# Contents

# Getting Started (WPF)

1/23/2019 • 2 minutes to read • Edit Online

Windows Presentation Foundation (WPF) is a UI framework that creates desktop client applications. The WPF development platform supports a broad set of application development features, including an application model, resources, controls, graphics, layout, data binding, documents, and security. It is a subset of the .NET Framework, so if you have previously built applications with the .NET Framework using ASP.NET or Windows Forms, the programming experience should be familiar. WPF uses the Extensible Application Markup Language (XAML) to provide a declarative model for application programming. This section has topics that introduce and help you get started with WPF.

## Where Should I Start?

| | |
|---|---|
| I want to jump right in... | Walkthrough: My first WPF desktop application |
| How do I design the application UI? | Designing XAML in Visual Studio |
| New to .NET? | Overview of the .NET Framework<br>.NET Framework Application Essentials<br>Getting Started with Visual C# and Visual Basic |
| Tell me more about WPF... | Introduction to WPF in Visual Studio<br>XAML Overview (WPF)<br>Controls<br>Data Binding Overview |
| Are you a Windows Forms developer? | Windows Forms Controls and Equivalent WPF Controls<br>WPF and Windows Forms Interoperation |

## See also

- Class Library
- Application Development
- .NET Framework Developer Center

# Introduction to WPF in Visual Studio

5/4/2018 • 2 minutes to read • Edit Online

Windows Presentation Foundation (WPF) in Visual Studio provides developers with a unified programming model for building line-of-business desktop applications on Windows.

Create Desktop Applications with Windows Presentation Foundation

Designing XAML in Visual Studio and Blend for Visual Studio

Introduction to WPF

WPF in the .NET Framework

Get Visual Studio

# What's New in WPF Version 4.5

1/30/2019 • 6 minutes to read • Edit Online

This topic contains information about new and enhanced features in Windows Presentation Foundation (WPF) version 4.5.

This topic contains the following sections:



## Ribbon control

WPF 4.5 ships with a Ribbon control that hosts a Quick Access Toolbar, Application Menu, and tabs. For more information, see the Ribbon Overview.

## Improved performance when displaying large sets of grouped data

UI virtualization occurs when a subset of user interface (UI) elements are generated from a larger number of data items based on which items are visible on the screen. The VirtualizingPanel defines the IsVirtualizingWhenGrouping attached property that enables UI Virtualization for grouped data. For more information about grouping data, see How to: Sort and Group Data Using a View in XAML. For more information about virtualizing grouped data, see the IsVirtualizingWhenGrouping attached property.

## New features for the VirtualizingPanel

1. You can specify whether a VirtualizingPanel, such as the VirtualizingStackPanel, displays partial items by using the ScrollUnit attached property. If ScrollUnit is set to Item, the VirtualizingPanel will only display items that are completely visible. If ScrollUnit is set to Pixel, the VirtualizingPanel can display partially visible items.

2. You can specify the size of the cache before and after the viewport when the VirtualizingPanel is virtualizing by using the CacheLength attached property. The cache is the amount of space above or below the viewport in which items are not virtualized. Using a cache to avoid generating UI elements as they're scrolled into view can improve performance. The cache is populated at a lower priority so that the application does not become unresponsive during the operation. The VirtualizingPanel.CacheLengthUnit property determines the unit of measurement that is used by VirtualizingPanel.CacheLength.

## Binding to static properties

You can use static properties as the source of a data binding. The data binding engine recognizes when the property's value changes if a static event is raised. For example, if the class `SomeClass` defines a static property called `MyProperty`, `SomeClass` can define a static event that is raised when the value of `MyProperty` changes. The static event can use either of the following signatures.

- `public static event EventHandler MyPropertyChanged;`
- `public static event EventHandler<PropertyChangedEventArgs> StaticPropertyChanged;`

Note that in the first case, the class exposes a static event named *PropertyName* `Changed` that passes EventArgs to the event handler. In the second case, the class exposes a static event named `StaticPropertyChanged` that passes PropertyChangedEventArgs to the event handler. A class that implements the static property can choose to raise property-change notifications using either method.

## Accessing collections on non-UI Threads

WPF enables you to access and modify data collections on threads other than the one that created the collection. This enables you to use a background thread to receive data from an external source, such as a database, and display the data on the UI thread. By using another thread to modify the collection, your user interface remains responsive to user interaction.

## Synchronously and Asynchronously validating data

The INotifyDataErrorInfo interface enables data entity classes to implement custom validation rules and expose validation results asynchronously. This interface also supports custom error objects, multiple errors per property, cross-property errors, and entity-level errors. For more information, see INotifyDataErrorInfo.

## Automatically updating the source of a data binding

If you use a data binding to update a data source, you can use the Delay property to specify an amount of time to pass after the property changes on the target before the source updates. For example, suppose that you have a Slider that has its Value property data two-way bound to a property of a data object and the UpdateSourceTrigger property is set to PropertyChanged. In this example, when the user moves the Slider, the source updates for each pixel that the Slider moves. The source object typically needs the value of the slider only when the slider's Value stops changing. To prevent updating the source too often, use Delay to specify that the source should not be updated until a certain amount of time elapses after the thumb stops moving.

## Binding to types that Implement ICustomTypeProvider

WPF supports data binding to objects that implement ICustomTypeProvider, also known as custom types. You can use custom types in the following cases.

1. As a PropertyPath in a data binding. For example, the Path property of a Binding can reference a property of a custom type.
2. As the value of the DataType property.

3. As a type that determines the automatically generated columns in a DataGrid.

## Retrieving data binding information from a binding expression

In certain cases, you might get the BindingExpression of a Binding and need information about the source and target objects of the binding. New APIs have been added to enable you to get the source or target object or the associated property. When you have a BindingExpression, use the following APIs to get information about the target and source.

| TO FIND THIS VALUE OF THE BINDING | USE THIS API |
|---|---|
| The target object | BindingExpressionBase.Target |
| The target property | BindingExpressionBase.TargetProperty |
| The source object | BindingExpression.ResolvedSource |
| The source property | BindingExpression.ResolvedSourcePropertyName |
| Whether the BindingExpression belongs to a BindingGroup | BindingExpressionBase.BindingGroup |
| The owner of a BindingGroup | Owner |

## Checking for a valid DataContext object

There are cases where the DataContext of an item container in an ItemsControl becomes disconnected. An item container is the UI element that displays an item in an ItemsControl. When an ItemsControl is data bound to a collection, an item container is generated for each item. In some cases, item containers are removed from the visual tree. Two typical cases where an item container is removed are when an item is removed from the underlying collection and when virtualization is enabled on the ItemsControl. In these cases, the DataContext property of the item container will be set to the sentinel object that is returned by the BindingOperations.DisconnectedSource static property. You should check whether the DataContext is equal to the DisconnectedSource object before accessing the DataContext of an item container.

## Repositioning data as the data's values change (Live shaping)

A collection of data can be grouped, sorted, or filtered. WPF 4.5 enables the data to be rearranged when the data is modified. For example, suppose that an application uses a DataGrid to list stocks in a stock market and the stocks are sorted by stock value. If live sorting is enabled on the stocks' CollectionView, a stock's position in the DataGrid moves when the value of the stock becomes greater or less than another stock's value. For more information, see the ICollectionViewLiveShaping interface.

## Improved Support for Establishing a Weak Reference to an Event

Implementing the weak event pattern is now easier because subscribers to events can participate in it without implementing an extra interface. The generic WeakEventManager class also enables subscribers to participate in the weak event pattern if a dedicated WeakEventManager does not exist for a certain event. For more information, see Weak Event Patterns.

## New methods for the Dispatcher class

The Dispatcher class defines new methods for synchronous and asynchronous operations. The synchronous Invoke method defines overloads that take an Action or Func<TResult> parameter. The new asynchronous method,

InvokeAsync, also takes an Action or Func<TResult> as the callback parameter and returns a DispatcherOperation or DispatcherOperation<TResult>. The DispatcherOperation and DispatcherOperation<TResult> classes define a Task property. When you call InvokeAsync, you can use the `await` keyword with either the DispatcherOperation or the associated Task. If you need to wait synchronously for the Task that is returned by a DispatcherOperation or DispatcherOperation<TResult>, call the DispatcherOperationWait extension method. Calling Task.Wait will result in a deadlock if the operation is queued on a calling thread. For more information about using a Task to perform asynchronous operations, see Task Parallelism (Task Parallel Library).

## Markup Extensions for Events

WPF 4.5 supports markup extensions for events. While WPF does not define a markup extension to be used for events, third parties are able to create a markup extension that can be used with events.

## See also

- What's New in the .NET Framework

# Walkthrough: My first WPF desktop application

11/20/2018 • 18 minutes to read • Edit Online

This article shows you how to develop a simple Windows Presentation Foundation (WPF) application that includes the elements that are common to most WPF applications: Extensible Application Markup Language (XAML) markup, code-behind, application definitions, controls, layout, data binding, and styles.

This walkthrough includes the following steps:

- Use XAML to design the appearance of the application's user interface (UI).
- Write code to build the application's behavior.
- Create an application definition to manage the application.
- Add controls and create the layout to compose the application UI.
- Create styles for a consistent appearance throughout an application's UI.
- Bind the UI to data to both populate the UI from data and keep the data and UI synchronized.

By the end of the walkthrough, you'll have built a standalone Windows application that allows users to view expense reports for selected people. The application is composed of several WPF pages that are hosted in a browser-style window.

> **TIP**
> The sample code that is used to build this walkthrough is available for both Visual Basic and C# at Introduction to Building WPF Applications.

## Prerequisites

- Visual Studio 2017 or later

  For more information about installing the latest version of Visual Studio, see Install Visual Studio.

## Create the application project

The first step is to create the application infrastructure, which includes an application definition, two pages, and an image.

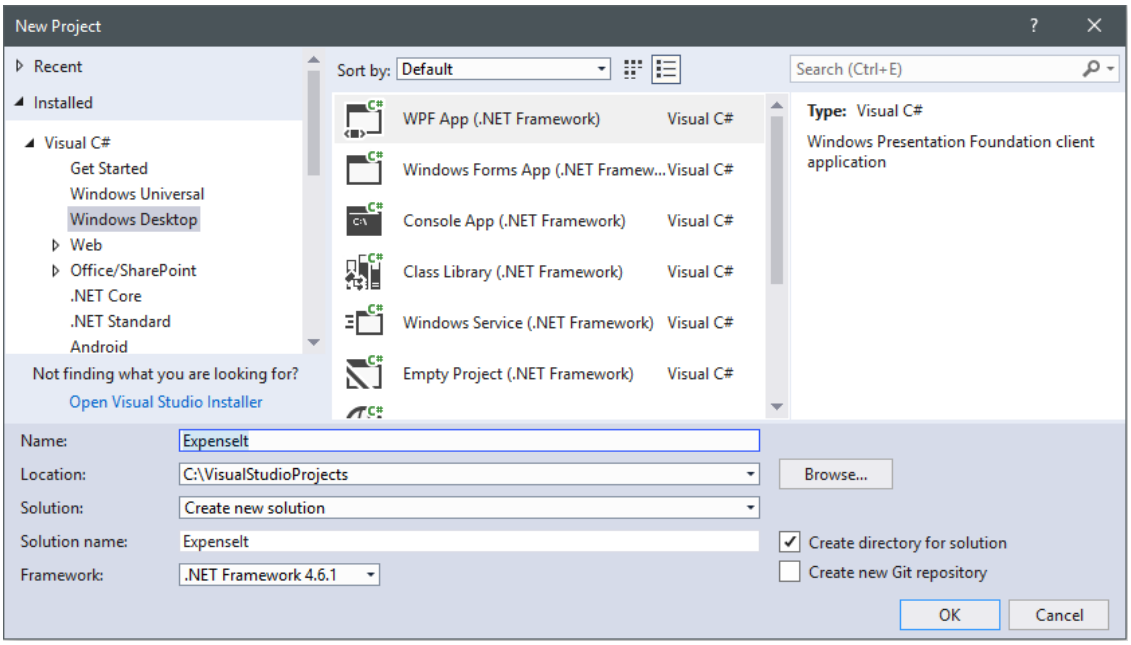
1. Create a new WPF Application project in Visual Basic or Visual C# named `ExpenseIt`:

   a. Open Visual Studio and select **File** > **New** > **Project**.

      The **New Project** dialog opens.

   b. Under the **Installed** category, expand either the **Visual C#** or **Visual Basic** node, and then select **Windows Desktop**.

   c. Select the **WPF App (.NET Framework)** template. Enter the name `ExpenseIt` and then select **OK**.




Visual Studio creates the project and opens the designer for the default application window named **MainWindow.xaml**.

> **NOTE**
>
> This walkthrough uses the DataGrid control that is available in the .NET Framework 4 and later. Be sure that your project targets the .NET Framework 4 or later. For more information, see How to: Target a Version of the .NET Framework.

2. Open *Application.xaml* (Visual Basic) or *App.xaml* (C#).

   This XAML file defines a WPF application and any application resources. You also use this file to specify the UI that automatically shows when the application starts; in this case, *MainWindow.xaml*.

   Your XAML should look like this in Visual Basic:

```
<Application x:Class="Application"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

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

   Or like this in C#:

```
<Application x:Class="ExpenseIt.App"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     StartupUri="MainWindow.xaml">
    <Application.Resources>

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

3. Open *MainWindow.xaml*.

   This XAML file is the main window of your application and displays content created in pages. The Window

class defines the properties of a window, such as its title, size, or icon, and handles events, such as closing or hiding.

4. Change the Window element to a NavigationWindow, as shown in the following XAML:

```
<NavigationWindow x:Class="ExpenseIt.MainWindow"
     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
     ...
</NavigationWindow>
```

   This app navigates to different content depending on the user input. This is why the main Window needs to be changed to a NavigationWindow. NavigationWindow inherits all the properties of Window. The NavigationWindow element in the XAML file creates an instance of the NavigationWindow class. For more information, see Navigation overview.

5. Change the following properties on the NavigationWindow element:

   - Set the Title property to "`ExpenseIt`".
   - Set the Width property to 500 pixels.
   - Set the Height property to 350 pixels.
   - Remove the Grid elements between the NavigationWindow tags.

   Your XAML should look like this in Visual Basic:

```
<NavigationWindow x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ExpenseIt" Height="350" Width="500">

</NavigationWindow>
```

   Or like this in C#:

```
<NavigationWindow x:Class="ExpenseIt.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ExpenseIt" Height="350" Width="500">

</NavigationWindow>
```

6. Open *MainWindow.xaml.vb* or *MainWindow.xaml.cs*.

   This file is a code-behind file that contains code to handle the events declared in *MainWindow.xaml*. This file contains a partial class for the window defined in XAML.

7. If you are using C#, change the `MainWindow` class to derive from NavigationWindow. (In Visual Basic, this happens automatically when you change the window in XAML.)

   Your code should look like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace ExpenseIt
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : NavigationWindow
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

```
Class MainWindow

End Class
```

**TIP**

You can toggle the code language of the sample code between C# and Visual Basic in the **Language** drop-down on the upper right side of this article.

## Add files to the application

In this section, you'll add two pages and an image to the application.

1. Add a new WPF page to the project, and name it `ExpenseItHome.xaml`:

   a. In **Solution Explorer**, right-click on the `ExpenseIt` project node and choose **Add** > **Page**.

   b. In the **Add New Item** dialog, the **Page (WPF)** template is already selected. Enter the name `ExpenseItHome`, and then select **Add**.

   This page is the first page that's displayed when the application is launched. It will show a list of people to select from, to show an expense report for.

2. Open `ExpenseItHome.xaml`.

3. Set the Title to "`ExpenseIt - Home`".

   Your XAML should look like this in Visual Basic:

```
<Page x:Class="ExpenseItHome"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  mc:Ignorable="d"
  d:DesignHeight="300" d:DesignWidth="300"
  Title="ExpenseIt - Home">
    <Grid>

    </Grid>
</Page>
```

Or this in C#:

```
<Page x:Class="ExpenseIt.ExpenseItHome"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
    Title="ExpenseIt - Home">

    <Grid>

    </Grid>
</Page>
```

4. Open *MainWindow.xaml*.
5. Set the Source property on the NavigationWindow to "`ExpenseItHome.xaml`".

   This sets `*ExpenseItHome.xaml*` to be the first page opened when the application starts. Your XAML should look like this in Visual Basic:

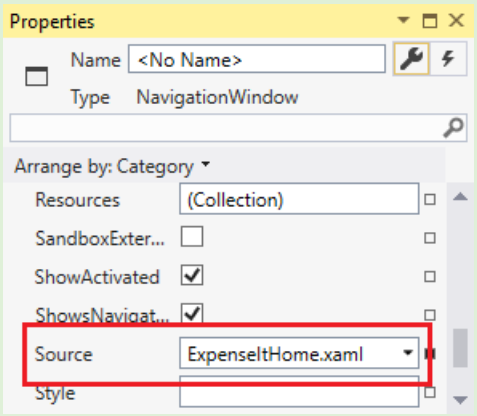
```
<NavigationWindow x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ExpenseIt" Height="350" Width="500" Source="ExpenseItHome.xaml">

</NavigationWindow>
```

   Or this in C#:

```
<NavigationWindow x:Class="ExpenseIt.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ExpenseIt" Height="350" Width="500" Source="ExpenseItHome.xaml">

</NavigationWindow>
```

**TIP**

You can also set the **Source** property in the **Miscellaneous** category of the **Properties** window.




6. Add another new WPF page to the project, and name it *ExpenseReportPage.xaml*::

   a. In **Solution Explorer**, right-click on the `ExpenseIt` project node and choose **Add** > **Page**.

   b. In the **Add New Item** dialog, the **Page (WPF)** template is already selected. Enter the name **ExpenseReportPage**, and then select **Add**.

   This page will show the expense report for the person that is selected on the `ExpenseItHome` page.

7. Open *ExpenseReportPage.xaml*.

8. Set the Title to "`ExpenseIt - View Expense`".

   Your XAML should look like this in Visual Basic:

```
<Page x:Class="ExpenseReportPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
      Title="ExpenseIt - View Expense">
    <Grid>

    </Grid>
</Page>
```

   Or this in C#:

```
<Page x:Class="ExpenseIt.ExpenseReportPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
    Title="ExpenseIt - View Expense">

    <Grid>

    </Grid>
</Page>
```

9. Open *ExpenseItHome.xaml.vb* and *ExpenseReportPage.xaml.vb*, or *ExpenseItHome.xaml.cs* and *ExpenseReportPage.xaml.cs*.

   When you create a new Page file, Visual Studio automatically creates a *code-behind* file. These code-behind files handle the logic for responding to user input.

   Your code should look like this for `ExpenseItHome`:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace ExpenseIt
{
    /// <summary>
    /// Interaction logic for ExpenseItHome.xaml
    /// </summary>
    public partial class ExpenseItHome : Page
    {
        public ExpenseItHome()
        {
            InitializeComponent();
        }
    }
}
```

```
Class ExpenseItHome

End Class
```

   And like this for **ExpenseReportPage**:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace ExpenseIt
{
    /// <summary>
    /// Interaction logic for ExpenseReportPage.xaml
    /// </summary>
    public partial class ExpenseReportPage : Page
    {
        public ExpenseReportPage()
        {
            InitializeComponent();
        }
    }
}
```

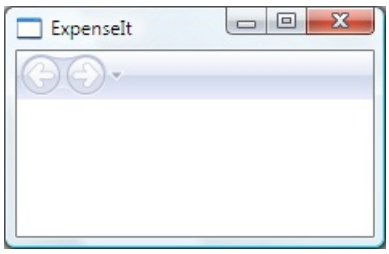
```
Class ExpenseReportPage

End Class
```

10. Add an image named *watermark.png* to the project. You can create your own image, copy the file from the sample code, or get it here.
11. Right-click on the project node and select **Add** > **Existing Item**, or press **Shift**+**Alt**+**A**.
12. In the **Add Existing Item** dialog, browse to the image file you want to use, and then select **Add**.

## Build and run the application

1. To build and run the application, press **F5** or select **Start Debugging** from the **Debug** menu.

   The following illustration shows the application with the NavigationWindow buttons:

2. Close the application to return to Visual Studio.

## Create the layout

Layout provides an ordered way to place UI elements, and also manages the size and position of those elements when a UI is resized. You typically create a layout with one of the following layout controls:

- Canvas

- DockPanel
- Grid
- StackPanel
- VirtualizingStackPanel
- WrapPanel

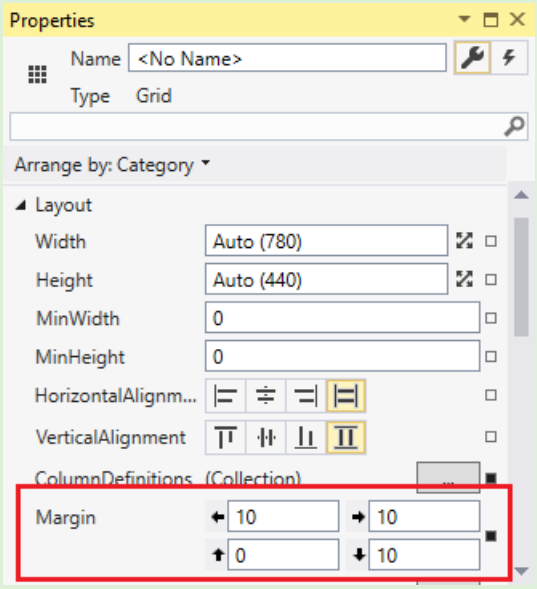
Each of these layout controls supports a special type of layout for its child elements. `ExpenseIt` pages can be resized, and each page has elements that are arranged horizontally and vertically alongside other elements. Consequently, the Grid is the ideal layout element for the application.

> **TIP**
> For more information about Panel elements, see Panels overview. For more information about layout, see Layout.

In the section, you create a single-column table with three rows and a 10-pixel margin by adding column and row definitions to the Grid in `ExpenseItHome.xaml`.

1. Open `ExpenseItHome.xaml`.
2. Set the Margin property on the Grid element to "10,0,10,10", which corresponds to left, top, right and bottom margins:

```
<Grid Margin="10,0,10,10">
```

> **TIP**
> You can also set the **Margin** values in the **Properties** window, under the **Layout** category:
>
> 
> 

3. Add the following XAML between the Grid tags to create the row and column definitions:

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

The Height of two rows is set to Auto, which means that the rows are sized based on the content in the rows. The default Height is Star sizing, which means that the row height is a weighted proportion of the available space. For example if two rows each have a Height of "*", they each have a height that is half of the available space.

Your Grid should now look like the following XAML:

```
<Grid Margin="10,0,10,10">
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition />
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
</Grid>
```

## Add controls

In this section, you'll update the home page UI to show a list of people that a user can select from to show the expense report for. Controls are UI objects that allow users to interact with your application. For more information, see Controls.

To create this UI, you'll add the following elements to `ExpenseItHome.xaml`:

- ListBox (for the list of people).
- Label (for the list header).
- Button (to click to view the expense report for the person that is selected in the list).

Each control is placed in a row of the Grid by setting the Grid.Row attached property. For more information about attached properties, see Attached Properties Overview.

1. Open `ExpenseItHome.xaml`.
2. Add the following XAML somewhere between the Grid tags:

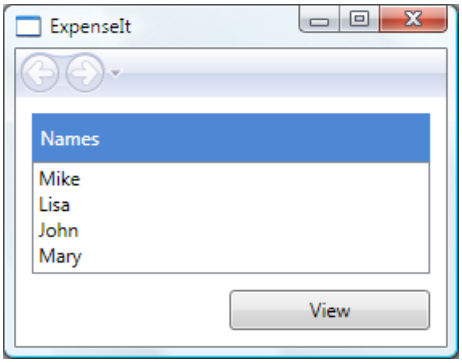
```
  <!-- People list -->
  <Border Grid.Column="0" Grid.Row="0" Height="35" Padding="5" Background="#4E87D4">
      <Label VerticalAlignment="Center" Foreground="White">Names</Label>
  </Border>
  <ListBox Name="peopleListBox" Grid.Column="0" Grid.Row="1">
      <ListBoxItem>Mike</ListBoxItem>
      <ListBoxItem>Lisa</ListBoxItem>
      <ListBoxItem>John</ListBoxItem>
      <ListBoxItem>Mary</ListBoxItem>
  </ListBox>

  <!-- View report button -->
  <Button Grid.Column="0" Grid.Row="2" Margin="0,10,0,0" Width="125"
Height="25" HorizontalAlignment="Right">View</Button>
```

**TIP**

You can also create the controls by dragging them from the **Toolbox** window onto the design window, and then setting their properties in the **Properties** window.

3. Build and run the application.

The following illustration shows the controls you just created:




## Add an image and a title

In this section, you'll update the home page UI with an image and a page title.

1. Open `ExpenseItHome.xaml`.

2. Add another column to the ColumnDefinitions with a fixed Width of 230 pixels:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="230" />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
```

3. Add another row to the RowDefinitions, for a total of four rows:

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

4. Move the controls to the second column by setting the Grid.Column property to 1 in each of the three controls (Border, ListBox, and Button).

5. Move each control down a row, by incrementing its Grid.Row value by 1.

   The XAML for the three controls now looks like this:

```
  <Border Grid.Column="1" Grid.Row="1" Height="35" Padding="5" Background="#4E87D4">
      <Label VerticalAlignment="Center" Foreground="White">Names</Label>
  </Border>
  <ListBox Name="peopleListBox" Grid.Column="1" Grid.Row="2">
      <ListBoxItem>Mike</ListBoxItem>
      <ListBoxItem>Lisa</ListBoxItem>
      <ListBoxItem>John</ListBoxItem>
      <ListBoxItem>Mary</ListBoxItem>
  </ListBox>

  <!-- View report button -->
  <Button Grid.Column="1" Grid.Row="3" Margin="0,10,0,0" Width="125"
Height="25" HorizontalAlignment="Right">View</Button>
```

6. Set the Background of the Grid to be the *watermark.png* image file, by adding the following XAML somewhere between the `<Grid>` and `</Grid>` tags:

```
<Grid.Background>
    <ImageBrush ImageSource="watermark.png"/>
</Grid.Background>
```

7. Before the Border element, add a Label with the content "View Expense Report". This is the title of the page.

```
<Label Grid.Column="1" VerticalAlignment="Center" FontFamily="Trebuchet MS"
        FontWeight="Bold" FontSize="18" Foreground="#0066cc">
    View Expense Report
</Label>
```

8. Build and run the application.

The following illustration shows the results of what you just added:


## Add code to handle events

1. Open `ExpenseItHome.xaml`.
2. Add a Click event handler to the Button element. For more information, see How to: Create a simple event handler.

```
  <!-- View report button -->
  <Button Grid.Column="1" Grid.Row="3" Margin="0,10,0,0" Width="125"
Height="25" HorizontalAlignment="Right" Click="Button_Click">View</Button>
```

3. Open `ExpenseItHome.xaml.vb` or `ExpenseItHome.xaml.cs`.
4. Add the following code to the `ExpenseItHome` class to add a button click event handler. The event handler opens the **ExpenseReportPage** page.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // View Expense Report
    ExpenseReportPage expenseReportPage = new ExpenseReportPage();
    this.NavigationService.Navigate(expenseReportPage);

}
```

```
Private Sub Button_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' View Expense Report
    Dim expenseReportPage As New ExpenseReportPage()
    Me.NavigationService.Navigate(expenseReportPage)

End Sub
```

## Create the UI for ExpenseReportPage

*ExpenseReportPage.xaml* displays the expense report for the person that's selected on the `ExpenseItHome` page. In this section, you'll create the UI for **ExpenseReportPage**. You'll also add background and fill colors to the various UI elements.

1. Open *ExpenseReportPage.xaml*.
2. Add the following XAML between the Grid tags:

```
<Grid.Background>
    <ImageBrush ImageSource="watermark.png" />
</Grid.Background>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="230" />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
</Grid.RowDefinitions>


<Label Grid.Column="1" VerticalAlignment="Center" FontFamily="Trebuchet MS"
FontWeight="Bold" FontSize="18" Foreground="#0066cc">
    Expense Report For:
</Label>
<Grid Margin="10" Grid.Column="1" Grid.Row="1">

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

    <!-- Name -->
    <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" Orientation="Horizontal">
        <Label Margin="0,0,0,5" FontWeight="Bold">Name:</Label>
        <Label Margin="0,0,0,5" FontWeight="Bold"></Label>
    </StackPanel>

    <!-- Department -->
    <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Orientation="Horizontal">
        <Label Margin="0,0,0,5" FontWeight="Bold">Department:</Label>
        <Label Margin="0,0,0,5" FontWeight="Bold"></Label>
    </StackPanel>

    <Grid Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="2" VerticalAlignment="Top"
          HorizontalAlignment="Left">
        <!-- Expense type and Amount table -->
        <DataGrid  AutoGenerateColumns="False" RowHeaderWidth="0" >
            <DataGrid.ColumnHeaderStyle>
                <Style TargetType="{x:Type DataGridColumnHeader}">
                    <Setter Property="Height" Value="35" />
                    <Setter Property="Padding" Value="5" />
                    <Setter Property="Background" Value="#4E87D4" />
                    <Setter Property="Foreground" Value="White" />
                </Style>
            </DataGrid.ColumnHeaderStyle>
            <DataGrid.Columns>
                <DataGridTextColumn Header="ExpenseType" />
                <DataGridTextColumn Header="Amount"  />
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Grid>
```

This UI is similar to `ExpenseItHome.xaml`, except the report data is displayed in a DataGrid.

3. Build and run the application.

> **NOTE**
>
> If you get an error that the DataGrid was not found or does not exist, make sure that your project targets the .NET Framework 4 or later. For more information, see How to: Target a Version of the .NET Framework.

4. Select the **View** button.

   The expense report page appears. Also notice that the back navigation button is enabled.

The following illustration shows the UI elements added to *ExpenseReportPage.xaml*.


## Style controls

The appearance of various elements is often the same for all elements of the same type in a UI. UI uses styles to make appearances reusable across multiple elements. The reusability of styles helps to simplify XAML creation and management. This section replaces the per-element attributes that were defined in previous steps with styles.

1. Open *Application.xaml* or *App.xaml*.
2. Add the following XAML between the Application.Resources tags:

```
<!-- Header text style -->
<Style x:Key="headerTextStyle">
    <Setter Property="Label.VerticalAlignment" Value="Center"></Setter>
    <Setter Property="Label.FontFamily" Value="Trebuchet MS"></Setter>
    <Setter Property="Label.FontWeight" Value="Bold"></Setter>
    <Setter Property="Label.FontSize" Value="18"></Setter>
    <Setter Property="Label.Foreground" Value="#0066cc"></Setter>
</Style>

<!-- Label style -->
<Style x:Key="labelStyle" TargetType="{x:Type Label}">
    <Setter Property="VerticalAlignment" Value="Top" />
    <Setter Property="HorizontalAlignment" Value="Left" />
    <Setter Property="FontWeight" Value="Bold" />
    <Setter Property="Margin" Value="0,0,0,5" />
</Style>

<!-- DataGrid header style -->
<Style x:Key="columnHeaderStyle" TargetType="{x:Type DataGridColumnHeader}">
    <Setter Property="Height" Value="35" />
    <Setter Property="Padding" Value="5" />
    <Setter Property="Background" Value="#4E87D4" />
    <Setter Property="Foreground" Value="White" />
</Style>

<!-- List header style -->
<Style x:Key="listHeaderStyle" TargetType="{x:Type Border}">
    <Setter Property="Height" Value="35" />
    <Setter Property="Padding" Value="5" />
    <Setter Property="Background" Value="#4E87D4" />
</Style>

<!-- List header text style -->
<Style x:Key="listHeaderTextStyle" TargetType="{x:Type Label}">
    <Setter Property="Foreground" Value="White" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="HorizontalAlignment" Value="Left" />
</Style>

<!-- Button style -->
<Style x:Key="buttonStyle" TargetType="{x:Type Button}">
    <Setter Property="Width" Value="125" />
    <Setter Property="Height" Value="25" />
    <Setter Property="Margin" Value="0,10,0,0" />
    <Setter Property="HorizontalAlignment" Value="Right" />
</Style>
```

This XAML adds the following styles:

- `headerTextStyle`: To format the page title Label.
- `labelStyle`: To format the Label controls.
- `columnHeaderStyle`: To format the DataGridColumnHeader.
- `listHeaderStyle`: To format the list header Border controls.
- `listHeaderTextStyle`: To format the list header Label.
- `buttonStyle`: To format the Button on `ExpenseItHome.xaml`.

Notice that the styles are resources and children of the Application.Resources property element. In this location, the styles are applied to all the elements in an application. For an example of using resources in a .NET Framework application, see Use Application Resources.

3. Open `ExpenseItHome.xaml`.

4. Replace everything between the Grid elements with the following XAML:

```
    <Grid.Background>
        <ImageBrush ImageSource="watermark.png"  />
    </Grid.Background>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="230" />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

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

    <!-- People list -->

    <Label Grid.Column="1" Style="{StaticResource headerTextStyle}" >
        View Expense Report
    </Label>

    <Border Grid.Column="1" Grid.Row="1" Style="{StaticResource listHeaderStyle}">
        <Label Style="{StaticResource listHeaderTextStyle}">Names</Label>
    </Border>
    <ListBox Name="peopleListBox" Grid.Column="1" Grid.Row="2">
        <ListBoxItem>Mike</ListBoxItem>
        <ListBoxItem>Lisa</ListBoxItem>
        <ListBoxItem>John</ListBoxItem>
        <ListBoxItem>Mary</ListBoxItem>
    </ListBox>

    <!-- View report button -->
    <Button Grid.Column="1" Grid.Row="3" Click="Button_Click" Style="{StaticResource
buttonStyle}">View</Button>
```

   The properties such as VerticalAlignment and FontFamily that define the look of each control are removed and replaced by applying the styles. For example, the `headerTextStyle` is applied to the "View Expense Report" Label.

5. Open *ExpenseReportPage.xaml*.

6. Replace everything between the Grid elements with the following XAML:

```
<Grid.Background>
    <ImageBrush ImageSource="watermark.png" />
</Grid.Background>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="230" />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
</Grid.RowDefinitions>


<Label Grid.Column="1" Style="{StaticResource headerTextStyle}">
    Expense Report For:
</Label>
<Grid Margin="10" Grid.Column="1" Grid.Row="1">

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

    <!-- Name -->
    <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" Orientation="Horizontal">
        <Label Style="{StaticResource labelStyle}">Name:</Label>
        <Label Style="{StaticResource labelStyle}"></Label>
    </StackPanel>

    <!-- Department -->
    <StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1"
Orientation="Horizontal">
        <Label Style="{StaticResource labelStyle}">Department:</Label>
        <Label Style="{StaticResource labelStyle}"></Label>
    </StackPanel>

    <Grid Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="2" VerticalAlignment="Top"
          HorizontalAlignment="Left">
        <!-- Expense type and Amount table -->
        <DataGrid ColumnHeaderStyle="{StaticResource columnHeaderStyle}"
                  AutoGenerateColumns="False" RowHeaderWidth="0" >
            <DataGrid.Columns>
                <DataGridTextColumn Header="ExpenseType" />
                <DataGridTextColumn Header="Amount"  />
            </DataGrid.Columns>
        </DataGrid>
    </Grid>
</Grid>
```

This adds styles to the Label and Border elements.

## Bind data to a control

In this section, you'll create the XML data that is bound to various controls.

1. Open `ExpenseItHome.xaml`.
2. After the opening Grid element, add the following XAML to create an XmlDataProvider that contains the data for each person:

```
<Grid.Resources>
```

```
<!-- Expense Report Data -->
<XmlDataProvider x:Key="ExpenseDataSource" XPath="Expenses">
    <x:XData>
        <Expenses xmlns="">
            <Person Name="Mike" Department="Legal">
                <Expense ExpenseType="Lunch" ExpenseAmount="50" />
                <Expense ExpenseType="Transportation" ExpenseAmount="50" />
            </Person>
            <Person Name="Lisa" Department="Marketing">
                <Expense ExpenseType="Document printing"
      ExpenseAmount="50"/>
                <Expense ExpenseType="Gift" ExpenseAmount="125" />
            </Person>
            <Person Name="John" Department="Engineering">
                <Expense ExpenseType="Magazine subscription"
     ExpenseAmount="50"/>
                <Expense ExpenseType="New machine" ExpenseAmount="600" />
                <Expense ExpenseType="Software" ExpenseAmount="500" />
            </Person>
            <Person Name="Mary" Department="Finance">
                <Expense ExpenseType="Dinner" ExpenseAmount="100" />
            </Person>
        </Expenses>
    </x:XData>
</XmlDataProvider>
```

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

The data is created as a Grid resource. Normally this would be loaded as a file, but for simplicity the data is added inline.

3. Within the `<Grid.Resources>` element, add the following DataTemplate, which defines how to display the data in the ListBox:

```
<Grid.Resources>
```

```
<!-- Name item template -->
<DataTemplate x:Key="nameItemTemplate">
    <Label Content="{Binding XPath=@Name}"/>
</DataTemplate>
```

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

For more information about data templates, see Data templating overview.

4. Replace the existing ListBox with the following XAML:

```
<ListBox Name="peopleListBox" Grid.Column="1" Grid.Row="2"
         ItemsSource="{Binding Source={StaticResource ExpenseDataSource}, XPath=Person}"
         ItemTemplate="{StaticResource nameItemTemplate}">
</ListBox>
```

This XAML binds the ItemsSource property of the ListBox to the data source and applies the data template

as the ItemTemplate.

## Connect data to controls

Next, you'll add code to retrieve the name that's selected on the `ExpenseItHome` page and pass it to the constructor of **ExpenseReportPage**. **ExpenseReportPage** sets its data context with the passed item, which is what the controls defined in *ExpenseReportPage.xaml* bind to.

1. Open *ExpenseReportPage.xaml.vb* or *ExpenseReportPage.xaml.cs*.
2. Add a constructor that takes an object so you can pass the expense report data of the selected person.

```
public partial class ExpenseReportPage : Page
{
    public ExpenseReportPage()
    {
        InitializeComponent();
    }

    // Custom constructor to pass expense report data
    public ExpenseReportPage(object data):this()
    {
        // Bind to expense report data.
        this.DataContext = data;
    }

}
```

```
Partial Public Class ExpenseReportPage
    Inherits Page
    Public Sub New()
        InitializeComponent()
    End Sub

    ' Custom constructor to pass expense report data
    Public Sub New(ByVal data As Object)
        Me.New()
        ' Bind to expense report data.
        Me.DataContext = data
    End Sub

End Class
```

3. Open `ExpenseItHome.xaml.vb` or `ExpenseItHome.xaml.cs`.
4. Change the Click event handler to call the new constructor passing the expense report data of the selected person.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // View Expense Report
    ExpenseReportPage expenseReportPage = new ExpenseReportPage(this.peopleListBox.SelectedItem);
    this.NavigationService.Navigate(expenseReportPage);

}
```

```
Private Sub Button_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' View Expense Report
    Dim expenseReportPage As New ExpenseReportPage(Me.peopleListBox.SelectedItem)
    Me.NavigationService.Navigate(expenseReportPage)

End Sub
```

## Style data with data templates

In this section, you'll update the UI for each item in the data-bound lists by using data templates.

1. Open *ExpenseReportPage.xaml*.
2. Bind the content of the "Name" and "Department" Label elements to the appropriate data source property. For more information about data binding, see Data binding overview.

```
<!-- Name -->
<StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="0" Orientation="Horizontal">
    <Label Style="{StaticResource labelStyle}">Name:</Label>
    <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=@Name}"></Label>
</StackPanel>

<!-- Department -->
<StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="1" Orientation="Horizontal">
    <Label Style="{StaticResource labelStyle}">Department:</Label>
    <Label Style="{StaticResource labelStyle}" Content="{Binding XPath=@Department}"></Label>
</StackPanel>
```

3. After the opening Grid element, add the following data templates, which define how to display the expense report data:

```
<!--Templates to display expense report data-->
<Grid.Resources>
    <!-- Reason item template -->
    <DataTemplate x:Key="typeItemTemplate">
        <Label Content="{Binding XPath=@ExpenseType}"/>
    </DataTemplate>
    <!-- Amount item template -->
    <DataTemplate x:Key="amountItemTemplate">
        <Label Content="{Binding XPath=@ExpenseAmount}"/>
    </DataTemplate>
</Grid.Resources>
```

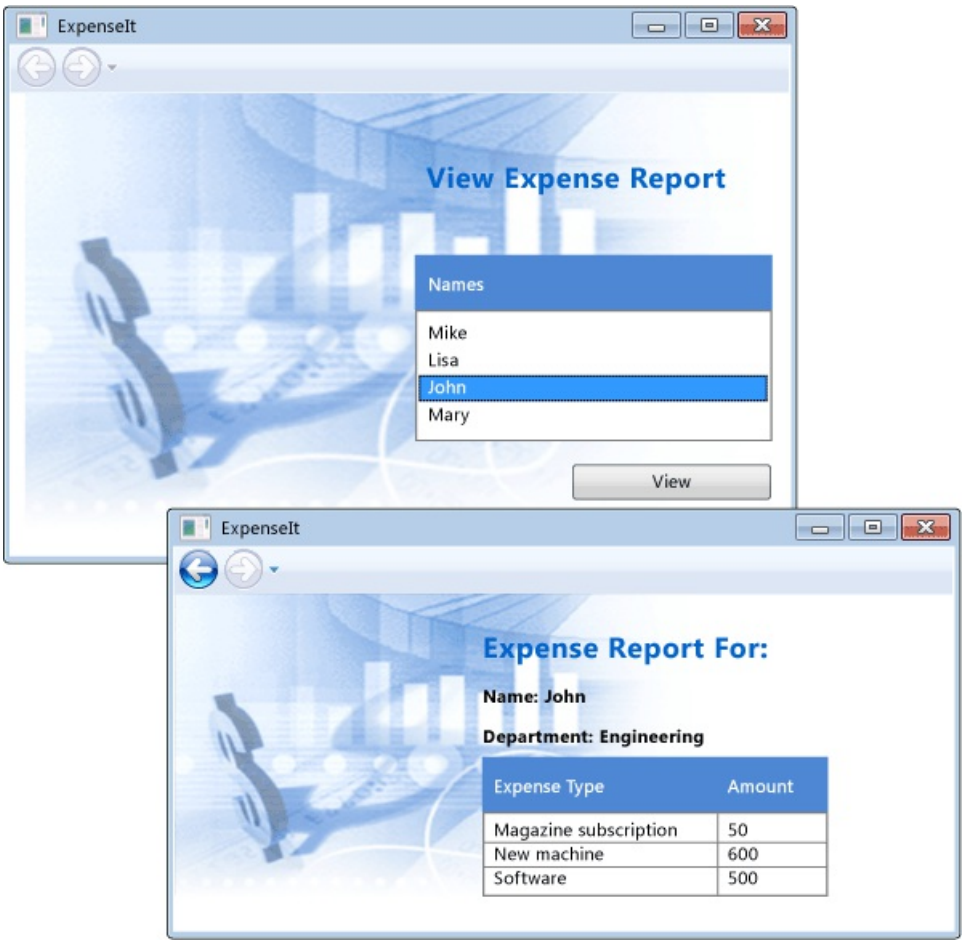
4. Replace the DataGridTextColumn elements with DataGridTemplateColumn under the DataGrid element and apply the templates to them.

```
<!-- Expense type and Amount table -->
<DataGrid ItemsSource="{Binding XPath=Expense}" ColumnHeaderStyle="{StaticResource columnHeaderStyle}"
AutoGenerateColumns="False" RowHeaderWidth="0" >

    <DataGrid.Columns>
        <DataGridTemplateColumn Header="ExpenseType" CellTemplate="{StaticResource typeItemTemplate}" />
        <DataGridTemplateColumn Header="Amount" CellTemplate="{StaticResource amountItemTemplate}" />
    </DataGrid.Columns>

</DataGrid>
```

5. Build and run the application.

6. Select a person and then select the **View** button.

The following illustration shows both pages of the `ExpenseIt` application with controls, layout, styles, data binding, and data templates applied:




> **NOTE**
>
> This sample demonstrates a specific feature of WPF and doesn't follow all best practices for things like security, localization, and accessibility. For comprehensive coverage of WPF and the .NET Framework application development best practices, see the following topics:
>
> - Accessibility
> - Security
> - WPF globalization and localization
> - WPF performance

## Next steps

In this walkthrough you learned a number of techniques for creating a UI using Windows Presentation Foundation (WPF). You should now have a basic understanding of the building blocks of a data-bound, .NET Framework application. For more information about the WPF architecture and programming models, see the following topics:

- WPF architecture
- XAML overview (WPF)
- Dependency properties overview
- Layout

For more information about creating applications, see the following topics:

- Application development
- Controls
- Data binding overview
- Graphics and multimedia
- Documents in WPF

## See also

- Panels overview
- Data templating overview
- Build a WPF application
- Styles and templates

# WPF Walkthroughs

8/31/2018 • 2 minutes to read • Edit Online

Walkthroughs give step-by-step instructions for common scenarios. This makes them a good place to start learning about the product or a particular feature area.

This topic contains links to Windows Presentation Foundation (WPF) walkthroughs.

## WPF Designer Walkthroughs

| TITLE | DESCRIPTION |
| --- | --- |
| Walkthrough: Building a Simple WPF Application with the WPF Designer | Demonstrates how to build a simple WPF application with the WPF Designer. |
| Walkthrough: Constructing a Dynamic Layout | Demonstrates creating a dynamic layout by using a Grid panel control. |
| Walkthrough: Creating a Resizable Application by Using the WPF Designer | Demonstrates how to create window layouts that are resizable by the user at run time. |
| Walkthrough: Creating a Data Binding by Using the WPF Designer | Demonstrates how to use the WPF Designer to create data bindings that connect data to a control. |
| Walkthrough: Using a DesignInstance to Bind to Data in the Designer | Demonstrates how to use the WPF Designer to create data bindings at design time for a data context that is assigned at run time. |

## WPF Walkthroughs

| TITLE | DESCRIPTION |
| --- | --- |
| Walkthrough: My first WPF desktop application | Demonstrates creating a WPF application using many of the common features of WPF including controls, layout and data binding. |
| Create a Button by Using XAML | Demonstrates how to create an animated button. |
| Create a Button by Using Microsoft Expression Blend | Demonstrates the process of creating a customized button by using Microsoft Expression Blend. |
| Walkthrough: Display Data from a SQL Server Database in a DataGrid Control | Demonstrates how to retrieve data from a SQL Server database and display that data in a DataGrid control. |

## Migration and Interoperability in WPF

| TITLE | DESCRIPTION |
| --- | --- |

| TITLE | DESCRIPTION |
| --- | --- |
| Walkthrough: Hosting a Windows Forms Control in WPF | Demonstrates how to host a Windows Forms System.Windows.Forms.MaskedTextBox control in a WPF application. |
| Walkthrough: Hosting a Windows Forms Composite Control in WPF | Demonstrates how to host a Windows Forms data-entry composite control in a WPF application. |
| Walkthrough: Hosting a WPF Composite Control in Windows Forms | Demonstrates how to host a WPF data-entry composite control in a Windows Forms application. |
| Walkthrough: Arranging Windows Forms Controls in WPF | Demonstrates how to use WPF layout features to arrange Windows Forms controls in a hybrid application. |
| Walkthrough: Binding to Data in Hybrid Applications | Demonstrates how to use data binding in hybrid applications that include both Windows Forms and WPF controls. |

## Related Sections

| TITLE | DESCRIPTION |
| --- | --- |
| Visual Studio Walkthroughs | Gives a related list of walkthroughs for all areas of programming in Visual Studio. |

# WPF community feedback

1/23/2019 • 4 minutes to read • Edit Online

Microsoft exposes a variety of community resources for you to learn about, discuss, and provide feedback on Windows Presentation Foundation (WPF). These resources include forums and the Visual Studio Developer Community site. Each community resource offers a different set of benefits. These benefits are described here, as are a set of best practices for using each to ensure the best response from the community at large and Microsoft in particular.

> **NOTE**
> Don't use the feedback section located at the bottom of each page to send product feedback. These links are for documentation feedback only.

## Forums

The WPF forum is the primary community resource for discussing and resolving issues. Forums facilitate discussion and problem resolution by offering a comprehensive set of supporting features that include:

- Searching.
- Discussion tracking.
- Rich formatting for text and code.
- Visual Studio integration.
- Most Valued Professional (MVP) and community involvement.
- Monitoring to ensure posts are responded to in the quickest possible time.

Another option for you to ask questions to the community about WPF is Stack Overflow.

### Forum best practices

Using the following best practices help to address issues posted to the WPF forum in the quickest possible time. These practices are applicable to all forums.

#### Search existing posts

Some issues occur widely enough that others have faced them before you. Consequently, you can solve your problem quickly, or you can add your input to an existing discussion.

#### Use meaningful titles

Concise, meaningful titles improve the discoverability of your posts. They also make it easier for other WPF forum community members to determine if they can solve your problem.

#### Include appropriate content

Describe the issue and how you've tried to work through it. If possible, include supporting code snippets, or the simplest possible sample that demonstrates your issue. All these details help to increase the chance your question will be answered quickly.

## Visual Studio Developer Community

Issues can sometimes be difficult to resolve, or irresolvable. Such situations arise because of bugs in the technology, difficulties applying the technology to particular scenarios, or lack of support for particular scenarios. This information is important to Microsoft, and can be provided via the Visual Studio Developer Community site.

Items posted on the WPF Product Feedback Center are routed to the WPF team's internal bug database. Consequently, it is the most reliable way to get your feedback to the WPF feature owner. In addition, you can validate and track suggestions and bugs as well as vote on them, which helps the WPF team to prioritize issues.

### Developer Community best practices

When posting to the Visual Studio Developer Community, searching existing posts, providing a meaningful title and appropriate content are important best practices, just as they are for posting to the WPF forum. The following are additional best practices you should also employ.

#### Search existing posts

Some issues occur widely enough that others have faced them before you. Consequently, you can solve your problem quickly, or you can add your input to an existing issue.

#### Use meaningful titles

Concise, meaningful titles increase the chance that your issue is directed to the most appropriate WPF team in the shortest amount of time. This is particularly important for a technology like WPF, which contains many interrelated features.

#### Describe how to reproduce your bug

When you post about a bug, it is important to include the following where relevant:

- Provide a clear description of the bug.
- Use code snippets to support the bug description.
- Provide a list of steps that demonstrate how to reproduce the bug.
- Include the smallest possible code sample that reproduces the bug.
- Mention whether the bug is consistently reproducible or not.
- Include relevant exception information.

If the bug is install or setup related, attach the relevant install logs and snapshots (files prefixed with "dd_" that are located in your %temp% folder).

For compile or build issues, attach the build logs. The MSBuild system can be configured to supports logging with various verbosities by using the /v: switch from the command line or by configuring the appropriate level from an Integrated Development Environment (IDE) like Visual Studio.

#### Provide environment information

Background information can often be useful for adding context to your post. In particular, mention the operating system platform, processor family, and architecture, such as "Windows 10 Version 1709, Intel(R) Xeon(R), x64."

If the issue you are posting about is related to rendering, you should also include graphics card and driver details, if possible. This information is important because WPF is a presentation framework.

#### Provide solution or project information

Bugs may pertain to the tools used to develop and build your applications and the types of applications you are building. Consequently, it can be useful to specify:

- The type of application you are building, such as:
  - Application (*.exe*) or library (*.dll*)
  - Extensible Application Markup Language (XAML) browser application (XBAP)
  - Loose XAML application
  - Standalone installed applications
  - Standalone ClickOnce-deployed applications
- The development tool, such as:
  - MSBuild
  - Expression Graphic Designer

    - Expression Interactive Designer
    - Visual Studio
- The solution configuration, such as:
    - A solution
    - A single project
    - A solution with multiple dependent projects
- Whether your application has language-specific or language-neutral resources. For example, did you specify the `UICulture` project property or localizable metadata for `Application`, `Page`, and `Resource` types?
- Whether you used the neutral language setting in the AssemblyInfo.cs or AssemblyInfo.vb file.

#### Provide scenario and impact information

Provide information about the scenario that manifests the bug and its impact. This information is highly important to the WPF team when it decides if, when, and how a problem should be fixed, or whether an acceptable workaround can be used instead.

Ordinarily, crash and data loss scenarios are high impact and, therefore, the easiest to prioritize. Some bugs, however, only show up in uncommon scenarios, which may also be mainline scenarios in some cases. Providing context around scenario and impact helps the WPF team make the right decision.

## See also

- How to report a problem with Visual Studio 2017