User interface design

in C#, with Windows Presentation Foundation (WPF)

# Lab 2 – Attaching interfaces to data – the small-project way

## Functional Goal

Connect elements in the user interface to data from TheTVDB.com to provide basic application functionality. Save program state across sessions.

## Learning Goals

* Begin to understand C# Data Binding
* Understand basic usage of Isolated Storage

## Prerequisites

* You’ll need to install Visual Studio (2012 Ultimate was used to create this lab) from the MSDN/DreamSpark service on ANGEL’s RosePortal.
* Code from Lab1 (a complete version may be available from your instructor)
* A basic understanding of serialization

## Submission Instructions

Submit answers to the **4** questions in this lab as a .pdf to the appropriate Moodle submission form.

## Get started: Connecting to TheTVDB

In Lab 1, you built a basic functional interface and used sample data to get an idea of what that interface looks like; but, Lab 1 was only half of the story. C# controls like the ListView have a lot of intelligence built in when it comes to displaying data – but, that intelligence isn’t much use without some real data. So, for this and subsequent labs, we’ll be using the API of TheTVDB.org, an open-source TV series database.

If you copy and paste code from another source, or type code from an example, and Visual Studio doesn’t have any using suggestions, you probably need to **Add a Reference.**

Protip

Working with other projects in C# is done through **References.** References can be to optional components of the .NET Framework, other projects in your solution, or **dynamic-link libraries (DLLs).** So to get started, let’s acquire and add to our project the TVDBlib DLL.

### Reference the TVDB API for C#

1. Go to <http://code.google.com/p/tvdblib/downloads/list> and download the most recent version of the library (the rar, not the exe).
2. In the downloaded file, located Library\bin\TvdbLib.dll and extract it to your solution, in a new folder called ”References”.
3. Open your solution in Visual Studio.
4. Right-click on the MyTVCompanion **project** (not the solution) and choose Add References. In the tree on the left, select Browse, then click Browse… near the bottom to locate the extracted file.
5. Click OK.

### Connect to the TVDB API

1. Create a user account at <http://www.thetvdb.com>. Once you’ve logged in, click account on the navigation bar.
2. Register for an API key at <http://thetvdb.com/?tab=apiregister> and make a note of it. If you forget it, visit <http://thetvdb.com/?tab=userinfo>.
3. In App.xaml:

Before the App() constructor, type prop and press Tab to let Visual Studio build you a template. These are called Snippets, and they’re quite helpful.

Protip

* 1. Create a new property of type TvdbHandler called TvdbHandler.
  2. In the constructor, assign a new instance of TvdbHandler using the API-key only constructor.

TvdbHandler = new TvdbHandler("49FF3082EF06CF50");



Question 1: What Snippet would you use to insert a new constructor? To write a line to the Console? (Hint: You can insert snippets from the context menu – right-click inside a method or class and choose Insert Snippet…) (6 points)

## Serialize and store: Prepare and persist a data structure for TV series storage

To serve as an effective TV companion, your app will need to remember the user’s watched shows. It will also need a data structure to share that list of shows between pages. So, let’s get that set up. You’re going to use IsolatedStorage to store the user’s shows between application runs, so you won’t have to handle file locations or permissions.

### Creating an ObservableCollection of shows

1. In App.xaml.cs (you may need to expand App.xaml in the Solution Explorer to see this file), create the following fields and properties.

private readonly IsolatedStorageFile \_isolatedStorage;

private const String ShowsFileName = "mydata.bin";

public ObservableCollection<TvdbSeries> Shows { get; private set; }

1. In the constructor, add the following code:

\_isolatedStorage = IsolatedStorageFile.GetUserStoreForAssembly();

if (\_isolatedStorage.FileExists(ShowsFileName))

{

using (var stream = \_isolatedStorage.OpenFile(ShowsFileName, FileMode.Open))

{

var deserializer = new BinaryFormatter();

Shows = (ObservableCollection<TvdbSeries>)deserializer.Deserialize(stream);

}

}

else { Shows = new ObservableCollection<TvdbSeries>(); }

Question 2: What does the code from step 2 do? If necessary, use MSDN (<http://msdn.microsoft.com>) to look up classes and **methods** you don’t understand. (9 points)

1. Create a new method using the following code:

private void AppExit(object sender, ExitEventArgs e)

{

var isolatedStorage = IsolatedStorageFile.GetUserStoreForAssembly();

new BinaryFormatter().Serialize(isolatedStorage.CreateFile(ShowsFileName), Shows);

}

1. In App.xaml, use the Properties panel to bind the Application’s Exit event to the method you just created.

If you’re using an external resource (e.g. a file, a network stream, etc.) in C#, you should always use a using block around your code. This ensures that the resource is properly disposed.

Protip

## Data binding: Search for and save shows in the Settings window

MSDN’s article about data binding (a worthwhile read, found here: <http://msdn.microsoft.com/en-us/library/ms750612.aspx>) says it best: “data binding provides a simple and consistent way for applications to present and interact with data.” To get started with data binding, you’ll attach data to the controls you created in SettingsWindow.xaml.

### Creating a Binding from the codebehind

1. In SettingsWindow.xaml, give the first ListView an x:Name: <ListView x:Name="SearchResults">
2. In the Window definition in SettingsWindow.xaml, underneath the two xmlns lines, add a DataContext: DataContext="{Binding RelativeSource={RelativeSource Self}}"
3. To make accessing the variables from App.xaml easier, create two new readonly properties:

public ObservableCollection<TvdbSeries> Shows { get { return ((App)Application.Current).Shows; } }

private TvdbHandler TvdbHandler { get { return ((App)Application.Current).TvdbHandler; } }

1. Add a new handler to the Search button’s Click event. In the codebehind, use the following code for the handler (your handler may have a different name):

private void SearchButtonClick(object sender, RoutedEventArgs e)

{

SearchResults.ItemsSource = TvdbHandler.SearchSeries(SearchBox.Text);  
}

1. Hover over the SearchSeries method and look at the return type – note that it’s a List. Because List implements IEnumerable, we can bind the ListView’s ItemsSource directly to the results of SearchSeries.

Now, run the program, go to the Settings page, type a show name (e.g. Big Bang Theory), and click search. Notice that nothing happened? That’s ok – we just told WPF what data to use, but we haven’t yet told it how to use it!

1. Replace the ListView.Items tag (and all the Items it contains) with this DataTemplate:

<ListView.ItemTemplate>

<DataTemplate>

<TextBlock Text="{Binding Path=SeriesName}" />

</DataTemplate>  
</ListView.ItemTemplate>

Rerun the program and repeat your search from before. Assuming you’re online, in a few seconds you should see a list of search results appear, because the binding you created told WPF which property of the SearchSeries data to use. Because SearchSeries returns a List<TvdbSearchResult>, you can bind to any property of a TvdbSearchResult. The Path field can contain just a property (as you’ve used it here), or a “dotted-down” path of properties if you need to dig deeper for the information you want.

Question 3: Why do we use an ObservableCollection instead of a List for the user’s watched shows? What interface does ObservableCollection implement but List does not that necessitates this decision? (6 points)

### Brief aside: add Shows to the collection

1. Add a new handler to the Add button’s Click event. In the codebehind, use the following code for the handler (your handler may have a different name).

private void AddButtonClick(object sender, RoutedEventArgs e)

{

if (SearchResults.SelectedIndex == -1) return;

var selected = SearchResults.SelectedItem as TvdbSearchResult;

Shows.Add(TvdbHandler.GetSeries(selected.Id, TvdbLanguage.DefaultLanguage, true, false, false));

}

### Creating a Binding from XAML

1. Add a new ItemsSource to the second ListView:
   1. <ListView x:Name="ShowsListView" ItemsSource="{Binding Shows}">

<ListView.ItemTemplate>

<DataTemplate>

<TextBlock Text="{Binding Path=SeriesName}" />

</DataTemplate>  
</ListView.ItemTemplate>

1. As before, replace the ListView.Items with a DataTemplate:

Notice that this DataTemplate’s binding doesn’t have a blue error underline. When your DataContext and ItemsSource are properly set up, Visual Studio can check your bindings at design time. Because of this, creating bindings in XAML is often a preferred solution.

Now, run your code one more time. The Settings window should now show the shows you selected before, if the serialization worked right. Try adding a new show and watch it update in real time.

## Your turn: Main window

You’ve now seen how to set up basic collection bindings. You’ll need the following event handlers and other code in MainWindow.xaml.cs (see next page). When you’re done, the following things should happen in the Main window:

* Shows with an episode on the current day (if any) should be listed when the application starts.
* Clicking on a date should show all shows with an episode on the current day.
  + Tip: {Binding} is a valid binding; Path is not required.
* Clicking on a date should change the Label that displays the currently selected date.

None of these changes require any codebehind code other than what you’ve been given.

Question 4: Submit the XAML code and a screenshot (if you’re not sure how to take a screenshot, type “Snipping Tool” in Start Search) of your Main Window with at least one show displayed on a date other than the date of the screenshot. If you are using a word processor that doesn’t retain Visual Studio’s text formatting on copy/paste, please take a screenshot of your code so that it remains properly formatted and colored. (9 points)

## Congratulations

You’re done. Don’t forget: submit answers to the **4** questions in this lab as a .pdf to the appropriate Moodle submission form.

public ObservableCollection<String> SelectedDayEpisodes { get; private set; }

public ObservableCollection<TvdbSeries> Shows { get; set; }

private TvdbHandler \_tvdbHandler;

public MainWindow()

{

Shows = ((App)Application.Current).Shows;

\_tvdbHandler = ((App)Application.Current).TvdbHandler;

SelectedDayEpisodes = new ObservableCollection<String>();

InitializeComponent();

Calendar.SelectedDate = DateTime.Today;

GetDayEpisodes(DateTime.Today);

}

private void GetDayEpisodes(DateTime day)

{

SelectedDayEpisodes.Clear();

var seriesThatAirToday = Shows.Where(series => series.AirsDayOfWeek == day.DayOfWeek).ToList();

var episodesToSort = (from series in seriesThatAirToday

let episode =

series.GetEpisodes(series.NumSeasons).FindLast((ep) => ep.FirstAired == day.Date)

where episode != default(TvdbEpisode)

select new EpisodeToSort() {episode = episode, series = series})

.ToList();

episodesToSort.Sort((c1, c2) =>

{

var c1Time = DateTime.Parse(c1.series.AirsTime);

var c2Time = DateTime.Parse(c2.series.AirsTime);

if (c1Time < c2Time) return -1;

if (c1Time > c2Time) return 1;

return 0;

});

foreach (var e in episodesToSort)

SelectedDayEpisodes.Add(e.series.AirsTime + " - " + e.series.SeriesName + ": " + e.episode.EpisodeName);

}

private void OnSettingsButtonClick(object sender, RoutedEventArgs e)

{

var window = new SettingsWindow();

window.Closed += (o, k) =>

GetDayEpisodes(Calendar.SelectedDate.GetValueOrDefault());

Window.Owner = this;

window.Show();

}

private void CalendarSelectedDatesChanged(object sender, System.Windows.Controls.SelectionChangedEventArgs e)

{

GetDayEpisodes((DateTime)e.AddedItems[0]);

}

private struct EpisodeToSort

{

public TvdbSeries series;

public TvdbEpisode episode;

}