

8NSS Assignment 01 Visual Basic Programming

**Ofiaich College**

**15/11/2011**

Paul Millar

Detailed Report Outlining The Solution Implemented In the Development of the Pay-Roll Project.

Contents

[Introduction 2](#_Toc309325562)

[MVVM Architecture 5](#_Toc309325563)

[Views 6](#_Toc309325564)

[The Splash Screen 6](#_Toc309325565)

[Logging In – The ‘*LoginView.xaml’* Class 7](#_Toc309325566)

[The Main Window – The ‘*PayrollView.xaml’* Class 8](#_Toc309325567)

[Updating Pay Packets – The ‘*PayPacketView.xaml’* Class 10](#_Toc309325568)

[Preferences – The ‘*SettingsView.xaml’* Class 11](#_Toc309325569)

[View Models 12](#_Toc309325570)

[The *‘LoginViewModel.vb’* Class 12](#_Toc309325571)

[The *‘PayPacketViewModel.vb’* Class 12](#_Toc309325572)

[The *‘PayrollViewModel.vb’* Class 12](#_Toc309325573)

[The *‘SettingsViewModel.vb’* Class 12](#_Toc309325574)

[Models (Entity) 13](#_Toc309325575)

[The *‘LoginModel.vb’* Class 13](#_Toc309325576)

[The *‘PayPacketModel.vb’* Class 13](#_Toc309325577)

[The ‘*PayTotalModel.vb’* Class 14](#_Toc309325578)

[The *‘SettingsModel.vb’* Class 14](#_Toc309325579)

[Abstract Classes 15](#_Toc309325580)

[Overview 15](#_Toc309325581)

[Updating The UI – The ‘*DataNotificationBase.vb’* Class 16](#_Toc309325582)

[Validation - The ‘*DataValidationBase.vb’* Class 17](#_Toc309325583)

[Object Editing - The ‘*ModelBase.vb’* Class 18](#_Toc309325584)

[The Data Access Layer 20](#_Toc309325585)

[Binding 21](#_Toc309325586)

[Data Binding 21](#_Toc309325587)

[Command Binding 22](#_Toc309325588)

[Attached Behaviours 23](#_Toc309325589)

[Xaml Styles & Templates 24](#_Toc309325590)

[Toolbar Buttons 25](#_Toc309325591)

[Styling the Data Grid 26](#_Toc309325592)

[LINQ Queries 27](#_Toc309325593)

[Problems 27](#_Toc309325594)

# Introduction

This is a simple WPF payroll application that implements a light-weight Model-View-View Model (MVVM) architecture. The solution also employs LINQ queries, Attached Behaviours, Command Binding, XAML Styles, Data & Control Templates. As our repository we are using random access data files. Multiple files can be created and opened for editing. In addition to the MVVM architecture, I have included additional folders to further separate solution concerns. We will go through each one and give a brief explanation for each.

It should be pointed out that all classes also have a comment to give an overview of its functionality, with each member in the class also containing a comment for a more specific explanation, for example ...

Private \_Index As Integer

''' <summary>

''' Represents The Position Within The Text File Of The Beginning Of Each Record.

''' </summary>

''' <remarks>This Will Allow Us To Edit and Delete Records Within Our Text File</remarks>

Public Property Index() As Integer

Get

Return \_Index

End Get

Set(ByVal value As Integer)

\_Index = value

' Once This Has Been Assigned An Index Number, This Is No Longer Considered A New Record.

\_IsNew = False

End Set

End Property

The final solution allows a user to ...

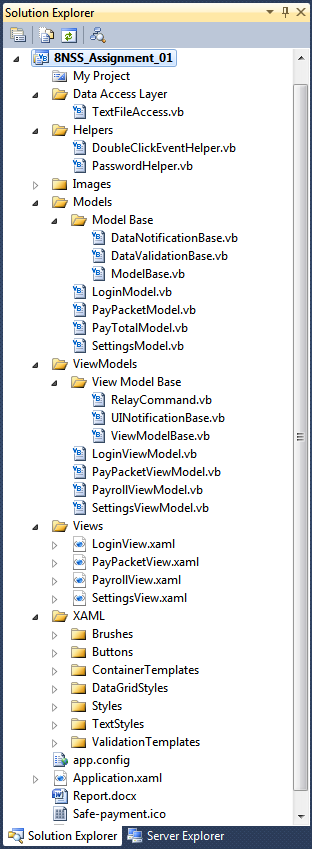
* Login
* Create, open and remove pay roll files.
* Add, edit and delete pay packet records.
* Change preferences in relation to minimum and maximum wage limits.
* Change preferences in relation to the ceiling limits for basic, time and half, and double time hours allowed.

The values for preferences are stored in the configuration file under the ‘MySettings’ section.

To login you’ll need to enter the following credentials ...

Username: student1

Password: ofiaich1

This is a quick look at how the overall solution is laid out ...

**Where to Start**

It all starts in the ‘Application\_Startup’ method within the ‘Application.xaml.vb’ file.

We first make a call to the ‘Init’ method. This routine configures some aspects of the application. For example, in WPF, the en-US culture is assumed by default, forcing us to instruct the application to get the current machine culture.

FrameworkElement.LanguageProperty.OverrideMetadata(GetType(FrameworkElement), New FrameworkPropertyMetadata(System.Windows.Markup.XmlLanguage.GetLanguage(CultureInfo.CurrentCulture.IetfLanguageTag)))

I have also registered a handler for the ‘GotFocusEvent’ event for text boxes. This will highlight the entire content of all text boxes when the user tabs into them.

We then display the splash screen for 3 seconds after which the login window is displayed.

logview = New LoginView

logviewmodel = New LoginViewModel

logview.DataContext = logviewmodel

logview.ShowDialog()

As you can see, we instantiated new objects for the view and the view model and then assigned the view model as the data context of the view.

At this point, program execution shifts to the login window. When it is closed, execution returns back to the ‘Application\_Startup’ method and we test to see if the correct details were entered ...

If logviewmodel.LoginCredentials.IsAuthentic = True Then

Me.RunPayroll()

Else

My.Application.Shutdown()

End If

If the credentials entered are valid, we open the Payroll window, otherwise we shut down the application.

We instantiate new objects for the payroll view and view model and assign the data context in the same manner as we did with the login window.

prview = New PayrollView

prviewmodel = New PayrollViewModel

prview.DataContext = prviewmodel

prview.Show()

# MVVM Architecture

Simply put, MVVM is a design pattern used in software engineering as a means of separating concerns that allow developer teams to work independently on different parts of a solution, and then bring these pieces together at runtime.

This pattern contains the following elements ...

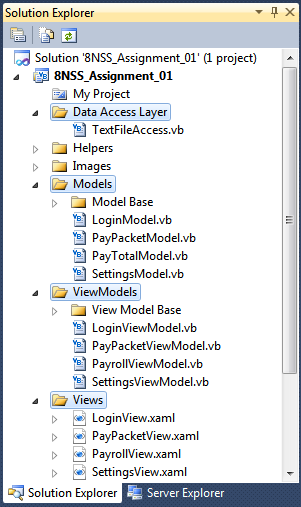
* Model: A representation of our data entities.
* View-Model A class that act as the data context for a view. It binds the Model to the View.
* View Your UI, Windows, User Controls, etc ...

In addition to this, I have also added a ‘Data Access Layer’ that persists all updates to our ‘data’ text files.

Simply putting it, the controls on your view (window) are bound to the properties in the model through the view model.

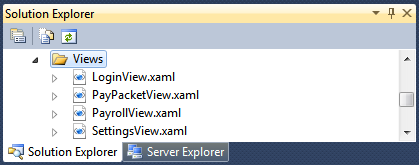
All validation should happen in your model so that validation is uniform throughout the application no matter where the model is used.

The MVVM structure used in this solution is laid out as follows



This is a light-weight MVVM architecture in that the solution has not been seperated into different projects.

## Views



There are four views or windows with the splash screen being just an image that’s displayed for a set amount of time.

### The Splash Screen



The splash screen is simply an image I made up using Photoshop. There are no window control involved. In the file ‘*Application.xaml.vb’* we simply instantiate a splash screen object, passing in the name of the image to display. I then created a timer object and set it’s ‘Interval’ property to 3 seconds. When this time has elapsed, the splash screen is closed.

Instantiate a new splash screen object and display it.

splashScreen = New SplashScreen("Images/PayProSplashScreen24.png")

splashScreen.Show(False)

Create a timer object that ‘kicks in’ after 3 seconds and calls the ‘CloseSplash’ method to close the splash screen.

timer = New Windows.Forms.Timer

timer.Interval = 3000

AddHandler timer.Tick, AddressOf CloseSplash

timer.Start()

The following method handles the ‘Tick’ event of the timer.

Private Sub CloseSplash(ByVal sender As Object, ByVal e As EventArgs)

Dim span As TimeSpan = New TimeSpan(0, 0, 0, 1, 0)

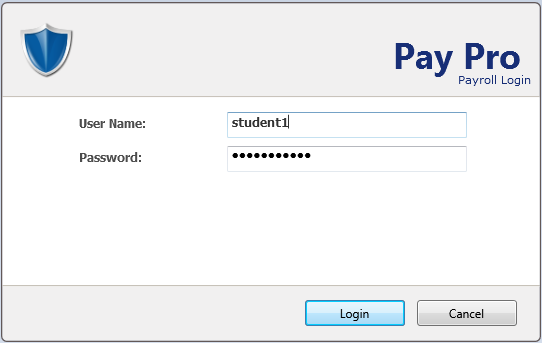
splashScreen.Close(span)

timer.Dispose()

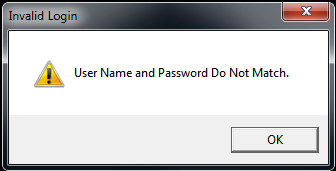
Me.RunLogin()

End Sub

### Logging In – The ‘*LoginView.xaml’* Class



This is our login window where the user must enter the correct username and password to gain access to the program. If incorrect credentials are entered, the following message appears ...



This message will display indefinitely until the user either enters the correct details or cancels.

The username and password are properties in the *‘LoginModel.vb’* class and we bind to these through the *‘LoginViewModel.vb’* class via a property reference to the class.

Public Property LoginCredentials() As LoginModel

We then use Xaml to do the actual binding...

Text="{Binding Path=LoginCredentials.Input\_UserName}"

Because WPF does not allow direct binding to a Password control, I had to use an ‘Attached Behaviour’ Property. This allowed me to attach a dependency property to the password control that essentially updated the contents of a textblock with the value entered by the user, I was then able to bind to that control. I used the *‘PasswordHelper.vb’* class to do this.

### The Main Window – The ‘*PayrollView.xaml’* Class

This is the heart of the application. We can create, open or delete payroll files, view the pay packets for that file, add, edit and delete pay packets, view overall totals and averages. An image of this window can be seen on the next page.

The buttons on the toolbar perform the following functions (each button has also a tooltip associated with them)...

 Shuts the application down.

 Opens the preferences window.

 Adds a new payroll file.

 Opens a payroll file.

 Deletes a payroll file.

 Adds a new pay packet.

 Edits the highlighted pay packet (you can also double click on the data grid row).

 Deletes the highlighted pay packet.

How I bound the buttons to their relevant command is covered under the ‘*Binding’* section.

All pay packets are displayed a Data Grid control. This control is bound to the ‘*paypacketCollection’* property which is an ObservableCollection, which can be found in the ‘*PayrollViewModel.vb’* class.

We bind to the ‘ItemsSource’ property of the data grid.

ItemsSource="{Binding paypacketCollection}"

And then bind each individual column

Binding="{Binding Path=Name}"

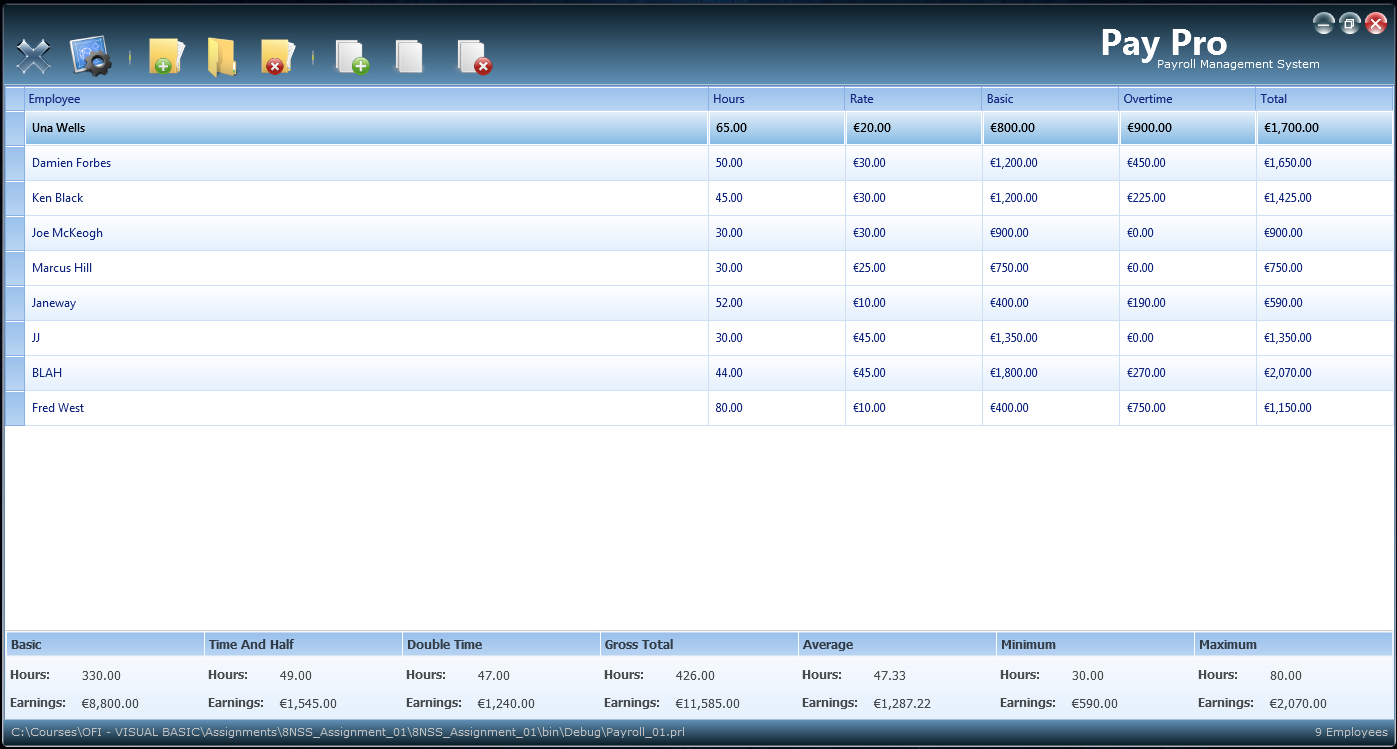
The data grid is not the standard control provided by .NET Framework, but rather it’s provided by the ‘WpfToolkit’ library, downloadable from the following site - <http://wpf.codeplex.com/>.

How I styled the data grid is explained later in the XAML Styles section. Sorting is also available just by clicking on the column headers.

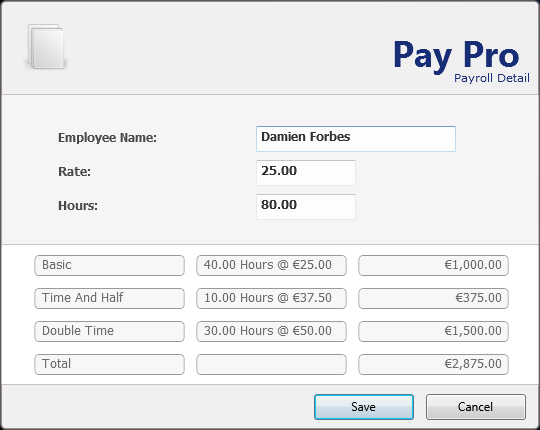
The totals are displayed in an Items control. We bind to the ‘*PayTotalCollection’* property in exactly the same manner as the data grid. However, in addition, I created a Xaml data template that is bound to the control.

ItemTemplate="{StaticResource PayTotalItemTemplate}"

This template can be found here ‘\XAML\ContainerTemplates\PayTotalItemTemplate.xaml’. It is in this template that we bind directly to the properties in the ‘*PayTotalModel.vb’* class.



### Updating Pay Packets – The ‘*PayPacketView.xaml’* Class



This window allows us to add or edit a pay packet object. It binds to the *‘PayPacketModel.vb’* class through the *‘PayPacketViewModel.vb’* class.

As the user changes either the rate or the hours, the values for basic, time and half, double time and overall totals are automatically recalculated. Validation for this window is explained later in the ‘*Validation - The ‘DataValidationBase.vb’ Class’* section but basically the following conditions must be met in order for the pay packet object to be valid.

* A name must be supplied.
* The rate cannot be lower than or greater than the minimum and maximum wages specified in the preferences window.
* Hours worked cannot be below zero or greater than the hours ceiling specified for double time.

The totals section of this form is just a ‘WrapPanel’ with each item being just a border with a TextBlock inside of it.

The Text block’s are bound to properties in the ‘*PayPacketModel.vb’* class through the *PayPacket* property in the ‘*PayPacketViewModel.vb’* class.

Public Property payPacket() As PayPacketModel

This binding is formatted to display the local machine currency.

Text="{Binding Path=payPacket.BasicEarnings, StringFormat={}{0:c}}"

### Preferences – The ‘*SettingsView.xaml’* Class

# 

The values entered here by the user govern the values that can be entered for each pay packet.

For example, in the above screen shot, we have specified that employees cannot earn less than €8.50 per hour or more than €50 per hour.

We have also specified that employees will not be paid for any hours worked over the ceilingvalue for double time (80 hours).

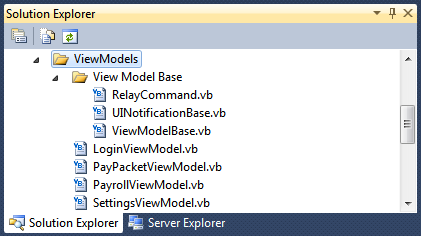
Validation criteria are as follows ...

* Minimum wage cannot be greater than the maximum wage.
* Double time ceiling cannot be equal to or less than Time and half ceiling.
* Time and half ceiling cannot be equal to or less than Flat week ceiling.

Controls on this window bind to the *‘SettingsModel.vb’* class through the *‘SettingsViewModel.vb’* class.

Any changes made here will still have an effect on the calculations of pay packets even if they were created before the change was made.

## View Models



### The *‘LoginViewModel.vb’* Class

This class binds the ‘*LoginModel.vb’* class to the ‘*LoginView.xaml’* class.

It has one constructor that instantiates a new *LoginModel* object.

\_LoginCredentials = New LoginModel

It has two *ICommand* properties that are bound to the ‘*Login’* and ‘*Cancel* buttons on the view. How these are bound is explained under the ‘*Binding’* section.

### The *‘PayPacketViewModel.vb’* Class

This class binds the ‘*PayPacketModel.vb’* class to the ‘*PayPacketView.xaml’* class.

‘*PayPacketView.xaml’* is used for adding and editing pay packets. In the constructor we pass in the pay packet object we want to change and also a reference to a ‘TextFileAccess’ object that is used to persist our data to the text file.

It has two *ICommand* properties that are bound to the ‘*Save* and ‘*Cancel* buttons on the view.

It also has a Boolean property called ‘*WasSuccessful’* that we use to test if the update went according to plan.

### The *‘PayrollViewModel.vb’* Class

This class binds both the ‘*PayPacketModel.vb’* class and the *PayTotalModel.vb’* to the ‘*PayrollView.xaml’* class.

This is where we calculate all the totals for each pay roll file. It also has several *ICommand* properties and methods for updating pay roll files and records.

### The *‘SettingsViewModel.vb’* Class

This class binds the ‘*SettingsModel.vb’* class to the ‘*SettingsView.xaml’* class.

It has one property that makes a reference to the ‘*SettingsModel’* and two ICommand properties and methods used for saving and cancelling any updates to the configuration file.

## Models (Entity)

### The *‘LoginModel.vb’* Class

This class is used to retrieve the username and password from the configurations file and then compare them with the values entered by the user. Again, this class only inherits from the ‘*DataNotificationBase’* class.

The class has three properties, *Username, Password and IsAuthentic*. IsAuthentic returns a Boolean value, ‘True’, if the credentials entered by the user match those in the configurations file, and ‘False’ if they don’t. The function ‘*Authenticate’* performs the actual comparison and also returns true if values match.

### The *‘PayPacketModel.vb’* Class

This class inherits from the ‘*ModelBase’* class. There are fifteen properties, of which the user only directly changes three of them, *Name, Rate and Hours.*

Most of the remaining properties concerning earnings for basic, time and half and double time pay are recalculated depending on the values entered for rate and hours worked. The ‘NotifyUI’ routine is also called in the ‘Set’ methods of both the Rate or Hours properties. This forces the UI to call the ‘Get’ methods of these read only properties, which in turn, forces a recalculation of the values.

To facilitate inserts, edits and deletes of pay packets, I introduced another property, ‘*Index’*. As every three lines in a text file represents our pay packet object, the index property marks the starting ordinal position of each record, i.e. where the ‘Name’ is specified. Typically the values in this property will be 0, 3, 6, 9, 12, etc ... We can then pass this position to the ‘TextFileAccess’ object which can then write to or remove from specific points in the file.

The constructor for this class takes a Boolean argument specifying if the user is inserting a new pay packet record, or not. This value is assigned to the ‘\_*IsNew’* Boolean variable and is used to distinguish between an insert and an edit. By default, this value is false (editing).

We instantiate a new model in the following way when we want to insert a new pay packet ...

Dim payObj As New PayPacketModel(True)

Otherwise we just omit the argument.

### The ‘*PayTotalModel.vb’* Class

This class is where we store all our totals for each payroll file.

This class only inherits from the ‘*DataNotificationBase’* class. As the user will not be directly updating this model, no validation is required, so we only need the functionality that updates the UI to display the most recent totals.

There are only three properties, *TotalsTitle, Hours and Earnings*.

The totals are calculated in the ‘*PayrollViewModel’* class. Within the ‘*RefreshTotals’* method, we add a new Totals object in the following manner ...

\_PayTotalCollection.Add(New PayTotalModel With {.TotalsTitle = "Basic", .Hours = Me.CalculateBasicHours, .Earnings = Me.CalculateBasicEarnings})

‘CalculateBasicHours’ and ‘CalculateBasicEarnings’ are functions that return a double value from a LINQ query.

We calculate the total ‘Basic’ hours worked by all employees like so ...

Private Function CalculateBasicHours() As Double

Return Aggregate p In \_paypacketCollection

Select p.BasicHours

Into Sum()

End Function

And we calculate the total ‘Basic’ earnings for all employees like this ...

Private Function CalculateBasicEarnings() As Double

Return Aggregate p In \_paypacketCollection

Select p.BasicEarnings

Into Sum()

End Function

We’ll cover LINQ queries in more depth later in the ‘*LINQ Queries’* section.

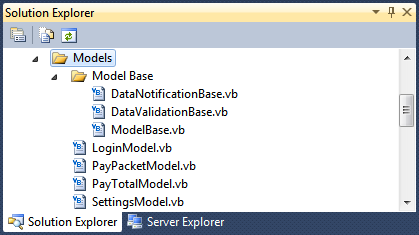
### The *‘SettingsModel.vb’* Class

This class allows the user to set their preferences in relation to minimum & maximum wage rates, and ceiling values for basic, time and half and double time hours. The values are stored in the configurations file. We use the ‘*Read’* method to read the values from the configurations file and the ‘*Save*’ method to persist values to the configurations file.

The ‘*UpdateUI’* method is called by all properties in their ‘Set’ methods because we need to validate multiple properties at the same time. For example, we need to validate both the minimum and maximum rates together, and the ceiling values must also be validated together.

This class inherits from the ‘*ModelBase’* class.

## Abstract Classes



### Overview

***Model ‘Base’ Classes***

We have three ‘Base’ or abstract classes, ‘*ModelBase.vb’*, ‘*DataNotificationBase.vb’* and *‘DataValidationBase.vb’* andare utilised by our ‘Data Entity’ classes. These are located in the ‘/Models/Base Models’ folder*.*

The ‘*ModelBase’* class inherits from the ‘*DataValidationBase’* class which in turn inherits from the ‘*DataNotificationBase’* class.

All three base classes provide additional functionality for our entity models such as updating values on the UI, validation, backup, modifications tracking, etc ...

***View Model ‘Base’ Classes***

There are two abstract classes that serve our View Model classes, ‘*UINotificationBase.vb’* and ‘*ViewModelBase.vb’*.

The ‘*UINotificationBase.vb’* class is exactly the same as the ‘*DataNotificationBase.vb’* class so we will cover this once (on page 16).

The ‘*ViewModelBase.vb’* class provides ICommand members for basic window control such as minimising, maximising, restoring and closing. It also provides handlers for the ‘*MouseLeftButtonDownEvent’* that allows us to drag a window across the screen and the ‘*MouseDoubleClickEvent’* for data grids.

As a lot of this class concerns binding, we will take a more detailed look at this in the ‘*Binding’* section.

.

### Updating The UI – The ‘*DataNotificationBase.vb’* Class

This class implements the ‘*INotifyPropertyChanged’* interface and is used to notify the UI that the value of a property has changed and that it must update the display. In the ‘Set’ method of a ‘Data Entity’ property, we inform the UI of a change by calling the ‘*ReportPropertyChanged’* method, passing in the name of the property whose value has changed.

ReportPropertyChanged("Hours")

The UI then makes a call to the property’s ‘Get’ method to retrieve the new value.

In the’ PayPacketModel’, the ‘NotifyUI’ routine is also called in the ‘Set’ methods of both the Rate or Hours properties. In that routine the following code snippet exists ...

ReportPropertyChanged("TimeAndHalfRate")

ReportPropertyChanged("DoubleRate")

ReportPropertyChanged("TimeAndHalfHours")

ReportPropertyChanged("DoubleTimeHours")

ReportPropertyChanged("BasicEarnings")

ReportPropertyChanged("TimeAndHalfEarnings")

ReportPropertyChanged("DoubleTimeEarnings")

ReportPropertyChanged("TotalEarnings")

This forces the UI to call the ‘Get’ methods of these read only properties, which in turn, forces a recalculation of the values concerning earnings, hours worked & the different rates, and then updates the controls on the UI.

### Validation - The ‘*DataValidationBase.vb’* Class

Before we continue, when applying validation, we must consider two things; the logic used to validate our models and how errors are displayed to the user. We’ll discuss the logic first and then the ‘Styles’ used to actually display errors on the UI.

This ‘*DataValidationBase.vb’* class inherits from the ‘*DataNotificationBase’* class and also implements the ‘*IDataErrorInfo’* Interface which is used to warn the user that the information they have just entered is invalid and that it must be corrected (if it is indeed invalid). Any errors are added to a collection ...

Private \_ValidationErrorsList As New Dictionary(Of String, String)

The ‘Dictionary’ represents a paired collection of *Keys* and *Values,* the keys must be unique. There are two class members that update this collection; ‘*AddError’* and ‘*RemoveError’*. We pass the property name to both methods. The error message is also passed to the ‘AddError’ method. As the information is being entered by the user, we always remove any previous errors for that property, and then re-evaluate. The first error encountered (if any) is then added.

For example, in the ‘*Set’* method of the ‘*Hours’* property in the ‘*PayPacketModel’* we have the following code ...

Me.RemoveError("Hours")

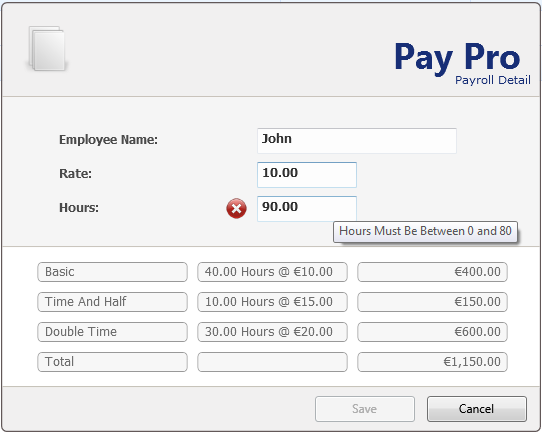
If value < 0 Or value > My.Settings.Hours\_DoubleTimeCeiling Then

Me.AddError("Hours", "Hours Must Be Between 0 and " & My.Settings.Hours\_DoubleTimeCeiling)

End If

Always remove any previous errors and then re-evaluate.

In the following screen shot, we’ve entered more hours than are permitted (only 80 hours are allowed in this case). Have a look at the tooltip and then look at the code above ...



As you can see, an error icon is shown to the left of the textbox and the ‘Save’ button is disabled. Also, if you hover the mouse over the textbox, a tooltip displays, informing the user of the specific error.

We use a Xaml ‘Style’ to display the error. Style resources are the equivalent to CSS files. In this case we apply the ‘Style\_ErrorInTextTooltip’ style to all textboxes, this can be found in the following location - \XAML\TextStyles\TextBoxStyles.xaml and employs the ‘InputValidationError’ control template which can be found here - \XAML\ValidationTemplates\ErrorTemplate.xaml.

In the style we have a trigger that fires when the property that is bound to the textbox has an error. Essentially it tests the ‘HasError’ property which returns true if there are errors. When this occurs, we set the tooltip to the error message.

<Trigger Property="Validation.HasError" Value="true">

<Setter Property="ToolTip" Value="{Binding RelativeSource={x:Static RelativeSource.Self}, Path=(Validation.Errors)[0].ErrorContent}"/>

</Trigger>

To disable the ‘Save’ button when we have an error, we test the ‘IsValid’ property which returns a boolean value that indicates if there are any errors (returns ‘True’ if there are no errors). In our view models (see *PayPacketViewModel.vb*), this is utilised by the ‘CanSaveExecute’ function of the ‘SaveCommand’. ‘SaveCommand’ is of type ‘*ICommand’,* that allows us to bind a command to a control in an MVVM environment. Simply put, the value returned from the ‘IsValid’ property determines whether we can execute a command, or not. This ensures that the information entered by the user is actually valid before being saved.

Private Function CanSaveExecute() As Boolean

Return \_payPacket.IsValid

End Function

### Object Editing - The ‘*ModelBase.vb’* Class

This class inherits from the ‘*DataValidationBase*’class and also implements the ‘*IEditableObject’* interface which allows us to commit or rollback changes to our data entity models. It has three methods that allow us to do that; *BeginEdit*, *CancelEdit* and *EndEdit*.

The *BeginEdit* method calls a backup method to make a copy of the data entity before changing it; BackupData().

*CancelEdit* calls a restore method to rollback the object back to its original state; BackupRestore().

*EndEdit* makes a call to clear the backup; BackupClear().

These methods are essential when directly editing an object in a DataGridRow and also if using a separate ‘Detail’ window to make changes.

Because all our entity models are different, we must manually override the backup methods within each model to add its own distinct backup implementation. We must also create some sort of structure to store a copy of the object.

There are three methods to override in the ‘*ModelBase.vb’* class...

Protected MustOverride Sub BackupData()

Protected MustOverride Sub BackupRestore()

Protected MustOverride Sub BackupClear()

In the ‘*PayPacketModel.vb’* class, I defined a structure to be our backup ...

Private Structure PayPacketBackup

Dim Name As String

Dim Hours As Double

Dim Rate As Double

End Structure

We create an instance of it ...

Private Backup As New PayPacketBackup

And backup the data ...

Protected Overrides Sub BackupData()

Backup.Name = Me.Name

Backup.Hours = Me.Hours

Backup.Rate = Me.Rate

End Sub

To restore our data we call this method ...

Protected Overrides Sub BackupRestore()

Me.Name = Backup.Name

Me.Hours = Backup.Hours

Me.Rate = Backup.Rate

End Sub

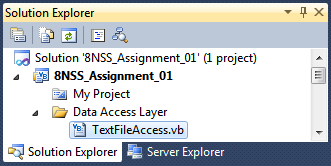
To clear the backup we call this method ...

Protected Overrides Sub BackupClear()

Backup = Nothing

End Sub

## The Data Access Layer



This folder contains only one class ‘TextFileAccess.vb’. The sole purpose of this class is to deal with all tasks related to our ‘data’ text files, whether that be creating, opening, reading from, writing to or removing the file from the computer altogether. This class at present only handles some basic functionality when dealing with text files but it can be adapted and added to, to increase its capabilities, and, prescribing to OOP principles can also be used to any program.

It works like this, We have a strongly typed List(Of String) that acts as a buffer between the program and the text file....

Private \_FileContent As List(Of String) = New List(Of String)

* Each line of our text file is read into this list.
* The program then reads from this list and creates its own ‘record’ objects.
* All writes are made the text file and the list is then repopulated.

The constructor takes 3 ‘String’ arguments - Dialog Title, Default Extension and Filter which are assigned to properties for used when opening or creating a file. In the ‘PayrollViewModel’ we instantiate the object as follows ...

Private \_dataAccess As TextFileAccess

Public ReadOnly Property DataAccess() As TextFileAccess

Get

If \_dataAccess Is Nothing Then

\_dataAccess = New TextFileAccess("Payroll File", ".prl", "Payroll Files|\*.prl")

End If

Return \_dataAccess

End Get

End Property

I used a property to reference the class object so that we could bind its ‘CurrentFileName’ property to a TextBlock on the main window that would inform the user of the file they are currently working on...



We use the following XAML code in the ‘*PayrollView.xaml’* file to bind ...

Text="{Binding DataAccess.CurrentFileName}"

# Binding

## Data Binding

For this example, I will show you how I bound the *UserName* property in the ‘*LoginModel.vb’* class to the *TextBox* control on the *LoginView* window.

Firstly, we declare the property in the *LoginModel* class...

Private \_Input\_UserName As String

''' <summary>

''' Gets / Sets The 'UserName' As Entered By The User On The UI.

''' </summary>

Public Property Input\_UserName() As String

Get

Return \_Input\_UserName

End Get

Set(ByVal value As String)

\_Input\_UserName = value

ReportPropertyChanged("Input\_UserName")

End Set

End Property

Secondly, we instantiate a *LoginModel* object in the *LoginViewModel* class*...*

Private \_LoginCredentials As LoginModel

''' <summary>

''' Property Used For Binding Our Login Model Data To The View.

''' </summary>

Public Property LoginCredentials() As LoginModel

Get

If \_LoginCredentials Is Nothing Then

\_LoginCredentials = New LoginModel

End If

Return \_LoginCredentials

End Get

Set(value As LoginModel)

\_LoginCredentials = value

End Set

End Property

Then, we bind to the TextBox control in the *LoginView*...

<TextBox Name="tbxUserName"

Width="240"

Style="{StaticResource TextBoxStyle}"

Text="{Binding Path=LoginCredentials.Input\_UserName}">

</TextBox>

Lastly, we knit al this together by assigning the *LoginViewModel* class as the data context for the *LoginView*.

logview = New LoginView

logviewmodel = New LoginViewModel

logview.DataContext = logviewmodel

## Command Binding

For this example, I will show you how I bound the ‘Login’ button to its *ICommand* in the ‘*LoginViewModel.vb’* class.

For this to work, we need to employ the services of the ‘*RelayCommand.vb’* class which implements the *ICommand* interface. Essentially it registers our command with the *CommandManager* class and provides command related functionality.

Firstly, we create a ICommand object and initialise it to a new RelayCommand. We pass in the address of the routine that act as the handler for the ‘Execute’ event. In this case it’s the ‘LoginExecute’ method.

Private \_LoginCommand As ICommand

''' <summary>

''' Property That Provides Command Binding For The View's 'Login' Button Control.

''' </summary>

Public ReadOnly Property LoginCommand() As ICommand

Get

If \_LoginCommand Is Nothing Then

\_LoginCommand = New RelayCommand(AddressOf LoginExecute)

End If

Return \_LoginCommand

End Get

End Property

''' <summary>

''' Attempts Login, Checks First That The Login Object Is Authentic.

''' </summary>

''' <remarks>Execution Logic For Attempting A Login.</remarks>

Private Sub LoginExecute()

If \_LoginCredentials.IsAuthentic Then

Me.Close()

Else

MessageBox.Show("User Name and Password Do Not Match.", "Invalid Login", MessageBoxButton.OK, MessageBoxImage.Exclamation)

End If

End Sub

We then simply bind the command property to the button control...

<Button Name="btnLogin"

Content="Login"

Style="{StaticResource UpdateButtonStyle}"

Command="{Binding Path=LoginCommand}"

IsDefault="True" />

## Attached Behaviours

I’ve just shown you how to bind a command to a control, but how do you bind a command to an event ?

In this case, I wanted to bind the ‘*EditCommand’* in the ‘*PayrollViewModel*.*vb’* class to the ‘*DoubleClickEvent’* of the payroll data grid in the ‘*PayrollView.xaml’* class.

The answer lay in what’s called an ‘Attached Behaviour Dependency Property’. Under the ‘*Helpers’* folder we have a static class called ‘*DoubleClickEventHelper.vb’*. This class simply allows us to create a property that can be attached to the data grid, that will then inform the control where to look for the handler for this event.

After creating the class, we simply reference the namespace of the class in our view...

xmlns:local="clr-namespace:Helpers.Events"

And attach the property to the data grid, pointing it to the command we want to execute.

local:DoubleClickEventHelper.HandleDoubleClick="True"

local:DoubleClickEventHelper.TheCommandToRun ="{Binding Path=EditCommand}">

I would point out that I found this solution online here...

<http://sachabarber.net/?p=532>

The solution was written in C# but I was able to port it to VB.

# Xaml Styles & Templates

Xaml styles and templates behave just like CSS files and can make development that much easier and quicker whilst providing a much richer UI experience for the end user. Styles and templates are created in a ‘*ResourceDictionary’.*

For an example of how we used styles, let’s have a look at how we styled TextBlocks (I used these instead of *Label* controls).

The style to be applied can be found here *‘\XAML\TextStyles\TextBlockStyles.xaml’*...

<!--Base Style-->

<Style x:Key="TextBlockBaseStyle" TargetType="TextBlock" >

<Setter Property="HorizontalAlignment" Value="Left" />

<Setter Property="VerticalAlignment" Value="Top"/>

<Setter Property="FontWeight" Value="Normal"/>

<Setter Property="Foreground" Value="Black" />

<Setter Property="FontFamily" Value="Tahoma" />

<Setter Property="Padding" Value="0"/>

<Setter Property="FontSize" Value="13"/>

<Setter Property="Height" Value="26" />

<Setter Property="Opacity" Value="0.7" />

<Setter Property="TextOptions.TextFormattingMode" Value="Display"/>

<Setter Property="TextOptions.TextRenderingMode" Value="ClearType"/>

</Style>

<!--Used On Forms-->

<Style x:Key="LabelTextBlock" TargetType="TextBlock" BasedOn="{StaticResource TextBlockBaseStyle}">

<Setter Property="FontWeight" Value="SemiBold"/>

<Setter Property="FontSize" Value="12"/>

<Setter Property="Margin" Value="26,8,0,0"/>

</Style>

As you can see, the ‘LabelTextBlock’ inherits all the properties from the ‘TextBlockBaseStyle’ style.

In the ‘*Application.xaml’* class we import the style so it can be used anywhere in our app.

<Application.Resources>

<ResourceDictionary>

<ResourceDictionary.MergedDictionaries>

...

<ResourceDictionary Source="/XAML/TextStyles/TextBlockStyles.xaml"/>

...

</ResourceDictionary.MergedDictionaries>

</ResourceDictionary>

</Application.Resources>

</Application>

In the ‘*LoginView.xaml’* window we applied the style to the ‘Username’ like so ...

<TextBlock Text="User Name:" Style="{StaticResource LabelTextBlock}"/>

‘StaticResource’ refers to a resource that doesn’t change during program execution.

## Toolbar Buttons

Toolbar buttons like the following...

Were created from png images that I downloaded from <http://www.iconspedia.com/> (and manipulated) and placed onto a button that had an ‘*invisible*‘ style applied to it. The style that I created can be found in the *‘\XAML\Buttons\TransparentButtonStyle.xaml’* resource file.

**The Glass Button**

I used the style from the following file -*‘\XAML\Buttons\GlassButtonSmallStyle.xaml’* to style the buttons on the main payroll window that are used for minimising, maximising and closing the window. The images inside, I made up in Photoshop.



The style inherits all the properties from the ‘*GlassButtonStyle.xaml’* resource which I downloaded from the following site - [http://www.codeproject.com/KB/WPF/glassbuttons.aspx](http://www.codeproject.com/KB/WPF/glassbuttons.aspx%20) .

Even so, I added my own implementations to the style so that the font changes colour when the mouse ‘hovers’ over them and when the buttons are ‘pressed’. The Y axis of any content will now also drop a notch to give the button a ‘sunk’ look when pressed.

Here’s a look at some of the additions I made to ‘*GlassButtonStyle.xaml’*.

<Trigger Property="IsMouseOver" Value="True">

<Setter Property="Foreground" Value="#F6F5EFA0"/>

<Setter Property="FontWeight" Value="SemiBold"/>

</Trigger>

<Trigger Property="IsEnabled" Value="False">

<Setter Property="Foreground" Value="Gray" />

<Setter Property="Opacity" Value="0.5"/>

</Trigger>

<Trigger Property="IsPressed" Value="True">

<Setter Property="Foreground" Value="#FF87BBE5"/>

<Setter TargetName="ButtonContent" Property="RenderTransform" >

<Setter.Value>

<TranslateTransform Y="1.0" />

</Setter.Value>

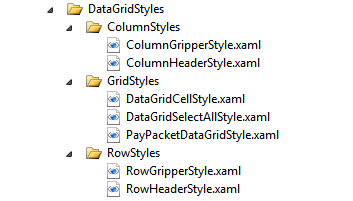
</Setter>

</Trigger>

I then created ‘*GlassButtonSmallStyle.xaml’* based on this style.

## Styling the Data Grid

The following styles and templates were used to style the main data grid.



The data grid is not the standard control provided by .NET Framework, but rather it’s provided by the ‘WpfToolkit’ library, downloadable from the following site - <http://wpf.codeplex.com/>. The reason for using this data grid was simply a matter of curiosity which sadly failed to find any significant changes or advantages over the standard data grid. A reference to this library had to be made in the solution which automatically copies the ‘WPFToolkit.dll’ file to the debug directory of the solution upon compiling.

I was able to get most of the style from [http://blog.smoura.com/introducing-the-wpf-toolkit-datagrid/](http://blog.smoura.com/introducing-the-wpf-toolkit-datagrid/%20) , I then dissected, altered and added my own implementations to suit my own needs.

Part of my own implementation was to create the following style that is applied to each individual cell of the data grid...

<Style x:Key="DataGridCellStyle" TargetType="{x:Type WpfToolkit:DataGridCell}">

<Setter Property="TextBlock.VerticalAlignment" Value="Stretch"/>

<Setter Property="TextBlock.Padding" Value="10,20,0,0"/>

<Setter Property="TextOptions.TextFormattingMode" Value="Display"/>

<Setter Property="TextOptions.TextRenderingMode" Value="ClearType"/>

<Setter Property="VerticalContentAlignment" Value="Center" />

<Style.Triggers>

<Trigger Property="IsSelected" Value="True">

<Setter Property="Background" Value="{StaticResource ItemBackgroundBrush\_IsSelected}" />

<Setter Property="BorderBrush" Value="{x:Null}" />

<Setter Property="Foreground" Value="Black"/>

<Setter Property="FontWeight" Value="SemiBold"/>

<Setter Property="FontSize" Value="12"/>

</Trigger>

</Style.Triggers>

</Style>

Which has the following effect on the ‘Phillip Wells’ row when selected...



# LINQ Queries

For totalling our payroll files I used LINQ queries. These following functions can be found in the ‘*PayrollViewModel.vb’* class.

Firstly, I declared a class variable to reference a pay packet object.

Private p As PayPacketModel

And below are examples of how I totalled the hours for all employees.

Total *hours*...

Private Function CalculateDoubleHours() As Double

Return Aggregate p In \_paypacketCollection

Select p.DoubleTimeHours

Into Sum()

End Function

Average hours...

Private Function CalculateAverageHours() As Double

Return Aggregate p In \_paypacketCollection

Select p.Hours

Into Average()

End Function

Minimum hours...

Private Function CalculateMinimumHours() As Double

Return Aggregate p In \_paypacketCollection

Into Min(p.Hours)

End Function

Maximum hours...

Private Function CalculateMaximumHours() As Double

Return Aggregate p In \_paypacketCollection

Into Max(p.Hours)

End Function

# Problems

Really the only problem I had was handling the *DoubleClickEvent* for the data grid which I discussed earlier on page 23 - ‘*Attached Behaviours’*.