

**Recueil d’exercices**

**Version 3.0**

Formation W097 - 2 :

Approfondissements de Silverlight

**© Winwise 2009.**   
Tous droits de reproduction, traduction et adaptation réservés pour tous pays

Sommaire

[1ère Partie : Création d’un Media Player 3](#_Toc242775578)

[Création du Média Player de base 3](#_Toc242775579)

[Mise en place des sous-titres 6](#_Toc242775580)

[Vidéos encodées avec les sous-titres 7](#_Toc242775581)

[Vidéos avec fichier de sous-titres 8](#_Toc242775582)

[2ème Partie : Création d’un contrôle personnalisé 11](#_Toc242775583)

[3ème Partie : Interaction DOM / Code Managé 22](#_Toc242775584)

[Ajouter la gestion du "Drag &Drop" 23](#_Toc242775585)

[Ajouter la gestion du zoom 26](#_Toc242775586)

[4ème Partie : Gestion de la navigation 29](#_Toc242775587)

[5ème Partie : Implémentation du mode Out Of Browser 37](#_Toc242775588)

[6ème Partie : Réalisation de tests unitaires 41](#_Toc242775589)

[Création des tests 42](#_Toc242775590)

[7ème Partie : Développement d’un contrôle accessible 50](#_Toc242775591)

[8ème Partie : Réalisation de tests d’interface 54](#_Toc242775592)

[9ème Partie : Isolated Storage & OpenFileDialog 56](#_Toc242775593)

[Utilisation du contrôle OpenFileDialog 56](#_Toc242775594)

[Utilisation du stockage isolé 60](#_Toc242775595)

[10ème Partie : Utilisation du BackgroundWorker 66](#_Toc242775596)

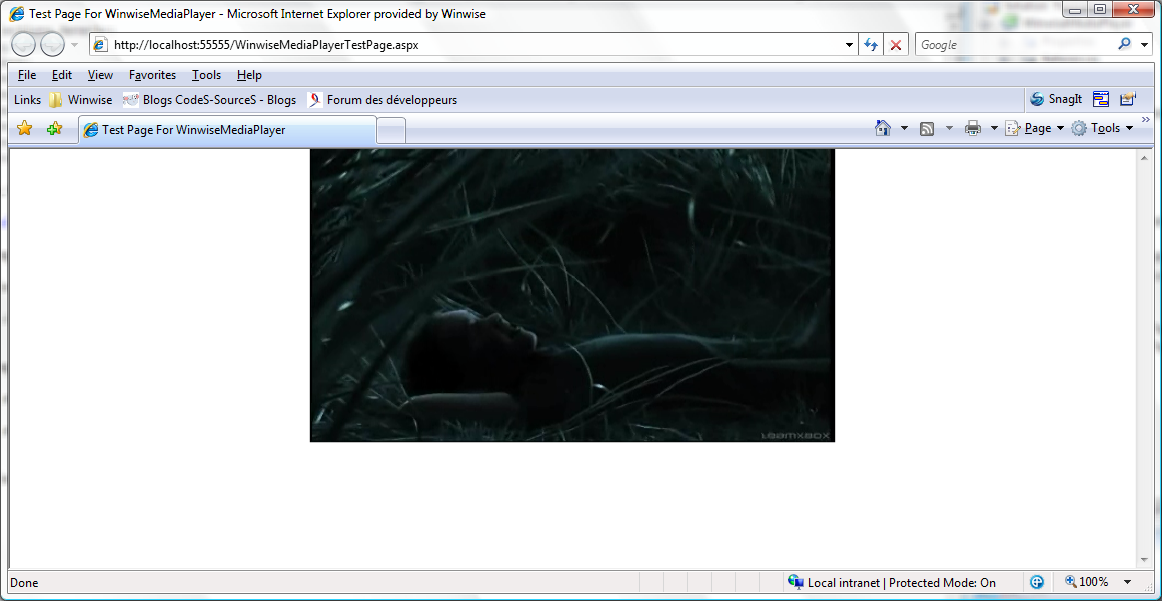
[11ème Partie : .Net RIA Services 73](#_Toc242775597)

# 1ère Partie : Création d’un Media Player

L’objectif de cet atelier est de vous permettre de développer votre premier lecteur multimédia, avec Silverlight. Après avoir vu les contrôles de base, nous verrons ensuite comment faire en sorte de pouvoir ajouter, très simplement, des sous-titres à notre vidéo.

## Création du Média Player de base

Afin de vous faire gagner du temps, nous avons déjà créé une solution que vous allez compléter au fur et à mesure. Ainsi, pour commencer, il vous suffit de double-cliquer sur le raccourci nommé "**Formation Avancée Silverlight**", situé sur le bureau. Une fois Visual Studio ouvert, assurez-vous que "**WinwiseMediaPlayer\_Web**" est bien définit comme projet de démarrage puis appuyez sur **F5** pour vous rendre compte de ce qui a déjà été mis en place. Si tout se passe bien, vous devriez voir un navigateur Internet se lancer, avec une vidéo :



Comme vous pouvez le constater en double-cliquant sur le fichier "**Page.xaml**", le contenu de l’interface graphique est très simple : une grille avec 2 lignes, l’une contenant un objet de type **MediaElement** (c’est l’objet qui nous permet de lire des fichiers audio et/ou vidéo avec Silverlight) et l’autre contenant un *StackPanel* vide :

<Grid x:Name="LayoutRoot" Background="White">

<Grid.RowDefinitions>

<RowDefinition Height="7\*" />

<RowDefinition Height="3\*" />

</Grid.RowDefinitions>

<MediaElement x:Name="media"

HorizontalAlignment="Center"

VerticalAlignment="Center"

Stretch="Uniform" />

<StackPanel Grid.Row="1" HorizontalAlignment="Center" VerticalAlignment="Center">

</StackPanel>

</Grid>

Au niveau du code behind associé à ce fichier XAML, il est lui aussi très simple: au chargement de l’application, nous indiquons simplement (au moyen de la propriété **Source**) quel est le fichier à jouer/utiliser dans le MediaElement :

void OnPageLoaded(object sender, RoutedEventArgs e)

{

this.media.Source = new Uri("http://localhost:55555/Videos/HaloWithSubtitles.wmv", UriKind.Absolute);

}

Vous pouvez constater qu’avec si peu de code, votre MediaPlayer est pour ainsi dire terminé. En effet, il reste encore des petites améliorations à apporter mais, d’un point de vue fonctionnalités (i.e. lire un fichier audio/vidéo), cela fonctionne aussi simplement, sans que vous ayez besoin de faire quoi que ce soit d’autre.

A présent, nous allons modifier légèrement notre interface pour ajouter les contrôles qui vont nous permettre de contrôler ce lecteur multimédia. Ainsi, ouvrez le fichier "*Page.xaml*" et sous la ligne suivante :

<StackPanel Grid.Row="1" HorizontalAlignment="Center" VerticalAlignment="Center">

Ajoutez le code ci-dessous :

<Button x:Name="btPlay"

Content="Play"

Click="btPlay\_Click" />

<Button x:Name="btPause"

Content="Pause"

Click="btPause\_Click" />

<Button x:Name="btStop"

Content="Stop"

Click="btStop\_Click" />

Ensuite, relocalisez la ligne déclarant votre StackPanel, modifiez sa hauteur (mettez là à 25) et changez l’orientation des éléments de façon à les empiler à l’horizontal. La ligne devient donc :

<StackPanel Grid.Row="1" Height="25" HorizontalAlignment="Center" VerticalAlignment="Center" Orientation="Horizontal">

A présent, ouvrez le fichier "**Page.xaml.cs**" et ajoutez le code suivant, qui vous permet de contrôler votre lecteur :

private void btPlay\_Click(object sender, RoutedEventArgs e)

{

this.media.Play();

}

private void btPause\_Click(object sender, RoutedEventArgs e)

{

this.media.Pause();

}

private void btStop\_Click(object sender, RoutedEventArgs e)

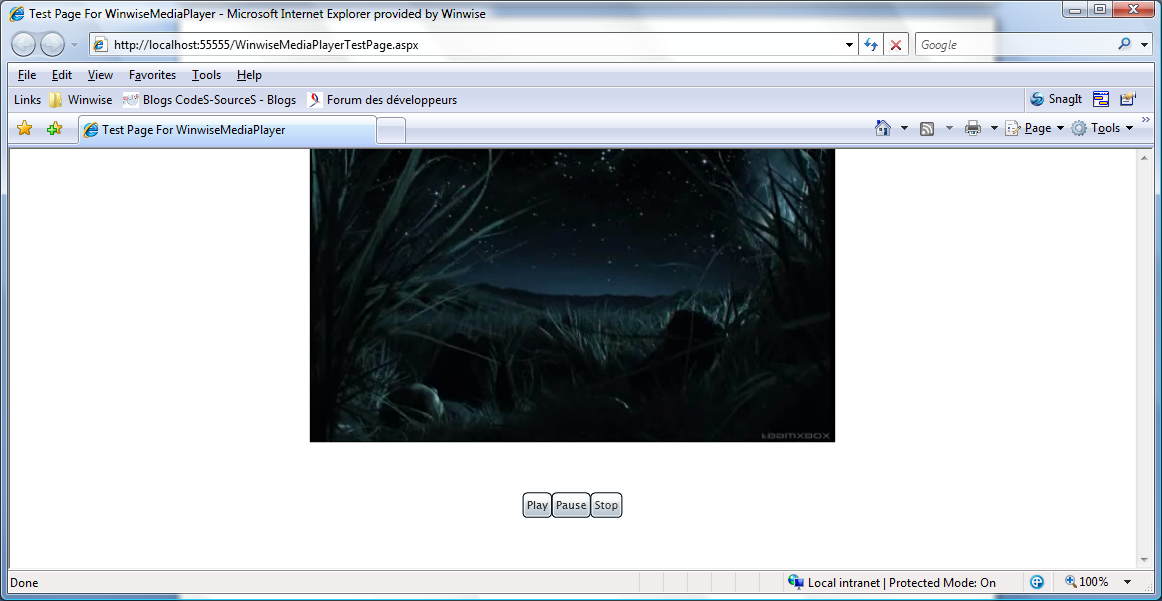
{

this.media.Stop();

}

Comme vous pouvez le constater, contrôler cet objet revient à appeler de simples méthodes (*Play*, *Pause* et *Stop*), tout comme si vous utiliser le lecteur multimédia de Windows.

A l’exécution, vous devriez donc obtenir ceci (avec, bien sûr, des boutons actifs) :



Il serait intéressant d’avoir une barre de progression(ou autre) permettant de savoir à quel endroit, de la vidéo, nous nous trouvons actuellement. Cependant actuellement, Silverlight ne permet pas de faire du Databinding sur d’autres éléments de l’interface graphique (comme cela est possible en WPF). Nous allons donc devoir trouver une parade pour pallier à ce problème.

Ainsi, dans votre fichier XAML, trouver le code suivant :

Click="btStop\_Click" />

Et rajoutez, juste après, cette ligne de code qui se contente de rajouter un TextBlock (équivalent d’un Label) dans l’interface :

<TextBlock x:Name="tb" Margin="5,0,0,0" />

Ensuite, retournez dans le fichier de code ("*Page.xaml.cs*") et, au-dessus de :

public Page()

Rajoutez le code suivant :

private System.Windows.Threading.DispatcherTimer timer;

Nous n’allons pas nous attarder sur le type **DispatcherTimer** pour le moment. Sachez juste qu’il s’agit tout simplement d’un "*timer*" particulier disponible avec WPF et Silverlight.

Après cela, sous la ligne suivante:

this.media.Source = new Uri("http://localhost:55555/Videos/HaloWithSubtitles.wmv", UriKind.Absolute);

Rajoutez ce code d’initialisation et de démarrage du timer:

timer = new System.Windows.Threading.DispatcherTimer();

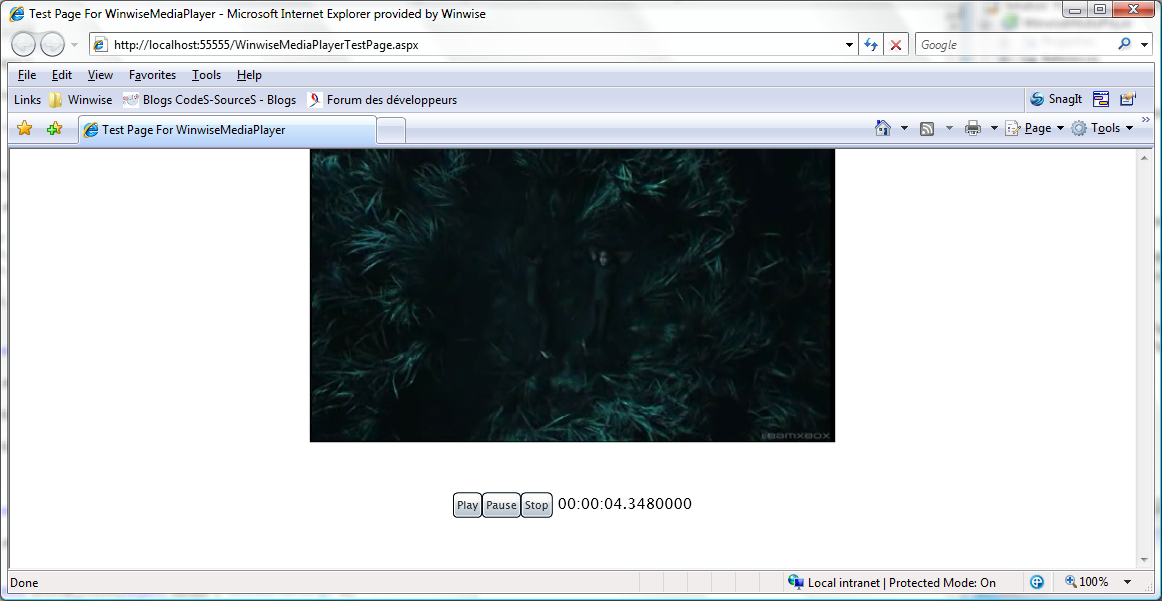
timer.Interval = new TimeSpan(0, 0, 0, 0, 1);

timer.Tick += (s, args) => this.tb.Text = this.media.Position.ToString();

timer.Start();

Note : *Vous noterez au passage l’utilisation des expressions lambda, nouveautés de C# 3, lors de l’abonnement à l’évènement Tick*.

Libre à vous ensuite d’enrichir le reste de l’application avec les tests, réinitialisations et démarrages nécessaires mais pour le moment, si vous lancez l’application (via la touche *F5* de votre clavier), vous devriez être en mesure de voir la position courante sur le fichier vidéo, via une zone de texte :



## Mise en place des sous-titres

La première partie de notre lecteur est à présent terminée. Nous allons maintenant voir comment nous pourrions rajouter des sous-titres sur cette application.

Il y a 2 façons d’implémenter les sous-titres sur une vidéo :

* Ré-encoder la vidéo en intégrant les sous-titres (sous forme de méta-data) : cette technique a l’avantage de permettre d’utiliser peu de code lorsque l’on souhaite accéder au texte mais impacte la taille de la vidéo (une vidéo de 3 Mo passe à environ 5 Mo avec une dizaine de sous-titres embarqués)
* Proposer un fichier de sous-titres à côté de la vidéo (fichier SRT) : la vidéo n’est pas modifiée, la modification est très simple et rapide. Cependant, un peu de code est nécessaire pour accéder au fichier contenant le texte.

### Vidéos encodées avec les sous-titres

Avant de commencer à accéder aux sous-titres, il faut un endroit pour les afficher. Nous allons donc utiliser un **TextBlock** qui contiendra le texte. Ainsi, dans votre fichier XAML, trouvez la ligne suivante :

<TextBlock x:Name="tb" Margin="5,0,0,0" />

Juste en dessous, ajoutez cette ligne:

<TextBlock x:Name="tbSubtitle" Margin="5,0,0,0" />

A présent, il nous faut écrire le code qui nous permettra d’accéder au texte du sous-titre, embarqué dans la vidéo.

Au-dessus de la ligne ci-dessous :

this.media.Source = new Uri("http://localhost:55555/Videos/HaloWithSubtitles.wmv", UriKind.Absolute);

Ajoutez cette ligne:

this.media.MarkerReached += new TimelineMarkerRoutedEventHandler(media\_MarkerReached);

Cette ligne est utilisée pour vous abonner à l’évènement "**MarkerReached**", autrement dit l’évènement qui survient à chaque fois qu’un marqueur est rencontré et remonté par l’objet de type *MediaElement*.

Il ne vous reste plus qu’à écrire la méthode "**media\_MarkerReached**", appelée lors de l’évènement "*MarkerReached*". Trouver les lignes :

timer.Start();

}

Et juste en dessous, rajoutez ce code :

private void media\_MarkerReached(object sender, TimelineMarkerRoutedEventArgs e)

{

if (e.Marker != null)

{

if (!string.IsNullOrEmpty(e.Marker.Text))

{

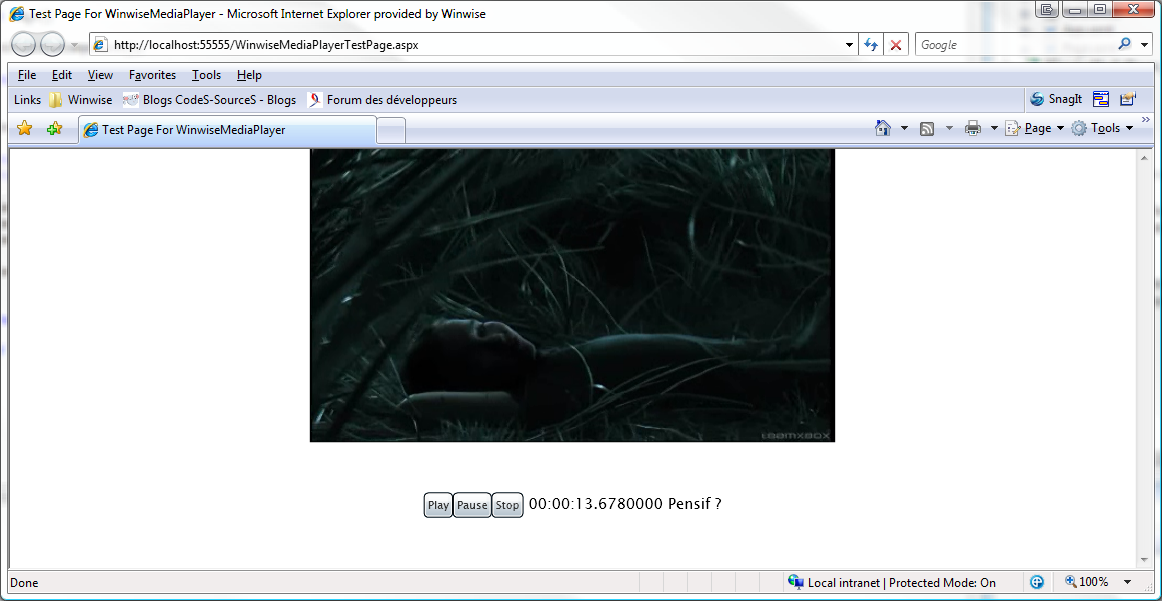
this.tbSubtitle.Text = e.Marker.Text;

}

}

}

Avec ce code, nous vérifions qu’un marqueur est bien présent puis nous accédons à sa propriété *Text*, que nous assignons à notre TextBlock. A l’exécution, l’évènement est bien déclenché à chaque fois qu’un marqueur est rencontré dans la vidéo (le premier se trouve à environ 7 secondes) :



### Vidéos avec fichier de sous-titres

Utiliser un fichier SRT est une technique très connue lorsque l’on a besoin de faire appel à des sous-titres. C’est par exemple utilisé avec les "trailers" ou d’autres types de fichiers vidéo…. Nous allons voir comment mettre en place cette technique dans le cadre de notre lecteur Silverlight.

La première chose à faire est d’indiquer à notre MediaElement quelle sera la nouvelle vidéo à utiliser, celle sans les sous-titres intégrée. Pour cela, remplacez la ligne suivante :

this.media.Source = new Uri("http://localhost:55555/Videos/HaloWithSubtitles.wmv", UriKind.Absolute);

Par celle-ci:

this.media.Source = new Uri("http://localhost:55555/Videos/HaloWithoutSubtitles.wmv", UriKind.Absolute);

Si vous exécutez l’application, vous remarquerez qu’aucun texte de sous-titres n’est affiché et, si vous utilisez les points d’arrêt dans la méthode "*media\_MarkerReached*", vous verrez également que ceux-ci ne sont pas déclenchés : il s’agit donc bel et bien d’une vidéo vide/vierge de tous marqueurs/sous-titres.

Commençons par charger les sous-titres dans l’application. Cette opération doit être faite lors de l’évènement "**MediaOpened**" de votre objet MediaElement car sinon, vous risquez d’être trop tôt ou trop tard.

Ainsi, trouvez la ligne suivante :

this.media.MarkerReached += new TimelineMarkerRoutedEventHandler(media\_MarkerReached);

Et juste en dessous, ajoutez ce code:

this.media.MediaOpened += new RoutedEventHandler(media\_MediaOpened);

Dans la méthode "media\_Opened", nous allons utiliser la classe SrtParser qui va nous fournir des marqueurs à ajouter dynamiquement à la video. Sous les lignes suivantes :

timer.Start();

}

Ajoutez ce code :

private void media\_MediaOpened(object sender, RoutedEventArgs e)

{

var srtParser = new SrtParser(new Uri("http://localhost:55555/SubTitles/SousTitresVideoHalo.srt", UriKind.Absolute));

srtParser.Loaded +=

(s, a) =>

{

foreach (var marker in srtParser.GetMarkers())

media.Markers.Add(marker);

};

srtParser.Load();

}

N’hésitez pas à examiner la classe SrtParser (déclarée dans le projet WinwiseMediaLibrary) pour comprendre comment le fichier SRT est parsé et tranformé en liste de TimelineMarkers.

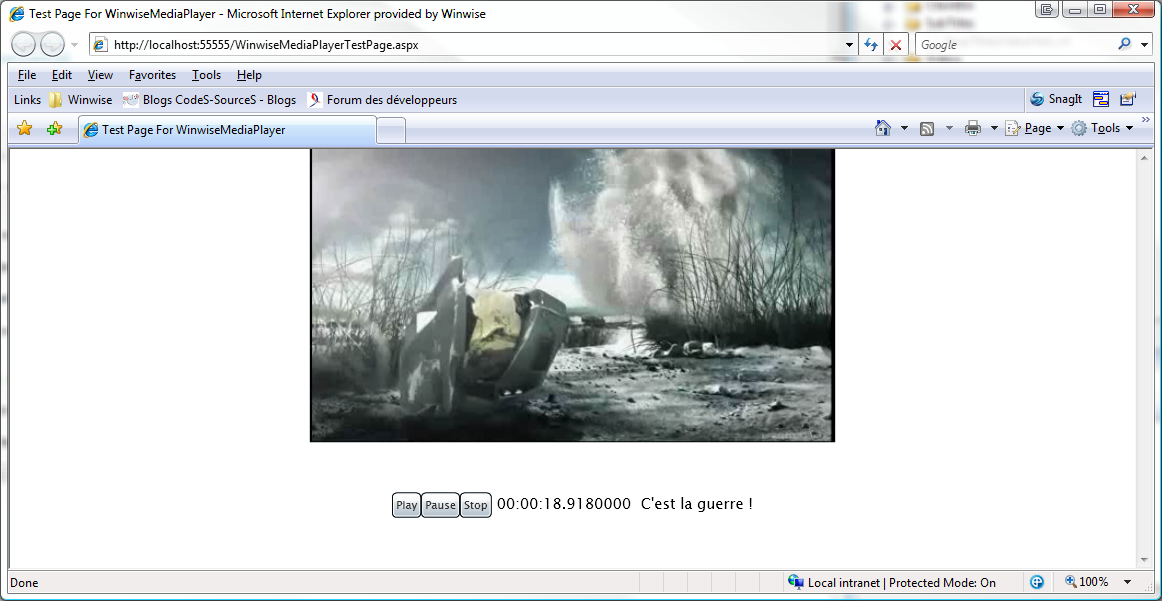
Pour que cela fonctionne sans problèmes, il ne faut pas une dernière petite chose : remplacez la ligne suivante :

if (!string.IsNullOrEmpty(e.Marker.Text))

Par celle-ci:

//if (!string.IsNullOrEmpty(e.Marker.Text))

A l’exécution, vos sous-titres sont téléchargés à l’ouverture de la vidéo et affichés par la suite:

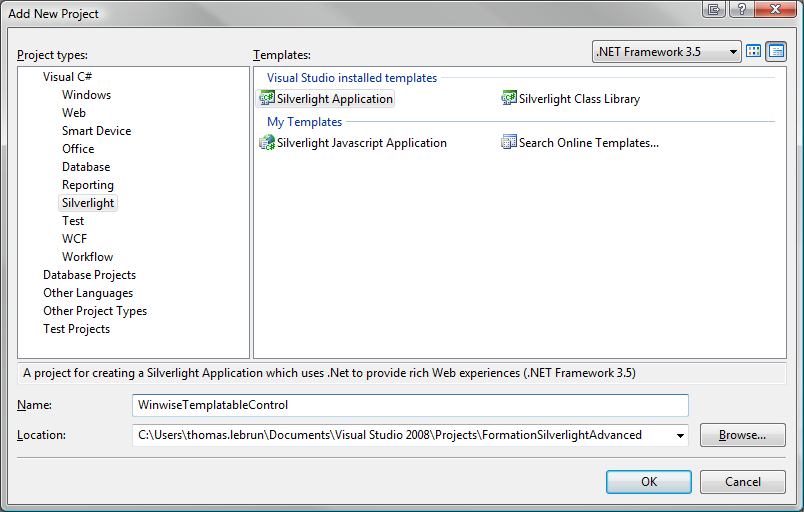
 

Note : Pour encoder une vidéo afin d’y intégrer des sous-titres, il existe plusieurs logiciels de disponibles. Celui de Microsoft se nomme ***Expression Encoder*** et vous permet d’encoder vos vidéos selon différents formats, de spécifier des marqueurs, de faire de l’overlay d’images, etc.….

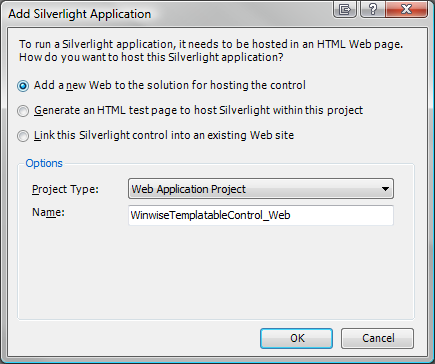
# 2ème Partie : Création d’un contrôle personnalisé

Dans cet atelier, nous allons voir comment créer, de A à Z, un contrôle en définissant l’interface graphique d’un côté et la logique applicative de l’autre.

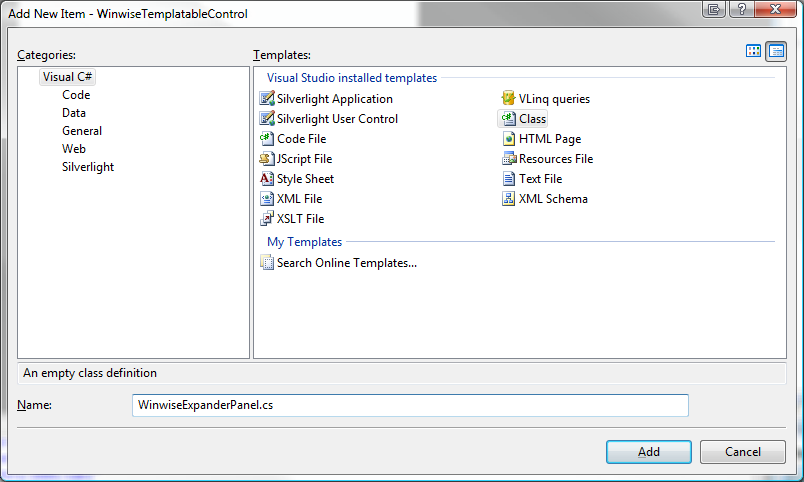
La première chose à faire est de créer un nouveau projet. Pour cela, dans Visual Studio, cliquez sur "*File*" => "*Add*" => "*Project*", sélectionnez "*Silverlight*" puis "*Silverlight Application*". Entrez le nom de "**WinwiseTemplatableControl**" et cliquez sur "OK" :



Dans la fenêtre qui apparait, sélectionnez "*Add a new Web to the solution for hosting the control*", sélectionnez "*Web Application Project*" puis cliquez sur "*OK*" :



Ajoutez une nouvelle classe sur l’application Silverlight : nous appellerons cette classe "**WinwiseExpanderPanel**" :



Ensuite, faîtes hériter cette classe nouvellement créée de la classe *ContentControl*. Votre code doit donc ressembler à ceci :

namespace WinwiseTemplatableControl

{

public class WinwiseExpanderPanel : ContentControl

{

}

}

A présent, nous allons rajouter les différents éléments "*templétisables*", autrement dit les différents éléments que nous allons pouvoir modifier ultérieurement dans le code XAML. Pour cela, nous allons utiliser l’attribut **TemplatePart**. Notre contrôle sera composé des éléments suivants :

* Un élément principal (*RootElement*) de type **FrameworkElement**
* Un élément "entête" (*HeaderElement*) de type **ContentControl**
* Un élément qui contiendra notre contenu principal (*ContentElement*) et qui sera de type **ContentControl**
* Un élément (*ButtonElement*), de type **ToggleButton**, qui pourra enrouler/dérouler le contenu principal
* 2 animations (*OpenAnimation* et *CloseAnimation*) de types **Storyboard**

Nous allons donc commencer par définir un certain nombre de constantes que nous utiliserons pour nommer les différents « TemplateParts » :

public class WinwiseExpanderPanel : ContentControl

{

private const string ButtonElement = "ButtonElement";

private const string Opened = "Opened";

private const string Closed = "Closed";

…

}

On pourra alors rajouter les attributs suivants à la classe WinwiseExpanderPanel :

[TemplatePart(Name = WinwiseExpanderPanel.ButtonElement, Type = typeof(ToggleButton))]

[TemplateVisualStateAttribute(Name = WinwiseExpanderPanel.Opened, GroupName = "CommonStates")]

[TemplateVisualStateAttribute(Name = WinwiseExpanderPanel.Closed, GroupName = "CommonStates")]

Une fois le contrôle chargé, son interface graphique est remplie à partir de son template XAML. Une fois le template graphique appliqué et chargé, la méthode **OnApplyTemplate** est appelée : nous allons mettre en place le code permettant de référencer les éléments graphiques définis en tant que TemplateParts dans cette méthode (que nous allons surcharger).

Sous la ligne suivante :

private const string Closed = "Closed";

Rajoutez ces lignes de code:

ToggleButton mButtonElement = null;

public WinwiseExpanderPanel()

: base()

{

DefaultStyleKey = typeof(WinwiseExpanderPanel);

}

public override void OnApplyTemplate()

{

base.OnApplyTemplate();

mButtonElement = this.GetTemplateChild(WinwiseExpanderPanel.ButtonElement) as ToggleButton;

// ChangeState();

}

Si vous n’arrivez pas à comprendre ce code, dîtes-vous simplement qu’il permet de stocker, dans des variables membres, une référence vers les contrôles présents dans le template.

A présent, nous allons créer 2 propriétés :

* Une propriété **Header**, qui contiendra ce que nous mettrons dans l’entête de notre contrôle
* Une propriété **IsOpened**, qui nous permettra de savoir si notre contrôle est ouvert ou fermé

Vous remarquez que nous ne créons pas de propriété pour le corps/contenu de notre contrôle. En effet, celui-ci héritant de *ContentControl*, il possède déjà une propriété *Content* !

Afin de pouvoir appliquer du binding, des styles, des templates, etc.… sur les 2 propriétés que nous allons créer, nous n’allons pas faire des propriétés .NET classiques mais plutôt des "**Dependency Property**" (qui sont utilisables par le moteur Silverlight dans le cas de binding ou autre).

Pour cela, sous la ligne suivante :

ToggleButton mButtonElement = null;

Ajoutez ce code :

public object Header

{

get { return (object)GetValue(HeaderProperty); }

set { SetValue(HeaderProperty, value); }

}

public static readonly DependencyProperty HeaderProperty =

DependencyProperty.Register("Header", typeof(object), typeof(WinwiseExpanderPanel), null);

public static readonly DependencyProperty IsOpenedProperty =

DependencyProperty.Register("IsOpened", typeof(bool), typeof(WinwiseExpanderPanel), new PropertyMetadata(new PropertyChangedCallback(OnIsOpenedPropertyChanged)));

public bool IsOpened

{

get

{

return (bool)GetValue(IsOpenedProperty);

}

set

{

SetValue(IsOpenedProperty, value);

}

}

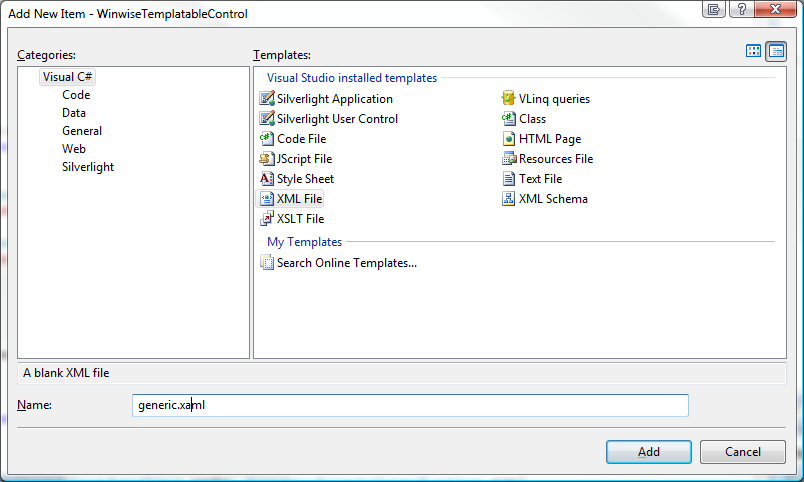
private static void OnIsOpenedPropertyChanged(DependencyObject sender, DependencyPropertyChangedEventArgs args)

{

}

Seule la deuxième propriété est un peu plus complexe car elle définit une "callback" (nommée *OnIsOpenedPropertyChanged*), autrement dit une méthode à appeler à chaque fois qu’elle est modifiée.

A présent, nous allons nous attaquer à définir l’interface de notre contrôle. Pour cela, tous les templates de contrôles personnalisés doivent se trouver dans un *ResourceDictionnary* nommé "**generic.xaml**" : il vous faut donc l’ajouter à votre projet, dans un répertoire nommé **Themes** (qu’il vous faut créer), et le marquer comme étant une ressource de l’application :





A présent, remplacez le contenu de ce fichier par celui-ci, qui va vous permettre de tester votre contrôle :

<ResourceDictionary

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

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

xmlns:Winwise="clr-namespace:WinwiseTemplatableControl;assembly=WinwiseTemplatableControl">

<Style TargetType="Winwise:WinwiseExpanderPanel">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Winwise:WinwiseExpanderPanel">

<Grid x:Name="RootElement">

<Grid.RowDefinitions>

<RowDefinition Height="25"/>

<RowDefinition Height="auto"/>

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="auto" />

<ColumnDefinition Width="25" />

</Grid.ColumnDefinitions>

<ContentControl Grid.Column="0"

Grid.Row="0"

x:Name="HeaderElement"

HorizontalAlignment="Left"

VerticalAlignment="Center"

Content="{TemplateBinding Header}"/>

<ToggleButton Grid.Row="0"

Grid.Column="1"

HorizontalAlignment="Center"

x:Name="ButtonElement"

Content="X" />

<ContentControl Grid.Row="1"

Grid.ColumnSpan="2"

x:Name="ContentElement"

Content="{TemplateBinding Content}" />

</Grid>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</ResourceDictionary>

Dans ce fichier, nous ne faisons que définir le **Template**, autrement dit la représentation graphique, de notre contrôle. Notez au passage l’utilisation de **TemplateBinding** qui nous permet de faire du binding entre ce que nous définissons dans notre interface graphique (*Page.xaml*) et ce que nous mettons dans le template.

Une fois le template de test mis en place, il nous reste à définir l’interface graphique. Ouvrez le fichier "*Page.xaml*" et, sous la ligne :

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

Ajoutez cet extrait de code :

xmlns:Winwise="clr-namespace:WinwiseTemplatableControl;assembly=WinwiseTemplatableControl"

Ensuite, remplacez le contenu du contrôle utilisateur par celui-ci:

<Winwise:WinwiseExpanderPanel x:Name="PanelExp" Header="Entête">

<Winwise:WinwiseExpanderPanel.Content>

<Grid Background="Red">

<TextBlock Text="Contenu" />

</Grid>

</Winwise:WinwiseExpanderPanel.Content>

</Winwise:WinwiseExpanderPanel>

Maintenant, nous allons nous attaquer à ajouter le code permettant de modifier le visuel de notre contrôle, en fonction de son état. Commencez par modifier le code de la méthode OnIsOpenedPropertyChanged pour modifier l’état du ToggleButton :

private static void OnIsOpenedPropertyChanged(DependencyObject sender, DependencyPropertyChangedEventArgs args)

{

var panel = sender as WinwiseExpanderPanel;

if (panel == null || panel.mButtonElement == null)

{

return;

}

panel.mButtonElement.IsChecked = (bool)args.NewValue;

// panel.ChangeState() ;

}

Puis, celui de la méthode OnApplyTemplate : (ajoutez ce code)

if (mButtonElement != null)

{

mButtonElement.Checked += (s, args) => IsOpened = true;

mButtonElement.Unchecked += (s, args) => IsOpened = false;

}

Vous pouvez remarquer dans la méthode OnApplyTemplate et dans OnIsOpenedPropertyChanged un appel commenté à la méthode ChangeState. Cette méthode va nous permettre de contrôler le « Visual State » courant du contrôle en utilisant la classe VisualStateManager. Implémentez la méthode suivante et dé-commentez ces appels :

private void ChangeState()

{

if (IsOpened)

{

VisualStateManager.GoToState(this, WinwiseExpanderPanel.Opened, true);

}

else

{

VisualStateManager.GoToState(this, WinwiseExpanderPanel.Closed, true);

}

}

Et voilà, toute la partie "logique" du contrôle est terminée. Il ne nous reste plus qu’à :

* Définir le template final du contrôle
* Indiquer de quoi ce contrôle doit-être composé.

Pour la première étape, rien de plus simple : dans le fichier "**generic.xaml**", remplacez le contenu du fichier par celui-ci (nous ne nous attarderons pas sur le contenu de ce fichier étant donné qu’il s’agit principalement de design et que sa logique a déjà été expliquée précédemment. Pour plus de facilité, vous trouverez le contenu ci-dessous dans le fichier « c:\templateexpander.xaml ») :

<ResourceDictionary

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

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

xmlns:vsm="clr-namespace:System.Windows;assembly=System.Windows"

xmlns:Winwise="clr-namespace:WinwiseTemplatableControl;assembly=WinwiseTemplatableControl">

<Style TargetType="Winwise:WinwiseExpanderPanel">

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="Winwise:WinwiseExpanderPanel">

<Grid x:Name="RootElement">

<vsm:VisualStateManager.VisualStateGroups>

<vsm:VisualStateGroup x:Name="CommonStates">

<vsm:VisualState x:Name="Opened">

<Storyboard>

<DoubleAnimationUsingKeyFrames Storyboard.TargetName="ContentParent" Storyboard.TargetProperty="(UIElement.RenderTransform).ScaleY">

<SplineDoubleKeyFrame KeyTime="0:0:0.5" Value="1.0"/>

</DoubleAnimationUsingKeyFrames>

</Storyboard>

</vsm:VisualState>

<vsm:VisualState x:Name="Closed">

<Storyboard>

<DoubleAnimationUsingKeyFrames Storyboard.TargetName="ContentParent" Storyboard.TargetProperty="(UIElement.RenderTransform).ScaleY">

<SplineDoubleKeyFrame KeyTime="0:0:0.5" Value="0"/>

</DoubleAnimationUsingKeyFrames>

</Storyboard>

</vsm:VisualState>

</vsm:VisualStateGroup>

</vsm:VisualStateManager.VisualStateGroups>

<Grid.RowDefinitions>

<RowDefinition Height="25"/>

<RowDefinition Height="auto"/>

</Grid.RowDefinitions>

<Grid Grid.Row="0" Background="#FF607DAF">

<Grid.ColumnDefinitions>

<ColumnDefinition Width="\*"/>

<ColumnDefinition Width="25"/>

</Grid.ColumnDefinitions>

<ContentControl x:Name="HeaderElement" HorizontalAlignment="Left"

Margin="5,0,0,0"

VerticalAlignment="Center"

Foreground="White" FontFamily="Verdana"

FontSize="13" FontWeight="Bold"

Content="{TemplateBinding Header}"/>

<ToggleButton Grid.Column="1" HorizontalAlignment="Center" x:Name="ButtonElement">

<ToggleButton.Template>

<ControlTemplate TargetType="ToggleButton">

<Grid x:Name="RootElement">

<vsm:VisualStateManager.VisualStateGroups>

<vsm:VisualStateGroup x:Name="CheckStates">

<vsm:VisualStateGroup.Transitions>

<vsm:VisualTransition GeneratedDuration="00:00:0.5" To="Checked"/>

<vsm:VisualTransition GeneratedDuration="00:00:0.5" To="Unchecked"/>

</vsm:VisualStateGroup.Transitions>

<vsm:VisualState x:Name="Checked">

<Storyboard>

<DoubleAnimationUsingKeyFrames Storyboard.TargetName="BorderButton"

Storyboard.TargetProperty="(Border.RenderTransform).Angle">

<SplineDoubleKeyFrame KeyTime="00:00:00" Value="180" />

</DoubleAnimationUsingKeyFrames>

</Storyboard>

</vsm:VisualState>

<vsm:VisualState x:Name="Unchecked">

<Storyboard>

<DoubleAnimationUsingKeyFrames Storyboard.TargetName="BorderButton"

Storyboard.TargetProperty="(Border.RenderTransform).Angle">

<SplineDoubleKeyFrame KeyTime="00:00:00" Value="360" />

</DoubleAnimationUsingKeyFrames>

</Storyboard>

</vsm:VisualState>

</vsm:VisualStateGroup>

</vsm:VisualStateManager.VisualStateGroups>

<Border x:Name="BorderButton" Width="15" Height="15" Background="#FF314977"

BorderThickness="1" BorderBrush="White" CornerRadius="15">

<Border.RenderTransform>

<RotateTransform CenterX="7.5" CenterY="7.5" Angle="0" />

</Border.RenderTransform>

<Canvas>

<Line X1="2" Y1="6" X2="6.5" Y2="2" Stroke="White" StrokeThickness="1" />

<Line X1="6.5" Y1="2" X2="11" Y2="6" Stroke="White" StrokeThickness="1" />

<Line X1="2" Y1="10" X2="6.5" Y2="6" Stroke="White" StrokeThickness="1" />

<Line X1="6.5" Y1="6" X2="11" Y2="10" Stroke="White" StrokeThickness="1" />

</Canvas>

</Border>

</Grid>

</ControlTemplate>

</ToggleButton.Template>

</ToggleButton>

</Grid>

<Grid x:Name="ContentParent" Grid.Row="1" Background="#FFF5F5FF">

<Grid.RenderTransform>

<ScaleTransform />

</Grid.RenderTransform>

<ContentControl x:Name="ContentElement"

Content="{TemplateBinding Content}"

HorizontalAlignment="Center">

<ContentControl.RenderTransform>

<ScaleTransform ScaleY="1" ScaleX="1" />

</ContentControl.RenderTransform>

</ContentControl>

</Grid>

</Grid>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</ResourceDictionary>

Pour la deuxième étape (définir de quoi est composé le contrôle), là encore la tâche est extrêmement simple : dans le fichier *Page.xaml*, remplacez ce bloc de code :

<Winwise:WinwiseExpanderPanel x:Name="PanelExp" Header="Entête">

<Winwise:WinwiseExpanderPanel.Content>

<Grid Background="Red">

<TextBlock Text="Contenu" />

</Grid>

</Winwise:WinwiseExpanderPanel.Content>

</Winwise:WinwiseExpanderPanel>

Par celui-là :

<Winwise:WinwiseExpanderPanel x:Name="PanelExp" Header="Cliquez sur les flèches pour ouvrir le volet" IsOpened="True">

<Winwise:WinwiseExpanderPanel.Content>

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="auto"/>

<RowDefinition Height="auto"/>

</Grid.RowDefinitions>

<TextBlock VerticalAlignment="Center" HorizontalAlignment="Center"

FontFamily="Verdana" FontSize="13" FontWeight="Bold" Margin="30,0,30,25">

<Run Text="Winwise - Formation 'Approfondissements de Silverlight'" />

</TextBlock>

<TextBlock VerticalAlignment="Center" HorizontalAlignment="Center"

FontFamily="Verdana" FontSize="13" FontWeight="Bold" Margin="30,30,30,10">

<Run Text="Atelier 'Création de contrôles personnalisés'" />

</TextBlock>

<Grid Grid.Row="1" Background="#FF3E5C92">

<HyperlinkButton Grid.Row="1" Content="Visiter le site Web de Winwise" NavigateUri="http://www.winwise.fr"

HorizontalAlignment="Right" Margin="0,5,5,5"

FontFamily="Verdana" FontSize="11" Foreground="#FFAEE0D5"/>

</Grid>

</Grid>

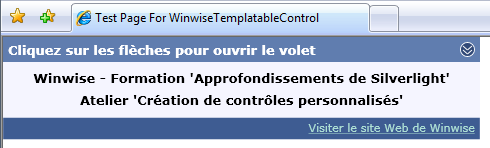
</Winwise:WinwiseExpanderPanel.Content>

</Winwise:WinwiseExpanderPanel>

Et voila ! A l’exécution, vous devriez donc voir votre contrôle apparaitre à l’écran :



Et si vous cliquez sur les flèches, les animations d’ouverture et de fermeture sont correctement déclenchées :



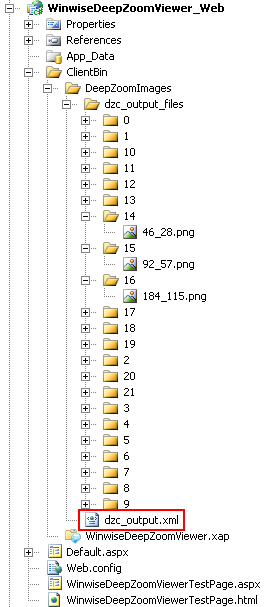
# 3ème Partie : Interaction DOM / Code Managé

Au cours de cet atelier, nous allons voir comment il est possible, depuis du code JavaScript, d’accéder et de manipuler du code managé. Bien sur, l’inverse est tout à fait possible (accéder au code JavaScript ou au code DOM depuis le code managé) mais suffisament simple pour ne pas être couvert dans cette partie.

Au cours de cet atelier, nous allons manipuler **DeepZoom**, une technologie qui a été ajoutée à Silverlight et qui permet de disposer d'un niveau de zoom sans perte de netteté. La technique consiste à utiliser plusieurs images conçues avec des niveaux de zoom différents, et à laisser Silverlight lisser le mouvement pour faire croire qu'on reste sur la même image.

Avec Silverlight, cette technologie est implémentée dans l’objet **MultiScaleImage** qui permet de contrôler l’agrandissement, la réduction et le zoom de vos images, comme vous allez le voir par la suite.

Dans la solution Visual Studio, assurez-vous que "**WinwiseDeepZoomViewer\_Web**" est bien définit comme projet de démarrage (clic droit sur le nom du projet => "*Set as StartUp Project*"). Ensuite, observez le contenu du projet et plus précisément le contenu du répertoire "**ClientBin**" : vous devriez y voir le fichier "*xap*", correspondant à l’application Silverlight, et un autre répertoire (nommé "*DeepZoomImages*") contenant des images et un fichier « *dzc\_output.xml* » :



C’est ce fichier, qui contient des métadonnées sur les images, que nous allons passer en paramètres à notre objet de type *MultiScaleImage*.

Ouvrez alors le fichier "*Page.xaml*" du projet "**WinwiseDeepZoomViewer**" et, sous la ligne :

<Grid x:Name="LayoutRoot" Background="White">

Ajoutez le code suivant:

<MultiScaleImage Source="DeepZoomImages/dzc\_output.xml" Height="300" Width="400" />

Si vous exécutez l’application, vous remarquerez que les images sont correctement chargées (avec une animation au démarrage) sans que vous ayez besoin de faire autre chose :



Vous remarquez cependant qu’il n’y a aucune interaction possible avec la souris. Or, pour une technologie censée proposer des niveaux de zoom sans perte de netteté, cela semble un peu nécessaire : nous allons donc remédier à ce problème tout de suite.

## Ajouter la gestion du "Drag &Drop"

Avant toute chose, nous allons agrandir la taille de l’application Silverlight et de notre composant. Pour cela, remplacez la ligne :

Width="400" Height="300">

Par :

Width="640" Height="480">

Puis remplacez :

<MultiScaleImage Source="DeepZoomImages/dzc\_output.xml"

Height="300" Width="400" />

Par:

<MultiScaleImage Source="DeepZoomImages/dzc\_output.xml"

Height="480" Width="640" />

A présent, nous allons nous abonner aux évènements *MouseLeftButtonDown*, *MouseLeftButtonUp* et *MouseMove* de notre objet MultiScaleImage, afin de pouvoir gérer le glisser/déplacer. Afin d’y arriver, localisez le code suivant :

<MultiScaleImage Source="DeepZoomImages/dzc\_output.xml"

Et ajoutez, juste en dessous, ces instructions :

x:Name="deepZoomObject"

MouseLeftButtonUp="MultiScaleImage\_MouseLeftButtonUp"

MouseLeftButtonDown="MultiScaleImage\_MouseLeftButtonDown"

MouseMove="MultiScaleImage\_MouseMove"

Maintenant, il nous faut aller dans le code behind pour écrire le code des méthodes appelées lors de ces évènements. Pour cela, rien de plus simple : ouvrez le fichier "*Page.xaml.cs*" et au dessus de la ligne suivante :

public Page()

Ajoutez ce code :

private bool dragInProgress = false;

private Point dragOffset;

private Point currentPosition;

Puis, sous la ligne :

InitializeComponent();

}

Ajoutez ceci :

private void MultiScaleImage\_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)

{

dragInProgress = false;

dragOffset = new Point();

currentPosition = new Point();

}

private void MultiScaleImage\_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)

{

dragInProgress = true;

dragOffset = e.GetPosition(this);

currentPosition = deepZoomObject.ViewportOrigin;

}

private void MultiScaleImage\_MouseMove(object sender, MouseEventArgs e)

{

if (dragInProgress)

{

Point newOrigin = new Point();

newOrigin.X = currentPosition.X - (((e.GetPosition(deepZoomObject).X - dragOffset.X) / deepZoomObject.ActualWidth) \* deepZoomObject.ViewportWidth);

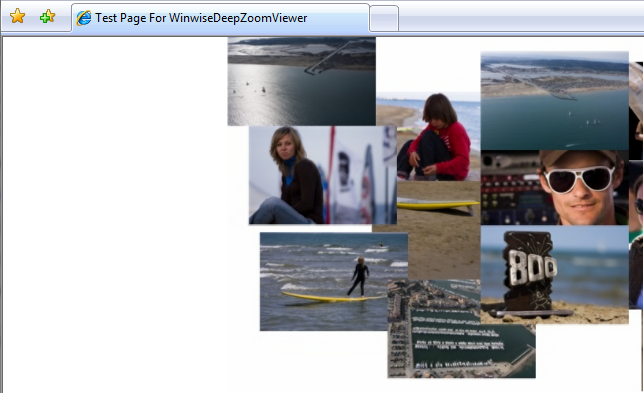
newOrigin.Y = currentPosition.Y - (((e.GetPosition(deepZoomObject).Y - dragOffset.Y) / deepZoomObject.ActualHeight) \* deepZoomObject.ViewportWidth);

deepZoomObject.ViewportOrigin = newOrigin;

}

}

Comme vous pouvez le constater, nous ne faisons rien d’extraordinaire dans cette partie mis à part déplacer le point d’origine de notre objet *MultiScaleImage*. A l’exécution, vous pouvez voir que vous avez alors la possibilité de glisser/déposer cet objet, via la souris :



## Ajouter la gestion du zoom

Nous allons à présent ajouter la gestion du zoom sur les images, au sein de notre application. Pour cela, nous allons utiliser la molette de la souris. Cependant, l’un des problèmes actuel de Silverlight (dans sa partie managée) est qu’il n’est pas capable de gérer le "**Mouse Wheel**", autrement dit le défilement avec la souris. Pour pallier à ce problème, il y a 2 options :

* Utiliser le JavaScript, car il s’agit d’une fonctionnalité que l’on peut gérer via ce langage
* Faire un pont entre le navigateur et Silverlight : l’évènement est intercepté par le navigateur qui informe .NET/Silverlight qui se charge d’appeler le code managé nécessaire.

C’est cette deuxième technique que nous allons mettre en œuvre et, pour commencer, la première chose à faire est, dans votre fichier de code behind, de trouver la ligne :

using System.Windows.Shapes;

Et de rajouter juste après :

using System.Windows.Browser;

Nous allons en effet utiliser des classes et méthodes qui font partis de cet espace de noms : il convient donc de le rajouter/l’importer afin d’éviter des erreurs éventuelles par la suite.

Maintenant, l’appel de méthode suivant:

InitializeComponent();

Rajoutez cette ligne de code :

HtmlPage.RegisterScriptableObject("WWDZClient", this);

Cette ligne est utilisée pour faire en sorte que toutes les méthodes/propriétés/évènements marqués avec l’attribut **ScriptableMember** seront accessibles en JavaScript.

A présent, il va nous falloir capturer l’évènement *MouseWheel* de la souris et, comme nous l’avons dit, nous allons faire cela en JavaScript. Nous allons commencer par rajouter un peu de code qui fera en sorte que notre JavaScript s’exécutera lorsque la page sera affichée.

Pour cela, ouvrez le fichier "**WinwiseDeepZoomViewerTestPage.aspx**" et remplacez la ligne :

<body style="height:100%;margin:0;">

Par cette ligne :

<body style="height:100%;margin:0;" onload="HandleLoad();">

Ensuite, au-dessus de la ligne suivante:

<html xmlns="http://www.w3.org/1999/xhtml" style="height:100%;">

Rajoutez le code suivant, qui vous permet de vous abonner à un évènement en JavaScript

<script type="text/javascript">

function HandleLoad()

{

window.onmousewheel = document.onmousewheel = OnMouseWheel;

if (window.addEventListener)

{

window.addEventListener('DOMMouseScroll', OnMouseWheel, false);

}

}

function OnMouseWheel()

{

if(!event)

{

event = window.event;

}

var slPlugin = document.getElementById("slControl");

if(slPlugin != null)

{

slPlugin.content.WWDZClient.DeepZoom\_MouseWheel(event.clientX, event.clientY, event.wheelDelta);

}

}

</script>

Il y a 2 choses intéressantes à noter dans la méthode **OnMouseWheel** (qui est la méthode appelée lorsque l’évènement *MouseWheel* est intercepté) : tout d’abord, on utilise la syntaxe suivante pour récupérer une référence vers notre objet Silverlight :

document.getElementById("slControl");

Ainsi il est obligatoire de donner un identifiant ( « slControl  ») à l’objet HTML « object » dans notre page HTML. Ensuite, vous avez sans doute remarqué cette ligne qui peut vous sembler surprenante :

slPlugin.content.WWDZClient.DeepZoom\_MouseWheel(

event.clientX, event.clientY, event.wheelDelta);

En fait, ici, nous accédons à notre objet Silverlight et, via la propriété **content**, nous accédons à son contenu. Ensuite, nous utilisons l’identifiant que nous avions mis lors de l’appel à la méthode **RegisterScriptableObject** (dans le constructeur du contrôle), ce qui nous permet d’accéder à n’importe quel élément marqué avec l’attribut *ScriptableMember*. Ici, nous essayons d’accéder à une méthode bien spécifique qui prend un certain nombre de paramètres (3 pour être précis). Il va donc nous falloir écrire la méthode **DeepZoom\_MouseWheel** et la marquer, via l’attribut, comme étant visible et accessible par JavaScript.

Pour cela, retournez dans le fichier "*Page.xaml.cs*" et, au-dessus du code suivant :

public Page()

Rajoutez cette méthode :

[ScriptableMember]

public void DeepZoom\_MouseWheel(double x, double y, int delta)

{

double dZoomFactor = 1.33;

if (delta < 0)

{

dZoomFactor = 1 / 1.33;

}

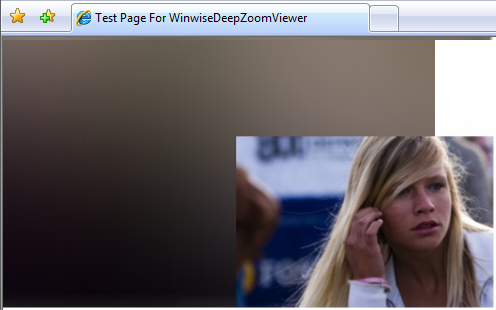
Point pz = deepZoomObject.ElementToLogicalPoint(new Point(x, y));

deepZoomObject.ZoomAboutLogicalPoint(dZoomFactor, pz.X, pz.Y);

}

Si on regarde de plus près cette méthode, on constate que le facteur de zoom, pour chaque défilement avec la molette de la souris, est définit à 33% (mais vous pouvez bien sur le changer). Ensuite, on effectue un zoom au niveau des coordonnées de la souris. Pour cela, on convertit les coordonnées de la souris en un point "logique" via un appel à la méthode **ElementToLogicalPoint**. Enfin, il ne reste plus qu’à zoomer sur ce point (ou plus précisément sur ces coordonnées) via la méthode **ZoomAboutLogicalPoint**.

A l’exécution, vous constatez que vous pouvez glisser/déposer et même zoomer via la molette. Si vous voulez savoir pourquoi cette technologie s’appelle *DeepZoom*, je vous conseille de zoomer (beaucoup !) en bas à droite : vous devriez être en mesure de voir une (toute petite) image incrustée au sein de la scène :

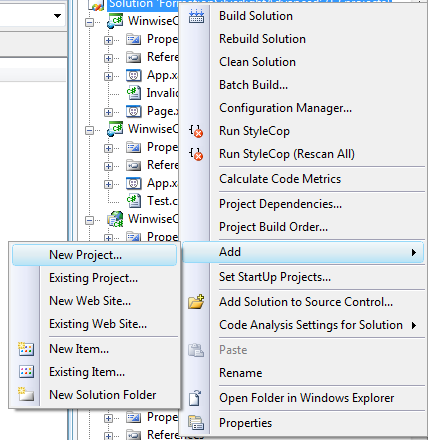


DeepZoom vous permet en effet d’incruster des images dans d’autres images, à des échelles/tailles différentes

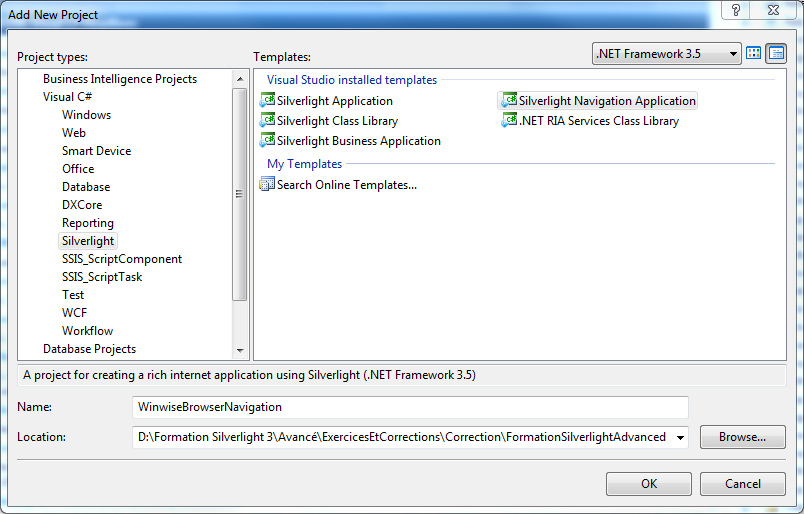
# 4ème Partie : Gestion de la navigation

Au cours de cet atelier, nous allons voir comme implémenter une logique de navigation au sein d’une application Silverlight.

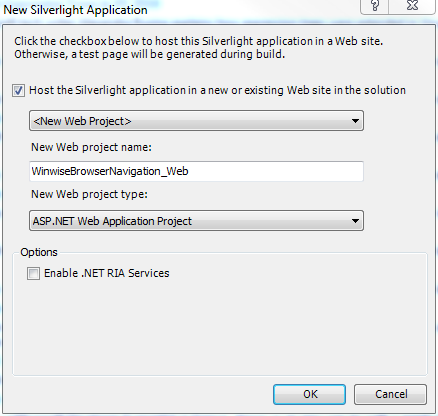
Dans la solution « FormationSilverlightAdvanced », créez un nouveau projet « WinwiseBrowserNavigation ». Pour cela clic-droit sur la solution puis « Add » et enfin « New project » :



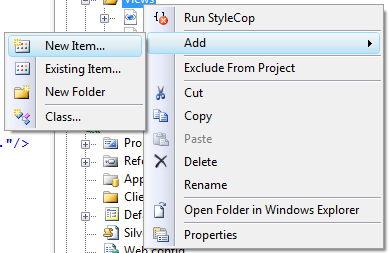
Dans le nœud « Silverlight », sélectionnez l’élément « Silverlight Navigation Application » et donnez comme nom « WinwiseBrowserNavigation » :



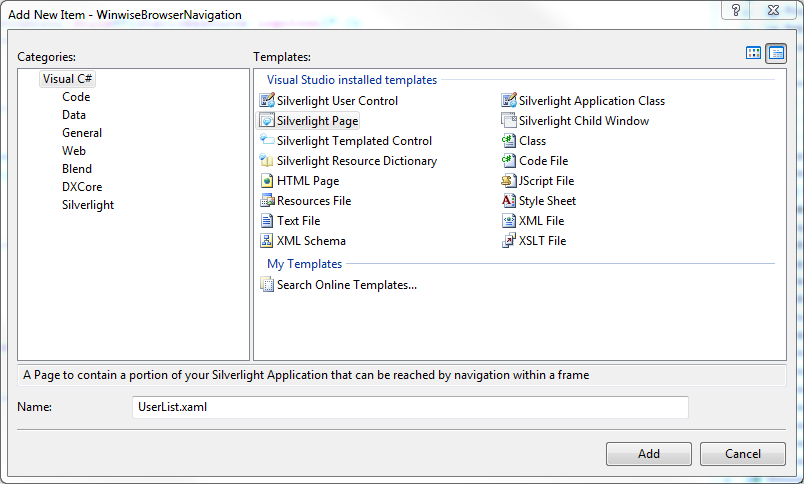
Une nouvelle fenêtre apparait alors pour vous demander où héberger l’application Silverlight. Une application web asp.net convient parfaitement :



Nous avons alors un projet avec différentes vues prédéfinies ainsi qu’une feuille de styles. Nous allons garder les vues en place et en ajoutez une. Ainsi faites un clic-droit sur le répertoire « Views » puis « Add » et enfin « New Item » :



Choisissez l’élément « Silverlight Page » et nommez le « UserList.xaml » :



Maintenant nous allons ajoutez un lien permettant d’accéder à cette page. Pour cela, dans le fichier « MainPage.xaml », ajoutez à l’élément nommé « LinksStackPanel » le séparateur et le lien suivant :

<Rectangle Style="{StaticResource DividerStyle}" />

<HyperlinkButton Style="{StaticResource LinkStyle}"

NavigateUri="/UserList"

TargetName="ContentFrame"

Content="UserList" />

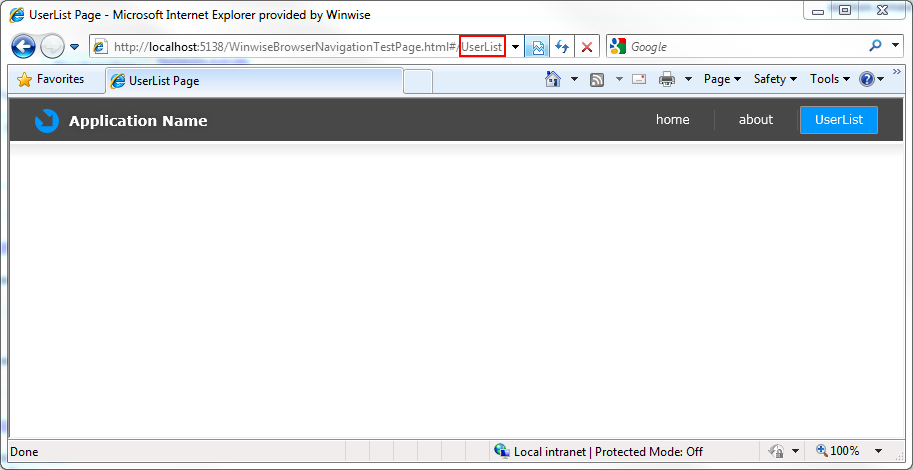
Dans le même fichier, on peut voir que au niveau de l’élément « Frame » qui contient la logique de navigation, le mapping des URI a été fait comme ceci :

<uriMapper:UriMapping Uri="/{pageName}"

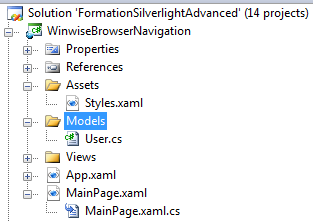
MappedUri="/Views/{pageName}.xaml" />

Ainsi, lorsque l’on va cliquer sur notre HyperlinkButton, nous n’allons pas être redirigés vers « /UserList » comme indiqué au niveau de l’HyperlinkButton mais vers « /Views/UserList.xaml » grâce au système de mapping d’URI.

Vous pouvez tester en lançant le projet que la navigation vers notre nouvelle page fonctionne bien :



Nous allons maintenant ajouter du contenu à notre page. Pour cela nous ajoutez une couche « Models » à notre solution qui représentera notre couche de données. Ajoutez un répertoire au projet « WinwiseBrowserNavigation » et ajoutez une classe « User » dans ce répertoire :



A cette cet objet métier « User », ajoutez différentes propriétés comme ci-dessous :

public string FirstName { get; set; }

public string LastName { get; set; }

public int Age { get; set; }

Nous allons également ajouter une autre classe au fichier « User.cs » nommée « Users ». Elle permettra d’accéder à une liste d’utilisateurs et également d’accéder à un utilisateur en particulier. Commencez par ajouter les 2 using suivant qui nous seront utiles par la suite :

using System.Collections.Generic;

using System.Linq;

Puis écrivons le code de cette classe d’accès aux données :

public static class Users

{

private static List<User> s\_users =

new List<User>

{

new User { Id = 0, Age = 29, FirstName = "Davy", LastName = "Frontigny" },

new User { Id = 1, Age = 26, FirstName = "Thomas", LastName = "Lebrun" },

new User { Id = 2, Age = 25, FirstName = "Simon", LastName = "Ferquel" }

};

public static List<User> GetUsers()

{

return s\_users;

}

public static User GetUser(int id)

{

return s\_users.FirstOrDefault(user => user.Id == id);

}

}

Maintenant retournons à notre page et ajoutons une « ListBox » à notre élément « Grid » qui va être « bindée » à notre liste d’utilisateurs :

<ListBox x:Name="ListBox"

ItemsSource="{Binding}">

<ListBox.ItemTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal">

<TextBlock Text="{Binding FirsstName}" />

<TextBlock Text="{Binding LastName}"

Margin="10,0,0,0" />

</StackPanel>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

Il ne nous reste plus qu’à indiquer la source de donnée de notre « ListBox ». Pour cela, dans le fichier « MainPage.xaml.cs », commençons pas nous abonner à l’évènement « Loaded » de la page, ainsi au niveau du constructeur, ajoutez la ligne suivante :

this.Loaded += new RoutedEventHandler(UserList\_Loaded);

Puis écrivons le code de cette méthode « UserList\_Loaded » :

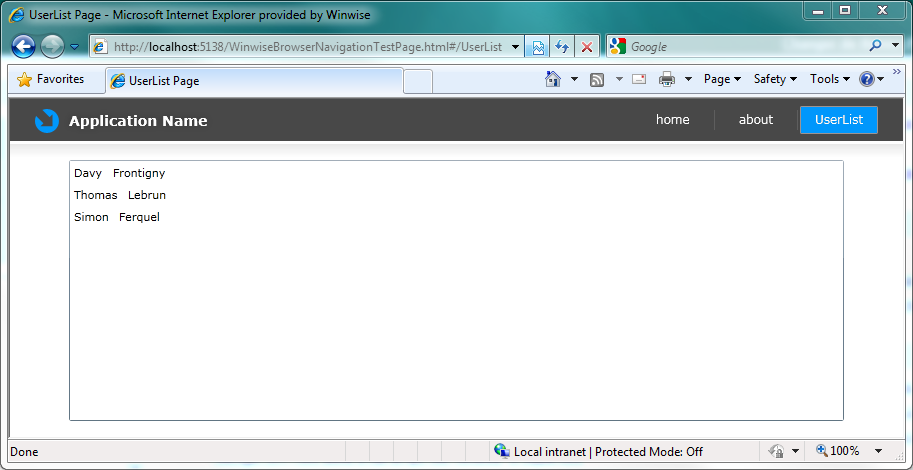
private void UserList\_Loaded(object sender, RoutedEventArgs e)

{

this.ListBox.DataContext = Models.Users.GetUsers();

}

Lancez et constatez que le binding s’est fait sans problème :



Nous allons maintenant mettre en œuvre le système de « Query String » supporté par le framework de Navigation. Nous allons l’utilisé afin de permettre l’accès à la fiche détaillée d’un utilisateur en particulier.

Pour cela, commencez par créer une nouvelle page nommée « UserDetail » en procédant comme auparavant. Cette page va nous permettre d’afficher l’ensemble des informations d’un utilisateur donné.

Définissez la vue comme ci-dessous pour l’affichage des informations utilisateurs :

<Grid x:Name="LayoutRoot" VerticalAlignment="Top" Height="200" >

<Grid.ColumnDefinitions>

<ColumnDefinition />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

<TextBlock Text="Id : " />

<TextBlock Grid.Column="1" Text="{Binding Id}" />

<TextBlock Text="Age : " Grid.Row="1" />

<TextBlock Text="{Binding Age}" Grid.Column="1" Grid.Row="1" />

<TextBlock Text="FirstName : " Grid.Row="2" />

<TextBlock Text="{Binding FirstName}" Grid.Row="2" Grid.Column="1" />

<TextBlock Text="LastName : " Grid.Row="3" />

<TextBlock Text="{Binding LastName}" Grid.Row="3" Grid.Column="1" />

</Grid>

Maintenant nous allons définir la logique à exécuter lorsqu’un utilisateur de la liste est sélectionné. Dans le fichier « UserList.xaml » abonnez à l’évènement « SelectionChanged » de la « ListBox » :

SelectionChanged="ListBox\_SelectionChanged"

Dans la callback de cet évènement nous allons récupérer l’id de l’utilisateur sélectionné puis demandé au système de navigation de naviguer vers une URI prenant en compte cet id :

private void ListBox\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

Models.User selectedUser = ((ListBox)sender).SelectedItem as Models.User;

this.NavigationService.Navigate(new Uri(String.Format("/User/{0}",selectedUser.Id), UriKind.Relative));

}

Donc si l’utilisateur possédant l’id « 1 » est sélectionné, nous sommes redirigé vers la page « /User/1 », le format de cette URI est intuitif et facile à exporter en dehors de l’application Silverlight mais elle correspond à une page qui n’existe pas, il va donc falloir la « mapper ». Dans le fichier « MainPage.xaml », sous la ligne :

<uriMapper:UriMapping Uri="" MappedUri="/Views/Home.xaml" />

Ajoutez la ligne suivante :

<uriMapper:UriMapping Uri="/User/{id}"

MappedUri="/Views/UserDetail.xaml?id={id}" />

Il est important de placer cette règle en 2ème position et non en dernier, sinon ce serait l’autre règle (« Uri="/{pageName}" MappedUri="/Views/{pageName}.xaml" ») qui serait exécutée.

Ainsi l’URI « /User/1 » va être mappé sur « /Views/UserDetail.xaml ?id=1 » qui est une URI valide avec un passage de paramètre correspondant à l’id de l’utilisateur.

Voyons maintenant comment prendre en compte ce paramètre. Dans le fichier « UserDetail.xaml.cs » au niveau de la méthode surchargée « OnNavigatedTo » (méthode appelée lorsque la page en question est atteinte), ajoutez le code suivant :

if (this.NavigationContext.QueryString.ContainsKey("id"))

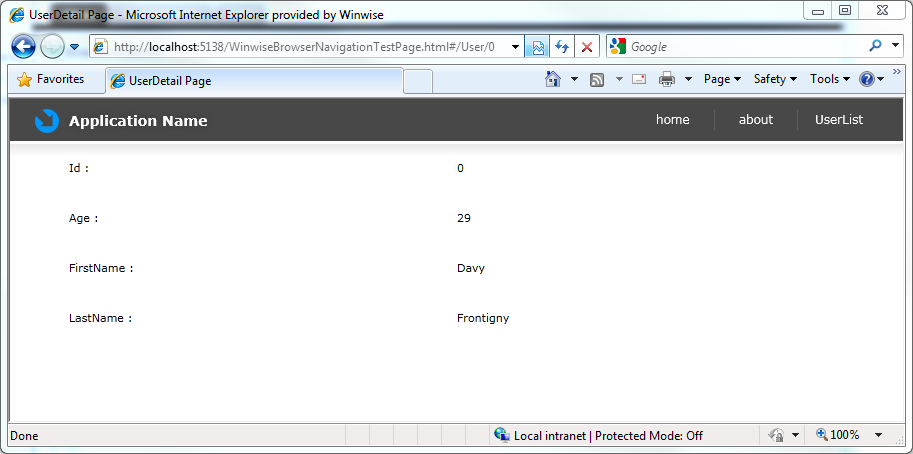
{

this.DataContext = Models.Users.GetUser(Convert.ToInt32(this.NavigationContext.QueryString["id"]));

}

Ainsi nous récupérons l’utilisateur associé à l’id passé en paramètre et nous mettons cet utilisateur en source de données de notre page.

Lancez la solution et sélectionnez un utilisateur :



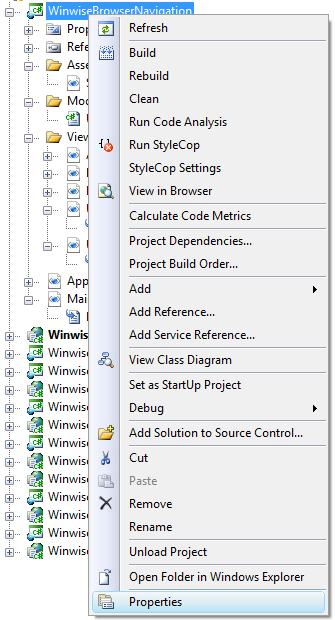
Essayez également les fonctionnalités « Précédent » et « Suivant » du navigateur.

Enfin vous pouvez essayer de copier l’url de la vue détaillée d’un utilisateur puis de fermer votre navigateur, l’ouvrir à nouveau coller l’url et accéder directement à la vue détaillée de l’utilisateur. Ceci est très utile pour permettre l’accès d’une partie d’une application Silverlight à partir d’un site web standard.

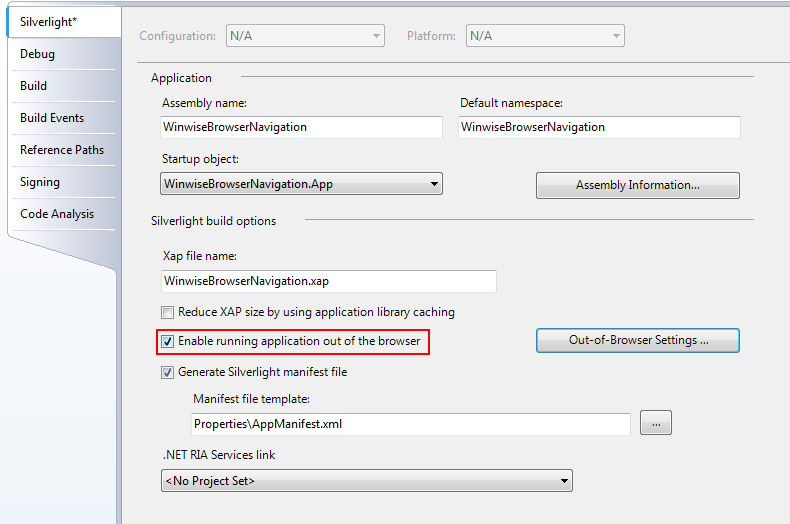
# 5ème Partie : Implémentation du mode Out Of Browser

Au cours de cet atelier nous allons voir comment mettre en œuvre la fonctionnalité « Out Of Browser » de Silverlight. Nous allons travailler sur la même application que pour l’atelier précédent.

La première étape est d’activer la fonctionnalité « Out Of Browser » sur l’application. Pour cela, faites un clic-droit sur le projet « WinwiseBrowserNavigation » puis « Properties » :



Puis cochez la case « Enable running application out of the browser »:



Vous pourrez remarquer que ceci a pour effet de créer un fichier xml dans le répertoire « Properties » nommé « OutOfBrowserSettings ». Ce fichier contient différentes informations de configuration de l’application en mode « Out Of Browser ». Ces informations peuvent être modifiés en cliquant sur le bouton « Out-Of-Browser Settings ».

Maintenant nous allons ajouter un bouton permettant d’installer l’application sur le bureau. Pour cela dans le fichier « MainPage.xaml » ajoutez un bouton en dernier élément de la « Grid » principale :

<Button x:Name="ButtonInstallApplication"

Content="Install application on desktop"

Margin="10"

HorizontalAlignment="Right"

VerticalAlignment="Bottom"

Click="Button\_Click"/>

Pour être notifié des différents changements d’état de l’application nous allons nous abonner à l’évènement « InstallStateChanged» de l’application au niveau du constructeur :

Application.Current.InstallStateChanged += new EventHandler(Current\_InstallStateChanged);

Il nous faut également vérifier cet état au lancement, pour cela abonnez à l’évènement « Loaded » du UserControl au niveau du constructeur :

this.Loaded += new RoutedEventHandler(MainPage\_Loaded);

Nous allons exécuter la même logique dans la callback de ces 2 évènements, ainsi nous allons créer une méthode utilitaire « CheckInstallState » pour prendre en compte l’état de l’application (en mode « Out Of Browser » le bouton d’installation ne doit pas être visible) :

private void CheckInstallState()

{

switch (Application.Current.InstallState)

{

case InstallState.InstallFailed:

this.ButtonInstallApplication.Visibility = Visibility.Visible;

break;

case InstallState.Installed:

this.ButtonInstallApplication.Visibility = Visibility.Collapsed;

break;

case InstallState.Installing:

this.ButtonInstallApplication.Visibility = Visibility.Collapsed;

break;

case InstallState.NotInstalled:

this.ButtonInstallApplication.Visibility = Visibility.Visible;

break;

default:

this.ButtonInstallApplication.Visibility = Visibility.Visible;

break;

}

}

Appelez cette méthode dans la callback des 2 évènements.

Puis nous allons définir la logique à exécuter lorsque l’utilisateur clique sur le bouton, elle est très simple, il suffit d’appeler la méthode statique « Install » de la classe « Application ». Cette méthode renvoie un booléen pour signifier si l’utilisateur accepte l’installation ou non :

private void Button\_Click(object sender, RoutedEventArgs e)

{

if (Application.Current.Install())

{

MessageBox.Show("Application successfully installed", "Application installation", MessageBoxButton.OK);

}

else

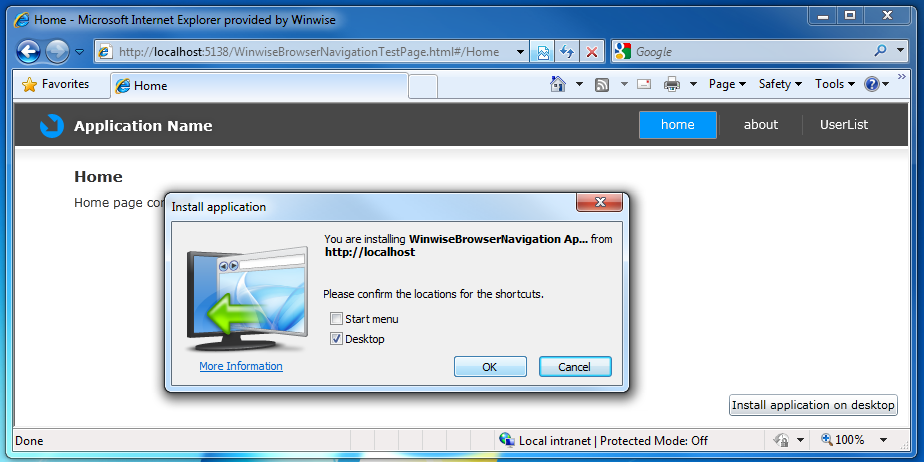
{

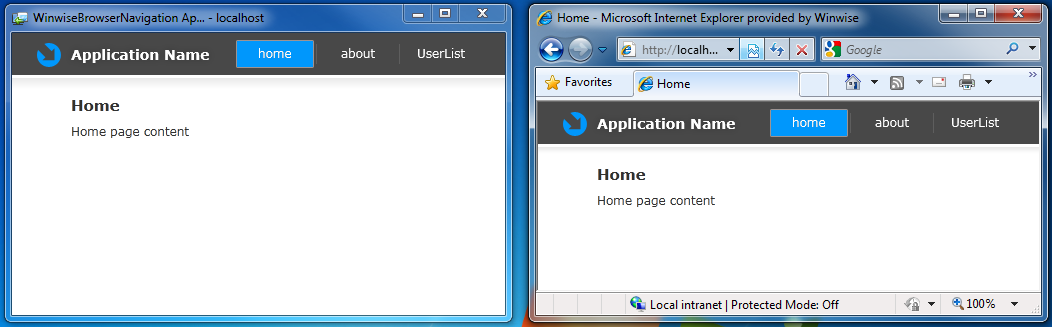
MessageBox.Show("You have cancelled the installation", "Application installation", MessageBoxButton.OK);

}

}

Lancez la solution puis installer l’application et désinstaller là grâce au menu contextuel du bouton droit, vous devez voir le bouton disparaitre puis apparaitre :

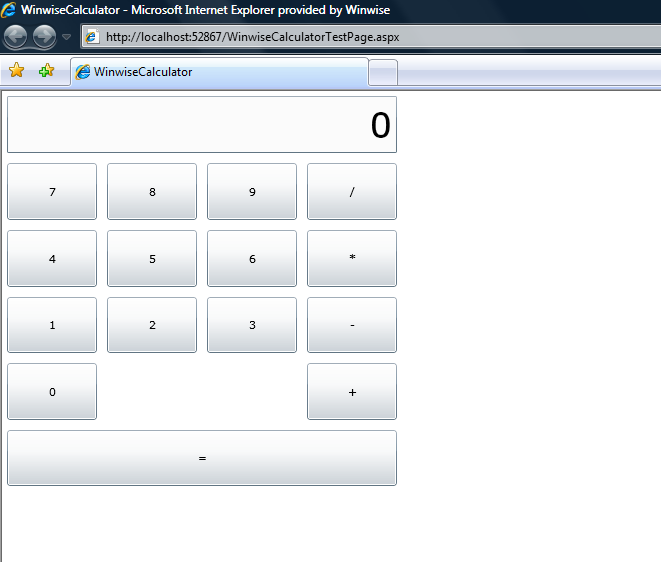




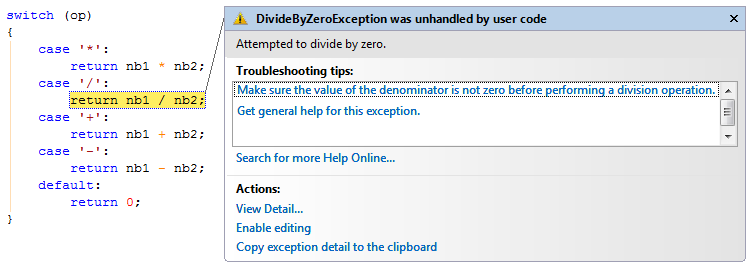
# 6ème Partie : Réalisation de tests unitaires

Au cours de cet atelier nous allons voir comment réaliser des tests unitaires pour une application Silverlight et ce afin de tester des fonctionnalités précises. Afin de se concentrer sur la réalisation de ces tests, nous mettons à votre disposition une application Silverlight déjà faite qui permet d’effectuer des calculs simples et nous allons voir comment créer des tests pour cette dernière.

Pour commencer, double-cliquez sur le raccourci nommé "**Formation Avancée Silverlight**", situé sur le bureau, si la solution « **FormationSilverlightAdvanced** » n’est pas déjà lancée. Dans la solution, vous trouverez un projet nommé « **WinwiseCalculator** ». Les fichiers « **Page.xaml** » et « **Page.xaml.cs** » de ce projet contiennent la définition de l’interface graphique de notre calculatrice et la logique associée, et le fichier « **InvalidIntergerException.cs** » la définition d’une exception personnalisée. Assurez-vous que le projet « **WinwiseCalculator\_Web**» est bien définit comme projet de démarrage puis appuyez sur **F5**. Si tout se passe bien vous devriez voir l’application se lancer dans un navigateur :



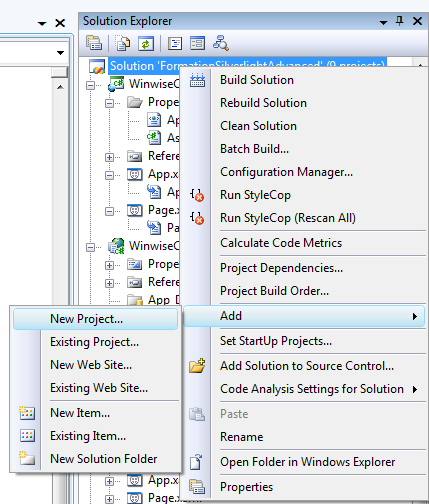
Comme dit précédemment, cette application est une calculatrice simple. Néanmoins, après quelques tests, vous vous rendrez compte que cette calculatrice ne gère pas certains cas particuliers comme par exemple la division par 0 :



Nous allons donc créer des tests unitaires afin de mettre en évidence certains bugs, de les corriger et *in fine* de valider le bon fonctionnement des différentes parties de notre application.

## Création des tests

Faites un clic droit sur la solution « **FormationSilverlightAdvanced** » puis cliquez sur **Add** et **New Project**:



Sélectionnez ensuite **Visual C#** puis dans la section **My Templates** de la partie **Templates** sélectionnez « **Silverlight Test Project** », donnez le nom « **WinwiseCalculator\_Test** » et cliquez sur « **OK** »:



Un nouveau projet est alors ajouté à la solution. Il contient différents fichiers dont un fichier nommé « **Test.cs** ». Ce dernier contient une classe décorée avec l’attribut « **TestClass** », c’est cette classe qui va contenir nos méthodes de test, chacune d’entre elles devra être décorée avec l’attribut « **TestMethod** ». A chaque méthode correspond un test. Nous allons commencer par effectuer des tests simples, c'est-à-dire valider le fonctionnement de l’addition, de la soustraction, de la division et de la multiplication. Pour cela nous allons faire appel à la méthode « **ComputeResult** » qui se trouve dans le fichier « **Page.xaml.c**s » de notre application silverlight « **WinwiseCalculator** », c’est cette méthode qui effectue les opérations arithmétiques. Néanmoins cette méthode est privée et donc inaccessible à l’extérieur de la classe, la solution ici est de passer cette méthode en **internal** et de rendre visible ce type de méthode pour notre assembly de test : « **WinwiseCalculator\_Test** ». Pour cela rajoutez la ligne suivante dans le fichier « **AssemblyInfo.cs** » qui se trouve dans le répertoire « **Properties** » du projet « **WinwiseCalculator** » :

[assembly: InternalsVisibleTo("WinwiseCalculator\_Test")]

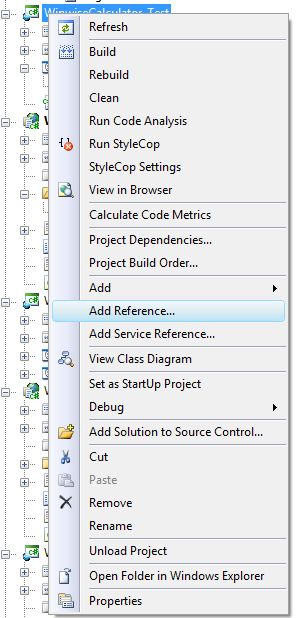
Puis remplacez le modificateur de la méthode « **ComputeResult** » par **internal** :

internal static int ComputeResult(string str1, string str2, char op)

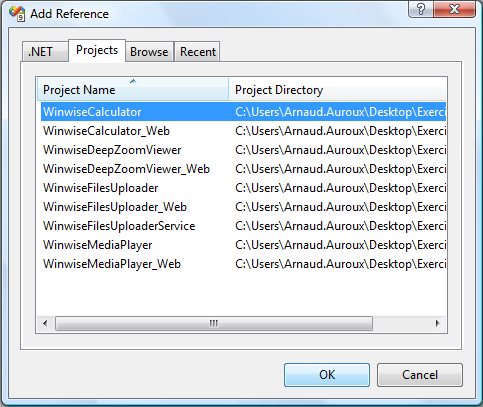
Notre méthode à tester pourra maintenant être visible à partir de notre assembly

« **WinwiseCalculator\_Test** ».

Enfin rajoutez la référence du projet « **WinwiseCalculator »** dans le projet « **WinwiseCalculator\_Test** ». Pour cela, faites un clic droit sur le projet « « **WinwiseCalculator\_Test** » puis sélectionner **Add Reference** :



Cliquez sur l’onglet **Projects** puis sélectionnez le projet « **WinwiseCalculator** » si ce n’est pas déjà le cas puis cliquez sur **OK** :



Dans le fichier « **Test.cs** » du projet « **WinwiseCalculator\_Test** », rajoutez le using correspondant au namespace de l’assembly « **WinwiseCalculator** » :

using WinwiseCalculator;

et remplacez le code de la classe « **Test** » par celui-ci :

[TestMethod]

public void TestAddition()

{

Assert.AreEqual(5, Page.ComputeResult("2", "3", '+'));

Assert.AreEqual(2, Page.ComputeResult("2", "0", '+'));

Assert.AreEqual(-1, Page.ComputeResult("5", "-6", '+'));

}

[TestMethod]

public void TestSubstraction()

{

Assert.AreEqual(8, Page.ComputeResult("10", "2", '-'));

Assert.AreEqual(5, Page.ComputeResult("5", "0", '-'));

Assert.AreEqual(3, Page.ComputeResult("2", "-1", '-'));

}

[TestMethod]

public void TestMultiplication()

{

Assert.AreEqual(12, Page.ComputeResult("3", "4", '\*'));

Assert.AreEqual(3, Page.ComputeResult("3", "1", '\*'));

Assert.AreEqual(0, Page.ComputeResult("5", "0", '\*'));

Assert.AreEqual(8, Page.ComputeResult("-2", "-4", '\*'));

}

[TestMethod]

public void TestValidInteger()

{

Page.ComputeResult("999999999999999", "9999999999999999999", '\*');

Page.ComputeResult("hello", "world", '+');

}

[TestMethod]

[ExpectedException(typeof(DivideByZeroException))]

public void TestDivision()

{

Assert.AreEqual(4, Page.ComputeResult("8", "2", '/'));

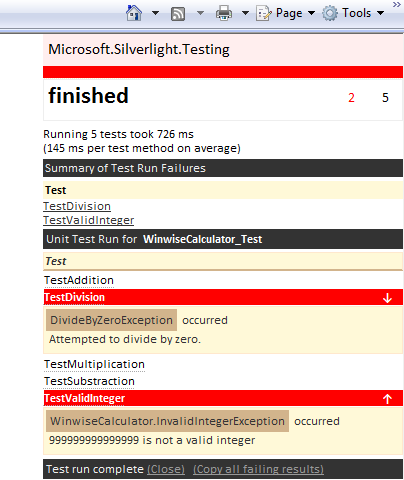
Assert.AreEqual(6, Page.ComputeResult("6", "1", '/'));

Assert.AreEqual(3, Page.ComputeResult("-6", "-2", '/'));

Page.ComputeResult("3", "0", '/');

}

Assurez vous que le projet « **WinwiseCalculator\_Test** » est bien définit comme projet de démarrage puis faites **Ctrl + F5** pour lancer l’application sans debug :



Un rapport sur les différents tests effectués s’affiche alors. On peut constater que plusieurs de nos tests ont échoué :

* Le test de la division.
* Le test de passage d’un entier invalide.

On a également des informations supplémentaires nous indiquant par exemple que c’est la division par 0 qui n’est pas gérée par notre application puisque une exception de type **DivideByZeroException** a été levée.

En ce qui concerne le test de la division par 0, ce n’est pas à notre méthode « **ComputeResult** » de gérer ce cas particulier mais bel et bien à la méthode appelante, donc on peut considérer qu’il est normal que la méthode « **ComputeResult** » lève une exception de type **DivideByZeroException** lorsqu’une division par 0 est faite. Nous allons donc ajouter un attribut à notre méthode de test pour spécifier que ce type d’exception est attendu lors de l’exécution de la méthode de test puis ensuite gérer cette exception dans la méthode appelante.

Ainsi rajoutez cette ligne en dessous de l’attribut « **TestMethod** » de la méthode « **TestDivision** » :

[ExpectedException(typeof(DivideByZeroException))]

Puis remplacez le code de la méthode « **Result** » par celui-ci :

private void Result()

{

if (this.currentOperator == null || this.leftOperand == null)

{

return;

}

this.rightOperand = this.EquationTextBox.Text;

try

{

int result = ComputeResult(this.leftOperand, this.rightOperand, this.currentOperator[0]);

this.EquationTextBox.Text = result.ToString();

this.leftOperand = this.endEquation ? null : result.ToString();

this.rightOperand = null;

this.currentOperator = null;

this.clear = true;

this.endEquation = false;

}

catch (Exception exception)

{

this.EquationTextBox.Text = exception.Message;

}

}

Ainsi, si une exception est levée dans la méthode « **ComputeResult** », le message de l’exception sera affiché dans la *textbox*.

Pour régler le problème de l’entier invalide, remplacer le code de la méthode « **ComputeResult** » par celui-ci :

internal static int ComputeResult(string str1, string str2, char op)

{

int nb1;

int nb2;

if (!int.TryParse(str1, out nb1))

{

throw new InvalidIntegerException(str1 + " is not a valid integer");

}

if (!int.TryParse(str2, out nb2))

{

throw new InvalidIntegerException(str2 + " is not a valid integer");

}

switch (op)

{

case '\*':

return nb1 \* nb2;

case '/':

return nb1 / nb2;

case '+':

return nb1 + nb2;

case '-':

return nb1 - nb2;

default:

return 0;

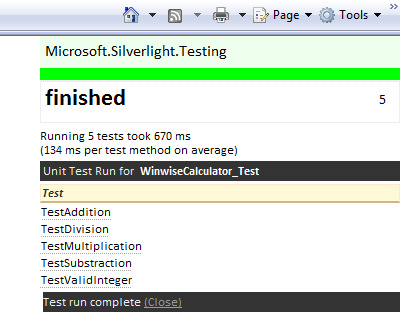
}

}

Ainsi, si un entier passé en paramètre n’est pas invalide, une exception personnalisée est levée, **InvalidIntegerException**. Il va donc falloir là aussi spécifier à notre méthode de test que cette exception est attendue. Ainsi rajoutez cette ligne en dessous de l’attribut « **TestMethod** » de la méthode « **TestValidInteger**» :

[ExpectedException(typeof(InvalidIntegerException))]

Retestez en faisant **Ctrl + F5** :



Cette fois-ci tous les tests sont passés avec succès.

# 7ème Partie : Développement d’un contrôle accessible

Au cours de cet atelier, nous allons voir comment rendre accessible un contrôle personnalisé. Nous allons nous appuyer sur le contrôle « WinwiseExpanderPanel » et ajouter le code nécessaire pour qu’il puisse être lu à partir d’un client.

Dans le fichier « WinwiseExpanderPanel.cs », ajoutez les using suivant afin de pouvoir utiliser différents types de l’UI Automation toolkit :

using System.Windows.Automation.Peers;

using System.Windows.Automation.Provider;

using System.Windows.Automation;

Ensuite nous allons créer une classe interne qui va être le « peer » de notre contrôle. C'est-à-dire une classe contenant le code utilisé par l’UI Automation pour examiner l’interface du contrôle afin d’interagir avec certains éléments qui le composent. Cette classe héritera de « FrameworkElementAutomationPeer » afin de surcharger les méthodes qui seront utilisés par l’UI Automation :

public class WinwiseExpanderPanelAutomationPeer : FrameworkElementAutomationPeer

{

}

Commencez par créer un membre pour stocker l’instance de notre contrôle qui va être mis à contribution au moment de l’utilisation du « peer » :

private WinwiseExpanderPanel m\_WinwiseExpanderPanel;

Et implémentez le constructeur de sorte qu’il prenne en entrée une référence vers un objet de type « WinwiseExpanderPanel » que l’on va sauvegarder dans le membre que l’on vient juste de créer et que l’on utilisera dans les différentes méthodes surchargées de l’UI automation :

public WinwiseExpanderPanelAutomationPeer(WinwiseExpanderPanel winwiseExpanderPanel) : base(winwiseExpanderPanel)

{

this.m\_WinwiseExpanderPanel = winwiseExpanderPanel;

}

Maintenant nous allons surcharger certaines méthodes de la classe « FrameworkElementAutomationPeer ». Pour développer un « automation peer » il faut au minimum surcharger les méthodes « GetClassNameCore » qui permet de donner le nom du contrôle que l’on veut rendre accessible et « GetAutomationControlTypeCore » qui permet de donner le type de contrôle se rapprochant le plus de la logique de notre contrôle :

protected override string GetClassNameCore()

{

return "WinwiseExpanderPanel";

}

protected override AutomationControlType GetAutomationControlTypeCore()

{

return AutomationControlType.Pane;

}

Nous allons également surcharger la méthode « GetPattern » qui permet pour un pattern donné de renvoyé le contrôle qui le supporte. Ici il n’y a pas de « sous contrôle » supportant un pattern précis donc nous allons simplement renvoyer « this » pour le pattern « Toggle » :

public override object GetPattern(PatternInterface patternInterface)

{

if (patternInterface == PatternInterface.Toggle)

{

return this;

}

return null;

}

Enfin nous allons surcharger la méthode « IsContentElementCore » qui renvoit un booléen pour spécifier si le contrôle a un contenu :

protected override bool IsContentElementCore()

{

return true;

}

Il faut maintenant implémenter le ou les interfaces qui vont définir quel UI patterns sont supportés par notre contrôle. Ici notre contrôle a 2 états (plié ou déplié) donc le « UI automation provider » de notre contrôle est un « ToggleProvider ». Ainsi il faut implémenter l’interface IToggleProvider. Pour ce faire, remplacer la ligne :

public class WinwiseExpanderPanelAutomationPeer : FrameworkElementAutomationPeer

Par la ligne suivante :

private class WinwiseExpanderPanelAutomationPeer : FrameworkElementAutomationPeer, IToggleProvider

Cette interface va nous obliger à implémenter une méthode et une propriété. Tout d’abord la méthode « Toggle » qui va inverser sa propriété IsOpened:

public void Toggle()

{

m\_WinwiseExpanderPanel.IsOpened = !m\_WinwiseExpanderPanel.IsOpened;

}

Ensuite la propriété « ToggleState » qui renvoi simplement l’état (« On » ou « Off ») de notre « expander » :

public ToggleState ToggleState

{

get

{

return m\_WinwiseExpanderPanel.IsOpened ? ToggleState.On : ToggleState.Off;

}

}

Enfin, au niveau de notre classe « WinwiseExpanderPanel », il nous faut surcharger la méthode « OnCreateAutomationPeer » pour qu’elle renvoie le « peer » que l’on vient juste d’implémenter :

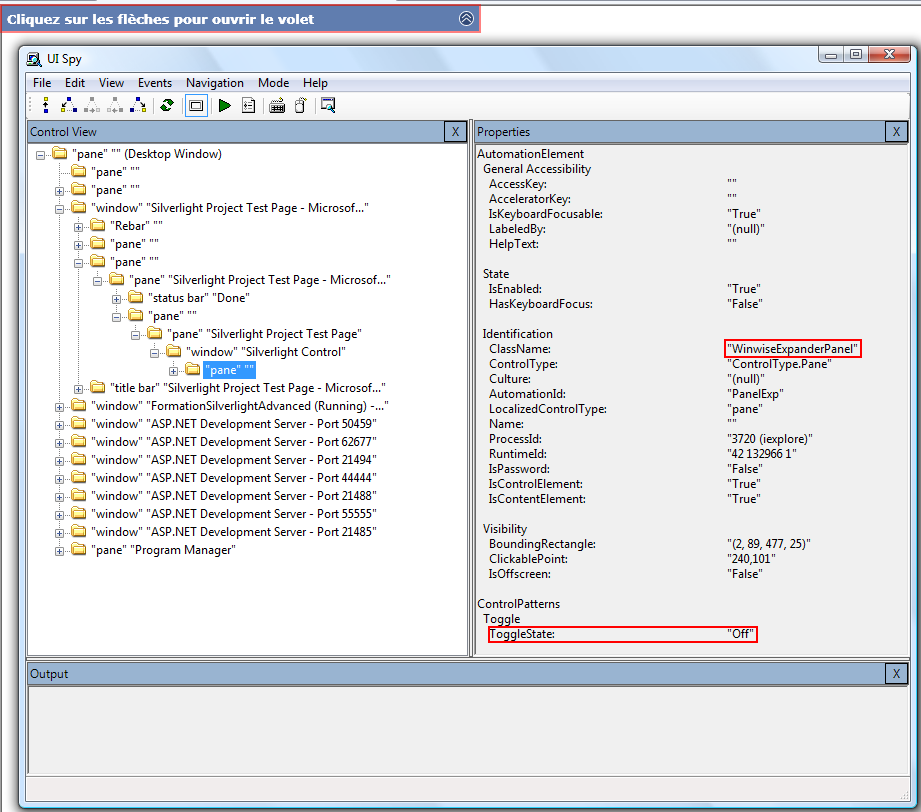
protected override AutomationPeer OnCreateAutomationPeer()

{

return new WinwiseExpanderPanelAutomationPeer(this);

}

Assurez-vous que le projet « WinwiseTemplatableControl\_Web » est défini comme projet de démarrage et lancez la solution en appuyant sur « F5 ». Une fois l’application démarrée, lancez « UISpy » dont le raccourci se situe sur le bureau et accéder au nœud correspondant à notre contrôle comme montré ci-dessous :



Si vous dépliez l’ « expander » et que vous rafraichissez l’affichage de « UISpy » vous verrez le « ToggleState » passé à « On ».

Sous UISpy vous pouvez également invoquer les différentes méthodes du modèle, pour cela effectuer un clic-droit sur le contrôle puis cliquez sur « Control pattern », dans la fenêtre qui apparait, vous avez une liste déroulante avec les différentes méthodes exposées par le modèle que vous pouvez appeler en appuyant sur le bouton « Call method ».

# 8ème Partie : Réalisation de tests d’interface

Au cours de cet atelier, nous allons voir comment réaliser des tests d’interface pour une application Silverlight et ce pour valider le bon fonctionnement de notre application au sens graphique du terme. Nous allons travailler sur la même application que pour la partie précédente et nous allons simplement tester si un clic sur un bouton modifie bien le texte de la *textbox* avec la bonne valeur, i.e. si un clic sur le bouton contenant le texte « 1 » affiche bien « 1 » dans la *textbox*.

Dans le fichier « **Test.cs** » du projet « **WinwiseCalculator\_Test »**, faites hériter la class **Test** de la classe **SilverlightTest** et ce afin de pouvoir visualiser l’évolution des tests sur notre application :

public class Test : SilverlightTest

Cette classe permet d’accéder à une propriété **TestPanel** qui est la zone où va s’afficher notre application dans la page de test. Ainsi rajouter le code suivant à la classe **Test**:

private Page page;

[TestInitialize]

public void Setup()

{

page = new Page();

TestPanel.Children.Add(page);

}

Cette méthode va avoir pour but d’être appelée en tout début de test avant les autres méthodes et c’est elle qui va nous permettre d’ajouter notre application dans la zone prévue à cet effet dans la page de test.

Ensuite ajoutez la méthode suivante à la classe :

[TestMethod]

[Asynchronous]

public void ClickButtonTest()

{

ButtonAutomationPeer buttonPeer = new ButtonAutomationPeer(page.Button1);

IInvokeProvider buttonProvider = (IInvokeProvider)buttonPeer;

EnqueueCallback(() => buttonProvider.Invoke());

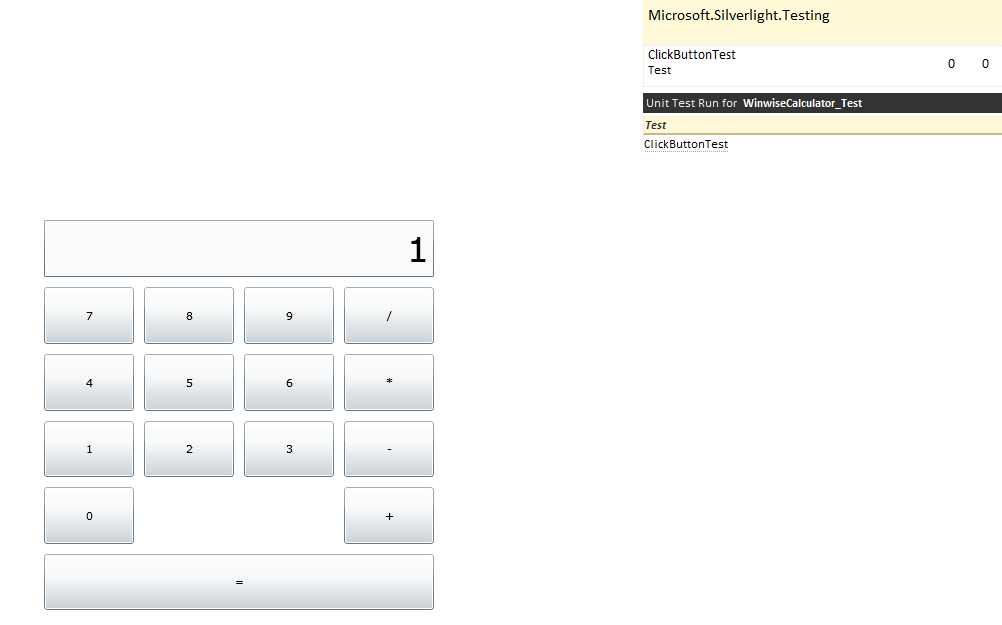
EnqueueSleep(2000);

EnqueueCallback(() => Assert.AreEqual("1", page.EquationTextBox.Text));

EnqueueTestComplete();

}

C’est notre méthode de test pour l’interface. Elle possède un attribut en plus que les autres méthodes de test que nous avons vues jusqu’ici qui est **Asynchronous** et qui spécifie simplement que le test va s’effectuer en asynchrone, en effet toutes les méthodes de test d’interface doivent être asynchrones. Dans cette méthode, nous utilisons la classe **ButtonAutomationPeer** qui va permettre d’exposer différents membres d’un bouton qui lui est donné à sa construction (en occurrence ici le bouton contenant le texte « 1 »). Les informations apportées par ces membres vont être utilisées par le toolkit dans l’exécution de certains patterns qui vont permettre l’automatisation des tests de l’interface. Ensuite nous utilisons un fournisseur qui va automatiser le clic sur le bouton puis nous vérifions que la *textbox* contient bien le texte « 1 » :



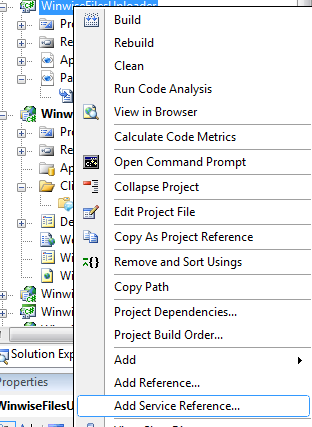
# 9ème Partie : Isolated Storage & OpenFileDialog

Dans cet atelier, nous allons voir comment utiliser le contrôle *OpenFileFialog* et le *stockage isolé*. Dans la majorité des cas, le contrôle de sélection de fichiers est utilisé pour créer des applications permettant d’uploader des fichiers : c’est donc ce que nous allons tenter de reproduire en Silverlight.

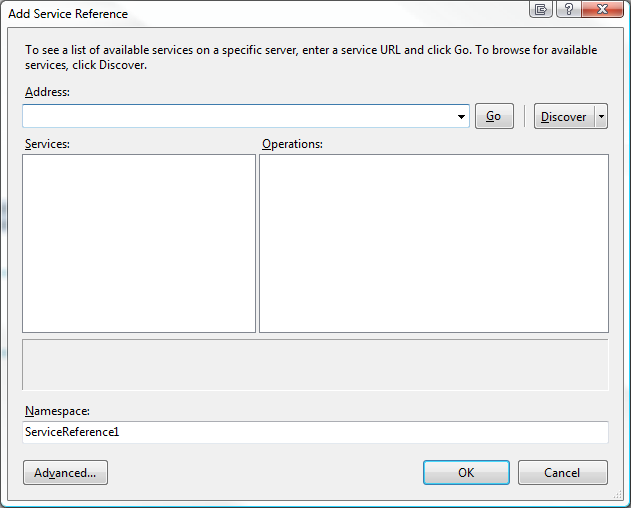
## Utilisation du contrôle OpenFileDialog

Dans la solution Visual Studio que vous utilisez se trouve un Web Service WCF qui va vous permettre de faire l’upload. Vous avez également un projet, **WinwiseFilesUploader**, dans lequel vous allez devoir appeler ce service Web.

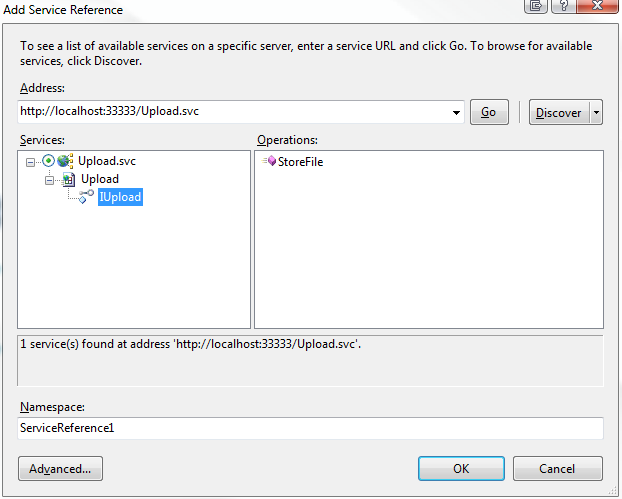
Commencez par définir le projet **WinwiseFilesUploader\_Web** comme projet de démarrage de votre solution. En seconde étapes, ajoutons une référence vers notre serveur WCF. Pour cela, faîtes un clic droit sur votre projet Silverlight et choisissez "*Add Service Reference*" :



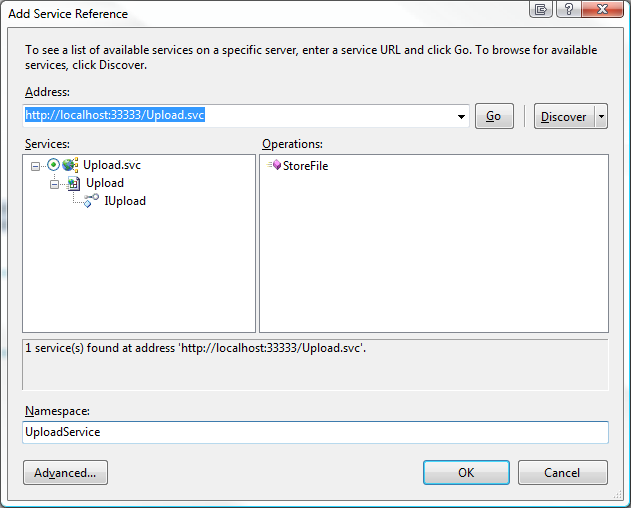
Dans la fenêtre qui apparait, cliquez sur le bouton "**Discover**"



Là, vous devriez voir apparaitre la liste de tous les Web Services présent dans la solution (il ne devrait y en avoir qu’un seul) : en dépliant l’arborescence, vous êtes en mesure de voir toutes les méthodes proposées par ce service :



Entrez alors le nom de "**UploadService**" sous la zone de texte nommée "*Namespace* :" et cliquez sur "*OK*" :



Ensuite, ouvrez le fichier *Page.xaml.cs*. Ajoutez les variables membres suivantes :

private byte[] FileContent = null;

private UploadService.UploadClient uploadClient = null;

Puis, sous la ligne :

InitializeComponent();

Rajoutez le code suivant :

this.btnBrowse.Click += new RoutedEventHandler(btnBrowse\_Click);

this.btnUpload.Click += new RoutedEventHandler(btnUpload\_Click);

uploadClient = new WinwiseFilesUploader.UploadService.UploadClient();

uploadClient.StoreFileCompleted += new EventHandler<System.ComponentModel.AsyncCompletedEventArgs>(uploadClient\_StoreFileCompleted);

Sur les 2 premières lignes de code, on ne fait que s’abonner à l’évènement *Click* de nos boutons puis on met en place le client WCF qui va appeler le service et on s’abonne à l’évènement *completed* de notre méthode WCF.

Ensuite, sous le constructeur de la classe *Page*, ajoutez ces lignes de code (qui ne sont rien d’autres que la déclaration des méthodes appelées lors des évènements *Click* des boutons) :

private void btnBrowse\_Click(object sender, RoutedEventArgs e)

{

OpenFileDialog ofd = new OpenFileDialog();

ofd.Multiselect = false;

if (ofd.ShowDialog().Value)

{

this.tbFileToUpload.Text = ofd.File.Name;

using (Stream streamFileToUpload = ofd.File.OpenRead())

{

FileContent = new byte[streamFileToUpload.Length];

streamFileToUpload.Read(FileContent, 0, (int)streamFileToUpload.Length);

}

}

}

private void btnUpload\_Click(object sender, RoutedEventArgs e)

{

UploadService.FileUpload fileUpload = new WinwiseFilesUploader.UploadService.FileUpload();

fileUpload.Name = this.tbFileToUpload.Text;

fileUpload.File = FileContent;

uploadClient.StoreFileAsync(fileUpload);

}

private void uploadClient\_StoreFileCompleted(object sender, System.ComponentModel.AsyncCompletedEventArgs e)

{

if (e.Error == null)

{

this.tbFileToUpload.Text = "Fichier uploadé !";

}

else

{

this.tbFileToUpload.Text = string.Format("Une erreur est survenue: {0}", e.Error.Message);

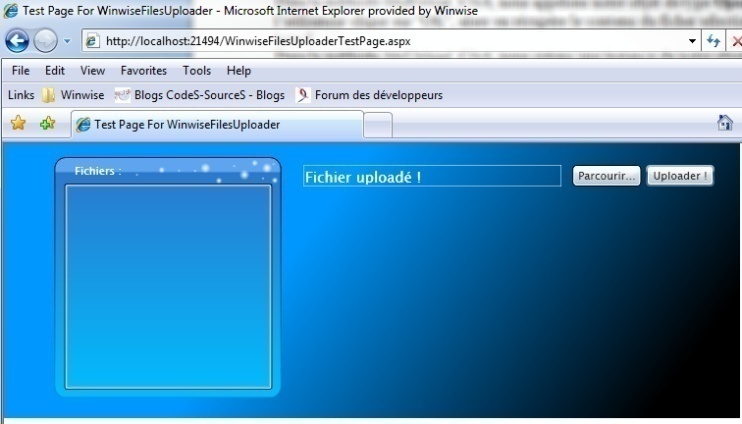
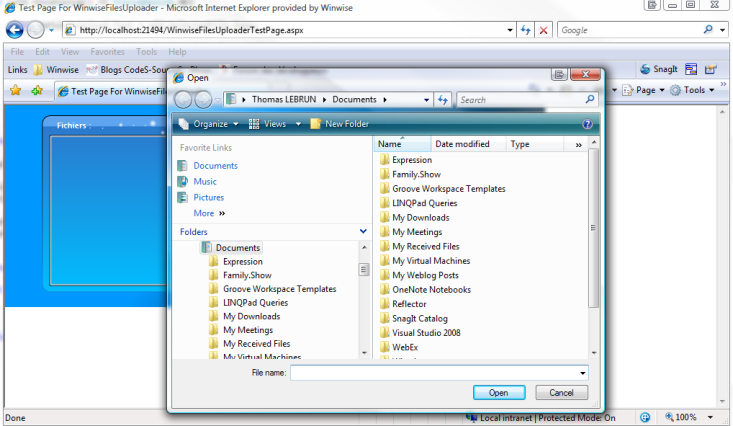
}

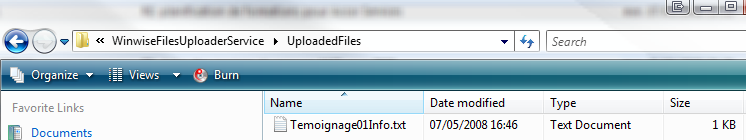
}

Dans la méthode *btnBrowse\_Click*, nous appelons notre objet de type **OpenFileDialog** et, si l’utilisateur clique sur "OK", alors on récupère le contenu du ficher sélectionné dans un tableau de byte.

Dans la méthode *btnUpload\_Click*, nous créons une instance de notre objet **FileUpload**, nous spécifions ses propriétés puis nous appelons la méthode **StoreFileAsync** de notre service WCF, qui n’est rien d’autre qu’un appel asynchrone de la méthode *StoreFile*. Nous nous abonnons également à l’évènement **StoreFileCompleted** (qui survient lorsque l’appel de la méthode est terminé) et nous vérifions qu’aucune erreur n’est survenue durant l’envoi du fichier sur le serveur.

A l’exécution, vous constatez qu’un clic sur le bouton "*Parcourir*…" affiche bien la boite de dialogue et un clic sur le bouton "*Upload*" envoi bien le fichier sur le serveur (vous pouvez le vérifier en regardant le contenu du répertoire "*UploadedFiles*" dans le projet du service WCF) :

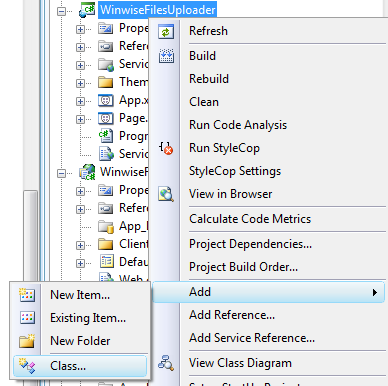




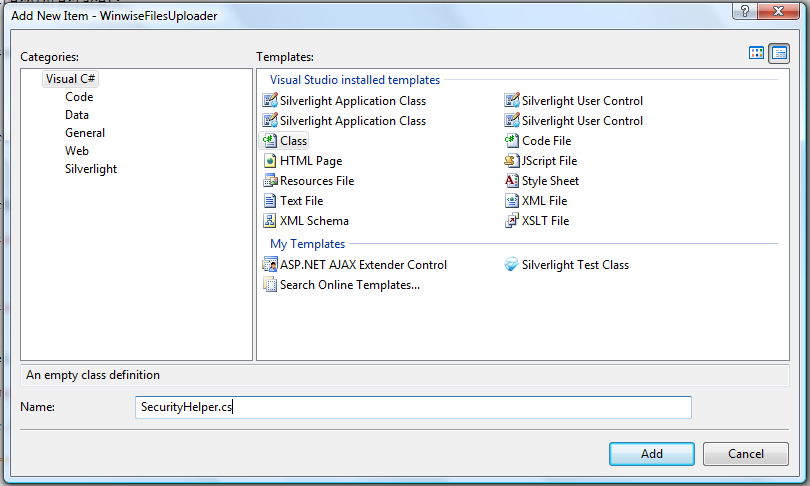
## Utilisation du stockage isolé

Nous allons à présent voir comment nous pouvons utiliser le stockage isolé pour sauvegarder des informations dans une application Silverlight. Nous allons également voir comment crypter les données que l’on va enregistrer dans cet espace.

Pour cela nous allons commencer par développer une classe « Helper » qui va effectuer les tâches relatives au cryptage des données. Faites un clic droit sur le projet « WinwiseFilesUploader » puis « Add » et enfin « Class » :



Entrez comme nom « SecurityHelper » et cliquez sur « Add » :



Cette classe est un « Helper » et ne sera utilisée que par l’assembly « WinwiseFileUploader » donc on peut la passer en « internal static ».

Remplacez cette ligne :

public class SecurityHelper

par celle-ci :

internal static class SecurityHelper

Nous allons utiliser le seul algorithme de cryptage à clé symétrique disponible avec Silverlight : l’algorithme AES. Tout d’abord nous allons créer une méthode qui va nous permettre de chiffrer la clé à partir d’un mot de passe. Commencez d’abord par ajoutez les « using » suivants :

using System.IO;

using System.Text;

using System.Security.Cryptography;

Puis ajoutez donc la méthode à la classe :

internal static byte[] GetHashKey(string password)

{

return new Rfc2898DeriveBytes(password, UTF8Encoding.UTF8.GetBytes("Some salt")).GetBytes(16);

}

Implémentons maintenant la méthode d’encryptions des données :

internal static string Encrypt(byte[] key, string dataToEncrypt)

{

AesManaged encryptor = new AesManaged

{

Key = key,

IV = key

};

using (MemoryStream encryptionStream = new MemoryStream())

{

using (CryptoStream encrypt = new CryptoStream(encryptionStream, encryptor.CreateEncryptor(), CryptoStreamMode.Write))

{

byte[] utfD1 = UTF8Encoding.UTF8.GetBytes(dataToEncrypt);

encrypt.Write(utfD1, 0, utfD1.Length);

encrypt.FlushFinalBlock();

encrypt.Close();

return Convert.ToBase64String(encryptionStream.ToArray());

}

}

}

Ici nous instancions un encrypteur AES à partir de la clé qui nous est passée en paramètre puis nous utilisons la classe « CryptoStream » qui va utiliser notre encrypteur pour écrire les données encryptés au sein d’un « Stream » que nous convertissons ensuite en une « String » encodé en base 64.

Logique quasi identique pour la décryption :

internal static string Decrypt(byte[] key, string encryptedString)

{

if (String.IsNullOrEmpty(encryptedString))

return String.Empty;

AesManaged decryptor = new AesManaged

{

Key = key,

IV = key

};

byte[] encryptedData = Convert.FromBase64String(encryptedString);

using (MemoryStream decryptionStream = new MemoryStream())

{

using (CryptoStream decrypt = new CryptoStream(decryptionStream, decryptor.CreateDecryptor(), CryptoStreamMode.Write))

{

decrypt.Write(encryptedData, 0, encryptedData.Length);

decrypt.Flush();

decrypt.Close();

byte[] decryptedData = decryptionStream.ToArray();

return UTF8Encoding.UTF8.GetString(decryptedData, 0, decryptedData.Length);

}

}

}

Passons ensuite au code pour l’upload des fichiers ainsi que pour l’enregistrement des données au sein du stockage isolé.

Commençons par rajouter quelques variables dans notre projet. Sous la ligne :

private UploadService.UploadClient uploadClient = null;

Rajoutez ce code:

private const string PASSWORD = "W!inw$e@2009";

private byte[] key;

// Emplacement de stockage

private System.IO.IsolatedStorage.IsolatedStorageSettings appSettings = System.IO.IsolatedStorage.IsolatedStorageSettings.ApplicationSettings;

Ensuite, sous la ligne:

uploadClient = new WinwiseFilesUploader.UploadService.UploadClient();

Rajoutez l’appel suivant:

InitializeIsolatedStorage();

A présent, localisez les lignes suivantes:

InitializeIsolatedStorage();

}

Et ajoutez en dessous cette méthode, qui sert à initialiser le stockage isolé, c’est-à-dire :

* Définir des paramètres si ceux-ci n’existent pas
* Remplir la ListBox avec le nom des fichiers déjà uploadés sur le serveur

private void InitializeIsolatedStorage()

{

this.key = SecurityHelper.GetHashKey(PASSWORD);

if (!appSettings.Contains("HistoryFileContent"))

{

appSettings.Add("HistoryFileContent", string.Empty);

}

ParseIsolatedStorageFile(SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"])));

}

Comme vous pouvez le voir, cette méthode commence par vérifier si le paramètre nommé **HistoryFileContent** existe et, si la réponse est non, elle le créé.

Ensuite, on ne fait que passer le contenu de ce paramètre à la méthode **ParseIsolatedStorageFile**, que vous devez implémenter, via ce code, juste en dessous :

private void ParseIsolatedStorageFile(string fileContent)

{

this.lbFilesUploaded.ItemsSource = null;

var results = from file in fileContent.Split(';')

select file;

if (results.Count() > 0)

{

this.lbFilesUploaded.ItemsSource = results.TakeWhile(s => !string.IsNullOrEmpty(s));

}

this.lbFilesUploaded.ItemsSource = results;

}

Cette méthode récupère le contenu passé en paramètre, le sépare en utilisant le point-virgule comme séparateur et affecte le résultat de cette séparation à la propriété ItemsSource de notre ListBox (tant que le contenu n’est pas une chaîne vide).

A présent, il nous faut modifier notre code pour insérer du contenu (le nom des fichiers) dans le stockage isolé, et rafraichir notre ListBox, à chaque fois qu’un fichier est uploadé avec succès.

Là encore, il s’agit d’une opération très simple : trouvez la ligne suivante :

this.tbFileToUpload.Text = "Fichier uploadé !";

Et rajouté juste avant :

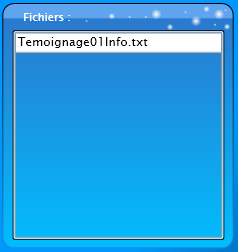
string contentFile = SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"]));

contentFile += string.Concat(this.tbFileToUpload.Text, ";");

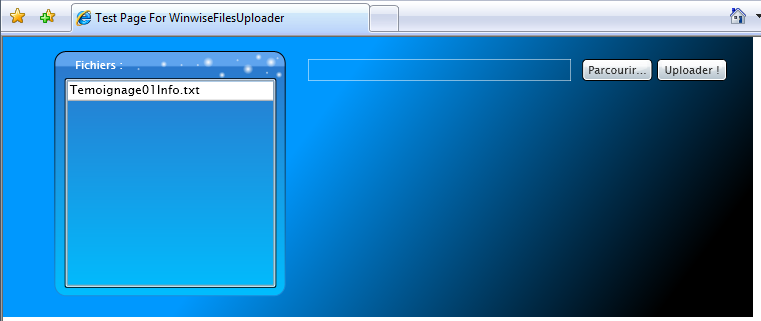
appSettings["HistoryFileContent"] = SecurityHelper.Encrypt(this.key, contentFile);

ParseIsolatedStorageFile(SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"])));

A l’exécution, vous constatez alors que dès qu’un fichier est correctement uploadé, son nom apparait dans la ListBox:



De plus, si vous fermez la fenêtre et que vous relancez l’application, vous constaterez que dès le démarrage, la lecture du fichier utilisé pour l’historique est un succès : notre application Silverlight est donc en mesure d’enregistrer des informations et de les réutiliser, même après une fermeture de l’application ou bien son ouverture avec un autre navigateur :



# 10ème Partie : Utilisation du BackgroundWorker

Dans cet atelier, nous allons voir comment tirer parti du composant nommé **BackgroundWorker**, qui permet d’effectuer des tâches en arrière-plan.

Pour cela, nous allons mettre en place un upload de fichiers multiples, avec Silverlight. Comme cet upload peut-être long, nous allons afficher sa progression à l’aide d’une *ProgressBar*.

Pour développer notre composant, nous allons repartir de l’application que nous venons de créer et la modifier un peu. La première chose à faire est de modifier notre interface graphique pour ajouter les autres zones de saisie et la barre de progression.

Ainsi, ouvrez le fichier *Page.xaml* et, sous la ligne :

<TextBox x:Name="tbFileToUpload" Height="22" Margin="305.3330078125,21.9950008392334,182,0" VerticalAlignment="Top" Text="" Background="Transparent" BorderBrush="White" Foreground="White" BorderThickness="0.5" />

Ajoutez ce code:

<TextBox x:Name="tbFileToUpload2" Height="22" Margin="305.3330078125,60,182,0" VerticalAlignment="Top" Text="" Background="Transparent" BorderBrush="White" Foreground="White" BorderThickness="0.5" />

<TextBox x:Name="tbFileToUpload3" Height="22" Margin="305.3330078125,100,182,0" VerticalAlignment="Top" Text="" Background="Transparent" BorderBrush="White" Foreground="White" BorderThickness="0.5" />

Ensuite, sous la ligne suivante:

<TextBox x:Name="tbFileToUpload3" Height="22" Margin="305.3330078125,100,182,0" VerticalAlignment="Top" Text="" Background="Transparent" BorderBrush="White" Foreground="White" BorderThickness="0.5" />

Rajoutez ce code, qui servira à afficher la progression de l’envoi des fichiers sur le serveur :

<ProgressBar x:Name="pb" Height="20" Width="260" Margin="305.3330078125,220,0,0" VerticalAlignment="Top" Minimum="0" Maximum="100" />

A présent, il va nous falloir modifier le code de notre application pour quelle fonctionne avec plusieurs fichiers à uploader. Pour cela, la première chose à faire est d’autoriser la multi-sélection dans la boite de dialogue de sélection de fichier. Ainsi, remplacez la ligne suivante :

ofd.Multiselect = false;

Par cette ligne :

ofd.Multiselect= true;

Ensuite, remplacez le code suivant :

if (ofd.ShowDialog().Value)

{

this.tbFileToUpload.Text = ofd.Files.Name;

using (Stream streamFileToUpload = ofd.SelectedFile.OpenRead())

{

FileContent = new byte[streamFileToUpload.Length];

streamFileToUpload.Read(FileContent, 0, (int)streamFileToUpload.Length);

}

}

Par ce code :

if (ofd.ShowDialog().Value)

{

foreach (var item in ofd.Files)

{

using (Stream streamFileToUpload = item.OpenRead())

{

FileContent = new byte[streamFileToUpload.Length];

streamFileToUpload.Read(FileContent, 0, (int)streamFileToUpload.Length);

listOfFile.Add(item.Name, FileContent);

}

}

if (listOfFile.Keys.Count == 3)

{

this.tbFileToUpload.Text = listOfFile.Keys.ToList()[0];

this.tbFileToUpload2.Text = listOfFile.Keys.ToList()[1];

this.tbFileToUpload3.Text = listOfFile.Keys.ToList()[2];

}}

Comme vous pouvez le voir, le code est très similaire à la version précédente hormis le fait que l’on itère maintenant sur l’ensemble des fichiers sélectionnés et, pour chacun de ces fichiers, on stocke son nom et son contenu (soute forme de tableau de byte) dans un dictionnaire. Il nous faut donc déclarer ce dictionnaire et, pour ce faire, nous allons ajouter la ligne suivante :

private Dictionary<string, byte[]> listOfFile = new Dictionary<string, byte[]>();

Juste en dessous de cette ligne:

private UploadService.UploadClient uploadClient = null;

Vous pouvez également remarquer que nous spécifions le contenu des zones de saisies en 4 étapes :

* Accès au dictionnaire
* Accès à la liste des clés
* Conversion de la liste des clés en liste générique
* Accès à l’élément qui nous intéresse dans cette liste

Note : Attention, bien que fonctionnel, ce code n’est pas le plus optimisé/performant. Il aurait été plus "sage" de faire la conversion en liste générique une seule fois et de réutiliser cette liste. De plus, vous pouvez remarquer que certaines valeurs sont fixées en dures : il serait intéressant d’optimiser ce point (et certains autres) dans le cadre d’un développement plus professionnel.

Maintenant, nous allons modifier le code de la méthode appelée lorsque l’utilisateur clique sur le bouton "*Upload*". Remplacez donc ces lignes :

UploadService.FileUpload fileUpload = new WinwiseFilesUploader.UploadService.FileUpload();

fileUpload.Name = this.tbFileToUpload.Text;

fileUpload.File = FileContent;

uploadClient.StoreFileAsync(fileUpload);

Par celles-ci :

worker = new System.ComponentModel.BackgroundWorker();

worker.WorkerReportsProgress = true;

worker.ProgressChanged += new System.ComponentModel.ProgressChangedEventHandler(worker\_ProgressChanged);

worker.DoWork += new System.ComponentModel.DoWorkEventHandler(worker\_DoWork);

worker.RunWorkerCompleted += new System.ComponentModel.RunWorkerCompletedEventHandler(worker\_RunWorkerCompleted);

worker.RunWorkerAsync();

Comme vous pouvez le voir, ces lignes sont utilisées pour déclarer un objet de type BackgroundWorker, s’abonner à certains de ces évènements et à le lancer. Pourquoi utiliser un BackgroundWorker ? Tout simplement car nous allons uploader plusieurs fichiers et, pour éviter que l’interface utilisateur ne soit gelée et ne rende la main qu’une fois les upload terminés, il faut les exécuter en tâche de fond : c’est le rôle du BackgroundWorker !

Pour que cela fonctionne, il ne faut pas oublier de rajouter la déclaration de l’objet. En dessous de la ligne suivante :

private Dictionary<string, byte[]> listOfFile = new Dictionary<string, byte[]>();

Rajoutez cette ligne :

private System.ComponentModel.BackgroundWorker worker = null;

A présent, il ne reste plus qu’à écrire le code des méthodes appelées lors des différents évènements du BackgroundWorker. Il y a 3 méthodes à écrire :

* worker\_DoWork
* worker\_ProgressChanged
* worker\_RunWorkerCompleted

Commençons par la plus simple : *worker\_ProgressChanged*. Cette méthode est utilisée pour appeler du code qui sera exécuté lorsque le système sera notifié que le composant a avancé dans son travail.

Ainsi, sous la ligne de code suivante :

worker.RunWorkerAsync();

}

Rajoutez cette méthode :

private void worker\_ProgressChanged(object sender, System.ComponentModel.ProgressChangedEventArgs e)

{

this.pb.Value = e.ProgressPercentage;

}

Comme vous pouvez le voir, elle est très simple : elle se contente de récupérer le pourcentage de progression du BackgroundWorker et de l’affecter à la propriété *Percentage* de la *ProgressBar*.

Note : L’évènement ProgressChanged est seul qui permet d’accéder aux contrôles de l’interface utilisateur. Si vous tentez de le faire dans un autre de ses évènements, vous aurez droit à une erreur "*Invalid cross thread call*"

A présent, ajoutons la méthode *worker\_RunWorkerCompleted*, qui sera appelée lorsque le composant aura finit son travail. Pour écrire cette méthode, sous la ligne :

this.pb.Value = e.ProgressPercentage;

}

Rajoutez ces lignes de code :

private void worker\_RunWorkerCompleted(object sender, System.ComponentModel.RunWorkerCompletedEventArgs e)

{

if (e.Error == null && !e.Cancelled && (this.pb.Value != 100))

{

this.pb.Value = 100;

ParseIsolatedStorageFile(SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"])));

}

}

Ici, vous pouvez constater que nous "trichons" : en effet, nous vérifions d’abord si tout s’est bien passé sans erreur et, si c’est bien le cas, alors nous affectons 100 à la valeur de notre ProgressBar puis nous appelons la méthode qui va lire le paramètre du stockage isolé pour remplir la ListBox avec la liste des fichiers uploadés.

A présent, il reste à écrire le code de *worker\_DoWork*, la méthode principale du BackgroundWorker. C’est en effet dans cette méthode que l’on va appeler le service WCF, de façon asynchrone. Pour écrire cette méthode, localisez les lignes suivantes :

ParseIsolatedStorageFile(SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"])));

}

}

Et rajoutez en dessous:

private void worker\_DoWork(object sender, System.ComponentModel.DoWorkEventArgs e)

{

for(int i = 0; i <= listOfFile.Count - 1; i++)

{

KeyValuePair<string, byte[]> item = listOfFile.ElementAt(i);

UploadService.FileUpload fileUpload = new WinwiseFilesUploader.UploadService.FileUpload();

fileUpload.Name = item.Key;

fileUpload.File = item.Value;

this.Dispatcher.BeginInvoke(() => uploadClient.StoreFileAsync(fileUpload, string.Concat(i, ";", fileUpload.Name)));

worker.ReportProgress(i \* Convert.ToInt32(100 / listOfFile.Count));

// Pause pour voir l'avancement

if (i != listOfFile.Count)

{

System.Threading.Thread.Sleep(1000);

}

}

}

Si on regarde de plus près cette méthode, on ne rend vite compte qu’elle n’est pas bien compliquée. En effet, elle est chargée de boucler sur l’ensemble des fichiers sélectionnés et, pour chaque fichier, elle appelle la méthode StoreFile de notre service. Ensuite, on informe le BackgroundWorker de son avancement et on fait une pause, pour que l’on puisse se rendre compte de l’avancement de la ProgressBar (étant donné que l’on envoie au serveur des fichiers qui font, au maximum, 2 Mo).

Il va maintenant falloir s’abonner à l’évènement indiquant la fin de l’upload de fichier. Pour cela, c’est très simple : sous la ligne :

uploadClient = new WinwiseFilesUploader.UploadService.UploadClient(binding, address);

Rajoutez ce code :

uploadClient.StoreFileCompleted += new EventHandler<System.ComponentModel.AsyncCompletedEventArgs>(uploadClient\_StoreFileCompleted);

A présent, il nous faut écrire le code de la méthode appelée (*uploadClient\_StoreFileCompleted*) pour le faire correspondre à nos besoins. Ainsi, remplacez ces lignes :

if (e.Error == null)

{

string contentFile = SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"]));

contentFile += string.Concat(this.tbFileToUpload.Text, ";");

appSettings["HistoryFileContent"] = SecurityHelper.Encrypt(this.key, contentFile);

ParseIsolatedStorageFile(SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"])));

this.tbFileToUpload.Text = "Fichier uploadé !";

}

else

{

this.tbFileToUpload.Text = string.Format("Une erreur est survenue: {0}", e.Error.Message);

}

Par celles-ci :

var parameters = Convert.ToString(e.UserState).Split(';');

int i = Convert.ToInt32(parameters[0]);

if (e.Error == null)

{

string contentFile = SecurityHelper.Decrypt(this.key, Convert.ToString(appSettings["HistoryFileContent"]));

contentFile += string.Concat(Convert.ToString(parameters[1]), ";");

appSettings["HistoryFileContent"] = SecurityHelper.Encrypt(this.key,contentFile);

if (i == 0)

{

this.tbFileToUpload.Text = "Fichier uploadé !";

}

else if (i == 1)

{

this.tbFileToUpload2.Text = "Fichier uploadé !";

}

else if (i == 2)

{

this.tbFileToUpload3.Text = "Fichier uploadé !";

}

}

else

{

string errorMsg = string.Format("Une erreur est survenue: {0}", e.Error.Message);

if (i == 0)

{

this.tbFileToUpload.Text = errorMsg;

}

else if (i == 1)

{

this.tbFileToUpload2.Text = errorMsg;

}

else if (i == 2)

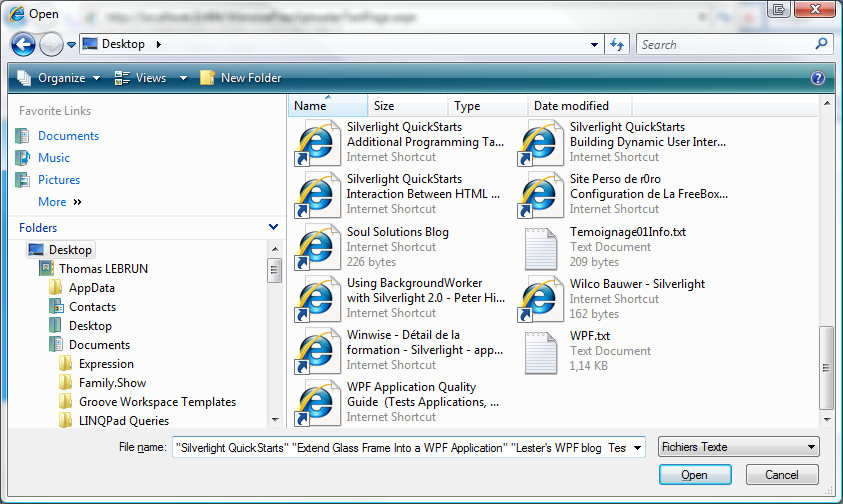
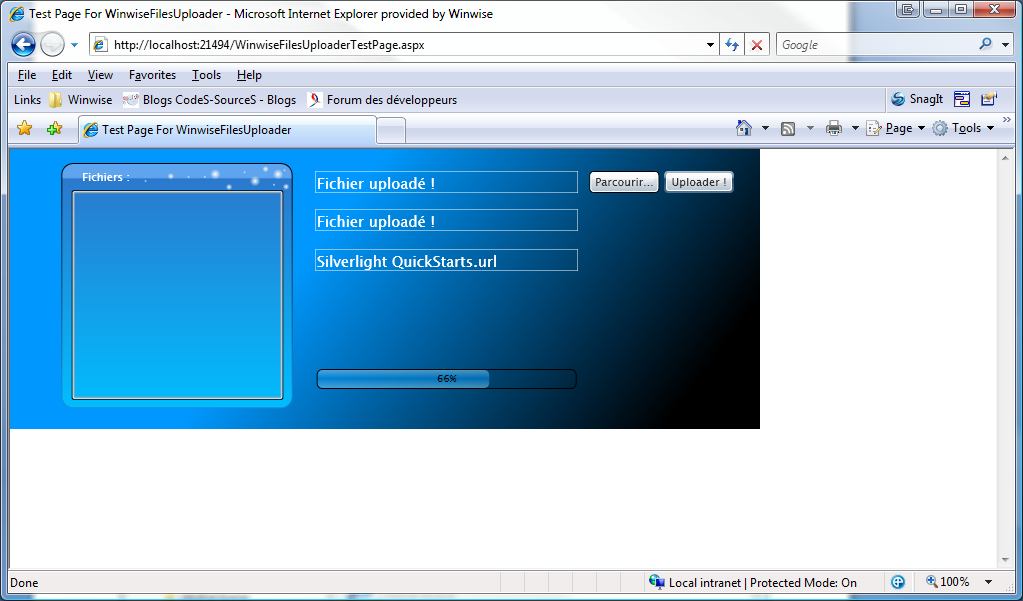
{

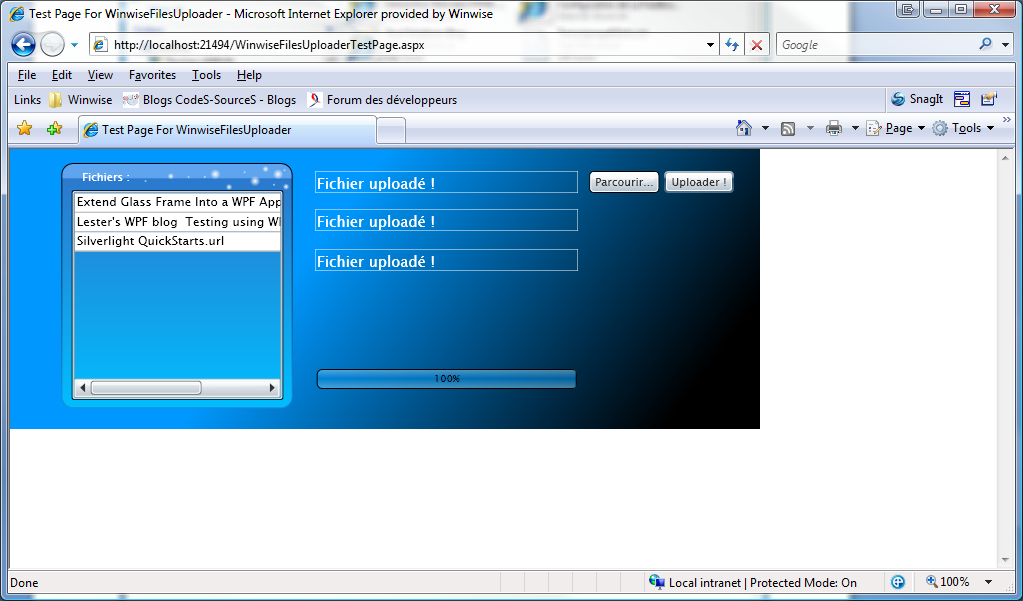
this.tbFileToUpload3.Text = errorMsg;

}

}

Si vous exécutez l’application (en appuyant simultanément sur Control + F5), vous verrez que vous pouvez bien choisir plusieurs fichiers (n’oubliez pas que nous sommes limité à 3 dans le cas présent). Ensuite, un clic sur le bouton "*Upload*" vous permettra d’envoyer les fichiers sur le serveur, tout en suivant la progression du téléchargement. Une fois que tout est terminé, la liste des fichiers uploadés apparait dans la ListBox :

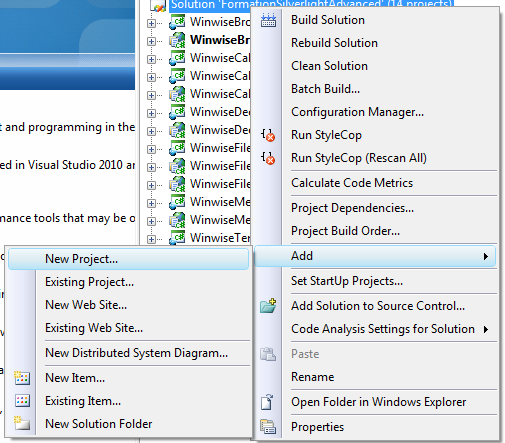
 



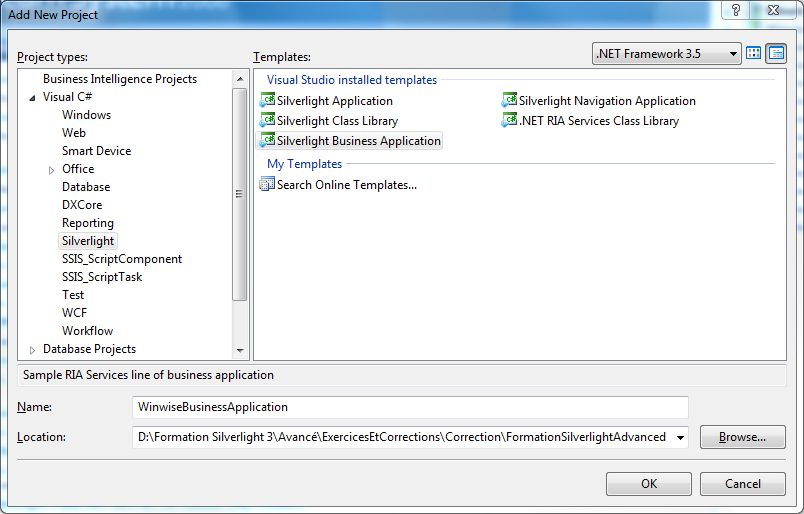
# 11ème Partie : .Net RIA Services

Dans cet atelier nous allons voir comment utiliser .Net Ria Services pour le développement d’une application métier et les différentes avantages que cela a.

Commencez par créer un nouveau projet. Pour cela clic-droit sur la solution puis « Add » et « New Project » :



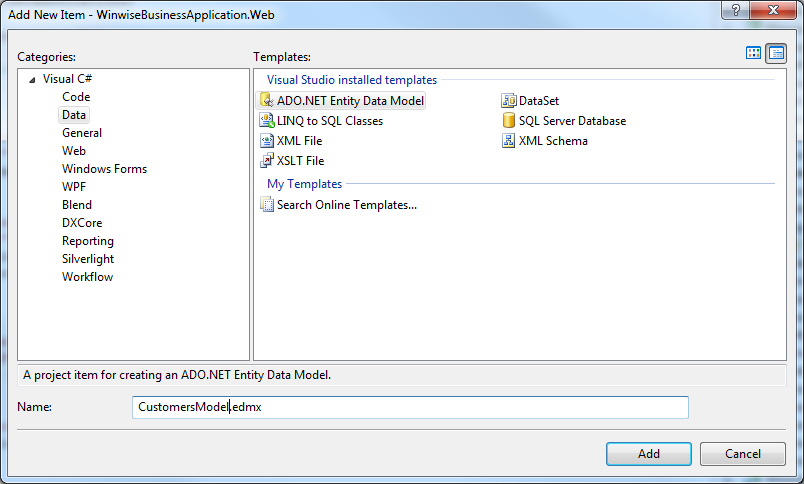
Ensuite dans le nœud Silverlight choisissez « Silverlight Business Application » et nommez le projet « WinwiseBusinessApplication » :



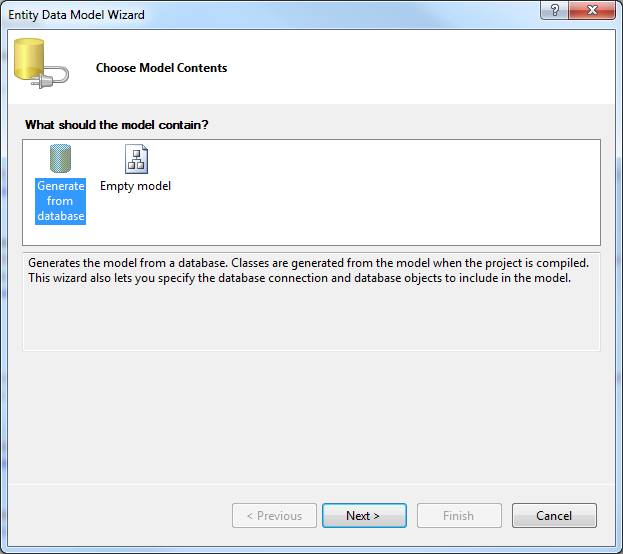
Ce template permet de créer la base pour une application métier. A noter que ce template se base au niveau de la présentation sur le template « Silverlight Navigation Application ».

Pour les besoins de cet atelier, nous allons utiliser la base de données « Northwind » qui est installée sur la machine.

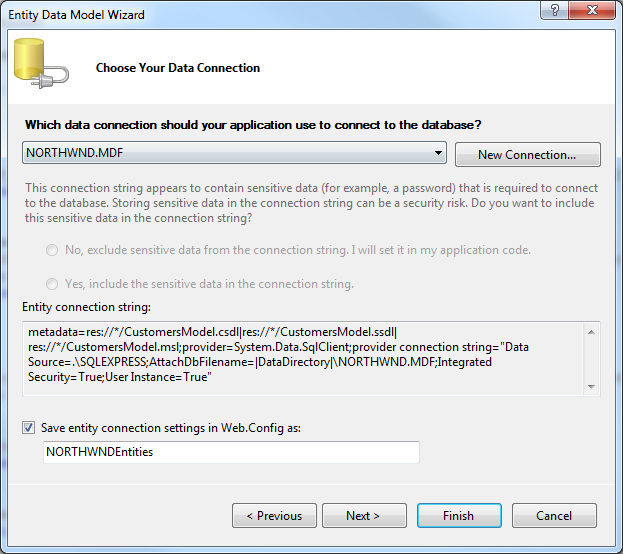
Pour tout ce qui est accès aux données, nous allons utiliser « Entity Framework ». Pour cela faites un clic-droit sur le projet « WinwiseBusinessApplication.Web » puis « Add » et « New Item ». Ensuite, dans le nœud « Data » choisissez « ADO.NET Entity Data Model »  et cliquez sur « Add »:



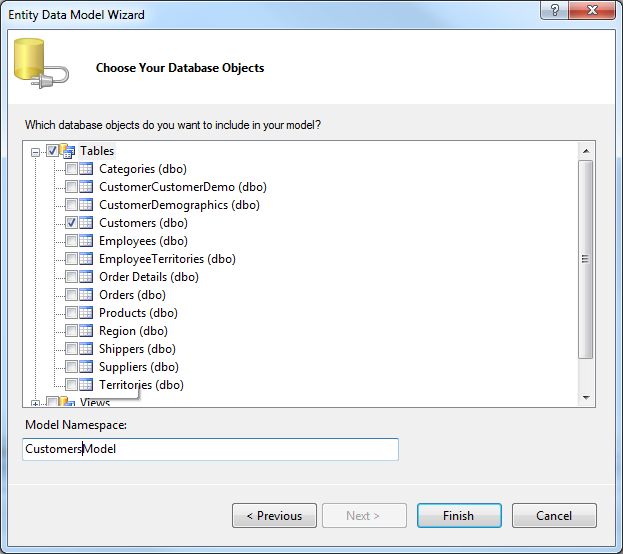
Un autre écran apparait alors vous demandant si vous désirez générer un model vide ou alors à partir d’une base de données, cliquez donc sur « Generate from database »  puis sur le bouton « Next »:



Sur l’écran suivant, vous pouvez paramétrer la connexion à utiliser pour vous connecter à la base de données, cliquez sur « New Connection », entrez « .\sqlexpress » comme nom de serveur et choisissez la base « Northwind » ::

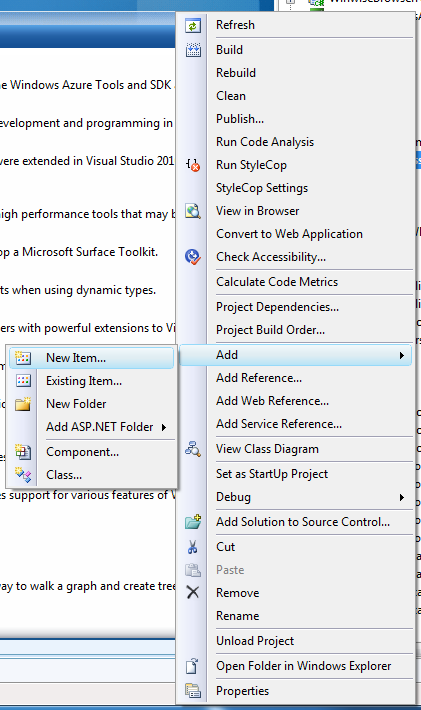


Ensuite vous allez désigner les tables auxquelles vous voulez accéder (nous allons travailler sur la table « Customers ») et également donnez le nom du namespace qui va être utilisé lors de la génération des entités, par exemple « CustomersModel » puis cliquez sur « Finish »:

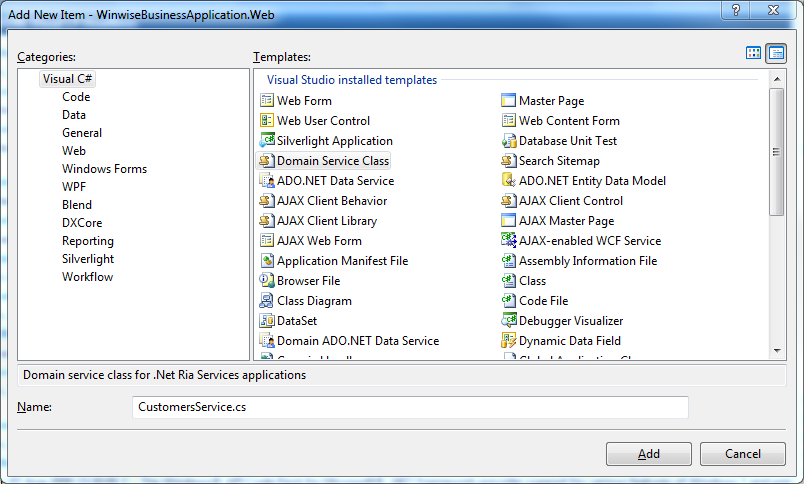


Il est maintenant nécessaire que vous compiliez la solution et ce pour que les différentes entités générées par l’opération précédente soient visible donc l’opération suivante.

Pour que cette couche d’accès aux données soit accessible, nous allons ajouter un élément de type « Domaine Service Class » qui fournira la logique nécessaire pour les opérations de C.R.U.D. Pour cela faites un clic-droit sur le projet « WinwiseBusinessApplication.Web » puis « Add » et enfin « New Item » :



Dans le nœud principal, choisissez « Domain Service Class »  et nommez cet élément, par exemple « CustomersService » :



Une fenêtre apparait alors vous permettant de configurer l’accès aux données, la case « Enable client access » vous permet d’exposer toute la logique de C.R.U.D vers Silverlight, vous pouvez ensuite sélectionnez les différentes entités auxquelles vous voulez accéder puis choisir de générer ou non automatiquement les opérations qui permettront d’ajouter, supprimer ou mettre à jour des éléments dans la table. Enfin la case « Generate associated classes for metadata » va permettre de générer du code qui permettra de créer des règles de validation sur différents champs et ces règles seront répercuter côté client. Cochez toutes les cases, cliquez sur le bouton « OK » et observez les différentes méthodes générées pour nous dans le fichier « CustomersService.cs ». Compilez maintenant la solution afin que le service puisse être visible depuis le client Silverlight (un proxy va être généré puis être inclut dans le projet Silverlight).

Maintenant voyons comment consommer ce service à partir de Silverlight, dans le fichier « Home.xaml » du répertoire « Views » dans le « WinwiseBusinessApplication » et remplacez ce « TextBlock » :

<TextBlock x:Name="ContentText" Style="{StaticResource ContentTextStyle}"

Text="Home page content"/>

Par une « DataGrid » et nommez là, par exemple « CustomersDataGrid » :

<data:DataGrid x:Name="CustomersDataGrid" />

Côté code-behind (fichier « Home.xaml.cs »), commencez par vous abonnez à l’évènement « Loaded » de la page dans le constructeur :

this.Loaded += new RoutedEventHandler(this.OnPageLoaded);

Puis ajoutez le using qui vous permettra d’accéder au proxy généré contenant la logique définit côté serveur :

using WinwiseBusinessApplication.Web;

puis définissez la callback comme ci-dessous :

private void OnPageLoaded(object sender, RoutedEventArgs e)

{

CustomersContext context = new CustomersContext();

this.CustomersDataGrid.ItemsSource = context.Customers;

context.Load(context.GetCustomersQuery());

}

Ainsi on commence par instancier le proxy, puis on définit la source de notre « DataGrid » (les différentes données présentes dans la table « Customer ») et enfin on charge les données en spécifiant la requête à exécuter pour cela. On aurait également très bien pu modifier la requête avec LINQ avant de charger les données pour par exemple ordonner les résultats par ville.

Lancez la solution et observez un début d’application métier n-tier développée comme une application 2-tiers ! :

