Localisation d'une application Silverlight

par Benjamin Roux (Retour aux articles)

Date de publication : 07/12/2009

Dernière mise à jour :

Cet article vous présentera comment localiser une application Silverlight.

1 - Introduction

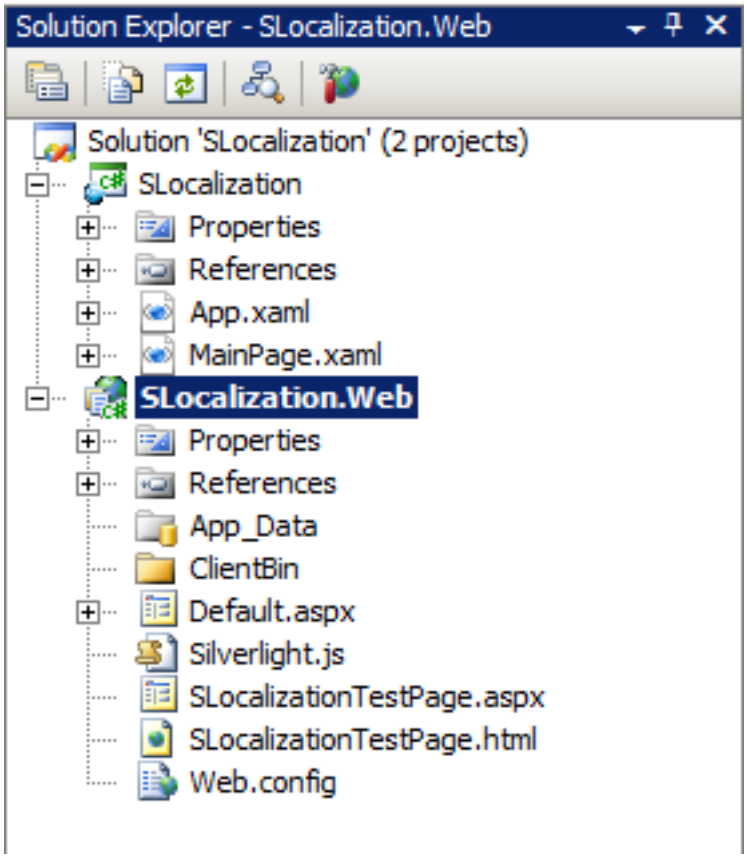
Dans cet article, nous allons voir comment localiser une application Silverlight.

2 - Comment ça marche

La localisation en Silverlight est basée sur l'utilisation de fichiers ressources. Nous avons un fichier ressource par langues supportées.

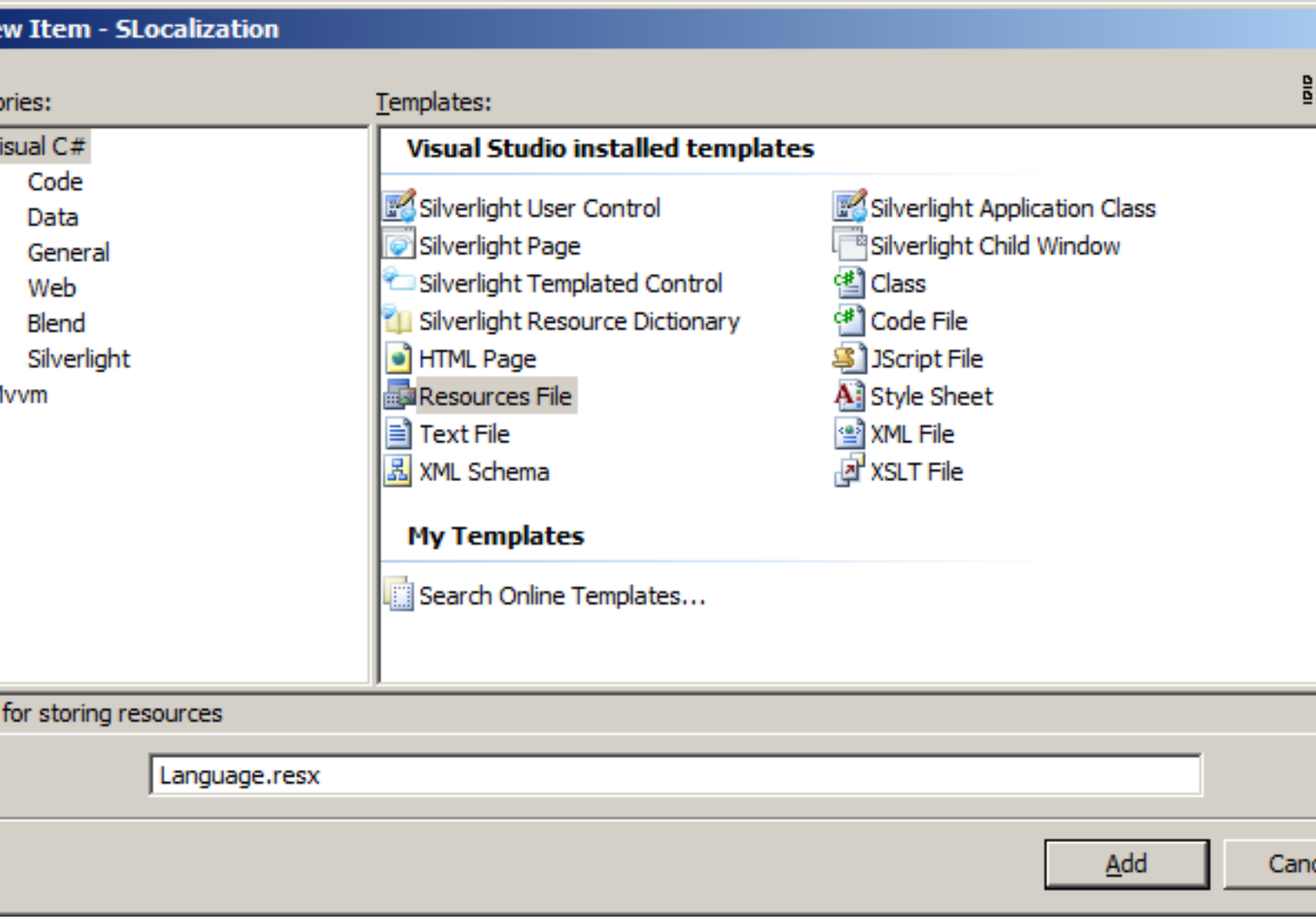
3 - Création des fichiers ressources

Premièrement, nous allons partir d'un projet Silverlight tout simple auquel nous allons rajouter des fichiers *resources*.

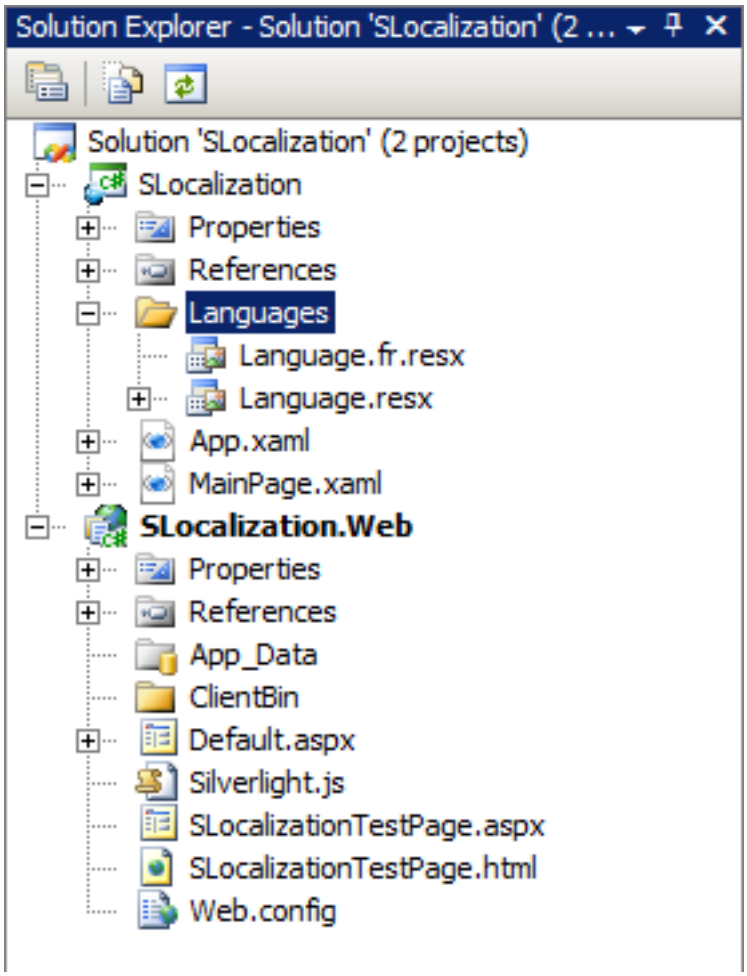


Il faut maintenant rajouter autant de fichiers ressources que nous avons de langages supportés par notre application.

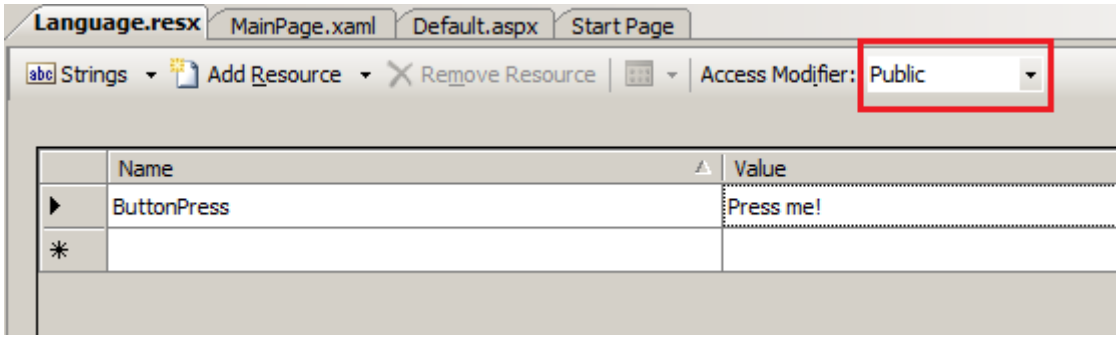
Pour le langage par défaut, nous allons appeler notre fichier *Language.resx*. Pour les autres langues, il faut respecter la convention *Language.xx.resx* où *xx* représente la culture de la langue contenue dans le fichier ressource. Par exemple, si votre fichier ressource contient les textes pour la langue française, le nom de votre fichier sera *Language.fr.resx*. On peut également spécifier pour une culture spécifique, par exemple *Language.fr-ca.resx*.



Votre projet devrait donc ressembler à ça maintenant :



Voici le contenu de *Language.resx*.



Notre application comportera donc un bouton dont le texte sera localisé.

À noter qu'il faille modifier le modificateur d'accès de la classe d'*Internal* à *Public*.

Un *bug* dans l'outil de génération nous oblige à modifier le fichier **.cs** créé à la main.

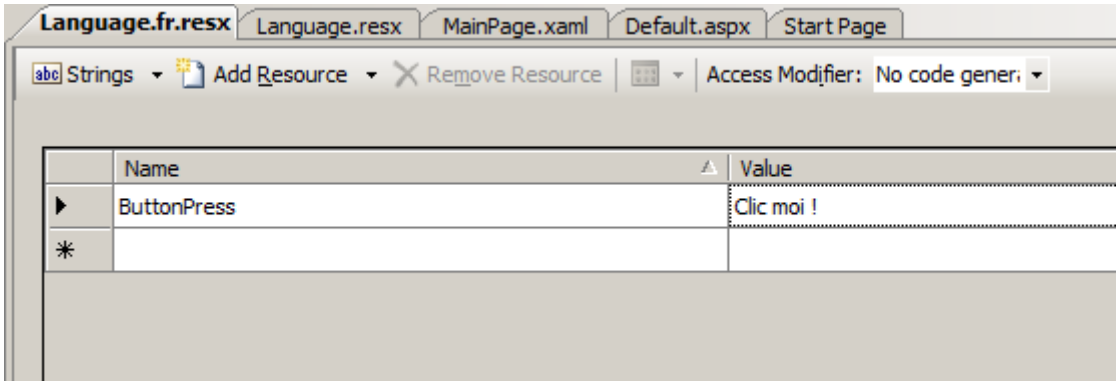
Ouvrez le **.cs** et modifiez le modificateur d'accès du constructeur de *internal* à *public*.

```
internal Language() {
```

devient donc :

```
public Language() {
```

Nous allons maintenant renseigner le fichier *Language.fr.resx*.



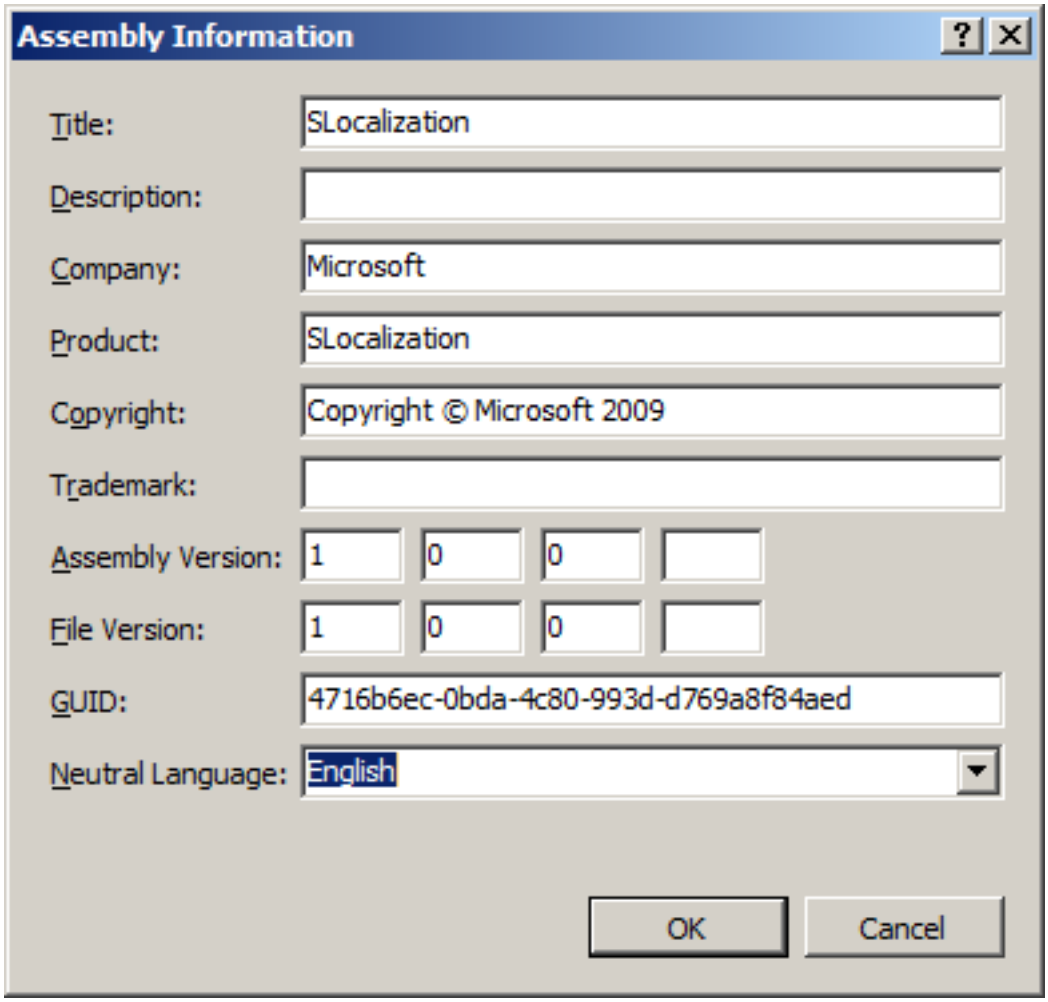
Nous venons de terminer la création de nos fichiers de ressources. Passons à la suite.

4 - Définition de la langue par défaut et des langues supportées

Nous allons maintenant définir la langue par défaut de notre application.

Clic droit sur notre projet -> Propriétés -> Assembly Info.

Il suffit de renseigner la case *Neutral Language*.



Cela rajoute simplement une ligne dans le fichier *AssemblyInfo.cs*.

```
[assembly: NeutralResourcesLanguageAttribute("en")]
```

La définition de langues supportées est un peu plus délicate (en espérant que ce soit amélioré dans une prochaine version).

Il faut tout d'abord décharger le projet : clic droit -> Unload Project.

Puis éditez le fichier **csproj** : clic droit -> Edit xxx.csproj.

Dans le fichier, repérez la section <SupportedCultures> et rajoutez les langues gérées par votre application, séparées par un « ; ».

Vous devriez obtenir quelque chose dans ce genre :

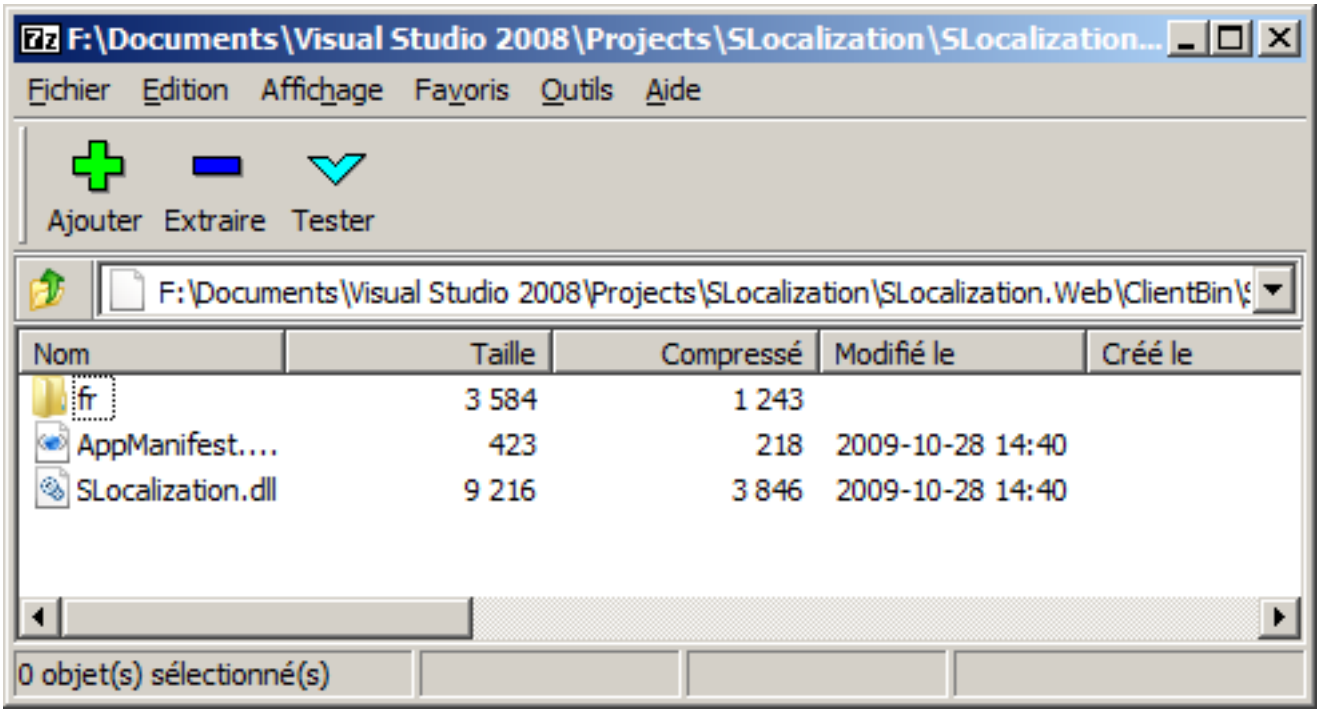
```
<SupportedCultures>fr</SupportedCultures>
```

Cette section pourrait également ressembler à ceci si notre application gérait plusieurs langues.

```
<SupportedCultures>fr;fr-CA;de;ru</SupportedCultures>
```

Rechargeons le projet : clic droit -> Reload Project.

Vous pouvez maintenant réaliser un petit test en compilant votre projet et en regardant le contenu du fichier **xap** généré.



On peut voir que Visual Studio a généré un répertoire **fr** qui contient les ressources liées à la langue française.

5 - Création du XAML localisé

Maintenant que nous avons créé nos ressources, nous allons les appliquer sur notre XAML.

Créons d'abord un simple bouton sans contenu.

```
<Button Width="100" Height="25" />
```

Ajoutons maintenant les références à notre fichier de ressource.

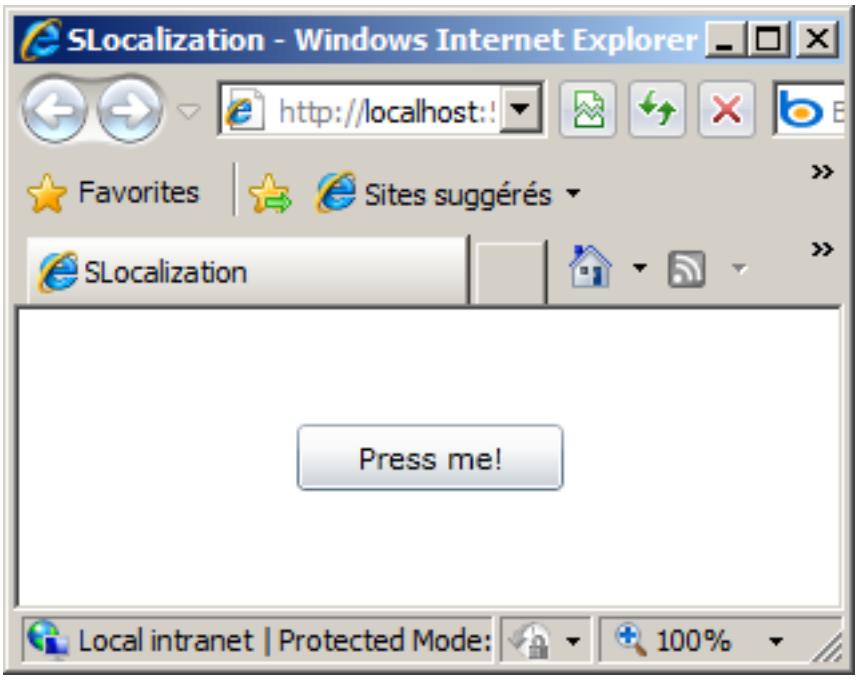
```
<UserControl x:Class="SLocalization.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://
schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:lang="clr-namespace:SLocalization.Languages"
    mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480">
    <UserControl.Resources>
        <lang:Language x:Key="Lang" />
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot">
        <Button Width="100" Height="25" />
    </Grid>
</UserControl>
```

Maintenant, le principe est de *binder* le contenu de notre bouton sur notre texte localisé.

```
<Button Content="{Binding Path=ButtonPress, Source={StaticResource Lang}}" Width="100" Height="25" />
```

On dit que le texte de notre bouton se trouve à la propriété *ButtonPress* de la ressource *Lang* qui n'est ni plus ni moins que notre fichier Resource.

Le résultat :



Testons maintenant en français.

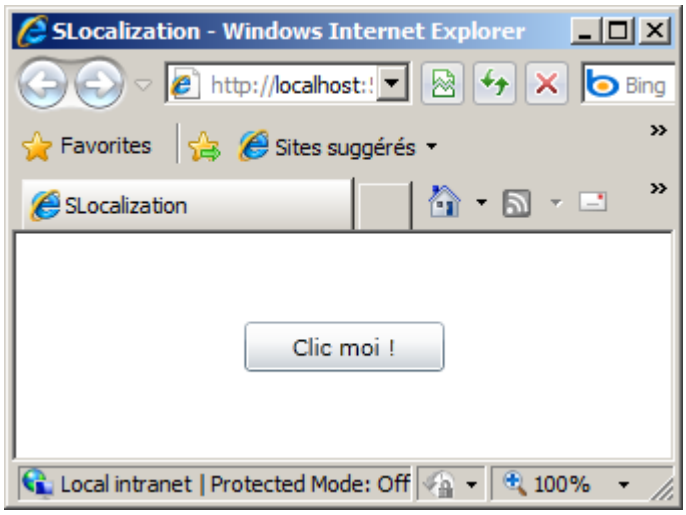
Dans le fichier **App.xaml.cs** :

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    Thread.CurrentThread.CurrentCulture = new CultureInfo("fr");
    Thread.CurrentThread.CurrentUICulture = new CultureInfo("fr");

    this.RootVisual = new MainPage();
}
```

Avant de créer notre application nous spécifions la culture.

Le résultat :



Le contenu de notre bouton est donc maintenant localisé !

6 - Initialisation de la langue

Pour initialiser la langue de l'application, le plus simple reste d'utiliser des paramètres d'initialisation ou bien des paramètres dans l'URL.

Paramètres dans l'URL :

```
private void Application_Startup(object sender, StartupEventArgs e)
{
    if (HtmlPage.Document.QueryString.Keys.Contains("lang"))
    {
        string lang = HtmlPage.Document.QueryString["lang"];
        Thread.CurrentThread.CurrentCulture = new CultureInfo(lang);
        Thread.CurrentThread.CurrentUICulture = new CultureInfo(lang);
    }

    this.RootVisual = new MainPage();
}
```

Utilisation : **http://localhost:55466/SLocalizationTestPage.aspx?lang=fr**

Paramètres d'initialisation :

```
if (e.InitParams.Keys.Contains("lang"))
{
    string lang = e.InitParams["lang"];
    Thread.CurrentThread.CurrentCulture = new CultureInfo(lang);
    Thread.CurrentThread.CurrentUICulture = new CultureInfo(lang);
}
```

Utilisation (dans la création du *plugin* dans la page **aspx** ou **html**) :

```
<param name="initParams" value="lang=fr" />
```

7 - Changement de langage au runtime

Avec la méthode décrite plus haut, la modification de la culture au runtime ne rafraîchit pas l'UI.

Pour obtenir ce résultat, il faut légèrement modifier notre code.

Tout d'abord, nous allons créer une classe *LangHelper* exposant notre fichier ressource. Le petit plus sera l'implémentation de l'interface *INotifyPropertyChanged*.

Le code est donc le suivant :

```
public class LangHelper : INotifyPropertyChanged
{
    private static Language mResource = new Language();

    public Language Resource
    {
        get { return mResource; }
    }

    public void ChangeCulture(string culture)
    {
        Thread.CurrentThread.CurrentCulture = new CultureInfo(culture);
        Thread.CurrentThread.CurrentUICulture = new CultureInfo(culture);
        NotifyPropertyChanged("Resource");
    }

    #region INotifyPropertyChanged Members

    public event PropertyChangedEventHandler PropertyChanged;

    private void NotifyPropertyChanged(string name)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(name));
        }
    }

    #endregion
}
```

On expose notre fichier ressource via une propriété, et créons une méthode permettant de modifier la culture de notre application et déclenchant l'évènement *PropertyChanged*.

Retour dans notre fichier xaml principal qui va subir quelques changements :

```
<UserControl x:Class="SLocalization.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://
schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:lang="clr-namespace:SLocalization.Helpers"
    mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480">
    <UserControl.Resources>
        <lang:LangHelper x:Key="Lang" />
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot">

  <Button Content="{Binding Path=Resource.ButtonPress, Source={StaticResource Lang}}" Width="100" Height="25" />
    </Grid>
</UserControl>
```

Le texte de notre bouton est maintenant toujours *bindé* sur la propriété ButtonPress de notre fichier de ressource, mais cette fois-ci en passant par une propriété intermédiaire.

Maintenant en modifiant la langue via la méthode *ChangeCulture* tous les bindings de notre UI seront rafraîchis.

Cette méthode peut facilement être adaptée dans une application utilisant le pattern MVVM.

8 - Localisation de propriétés non « bindable »

Il puisse arriver que l'on veuille localiser une propriété sur laquelle le *binding* n'est pas possible (une propriété n'étant pas une *Dependency Property* le plus souvent), comme le *header* d'une *DataGridTextColumn*.

Silverlight ne propose pas de méthode type pour ça, mais seulement des « astuces ».

Pour le cas présent, il va falloir passer par du code *behind*.

```
column1.Header = Languages.Language.ButtonPress;
```

Correspondant au code XAML suivant

```
<data:DataGridTextColumn x:Name="column1" . />
```

Le problème ici vient du changement de langue au *runtime*. Il faudra reprendre toutes les propriétés une par une pour remettre la valeur dedans.

On peut bien évidemment passer par une méthode qui fait tout pour nous, mais cela reste moins pratique que le *binding*.

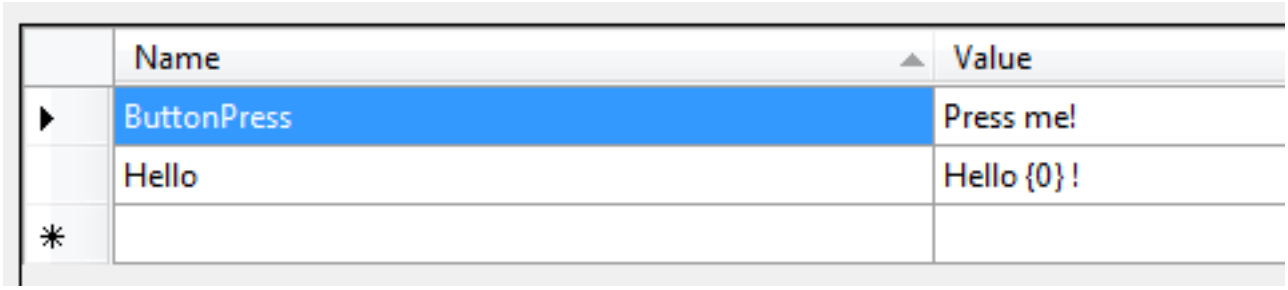
9 - Formatage de chaîne localisée

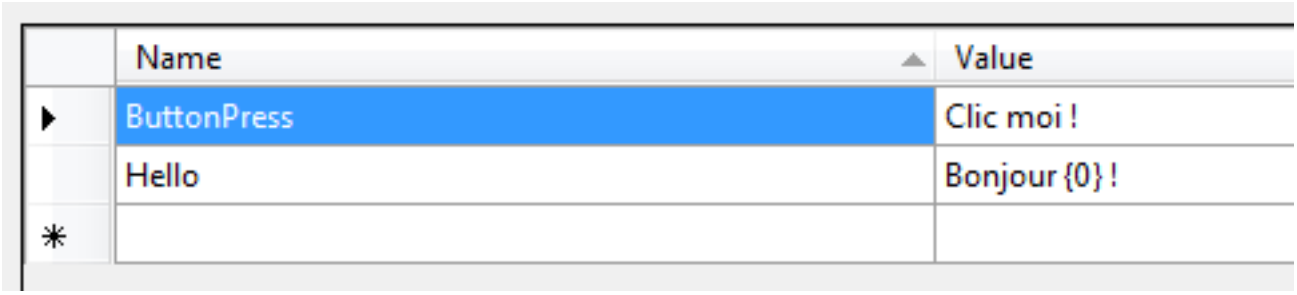
Il puisse arriver que l'on veuille localiser du texte contenant des paramètres.

Exemple, je veux afficher un *TextBlock* contenant un message de bienvenue pour l'utilisateur courant.

Selon la langue je veux donc afficher ***Bonjour*** ou ***Hello*** suivi du nom de l'utilisateur.

Pour ce faire, nous allons rajouter une entrée dans chaque fichier ressource.





Maintenant pour afficher notre texte, nous allons simplement passer par un *converter*.

```
public class TextFormatterConverter : IValueConverter
    {
        #region IValueConverter Members

        public object Convert(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
        {
            return string.Format(value.ToString(), parameter.ToString());
        }

        public object ConvertBack(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }

        #endregion
    }
```

Maintenant lors de notre *binding*, il faut simplement spécifier notre *converter* ainsi que son paramètre.

```
<UserControl x:Class="SLocalization.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://
schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:lang="clr-namespace:SLocalization.Helpers"
    xmlns:converter="clr-namespace:SLocalization.Converters"
    mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480">
```

```
    <UserControl.Resources>
        <lang:LangHelper x:Key="Lang" />
        <converter:TextFormatterConverter x:Key="TextFormatter" />
    </UserControl.Resources>
    <Grid x:Name="LayoutRoot">
        <TextBlock Text="{Binding Path=Resource.Hello,
                                  Source={StaticResource Lang},
                                  Converter={StaticResource TextFormatter},
                                  ConverterParameter=Benjamin}" />
    </Grid>
</UserControl>
```

Pour utiliser un paramètre variable, on peut définir le *binding* en code au lieu de passer par le XAML.

10 - Conclusion

La localisation d'une application Silverlight est, comme vous avez pu le voir, assez facile. La méthode proposée est quand même assez basique.

Tout ce processus de localisation devrait être amélioré dans une future version de Silverght et Xaml 2009.

Remerciements

Merci à toute l'équipe .NET pour leurs relectures et corrections.

Merci également à **Wachter** pour sa relecture et correction finale.