

An Introduction to the Windows Presentation Foundation with the Model-View-ViewModel

Part 2

Paul Grenyer

In part 1 of An Introduction to the Windows Presentation Foundation with Model-View-ViewModel [Part1] I introduced the Canon application, the source code which can be downloaded from my website [SourceCode]. I used it to introduce you to simple WPF UI development and the Model-View-ViewModel [MVVM] pattern including simple binding and commands. Figure 1 shows the GUI for Cannon.

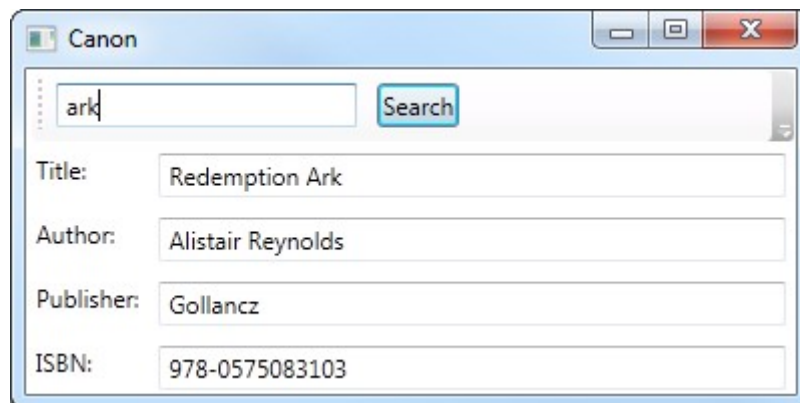


Figure 1: A Successful Search

Part 2 is focused around making the GUI look more aesthetically pleasing and introducing menus and tool bars and demonstrating system commands. I'll start off by introducing images.

Images

Currently the Canon application uses the standard icon in its title bar. It doesn't really make Canon stand out from any other Windows application. If you look on your (Windows 7 at least) task bar you'll see that all the open applications have an icon. If they all had the standard icon it would be difficult to tell them apart.

I use free icon libraries, like Silk Icon Set [SilkIcons], available on the internet for icons. I usually put images into an `Images` folder at the project level, so create one for the Canon project. Paste a suitable image (e.g. a 16x16 PNG) for the Canon icon into it and add the image to the project in the usual way. Make sure its Build Action property is set to `Resource`. Adding the image as an icon to the main window is done by setting the `Icon` attribute in the `Window` element:

```
<Window x:Class="Canon.View.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="{Binding AppTitle}"
  MinHeight="230"
  Height="230"
  MinWidth="450"
  Width="450"
```

```
FocusManager.FocusedElement="{Binding ElementName=searchBox}"
Icon="/Canon;component/images/lightbulb.png">
```

The format of the `Icon` attribute value is Microsoft Pack URI [PackURI] and consists of the following tokens:

`/Canon` The name of the resource file, including its path, relative to the root of the referenced assembly's project folder.

`;component` Specifies that the assembly being referred to is referenced from the local assembly.

`/images/lightbulb.png` The relative path to the image file.

Menus and Tool Bar Icons

As it stands the Canon application is not very useful as it only allows us to search for the two preloaded books. What it needs to be able to do next is save updates to those books and create new ones. Save actions are often invoked by a menu and/or tool bar button or via a keyboard shortcut. Next I'll show you how to add a menu, with menu items bound to commands, which share an icon with a tool bar button we'll add to a new tool bar. First add a menu to the top section of the dock panel:

```
<DockPanel>
  <Menu DockPanel.Dock="Top">

  </Menu>
  <ToolBarTray DockPanel.Dock="Top">

  </ToolBarTray>

</DockPanel>
```

Menus are declared in their parent component with the `Menu` element. In the case of a `DockPanel` they also inherit the `DockPanel.Dock` attribute which is set to `Top` to put it in the same place as the tool bar tray. The order of child elements is important. If you put the menu below the tool bar tray the menu will *appear* below the tool bar tray. Add a drop down menu by adding a `MenuItem` with the `Header` attribute set:

```
<Menu DockPanel.Dock="Top">
  <MenuItem Header="_File">

  </MenuItem>
</Menu>
```

The underscore in front of the `F` in `File` specifies that `F` is the short cut key for the File menu. To add an item to the drop down menu, add a child `MenuItem` element:

```
<Menu DockPanel.Dock="Top">
  <MenuItem Header="_File">
    <MenuItem Header="_Save" Command="{Binding RunSave}"/>
  </MenuItem>
</Menu>
```

The `Header` attribute specifies the name of the item and the command binding is the same as a button command binding. We also need to add the command to the view

model:

```
public class MainWindowViewModel : PropertyChangedBase
{
    ...
    public ICommand RunSave { get; private set; }
    ...

    public MainWindowViewModel(IBookRepository repo)
    {
        ...
        RunSave = new RelayCommand(o => Save(), o => canSave());
    }

    private bool canSave()
    {
        return true;
    }

    private void Save()
    {}
}
```

The `canSave` method just returns true for the time being. We'll put it to better use later. Menu items can also have images and the same image can be used for a tool bar button too. You could repeat the location of the image for both the menu item and the tool bar button, but a better solution is to add a resource:

```
<DockPanel>
    <DockPanel.Resources>
        <BitmapImage x:Key="SaveImage" UriSource="/Canon;component/images/disk.png" />
    </DockPanel.Resources>
</DockPanel>
```

This resource is added to the dock panel. Resources can be added to most components and are in scope within that component and all of its children. Before you add the `DockPanel.Resources` element, make sure you add a suitable image, called something like `disk.png`, to the images folder the name must match the name specified in `UriSource`. You can add all sorts of resources including the `BitmapImage` shown above. The `x:Key` attribute specifies the name that the resource will be referred to by when it's used by other components. The `UriSource` attribute is the path to the resource. It also uses Pack URI.

The `MenuItem.Icon` and `Image` child elements are required to add an image to a menu item:

```
<MenuItem Header="_Save" Command="{Binding RunSave}">
    <MenuItem.Icon>
        <Image Source="{StaticResource SaveImage}"/>
    </MenuItem.Icon>
</MenuItem>
```

The image to use is specified by the `Source` attribute of the `Image` element which maps to the `x:Key` attribute of the resources. The image is bound to the resource, so uses curly braces. The resource is static as it is known at compile time, so uses the `StaticResource` keyword followed by the name of the resource. If you run the application now you will see the image next to the new menu item. The same image can

be used as a tool bar icon. Add a new tool bar under the existing one. Add a button with a Command binding to the tool bar and an Image element that binds to the save image.

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
    <StackPanel Orientation="Horizontal">
      ...
    </StackPanel>
  </ToolBar>
  <ToolBar>
    <Button Command="{Binding RunSave}">
      <Image Source="{StaticResource SaveImage}" />
    </Button>
  </ToolBar>
</ToolBarTray>
```

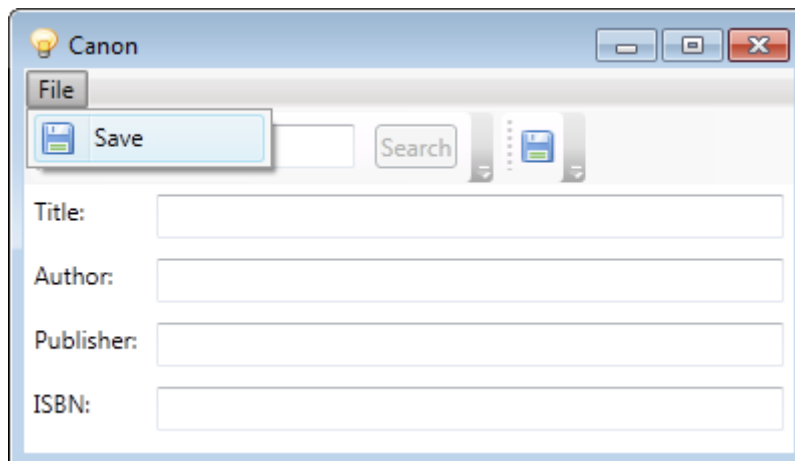


Figure 2: Menus and Icon Toolbar

The save menu item and button do not currently save. The simplest way to save a book is to create a new `Book` instance, initialise it from the UI fields and pass it to the `Save` method of the repository:

```
private void Save()
{
    repo.Save(new Book{Title = Title, Author = Author, Publisher = Publisher, ISBN = ISBN});
}
```

Can you spot the flaw? The `Id` is not set, which means every time you save a new book instance will be created, even if it has exactly the same field values as an existing one. To get around this, we need to keep a reference to the loaded book:

```
public class MainWindowViewModel : PropertyChangedBase
{
    ...
    private Book currentBook;
    ...

    public MainWindowViewModel(IBookRepository repo)
    {
        ...
        currentBook = new Book();
        ...
    }
}
```

```

private void Search()
{
    var book = repo.Search(SearchText);
    if (book != null)
    {
        currentBook = book;

        Title = book.Title;
        Author = book.Author;
        Publisher = book.Publisher;
        ISBN = book.ISBN;

        OnPropertyChanged("Title");
        OnPropertyChanged("Author");
        OnPropertyChanged("Publisher");
        OnPropertyChanged("ISBN");
    }
}

...
private void Save()
{
    currentBook = repo.Save(new Book
    {
        Id = currentBook.Id,
        Title = Title,
        Author = Author,
        Publisher = Publisher,
        ISBN = ISBN
    });
}
}

```

To hold the reference we add a book field called `currentBook` to the `MainWindowViewModel`. We default initialise it in the constructor to make sure it is valid even if a book has not been loaded yet. Then if we find a book when we search for one we set the `currentBook` reference to the new book. Finally when we save the new book we use the `Id` from `currentBook` to create a new book instance. After a successful save we set `currentBook` to the new book instance. Try it out and see if you can spot the further flaw.

The only way to create a new book is to enter values into all the fields and save before searching for a book and even then you can only do it once. What we need is a new book menu item, image and tool bar button:

```

<DockPanel>
  <DockPanel.Resources>
    <BitmapImage x:Key="SaveImage" UriSource="/Canon;component/images/disk.png" />
    <BitmapImage x:Key="NewImage" UriSource="/Canon;component/images/add.png" />
  </DockPanel.Resources>
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">
      <MenuItem Header="_New" Command="{Binding RunNew}">
        <MenuItem.Icon>
          <Image Source="{StaticResource NewImage}" />
        </MenuItem.Icon>
      </MenuItem>
      <MenuItem Header="_Save" Command="{Binding RunSave}">
        <MenuItem.Icon>

```

```

        <Image Source="{StaticResource SaveImage}"/>
    </MenuItem.Icon>
</MenuItem>
</MenuItem>
</Menu>
<ToolBarTray DockPanel.Dock="Top">
    ...
<ToolBar>
    <Button Command="{Binding RunNew}">
        <Image Source="{StaticResource NewImage}" />
    </Button>
    <Button Command="{Binding RunSave}">
        <Image Source="{StaticResource SaveImage}" />
    </Button>
</ToolBar>
</ToolBarTray>

```

and a new Command like `RunSave` and `RunSearch`. The difference with `RunNew` is that it does not need a `canNew` method as it is always permitted to create a new book:

```
RunNew = new RelayCommand(o => New());
```

You could create a `canNew` method hard coded to return `true` for consistency if you wanted too. The implementation of `New` looks like this:

```

private void New()
{
    Update(new Book());
}

private void Update(Book book)
{
    currentBook = book;

    Title = book.Title;
    Author = book.Author;
    Publisher = book.Publisher;
    ISBN = book.ISBN;

    OnPropertyChanged("Title");
    OnPropertyChanged("Author");
    OnPropertyChanged("Publisher");
    OnPropertyChanged("ISBN");
}

```

The `Update` method is duplication of the code in the `Search` method, so the `Search` method can be refactored to remove the duplication:

```

private void Search()
{
    var book = repo.Search(SearchText);
    if (book != null)
    {
        Update(book);
    }
}

```

If you run the application now you can create, save and search for new books.

System Commands

WPF supports a range of system commands for operations including cutting, copying and pasting. This means you can add standard functionality without having to implement the details. For example you can add an edit menu:

```
<MenuItem Header="_Edit">
    <MenuItem Header="Cut" Command="ApplicationCommands.Cut" />
    <MenuItem Header="Copy" Command="ApplicationCommands.Copy" />
    <MenuItem Header="Paste" Command="ApplicationCommands.Paste" />
</MenuItem>
```

You can of course add images and a corresponding tool bar in the way already described. Here we've replaced the command bindings with the system commands for cut, copy and paste. If you run the application you'll find cut, copy and paste just work as expected. WPF In Action [WPFInAction], the book in Introduced in part 1, goes into the system commands in more detail¹.

Not all system commands are as straight forward. Unfortunately if you add the system `Close` command to the file menu:

```
<MenuItem Header="Close" Command="ApplicationCommands.Close" />
```

it is not enabled and does not close the application. What is missing is a command binding and handler methods:

```
<Window>
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Close" Executed="CloseCommandHandler"/>
    </Window.CommandBindings>
    ...
</Window>
```

The `CommandBinding` element uses its `Command` and `Executed` attributes to map the `Close` system command to the `CloseCommandHandler` handler, which is defined in the `MainWindow` class:

```
private void CloseCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
    Close();
}
```

Clearly `CloseCommandHandler` just calls the `Close` method on the window to close it and consequently the application.

Detecting Changes

Do you remember that earlier on we implemented a not particularly helpful binding for the Canon window title? Do you also remember the `canSave` method that always returns `true`? It would be far more useful for the user to only be able to save when there were changes to be saved and for the window title to indicate when there are changes to be saved:

```
public string AppTitle
```

¹ See chapter 10, Commands

```

{
    get
    {
        return string.Format("Canon{0}", IsDirty ? " *" : "");
    }
}
...
private bool canSave()
{
    return IsDirty;
}

```

`IsDirty` is a boolean property that indicates if any changes have been made. In the case of `AppTitle` it is used to determine whether an asterisk should be appended to the title when there are changes and in the case `canSave` it is just returned to indicate if the command should be enabled.

```

public bool IsDirty
{
    get
    {
        return !currentBook.Title.Equals(Title) ||
               !currentBook.Author.Equals(Author) ||
               !currentBook.Publisher.Equals(Publisher) ||
               !currentBook.ISBN.Equals(ISBN);
    }
}

```

The `IsDirty` property compares the current book fields against the equivalent UI fields to determine if there are any changes. Unfortunately this leads to some more verbose changes to the UI field properties to get the title and save command to update in real time:

```

private string title = string.Empty;
public string Title
{
    get
    {
        return title;
    }
    set
    {
        title = value;
        OnPropertyChanged("Title");
        OnChange();
    }
}
...
private void OnChange()
{
    OnPropertyChanged("AppTitle");
}

```

I have only shown the changes for the `Title` property, but the `Author`, `Publisher` and `ISBN` properties must be changed in the same way. Instead of using the default property implementation we have to implement our own `set` method so that when the property is updated we can tell WPF to also update the window title. This means we also need to *separately* store the property value, which is initialised to an empty string to match the default `Book` instance, and implement a `get` method too. One advantage is that we can also move the WPF notification that the property has changed to the property itself so that

we don't need to remember to call `OnPropertyChanged` anywhere else in the code where we assign the property. So the `Update` method is reduced to:

```
private void Update(Book book)
{
    currentBook = book;

    Title = book.Title;
    Author = book.Author;
    Publisher = book.Publisher;
    ISBN = book.ISBN;
}
```

The window title also needs to be updated when a book is saved as there are no longer any changes:

```
private void Save()
{
    currentBook = repo.Save(new Book
    {
        Id = currentBook.Id,
        Title = Title,
        Author = Author,
        Publisher = Publisher,
        ISBN = ISBN
    });
    OnChange();
}
```

Finally

This is where this second article leaves the Canon application. There is more to do, but that falls outside the scope of an introductory article. In part 1 I already introduced you to simple WPF UI development and the Model-View-ViewModel pattern including simple binding and commands. In this part I explained how to make WPF GUIs more aesthetically pleasing with the use of images and more user friendly with the use of menus and toolbars and showed how to implement those menus and toolbars with custom and system commands.

In future articles I will cover unit testing and patterns for maintaining the separation between the view model and the view when you want to display message boxes and child windows or use custom controls.

References

[Part1] In Introduction to the Windows Presentation Foundation with Model-View-ViewModel – Part 1: http://paulgrenyer.net/Introduction_to_WPF_with_MVVM_-_Part_1.pdf

[SourceCode] Canon 0.0.1 Source Code: <http://paulgrenyer.net/dnld/Canon-0.0.1.zip>

[MVVM] WPF Apps With The Model-View-ViewModel Design Pattern by Josh Smith. MSDN Magazine: <http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>

[SilkIcons] Silk Icon Set from Mark James: <http://www.famfamfam.com/lab/icons/silk/>

[PackURI] Pack URIs in WPF: <http://msdn.microsoft.com/en-us/library/aa970069.aspx>

[WPFIInAction] WPF In Action with Visual Studio 2008 by Arlen Feldman and Maxx Daymon. Manning. ISBN: 978-1933988221