

The Window

When creating a WPF application, the first thing you will meet is the Window class. It serves as the root of a window and provides you with the standard border, title bar and maximize, minimize and close buttons. A WPF window is a combination of a XAML (.xaml) file, where the <Window> element is the root, and a CodeBehind (.cs) file. If you're using Visual Studio (Express) and you create a new WPF application, it will create a default window for you, which will look something like this:

<Window x:Class="WpfApplication1.Window1"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Window1" Height="300" Width="300">

<Grid>

</Grid>

</Window>

The *x:class* attribute tells the XAML file which class to use, in this case Window1, which Visual Studio has created for us as well. You will find it in the project tree in VS, as a child node of the XAML file. By default, it looks something like this:

using System;

using System.Windows;

using System.Windows.Controls;

*//…more using statements*

namespace WpfApplication1

{

/// <summary>

/// Interaction logic for Window1.xaml

/// </summary>

public partial class Window1 : Window

{

public Window1()

{

InitializeComponent();

}

}

}

the Window1 class is definied as partial, because it's being combined with your XAML file in runtime to give you the full window. This is actually what the call to InitializeComponent() does, which is why it's required to get a full functioning window up and running.

You will also notice that Visual Studio has created a Grid control for us inside the Window. The Grid is one of the WPF panels, and while it could be any panel or control, the Window can only have ONE child control, so a Panel, which in turn can contain multiple child controls, is usually a good choice

# Working with App.xaml

App.xaml is the declarative starting point of your application. Visual Studio will automatically create it for you when you start a new WPF application, including a Code-behind file called App.xaml.cs. They work much like for a Window, where the two files are partial classes, working together to allow you to work in both markup (XAML) and Code-behind.

App.xaml.cs extends the Application class, which is a central class in a WPF Windows application. .NET will go to this class for starting instructions and then start the desired Window or Page from there. This is also the place to subscribe to important application events, like application start, unhandled exceptions and so on.

**[WPF Tutorial](https://wpf-tutorial.com/)**

# WPF vs. WinForms

While they do serve the same purpose, there is a LOT of differences between them. If you have never worked with WinForms before, and especially if WPF is your very first GUI framework, you may skip this chapter, but if you're interested in the differences then read on.

The single most important difference between WinForms and WPF is the fact that while WinForms is simply a layer on top of the standard Windows controls (e.g. a TextBox), WPF is buil t from scratch and doesn't rely on standard Windows controls in almost all situations. This might seem like a subtle difference, but it really isn't, which you will definitely notice if you have ever worked with a framework that depends on Win32/WinAPI.

A great example of this is a button with an image and text on it. This is not a standard Windows control, so WinForms doesn't offer you this possibility out of the box. Instead you will have to draw the image yourself, implement your own button that supports images or use a 3rd party control. With WPF, a button can contain anything because it's essentially a border with content and various states (e.g. untouched, hovered, pressed). The WPF button is "look-less", as are most other WPF controls, which means that it can contain a range of other controls inside of it. You want a button with an image and some text? Just put an Image and a TextBlock control inside of the button and you're done! You simply don’t get this kind of flexibility out of the standard WinForms controls, which is why there's a big market for rather simple implementations of controls like buttons with images and so on.

The drawback to this flexibility is that sometimes you will have to work harder to achieve something that was very easy with WinForms, because it was created for just the scenario you need it for. At least that's how it feels in the beginning, where you find yourself creating templates to make a ListView with an image and some nicely aligned text, something that the WinForms ListViewItem does in a single line of code.

This was just one difference, but as you work with WPF, you will realize that it is in fact the underlying reason for many of the other differences - WPF is simply just doing things in its own way, for better and for worse. You're no longer constrained to doing things the Windows way, but to get this kind of flexibility, you pay with a little more work when you're really just looking to do things the Windows way.

The following is a completely subjective list of the key advantages for WPF and WinForms. It should give you a better idea of what you're going into.

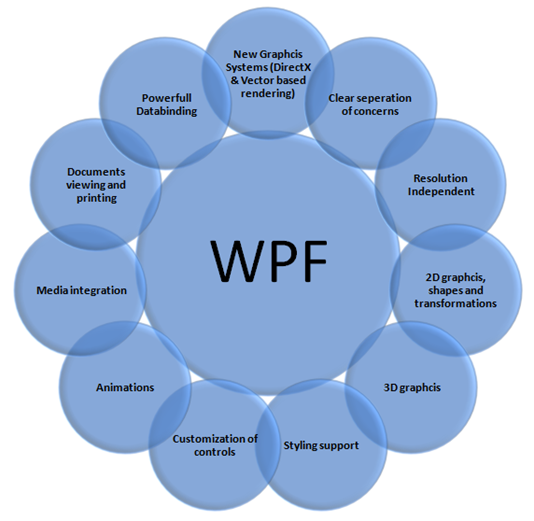
## WPF advantages

* It's newer and thereby more in tune with current standards
* Microsoft is using it for a lot of new applications, e.g. Visual Studio
* It's more flexible, so you can do more things without having to write or buy new controls
* When you do need to use 3rd party controls, the developers of these controls will likely be more focused on WPF because it's newer
* XAML makes it easy to create and edit your GUI, and allows the work to be split between a designer (XAML) and a programmer (C#, VB.NET etc.)
* Databinding, which allows you to get a more clean separation of data and layout
* Uses hardware acceleration for drawing the GUI, for better performance
* It allows you to make user interfaces for both Windows applications and web applications (Silverlight/XBAP)

## WinForms advantages

* It's older and thereby more tried and tested
* There are already a lot of 3rd party controls that you can buy or get for free
* The designer in Visual Studio is still, as of writing, better for WinForms than for WPF, where you will have to do more of the work yourself with WPF

Windows Presentation Framework is a next generation UI framework to create applications with a rich user experience. It is part of the .NET framework 3.0 and higher. It includes application UI, 2D graphics, 3D graphics and multimedia. It takes advantage of hardware acceleration of modern graphic cards. WPF makes the UI faster, scalable and resolution independent.



## Features of WPF

The list of WPF features is given below –

### Resolution Independence

WPF is resolution independence since all measures in WPF are logical units not pixels. A logical unit is a 1/96 of an inch. So, with changing the screen resolution setting in WPF each control will look same for each resolution. It is not based on Dots per inch (DPI) setting of the device.

### Separation of appearance and behaviours

WPF separates the appearance of an UI from its behaviour. The appearance is specified by XAML and behaviour is specified by a managed programming language like C# or VB.

### Built-In support for graphics and animation

WPF applications run within DirectX environment, hence it has major support of graphics and animation capabilities. WPF has a separate set of classes that are specifically deal with animation effects and graphics.

### Supports for Audio and Video

WPF has support for playing any audio or video file supported by Windows Media Player. It also gives you the tools to integrate video content into your rich UI such as placing a video window on a spinning 3-D cube.

### Highly customizable

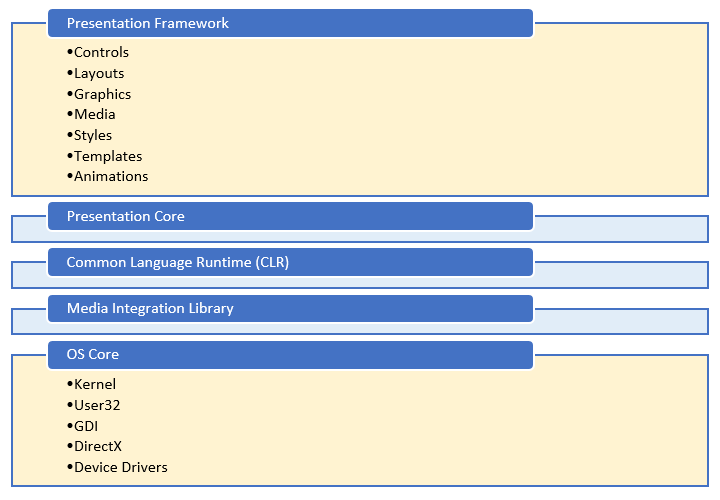
WPF supports separation of appearance and behaviours; hence you can easily change the look of a control or a set of controls. This concept of styling controls in WPF, is almost like CSS in HTML.

In WPF, you can store styles, controls, animations, and even any object as a resource and you may associate that resource to the controls. Each resource is declared once when the form loads itself.

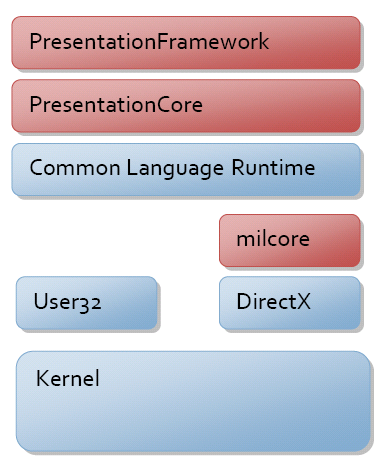
# The WPF Architecture

WPF uses a layered architecture that includes managed, unmanaged, and the core APIs in five different layers called **Presentation Framework**, **Presentation Core**, **Common Language Runtime**, **Media Integration Library**, and **OS Core**. The programming model is exposed through the managed code.

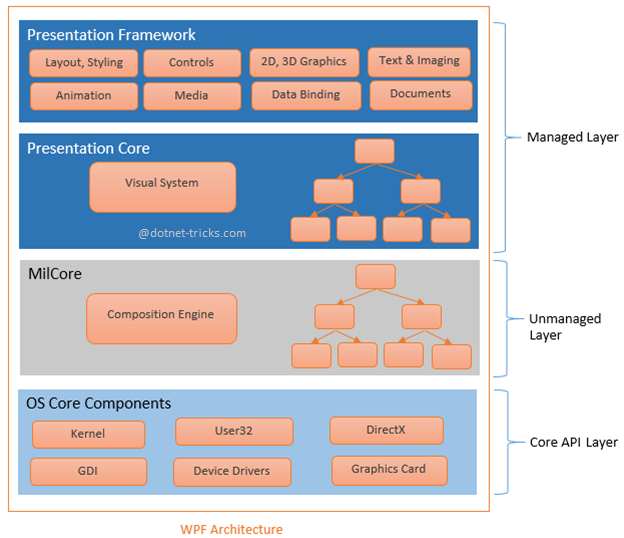
In the following diagram, you can see a clear picture of the architecture:



The major components of WPF are illustrated in the figure below. The red sections of the diagram (PresentationFramework, PresentationCore, and milcore) are the major code portions of WPF. Of these, only one is an unmanaged component – milcore. Milcore is written in unmanaged code in order to enable tight integration with DirectX. All display in WPF is done through the DirectX engine, allowing for efficient hardware and software rendering. WPF also required fine control over memory and execution. The composition engine in milcore is extremely performance sensitive, and required giving up many advantages of the CLR to gain performance.



Windows Presentation Framework is a next generation UI framework to create applications with a rich user experience. It is part of the .NET framework 3.0 and higher. WPF architecture is a layered architecture which have Managed, Unmanaged and Core API layers as shown in below fig.



### Managed Layer

Managed layer has two main components – Presentation Framework and Presentation Core.

* 1. Presentation Framework provides the required functionalities that we need to build the WPF applications such as controls, data bindings, styling, shapes, media, documents, annotations, animation and more. PresentationFamework.dll is responsible for this purpose.
  2. Presentation Core acts as a managed wrapper around MILCore and provides public interface for MIL. Presentation Core is the home for WPF Visual System and provides classes for creating application visual tree. The Visual System creates visual tree which contains applications Visual Elements and rendering instructions. PresentationCore.dll is responsible for this purpose.

### Unmanaged Layer

This layer is also called milcore or Media Integration Library Core. MilCore is written in unmanaged code in order to enable tight integration with DirectX. DirectX engine is underlying technology used in WPF to display all graphics, allowing for efficient hardware and software rendering. MIL has Composition System that receives rendering instructions from Visual System and translates into data that can be understood by DirectX to render user interface.

### Core API Layer

This layer has OS core components like Kernel, User32, GDI, Device Drivers, Graphic cards etc. These components are used by the application to access low level APIs. User32 manages memory and process separation.

**DirectX**, simply put, is software developed by Microsoft that talks to a PC's hardware components. Specifically, it's a collection of application programming interfaces, or APIs, designed to handle tasks related to rendering 2D and 3D vector graphics, rendering video and playing audio on the Windows platform.