Contents

Application Development

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

Windows Presentation Foundation (WPF) is a presentation framework that can be used to develop the following types of applications:

- Standalone Applications (traditional style Windows applications built as executable assemblies that are installed to and run from the client computer).
- XAML browser applications (XBAPs) (applications composed of navigation pages that are built as executable assemblies and hosted by Web browsers such as Microsoft Internet Explorer or Mozilla Firefox).
- Custom Control Libraries (non-executable assemblies containing reusable controls).
- Class Libraries (non-executable assemblies that contain reusable classes).

> **NOTE**
> Using WPF types in a Windows service is strongly discouraged. If you attempt to use these features in a Windows service, they may not work as expected.

To build this set of applications, WPF implements a host of services. This topic provides an overview of these services and where to find more information.

Application Management

Executable WPF applications commonly require a core set of functionality that includes the following:

- Creating and managing common application infrastructure (including creating an entry point method and a Windows message loop to receive system and input messages).
- Tracking and interacting with the lifetime of an application.
- Retrieving and processing command-line parameters.
- Sharing application-scope properties and UI resources.
- Detecting and processing unhandled exceptions.
- Returning exit codes.
- Managing windows in standalone applications.
- Tracking navigation in XAML browser applications (XBAPs), and standalone applications with navigation windows and frames.

These capabilities are implemented by the Application class, which you add to your applications using an *application definition*.

For more information, see Application Management Overview.

WPF Application Resource, Content, and Data Files

WPF extends the core support in the Microsoft .NET Framework for embedded resources with support for three kinds of non-executable data files: resource, content, and data. For more information, see WPF Application

Resource, Content, and Data Files.

A key component of the support for WPF non-executable data files is the ability to identify and load them using a unique URI. For more information, see Pack URIs in WPF.

Windows and Dialog Boxes

Users interact with WPF standalone applications through windows. The purpose of a window is to host application content and expose application functionality that usually allows users to interact with the content. In WPF, windows are encapsulated by the Window class, which supports:

- Creating and showing windows.
- Establishing owner/owned window relationships.
- Configuring window appearance (for example, size, location, icons, title bar text, border).
- Tracking and interacting with the lifetime of a window.

For more information, see WPF Windows Overview.

Window supports the ability to create a special type of window known as a dialog box. Both modal and modeless types of dialog boxes can be created.

For convenience, and the benefits of reusability and a consistent user experience across applications, WPF exposes three of the common Windows dialog boxes: OpenFileDialog, SaveFileDialog, and PrintDialog.

A message box is a special type of dialog box for showing important textual information to users, and for asking simple Yes/No/OK/Cancel questions. You use the MessageBox class to create and show message boxes.

For more information, see Dialog Boxes Overview.

Navigation

WPF supports Web-style navigation using pages (Page) and hyperlinks (Hyperlink). Navigation can be implemented in a variety of ways that include the following:

- Standalone pages that are hosted in a Web browser.
- Pages compiled into an XBAP that is hosted in a Web browser.
- Pages compiled into a standalone application and hosted by a navigation window (NavigationWindow).
- Pages that are hosted by a frame (Frame), which may be hosted in a standalone page, or a page compiled into either an XBAP or a standalone application.

To facilitate navigation, WPF implements the following:

- NavigationService, the shared navigation engine for processing navigation requests that is used by Frame, NavigationWindow, and XBAPs to support intra-application navigation.
- Navigation methods to initiate navigation.
- Navigation events to track and interact with navigation lifetime.
- Remembering back and forward navigation using a journal, which can also be inspected and manipulated.

For information, see Navigation Overview.

WPF also supports a special type of navigation known as structured navigation. Structured navigation can be used to call one or more pages that return data in a structured and predictable way that is consistent with calling functions. This capability depends on the PageFunction<T> class, which is described further in Structured Navigation Overview. PageFunction<T> also serves to simplify the creation of complex navigation topologies, which are described in Navigation Topologies Overview.

Hosting

XBAPs can be hosted in Microsoft Internet Explorer or Firefox. Each hosting model has its own set of considerations and constraints that are covered in Hosting.

Build and Deploy

Although simple WPF applications can be built from a command prompt using command-line compilers, WPF integrates with Microsoft Visual Studio to provide additional support that simplified the development and build process. For more information, see Building a WPF Application.

Depending on the type of application you build, there are one or more deployment options to choose from. For more information, see Deploying a WPF Application.

Related Topics

TITLE	DESCRIPTION
Application Management Overview	Provides an overview of the Application class including managing application lifetime, windows, application resources, and navigation.
Windows in WPF	Provides details of managing windows in your application including how to use the Window class and dialog boxes.
Navigation Overview	Provides an overview of managing navigation between pages of your application.
Hosting	Provides an overview of XAML browser applications (XBAPs).
Build and Deploy	Describes how to build and deploy your WPF application.
Introduction to WPF in Visual Studio	Describes the main features of WPF.
Walkthrough: My first WPF desktop application	A walkthrough that shows how to create a WPF application using page navigation, layout, controls, images, styles, and binding.

Application Management Overview

1/29/2019 • 20 minutes to read • Edit Online

All applications tend to share a common set of functionality that applies to application implementation and management. This topic provides an overview of the functionality in the Application class for creating and managing applications.

The Application Class

In WPF, common application-scoped functionality is encapsulated in the Application class. The Application class includes the following functionality:

- Tracking and interacting with application lifetime.
- Retrieving and processing command-line parameters.
- Detecting and responding to unhandled exceptions.
- Sharing application-scope properties and resources.
- Managing windows in standalone applications.
- Tracking and managing navigation.

How to Perform Common Tasks Using the Application Class

If you are not interested in all of the details of the Application class, the following table lists some of the common tasks for Application and how to accomplish them. By viewing the related API and topics, you can find more information and sample code.

TASK	APPROACH
Get an object that represents the current application	Use the Application.Current property.
Add a startup screen to an application	See Add a Splash Screen to a WPF Application.
Start an application	Use the Application.Run method.
Stop an application	Use the Shutdown method of the Application.Current object.
Get arguments from the command line	Handle the Application.Startup event and use the StartupEventArgs.Args property. For an example, see the Application.Startup event.
Get and set the application exit code	Set the ExitEventArgs.ApplicationExitCode property in the Application.Exit event handler or call the Shutdown method and pass in an integer.
Detect and respond to unhandled exceptions	Handle the DispatcherUnhandledException event.
Get and set application-scoped resources	Use the Application.Resources property.

TASK	APPROACH
Use an application-scope resource dictionary	See Use an Application-Scope Resource Dictionary.
Get and set application-scoped properties	Use the Application.Properties property.
Get and save an application's state	See Persist and Restore Application-Scope Properties Across Application Sessions.
Manage non-code data files, including resource files, content files, and site-of-origin files.	See WPF Application Resource, Content, and Data Files.
Manage windows in standalone applications	See WPF Windows Overview.
Track and manage navigation	See Navigation Overview.

The Application Definition

To utilize the functionality of the Application class, you must implement an application definition. A WPF application definition is a class that derives from Application and is configured with a special MSBuild setting.

Implementing an Application Definition

A typical WPF application definition is implemented using both markup and code-behind. This allows you to use markup to declaratively set application properties, resources, and register events, while handling events and implementing application-specific behavior in code-behind.

The following example shows how to implement an application definition using both markup and code-behind:

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

```
using System.Windows;

namespace SDKSample
{
    public partial class App : Application { }
}
```

```
Imports Microsoft.VisualBasic
Imports System.Windows

Namespace SDKSample
    Partial Public Class App
        Inherits Application
    End Class
End Namespace
```

To allow a markup file and code-behind file to work together, the following needs to happen:

- In markup, the `Application` element must include the `x:Class` attribute. When the application is built, the existence of `x:Class` in the markup file causes MSBuild to create a `partial` class that derives from Application and has the name that is specified by the `x:Class` attribute. This requires the addition of an

XML namespace declaration for the XAML schema (`xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`).

- In code-behind, the class must be a `partial` class with the same name that is specified by the `x:Class` attribute in markup and must derive from Application. This allows the code-behind file to be associated with the `partial` class that is generated for the markup file when the application is built (see Building a WPF Application).

NOTE

When you create a new WPF Application project or WPF Browser Application project using Visual Studio, an application definition is included by default and is defined using both markup and code-behind.

This code is the minimum that is required to implement an application definition. However, an additional MSBuild configuration needs to be made to the application definition before building and running the application.

Configuring the Application Definition for MSBuild

Standalone applications and XAML browser applications (XBAPs) require the implementation of a certain level of infrastructure before they can run. The most important part of this infrastructure is the entry point. When an application is launched by a user, the operating system calls the entry point, which is a well-known function for starting applications.

Traditionally, developers have needed to write some or all of this code for themselves, depending on the technology. However, WPF generates this code for you when the markup file of your application definition is configured as an MSBuild `ApplicationDefinition` item, as shown in the following MSBuild project file:

```
<Project
  DefaultTargets="Build"
                        xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  ...
  <ApplicationDefinition Include="App.xaml" />
  <Compile Include="App.xaml.cs" />
  ...
</Project>
```

Because the code-behind file contains code, it is marked as an MSBuild `Compile` item, as is normal.

The application of these MSBuild configurations to the markup and code-behind files of an application definition causes MSBuild to generate code like the following:

```csharp
using System;
using System.Windows;

namespace SDKSample
{
    public class App : Application
    {
        public App() { }
        [STAThread]
        public static void Main()
        {
            // Create new instance of application subclass
            App app = new App();

            // Code to register events and set properties that were
            // defined in XAML in the application definition
            app.InitializeComponent();

            // Start running the application
            app.Run();
        }

        public void InitializeComponent()
        {
            // Initialization code goes here.
        }
    }
}
```

```vb
Imports System.Windows

Namespace SDKSample
    Public Class App
        Inherits Application
        Public Sub New()
        End Sub
        <STAThread>
        Public Shared Sub Main()
            ' Create new instance of application subclass
            Dim app As New App()

            ' Code to register events and set properties that were
            ' defined in XAML in the application definition
            app.InitializeComponent()

            ' Start running the application
            app.Run()
        End Sub

        Public Sub InitializeComponent()
            ' Initialization code goes here.
        End Sub
    End Class
End Namespace
```

The resulting code augments your application definition with additional infrastructure code, which includes the entry-point method `Main`. The STAThreadAttribute attribute is applied to the `Main` method to indicate that the main UI thread for the WPF application is an STA thread, which is required for WPF applications. When called, `Main` creates a new instance of `App` before calling the `InitializeComponent` method to register the events and set the properties that are implemented in markup. Because `InitializeComponent` is generated for you, you don't need to explicitly call `InitializeComponent` from an application definition like you do for Page and Window implementations. Finally, the Run method is called to start the application.

Getting the Current Application

Because the functionality of the Application class are shared across an application, there can be only one instance of the Application class per AppDomain. To enforce this, the Application class is implemented as a singleton class (see Implementing Singleton in C#), which creates a single instance of itself and provides shared access to it with the `static` Current property.

The following code shows how to acquire a reference to the Application object for the current AppDomain.

```
// Get current application
Application current = App.Current;
```

```
' Get current application
Dim current As Application = App.Current
```

Current returns a reference to an instance of the Application class. If you want a reference to your Application derived class you must cast the value of the Current property, as shown in the following example.

```
// Get strongly-typed current application
App app = (App)App.Current;
```

```
' Get strongly-typed current application
Dim appCurrent As App = CType(App.Current, App)
```

You can inspect the value of Current at any point in the lifetime of an Application object. However, you should be careful. After the Application class is instantiated, there is a period during which the state of the Application object is inconsistent. During this period, Application is performing the various initialization tasks that are required by your code to run, including establishing application infrastructure, setting properties, and registering events. If you try to use the Application object during this period, your code may have unexpected results, particularly if it depends on the various Application properties being set.

When Application completes its initialization work, its lifetime truly begins.

Application Lifetime

The lifetime of a WPF application is marked by several events that are raised by Application to let you know when your application has started, has been activated and deactivated, and has been shut down.

Splash Screen

Starting in the .NET Framework 3.5 SP1, you can specify an image to be used in a startup window, or *splash screen*. The SplashScreen class makes it easy to display a startup window while your application is loading. The SplashScreen window is created and shown before Run is called. For more information, see Application Startup Time and Add a Splash Screen to a WPF Application.

Starting an Application

After Run is called and the application is initialized, the application is ready to run. This moment is signified when the Startup event is raised:

```
using System.Windows;

namespace SDKSample
{
    public partial class App : Application
    {
        void App_Startup(object sender, StartupEventArgs e)
        {
            // Application is running
        }
    }
}
```

```
Imports System.Windows

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private Sub App_Startup(ByVal sender As Object, ByVal e As StartupEventArgs)
            ' Application is running
        End Sub
    End Class
End Namespace
```

At this point in an application's lifetime, the most common thing to do is to show a UI.

Showing a User Interface

Most standalone Windows applications open a Window when they begin running. The Startup event handler is one location from which you can do this, as demonstrated by the following code.

```
<Application
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.App"
  Startup="App_Startup" />
```

```
using System.Windows;

namespace SDKSample
{
    public partial class App : Application
    {
        void App_Startup(object sender, StartupEventArgs e)
        {
            // Open a window
            MainWindow window = new MainWindow();
            window.Show();
        }
    }
}
```

```
Imports Microsoft.VisualBasic
Imports System.Windows

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private Sub App_Startup(ByVal sender As Object, ByVal e As StartupEventArgs)
            ' Open a window
            Dim window As New MainWindow()
            window.Show()
        End Sub
    End Class
End Namespace
```

NOTE

The first Window to be instantiated in a standalone application becomes the main application window by default. This Window object is referenced by the Application.MainWindow property. The value of the MainWindow property can be changed programmatically if a different window than the first instantiated Window should be the main window.

When an XBAP first starts, it will most likely navigate to a Page. This is shown in the following code.

```
<Application
  x:Class="SDKSample.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Startup="App_Startup" />
```

```
using System;
using System.Windows;
using System.Windows.Navigation;

namespace SDKSample
{
    public partial class App : Application
    {
        void App_Startup(object sender, StartupEventArgs e)
        {
            ((NavigationWindow)this.MainWindow).Navigate(new Uri("HomePage.xaml", UriKind.Relative));
        }
    }
}
```

```
Imports System
Imports System.Windows
Imports System.Windows.Navigation

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private Sub App_Startup(ByVal sender As Object, ByVal e As StartupEventArgs)
            CType(Me.MainWindow, NavigationWindow).Navigate(New Uri("HomePage.xaml", UriKind.Relative))
        End Sub
    End Class
End Namespace
```

If you handle Startup to only open a Window or navigate to a Page, you can set the `StartupUri` attribute in markup instead.

The following example shows how to use the StartupUri from a standalone application to open a Window.

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

The following example shows how to use StartupUri from an XBAP to navigate to a Page.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    StartupUri="HomePage.xaml" />
```

This markup has the same effect as the previous code for opening a window.

NOTE

For more information on navigation, see Navigation Overview.

You need to handle the Startup event to open a Window if you need to instantiate it using a non-default constructor, or you need to set its properties or subscribe to its events before showing it, or you need to process any command-line arguments that were supplied when the application was launched.

Processing Command-Line Arguments

In Windows, standalone applications can be launched from either a command prompt or the desktop. In both cases, command-line arguments can be passed to the application. The following example shows an application that is launched with a single command-line argument, "/StartMinimized":

```
wpfapplication.exe /StartMinimized
```

During application initialization, WPF retrieves the command-line arguments from the operating system and passes them to the Startup event handler via the Args property of the StartupEventArgs parameter. You can retrieve and store the command-line arguments using code like the following.

```
<Application
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.App"
  Startup="App_Startup" />
```

```csharp
using System.Windows;

namespace SDKSample
{
    public partial class App : Application
    {
        void App_Startup(object sender, StartupEventArgs e)
        {
            // Application is running
            // Process command line args
            bool startMinimized = false;
            for (int i = 0; i != e.Args.Length; ++i)
            {
                if (e.Args[i] == "/StartMinimized")
                {
                    startMinimized = true;
                }
            }

            // Create main application window, starting minimized if specified
            MainWindow mainWindow = new MainWindow();
            if (startMinimized)
            {
                mainWindow.WindowState = WindowState.Minimized;
            }
            mainWindow.Show();
        }
    }
}
```

```vb
Imports Microsoft.VisualBasic
Imports System.Windows

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private Sub App_Startup(ByVal sender As Object, ByVal e As StartupEventArgs)
            ' Application is running
            ' Process command line args
            Dim startMinimized As Boolean = False
            Dim i As Integer = 0
            Do While i <> e.Args.Length
                If e.Args(i) = "/StartMinimized" Then
                    startMinimized = True
                End If
                i += 1
            Loop

            ' Create main application window, starting minimized if specified
            Dim mainWindow As New MainWindow()
            If startMinimized Then
                mainWindow.WindowState = WindowState.Minimized
            End If
            mainWindow.Show()
        End Sub
    End Class
End Namespace
```

The code handles Startup to check whether the **/StartMinimized** command-line argument was provided; if so, it opens the main window with a WindowState of Minimized. Note that because the WindowState property must be set programmatically, the main Window must be opened explicitly in code.

XBAPs cannot retrieve and process command-line arguments because they are launched using ClickOnce deployment (see Deploying a WPF Application). However, they can retrieve and process query string parameters from the URLs that are used to launch them.

Application Activation and Deactivation

Windows allows users to switch between applications. The most common way is to use the ALT+TAB key combination. An application can only be switched to if it has a visible Window that a user can select. The currently selected Window is the *active window* (also known as the *foreground window*) and is the Window that receives user input. The application with the active window is the *active application* (or *foreground application*). An application becomes the active application in the following circumstances:

- It is launched and shows a Window.
- A user switches from another application by selecting a Window in the application.

You can detect when an application becomes active by handling the Application.Activated event.

Likewise, an application can become inactive in the following circumstances:

- A user switches to another application from the current one.
- When the application shuts down.

You can detect when an application becomes inactive by handling the Application.Deactivated event.

The following code shows how to handle the Activated and Deactivated events to determine whether an application is active.

```
<Application
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.App"
  StartupUri="MainWindow.xaml"
  Activated="App_Activated"
  Deactivated="App_Deactivated" />
```

```
using System;
using System.Windows;

namespace SDKSample
{
    public partial class App : Application
    {
        bool isApplicationActive;

        void App_Activated(object sender, EventArgs e)
        {
            // Application activated
            this.isApplicationActive = true;
        }

        void App_Deactivated(object sender, EventArgs e)
        {
            // Application deactivated
            this.isApplicationActive = false;
        }
    }
}
```

```
Imports Microsoft.VisualBasic
Imports System
Imports System.Windows

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private isApplicationActive As Boolean

        Private Sub App_Activated(ByVal sender As Object, ByVal e As EventArgs)
            ' Application activated
            Me.isApplicationActive = True
        End Sub

        Private Sub App_Deactivated(ByVal sender As Object, ByVal e As EventArgs)
            ' Application deactivated
            Me.isApplicationActive = False
        End Sub
    End Class
End Namespace
```

A Window can also be activated and deactivated. See Window.Activated and Window.Deactivated for more information.

> **NOTE**
> Neither Application.Activated nor Application.Deactivated is raised for XBAPs.

Application Shutdown

The life of an application ends when it is shut down, which can occur for the following reasons:

- A user closes every Window.
- A user closes the main Window.
- A user ends the Windows session by logging off or shutting down.
- An application-specific condition has been met.

To help you manage application shutdown, Application provides the Shutdown method, the ShutdownMode property, and the SessionEnding and Exit events.

> **NOTE**
> Shutdown can only be called from applications that have UIPermission. Standalone WPF applications always have this permission. However, XBAPs running in the Internet zone partial-trust security sandbox do not.

Shutdown Mode

Most applications shut down either when all the windows are closed or when the main window is closed. Sometimes, however, other application-specific conditions may determine when an application shuts down. You can specify the conditions under which your application will shut down by setting ShutdownMode with one of the following ShutdownMode enumeration values:

- OnLastWindowClose
- OnMainWindowClose
- OnExplicitShutdown

The default value of ShutdownMode is OnLastWindowClose, which means that an application automatically shuts down when the last window in the application is closed by the user. However, if your application should be shut down when the main window is closed, WPF automatically does that if you set ShutdownMode to OnMainWindowClose. This is shown in the following example.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.App"
    ShutdownMode="OnMainWindowClose" />
```

When you have application-specific shutdown conditions, you set ShutdownMode to OnExplicitShutdown. In this case, it is your responsibility to shut an application down by explicitly calling the Shutdown method; otherwise, your application will continue running even if all the windows are closed. Note that Shutdown is called implicitly when the ShutdownMode is either OnLastWindowClose or OnMainWindowClose.

NOTE

ShutdownMode can be set from an XBAP, but it is ignored; an XBAP is always shut down when it is navigated away from in a browser or when the browser that hosts the XBAP is closed. For more information, see Navigation Overview.

Session Ending

The shutdown conditions that are described by the ShutdownMode property are specific to an application. In some cases, though, an application may shut down as a result of an external condition. The most common external condition occurs when a user ends the Windows session by the following actions:

- Logging off
- Shutting down
- Restarting
- Hibernating

To detect when a Windows session ends, you can handle the SessionEnding event, as illustrated in the following example.

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

```
using System.Windows;

namespace SDKSample
{
    public partial class App : Application
    {
        void App_SessionEnding(object sender, SessionEndingCancelEventArgs e)
        {
            // Ask the user if they want to allow the session to end
            string msg = string.Format("{0}. End session?", e.ReasonSessionEnding);
            MessageBoxResult result = MessageBox.Show(msg, "Session Ending", MessageBoxButton.YesNo);

            // End session, if specified
            if (result == MessageBoxResult.No)
            {
                e.Cancel = true;
            }
        }
    }
}
```

```
Imports Microsoft.VisualBasic
Imports System.Windows

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private Sub App_SessionEnding(ByVal sender As Object, ByVal e As SessionEndingCancelEventArgs)
            ' Ask the user if they want to allow the session to end
            Dim msg As String = String.Format("{0}. End session?", e.ReasonSessionEnding)
            Dim result As MessageBoxResult = MessageBox.Show(msg, "Session Ending", MessageBoxButton.YesNo)

            ' End session, if specified
            If result = MessageBoxResult.No Then
                e.Cancel = True
            End If
        End Sub
    End Class
End Namespace
```

In this example, the code inspects the ReasonSessionEnding property to determine how the Windows session is ending. It uses this value to display a confirmation message to the user. If the user does not want the session to end, the code sets Cancel to `true` to prevent the Windows session from ending.

> **NOTE**
> SessionEnding is not raised for XBAPs.

Exit

When an application shuts down, it may need to perform some final processing, such as persisting application state. For these situations, you can handle the Exit event, as the `App_Exit` event handler does in the following example. It is defined as an event handler in the *App.xaml* file. Its implementation is highlighted in the *App.xaml.cs* and *Application.xaml.vb* files.

```xml
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.App"
    StartupUri="MainWindow.xaml"
    Startup="App_Startup"
    Exit="App_Exit">
    <Application.Resources>
        <SolidColorBrush x:Key="ApplicationScopeResource" Color="White"></SolidColorBrush>
    </Application.Resources>
</Application>
```

```
using System.Windows;
using System.IO;
using System.IO.IsolatedStorage;

namespace SDKSample
{
    public partial class App : Application
    {
        string filename = "App.txt";

        public App()
        {
            // Initialize application-scope property
            this.Properties["NumberOfAppSessions"] = 0;
        }

        private void App_Startup(object sender, StartupEventArgs e)
        {
            // Restore application-scope property from isolated storage
            IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForDomain();
            try
            {
                using (IsolatedStorageFileStream stream = new IsolatedStorageFileStream(filename,
FileMode.Open, storage))
                using (StreamReader reader = new StreamReader(stream))
                {
                    // Restore each application-scope property individually
                    while (!reader.EndOfStream)
                    {
                        string[] keyValue = reader.ReadLine().Split(new char[] {','});
                        this.Properties[keyValue[0]] = keyValue[1];
                    }
                }
            }
            catch (FileNotFoundException ex)
            {
                // Handle when file is not found in isolated storage:
                // * When the first application session
                // * When file has been deleted
            }
        }

        private void App_Exit(object sender, ExitEventArgs e)
        {
            // Persist application-scope property to isolated storage
            IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForDomain();
            using (IsolatedStorageFileStream stream = new IsolatedStorageFileStream(filename, FileMode.Create,
storage))
            using (StreamWriter writer = new StreamWriter(stream))
            {
                // Persist each application-scope property individually
                foreach (string key in this.Properties.Keys)
                {
                    writer.WriteLine("{0},{1}", key, this.Properties[key]);
                }
            }
        }
    }
}
```

```
Imports System.IO
Imports System.IO.IsolatedStorage

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private filename As String = "App.txt"

        Public Sub New()
            ' Initialize application-scope property
            Me.Properties("NumberOfAppSessions") = 0
        End Sub

        Private Sub App_Startup(ByVal sender As Object, ByVal e As StartupEventArgs)
            ' Restore application-scope property from isolated storage
            Dim storage As IsolatedStorageFile = IsolatedStorageFile.GetUserStoreForDomain()
            Try
                Using stream As New IsolatedStorageFileStream(filename, FileMode.Open, storage)
                Using reader As New StreamReader(stream)
                    ' Restore each application-scope property individually
                    Do While Not reader.EndOfStream
                        Dim keyValue() As String = reader.ReadLine().Split(New Char() {","c})
                        Me.Properties(keyValue(0)) = keyValue(1)
                    Loop
                End Using
                End Using
            Catch ex As FileNotFoundException
                ' Handle when file is not found in isolated storage:
                ' * When the first application session
                ' * When file has been deleted
            End Try
        End Sub

        Private Sub App_Exit(ByVal sender As Object, ByVal e As ExitEventArgs)
            ' Persist application-scope property to isolated storage
            Dim storage As IsolatedStorageFile = IsolatedStorageFile.GetUserStoreForDomain()
            Using stream As New IsolatedStorageFileStream(filename, FileMode.Create, storage)
            Using writer As New StreamWriter(stream)
                ' Persist each application-scope property individually
                For Each key As String In Me.Properties.Keys
                    writer.WriteLine("{0},{1}", key, Me.Properties(key))
                Next key
            End Using
            End Using
        End Sub
    End Class
End Namespace
```

For the complete example, see Persist and Restore Application-Scope Properties Across Application Sessions.

Exit can be handled by both standalone applications and XBAPs. For XBAPs, Exit is raised when in the following circumstances:

- An XBAP is navigated away from.
- In Internet Explorer 7, when the tab that is hosting the XBAP is closed.
- When the browser is closed.

Exit Code

Applications are mostly launched by the operating system in response to a user request. However, an application can be launched by another application to perform some specific task. When the launched application shuts down, the launching application may want to know the condition under which the launched application shut down. In these situations, Windows allows applications to return an application exit code on shutdown. By default, WPF applications return an exit code value of 0.

NOTE

When you debug from Visual Studio, the application exit code is displayed in the **Output** window when the application shuts down, in a message that looks like the following:

```
The program '[5340] AWPFApp.vshost.exe: Managed' has exited with code 0 (0x0).
```

You open the **Output** window by clicking **Output** on the **View** menu.

To change the exit code, you can call the Shutdown(Int32) overload, which accepts an integer argument to be the exit code:

```
// Shutdown and return a non-default exit code
Application.Current.Shutdown(-1);
```

```
' Shutdown and return a non-default exit code
Application.Current.Shutdown(-1)
```

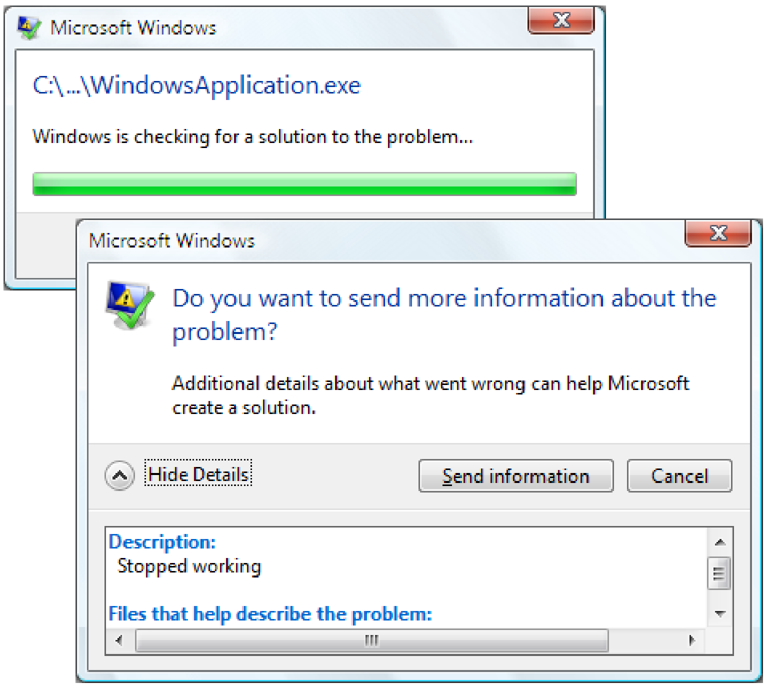
You can detect the value of the exit code, and change it, by handling the Exit event. The Exit event handler is passed an ExitEventArgs which provides access to the exit code with the ApplicationExitCode property. For more information, see Exit.

NOTE

You can set the exit code in both standalone applications and XBAPs. However, the exit code value is ignored for XBAPs.

Unhandled Exceptions

Sometimes an application may shut down under abnormal conditions, such as when an unanticipated exception is thrown. In this case, the application may not have the code to detect and process the exception. This type of exception is an unhandled exception; a notification similar to that shown in the following figure is displayed before the application is closed.


Microsoft Windows

C:\...\WindowsApplication.exe

Windows is checking for a solution to the problem...

Microsoft Windows

Do you want to send more information about the problem?

Additional details about what went wrong can help Microsoft create a solution.

Description:
Stopped working

Files that help describe the problem:

From the user experience perspective, it is better for an application to avoid this default behavior by doing some or all of the following:

- Displaying user-friendly information.
- Attempting to keep an application running.
- Recording detailed, developer-friendly exception information in the Windows event log.

Implementing this support depends on being able to detect unhandled exceptions, which is what the DispatcherUnhandledException event is raised for.

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

```
using System.Windows;
using System.Windows.Threading;

namespace SDKSample
{
    public partial class App : Application
    {
        void App_DispatcherUnhandledException(object sender, DispatcherUnhandledExceptionEventArgs e)
        {
            // Process unhandled exception

            // Prevent default unhandled exception processing
            e.Handled = true;
        }
    }
}
```

```
Imports System.Windows
Imports System.Windows.Threading

Namespace SDKSample
    Partial Public Class App
        Inherits Application
        Private Sub App_DispatcherUnhandledException(ByVal sender As Object, ByVal e As
DispatcherUnhandledExceptionEventArgs)
            ' Process unhandled exception

            ' Prevent default unhandled exception processing
            e.Handled = True
        End Sub
    End Class
End Namespace
```

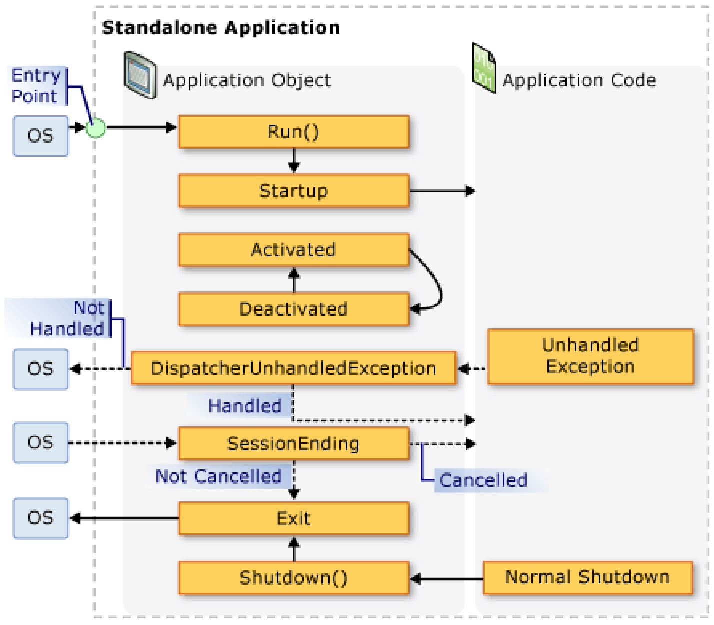
The DispatcherUnhandledException event handler is passed a DispatcherUnhandledExceptionEventArgs parameter that contains contextual information regarding the unhandled exception, including the exception itself (DispatcherUnhandledExceptionEventArgs.Exception). You can use this information to determine how to handle the exception.

When you handle DispatcherUnhandledException, you should set the DispatcherUnhandledExceptionEventArgs.Handled property to `true`; otherwise, WPF still considers the exception to be unhandled and reverts to the default behavior described earlier. If an unhandled exception is raised and either the DispatcherUnhandledException event is not handled, or the event is handled and Handled is set to `false`, the application shuts down immediately. Furthermore, no other Application events are raised. Consequently, you need to handle DispatcherUnhandledException if your application has code that must run before the application shuts down.

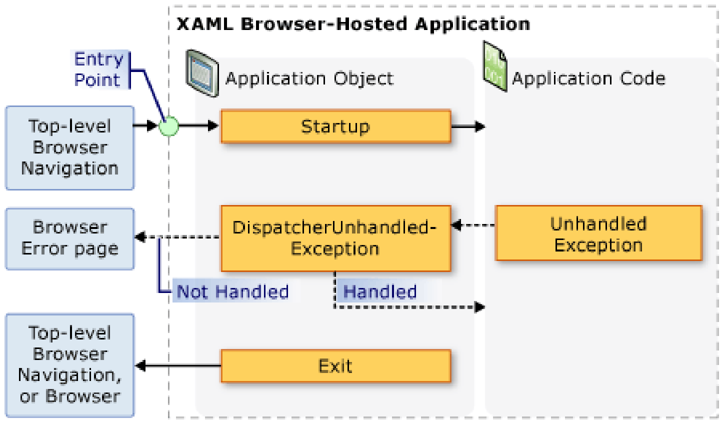
Although an application may shut down as a result of an unhandled exception, an application usually shuts down in response to a user request, as discussed in the next section.

Application Lifetime Events

Standalone applications and XBAPs don't have exactly the same lifetimes. The following figure illustrates the key events in the lifetime of a standalone application and shows the sequence in which they are raised.



Likewise, the following figure illustrates the key events in the lifetime of an XBAP, and shows the sequence in which they are raised.


XAML Browser-Hosted Application
Entry Point
Application Object
Application Code
Top-level Browser Navigation
Startup
Browser Error page
DispatcherUnhandled-Exception
Unhandled Exception
Not Handled
Handled
Top-level Browser Navigation, or Browser
Exit

See also

- Application
- WPF Windows Overview
- Navigation Overview
- WPF Application Resource, Content, and Data Files
- Pack URIs in WPF
- Application Model: How-to Topics
- Application Development

WPF Application Resource, Content, and Data Files

1/23/2019 • 11 minutes to read • Edit Online

Microsoft Windows applications often depend on files that contain non-executable data, such as Extensible Application Markup Language (XAML), images, video, and audio. Windows Presentation Foundation (WPF) offers special support for configuring, identifying, and using these types of data files, which are called application data files. This support revolves around a specific set of application data file types, including:

- **Resource Files**: Data files that are compiled into either an executable or library WPF assembly.
- **Content Files**: Standalone data files that have an explicit association with an executable WPF assembly.
- **Site of Origin Files**: Standalone data files that have no association with an executable WPF assembly.

One important distinction to make between these three types of files is that resource files and content files are known at build time; an assembly has explicit knowledge of them. For site of origin files, however, an assembly may have no knowledge of them at all, or implicit knowledge through a pack uniform resource identifier (URI) reference; the case of the latter, there is no guarantee that the referenced site of origin file actually exists.

To reference application data files, Windows Presentation Foundation (WPF) uses the Pack uniform resource identifier (URI) Scheme, which is described in detail in Pack URIs in WPF).

This topic describes how to configure and use application data files.

Resource Files

If an application data file must always be available to an application, the only way to guarantee availability is to compile it into an application's main executable assembly or one of its referenced assemblies. This type of application data file is known as a *resource file*.

You should use resource files when:

- You don't need to update the resource file's content after it is compiled into an assembly.
- You want to simplify application distribution complexity by reducing the number of file dependencies.
- Your application data file needs to be localizable (see WPF Globalization and Localization Overview).

NOTE

The resource files described in this section are different than the resource files described in XAML Resources and different than the embedded or linked resources described in Managing Application Resources (.NET).

Configuring Resource Files

In WPF, a resource file is a file that is included in an Microsoft build engine (MSBuild) project as a `Resource` item.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ... >
  ...
  <ItemGroup>
    <Resource Include="ResourceFile.xaml" />
  </ItemGroup>
  ...
</Project>
```

NOTE

In Microsoft Visual Studio, you create a resource file by adding a file to a project and setting its `Build Action` to `Resource`.

When the project is built, MSBuild compiles the resource into the assembly.

Using Resource Files

To load a resource file, you can call the GetResourceStream method of the Application class, passing a pack URI that identifies the desired resource file. GetResourceStream returns a StreamResourceInfo object, which exposes the resource file as a Stream and describes its content type.

As an example, the following code shows how to use GetResourceStream to load a Page resource file and set it as the content of a Frame (`pageFrame`):

```
// Navigate to xaml page
Uri uri = new Uri("/PageResourceFile.xaml", UriKind.Relative);
StreamResourceInfo info = Application.GetResourceStream(uri);
System.Windows.Markup.XamlReader reader = new System.Windows.Markup.XamlReader();
Page page = (Page)reader.LoadAsync(info.Stream);
this.pageFrame.Content = page;
```

```
' Navigate to xaml page
Dim uri As New Uri("/PageResourceFile.xaml", UriKind.Relative)
Dim info As StreamResourceInfo = Application.GetResourceStream(uri)
Dim reader As New System.Windows.Markup.XamlReader()
Dim page As Page = CType(reader.LoadAsync(info.Stream), Page)
Me.pageFrame.Content = page
```

While calling GetResourceStream gives you access to the Stream, you need to perform the additional work of converting it to the type of the property that you'll be setting it with. Instead, you can let WPF take care of opening and converting the Stream by loading a resource file directly into the property of a type using code.

The following example shows how to load a Page directly into a Frame (`pageFrame`) using code.

```
Uri pageUri = new Uri("/PageResourceFile.xaml", UriKind.Relative);
this.pageFrame.Source = pageUri;
```

```
Dim pageUri As New Uri("/PageResourceFile.xaml", UriKind.Relative)
Me.pageFrame.Source = pageUri
```

The following example is the markup equivalent of the preceding example.

```
<Frame Name="pageFrame" Source="PageResourceFile.xaml" />
```

Application Code Files as Resource Files

A special set of WPF application code files can be referenced using pack URIs, including windows, pages, flow documents, and resource dictionaries. For example, you can set the Application.StartupUri property with a pack URI that references the window or page that you would like to load when an application starts.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    StartupUri="SOOPage.xaml" />
```

You can do this when a XAML file is included in an Microsoft build engine (MSBuild) project as a `Page` item.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ... >
  ...
  <ItemGroup>
    <Page Include="MainWindow.xaml" />
  </ItemGroup>
  ...
</Project>
```

NOTE

In Visual Studio, you add a new Window, NavigationWindow, Page, FlowDocument, or ResourceDictionary to a project, the `Build Action` for the markup file will default to `Page`.

When a project with `Page` items is compiled, the XAML items are converted to binary format and compiled into the associated assembly. Consequently, these files can be used in the same way as typical resource files.

NOTE

If a XAML file is configured as a `Resource` item, and does not have a code-behind file, the raw XAML is compiled into an assembly rather than a binary version of the raw XAML.

Content Files

A *content file* is distributed as a loose file alongside an executable assembly. Although they are not compiled into an assembly, assemblies are compiled with metadata that establishes an association with each content file.

You should use content files when your application requires a specific set of application data files that you want to be able to update without recompiling the assembly that consumes them.

Configuring Content Files

To add a content file to a project, an application data file must be included as a `Content` item. Furthermore, because a content file is not compiled directly into the assembly, you need to set the MSBuild `CopyToOutputDirectory` metadata element to specify that the content file is copied to a location that is relative to the built assembly. If you want the resource to be copied to the build output folder every time a project is built, you set the `CopyToOutputDirectory` metadata element with the `Always` value. Otherwise, you can ensure that only the newest version of the resource is copied to the build output folder by using the `PreserveNewest` value.

The following shows a file that is configured as a content file which is copied to the build output folder only when a new version of the resource is added to the project.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ... >
  ...
  <ItemGroup>
    <Content Include="ContentFile.xaml">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
  </ItemGroup>
  ...
</Project>
```

NOTE

In Visual Studio, you create a content file by adding a file to a project and setting its `Build Action` to `Content`, and set its `Copy to Output Directory` to `Copy always` (same as `Always`) and `Copy if newer` (same as `PreserveNewest`).

When the project is built, an AssemblyAssociatedContentFileAttribute attribute is compiled into the metadata of the assembly for each content file.

```
[assembly: AssemblyAssociatedContentFile("ContentFile.xaml")]
```

The value of the AssemblyAssociatedContentFileAttribute implies the path to the content file relative to its position in the project. For example, if a content file was located in a project subfolder, the additional path information would be incorporated into the AssemblyAssociatedContentFileAttribute value.

```
[assembly: AssemblyAssociatedContentFile("Resources/ContentFile.xaml")]
```

The AssemblyAssociatedContentFileAttribute value is also the value of the path to the content file in the build output folder.

Using Content Files

To load a content file, you can call the GetContentStream method of the Application class, passing a pack URI that identifies the desired content file. GetContentStream returns a StreamResourceInfo object, which exposes the content file as a Stream and describes its content type.

As an example, the following code shows how to use GetContentStream to load a Page content file and set it as the content of a Frame (`pageFrame`).

```
// Navigate to xaml page
Uri uri = new Uri("/PageContentFile.xaml", UriKind.Relative);
StreamResourceInfo info = Application.GetContentStream(uri);
System.Windows.Markup.XamlReader reader = new System.Windows.Markup.XamlReader();
Page page = (Page)reader.LoadAsync(info.Stream);
this.pageFrame.Content = page;
```

```
' Navigate to xaml page
Dim uri As New Uri("/PageContentFile.xaml", UriKind.Relative)
Dim info As StreamResourceInfo = Application.GetContentStream(uri)
Dim reader As New System.Windows.Markup.XamlReader()
Dim page As Page = CType(reader.LoadAsync(info.Stream), Page)
Me.pageFrame.Content = page
```

While calling GetContentStream gives you access to the Stream, you need to perform the additional work of converting it to the type of the property that you'll be setting it with. Instead, you can let WPF take care of opening and converting the Stream by loading a resource file directly into the property of a type using code.

The following example shows how to load a Page directly into a Frame (`pageFrame`) using code.

```
Uri pageUri = new Uri("/PageContentFile.xaml", UriKind.Relative);
this.pageFrame.Source = pageUri;
```

```
Dim pageUri As New Uri("/PageContentFile.xaml", UriKind.Relative)
Me.pageFrame.Source = pageUri
```

The following example is the markup equivalent of the preceding example.

```
<Frame Name="pageFrame" Source="PageContentFile.xaml" />
```

Site of Origin Files

Resource files have an explicit relationship with the assemblies that they are distributed alongside, as defined by the AssemblyAssociatedContentFileAttribute. But, there are times when you may want to establish either an implicit or non-existent relationship between an assembly and an application data file, including when:

- A file doesn't exist at compile time.
- You don't know what files your assembly will require until run time.
- You want to be able to update files without recompiling the assembly that they are associated with.
- Your application uses large data files, such as audio and video, and you only want users to download them if they choose to.

It is possible to load these types of files by using traditional URI schemes, such as the file:/// and http:// schemes.

```
<Image Source="file:///C:/DataFile.bmp" />
<Image Source="http://www.datafilewebsite.com/DataFile.bmp" />
```

However, the file:/// and http:// schemes require your application to have full trust. If your application is a XAML browser application (XBAP) that was launched from the Internet or intranet, and it requests only the set of permissions that are allowed for applications launched from those locations, loose files can only be loaded from the application's site of origin (launch location). Such files are known as *site of origin* files.

Site of origin files are the only option for partial trust applications, although are not limited to partial trust applications. Full trust applications may still need to load application data files that they do not know about at build time; while full trust applications could use file:///, it is likely that the application data files will be installed in the same folder as, or a subfolder of, the application assembly. In this case, using site of origin referencing is easier than using file:///, because using file:/// requires you to work out the full path the file.

NOTE

Site of origin files are not cached with an XAML browser application (XBAP) on a client machine, while content files are. Consequently, they are only downloaded when specifically requested. If an XAML browser application (XBAP) application has large media files, configuring them as site of origin files means the initial application launch is much faster, and the files are only downloaded on demand.

Configuring Site of Origin Files

If your site of origin files are non-existent or unknown at compile time, you need to use traditional deployment mechanisms for ensuring the required files are available at run time, including using either the `XCopy` command-line program or the Microsoft Windows Installer.

If you do know at compile time the files that you would like to be located at the site of origin, but still want to avoid an explicit dependency, you can add those files to an Microsoft build engine (MSBuild) project as `None` item. As with content files, you need to set the MSBuild `CopyToOutputDirectory` attribute to specify that the site of origin file is copied to a location that is relative to the built assembly, by specifying either the `Always` value or the `PreserveNewest` value.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" ... >
  ...
  <None Include="PageSiteOfOriginFile.xaml">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
  ...
</Project>
```

NOTE

In Visual Studio, you create a site of origin file by adding a file to a project and setting its `Build Action` to `None`.

When the project is built, MSBuild copies the specified files to the build output folder.

Using Site of Origin Files

To load a site of origin file, you can call the GetRemoteStream method of the Application class, passing a pack URI that identifies the desired site of origin file. GetRemoteStream returns a StreamResourceInfo object, which exposes the site of origin file as a Stream and describes its content type.

As an example, the following code shows how to use GetRemoteStream to load a Page site of origin file and set it as the content of a Frame (`pageFrame`).

```
// Navigate to xaml page
Uri uri = new Uri("/SiteOfOriginFile.xaml", UriKind.Relative);
StreamResourceInfo info = Application.GetRemoteStream(uri);
System.Windows.Markup.XamlReader reader = new System.Windows.Markup.XamlReader();
Page page = (Page)reader.LoadAsync(info.Stream);
this.pageFrame.Content = page;
```

```
' Navigate to xaml page
Dim uri As New Uri("/SiteOfOriginFile.xaml", UriKind.Relative)
Dim info As StreamResourceInfo = Application.GetRemoteStream(uri)
Dim reader As New System.Windows.Markup.XamlReader()
Dim page As Page = CType(reader.LoadAsync(info.Stream), Page)
Me.pageFrame.Content = page
```

While calling GetRemoteStream gives you access to the Stream, you need to perform the additional work of converting it to the type of the property that you'll be setting it with. Instead, you can let WPF take care of opening and converting the Stream by loading a resource file directly into the property of a type using code.

The following example shows how to load a Page directly into a Frame (`pageFrame`) using code.

```
Uri pageUri = new Uri("pack://siteoforigin:,,,/SiteOfOriginFile.xaml", UriKind.Absolute);
this.pageFrame.Source = pageUri;
```

```
Dim pageUri As New Uri("pack://siteoforigin:,,,/Subfolder/SiteOfOriginFile.xaml", UriKind.Absolute)
Me.pageFrame.Source = pageUri
```

The following example is the markup equivalent of the preceding example.

```
<Frame Name="pageFrame" Source="pack://siteoforigin:,,,/SiteOfOriginFile.xaml" />
```

Rebuilding After Changing Build Type

After you change the build type of an application data file, you need to rebuild the entire application to ensure those changes are applied. If you only build the application, the changes are not applied.

See also

- Pack URIs in WPF

Pack URIs in WPF

1/23/2019 • 13 minutes to read • Edit Online

In Windows Presentation Foundation (WPF), uniform resource identifiers (URIs) are used to identify and load files in many ways, including the following:

- Specifying the user interface (UI) to show when an application first starts.
- Loading images.
- Navigating to pages.
- Loading non-executable data files.

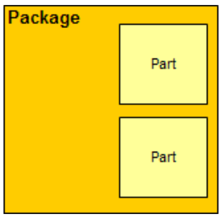
Furthermore, URIs can be used to identify and load files from a variety of locations, including the following:

- The current assembly.
- A referenced assembly.
- A location relative to an assembly.
- The application's site of origin.

To provide a consistent mechanism for identifying and loading these types of files from these locations, WPF leverages the extensibility of the *pack URI scheme*. This topic provides an overview of the scheme, covers how to construct pack URIs for a variety of scenarios, discusses absolute and relative URIs and URI resolution, before showing how to use pack URIs from both markup and code.

The Pack URI Scheme

The pack URI scheme is used by the Open Packaging Conventions (OPC) specification, which describes a model for organizing and identifying content. The key elements of this model are packages and parts, where a *package* is a logical container for one or more logical *parts*. The following figure illustrates this concept.



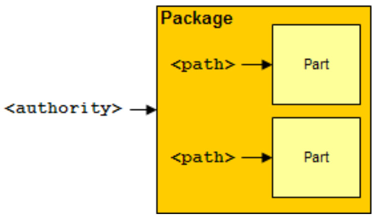
To identify parts, the OPC specification leverages the extensibility of RFC 2396 (Uniform Resource Identifiers (URI): Generic Syntax) to define the pack URI scheme.

The scheme that is specified by a URI is defined by its prefix; http, ftp, and file are well-known examples. The pack URI scheme uses "pack" as its scheme, and contains two components: authority and path. The following is the format for a pack URI.

pack://*authority*/*path*

The *authority* specifies the type of package that a part is contained by, while the *path* specifies the location of a part within a package.

This concept is illustrated by the following figure:

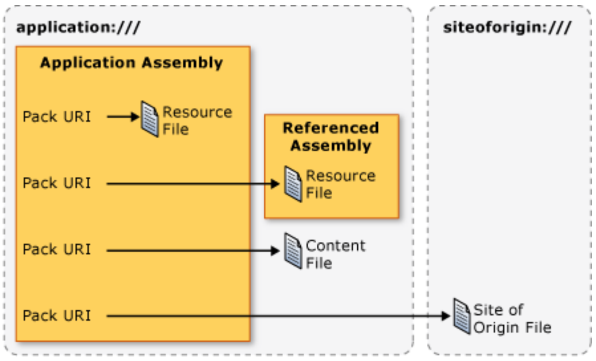


Packages and parts are analogous to applications and files, where an application (package) can include one or more files (parts), including:

- Resource files that are compiled into the local assembly.
- Resource files that are compiled into a referenced assembly.
- Resource files that are compiled into a referencing assembly.

- Content files.
- Site of origin files.

To access these types of files, WPF supports two authorities: application:/// and siteoforigin:///. The application:/// authority identifies application data files that are known at compile time, including resource and content files. The siteoforigin:/// authority identifies site of origin files. The scope of each authority is shown in the following figure.



NOTE

The authority component of a pack URI is an embedded URI that points to a package and must conform to RFC 2396. Additionally, the "/" character must be replaced with the "," character, and reserved characters such as "%" and "?" must be escaped. See the OPC for details.

The following sections explain how to construct pack URIs using these two authorities in conjunction with the appropriate paths for identifying resource, content, and site of origin files.

Resource File Pack URIs

Resource files are configured as MSBuild `Resource` items and are compiled into assemblies. WPF supports the construction of pack URIs that can be used to identify resource files that are either compiled into the local assembly or compiled into an assembly that is referenced from the local assembly.

Local Assembly Resource File

The pack URI for a resource file that is compiled into the local assembly uses the following authority and path:

- **Authority**: application:///.
- **Path**: The name of the resource file, including its path, relative to the local assembly project folder root.

The following example shows the pack URI for a XAML resource file that is located in the root of the local assembly's project folder.

```
pack://application:,,,/ResourceFile.xaml
```

The following example shows the pack URI for a XAML resource file that is located in a subfolder of the local assembly's project folder.

```
pack://application:,,,/Subfolder/ResourceFile.xaml
```

Referenced Assembly Resource File

The pack URI for a resource file that is compiled into a referenced assembly uses the following authority and path:

- **Authority**: application:///.
- **Path**: The name of a resource file that is compiled into a referenced assembly. The path must conform to the following format:

  AssemblyShortName{*;Version*}[*;PublicKey*];component/*Path*

  - **AssemblyShortName**: the short name for the referenced assembly.
  - **;Version** [optional]: the version of the referenced assembly that contains the resource file. This is used when two or more referenced assemblies with the same short name are loaded.
  - **;PublicKey** [optional]: the public key that was used to sign the referenced assembly. This is used when two or more referenced assemblies with the same short name are loaded.
  - **;component**: specifies that the assembly being referred to is referenced from the local assembly.
  - **/Path**: the name of the resource file, including its path, relative to the root of the referenced assembly's project folder.

The following example shows the pack URI for a XAML resource file that is located in the root of the referenced assembly's project folder.

```
pack://application:,,,/ReferencedAssembly;component/ResourceFile.xaml
```

The following example shows the pack URI for a XAML resource file that is located in a subfolder of the referenced assembly's project folder.

```
pack://application:,,,/ReferencedAssembly;component/Subfolder/ResourceFile.xaml
```

The following example shows the pack URI for a XAML resource file that is located in the root folder of a referenced, version-specific assembly's project folder.

```
pack://application:,,,/ReferencedAssembly;v1.0.0.1;component/ResourceFile.xaml
```

Note that the pack URI syntax for referenced assembly resource files can be used only with the application:/// authority. For example, the following is not supported in WPF.

```
pack://siteoforigin:,,,/SomeAssembly;component/ResourceFile.xaml
```

Content File Pack URIs

The pack URI for a content file uses the following authority and path:

- **Authority**: application:///.
- **Path**: The name of the content file, including its path relative to the file system location of the application's main executable assembly.

The following example shows the pack URI for a XAML content file, located in the same folder as the executable assembly.

```
pack://application:,,,/ContentFile.xaml
```

The following example shows the pack URI for a XAML content file, located in a subfolder that is relative to the application's executable assembly.

```
pack://application:,,,/Subfolder/ContentFile.xaml
```

NOTE

HTML content files cannot be navigated to. The URI scheme only supports navigation to HTML files that are located at the site of origin.

Site of Origin Pack URIs

The pack URI for a site of origin file uses the following authority and path:

- **Authority**: siteoforigin:///.
- **Path**: The name of the site of origin file, including its path relative to the location from which the executable assembly was launched.

The following example shows the pack URI for a XAML site of origin file, stored in the location from which the executable assembly is launched.

```
pack://siteoforigin:,,,/SiteOfOriginFile.xaml
```

The following example shows the pack URI for a XAML site of origin file, stored in subfolder that is relative to the location from which the application's executable assembly is launched.

```
pack://siteoforigin:,,,/Subfolder/SiteOfOriginFile.xaml
```

Page Files

XAML files that are configured as MSBuild `Page` items are compiled into assemblies in the same way as resource files. Consequently, MSBuild `Page` items can be identified using pack URIs for resource files.

The types of XAML files that are commonly configured as MSBuild `Page` items have one of the following as their root element:

- System.Windows.Window
- System.Windows.Controls.Page
- System.Windows.Navigation.PageFunction<T>
- System.Windows.ResourceDictionary

- System.Windows.Documents.FlowDocument
- System.Windows.Controls.UserControl

Absolute vs. Relative Pack URIs

A fully qualified pack URI includes the scheme, the authority, and the path, and it is considered an absolute pack URI. As a simplification for developers, XAML elements typically allow you to set appropriate attributes with a relative pack URI, which includes only the path.

For example, consider the following absolute pack URI for a resource file in the local assembly.

`pack://application:,,,/ResourceFile.xaml`

The relative pack URI that refers to this resource file would be the following.

`/ResourceFile.xaml`

NOTE

Because site of origin files are not associated with assemblies, they can only be referred to with absolute pack URIs.

By default, a relative pack URI is considered relative to the location of the markup or code that contains the reference. If a leading backslash is used, however, the relative pack URI reference is then considered relative to the root of the application. For example, consider the following project structure.

`App.xaml`

`Page2.xaml`

`\SubFolder`

`+ Page1.xaml`

`+ Page2.xaml`

If Page1.xaml contains a URI that references *Root*\SubFolder\Page2.xaml, the reference can use the following relative pack URI.

`Page2.xaml`

If Page1.xaml contains a URI that references *Root*\Page2.xaml, the reference can use the following relative pack URI.

`/Page2.xaml`

Pack URI Resolution

The format of pack URIs makes it possible for pack URIs for different types of files to look the same. For example, consider the following absolute pack URI.

`pack://application:,,,/ResourceOrContentFile.xaml`

This absolute pack URI could refer to either a resource file in the local assembly or a content file. The same is true for the following relative URI.

`/ResourceOrContentFile.xaml`

In order to determine the type of file that a pack URI refers to, WPF resolves URIs for resource files in local assemblies and content files by using the following heuristics:

1. Probe the assembly metadata for an AssemblyAssociatedContentFileAttribute attribute that matches the pack URI.
2. If the AssemblyAssociatedContentFileAttribute attribute is found, the path of the pack URI refers to a content file.
3. If the AssemblyAssociatedContentFileAttribute attribute is not found, probe the set resource files that are compiled into the local assembly.
4. If a resource file that matches the path of the pack URI is found, the path of the pack URI refers to a resource file.
5. If the resource is not found, the internally created Uri is invalid.

URI resolution does not apply for URIs that refer to the following:

- Content files in referenced assemblies: these file types are not supported by WPF.
- Embedded files in referenced assemblies: URIs that identify them are unique because they include both the name of the referenced

assembly and the `;component` suffix.

- Site of origin files: URIs that identify them are unique because they are the only files that can be identified by pack URIs that contain the siteoforigin:/// authority.

One simplification that pack URI resolution allows is for code to be somewhat independent of the locations of resource and content files. For example, if you have a resource file in the local assembly that is reconfigured to be a content file, the pack URI for the resource remains the same, as does the code that uses the pack URI.

Programming with Pack URIs

Many WPF classes implement properties that can be set with pack URIs, including:

- Application.StartupUri
- Frame.Source
- NavigationWindow.Source
- Hyperlink.NavigateUri
- Window.Icon
- Image.Source

These properties can be set from both markup and code. This section demonstrates the basic constructions for both and then shows examples of common scenarios.

Using Pack URIs in Markup

A pack URI is specified in markup by setting the element of an attribute with the pack URI. For example:

`<element attribute="pack://application:,,,/File.xaml" />`

Table 1 illustrates the various absolute pack URIs that you can specify in markup.

Table 1: Absolute Pack URIs in Markup

FILE	ABSOLUTE PACK URI
Resource file - local assembly	`"pack://application:,,,/ResourceFile.xaml"`
Resource file in subfolder - local assembly	`"pack://application:,,,/Subfolder/ResourceFile.xaml"`
Resource file - referenced assembly	`"pack://application:,,,/ReferencedAssembly;component/ResourceFile.xaml"`
Resource file in subfolder of referenced assembly	`"pack://application:,,,/ReferencedAssembly;component/Subfolder/ResourceFil`
Resource file in versioned referenced assembly	`"pack://application:,,,/ReferencedAssembly;v1.0.0.0;component/ResourceFile`
Content file	`"pack://application:,,,/ContentFile.xaml"`
Content file in subfolder	`"pack://application:,,,/Subfolder/ContentFile.xaml"`
Site of origin file	`"pack://siteoforigin:,,,/SOOFile.xaml"`
Site of origin file in subfolder	`"pack://siteoforigin:,,,/Subfolder/SOOFile.xaml"`

Table 2 illustrates the various relative pack URIs that you can specify in markup.

Table 2: Relative Pack URIs in Markup

FILE	RELATIVE PACK URI
Resource file in local assembly	`"/ResourceFile.xaml"`
Resource file in subfolder of local assembly	`"/Subfolder/ResourceFile.xaml"`
Resource file in referenced assembly	`"/ReferencedAssembly;component/ResourceFile.xaml"`

FILE	RELATIVE PACK URI
Resource file in subfolder of referenced assembly	`"/ReferencedAssembly;component/Subfolder/ResourceFile.xaml"`
Content file	`"/ContentFile.xaml"`
Content file in subfolder	`"/Subfolder/ContentFile.xaml"`

Using Pack URIs in Code

You specify a pack URI in code by instantiating the Uri class and passing the pack URI as a parameter to the constructor. This is demonstrated in the following example.

```
Uri uri = new Uri("pack://application:,,,/File.xaml");
```

By default, the Uri class considers pack URIs to be absolute. Consequently, an exception is raised when an instance of the Uri class is created with a relative pack URI.

```
Uri uri = new Uri("/File.xaml");
```

Fortunately, the Uri(String, UriKind) overload of the Uri class constructor accepts a parameter of type UriKind to allow you to specify whether a pack URI is either absolute or relative.

```
// Absolute URI (default)
Uri absoluteUri = new Uri("pack://application:,,,/File.xaml", UriKind.Absolute);
// Relative URI
Uri relativeUri = new Uri("/File.xaml",
                        UriKind.Relative);
```

You should specify only Absolute or Relative when you are certain that the provided pack URI is one or the other. If you don't know the type of pack URI that is used, such as when a user enters a pack URI at run time, use RelativeOrAbsolute instead.

```
// Relative or Absolute URI provided by user via a text box
TextBox userProvidedUriTextBox = new TextBox();
Uri uri = new Uri(userProvidedUriTextBox.Text, UriKind.RelativeOrAbsolute);
```

Table 3 illustrates the various relative pack URIs that you can specify in code by using System.Uri.

Table 3: Absolute Pack URIs in Code

FILE	ABSOLUTE PACK URI
Resource file - local assembly	`Uri uri = new Uri("pack://application:,,,/ResourceFile.xaml", UriKind.Absolute);`
Resource file in subfolder - local assembly	`Uri uri = new Uri("pack://application:,,,/Subfolder/ResourceFile.xaml", UriKind.Absolute);`
Resource file - referenced assembly	`Uri uri = new Uri("pack://application:,,,/ReferencedAssembly;component/ResourceFile.xaml UriKind.Absolute);`
Resource file in subfolder of referenced assembly	`Uri uri = new Uri("pack://application:,,,/ReferencedAssembly;component/Subfolder/Resourc UriKind.Absolute);`
Resource file in versioned referenced assembly	`Uri uri = new Uri("pack://application:,,,/ReferencedAssembly;v1.0.0.0;component/Resource UriKind.Absolute);`
Content file	`Uri uri = new Uri("pack://application:,,,/ContentFile.xaml", UriKind.Absolute);`
Content file in subfolder	`Uri uri = new Uri("pack://application:,,,/Subfolder/ContentFile.xaml", UriKind.Absolute);`

FILE	ABSOLUTE PACK URI
Site of origin file	`Uri uri = new Uri("pack://siteoforigin:,,,/SOOFile.xaml", UriKind.Absolute);`
Site of origin file in subfolder	`Uri uri = new Uri("pack://siteoforigin:,,,/Subfolder/SOOFile.xaml", UriKind.Absolute);`

Table 4 illustrates the various relative pack URIs that you can specify in code using System.Uri.

Table 4: Relative Pack URIs in Code

FILE	RELATIVE PACK URI
Resource file - local assembly	`Uri uri = new Uri("/ResourceFile.xaml", UriKind.Relative);`
Resource file in subfolder - local assembly	`Uri uri = new Uri("/Subfolder/ResourceFile.xaml", UriKind.Relative);`
Resource file - referenced assembly	`Uri uri = new Uri("/ReferencedAssembly;component/ResourceFile.xaml", UriKind.Relative);`
Resource file in subfolder - referenced assembly	`Uri uri = new Uri("/ReferencedAssembly;component/Subfolder/ResourceFile.xaml", UriKind.Relative);`
Content file	`Uri uri = new Uri("/ContentFile.xaml", UriKind.Relative);`
Content file in subfolder	`Uri uri = new Uri("/Subfolder/ContentFile.xaml", UriKind.Relative);`

Common Pack URI Scenarios

The preceding sections have discussed how to construct pack URIs to identify resource, content, and site of origin files. In WPF, these constructions are used in a variety of ways, and the following sections cover several common usages.

Specifying the UI to Show When an Application Starts

StartupUri specifies the first UI to show when a WPF application is launched. For standalone applications, the UI can be a window, as shown in the following example.

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

Standalone applications and XAML browser applications (XBAPs) can also specify a page as the initial UI, as shown in the following example.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    StartupUri="HomePage.xaml" />
```

If the application is a standalone application and a page is specified with StartupUri, WPF opens a NavigationWindow to host the page. For XBAPs, the page is shown in the host browser.

Navigating to a Page

The following example shows how to navigate to a page.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  WindowTitle="Page With Hyperlink"
  WindowWidth="250"
  WindowHeight="250">
```

```
<Hyperlink NavigateUri="UriOfPageToNavigateTo.xaml">
  Navigate to Another Page
</Hyperlink>
```

```
</Page>
```

For more information on the various ways to navigate in WPF, see Navigation Overview.

Specifying a Window Icon

The following example shows how to use a URI to specify a window's icon.

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.MainWindow"
    Icon="WPFIcon1.ico">
</Window>
```

For more information, see Icon.

Loading Image, Audio, and Video Files

WPF allows applications to use a wide variety of media types, all of which can be identified and loaded with pack URIs, as shown in the following examples.

```
<MediaElement Stretch="Fill" LoadedBehavior="Play" Source="pack://siteoforigin:,,,/Media/bee.wmv" />
```

```
<MediaElement Stretch="Fill" LoadedBehavior="Play" Source="pack://siteoforigin:,,,/Media/ringin.wav" />
```

```
<Image Source="Images/Watermark.png" />
```

For more information on working with media content, see Graphics and Multimedia.

Loading a Resource Dictionary from the Site of Origin

Resource dictionaries (ResourceDictionary) can be used to support application themes. One way to create and manage themes is to create multiple themes as resource dictionaries that are located at an application's site of origin. This allows themes to be added and updated without recompiling and redeploying an application. These resource dictionaries can be identified and loaded using pack URIs, which is shown in the following example.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    StartupUri="HomePage.xaml">
  <Application.Resources>
    <ResourceDictionary Source="pack://siteoforigin:,,,/PageTheme.xaml" />
  </Application.Resources>
</Application>
```

For an overview of themes in WPF, see Styling and Templating.

See also

- WPF Application Resource, Content, and Data Files

How to: Add a Splash Screen to a WPF Application

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

This topic shows how to add a startup window, or *splash screen*, to a Windows Presentation Foundation (WPF) application.

To add an existing image as a splash screen

1. Create or find an image that you want to use for the splash screen. You can use any image format that is supported by the Windows Imaging Component (WIC). For example, you can use the BMP, GIF, JPEG, PNG, or TIFF format.

2. Add the image file to the WPF Application project.

3. In **Solution Explorer**, select the image.

4. In the Properties window, click the drop-down arrow for the **Build Action** property.

5. Select **SplashScreen** from the drop-down list.

6. Press **F5** to build and run the application.

   The splash screen image appears in the center of the screen, and then fades when the main application window appears.

To exclude the splash screen from build

1. In **Solution Explorer**, select the splash screen image.

2. In the **Properties** window, set the **Build Action** to **None**.

To remove the splash screen from an application

In **Solution Explorer**, delete the splash screen image.

See also

- SplashScreen
- How to: Add Existing Items to a Project

How to: Use an Application-Scope Resource Dictionary

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

This example shows how to define and use an application-scope custom resource dictionary.

Example

Application exposes an application-scope store for shared resources: Resources. By default, the Resources property is initialized with an instance of the ResourceDictionary type. You use this instance when you get and set application-scope properties using Resources. For more information, see How to: Get and Set an Application-Scope Resource.

If you have multiple resources that you set using Resources, you can instead use a custom resource dictionary to store those resources and set Resources with it instead. The following shows how you declare a custom resource dictionary using XAML.

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
    <SolidColorBrush x:Key="StandardSolidColorBrush" Color="Blue" />
    <LinearGradientBrush x:Key="StandardLinearGradientBrush" StartPoint="0.0,0.0" EndPoint="1.0,1.0">
        <LinearGradientBrush.GradientStops>
            <GradientStop Color="White" Offset="0" />
            <GradientStop Color="Black" Offset="1" />
        </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
</ResourceDictionary>
```

Swapping entire resource dictionaries using Resources allows you to support application-scope themes, where each theme is encapsulated by a single resource dictionary. The following example shows how to set the ResourceDictionary.

```
<!--Set the Application ResourceDictionary-->
<Application.Resources>
    <ResourceDictionary Source="MyResourceDictionary.xaml" />
</Application.Resources>
```

The following shows how you can get application-scope resources from the resource dictionary exposed by Resources in XAML.

```
<!--Set the brush as a StaticResource from the ResourceDictionary-->
<Rectangle Name="Rect" Height="200" Width="100" Fill="{StaticResource ResourceKey=StandardSolidColorBrush}" />
```

The following shows how you can also get the resources in code.

```
//Get a resource from the ResourceDictionary in code
Brush gradientBrush = (Brush)Application.Current.FindResource("StandardLinearGradientBrush");
```

```
'Get a resource from the ResourceDictionary in code
Dim GradientBrush As Brush = Application.Current.FindResource("StandardLinearGradientBrush")
```

There are two considerations to make when using Resources. First, the dictionary *key* is an object, so you must use exactly the same object instance when both setting and getting a property value. (Note that the key is case-sensitive when using a string.) Second, the dictionary *value* is an object, so you will have to convert the value to the desired type when getting a property value.

See also

- ResourceDictionary
- Resources
- XAML Resources
- Merged Resource Dictionaries

How to: Persist and Restore Application-Scope Properties Across Application Sessions

1/29/2019 • 2 minutes to read • Edit Online

This example shows how to persist application-scope properties when an application shuts down, and how to restore application-scope properties when an application is next launch.

Example

The application persists application-scope properties to, and restores them from, isolated storage. Isolated storage is a protected storage area that can safely be used by applications without file access permission. The *App.xaml* file defines the `App_Startup` method as the handler for the Application.Startup event, and the `App_Exit` method as the handler for the Application.Exit event, as shown in the highlighted lines of the first example. The second example shows a portion of the *App.xaml.cs* and *App.xaml.vb* files that highlights the code for the `App_Startup` method, which restores application-scope properties, and the `App_Exit` method, which saves application-scope properties.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.App"
    StartupUri="MainWindow.xaml"
    Startup="App_Startup"
    Exit="App_Exit">
    <Application.Resources>
        <SolidColorBrush x:Key="ApplicationScopeResource" Color="White"></SolidColorBrush>
    </Application.Resources>
</Application>
```

```csharp
using System.Windows;
using System.IO;
using System.IO.IsolatedStorage;

namespace SDKSample
{
    public partial class App : Application
    {
        string filename = "App.txt";

        public App()
        {
            // Initialize application-scope property
            this.Properties["NumberOfAppSessions"] = 0;
        }

        private void App_Startup(object sender, StartupEventArgs e)
        {
            // Restore application-scope property from isolated storage
            IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForDomain();
            try
            {
                using (IsolatedStorageFileStream stream = new IsolatedStorageFileStream(filename,
FileMode.Open, storage))
                using (StreamReader reader = new StreamReader(stream))
                {
                    // Restore each application-scope property individually
                    while (!reader.EndOfStream)
                    {
                        string[] keyValue = reader.ReadLine().Split(new char[] {','});
                        this.Properties[keyValue[0]] = keyValue[1];
                    }
                }
            }
            catch (FileNotFoundException ex)
            {
                // Handle when file is not found in isolated storage:
                // * When the first application session
                // * When file has been deleted
            }
        }

        private void App_Exit(object sender, ExitEventArgs e)
        {
            // Persist application-scope property to isolated storage
            IsolatedStorageFile storage = IsolatedStorageFile.GetUserStoreForDomain();
            using (IsolatedStorageFileStream stream = new IsolatedStorageFileStream(filename, FileMode.Create,
storage))
            using (StreamWriter writer = new StreamWriter(stream))
            {
                // Persist each application-scope property individually
                foreach (string key in this.Properties.Keys)
                {
                    writer.WriteLine("{0},{1}", key, this.Properties[key]);
                }
            }
        }
    }
}
```

Windows in WPF Applications

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

Users interact with applications through windows. The fundamental purpose of a window is to host and display content. The type of content that a window hosts depends on the type of data that an application operates over, which can include media, Extensible Application Markup Language (XAML) pages, Web pages, documents, database tables and records, and system information.

In This Section

WPF Windows Overview
Dialog Boxes Overview
How-to Topics

Reference

Window

NavigationWindow

Related Sections

Application Management Overview
Navigation Overview
Hosting
Build and Deploy

WPF Windows Overview

1/23/2019 • 21 minutes to read • Edit Online

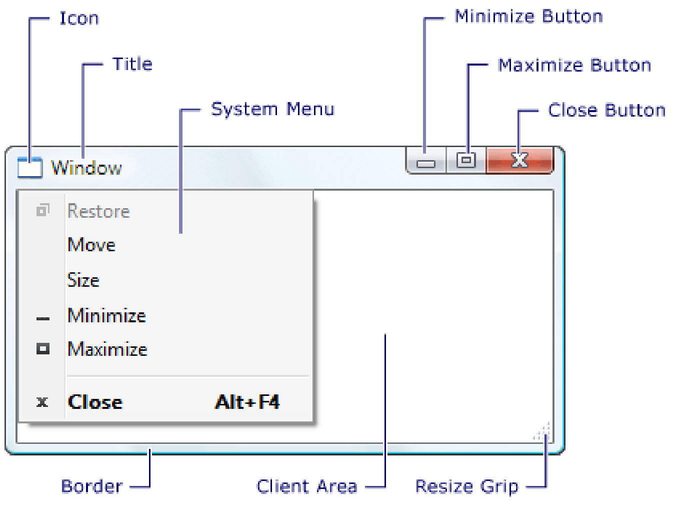
Users interact with Windows Presentation Foundation (WPF) standalone applications through windows. The primary purpose of a window is to host content that visualizes data and enables users to interact with data. Standalone WPF applications provide their own windows by using the Window class. This topic introduces Window before covering the fundamentals of creating and managing windows in standalone applications.

NOTE

Browser-hosted WPF applications, including XAML browser applications (XBAPs) and loose Extensible Application Markup Language (XAML) pages, don't provide their own windows. Instead, they are hosted in windows provided by Windows Internet Explorer. See WPF XAML Browser Applications Overview.

The Window Class

The following figure illustrates the constituent parts of a window.



A window is divided into two areas: the non-client area and client area.

The *non-client area* of a window is implemented by WPF and includes the parts of a window that are common to most windows, including the following:

- A border.
- A title bar.
- An icon.
- Minimize, Maximize, and Restore buttons.
- A Close button.
- A System menu with menu items that allow users to minimize, maximize, restore, move, resize, and close a window.

The *client area* of a window is the area within a window's non-client area and is used by developers to add application-specific content, such as menu bars, tool bars, and controls.

In WPF, a window is encapsulated by the Window class that you use to do the following:

- Display a window.
- Configure the size, position, and appearance of a window.
- Host application-specific content.
- Manage the lifetime of a window.

Implementing a Window

The implementation of a typical window comprises both appearance and behavior, where *appearance* defines how a window looks to users and *behavior* defines the way a window functions as users interact with it. In WPF, you can implement the appearance and behavior of a window using either code or XAML markup.

In general, however, the appearance of a window is implemented using XAML markup, and its behavior is implemented using code-behind, as shown in the following example.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.MarkupAndCodeBehindWindow">

  <!-- Client area (for content) -->

</Window>
```

```
using System.Windows;

namespace SDKSample
{
    public partial class MarkupAndCodeBehindWindow : Window
    {
        public MarkupAndCodeBehindWindow()
        {
            InitializeComponent();
        }
    }
}
```

```
Imports System.Windows

Namespace SDKSample
    Partial Public Class MarkupAndCodeBehindWindow
        Inherits Window
        Public Sub New()
            InitializeComponent()
        End Sub
    End Class
End Namespace
```

To enable a XAML markup file and code-behind file to work together, the following are required:

- In markup, the `Window` element must include the `x:Class` attribute. When the application is built, the existence of `x:Class` in the markup file causes Microsoft build engine (MSBuild) to create a `partial` class that derives from Window and has the name that is specified by the `x:Class` attribute. This requires the addition of an XML namespace declaration for the XAML schema (`xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`). The generated `partial` class implements the

`InitializeComponent` method, which is called to register the events and set the properties that are implemented in markup.

- In code-behind, the class must be a `partial` class with the same name that is specified by the `x:Class` attribute in markup, and it must derive from Window. This allows the code-behind file to be associated with the `partial` class that is generated for the markup file when the application is built (see Building a WPF Application).
- In code-behind, the Window class must implement a constructor that calls the `InitializeComponent` method. `InitializeComponent` is implemented by the markup file's generated `partial` class to register events and set properties that are defined in markup.

> **NOTE**
>
> When you add a new Window to your project by using Microsoft Visual Studio, the Window is implemented using both markup and code-behind, and includes the necessary configuration to create the association between the markup and code-behind files as described here.

With this configuration in place, you can focus on defining the appearance of the window in XAML markup and implementing its behavior in code-behind. The following example shows a window with a button, implemented in XAML markup, and an event handler for the button's Click event, implemented in code-behind.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.MarkupAndCodeBehindWindow">
  <!-- Client area (for content) -->
  <Button Click="button_Click">Click This Button</Button>
</Window>
```

```
using System.Windows;

namespace SDKSample
{
    public partial class MarkupAndCodeBehindWindow : Window
    {
        public MarkupAndCodeBehindWindow()
        {
            InitializeComponent();
        }

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

```
Imports System.Windows

Namespace SDKSample
    Partial Public Class MarkupAndCodeBehindWindow
        Inherits Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub button_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            MessageBox.Show("Button was clicked.")
        End Sub
    End Class
End Namespace
```

Configuring a Window Definition for MSBuild

How you implement your window determines how it is configured for MSBuild. For a window that is defined using both XAML markup and code-behind:

- XAML markup files are configured as MSBuild `Page` items.
- Code-behind files are configured as MSBuild `Compile` items.

This is shown in the following MSBuild project file.

```
<Project ...
                xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    ...
    <Page Include="MarkupAndCodeBehindWindow.xaml" />
    <Compile Include=" MarkupAndCodeBehindWindow.xaml.cs" />
    ...
</Project>
```

For information about building WPF applications, see Building a WPF Application.

Window Lifetime

As with any class, a window has a lifetime that begins when it is first instantiated, after which it is opened, activated and deactivated, and eventually closed.

Opening a Window

To open a window, you first create an instance of it, which is demonstrated in the following example.

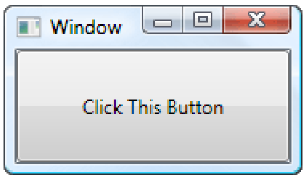
```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.App"
    Startup="app_Startup">
</Application>
```

```
using System.Windows;
namespace SDKSample
{
    public partial class App : Application
    {
        void app_Startup(object sender, StartupEventArgs e)
        {
            // Create a window
            MarkupAndCodeBehindWindow window = new MarkupAndCodeBehindWindow();

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

In this example, the `MarkupAndCodeBehindWindow` is instantiated when the application starts, which occurs when the Startup event is raised.

When a window is instantiated, a reference to it is automatically added to a list of windows that is managed by the Application object (see Application.Windows). Additionally, the first window to be instantiated is, by default, set by Application as the main application window (see Application.MainWindow).

The window is finally opened by calling the Show method; the result is shown in the following figure.



A window that is opened by calling Show is a modeless window, which means that the application operates in a mode that allows users to activate other windows in the same application.

NOTE

ShowDialog is called to open windows such as dialog boxes modally. See Dialog Boxes Overview for more information.

When Show is called, a window performs initialization work before it is shown to establish infrastructure that allows it to receive user input. When the window is initialized, the SourceInitialized event is raised and the window is shown.

As a shortcut, StartupUri can be set to specify the first window that is opened automatically when an application starts.

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

When the application starts, the window specified by the value of StartupUri is opened modelessly; internally, the window is opened by calling its Show method.

Window Ownership

A window that is opened by using the Show method does not have an implicit relationship with the window that created it; users can interact with either window independently of the other, which means that either window can do the following:

- Cover the other (unless one of the windows has its Topmost property set to `true`).
- Be minimized, maximized, and restored without affecting the other.

Some windows require a relationship with the window that opens them. For example, an Integrated Development Environment (IDE) application may open property windows and tool windows whose typical behavior is to cover the window that creates them. Furthermore, such windows should always close, minimize, maximize, and restore in concert with the window that created them. Such a relationship can be established by making one window *own* another, and is achieved by setting the Owner property of the *owned window* with a reference to the *owner window*. This is shown in the following example.

```
// Create a window and make this window its owner
Window ownedWindow = new Window();
ownedWindow.Owner = this;
ownedWindow.Show();
```

```
' Create a window and make this window its owner
Dim ownedWindow As New Window()
ownedWindow.Owner = Me
ownedWindow.Show()
```

After ownership is established:

- The owned window can reference its owner window by inspecting the value of its Owner property.
- The owner window can discover all the windows it owns by inspecting the value of its OwnedWindows property.

Preventing Window Activation

There are scenarios where windows should not be activated when shown, such as conversation windows of an Internet messenger-style application or notification windows of an email application.

If your application has a window that shouldn't be activated when shown, you can set its ShowActivated property to `false` before calling the Show method for the first time. As a consequence:

- The window is not activated.
- The window's Activated event is not raised.
- The currently activated window remains activated.

The window will become activated, however, as soon as the user activates it by clicking either the client or non-client area. In this case:

- The window is activated.
- The window's Activated event is raised.
- The previously activated window is deactivated.
- The window's Deactivated and Activated events are subsequently raised as expected in response to user actions.

Window Activation

When a window is first opened, it becomes the active window (unless it is shown with ShowActivated set to `false`). The *active window* is the window that is currently capturing user input, such as key strokes and mouse clicks. When a window becomes active, it raises the Activated event.

NOTE

When a window is first opened, the Loaded and ContentRendered events are raised only after the Activated event is raised. With this in mind, a window can effectively be considered opened when ContentRendered is raised.

After a window becomes active, a user can activate another window in the same application, or activate another application. When that happens, the currently active window becomes deactivated and raises the Deactivated event. Likewise, when the user selects a currently deactivated window, the window becomes active again and Activated is raised.

One common reason to handle Activated and Deactivated is to enable and disable functionality that can only run when a window is active. For example, some windows display interactive content that requires constant user input or attention, including games and video players. The following example is a simplified video player that demonstrates how to handle Activated and Deactivated to implement this behavior.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.CustomMediaPlayerWindow"
    Activated="window_Activated"
    Deactivated="window_Deactivated">

    <!-- Media Player -->
    <MediaElement
      Name="mediaElement"
      Stretch="Fill"
      LoadedBehavior="Manual"
      Source="numbers.wmv" />

</Window>
```

```
using System;
using System.Windows;

namespace SDKSample
{
    public partial class CustomMediaPlayerWindow : Window
    {
        public CustomMediaPlayerWindow()
        {
            InitializeComponent();
        }

        void window_Activated(object sender, EventArgs e)
        {
            // Recommence playing media if window is activated
            this.mediaElement.Play();
        }

        void window_Deactivated(object sender, EventArgs e)
        {
            // Pause playing if media is being played and window is deactivated
            this.mediaElement.Pause();
        }
    }
}
```

```
Imports System
Imports System.Windows

Namespace SDKSample
    Partial Public Class CustomMediaPlayerWindow
        Inherits Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub window_Activated(ByVal sender As Object, ByVal e As EventArgs)
            ' Recommence playing media if window is activated
            Me.mediaElement.Play()
        End Sub

        Private Sub window_Deactivated(ByVal sender As Object, ByVal e As EventArgs)
            ' Pause playing if media is being played and window is deactivated
            Me.mediaElement.Pause()
        End Sub
    End Class
End Namespace
```

Other types of applications may still run code in the background when a window is deactivated. For example, a mail client may continue polling the mail server while the user is using other applications. Applications like these often provide different or additional behavior while the main window is deactivated. With respect to the mail program, this may mean both adding the new mail item to the inbox and adding a notification icon to the system tray. A notification icon need only be displayed when the mail window isn't active, which can be determined by inspecting the IsActive property.

If a background task completes, a window may want to notify the user more urgently by calling Activate method. If the user is interacting with another application activated when Activate is called, the window's taskbar button flashes. If a user is interacting with the current application, calling Activate will bring the window to the foreground.

NOTE

You can handle application-scope activation using the Application.Activated and Application.Deactivated events.

Closing a Window

The life of a window starts coming to an end when a user closes it. A window can be closed by using elements in the non-client area, including the following:

- The **Close** item of the **System** menu.
- Pressing ALT+F4.
- Pressing the **Close** button.

You can provide additional mechanisms to the client area to close a window, the more common of which include the following:

- An **Exit** item in the **File** menu, typically for main application windows.
- A **Close** item in the **File** menu, typically on a secondary application window.
- A **Cancel** button, typically on a modal dialog box.
- A **Close** button, typically on a modeless dialog box.

To close a window in response to one of these custom mechanisms, you need to call the Close method. The following example implements the ability to close a window by choosing the **Exit** on the **File** menu.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.WindowWithFileExit">

  <Menu>
    <MenuItem Header="_File">
      <MenuItem Header="E_xit" Click="fileExitMenuItem_Click" />
    </MenuItem>
  </Menu>

</Window>
```

```
using System.Windows;

namespace SDKSample
{
    public partial class WindowWithFileExit : System.Windows.Window
    {
        public WindowWithFileExit()
        {
            InitializeComponent();
        }

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

```

Imports System.Windows

Namespace SDKSample
    Partial Public Class WindowWithFileExit
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub fileExitMenuItem_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            ' Close this window
            Me.Close()
        End Sub
    End Class
End Namespace
```

When a window closes, it raises two events: Closing and Closed.

Closing is raised before the window closes, and it provides a mechanism by which window closure can be prevented. One common reason to prevent window closure is if window content contains modified data. In this situation, the Closing event can be handled to determine whether data is dirty and, if so, to ask the user whether to either continue closing the window without saving the data or to cancel window closure. The following example shows the key aspects of handling Closing.

```
using System; // EventArgs
using System.ComponentModel; // CancelEventArgs
using System.Windows; // window

namespace CSharp
{
    public partial class DataWindow : Window
    {
        // Is data dirty
        bool isDataDirty = false;

        public DataWindow()
        {
            InitializeComponent();
        }

        void documentTextBox_TextChanged(object sender, EventArgs e)
        {
            this.isDataDirty = true;
        }

        void DataWindow_Closing(object sender, CancelEventArgs e)
        {
            MessageBox.Show("Closing called");

            // If data is dirty, notify user and ask for a response
            if (this.isDataDirty)
            {
                string msg = "Data is dirty. Close without saving?";
                MessageBoxResult result =
                  MessageBox.Show(
                    msg,
                    "Data App",
                    MessageBoxButton.YesNo,
                    MessageBoxImage.Warning);
                if (result == MessageBoxResult.No)
                {
                    // If user doesn't want to close, cancel closure
                    e.Cancel = true;
                }
            }
        }
    }
}
```

```
Imports System ' EventArgs
Imports System.ComponentModel ' CancelEventArgs
Imports System.Windows ' window

Namespace VisualBasic
    Partial Public Class DataWindow
        Inherits Window
        ' Is data dirty
        Private isDataDirty As Boolean = False

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub documentTextBox_TextChanged(ByVal sender As Object, ByVal e As EventArgs)
            Me.isDataDirty = True
        End Sub

        Private Sub DataWindow_Closing(ByVal sender As Object, ByVal e As CancelEventArgs)
            MessageBox.Show("Closing called")

            ' If data is dirty, notify user and ask for a response
            If Me.isDataDirty Then
                Dim msg As String = "Data is dirty. Close without saving?"
                Dim result As MessageBoxResult = MessageBox.Show(msg, "Data App", MessageBoxButton.YesNo,
MessageBoxImage.Warning)
                If result = MessageBoxResult.No Then
                    ' If user doesn't want to close, cancel closure
                    e.Cancel = True
                End If
            End If
        End Sub
    End Class
End Namespace
```

The Closing event handler is passed a CancelEventArgs, which implements the `Boolean` Cancel property that you set to `true` to prevent a window from closing.

If Closing is not handled, or it is handled but not canceled, the window will close. Just before a window actually closes, Closed is raised. At this point, a window cannot be prevented from closing.

NOTE

An application can be configured to shut down automatically when either the main application window closes (see MainWindow) or the last window closes. For details, see ShutdownMode.

While a window can be explicitly closed through mechanisms provided in the non-client and client areas, a window can also be implicitly closed as a result of behavior in other parts of the application or Windows, including the following:

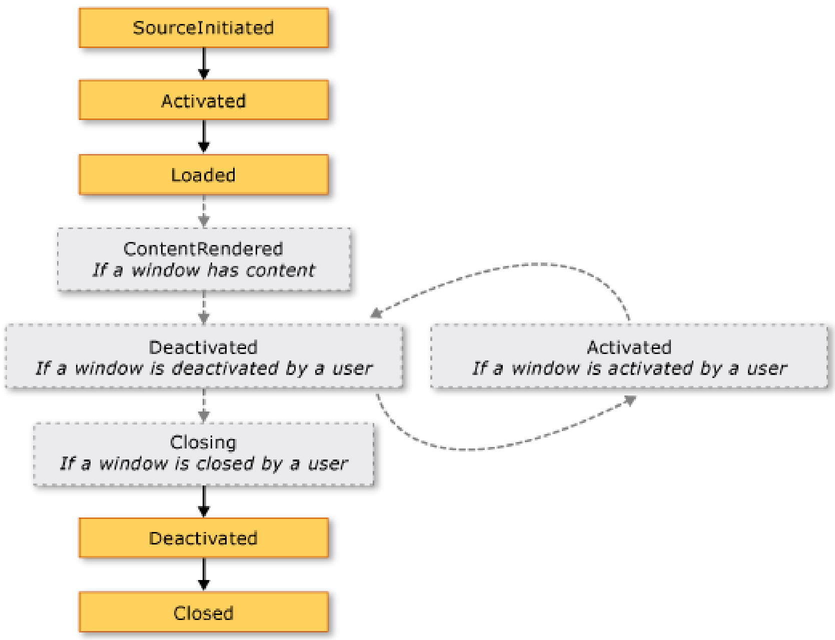
- A user logs off or shuts down Windows.
- A window's owner closes (see Owner).
- The main application window is closed and ShutdownMode is OnMainWindowClose.
- Shutdown is called.

NOTE

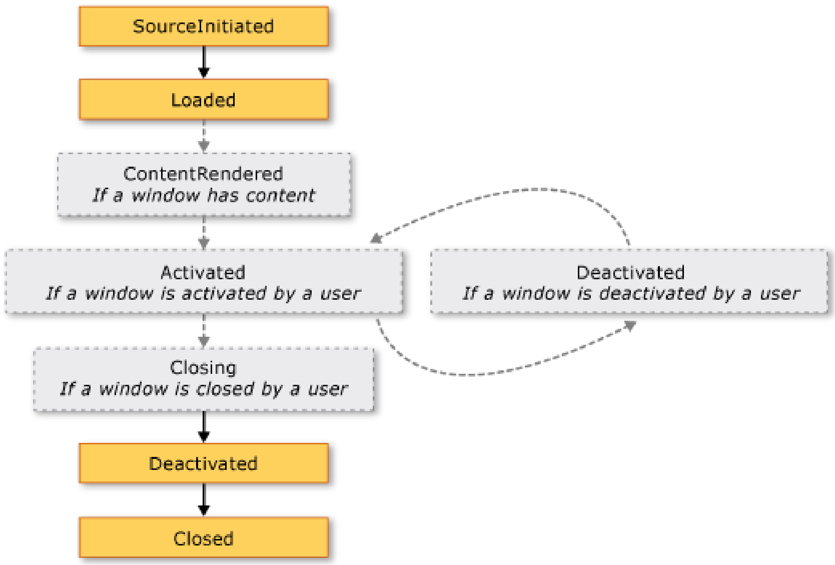
A window cannot be reopened after it is closed.

Window Lifetime Events

The following illustration shows the sequence of the principal events in the lifetime of a window.



The following illustration shows the sequence of the principal events in the lifetime of a window that is shown without activation (ShowActivated is set to `false` before the window is shown).



Window Location

While a window is open, it has a location in the x and y dimensions relative to the desktop. This location can be determined by inspecting the Left and Top properties, respectively. You can set these properties to change the location of the window.

You can also specify the initial location of a Window when it first appears by setting the WindowStartupLocation property with one of the following WindowStartupLocation enumeration values:

- CenterOwner (default)

- CenterScreen
- Manual

If the startup location is specified as Manual, and the Left and Top properties have not been set, Window will ask Windows for a location to appear in.

Topmost Windows and Z-Order

Besides having an x and y location, a window also has a location in the z dimension, which determines its vertical position with respect to other windows. This is known as the window's z-order, and there are two types: normal z-order and topmost z-order. The location of a window in the *normal z-order* is determined by whether it is currently active or not. By default, a window is located in the normal z-order. The location of a window in the *topmost z-order* is also determined by whether it is currently active or not. Furthermore, windows in the topmost z-order are always located above windows in the normal z-order. A window is located in the topmost z-order by setting its Topmost property to `true`.

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

Within each z-order, the currently active window appears above all other windows in the same z-order.

Window Size

Besides having a desktop location, a window has a size that is determined by several properties, including the various width and height properties and SizeToContent.

MinWidth, Width, and MaxWidth are used to manage the range of widths that a window can have during its lifetime, and are configured as shown in the following example.

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

Window height is managed by MinHeight, Height, and MaxHeight, and are configured as shown in the following example.

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

Because the various width values and height values each specify a range, it is possible for the width and height of a resizable window to be anywhere within the specified range for the respective dimension. To detect its current width and height, inspect ActualWidth and ActualHeight, respectively.

If you'd like the width and height of your window to have a size that fits to the size of the window's content, you can use the SizeToContent property, which has the following values:

- Manual. No effect (default).
- Width. Fit to content width, which has the same effect as setting both MinWidth and MaxWidth to the width of the content.

- Height. Fit to content height, which has the same effect as setting both MinHeight and MaxHeight to the height of the content.
- WidthAndHeight. Fit to content width and height, which has the same effect as setting both MinHeight and MaxHeight to the height of the content, and setting both MinWidth and MaxWidth to the width of the content.

The following example shows a window that automatically sizes to fit its content, both vertically and horizontally, when first shown.

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

The following example shows how to set the SizeToContent property in code to specify how a window resizes to fit its content .

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

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

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

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

```
' Manually alter window height and width
Me.SizeToContent = SizeToContent.Manual

' Automatically resize width relative to content
Me.SizeToContent = SizeToContent.Width

' Automatically resize height relative to content
Me.SizeToContent = SizeToContent.Height

' Automatically resize height and width relative to content
Me.SizeToContent = SizeToContent.WidthAndHeight
```

Order of Precedence for Sizing Properties

Essentially, the various sizes properties of a window combine to define the range of width and height for a resizable window. To ensure a valid range is maintained, Window evaluates the values of the size properties using the following orders of precedence.

For Height Properties:

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

For Width Properties:

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

The order of precedence can also determine the size of a window when it is maximized, which is managed with the WindowState property.

Window State

During the lifetime of a resizable window, it can have three states: normal, minimized, and maximized. A window with a *normal* state is the default state of a window. A window with this state allows a user to move and resize it by using a resize grip or the border, if it is resizable.

A window with a *minimized* state collapses to its task bar button if ShowInTaskbar is set to `true`; otherwise, it collapses to the smallest possible size it can be and relocates itself to the bottom-left corner of the desktop. Neither type of minimized window can be resized using a border or resize grip, although a minimized window that isn't shown in the task bar can be dragged around the desktop.

A window with a *maximized* state expands to the maximum size it can be, which will only be as large as its MaxWidth, MaxHeight, and SizeToContent properties dictate. Like a minimized window, a maximized window cannot be resized by using a resize grip or by dragging the border.

NOTE

The values of the Top, Left, Width, and Height properties of a window always represent the values for the normal state, even when the window is currently maximized or minimized.

The state of a window can be configured by setting its WindowState property, which can have one of the following WindowState enumeration values:

- Normal (default)
- Maximized
- Minimized

The following example shows how to create a window that is shown as maximized when it opens.

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

In general, you should set WindowState to configure the initial state of a window. Once a resizable window is shown, users can press the minimize, maximize, and restore buttons on the window's title bar to change the window state.

Window Appearance

You change the appearance of the client area of a window by adding window-specific content to it, such as buttons, labels, and text boxes. To configure the non-client area, Window provides several properties, which include Icon to set a window's icon and Title to set its title.

You can also change the appearance and behavior of non-client area border by configuring a window's resize mode, window style, and whether it appears as a button in the desktop task bar.

Resize Mode

Depending on the WindowStyle property, you can control how (and if) users can resize the window. The choice of window style affects whether a user can resize the window by dragging its border with the mouse, whether the **Minimize**, **Maximize**, and **Resize** buttons appear on the non-client area, and, if they do appear, whether they are enabled.

You can configure how a window resizes by setting its ResizeMode property, which can be one of the following ResizeMode enumeration values:

- NoResize
- CanMinimize
- CanResize (default)
- CanResizeWithGrip

As with WindowStyle, the resize mode of a window is unlikely to change during its lifetime, which means that you'll most likely set it from XAML markup.

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

Note that you can detect whether a window is maximized, minimized, or restored by inspecting the WindowState property.

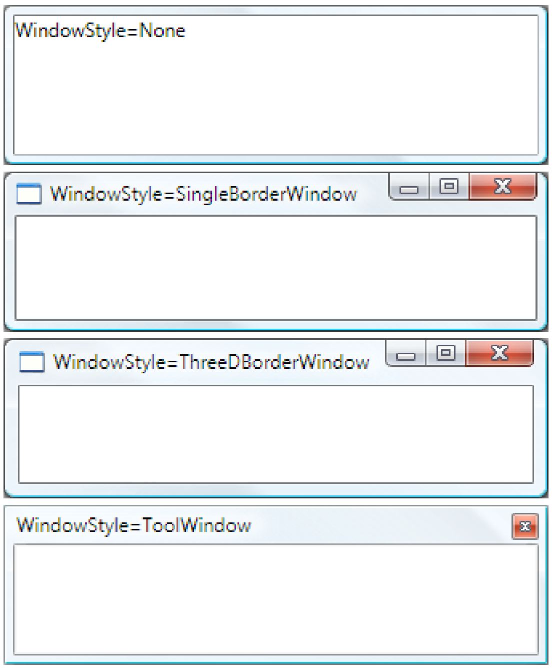
Window Style

The border that is exposed from the non-client area of a window is suitable for most applications. However, there are circumstances where different types of borders are needed, or no borders are needed at all, depending on the type of window.

To control what type of border a window gets, you set its WindowStyle property with one of the following values of the WindowStyle enumeration:

- None
- SingleBorderWindow (default)
- ThreeDBorderWindow
- ToolWindow

The effect of these window styles are illustrated in the following figure.


WindowStyle=None
WindowStyle=SingleBorderWindow
WindowStyle=ThreeDBorderWindow
WindowStyle=ToolWindow

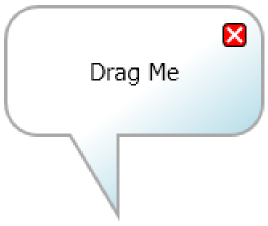
You can set WindowStyle using either XAML markup or code; because it is unlikely to change during the lifetime of a window, you will most likely configure it using XAML markup.

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

Non-Rectangular Window Style

There are also situations where the border styles that WindowStyle allows you to have are not sufficient. For example, you may want to create an application with a non-rectangular border, like Microsoft Windows Media Player uses.

For example, consider the speech bubble window shown in the following figure.



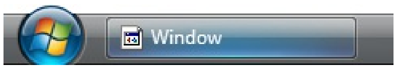
This type of window can be created by setting the WindowStyle property to None, and by using special support that Window has for transparency.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    WindowStyle="None"
    AllowsTransparency="True"
    Background="Transparent">
</Window>
```

This combination of values instructs the window to render completely transparent. In this state, the window's non-client area adornments (the Close menu, Minimize, Maximize, and Restore buttons, and so on) cannot be used. Consequently, you need to provide your own.

Task Bar Presence

The default appearance of a window includes a task bar button, like the one shown in the following figure.



Some types of windows don't have a task bar button, such as message boxes and dialog boxes (see Dialog Boxes Overview). You can control whether the task bar button for a window is shown by setting the ShowInTaskbar property (`true` by default).

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

Security Considerations

Window requires `UnmanagedCode` security permission to be instantiated. For applications installed on and launched from the local machine, this falls within the set of permissions that are granted to the application.

However, this falls outside the set of permissions granted to applications that are launched from the Internet or Local intranet zone using ClickOnce. Consequently, users will receive a ClickOnce security warning and will need to elevate the permission set for the application to full trust.

Additionally, XBAPs cannot show windows or dialog boxes by default. For a discussion on standalone application security considerations, see WPF Security Strategy - Platform Security.

Other Types of Windows

NavigationWindow is a window that is designed to host navigable content. For more information, see Navigation Overview).

Dialog boxes are windows that are often used to gather information from a user to complete a function. For example, when a user wants to open a file, the **Open File** dialog box is usually displayed by an application to get the file name from the user. For more information, see Dialog Boxes Overview.

See also

- Window
- MessageBox
- NavigationWindow
- Application
- Dialog Boxes Overview
- Building a WPF Application

Dialog Boxes Overview

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

Standalone applications typically have a main window that both displays the main data over which the application operates and exposes the functionality to process that data through user interface (UI) mechanisms like menu bars, tool bars, and status bars. A non-trivial application may also display additional windows to do the following:

- Display specific information to users.
- Gather information from users.
- Both display and gather information.

These types of windows are known as *dialog boxes*, and there are two types: modal and modeless.

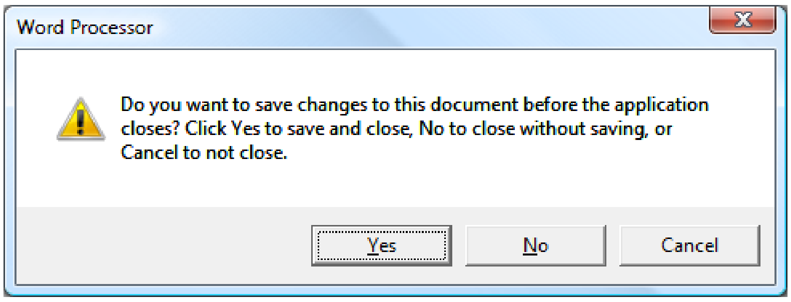
A *modal* dialog box is displayed by a function when the function needs additional data from a user to continue. Because the function depends on the modal dialog box to gather data, the modal dialog box also prevents a user from activating other windows in the application while it remains open. In most cases, a modal dialog box allows a user to signal when they have finished with the modal dialog box by pressing either an **OK** or **Cancel** button. Pressing the **OK** button indicates that a user has entered data and wants the function to continue processing with that data. Pressing the **Cancel** button indicates that a user wants to stop the function from executing altogether. The most common examples of modal dialog boxes are shown to open, save, and print data.

A *modeless* dialog box, on the other hand, does not prevent a user from activating other windows while it is open. For example, if a user wants to find occurrences of a particular word in a document, a main window will often open a dialog box to ask a user what word they are looking for. Since finding a word doesn't prevent a user from editing the document, however, the dialog box doesn't need to be modal. A modeless dialog box at least provides a **Close** button to close the dialog box, and may provide additional buttons to execute specific functions, such as a **Find Next** button to find the next word that matches the find criteria of a word search.

Windows Presentation Foundation (WPF) allows you to create several types of dialog boxes, including message boxes, common dialog boxes, and custom dialog boxes. This topic discusses each, and the Dialog Box Sample provides matching examples.

Message Boxes

A *message box* is a dialog box that can be used to display textual information and to allow users to make decisions with buttons. The following figure shows a message box that displays textual information, asks a question, and provides the user with three buttons to answer the question.



Do you want to save changes to this document before the application closes? Click Yes to save and close, No to close without saving, or Cancel to not close.

To create a message box, you use the MessageBox class. MessageBox lets you configure the message box text, title, icon, and buttons, using code like the following.

```csharp
// Configure the message box to be displayed
string messageBoxText = "Do you want to save changes?";
string caption = "Word Processor";
MessageBoxButton button = MessageBoxButton.YesNoCancel;
MessageBoxImage icon = MessageBoxImage.Warning;
```

```vb
' Configure the message box to be displayed
Dim messageBoxText As String = "Do you want to save changes?"
Dim caption As String = "Word Processor"
Dim button As MessageBoxButton = MessageBoxButton.YesNoCancel
Dim icon As MessageBoxImage = MessageBoxImage.Warning
```

To show a message box, you call the `static` Show method, as demonstrated in the following code.

```csharp
// Display message box
MessageBox.Show(messageBoxText, caption, button, icon);
```

```vb
' Display message box
MessageBox.Show(messageBoxText, caption, button, icon)
```

When code that shows a message box needs to detect and process the user's decision (which button was pressed), the code can inspect the message box result, as shown in the following code.

```csharp
// Display message box
MessageBoxResult result = MessageBox.Show(messageBoxText, caption, button, icon);

// Process message box results
switch (result)
{
    case MessageBoxResult.Yes:
        // User pressed Yes button
        // ...
        break;
    case MessageBoxResult.No:
        // User pressed No button
        // ...
        break;
    case MessageBoxResult.Cancel:
        // User pressed Cancel button
        // ...
        break;
}
```

```
' Display message box
Dim result As MessageBoxResult = MessageBox.Show(messageBoxText, caption, button, icon)

' Process message box results
Select Case result
    Case MessageBoxResult.Yes
        ' User pressed Yes button
        ' ...
    Case MessageBoxResult.No
        ' User pressed No button
        ' ...
    Case MessageBoxResult.Cancel
        ' User pressed Cancel button
        ' ...
End Select
```

For more information on using message boxes, see MessageBox, MessageBox Sample, and Dialog Box Sample.

Although MessageBox may offer a simple dialog box user experience, the advantage of using MessageBox is that is the only type of window that can be shown by applications that run within a partial trust security sandbox (see Security), such as XAML browser applications (XBAPs).

Most dialog boxes display and gather more complex data than the result of a message box, including text, selection (check boxes), mutually exclusive selection (radio buttons), and list selection (list boxes, combo boxes, drop-down list boxes). For these, Windows Presentation Foundation (WPF) provides several common dialog boxes and allows you to create your own dialog boxes, although the use of either is limited to applications running with full trust.

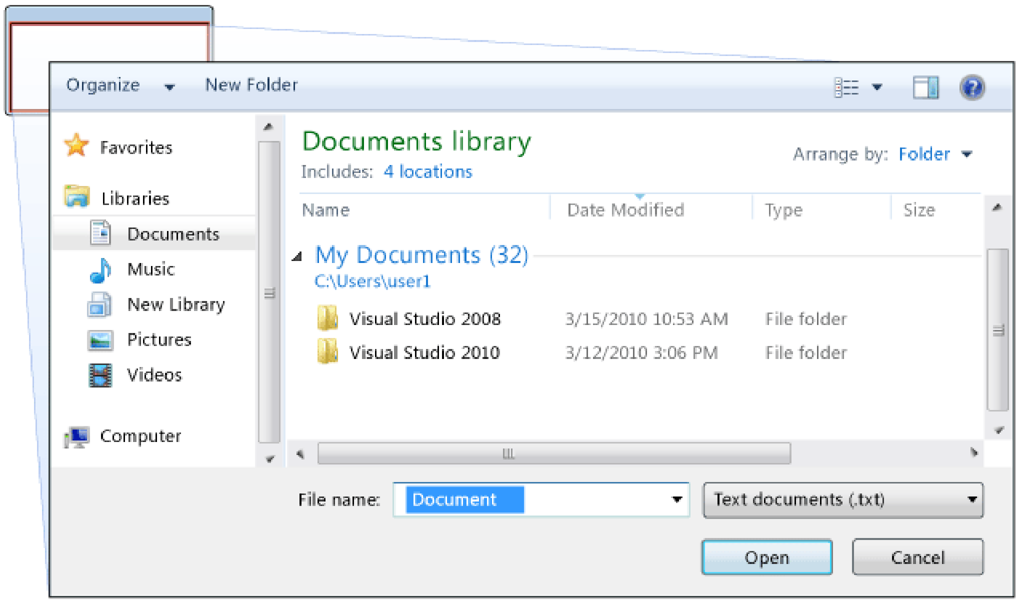
Common Dialog Boxes

Windows implements a variety of reusable dialog boxes that are common to all applications, including dialog boxes for opening files, saving files, and printing. Since these dialog boxes are implemented by the operating system, they can be shared among all the applications that run on the operating system, which helps user experience consistency; when users are familiar with the use of an operating system-provided dialog box in one application, they don't need to learn how to use that dialog box in other applications. Because these dialog boxes are available to all applications and because they help provide a consistent user experience, they are known as *common dialog boxes*.

Windows Presentation Foundation (WPF) encapsulates the open file, save file, and print common dialog boxes and exposes them as managed classes for you to use in standalone applications. This topic provides a brief overview of each.

Open File Dialog

The open file dialog box, shown in the following figure, is used by file opening functionality to retrieve the name of a file to open.



The common open file dialog box is implemented as the OpenFileDialog class and is located in the Microsoft.Win32 namespace. The following code shows how to create, configure, and show one, and how to process the result.

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

// Show open file dialog box
Nullable<bool> result = dlg.ShowDialog();

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

```
' Configure open file dialog box
Dim dlg As New Microsoft.Win32.OpenFileDialog()
dlg.FileName = "Document" ' Default file name
dlg.DefaultExt = ".txt" ' Default file extension
dlg.Filter = "Text documents (.txt)|*.txt" ' Filter files by extension

' Show open file dialog box
Dim result? As Boolean = dlg.ShowDialog()

' Process open file dialog box results
If result = True Then
    ' Open document
    Dim filename As String = dlg.FileName
End If
```

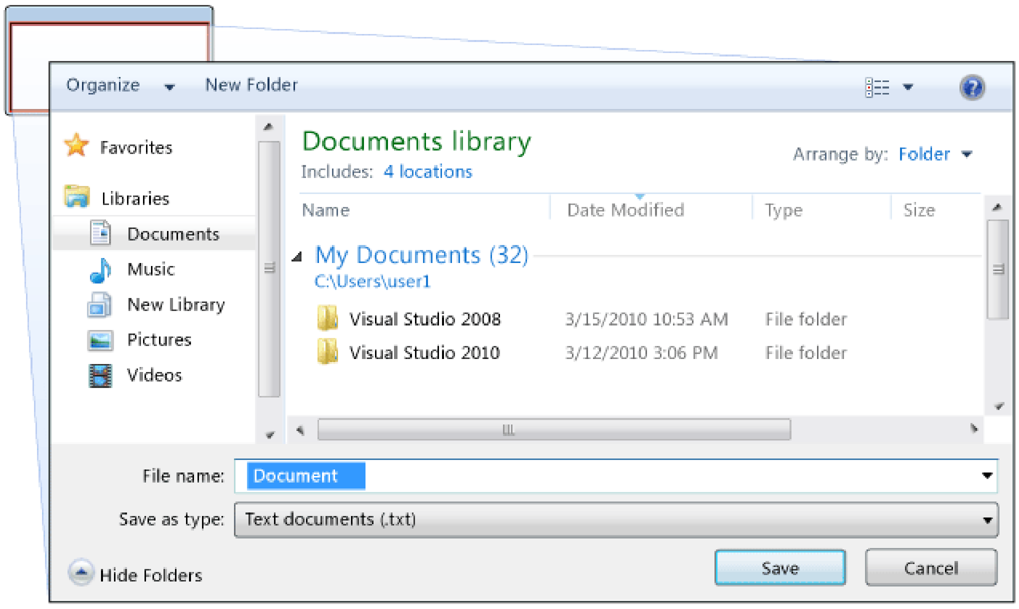
For more information on the open file dialog box, see Microsoft.Win32.OpenFileDialog.

NOTE

OpenFileDialog can be used to safely retrieve file names by applications running with partial trust (see Security).

Save File Dialog Box

The save file dialog box, shown in the following figure, is used by file saving functionality to retrieve the name of a file to save.



The common save file dialog box is implemented as the SaveFileDialog class, and is located in the Microsoft.Win32 namespace. The following code shows how to create, configure, and show one, and how to process the result.

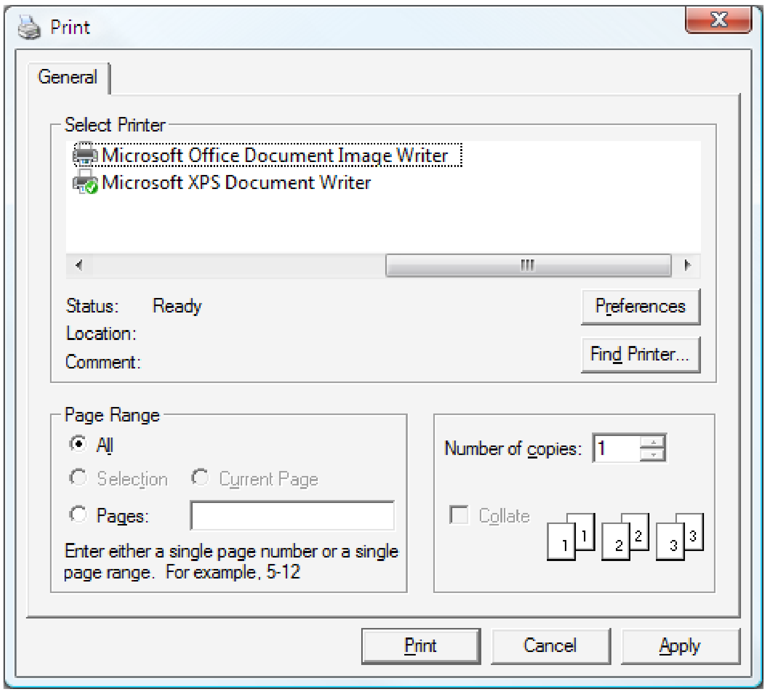
```
// Configure save file dialog box
Microsoft.Win32.SaveFileDialog dlg = new Microsoft.Win32.SaveFileDialog();
dlg.FileName = "Document"; // Default file name
dlg.DefaultExt = ".txt"; // Default file extension
dlg.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension

// Show save file dialog box
Nullable<bool> result = dlg.ShowDialog();

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

```
' Configure save file dialog box
Dim dlg As New Microsoft.Win32.SaveFileDialog()
dlg.FileName = "Document" ' Default file name
dlg.DefaultExt = ".txt" ' Default file extension
dlg.Filter = "Text documents (.txt)|*.txt" ' Filter files by extension

' Show save file dialog box
Dim result? As Boolean = dlg.ShowDialog()

' Process save file dialog box results
If result = True Then
    ' Save document
    Dim filename As String = dlg.FileName
End If
```

For more information on the save file dialog box, see Microsoft.Win32.SaveFileDialog.

Print Dialog Box

The print dialog box, shown in the following figure, is used by printing functionality to choose and configure the printer that a user would like to print data to.



The common print dialog box is implemented as the PrintDialog class, and is located in the System.Windows.Controls namespace. The following code shows how to create, configure, and show one.

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

// Show save file dialog box
Nullable<bool> result = dlg.ShowDialog();

// Process save file dialog box results
if (result == true)
{
    // Print document
}
```

```
' Configure printer dialog box
Dim dlg As New PrintDialog()
dlg.PageRangeSelection = PageRangeSelection.AllPages
dlg.UserPageRangeEnabled = True

' Show save file dialog box
Dim result? As Boolean = dlg.ShowDialog()

' Process save file dialog box results
If result = True Then
    ' Print document
End If
```

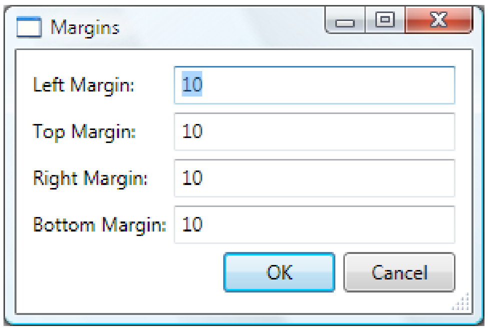
For more information on the print dialog box, see System.Windows.Controls.PrintDialog. For detailed discussion of printing in WPF, see Printing Overview.

Custom Dialog Boxes

While common dialog boxes are useful, and should be used when possible, they do not support the requirements of domain-specific dialog boxes. In these cases, you need to create your own dialog boxes. As we'll see, a dialog box is a window with special behaviors. Window implements those behaviors and, consequently, you use Window to create custom modal and modeless dialog boxes.

Creating a Modal Custom Dialog Box

This topic shows how to use Window to create a typical modal dialog box implementation, using the `Margins` dialog box as an example (see Dialog Box Sample). The `Margins` dialog box is shown in the following figure.



Configuring a Modal Dialog Box

The user interface for a typical dialog box includes the following:

- The various controls that are required to gather the desired data.
- Showing an **OK** button that users click to close the dialog box, return to the function, and continue processing.
- Showing a **Cancel** button that users click to close the dialog box and stop the function from further processing.
- Showing a **Close** button in the title bar.
- Showing an icon.
- Showing **Minimize**, **Maximize**, and **Restore** buttons.
- Showing a **System** menu to minimize, maximize, restore, and close the dialog box.
- Opening above and in the center of the window that opened the dialog box.
- Dialog boxes should be resizable where possible so, to prevent the dialog box from being too small, and to provide the user with a useful default size, you need to set both default and a minimum dimensions respectively.
- Pressing the ESC key should be configured as a keyboard shortcut that causes the **Cancel** button to be pressed. This is achieved by setting the IsCancel property of the **Cancel** button to `true`.
- Pressing the ENTER (or RETURN) key should be configured as a keyboard shortcut that causes the **OK** button to be pressed. This is achieved by setting the IsDefault property of the **OK** button `true`.

The following code demonstrates this configuration.

```xml
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.MarginsDialogBox"
    xmlns:local="clr-namespace:SDKSample"
    Title="Margins"
    Height="190"
    Width="300"
    MinHeight="10"
    MinWidth="300"
    ResizeMode="CanResizeWithGrip"
    ShowInTaskbar="False"
    WindowStartupLocation="CenterOwner"
    FocusManager.FocusedElement="{Binding ElementName=leftMarginTextBox}">

  <Grid>
```

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

  </Grid >

</Window>
```

```csharp
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;

namespace SDKSample
{
    public partial class MarginsDialogBox : Window
    {
        public MarginsDialogBox()
        {
            InitializeComponent();
        }
```

```vb
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input

Namespace SDKSample


Public Class MarginsDialogBox
    Inherits Window
    Public Sub New()
        Me.InitializeComponent()
    End Sub
```

```csharp
    }
}
```

```
End Class

End Namespace
```

The user experience for a dialog box also extends into the menu bar of the window that opens the dialog box. When a menu item runs a function that requires user interaction through a dialog box before the function can continue, the menu item for the function will have an ellipsis in its header, as shown here.

```
<!--Main Window-->
```

```
<MenuItem Name="formatMarginsMenuItem" Header="_Margins..." Click="formatMarginsMenuItem_Click" />
```

When a menu item runs a function that displays a dialog box which does not require user interaction, such as an About dialog box, an ellipsis is not required.

Opening a Modal Dialog Box

A dialog box is typically shown as a result of a user selecting a menu item to perform a domain-specific function, such as setting the margins of a document in a word processor. Showing a window as a dialog box is similar to showing a normal window, although it requires additional dialog box-specific configuration. The entire process of instantiating, configuring, and opening a dialog box is shown in the following code.

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Win32;

namespace SDKSample
{
    public partial class MainWindow : Window
    {
```

```


Imports System
Imports System.ComponentModel
Imports System.Windows
Imports System.Windows.Controls
Imports Microsoft.Win32

Namespace SDKSample

Public Class MainWindow
    Inherits Window
```

```
void formatMarginsMenuItem_Click(object sender, RoutedEventArgs e)
{
    // Instantiate the dialog box
    MarginsDialogBox dlg = new MarginsDialogBox();

    // Configure the dialog box
    dlg.Owner = this;
    dlg.DocumentMargin = this.documentTextBox.Margin;

    // Open the dialog box modally
    dlg.ShowDialog();
```

```
Private Sub formatMarginsMenuItem_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Instantiate the dialog box
    Dim dlg As New MarginsDialogBox

    ' Configure the dialog box
    dlg.Owner = Me
    dlg.DocumentMargin = Me.documentTextBox.Margin

    ' Open the dialog box modally
    dlg.ShowDialog()
```

```
}
```

```
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

Here, the code is passing default information (the current margins) to the dialog box. It is also setting the Window.Owner property with a reference to the window that is showing the dialog box. In general, you should always set the owner for a dialog box to provide window state-related behaviors that are common to all dialog boxes (see WPF Windows Overview for more information).

> **NOTE**
>
> You must provide an owner to support user interface (UI) automation for dialog boxes (see UI Automation Overview).

After the dialog box is configured, it is shown modally by calling the ShowDialog method.

Validating User-Provided Data

When a dialog box is opened and the user provides the required data, a dialog box is responsible for ensuring that the provided data is valid for the following reasons:

- From a security perspective, all input should be validated.
- From a domain-specific perspective, data validation prevents erroneous data from being processed by the

code, which could potentially throw exceptions.

- From a user-experience perspective, a dialog box can help users by showing them which data they have entered is invalid.
- From a performance perspective, data validation in a multi-tier application can reduce the number of round trips between the client and the application tiers, particularly when the application is composed of Web services or server-based databases.

To validate a bound control in WPF, you need to define a validation rule and associate it with the binding. A validation rule is a custom class that derives from ValidationRule. The following example shows a validation rule, `MarginValidationRule`, which checks that a bound value is a Double and is within a specified range.

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

namespace SDKSample
{
    public class MarginValidationRule : ValidationRule
    {
        double minMargin;
        double maxMargin;

        public double MinMargin
        {
            get { return this.minMargin; }
            set { this.minMargin = value; }
        }

        public double MaxMargin
        {
            get { return this.maxMargin; }
            set { this.maxMargin = value; }
        }

        public override ValidationResult Validate(object value, CultureInfo cultureInfo)
        {
            double margin;

            // Is a number?
            if (!double.TryParse((string)value, out margin))
            {
                return new ValidationResult(false, "Not a number.");
            }

            // Is in range?
            if ((margin < this.minMargin) || (margin > this.maxMargin))
            {
                string msg = string.Format("Margin must be between {0} and {1}.", this.minMargin,
this.maxMargin);
                return new ValidationResult(false, msg);
            }

            // Number is valid
            return new ValidationResult(true, null);
        }
    }
}
```

```
Imports System.Globalization
Imports System.Windows.Controls

Namespace SDKSample

Public Class MarginValidationRule
    Inherits ValidationRule

    Private _maxMargin As Double
    Private _minMargin As Double

    Public Property MaxMargin() As Double
        Get
            Return Me._maxMargin
        End Get
        Set(ByVal value As Double)
            Me._maxMargin = value
        End Set
    End Property

    Public Property MinMargin() As Double
        Get
            Return Me._minMargin
        End Get
        Set(ByVal value As Double)
            Me._minMargin = value
        End Set
    End Property

    Public Overrides Function Validate(ByVal value As Object, ByVal cultureInfo As CultureInfo) As
ValidationResult

        Dim margin As Double

        ' Is a number?
        If Not Double.TryParse(CStr(value), margin) Then
            Return New ValidationResult(False, "Not a number.")
        End If

        ' Is in range?
        If ((margin < Me.MinMargin) OrElse (margin > Me.MaxMargin)) Then
            Dim msg As String = String.Format("Margin must be between {0} and {1}.", Me.MinMargin,
Me.MaxMargin)
            Return New ValidationResult(False, msg)
        End If

        ' Number is valid
        Return New ValidationResult(True, Nothing)

    End Function

End Class

End Namespace
```

In this code, the validation logic of a validation rule is implemented by overriding the Validate method, which validates the data and returns an appropriate ValidationResult.

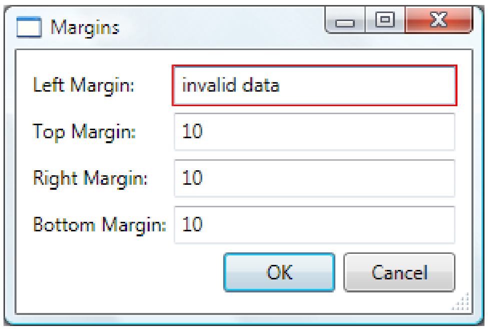
To associate the validation rule with the bound control, you use the following markup.

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.MarginsDialogBox"
    xmlns:local="clr-namespace:SDKSample"
    Title="Margins"
    Height="190"
    Width="300"
    MinHeight="10"
    MinWidth="300"
    ResizeMode="CanResizeWithGrip"
    ShowInTaskbar="False"
    WindowStartupLocation="CenterOwner"
    FocusManager.FocusedElement="{Binding ElementName=leftMarginTextBox}">

  <Grid>
```

```
<Label Grid.Column="0" Grid.Row="0">Left Margin:</Label>
<TextBox Name="leftMarginTextBox" Grid.Column="1" Grid.Row="0">
  <TextBox.Text>
    <Binding Path="Left" UpdateSourceTrigger="PropertyChanged">
      <Binding.ValidationRules>
        <local:MarginValidationRule MinMargin="0" MaxMargin="10" />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

```
</Window>
```

Once the validation rule is associated, WPF will automatically apply it when data is entered into the bound control. When a control contains invalid data, WPF will display a red border around the invalid control, as shown in the following figure.



WPF does not restrict a user to the invalid control until they have entered valid data. This is good behavior for a dialog box; a user should be able to freely navigate the controls in a dialog box whether or not data is valid. However, this means a user can enter invalid data and press the **OK** button. For this reason, your code also needs to validate all controls in a dialog box when the **OK** button is pressed by handling the Click event.

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

namespace SDKSample
{
    public partial class MarginsDialogBox : Window
    {
```

```
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input

Namespace SDKSample



Public Class MarginsDialogBox
    Inherits Window
```

```
void okButton_Click(object sender, RoutedEventArgs e)
{
    // Don't accept the dialog box if there is invalid data
    if (!IsValid(this)) return;
```

```
Private Sub okButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Don't accept the dialog box if there is invalid data
    If Not Me.IsValid(Me) Then Return
```

```
        }

        // Validate all dependency objects in a window
        bool IsValid(DependencyObject node)
        {
            // Check if dependency object was passed
            if (node != null)
            {
                // Check if dependency object is valid.
                // NOTE: Validation.GetHasError works for controls that have validation rules attached
                bool isValid = !Validation.GetHasError(node);
                if (!isValid)
                {
                    // If the dependency object is invalid, and it can receive the focus,
                    // set the focus
                    if (node is IInputElement) Keyboard.Focus((IInputElement)node);
                    return false;
                }
            }

            // If this dependency object is valid, check all child dependency objects
            foreach (object subnode in LogicalTreeHelper.GetChildren(node))
            {
                if (subnode is DependencyObject)
                {
                    // If a child dependency object is invalid, return false immediately,
                    // otherwise keep checking
                    if (IsValid((DependencyObject)subnode) == false) return false;
                }
            }

            // All dependency objects are valid
            return true;
        }
    }
}
```

```
    End Sub

    ' Validate all dependency objects in a window
    Private Function IsValid(ByVal node As DependencyObject) As Boolean

        ' Check if dependency object was passed and if dependency object is valid.
        ' NOTE: Validation.GetHasError works for controls that have validation rules attached
        If ((Not node Is Nothing) AndAlso Validation.GetHasError(node)) Then
            ' If the dependency object is invalid, and it can receive the focus,
            ' set the focus
            If TypeOf node Is IInputElement Then
                Keyboard.Focus(DirectCast(node, IInputElement))
            End If
            Return False
        End If

        ' If this dependency object is valid, check all child dependency objects
        Dim subnode As Object
        For Each subnode In LogicalTreeHelper.GetChildren(node)
            If (TypeOf subnode Is DependencyObject AndAlso Not Me.IsValid(DirectCast(subnode,
DependencyObject))) Then
                ' If a child dependency object is invalid, return false immediately,
                ' otherwise keep checking
                Return False
            End If
        Next

        ' All dependency objects are valid
        Return True

    End Function

End Class

End Namespace
```

This code enumerates all dependency objects on a window and, if any are invalid (as returned by GetHasError, the invalid control gets the focus, the `IsValid` method returns `false`, and the window is considered invalid.

Once a dialog box is valid, it can safely close and return. As part of the return process, it needs to return a result to the calling function.

Setting the Modal Dialog Result

Opening a dialog box using ShowDialog is fundamentally like calling a method: the code that opened the dialog box using ShowDialog waits until ShowDialog returns. When ShowDialog returns, the code that called it needs to decide whether to continue processing or stop processing, based on whether the user pressed the **OK** button or the **Cancel** button. To facilitate this decision, the dialog box needs to return the user's choice as a Boolean value that is returned from the ShowDialog method.

When the **OK** button is clicked, ShowDialog should return `true`. This is achieved by setting the DialogResult property of the dialog box when the **OK** button is clicked.

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

namespace SDKSample
{
    public partial class MarginsDialogBox : Window
    {
```

```
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input

Namespace SDKSample


Public Class MarginsDialogBox
    Inherits Window
```

```
void okButton_Click(object sender, RoutedEventArgs e)
{
```

```
Private Sub okButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
```

```
    // Dialog box accepted
    this.DialogResult = true;
}
```

```
    ' Dialog box accepted
    MyBase.DialogResult = New Nullable(Of Boolean)(True)
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

Note that setting the DialogResult property also causes the window to close automatically, which alleviates the need to explicitly call Close.

When the **Cancel** button is clicked, ShowDialog should return `false`, which also requires setting the DialogResult property.

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

namespace SDKSample
{
    public partial class MarginsDialogBox : Window
    {
```

```
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input

Namespace SDKSample


Public Class MarginsDialogBox
    Inherits Window
```

```
void cancelButton_Click(object sender, RoutedEventArgs e)
{
    // Dialog box canceled
    this.DialogResult = false;
}
```

```
Private Sub cancelButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Dialog box canceled
    Me.DialogResult = False
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

When a button's IsCancel property is set to `true` and the user presses either the **Cancel** button or the ESC key, DialogResult is automatically set to `false`. The following markup has the same effect as the preceding code, without the need to handle the Click event.

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

A dialog box automatically returns `false` when a user presses the **Close** button in the title bar or chooses the **Close** menu item from the **System** menu.

Processing Data Returned from a Modal Dialog Box

When DialogResult is set by a dialog box, the function that opened it can get the dialog box result by inspecting the DialogResult property when ShowDialog returns.

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Win32;

namespace SDKSample
{
    public partial class MainWindow : Window
    {
```

```
Imports System
Imports System.ComponentModel
Imports System.Windows
Imports System.Windows.Controls
Imports Microsoft.Win32

Namespace SDKSample

Public Class MainWindow
    Inherits Window
```

```
void formatMarginsMenuItem_Click(object sender, RoutedEventArgs e)
{
```

```
Private Sub formatMarginsMenuItem_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
```

```
    // Process data entered by user if dialog box is accepted
    if (dlg.DialogResult == true)
    {
        // Update fonts
        this.documentTextBox.Margin = dlg.DocumentMargin;
    }
}
```

```
    ' Process data entered by user if dialog box is accepted
    If (dlg.DialogResult.GetValueOrDefault = True) Then
        Me.documentTextBox.Margin = dlg.DocumentMargin
    End If
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

If the dialog result is `true`, the function uses that as a cue to retrieve and process the data provided by the user.

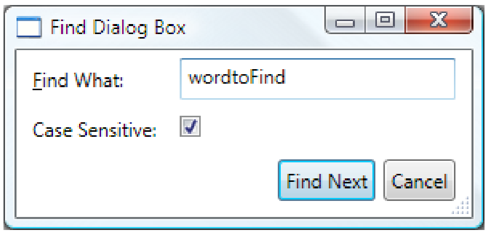
NOTE

After ShowDialog has returned, a dialog box cannot be reopened. Instead, you need to create a new instance.

If the dialog result is `false`, the function should end processing appropriately.

Creating a Modeless Custom Dialog Box

A modeless dialog box, such as the Find Dialog Box shown in the following figure, has the same fundamental appearance as the modal dialog box.



However, the behavior is slightly different, as described in the following sections.

Opening a Modeless Dialog Box

A modeless dialog box is opened by calling the Show method.

```
<!--Main Window-->
```

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Win32;

namespace SDKSample
{
    public partial class MainWindow : Window
    {
```

```


Imports System
Imports System.ComponentModel
Imports System.Windows
Imports System.Windows.Controls
Imports Microsoft.Win32

Namespace SDKSample

Public Class MainWindow
    Inherits Window
```

```
void editFindMenuItem_Click(object sender, RoutedEventArgs e)
{
    // Instantiate the dialog box
    FindDialogBox dlg = new FindDialogBox(this.documentTextBox);

    // Configure the dialog box
    dlg.Owner = this;
    dlg.TextFound += new TextFoundEventHandler(dlg_TextFound);

    // Open the dialog box modally
    dlg.Show();
}
```

```
Private Sub editFindMenuItem_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Dim dlg As New FindDialogBox(Me.documentTextBox)
    dlg.Owner = Me
    AddHandler dlg.TextFound, New TextFoundEventHandler(AddressOf Me.dlg_TextFound)
    dlg.Show()
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

Unlike ShowDialog, Show returns immediately. Consequently, the calling window cannot tell when the modeless dialog box is closed and, therefore, does not know when to check for a dialog box result or get data from the dialog box for further processing. Instead, the dialog box needs to create an alternative way to return data to the calling window for processing.

Processing Data Returned from a Modeless Dialog Box

In this example, the `FindDialogBox` may return one or more find results to the main window, depending on the text being searched for without any specific frequency. As with a modal dialog box, a modeless dialog box can return results using properties. However, the window that owns the dialog box needs to know when to check those properties. One way to enable this is for the dialog box to implement an event that is raised whenever text is found. `FindDialogBox` implements the `TextFoundEvent` for this purpose, which first requires a delegate.

```
using System;
namespace SDKSample
{
    public delegate void TextFoundEventHandler(object sender, EventArgs e);
}
```

```
Namespace SDKSample
Public Delegate Sub TextFoundEventHandler(ByVal sender As Object, ByVal e As EventArgs)
End Namespace
```

Using the `TextFoundEventHandler` delegate, `FindDialogBox` implements the `TextFoundEvent`.

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Text.RegularExpressions;

namespace SDKSample
{
    public partial class FindDialogBox : Window
    {
        public event TextFoundEventHandler TextFound;
        protected virtual void OnTextFound()
        {
            TextFoundEventHandler textFound = this.TextFound;
            if (textFound != null) textFound(this, EventArgs.Empty);
        }
```

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Text.RegularExpressions

Namespace SDKSample

Public Class FindDialogBox
    Inherits Window
    Public Event TextFound As TextFoundEventHandler
    Protected Overridable Sub OnTextFound()
        RaiseEvent TextFound(Me, EventArgs.Empty)
    End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

Consequently, `Find` can raise the event when a search result is found.

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Text.RegularExpressions;

namespace SDKSample
{
    public partial class FindDialogBox : Window
    {
```

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Text.RegularExpressions

Namespace SDKSample

Public Class FindDialogBox
    Inherits Window
```

```
void findNextButton_Click(object sender, RoutedEventArgs e)
{
```

```
Private Sub findNextButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
```

```
// Text found
this.index = match.Index;
this.length = match.Length;
OnTextFound();
```

```
Me.Index = match.Index
Me.Length = match.Length
RaiseEvent TextFound(Me, EventArgs.Empty)
```

```
}
```

```
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

The owner window then needs to register with and handle this event.

```
using System;
using System.ComponentModel;
using System.Windows;
using System.Windows.Controls;
using Microsoft.Win32;

namespace SDKSample
{
    public partial class MainWindow : Window
    {
```

```


Imports System
Imports System.ComponentModel
Imports System.Windows
Imports System.Windows.Controls
Imports Microsoft.Win32

Namespace SDKSample

Public Class MainWindow
    Inherits Window
```

```
        void dlg_TextFound(object sender, EventArgs e)
        {
            // Get the find dialog box that raised the event
            FindDialogBox dlg = (FindDialogBox)sender;

            // Get find results and select found text
            this.documentTextBox.Select(dlg.Index, dlg.Length);
            this.documentTextBox.Focus();
        }
    }
}
```

```
    Private Sub dlg_TextFound(ByVal sender As Object, ByVal e As EventArgs)
        Dim dlg As FindDialogBox = DirectCast(sender, FindDialogBox)
        Me.documentTextBox.Select(dlg.Index, dlg.Length)
        Me.documentTextBox.Focus()
    End Sub

End Class

End Namespace
```

Closing a Modeless Dialog Box

Because DialogResult does not need to be set, a modeless dialog can be closed using system provide mechanisms, including the following:

- Clicking the **Close** button in the title bar.
- Pressing ALT+F4.
- Choosing **Close** from the **System** menu.

Alternatively, your code can call Close when the **Close** button is clicked.

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Text.RegularExpressions;

namespace SDKSample
{
    public partial class FindDialogBox : Window
    {
```

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Text.RegularExpressions

Namespace SDKSample

Public Class FindDialogBox
    Inherits Window
```

```
        void closeButton_Click(object sender, RoutedEventArgs e)
        {
            // Close dialog box
            this.Close();
        }
    }
}
```

```
    Private Sub closeButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
        MyBase.Close()
    End Sub
End Class

End Namespace
```

See also

- Popup Overview
- Dialog Box Sample
- ColorPicker Custom Control Sample

Window Management How-to Topics

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

The following topics show how to manage Windows Presentation Foundation (WPF) windows.

In This Section

Automatically Size a Window to Fit Its Content
Get all Windows in an Application
Get and Set the Main Application Window
Open a Dialog Box
Open a Message Box
Open a Window
Return a Dialog Box Result

Related Sections

Application Management Overview

Navigation Overview

Hosting

Build and Deploy

How to: Automatically Size a Window to Fit Its Content

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

This example shows how to set the SizeToContent property to specify how a window resizes to fit its content.

Example

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

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

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

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

```
' Manually alter window height and width
Me.SizeToContent = SizeToContent.Manual

' Automatically resize width relative to content
Me.SizeToContent = SizeToContent.Width

' Automatically resize height relative to content
Me.SizeToContent = SizeToContent.Height

' Automatically resize height and width relative to content
Me.SizeToContent = SizeToContent.WidthAndHeight
```

How to: Get all Windows in an Application

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

This example shows how to get all Window objects in an application.

Example

Every instantiated Window object, whether visible or not, is automatically added to a collection of window references that is managed by Application, and exposed from Windows.

You can enumerate Windows to get all instantiated windows using the following code:

```
foreach( Window window in Application.Current.Windows ) {
  Console.WriteLine(window.Title);
}
```

```
For Each window As Window In Application.Current.Windows
  Console.WriteLine(window.Title)
Next window
```

How to: Get and Set the Main Application Window

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

This example shows how to get and set the main application window.

Example

The first Window that is instantiated within a Windows Presentation Foundation (WPF) application is automatically set by Application as the main application window. The first Window to be instantiated will most likely be the window that is specified as the startup uniform resource identifier (URI) (see StartupUri).

The first Window could also be instantiated using code. One example is opening a window during application startup, like the following:

```
public partial class App : Application
{
    void App_Startup(object sender, StartupEventArgs e)
    {
        MainWindow window = new MainWindow();
        window.Show();
    }
}
```

```
Partial Public Class App
    Inherits Application
    Private Sub App_Startup(ByVal sender As Object, ByVal e As StartupEventArgs)
        Dim window As New MainWindow()
        window.Show()
    End Sub
End Class
```

Sometimes, the first instantiated Window is not actually the main application window e.g. a splash screen. In this case, you can specify the main application window using markup, like the following:

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="StartupWindow.xaml"
    >
  <Application.MainWindow>
    <NavigationWindow Source="MainPage.xaml" Visibility="Visible"></NavigationWindow>
  </Application.MainWindow>
</Application>
```

Whether the main window is specified automatically or manually, you can get the main window from MainWindow using the following code, like the following:

```
// Get the main window
Window mainWindow = this.MainWindow;
```

```
' Get the main window
Dim mainWindow As Window = Me.MainWindow
```

How to: Open a Dialog Box

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

This example shows how to open a dialog box.

Example

A dialog box is a window that is opened by instantiating Window and calling the ShowDialog method. ShowDialog opens a window and doesn't return until the new dialog box has been closed. This type of window is also known as a *modal* window, and restricts user input.

```
CustomDialogBox dialogBox = new CustomDialogBox();
dialogBox.ShowDialog(); // Returns when dialog box has closed
```

```
Dim dialogBox As New CustomDialogBox()
dialogBox.ShowDialog() ' Returns when dialog box has closed
```

.NET Framework Security

Calling ShowDialog requires permission to use all windows and user input events without restriction.

See also

- Return a Dialog Box Result

How to: Open a Message Box

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

This example shows how to open a message box.

Example

A message box is a prefabricated modal dialog box for displaying information to users. A message box is opened by calling the static Show method of the MessageBox class. When Show is called, the message is passed using a string parameter. Several overloads of Show allow you to configure how a message box will appear (see MessageBox).

```
private void ShowMessageBoxButton_Click(object sender, RoutedEventArgs e)
{
    // Configure message box
    string message = "Hello, MessageBox!";
    // Show message box
    MessageBoxResult result = MessageBox.Show(message);
}
```

```
Private Sub showMessageBoxButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Configure message box
    Dim message As String = "Hello, MessageBox!"
    ' Show message box
    Dim result As MessageBoxResult = MessageBox.Show(message)
End Sub
```

See also

- MessageBox Sample

How to: Open a Window

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

This example shows how to open a window.

Example

A window is opened by instantiating Window and calling the Show method. Show opens a window and returns immediately without waiting for the new window to close. This type of window is also known as a *modeless* window, and doesn't restrict user input.

```
CustomWindow window = new CustomWindow();
window.Show(); // Returns immediately
```

```
Dim window As New CustomWindow()
window.Show() ' Returns immediately
```

.NET Framework Security

Instantiating Window requires permission to call unsafe native methods (see Window).

How to: Return a Dialog Box Result

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

This example shows how to retrieve the dialog result for a window that is opened by calling ShowDialog.

Example

Before a dialog box closes, its DialogResult property should be set with a Nullable<T>Boolean that indicates how the user closed the dialog box. This value is returned by ShowDialog to allow client code to determine how the dialog box was closed and, consequently, how to process the result.

NOTE

DialogResult can only be set if a window was opened by calling ShowDialog.

```
DialogBoxWithResult dialogBoxWithResult = new DialogBoxWithResult();
// Open dialog box and retrieve dialog result when ShowDialog returns
bool? dialogResult = dialogBoxWithResult.ShowDialog();
switch (dialogResult)
{
    case true:
        // User accepted dialog box
        break;
    case false:
        // User canceled dialog box
        break;
    default:
        // Indeterminate
        break;
}
```

```
Dim dialogBoxWithResult As New DialogBoxWithResult()
' Open dialog box and retrieve dialog result when ShowDialog returns
Dim dialogResult? As Boolean = dialogBoxWithResult.ShowDialog()
Select Case dialogResult
    Case True
        ' User accepted dialog box
    Case False
        ' User canceled dialog box
    Case Else
        ' Indeterminate
End Select
```

.NET Framework Security

Calling ShowDialog requires permission to use all windows and user input events without restriction.

Navigation Overview

1/23/2019 • 36 minutes to read • Edit Online

Windows Presentation Foundation (WPF) supports browser-style navigation that can be used in two types of applications: standalone applications and XAML browser applications (XBAPs). To package content for navigation, WPF provides the Page class. You can navigate from one Page to another declaratively, by using a Hyperlink, or programmatically, by using the NavigationService. WPF uses the journal to remember pages that have been navigated from and to navigate back to them.

Page, Hyperlink, NavigationService, and the journal form the core of the navigation support offered by WPF. This overview explores these features in detail before covering advanced navigation support that includes navigation to loose Extensible Application Markup Language (XAML) files, HTML files, and objects.

> **NOTE**
> In this topic, the term "browser" refers only to browsers that can host WPF applications, which currently includes Microsoft Internet Explorer and Firefox. Where specific WPF features are supported only by a particular browser, the browser version is referred to.

Navigation in WPF Applications

This topic provides an overview of the key navigation capabilities in WPF. These capabilities are available to both standalone applications and XBAPs, although this topic presents them within the context of an XBAP.

> **NOTE**
> This topic doesn't discuss how to build and deploy XBAPs. For more information on XBAPs, see WPF XAML Browser Applications Overview.

This section explains and demonstrates the following aspects of navigation:

- Implementing a Page
- Configuring a Start Page
- Configuring the Host Window's Title, Width, and Height
- Hyperlink Navigation
- Fragment Navigation
- Navigation Service
- Programmatic Navigation with the Navigation Service
- Navigation Lifetime
- Remembering Navigation with the Journal
- Page Lifetime and the Journal
- Retaining Content State with Navigation History
- Cookies

- Structured Navigation

Implementing a Page

In WPF, you can navigate to several content types that include .NET Framework objects, custom objects, enumeration values, user controls, XAML files, and HTML files. However, you'll find that the most common and convenient way to package content is by using Page. Furthermore, Page implements navigation-specific features to enhance their appearance and simplify development.

Using Page, you can declaratively implement a navigable page of XAML content by using markup like the following.

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

A Page that is implemented in XAML markup has `Page` as its root element and requires the WPFXML namespace declaration. The `Page` element contains the content that you want to navigate to and display. You add content by setting the `Page.Content` property element, as shown in the following markup.

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Page.Content>
    <!-- Page Content -->
    Hello, Page!
  </Page.Content>
</Page>
```

`Page.Content` can only contain one child element; in the preceding example, the content is a single string, "Hello, Page!" In practice, you will usually use a layout control as the child element (see Layout) to contain and compose your content.

The child elements of a `Page` element are considered to be the content of a Page and, consequently, you don't need to use the explicit `Page.Content` declaration. The following markup is the declarative equivalent to the preceding sample.

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <!-- Page Content -->
  Hello, Page!
</Page>
```

In this case, `Page.Content` is automatically set with the child elements of the `Page` element. For more information, see WPF Content Model.

A markup-only Page is useful for displaying content. However, a Page can also display controls that allow users to interact with the page, and it can respond to user interaction by handling events and calling application logic. An interactive Page is implemented by using a combination of markup and code-behind, as shown in the following example.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.HomePage">
  Hello, from the XBAP HomePage!
</Page>
```

```csharp
using System.Windows.Controls;

namespace SDKSample
{
    public partial class HomePage : Page
    {
        public HomePage()
        {
            InitializeComponent();
        }
    }
}
```

```vb
Imports System.Windows.Controls

Namespace SDKSample
    Partial Public Class HomePage
        Inherits Page
        Public Sub New()
            InitializeComponent()
        End Sub
    End Class
End Namespace
```

To allow a markup file and code-behind file to work together, the following configuration is required:

- In markup, the `Page` element must include the `x:Class` attribute. When the application is built, the existence of `x:Class` in the markup file causes Microsoft build engine (MSBuild) to create a `partial` class that derives from Page and has the name that is specified by the `x:Class` attribute. This requires the addition of an XML namespace declaration for the XAML schema (`xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`). The generated `partial` class implements `InitializeComponent`, which is called to register the events and set the properties that are implemented in markup.

- In code-behind, the class must be a `partial` class with the same name that is specified by the `x:Class` attribute in markup, and it must derive from Page. This allows the code-behind file to be associated with the `partial` class that is generated for the markup file when the application is built (see Building a WPF Application).

- In code-behind, the Page class must implement a constructor that calls the `InitializeComponent` method. `InitializeComponent` is implemented by the markup file's generated `partial` class to register events and set properties that are defined in markup.

NOTE

When you add a new Page to your project using Microsoft Visual Studio, the Page is implemented using both markup and code-behind, and it includes the necessary configuration to create the association between the markup and code-behind files as described here.

Once you have a Page, you can navigate to it. To specify the first Page that an application navigates to, you need to configure the start Page.

Configuring a Start Page

XBAPs require a certain amount of application infrastructure to be hosted in a browser. In WPF, the Application class is part of an application definition that establishes the required application infrastructure (see Application

Management Overview).

An application definition is usually implemented using both markup and code-behind, with the markup file configured as an MSBuild `ApplicationDefinition` item. The following is an application definition for an XBAP.

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

```
using System.Windows;

namespace SDKSample
{
    public partial class App : Application { }
}
```

```
Imports System.Windows

Namespace SDKSample
    Partial Public Class App
        Inherits Application
    End Class
End Namespace
```

An XBAP can use its application definition to specify a start Page, which is the Page that is automatically loaded when the XBAP is launched. You do this by setting the StartupUri property with the uniform resource identifier (URI) for the desired Page.

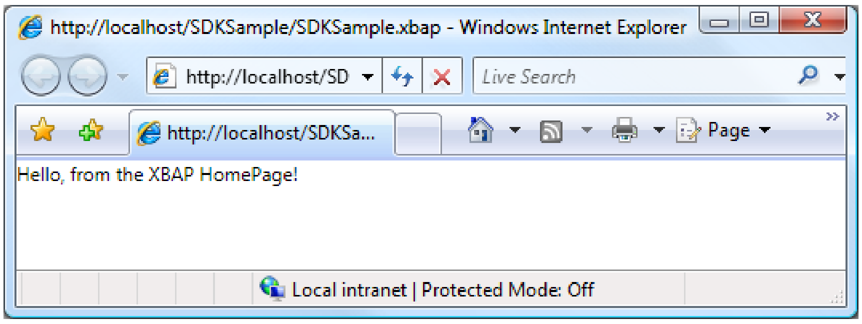
NOTE

In most cases, the Page is either compiled into or deployed with an application. In these cases, the URI that identifies a Page is a pack URI, which is a URI that conforms to the *pack* scheme. Pack URIs are discussed further in Pack URIs in WPF. You can also navigate to content using the http scheme, which is discussed below.

You can set StartupUri declaratively in markup, as shown in the following example.

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

In this example, the `StartupUri` attribute is set with a relative pack URI that identifies HomePage.xaml. When the XBAP is launched, HomePage.xaml is automatically navigated to and displayed. This is demonstrated by the following figure, which shows an XBAP that was launched from a Web server.


http://localhost/SDKSample/SDKSample.xbap - Windows Internet Explorer
http://localhost/SD
Live Search
http://localhost/SDKSa...
Page
Local intranet | Protected Mode: Off

Hello, from the XBAP HomePage!

NOTE

For more information regarding the development and deployment of XBAPs, see WPF XAML Browser Applications Overview and Deploying a WPF Application.

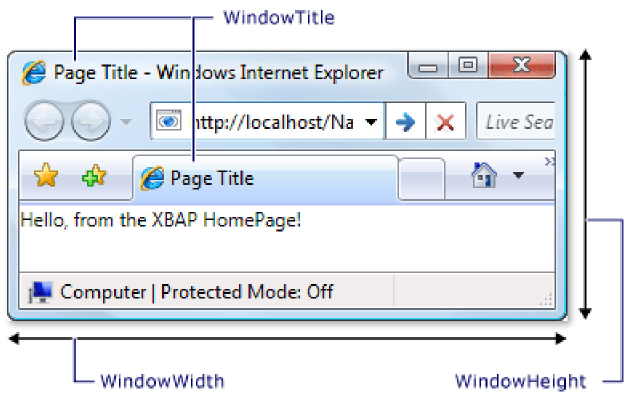
Configuring the Host Window's Title, Width, and Height

One thing you may have noticed from the previous figure is that the title of both the browser and the tab panel is the URI for the XBAP. Besides being long, the title is neither attractive nor informative. For this reason, Page offers a way for you to change the title by setting the WindowTitle property. Furthermore, you can configure the width and height of the browser window by setting WindowWidth and WindowHeight, respectively.

WindowTitle, WindowWidth, and WindowHeight can be set declaratively in markup, as shown in the following example.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.HomePage"
    WindowTitle="Page Title"
    WindowWidth="500"
    WindowHeight="200">
  Hello, from the XBAP HomePage!
</Page>
```

The result is shown in the following figure.



Hello, from the XBAP HomePage!

Hyperlink Navigation

A typical XBAP comprises several pages. The simplest way to navigate from one page to another is to use a Hyperlink. You can declaratively add a Hyperlink to a Page by using the `Hyperlink` element, which is shown in the following markup.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  WindowTitle="Page With Hyperlink"
  WindowWidth="250"
  WindowHeight="250">
```

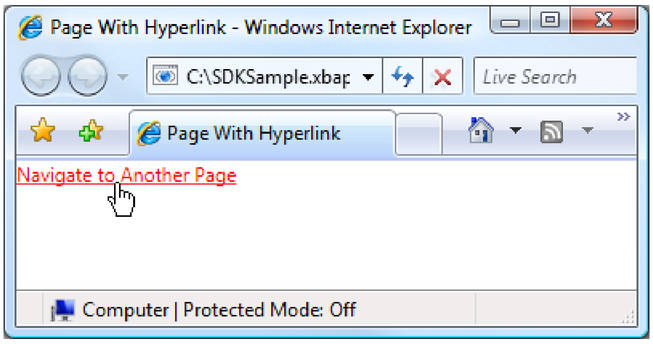
```
<Hyperlink NavigateUri="UriOfPageToNavigateTo.xaml">
  Navigate to Another Page
</Hyperlink>
```

```
</Page>
```

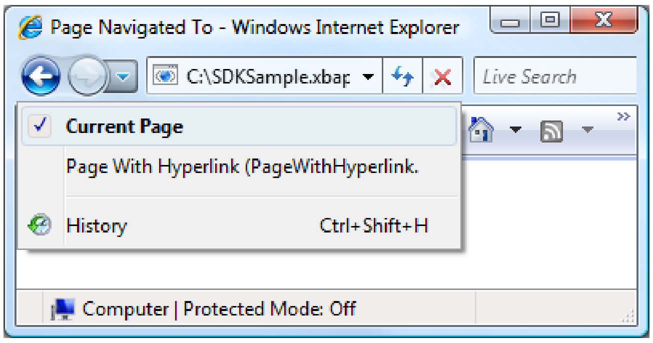
A `Hyperlink` element requires the following:

- The pack URI of the Page to navigate to, as specified by the `NavigateUri` attribute.
- Content that a user can click to initiate the navigation, such as text and images (for the content that the `Hyperlink` element can contain, see Hyperlink).

The following figure shows an XBAP with a Page that has a Hyperlink.


Page With Hyperlink - Windows Internet Explorer
C:\SDKSample.xbap
Live Search
Page With Hyperlink
Navigate to Another Page
Computer | Protected Mode: Off

As you would expect, clicking the Hyperlink causes the XBAP to navigate to the Page that is identified by the `NavigateUri` attribute. Additionally, the XBAP adds an entry for the previous Page to the Recent Pages list in Internet Explorer. This is shown in the following figure.


Page Navigated To - Windows Internet Explorer
C:\SDKSample.xbap
Live Search
Current Page
Page With Hyperlink (PageWithHyperlink.
History
Ctrl+Shift+H
Computer | Protected Mode: Off

As well as supporting navigation from one Page to another, Hyperlink also supports fragment navigation.

Fragment Navigation

Fragment navigation is the navigation to a content fragment in either the current Page or another Page. In WPF, a content fragment is the content that is contained by a named element. A named element is an element that has its `Name` attribute set. The following markup shows a named `TextBlock` element that contains a content fragment.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    WindowTitle="Page With Fragments" >
```

```
<!-- Content Fragment called "Fragment1" -->
<TextBlock Name="Fragment1">
  Ea vel dignissim te aliquam facilisis ...
</TextBlock>
```

```
</Page>
```

For a Hyperlink to navigate to a content fragment, the `NavigateUri` attribute must include the following:

- The URI of the Page with the content fragment to navigate to.
- A "#" character.
- The name of the element on the Page that contains the content fragment.

A fragment URI has the following format.

PageURI `#` *ElementName*

The following shows an example of a `Hyperlink` that is configured to navigate to a content fragment.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  WindowTitle="Page That Navigates To Fragment" >
```

```
<Hyperlink NavigateUri="PageWithFragments.xaml#Fragment1">
  Navigate To pack Fragment
</Hyperlink>
```

```
</Page>
```

NOTE

This section describes the default fragment navigation implementation in WPF. WPF also allows you to implement your own fragment navigation scheme which, in part, requires handling the NavigationService.FragmentNavigation event.

IMPORTANT

You can navigate to fragments in loose XAML pages (markup-only XAML files with `Page` as the root element) only if the pages can be browsed via HTTP.

However, a loose XAML page can navigate to its own fragments.

Navigation Service

While Hyperlink allows a user to initiate navigation to a particular Page, the work of locating and downloading the page is performed by the NavigationService class. Essentially, NavigationService provides the ability to process a navigation request on behalf of client code, such as the Hyperlink. Additionally, NavigationService implements higher-level support for tracking and influencing a navigation request.

When a Hyperlink is clicked, WPF calls NavigationService.Navigate to locate and download the Page at the specified pack URI. The downloaded Page is converted to a tree of objects whose root object is an instance of the downloaded Page. A reference to the root Page object is stored in the NavigationService.Content property. The pack URI for the content that was navigated to is stored in the NavigationService.Source property, while the NavigationService.CurrentSource stores the pack URI for the last page that was navigated to.

> **NOTE**
>
> It is possible for a WPF application to have more than one currently active NavigationService. For more information, see Navigation Hosts later in this topic.

Programmatic Navigation with the Navigation Service

You don't need to know about NavigationService if navigation is implemented declaratively in markup using Hyperlink, because Hyperlink uses the NavigationService on your behalf. This means that, as long as either the direct or indirect parent of a Hyperlink is a navigation host (see Navigation Hosts), Hyperlink will be able to find and use the navigation host's navigation service to process a navigation request.

However, there are situations when you need to use NavigationService directly, including the following:

- When you need to instantiate a Page using a non-default constructor.
- When you need to set properties on the Page before you navigate to it.
- When the Page that needs to be navigated to can only be determined at run time.

In these situations, you need to write code to programmatically initiate navigation by calling the Navigate method of the NavigationService object. That requires getting a reference to a NavigationService.

Getting a Reference to the NavigationService

For reasons that are covered in the Navigation Hosts section, a WPF application can have more than one NavigationService. This means that your code needs a way to find a NavigationService, which is usually the NavigationService that navigated to the current Page. You can get a reference to a NavigationService by calling the `static` NavigationService.GetNavigationService method. To get the NavigationService that navigated to a particular Page, you pass a reference to the Page as the argument of the GetNavigationService method. The following code shows how to get the NavigationService for the current Page.

```
using System.Windows.Navigation;
```

```
// Get a reference to the NavigationService that navigated to this Page
NavigationService ns = NavigationService.GetNavigationService(this);
```

```
' Get a reference to the NavigationService that navigated to this Page
Dim ns As NavigationService = NavigationService.GetNavigationService(Me)
```

As a shortcut for finding the NavigationService for a Page, Page implements the NavigationService property. This is shown in the following example.

```
using System.Windows.Navigation;
```

```
// Get a reference to the NavigationService that navigated to this Page
NavigationService ns = this.NavigationService;
```

```
' Get a reference to the NavigationService that navigated to this Page
Dim ns As NavigationService = Me.NavigationService
```

NOTE

A Page can only get a reference to its NavigationService when Page raises the Loaded event.

Programmatic Navigation to a Page Object

The following example shows how to use the NavigationService to programmatically navigate to a Page. Programmatic navigation is required because the Page that is being navigated to can only be instantiated using a single, non-default constructor. The Page with the non-default constructor is shown in the following markup and code.

```
<Page
    x:Class="SDKSample.PageWithNonDefaultConstructor"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="PageWithNonDefaultConstructor">

  <!-- Content goes here -->

</Page>
```

```
using System.Windows.Controls;

namespace SDKSample
{
    public partial class PageWithNonDefaultConstructor : Page
    {
        public PageWithNonDefaultConstructor(string message)
        {
            InitializeComponent();

            this.Content = message;
        }
    }
}
```

```

Namespace SDKSample
    Partial Public Class PageWithNonDefaultConstructor
        Inherits Page
        Public Sub New(ByVal message As String)
            InitializeComponent()

            Me.Content = message
        End Sub
    End Class
End Namespace
```

The Page that navigates to the Page with the non-default constructor is shown in the following markup and code.

```xml
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.NSNavigationPage">

  <Hyperlink Click="hyperlink_Click">
    Navigate to Page with Non-Default Constructor
  </Hyperlink>

</Page>
```

```csharp
using System.Windows;
using System.Windows.Controls;
using System.Windows.Navigation;

namespace SDKSample
{
    public partial class NSNavigationPage : Page
    {
        public NSNavigationPage()
        {
            InitializeComponent();
        }

        void hyperlink_Click(object sender, RoutedEventArgs e)
        {
            // Instantiate the page to navigate to
            PageWithNonDefaultConstructor page = new PageWithNonDefaultConstructor("Hello!");

            // Navigate to the page, using the NavigationService
            this.NavigationService.Navigate(page);
        }
    }
}
```

```vb
Namespace SDKSample
    Partial Public Class NSNavigationPage
        Inherits Page
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub hyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            ' Instantiate the page to navigate to
            Dim page As New PageWithNonDefaultConstructor("Hello!")

            ' Navigate to the page, using the NavigationService
            Me.NavigationService.Navigate(page)
        End Sub
    End Class
End Namespace
```

When the Hyperlink on this Page is clicked, navigation is initiated by instantiating the Page to navigate to using the non-default constructor and calling the NavigationService.Navigate method. Navigate accepts a reference to the object that the NavigationService will navigate to, rather than a pack URI.

Programmatic Navigation with a Pack URI

If you need to construct a pack URI programmatically (when you can only determine the pack URI at run time, for example), you can use the NavigationService.Navigate method. This is shown in the following example.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.NSUriNavigationPage">
  <Hyperlink Click="hyperlink_Click">Navigate to Page by Pack URI</Hyperlink>
</Page>
```

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

namespace SDKSample
{
    public partial class NSUriNavigationPage : Page
    {
        public NSUriNavigationPage()
        {
            InitializeComponent();
        }

        void hyperlink_Click(object sender, RoutedEventArgs e)
        {
            // Create a pack URI
            Uri uri = new Uri("AnotherPage.xaml", UriKind.Relative);

            // Get the navigation service that was used to
            // navigate to this page, and navigate to
            // AnotherPage.xaml
            this.NavigationService.Navigate(uri);
        }
    }
}
```

```
Namespace SDKSample
    Partial Public Class NSUriNavigationPage
        Inherits Page
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub hyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            ' Create a pack URI
            Dim uri As New Uri("AnotherPage.xaml", UriKind.Relative)

            ' Get the navigation service that was used to
            ' navigate to this page, and navigate to
            ' AnotherPage.xaml
            Me.NavigationService.Navigate(uri)
        End Sub
    End Class
End Namespace
```

Refreshing the Current Page

A Page is not downloaded if it has the same pack URI as the pack URI that is stored in the NavigationService.Source property. To force WPF to download the current page again, you can call the NavigationService.Refresh method, as shown in the following example.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.NSRefreshNavigationPage">
 <Hyperlink Click="hyperlink_Click">Refresh this page</Hyperlink>
</Page>
```

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

namespace SDKSample
{
    public partial class NSRefreshNavigationPage : Page
    {
```

```
Namespace SDKSample
    Partial Public Class NSRefreshNavigationPage
        Inherits Page
```

```
        void hyperlink_Click(object sender, RoutedEventArgs e)
        {
            // Force WPF to download this page again
            this.NavigationService.Refresh();
        }
    }
}
```

```
        Private Sub hyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            ' Force WPF to download this page again
            Me.NavigationService.Refresh()
        End Sub
    End Class
End Namespace
```

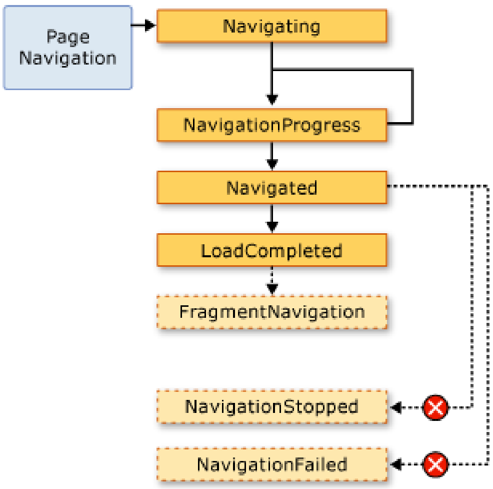
Navigation Lifetime

There are many ways to initiate navigation, as you've seen. When navigation is initiated, and while navigation is in progress, you can track and influence the navigation using the following events that are implemented by NavigationService:

- Navigating. Occurs when a new navigation is requested. Can be used to cancel the navigation.
- NavigationProgress. Occurs periodically during a download to provide navigation progress information.
- Navigated. Occurs when the page has been located and downloaded.
- NavigationStopped. Occurs when the navigation is stopped (by calling StopLoading), or when a new navigation is requested while a current navigation is in progress.
- NavigationFailed. Occurs when an error is raised while navigating to the requested content.
- LoadCompleted. Occurs when content that was navigated to is loaded and parsed, and has begun rendering.
- FragmentNavigation. Occurs when navigation to a content fragment begins, which happens:

- Immediately, if the desired fragment is in the current content.
- After the source content has been loaded, if the desired fragment is in different content.

The navigation events are raised in the order that is illustrated by the following figure.



In general, a Page isn't concerned about these events. It is more likely that an application is concerned with them and, for that reason, these events are also raised by the Application class:

- Application.Navigating
- Application.NavigationProgress
- Application.Navigated
- Application.NavigationFailed
- Application.NavigationStopped
- Application.LoadCompleted
- Application.FragmentNavigation

Every time NavigationService raises an event, the Application class raises the corresponding event. Frame and NavigationWindow offer the same events to detect navigation within their respective scopes.

In some cases, a Page might be interested in these events. For example, a Page might handle the NavigationService.Navigating event to determine whether or not to cancel navigation away from itself. This is shown in the following example.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.CancelNavigationPage">
  <Button Click="button_Click">Navigate to Another Page</Button>
</Page>
```

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

namespace SDKSample
{
    public partial class CancelNavigationPage : Page
    {
        public CancelNavigationPage()
        {
            InitializeComponent();

            // Can only access the NavigationService when the page has been loaded
            this.Loaded += new RoutedEventHandler(CancelNavigationPage_Loaded);
            this.Unloaded += new RoutedEventHandler(CancelNavigationPage_Unloaded);
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Force WPF to download this page again
            this.NavigationService.Navigate(new Uri("AnotherPage.xaml", UriKind.Relative));
        }

        void CancelNavigationPage_Loaded(object sender, RoutedEventArgs e)
        {
            this.NavigationService.Navigating += new
NavigatingCancelEventHandler(NavigationService_Navigating);
        }

        void CancelNavigationPage_Unloaded(object sender, RoutedEventArgs e)
        {
            this.NavigationService.Navigating -= new
NavigatingCancelEventHandler(NavigationService_Navigating);
        }

        void NavigationService_Navigating(object sender, NavigatingCancelEventArgs e)
        {
            // Does the user really want to navigate to another page?
            MessageBoxResult result;
            result = MessageBox.Show("Do you want to leave this page?", "Navigation Request",
MessageBoxButton.YesNo);

            // If the user doesn't want to navigate away, cancel the navigation
            if (result == MessageBoxResult.No) e.Cancel = true;
        }
    }
}
```

```
Namespace SDKSample
    Partial Public Class CancelNavigationPage
        Inherits Page
        Public Sub New()
            InitializeComponent()

            ' Can only access the NavigationService when the page has been loaded
            AddHandler Loaded, AddressOf CancelNavigationPage_Loaded
            AddHandler Unloaded, AddressOf CancelNavigationPage_Unloaded
        End Sub

        Private Sub button_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            ' Force WPF to download this page again
            Me.NavigationService.Navigate(New Uri("AnotherPage.xaml", UriKind.Relative))
        End Sub

        Private Sub CancelNavigationPage_Loaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
            AddHandler NavigationService.Navigating, AddressOf NavigationService_Navigating
        End Sub

        Private Sub CancelNavigationPage_Unloaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
            RemoveHandler NavigationService.Navigating, AddressOf NavigationService_Navigating
        End Sub

        Private Sub NavigationService_Navigating(ByVal sender As Object, ByVal e As
NavigatingCancelEventArgs)
            ' Does the user really want to navigate to another page?
            Dim result As MessageBoxResult
            result = MessageBox.Show("Do you want to leave this page?", "Navigation Request",
MessageBoxButton.YesNo)

            ' If the user doesn't want to navigate away, cancel the navigation
            If result = MessageBoxResult.No Then
                e.Cancel = True
            End If
        End Sub
    End Class
End Namespace
```

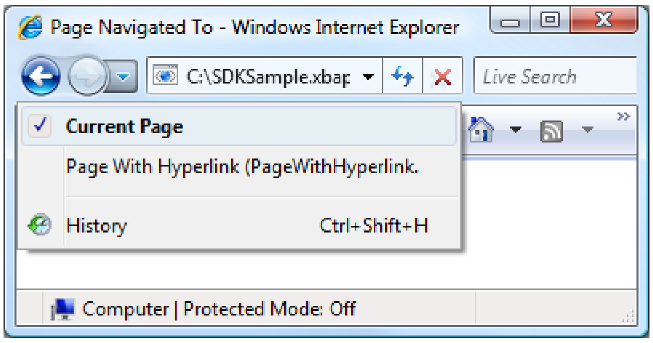
If you register a handler with a navigation event from a Page, as the preceding example does, you must also unregister the event handler. If you don't, there may be side effects with respect to how WPF navigation remembers Page navigation using the journal.

Remembering Navigation with the Journal

WPF uses two stacks to remember the pages that you have navigated from: a back stack and a forward stack. When you navigate from the current Page to a new Page or forward to an existing Page, the current Page is added to the *back stack*. When you navigate from the current Page back to the previous Page, the current Page is added to the *forward stack*. The back stack, the forward stack, and the functionality to manage them, are collectively referred to as the journal. Each item in the back stack and the forward stack is an instance of the JournalEntry class, and is referred to as a *journal entry*.

Navigating the Journal from Internet Explorer

Conceptually, the journal operates the same way that the **Back** and **Forward** buttons in Internet Explorer do. These are shown in the following figure.


Page Navigated To - Windows Internet Explorer
C:\SDKSample.xbap
Live Search
Current Page
Page With Hyperlink (PageWithHyperlink.
History
Ctrl+Shift+H
Computer | Protected Mode: Off

For XBAPs that are hosted by Internet Explorer, WPF integrates the journal into the navigation UI of Internet Explorer. This allows users to navigate pages in an XBAP by using the **Back**, **Forward**, and **Recent Pages** buttons in Internet Explorer. The journal is not integrated into Microsoft Internet Explorer 6 in the same way it is for Internet Explorer 7 or Internet Explorer 8. Instead, WPF renders a substitute navigation UI.

IMPORTANT

In Internet Explorer, when a user navigates away from and back to an XBAP, only the journal entries for pages that were not kept alive are retained in the journal. For discussion on keeping pages alive, see Page Lifetime and the Journal later in this topic.

By default, the text for each Page that appears in the **Recent Pages** list of Internet Explorer is the URI for the Page. In many cases, this is not particularly meaningful to the user. Fortunately, you can change the text using one the following options:

1. The attached `JournalEntry.Name` attribute value.
2. The `Page.Title` attribute value.
3. The `Page.WindowTitle` attribute value and the URI for the current Page.
4. The URI for the current Page. (Default)

The order in which the options are listed matches the order of precedence for finding the text. For example, if `JournalEntry.Name` is set, the other values are ignored.

The following example uses the `Page.Title` attribute to change the text that appears for a journal entry.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.PageWithTitle"
    Title="This is the title of the journal entry for this page.">
```

```
</Page>
```

```
using System.Windows.Controls;

namespace SDKSample
{
    public partial class PageWithTitle : Page
    {
```

```
Namespace SDKSample
    Partial Public Class PageWithTitle
        Inherits Page
```

```
    }
}
```

```
    End Class
End Namespace
```

Navigating the Journal Using WPF

Although a user can navigate the journal by using the **Back**, **Forward**, and **Recent Pages** in Internet Explorer, you can also navigate the journal using both declarative and programmatic mechanisms provided by WPF. One reason to do this is to provide custom navigation UIs in your pages.

You can declaratively add journal navigation support by using the navigation commands exposed by NavigationCommands. The following example demonstrates how to use the `BrowseBack` navigation command.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.NavigationCommandsPage">
```

```
<Hyperlink Command="NavigationCommands.BrowseBack">Back</Hyperlink>
```

```
<Hyperlink Command="NavigationCommands.BrowseForward">Forward</Hyperlink>
```

```
</Page>
```

You can programmatically navigate the journal by using one of the following members of the NavigationService class:

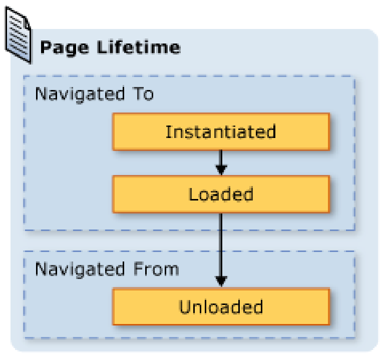
- GoBack
- GoForward
- CanGoBack
- CanGoForward

The journal can also be manipulated programmatically, as discussed in Retaining Content State with Navigation History later in this topic.

Page Lifetime and the Journal

Consider an XBAP with several pages that contain rich content, including graphics, animations, and media. The memory footprint for pages like these could be quite large, particularly if video and audio media are used. Given that the journal "remembers" pages that have been navigated to, such an XBAP could quickly consume a large and noticeable amount of memory.

For this reason, the default behavior of the journal is to store Page metadata in each journal entry rather than a reference to a Page object. When a journal entry is navigated to, its Page metadata is used to create a new instance of the specified Page. As a consequence, each Page that is navigated has the lifetime that is illustrated by the following figure.

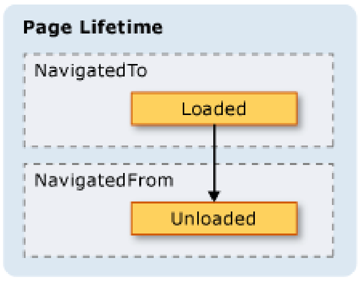


Although using the default journaling behavior can save on memory consumption, per-page rendering performance might be reduced; reinstantiating a Page can be time-intensive, particularly if it has a lot of content. If you need to retain a Page instance in the journal, you can draw on two techniques for doing so. First, you can programmatically navigate to a Page object by calling the NavigationService.Navigate method.

Second, you can specify that WPF retain an instance of a Page in the journal by setting the KeepAlive property to `true` (the default is `false`). As shown in the following example, you can set KeepAlive declaratively in markup.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.KeepAlivePage"
    KeepAlive="True">

  An instance of this page is stored in the journal.

</Page>
```

The lifetime of a Page that is kept alive is subtly different from one that is not. The first time a Page that is kept alive is navigated to, it is instantiated just like a Page that is not kept alive. However, because an instance of the Page is retained in the journal, it is never instantiated again for as long as it remains in the journal. Consequently, if a Page has initialization logic that needs to be called every time the Page is navigated to, you should move it from the constructor into a handler for the Loaded event. As shown in the following figure, the Loaded and Unloaded events are still raised each time a Page is navigated to and from, respectively.



When a Page is not kept alive, you should not do either of the following:

- Store a reference to it, or any part of it.
- Register event handlers with events that are not implemented by it.

Doing either of these will create references that force the Page to be retained in memory, even after it has been removed from the journal.

In general, you should prefer the default Page behavior of not keeping a Page alive. However, this has state implications that are discussed in the next section.

Retaining Content State with Navigation History

If a Page is not kept alive, and it has controls that collect data from the user, what happens to the data if a user navigates away from and back to the Page? From a user experience perspective, the user should expect to see the data they entered previously. Unfortunately, because a new instance of the Page is created with each navigation, the controls that collected the data are reinstantiated and the data is lost.

Fortunately, the journal provides support for remembering data across Page navigations, including control data. Specifically, the journal entry for each Page acts as a temporary container for the associated Page state. The following steps outline how this support is used when a Page is navigated from:

1. An entry for the current Page is added to the journal.

2. The state of the Page is stored with the journal entry for that page, which is added to the back stack.

3. The new Page is navigated to.

When the page Page is navigated back to, using the journal, the following steps take place:

1. The Page (the top journal entry on the back stack) is instantiated.

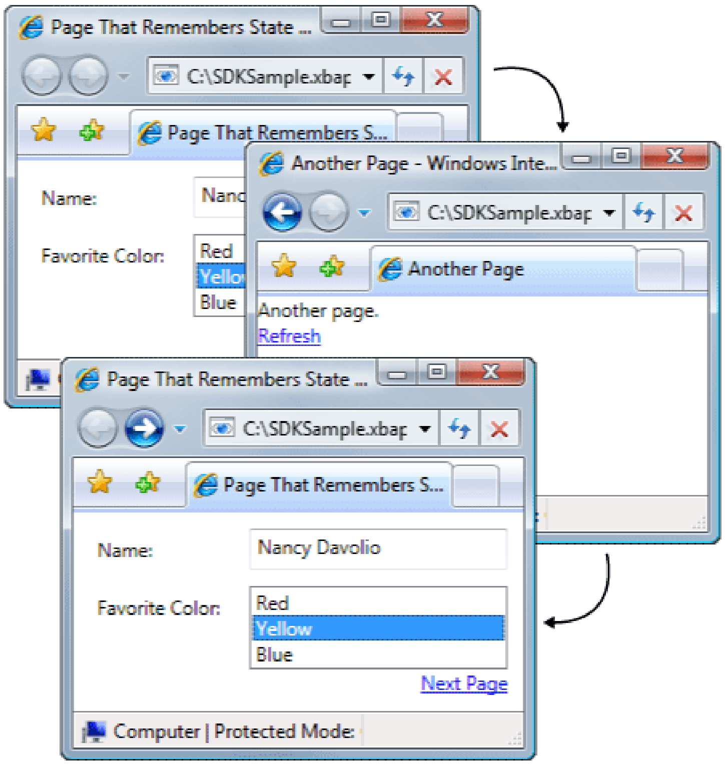
2. The Page is refreshed with the state that was stored with the journal entry for the Page.

3. The Page is navigated back to.

WPF automatically uses this support when the following controls are used on a Page:

- CheckBox
- ComboBox
- Expander
- Frame
- ListBox
- ListBoxItem
- MenuItem
- ProgressBar
- RadioButton
- Slider
- TabControl
- TabItem
- TextBox

If a Page uses these controls, data entered into them is remembered across Page navigations, as demonstrated by the **Favorite Color**ListBox in the following figure.



When a Page has controls other than those in the preceding list, or when state is stored in custom objects, you need to write code to cause the journal to remember state across Page navigations.

If you need to remember small pieces of state across Page navigations, you can use dependency properties (see DependencyProperty) that are configured with the FrameworkPropertyMetadata.Journal metadata flag.

If the state that your Page needs to remember across navigations comprises multiple pieces of data, you may find it less code intensive to encapsulate your state in a single class and implement the IProvideCustomContentState interface.

If you need to navigate through various states of a single Page, without navigating from the Page itself, you can use IProvideCustomContentState and NavigationService.AddBackEntry.

Cookies

Another way that WPF applications can store data is with cookies, which are created, updated, and deleted by using the SetCookie and GetCookie methods. The cookies that you can create in WPF are the same cookies that other types of Web applications use; cookies are arbitrary pieces of data that are stored by an application on a client machine either during or across application sessions. Cookie data typically takes the form of a name/value pair in the following format.

Name `=` *Value*

When the data is passed to SetCookie, along with the Uri of the location for which the cookie should be set, a cookie is created in-memory, and it is only available for the duration of the current application session. This type of cookie is referred to as a *session cookie*.

To store a cookie across application sessions, an expiration date must be added to the cookie, using the following format.

NAME `=` *VALUE* `; expires=DAY, DD-MMM-YYYY HH:MM:SS GMT`

A cookie with an expiration date is stored in the current Windows installation's Temporary Internet Files folder until the cookie expires. Such a cookie is known as a *persistent cookie* because it persists across application sessions.

You retrieve both session and persistent cookies by calling the GetCookie method, passing the Uri of the location where the cookie was set with the SetCookie method.

The following are some of the ways that cookies are supported in WPF:

- WPF standalone applications and XBAPs can both create and manage cookies.
- Cookies that are created by an XBAP can be accessed from the browser.
- XBAPs from the same domain can create and share cookies.
- XBAPs and HTML pages from the same domain can create and share cookies.
- Cookies are dispatched when XBAPs and loose XAML pages make Web requests.
- Both top-level XBAPs and XBAPs hosted in IFRAMES can access cookies.
- Cookie support in WPF is the same for all supported browsers.
- In Internet Explorer, P3P policy that pertains to cookies is honored by WPF, particularly with respect to first-party and third-party XBAPs.

Structured Navigation

If you need to pass data from one Page to another, you can pass the data as arguments to a non-default constructor of the Page. Note that if you use this technique, you must keep the Page alive; if not, the next time you navigate to the Page, WPF reinstantiates the Page by using the default constructor.

Alternatively, your Page can implement properties that are set with the data that needs to be passed. Things become tricky, however, when a Page needs to pass data back to the Page that navigated to it. The problem is that navigation doesn't natively support mechanisms for guaranteeing that a Page will be returned to after it is navigated from. Essentially, navigation doesn't support call/return semantics. To solve this problem, WPF provides the PageFunction<T> class that you can use to ensure that a Page is returned to in a predictable and structured fashion. For more information, see Structured Navigation Overview.

The NavigationWindow Class

To this point, you've seen the gamut of navigation services that you are most likely to use to build applications with navigable content. These services were discussed in the context of XBAPs, although they are not limited to XBAPs. Modern operating systems and Windows applications take advantage of the browser experience of modern users to incorporate browser-style navigation into standalone applications. Common examples include:

- **Word Thesaurus**: Navigate word choices.
- **File Explorer**: Navigate files and folders.
- **Wizards**: Breaking down a complex task into multiple pages that can be navigated between. An example is the Windows Components Wizard that handles adding and removing Windows features.

To incorporate browser-style navigation into your standalone applications, you can use the NavigationWindow class. NavigationWindow derives from Window and extends it with the same support for navigation that XBAPs provide. You can use NavigationWindow as either the main window of your standalone application or as a secondary window such as a dialog box.

To implement a NavigationWindow, as with most top-level classes in WPF (Window, Page, and so on), you use a combination of markup and code-behind. This is shown in the following example.

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

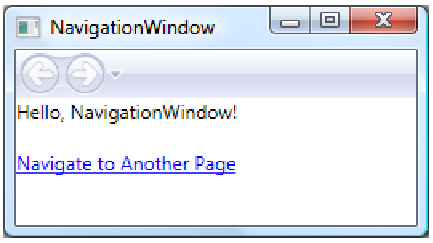
```
using System.Windows.Navigation;

namespace SDKSample
{
    public partial class MainWindow : NavigationWindow
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

```
Namespace SDKSample
    Partial Public Class MainWindow
        Inherits NavigationWindow
        Public Sub New()
            InitializeComponent()
        End Sub
    End Class
End Namespace
```

This code creates a NavigationWindow that automatically navigates to a Page (HomePage.xaml) when the NavigationWindow is opened. If the NavigationWindow is the main application window, you can use the `StartupUri` attribute to launch it. This is shown in the following markup.

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

The following figure shows the NavigationWindow as the main window of a standalone application.



From the figure, you can see that the NavigationWindow has a title, even though it wasn't set in the NavigationWindow implementation code from the preceding example. Instead, the title is set using the WindowTitle property, which is shown in the following code.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Home Page"
    WindowTitle="NavigationWindow">
```

```
</Page>
```

Setting the WindowWidth and WindowHeight properties also affects the NavigationWindow.

Usually, you implement your own NavigationWindow when you need to customize either its behavior or its appearance. If you do neither, you can use a shortcut. If you specify the pack URI of a Page as the StartupUri in a standalone application, Application automatically creates a NavigationWindow to host the Page. The following markup shows how to enable this.

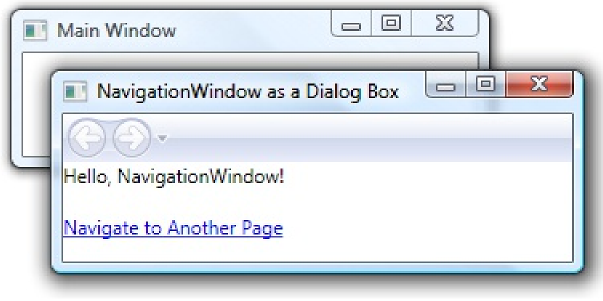
```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    StartupUri="HomePage.xaml" />
```

If you want a secondary application window such as a dialog box to be a NavigationWindow, you can use the code in the following example to open it.

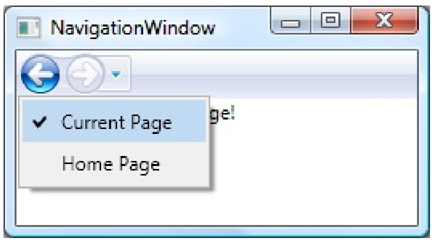
```
// Open a navigation window as a dialog box
NavigationWindowDialogBox dlg = new NavigationWindowDialogBox();
dlg.Source = new Uri("HomePage.xaml", UriKind.Relative);
dlg.Owner = this;
dlg.ShowDialog();
```

```
' Open a navigation window as a dialog box
Dim dlg As New NavigationWindowDialogBox()
dlg.Source = New Uri("HomePage.xaml", UriKind.Relative)
dlg.Owner = Me
dlg.ShowDialog()
```

The following figure shows the result.



As you can see, NavigationWindow displays Internet Explorer-style **Back** and **Forward** buttons that allow users to navigate the journal. These buttons provide the same user experience, as shown in the following figure.



If your pages provide their own journal navigation support and UI, you can hide the **Back** and **Forward** buttons displayed by NavigationWindow by setting the value of the ShowsNavigationUI property to `false`.

Alternatively, you can use customization support in WPF to replace the UI of the NavigationWindow itself.

The Frame Class

Both the browser and NavigationWindow are windows that host navigable content. In some cases, applications have content that does not need to be hosted by an entire window. Instead, such content be hosted inside other content. You can insert navigable content into other content by using the Frame class. Frame provides the same support as NavigationWindow and XBAPs.

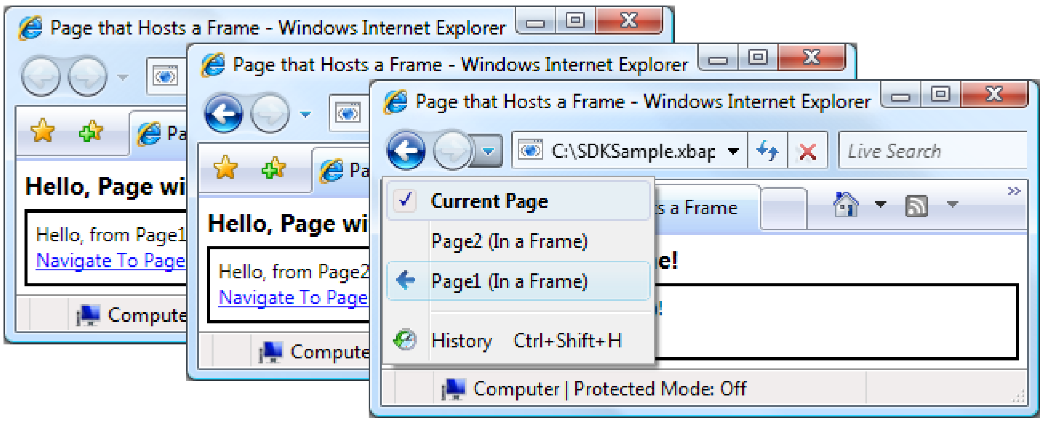
The following example shows how to add a Frame to a Page declaratively by using the `Frame` element.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  WindowTitle="Page that Hosts a Frame"
  WindowWidth="250"
  WindowHeight="250">
```

```
<Frame Source="FramePage1.xaml" />
```

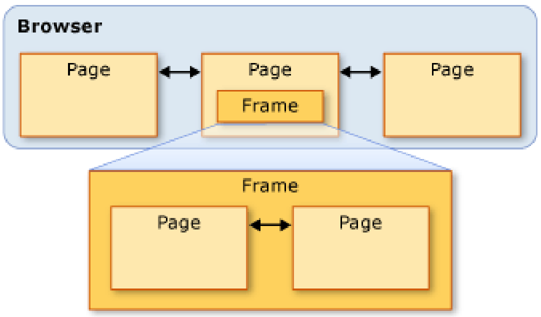
```
</Page>
```

This markup sets the `Source` attribute of the `Frame` element with a pack URI for the Page that the Frame should initially navigate to. The following figure shows an XBAP with a Page that has a Frame that has navigated between several pages.



You don't only have to use Frame inside the content of a Page. It is also common to host a Frame inside the content of a Window.

By default, Frame only uses its own journal in the absence of another journal. If a Frame is part of content that is hosted inside either a NavigationWindow or an XBAP, Frame uses the journal that belongs to the NavigationWindow or XBAP. Sometimes, though, a Frame might need to be responsible for its own journal. One reason to do so is to allow journal navigation within the pages that are hosted by a Frame. This is illustrated by the following figure.



In this case, you can configure the Frame to use its own journal by setting the JournalOwnership property of the

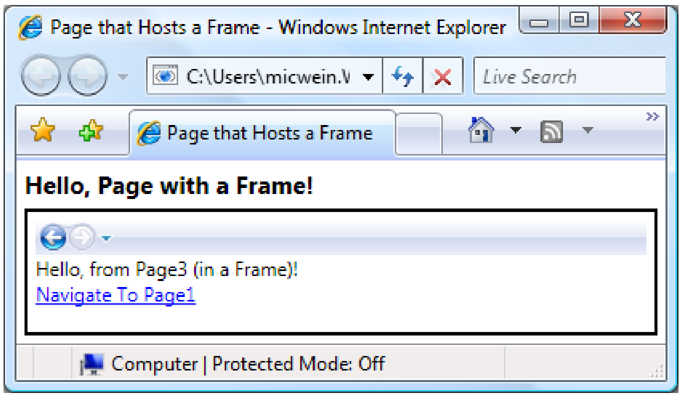
Frame to OwnsJournal. This is shown in the following markup.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  WindowTitle="Page that Hosts a Frame"
  WindowWidth="250"
  WindowHeight="250">
```

```
<Frame Source="FramePage1.xaml" JournalOwnership="OwnsJournal" />
```

```
</Page>
```

The following figure illustrates the effect of navigating within a Frame that uses its own journal.


Page that Hosts a Frame - Windows Internet Explorer
C:\Users\micwein.\
Live Search
Page that Hosts a Frame
Computer | Protected Mode: Off

Hello, Page with a Frame!

Hello, from Page3 (in a Frame)!
Navigate To Page1

Notice that the journal entries are shown by the navigation UI in the Frame, rather than by Internet Explorer.

NOTE

If a Frame is part of content that is hosted in a Window, Frame uses its own journal and, consequently, displays its own navigation UI.

If your user experience requires a Frame to provide its own journal without showing the navigation UI, you can hide the navigation UI by setting the NavigationUIVisibility to Hidden. This is shown in the following markup.

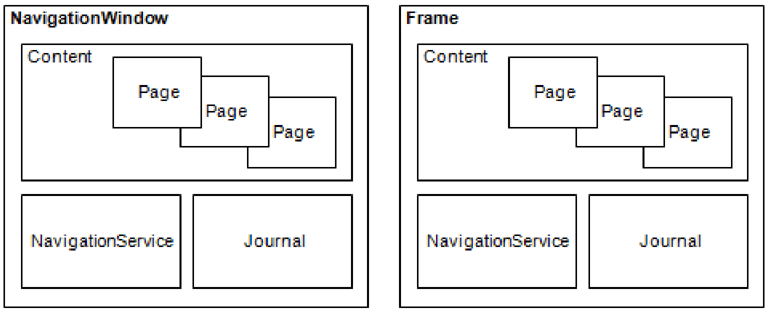
```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  WindowTitle="Page that Hosts a Frame"
  WindowWidth="250"
  WindowHeight="250">
```

```
<Frame
  Source="FramePage1.xaml"
  JournalOwnership="OwnsJournal"
  NavigationUIVisibility="Hidden" />
```

```
</Page>
```

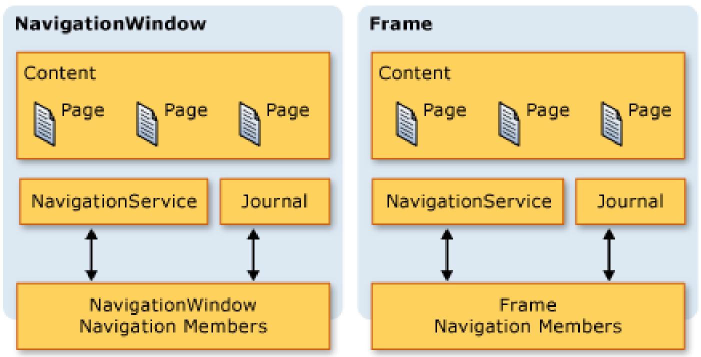
Navigation Hosts

Frame and NavigationWindow are classes that are known as navigation hosts. A *navigation host* is a class that can navigate to and display content. To accomplish this, each navigation host uses its own NavigationService and journal. The basic construction of a navigation host is shown in the following figure.



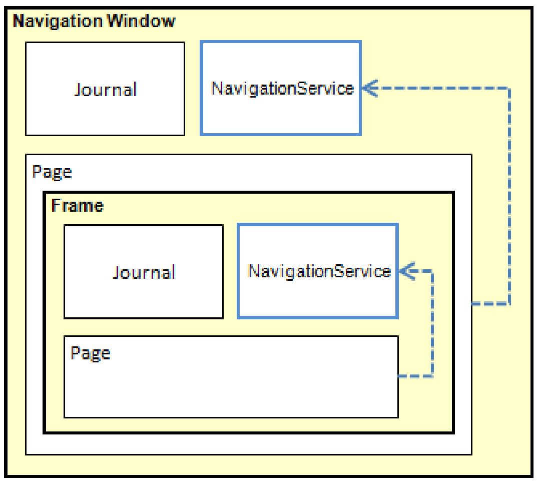
Essentially, this allows NavigationWindow and Frame to provide the same navigation support that an XBAP provides when hosted in the browser.

Besides using NavigationService and a journal, navigation hosts implement the same members that NavigationService implements. This is illustrated by the following figure.



This allows you to program navigation support directly against them. You may consider this if you need to provide a custom navigation UI for a Frame that is hosted in a Window. Furthermore, both types implement additional, navigation-related members, including `BackStack` (NavigationWindow.BackStack, Frame.BackStack) and `ForwardStack` (NavigationWindow.ForwardStack, Frame.ForwardStack), which allow you to enumerate the journal entries in the back stack and forward stack, respectively.

As mentioned earlier, more than one journal can exist within an application. The following figure provides an example of when this can happen.



Navigating to Content Other than XAML Pages

Throughout this topic, Page and pack XBAPs have been used to demonstrate the various navigation capabilities of WPF. However, a Page that is compiled into an application is not the only type of content that can be navigated to, and pack XBAPs aren't the only way to identify content.

As this section demonstrates, you can also navigate to loose XAML files, HTML files, and objects.

Navigating to Loose XAML Files

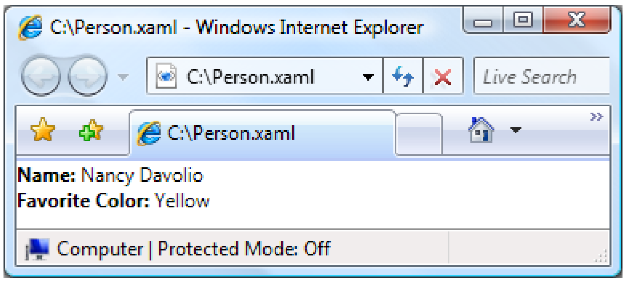
A loose XAML file is a file with the following characteristics:

- Contains only XAML (that is, no code).
- Has an appropriate namespace declaration.
- Has the .xaml file name extension.

For example, consider the following content that is stored as a loose XAML file, Person.xaml.

```
<!-- Person.xaml -->
<TextBlock xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <TextBlock FontWeight="Bold">Name:</TextBlock>
  <TextBlock>Nancy Davolio</TextBlock>
  <LineBreak />
  <TextBlock FontWeight="Bold">Favorite Color:</TextBlock>
  <TextBlock>Yellow</TextBlock>
</TextBlock>
```

When you double-click the file, the browser opens and navigates to and displays the content. This is shown in the following figure.


C:\Person.xaml - Windows Internet Explorer
C:\Person.xaml
Live Search
C:\Person.xaml
Name: Nancy Davolio
Favorite Color: Yellow
Computer | Protected Mode: Off

You can display a loose XAML file from the following:

- A Web site on the local machine, the intranet, or the Internet.
- A Universal Naming Convention (UNC) file share.
- The local disk.

A loose XAML file can be added to the browser's favorites, or be the browser's home page.

> **NOTE**
>
> For more information about publishing and launching loose XAML pages, see Deploying a WPF Application.

One limitation with respect to loose XAML is that you can only host content that is safe to run in partial trust. For example, `Window` cannot be the root element of a loose XAML file. For more information, see WPF Partial Trust Security.

Navigating to HTML Files by Using Frame

As you might expect, you can also navigate to HTML. You simply need to provide a URI that uses the http scheme. For example, the following XAML shows a Frame that navigates to an HTML page.

```
<Frame Source="http://www.microsoft.com/default.aspx" />
```

Navigating to HTML requires special permissions. For example, you can't navigate from an XBAP that is running in the Internet zone partial trust security sandbox. For more information, see WPF Partial Trust Security.

Navigating to HTML Files by Using the WebBrowser Control

The WebBrowser control supports HTML document hosting, navigation and script/managed code interoperability. For detailed information regarding the WebBrowser control, see WebBrowser.

Like Frame, navigating to HTML using WebBrowser requires special permissions. For example, from a partial-trust application, you can navigate only to HTML located at the site of origin. For more information, see WPF Partial Trust Security.

Navigating to Custom Objects

If you have data that is stored as custom objects, one way to display that data is to create a Page with content that is bound to those objects (see Data Binding Overview). If you don't need the overhead of creating an entire page just to display the objects, you can navigate directly to them instead.

Consider the `Person` class that is implemented in the following code.

```
using System.Windows.Media;

namespace SDKSample
{
    public class Person
    {
        string name;
        Color favoriteColor;

        public Person() { }
        public Person(string name, Color favoriteColor)
        {
            this.name = name;
            this.favoriteColor = favoriteColor;
        }

        public string Name
        {
            get { return this.name; }
            set { this.name = value; }
        }

        public Color FavoriteColor
        {
            get { return this.favoriteColor; }
            set { this.favoriteColor = value; }
        }
    }
}
```

```
Namespace SDKSample
    Public Class Person
        Private _name As String
        Private _favoriteColor As Color

        Public Sub New()
        End Sub
        Public Sub New(ByVal name As String, ByVal favoriteColor As Color)
            Me._name = name
            Me._favoriteColor = favoriteColor
        End Sub

        Public Property Name() As String
            Get
                Return Me._name
            End Get
            Set(ByVal value As String)
                Me._name = value
            End Set
        End Property

        Public Property FavoriteColor() As Color
            Get
                Return Me._favoriteColor
            End Get
            Set(ByVal value As Color)
                Me._favoriteColor = value
            End Set
        End Property
    End Class
End Namespace
```

To navigate to it, you call the NavigationWindow.Navigate method, as demonstrated by the following code.

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="SDKSample.HomePage"
  WindowTitle="Page that Navigates to an Object">
```

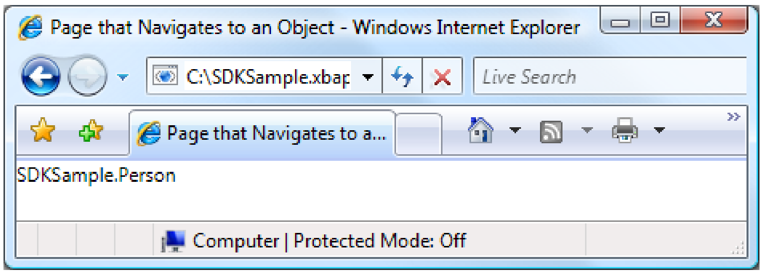
```
<Hyperlink Name="hyperlink" Click="hyperlink_Click">
  Navigate to Nancy Davolio
</Hyperlink>
```

```
</Page>
```

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

namespace SDKSample
{
    public partial class HomePage : Page
    {
        public HomePage()
        {
            InitializeComponent();
        }

        void hyperlink_Click(object sender, RoutedEventArgs e)
        {
            Person person = new Person("Nancy Davolio", Colors.Yellow);
            this.NavigationService.Navigate(person);
        }
    }
}
```

```
Namespace SDKSample
    Partial Public Class HomePage
        Inherits Page
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub hyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            Dim person As New Person("Nancy Davolio", Colors.Yellow)
            Me.NavigationService.Navigate(person)
        End Sub
    End Class
End Namespace
```

The following figure shows the result.

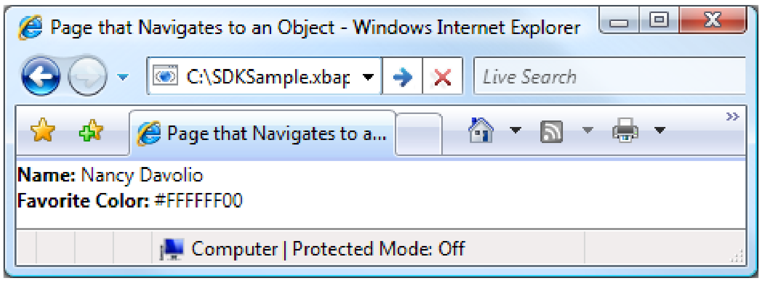

Page that Navigates to an Object - Windows Internet Explorer
C:\SDKSample.xbap
Live Search
Page that Navigates to a...
Computer | Protected Mode: Off

SDKSample.Person

From this figure, you can see that nothing useful is displayed. In fact, the value that is displayed is the return value of the `ToString` method for the **Person** object; by default, this is the only value that WPF can use to represent your object. You could override the `ToString` method to return more meaningful information, although it will still only be a string value. One technique you can use that takes advantage of the presentation capabilities of WPF is to use a data template. You can implement a data template that WPF can associate with an object of a particular type. The following code shows a data template for the `Person` object.

```
<Application
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:SDKSample"
    x:Class="SDKSample.App"
    StartupUri="HomePage.xaml">

  <Application.Resources>

    <!-- Data Template for the Person Class -->
    <DataTemplate DataType="{x:Type local:Person}">
      <TextBlock xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
        <TextBlock FontWeight="Bold">Name:</TextBlock>
        <TextBlock Text="{Binding Path=Name}" />
        <LineBreak />
        <TextBlock FontWeight="Bold">Favorite Color:</TextBlock>
        <TextBlock Text="{Binding Path=FavoriteColor}" />
      </TextBlock>
    </DataTemplate>

  </Application.Resources>

</Application>
```

Here, the data template is associated with the `Person` type by using the `x:Type` markup extension in the `DataType` attribute. The data template then binds `TextBlock` elements (see TextBlock) to the properties of the `Person` class. The following figure shows the updated appearance of the `Person` object.


Page that Navigates to an Object - Windows Internet Explorer
C:\SDKSample.xbap
Live Search
Page that Navigates to a...
Computer | Protected Mode: Off

Name: Nancy Davolio
Favorite Color: #FFFFFF00

An advantage of this technique is the consistency you gain by being able to reuse the data template to display your objects consistently anywhere in your application.

For more information on data templates, see Data Templating Overview.

Security

WPF navigation support allows XBAPs to be navigated to across the Internet, and it allows applications to host third-party content. To protect both applications and users from harmful behavior, WPF provides a variety of security features that are discussed in Security and WPF Partial Trust Security.

See also

- SetCookie
- GetCookie
- Application Management Overview
- Pack URIs in WPF
- Structured Navigation Overview
- Navigation Topologies Overview
- How-to Topics
- Deploying a WPF Application

Structured Navigation Overview

1/23/2019 • 10 minutes to read • Edit Online

Content that can be hosted by an XAML browser application (XBAP), a Frame, or a NavigationWindow is composed of pages that can be identified by pack uniform resource identifiers (URIs) and navigated to by hyperlinks. The structure of pages and the ways in which they can be navigated, as defined by hyperlinks, is known as a navigation topology. Such a topology suits a variety of application types, particularly those that navigate through documents. For such applications, the user can navigate from one page to another page without either page needing to know anything about the other.

However, other types of applications have pages that do need to know when they have been navigated between. For example, consider a human resources application that has one page to list all the employees in an organization—the "List Employees" page. This page could also allow users to add a new employee by clicking a hyperlink. When clicked, the page navigates to an "Add an Employee" page to gather the new employee's details and return them to the "List Employees" page to create the new employee and update the list. This style of navigation is similar to calling a method to perform some processing and return a value, which is known as structured programming. As such, this style of navigation is known as *structured navigation*.

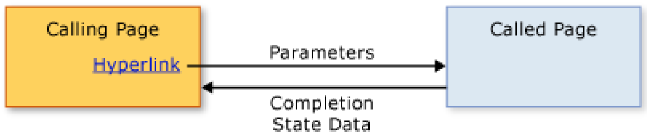
The Page class doesn't implement support for structured navigation. Instead, the PageFunction<T> class derives from Page and extends it with the basic constructs required for structured navigation. This topic shows how to establish structured navigation using PageFunction<T>.

Structured Navigation

When one page calls another page in a structured navigation, some or all of the following behaviors are required:

- The calling page navigates to the called page, optionally passing parameters required by the called page.
- The called page, when a user has completed using the calling page, returns specifically to the calling page, optionally:
  - Returning state information that describes how the calling page was completed (for example, whether a user pressed an OK button or a Cancel button).
  - Returning that data that was collected from the user (for example, new employee details).
- When the calling page returns to the called page, the called page is removed from navigation history to isolate one instance of a called page from another.

These behaviors are illustrated by the following figure.



You can implement these behaviors by using a PageFunction<T> as the called page.

Structured Navigation with PageFunction

This topic shows how to implement the basic mechanics of structured navigation involving a single PageFunction<T>. In this sample, a Page calls a PageFunction<T> to get a String value from the user and return it.

Creating a Calling Page

The page that calls a PageFunction<T> can be either a Page or a PageFunction<T>. In this example, it is a Page, as shown in the following code.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="StructuredNavigationSample.CallingPage"
    WindowTitle="Calling Page"
    WindowWidth="250" WindowHeight="150">
```

```
</Page>
```

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

namespace StructuredNavigationSample
{
    public partial class CallingPage : Page
    {
        public CallingPage()
        {
            InitializeComponent();
```

```
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Navigation

Namespace StructuredNavigationSample

Public Class CallingPage
    Inherits Page
    Public Sub New()
        Me.InitializeComponent()
```

```
}
```

```
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

Creating a Page Function to Call

Because the calling page can use the called page to collect and return data from the user, PageFunction<T> is implemented as a generic class whose type argument specifies the type of the value that the called page will return. The following code shows the initial implementation of the called page, using a PageFunction<T>, which returns a String.

```xml
<PageFunction
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    x:Class="StructuredNavigationSample.CalledPageFunction"
    x:TypeArguments="sys:String"
    Title="Page Function"
    WindowWidth="250" WindowHeight="150">

  <Grid Margin="10">

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

    <!-- Data -->
    <Label Grid.Column="0" Grid.Row="0">DataItem1:</Label>
    <TextBox Grid.Column="1" Grid.Row="0" Name="dataItem1TextBox"></TextBox>

    <!-- Accept/Cancel buttons -->
    <TextBlock Grid.Column="1" Grid.Row="1" HorizontalAlignment="Right">
      <Button Name="okButton" IsDefault="True" MinWidth="50">OK</Button>
      <Button Name="cancelButton" IsCancel="True" MinWidth="50">Cancel</Button>
    </TextBlock>

  </Grid>

</PageFunction>
```

```csharp
using System;
using System.Windows;
using System.Windows.Navigation;

namespace StructuredNavigationSample
{
    public partial class CalledPageFunction : PageFunction<String>
    {
        public CalledPageFunction()
        {
            InitializeComponent();
        }
```

```vb
Imports System
Imports System.Windows
Imports System.Windows.Navigation

Namespace StructuredNavigationSample

Public Class CalledPageFunction
    Inherits PageFunction(Of String)
    Public Sub New()
        Me.InitializeComponent()
    End Sub
```

```csharp
    }
}
```

```
End Class

End Namespace
```

The declaration of a PageFunction<T> is similar to the declaration of a Page with the addition of the type arguments. As you can see from the code example, the type arguments are specified in both XAML markup, using the `x:TypeArguments` attribute, and code-behind, using standard generic type argument syntax.

You don't have to use only .NET Framework classes as type arguments. A PageFunction<T> could be called to gather domain-specific data that is abstracted as a custom type. The following code shows how to use a custom type as a type argument for a PageFunction<T>.

```
namespace SDKSample
{
    public class CustomType
    {
```

```
Public Class CustomType
```

```
    }
}
```

```
End Class
```

```
<PageFunction
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:SDKSample"
    x:Class="SDKSample.CustomTypePageFunction"
    x:TypeArguments="local:CustomType">
```

```
</PageFunction>
```

```
using System.Windows.Navigation;

namespace SDKSample
{
    public partial class CustomTypePageFunction : PageFunction<CustomType>
    {
```

```
Partial Public Class CustomTypePageFunction
    Inherits System.Windows.Navigation.PageFunction(Of CustomType)
```

```
    }
}
```

```
End Class
```

The type arguments for the PageFunction<T> provide the foundation for the communication between a calling page and the called page, which are discussed in the following sections.

As you'll see, the type that is identified with the declaration of a PageFunction<T> plays an important role in returning data from a PageFunction<T> to the calling page.

Calling a PageFunction and Passing Parameters

To call a page, the calling page must instantiate the called page and navigate to it using the Navigate method. This allows the calling page to pass initial data to the called page, such as default values for the data being gathered by the called page.

The following code shows the called page with a non-default constructor to accept parameters from the calling page.

```
using System;
using System.Windows;
using System.Windows.Navigation;

namespace StructuredNavigationSample
{
    public partial class CalledPageFunction : PageFunction<String>
    {
```

```
Imports System
Imports System.Windows
Imports System.Windows.Navigation

Namespace StructuredNavigationSample

Public Class CalledPageFunction
    Inherits PageFunction(Of String)
```

```
public CalledPageFunction(string initialDataItem1Value)
{
    InitializeComponent();
```

```
Public Sub New(ByVal initialDataItem1Value As String)
    Me.InitializeComponent()
```

```
    // Set initial value
    this.dataItem1TextBox.Text = initialDataItem1Value;
}
```

```
    ' Set initial value
    Me.dataItem1TextBox.Text = initialDataItem1Value
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

The following code shows the calling page handling the Click event of the Hyperlink to instantiate the called page and pass it an initial string value.

```
<Hyperlink Name="pageFunctionHyperlink">Call Page Function</Hyperlink>
```

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

namespace StructuredNavigationSample
{
    public partial class CallingPage : Page
    {
        public CallingPage()
        {
            InitializeComponent();
            this.pageFunctionHyperlink.Click += new RoutedEventHandler(pageFunctionHyperlink_Click);
        }
        void pageFunctionHyperlink_Click(object sender, RoutedEventArgs e)
        {

            // Instantiate and navigate to page function
            CalledPageFunction CalledPageFunction = new CalledPageFunction("Initial Data Item Value");
```

```
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Navigation

Namespace StructuredNavigationSample

Public Class CallingPage
    Inherits Page
    Public Sub New()
        Me.InitializeComponent()
        AddHandler Me.pageFunctionHyperlink.Click, New RoutedEventHandler(AddressOf
Me.pageFunctionHyperlink_Click)
    End Sub
    Private Sub pageFunctionHyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
```

```
}
```

```
End Sub
```

```
    }
}
```

```
End Class

End Namespace
```

You are not required to pass parameters to the called page. Instead, you could do the following:

- From the calling page:
  1. Instantiate the called PageFunction<T> using the default constructor.
  2. Store the parameters in Properties.
  3. Navigate to the called PageFunction<T>.
- From the called PageFunction<T>:
  - Retrieve and use the parameters stored in Properties.

But, as you'll see shortly, you'll still need use code to instantiate and navigate to the called page to collect the data returned by the called page. For this reason, the PageFunction<T> needs to be kept alive; otherwise, the next time you navigate to the PageFunction<T>, WPF instantiates the PageFunction<T> using the default constructor.

Before the called page can return, however, it needs to return data that can be retrieved by the calling page.

Returning Task Result and Task Data from a Task to a Calling Page

Once the user has finished using the called page, signified in this example by pressing either the OK or Cancel buttons, the called page needs to return. Since the calling page used the called page to collect data from the user, the calling page requires two types of information:

1. Whether the user canceled the called page (by pressing either the OK button or the Cancel button in this example). This allows the calling page to determine whether to process the data that the calling page gathered from the user.
2. The data that was provided by the user.

To return information, PageFunction<T> implements the OnReturn method. The following code shows how to call it.

```
using System;
using System.Windows;
using System.Windows.Navigation;

namespace StructuredNavigationSample
{
    public partial class CalledPageFunction : PageFunction<String>
    {
```

```
Imports System
Imports System.Windows
Imports System.Windows.Navigation

Namespace StructuredNavigationSample

Public Class CalledPageFunction
    Inherits PageFunction(Of String)
```

```
        void okButton_Click(object sender, RoutedEventArgs e)
        {
            // Accept when Ok button is clicked
            OnReturn(new ReturnEventArgs<string>(this.dataItem1TextBox.Text));
        }

        void cancelButton_Click(object sender, RoutedEventArgs e)
        {
            // Cancel
            OnReturn(null);
        }
    }
}
```

```
    Private Sub okButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
        ' Accept when Ok button is clicked
        Me.OnReturn(New ReturnEventArgs(Of String)(Me.dataItem1TextBox.Text))
    End Sub

    Private Sub cancelButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
        ' Cancel
        Me.OnReturn(Nothing)
    End Sub
End Class

End Namespace
```

In this example, if a user presses the Cancel button, a value of `null` is returned to the calling page. If the OK button is pressed instead, the string value provided by the user is returned. OnReturn is a `protected virtual` method that you call to return your data to the calling page. Your data needs to be packaged in an instance of the generic ReturnEventArgs<T> type, whose type argument specifies the type of value that Result returns. In this way, when you declare a PageFunction<T> with a particular type argument, you are stating that a PageFunction<T> will return an instance of the type that is specified by the type argument. In this example, the type argument and, consequently, the return value is of type String.

When OnReturn is called, the calling page needs some way of receiving the return value of the PageFunction<T>. For this reason, PageFunction<T> implements the Return event for calling pages to handle. When OnReturn is called, Return is raised, so the calling page can register with Return to receive the notification.

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

namespace StructuredNavigationSample
{
    public partial class CallingPage : Page
    {
```

```
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Navigation

Namespace StructuredNavigationSample

Public Class CallingPage
    Inherits Page
```

```
        void pageFunctionHyperlink_Click(object sender, RoutedEventArgs e)
        {

            // Instantiate and navigate to page function
            CalledPageFunction CalledPageFunction = new CalledPageFunction("Initial Data Item Value");
            CalledPageFunction.Return += pageFunction_Return;
            this.NavigationService.Navigate(CalledPageFunction);
        }
        void pageFunction_Return(object sender, ReturnEventArgs<string> e)
        {
            this.pageFunctionResultsTextBlock.Visibility = Visibility.Visible;

            // Display result
            this.pageFunctionResultsTextBlock.Text = (e != null ? "Accepted" : "Canceled");

            // If page function returned, display result and data
            if (e != null)
            {
                this.pageFunctionResultsTextBlock.Text += "\n" + e.Result;
            }
        }
    }
}
```

```
    Private Sub pageFunctionHyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
        ' Instantiate and navigate to page function
        Dim calledPageFunction As New CalledPageFunction("Initial Data Item Value")
        AddHandler calledPageFunction.Return, New ReturnEventHandler(Of String)(AddressOf
Me.calledPageFunction_Return)
        MyBase.NavigationService.Navigate(calledPageFunction)
    End Sub
    Private Sub calledPageFunction_Return(ByVal sender As Object, ByVal e As ReturnEventArgs(Of String))

        Me.pageFunctionResultsTextBlock.Visibility = Windows.Visibility.Visible

        ' Display result
        Me.pageFunctionResultsTextBlock.Text = IIf((Not e Is Nothing), "Accepted", "Canceled")

        ' If page function returned, display result and data
        If (Not e Is Nothing) Then
            Me.pageFunctionResultsTextBlock.Text = (Me.pageFunctionResultsTextBlock.Text & ChrW(10) &
e.Result)
        End If

    End Sub
End Class

End Namespace
```

Removing Task Pages When a Task Completes

When a called page returns, and the user didn't cancel the called page, the calling page will process the data that was provided by the user and also returned from the called page. Data acquisition in this way is usually an isolated activity; when the called page returns, the calling page needs to create and navigate to a new calling page to capture more data.

However, unless a called page is removed from the journal, a user will be able to navigate back to a previous instance of the calling page. Whether a PageFunction<T> is retained in the journal is determined by the RemoveFromJournal property. By default, a page function is automatically removed when OnReturn is called because RemoveFromJournal is set to `true`. To keep a page function in navigation history after OnReturn is called, set RemoveFromJournal to `false`.

Other Types of Structured Navigation

This topic illustrates the most basic use of a PageFunction<T> to support call/return structured navigation. This foundation provides you with the ability to create more complex types of structured navigation.

For example, sometimes multiple pages are required by a calling page to gather enough data from a user or to perform a task. The use of multiple pages is referred to as a "wizard".

In other cases, applications may have complex navigation topologies that depend on structured navigation to operate effectively. For more information, see Navigation Topologies Overview.

See also

- PageFunction<T>
- NavigationService
- Navigation Topologies Overview

Navigation Topologies Overview

1/23/2019 • 5 minutes to read • Edit Online

This overview provides an introduction to navigation topologies in WPF. Three common navigation topologies, with samples, are subsequently discussed.

NOTE

Before reading this topic, you should be familiar with the concept of structured navigation in WPF using page functions. For more information on both of these topics, see Structured Navigation Overview.

This topic contains the following sections:

Navigation Topologies

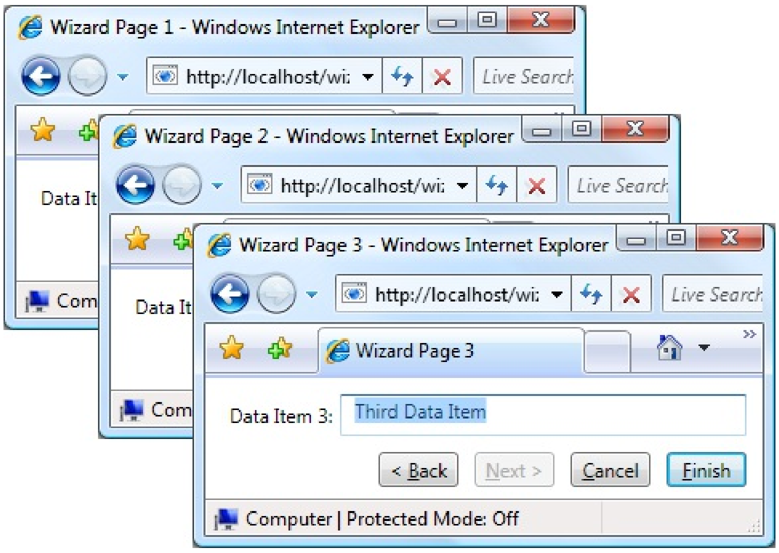
In WPF, navigation typically consists of pages (Page) with hyperlinks (Hyperlink) that navigate to other pages when clicked. Pages that are navigated to are identified by uniform resource identifiers (URIs) (see Pack URIs in WPF). Consider the following simple example that shows pages, hyperlinks, and uniform resource identifiers (URIs):

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" WindowTitle="Page1">
  <Hyperlink NavigateUri="Page2.xaml">Navigate to Page2</Hyperlink>
</Page>
```

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" WindowTitle="Page2">
  <Hyperlink NavigateUri="Page1.xaml">Navigate to Page1</Hyperlink>
</Page>
```

These pages are arranged in a *navigation topology* whose structure is determined by how you can navigate between the pages. This particular navigation topology is suitable in simple scenarios, although navigation can require more complex topologies, some of which can only be defined when an application is running.

This topic covers three common navigation topologies: *fixed linear*, *fixed hierarchical*, and *dynamically generated*. Each navigation topology is demonstrated with a sample that has a UI like the one that is shown in the following figure:


Wizard Page 1 - Windows Internet Explorer
Wizard Page 2 - Windows Internet Explorer
Wizard Page 3 - Windows Internet Explorer
http://localhost/wi:
Live Search
Wizard Page 3
Data Item 3:
Third Data Item
< Back
Next >
Cancel
Finish
Computer | Protected Mode: Off

Structured Navigation Topologies

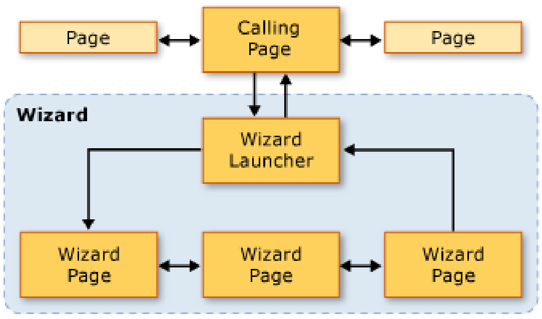
There are two broad types of navigation topologies:

- **Fixed Topology**: defined at compile time and does not change at run time. Fixed topologies are useful for navigation through a fixed sequence of pages in either a linear or hierarchical order.
- **Dynamic Topology**: defined at run time based on input that is collected from the user, the application, or the system. Dynamic topologies are useful when pages can be navigated in different sequences.

Although it is possible to create navigation topologies using pages, the samples use page functions because they provide additional support that simplifies support for passing and returning data through the pages of a topology.

Navigation over a Fixed Linear Topology

A fixed linear topology is analogous to the structure of a wizard that has one or more wizard pages that are navigated in a fixed sequence. The following figure shows the high-level structure and flow of a wizard with a fixed linear topology.



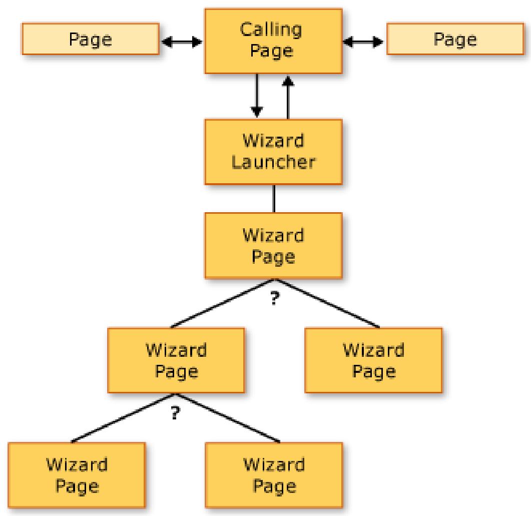
The typical behaviors for navigating over a fixed linear topology include the following:

- Navigating from the calling page to a launcher page that initializes the wizard and navigates to the first wizard page. A launcher page (a UI-less PageFunction<T>) is not required, since a calling page can call the first wizard page directly. Using a launcher page, however, can simplify wizard initialization, particularly if initialization is complex.
- Users can navigate between pages by using Back and Forward buttons (or hyperlinks).
- Users can navigate between pages using the journal.
- Users can cancel the wizard from any wizard page by pressing a Cancel button.
- Users can accept the wizard on the last wizard page by pressing a Finish button.

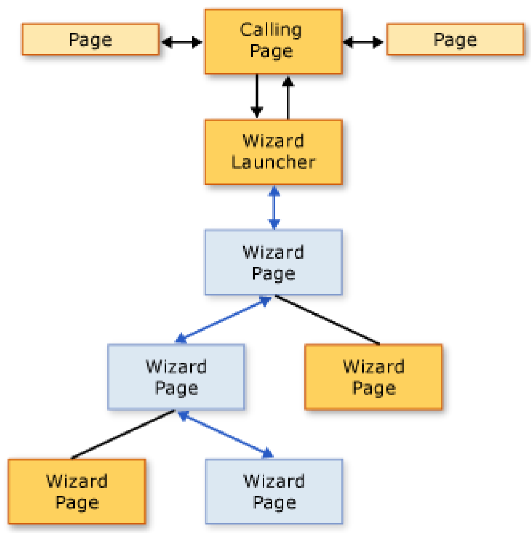
- If a wizard is canceled, the wizard returns an appropriate result, and does not return any data.
- If a user accepts a wizard, the wizard returns an appropriate result, and returns the data it collected.
- When the wizard is complete (accepted or canceled), the pages that the wizard comprises are removed from the journal. This keeps each instance of the wizard isolated, thereby avoiding potential data or state anomalies.

Dynamic Navigation over a Fixed Hierarchical Topology

In some applications, pages allow navigation to two or more other pages, as shown in the following figure.



This structure is known as a fixed hierarchical topology, and the sequence in which the hierarchy is traversed is often determined at run time by either the application or the user. At run time, each page in the hierarchy that allows navigation to two or more other pages gathers the data required to determine which page to navigate to. The following figure illustrates one of several possible navigation sequences based on the previous figure.



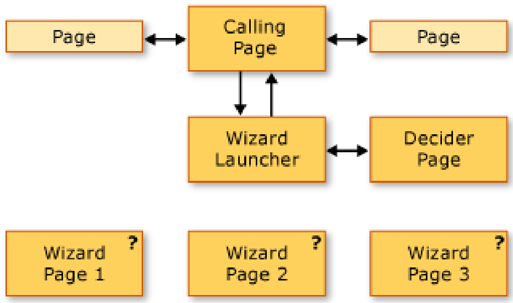
Even though the sequence in which pages in a fixed hierarchical structure are navigated is determined at run time, the user experience is the same as the user experience for a fixed linear topology:

- Navigating from the calling page to a launcher page that initializes the wizard and navigates to the first wizard page. A launcher page (a UI-less PageFunction<T>) is not required, since a calling page can call the first wizard page directly. Using a launcher page, however, can simplify wizard initialization, particularly if initialization is complex.
- Users can navigate between pages by using Back and Forward buttons (or hyperlinks).

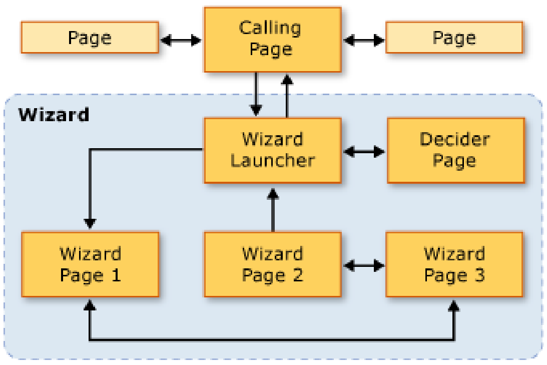
- Users can navigate between pages using the journal.
- Users can change the navigation sequence if they navigate back through the journal.
- Users can cancel the wizard from any wizard page by pressing a Cancel button.
- Users can accept the wizard on the last wizard page by pressing a Finish button.
- If a wizard is canceled, the wizard returns an appropriate result, and does not return any data.
- If a user accepts a wizard, the wizard returns an appropriate result, and returns the data it collected.
- When the wizard is complete (accepted or canceled), the pages that the wizard comprises are removed from the journal. This keeps each instance of the wizard isolated, thereby avoiding potential data or state anomalies.

Navigation over a Dynamically Generated Topology

In some applications, the sequence in which two or more pages are navigated can only be determined at run time, whether by the user, the application, or external data. The following figure illustrates a set of pages with an undetermined navigation sequence.



The next figure illustrates a navigation sequence that was chosen by the user at run time.



The navigation sequence is known as a dynamically generated topology. For the user, as with the other navigation topologies, the user experience is the same as it is for the previous topologies:

- Navigating from the calling page to a launcher page that initializes the wizard and navigates to the first wizard page. A launcher page (a UI-less PageFunction<T>) is not required, since a calling page can call the first wizard page directly. Using a launcher page, however, can simplify wizard initialization, particularly if initialization is complex.
- Users can navigate between pages by using Back and Forward buttons (or hyperlinks).
- Users can navigate between pages using the journal.
- Users can cancel the wizard from any wizard page by pressing a Cancel button.
- Users can accept the wizard on the last wizard page by pressing a Finish button.

- If a wizard is canceled, the wizard returns an appropriate result, and does not return any data.
- If a user accepts a wizard, the wizard returns an appropriate result, and returns the data it collected.
- When the wizard is complete (accepted or canceled), the pages that the wizard comprises are removed from the journal. This keeps each instance of the wizard isolated, thereby avoiding potential data or state anomalies.

See also

- Page
- PageFunction<T>
- NavigationService
- Structured Navigation Overview

Navigation How-to Topics

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

The following topics show how to use Windows Presentation Foundation (WPF) navigation.

In This Section

Call a Page Function
Get the Return Value of a Page Function
Navigate Forward or Back Through Navigation History
Return from a Page Function

Related Sections

Navigation Overview

Structured Navigation Overview

How to: Call a Page Function

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

This example shows how to call a page function from a Extensible Application Markup Language (XAML) page.

Example

You can navigate to a page function using a uniform resource identifier (URI), just as you can when you navigate to a page. This is shown in the following example.

```
// Navigate to a page function like a page
Uri pageFunctionUri = new Uri("GetStringPageFunction.xaml", UriKind.Relative);
this.NavigationService.Navigate(pageFunctionUri);
```

```
' Navigate to a page function like a page
Dim pageFunctionUri As New Uri("GetStringPageFunction.xaml", UriKind.Relative)
Me.NavigationService.Navigate(pageFunctionUri)
```

If you need to pass data to the page function, you can create an instance of it and pass the data by setting a property. Or, as the following example shows, you can pass the data using a non-default constructor.

```
<Page x:Class="UsingPageFunctionsSample.CallingPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CallingPage"
    >
    <Hyperlink Name="callPageFunctionHyperlink" Click="callPageFunctionHyperlink_Click">Call Page
Function</Hyperlink>
</Page>
```

```
void callPageFunctionHyperlink_Click(object sender, RoutedEventArgs e)
{
    // Call a page function
    GetStringPageFunction pageFunction = new GetStringPageFunction("initialValue");
    this.NavigationService.Navigate(pageFunction);
}
```

```
Private Sub callPageFunctionHyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Call a page function
    Dim pageFunction As New GetStringPageFunction("initialValue")
    Me.NavigationService.Navigate(pageFunction)
End Sub
```

See also

- PageFunction<T>

How to: Navigate to a Page

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

This example illustrates several ways in which a page can be navigated to from a NavigationWindow.

Example

It is possible for a NavigationWindow to navigate to a page using one of the following:

- The Source property.
- The Navigate method.

```
// Navigate to URI using the Source property
this.Source = new Uri("HomePage.xaml", UriKind.Relative);

// Navigate to URI using the Navigate method
this.Navigate(new Uri("HomePage.xaml", UriKind.Relative));

// Navigate to object using the Navigate method
this.Navigate(new HomePage());
```

```
' Navigate to URI using the Source property
Me.Source = New Uri("HomePage.xaml", UriKind.Relative)

' Navigate to URI using the Navigate method
Me.Navigate(New Uri("HomePage.xaml", UriKind.Relative))

' Navigate to object using the Navigate method
Me.Navigate(New HomePage())
```

NOTE

Uniform resource identifiers (URIs) can be either relative or absolute. For more information, see Pack URIs in WPF.

See also

- Frame
- Page
- NavigationService

How to: Refresh a Page

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

This example shows how to call the Refresh method to refresh the current content in a NavigationWindow.

Example

Refresh refreshes the current content in a NavigationWindow to be reloaded from its source.

```
void navigateRefreshButton_Click(object sender, RoutedEventArgs e)
{
    this.Refresh();
}
```

```
Private Sub navigateRefreshButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Me.Refresh()
End Sub
```

How to: Stop a Page from Loading

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

This example shows how to call the StopLoading method to stop navigation to content before it has finished being downloaded.

Example

StopLoading stops the download of the requested content, and causes the NavigationStopped event to be raised.

```
void navigateStopButton_Click(object sender, RoutedEventArgs e)
{
    this.StopLoading();
}
```

```
Private Sub navigateStopButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Me.StopLoading()
End Sub
```

How to: Navigate Back Through Navigation History

7/23/2018 • 2 minutes to read • Edit Online

This example illustrates how to navigate to entries in back navigation history.

Example

Code that is running from content that is hosted in a NavigationWindow, Frame using NavigationService, or Windows Internet Explorer can navigate back through navigation history, one entry at a time.

Navigating back one entry requires first checking that there are entries in back navigation history, by inspecting the **CanGoBack** property, before navigating back one entry, by calling the **GoBack** method. This is illustrated in the following example:

```
void navigateBackButton_Click(object sender, RoutedEventArgs e)
{
    // Navigate back one page from this page, if there is an entry
    // in back navigation history
    if (this.NavigationService.CanGoBack)
    {
        this.NavigationService.GoBack();
    }
    else
    {
        MessageBox.Show("No entries in back navigation history.");
    }
}
```

```
Private Sub navigateBackButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Navigate back one page from this page, if there is an entry
    ' in back navigation history
    If Me.NavigationService.CanGoBack Then
        Me.NavigationService.GoBack()
    Else
        MessageBox.Show("No entries in back navigation history.")
    End If
End Sub
```

CanGoBack and **GoBack** are implemented by NavigationWindow, Frame, and NavigationService.

NOTE

If you call **GoBack**, and there are no entries in back navigation history, an InvalidOperationException is raised.

How to: Return from a Page Function

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

This example shows how to return a result from a page function.

Example

To return from a page function, you need to call OnReturn and pass an instance of ReturnEventArgs<T>.

```
<PageFunction
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib"
    x:Class="UsingPageFunctionsSample.GetStringPageFunction"
    x:TypeArguments="sys:String"
    Title="GetStringPageFunction">
```

```
</PageFunction>
```

```
public partial class GetStringPageFunction : PageFunction<String>
{
    public GetStringPageFunction()
    {
        InitializeComponent();
    }

    public GetStringPageFunction(string initialValue) : this()
    {
        this.stringTextBox.Text = initialValue;
    }

    void okButton_Click(object sender, RoutedEventArgs e)
    {
        // Page function is accepted, so return a result
        OnReturn(new ReturnEventArgs<string>(this.stringTextBox.Text));
    }

    void cancelButton_Click(object sender, RoutedEventArgs e)
    {
        // Page function is cancelled, so don't return a result
        OnReturn(new ReturnEventArgs<string>(null));
    }
}
```

```
Partial Public Class GetStringPageFunction
    Inherits PageFunction(Of String)
    Public Sub New()
        InitializeComponent()
    End Sub

    Public Sub New(ByVal initialValue As String)
        Me.New()
        Me.stringTextBox.Text = initialValue
    End Sub

    Private Sub okButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
        ' Page function is accepted, so return a result
        OnReturn(New ReturnEventArgs(Of String)(Me.stringTextBox.Text))
    End Sub

    Private Sub cancelButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
        ' Page function is cancelled, so don't return a result
        OnReturn(New ReturnEventArgs(Of String)(Nothing))
    End Sub
End Class
```

See also

- PageFunction<T>

How to: Get the Return Value of a Page Function

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

This example shows how to get the result that is returned by a page function.

Example

To get the result that is returned from a page function, you need to handle Return of the page function you are calling.

```
<Page x:Class="UsingPageFunctionsSample.CallingPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CallingPage"
    >
    <Hyperlink Name="callPageFunctionHyperlink" Click="callPageFunctionHyperlink_Click">Call Page
Function</Hyperlink>
</Page>
```

```
void callPageFunctionAndReturnHyperlink_Click(object sender, RoutedEventArgs e)
{
    // Call a page function and hook up page function's return event to get result
    GetStringPageFunction pageFunction = new GetStringPageFunction();
    pageFunction.Return += new ReturnEventHandler<String>(GetStringPageFunction_Returned);
    this.NavigationService.Navigate(pageFunction);
}
void GetStringPageFunction_Returned(object sender, ReturnEventArgs<String> e)
{
    // Get the result, if a result was passed.
    if (e.Result != null)
    {
        Console.WriteLine(e.Result);
    }
}
```

```
Private Sub callPageFunctionAndReturnHyperlink_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Call a page function and hook up page function's return event to get result
    Dim pageFunction As New GetStringPageFunction()
    AddHandler pageFunction.Return, AddressOf GetStringPageFunction_Returned
    Me.NavigationService.Navigate(pageFunction)
End Sub
Private Sub GetStringPageFunction_Returned(ByVal sender As Object, ByVal e As ReturnEventArgs(Of String))
    ' Get the result, if a result was passed.
    If e.Result IsNot Nothing Then
        Console.WriteLine(e.Result)
    End If
End Sub
```

See also

- PageFunction<T>

How to: Navigate Forward or Back Through Navigation History

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

This example illustrates how to navigate forward or back to entries in navigation history.

Example

Code that runs from content in the following hosts can navigate forward or back through navigation history, one entry at a time.

- NavigationWindow using NavigationService
- Frame using NavigationService
- Windows Internet Explorer

Before you can navigate forward one entry, you must first check that there are entries in forward navigation history by inspecting the **CanGoForward** property. To navigate forward one entry, you call the **GoForward** method. This is illustrated in the following example:

```
void navigateForwardButton_Click(object sender, RoutedEventArgs e)
{
    // Navigate forward one page from this page, if there is an entry
    // in forward navigation history
    if (this.NavigationService.CanGoForward)
    {
        this.NavigationService.GoForward();
    }
    else
    {
        MessageBox.Show("No entries in forward navigation history.");
    }
}
```

```
Private Sub navigateForwardButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Navigate forward one page from this page, if there is an entry
    ' in forward navigation history
    If Me.NavigationService.CanGoForward Then
        Me.NavigationService.GoForward()
    Else
        MessageBox.Show("No entries in forward navigation history.")
    End If
End Sub
```

Before you can navigate back one entry, you must first check that there are entries in back navigation history by inspecting the **CanGoBack** property. To navigate back one entry, you call the **GoBack** method. This is illustrated in the following example:

```csharp
void navigateBackButton_Click(object sender, RoutedEventArgs e)
{
    // Navigate back one page from this page, if there is an entry
    // in back navigation history
    if (this.NavigationService.CanGoBack)
    {
        this.NavigationService.GoBack();
    }
    else
    {
        MessageBox.Show("No entries in back navigation history.");
    }
}
```

```vb
Private Sub navigateBackButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Navigate back one page from this page, if there is an entry
    ' in back navigation history
    If Me.NavigationService.CanGoBack Then
        Me.NavigationService.GoBack()
    Else
        MessageBox.Show("No entries in back navigation history.")
    End If
End Sub
```

CanGoForward, **GoForward**, **CanGoBack**, and **GoBack** are implemented by NavigationWindow, Frame, and NavigationService.

NOTE

If you call **GoForward**, and there are no entries in forward navigation history, or if you call **GoBack**, and there are no entries in back navigation history, an InvalidOperationException is thrown.

How to: Determine If a Page is Browser Hosted

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

This example demonstrates how to determine if a Page is hosted in a browser.

Example

A Page can be host agnostic and, consequently, can be loaded into several different types of hosts, including a Frame, a NavigationWindow, or a browser. This can happen when you have a library assembly that contains one or more pages, and which is referenced by multiple standalone and browsable (XAML browser application (XBAP)) host applications.

The following example demonstrates how to use BrowserInteropHelper.IsBrowserHosted to determine if a Page is hosted in a browser.

```
// Detect if browser hosted
if (BrowserInteropHelper.IsBrowserHosted)
{
    // Note: can only inspect BrowserInteropHelper.Source property if page is browser-hosted.
    this.dataTextBlock.Text = "Is Browser Hosted: " + BrowserInteropHelper.Source.ToString();
}
else
{
    this.dataTextBlock.Text = "Is not browser hosted";
}
```

```
' Detect if browser hosted
If BrowserInteropHelper.IsBrowserHosted Then
    ' Note: can only inspect BrowserInteropHelper.Source property if page is browser-hosted.
    Me.dataTextBlock.Text = "Is Browser Hosted: " & BrowserInteropHelper.Source.ToString()
Else
    Me.dataTextBlock.Text = "Is not browser hosted"
End If
```

See also

- Frame
- Page

How to: Use mailto: to Send Mail From a Page

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

This example shows how to use Hyperlink in conjunction with a **mailto:**uniform resource identifier (URI).

Example

The following code shows how to use a **mailto:**uniform resource identifier (URI) to open a new mail window that contains an email address, and email address and a subject, and an email address, subject, and body.

```
<Hyperlink NavigateUri="mailto:username@domainname">mailto: With Email Address</Hyperlink>
<Hyperlink NavigateUri="mailto:username@domainname?subject=SubjectText">mailto: With Email Address and
Subject</Hyperlink>
<Hyperlink NavigateUri="mailto:username@domainname?subject=SubjectText&amp;body=BodyText">mailto: With Email
Address, Subject, and Body</Hyperlink>
```

See also

- Pack URIs in WPF

How to: Set the Title of a Window from a Page

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

This example shows how to set the title of the window in which a Page is hosted.

Example

A page can change the title of the window that is hosting it by setting the WindowTitle property, like so:

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Page Title"
    WindowTitle="Window Title"
    >
</Page>
```

NOTE

Setting the Title property of a page does not change the value of the window title. Instead, Title specifies the name of a page entry in navigation history.

How to: Set the Width of a Window from a Page

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

This example illustrates how to set the width of the window from a Page.

Example

A Page can set the width of its host window by setting WindowWidth. This property allows the Page to not have explicit knowledge of the type of window that hosts it.

NOTE

To set the width of a window using WindowWidth, a Page must be the child of a window.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="SetWindowWidthPage"
    WindowWidth="500"
    >
</Page>
```

How to: Set the Height of a Window from a Page

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

This example illustrates how to set the height of the window from a Page.

Example

A Page can set the height of its host window by setting WindowHeight. This property allows the Page to not have explicit knowledge of the type of window that hosts it.

NOTE

To set the height of a window using WindowHeight, a Page must be the child of a window.

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="SetWindowHeightPage"
    WindowHeight="500"
    >
</Page>
```

WPF Add-Ins Overview

1/23/2019 • 16 minutes to read • Edit Online

The .NET Framework includes an add-in model that developers can use to create applications that support add-in extensibility. This add-in model allows the creation of add-ins that integrate with and extend application functionality. In some scenarios, applications also need to display user interfaces that are provided by add-ins. This topic shows how WPF augments the .NET Framework add-in model to enable these scenarios, the architecture behind it, its benefits, and its limitations.

Prerequisites

Familiarity with the .NET Framework add-in model is required. For more information, see Add-ins and Extensibility.

Add-Ins Overview

In order to avoid the complexities of application recompilation and redeployment to incorporate new functionality, applications implement extensibility mechanisms that allow developers (both first-party and third-party) to create other applications that integrate with them. The most common way to support this type of extensibility is through the use of add-ins (also known as "add-ons" and "plug-ins"). Examples of real-world applications that expose extensibility with add-ins include:

- Internet Explorer add-ons.
- Windows Media Player plug-ins.
- Visual Studio add-ins.

For example, the Windows Media Player add-in model allows third-party developers to implement "plug-ins" that extend Windows Media Player in a variety of ways, including creating decoders and encoders for media formats that are not supported natively by Windows Media Player (for example, DVD, MP3), audio effects, and skins. Each add-in model is built to expose the functionality that is unique to an application, although there are several entities and behaviors that are common to all add-in models.

The three main entities of typical add-in extensibility solutions are *contracts*, *add-ins*, and *host applications*. Contracts define how add-ins integrate with host applications in two ways:

- Add-ins integrate with functionality that is implemented by host applications.
- Host applications expose functionality for add-ins to integrate with.

In order for add-ins to be used, host applications need to find them and load them at run time. Consequently, applications that support add-ins have the following additional responsibilities:

- **Discovery**: Finding add-ins that adhere to contracts supported by host applications.
- **Activation**: Loading, running, and establishing communication with add-ins.
- **Isolation**: Using either application domains or processes to establish isolation boundaries that protect applications from potential security and execution problems with add-ins.
- **Communication**: Allowing add-ins and host applications to communicate with each other across isolation boundaries by calling methods and passing data.

- **Lifetime Management**: Loading and unloading application domains and processes in a clean, predictable manner (see Application Domains).
- **Versioning**: Ensuring that host applications and add-ins can still communicate when new versions of either are created.

Ultimately, developing a robust add-in model is a non-trivial undertaking. For this reason, the .NET Framework provides an infrastructure for building add-in models.

> **NOTE**
>
> For more detailed information on add-ins, see Add-ins and Extensibility.

.NET Framework Add-In Model Overview

The .NET Framework add-in model, found in the System.AddIn namespace, contains a set of types that are designed to simplify the development of add-in extensibility. The fundamental unit of the .NET Framework add-in model is the *contract*, which defines how a host application and an add-in communicate with each other. A contract is exposed to a host application using a host-application-specific *view* of the contract. Likewise, an add-in-specific *view* of the contract is exposed to the add-in. An *adapter* is used to allow a host application and an add-in to communicate between their respective views of the contract. Contracts, views, and adapters are referred to as segments, and a set of related segments constitutes a *pipeline*. Pipelines are the foundation upon which the .NET Framework add-in model supports discovery, activation, security isolation, execution isolation (using both application domains and processes), communication, lifetime management, and versioning.

The sum of this support allows developers to build add-ins that integrate with the functionality of a host application. However, some scenarios require host applications to display user interfaces provided by add-ins. Because each presentation technology in the .NET Framework has its own model for implementing user interfaces, the .NET Framework add-in model does not support any particular presentation technology. Instead, WPF extends the .NET Framework add-in model with UI support for add-ins.

WPF Add-Ins

WPF, in conjunction with the .NET Framework add-in model, allows you to address a wide variety of scenarios that require host applications to display user interfaces from add-ins. In particular, these scenarios are addressed by WPF with the following two programming models:

1. **The add-in returns a UI**. An add-in returns a UI to the host application via a method call, as defined by the contract. This scenario is used in the following cases:
    - The appearance of a UI that is returned by an add-in is dependent on either data or conditions that exist only at run time, such as dynamically generated reports.
    - The UI for services provided by an add-in differs from the UI of the host applications that can use the add-in.
    - The add-in primarily performs a service for the host application, and reports status to the host application with a UI.
2. **The add-in is a UI**. An add-in is a UI, as defined by the contract. This scenario is used in the following cases:
    - An add-in doesn't provide services other than being displayed, such as an advertisement.
    - The UI for services provided by an add-in is common to all host applications that can use that add-in, such as a calculator or color picker.

These scenarios require that UI objects can be passed between host application and add-in application domains. Since the .NET Framework add-in model relies on remoting to communicate between application domains, the objects that are passed between them must be remotable.

A remotable object is an instance of a class that does one or more of the following:

- Derives from the MarshalByRefObject class.
- Implements the ISerializable interface.
- Has the SerializableAttribute attribute applied.

NOTE

For more information regarding the creation of remotable .NET Framework objects, see Making Objects Remotable.

The WPF UI types are not remotable. To solve the problem, WPF extends the .NET Framework add-in model to enable WPF UI created by add-ins to be displayed from host applications. This support is provided by WPF by two types: the INativeHandleContract interface and two static methods implemented by the FrameworkElementAdapters class: ContractToViewAdapter and ViewToContractAdapter. At a high level, these types and methods are used in the following manner:

1. WPF requires that user interfaces provided by add-ins are classes that derive directly or indirectly from FrameworkElement, such as shapes, controls, user controls, layout panels, and pages.
2. Wherever the contract declares that a UI will be passed between the add-in and the host application, it must be declared as an INativeHandleContract (not a FrameworkElement); INativeHandleContract is a remotable representation of the add-in UI that can be passed across isolation boundaries.
3. Before being passed from the add-in's application domain, a FrameworkElement is packaged as an INativeHandleContract by calling ViewToContractAdapter.
4. After being passed to the host application's application domain, the INativeHandleContract must be repackaged as a FrameworkElement by calling ContractToViewAdapter.

How INativeHandleContract, ContractToViewAdapter, and ViewToContractAdapter are used depends on the specific scenario. The following sections provide details for each programming model.

Add-In Returns a User Interface

For an add-in to return a UI to a host application, the following are required:

1. The host application, add-in, and pipeline must be created, as described by the .NET Framework Add-ins and Extensibility documentation.
2. The contract must implement IContract and, to return a UI, the contract must declare a method with a return value of type INativeHandleContract.
3. The UI that is passed between the add-in and the host application must directly or indirectly derive from FrameworkElement.
4. The UI that is returned by the add-in must be converted from a FrameworkElement to an INativeHandleContract before crossing the isolation boundary.
5. The UI that is returned must be converted from an INativeHandleContract to a FrameworkElement after crossing the isolation boundary.
6. The host application displays the returned FrameworkElement.

For an example that demonstrates how to implement an add-in that returns a UI, see Create an Add-In That Returns a UI.

Add-In Is a User Interface

When an add-in is a UI, the following are required:

1. The host application, add-in, and pipeline must be created, as described by the .NET Framework Add-ins and Extensibility documentation.

2. The contract interface for the add-in must implement INativeHandleContract.

3. The add-in that is passed to the host application must directly or indirectly derive from FrameworkElement.

4. The add-in must be converted from a FrameworkElement to an INativeHandleContract before crossing the isolation boundary.

5. The add-in must be converted from an INativeHandleContract to a FrameworkElement after crossing the isolation boundary.

6. The host application displays the returned FrameworkElement.

For an example that demonstrates how to implement an add-in that is a UI, see Create an Add-In That Is a UI.

Returning Multiple UIs from an Add-In

Add-ins often provide multiple user interfaces for host applications to display. For example, consider an add-in that is a UI that also provides status information to the host application, also as a UI. An add-in like this can be implemented by using a combination of techniques from both the Add-In Returns a User Interface and Add-In Is a User Interface models.

Add-Ins and XAML Browser Applications

In the examples so far, the host application has been an installed standalone application. But XAML browser applications (XBAPs) can also host add-ins, albeit with the following additional build and implementation requirements:

- The XBAP application manifest must be configured specially to download the pipeline (folders and assemblies) and add-in assembly to the ClickOnce application cache on the client machine, in the same folder as the XBAP.

- The XBAP code to discover and load add-ins must use the ClickOnce application cache for the XBAP as the pipeline and add-in location.

- The XBAP must load the add-in into a special security context if the add-in references loose files that are located at the site of origin; when hosted by XBAPs, add-ins can only reference loose files that are located at the host application's site of origin.

These tasks are described in detail in the following subsections.

Configuring the Pipeline and Add-In for ClickOnce Deployment

XBAPs are downloaded to and run from a safe folder in the ClickOnce deployment cache. In order for an XBAP to host an add-in, the pipeline and add-in assembly must also be downloaded to the safe folder. To achieve this, you need to configure the application manifest to include both the pipeline and add-in assembly for download. This is most easily done in Visual Studio, although the pipeline and add-in assembly needs to be in the host XBAP project's root folder in order for Visual Studio to detect the pipeline assemblies.

Consequently, the first step is to build the pipeline and add-in assembly to the XBAP project's root by setting the build output of each pipeline assembly and add-in assembly projects. The following table shows the build output paths for pipeline assembly projects and add-in assembly project that are in the same solution and root folder as the host XBAP project.

Table 1: Build Output Paths for the Pipeline Assemblies That Are Hosted by an XBAP

PIPELINE ASSEMBLY PROJECT	BUILD OUTPUT PATH
Contract	`..\HostXBAP\Contracts\`
Add-In View	`..\HostXBAP\AddInViews\`
Add-In-Side Adapter	`..\HostXBAP\AddInSideAdapters\`
Host-Side Adapter	`..\HostXBAP\HostSideAdapters\`
Add-In	`..\HostXBAP\AddIns\WPFAddIn1`

The next step is to specify the pipeline assemblies and add-in assembly as the XBAPs content files in Visual Studio by doing the following:

1. Including the pipeline and add-in assembly in the project by right-clicking each pipeline folder in Solution Explorer and choosing **Include In Project**.

2. Setting the **Build Action** of each pipeline assembly and add-in assembly to **Content** from the **Properties** window.

The final step is to configure the application manifest to include the pipeline assembly files and add-in assembly file for download. The files should be located in folders at the root of the folder in the ClickOnce cache that the XBAP application occupies. The configuration can be achieved in Visual Studio by doing the following:

1. Right-click the XBAP project, click **Properties**, click **Publish**, and then click the **Application Files** button.

2. In the **Application Files** dialog, set the **Publish Status** of each pipeline and add-in DLL to **Include (Auto)**, and set the **Download Group** for each pipeline and add-in DLL to **(Required)**.

Using the Pipeline and Add-In from the Application Base

When the pipeline and add-in are configured for ClickOnce deployment, they are downloaded to the same ClickOnce cache folder as the XBAP. To use the pipeline and add-in from the XBAP, the XBAP code must get them from the application base. The various types and members of the .NET Framework add-in model for using pipelines and add-ins provide special support for this scenario. Firstly, the path is identified by the ApplicationBase enumeration value. You use this value with overloads of the pertinent add-in members for using pipelines that include the following:

- AddInStore.FindAddIns(Type, PipelineStoreLocation)

- AddInStore.FindAddIns(Type, PipelineStoreLocation, String[])

- AddInStore.Rebuild(PipelineStoreLocation)

- AddInStore.Update(PipelineStoreLocation)

Accessing the Host's Site of Origin

To ensure that an add-in can reference files from the site of origin, the add-in must be loaded with security isolation that is equivalent to the host application. This security level is identified by the AddInSecurityLevel.Host enumeration value, and passed to the Activate method when an add-in is activated.

WPF Add-In Architecture

At the highest level, as we've seen, WPF enables .NET Framework add-ins to implement user interfaces (that derive directly or indirectly from FrameworkElement) using INativeHandleContract, ViewToContractAdapter and ContractToViewAdapter. The result is that the host application is returned a FrameworkElement that is displayed from UI in the host application.

For simple UI add-in scenarios, this is as much detail as a developer needs. For more complex scenarios, particularly those that try to utilize additional WPF services such as layout, resources, and data binding, more detailed knowledge of how WPF extends the .NET Framework add-in model with UI support is required to understand its benefits and limitations.

Fundamentally, WPF doesn't pass a UI from an add-in to a host application; instead, WPF passes the Win32 window handle for the UI by using WPF interoperability. As such, when a UI from an add-in is passed to a host application, the following occurs:

- On the add-in side, WPF acquires a window handle for the UI that will be displayed by the host application. The window handle is encapsulated by an internal WPF class that derives from HwndSource and implements INativeHandleContract. An instance of this class is returned by ViewToContractAdapter and is marshaled from the add-in's application domain to the host application's application domain.

- On the host application side, WPF repackages the HwndSource as an internal WPF class that derives from HwndHost and consumes INativeHandleContract. An instance of this class is returned by ContractToViewAdapter to the host application.

HwndHost exists to display user interfaces, identified by window handles, from WPF user interfaces. For more information, see WPF and Win32 Interoperation.

In summary, INativeHandleContract, ViewToContractAdapter, and ContractToViewAdapter exist to allow the window handle for a WPF UI to be passed from an add-in to a host application, where it is encapsulated by a HwndHost and displayed the host application's UI.

NOTE

Because the host application gets an HwndHost, the host application cannot convert the object that is returned by ContractToViewAdapter to the type it is implemented as by the add-in (for example, a UserControl).

By its nature, HwndHost has certain limitations that affect how host applications can use them. However, WPF extends HwndHost with several capabilities for add-in scenarios. These benefits and limitations are described below.

WPF Add-In Benefits

Because WPF add-in user interfaces are displayed from host applications using an internal class that derives from HwndHost, those user interfaces are constrained by the capabilities of HwndHost with respect to WPF UI services such as layout, rendering, data binding, styles, templates, and resources. However, WPF augments its internal HwndHost subclass with additional capabilities that include the following:

- Tabbing between a host application's UI and an add-in's UI. Note that the "add-in is a UI" programming model requires the add-in-side adapter to override QueryContract to enable tabbing, whether the add-in is fully trusted or partially trusted.

- Honoring accessibility requirements for add-in user interfaces that are displayed from host application user interfaces.

- Enabling WPF applications to run safely in multiple application domain scenarios.

- Preventing illegal access to add-in UI window handles when add-ins run with security isolation (that is, a partial-trust security sandbox). Calling ViewToContractAdapter ensures this security:
  - For the "add-in returns a UI" programming model, the only way to pass the window handle for an add-in UI across the isolation boundary is to call ViewToContractAdapter.
  - For the "add-in is a UI" programming model, overriding QueryContract on the add-in-side adapter and calling ViewToContractAdapter (as shown in the preceding examples) is required, as is calling the add-in-side adapter's `QueryContract` implementation from the host-side adapter.
- Providing multiple application domain execution protection. Due to limitations with application domains, unhandled exceptions that are thrown in add-in application domains cause the entire application to crash, even though the isolation boundary exists. However, WPF and the .NET Framework add-in model provide a simple way to work around this problem and improve application stability. A WPF add-in that displays a UI creates a Dispatcher for the thread that the application domain runs on, if the host application is a WPF application. You can detect all unhandled exceptions that occur in the application domain by handling the UnhandledException event of the WPF add-in's Dispatcher. You can get the Dispatcher from the CurrentDispatcher property.

WPF Add-In Limitations

Beyond the benefits that WPF adds to the default behaviors supplied by HwndSource, HwndHost, and window handles, there are also limitations for add-in user interfaces that are displayed from host applications:

- Add-in user interfaces displayed from a host application do not respect the host application's clipping behavior.
- The concept of *airspace* in interoperability scenarios also applies to add-ins (see Technology Regions Overview).
- A host application's UI services, such as resource inheritance, data binding, and commanding, are not automatically available to add-in user interfaces. To provide these services to the add-in, you need to update the pipeline.
- An add-in UI cannot be rotated, scaled, skewed, or otherwise affected by a transformation (see Transforms Overview).
- Content inside add-in user interfaces that is rendered by drawing operations from the System.Drawing namespace can include alpha blending. However, both an add-in UI and the host application UI that contains it must be 100% opaque; in other words, the `Opacity` property on both must be set to 1.
- If the AllowsTransparency property of a window in the host application that contains an add-in UI is set to `true`, the add-in is invisible. This is true even if the add-in UI is 100% opaque (that is, the `Opacity` property has a value of 1).
- An add-in UI must appear on top of other WPF elements in the same top-level window.
- No portion of an add-in's UI can be rendered using a VisualBrush. Instead, the add-in may take a snapshot of the generated UI to create a bitmap that can be passed to the host application using methods defined by the contract.
- Media files cannot be played from a MediaElement in an add-in UI.
- Mouse events generated for the add-in UI are neither received nor raised by the host application, and the `IsMouseOver` property for host application UI has a value of `false`.
- When focus shifts between controls in an add-in UI, the `GotFocus` and `LostFocus` events are neither received nor raised by the host application.

- The portion of a host application that contains an add-in UI appears white when printed.
- All dispatchers (see Dispatcher) created by the add-in UI must be shut down manually before the owner add-in is unloaded if the host application continues execution. The contract can implement methods that allow the host application to signal the add-in before the add-in is unloaded, thereby allowing the add-in UI to shut down its dispatchers.
- If an add-in UI is an InkCanvas or contains an InkCanvas, you cannot unload the add-in.

Performance Optimization

By default, when multiple application domains are used, the various .NET Framework assemblies required by each application are all loaded into that application's domain. As a result, the time required for creating new application domains and starting applications in them might affect performance. However, the .NET Framework provides a way for you to reduce start times by instructing applications to share assemblies across application domains if they are already loaded. You do this by using the LoaderOptimizationAttribute attribute, which must be applied to the entry point method (`Main`). In this case, you must use only code to implement your application definition (see Application Management Overview).

See also

- LoaderOptimizationAttribute
- Add-ins and Extensibility
- Application Domains
- .NET Framework Remoting Overview
- Making Objects Remotable
- How-to Topics

How-to Topics

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

The following topics show how to create Windows Presentation Foundation (WPF) add-ins.

In This Section

Create an Add-In That Returns a UI
Create an Add-In That Is a UI

Related Sections

WPF Add-Ins Overview

How to: Create an Add-In That Returns a UI

1/23/2019 • 8 minutes to read • Edit Online

This example shows how to create an add-in that returns a Windows Presentation Foundation (WPF) to a host WPF standalone application.

The add-in returns a UI that is a WPF user control. The content of the user control is a single button that, when clicked, displays a message box. The WPF standalone application hosts the add-in and displays the user control (returned by the add-in) as the content of the main application window.

Prerequisites

This example highlights the WPF extensions to the .NET Framework add-in model that enable this scenario, and assumes the following:

- Knowledge of the .NET Framework add-in model, including pipeline, add-in, and host development. If you are unfamiliar with these concepts, see Add-ins and Extensibility. For a tutorial that demonstrates the implementation of a pipeline, an add-in, and a host application, see Walkthrough: Creating an Extensible Application.
- Knowledge of the WPF extensions to the .NET Framework add-in model, which can be found here: WPF Add-Ins Overview.

Example

To create an add-in that returns a WPF UI requires specific code for each pipeline segment, the add-in, and the host application.

Implementing the Contract Pipeline Segment

A method must be defined by the contract for returning a UI, and its return value must be of type INativeHandleContract. This is demonstrated by the `GetAddInUI` method of the `IWPFAddInContract` contract in the following code.

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;

namespace Contracts
{
    /// <summary>
    /// Defines the services that an add-in will provide to a host application
    /// </summary>
    [AddInContract]
    public interface IWPFAddInContract : IContract
    {
        // Return a UI to the host application
        INativeHandleContract GetAddInUI();
    }
}
```

```
Imports System.AddIn.Contract
Imports System.AddIn.Pipeline

Namespace Contracts
    ''' <summary>
    ''' Defines the services that an add-in will provide to a host application
    ''' </summary>
    <AddInContract>
    Public Interface IWPFAddInContract
        Inherits IContract
        ' Return a UI to the host application
        Function GetAddInUI() As INativeHandleContract
    End Interface
End Namespace
```

Implementing the Add-In View Pipeline Segment

Because the add-in implements the UIs it provides as subclasses of FrameworkElement, the method on the add-in view that correlates to `IWPFAddInView.GetAddInUI` must return a value of type FrameworkElement. The following code shows the add-in view of the contract, implemented as an interface.

```
using System.AddIn.Pipeline;
using System.Windows;

namespace AddInViews
{
    /// <summary>
    /// Defines the add-in's view of the contract
    /// </summary>
    [AddInBase]
    public interface IWPFAddInView
    {
        // The add-in's implementation of this method will return
        // a UI type that directly or indirectly derives from
        // FrameworkElement.
        FrameworkElement GetAddInUI();
    }
}
```

```
Imports System.AddIn.Pipeline
Imports System.Windows

Namespace AddInViews
    ''' <summary>
    ''' Defines the add-in's view of the contract
    ''' </summary>
    <AddInBase>
    Public Interface IWPFAddInView
        ' The add-in's implementation of this method will return
        ' a UI type that directly or indirectly derives from
        ' FrameworkElement.
        Function GetAddInUI() As FrameworkElement
    End Interface
End Namespace
```

Implementing the Add-In-Side Adapter Pipeline Segment

The contract method returns an INativeHandleContract, but the add-in returns a FrameworkElement (as specified by the add-in view). Consequently, the FrameworkElement must be converted to an INativeHandleContract before crossing the isolation boundary. This work is performed by the add-in-side adapter by calling ViewToContractAdapter, as shown in the following code.

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;
using System.Windows;

using AddInViews;
using Contracts;

namespace AddInSideAdapters
{
    /// <summary>
    /// Adapts the add-in's view of the contract to the add-in contract
    /// </summary>
    [AddInAdapter]
    public class WPFAddIn_ViewToContractAddInSideAdapter : ContractBase, IWPFAddInContract
    {
        IWPFAddInView wpfAddInView;

        public WPFAddIn_ViewToContractAddInSideAdapter(IWPFAddInView wpfAddInView)
        {
            // Adapt the add-in view of the contract (IWPFAddInView)
            // to the contract (IWPFAddInContract)
            this.wpfAddInView = wpfAddInView;
        }

        public INativeHandleContract GetAddInUI()
        {
            // Convert the FrameworkElement from the add-in to an INativeHandleContract
            // that will be passed across the isolation boundary to the host application.
            FrameworkElement fe = this.wpfAddInView.GetAddInUI();
            INativeHandleContract inhc = FrameworkElementAdapters.ViewToContractAdapter(fe);
            return inhc;
        }
    }
}
```

```vb
Imports System.AddIn.Contract
Imports System.AddIn.Pipeline
Imports System.Windows

Imports AddInViews
Imports Contracts

Namespace AddInSideAdapters
    ''' <summary>
    ''' Adapts the add-in's view of the contract to the add-in contract
    ''' </summary>
    <AddInAdapter>
    Public Class WPFAddIn_ViewToContractAddInSideAdapter
        Inherits ContractBase
        Implements IWPFAddInContract
        Private wpfAddInView As IWPFAddInView

        Public Sub New(ByVal wpfAddInView As IWPFAddInView)
            ' Adapt the add-in view of the contract (IWPFAddInView)
            ' to the contract (IWPFAddInContract)
            Me.wpfAddInView = wpfAddInView
        End Sub

        Public Function GetAddInUI() As INativeHandleContract Implements IWPFAddInContract.GetAddInUI
            ' Convert the FrameworkElement from the add-in to an INativeHandleContract
            ' that will be passed across the isolation boundary to the host application.
            Dim fe As FrameworkElement = Me.wpfAddInView.GetAddInUI()
            Dim inhc As INativeHandleContract = FrameworkElementAdapters.ViewToContractAdapter(fe)
            Return inhc
        End Function
    End Class
End Namespace
```

Implementing the Host View Pipeline Segment

Because the host application will display a FrameworkElement, the method on the host view that correlates to `IWPFAddInHostView.GetAddInUI` must return a value of type FrameworkElement. The following code shows the host view of the contract, implemented as an interface.

```csharp
using System.Windows;

namespace HostViews
{
    /// <summary>
    /// Defines the host's view of the add-in
    /// </summary>
    public interface IWPFAddInHostView
    {
        // The view returns as a class that directly or indirectly derives from
        // FrameworkElement and can subsequently be displayed by the host
        // application by embedding it as content or sub-content of a UI that is
        // implemented by the host application.
        FrameworkElement GetAddInUI();
    }
}
```

```vb
Imports System.Windows

Namespace HostViews
    ''' <summary>
    ''' Defines the host's view of the add-in
    ''' </summary>
    Public Interface IWPFAddInHostView
        ' The view returns as a class that directly or indirectly derives from
        ' FrameworkElement and can subsequently be displayed by the host
        ' application by embedding it as content or sub-content of a UI that is
        ' implemented by the host application.
        Function GetAddInUI() As FrameworkElement
    End Interface
End Namespace
```

Implementing the Host-Side Adapter Pipeline Segment

The contract method returns an INativeHandleContract, but the host application expects a FrameworkElement (as specified by the host view). Consequently, the INativeHandleContract must be converted to a FrameworkElement after crossing the isolation boundary. This work is performed by the host-side adapter by calling ContractToViewAdapter, as shown in the following code.

```csharp
using System.AddIn.Contract;
using System.AddIn.Pipeline;
using System.Windows;

using Contracts;
using HostViews;

namespace HostSideAdapters
{
    /// <summary>
    /// Adapts the add-in contract to the host's view of the add-in
    /// </summary>
    [HostAdapter]
    public class WPFAddIn_ContractToViewHostSideAdapter : IWPFAddInHostView
    {
        IWPFAddInContract wpfAddInContract;
        ContractHandle wpfAddInContractHandle;

        public WPFAddIn_ContractToViewHostSideAdapter(IWPFAddInContract wpfAddInContract)
        {
            // Adapt the contract (IWPFAddInContract) to the host application's
            // view of the contract (IWPFAddInHostView)
            this.wpfAddInContract = wpfAddInContract;

            // Prevent the reference to the contract from being released while the
            // host application uses the add-in
            this.wpfAddInContractHandle = new ContractHandle(wpfAddInContract);
        }

        public FrameworkElement GetAddInUI()
        {
            // Convert the INativeHandleContract that was passed from the add-in side
            // of the isolation boundary to a FrameworkElement
            INativeHandleContract inhc = this.wpfAddInContract.GetAddInUI();
            FrameworkElement fe = FrameworkElementAdapters.ContractToViewAdapter(inhc);
            return fe;
        }
    }
}
```

```
Imports System.AddIn.Contract
Imports System.AddIn.Pipeline
Imports System.Windows

Imports Contracts
Imports HostViews

Namespace HostSideAdapters
    ''' <summary>
    ''' Adapts the add-in contract to the host's view of the add-in
    ''' </summary>
    <HostAdapter>
    Public Class WPFAddIn_ContractToViewHostSideAdapter
        Implements IWPFAddInHostView
        Private wpfAddInContract As IWPFAddInContract
        Private wpfAddInContractHandle As ContractHandle

        Public Sub New(ByVal wpfAddInContract As IWPFAddInContract)
            ' Adapt the contract (IWPFAddInContract) to the host application's
            ' view of the contract (IWPFAddInHostView)
            Me.wpfAddInContract = wpfAddInContract

            ' Prevent the reference to the contract from being released while the
            ' host application uses the add-in
            Me.wpfAddInContractHandle = New ContractHandle(wpfAddInContract)
        End Sub

        Public Function GetAddInUI() As FrameworkElement Implements IWPFAddInHostView.GetAddInUI
            ' Convert the INativeHandleContract that was passed from the add-in side
            ' of the isolation boundary to a FrameworkElement
            Dim inhc As INativeHandleContract = Me.wpfAddInContract.GetAddInUI()
            Dim fe As FrameworkElement = FrameworkElementAdapters.ContractToViewAdapter(inhc)
            Return fe
        End Function
    End Class
End Namespace
```

Implementing the Add-In

With the add-in-side adapter and add-in view created, the add-in (`WPFAddIn1.AddIn`) must implement the `IWPFAddInView.GetAddInUI` method to return a FrameworkElement object (a UserControl in this example). The implementation of the UserControl, `AddInUI`, is shown by the following code.

```
    <UserControl
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WPFAddIn1.AddInUI">

    <StackPanel>
        <Button Click="clickMeButton_Click" Content="Click Me!" />
    </StackPanel>

</UserControl>
```

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

namespace WPFAddIn1
{
    public partial class AddInUI : UserControl
    {
        public AddInUI()
        {
            InitializeComponent();
        }

        void clickMeButton_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Hello from WPFAddIn1");
        }
    }
}
```

```vb
Imports System.Windows
Imports System.Windows.Controls

Namespace WPFAddIn1
    Partial Public Class AddInUI
        Inherits UserControl
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub clickMeButton_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
            MessageBox.Show("Hello from WPFAddIn1")
        End Sub
    End Class
End Namespace
```

The implementation of the `IWPFAddInView.GetAddInUI` by the add-in simply needs to return a new instance of `AddInUI`, as shown by the following code.

```csharp
using System.AddIn;
using System.Windows;

using AddInViews;

namespace WPFAddIn1
{
    /// <summary>
    /// Add-In implementation
    /// </summary>
    [AddIn("WPF Add-In 1")]
    public class WPFAddIn : IWPFAddInView
    {
        public FrameworkElement GetAddInUI()
        {
            // Return add-in UI
            return new AddInUI();
        }
    }
}
```

```
Imports System.AddIn
Imports System.Windows

Imports AddInViews

Namespace WPFAddIn1
    ''' <summary>
    ''' Add-In implementation
    ''' </summary>
    <AddIn("WPF Add-In 1")>
    Public Class WPFAddIn
        Implements IWPFAddInView
        Public Function GetAddInUI() As FrameworkElement Implements IWPFAddInView.GetAddInUI
            ' Return add-in UI
            Return New AddInUI()
        End Function
    End Class
End Namespace
```

Implementing the Host Application

With the host-side adapter and host view created, the host application can use the .NET Framework add-in model to open the pipeline, acquire a host view of the add-in, and call the `IWPFAddInHostView.GetAddInUI` method. These steps are shown in the following code.

```
// Get add-in pipeline folder (the folder in which this application was launched from)
string appPath = Environment.CurrentDirectory;

// Rebuild visual add-in pipeline
string[] warnings = AddInStore.Rebuild(appPath);
if (warnings.Length > 0)
{
    string msg = "Could not rebuild pipeline:";
    foreach (string warning in warnings) msg += "\n" + warning;
    MessageBox.Show(msg);
    return;
}

// Activate add-in with Internet zone security isolation
Collection<AddInToken> addInTokens = AddInStore.FindAddIns(typeof(IWPFAddInHostView), appPath);
AddInToken wpfAddInToken = addInTokens[0];
this.wpfAddInHostView = wpfAddInToken.Activate<IWPFAddInHostView>(AddInSecurityLevel.Internet);

// Get and display add-in UI
FrameworkElement addInUI = this.wpfAddInHostView.GetAddInUI();
this.addInUIHostGrid.Children.Add(addInUI);
```

```vb
' Get add-in pipeline folder (the folder in which this application was launched from)
Dim appPath As String = Environment.CurrentDirectory

' Rebuild visual add-in pipeline
Dim warnings() As String = AddInStore.Rebuild(appPath)
If warnings.Length > 0 Then
    Dim msg As String = "Could not rebuild pipeline:"
    For Each warning As String In warnings
        msg &= vbLf & warning
    Next warning
    MessageBox.Show(msg)
    Return
End If

' Activate add-in with Internet zone security isolation
Dim addInTokens As Collection(Of AddInToken) = AddInStore.FindAddIns(GetType(IWPFAddInHostView), appPath)
Dim wpfAddInToken As AddInToken = addInTokens(0)
Me.wpfAddInHostView = wpfAddInToken.Activate(Of IWPFAddInHostView)(AddInSecurityLevel.Internet)

' Get and display add-in UI
Dim addInUI As FrameworkElement = Me.wpfAddInHostView.GetAddInUI()
Me.addInUIHostGrid.Children.Add(addInUI)
```

See also

- Add-ins and Extensibility
- WPF Add-Ins Overview

How to: Create an Add-In That Is a UI

1/23/2019 • 5 minutes to read • Edit Online

This example shows how to create an add-in that is a Windows Presentation Foundation (WPF) which is hosted by a WPF standalone application.

The add-in is a UI that is a WPF user control. The content of the user control is a single button that, when clicked, displays a message box. The WPF standalone application hosts the add-in UI as the content of the main application window.

Prerequisites

This example highlights the WPF extensions to the .NET Framework add-in model that enable this scenario, and assumes the following:

- Knowledge of the .NET Framework add-in model, including pipeline, add-in, and host development. If you are unfamiliar with these concepts, see Add-ins and Extensibility. For a tutorial that demonstrates the implementation of a pipeline, an add-in, and a host application, see Walkthrough: Creating an Extensible Application.
- Knowledge of the WPF extensions to the .NET Framework add-in model. See WPF Add-Ins Overview.

Example

To create an add-in that is a WPF UI requires specific code for each pipeline segment, the add-in, and the host application.

Implementing the Contract Pipeline Segment

When an add-in is a UI, the contract for the add-in must implement INativeHandleContract. In the example, `IWPFAddInContract` implements INativeHandleContract, as shown in the following code.

```
using System.AddIn.Contract;
using System.AddIn.Pipeline;

namespace Contracts
{
    /// <summary>
    /// Defines the services that an add-in will provide to a host application.
    /// In this case, the add-in is a UI.
    /// </summary>
    [AddInContract]
    public interface IWPFAddInContract : INativeHandleContract {}
}
```

Implementing the Add-In View Pipeline Segment

Because the add-in is implemented as a subclass of the FrameworkElement type, the add-in view must also subclass FrameworkElement. The following code shows the add-in view of the contract, implemented as the `WPFAddInView` class.

```
using System.AddIn.Pipeline;
using System.Windows.Controls;

namespace AddInViews
{
    /// <summary>
    /// Defines the add-in's view of the contract.
    /// </summary>
    [AddInBase]
    public class WPFAddInView : UserControl { }
}
```

Here, the add-in view is derived from UserControl. Consequently, the add-in UI should also derive from UserControl.

Implementing the Add-In-Side Adapter Pipeline Segment

While the contract is an INativeHandleContract, the add-in is a FrameworkElement (as specified by the add-in view pipeline segment). Therefore, the FrameworkElement must be converted to an INativeHandleContract before crossing the isolation boundary. This work is performed by the add-in-side adapter by calling ViewToContractAdapter, as shown in the following code.

```
using System;
using System.AddIn.Contract;
using System.AddIn.Pipeline;
using System.Security.Permissions;

using AddInViews;
using Contracts;

namespace AddInSideAdapters
{
    /// <summary>
    /// Adapts the add-in's view of the contract to the add-in contract
    /// </summary>
    [AddInAdapter]
    public class WPFAddIn_ViewToContractAddInSideAdapter : ContractBase, IWPFAddInContract
    {
        WPFAddInView wpfAddInView;

        public WPFAddIn_ViewToContractAddInSideAdapter(WPFAddInView wpfAddInView)
        {
            // Adapt the add-in view of the contract (WPFAddInView)
            // to the contract (IWPFAddInContract)
            this.wpfAddInView = wpfAddInView;
        }

        /// <summary>
        /// ContractBase.QueryContract must be overridden to:
        /// * Safely return a window handle for an add-in UI to the host
        ///   application's application.
        /// * Enable tabbing between host application UI and add-in UI, in the
        ///   "add-in is a UI" scenario.
        /// </summary>
        public override IContract QueryContract(string contractIdentifier)
        {
            if (contractIdentifier.Equals(typeof(INativeHandleContract).AssemblyQualifiedName))
            {
                return FrameworkElementAdapters.ViewToContractAdapter(this.wpfAddInView);
            }

            return base.QueryContract(contractIdentifier);
        }

        /// <summary>
        /// GetHandle is called by the WPF add-in model from the host application's
        /// application domain to get the window handle for an add-in UI from the
        /// add-in's application domain. GetHandle is called if a window handle isn't
        /// returned by other means ie overriding ContractBase.QueryContract,
        /// as shown above.
        /// NOTE: This method requires UnmanagedCodePermission to be called
        ///       (full-trust by default), to prevent illegal window handle
        ///       access in partially trusted scenarios. If the add-in could
        ///       run in a partially trusted application domain
        ///       (eg AddInSecurityLevel.Internet), you can safely return a window
        ///       handle by overriding ContractBase.QueryContract, as shown above.
        /// </summary>
        [SecurityPermissionAttribute(SecurityAction.Demand, Flags = SecurityPermissionFlag.UnmanagedCode)]
        public IntPtr GetHandle()
        {
            return FrameworkElementAdapters.ViewToContractAdapter(this.wpfAddInView).GetHandle();
        }
    }
}
```

In the add-in model where an add-in returns a UI (see Create an Add-In That Returns a UI), the add-in adapter converted the FrameworkElement to an INativeHandleContract by calling ViewToContractAdapter. ViewToContractAdapter must also be called in this model, although you need to implement a method from which to write the code to call it. You do this by overriding QueryContract and implementing the code that calls ViewToContractAdapter if the code that is calling QueryContract is expecting an INativeHandleContract. In this case, the caller will be the host-side adapter, which is covered in a subsequent subsection.

> **NOTE**
>
> You also need to override QueryContract in this model to enable tabbing between host application UI and add-in UI. For more information, see "WPF Add-In Limitations" in WPF Add-Ins Overview.

Because the add-in-side adapter implements an interface that derives from INativeHandleContract, you also need to implement GetHandle, although this is ignored when QueryContract is overridden.

Implementing the Host View Pipeline Segment

In this model, the host application typically expects the host view to be a FrameworkElement subclass. The host-side adapter must convert the INativeHandleContract to a FrameworkElement after the INativeHandleContract crosses the isolation boundary. Because a method isn't being called by the host application to get the FrameworkElement, the host view must "return" the FrameworkElement by containing it. Consequently, the host view must derive from a subclass of FrameworkElement that can contain other UIs, such as UserControl. The following code shows the host view of the contract, implemented as the `WPFAddInHostView` class.

Implementing the Host-Side Adapter Pipeline Segment

While the contract is an INativeHandleContract, the host application expects a UserControl (as specified by the host view). Consequently, the INativeHandleContract must be converted to a FrameworkElement after crossing the isolation boundary, before being set as content of the host view (which derives from UserControl).

This work is performed by the host-side adapter, as shown in the following code.

As you can see, the host-side adapter acquires the INativeHandleContract by calling the add-in-side adapter's QueryContract method (this is the point where the INativeHandleContract crosses the isolation boundary).

The host-side adapter then converts the INativeHandleContract to a FrameworkElement by calling ContractToViewAdapter. Finally, the FrameworkElement is set as the content of the host view.

Implementing the Add-In

With the add-in-side adapter and add-in view in place, the add-in can be implemented by deriving from the add-in view, as shown in the following code.

From this example, you can see one interesting benefit of this model: add-in developers only need to implement the add-in (since it is the UI as well), rather than both an add-in class and an add-in UI.

Implementing the Host Application

With the host-side adapter and host view created, the host application can use the .NET Framework add-in model to open the pipeline and acquire a host view of the add-in. These steps are shown in the following code.

The host application uses typical .NET Framework add-in model code to activate the add-in, which implicitly returns the host view to the host application. The host application subsequently displays the host view (which is a UserControl) from a Grid.

The code for processing interactions with the add-in UI runs in the add-in's application domain. These interactions include the following:

- Handling the ButtonClick event.

- Showing the MessageBox.

This activity is completely isolated from the host application.

See also

- Add-ins and Extensibility
- WPF Add-Ins Overview

Hosting WPF Applications

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

WPF XAML Browser Applications (XBAPs) are rich-client applications that can be deployed to a Web server and started in a browser. The WPF Host (PresentationHost.exe) is registered as the shell and MIME handler for XBAP and XAML files. Therefore, Internet Explorer knows to start the WPF Host when an XBAP is launched. Firefox users can install Firefox add-ons that enable Firefox to host XBAPs as well. An XBAP can be hosted in other browsers or stand-alone applications by using the native browser hosting APIs provided by WPF.

In This Section

WPF XAML Browser Applications Overview
WPF Host (PresentationHost.exe)
Firefox Add-ons to Support .NET Application Deployment
Native WPF Browser Hosting Support APIs

Related Sections

Application Management Overview
Windows in WPF
Navigation Overview
Build and Deploy

WPF XAML Browser Applications Overview

1/23/2019 • 9 minutes to read • Edit Online

XAML browser applications (XBAPs) combines features of both Web applications and rich-client applications. Like Web applications, XBAPs can be deployed to a Web server and started from Internet Explorer or Firefox. Like rich-client applications, XBAPs can take advantage of the capabilities of WPF. Developing XBAPs is also similar to rich-client development. This topic provides a simple, high-level introduction to XBAP development and describes where XBAP development differs from standard rich-client development.

This topic contains the following sections:

Creating a New XAML Browser Application (XBAP)

The simplest way to create a new XBAP project is with Microsoft Visual Studio. When creating a new project, select **WPF Browser Application** from the list of templates. For more information, see How to: Create a New WPF Browser Application Project.

When you run the XBAP project, it opens in a browser window instead of a stand-alone window. When you debug the XBAP from Visual Studio, the application runs with Internet zone permission and will therefore throw security exceptions if those permissions are exceeded. For more information, see Security and WPF Partial Trust Security.

NOTE

If you are not developing with Visual Studio or want to learn more about the project files, see Building a WPF Application.

Deploying an XBAP

When you build an XBAP, the output includes the following three files:

FILE	DESCRIPTION
Executable (.exe)	This contains the compiled code and has an .exe extension.
Application manifest (.manifest)	This contains metadata associated with the application and has a .manifest extension.
Deployment manifest (.xbap)	This file contains the information that ClickOnce uses to deploy the application and has the .xbap extension.

You deploy XBAPs to a Web server, for example Microsoft Internet Information Services (IIS) 5.0 or later versions. You do not have to install the .NET Framework on the Web server, but you do have to register the WPF

Multipurpose Internet Mail Extensions (MIME) types and file name extensions. For more information, see Configure IIS 5.0 and IIS 6.0 to Deploy WPF Applications.

To prepare your XBAP for deployment, copy the .exe and the associated manifests to the Web server. Create an HTML page that contains a hyperlink to open the deployment manifest, which is the file that has the .xbap extension. When the user clicks the link to the .xbap file, ClickOnce automatically handles the mechanics of downloading and starting the application. The following example code shows an HTML page that contains a hyperlink that points to an XBAP.

```
<html>
    <head></head>
    <body>
        <a href="XbapEx.xbap">Click this link to launch the application</a>
    </body>
</html>
```

You can also host an XBAP in the frame of a Web page. Create a Web page with one or more frames. Set the source property of a frame to the deployment manifest file. If you want to use the built-in mechanism to communicate between the hosting Web page and the XBAP, you must host the application in a frame. The following example code shows an HTML page with two frames, the source for the second frame is set to an XBAP.

```
<html>
    <head>
        <title>A page with frames</title>
    </head>
    <frameset cols="50%,50%">
        <frame src="introduction.htm">
        <frame src="XbapEx.xbap">
    </frameset>
</html>
```

Clearing Cached XBAPs

In some situations after rebuilding and starting your XBAP, you may find that an earlier version of the XBAP is opened. For example, this behavior may occur when your XBAP assembly version number is static and you start the XBAP from the command line. In this case, because the version number between the cached version (the version that was previously started) and the new version remains the same, the new version of the XBAP is not downloaded. Instead, the cached version is loaded.

In these situations, you can remove the cached version by using the **Mage** command (installed with Visual Studio or the Windows SDK) at the command prompt. The following command clears the application cache.

```
Mage.exe -cc
```

This command guarantees that the latest version of your XBAP is started. When you debug your application in Visual Studio, the latest version of your XBAP should be started. In general, you should update your deployment version number with each build. For more information about Mage, see Mage.exe (Manifest Generation and Editing Tool).

Communicating with the Host Web Page

When the application is hosted in an HTML frame, you can communicate with the Web page that contains the XBAP. You do this by retrieving the HostScript property of BrowserInteropHelper. This property returns a script object that represents the HTML window. You can then access the properties, methods, and events on the window object by using regular dot syntax. You can also access script methods and global variables. The following example shows how to retrieve the script object and close the browser.

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Retrieve the script object. The XBAP must be hosted in a frame or
    // the HostScript object will be null.
    var scriptObject = BrowserInteropHelper.HostScript;

    // Call close to close the browser window.
    scriptObject.Close();
}
```

```
Private Sub Button_Click(ByVal sender As Object, ByVal e As RoutedEventArgs)
    ' Retrieve the script object  The XBAP must be hosted in a frame or
    ' the HostScript object will be null.
    Dim scriptObject = BrowserInteropHelper.HostScript

    ' Call close to close the browser window.
    scriptObject.Close()
End Sub
```

Debugging XBAPs that Use HostScript

If your XBAP uses the HostScript object to communicate with the HTML window, there are two settings that you must specify to run and debug the application in Visual Studio. The application must have access to its site of origin and you must start the application with the HTML page that contains the XBAP. The following steps describe how to check these two settings:

1. In Visual Studio, open the project properties.
2. On the **Security** tab, click **Advanced**.

   The Advanced Security Settings dialog box appears.

3. Make sure that the **Grant the application access to its site of origin** check box is checked and then click **OK**.
4. On the **Debug** tab, select the **Start browser with URL** option and specify the URL for the HTML page that contains the XBAP.
5. In Internet Explorer, click the **Tools** button and then select **Internet Options**.

   The Internet Options dialog box appears.

6. Click the **Advanced** tab.
7. In the **Settings** list under **Security**, check the **Allow active content to run in files on My Computer** check box.
8. Click **OK**.

   The changes will take effect after you restart Internet Explorer.

Caution

Enabling active content in Internet Explorer may put your computer at risk. If you do not want to change your Internet Explorer security settings, you can launch the HTML page from a server and attach the Visual Studio debugger to the process.

XBAP Security Considerations

XBAPs typically execute in a partial-trust security sandbox that is restricted to the Internet zone permission set.

Consequently, your implementation must support the subset of WPF elements that are supported in the Internet zone or you must elevate the permissions of your application. For more information, see Security.

When you use a WebBrowser control in your application, WPF internally instantiates the native WebBrowser ActiveX control. When your application is a partial-trust XBAP running in Internet Explorer, the ActiveX control runs in a dedicated thread of the Internet Explorer process. Therefore, the following limitations apply:

- The WebBrowser control should provide behavior similar to the host browser, including security restrictions. Some of these security restrictions can be controlled through the Internet Explorer security settings. For more information, see Security.
- An exception is thrown when an XBAP is loaded cross-domain in an HTML page.
- Input is on a separate thread from the WPF WebBrowser, so keyboard input cannot be intercepted and the IME state is not shared.
- The timing or order of navigation may be different due to the ActiveX control running on another thread. For example, navigating to a page is not always cancelled by starting another navigation request.
- A custom ActiveX control may have trouble with communication since the WPF application is running in a separate thread.
- MessageHook does not get raised because HwndHost cannot subclass a window running in another thread or process.

Creating a Full-Trust XBAP

If your XBAP requires full trust, you can change your project to enable this permission. The following steps describe how to enable full trust:

1. In Visual Studio, open the project properties.
2. On the **Security** tab, select the **This is a full trust application** option.

This setting makes the following changes:

- In the project file, the `<TargetZone>` element value is changed to `Custom`.
- In the application manifest (app.manifest), an `Unrestricted="true"` attribute is added to the `PermissionSet` element.

```
<PermissionSet class="System.Security.PermissionSet"
               version="1"
               ID="Custom"
               SameSite="site"
               Unrestricted="true" />
```

Deploying a Full-Trust XBAP

When you deploy a full-trust XBAP that does not follow the ClickOnce Trusted Deployment model, the behavior when the user runs the application will depend on the security zone. In some cases, the user will receive a warning when they attempt to install it. The user can choose to continue or cancel the installation. The following table describes the behavior of the application for each security zone and what you have to do for the application to receive full trust.

SECURITY ZONE	BEHAVIOR	GETTING FULL TRUST
Local computer	Automatic full trust	No action is needed.

SECURITY ZONE	BEHAVIOR	GETTING FULL TRUST
Intranet and trusted sites	Prompt for full trust	Sign the XBAP with a certificate so that the user sees the source in the prompt.
Internet	Fails with "Trust Not Granted"	Sign the XBAP with a certificate.

NOTE

The behavior described in the previous table is for full-trust XBAPs that do not follow the ClickOnce Trusted Deployment model.

It is recommended that you use the ClickOnce Trusted Deployment model for deploying a full-trust XBAP. This model allows your XBAP to be granted full trust automatically, regardless of the security zone, so that the user is not prompted. As part of this model, you must sign your application with a certificate from a trusted publisher. For more information, see Trusted Application Deployment Overview and Introduction to Code Signing.

XBAP Start Time Performance Considerations

An important aspect of XBAP performance is its start time. If an XBAP is the first WPF application to load, the *cold start* time can be ten seconds or more. This is because the progress page is rendered by WPF, and both the CLR and WPF must be cold-started to display the application.

Starting in .NET Framework 3.5 SP1, XBAP cold-start time is mitigated by displaying an unmanaged progress page early in the deployment cycle. The progress page appears almost immediately after the application is started, because it is displayed by native hosting code and rendered in HTML.

In addition, improved concurrency of the ClickOnce download sequence improves start time by up to ten percent. After ClickOnce downloads and validates manifests, the application download starts, and the progress bar starts to update.

See also

- Configure Visual Studio to Debug a XAML Browser Application to Call a Web Service
- Deploying a WPF Application

WPF Host (PresentationHost.exe)

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

Windows Presentation Foundation (WPF) Host (PresentationHost.exe) is the application that enables WPF applications to be hosted in compatible browsers (including Microsoft Internet Explorer 6 and later). By default, Windows Presentation Foundation (WPF) Host is registered as the shell and MIME handler for browser-hosted WPF content, which includes:

- Loose (uncompiled) XAML files (.xaml).
- XAML browser application (XBAP) (.xbap).

For files of these types, Windows Presentation Foundation (WPF) Host:

- Launches the registered HTML handler to host the Windows Presentation Foundation (WPF) content.
- Loads the right versions of the required common language runtime (CLR) and Windows Presentation Foundation (WPF) assemblies.
- Ensures the appropriate permission levels for the zone of deployment are in place.

This topic describes the command line parameters that can be used with PresentationHost.exe.

Usage

`PresentationHost.exe [parameters] uri|filename`

Parameters

PARAMETER	DESCRIPTION
filename	The path of the file to be activated. Can also be a URI.
-debug	When activating an application, does not commit it to or run it from the store. This only works when a local file is activated.
-debugSecurityZoneURL <url>	Used with a URL value to indicate to PresentationHost.exe that an application should be debugged as if it were deployed from the specified URL. This determines both the deployment zone and the site of origin.
-embedding	Required by OLE. If the `-event` or `-debug` parameter are specified, it is not necessary to specify the `-embedding` parameter, since that parameter is set internally.
-event <eventname>	Open the event with this name and signal it when PresentationHost.exe is initialized and ready to host WPF content. PresentationHost.exe will terminate if there was an error opening the event, such as if it has not already been created.

PARAMETER	DESCRIPTION
-launchApplication <url>	Launches a standalone ClickOnce application from the specified URL. Internet Explorer and WinINet security policy concerning .NET applications are applied.

Scenarios

Shell Handler

```
PresentationHost.exe example.xbap
```

MIME Handler

```
PresentationHost.exe -embedding example.xbap
```

Visual Studio Debugging

```
PresentationHost.exe -debug example.xbap
```

Visual Studio Debugging In Zone

```
PresentationHost.exe -debug -debugSecurityZoneURL http://www.example.com c:\folderpath\example.xbap
```

See also

- Security

Firefox Add-ons to Support .NET Application Deployment

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

The Windows Presentation Foundation (WPF) plug-in for Firefox and the .NET Framework Assistant for Firefox enable XAML browser applications (XBAPs), loose XAML, and ClickOnce applications to work with the Mozilla Firefox browser.

WPF Plug-in for Firefox

The WPF plug-in for Firefox enables XBAPs and loose XAML files to be navigated to and run at the top-level or in an HTML IFRAME in the Firefox browser. An XBAP is a WPF application that can be published to a Web server and launched within supported browsers. Loose XAML is a XAML-only file that can be navigated to and displayed in supported browsers, much like an XML file.

The WPF plug-in for Firefox is installed with the .NET Framework 3.5. Window 7 includes the .NET Framework 3.5, but does not include the WPF plug-in for Firefox. You cannot install the WPF plug-in for Firefox on Windows 7.

The .NET Framework 4 does not include the WPF plug-in for Firefox. However, if both the .NET Framework 3.5 and .NET Framework 4 are installed, the WPF plug-in for Firefox is installed with the .NET Framework 3.5. Therefore .NET Framework 4 applications will still run because the WPF Host will load the correct version of the framework. For more information, see WPF Host (PresentationHost.exe).

.NET Framework Assistant for Firefox

The .NET Framework Assistant for Firefox enables stand-alone ClickOnce applications to run from the Firefox browser. The .NET Framework Assistant for Firefox functions identically when it is installed before and after the Firefox browser. When the Firefox browser is launched and the .NET Framework 3.5 SP1 is installed, Firefox finds and installs the .NET Framework Assistant for Firefox. Users can configure the .NET Framework Assistant for Firefox to do the following:

- Prompt before running the ClickOnce application.
- Report all installed versions of the .NET Framework or just the latest version.

The .NET Framework Assistant for Firefox is included with the .NET Framework 3.5 SP1. For information about removing the .NET Framework Assistant for Firefox, see How to remove the .NET Framework Assistant for Firefox.

See also

- Deploying a WPF Application
- WPF XAML Browser Applications Overview
- Detect Whether the WPF Plug-In for Firefox Is Installed

Native WPF Browser Hosting Support APIs

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

Hosting of WPF applications in Web browsers is facilitated by an Active Document server (also known as a DocObject) registered out of the WPF Host. Internet Explorer can directly activate and integrate with an Active Document. For hosting of XBAPs and loose XAML documents in Mozilla browsers, WPF provides an NPAPI plugin, which provides a similar hosting environment to the WPF Active Document server as Internet Explorer does. However, the easiest practical way to host XBAPs and XAML documents in other browsers and standalone applications is via the Internet Explorer Web Browser control. The Web Browser control provides the complex Active Document server hosting environment, yet it enables its own host to customize and extend that environment and communicate directly with the current Active Document object.

The WPF Active Document server implements several common hosting interfaces, including IOleObject, IOleDocument, IOleInPlaceActiveObject, IPersistMoniker, IOleCommandTarget. When hosted in the Web Browser control, these interfaces can be queries from the object returned by the IWebBrowser2::Document property.

IOleCommandTarget

WPF Active Document server's implementation of IOleCommandTarget supports numerous navigation-related and browser-specific commands of the standard OLE command group (with a null command group GUID). In addition, it recognizes a custom command group called CGID_PresentationHost. Currently, there is only one command defined within this group.

```
DEFINE_GUID(CGID_PresentationHost, 0xd0288c55, 0xd6, 0x4f5e, 0xa8, 0x51, 0x79, 0xde, 0xc5, 0x1b, 0x10, 0xec);
enum PresentationHostCommands {
   PHCMDID_TABINTO = 1
};
```

PHCMDID_TABINTO instructs PresentationHost to switch focus to the first or last focusable element in its content, depending on the state of the Shift key.

In This Section

IEnumRAWINPUTDEVICE
IWpfHostSupport

IEnumRAWINPUTDEVICE

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

This interface enumerates the raw input devices, and is only used by PresentationHost.exe.

NOTE

This API is only intended and supported for use on the local client machine

Members

MEMBER	DESCRIPTION
IEnumRAWINPUTDEVIC:Next	Enumerates the next `celt` elements (that is, RAWINPUTDEVICE structures) in the enumerator's list, returning them in `rgelt` along with the actual number of enumerated elements in `pceltFetched`.
IEnumRAWINPUTDEVIC:Skip	Instructs the enumerator to skip the next `celt` elements in the enumeration so that the next call to IEnumRAWINPUTDEVIC:Next will not return those elements.
IEnumRAWINPUTDEVIC:Reset	Resets the enumeration sequence to the beginning.
IEnumRAWINPUTDEVIC:Clone	Creates another raw input device enumerator with the same state as the current enumerator to iterate over the same list.

See also

- About Raw Input

IEnumRAWINPUTDEVIC:Next

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

Enumerates the next `celt` RAWINPUTDEVICE structures in the enumerator's list, returning them in `rgelt` along with the actual number of enumerated elements in `pceltFetched`.

Syntax

```
HRESULT Next(
      [in] ULONG celt,
      [out, size_is(celt), length_is(*pceltFetched)] RAWINPUTDEVICE *rgelt,
      [out] ULONG *pceltFetched);
```

Parameters

`celt`

[in] Number of RAWINPUTDEVICE structures returned in `rgelt`.

`rgelt`

[out] Array of size celt (or larger) to receive enumerated RAWINPUTDEVICE structures.

`pceltFetched`

[out] Pointer to the number of elements actually supplied in `rgelt`. Caller can pass in `NULL` if `rgelt` is one.

Property Value/Return Value

HRESULT: S_OK if the number of elements supplied is `celt`; S_FALSE otherwise.

IEnumRAWINPUTDEVIC:Skip

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

Instructs the enumerator to skip the next `celt` elements in the enumeration so that the next call to IEnumRAWINPUTDEVIC:Next will not return those elements.

Syntax

```
HRESULT Skip( [in] ULONG celt);
```

Parameters

`celt`

[in] Number of elements to be skipped.

Property Value/Return Value

HRESULT: S_OK if the number of elements supplied is `celt`; S_FALSE otherwise.

IEnumRAWINPUTDEVIC:Reset

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

Resets the enumeration sequence to the beginning.

Syntax

```
HRESULT Reset();
```

Property Value/Return Value

HRESULT: S_OK.

IEnumRAWINPUTDEVIC:Clone

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

Creates another raw input device enumerator with the same state as the current enumerator to iterate over the same list.

Syntax

```
HRESULT Clone( [out] IEnumRAWINPUTDEVICE **ppenum);
```

Parameters

`ppenum`

[out] Address of output variable that receives the IEnumRAWINPUTDEVICE interface pointer. If the method is unsuccessful, the value of this output variable is undefined.

Property Value/Return Value

HRESULT: This method supports the standard return values E_INVALIDARG, E_OUTOFMEMORY, and E_UNEXPECTED.

Remarks

This method makes it possible to record a point in the enumeration sequence in order to return to that point at a later time. The caller must release this new enumerator separately from the first enumerator.

IWpfHostSupport

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

Applications that host Windows Presentation Foundation (WPF) content via PresentationHost.exe implement this interface to provide a point of integration between the host and PresentationHost.exe.

Remarks

Win32 applications such as Web browsers can host WPF content, including XAML browser applications (XBAPs) and loose XAML. To host WPF content, Win32 applications create an instance of the WebBrowser control. To be hosted, WPF creates an instance of PresentationHost.exe, which provides the hosted WPF content to the host for display in the WebBrowser control.

The integration enabled by `IWpfHostSupport` allows PresentationHost.exe to:

- Discover and register with the raw input devices (Human Interface Devices) that the host application is interested in.
- Receive input messages from the registered raw input devices and forward appropriate messages to the host application.
- Query the host application for custom progress and error user interfaces.

NOTE
This API is only intended and supported for use on the local client machine

Members

MEMBER	DESCRIPTION
GetRawInputDevices	Allows PresentationHost.exe to discover the raw input devices (Human Interface Devices) that the host application is interested in.
FilterInputMessage	Called by PresentationHost.exe whenever a message is received unless E_NOTIMPL is returned.
GetCustomUI	By default, PresentationHost.exe provides its own deployment progress and deployment error user interfaces that are displayed when WPF content is deployed.

GetRawInputDevices

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

Allows PresentationHost.exe to discover the raw input devices (Human Interface Devices) that the host application is interested in.

Syntax

```
HRESULT GetRawInputDevices( [out] IEnumRAWINPUTDEVICE **ppEnum );
```

Parameters

`ppEnum`

[out] A pointer to an IEnumRAWINPUTDEVICE for enumerating the raw input devices.

Property Value/Return Value

HRESULT:

S_OK - IEnumRAWINPUTDEVICE will only be used by PresentationHost.exe if S_OK is returned.

E_NOTIMPL

Remarks

Raw input devices are the set of input devices that includes keyboards, mice, and less traditional devices like remote controls.

Once the list of raw input devices has been retrieved, PresentationHost.exe registers with the devices to receive WM_INPUT notification messages.

See also

- GetRawInputDeviceList
- FilterInputMessage

FilterInputMessage

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

Called by PresentationHost.exe whenever a message is received unless E_NOTIMPL is returned.

Syntax

```
HRESULT FilterInputMessage( [in] MSG* pMsg ) ;
```

Parameters

`pMsg`

[in] The WM_INPUT message sent to the window that is getting raw input.

Property Value/Return Value

HRESULT:

S_OK - The filter did not process the message and further processing may occur.

S_FALSE - The filter processed this message and no further processing should occur.

E_NOTIMPL – If this value is returned, FilterInputMessage is not called again. This might be returned from a host application that is only interested in providing custom progress and error user interfaces to PresentationHost.exe is not interested in being forwarded raw input messages from PresentationHost.exe.

Remarks

PresentationHost.exe is the target of various raw input devices, including keyboard, mice, and remote controls. Sometimes, behavior in the host application is dependent on input that would otherwise be consumed by PresentationHost.exe. For example, a host application may depend on receiving certain input messages to determine whether or not to display specific user interface elements.

To allow the host application to receive the necessary input messages to provide these behaviors, PresentationHost.exe forwards appropriate raw input messages to the hosted application by calling FilterInputMessage.

The hosted application receives raw input messages by registering with the set of raw input devices (Human Interface Devices) returned by GetRawInputDevices.

See also

- WM_INPUT Notification

GetCustomUI

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

Called by PresentationHost.exe to get custom progress and error messages from the host, if implemented.

Syntax

```
HRESULT GetCustomUI( [out] BSTR* pwzProgressAssemblyName, [out] BSTR* pwzProgressClassName, [out] BSTR*
pwzErrorAssemblyName, [out] BSTR* pwzErrorClassName );
```

Parameters

`pwzProgressAssemblyName`

[out] A pointer to the assembly that contains the host-supplied progress user interface.

`pwzProgressClassName`

[out] The name of the class that is the host-supplied progress user interface, preferably a XAML file with Page is its top-level element. This class resides in the assembly that is specified by `pwzProgressAssemblyName`.

`pwzErrorAssemblyName`

[out] A pointer to the assembly that contains the host-supplied error user interface.

`pwzErrorClassName`

[out] The name of the class that is the host-supplied error user interface, preferably a XAML file with Page is its top-level element. This class resides in the assembly that is specified by `pwzErrorAssemblyName`.

Property Value/Return Value

HRESULT: Ignored.

Remarks

A host application may have a specific theme that PresentationHost.exe's default user interfaces may not conform to. If this is the case, the host application can implement GetCustomUI to return progress and error user interfaces to PresentationHost.exe. PresentationHost.exe will always call GetCustomUI before using its default user interfaces.

This function is called once during PresentationHost's initialization.

See also

- IWpfHostSupport

Building and Deploying WPF Applications

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

The build and deployment model provides the capability to build and deploy applications locally and remotely, including the following:

- MSBuild: the .NET build system located in the Microsoft.Build.Tasks.Windows namespace.
- Resources: working with UI resources.
- ClickOnce Deployment: the .NET publishing and deployment system.

In This Section

Building a WPF Application
Deploying a WPF Application
How-to Topics

Reference

MSBuild

Related Sections

Application Management Overview
Windows in WPF
Navigation Overview
WPF XAML Browser Applications Overview
Hosting

Building a WPF Application (WPF)

1/23/2019 • 8 minutes to read • Edit Online

Windows Presentation Foundation (WPF) applications can be built as .NET Framework executables (.exe), libraries (.dll), or a combination of both types of assemblies. This topic introduces how to build WPF applications and describes the key steps in the build process.

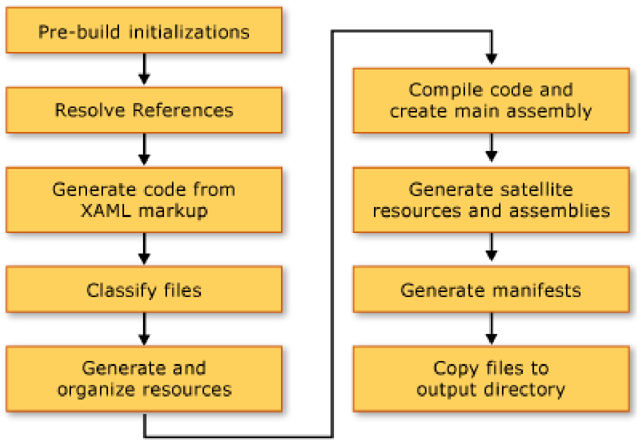
Building a WPF Application

A WPF application can be compiled in the following ways:

- Command-line. The application must contain only code (no XAML) and an application definition file. For more information, see Command-line Building With csc.exe or Building from the Command Line (Visual Basic).
- Microsoft Build Engine (MSBuild). In addition to the code and XAML files, the application must contain an MSBuild project file. For more information, see "MSBuild".
- Visual Studio. Visual Studio is an integrated development environment that compiles WPF applications with MSBuild and includes a visual designer for creating UI. For more information, see Application Development in Visual Studio and Design XAML in Visual Studio.

WPF Build Pipeline

When a WPF project is built, the combination of language-specific and WPF-specific targets are invoked. The process of executing these targets is called the build pipeline, and the key steps are illustrated by the following figure.


Pre-build initializations
Resolve References
Generate code from XAML markup
Classify files
Generate and organize resources
Compile code and create main assembly
Generate satellite resources and assemblies
Generate manifests
Copy files to output directory

Pre-Build Initializations

Before building, MSBuild determines the location of important tools and libraries, including the following:

- The .NET Framework.
- The Windows SDK directories.
- The location of WPF reference assemblies.
- The property for the assembly search paths.

The first location where MSBuild searches for assemblies is the reference assembly directory (%ProgramFiles%\Reference Assemblies\Microsoft\Framework\v3.0\). During this step, the build process also initializes the various properties and item groups and performs any required cleanup work.

Resolving References

The build process locates and binds the assemblies required to build the application project. This logic is contained in the `ResolveAssemblyReference` task. All assemblies declared as `Reference` in the project file are provided to the task along with information on the search paths and metadata on assemblies already installed on the system. The task looks up assemblies and uses the installed assembly's metadata to filter out those core WPF assemblies that need not show up in the output manifests. This is done to avoid redundant information in the ClickOnce manifests. For example, since PresentationFramework.dll can be considered representative of an application built on and for the WPF and moreover since all WPF assemblies exist at the same location on every machine that has the .NET Framework installed, there is no need to include all information on all .NET Framework reference assemblies in the manifests.

Markup Compilation—Pass 1

In this step, XAML files are parsed and compiled so that the runtime does not spend time parsing XML and validating property values. The compiled XAML file is pre-tokenized so that, at run time, loading it should be much faster than loading a XAML file.

During this step, the following activities take place for every XAML file that is a `Page` build item:

1. The XAML file is parsed by the markup compiler.

2. A compiled representation is created for that XAML and copied to the obj\Release folder.

3. A CodeDOM representation of a new partial class is created and copied to the obj\Release folder.

In addition, a language-specific code file is generated for every XAML file. For example, for a Page1.xaml page in a Visual Basic project, a Page1.g.vb is generated; for a Page1.xaml page in a C# project, a Page1.g.cs is generated. The ".g" in the file name indicates the file is generated code that has a partial class declaration for the top-level element of the markup file (such as `Page` or `Window`). The class is declared with the `partial` modifier in C# (`Extends` in Visual Basic) to indicate there is another declaration for the class elsewhere, usually in the code-behind file Page1.xaml.cs.

The partial class extends from the appropriate base class (such as Page for a page) and implements the System.Windows.Markup.IComponentConnector interface. The IComponentConnector interface has methods to initialize a component and connect names and events on elements in its content. Consequently, the generated code file has a method implementation like the following:

```
public void InitializeComponent() {
    if (_contentLoaded) {
        return;
    }
    _contentLoaded = true;
    System.Uri resourceLocater =
        new System.Uri(
            "window1.xaml",
            System.UriKind.RelativeOrAbsolute);
    System.Windows.Application.LoadComponent(this, resourceLocater);
}
```

```
Public Sub InitializeComponent() _

    If _contentLoaded Then
        Return
    End If

    _contentLoaded = True
    Dim resourceLocater As System.Uri = _
        New System.Uri("mainwindow.xaml", System.UriKind.Relative)

    System.Windows.Application.LoadComponent(Me, resourceLocater)

End Sub
```

By default, markup compilation runs in the same AppDomain as the MSBuild engine. This provides significant performance gains. This behavior can be toggled with the `AlwaysCompileMarkupFilesInSeparateDomain` property. This has the advantage of unloading all reference assemblies by unloading the separate AppDomain.

Markup Compilation—Pass 2

Not all XAML pages are compiled at during pass 1 of markup compilation. XAML files that have locally defined type references (references to types defined in code elsewhere in the same project) are exempt from compilation at this time. This is because those locally defined types exist only in source and have not yet been compiled. In order to determine this, the parser uses heuristics that involve looking for items such as `x:Name` in the markup file. When such an instance is found, that markup file's compilation is postponed until the code files have been compiled, after which, the second markup compilation pass processes these files.

File Classification

The build process puts output files into different resource groups based on which application assembly they will be placed in. In a typical nonlocalized application, all data files marked as `Resource` are placed in the main assembly (executable or library). When `UICulture` is set in the project, all compiled XAML files and those resources specifically marked as language-specific are placed in the satellite resource assembly. Furthermore, all language-neutral resources are placed in the main assembly. In this step of the build process, that determination is made.

The `ApplicationDefinition`, `Page`, and `Resource` build actions in the project file can be augmented with the `Localizable` metadata (acceptable values are `true` and `false`), which dictates whether the file is language-specific or language-neutral.

Core Compilation

The core compile step involves compilation of code files. This is orchestrated by logic in the language-specific targets files Microsoft.CSharp.targets and Microsoft.VisualBasic.targets. If heuristics have determined that a single pass of the markup compiler is sufficient, then the main assembly is generated. However, if one or more XAML files in the project have references to locally defined types, then a temporary .dll file is generated so the final application assemblies may be created after the second pass of markup compilation is complete.

Manifest Generation

At the end of the build process, after all the application assemblies and content files are ready, the ClickOnce manifests for the application are generated.

The deployment manifest file describes the deployment model: the current version, update behavior, and publisher identity along with digital signature. This manifest is intended to be authored by administrators who handle deployment. The file extension is .xbap (for XAML browser applications (XBAPs)) and .application for installed applications. The former is dictated by the `HostInBrowser` project property and as a result the manifest identifies the application as browser-hosted.

The application manifest (an .exe.manifest file) describes the application assemblies and dependent libraries and lists permissions required by the application. This file is intended to be authored by the application developer. In order to launch a ClickOnce application, a user opens the application's deployment manifest file.

These manifest files are always created for XBAPs. For installed applications, they are not created unless the `GenerateManifests` property is specified in the project file with value `true`.

XBAPs get two additional permissions over and above those permissions assigned to typical Internet zone applications: WebBrowserPermission and MediaPermission. The WPF build system declares those permissions in the application manifest.

Incremental Build Support

The WPF build system provides support for incremental builds. It is fairly intelligent about detecting changes made to markup or code, and it compiles only those artifacts affected by the change. The incremental build mechanism uses the following files:

- An $(*AssemblyName*)_MarkupCompiler.Cache file to maintain current compiler state.
- An $(*AssemblyName*)_MarkupCompiler.lref file to cache the XAML files with references to locally defined types.

The following is a set of rules governing incremental build:

- The file is the smallest unit at which the build system detects change. So, for a code file, the build system cannot tell if a type was changed or if code was added. The same holds for project files.
- The incremental build mechanism must be cognizant that a XAML page either defines a class or uses other classes.
- If `Reference` entries change, then recompile all pages.
- If a code file changes, recompile all pages with locally defined type references.
- If a XAML file changes:
  - If XAML is declared as `Page` in the project: if the XAML does not have locally defined type references, recompile that XAML plus all XAML pages with local references; if the XAML has local references, recompile all XAML pages with local references.
  - If XAML is declared as `ApplicationDefinition` in the project: recompile all XAML pages (reason: each XAML has reference to an Application type that may have changed).
- If the project file declares a code file as application definition instead of a XAML file:
  - Check if the `ApplicationClassName` value in the project file has changed (is there a new application type?). If so, recompile the entire application.
  - Otherwise, recompile all XAML pages with local references.
- If a project file changes: apply all preceding rules and see what needs to be recompiled. Changes to the following properties trigger a complete recompile: `AssemblyName`, `IntermediateOutputPath`, `RootNamespace`, and `HostInBrowser`.

The following recompile scenarios are possible:

- The entire application is recompiled.
- Only those XAML files that have locally defined type references are recompiled.
- Nothing is recompiled (if nothing in the project has changed).

See also

- Deploying a WPF Application
- WPF MSBuild Reference
- Pack URIs in WPF
- WPF Application Resource, Content, and Data Files

Deploying a WPF Application (WPF)

1/23/2019 • 5 minutes to read • Edit Online

After Windows Presentation Foundation (WPF) applications are built, they need to be deployed. Windows and the .NET Framework include several deployment technologies. The deployment technology that is used to deploy a WPF application depends on the application type. This topic provides a brief overview of each deployment technology, and how they are used in conjunction with the deployment requirements of each WPF application type.

Deployment Technologies

Windows and the .NET Framework include several deployment technologies, including:

- XCopy deployment.
- Windows Installer deployment.
- ClickOnce deployment.

XCopy Deployment

XCopy deployment refers to the use of the XCopy command-line program to copy files from one location to another. XCopy deployment is suitable under the following circumstances:

- The application is self-contained. It does not need to update the client to run.
- Application files must be moved from one location to another, such as from a build location (local disk, UNC file share, and so on) to a publish location (Web site, UNC file share, and so on).
- The application does not require shell integration (Start menu shortcut, desktop icon, and so on).

Although XCopy is suitable for simple deployment scenarios, it is limited when more complex deployment capabilities are required. In particular, using XCopy often incurs the overhead for creating, executing, and maintaining scripts for managing deployment in a robust way. Furthermore, XCopy does not support versioning, uninstallation, or rollback.

Windows Installer

Windows Installer allows applications to be packaged as self-contained executables that can be easily distributed to clients and run. Furthermore, Windows Installer is installed with Windows and enables integration with the desktop, the Start menu, and the Programs control panel.

Windows Installer simplifies the installation and uninstallation of applications, but it does not provide facilities for ensuring that installed applications are kept up-to-date from a versioning standpoint.

For more information about Windows Installer, see Windows Installer Deployment.

ClickOnce Deployment

ClickOnce enables Web-style application deployment for non-Web applications. Applications are published to and deployed from Web or file servers. Although ClickOnce does not support the full range of client features that Windows Installer-installed applications do, it does support a subset that includes the following:

- Integration with the Start menu and Programs control panel.
- Versioning, rollback, and uninstallation.
- Online install mode, which always launches an application from the deployment location.

- Automatic updating when new versions are released.
- Registration of file extensions.

For more information about ClickOnce, see ClickOnce Security and Deployment.

Deploying WPF Applications

The deployment options for a WPF application depend on the type of application. From a deployment perspective, WPF has three significant application types:

- Standalone applications.
- Markup-only XAML applications.
- XAML browser applications (XBAPs).

Deploying Standalone Applications

Standalone applications are deployed using either ClickOnce or Windows Installer. Either way, standalone applications require full trust to run. Full trust is automatically granted to standalone applications that are deployed using Windows Installer. Standalone applications that are deployed using ClickOnce are not automatically granted full trust. Instead, ClickOnce displays a security warning dialog that users must accept before a standalone application is installed. If accepted, the standalone application is installed and granted full trust. If not, the standalone application is not installed.

Deploying Markup-Only XAML Applications

Markup-only XAML pages are usually published to Web servers, like HTML pages, and can be viewed using Internet Explorer. Markup-only XAML pages run within a partial-trust security sandbox with restrictions that are defined by the Internet zone permission set. This provides an equivalent security sandbox to HTML-based Web applications.

For more information about security for WPF applications, see Security.

Markup-only XAML pages can be installed to the local file system by using either XCopy or Windows Installer. These pages can be viewed using Internet Explorer or Windows Explorer.

For more information about XAML, see XAML Overview (WPF).

Deploying XAML Browser Applications

XBAPs are compiled applications that require the following three files to be deployed:

- *ApplicationName*.exe: The executable assembly application file.
- *ApplicationName*.xbap: The deployment manifest.
- *ApplicationName*.exe.manifest: The application manifest.

NOTE

For more information about deployment and application manifests, see Building a WPF Application.

These files are produced when an XBAP is built. For more information, see How to: Create a New WPF Browser Application Project. Like markup-only XAML pages, XBAPs are typically published to a Web server and viewed using Internet Explorer.

XBAPs can be deployed to clients using any of the deployment techniques. However, ClickOnce is recommended since it provides the following capabilities:

1. Automatic updates when a new version is published.
2. Elevation privileges for the XBAP running with full trust.

By default, ClickOnce publishes application files with the .deploy extension. This can be problematic, but can be disabled. For more information, see Server and Client Configuration Issues in ClickOnce Deployments.

For more information about deploying XAML browser applications (XBAPs), see WPF XAML Browser Applications Overview.

Installing the .NET Framework

To run a WPF application, the Microsoft .NET Framework must be installed on the client. Internet Explorer automatically detects whether clients are installed with .NET Framework when WPF browser-hosted applications are viewed. If the .NET Framework is not installed, Internet Explorer prompts users to install it.

To detect whether the .NET Framework is installed, Internet Explorer includes a bootstrapper application that is registered as the fallback Multipurpose Internet Mail Extensions (MIME) handler for content files with the following extensions: .xaml, .xps, .xbap, and .application. If you navigate to these file types and the .NET Framework is not installed on the client, the bootstrapper application requests permission to install it. If permission is not provided, neither the .NET Framework nor the application is installed.

If permission is granted, Internet Explorer downloads and installs the .NET Framework using the Microsoft Background Intelligent Transfer Service (BITS). After successful installation of the .NET Framework, the originally requested file is opened in a new browser window.

.NET Framework auto-detection is available on Windows Vista, Microsoft Windows XP Service Pack 2 (SP2), and Microsoft Windows Server 2003 (SP1) clients that have Internet Explorer 7 or later installed.

For more information, see Deploying the .NET Framework and Applications.

See also

- Building a WPF Application
- Security

Build and Deploy How-to Topics

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

The following topics show how to create project files for the various WPF application types.

In This Section

Configure IIS 5.0 and IIS 6.0 to Deploy WPF Applications
Configure Visual Studio to Debug a XAML Browser Application to Call a Web Service
Determine the Installed Version of WPF
Detect Whether the .NET Framework 3.0 Is Installed
Detect Whether the .NET Framework 3.5 Is Installed
Detect Whether the WPF Plug-In for Firefox Is Installed

Related Sections

Building a WPF Application

Deploying a WPF Application

How to: Create a New WPF Application Project

How to: Create a New WPF Browser Application Project

How to: Configure IIS 5.0 and IIS 6.0 to Deploy WPF Applications

5/4/2018 • 3 minutes to read • Edit Online

You can deploy a Windows Presentation Foundation (WPF) application from most Web servers, as long as they are configured with the appropriate Multipurpose Internet Mail Extensions (MIME) types. By default, Microsoft Internet Information Services (IIS) 7.0 is configured with these MIME types, but Microsoft Internet Information Services (IIS) 5.0 and Microsoft Internet Information Services (IIS) 6.0 are not.

This topic describes how to configure Microsoft Internet Information Services (IIS) 5.0 and Microsoft Internet Information Services (IIS) 6.0 to deploy WPF applications.

NOTE

You can check the *UserAgent* string in the registry to determine whether a system has .NET Framework installed. For details and a script that examines the *UserAgent* string to determine whether .NET Framework is installed on a system, see Detect Whether the .NET Framework 3.0 Is Installed.

Adjust the Content Expiration Setting

You should adjust the content expiration setting to 1 minute. The following procedure outlines how to do this with IIS.

1. Click the **Start** menu, point to **Administrative Tools**, and click **Internet Information Services (IIS) Manager**. You can also launch this application from the command line with "%SystemRoot%\system32\inetsrv\iis.msc".
2. Expand the IIS tree until you find the **Default Web site** node.
3. Right-click **Default Web site** and select **Properties** from the context menu.
4. Select the **HTTP Headers** tab and click "Enable Content Expiration".
5. Set the content to expire after 1 minute.

Register MIME Types and File Extensions

You must register several MIME types and file extensions so that the browser on the client's system can load the correct handler. You need to add the following types:

EXTENSION	MIME TYPE
.manifest	application/manifest
.xaml	application/xaml+xml
.application	application/x-ms-application
.xbap	application/x-ms-xbap

EXTENSION	MIME TYPE
.deploy	application/octet-stream
.xps	application/vnd.ms-xpsdocument

NOTE

You do not need to register MIME types or file extensions on client systems. They are registered automatically when you install Microsoft .NET Framework.

The following Microsoft Visual Basic Scripting Edition (VBScript) sample automatically adds the necessary MIME types to IIS. To use the script, copy the code to a .vbs file on your server. Then, run the script by running the file from the command line or double-clicking the file in Microsoft Windows Explorer.

```vb
' This script adds the necessary Windows Presentation Foundation MIME types
' to an IIS Server.
' To use this script, just double-click or execute it from a command line.
' Running this script multiple times results in multiple entries in the IIS MimeMap.

Dim MimeMapObj, MimeMapArray, MimeTypesToAddArray, WshShell, oExec
Const ADS_PROPERTY_UPDATE = 2

' Set the MIME types to be added
MimeTypesToAddArray = Array(".manifest", "application/manifest", ".xaml", _
    "application/xaml+xml", ".application", "application/x-ms-application", _
    ".deploy", "application/octet-stream", ".xbap", "application/x-ms-xbap", _
    ".xps", "application/vnd.ms-xpsdocument")

' Get the mimemap object
Set MimeMapObj = GetObject("IIS://LocalHost/MimeMap")

' Call AddMimeType for every pair of extension/MIME type
For counter = 0 to UBound(MimeTypesToAddArray) Step 2
    AddMimeType MimeTypesToAddArray(counter), MimeTypesToAddArray(counter+1)
Next

' Create a Shell object
Set WshShell = CreateObject("WScript.Shell")

' Stop and Start the IIS Service
Set oExec = WshShell.Exec("net stop w3svc")
Do While oExec.Status = 0
    WScript.Sleep 100
Loop

Set oExec = WshShell.Exec("net start w3svc")
Do While oExec.Status = 0
    WScript.Sleep 100
Loop

Set oExec = Nothing

' Report status to user
WScript.Echo "Windows Presentation Foundation MIME types have been registered."

' AddMimeType Sub
Sub AddMimeType (Ext, MType)

    ' Get the mappings from the MimeMap property.
    MimeMapArray = MimeMapObj.GetEx("MimeMap")

    ' Add a new mapping.
    i = UBound(MimeMapArray) + 1
    Redim Preserve MimeMapArray(i)
    Set MimeMapArray(i) = CreateObject("MimeMap")
    MimeMapArray(i).Extension = Ext
    MimeMapArray(i).MimeType = MType
    MimeMapObj.PutEx ADS_PROPERTY_UPDATE, "MimeMap", MimeMapArray
    MimeMapObj.SetInfo

End Sub
```

NOTE

Running this script multiple times creates multiple MIME map entries in the Microsoft Internet Information Services (IIS) 5.0 or Microsoft Internet Information Services (IIS) 6.0 metabase.

After you have run this script, you may not see additional MIME types from the Microsoft Internet Information

Services (IIS) 5.0 or Microsoft Internet Information Services (IIS) 6.0 Microsoft Management Console (MMC). However, these MIME types have been added to the Microsoft Internet Information Services (IIS) 5.0 or Microsoft Internet Information Services (IIS) 6.0 metabase. The following script will display all the MIME types in the Microsoft Internet Information Services (IIS) 5.0 or Microsoft Internet Information Services (IIS) 6.0 metabase.

```
' This script lists the MIME types for an IIS Server.
' To use this script, just double-click or execute it from a command line
' by calling cscript.exe

dim mimeMapEntry, allMimeMaps

' Get the mimemap object.
Set mimeMapEntry = GetObject("IIS://localhost/MimeMap")
allMimeMaps = mimeMapEntry.GetEx("MimeMap")

' Display the mappings in the table.
For Each mimeMap In allMimeMaps
    WScript.Echo(mimeMap.MimeType & " (" & mimeMap.Extension + ")")
Next
```

Save the script as a `.vbs` file (for example, `DiscoverIISMimeTypes.vbs`) and run it from the command prompt using the following command:

```
cscript DiscoverIISMimeTypes.vbs
```

How to: Configure Visual Studio to Debug a XAML Browser Application to Call a Web Service

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

XAML browser applications (XBAPs) run within a partial-trust security sandbox that is restricted to the Internet zone set of permissions. This permission set restricts Web service calls to only Web services that are located at the XBAP application's site of origin. When an XBAP is debugged from Visual Studio 2005, though, it is not considered to have the same site of origin as the Web service it references. This causes security exceptions to be raised when the XBAP attempts to call the Web service. However, a Visual Studio 2005 XAML Browser Application (WPF) project can be configured to simulate having the same site of origin as the Web service it calls while debugging. This allows the XBAP to safely call the Web service without causing security exceptions.

Configuring Visual Studio

To configure Visual Studio 2005 to debug an XBAP that calls a Web service:

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. In the **Project Designer**, click the **Debug** tab.
3. In the **Start Action** section, select **Start external program** and enter the following:

   `C:\WINDOWS\System32\PresentationHost.exe`

4. In the **Start Options** section, enter the following into the **Command line arguments** text box:

   `-debug` *filename*

   The *filename* value for the **-debug** parameter is the .xbap filename; for example:

   `-debug c:\example.xbap`

NOTE
This is the default configuration for solutions that are created with the Visual Studio 2005 XAML Browser Application (WPF) project template.

1. With a project selected in **Solution Explorer**, on the **Project** menu, click **Properties**.
2. In the **Project Designer**, click the **Debug** tab.
3. In the **Start Options** section, add the following command-line parameter to the **Command line arguments** text box:

   `-debugSecurityZoneURL` *URL*

   The *URL* value for the **-debugSecurityZoneURL** parameter is the URL for the location that you want to simulate as being the site of origin of your application.

As an example, consider a XAML browser application (XBAP) that uses a Web service with the following URL:

`http://services.msdn.microsoft.com/ContentServices/ContentService.asmx`

The site of origin URL for this Web service is:

```
http://services.msdn.microsoft.com
```

Consequently, the complete **-debugSecurityZoneURL** command-line parameter and value is:

```
-debugSecurityZoneURL http://services.msdn.microsoft.com
```

See also

- WPF Host (PresentationHost.exe)

How to: Determine the Installed Version of WPF

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

The version number for the current installed version of Windows Presentation Foundation (WPF) is located in the **Registry**.

To find the version number:

1. On the **Start** menu, click **Run**.

2. In **Open**, type **regedit.exe**.

3. Open the following key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v3.0\Setup\Windows Presentation Foundation
```

The WPF version number is stored in the **Version** value.

How to: Detect Whether the .NET Framework 3.0 Is Installed

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

Before administrators can deploy Microsoft .NET Framework applications on a system, they must first confirm that the .NET Framework runtime is present. This topic provides a script written in HTML/JavaScript that administrators can use to determine whether the .NET Framework is present on a system.

NOTE

For more detailed information on installing, deploying, and detecting the Microsoft .NET Framework, see the discussion in Deploying Microsoft .NET Framework Version 3.0.

Detect the ".NET CLR" User-Agent String

When .NET Framework is installed, the MSI adds ".NET CLR" and the version number to the UserAgent string. The following example shows a script embedded in a simple HTML page. The script searches the UserAgent string to determine whether .NET Framework is installed, and displays a status message on the results of the search.

```
<HTML>
  <HEAD>
    <TITLE>Test for the .NET Framework 3.0</TITLE>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8" />
    <SCRIPT LANGUAGE="JavaScript">
    <!--
    var dotNETRuntimeVersion = "3.0.04425.00";

    function window::onload()
    {
      if (HasRuntimeVersion(dotNETRuntimeVersion))
      {
        result.innerText =
          "This machine has the correct version of the .NET Framework 3.0: "
          + dotNETRuntimeVersion
      }
      else
      {
        result.innerText =
          "This machine does not have the correct version of the .NET Framework 3.0."
      }
      result.innerText += "\n\nThis machine's userAgent string is: " +
        navigator.userAgent + ".";
    }

    //
    // Retrieve the version from the user agent string and
    // compare with the specified version.
    //
    function HasRuntimeVersion(versionToCheck)
    {
      var userAgentString =
        navigator.userAgent.match(/.NET CLR [0-9.]+/g);

      if (userAgentString != null)
      {
        var i;
```

```
          for (i = 0; i < userAgentString.length; ++i)
          {
            if (CompareVersions(GetVersion(versionToCheck),
              GetVersion(userAgentString[i])) <= 0)
              return true;
          }
        }

        return false;
      }

      //
      // Extract the numeric part of the version string.
      //
      function GetVersion(versionString)
      {
        var numericString =
          versionString.match(/([0-9]+)\.([0-9]+)\.([0-9]+)/i);
        return numericString.slice(1);
      }

      //
      // Compare the 2 version strings by converting them to numeric format.
      //
      function CompareVersions(version1, version2)
      {
        for (i = 0; i < version1.length; ++i)
        {
          var number1 = new Number(version1[i]);
          var number2 = new Number(version2[i]);

          if (number1 < number2)
            return -1;

          if (number1 > number2)
            return 1;
        }

        return 0;
      }

      -->
      </SCRIPT>
    </HEAD>

    <BODY>
      <div id="result" />
    </BODY>
  </HTML>
```

If the search for the ".NET CLR " version is successful, the following type of status message appears:

```
This machine has the correct version of the .NET Framework 3.0: 3.0.04425.00
```

```
This machine's userAgent string is: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322;
InfoPath.1; .NET CLR 2.0.50727; .NET CLR 3.0.04425.00).
```

Otherwise, the following type of status message appears:

```
This machine does not have correct version of the .NET Framework 3.0.
```

```
This machine's userAgent string is: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322;
InfoPath.1; .NET CLR 2.0.50727).
```

How to: Detect Whether the .NET Framework 3.5 Is Installed

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

Before administrators can deploy Windows Presentation Foundation (WPF) applications on a system that targets the .NET Framework 3.5, they must first confirm that the .NET Framework 3.5 runtime is present. This topic provides a script written in HTML/JavaScript that administrators can use to determine whether the .NET Framework 3.5 is present on a system.

NOTE

For more detailed information on installing, deploying, and detecting the .NET Framework, see Install the .NET Framework for developers.

Example

When the .NET Framework 3.5 is installed, the MSI adds ".NET CLR" and the version number to the UserAgent string. The following example shows a script embedded in a simple HTML page. The script searches the UserAgent string to determine whether the .NET Framework 3.5 is installed, and displays a status message on the results of the search.

NOTE

This script is designed for Internet Explorer. Other browsers may not include .NET CLR information in the UserAgent string.

```
<HTML>
  <HEAD>
    <TITLE>Test for the .NET Framework 3.5</TITLE>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8" />
    <SCRIPT LANGUAGE="JavaScript">
    <!--
    var dotNETRuntimeVersion = "3.5.0.0";

    function window::onload()
    {
      if (HasRuntimeVersion(dotNETRuntimeVersion))
      {
        result.innerText =
          "This machine has the correct version of the .NET Framework 3.5."
      }
      else
      {
        result.innerText =
          "This machine does not have the correct version of the .NET Framework 3.5." +
          " The required version is v" + dotNETRuntimeVersion + ".";
      }
      result.innerText += "\n\nThis machine's userAgent string is: " +
        navigator.userAgent + ".";
    }

    //
    // Retrieve the version from the user agent string and
    // compare with the specified version.
    //
    function HasRuntimeVersion(versionToCheck)
    {
      var userAgentString =
        navigator.userAgent.match(/.NET CLR [0-9.]+/g);

      if (userAgentString != null)
      {
        var i;

        for (i = 0; i < userAgentString.length; ++i)
        {
          if (CompareVersions(GetVersion(versionToCheck),
            GetVersion(userAgentString[i])) <= 0)
            return true;
        }
      }

      return false;
    }

    //
    // Extract the numeric part of the version string.
    //
    function GetVersion(versionString)
    {
      var numericString =
        versionString.match(/([0-9]+)\.([0-9]+)\.([0-9]+)/i);
      return numericString.slice(1);
    }

    //
    // Compare the 2 version strings by converting them to numeric format.
    //
    function CompareVersions(version1, version2)
    {
      for (i = 0; i < version1.length; ++i)
      {
        var number1 = new Number(version1[i]);
        var number2 = new Number(version2[i]);

        if (number1 < number2)
          return -1;

        if (number1 > number2)
          return 1;
      }

      return 0;
    }

    -->
    </SCRIPT>
  </HEAD>

  <BODY>
    <div id="result" />
  </BODY>
</HTML>
```

If the search for the ".NET CLR " version is successful, the following type of status message appears:

```
This machine has the correct version of the .NET Framework 3.5.
```

```
This machine's userAgent string is: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; InfoPath.2; .NET CLR 3.0.590; .NET CLR 3.5.20726; MS-RTC LM 8).
```

Otherwise, the following type of status message appears:

```
This machine does not have the correct version of the .NET Framework 3.5. The required version is v3.5.0.0.
```

```
This machine's userAgent string is: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; SLCC1; .NET CLR
2.0.50727; .NET CLR 1.1.4322; InfoPath.2; .NET CLR 3.0.590; MS-RTC LM 8).
```

See also

- Detect Whether the .NET Framework 3.0 Is Installed

How to: Detect Whether the WPF Plug-In for Firefox Is Installed

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

The Windows Presentation Foundation (WPF) plug-in for Firefox enables XAML browser applications (XBAPs) and loose XAML files to run in the Mozilla Firefox browser. This topic provides a script written in HTML and JavaScript that administrators can use to determine whether the WPF plug-in for Firefox is installed.

NOTE

For more information about installing, deploying, and detecting the .NET Framework, see Install the .NET Framework for developers.

Example

When the .NET Framework 3.5 is installed, the client computer is configured with a WPF plug-in for Firefox. The following example script checks for the WPF plug-in for Firefox and then displays an appropriate status message.

```
<HTML>

  <HEAD>
    <TITLE>Test for the WPF plug-in for Firefox</TITLE>
    <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=utf-8" />
    <SCRIPT type="text/javascript">
    <!--
    function OnLoad()
    {

       // Check for the WPF plug-in for Firefox and report
       var msg = "The WPF plug-in for Firefox is ";
       var wpfPlugin = navigator.plugins["Windows Presentation Foundation"];
       if( wpfPlugin != null ) {
          document.writeln(msg + " installed.");
       }
       else {
          document.writeln(msg + " not installed. Please install or reinstall the .NET Framework 3.5.");
       }
    }
    -->
    </SCRIPT>
  </HEAD>

  <BODY onload="OnLoad()" />

</HTML>
```

If the check for the WPF plug-in for Firefox is successful, the following status message is displayed:

```
The WPF plug-in for Firefox is installed.
```

Otherwise, the following status message is displayed:

```
The WPF plug-in for Firefox is not installed. Please install or reinstall the .NET Framework 3.5.
```

See also

- Detect Whether the .NET Framework 3.0 Is Installed
- Detect Whether the .NET Framework 3.5 Is Installed
- WPF XAML Browser Applications Overview