Aller plus loin avec WPF

WPF est une technologie dont la structure permet de pousser très loin la personnalisation d'une interface graphique. Visual Studio est fourni avec de nombreux outils, dont **Blend**, qui se révèle être une aide précieuse pour l'élaboration des interfaces graphiques.

WPF permet aussi, grâce aux **bindings** notamment, un **découplage** presque parfait entre la partie visuelle d'une application et le code métier qui lui est associé. La mise en œuvre de cette séparation est le plus souvent effectuée à l'aide du patron de conception **MVVM** (*Model* - *View* - *ViewModel*).

1. Introduction à l'utilisation de Blend

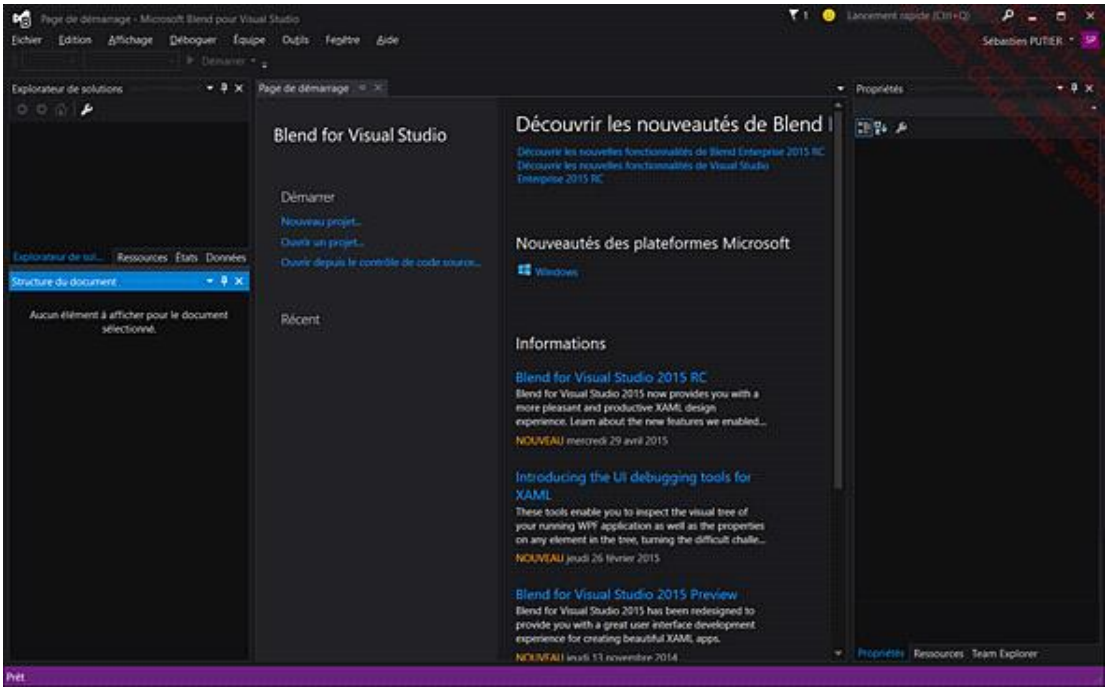
Blend pour Visual Studio est un outil apparu initialement sous le nom d'**Expression Blend** dans la suite Expression de Microsoft en 2007. Il est depuis ses débuts destiné à produire du code XAML pour les applications **WPF**, puis son champ d'action a été élargi pour inclure **Silverlight** et **Windows Phone 7**. Après sa version 4, il a été introduit dans la gamme d'outils livrés avec Visual Studio, et l'ensemble de technologies qu'il supporte a évolué de manière à inclure les applications **Windows Phone 8** et **Windows 8**. Aujourd'hui, Blend peut même être utilisé pour créer des interfaces d'application Windows Phone ou Windows Store en **HTML5** !

Blend peut être décrit comme étant une version orientée design de Visual Studio. En effet, Visual Studio possède un concepteur visuel basique et des dizaines d'outils destinés à aider le développeur dans sa tâche d'écriture de code. Blend est quant à lui constitué d'un éditeur de code ainsi que d'un concepteur visuel complété par de nombreux outils graphiques. Le fossé entre Visual Studio et Blend se comble de plus en plus à la sortie de chaque nouvelle version, la dernière apportant le support d'IntelliSense et une interface très proche de celle de Visual Studio, ce qui favorise une courbe d'apprentissage rapide.

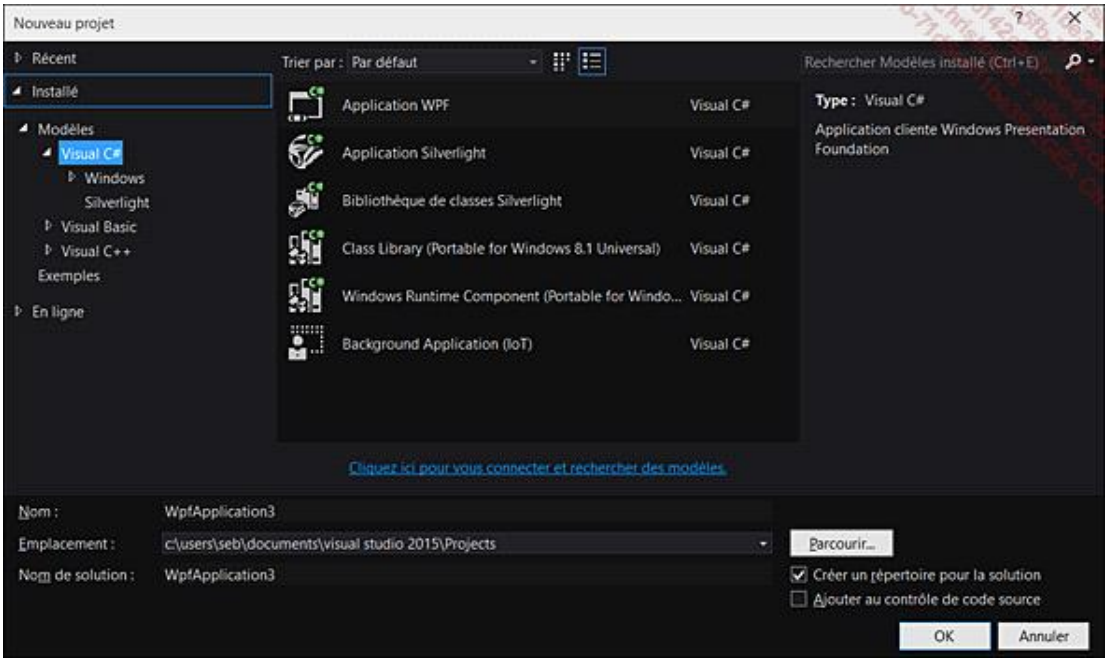
a. L'interface

L'interface générale de Blend 2015 est identique à l'interface de Visual Studio, à la différence près que le thème par défaut pour cet outil est spécifique à Blend et est à dominante noire, de manière à visualiser au mieux l'interface graphique en cours d'édition.

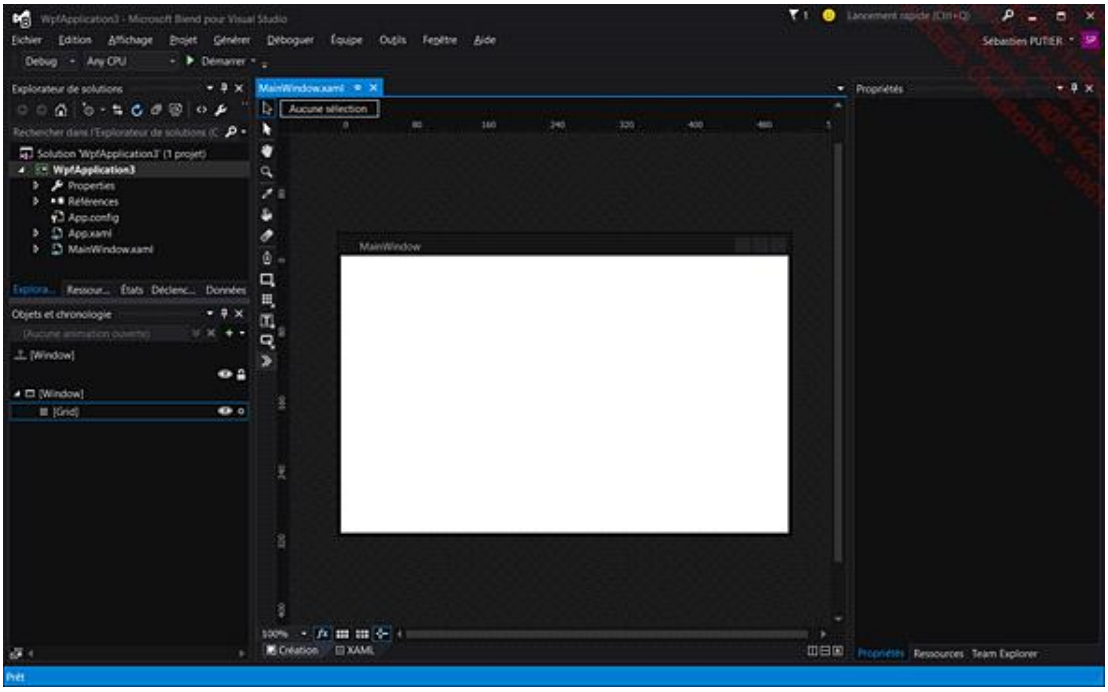
Au lancement de Blend, l'écran principal est affiché, et une fenêtre modale bleue permettant la création d'un nouveau projet ou l'ouverture d'un projet existant est placée au premier plan.



Le choix de l'option **Nouveau projet...** sur l'écran de démarrage ouvre la fenêtre suivante. Celle-ci permet de choisir le type de projet sur lequel travailler. De nombreuses possibilités sont proposées, parmi lesquelles des applications WPF, Windows Store, Silverlight ou Windows Phone.



Le choix de l'option **Application WPF** amène à l'écran suivant, dont l'organisation devrait vous rappeler Visual Studio. Il présente par défaut trois zones réparties sur la gauche, au centre et à droite.

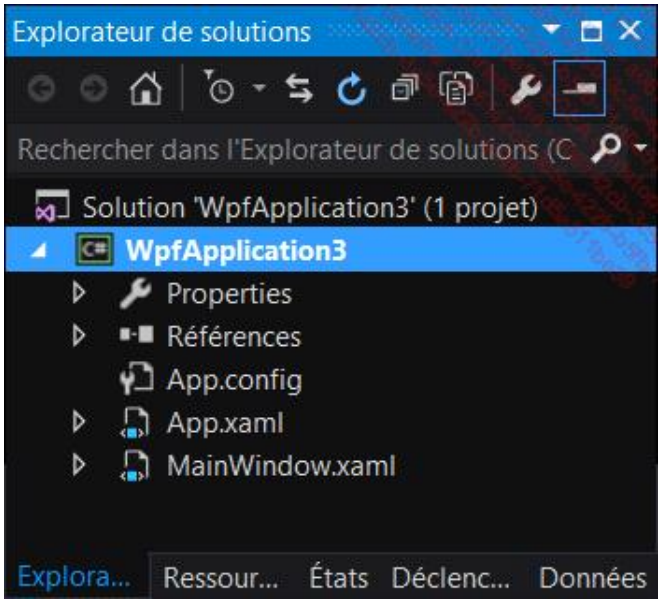


Au centre, on trouve une surface de travail représentant la fenêtre en cours de développement. Il est possible de zoomer aisément à l'aide de la molette de la souris afin de mieux visualiser les éléments qui seront placés dans cette zone.

Différentes fenêtres ancrées sur la gauche et la droite contiennent des outils permettant de visualiser ou modifier certains éléments de l'application.

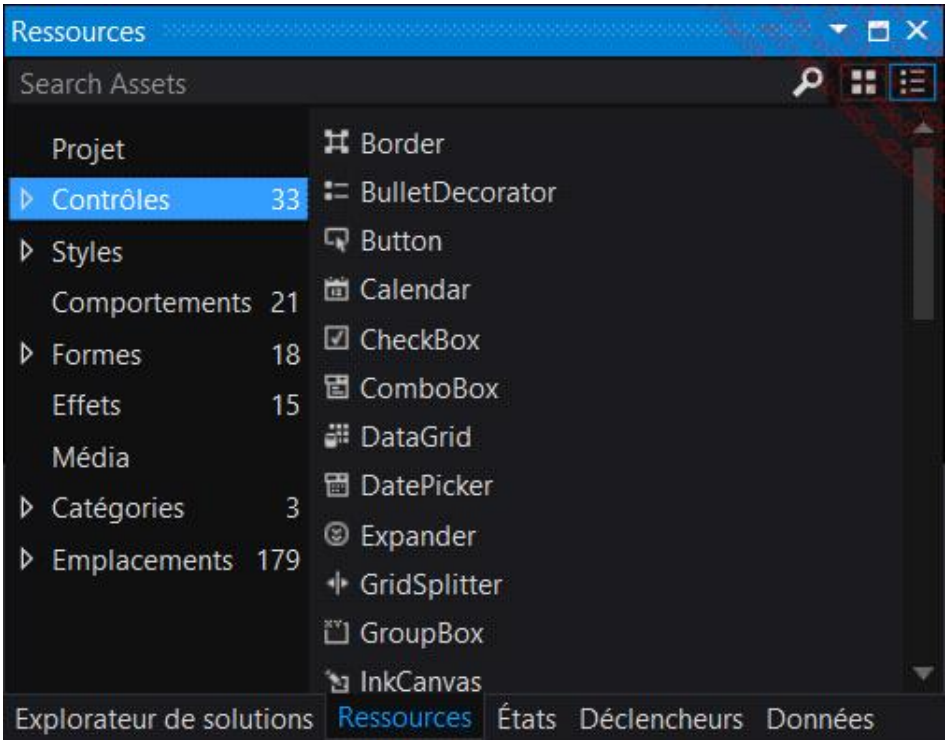
Explorateur de solutions

Comme dans Visual Studio, cet outil contient un arbre représentant la structure du projet en cours d'édition. Il permet entre autres de naviguer dans les projets et d'ouvrir les fichiers qu'ils contiennent.



Ressources

Les différents éléments listés dans cet onglet sont les contrôles (majoritairement graphiques), styles visuels, comportements ou éléments graphiques (images, vidéos...) utilisables dans l'application WPF. Ils sont regroupés selon différentes catégories représentant des emplacements physiques ou des groupements logiques.



États

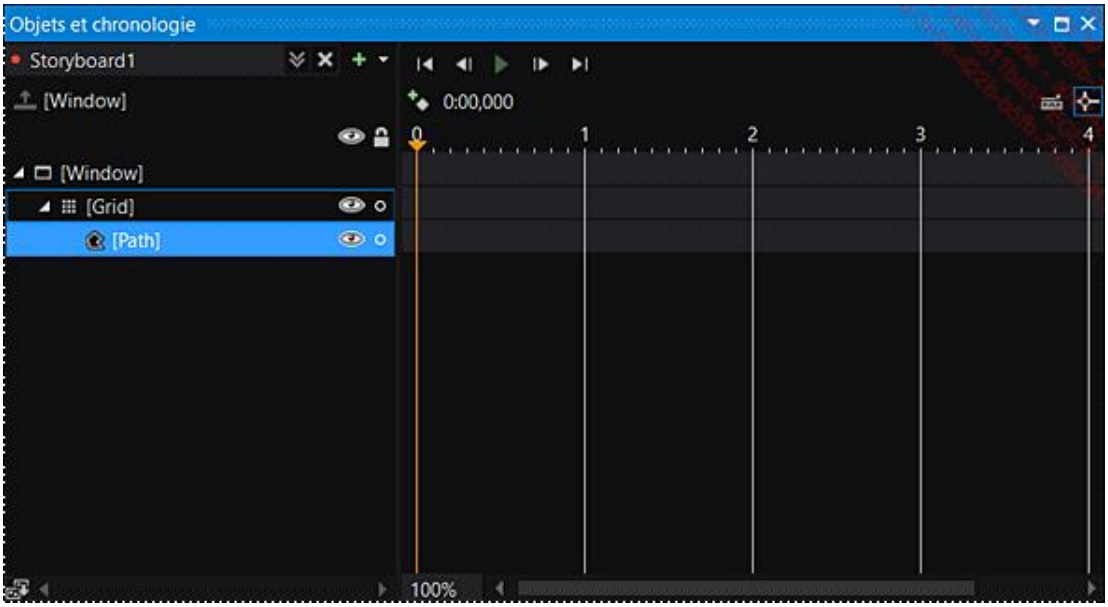
Les contrôles graphiques peuvent avoir différents états visuels définis en XAML. Chacun regroupe un ensemble de valeurs de propriétés à appliquer lorsque l'état visuel du contrôle est modifié.

Déclencheurs

Il est possible de déclarer avec XAML des modifications devant être appliquées lorsqu'une ou plusieurs conditions sont réunies. Ces modifications sont regroupées dans des déclencheurs (ou Triggers). L'onglet **Déclencheurs** permet de créer et gérer ces objets sans utiliser directement XAML.

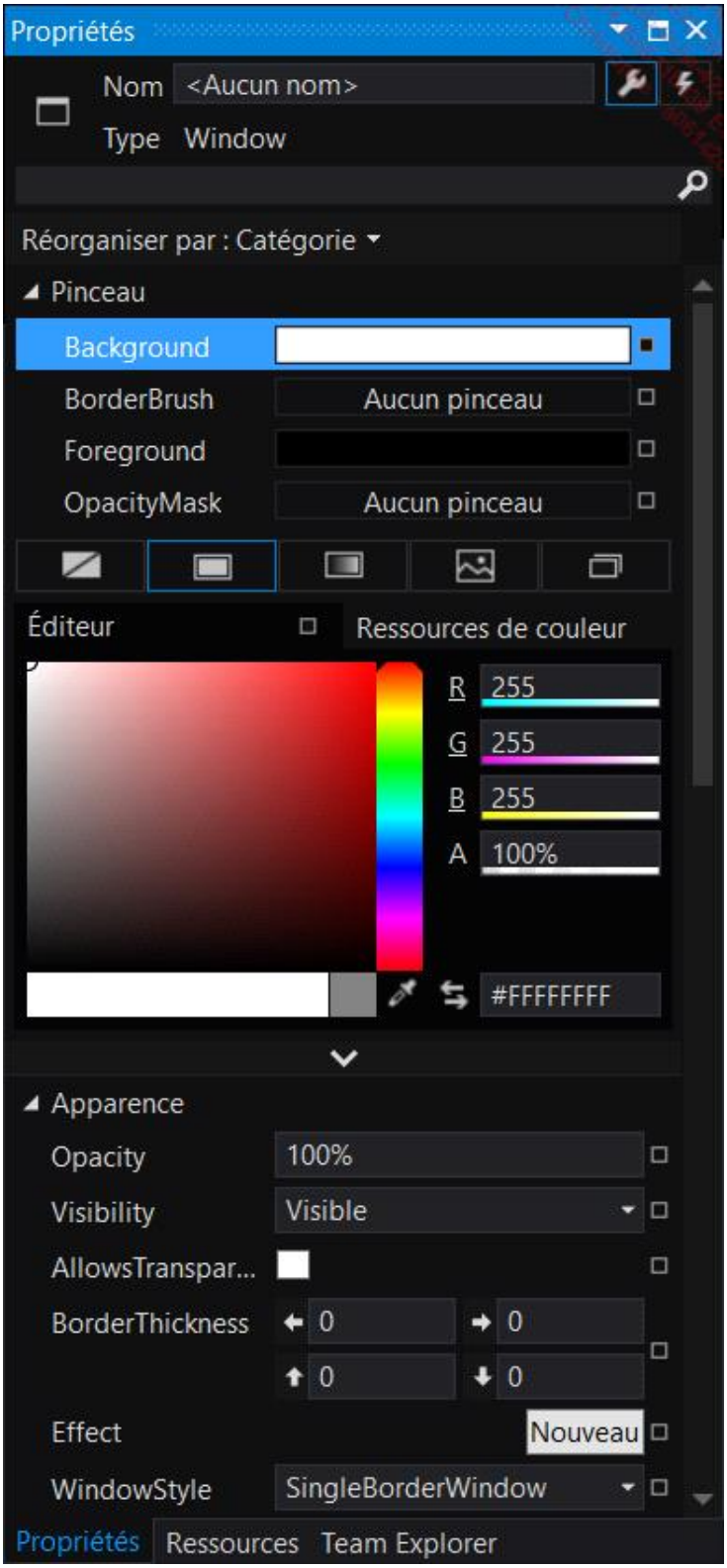
Objets et chronologie

Cet onglet a deux objectifs : la visualisation de l'arbre logique formé par les contrôles d'une fenêtre et la génération de scénarios pour des animations (appelés communément *Storyboards*). Les animations pouvant modifier des propriétés de plusieurs contrôles, cet onglet permet de visualiser ces changements sous la forme d'un chronogramme (diagramme suivant une ligne temporelle).



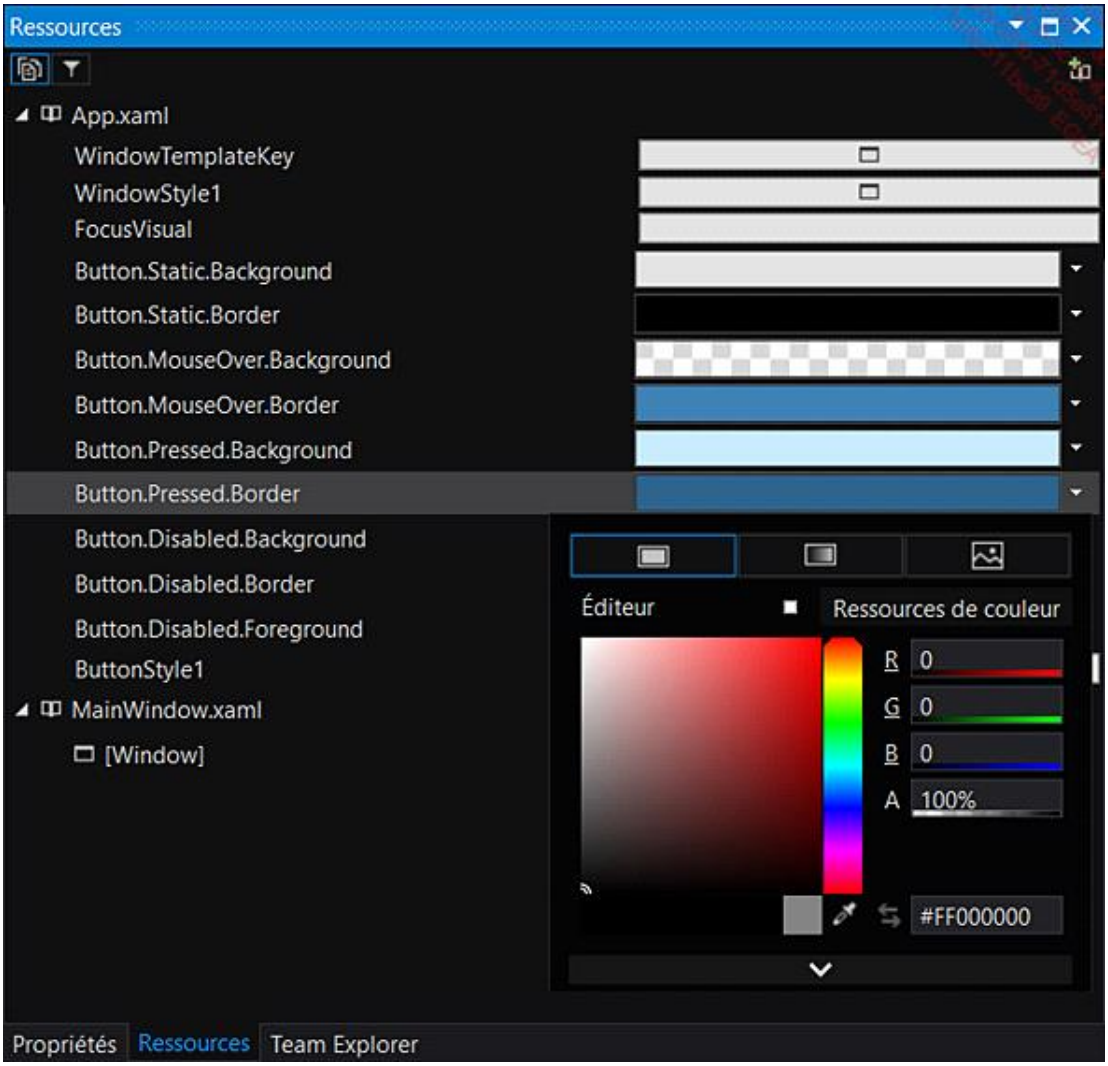
Propriétés

Cet onglet permet la modification directe des propriétés du contrôle sélectionné dans l'onglet **Objets et chronologie**. Il est similaire à l'onglet **Propriétés** disponible dans Visual Studio, mais contient des fonctionnalités supplémentaires, comme la sélection d'une ressource dans une liste de toutes les ressources accessibles.



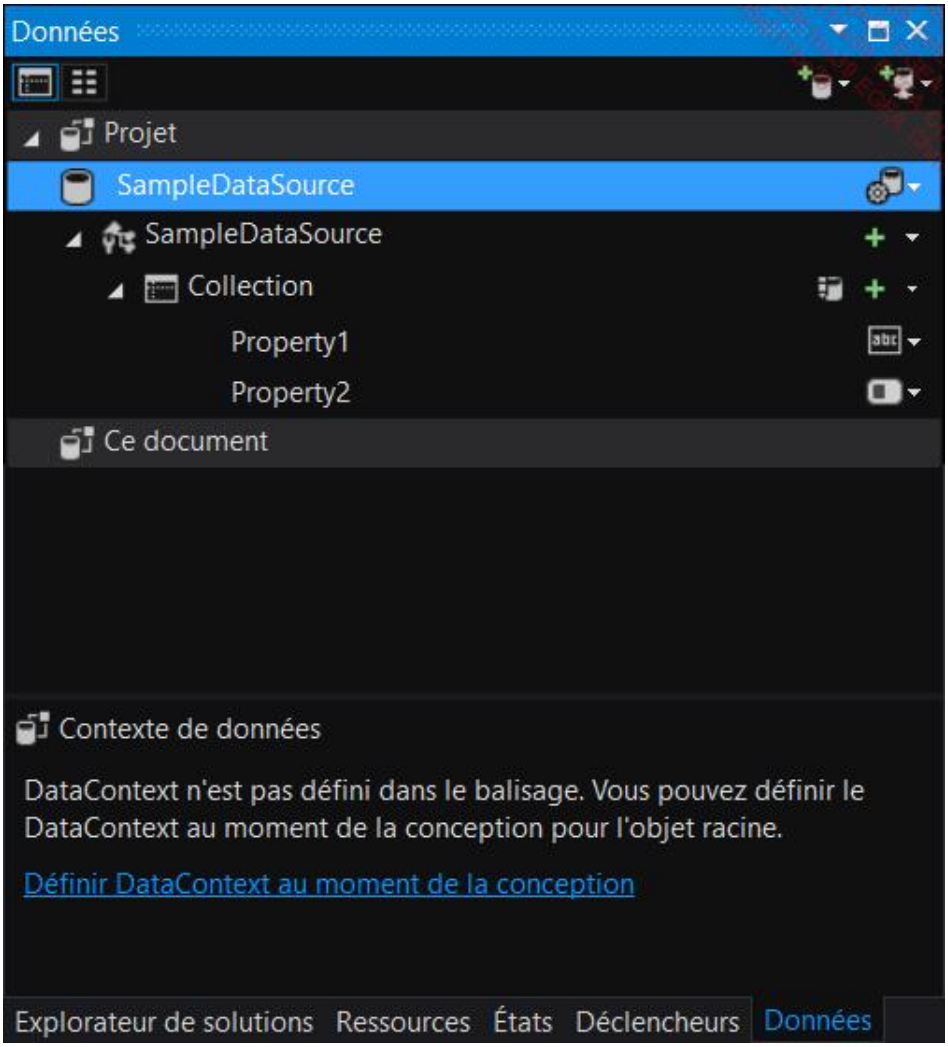
Ressources

Cette seconde fenêtre nommée **Ressources** permet de lister les différentes ressources XAML du projet courant, qu'elles soient graphiques, textuelles, numériques ou de n'importe quel autre type. Il est possible, si leur type est supporté par Blend, de les éditer directement.



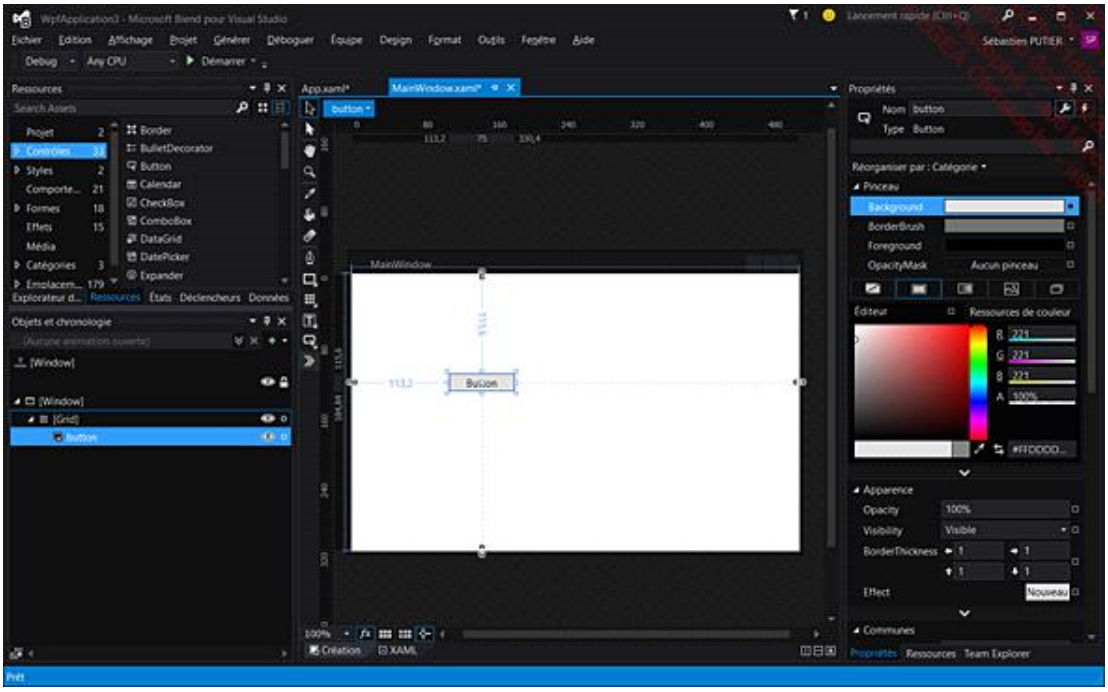
Données

Ce dernier onglet permet de créer des échantillons de données utilisables pendant la conception visuelle de l'application. Les contrôles peuvent être remplis avec ces données afin qu'il ne soit pas nécessaire de lancer l'application pour vérifier l'effet d'une modification.



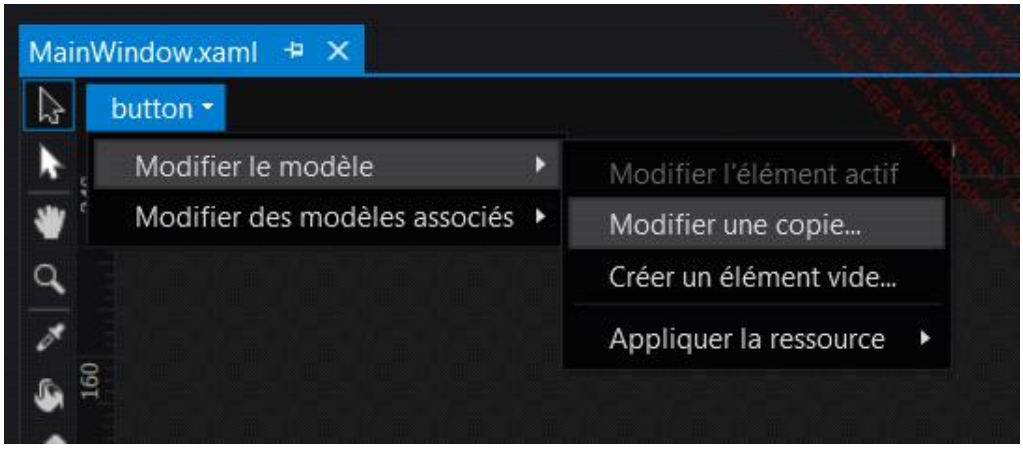
b. Ajout et modification de contrôles visuels

L'ajout de contrôles sur la surface de dessin est effectué par un glisser-déposer de l'onglet **Composants** vers l'emplacement de destination. Une fois créé, le contrôle apparaît dans l'onglet **Objets et chronologie** et peut être sélectionné.

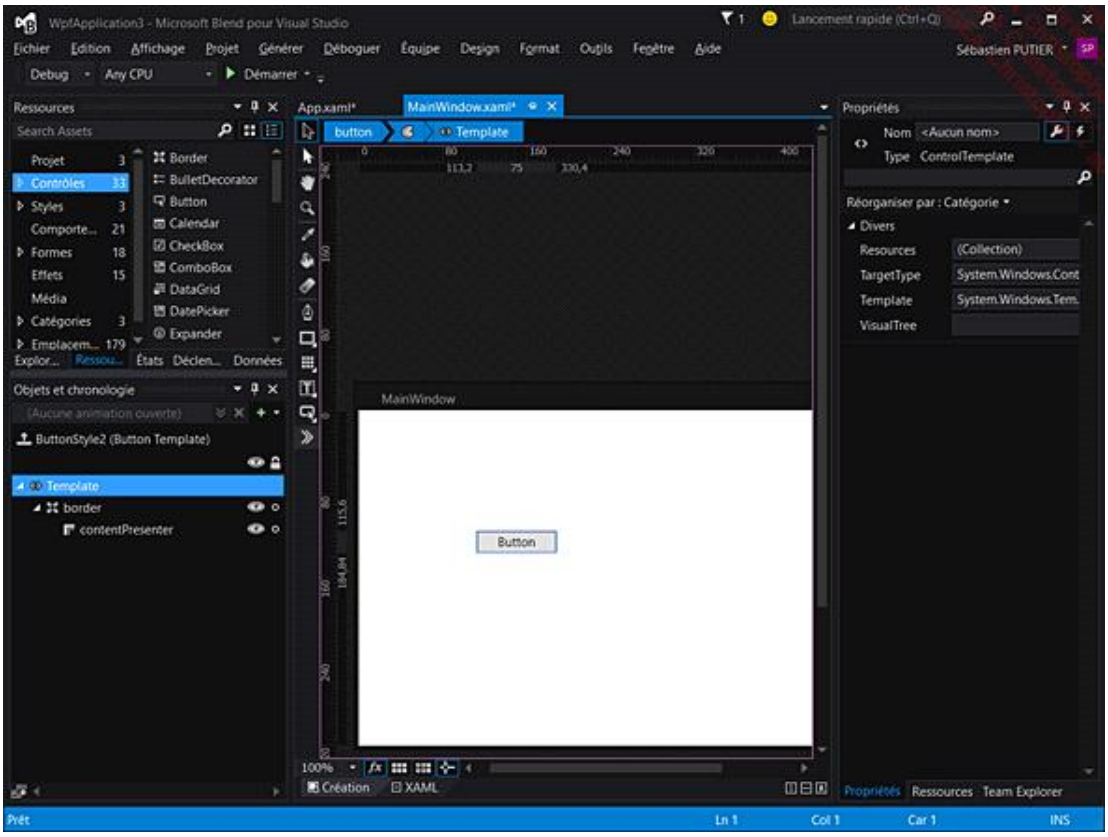


Lorsqu'il est sélectionné, l'onglet **Propriétés** liste les propriétés du contrôle et permet leur édition. De la même manière, les onglets **États** et **Déclencheurs** sont mis à jour pour refléter les états existants du contrôle et les déclencheurs qui lui sont associés.

Lorsque le contrôle sélectionné possède un template, il est possible de l'éditer à l'aide du menu situé au-dessus de la surface de dessin.



Sélectionner l'option **Modifier le modèle - Modifier une copie** modifie l'arbre logique de l'onglet **Objets et chronologie** : Blend est placé en mode "édition d'un template", et les différents onglets composant l'interface sont actualisés pour refléter les données du template du contrôle.



L'utilisation de Blend comme outil d'édition graphique peut permettre un gain de temps considérable lorsqu'elle est maîtrisée. Je ne peux que vous conseiller de passer du temps à apprendre comment éditer les différents éléments composant une interface WPF à l'aide de Blend. Cet outil est véritablement un ami du développeur, et bien que son utilisation ait ici seulement été survolée, vous pouvez déjà probablement imaginer toute sa puissance.

2. Introduction à MVVM

MVVM (*Model - View - ViewModel*) est un patron de conception facilitant la séparation de l'interface graphique et du code contenant la logique de l'application. Il est apparu avec l'avènement de WPF et est aujourd'hui indissociable de la notion de **binding**. C'est en effet grâce à l'existence de ces liaisons de données que la séparation peut être aussi

efficace entre les vues et les données qu'elles affichent.

a. Présentation

Ce modèle de développement définit une structure applicative découpée en trois couches distinctes : `View`, `Model` et `ViewModel`.

View

Cette couche contient le code relatif à l'interface graphique. Elle est composée des fichiers de code XAML et des fichiers de code-behind C# associés. Aucune donnée autre que spécifique à l'interface ne devrait être manipulée dans cette couche de l'application. Elle est liée à la logique de l'application à travers les bindings.

Model

On trouve dans cette couche la définition et la manipulation des données de l'application. Les traitements purement définis par le métier de l'application se trouvent tous dans cette couche.

ViewModel

Un `ViewModel` (modèle de vue) est une adaptation de données issues de la couche `Model` dans l'objectif de leur présentation par la couche `View`. On peut aussi trouver ici la définition des commandes utilisées par l'interface graphique.

b. Les interfaces INotifyPropertyChanged et INotifyCollectionChanged

WPF permet de lier une ou plusieurs propriétés d'un contrôle à des données d'un `ViewModel` à l'aide des expressions de binding.

Celles-ci sont évaluées initialement à la fin du chargement de la fenêtre. Ainsi, si une propriété du `ViewModel` est valorisée durant ce chargement, sa valeur sera bien passée aux objets qui lui sont liés. Mais si cette valeur est modifiée par l'intermédiaire de code C# après la première évaluation de l'expression de binding, la modification ne sera pas répercutée.

Pour que le comportement des bindings soit constant, il est nécessaire que le `ViewModel` implémente l'interface `INotifyPropertyChanged`. En effet, WPF utilise l'événement `PropertyChanged` des objets implémentant cette interface afin de savoir qu'une modification a eu lieu et, si nécessaire, la répercuter. L'événement `PropertyChanged` doit bien évidemment être déclenché à partir du `ViewModel` lorsqu'une valeur de propriété est modifiée.

Le cas des collections est légèrement différent. Les éléments contenus dans une collection peuvent être modifiés sans que la valeur intrinsèque de la collection ne change. Dans ce cas, l'utilisation de l'interface `INotifyPropertyChanged` ne résout donc pas le problème : c'est l'interface `INotifyCollectionChanged` qui prend le relais pour indiquer une modification d'élément "enfant". Les collections implémentant cette interface déclenchent un événement `CollectionChanged` lorsqu'un changement se produit sur l'un de leurs éléments. Le moteur de WPF intercepte cet événement et propage l'information jusqu'aux contrôles liés à la collection.

Une seule collection implémentant cette interface est disponible nativement dans le framework .NET : `ObservableCollection<T>`.

c. Commandes

Une commande représente une action qui peut être fournie à certains contrôles à l'aide d'une expression de binding. Il s'agit d'une encapsulation d'un ou deux délégués. Le premier définit le traitement à effectuer au déclenchement de l'action. Le second, optionnel, définit une fonction qui renvoie une valeur booléenne. Celle-ci indique s'il est possible d'exécuter l'action dans le contexte courant.

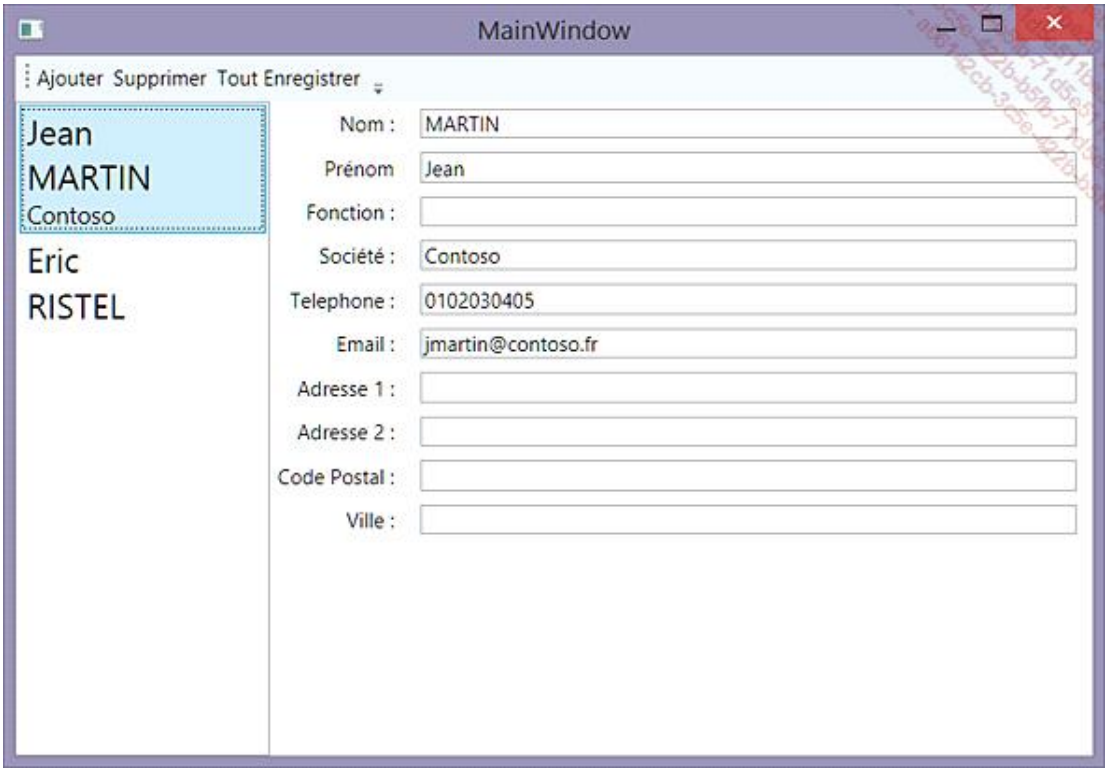
Les contrôles `Button` et `MenuItem` permettent notamment l'utilisation de commandes à travers leur propriété `Command`. Dans ces deux cas, la commande est exécutée lorsque l'utilisateur clique sur le contrôle.

Les commandes sont très utilisées dans un contexte impliquant MVVM, car elles permettent de déplacer aisément des traitements non liés à l'interface hors de la couche `View`.

d. Mise en œuvre

Pour étudier la mise en œuvre des concepts liés au patron de conception MVVM, nous allons développer une application de type "carnet d'adresses". Cette étude de cas vous permettra de voir l'utilisation des bindings, la mise en place d'un `ViewModel` et enfin la logique à adopter pour maintenir la séparation entre l'interface graphique et la logique de l'application. Le stockage des informations sera effectué dans un fichier texte utilisant la tabulation comme séparateur entre deux données d'un enregistrement. Après l'étude des chapitres portant sur ADO.NET et LINQ, vous serez en mesure de créer votre propre couche de persistance pour le stockage des informations en base de données.

L'application que nous allons développer ensemble aura l'aspect suivant :



Structure de l'application

La première étape du développement correspond à la création et la structuration du projet.

- Commencez par créer un nouveau projet d'**Application WPF** nommé **CarnetAdressesMVVM**.

L'architecture MVVM propose comme convention de structurer l'application en créant trois dossiers permettant d'isoler chacune de ses couches.

- Créez trois dossiers dans le projet et nommez-les **View**, **Model** et **ViewModel**.
- Déplacez le fichier `MainWindow.xaml` dans le dossier `View` avec une opération de glisser-déposer.

Afin de respecter les conventions de développement de C#, il faut modifier l'espace de noms associé à la classe `MainWindow`.

- Dans `MainWindow.xaml.cs`, remplacez la ligne :

```
namespace CarnetAdressesMVVM
```

par :

```
namespace CarnetAdressesMVVM.View
```

- Dans `MainWindow.xaml`, reportez la modification de l'espace de noms en modifiant le nom de classe associé au code XAML.

```
<Window x:Class="CarnetAdressesMVVM.MainWindow"
```

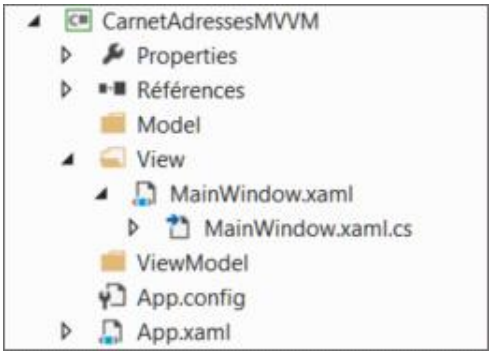
devient :

```
<Window x:Class="CarnetAdressesMVVM.View.MainWindow"
```

- Dans le fichier `App.xaml`, modifiez l'attribut `StartupUri` de la balise `Application` de manière à refléter le nouveau chemin de la page dans l'arborescence de l'application.

```
StartupUri="View/MainWindow.xaml"
```

À ce stade, la structure de l'application dans l'**explorateur de solutions** est la suivante :



La couche Model

Cette couche a pour responsabilité la structuration et l'enregistrement des données dans une base de données, dans un fichier ou, dans le cas présent, en mémoire.

Il convient tout d'abord de créer un type de données permettant de stocker les informations relatives à un contact : nom, prénom, adresse, etc.

- Ajoutez une classe nommée `ContactModel` dans le dossier `Model` du projet.

```
internal class ContactModel
{
    public string Prenom { get; set; }
    public string Nom { get; set; }
    public string Telephone { get; set; }
    public string Email { get; set; }
    public string Adresse { get; set; }
    public string Fonction { get; set; }
    public string Societe { get; set; }
}
```

Il est également nécessaire, pour une séparation maximale des responsabilités, de créer une classe dont l'objectif est le stockage et la récupération des différents contacts enregistrés.

- Ajoutez une classe nommée `GestionnaireDonnees` dans le dossier `Model` du projet.

Cette classe lit et enregistre les données dans le fichier *contacts.txt* enregistré dans le même dossier que l'exécutable.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;

namespace CarnetAdressesMVVM.Model
{
    internal class GestionnaireDonnees
    {
        private const string fichierDonnees = "contacts.txt";

        public Contact[] LireTousLesEnregistrements()
```

```
        {
            if (!File.Exists(fichierDonnees))
                return new Contact[0];

            string[] enregistrements =
File.ReadAllLines(fichierDonnees);

            Contact[] resultat = new
Contact[enregistrements.Length];

            for (int i = 0; i < enregistrements.Length; i++)
            {
                string[] champs = enregistrements[i].Split('\t');

                Contact contact = new Contact
                {
                    Prenom = champs[0],
                    Nom = champs[1],
                    Fonction = champs[2],
                    Societe = champs[3],
                    Telephone = champs[4],
                    Email = champs[5],
                    Adresse1 = champs[6],
                    Adresse2 = champs[7],
                    CodePostal = champs[8],
                    Ville = champs[9],
                };
                resultat[i] = contact;
            }

            return resultat;
        }

        public void Enregistrer(IEnumerable<Contact> contacts)
        {
            StringBuilder builder = new StringBuilder();

            foreach (Contact c in contacts)
            {
                string enregistrement =
string.Format("{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}\t{7}\t{8}\t{9}",
c.Prenom, c.Nom, c.Fonction, c.Societe, c.Telephone, c.Email,
c.Adresse1, c.Adresse2, c.CodePostal, c.Ville);

                builder.AppendLine(enregistrement);
            }


            File.WriteAllText(fichierDonnees, builder.ToString());
        }
    }
}
```

Passons à présent à la création de l'interface graphique.

Modifiez le fichier `MainWindow.xaml` de manière à avoir une barre d'outils pour les trois boutons d'action, une `ListBox`, et enfin une `Grid` contenant tous les champs éditables.

```
<Window x:Class="CarnetAdressesMVVM.View.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="450" Width="650" >

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="150" />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <!-- Boutons d'action (Ajouter, Supprimer, Tout
enregistrer) -->
        <ToolBarTray Grid.Row="0" Grid.Column="0"
Grid.ColumnSpan="2">
            <ToolBar>
                <Button Content="Ajouter" />
                <Button Content="Supprimer" />
                <Button Content="Tout enregistrer" />
            </ToolBar>
        </ToolBarTray>

        <!-- Liste des contacts -->
        <ListBox x:Name="listBoxContacts" Grid.Row="1"
Grid.Column="0" />

        <!-- Détail du contact sélectionné -->
        <Grid Margin="0" Grid.Row="1" Grid.Column="1">
            <Grid.Resources>
                <Style TargetType="Label">
                    <Setter Property="Margin" Value="0,0,10,0" />
                    <Setter Property="VerticalAlignment"
Value="Center" />
                    <Setter Property="HorizontalAlignment"
Value="Right" />
                </Style>
                <Style TargetType="TextBox">
                    <Setter Property="Margin" Value="0,0,10,0" />
                    <Setter Property="VerticalAlignment"
Value="Center" />
                </Style>
            </Grid.Resources>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
```

```
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto"/>
                <RowDefinition/>
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <Label Content="Nom :" Grid.Row="1" />
            <TextBox Grid.Column="1" Grid.Row="1" />
            <Label Content="Prénom" Grid.Row="2" />
            <TextBox Grid.Column="1" Grid.Row="2" />
            <Label Content="Fonction :" Grid.Row="3"/>
            <TextBox Grid.Column="1" Grid.Row="3" />
            <Label Content="Société :" Grid.Row="4"/>
            <TextBox Grid.Column="1" Grid.Row="4" />
            <Label Content="Telephone :" Grid.Row="5" />
            <TextBox Grid.Column="1" Grid.Row="5" />
            <Label Content="Email :" Grid.Row="6" />
            <TextBox Grid.Column="1" Grid.Row="6" />
            <Label Content="Adresse 1 :" Grid.Row="7" />
            <TextBox Grid.Column="1" Grid.Row="7" />
            <Label Content="Adresse 2 :" Grid.Row="8" />
            <TextBox Grid.Column="1" Grid.Row="8" />
            <Label Content="Code Postal :" Grid.Row="9" />
            <TextBox Grid.Column="1" Grid.Row="9" />
            <Label Content="Ville :" Grid.Row="10" />
            <TextBox Grid.Column="1" Grid.Row="10" />
        </Grid>
    </Grid>
</Window>
```

Les balises `Style` permettent de grouper certaines propriétés relatives à un type de contrôle. Les styles peuvent être utilisés nommément. Ils peuvent aussi être appliqués automatiquement à tous les éléments correspondant à leur propriété `TargetType`, et qui sont situés plus bas que ce style dans la branche de l'arborescence. Cette dernière solution est celle utilisée ici. Un style permet de spécifier certaines propriétés communes à tous les objets `Label`, tandis qu'un autre style définit des valeurs de propriétés pour les contrôles `TextBox`.

À partir d'ici, nous allons travailler simultanément sur les couches `View` et `ViewModel` afin de montrer les mécanismes utilisés pour mettre en place une architecture MVVM.

→ Créez une classe nommée `MainViewModel` dans le dossier `ViewModel` de l'application.

C'est cette classe qui portera les données à afficher dans l'interface graphique. Elle comportera aussi d'autres éléments nécessaires à la gestion des actions exécutées par les boutons de la barre d'outils, ainsi que des propriétés qui seront utiles à d'autres traitements.

Le premier point à aborder est la récupération des données enregistrées à l'aide de la classe `GestionnaireDonnees`. Cette opération est effectuée dans le constructeur de la classe `MainViewModel` et valorise une propriété nommée `ListeContacts`. Celle-ci est de type `ObservableCollection<Contact>` car, comme étudié un peu plus tôt, ceci permet des mises à jour automatiques de l'interface graphique. Pour la même raison, nous allons également implémenter l'interface `INotifyPropertyChanged` sur cette classe.

```
public class MainViewModel : INotifyPropertyChanged
{
    private GestionnaireDonnees gestionnaireDonnees;

    private ObservableCollection<Contact> listeContacts;

    public ObservableCollection<Contact> ListeContacts
    {
        get { return listeContacts; }
        set
        {
            listeContacts = value;
            OnPropertyChanged("ListeContacts");
        }
    }

    public MainViewModel()
    {
        gestionnaireDonnees = new GestionnaireDonnees();

        Contact[] contacts =
gestionnaireDonnees.LireTousLesEnregistrements();
        ListeContacts = new
ObservableCollection<Contact>(contacts);
    }

    public event PropertyChangedEventHandler PropertyChanged;

    private void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Pour obtenir l'affichage des contacts dans l'application, il convient de définir un objet `MainViewModel` comme étant le contexte de données de la fenêtre.

- Dans le fichier `MainWindow.xaml.cs`, modifiez le constructeur de la classe de la manière suivante :

```
public MainWindow()
{
    InitializeComponent();
    DataContext = new CarnetAdressesMVVM.ViewModel.MainViewModel();
```

```
}
```

La ligne ajoutée est la seule qui liera la partie `View` à la couche `ViewModel`.

Il existe des techniques avancées permettant de ne pas les lier directement, mais elles ne font pas l'objet de cet ouvrage.

Une fois le contexte de données valorisé, il faut mettre en place le `Binding` et le `DataTemplate` permettant d'afficher les contacts dans la `ListBox`.

Modifiez la définition du contrôle `ListBox` comme décrit ci-dessous.

```
<ListBox x:Name="listBoxContacts" Grid.Row="1" Grid.Column="0"
ItemsSource="{Binding ListeContacts}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Vertical">
                <TextBlock Text="{Binding Prenom}" FontSize="20" />
                <TextBlock Text="{Binding Nom}" FontSize="20" />
                <TextBlock Text="{Binding Societe}" FontSize="14" />
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

L'ajout, la suppression et l'enregistrement des contacts doivent être effectués à partir de la barre d'outils. Pour chacun de ces boutons, il serait possible de créer un gestionnaire pour l'événement `Click`, mais l'architecture MVVM préconise l'utilisation de **commandes**. Ces traitements qui ne sont pas liés à l'interface graphique sont ainsi déportés dans le `ViewModel`.

Le contrôle `Button` possède une propriété `Command` qu'il convient de valoriser à l'aide d'un `Binding`. Cette propriété accepte une valeur de type `ICommand`, c'est-à-dire un objet qui implémente cette interface. Le framework .NET ne fournit malheureusement pas d'implémentation convenant à notre besoin, c'est pourquoi il est nécessaire de la créer dans le projet. L'implémentation proposée ci-dessous est très simple mais conviendra parfaitement à notre application.

Créez une classe `Command` dans le dossier `ViewModel` du projet.

```
using System;
using System.Windows.Input;

namespace CarnetAdressesMVVM.ViewModel
{
    public class Command : Icommand
    {
        readonly Action<object> actionAExecuter;

        public Command(Action<object> execute)
            : this(execute, null)
        {
        }
```

```
        public Command(Action<object> execute, Predicate<object>
canExecute)
        {
            actionAExecuter = execute;
        }
        public bool CanExecute(object parameter)
        {
            return true;
        }

        public event EventHandler CanExecuteChanged;
        public void Execute(object parameter)
        {
            actionAExecuter(parameter);
        }
    }
}
```

De nombreuses autres implémentations plus ou moins riches ont vu le jour depuis la création de WPF. Une des plus utilisées est la classe `RelayCommand` fournie dans le **MVVM Light Toolkit**, édité par Galasoft.

- Dans la définition de la classe `MainViewModel`, ajoutez trois propriétés publiques de type `ICommand` (ou `Command`) qui correspondront aux trois boutons de la barre d'outils.

```
public ICommand CommandeNouveauContact { get; set; }
public ICommand CommandeSupprimerContact { get; set; }
public ICommand CommandeToutEnregistrer { get; set; }
```

Chacune de ces commandes nécessite la création d'une méthode qui contient les instructions à exécuter.

```
private void ActionNouveauContact(object parametre)
{
    Contact contact = new Contact { Nom = "Nom", Prenom =
"Prénom" };
    ListeContacts.Add(contact);
}

private void ActionToutEnregistrer(object parametre)
{
    gestionnaireDonnees.Enregistrer(ListeContacts);
}

private void ActionSupprimerContact(object parametre)
{

}
```

La procédure de suppression d'un contact ne peut pas être implémentée en l'état. En effet, l'objet à supprimer, qui est l'objet sélectionné dans la `ListBox`, ne nous est pas connu. La création d'une propriété dans le `ViewModel` et l'ajout d'un `Binding` dans la définition de la `ListBox` permettent de régler ce problème.

```
private Contact contactSelectionne;
public Contact ContactSelectionne
{
    get { return contactSelectionne; }
    set
    {
          contactSelectionne = value;
          OnPropertyChanged("ContactSelectionne");
   }
}
```

```
<ListBox ... SelectedItem="{Binding ContactSelectionne,
Mode=TwoWay}">
```

Enfin, nous pouvons finir l'implémentation de la méthode `ActionSupprimerContact` et lier les commandes aux boutons à l'aide de `Bindings`.

```
private void ActionSupprimerContact(object parametre)
{
    if (ContactSelectionne != null)
           ListeContacts.Remove(ContactSelectionne);
}
```

```
<ToolBar>
    <Button Content="Ajouter" Command="{Binding
CommandeNouveauContact}" />
    <Button Content="Supprimer" Command="{Binding
CommandeSupprimerContact}" />
    <Button Content="Tout enregistrer" Command="{Binding
CommandeToutEnregistrer}" />
</ToolBar>
```

À ce stade, il est possible de créer et supprimer des enregistrements, mais impossible de les éditer. Les champs de saisie ne sont en effet liés à aucun contact. Pour remédier à cette situation, il faut définir le contexte de ces éléments comme étant le contact sélectionné dans la `ListBox`. La valorisation de la propriété `DataContext` de la `Grid` contenant les champs de saisie semble être une bonne solution. La valeur passée est un `Binding` pointant sur la propriété `SelectedItem` de la `ListBox`.

```
<Grid Margin="0" Grid.Row="1" Grid.Column="1"
      DataContext="{Binding ElementName=listBoxContacts,
Path=SelectedItem}">
```

Vous pouvez ensuite associer un `Binding` à chaque contrôle `TextBox`, en mode `TwoWay` pour vous assurer que les modifications effectuées seront remontées jusqu'aux propriétés correspondantes de l'objet `Contact`.

```
<Label Content="Nom :" Grid.Row="1" />
```

```
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Nom,
Mode=TwoWay}" />
<Label Content="Prénom" Grid.Row="2" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Prenom,
Mode=TwoWay}" />
<Label Content="Fonction :" Grid.Row="3"/>
<TextBox Grid.Column="1" Grid.Row="3" Text="{Binding Fonction,
Mode=TwoWay}" />
<Label Content="Société :" Grid.Row="4"/>
<TextBox Grid.Column="1" Grid.Row="4" Text="{Binding Societe,
Mode=TwoWay}" />
<Label Content="Telephone :" Grid.Row="5"/>
<TextBox Grid.Column="1" Grid.Row="5" Text="{Binding Telephone,
Mode=TwoWay}" />
<Label Content="Email :" Grid.Row="6"/>
<TextBox Grid.Column="1" Grid.Row="6" Text="{Binding Email,
Mode=TwoWay}" />
<Label Content="Adresse 1 :" Grid.Row="7" />
<TextBox Grid.Column="1" Grid.Row="7" Text="{Binding Adresse1,
Mode=TwoWay}" />
<Label Content="Adresse 2 :" Grid.Row="8" />
<TextBox Grid.Column="1" Grid.Row="8" Text="{Binding Adresse2,
Mode=TwoWay}" />
<Label Content="Code Postal :" Grid.Row="9" />
<TextBox Grid.Column="1" Grid.Row="9" Text="{Binding CodePostal,
Mode=TwoWay}" />
<Label Content="Ville :" Grid.Row="10" />
<TextBox Grid.Column="1" Grid.Row="10" Text="{Binding Ville,
Mode=TwoWay}" />
```

L'application est maintenant fonctionnelle. Quelques détails peuvent être néanmoins gênants :

- Il est possible de cliquer sur le bouton `Supprimer` lorsqu'aucun contact n'est sélectionné.
- Il est également possible de placer du texte dans les champs de saisie lorsqu'aucun contact n'est sélectionné.

Une solution à ce problème est de valoriser la propriété `IsEnabled` de ces contrôles à l'aide d'un `Binding` pointant sur une propriété booléenne du `ViewModel`.

- Ajoutez la propriété `ActiverSuppressionEtEdition` au `ViewModel` et modifiez la définition de la propriété `ContactSelectionne` de manière à ce que sa modification valorise `ActiverSuppressionEtEdition`.

```
private bool activerSuppressionEtEdition;

public bool ActiverSuppressionEtEdition
{
    get { return activerSuppressionEtEdition; }
    set
    {
          activerSuppressionEtEdition = value;
          OnPropertyChanged("ActiverSuppressionEtEdition");
    }
```

```
}

public Contact ContactSelectionne
{
    get { return contactSelectionne; }
    set
    {
          contactSelectionne = value;
          OnPropertyChanged("ContactSelectionne");

          if (contactSelectionne == null)
                ActiverSuppressionEtEdition = false;
          else
                ActiverSuppressionEtEdition = true;
    }
}
```

Modifiez la définition du bouton de suppression pour prendre en compte cette désactivation.

```
<Button Content="Supprimer" Command="{Binding
CommandeSupprimerContact}" IsEnabled="{Binding
ActiverSuppressionEtEdition}" />
```

Pour les champs de saisie, la situation est légèrement différente : leur contexte de données n'est pas le `ViewModel` mais un objet `Contact`. Pour modifier localement le contexte de données, il est nécessaire de passer par un élément dont le contexte de données est le `ViewModel`. L'objet `Window` racine est ainsi tout indiqué pour réaliser cette opération.

Nommez l'objet `Window` à l'aide d'un attribut `x:Name` puis créez la liaison entre la propriété `IsEnabled` et la propriété `ActiverSuppressionEtEdition` dans le style ciblant les contrôles `TextBox`, afin d'éviter de dupliquer le code pour chaque `TextBox`.

```
<Window ... x:Name="fenetre">
<Style TargetType="TextBox">
    <Setter Property="Margin" Value="0,0,10,0" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="IsEnabled" Value="{Binding
ElementName=fenetre,
Path=DataContext.ActiverSuppressionEtEdition}" />
</Style>
```

Vous avez réalisé une application de gestion de contacts de A à Z en utilisant l'architecture MVVM.

Bien que nécessitant un peu plus de travail, ainsi qu'un changement dans le mode de pensée, l'utilisation de ce patron de conception cloisonne nettement mieux les différentes couches de l'application et facilite grandement la maintenance du code.

Le code source de l'exemple détaillé ici est disponible en téléchargement depuis la page Informations générales.