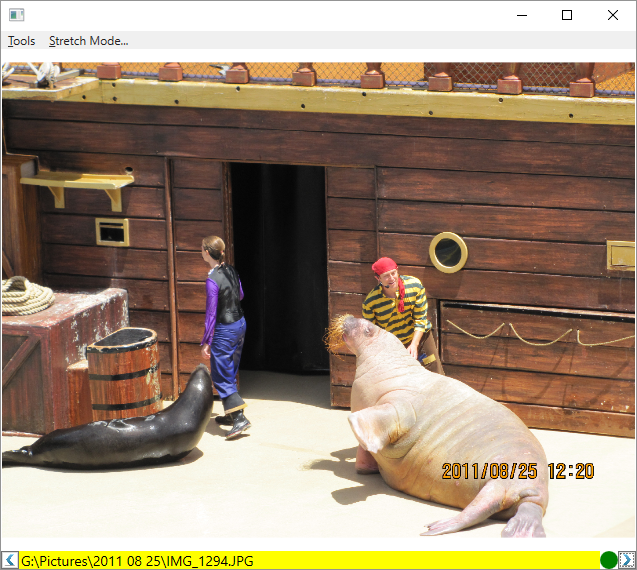
Rotating picture display

Your pictures that reside on your computer disk and network disks

# Introduction

The purpose of this solution is to retrieve files in directories and subdirectories that you provide and display them in a random order.

## Description of the interface



The display is composed of

* Menu
* Picture
* Bottom bar appearing mostly in yellow background, contains both information and allows actions on the picture set

The Menu items allow you to change all the factors provided in the configuration file.

The picture is “clickable”, mouse selection of the picture toggles the rotate / hold-still mode of picture rotation. The rotation mode is accompanied with an indicator represented as a traffic-light circle kind of indicator, in the bottom bar. Green for rotating automatically mode, Red for holding-still.

The bottom bar allows you to move back and forth through the pictures in the order they were displayed, using the left arrow and the right arrow at each end of the bottom bar. The bottom bar also contains the full path of the picture displayed.

# Installation

Modify the App.config file by modifying the following, value, entries in the appSettings section:

<appSettings>

<!-- Directories where the system will look for pictures to display.

 Directories are semicolon separated.  Ex: value="c:\pic;g:\pic;m:\pic" -->

<add key="Initial Folders" value="G:\Pictures"/>

<!-- Depth of stack keeping the displayed pictures, meaning you may go

 back up to "Max picture tracker depth" of displayed pictures -->

<add key="Max picture tracker depth" value="1000"/>

<!-- These are the only extensions that the system will consider.

 Extensions are semicolon separated.

Extensions must start with a period (".").  -->

<add key="Still pictures" value=".jpg;.bmp;.gif;.png;.psd;.tif"/>

<add key="Motion pictures" value=".mov;.avi;.mpg;.mp4;.wmv;.3gp"/>

<!-- Image stretch may be:

 "Fill" - Stretch the picture height and width independently

 "None" - Original size of height and width is maintained

 "Uniform" Stretches the height and width uniformly until the one of the

directions equals the height or the width of the window

 "UniformToFill" Stretches the height and width uniformly passed the

the first dimension to reach the window height or width

and until the second dimension reaches the height or

width of the window -->

<add key="Image stretch" value="Uniform"/>

<!-- Time to wait between display of one picture to the next

 Value may contain fraction of a second.  -->

<add key="Timespan between pictures [Seconds]" value="10"/>

<!-- The first picture to be displayed is treated differently then the rest

 of the pictures.  However, if you leave the value of "First picture to

display" blank then the system will treat the first picture like it

 treats the rest of the picture, choose it randomly.  Though not having

 first picture may mean that your wait for "Timespan between pictures

 [Seconds]" before the first picture appears. -->

<add key="First picture to display" value="G:\Pictures\Ben\IMG\_0840-1.JPG"/>

<!-- The system will start with automatic rotation of picture if the value

 to "On start image rotating" is true otherwise the first picture will

 be frozen until you change the running status by selecting (click) on   the picture or select the forward arrow -->

<add key="On start image rotating" value="True"/>

</appSettings>

Also modify the value entry of file type="log4net.Util.PatternString", part of the appender entry in the log4net config section:

<log4net>

...

<appender name="AppRollingFile" type="log4net.Appender.RollingFileAppender"> <!-- Set the value to the directory where you care for messages to go into.

The %date{yyyyMMdd} translates to the date value, 4 digit year,

2 digit month and 2 digit day. Date value when the program started

executing -->

<file type="log4net.Util.PatternString"

value="G:\Logs\RotatePictures\RotatePictures.%date{yyyyMMdd}.log"/>

# The code

The solution is a relatively simple MVVM project, resulting in an executable that is about 100KB.

## Model

The model, represented by the PictureModel class, is a file name repository. PictureModel relies on the PictureCollection class that stores all the file names in the directories provided in the "Initial Folders" of the configuration file, filtered by the allowable extensions.

The model also relies on the SelectionTracker class that stores the file names that were displayed and allow for back and forth motion of pictures using the arrows in the bottom bar.

The ViewModel, MainWindowViewModel, holds a reference to the model, though it does not instantiate the class. which is left to a factory:

\_model = (PictureModel)ModelFactory.Inst.Create("PictureFileRepository");

The model, PictureModel, loads the pictures asynchronously:

\_taskModel = Task.Run(() => RetrievePictures(), \_cts.Token);

Thereafter the pictures are selected in a random fashion:

var cnt = \_picCollection.Count;

var index = \_rand.Next(cnt);

return \_picCollection[index];

## ViewModel

The ViewModel, for example MainWindowViewModel drives the View, MainWindow. The ViewModel section includes other window controls, that are used to display and alter information provided in the .config file.

### Communication between the ViewModels

The purpose of the MVVM pattern is to separate concerns between the disparate logical components, so the View is responsible for the display of information only which is separate from the model that is responsible for retrieving and saving data which is separate from the “glue” between them called the ViewModel.

However, we need to communicate between the various ViewModel components and we should not let one ViewModel component hold an instance of another ViewModel component. The communication between the ViewModel components is done through a Messenger, creatively named Messenger.

The recipient of a message needs to register and potentially unregister the message. Between the time that the message is registered and before it is unregistered, any class can send the registered message to be handled by the class that registered to handle the message.

The ViewModel FileTypesToRotateViewModel registers the message SelectedMetadataMessage and processes requests as follows:

**Step 1**

Register the message:

Messenger.DefaultMessenger.Register<SelectedMetadataMessage>(

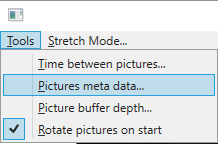
this, OnMetaDataProcess, MessageContext.SelectedMetadataViewModel);

The third (3rd) parameter, MessageContext.SelectedMetadataViewModel, is a string used as context for the message. Context, in this scenario, is a differentiating mechanism for messages.

This registering class, FileTypesToRotateViewModel, is not instantiated until the Main Window, MainWindowViewModel, asks for it be instantiated, and as such it cannot receive nor handle messages until it is instantiated.

**Step 2**

Upon selecting the menu item Tools / Pictures meta data…



The command processing for this menu item, part of the MainWindowViewModel, calls:

\_pictureMetadataService.ShowDetailDialog(metaData);

Where the parameter, metaData, is the data needed to populate the View, FileTypesToRotateView and \_pictureMetadataService is an instance of the service, FileTypeToRotateService, responsible for displaying the FileTypesToRotateView window, as a dialog. The metaData parameter will be passed on in the SelectedMetadataMessage message, after the FileTypesToRotateView window was created by the service, FileTypeToRotateService. The metaData will be sent via the following message:

Messenger.DefaultMessenger.Send(

new SelectedMetadataMessage(metadata),

MessageContext.SelectedMetadataViewModel);

**Step 3**

FileTypesToRotateViewModel’s callback method, OnMetaDataProcess (see **Step 1** registration), processes the message, in our case the callback displays the information within the window, FileTypesToRotateView.

**On the way back**

When the user selects the OK button, in the FileTypesToRotateView class, the above three (3) steps are used again to update the result of the window. The main window’s ViewModel, MainWindowViewModel, register this same SelectedMetadataMessage but a different context.

**Step 1**

Messenger.DefaultMessenger.Register<SelectedMetadataMessage>(

this, OnSetMetadataAction, MessageContext.SetMetadata);

**Step 2**

The handling of the OK button in the FileTypesToRotateViewModel sends:

Messenger.DefaultMessenger.Send(

new SelectedMetadataMessage(metadata),

MessageContext.SetMetadata);

Then still part of handling the OK button, FileTypesToRotateViewModel unregisters from the SelectedMetadataMessage message:

Messenger.DefaultMessenger.Unregister(

this,

MessageContext.SelectedMetadataViewModel);

The Unregister is necessary, otherwise every time the window FileTypesToRotateView is instantiated a new call to the register will happen which will duplicate and triplicate the handling.

**Step 3**

MainWindowViewModel handles the updates from the FileTypesToRotateViewModel.

## View

### Satisfying the MVVM equation

The views are xaml defined controls. Each View control instantiates the appropriate ViewModel class and assigns its DataContext to it. However, the View does not instantiate the ViewModel directly, it calls a factory class.

DataContext = VmFactory.Inst.Create(this);

The VmFactory class is a simple class using reflection to instantiate the ViewModel class.

### Handling Errors

The FileTypesToRotateView needs to handle errors. The XAML file adds the following template

<Grid.Resources>

<Style TargetType="{x:Type TextBox}">

<Setter Property="Validation.ErrorTemplate">

<Setter.Value>

<ControlTemplate>

<StackPanel>

<AdornedElementPlaceholder>

<Border BorderBrush="Red" BorderThickness="2"/>

</AdornedElementPlaceholder>

<ItemsControl ItemsSource="{Binding}">

<ItemsControl.ItemTemplate>

<DataTemplate>

<TextBlock x:Name="ErrorText"

Text="{Binding ErrorContent}"

Foreground="Red"/>

</DataTemplate>

</ItemsControl.ItemTemplate>

</ItemsControl>

</StackPanel>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</Grid.Resources>

Which paints the border of the TextBox containing the error red and displays the error message under the TextBox.

The TextBox that is needs to opt into the error program needs to mark its binding with ValidatesOnDataErrors=True, and potentially ValidatesOnExceptions=True, as follows:

<TextBox x:Name="InitialFolder" Grid.Row="1" Grid.Column="1"

Grid.ColumnSpan="2" Margin="0,3,2,16"

 Text="{Binding PictureFolders,

Mode=TwoWay, UpdateSourceTrigger=PropertyChanged,

ValidatesOnDataErrors=True, ValidatesOnExceptions=True}"

 ToolTip=

"You may specify more than one full-pathed folder.  Folders are semicolon separated"

ToolTipService.ShowDuration="30000" />

Then the VM, FileTypesToRotateViewModel, in addition to implementing the INotifyPropertyChanged it needs to implement IDataErrorInfo.

I accomplished it through a class, ViewModelBase, that FileTypesToRotateViewModel inherits.

The ViewModelBase implements both INotifyPropertyChanged and IDataErrorInfo. We will concentrate on the IDataErrorInfo. The IDataErrorInfo calls for two (2) properties:

public string this[string propertyName] { get { ... } }

public string Error { get { ... } }

They way the ViewModelBase class implements it is separating the base class from the rules. The base class defines an ErrBinder class and a dictionary of (property name, ErrBinder instance), \_errBinderMap:

private readonly Dictionary<string, ErrBinder> \_errBinderMap = new Dictionary<string, ErrBinder>();

Then a rule agnostic AddRule method:

protected void AddRule(string propertyName, Func<bool> ruleDelegate, string errorMessage)

{

var rv = new ErrBinder(ruleDelegate, errorMessage);

\_errBinderMap.Add(propertyName, rv);

}

That is used by the VM class, FileTypesToRotateViewModel. This VM class knows what the rules should be and as such defines them.

The ErrBinder class’s main function is to allow an error evaluation upon request through the method: ErrEvaluate().

private class ErrBinder

{

private readonly Func<bool> \_ruleValidate;

private readonly string \_message;

internal ErrBinder(Func<bool> ruleValidate, string message)

{

\_ruleValidate = ruleValidate;

\_message = message;

}

internal string Error { get; set; }

internal bool HasError { get; set; }

internal void ErrEvaluate()

{

Error = null;

HasError = false;

try

{

var rc = \_ruleValidate();

HasError = !rc;

if (rc) return;

Error = \_message;

HasError = true;

}

catch (Exception e)

{

Error = e.Message;

HasError = true;

}

}

}

Now filling in the implementation of IDataErrorInfo is easy:

public string this[string propertyName]

{

get

{

if (!\_errBinderMap.ContainsKey(propertyName)) return null;

\_errBinderMap[propertyName].ErrEvaluate();

return \_errBinderMap[propertyName].Error;

}

}

public string Error

{

get

{

var errors = \_errBinderMap.Values.Where(b => b.HasError).Select(b => b.Error);

return string.Join(Environment.NewLine, errors);

}

}

In addition we need a HasErrors property, this method will allow us to set the CanExecute method on the OK button of the form.

protected bool HasErrors

{

get

{

var values = \_errBinderMap.Values.ToList();

values.ForEach(b => b.ErrEvaluate());

return values.Any(b => b.HasError);

}

}

# Points of interest

There are two types of extensions that we filter on: the still pictures extensions and the motion pictures extensions, therefore we need to filter all files based on the union of both sets of extensions. Linq provides a Union() method. This Linq Union() needs to compares objects for equality, so in our extensions case it is sting comparison which by default is case sensitive. We are looking for case insensitive comparison, “.mov” is the same as “.MOV”.

Linq provides a Union() overload method with the following signature:

public static IEnumerable<TSource> [Union](http://referencesource.microsoft.com/#System.Core/System/Linq/Enumerable.cs,d54fdb15d52500f1)<TSource>(this IEnumerable<TSource> first,

IEnumerable<TSource> second, IEqualityComparer<TSource> comparer)

See the site <http://sourceof.net>, where Microsoft publishes the source code to .Net, then search for Enumerable.Union, searching for Union only yields too many hits.

At this point we can provide a specific solution for our problem at hand but what we really would like is a generic solution where we could write the Union() method as follows:

\_stillExt.Union(\_motionExt,

(x, y) => string.Compare(x, y, StringComparison.OrdinalIgnoreCase) == 0);

This is possible by defining our own LinqExtensions static class

public static class LinqExtensions

{

public static IEnumerable<TSource> Union<TSource>(

this IEnumerable<TSource> first,

IEnumerable<T> second,

Func<TSource, TSource, bool> cmpr)

{

// This method calls Linq’s Union

return first.Union(second, new LinqComparer<TSource>(...));

}

}

LinqComparer will be a class that inherits from IEqualityComparer<in T> (see <http://sourceof.net>). The IEqualityComparer<in T> interface demands implementation of:

bool Equals(T x, T y);

int GetHashCode(T obj);

So, for example:

public class LinqComparer<TSource> : IEqualityComparer<TSource>

{

private readonly Func<TSource, TSource, bool> \_linqCmp;

public LinqComparer(Func<TSource, TSource, bool> cmp)

{

\_linqCmp = cmp ?? throw new ArgumentException(

@"comparison function may not be null ", nameof(cmp));

}

public bool Equals(TSource x, TSource y) => \_linqCmp(x, y);

public int GetHashCode(TSource x) => 0;

}

This last GetHashCode(TSource x) => 0 is an issue for a large set, a return of 0 for every GetHashCode() means that a full comparison will be done on every comparison, as opposed to leverage integer comparison via the HashCode.

As such we will include a different approach. LinqComparer will be as follows:

public class LinqComparer<TSource> : IEqualityComparer<TSource>

{

private readonly Func<TSource, TSource, bool> \_linqCmp;

private readonly Func<TSource, int> \_hashCode;

public LinqComparer(Func<TSource, TSource, bool> cmp,

Func<TSource, int> hashCode = null)

{

\_linqCmp = cmp ?? throw new ArgumentException(

@"comparison function may not be null", nameof(cmp));

\_hashCode = hashCode ?? (T => 0);

}

public bool Equals(TSource x, TSource y) => \_linqCmp(x, y);

public int GetHashCode(TSource x) => \_hashCode(x);

}

And the LinqExtensions class will be:

public static class LinqExtensions

{

public static IEnumerable<T> Union<T>(

this IEnumerable<T> a,

IEnumerable<T> b,

Func<T, T, bool> cmpr,

Func<T, int> hashCode = null) =>

a.Union(b, new LinqComparer<T>(cmpr, hashCode));

}

Now calling the Union method is formed as follows:

\_stillExt.Union(\_motionExt,

(x, y) => string.Compare(x, y, StringComparison.OrdinalIgnoreCase) == 0),

x => x.ToLower().GetHashCode());

And LinqComparer is as follows:

public class LinqComparer<TSource> : IEqualityComparer<TSource>

{

private readonly Func<TSource, TSource, bool> \_linqCmp;

private readonly Func<TSource, int> \_hashCode;

public LinqComparer(Func<TSource, TSource, bool> cmp,

Func<TSource, int> hashCode = null)

{

\_linqCmp = cmp ?? throw new ArgumentException(

@"comparison function may not be null", nameof(cmp));

\_hashCode = hashCode ?? (T => 0);

}

public bool Equals(TSource x, TSource y) => \_linqCmp(x, y);

public int GetHashCode(TSource x) => \_hashCode(x);

}

Achieving our goal of using a lambda expression for the comparison of the Union() method.

# Conclusion

We have achieved separation of concerns and a working picture rotation application.