The icon  indicates that there is a step-by-step introduction in the screenshot of Expression Blend. The number beside it is the figure number. We suggest you to always have a look at the figure first if there is one.

The **DockPanel** tool is not shown in the **ToolBox** by default, so you should first add the **DockPanel** tool to the **ToolBox**: click the **Asset Library** button in the **ToolBox** to open the **Asset Library** dialog box which displays all the tools available for the project. To quickly locate the **DockPanel** tool, type "**DockPanel**" into the text box in the upper-left corner, then click **DockPanel** shown in the dialog box.

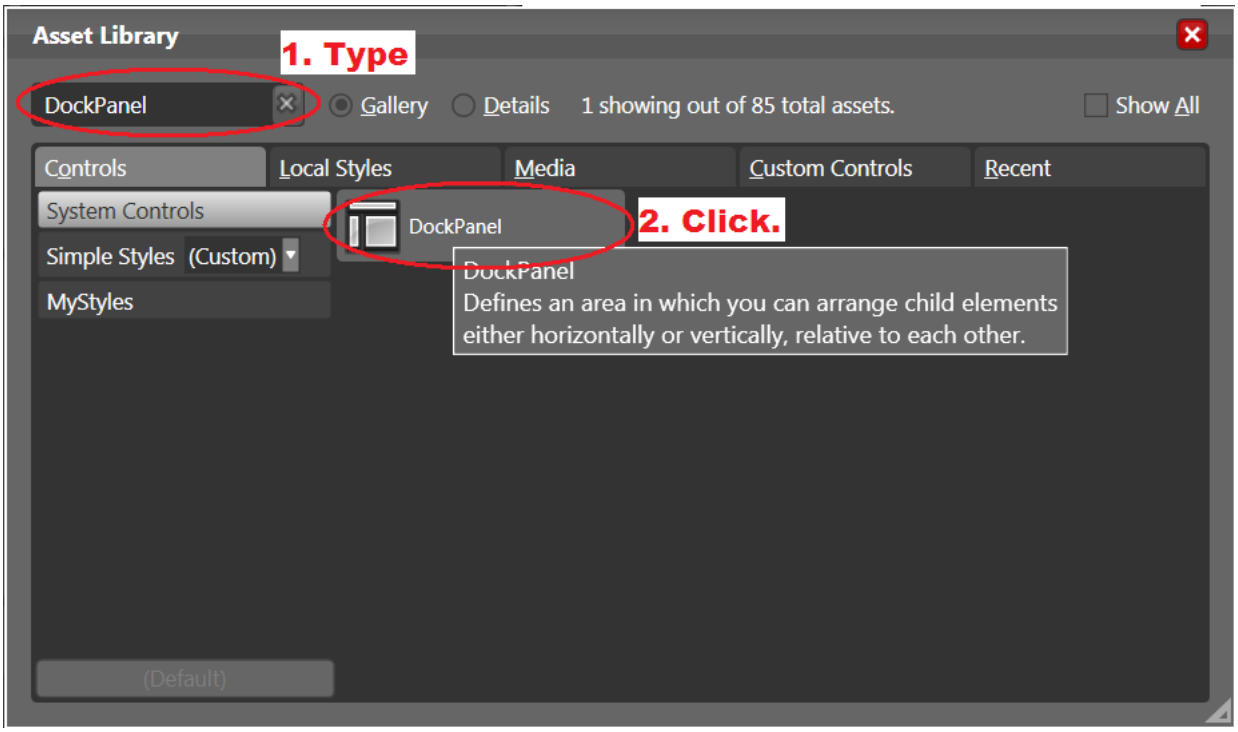


Figure 7 Adding the DockPanel tool to the ToolBox.

Now you can see the icon for the **DockPanel** tool in the **ToolBox.**

3. Add a **DockPanel** control to **LayoutRoot**. 8 9

There are two common ways to add a child control to a parent control, both variants require the parent control to be activated first. With the parent control activated, perform either of the following operations:

a. Click the tool for the child control in the **ToolBox**, move mouse to the position on the artboard where you want the control to be located, and then draw a control of a wanted size while holding down the left mouse button (see Figure 8).

b. Double-click the tool for the child control in the **ToolBox**, after that the control is automatically added to the currently active control with its default size (see Figure 9).

Now you should be able to add a **DockPanel** control to **LayoutRoot**.

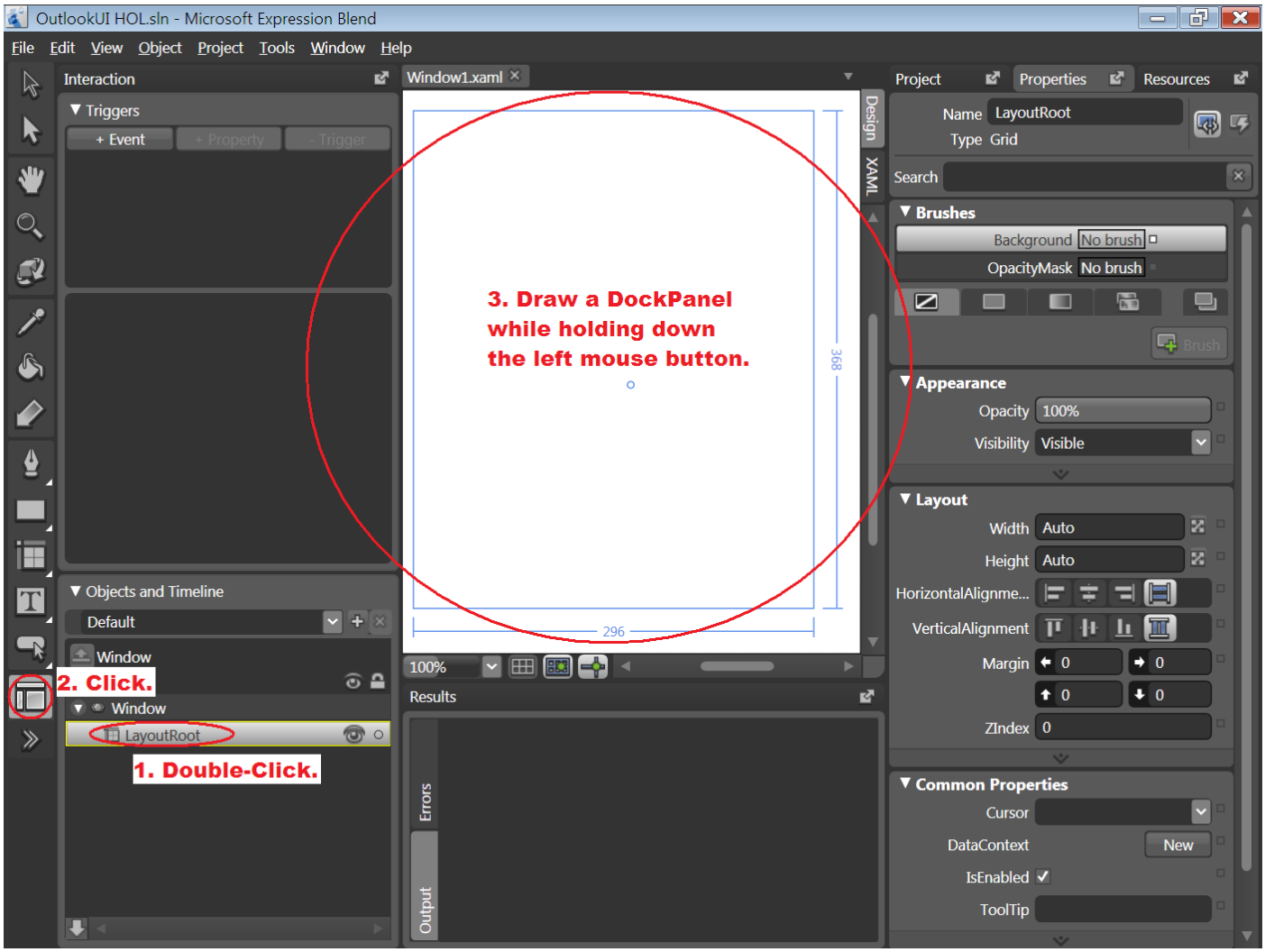


Figure 8 Adding a DockPanel control to LayoutRoot.

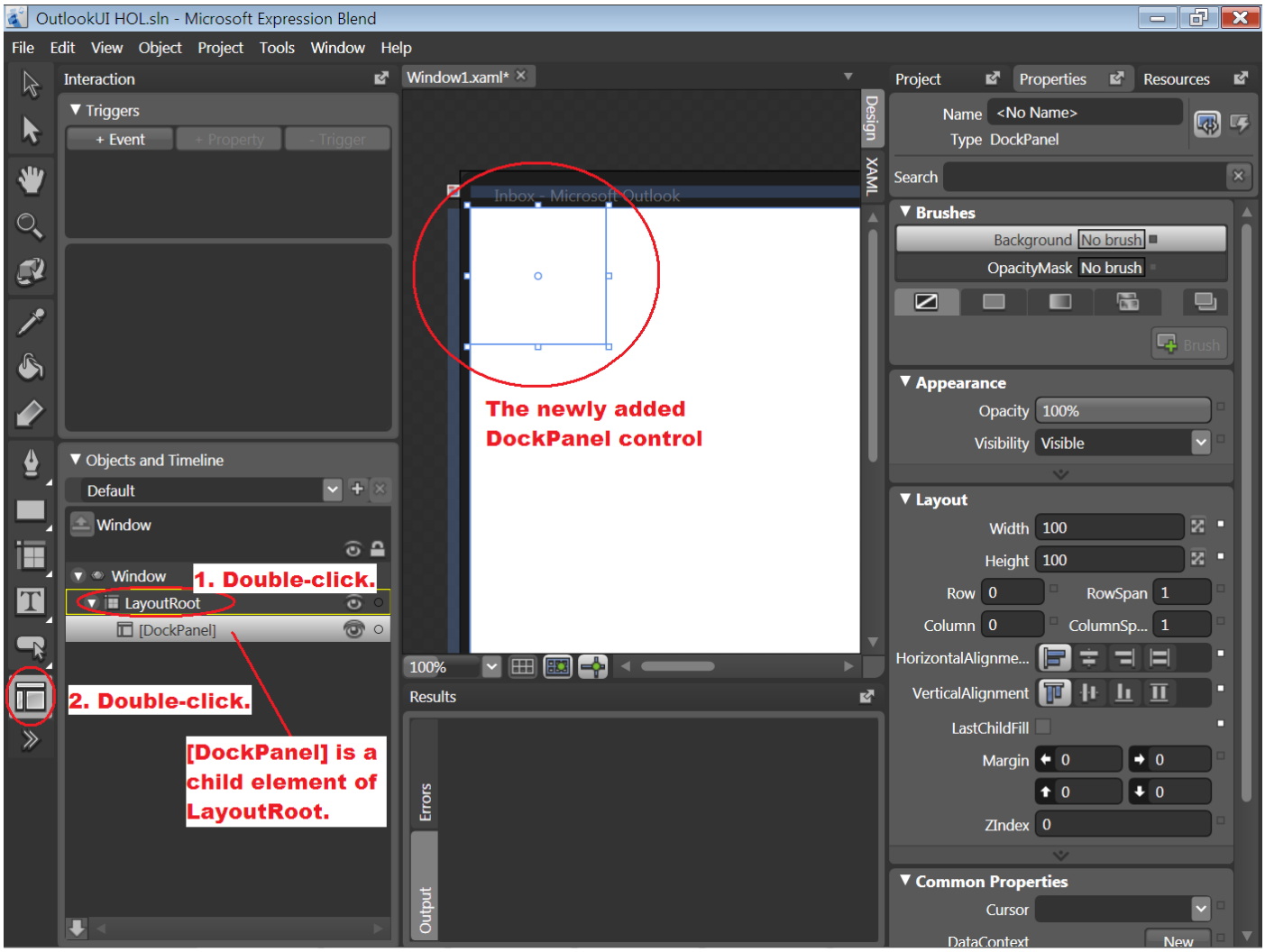


Figure 9 Adding a DockPanel control to LayoutRoot.

4. Configure the Layout properties of **[DockPanel]**. 10

Alignment, Margin and Padding

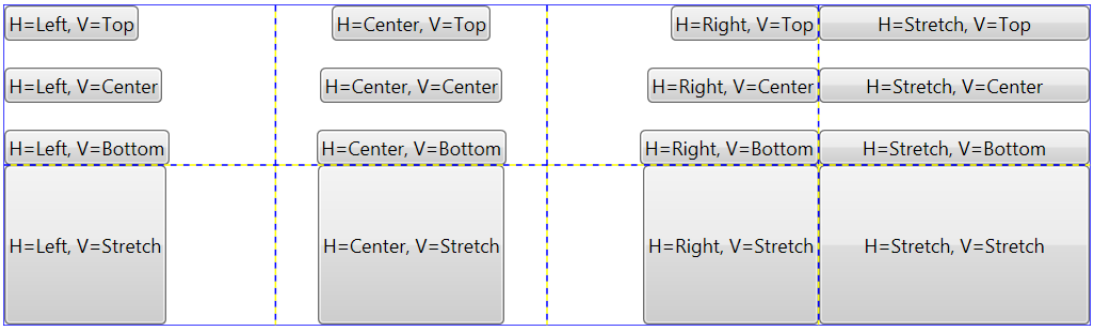
The **FrameworkElement** class exposes several properties that are used to precisely position child elements. This topic discusses four of the most important properties: **HorizontalAlignment**, **VerticalAlignment**, **Margin** and **Padding**.

Alignment Properties

The **HorizontalAlignment** and **VerticalAlignment** properties describe how a child element should be positioned within a parent element's allocated layout space. The **HorizontalAlignment** property declares the horizontal alignment characteristics to apply to child elements. The possible values of the **HorizontalAlignment** property are **Left**, **Center**, **Right** and **Stretch**.

The **VerticalAlignment** property describes the vertical alignment characteristics to apply to child elements. The possible values for the **VerticalAlignment** property are **Top**, **Center**, **Bottom** and **Stretch**.

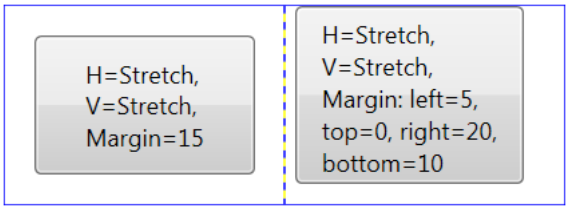
The following figure shows different value combinations of **HorizontalAlignment** (H) and **VerticalAlignment** (V) of the **Button** controls which are child elements of the **Grid** control, and each **Button** is positioned in a certain grid cell.



Margin Properties

The **Margin** property describes the distance between an element and its child or peers. Margin values can be uniform, by using syntax like **Margin="15"**. With this syntax, a uniform Margin of 15 device independent pixels would be applied to the element. Margin values can also take the form of four distinct values, each value describing a distinct margin to apply to the left, top, right, and bottom (in that order), like **Margin="0,10,5,25"**.

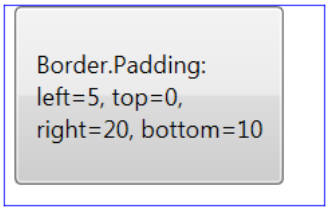
The following figure shows two **Button** controls which are child elements of a **Grid** control. The left **Button** has a uniform Margin value 15, and the right **Button** has four distinct Margin values with **Margin="5, 0, 20, 10"**.



Padding Property

The **Padding** property is exposed on only a few classes, primarily as a convenience: **Block**, **Border**, **Control**, and **TextBlock** are samples of classes that expose a **Padding** property. The **Padding** property enlarges the effective size of a child element by the specified **Thickness** value.

The following figure shows a **Border** control which contains a button. The **Padding** value of the **Border** control is **"5, 0, 20, 10"**.



Since we want **[DockPanel]** to be the root container of all our controls, we must make it cover the entire area of **LayoutRoot**. With **[DockPanel]** selected, go to the **Properties** panel, type "**Auto**" into the **Width** and **Height** text boxes, click the two "**Stretch**" buttons for the **HorizontalAlignment** and **VerticalAlignment** properties, and set the left, right, top, bottom margins to 0.

What we do here is actually resetting these properties to their default values, so you can also click the small square beside the property to open the context menu, and choose **Reset**, then the property's value is automatically set to the default value.

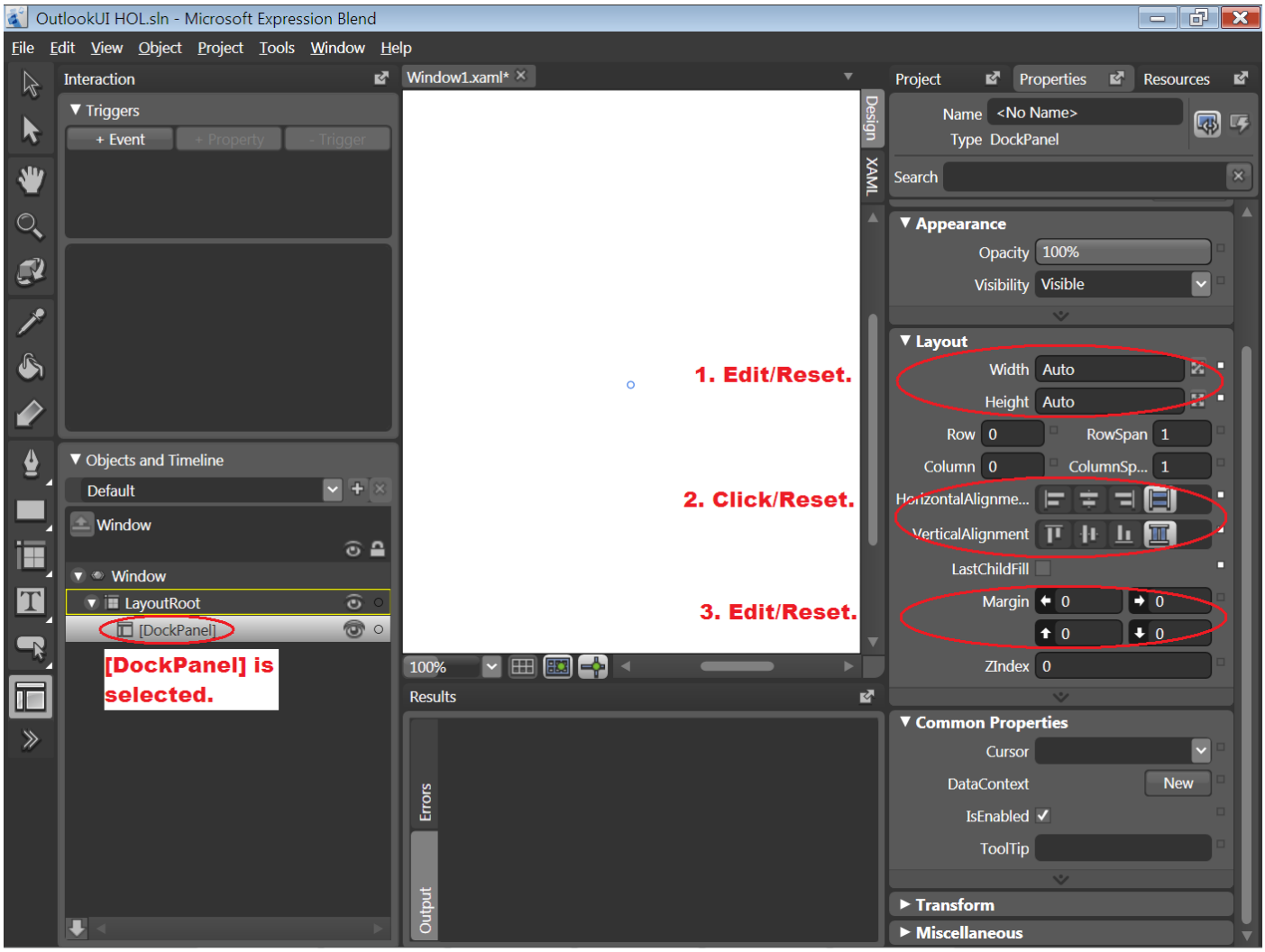


Figure 10 Configuring the properties of [DockPanel].

Now we have a **DockPanel** control, and we are going to fill the **[DockPanel]** control with a **ToolBarTray**, a **StatusBar** and a **Grid.**

Task 4: Adding a ToolBarTray, a StatusBar and a Grid to [DockPanel]

Our project is shipped with a custom **ToolBarTray** control and a custom **StatusBar** control, both are defined as user controls called **MyToolBarTrayControl** and **MyStatusBarControl**.

1. Add a **MyToolBarTrayControl** to **[DockPanel]**.

    1) Add the **MyToolBarTrayControl** tool to the **ToolBox**. 

    Open the **Asset Library** dialog box, select the **Custom Controls** tab, type **MyToolBarTrayControl** into the text box, and click **MyToolBarTrayControl**. Now you can see the tool in the **ToolBox**.

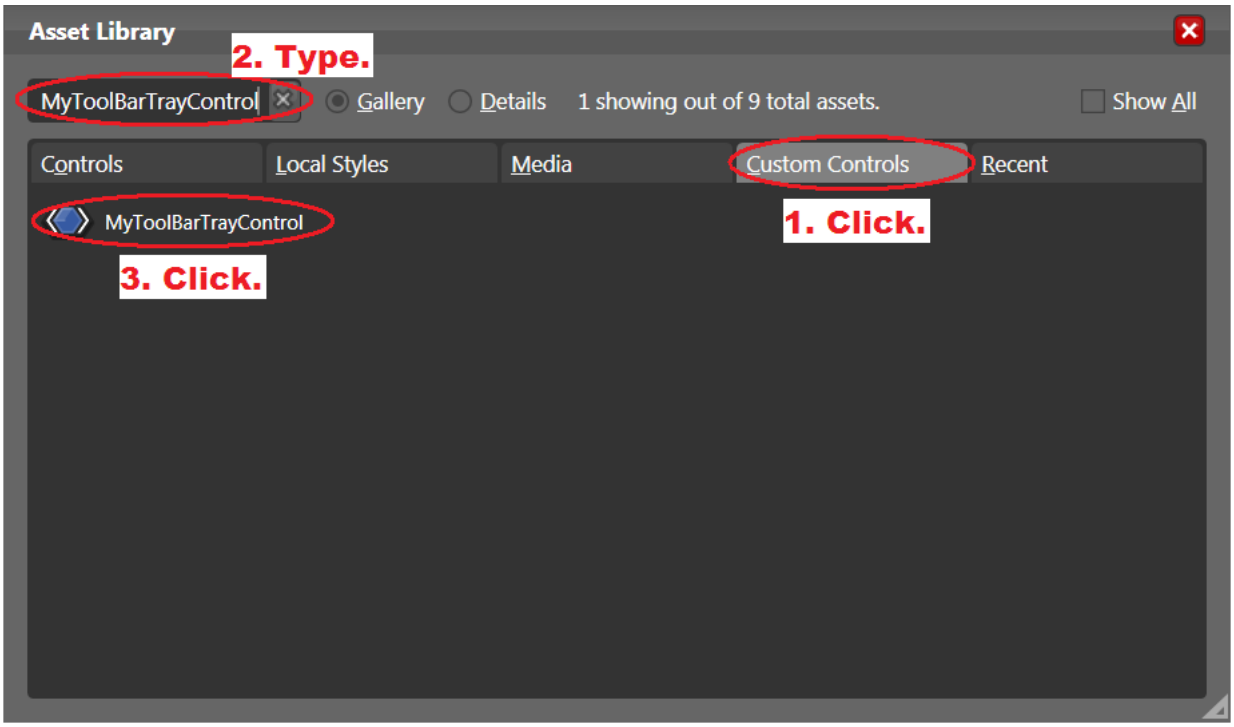


Figure 11 Adding the MyToolBarTrayControl tool to the ToolBox.

2) Add a **MyToolBarTrayControl** to **[DockPanel]** and set its **Dock** property to **Top**. 

Activate **[DockPanel]**, double-click the **MyToolBarTrayControl** tool in the **ToolBox**, and set the **Dock** property of **[MyToolBarTrayControl]** to **Top**.

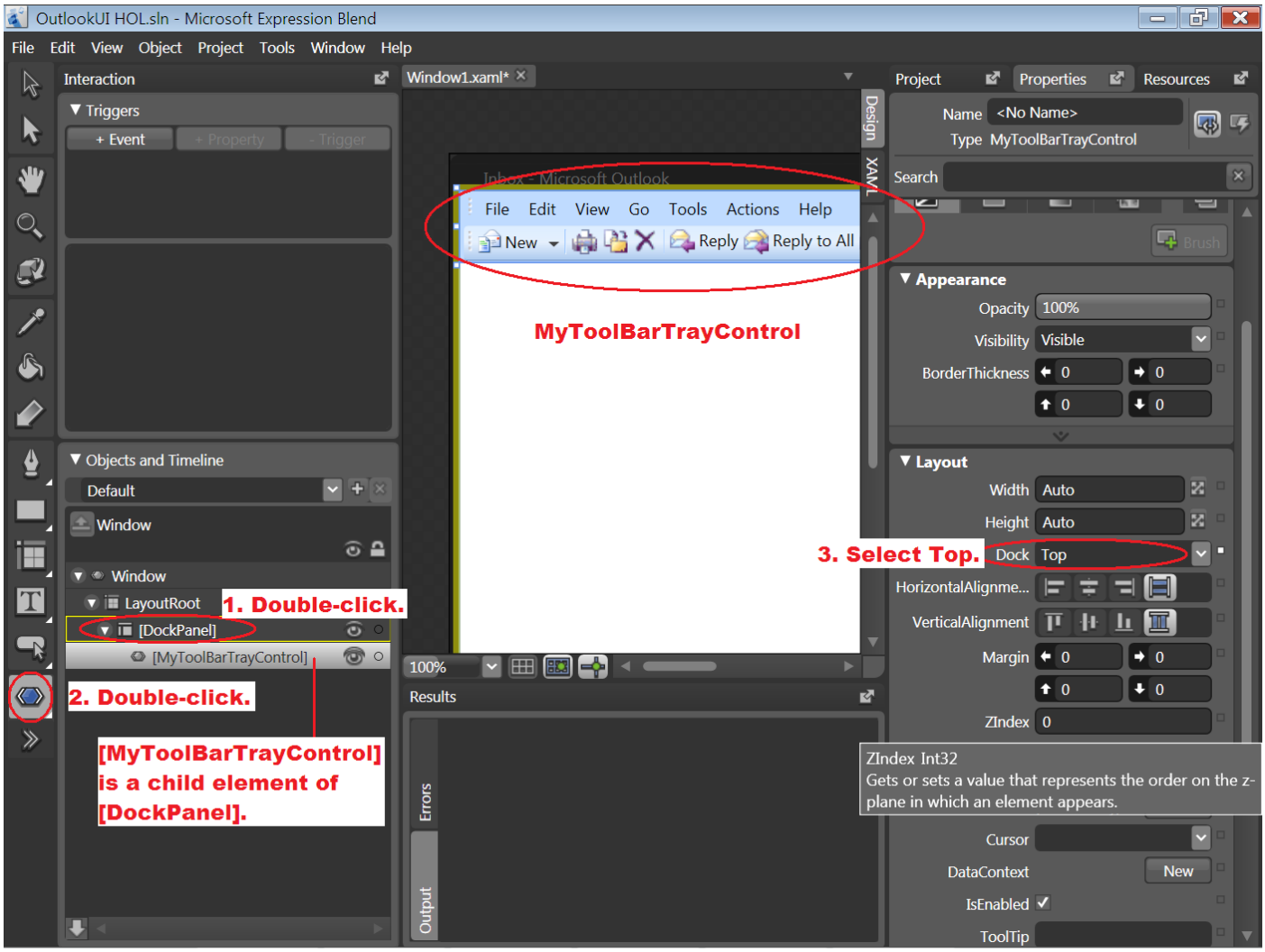


Figure 12 Adding a MyToolBarTrayControl control to [DockPanel].

2. Add a **MyStatusBarControl** to **[DockPanel]**.

   With **[DockPanel]** activated,

   1) add the **MyStatusBarControl** tool to the **ToolBox**.
   2) add a **MyStatusBarControl** to **[DockPanel]** and set its **Dock** property to **Bottom**.

3. Add a **Grid** control to **[DockPanel]**. 

   A **Grid** is a very commonly used layout container which consists of columns and rows, and a child control can be positioned into one cell or span more than one column resp. row. Let's add a **Grid** control to **[DockPanel]**:

   1) With **[DockPanel]** activated, double-click the **Grid** tool in the **ToolBox**.
   2) Reset the **Width** property of **[Grid]** and set its **Dock** property to **Bottom**.

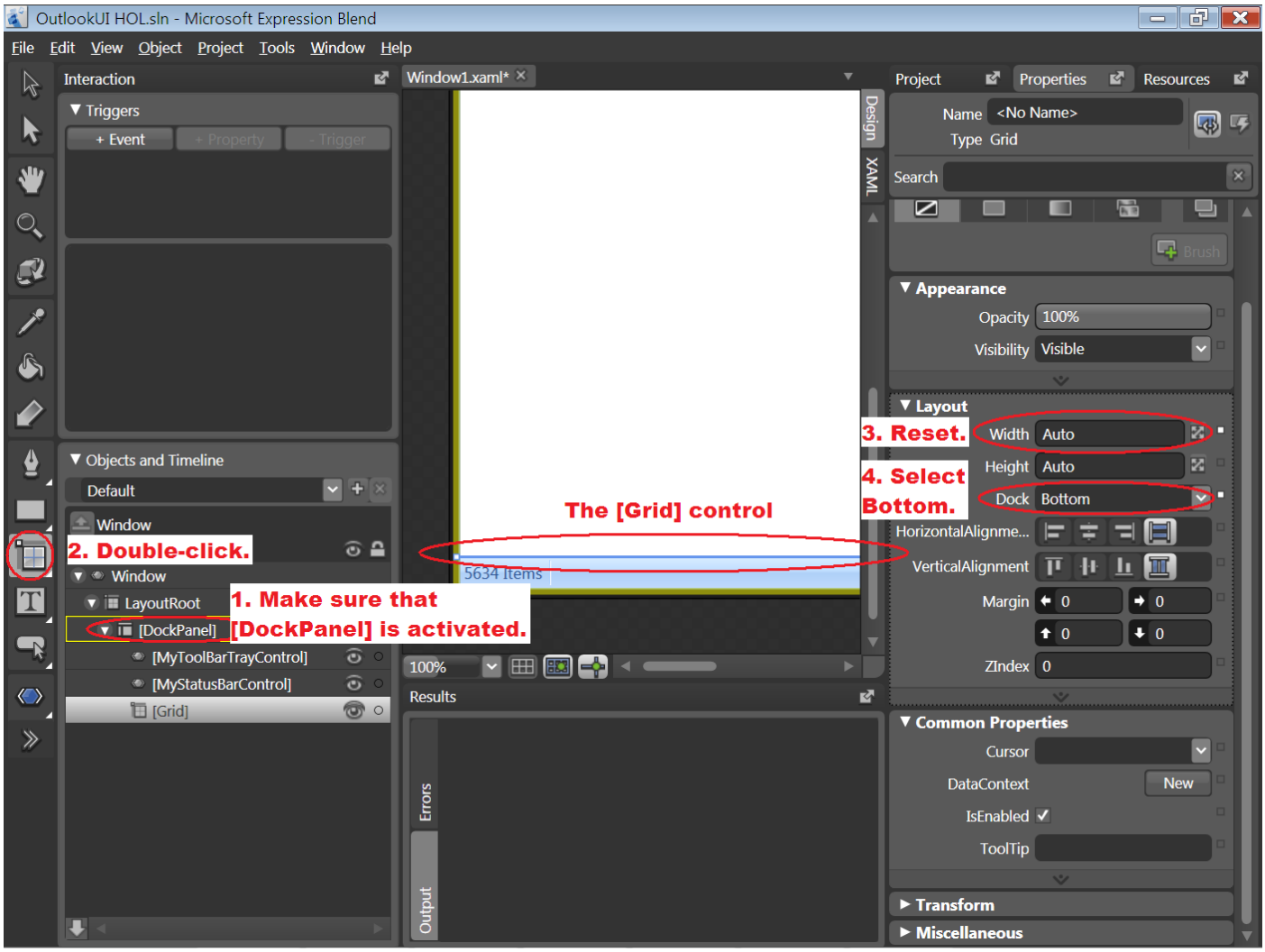


Figure 13 Adding a Grid control to [DockPanel].

4. Set **[DockPanel].LastChildFill** to **True**. 14

   At this moment, the **[Grid]** control does not entirely fill the area between the toolbar and the status bar. To make this effect, we go back to the **Properties** of the **[DockPanel]** control, and check the **LastChildFill** property.

DockPanel.LastChildFill

If you set the **DockPanel.LastChildFill** property to **true**, the last child element of a **DockPanel** always fills the remaining space, regardless of any other dock value that you set on the last child element. In our case, **[Grid]** is added to **[DockPanel]** as the last child element, so it will fill the area between **[MyToolBarTrayControl]** and **[MyStatusBarControl]**.

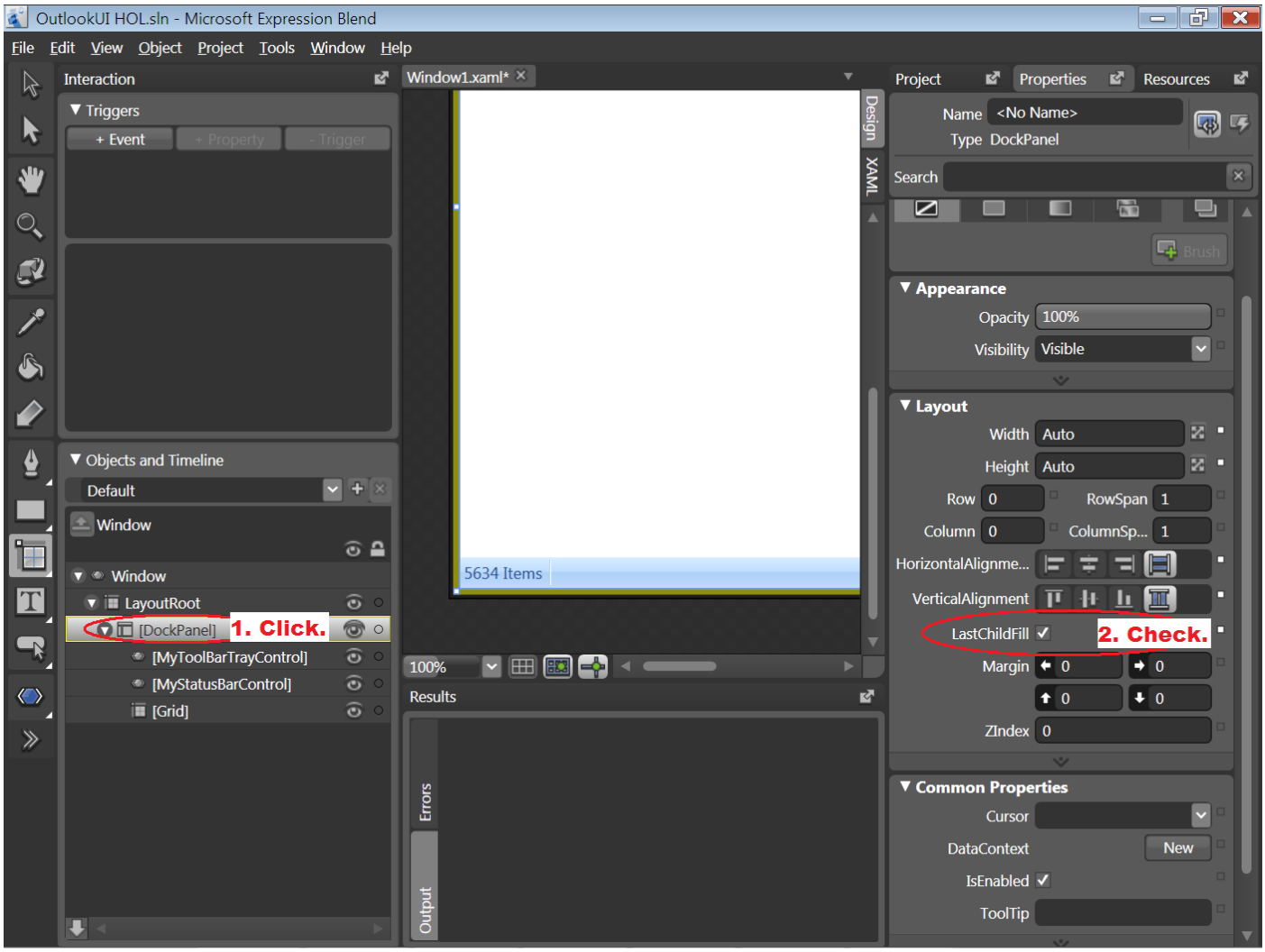


Figure 14 Setting the LastChildFill property of the [DockPanel] control to True.

5. Group **[Grid]** into a **Border** control to add a blue border around **[Grid]**. 15

   Under **Objects and Timeline**, right-click **[Grid]** to open the context menu, select **Group Into** > **Border**.

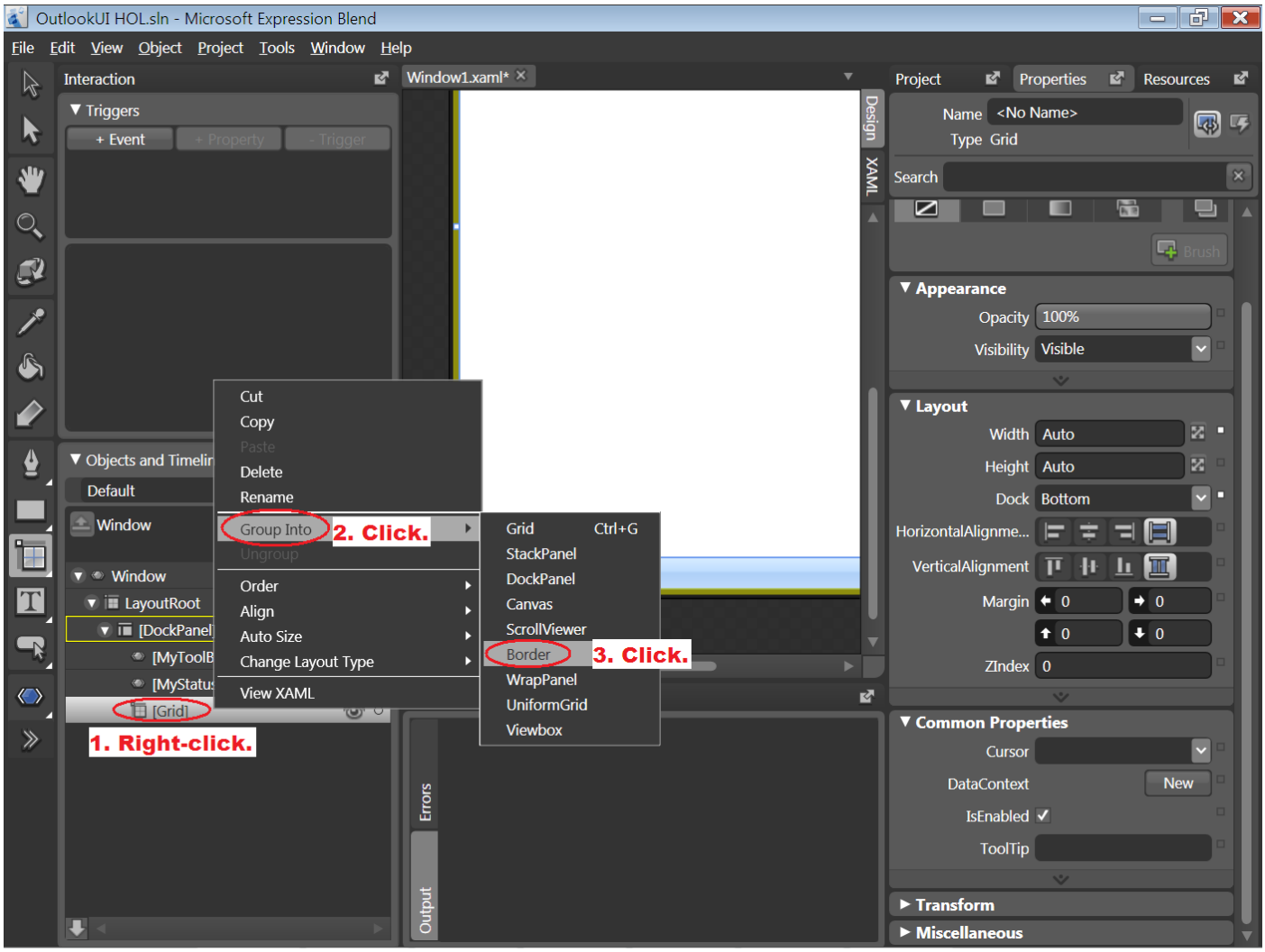


Figure 15 Grouping the [Grid] control into a Border.

6. With **[Border]** selected, configure the properties of **[Border]** as follows:  **16**

Brushes

- **BorderBrush = MyBorderBlueGradientBrush**

Appearance

- **BorderThickness: left = 5, right = 5, top = 3, bottom = 5**

Layout

- **Width = Auto, Height = Auto**
- **Dock = Bottom**

The controls' hierarchy has been updated like the following: **[DockPanel]** > **[Border]** > **[Grid]**. That's why we must specify the **Dock** property for **[Border]** instead of for **[Grid]** now.

- HorizontalAlignment = Stretch, VerticalAlignment = Stretch
- Margin = 0, 0, 0, 0

For the sake of completeness, we list the commonly used properties of the control being edited, regardless of whether the property value is the default or not. You only need to pay attention to the properties which are assigned custom values, these properties are in bold.

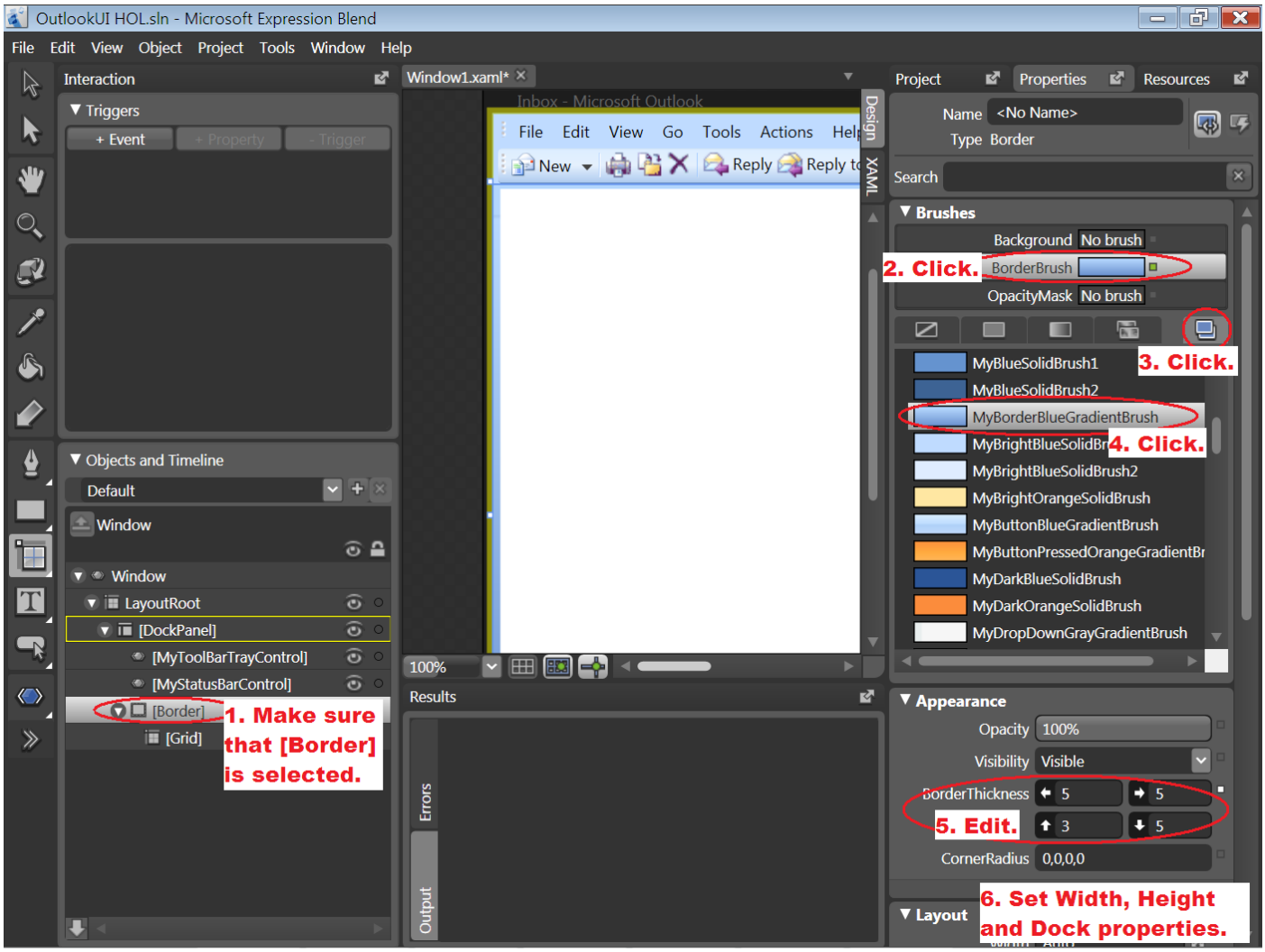


Figure 16 Configuring the properties of [Border].

7. With **[Grid]** selected, configure the properties of **[Grid]** as follows:

   Layout

   - **Width = Auto, Height = Auto**

Task 5: Dividing [Grid] into Columns and Rows & Adding GridSplitters to [Grid]

The next step is to divide **[Grid]** into three columns and two rows and then add **GridSplitter**s to **[Grid]**.

1. Divide **[Grid]** into three columns and two rows. 17
    1) Under **Objects and Timeline**, double-click **[Grid]** to activate it.

Press **F4** to get the whole picture of **[Grid]**. To return to the original view, press **F4** again.

2) Click the **Selection** tool in the **ToolBox**.
3) On the artboard, move the mouse pointer over the thick blue ruler area at the top of **[Grid]**. An orange column ruler will follow your mouse pointer and indicate where a new column divider would be placed should you click. Click to create a new column divider. A blue column divider appears inside **[Grid]**.
4) Create a second column divider.
5) Move the mouse pointer over the thick blue ruler area at the left of **[Grid]**, click to create a row divider.

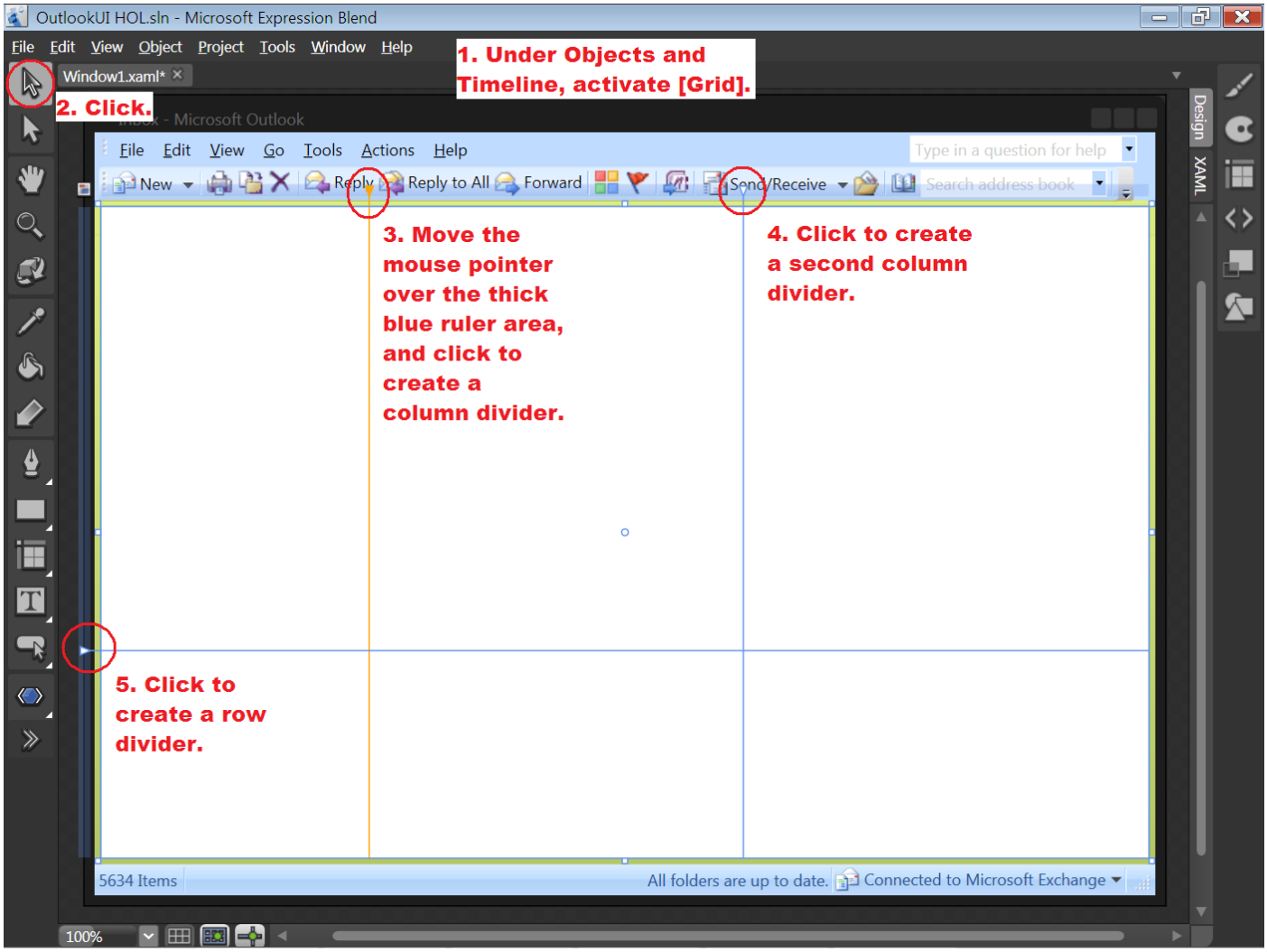


Figure 17 Creating two column dividers and one row divider for [Grid].

Since we want the user to be able to adjust the column width and the row height at runtime, we must add **GridSplitter**s to **[Grid]**.

2. Add a vertical **GridSplitter** control on the left side of the second column of **[Grid]**.  **18**

    1) Activate **[Grid]** if it is not active.
    2) Add the **GridSplitter** tool from the **Asset Library** to the **ToolBox**.

       This time when you open the **Asset Library** dialog box, the current tab is **Custom Controls**. Since the **GridSplitter** tool is to be found under the **Controls** category, you must first switch to **Controls** tab to get the **GridSplitter** tool.
    3) Click the **GridSplitter** tool in the **ToolBox**.
    4) On the artboard, draw a vertical **GridSplitter** control on the left side of the second column.
    5) Configure the properties of **[GridSplitter]** as follows:

        - **Name = myMainGridFirstColumnSplitter**

Unique control name for programmatic reference

In WPF, not all the controls have to be named. Under **Objects and Timeline**, unnamed controls are shown as **[*Control*]**.

However, if you want to refer to a control programmatically in VB/C# code, then you must give this control a unique name. In our case, as you will see, we must write event-handling code for this **GridSplitter**'s events to enable a more advanced behavior, that's why it must be named.

XAML and C# code are case-sensitive

Since XAML and C# code are both case-sensitive, you must pay attention to lowercase and uppercase letters while editing XAML files resp. C# code.

Brushes

- **Background = MyBorderBlueGradientBrush**

Layout

- **Width = 5, Height = Auto**
- **Row = 0** (zero-based index), **RowSpan = 2**, **Column = 1**, **ColumnSpan = 1**

Grid.RowSpan and Grid.ColumnSpan

The **Grid.RowSpan** property gets or sets a value that indicates the total number of rows that child content spans within a **Grid**. The **Grid.ColumnSpan** property gets or sets a value that indicates the total number of columns that child content spans within a **Grid**.

- **HorizontalAlignment = Left** (this configures the **[GridSplitter]** element to line up with the left edge of the second column), **VerticalAlignment = Stretch**
- **Margin = 0, 0, 0, 0**

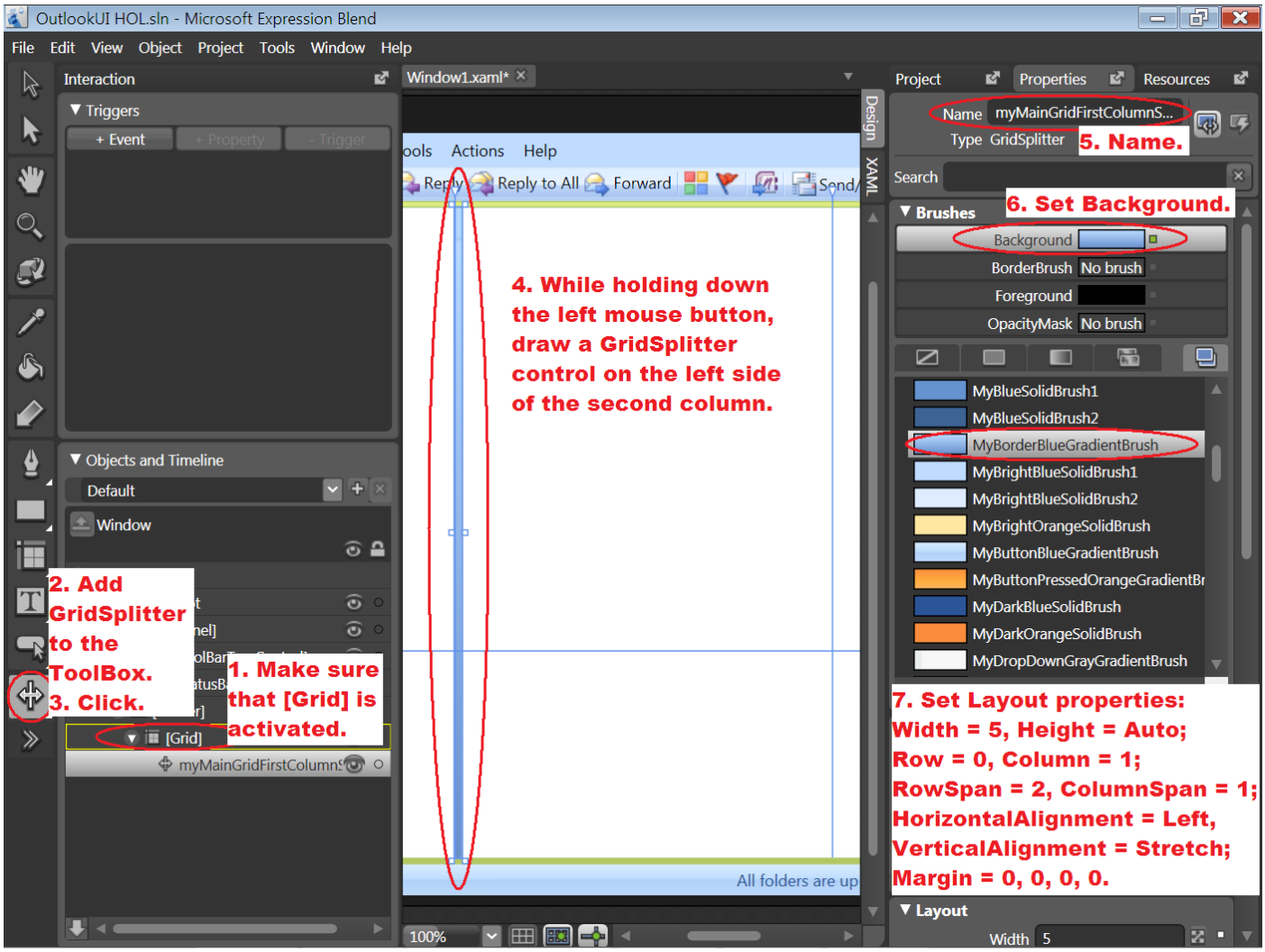


Figure 18 Adding a GridSplitter control to [Grid] on the left side of the second column.

3. Add a second vertical **GridSplitter** control on the right side of the second column of **[Grid]**, and then add a horizontal **GridSplitter** control in the first column and at the bottom of the first row of **[Grid]**.

   As you already know how to add a **GridSplitter** to **[Grid]**, you don't need to take time for the second and the third one. We have prepared XAML snippet which you can copy and then paste directly into the **Window1.xaml** file. The following steps show you how to do this:

   1) Switch to the **XAML View** of **[Grid]**.  **19**

      Under **Objects and Timeline**, right-click **[Grid]** to open the context menu, choose **View XAML** to switch to the **XAML View**.

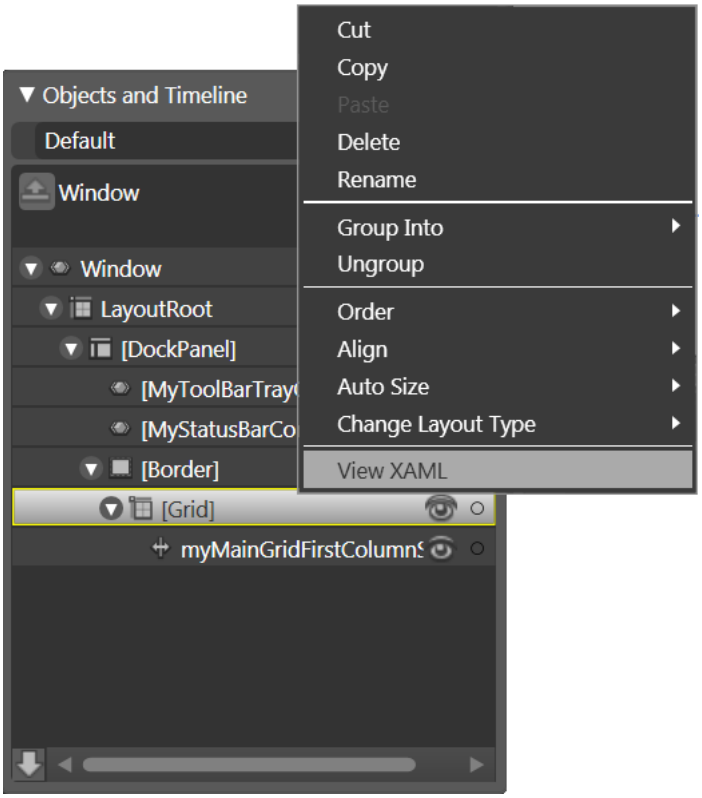


Figure 19 Switch to the XAML View of [Grid].

As you see, the XAML code for the **[Grid]** element is highlighted in blue in the background. Press **F4** to get a larger view of the file.

2) Add two **GridSplitter** elements to the **[Grid]** element. 

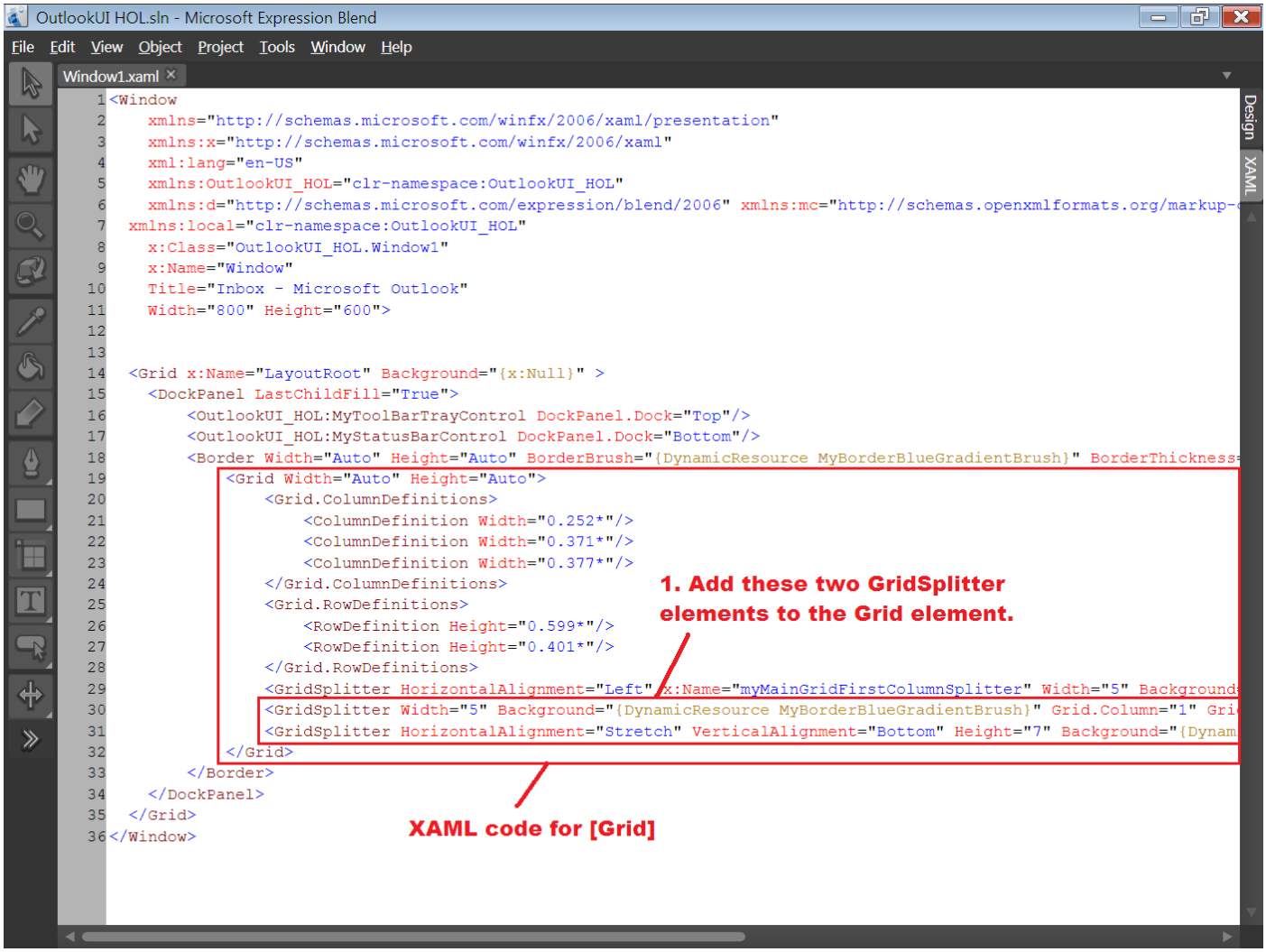


Figure 20 Adding two GridSplitter elements to [Grid].

Generating XAML in Design view or in XAML view

In **Design** view, you can design your application visually and let Expression Blend generate the XAML for you. Editing XAML directly may result in parsing errors in your application that you will have to fix before Expression Blend can correctly display your documents on the artboard in **Design** view or before you can build and run the application. With this caution in mind, working between **Design** view and **XAML** view can be an effective method for learning the basics of XAML.

In **XAML** view, you can type in the code for new elements, or you can select existing code and then cut or copy from it or paste into it, just as you would in a word processing program. You can also go to specific lines in the XAML or find and replace text, by using the **Go To**, **Find**, **Find Next**, and **Replace** commands on the **Edit** menu.

Changes that you make in **XAML** view are automatically updated in **Design** view, and changes that you make in **Design** view are automatically updated in **XAML** view.

Expression Blend updates syntax errors in the **Results** pane when you save the file or when you switch between views (**Design** view and **XAML** view).

Add the following XAML code for these two **GridSplitter** elements after the first **GridSplitter** element:

basic5.txt

```
<GridSplitter Width="5" Background="{DynamicResource MyBorderBlueGradientBrush}"
Grid.Column="1" Grid.RowSpan="2"/>
```

```
<GridSplitter HorizontalAlignment="Stretch" VerticalAlignment="Bottom" Height="7"
Background="{DynamicResource MyHGridSplitterBlueGradientBrush}"/>
```

Switch to **Design View** and press **F4** again to return to the normal user interface of **Expression Blend**.

It's time to build and run your application! Press **F5** and wait until your first WPF application comes into sight. Your application should look like Figure 21. Move mouse over the toolbar, click the mouse button on the menus, on the buttons, on the images and in the text boxes, drag the vertical **GridSplitter**s left and right, drag the horizontal **GridSplitter** up and down.

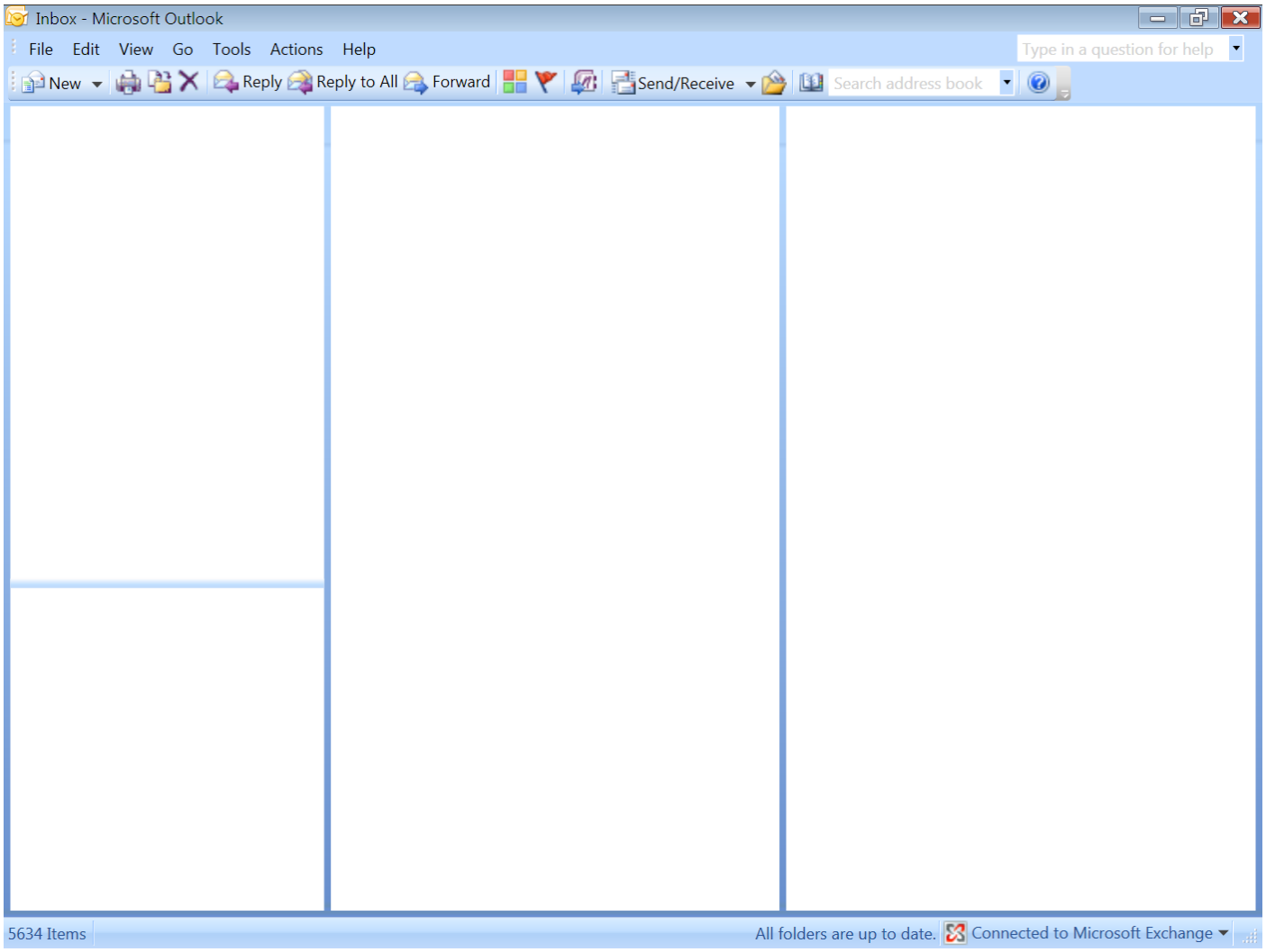


Figure 21 The application's user interface after task 5.

Task 6: Adding a StackPanel and its Child Controls to the Bottom Left Cell of [Grid]

In this task, you will learn to add a **StackPanel** control and its child controls to the bottom left cell of **[Grid]** (see Figure 22).

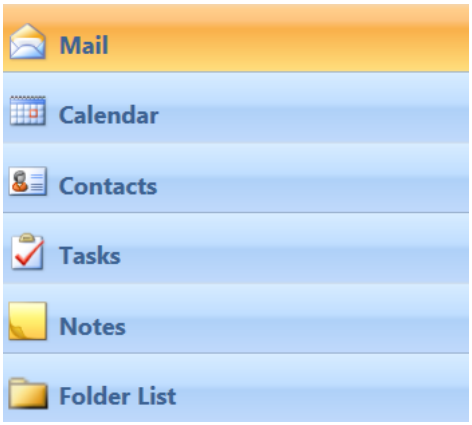


Figure 22 The StackPanel with its child controls.

The structure of the above element looks like the following (see Figure 23): A vertical **StackPanel** control sits at the topmost level, it contains 6 buttons. Each button contains again a horizontal **StackPanel** control which is used to align the **Image** and the **TextBlock** controls.

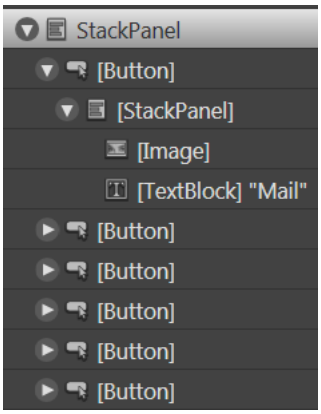


Figure 23 Hierarchical control structure of the StackPanel and its child controls.

1. Add a **StackPanel** control to **[Grid]** and configure its properties as follows:  24

   Layout

   - **Width = Auto, Height = Auto**
   - **Row = 1**, RowSpan = 1, Column = 0, ColumnSpan = 1
   - Orientation = Vertical
   - **HorizontalAlignment = Stretch, VerticalAlignment = Top**
   - **Margin = 0, 0, 0, 0**

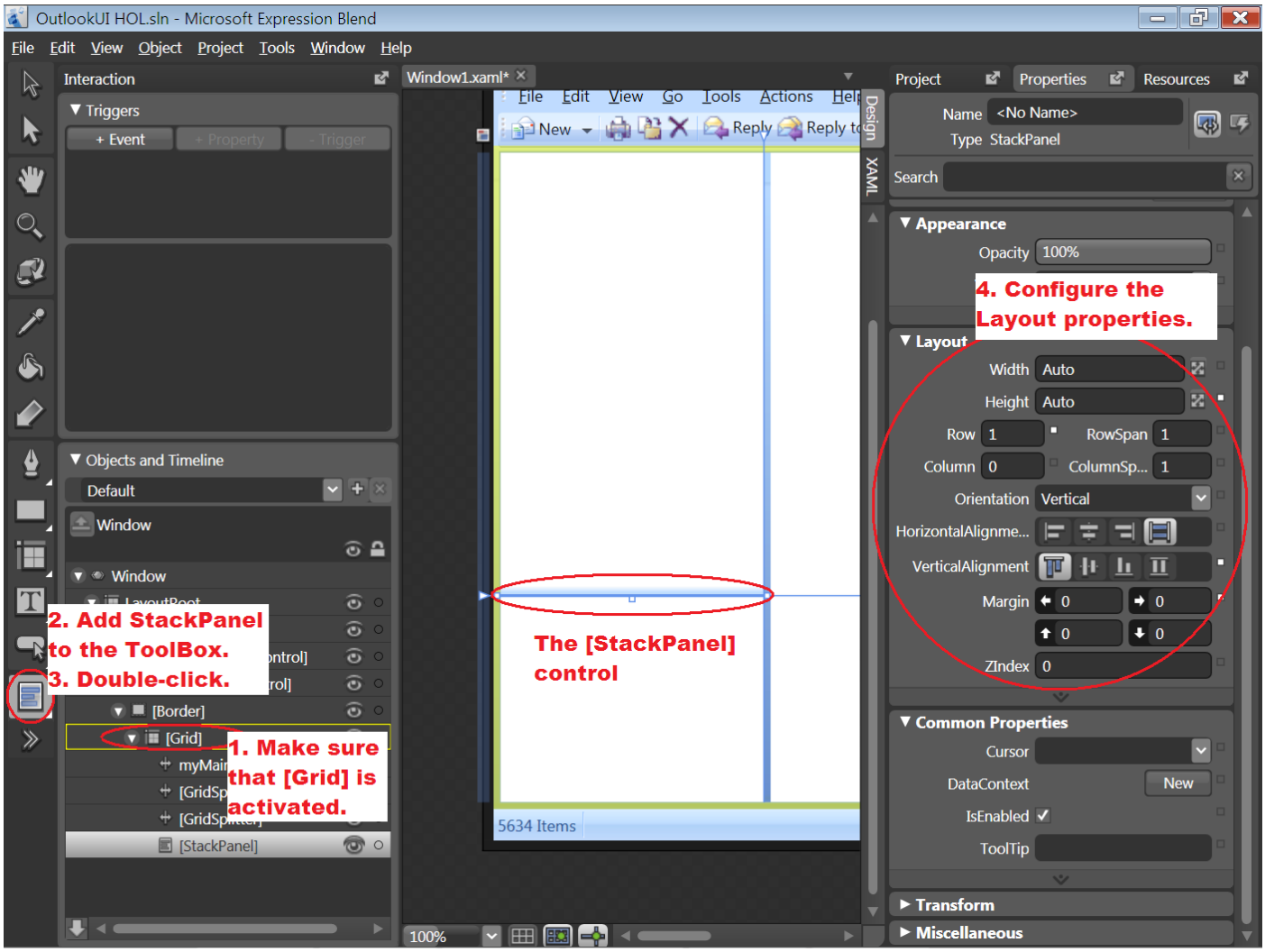


Figure 24 Adding a StackPanel control to the [Grid] control.

2. Add a **Button** control to **[StackPanel]** and configure its properties as follows: 25

Brushes

- **Background = MySelectedButtonOrangeGradientBrush**
- **BorderBrush = MyBlueSolidBrush2**

Appearance

- **BorderThickness: left = 0, right = 0, top = 0.2, bottom = 0.2**

Layout

- Width = Auto, **Height = 35**
- HorizontalAlignment = Stretch, VerticalAlignment = Stretch
- Margin = 0, 0, 0, 0
- **HorizontalContentAlignment = Left**, VerticalContentAlignment = Center

  To see these two properties you need to expand the **Layout** category by clicking the arrow under the **ZIndex** property.

Control.HorizontalContentAlignment and Control.VerticalContentAlignment properties

The **Control.HorizontalContentAlignment** property gets or sets the horizontal alignment of a control's content, similar to **FrameworkElement.HorizontalAlignment**, it also has four possible values: **Left**, **Center**, **Right** and **Stretch**.

The **Control.VerticalContentAlignment** property gets or sets the vertical alignment of a control's content, similar to **FrameworkElement.VerticalAlignment**, it also has four possible values: **Top**, **Center**, **Bottom** and **Stretch**.

- Padding: **left = 3**, right = 1, top = 1, bottom = 1

  To see this property you also need to expand the **Layout** category.

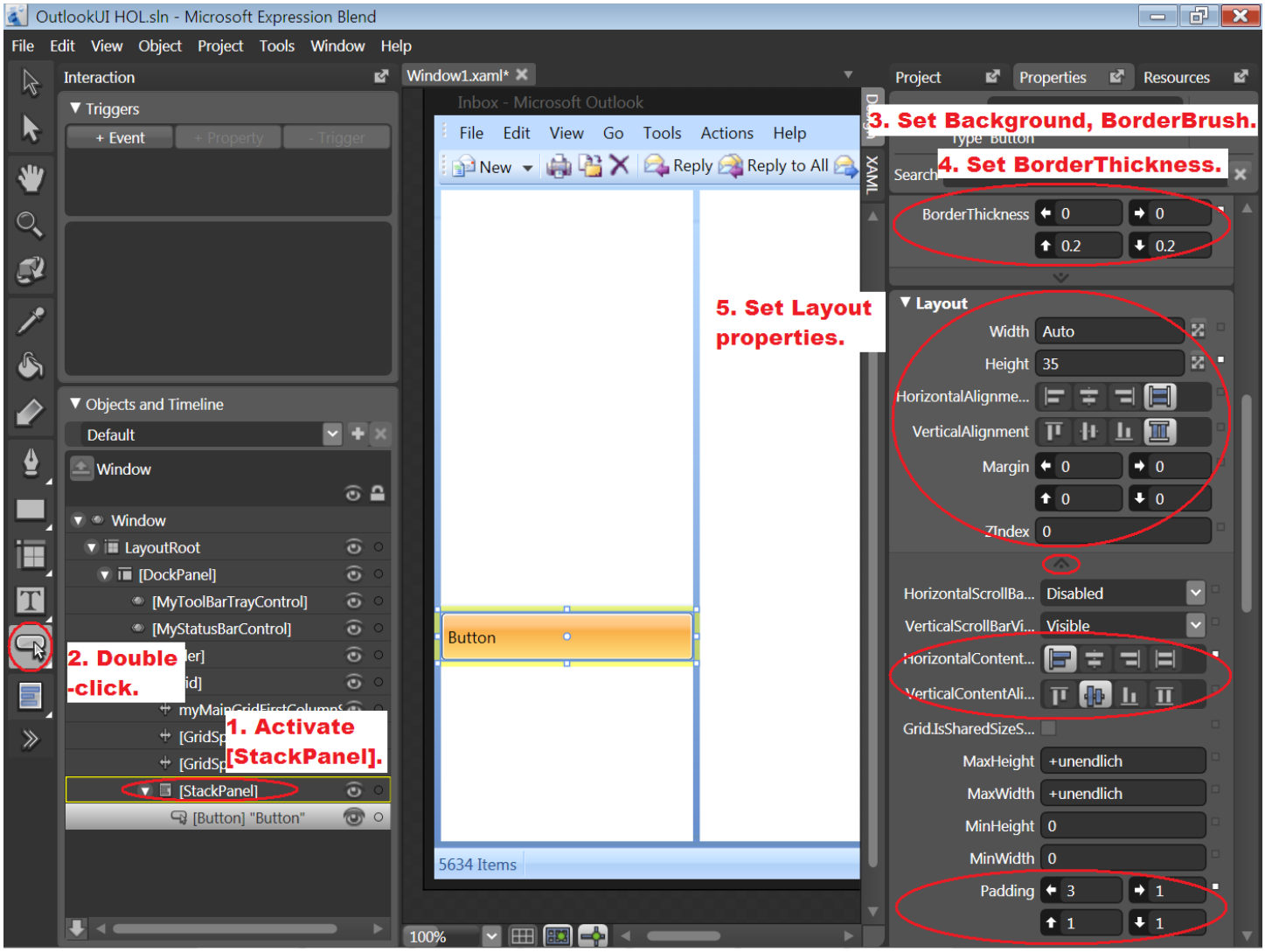


Figure 25 Adding a Button control to [StackPanel].

In order that the **[Button]** controls have the same look and feel as the one in MS Outlook 2007, we have defined our own style called **MyBottomLeftButtonStyle** for these buttons.

Style

A **style** allows you to define different property values for different states which a control is currently in. For example, in case of a button, we can set its **Background** property to Blue for its **Default** state, and we can also set its **Background** property to Orange for the case that the user moves mouse over it, and finally, the **Background** property can be Red, when the user clicks it.

Do the following to set the **Style** property of **[Button]** to **MyBottomLeftButtonStyle:**

Miscellaneous

- **Style = MyBottomLeftButtonStyle** 

  Expand the **Miscellaneous** category, click the small square beside the **Style** property to open the context menu, click **Local Resource** and select **MyBottomLeftButtonStyle**.

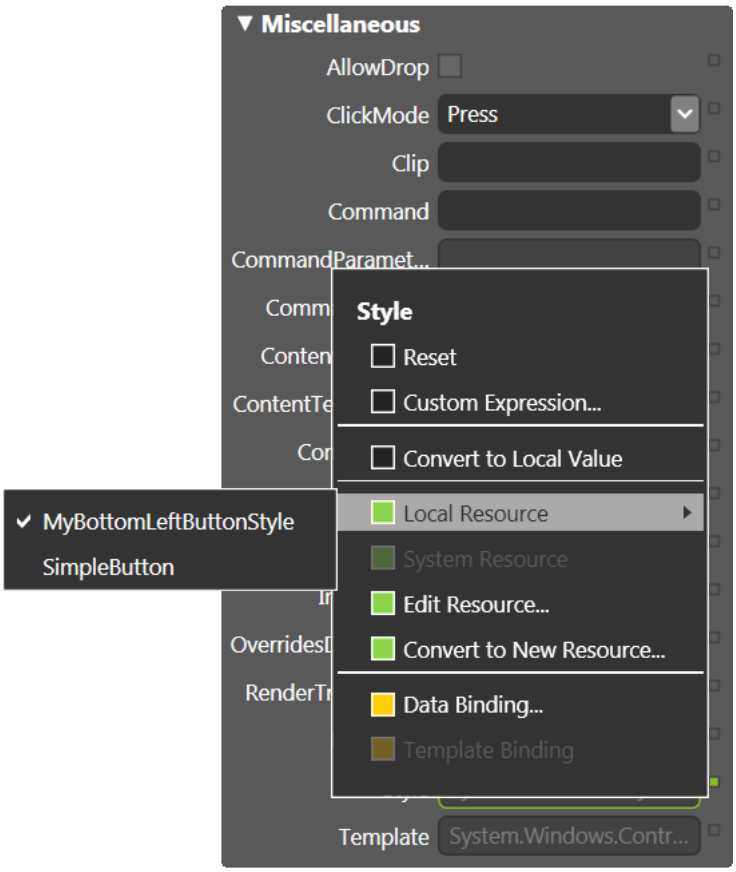


Figure 26 Setting the Style property of the [Button] control to MyBottomLeftButtonStyle.

3. Add a **StackPanel** control to **[Button]**.

   First activate **[Button]**, then double-click the **StackPanel** tool  in the **ToolBox**, then configure the properties of **[StackPanel]** as follows:

   Layout

   - **Width = Auto, Height = 26**
   - **Orientation = Horizontal**

   The **[StackPanel]** control added in this step should contain two child elements: one **Image** control and one **TextBlock** control.

4. Add an **Image** control to **[StackPanel]** created in step 3.
   1) Activate **[StackPanel]** located under **[Button]**.
   2) Switch to the **Project** tab.
   3) Under **OutlookUI HOL**, expand the **graphics** folder, and double-click the image file **mail.ico**.
   4) Configure the properties of the **[Image]** control as follows:

      Layout

      - **Width = 21, Height = 21**
      - **Margin**: left = 0, right = 0, **top = 5**, bottom = 0

5. Add a **TextBlock** control to [**StackPanel]** created in step 3.
    1) Activate **[StackPanel]** located under **[Button]** if it is not active.
    2) Select the **TextBlock** tool in the **ToolBox**:  **27**

       Right-click the small triangle included in the **TextBox** tool in the **ToolBox** to get a list of other similar assets, and then select **TextBlock**.

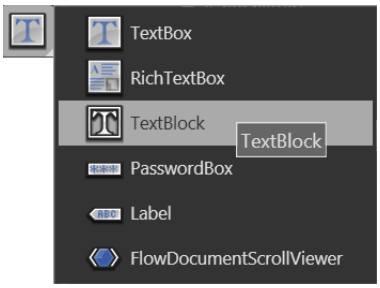


Figure 27 Selecting the TextBlock tool.

    3) Double-click the **TextBlock** tool in the **ToolBox** to add a **TextBlock** control to **[StackPanel]**.
    4) Configure the properties of the **[TextBlock]** control as follows:

Expression Blend does not properly display the **Properties editor** right after you have added a **TextBlock** control. Just select another control, and then select the **TextBlock** control again to edit its properties. Remember this trick.

Brushes

- **Foreground = MyBlueSolidBrush2**

Layout

- **HorizontalAlignment = Left, VerticalAlignment = Center**
- **Margin**: **left = 4**, right = 0, **top = 7**, bottom = 0

Common Properties

- **Text = Mail**

Text

- **FontWeight = Bold**

6. Add another five buttons.

   Since the six buttons have the same structure and they only differ in the **Image.Source** property and the **TextBlock.Text** property, we can use Copy & Paste technique to save us a lot of work:
    1) Under **Objects and Timeline**, right-click **[Button]** to get the context menu displayed, click **Copy**.
    2) Activate **[StackPanel]** in which **[Button]** is contained.
    3) Right-click **[StackPanel]** to get the context menu displayed, click **Paste**.
    4) Repeat step 3) for another four times.
    5) Set the **Background** property to **MyButtonBlueGradientBrush** and reset the **Width** property for the last five buttons.  **28**

One nice thing in **Expression Blend** is that you can configure common properties of several selected controls in one go.

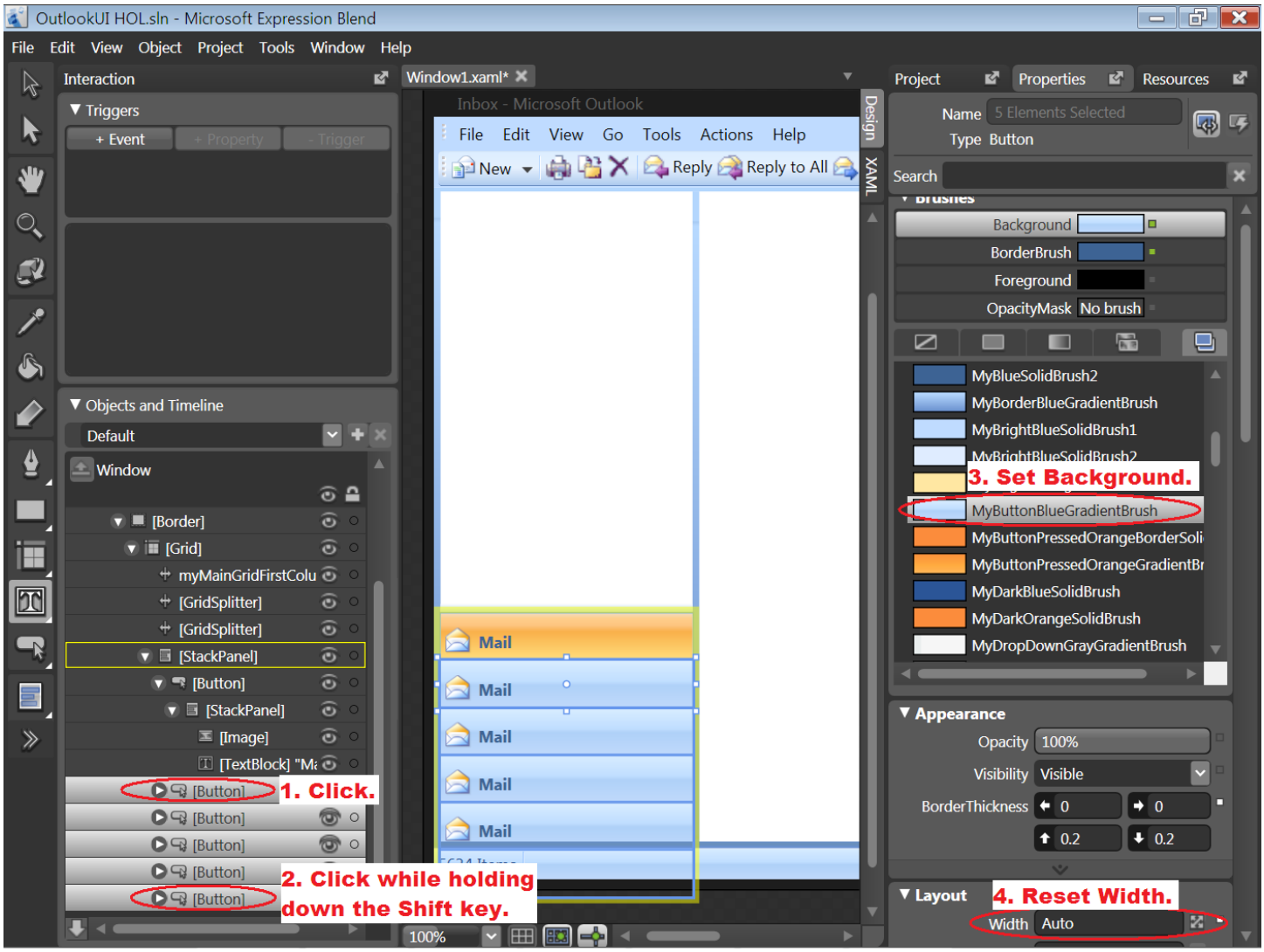


Figure 28 Setting the Background and Width properties of the last five buttons.

We suggest you to do the following optional step after you have completed the basic resp. advanced course: correcting the **Image.Source** and the **TextBlock.Text** properties for the last five buttons. The properties' values are:

- Image.Source = graphics\calendar.ico, TextBlock.Text = Calendar
- Image.Source = graphics\contacts.ico, TextBlock.Text = Contacts
- Image.Source = graphics\tasks.ico, TextBlock.Text = Tasks
- Image.Source = graphics\notes.ico, TextBlock.Text = Notes
- Image.Source = graphics\folder.ico, TextBlock.Text = Folder List

Build and run the project (**F5**). Drag the horizontal **GridSplitter** up and down, as you see, there are three problems with the dynamic behavior of the **[StackPanel]** control:

1. With great probability only part of the button at the bottom is displayed.
2. The horizontal **GridSplitter** updates its vertical position pixel by pixel when you drag it, so that the buttons are not always displayed entirely.
3. The **[StackPanel]** control does not stick to the bottom when you drag the horizontal **GridSplitter** high enough.

The first problem can be solved by making the second row of **[Grid]** high enough to let the whole **[StackPanel]** be displayed when the application starts up (see steps 1-4 described below).

To solve the second problem, do the following: Go to the **Properties** editor of the horizontal **GridSplitter** control, under **Common Properties** category, set its **DragIncrement** property to the **Height** value of our buttons, that is, **35**.

Data binding shows its strength when we try to solve the third problem.

Data Binding

Data binding is the process of connecting a data source to user interface components. This means that whenever the data changes, the interface components will optionally reflect these changes and vice versa. Two essential components of a data binding are one source and one target. The source can be an XML data source, another control, etc., and the target is a control.

In our case, we want to bind the **Grid.Row[1].MaxHeight** property to the value of the **StackPanel.ActualHeight** property:

1. Select the **[Grid]** control.
2. In the **Properties** editor, expand the **Layout** category. Locate the **RowDefinitions (Collection)** property, click the button  , a dialog box is then displayed.
3. Set the **Height** property of the first row to **0.5***. 

Star-sizing

Star-sizing indicates that the sizes of the rows are proportional to each other. For example, a **Height** of "3*" will produce a row that is three times the size of a row whose **Height** is set to "*".

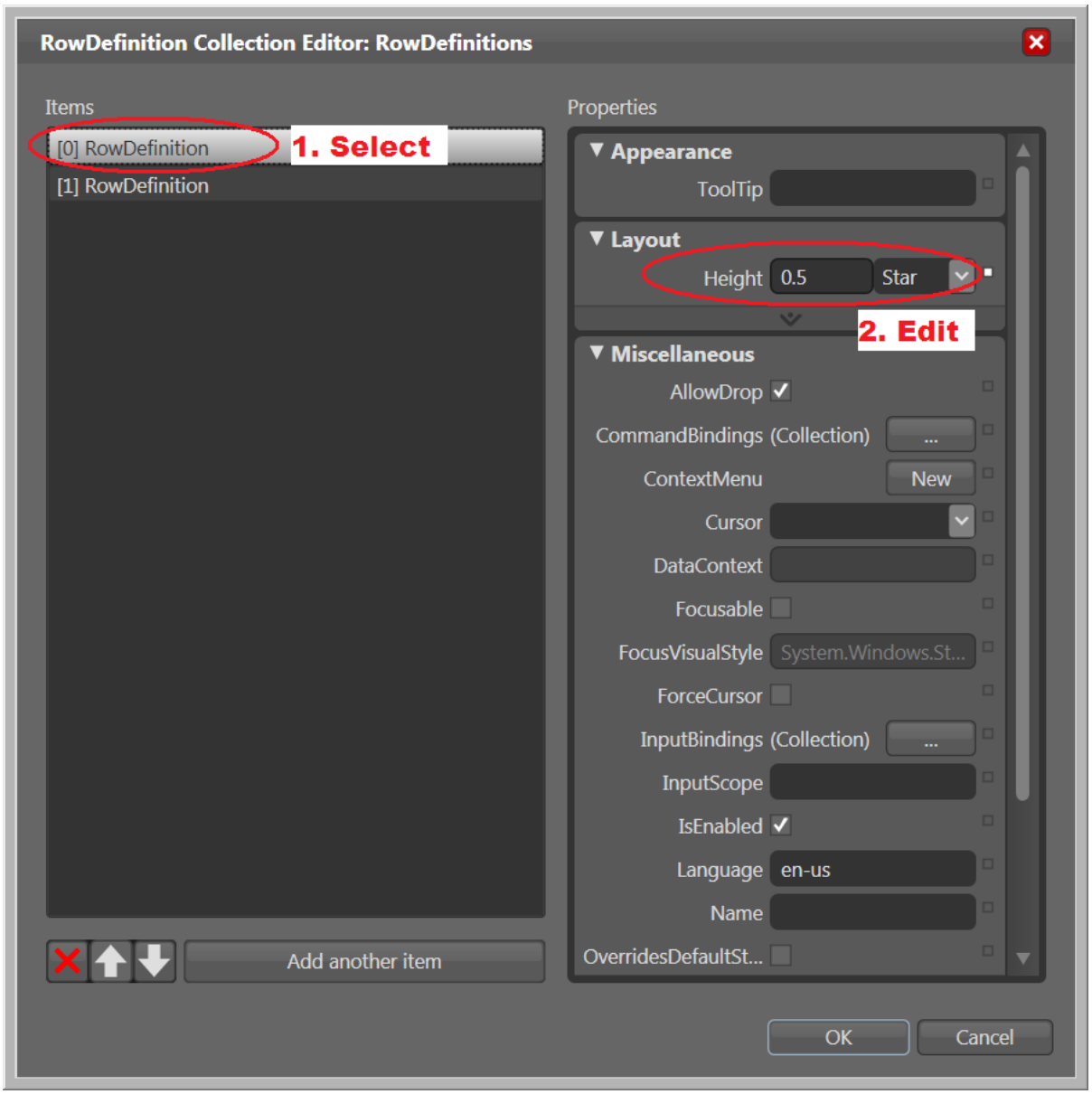


Figure 29 Setting the Height property of the first row to 0.5*.

4. Configure the properties of the second row **[1] RowDefinition** as follows: 

   - **Height = 0.5***
   - **MinHeight = 35**, this is the **Height** value of the **Button**, in order to avoid that the **StackPanel** gets hidden under the screen.

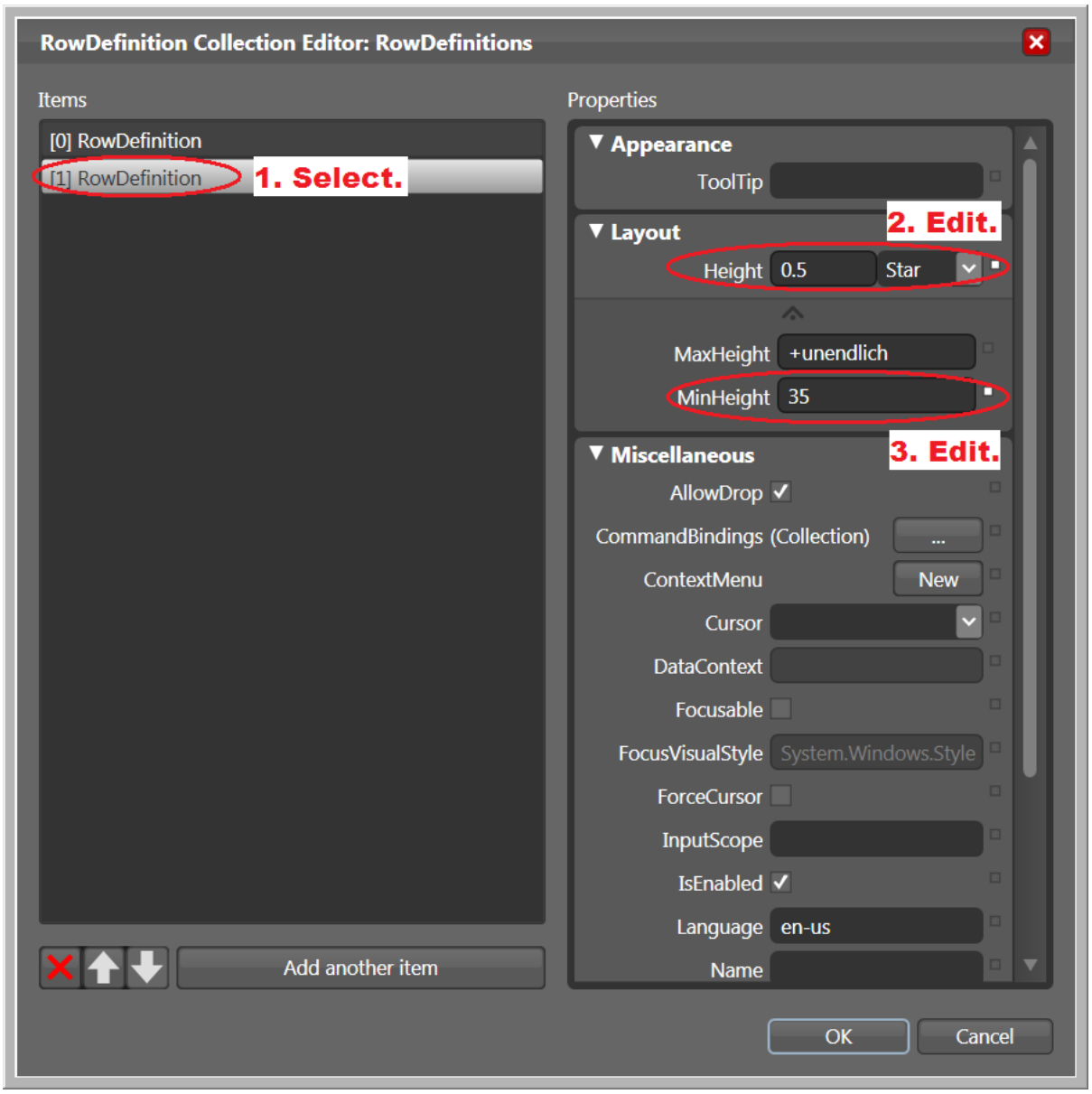


Figure 30 Editing the properties of the second row of [Grid].

5. Bind the **MaxHeight** property of the second row to the value of **[StackPanel].ActualHeight**.  **31**

   Click the small square button beside the **MaxHeight** property to open the context menu, choose the submenu item **Data Binding …**, then a dialog box opens, perform the steps described in Figure 31 to do data binding.

In normal cases, a yellow bounding box like 

will be shown beside the property name to indicate that the property is data bound. However, due to a bug in **Expression Blend**, you may not see it after data binding the **MaxHeight** property of **[1] RowDefinition**. Just ignore it.

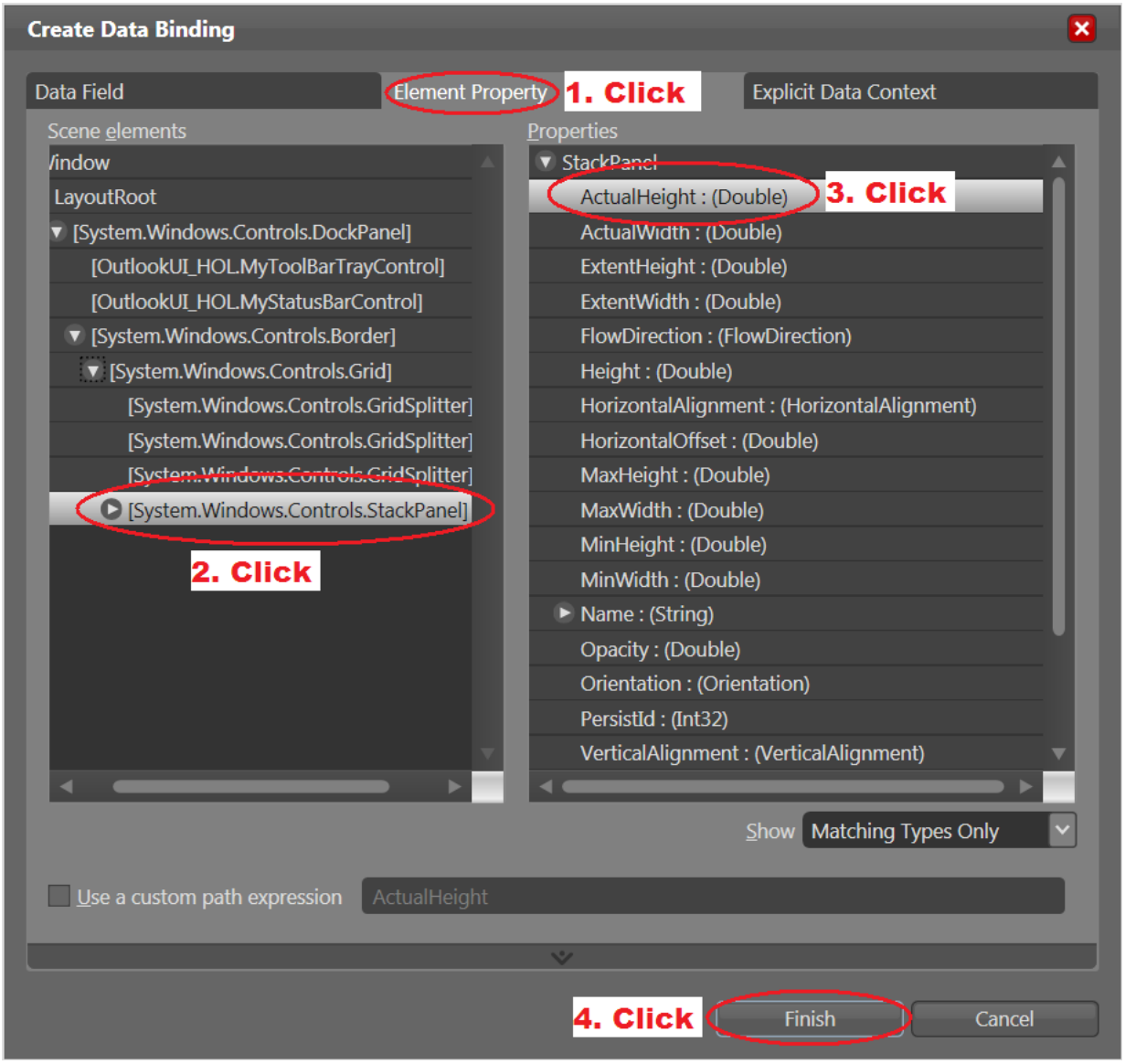


Figure 31 Binding the MaxHeight property of the second row to the value of the ActualHeight property of [StackPanel].

Click **OK** to close the **RowDefinition Collection Editor** dialog box.

After having successfully configured the row height of **[Grid]**, you may ask the question: how about the column width? Yes, we also need to do something with the first column: we want to keep **[StackPanel]** visible when we drag the first column divider towards the left side of **[Grid]**. So let's set the **MinWidth** property of the first column to **27** so that the images are always in our sight:

1. Select the **[Grid]** control.
2. In the **Properties** editor, expand the **Layout** category. Locate the **ColumnDefinitions (Collection)** property, click the button  , a dialog box is then displayed.
3. Set **MinWidth** = **27** and **Name** = **myMainGridFirstColumn** for **[0] ColumnDefinition**. 

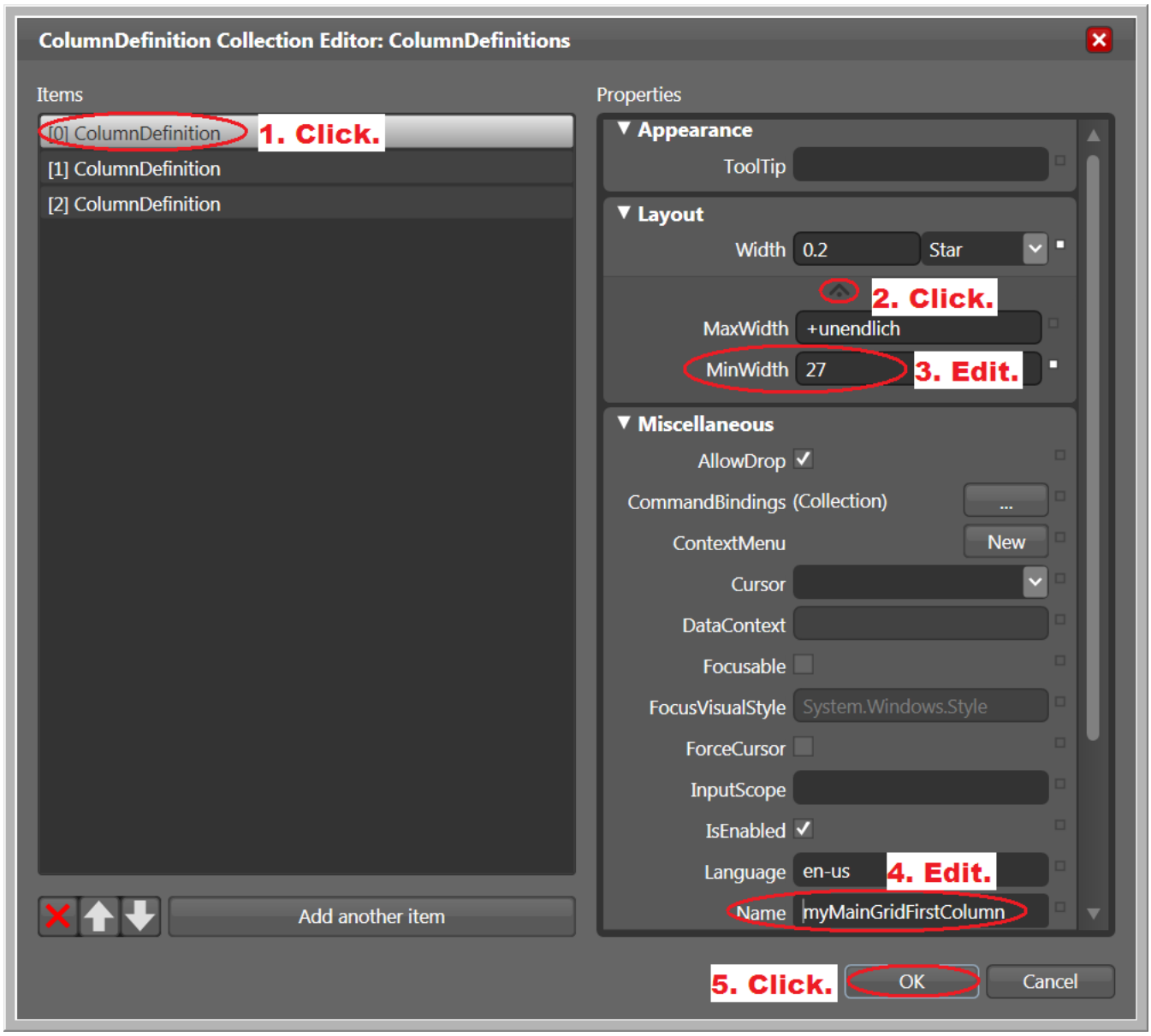


Figure 32 Setting the MinWidth and Name properties of the first column of [Grid].

Build and run your application (**F5**), make sure that the application behaves as you expect. Convince yourself that you should continue with our project.

Task 7: Adding MyFoldersExpandersControl to the Upper Left Cell of [Grid]

The two expanders for Favorites Folders and Mail Folders are available as **Custom Control** called **MyFoldersExpandersControl**.

It's time to raise your confidence in WPF applications! Try your best to add a **MyFoldersExpandersControl** to **[Grid]** and configure its properties as follows:

- **Name = myFoldersExpandersControl**

Brushes

- **BorderBrush = MyDarkBlueSolidBrush**

Appearance

- **BorderThickness = 0.5, 0.5, 0.5, 0.5**

Layout

- Width = Auto, Height = Auto
- Row = 0, RowSpan = 1, Column = 0, ColumnSpan = 1
- **HorizontalAlignment = Stretch, VerticalAlignment = Stretch**
- **Margin**: left = 0, right = 0, **top = 32**, **bottom = 7**

Task 8: Adding a ToggleButton to Show/Hide the Sidebar

Figure 1 shows the application user interface whose **[Grid]** has three columns. However, from user experience we know that most of the time we focus on the second column which shows all the items of a list with details like mail sender, receiver, subject, etc., and the third column which displays the content of an E-mail. So why not make more room for the second and the third columns?

Our approach is creating a **ToggleButton** control above the two expanders we added in the last task (see Figure 33).

Figure 33 A DockPanel control with a TextBlock and a ToggleButton.

If the **ToggleButton** is unchecked, then we have the normal view with the three columns in **[Grid]**; if the **ToggleButton** is checked, then the first column is minimized and a sidebar which contains a button for **Navigation Pane** and a button for **Inbox** is displayed (see Figure 34).

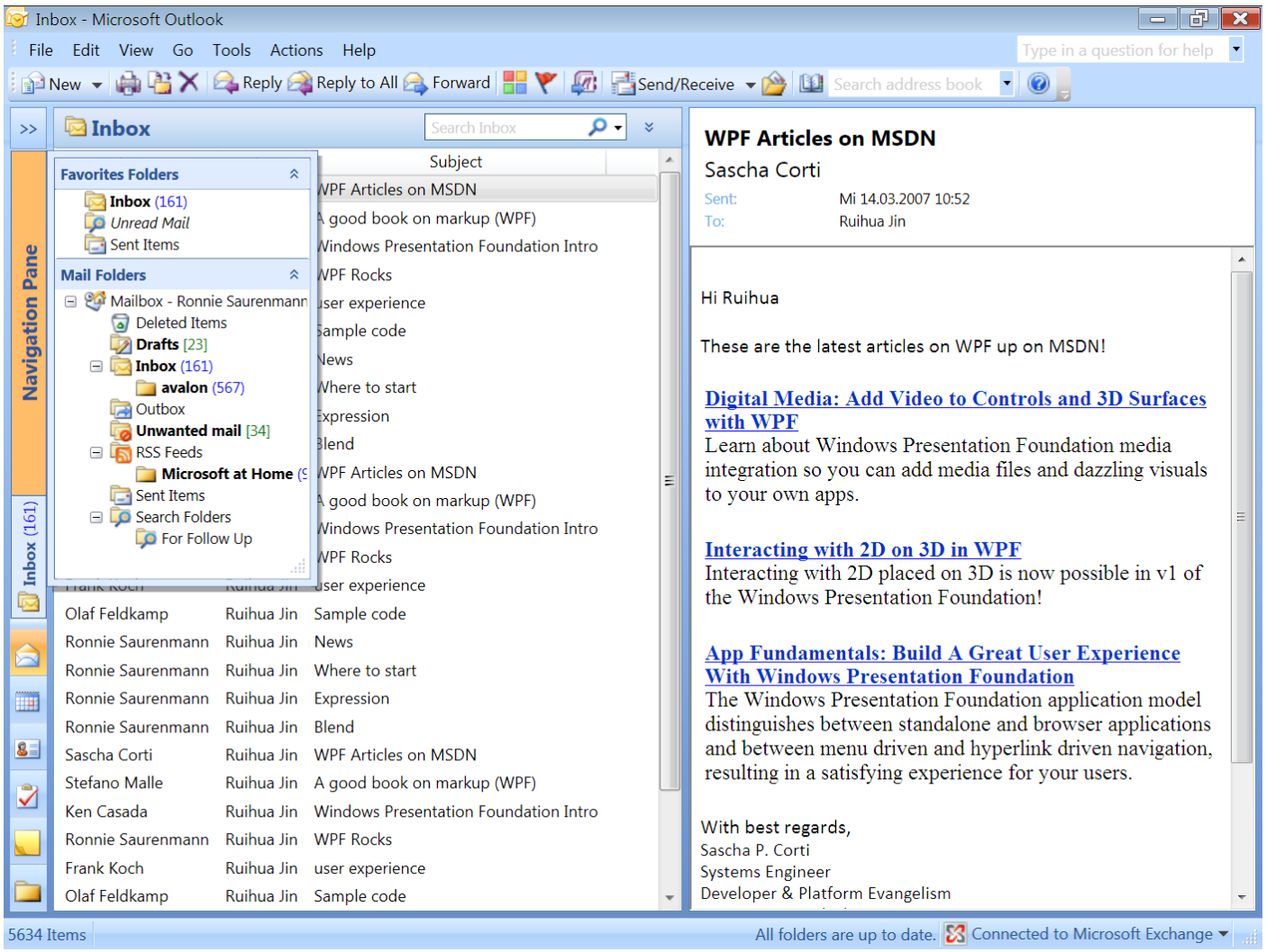


Figure 34 OutlookUI with a sidebar and an expanded navigation pane.

This part of work is more challenging and exciting than all what we have done so far, let's go!

1. Add a **DockPanel** control to **[Grid]** and name it **dp1**.
2. Group the **DockPanel dp1** into a **Border**.

   Under **Objects and Timeline**, right-click **dp1** to open the context menu, select **Group Into** > **Border.**

3. Create a custom solid brush for the **BorderBrush** of **[Border]**.

Solid Brush

A **solid brush** is a logical brush that contains 64 pixels of the same color. After creating the solid brush, the application can select it into its device context and use it to paint filled shapes.

With **[Border]** selected, click **BorderBrush** under **Brushes** category, then click to open the solid brush editor, type the following red, green, blue, alpha values for the color we want:

R = 43, G = 85, B = 151, A = 100%

This combination of red, green, blue, alpha values is equal to the Hex value **#FF2B5597**, so alternatively you can directly type this value into the corresponding text box.

Figure 35 Editing a custom solid brush called MyBlueBorderSolidBrush.

Next we want to convert this brush into a resource called **MyBlueBorderSolidBrush**, this step allows us to reuse the brush for properties like **Background**, **BorderBrush**, **Foreground** of other controls. To achieve this goal, click

, then a dialog box opens as shown in Figure 36. Type **MyBlueBorderSolidBrush** into the text box for **Resource name**, and choose **MyStyles.xaml** as the resource dictionary. Click **OK**.

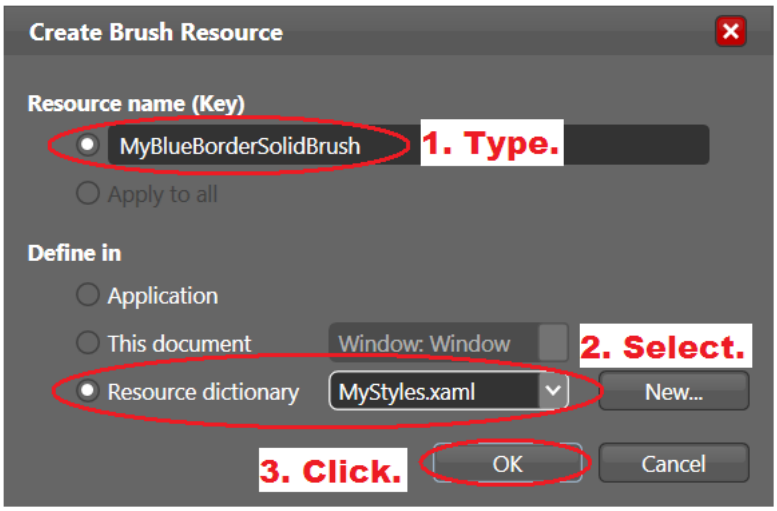


Figure 36 Converting the MyBlueBorderSolidBrush brush into a resource.

4. Create a custom linear gradient brush for the **Background** property of **[Border]**. **37,38,39,40,41**

Linear Gradient Brush

Unlike a solid brush, a **linear gradient brush** uses more than one color. You define different colors in different positions, and between two neighboring colors, both colors merge to create a transition or fading effect.

1) With **[Border]** selected, click **Background** under **Brushes** category, and then click to open the gradient brush editor as shown in Figure 37.

Figure 37 Gradient brush editor with default values.

As you see, the default gradient brush has gradually changing colors from black to white. There are two gradient 'stops' , one all the way on the left, and one on the right. You can add additional stops by clicking anywhere in the gradient editor band. You can also delete gradient stops by clicking and dragging them downward off the gradient editor.

Our custom gradient brush should look like the following:

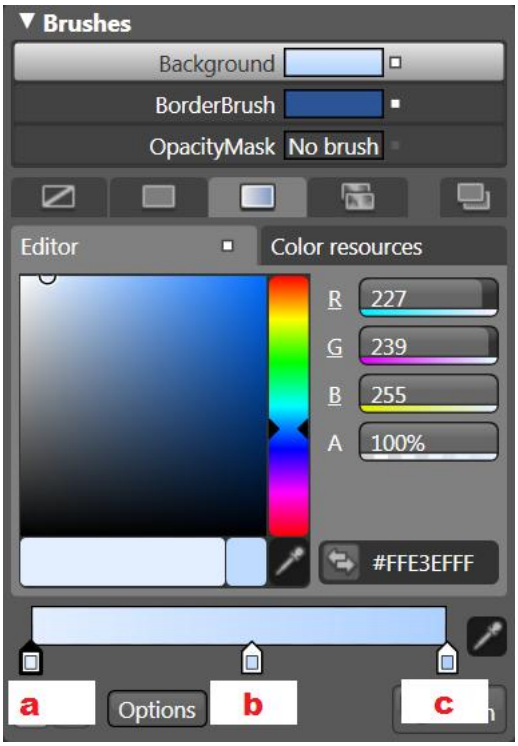


Figure 38 The gradient brush MyBlueBackgroundGradientBrush.

To create our own gradient brush, do the following:

2) Click the first gradient stop in position **a** as shown in Figure 38. Enter the following values:

   R = 227, **G = 239**, **B = 255**, **A = 100%** or **Hex value = #FFE3EFFF**

3) Click in the gradient editor band to add a third gradient stop, and drag it to position **b** as shown in Figure 38. Enter the following values:

   R = 199, **G = 223**, **B = 255**, **A = 100%** or **Hex value = #FFC7DFFF**

4) Click the third gradient stop in position **c** as shown in Figure 38, and enter the following values:

   R = 174, **G = 209**, **B = 255**, **A = 100%** or **Hex value = #FFAED1FF**

5) Modify the gradient vector:

   Each gradient is mapped along a vector, you modify the gradient vector to define the start and end points of the gradient.

   To get the gradient vector displayed in the control, click the **Brush Transform** tool in the **ToolBox** (see Figure 39).

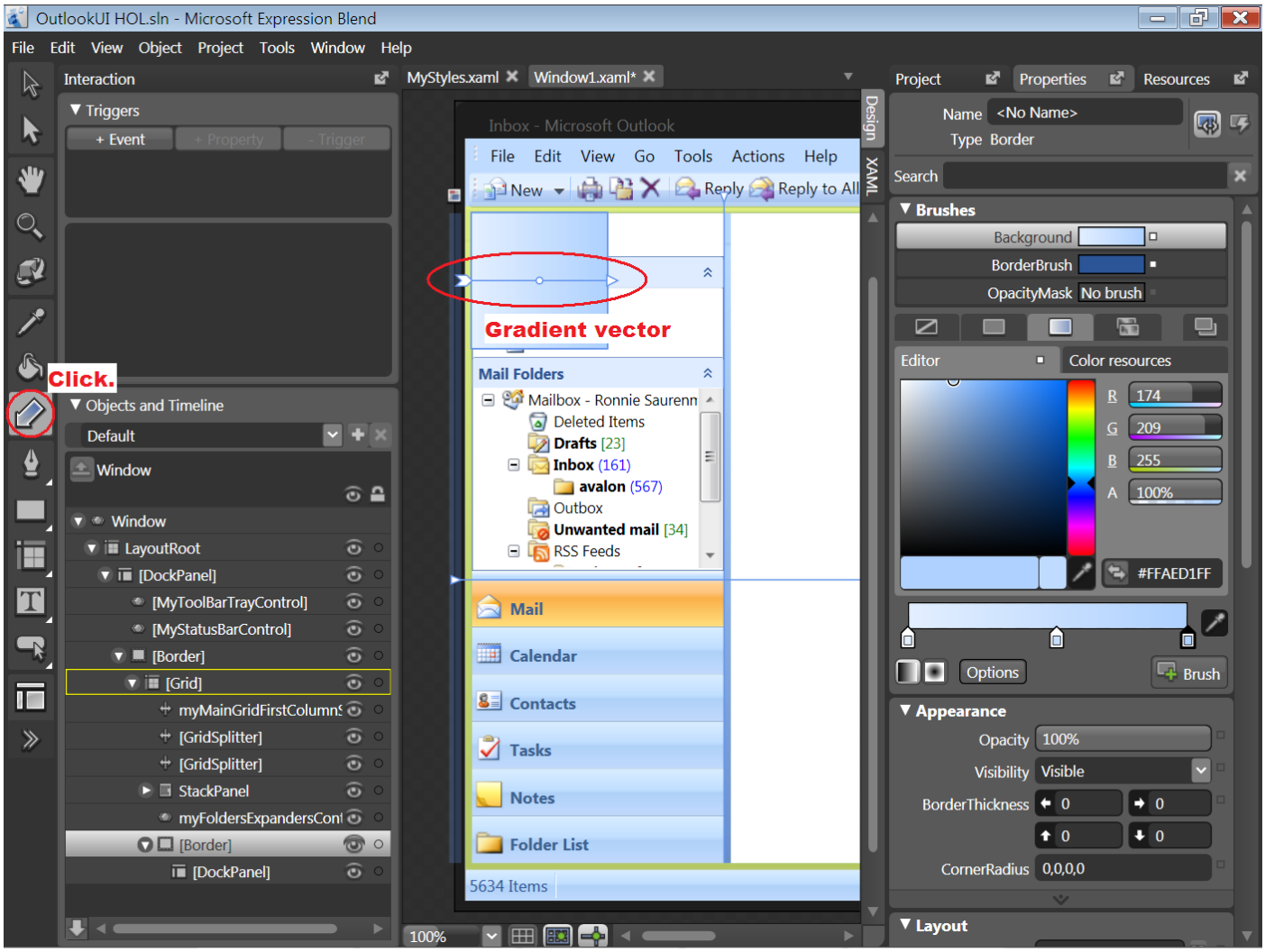


Figure 39 Selecting the Brush Transform tool to modify the gradient vector.

The default gradient vector is horizontal, which means that the gradient goes from left to right. To change the start and end points of the gradient, you need to drag the tail or head of the vector. Position mouse to right of the vector head until you see a rotation adorner, move mouse cursor clockwise while holding down the left mouse button and the shift key, stop when the gradient vector becomes vertical as shown in Figure 40.

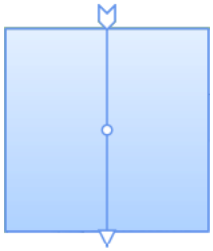


Figure 40 The gradient vector has been rotated to a vertical position.

6) To convert this brush to a resource for later reuse, click Brush to open the dialog box, and enter **MyBlueBackgroundGradientBrush** for resource name, and select **MyStyles.xaml** as resource dictionary (see Figure 41).

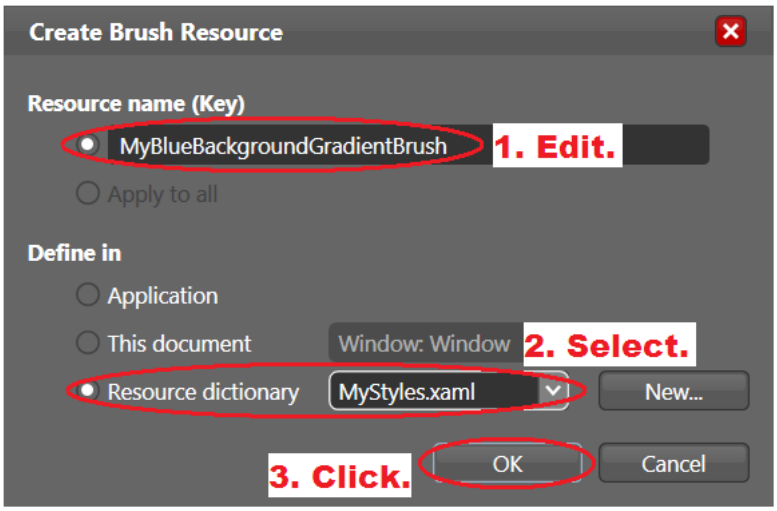


Figure 41 Converting the gradient brush to a resource.

5. Configure other properties of **[Border]** as follows:

   Appearance

   - **BorderThickness = 0.5, 0.5, 0.5, 0.5**

   Layout

   - **Width = Auto, Height = 31**
   - **HorizontalAlignment = Stretch, VerticalAlignment = Top**
   - **Margin = 0, 0, 0, 0**

6. With **dp1** selected, configure the properties of **dp1** as follows:

   Layout

   - **Width = Auto, Height = Auto**
   - HorizontalAlignment = Stretch, VerticalAlignment = Stretch
   - Margin = 0, 0, 0, 0

7. With **dp1** activated, add a **TextBlock** control to **dp1** with the following properties:

   Brushes

   - **Foreground = MyDarkBlueSolidBrush**

   Layout

   - Width = Auto, Height = Auto
   - Dock = Left
   - HorizontalAlignment = Stretch, **VerticalAlignment = Center**
   - **Margin**: **left = 4**, right = 0, top = 0, bottom = 0

   Common Properties

   - **Text = Mail**

   Text

   - **FontSize = 16, FontWeight = Bold**
   - **TextWrapping = NoWrap**

8. Add a **ToggleButton** control to **dp1** with the following properties:

   - **Name = myMainGridToggleButton**

   Brushes

   - **Background = No brush**

- **BorderBrush = No brush** 

Layout

- **Width = 26**, Height = Auto
- **Dock = Right**
- HorizontalAlignment = Stretch, **VerticalAlignment = Center**
- Margin = 0, 0, 0, 0

Now we have a **ToggleButton**, but it does not look like what we want it to, we want « instead of a rectangular button. A great advantage of WPF is that controls are not coerced to have a predefined appearance or behavior as we get used to, we can actually customize and style controls to our liking to make them perform the way we want. For example, a **Button** control can be rectangular, can be round, or even can have the form of an airplane.

Controls, Control Templates and Styles

Controls are the basic components of the user interface in applications, handling most user interactions with the application. Windows Presentation Foundation contains a number of default control templates which define the appearance of windows controls under different themes. You can change the structure and appearance of a control by editing the control template for the control.

Each control consists of two primary parts—a template and a style.

A **template** is the element tree which is used to build a control. You can edit a control template and rearrange, add or delete the elements (or parts) within the control.

A **style** allows you to set how the default attributes are defined for a control. These include states such as the pressed look of a button, as well as the default colors of the button. Styles are essentially "property bags" which, in many cases, also designate what template the control will use as well.

9. Do the following to define our own double-arrow-ToggleButton:

   1) Enter the **Control Template Editor** for **myMainGridToggleButton**.  42  43

      Under **Objects and Timeline**, right-click **myMainGridToggleButton** to open the context menu, select the submenu item **Edit Control Parts (Template)** > **Edit a Copy ...** (you cannot modify the system styles and templates, so make copies of them).

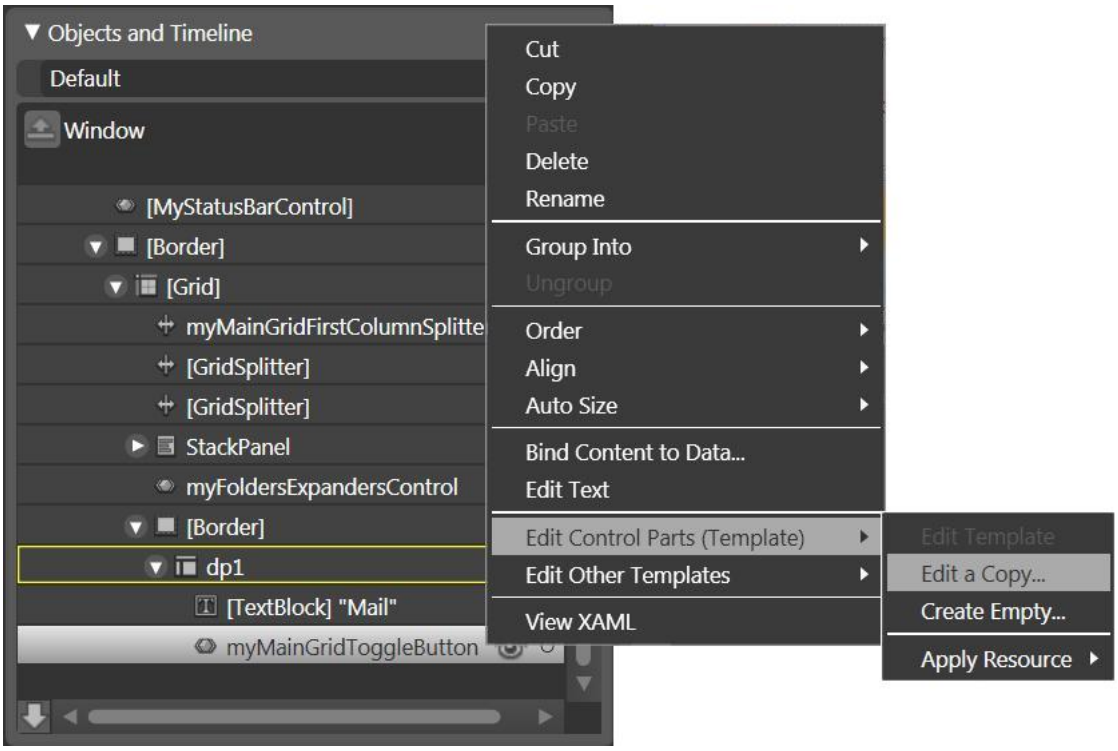


Figure 42 Entering the Control Template Editor for myMainGridToggleButton.

Then a dialog box opens, here you specify the name of the style as **MyDoubleArrowToggleButtonStyle**, and select **MyStyles.xaml** as the resource dictionary. Click **OK**.

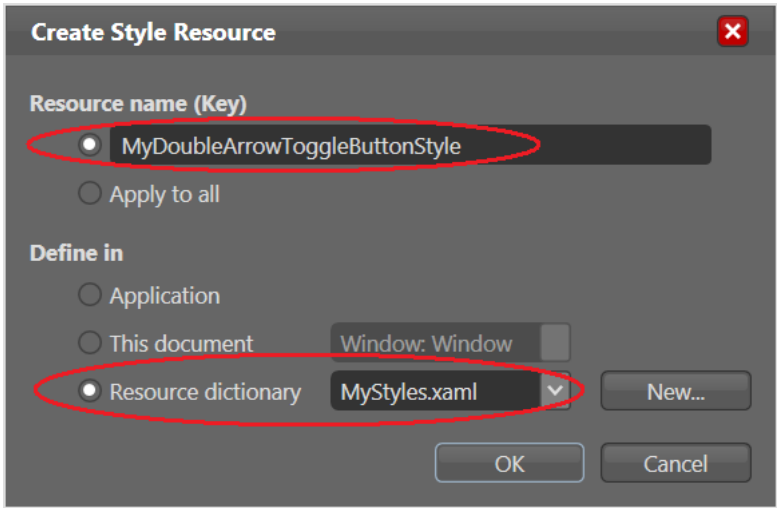
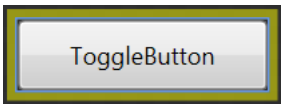


Figure 43 Naming the style and selecting a resource dictionary.

As you see, the **ToggleButton** template 

has a **Chrome** which contains a **[ContentPresenter]** control (see Figure 44).

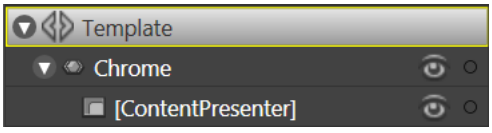


Figure 44 Hierarchical control structure of the original ToggleButton control.

2) To delete **Chrome**, right-click **Chrome** to open the context menu, select **Delete**.

3) With **Template** activated, add a **Grid** control to **Template** and configure its properties as follows:

Brushes

- **Background = an arbitrary solid color brush with Alpha = 0%**

  Under **Brushes** category, click **Background**, and then click  to enter the **Solid color brush editor**. Choose an arbitrary color and set **A = 0%**. An **alpha** value of **zero** represents full transparency.

You may wonder why we perform this step to achieve an effect of no effect. As you will see after we add two **Path** controls to **[Grid]**, if the **[Grid]** control does not have a **Background** color, then the user must click exactly on the **Path**s to check resp. uncheck the **ToggleButton**, which is inconvenient. A transparent color as background color of **[Grid]** solves this problem.

Layout

- **Width = 26**, Height = Auto

4) With **[Grid]** activated, draw a path in **[Grid]** on the artboard. 

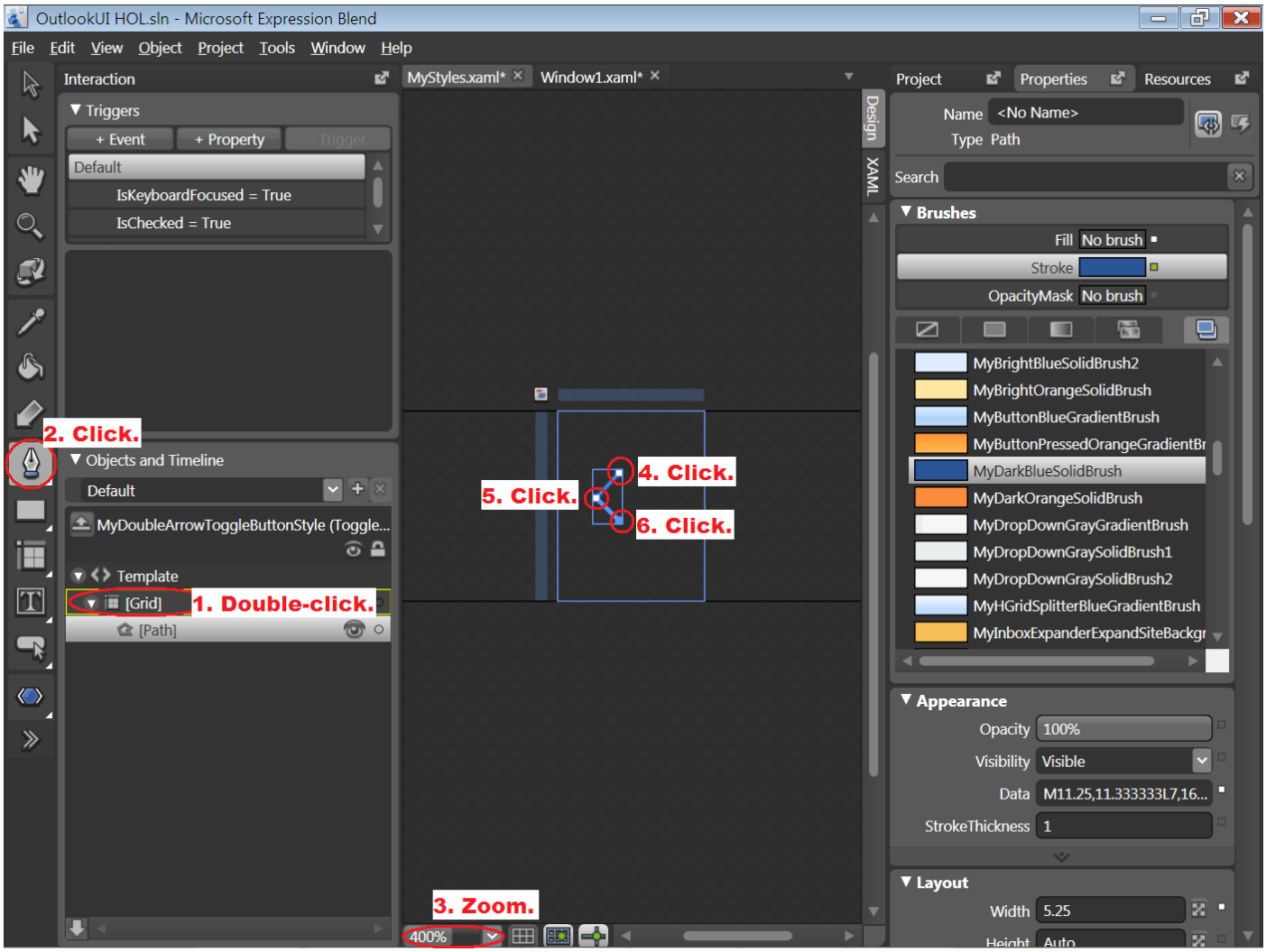


Figure 45 Drawing a path inside [Grid].

5) Configure the properties of the **[Path]** control as follows:

Brushes

- **Fill = No brush**
- **Stroke = MyDarkBlueSolidBrush**

6) Create a second path in **[Grid]**. 46

Under **Objects and Timeline**, right-click **[Path]** to open the context menu, select **Copy**. Then right-click **[Grid]** to open the context menu, select **Paste**.

With the second **[Path]** selected, drag it right as described in Figure 46.

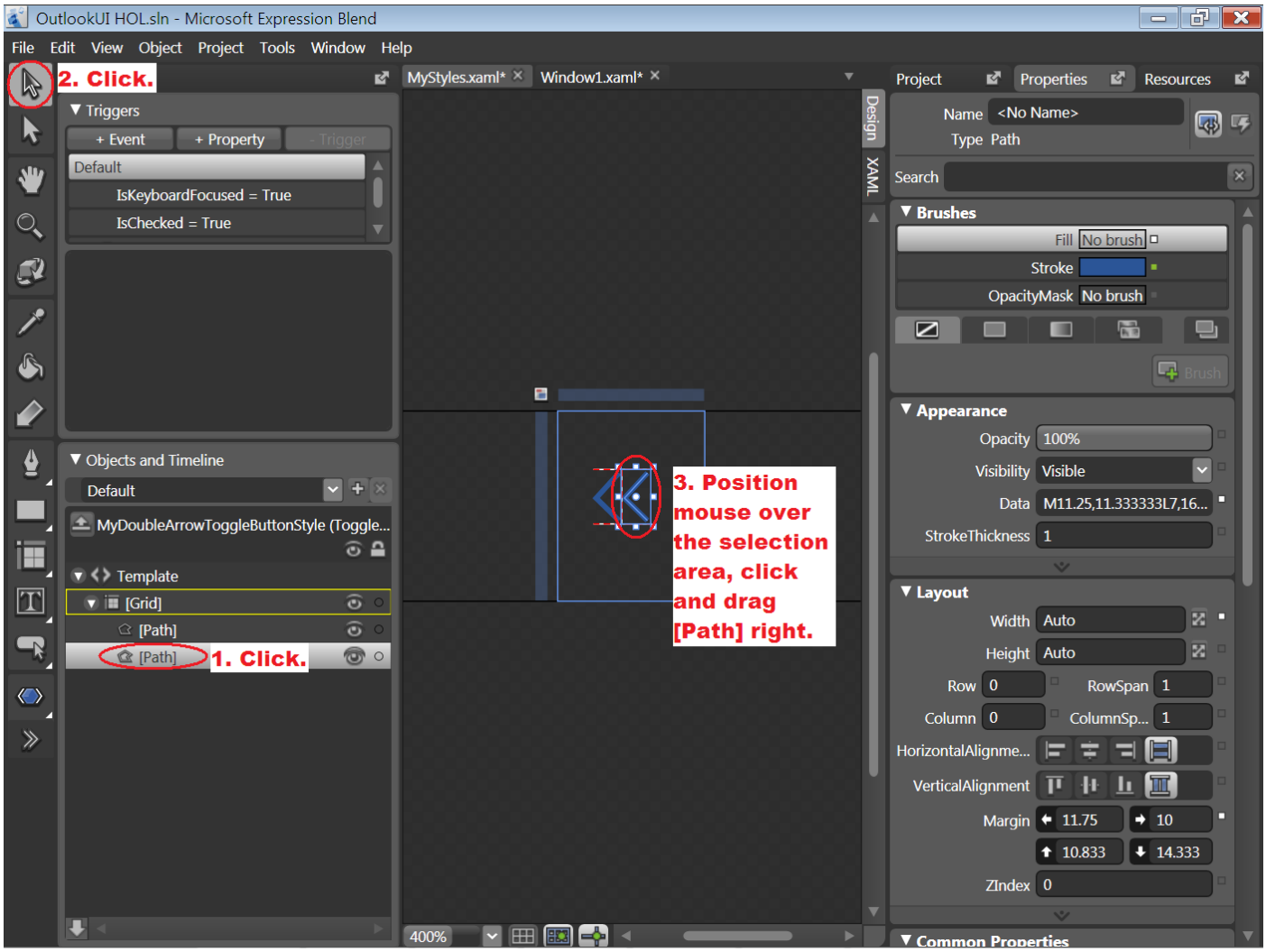


Figure 46 Moving the second path right.

7) Add and edit the property trigger **IsChecked = True:** 47 48

Since we want our double-arrow-ToggleButton to point left or right, depending on whether the user checks it or unchecks it, we need to add a property trigger which is activated when the **IsChecked** property becomes **True**.

Property Trigger

Property trigger is the mechanism by which a change in one property triggers either an *instant* or an *animated* change in another property.

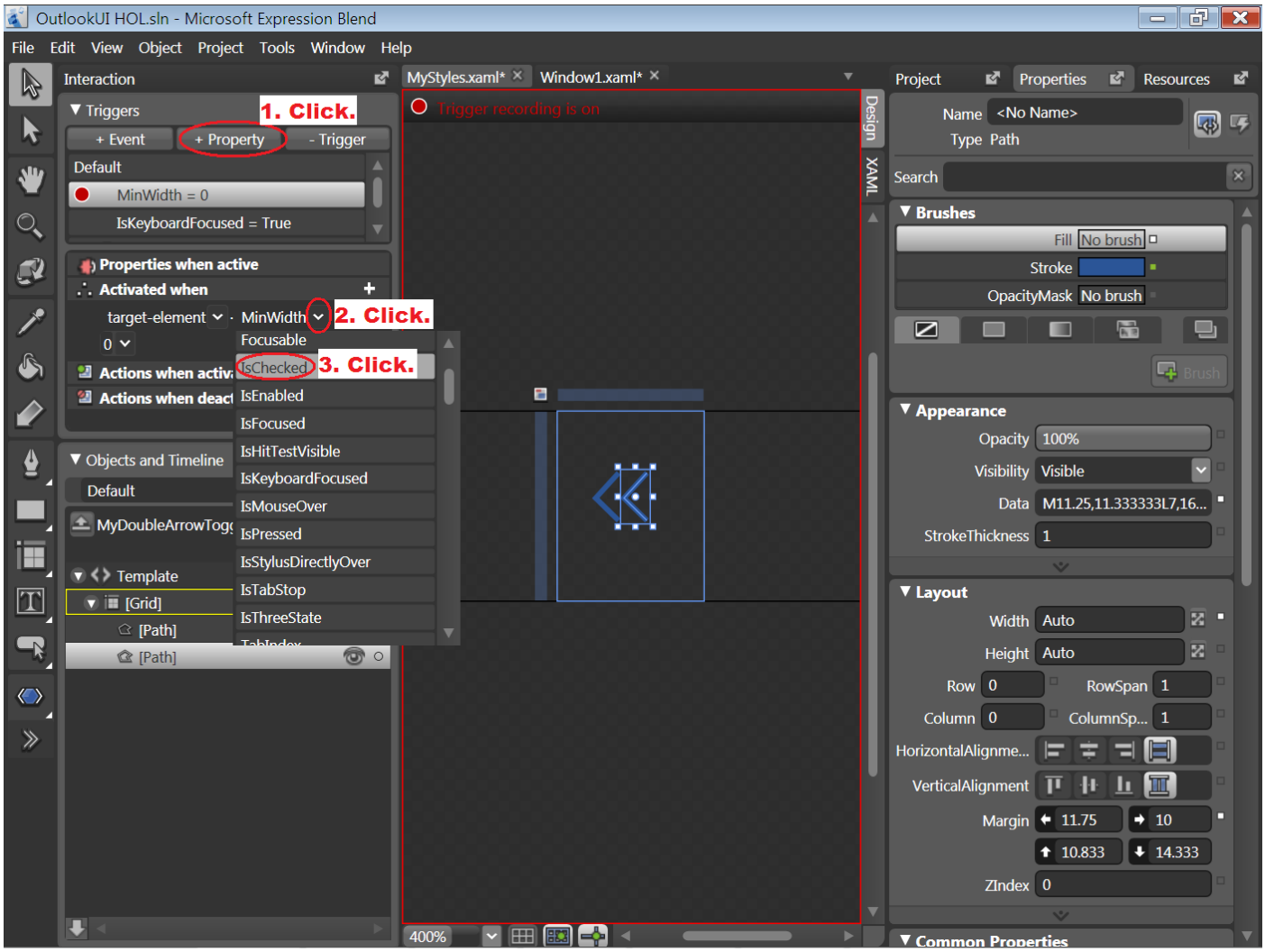


Figure 47 Adding the property trigger IsChecked = True.

MinWidth = 0 is the default property trigger, change **MinWidth = 0** to **IsChecked = True** as described in Figure 47. As soon as the property trigger IsChecked = True is added to the list of triggers, Trigger recording is on is displayed on the screen, which means that all the changes you make from now on are recorded in the property trigger, they will take effect when the related property gets the specified value. In our case, when **ToggleButton.IsChecked = True**, the double-arrow points to the opposite direction. The property changes for both **[Path]** controls are:

Transform

- Under **RenderTransform** subcategory, **Angle = 180**.

Consult Figure 48 to configure the properties of both **[Path]** controls.

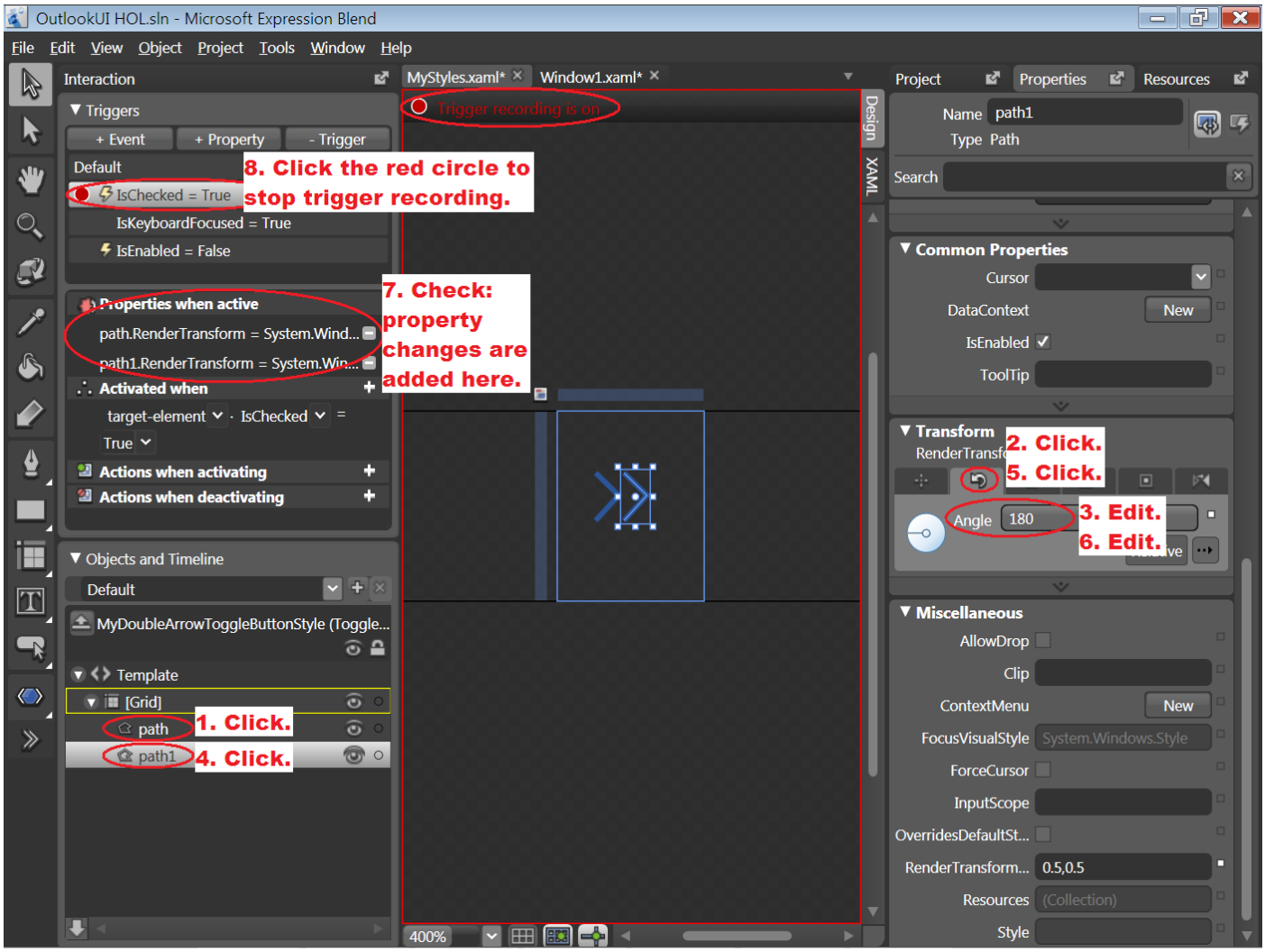


Figure 48 Editing the property trigger IsChecked = True.

8) Under **Objects and Timeline**, click  (Scope up) to exit the Control Template Editor.

Build and run the application (**F5**), make sure that the **ToggleButton** works as expected.

The next step is to add a sidebar. Perform the following steps:

1. With **[Grid]** activated, add a custom control **MySidebarControl** to **[Grid]**.
2. Configure the properties of **[MySidebarControl]** as follows:
   - **Name = mySidebarControl**

   Brushes
   - **BorderBrush = MyDarkBlueSolidBrush**

   Appearance
   - **BorderThickness = 0.5, 0.5, 0.5, 0.5**

   Layout
   - Width = Auto, **Height = 27**
   - **HorizontalAlignment = Left, VerticalAlignment = Stretch**
   - **Margin**: left = 0, right = 0, **top = 32**, **bottom = 7**

Transform

Expand the **Transform** category, under the **LayoutTransform** subcategory: 49

- **Angle = -90**

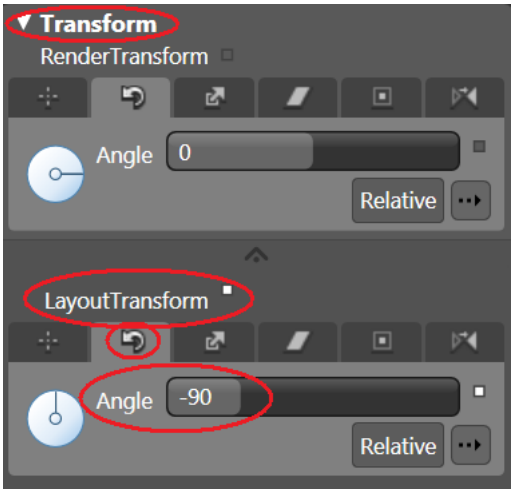


Figure 49 Editing the Angle property of LayoutTransform.

Appearance

- **Visibility = Hidden**

As described above, we want to get the first grid column minimized and the sidebar displayed when **myMainGridToggleButton** is checked. To get this effect, we need to write an event handler for the **myMainGridToggleButton.Checked** event. To go back to the default view, we should write an event handler for the **myMainGridToggleButton.Unchecked** event.

Event Handling

Event handling is a mechanism which allows flexible interactions between program components. To react to a certain event, you must register an event handler for the event, as soon as this event is triggered, your event handler is called. Typical events are GUI events like mouse clicks, key presses, etc.

1. Write an event handler for the **myMainGridToggleButton.Checked** event. 50

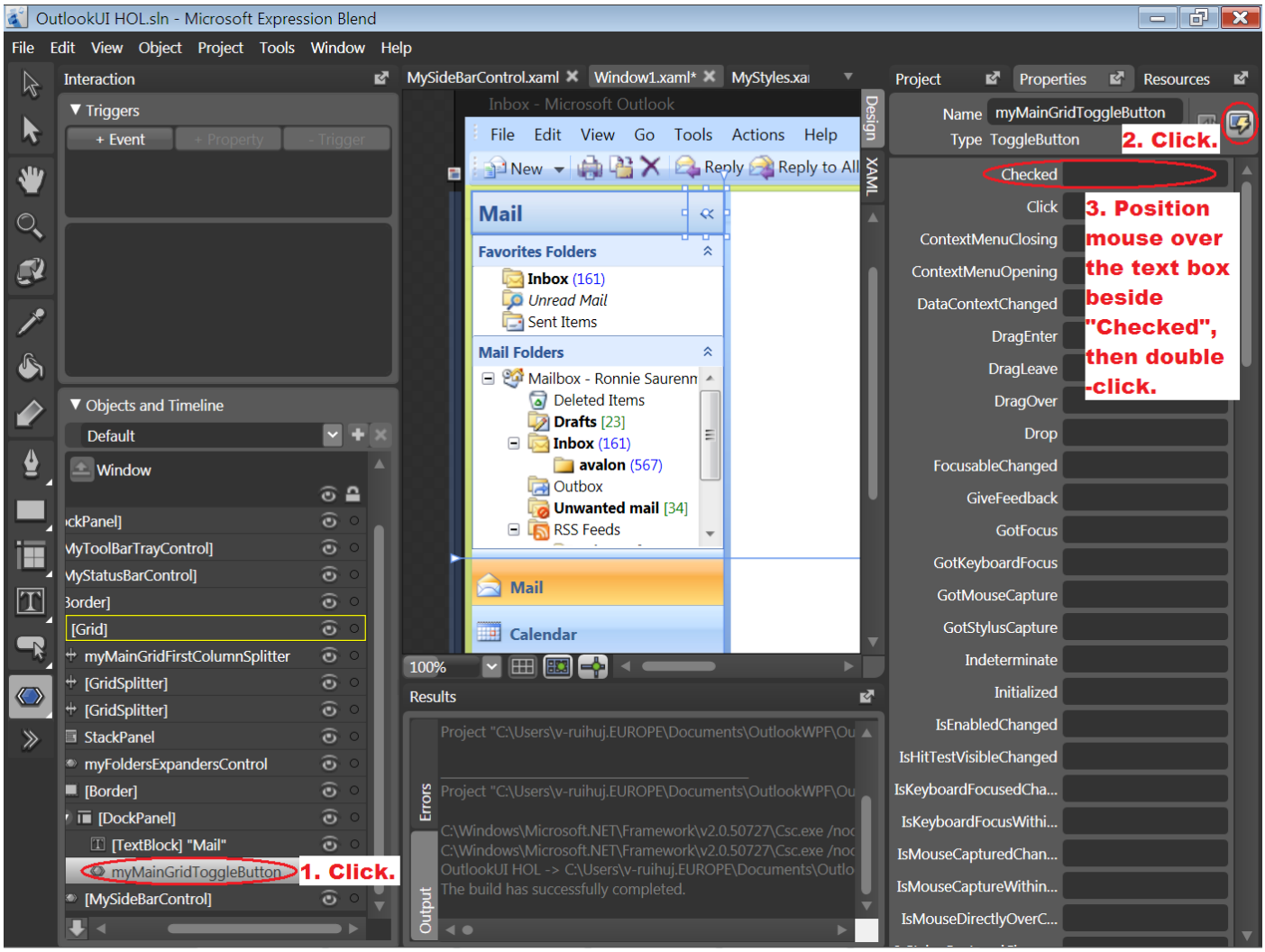


Figure 50 Adding an event handler for myMainGridToggleButton.Checked event.

After step 3 described in Figure 50, Expression Blend automatically names the event handler **myMainGridToggleButton_Checked**.

If you have Visual Studio installed, then Visual Studio will be started, and the method signature is automatically added into **Window1.xaml.cs**; otherwise, choose your favorite editor to edit the event handler.

In the **Window1.xaml.cs** file, the **Window1** class is defined in the **OutlookUI_HOL** namespace. Add the following code (bold) to the **myMainGridToggleButton_Checked** method in the **Window1** class:

basic8.txt

```
private GridLength gridLength;
private void myMainGridToggleButton_Checked(object sender, RoutedEventArgs e)
{
   /* store the current width of the first grid column */
   if (myMainGridFirstColumn.ActualWidth > myMainGridFirstColumn.MinWidth)
      gridLength = myMainGridFirstColumn.Width;
   /* set the first grid column width to its minimum width */
   myMainGridFirstColumn.Width = new GridLength(myMainGridFirstColumn.MinWidth);
   /* hide the two expanders for Favorites Folders and Mail Folders */
   myFoldersExpandersControl.Visibility = Visibility.Hidden;
   /* display mySidebarControl */
   mySidebarControl.Visibility = Visibility.Visible;
}
```

2. Write an event handler for the **myMainGridToggleButton.Unchecked** event.

basic8.txt

```
private void myMainGridToggleButton_Unchecked(object sender, RoutedEventArgs e)
{
  if (!myMainGridFirstColumnSplitter.IsDragging)
    myMainGridFirstColumn.Width = gridLength;  // restore the first column width
  myFoldersExpandersControl.Visibility = Visibility.Visible;
  mySidebarControl.Visibility = Visibility.Hidden;
}
```

3. Besides **myMainGridToggleButton**, the first grid column divider **myMainGridFirstColumnSplitter** also changes the width of the first column when you drag it, that's why we need to write an event handler for the **myMainGridFirstColumnSplitter.DragDelta** event:

basic8.txt

```
private void myMainGridFirstColumnSplitter_DragDelta(
       object sender, System.Windows.Controls.Primitives.DragDeltaEventArgs e)
{
  /** trigger the myMainGridToggleButton.Checked event when
    * the first grid column width reaches its minimum */
  if (myMainGridFirstColumn.ActualWidth <= myMainGridFirstColumn.MinWidth)
    myMainGridToggleButton.IsChecked = true;
  /** trigger the myMainGridToggleButton.Unchecked event when
    * the first grid column width exceeds its minimum */
  else
    myMainGridToggleButton.IsChecked = false;
}
```

Build and run your application (**F5**). Drag the first column divider left and right, check the **ToggleButton**. The sidebar should pop up and get hidden as expected. You may complain that the text **Mail** keeps sticking in its position, don't worry, there is an easy way to solve this problem, just do the following:

4. Set the order of **myMainGridToggleButton**. **51**

Right-click **myMainGridToggleButton** to open the context menu, select **Order** > **Send to Back**. You can see that **myMainGridToggleButton** is moved above **[TextBlock] "Mail"**, this means that the **"Mail" TextBlock** becomes the last child of the **DockPanel dp1**. With **dp1.LastChildFill** set to **True**, **myMainGridToggleButton** is positioned at first, then the **TextBlock** fills the remaining area.

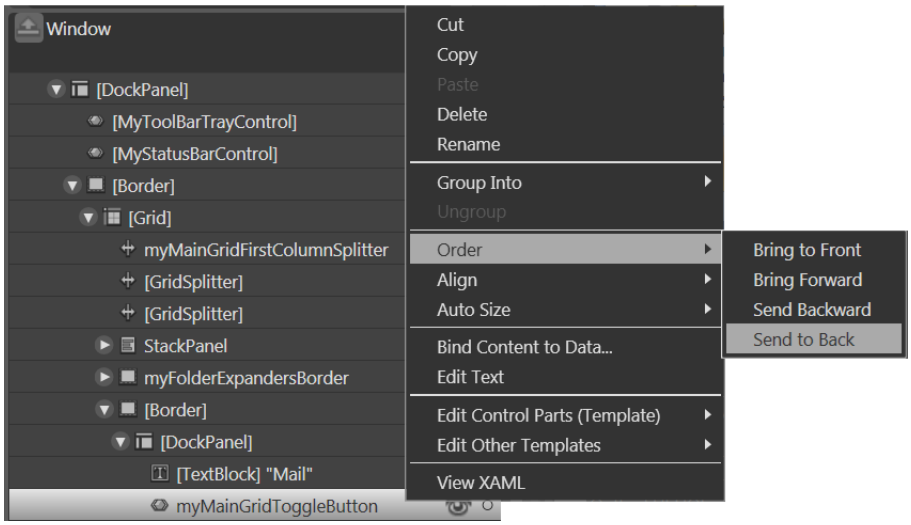


Figure 51 Setting the order of myMainGridToggleButton.

Task 9: Adding MyInboxExpanderControl to the Second Column of [Grid]

The second column of our layout container **[Grid]** has an **Expander** control and a **ListView** control which shows all the items of the Inbox.

1. With **[Grid]** activated, add a **DockPanel** control to **[Grid]** and name it **dp2.**

Since we align two controls vertically in one direction, a **StackPanel** as parent control would be more proper. However, practice reveals that a **StackPanel** does not deal well with a **ScrollViewer** control contained in it, the scrollbar is not displayed even when the content of a **ScrollViewer** control exceeds the range of the **StackPanel** control. A **DockPanel** instead does not suffer from this problem.

2. Group the **DockPanel dp2** into a **Border**.
3. Configure the properties of **[Border]** as follows:

   Brushes

   - **BorderBrush = MyDarkBlueSolidBrush**

   Appearance

   - **BorderThickness = 0.5, 0.5, 0.5, 0.5**

   Layout

   - **Width = Auto, Height = Auto**
   - Row = 0, **RowSpan = 2**, **Column = 1**, ColumnSpan = 1
   - **HorizontalAlignment = Stretch, VerticalAlignment = Stretch**
   - **Margin**: **left = 5**, **right = 5**, top = 0, bottom = 0
4. Configure the properties of **dp2** as follows:

   Layout

   - **Width = Auto, Height = Auto**
   - HorizontalAlignment = Stretch, VerticalAlignment = Stretch
   - **LastChildFill = True**
   - Margin = 0, 0, 0, 0
5. With **dp2** activated, add a custom control called **MyInboxExpanderControl** to **dp2**, and configure its properties as follows:

   Layout

   - **Dock = Top**
   - HorizontalAlignment = Stretch, **VerticalAlignment = Top**

Task 10: Adding a ListView Control to Display Mails

In this task, you will learn how to bind a control to an external XML data source.

1. With the **DockPanel dp2** activated, add a **ListView** control to **dp2**, and configure its properties as follows:

   Brushes

   - **BorderBrush = No brush**
   - **Foreground = Black** (In the solid color brush editor, set **R = 0**, **G = 0**, **B = 0**, **A = 100%**).

   Appearance

   - **BorderThickness = 0, 0, 0, 0**

   Layout

   - **Dock = Top**
   - Margin = 0, 0, 0, 0

   Common Properties

   - IsSynchronizedWithCurrentItem = True (Checked)
   - **SelectedIndex = 0**

2. Add a new XML data source to the Project. 
   Switch to the **Project** tab, under the **Data** section, click **+XML** button, a dialog box opens.

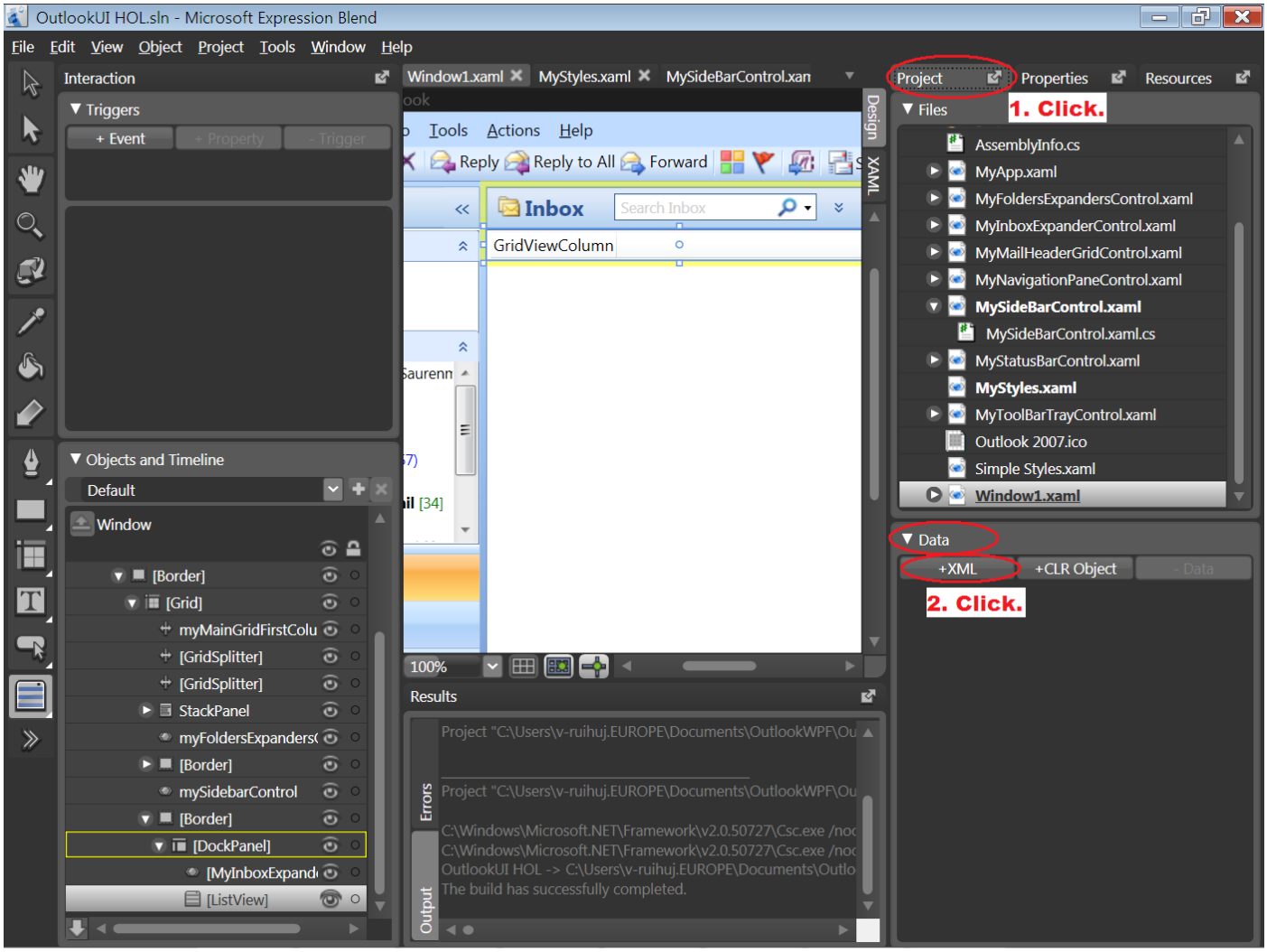


Figure 52 Opening *Add XML Data Source* dialog box.

In the dialog box, set the URL for XML data to **inbox.xml** (**inbox.xml** has already been configured as **Resource** of the project). Click **OK** to close the dialog box.

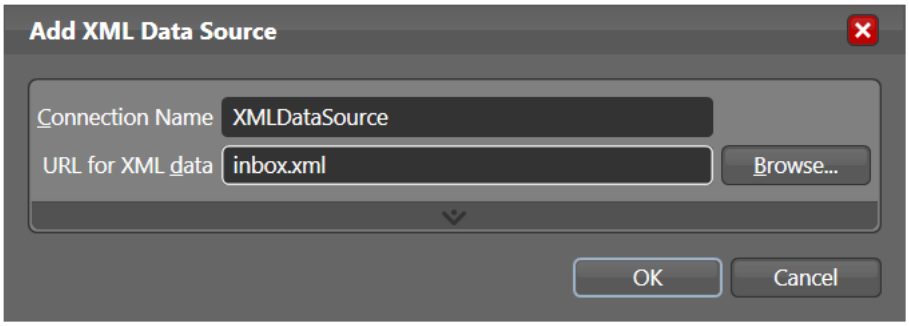


Figure 53 Setting the URL for XML data.

As you see, a new data source called **inboxDS** has been added to the project. The root element is **inbox**, within the **inbox** element there is an array of **mail** elements, and each **mail** element contains several child elements which specify the properties of the related mail (see Figure 54).

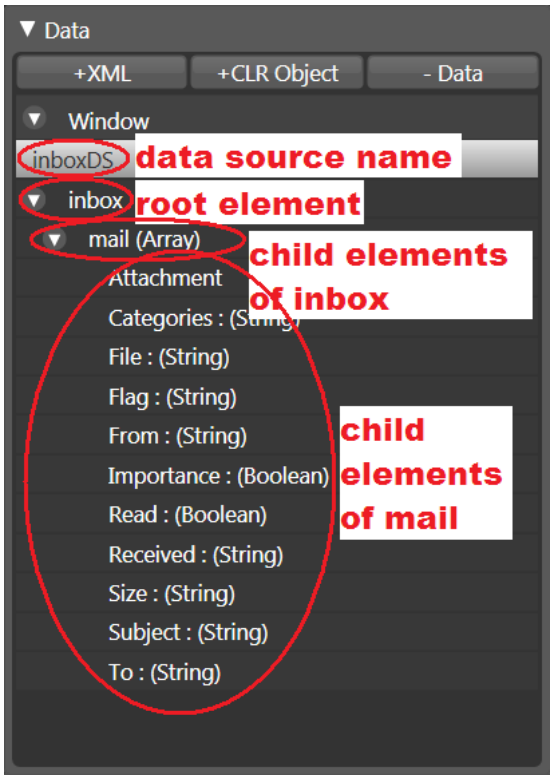


Figure 54 The XML data structure of inboxDS.

3. Data bind the data context of **LayoutRoot** to **inboxDS.**  **55**

Data Context

Data context is a concept that allows elements (like **[ListView]** in our project) to inherit information from their parent elements (like **LayoutRoot** in our project) about the data source that is used for binding, as well as other characteristics of the binding, such as the path.

Select **LayoutRoot**, under **Common Properties** category, click the small square beside the **DataContext** property to open the context menu, select **Data Binding…**.

The **Create Data Binding** dialog box opens. Perform the steps described in Figure 55 to bind the **LayoutRoot.DataContext** property to **/inbox/mail** of **inboxDS**.

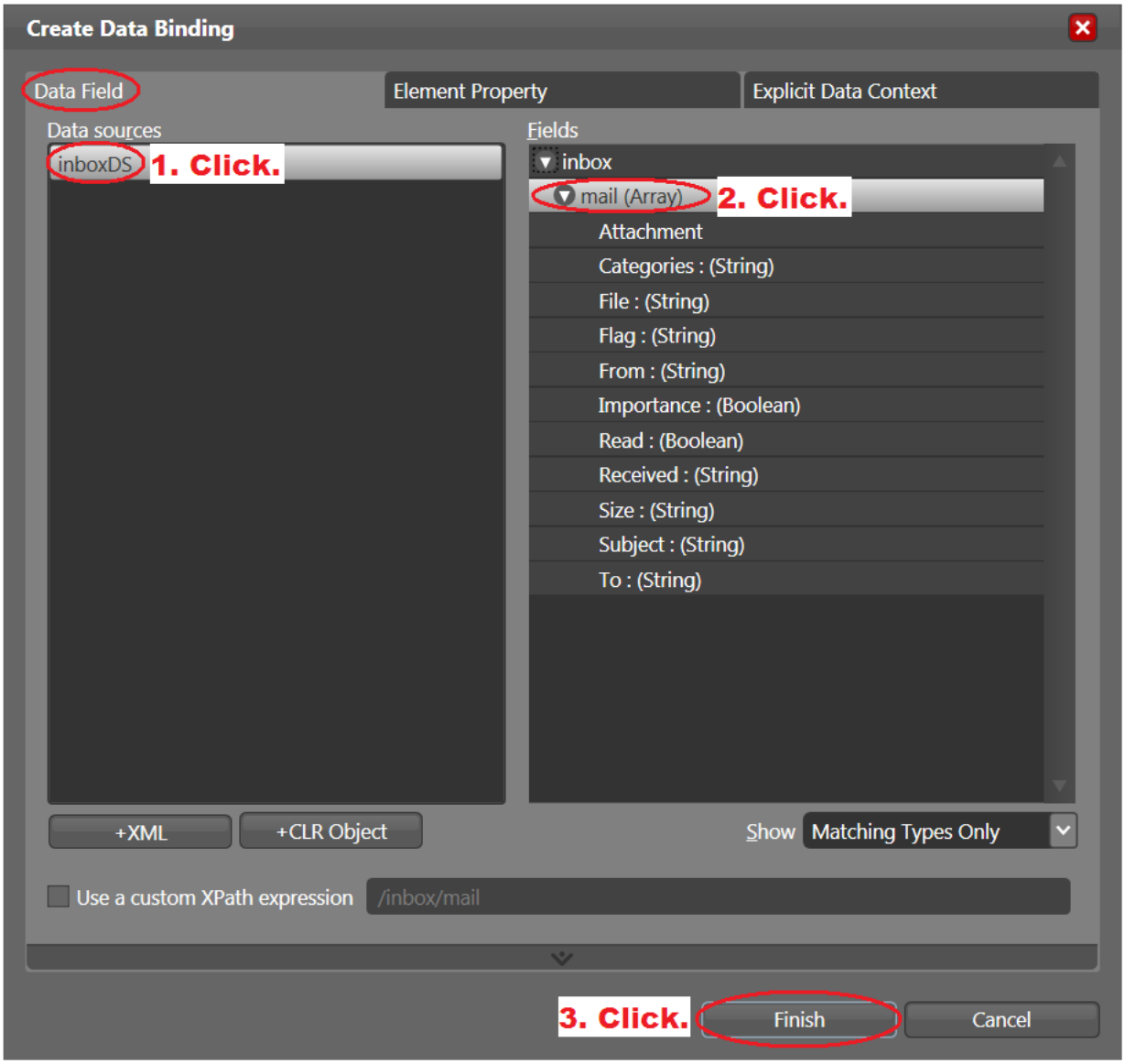


Figure 55 Data binding LayoutRoot.DataContext to /inbox/mail of inboxDS.

As soon as you have bound the **LayoutRoot.DataContext** to an XML data source, the child elements of **LayoutRoot** (like **[ListView]**) automatically inherit the data source information from their parent control. This can be confirmed if you go to the Properties Editor of **[ListView]**, and see that beside the **DataContext** property, **(XmlDataCollection)** has been newly added.

4. Data bind the **[ListView].ItemsSource** property to the explicit data context.  56

   With **[ListView]** selected, click the small square beside the **ItemsSource** property under the **Common Properties** category, select **Data Binding…** to open the **Create Data Binding** dialog box. Perform the steps described in Figure 56 to bind **[ListView].ItemsSource** to the **mail** element of the explicit data context (you select the "Explicit Data Context" option, as you are binding **ItemsSource** to the **DataContext** inherited from the parent control of **[ListView]**).

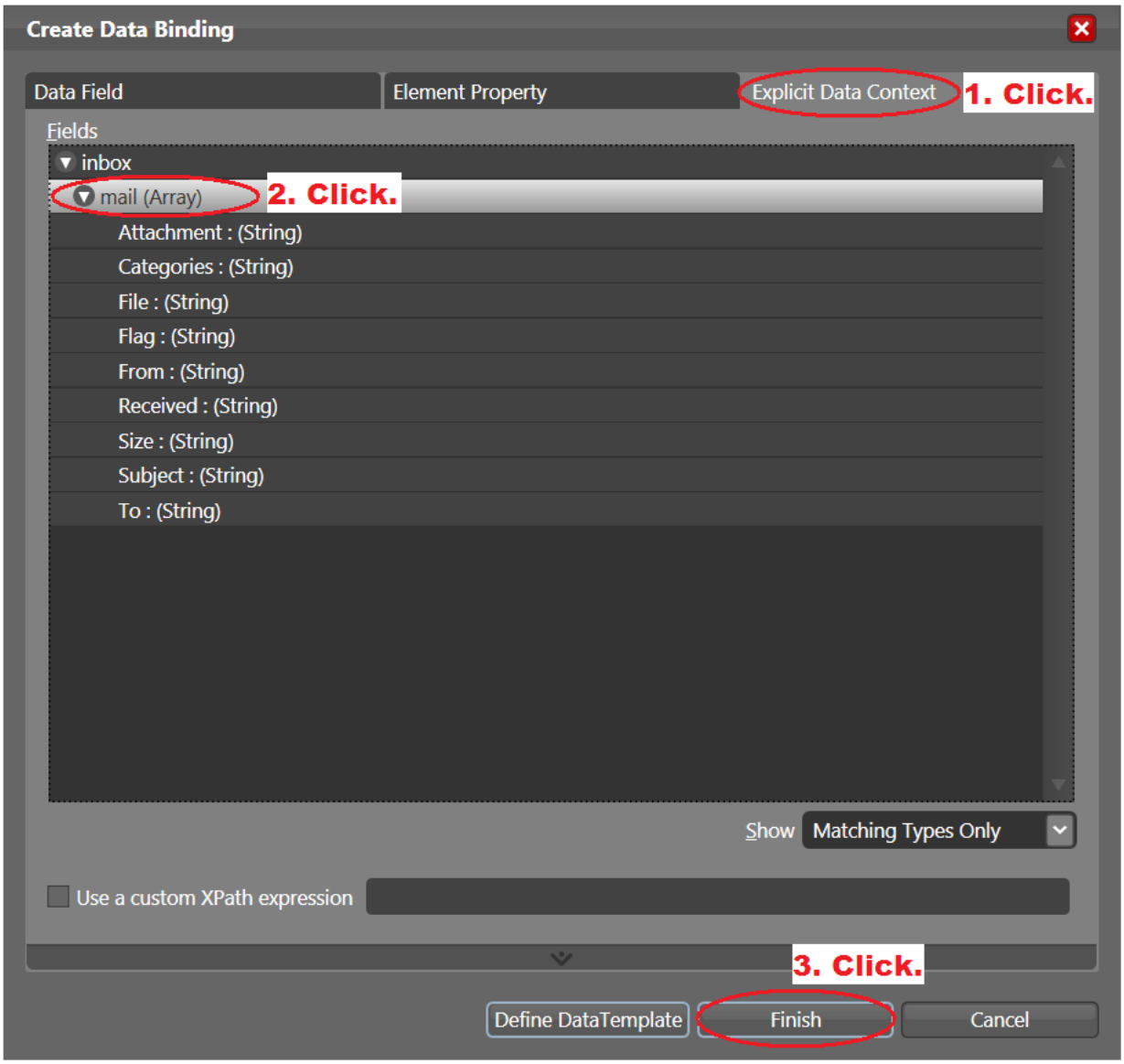


Figure 56 Data binding [ListView].ItemsSource to the explicit data context.

5. Create list columns for showing mail properties like sender, receiver and subject:

   By default the **ListView.View** is a **GridView** which allows you to create columns and show values of a certain property in the related column. Perform the following steps to create three columns **From**, **To** and **Subject**.

   1) Under **Miscellaneous** category, expand the **View (GridView)** property editor and click the button to open the **GridViewColumn Collection Editor** dialog box. 57

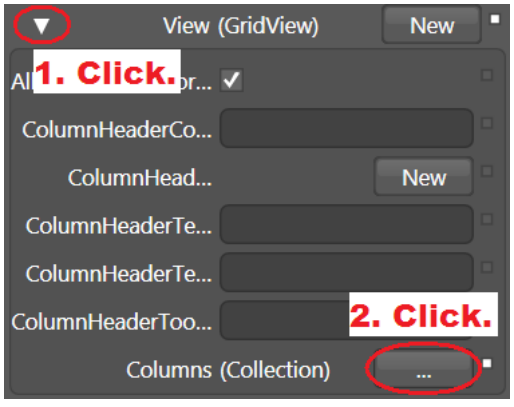


Figure 57 Opening GridViewColumn Collection Editor.

2) By default, there is already a column called **GridViewColumn** in the **GridView**. Set the property value of **Header** to "**From**".

3) Add a second column with **Header = To**. 

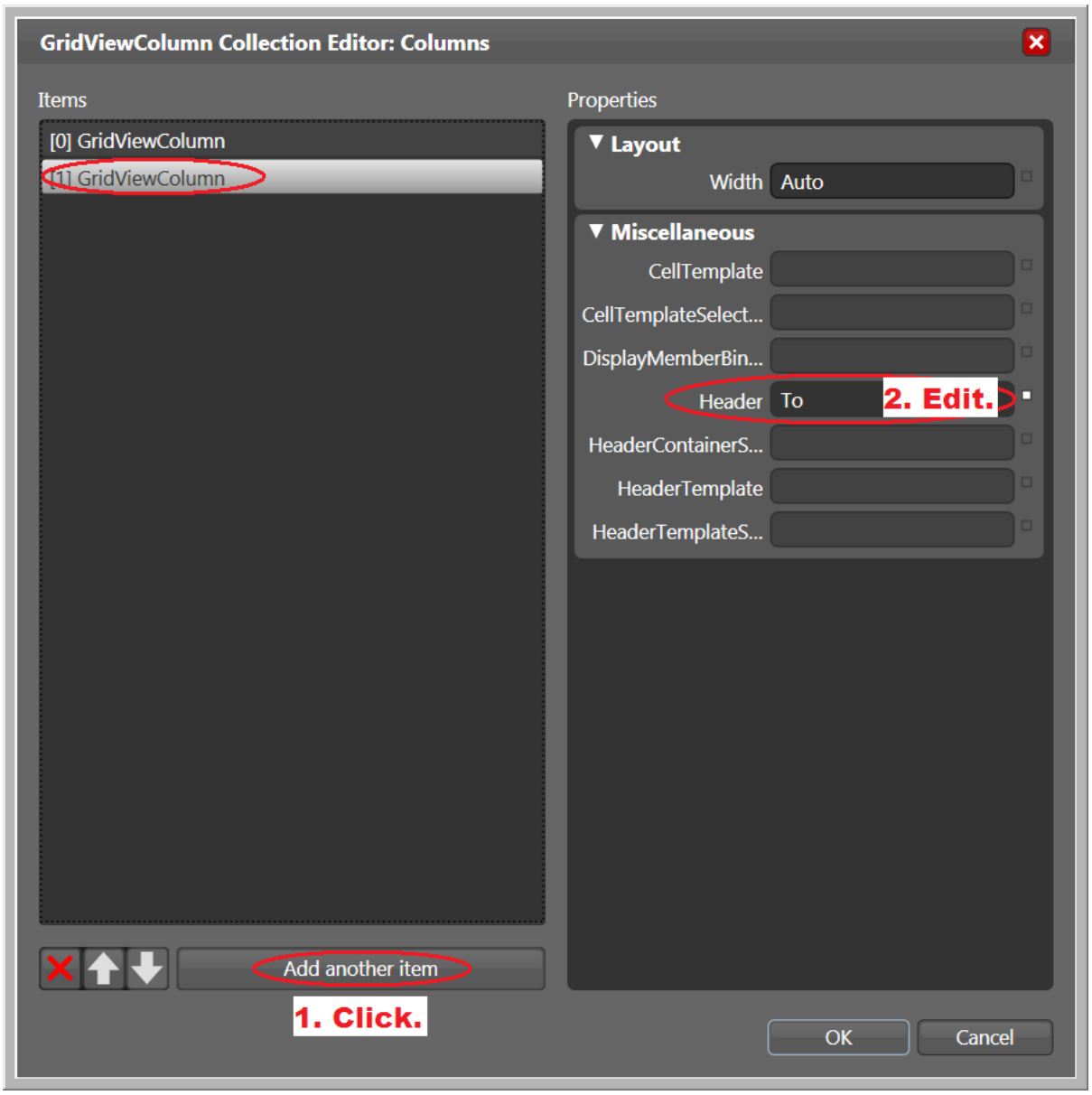


Figure 58 Adding a second column with Header = To.

4) Add a third column with **Header = Subject**.

5) Click **OK** to close the dialog box.

6) Data bind the **DisplayMemberBinding** property of each column to a mail property:

Due to a bug in Expression Blend we cannot do data binding for the **GridViewColumn.DisplayMemberBinding** property in Expression Blend, which means that we have to edit XAML file directly.

Under **Objects and Timeline**, right-click **[ListView]** to open the context menu, select **View XAML** to switch to the XAML view of **Window1.xaml**, press **F4** to get a larger view of the document. As you see, the **ListView** element is highlighted in blue in the background, and we need to add the following texts (in red) to data bind the **DisplayMemberBinding** property of each column to **From**, **To** and **Subject** elements of the XML data source.

basic10.txt

```
<ListView DataContext="{Binding Path=DataContext, ElementName=LayoutRoot,
Mode=Default}" BorderBrush="{DynamicResource MyDarkBlueSolidBrush}"
BorderThickness="0,0.5,0,0" DockPanel.Dock="Top" ItemsSource="{Binding Mode=OneWay}"
IsSynchronizedWithCurrentItem="True">
 <ListView.View>
    <GridView>
      <GridViewColumn Header="From" DisplayMemberBinding="{Binding XPath=From}" />
      <GridViewColumn Header="To" DisplayMemberBinding="{Binding XPath=To}" />
      <GridViewColumn Header="Subject" DisplayMemberBinding="{Binding XPath=Subject}"
/>
    </GridView>
 </ListView.View>
</ListView>
```

Build and run the project (**F5**), make sure that the list displays the **From**, **To** and **Subject** properties of each mail.

Task 11: Adding Controls to the Third Column of [Grid] to Display Mail Contents

The third column of **[Grid]** is used to display the details of the currently selected mail.

1. With **[Grid]** activated, add a **DockPanel** control to **[Grid]** and name it **dp3**.
2. Group the **DockPanel dp3** into a **Border**.
3. Configure the properties of **[Border]** as follows:

   Brushes

   - **BorderBrush = MyDarkBlueSolidBrush**

   Appearance

   - **BorderThickness = 0.5, 0.5, 0.5, 0.5**

   Layout

   - **Width = Auto, Height = Auto**
   - Row = 0, **RowSpan = 2**, **Column = 2**, ColumnSpan = 1
   - **HorizontalAlignment = Stretch, VerticalAlignment = Stretch**
   - **Margin = 0, 0, 0, 0**
4. Configure the properties of **dp3** as follows:

   Layout

   - **Width = Auto, Height = Auto**
   - HorizontalAlignment = Stretch, VerticalAlignment = Stretch
   - **LastChildFill = True**
   - Margin = 0, 0, 0, 0
5. With **dp3** activated, add a custom control **MyMailHeaderGridControl** to **dp3**, and configure its property as follows:

   Layout

   - **Dock = Top**
   - HorizontalAlignment = Stretch, **VerticalAlignment = Top**

   Common Properties

   - DataContext = MS.Internal.Data.XmlDataCollection

     Since **[MyMailHeaderGridControl]** is also a child control of **LayoutRoot**, it automatically inherits the data source information from **LayoutRoot.DataContext**. In the Properties Editor of **[MyMailHeaderGridControl]**, you can see **DataContext (XmlDataCollection)**, that's why no explicit data binding of **[MyMailHeaderGridControl].DataContext** is needed here.

     The **MyMailHeaderGridControl** has four **TextBlock** controls for displaying **Subject**, **From**, **Received** and **To** elements of the corresponding mail. Data bindings of the **TextBlock.Text** properties to **XPath=Subject**, **XPath=From**, **XPath=Received** and **XPath=To** have already been defined in the custom control.

6. With **dp3** activated, add a **Frame** control to **dp3**, and configure its properties as follows:

Frame

Frame is a content control that provides the ability to navigate to and display content. Content can be any type of .NET Framework 3.0 object and HTML files. In our project, the E-mails are HTML files located under the directory ***project folder*\OutlookUI HOL\Initial Projects*Current Project*\resources**.

Layout

- **Dock = Top**

Common Properties

- **NavigationUIVisiblity = Hidden**
- DataContext = MS.Internal.Data.XmlDataCollection (inherited from **LayoutRoot.DataContext**)

7. Data bind the **[Frame].Source** property to **XPath=File** and convert each **File** element to its corresponding file location which is then the actual value of the **[Frame].Source** property.

   In **inbox.xml**, the files for the mails are specified as **<File>mail*.htm</File>** in the **<mail>** elements. Since the actual file locations for these mails are ***project folder*\OutlookUI HOL\Initial Projects*Current Project*\resources\mail*.htm**, we need to first convert **mail*.htm** to the corresponding absolute path ***project folder*\OutlookUI HOL\Initial Projects*Current Project*\resources\mail*.htm** before setting the **[Frame].Source** property. This is done using a **Converter**.

Converter

To convert data during binding, you must create a class that implements the **IValueConverter** interface, which includes the **Convert** and **ConvertBack** methods. Converters can change data from one type to another, translate data based on cultural information, or modify other aspects of the presentation.

When implementing the **IValueConverter** interface, it is a good practice to decorate the implementation with a **ValueConversion** attribute to indicate to development tools the data types involved in the conversion.

1) Add the following **MyFrameSourceConverter** class to the **OutlookUI_HOL** namespace in **Window1.xaml.cs**:

basic11.txt

```
[ValueConversion(typeof(string), typeof(string))]
public class MyFrameSourceConverter : IValueConverter
{
   public object Convert(object value, Type targetType,
                         object parameter,
                         System.Globalization.CultureInfo culture)
   {
      string file = System.IO.Path.GetFullPath("..\\..\\.") +
                         "\\resources\\" + (string)value;
      return file;
   }

   public object ConvertBack(object value, Type targetType,
                         object parameter,
                         System.Globalization.CultureInfo culture)
   {
      return null;
   }
}
```

2) Build the project (**F5**) to make the above class be compiled.

3) Add the following text (in red) to **Window.Resources** in **Window1.xaml** to register the above converter as a resource:

basic11.txt

```
<Window … xmlns:local="clr-namespace:OutlookUI_HOL" … >
  <Window.Resources>
    <XmlDataProvider x:Key="inboxDS" d:IsDataSource="True"
                     Source="inbox.xml"/>
    <local:MyFrameSourceConverter x:Key="MyFrameSourceConverter" />
  </Window.Resources>
```

4) With **[Frame]** selected, click the small square beside the **Source** property to open the context menu, select **Custom Expression…** to open the dialog box, type

basic11.txt

```
{Binding XPath=File, Converter={StaticResource MyFrameSourceConverter}}
```

into the text box and press **Return**.

Build and run the application (**F5**). Select the list items one after another, make sure that the **[MyMailHeaderGridControl]** and the **[Frame]** controls update their contents as expected.

The following is a summary of what we have done in Task 10 and 11 in order to achieve **Master-Details View**:

1. Adding an XML data source to the project.
2. Data binding **LayoutRoot.DataContext** to the path **/inbox/mail** of the XML data source **inboxDS.**

   All the child elements of **LayoutRoot** then inherit the data source information from **LayoutRoot.DataContext** automatically. Their **DataContext** properties are all implicitly set to **MS.Internal.Data.XmlDataCollection**.
3. Data binding the **[ListView].ItemsSource** property to the explicit data context.
4. Setting the **[ListView].IsSynchronizedWithCurrentItem** property, inherited from **Selector**, to **true** to ensure that **[MyMailHeaderGridControl]** and **[Frame]** always refer to the selected item in **[ListView]**.

Selector.IsSynchronizedWithCurrentItem Property

The **Selector.IsSynchronizedWithCurrentItem** property gets or sets a value that indicates whether a **Selector** should keep the **SelectedItem** synchronized with the current item in the **Items** property.

You can set the **IsSynchronizedWithCurrentItem** property to **true** to ensure that the item selected always corresponds to the **CurrentItem** property in the **ItemCollection**. For example, suppose that there are two **ListView** controls with their **ItemsSource** property set to the same source. Set **IsSynchronizedWithCurrentItem** to **true** on both **ListView** controls to ensure that the selected item in each **ListView** is the same.

5. Writing **Binding** declarations for the elements which display mail information like sender, receiver, subject and mail content.

   1) For the **GridViewColumn** of **GridView** in **[ListView]** we have **Binding** declarations like

   ```
   <GridViewColumn Header="To" DisplayMemberBinding="{Binding XPath=To}" />
   ```

   2) For the **TextBlock** controls in **[MyMailHeaderGridControl]** we have **Binding** declarations like

   ```
   <TextBlock Text="{Binding XPath=Subject}" />
   ```

   3) For the **Frame** control we have the **Binding** declaration

   ```
   <Frame Source="{Binding XPath=File}" />
   ```

Congratulations! You have done great work! If you are eager for more WPF features, just continue with our Hands-On Lab. In the following advanced courses, you will learn how to customize a **Button** control and a **ListView** control.

Building the Sidebar (Advanced)

In the basic course you were asked to add the custom control **MySidebarControl** which consists of two buttons. It may seem simple to add two buttons, however, as you will see, there is more to be done to achieve an Outlook-style sidebar.

In this advanced course you will learn about

- customizing the control template of a **Button** control,
- template binding,
- **ContentPresenter**,
- multi-condition property triggers, and
- writing event handlers to display resp. hide the **MyNavigationPaneControl**.

First we must delete the old **mySidebarControl** from **[Grid]**: Under **Objects and Timeline**, right-click **mySidebarControl** to open the context menu, choose **Delete** to delete **mySidebarControl**.

Task 1: Building Sidebar Buttons

Figure 59 shows the look of the sidebar, its hierarchical control structure is depicted in Figure 60.

Figure 59 The sidebar.

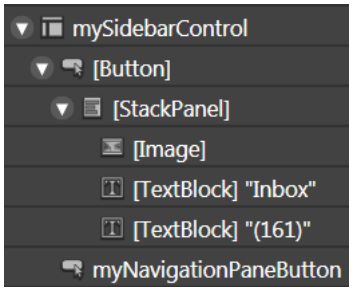


Figure 60 The hierarchical control structure of the sidebar.

1. With **[Grid]** activated, add a **DockPanel** control to **[Grid]**, configure its properties as follows:
   - **Name = mySidebarControl**

   Layout
   - **Width = Auto, Height = 27**
   - **HorizontalAlignment = Left, VerticalAlignment = Stretch**
   - **LastChildFill = True** (Checked)

- **Margin: left = 0, right = 0, top = 32, bottom = 7**

2. Add a **Button** control to **mySidebarControl**, configure its properties as follows:

   Brushes

   - **Background = MyBrightBlueSolidBrush2**
   - **BorderBrush = MyDarkBlueSolidBrush**

   Appearance

   - **BorderThickness = 0.5, 0.5, 0.5, 0.5**

3. Add a **StackPanel** control to **[Button]**, configure its properties as follows:

   Layout

   - **Width = Auto, Height = Auto**
   - **Orientation = Horizontal**

4. Add an **Image** control to **[StackPanel]**, configure its properties as follows:

   Layout

   - **Width = 16, Height = 16**
   - **HorizontalAlignment = Center, VerticalAlignment = Center**
   - **Margin**: **left = 4**, **right = 2**, top = 0, bottom = 0

   Common Properties

   - **Source = graphics\inbox.gif**
   - **Stretch = Fill**

5. Add a **TextBlock** control to **[StackPanel]**, configure its properties as follows:

   Brushes

   - **Foreground = MyDarkBlueSolidBrush**

   Layout

   - **HorizontalAlignment = Center, VerticalAlignment = Center**
   - **Margin**: **left = 2**, **right = 2**, top = 0, bottom = 0

   Common Properties

   - **Text = Inbox**

   Text

   - **FontWeight = Bold**
   - **TextWrapping = NoWrap**

6. Add a **TextBlock** control to **[StackPanel]**, configure its properties as follows:

   Brushes

   - **Foreground = #FF0000FF** (a blue solid color brush) 

Figure 61 Setting the Foreground property of the TextBlock control.

Layout

- **HorizontalAlignment = Center, VerticalAlignment = Center**
- **Margin**: **left = 2**, **right = 4**, top = 0, bottom = 0

Common Properties

- **Text = (161)**

Text

- **TextWrapping = NoWrap**

7. Add a **Button** control to **mySidebarControl**, configure its properties as follows:
   - **Name = myNavigationPaneButton**

   Brushes

   - **Background = MyBrightBlueSolidBrush2**
   - **BorderBrush = MyDarkBlueSolidBrush**
   - **Foreground = MyDarkBlueSolidBrush**

   Appearance

   - **BorderThickness = 0.5, 0.5, 0.5, 0.5**

   Layout

   - **Dock = Right**

   Common Properties

   - **Content = Navigation Pane**

   Text

   - **FontSize = 15**
   - **FontWeight = Bold**

8. Rotate the **mySidebarControl** control vertical.

   With **mySidebarControl** selected, expand the **Transform** category, under **LayoutTransform** subcategory, set the following properties:

   - **Rotate** : **Angle = -90**

9. Rotate the **[Image]** control created in step 4 90 degrees clockwise:

   With the **[Image]** control selected, expand the **Transform** category, under **LayoutTransform** subcategory, set the following properties:

   - **Rotate** : **Angle = 90**

Build and run the application (**F5**). Move mouse over the buttons in the sidebar, and click them. As you see, the two buttons have a different look and feel from the one of Outlook, so let's start configuring our own control template for the two buttons.

Task 2: Editing Control Template for the Sidebar Buttons

1. Open the control template editor for **[Button]**. 62 63

   Right-click **[Button]** to open the context menu, select **Edit Control Parts (Template)** > **Edit a Copy…**. In the dialog box, name the style **MySidebarButtonStyle**, and choose **MyStyles.xaml** as the resource dictionary.

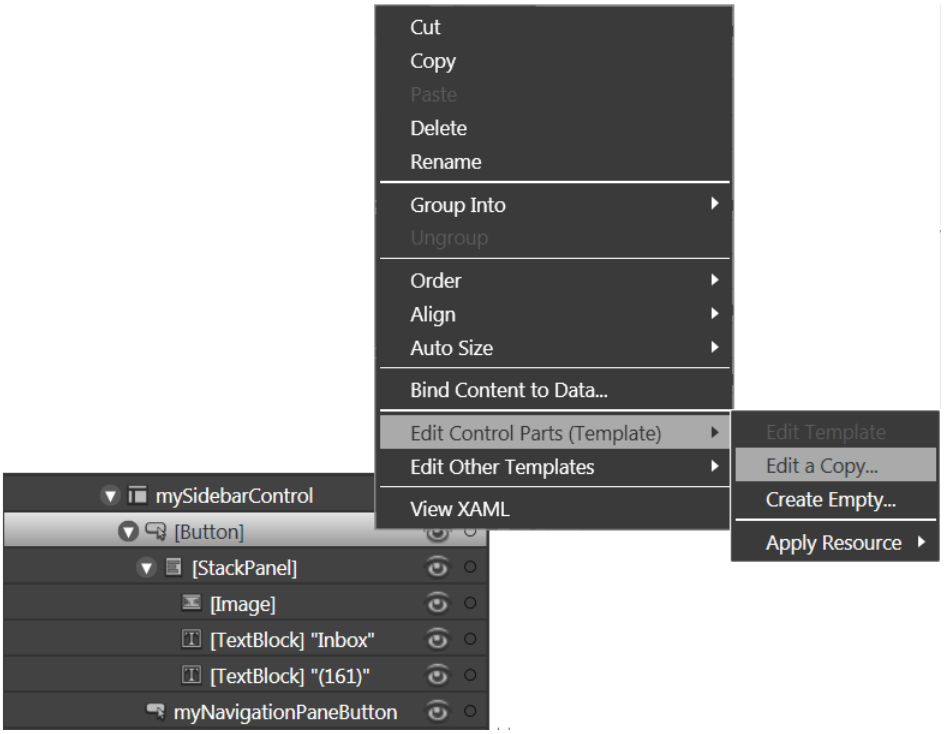


Figure 62 Opening the control template editor for the [Button] control.

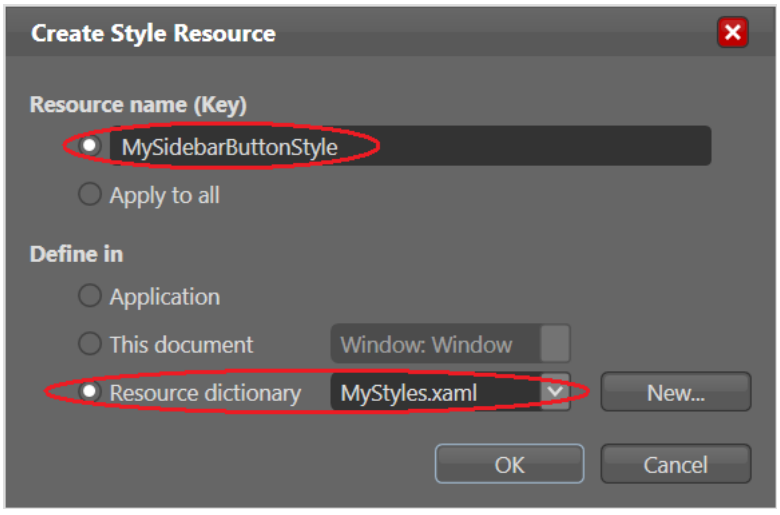


Figure 63 Defining the name and the resource dictionary for the MySidebarButtonStyle style.

The default control template for a button has the control structure as shown in Figure 64:

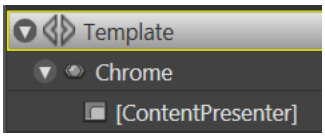


Figure 64 The control structure of the default button style.

For our project, we want the control structure to look like the following:

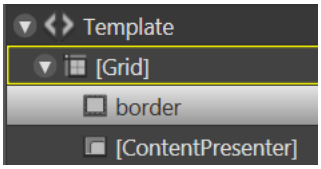


Figure 65 The control structure of our custom button style.

2. Under **Objects and Timeline**, right-click **Chrome** to open the context menu, choose **Delete** to delete **Chrome**.
3. Add a **Grid** control to the **Template**.
4. Add a **Border** control which you can find in the **Asset Library** to the **[Grid]** control, configure its properties as follows:

   Brushes

   - **Background = {TemplateBinding Background}**  **66**

     Click the small square beside the **Background** property to open the context menu, choose the submenu item **Template Binding** > **Background**.

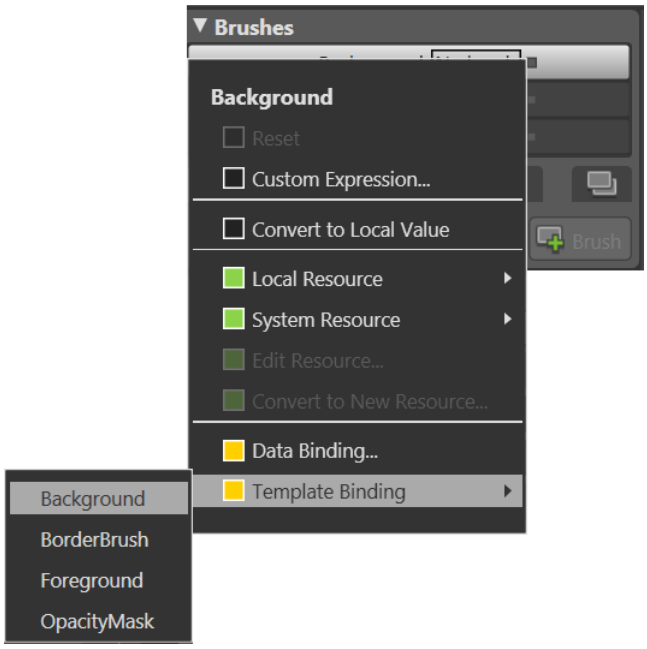


Figure 66 Template binding for the Background property.

The small square beside the **Background** property has turned orange, which indicates that the **Background** property is template-bound.

Template Binding

Template binding is used to bind properties in the template to the properties of the control to which the template is applied. In our case, template binding of **Border.Background** to **Button.Background** has the effect that when we set the **Background** property of a button whose style is set to **MySidebarButtonStyle**, we are actually referring to the **Background** property of the **Border** control in this control template.

- **BorderBrush = {TemplateBinding BorderBrush}**

Appearance

- **BorderThickness = {TemplateBinding BorderThickness}**

Layout

- **Margin = 0, 0, 0, 0**

5. Add a **ContentPresenter** control which you can find in the **Asset Library** to the **[Grid]** control, configure its properties as follows:

Layout

- **HorizontalAlignment = Center, VerticalAlignment = Center**
- **Margin = 0, 0, 0, 0**

Content Presenter

A content presenter is a placeholder in the control template to display the content of the control to which the template is applied. This might be the value of a **Content** property (in a **Button** for example), or a **Text** property (in a **TextBox**).

6. Add and customize the property trigger **IsPressed = True**. 67 68

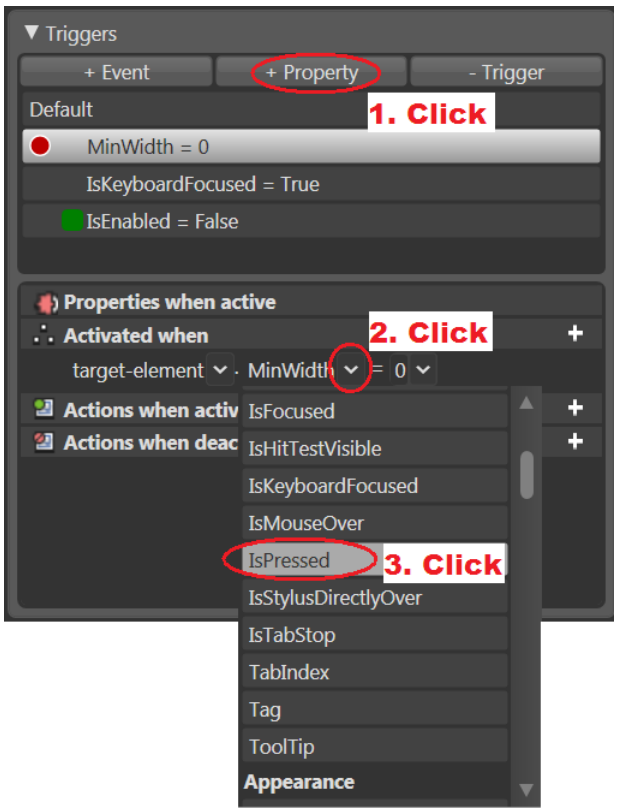


Figure 67 Adding the property trigger IsPressed = True.

As soon as a property trigger is added, its trigger recording is on, which records all the property changes for the case that the property trigger is activated.

Set the **Background** property of the **border** control to **MyDarkOrangeSolidBrush**, and then stop the trigger recording (see Figure 68).

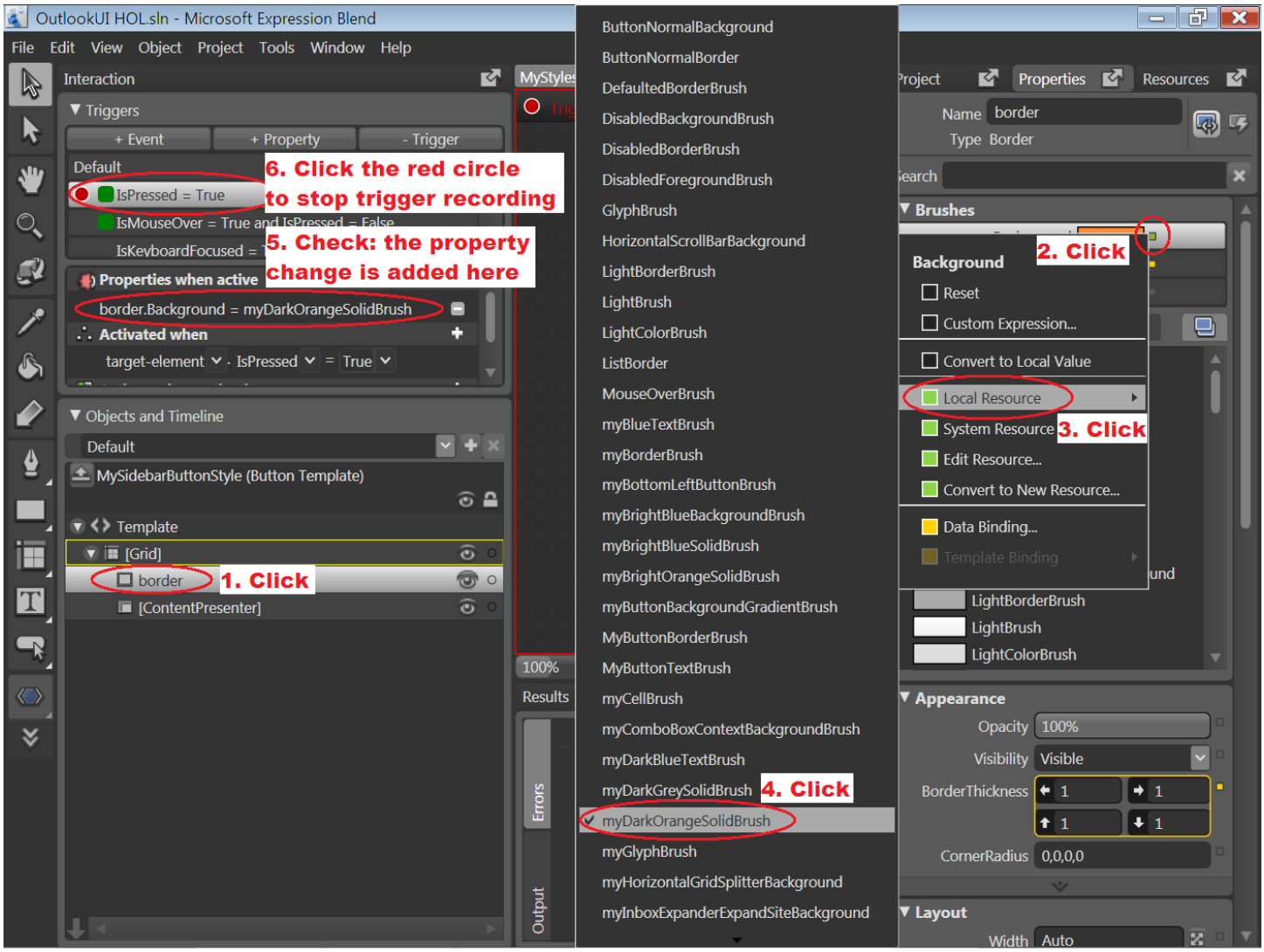


Figure 68 Customizing the property trigger IsPressed = True.

7. Add and customize the property trigger **IsMouseOver = True and IsPressed = False**.

   First add a property trigger **IsMouseOver = True** as described for **IsPressed = True** (see Figure 67), then click the button ✚ in ∴ Activated when ✚, then change **MinWidth** to **IsPressed** and change **True** to **False**. In this way you get a Boolean expression like **e1 and e2**, in our case, the expression means that the property trigger is activated, when the user moves mouse over the button and does not click it.

   With trigger recording on, set the **Background** property of the **border** control to **MyBrightOrangeSolidBrush**, and stop trigger recording.

8. Under **Objects and Timeline**, click ⬆ to exit the control template editor for **MySidebarButtonStyle**.

9. Apply **MySidebarButtonStyle** to **myNavigationPaneButton**. 69

   With **myNavigationPaneButton** selected, set the **Style** property under **Miscellaneous** category to **MySidebarButtonStyle**.

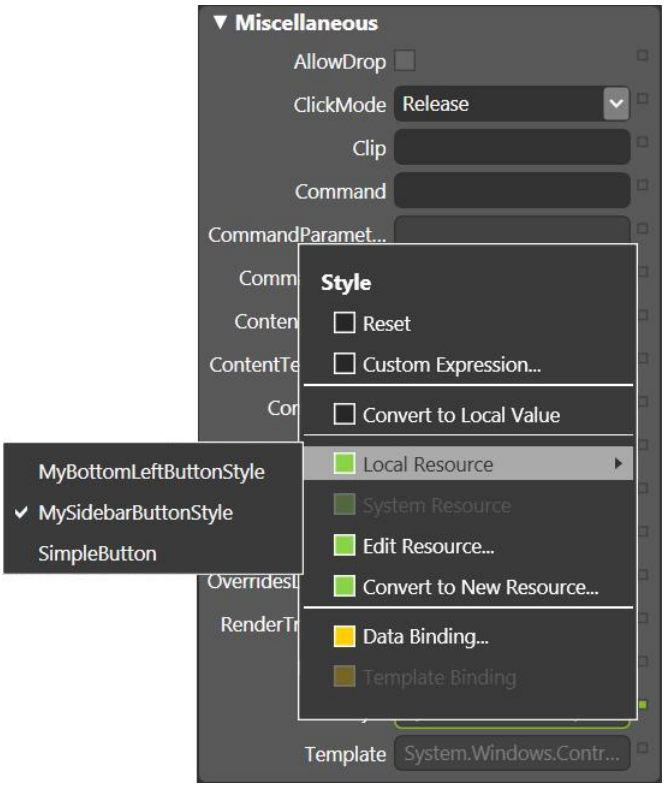


Figure 69 Setting the Style property of myNavigationPaneButton to MySidebarButtonStyle.

Build and run the application (**F5**). Move mouse over the buttons and click the buttons, make sure that the buttons have different background colors under different states.

10. Under **Appearance** category, set the **Visibility** property of **mySidebarControl** to **Hidden**.

Task 3: Writing Event Handlers to Display/Hide the Navigation Pane

1. Add the custom control **MyNavigationPaneControl** to **[Grid]**, configure its properties as follows:
   - **Name = myNavigationPaneControl**

   Appearance
   - **Visibility = Hidden**

   Layout
   - **Width = Auto**, Height = Auto
   - Row = 0, **RowSpan = 2,** Column = 0, **ColumnSpan = 3**
   - **Margin: left = 27**, right = 0, **top = 32**, bottom = 0
2. Write an event handler for the **myNavigationPaneButton.Click** event and add additional programmatic logic to **Window1.xaml.cs**:

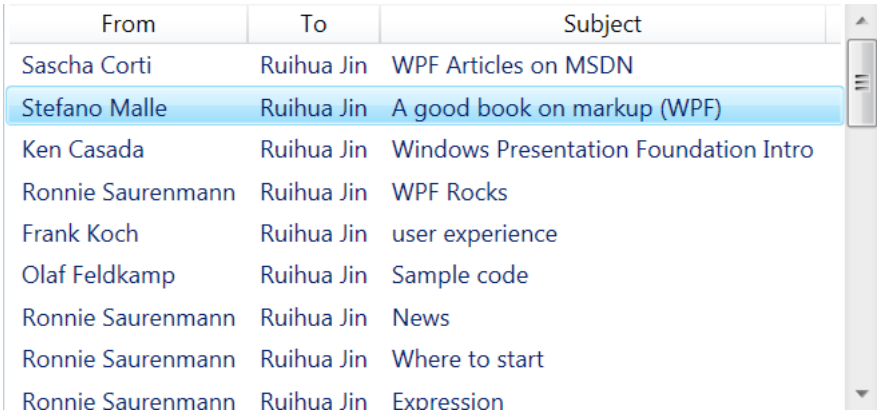
   sidebar.txt

```
private MouseButtonEventHandler mbEventHandler = null;
private DependencyPropertyChangedEventHandler visibilityEventHandler = null;

private void myNavigationPaneButton_Click(object sender, RoutedEventArgs e)
{
  if (mbEventHandler == null) {
    mbEventHandler =
        new MouseButtonEventHandler(LayoutRoot_PreviewMouseLeftButtonUp);
  }
```

```
    if (visibilityEventHandler == null) {
      visibilityEventHandler = new
DependencyPropertyChangedEventHandler(myNavigationPaneControl_IsVisibleChanged);
    }
    if (myNavigationPaneControl.Visibility == Visibility.Hidden) {
      myNavigationPaneButton.SetValue(Button.BackgroundProperty,
                  (Brush)MyApp.Current.Resources["MyOrangeSolidBrush"]);
      myNavigationPaneControl.Visibility = Visibility.Visible;
      LayoutRoot.PreviewMouseLeftButtonUp += mbEventHandler;
      myNavigationPaneControl.IsVisibleChanged += visibilityEventHandler;
    } else {
      hideNavigationPane();
    }
  }

  private void myNavigationPaneControl_IsVisibleChanged(object sender,
                          DependencyPropertyChangedEventArgs e)
  {
    /* if myNavigationPaneControl becomes hidden, then
     * the look of the application UI is to be reset and
     * the two event handlers are to be deregistered, which is done in the method
     * hideNavigationPane().
     */
    if (myNavigationPaneControl.Visibility == Visibility.Hidden) {
      hideNavigationPane();
    }
  }

  private void LayoutRoot_PreviewMouseLeftButtonUp(object sender,
                    System.Windows.Input.MouseButtonEventArgs e)
  {
    /* if the user clicks somewhere on LayoutRoot except on myNavigationPaneButton
     * and except on myNavigationPaneControl, and
     * myNavigationPaneControl is visible, then
     * myNavigationPaneControl is to be hidden.
     */
    if (e.OriginalSource != myNavigationPaneButton
         && e.Source != myNavigationPaneControl) {
      if (myNavigationPaneControl.Visibility == Visibility.Visible) {
        hideNavigationPane();
      }
    }
  }

  private void hideNavigationPane()
  {
    myNavigationPaneButton.SetValue(Button.BackgroundProperty,
              (Brush)MyApp.Current.Resources["MyBrightBlueSolidBrush2"]);
    myNavigationPaneControl.Visibility = Visibility.Hidden;
    LayoutRoot.PreviewMouseLeftButtonUp -= mbEventHandler;
    myNavigationPaneControl.IsVisibleChanged -= visibilityEventHandler;
  }
```

Customizing the [ListView] Control (Advanced)

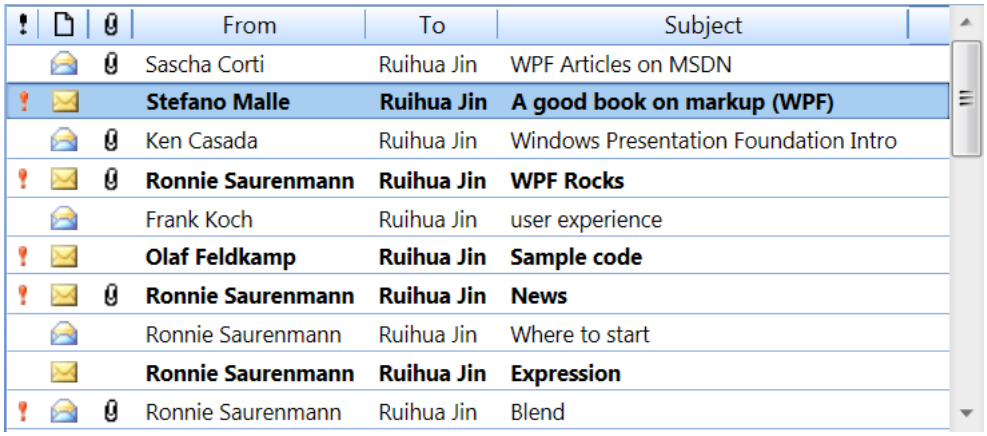
In the basic course, we have added list items to the **[ListView]** control without considering the style of **[ListView]**. The default style of a **ListView** control looks like the following:



From	To	Subject
Sascha Corti	Ruihua Jin	WPF Articles on MSDN
Stefano Malle	Ruihua Jin	A good book on markup (WPF)
Ken Casada	Ruihua Jin	Windows Presentation Foundation Intro
Ronnie Saurenmann	Ruihua Jin	WPF Rocks
Frank Koch	Ruihua Jin	user experience
Olaf Feldkamp	Ruihua Jin	Sample code
Ronnie Saurenmann	Ruihua Jin	News
Ronnie Saurenmann	Ruihua Jin	Where to start
Ronnie Saurenmann	Ruihua Jin	Expression

Figure 70 The [ListView] control with default style.

Our goal is to achieve the MS Outlook 2007-Style:



From	To	Subject
Sascha Corti	Ruihua Jin	WPF Articles on MSDN
Stefano Malle	**Ruihua Jin**	**A good book on markup (WPF)**
Ken Casada	Ruihua Jin	Windows Presentation Foundation Intro
Ronnie Saurenmann	**Ruihua Jin**	**WPF Rocks**
Frank Koch	Ruihua Jin	user experience
Olaf Feldkamp	**Ruihua Jin**	**Sample code**
Ronnie Saurenmann	**Ruihua Jin**	**News**
Ronnie Saurenmann	Ruihua Jin	Where to start
Ronnie Saurenmann	**Ruihua Jin**	**Expression**
Ronnie Saurenmann	Ruihua Jin	Blend

Figure 71 The Outlook-Style ListView.

In this advanced course you will learn about

- customizing the style for column headers
- customizing the style for list items
- adding images to the columns to represent data
- sorting list

Among the rest, two new important concepts involved are

- data triggers,
- data templating

Task 1: Editing the Style of Column Headers

1. Enter the **Template Editor** for **[ListView]**.

   Under **Objects and Timeline**, right-click **[ListView]** to open the context menu, choose **Edit Control Parts (Template)** > **Edit a Copy…**. In the dialog box, name the style **MyListViewStyle**, and choose **MyStyles.xaml** as resource dictionary.

   The default control structure of a **ListView** control looks like the following:

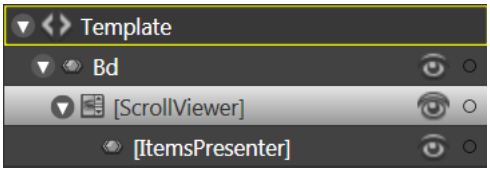


Figure 72 The default control structure of the [ListView] control.

2. Edit **MyListViewScrollViewerStyle.**

   1) Enter the **Template Editor** for **[ScrollViewer]**.

      Under **Objects and Timeline**, right-click **[ScrollViewer]** to open the context menu, choose **Edit Control Parts (Template)** > **Edit a Copy…**. In the dialog box, name the style **MyListViewScrollViewerStyle**, and choose **MyStyles.xaml** as resource dictionary.

      The default control structure of a **ScrollViewer** control looks like the following:

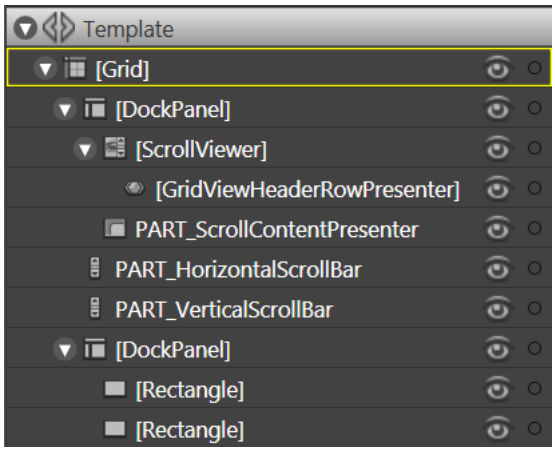


Figure 73 The control structure of the [ScrollViewer] control.

   2) Configure the properties of **[GridViewHeaderRowPresenter]** as follows:

      Layout

      - **Margin = 0, 0, 0, 0**

3. Edit **MyGridViewColumnHeaderStyle.**

   1) Enter the **Style Editor** for **ColumnHeaderContainerStyle**.  74

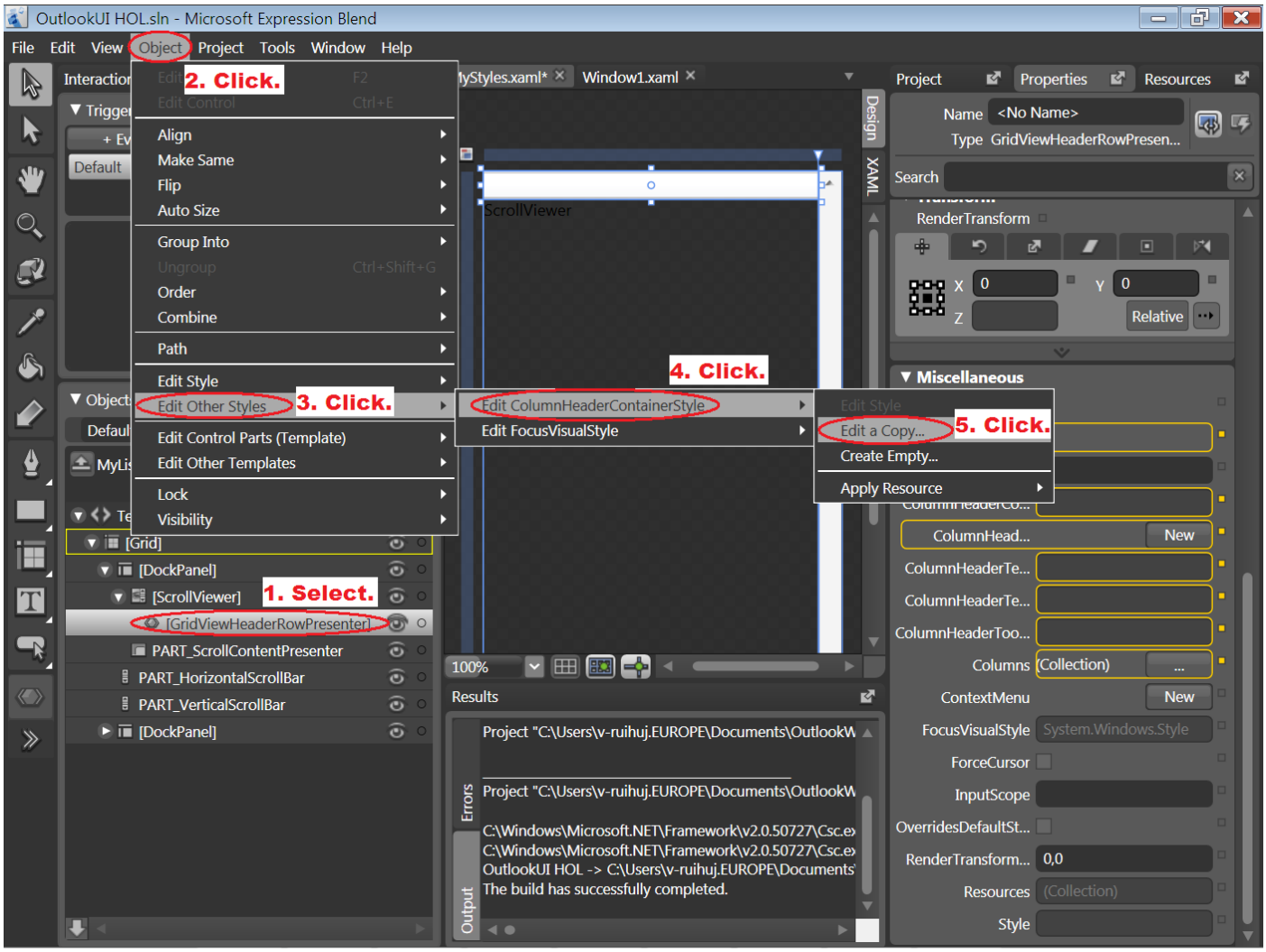


Figure 74 Entering the Style Editor for ColumnHeaderContainerStyle.

In the dialog box, name the style **MyGridViewColumnHeaderStyle**, and choose **MyStyles.xaml** as resource dictionary.

2) Configure the properties of **MyGridViewColumnHeaderStyle** as follows:

Brushes

- **Background = MyGridViewColumnHeaderBlueGradientBrush**
- **BorderBrush = MyDarkBlueSolidBrush**

Appearance

- **BorderThickness:** left = 0, right = 0, **top = 0.5, bottom = 0.5**

4. Edit **MyGridViewColumnHeaderControlTemplate**.

1) Enter the **Template Editor** for **MyGridViewColumnHeaderStyle**.

Under **Objects and Timeline**, right-click **Style** to open the context menu, choose **Edit Control Parts (Template) > Edit a Copy…**. In the dialog box, name the control template **MyGridViewColumnHeaderControlTemplate**, and choose **MyStyles.xaml** as resource dictionary.

The default control structure of **MyGridViewColumnHeaderControlTemplate** looks like the following:

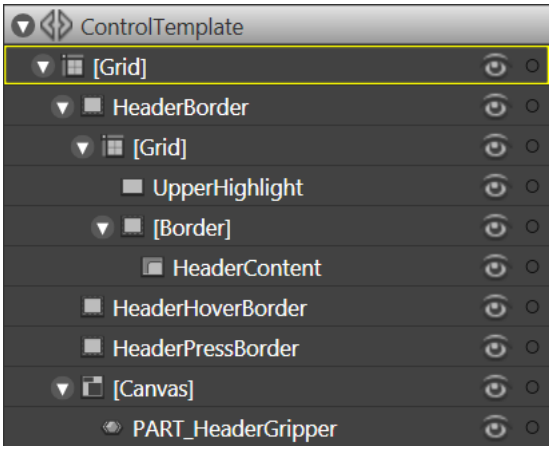


Figure 75 The default control structure of MyGridViewColumnHeaderControlTemplate.

2) Delete **UpperHighlight**, **HeaderHoverBorder** and **HeaderPressBorder**.

3) Configure the properties of **HeaderBorder** as follows:

Brushes

- **Background = MyGridViewColumnHeaderBlueGradientBrush**

Appearance

- **BorderThickness = 0, 0, 0, 0**

4) Configure the properties of **[Border]** which is the parent control of **HeaderContent** as follows:

Brushes

- **BorderBrush = MyDarkBlueSolidBrush**

Appearance

- **BorderThickness:** left = 0, right = 0, **top = 0.5, bottom = 0.5**

5) Configure the properties of **PART_HeaderGripper** which is used to denote the end of a column header as follows:

Brushes

- **Background = MyBlueSolidBrush1**

Layout

- **Width = 1, Height = 16**
- **Left = {Binding Path=ActualWidth, ElementName=Grid, Mode=Default}**  **76**

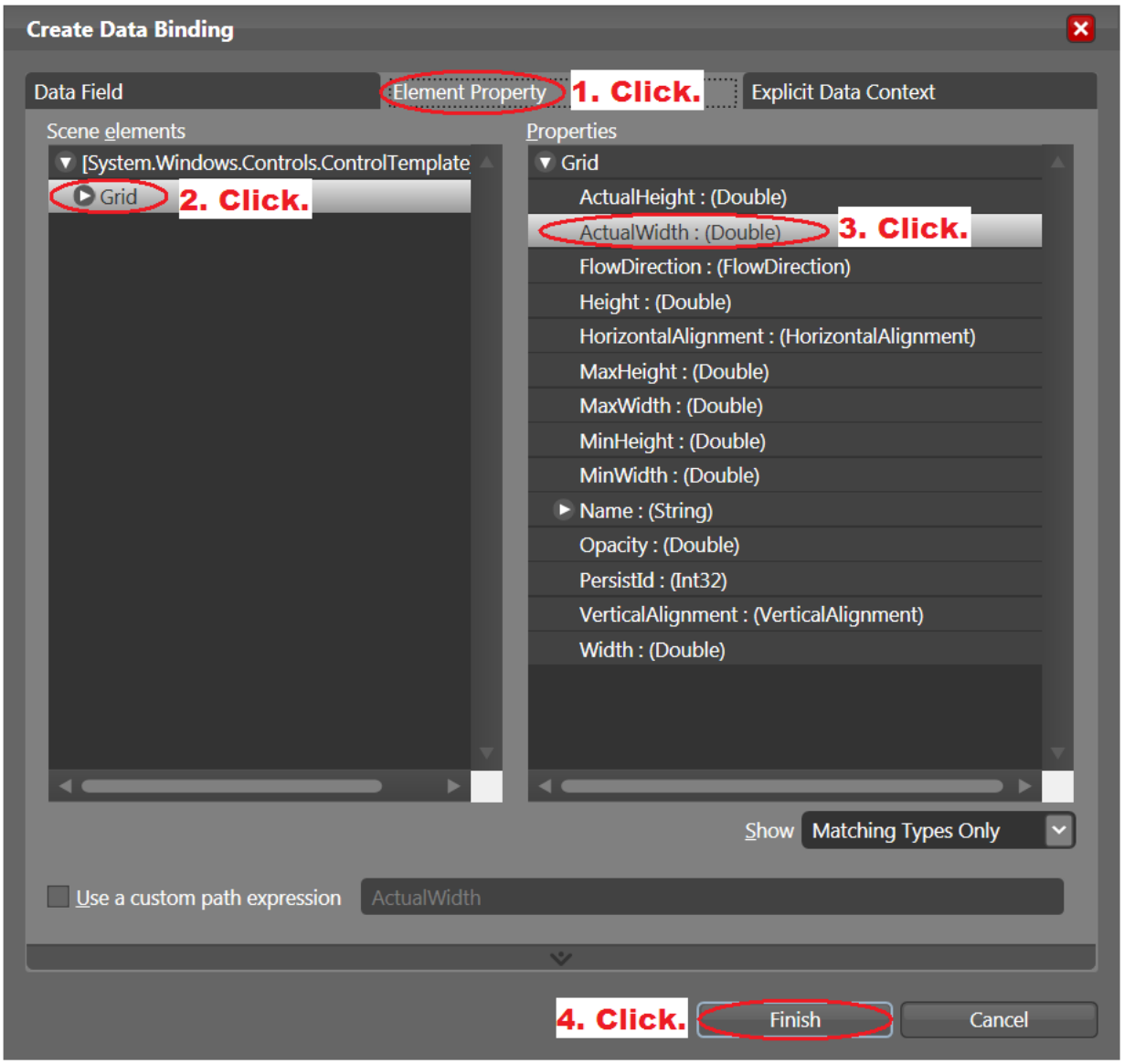


Figure 76 Data binding PART_HeaderGripper.Left to Grid.ActualWidth.

- **Margin: left = -1, right = 0, top = 2, bottom = 2**

6) Delete the property changes of the property trigger **IsMouseOver = True**. 77

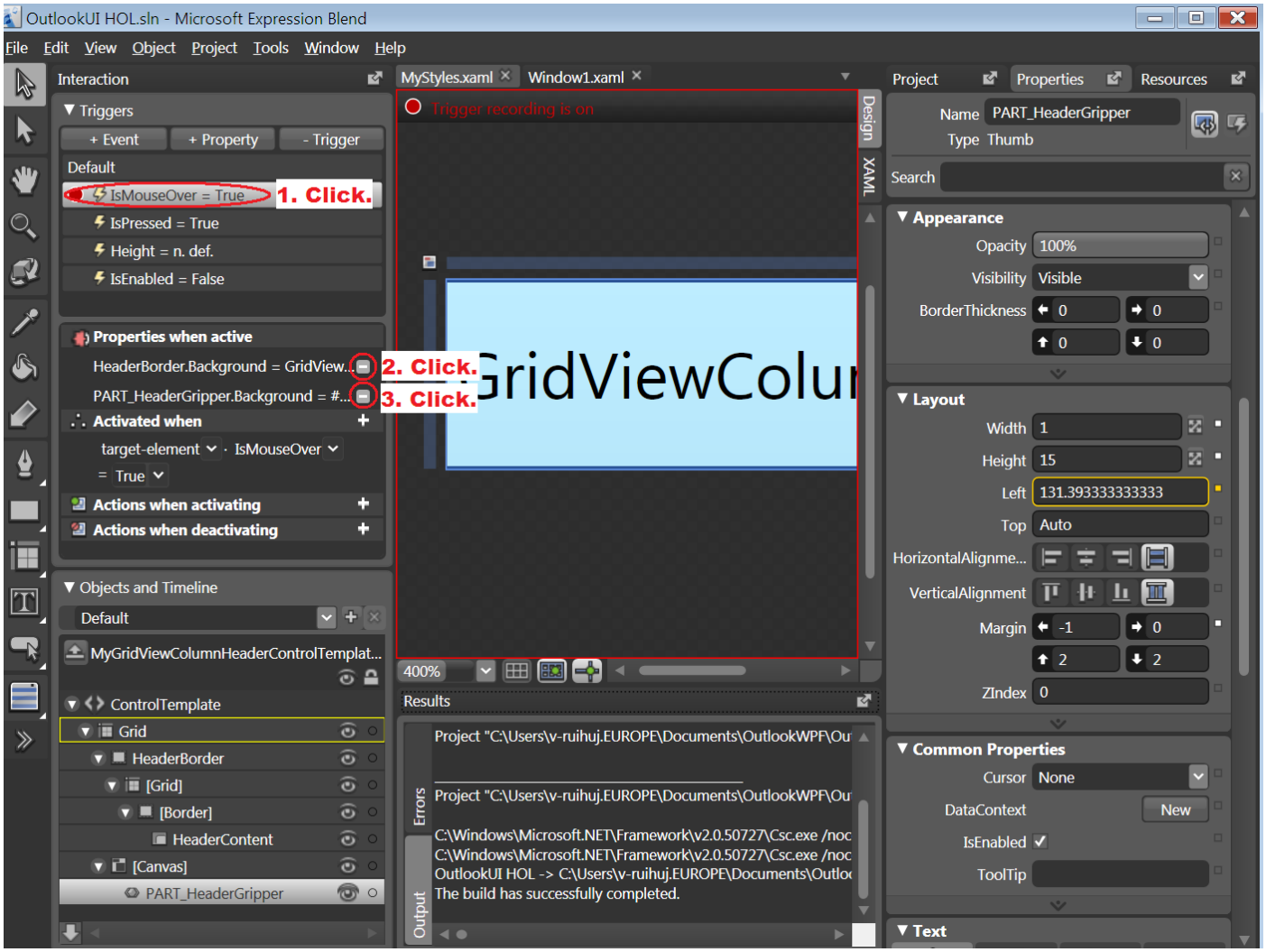


Figure 77 Deleting the default property changes of the property trigger IsMouseOver = True.

7) Edit the property trigger **IsPressed = True**.  78

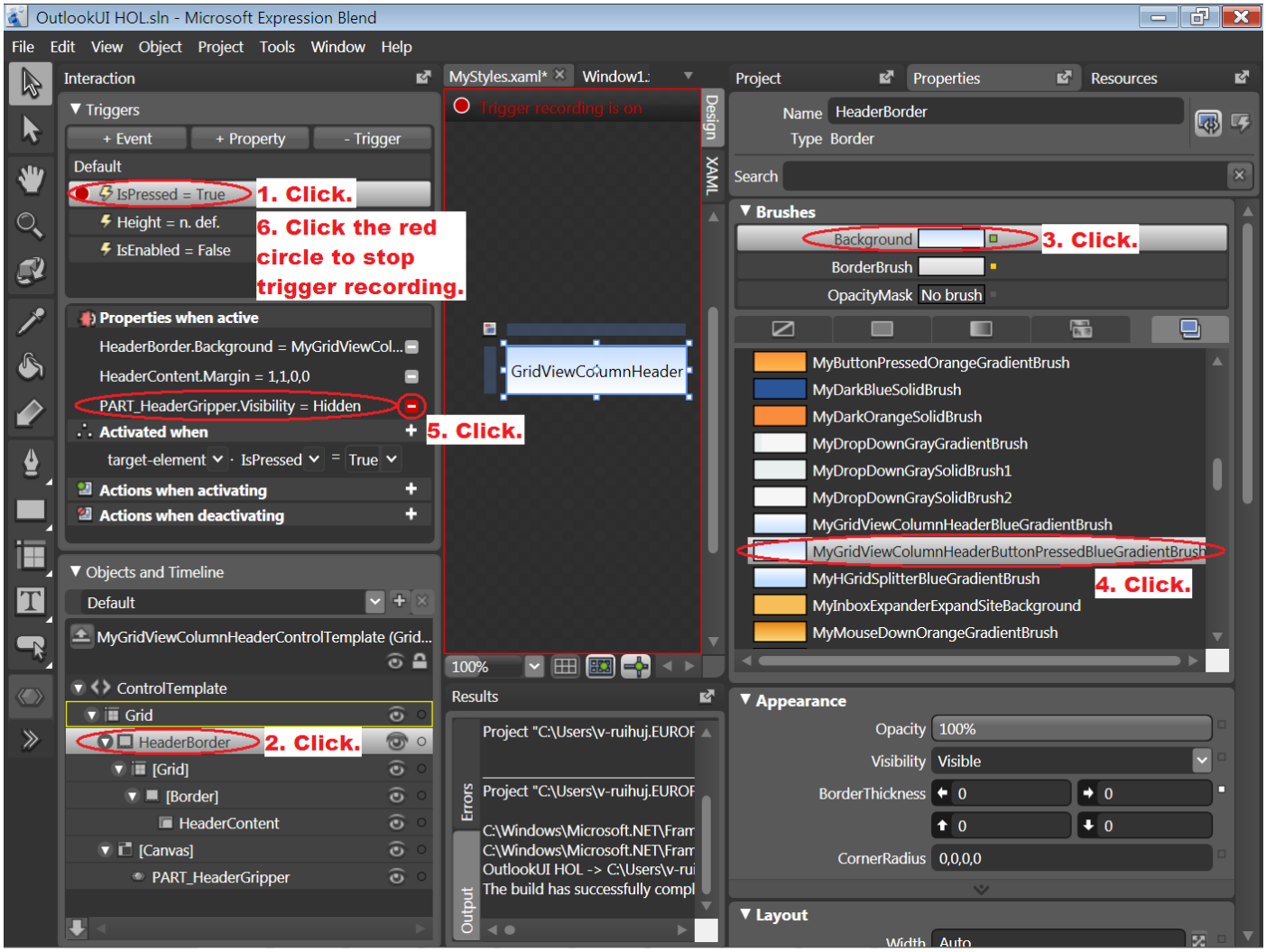


Figure 78 Editing the property trigger IsPressed = True.

8) Under **Objects and Timeline**, click  (Scope up) to exit the **Template Editor** for **MyGridViewColumnHeaderControlTemplate.**

Build and run the application (**F5**), make sure that the column headers **From**, **To** and **Subject** have the same look-and-feel as the one in MS Outlook 2007.

Task 2: Editing ItemContainerStyle of [ListView]

In this task we configure the **ItemContainerStyle** of **[ListView]** to customize the look of a selected list item and the one of unselected list items. Furthermore, like in MS Outlook, we will make the unread E-mails be displayed in bold.

1. Enter the **Style Editor** for **ItemContainerStyle** of **[ListView]**.

   With **[ListView]** selected, open the **Object** menu, select **Edit Other Styles** > **Edit ItemContainerStyle** > **Create Empty…**. In the dialog box, name the style **MyListViewItemContainerStyle**, and choose **MyStyles.xaml** as resource dictionary.

2. Configure the properties as follows:

   Layout

   - **HorizontalContentAlignment = Stretch**
   - **Padding: left = 0,** right = 0, **top = 1.5, bottom = 1.5**

   Text

   - **FontSize = 11**

3. Edit the control template for **MyListViewItemContainerStyle**.

   1) Enter the control template editor for **MyListViewItemContainerControlTemplate**.

      Under **Objects and Timeline**, right-click **Style** to open the context menu, choose **Edit Control Parts (Template)** > **Edit a Copy…**. In the dialog box, name the control template **MyListViewItemContainerControlTemplate**, and choose **MyStyles.xaml** as resource dictionary.

   2) Delete **[ContentPresenter]**.

   3) With **Bd** activated, add **GridViewRowPresenter** to **Bd**.

   4) Configure the properties of **Bd** as follows:

      Brushes

      - **BorderBrush = MyBlueSolidBrush1**

      Appearance

      - **BorderThickness**: left = 0, right = 0, top = 0, **bottom = 0.5**

   5) Delete the default property changes of the property trigger **IsSelected = True**, and then add the following property changes to the property trigger **IsSelected = True**:

      Brushes

      - **Bd.Background = MySelectedListItemBlueSolidBrush**
      - **Bd.BorderBrush = Black** (Enter **Black** into the **Custom Expression** dialog box)

      Appearance

      - **Bd.BorderThickness = 0.5, 0.5, 0.5, 0.5** (Enter **0.5** into the **Custom Expression** dialog box)

      Layout

      - **Bd.Margin:** left = 0, right = 0, **top = -1**, bottom = 0

   6) Delete the default property changes of the property trigger

      IsSelected = True and IsSelectionActive = False, and then add the following property changes to the property trigger:

      Brushes

      - **Bd.Background = MyGraySolidBrush**
      - **Bd.BorderBrush = MyBlueSolidBrush1**

   7) Exit the control template editor.

4. Make the unread E-mails be displayed in bold.

   With **MyStyles.xaml** being the current editable file, switch to the **XAML View**. Locate **MyListViewItemContainerStyle** (open the **Find** dialog box by pressing **Ctrl+F**, then type **MyListViewItemContainerStyle** into the text box and click **Find Next**), and then add the following red text to **MyListViewItemContainerStyle**:


```
<Style x:Key="MyListViewItemContainerStyle" TargetType="{x:Type ListViewItem}">
  <Setter Property="HorizontalContentAlignment" Value="Stretch"/>
  <Setter Property="Padding" Value="0,1.5,0,1.5"/>
  <Setter Property="FontSize" Value="11"/>
  <Setter Property="Template" Value="{DynamicResource
MyListViewItemContainerControlTemplate}"/>
  <Style.Triggers>
    <DataTrigger Binding="{Binding XPath=Read}" Value="false">
      <Setter Property="TextBlock.FontWeight" Value="Bold"/>
    </DataTrigger>
  </Style.Triggers>
</Style>
```

DataTrigger

Style, **ControlTemplate**, and **DataTemplate** all have a triggers collection. A **DataTrigger** allows you to set property values when the property value of the data object matches a specified value. In our example, we set the font weight of a list item to bold if the Read property of the corresponding mail has the value false.

Note that you must specify both the **Binding** and **Value** properties on a **DataTrigger** for the data trigger to be meaningful. If one or both of the properties are not specified, an exception is thrown. The **Setter**s are used to specify the property values which take effect when the related data object meets the specified condition.

Task 3: Adding Image Columns for Importance, Read, Attachment to [ListView]

The first column of **[ListView]** should display ! if the E-mail is marked as important. To do this, we need to introduce another important concept in WPF: **Data Templating**.

Data Templating

DataTemplate is about the presentation and appearance of the data objects and is one of the many features provided by the WPF styling and templating model. What you specify in your **DataTemplate** becomes the visual structure of your data object. WPF controls have built-in functionality to support the customization of data presentation.

1. Create a data template **MyImportanceCellDataTemplate**.

   Our data template for the cells in the **Importance** column has an **Image** control called **myImage**, it is used to show whether the mail is marked as important or not. The **Image** control **myImage** shows no image if the **Importance** element of the related **mail** has the value **false**, and it shows "**graphics\important.gif**" if **Importance** is equal to **true**. This means that the value of **myImage.Source** depends on the value of **Importance** of the **mail**, which triggers us to use a data trigger to define different property values for different cases.

   Switch to the **XAML View** of **[ListView]**, and add the following red text to **[ListView]** (since the data template is only used in **[ListView]**, it makes sense to define the data template locally):


```
<ListView … >
  <ListView.Resources>
    <DataTemplate x:Key="MyImportanceCellDataTemplate">
      <Image x:Name="myImage" Stretch="None" />
      <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding XPath=Importance}" Value="true" >
          <Setter Property="Source" TargetName="myImage"
                  Value="graphics\important.gif" />
        </DataTrigger>
      </DataTemplate.Triggers>
    </DataTemplate>
  </ListView.Resources>
  <ListView.View>…</ListView.View>
</ListView>
```

2. Add the following red text to **[ListView]** in **Window1.xaml** to create an image column:


```
<ListView … >
  …
  <ListView.View>
    <GridView>
      <GridViewColumn CellTemplate="{DynamicResource
                                     MyImportanceCellDataTemplate}" >
        <GridViewColumn.Header>
          <Image Source="graphics\importance.gif" Stretch="None" />
        </GridViewColumn.Header>
      </GridViewColumn>
      …
    </GridView>
  </ListView.View>
</ListView>
```

The second column of **[ListView]** shows  if the mail is read (**Read = true**), and  if the mail has not been read by the user (**Read = false**).

1. Add the following text to **ListView.Resources** of **[ListView]**:

```
<DataTemplate x:Key="MyReadCellDataTemplate">
  <Image x:Name="myImage" Stretch="None" />
  <DataTemplate.Triggers>
    <DataTrigger Binding="{Binding XPath=Read}" Value="true" >
      <Setter Property="Source" TargetName="myImage"
              Value="graphics\read.gif" />
    </DataTrigger>
    <DataTrigger Binding="{Binding XPath=Read}" Value="false" >
      <Setter Property="Source" TargetName="myImage"
              Value="graphics\unread.gif" />
    </DataTrigger>
  </DataTemplate.Triggers>
</DataTemplate>
```

2. Add the following red text to **[ListView]** in **Window1.xaml** to create a second image column:

```
<ListView … >
  <ListView.View>
    <GridView>
      …
      <GridViewColumn CellTemplate="{DynamicResource MyReadCellDataTemplate}">
        <GridViewColumn.Header>
          <Image Source="graphics\mailType.gif" Stretch="None" />
        </GridViewColumn.Header>
      </GridViewColumn>
      <GridViewColumn Header="From" … />
    </GridView>
  </ListView.View>
</ListView>
```

The third column of **[ListView]** is also an image column, it is used to show whether the mail has an attachment sent with it. If yes (**mail.Attachment=*file name***), then  is displayed, otherwise (**mail.Attachment=-1**), no image is shown.

1. Add the following text to **ListView.Resources** of **[ListView]**:

```
<DataTemplate x:Key="MyAttachmentCellDataTemplate">
  <Image x:Name="myImage" Source="graphics\attachment.gif" Stretch="None" />
  <DataTemplate.Triggers>
    <DataTrigger Binding="{Binding XPath=Attachment}" Value="-1" >
      <Setter Property="Source" TargetName="myImage" Value="{x:Null}" />
    </DataTrigger>
  </DataTemplate.Triggers>
</DataTemplate>
```

2. Add the following red text to **[ListView]** in **Window1.xaml** to create a third image column:

```
<ListView … >
  <ListView.View>
    <GridView>
      …
      <GridViewColumn CellTemplate="{DynamicResource
                          MyAttachmentCellDataTemplate}">
        <GridViewColumn.Header>
          <Image Source="graphics\attachment.gif" Stretch="None" />
        </GridViewColumn.Header>
      </GridViewColumn>
      <GridViewColumn Header="From" … />
      …
    </GridView>
  </ListView.View>
</ListView>
```

Task 4: Sorting List

A list without sorting functionality is hardly a useful list, so the next step is to add the sorting functionality to allow the user to sort the list by **Importance**, **Read**, **Attachment**, **Sender**, **Receiver** and **Subject**.

The list is sorted when the user clicks certain column header. Depending on the sort order (ascending resp. descending) we add up resp. down arrow to the column header as shown in Figure 79:

Figure 79 Column header with up- resp. down-arrow.

The different presentations of the column headers are again achieved by using data templates.

1. Add the following data templates to **ListView.Resources** of **[ListView]**:

   listview4.txt

```
<DataTemplate x:Key="MyArrowUpColumnHeaderTemplate">
  <StackPanel Orientation="Horizontal" >
    <TextBlock Text="{Binding}" />
    <Path StrokeThickness="1" Fill="{DynamicResource MyBlueSolidBrush2}"
          Data="M 0,4 L 3.5,0 L 7,4 Z" Margin="10,7,0,0"/>
  </StackPanel>
</DataTemplate>

<DataTemplate x:Key="MyArrowDownColumnHeaderTemplate">
  <StackPanel Orientation="Horizontal" >
    <TextBlock Text="{Binding}" />
    <Path StrokeThickness="1" Fill="{DynamicResource MyBlueSolidBrush2}"
          Data="M 0,0 L 3.5,4 L 7,0 Z" Margin="10,7,0,0"/>
  </StackPanel>
</DataTemplate>
```

2. Add the following red text to the **[ListView]** element in **Window1.xaml** to register the event handler **myGridViewColumnHeader_Click**:

   listview4.txt

```
<ListView ButtonBase.Click="myGridViewColumnHeader_Click" x:Name="myListView" … >
```

3. Add the following code to the **Window1** class in **Window1.xaml.cs** for the sorting functionality of **myListView**:

   listview4.txt

```
private GridViewColumnHeader lastHeaderClicked = null;
private ListSortDirection lastDirection = ListSortDirection.Ascending;

private void myGridViewColumnHeader_Click(object sender, RoutedEventArgs e)
{
  GridViewColumnHeader headerClicked = e.OriginalSource as GridViewColumnHeader;
  ListSortDirection direction;

  if (headerClicked != null) {
    if (headerClicked.Role != GridViewColumnHeaderRole.Padding) {
      // determine the sort order
      if (headerClicked != lastHeaderClicked) {
        direction = ListSortDirection.Ascending;
      } else {
```

```
            if (lastDirection == ListSortDirection.Ascending) {
              direction = ListSortDirection.Descending;
            } else {
              direction = ListSortDirection.Ascending;
            }
          }

          // sort the list
          if (headerClicked.Column.Header is string) {
            string header = headerClicked.Column.Header as string;
            Sort(header, direction);
          } else if (headerClicked.Column.Header is Image) {
            Image header = headerClicked.Column.Header as Image;
            string type = header.Source.ToString();
            if (type.Contains("attachment")) {
              Sort("Attachment", direction);
            } else if (type.Contains("mailType")) {
              Sort("Read", direction);
            } else if (type.Contains("importance")) {
              Sort("Importance", direction);
            } else {
              return;
            }
          } else {
            return;
          }

          // Add arrow to the column header if it is a string
          if (headerClicked.Column.Header is string) {
            if (direction == ListSortDirection.Ascending) {
              headerClicked.Column.HeaderTemplate =
((ListView)sender).Resources["MyArrowUpColumnHeaderTemplate"] as DataTemplate;
            } else {
              headerClicked.Column.HeaderTemplate =
((ListView)sender).Resources["MyArrowDownColumnHeaderTemplate"] as DataTemplate;
            }
          }
          // Remove arrow from previously sorted header
          if (lastHeaderClicked != null && lastHeaderClicked != headerClicked) {
            lastHeaderClicked.Column.HeaderTemplate = null;
          }

          // Update sorting information
          lastHeaderClicked = headerClicked;
          lastDirection = direction;
        }
      }
    }

    private void Sort(string sortBy, ListSortDirection direction)
    {
      ICollectionView dataView =
           CollectionViewSource.GetDefaultView(myListView.ItemsSource);
      dataView.SortDescriptions.Clear();
      SortDescription sd = new SortDescription(sortBy, direction);
      dataView.SortDescriptions.Add(sd);
    }
```

It is really admirable that you have stayed the course and completed this Hands-On Lab. We hope that you have enjoyed the lab. WPF is a very nice framework for developing powerful Windows applications. If you now have the same opinion as we do, then our work is paid, and we will be glad if you could consider WPF as one option when you are designing your projects. Thank you!

More on Expression Blend and Windows Presentation Foundation

Microsoft Expression Blend is shipped with a detailed user guide. To read the user guide, open the **Help** menu, and then choose **User Guide**.

The website for Microsoft Expression Blend is

http://www.microsoft.com/Expression/products/overview.aspx?key=blend

For more Expression Blend trainings, go to

http://www.microsoft.com/Expression/kc/product.aspx?key=blend

Windows Presentation Foundation on MSDN:

http://msdn2.microsoft.com/en-us/library/ms754130.aspx