Retour d'expérience apprentissage

Synthèse Xamarin

INRAE - Cati Sicpa

Mathieu Barousse
31/07/2020

Table des matières

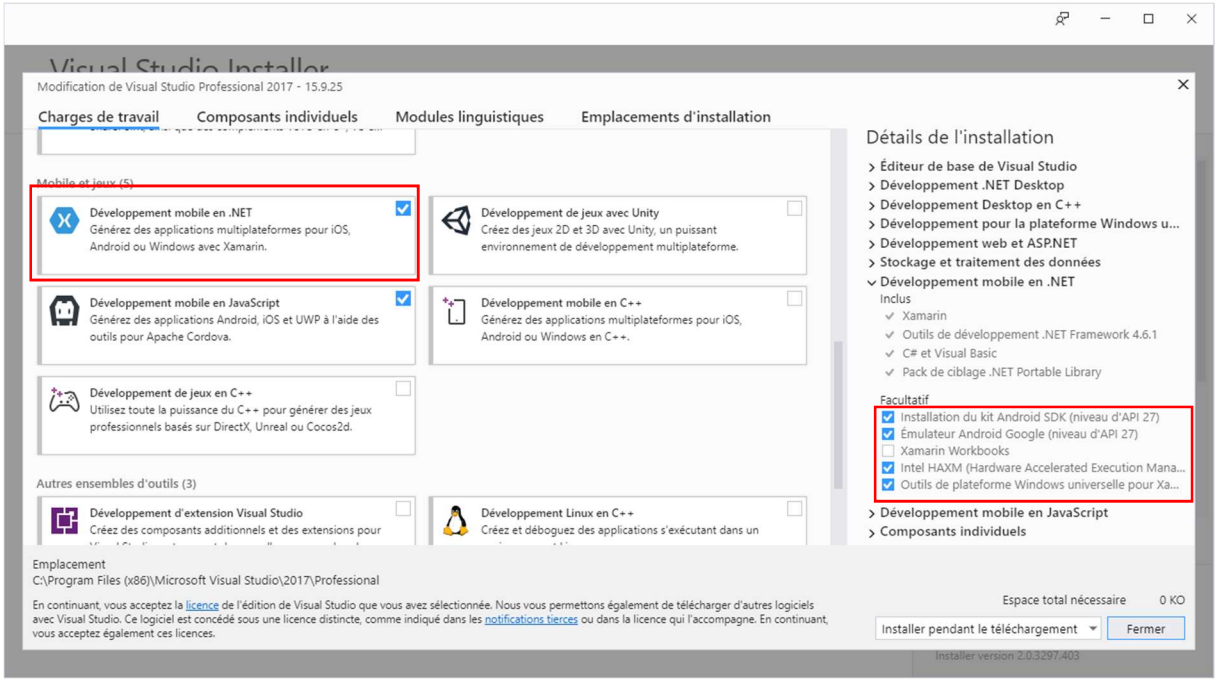
Présentation

Xamarin est une extension de la plateforme .NET, conçue dans le but de développer des applications multiplateformes. Nativement Xamarin supporte les plateformes suivantes :

- IOS 8 ou Supérieur
- Android 5.0 ou Supérieur
- Windows 10

Bien démarrer

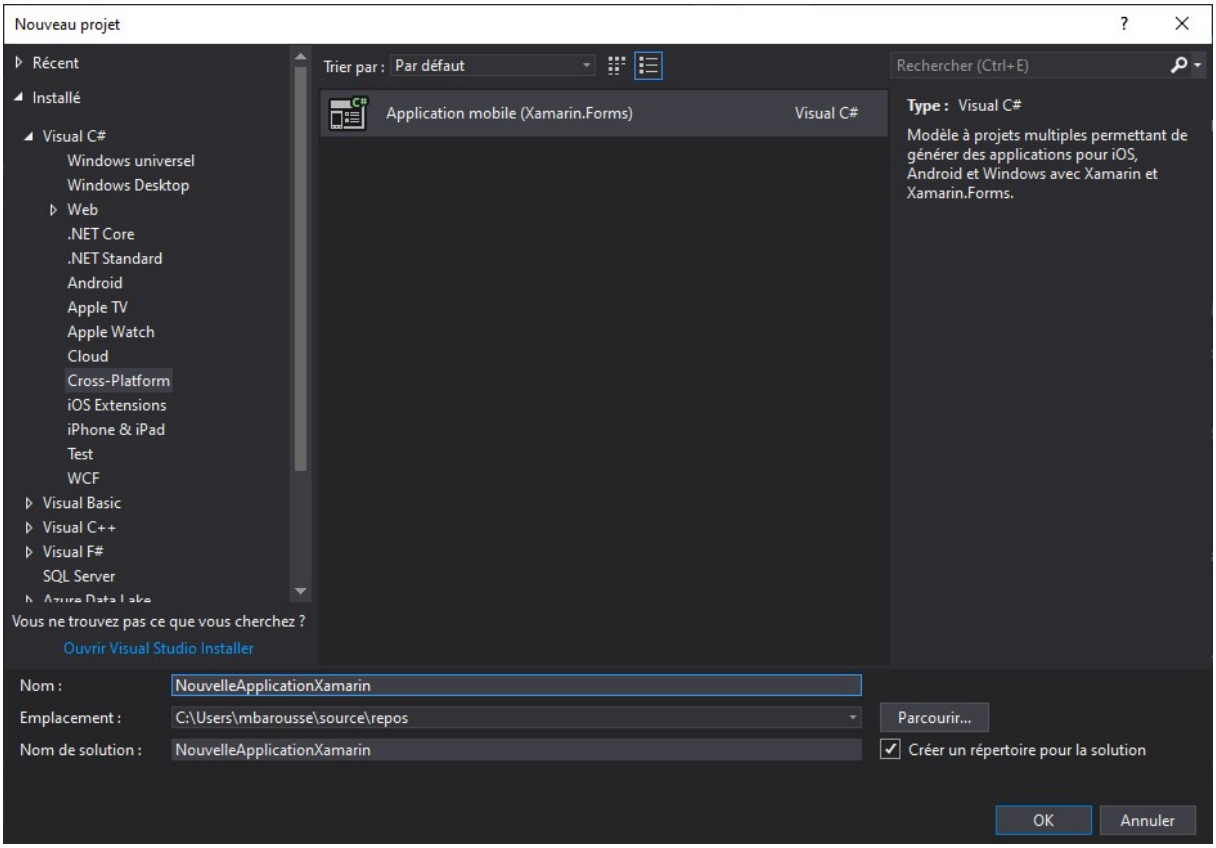
Pour réaliser une application Xamarin, on utilise Visual Studio 2017 ou plus récent et il est nécessaire au minimum la version 16299 de Windows 10. Pour la configuration de Visual Studio, sont requis :



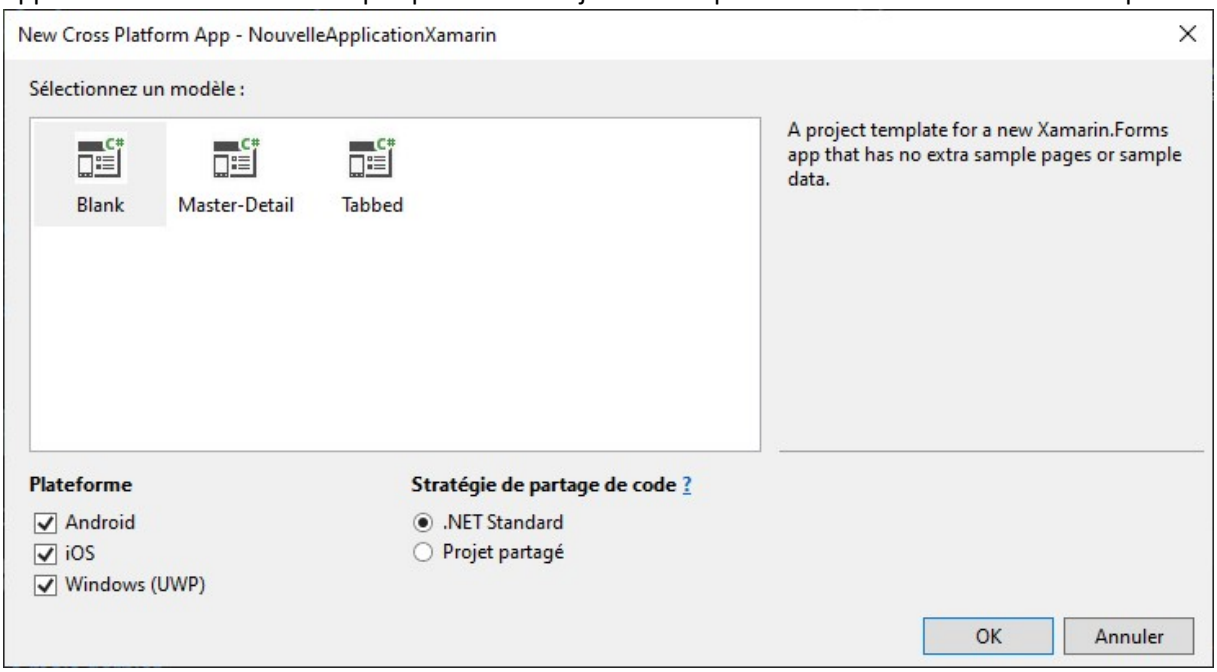
Le pack de « Développement mobile en .NET » ainsi que les options suivantes « Installation du kit Android SDK (niveau d'API 27) », « Émulateur Android Google (niveau d'API 27) », « Intel HAXM (Hardware Accelerated Execution Manager » et « Outils de plateforme Windows universelle pour Xamarin ».

Seul les émulateurs ayant une version supérieure à Android 6.0 (Lolipop) permet le débug depuis Visual Studio sinon il y est toujours possible de brancher un appareil Android et l'utiliser en débug USB. Pour savoir comment faire https://docs.microsoft.com/fr-fr/xamarin/android/get-started/installation/set-up-device-for-development

Pour démarrer un nouveau projet Xamarin, on va, premièrement, dans la fenêtre Nouveau Projet choisir *Application mobile (Xamarin.Forms)* dans Visual C# → Cross-Plateform. On nomme ensuite notre projet avant d'appuyer sur le bouton OK.

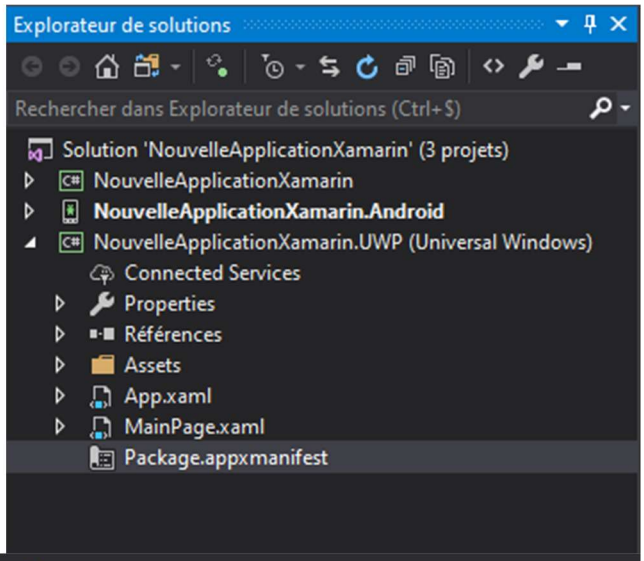


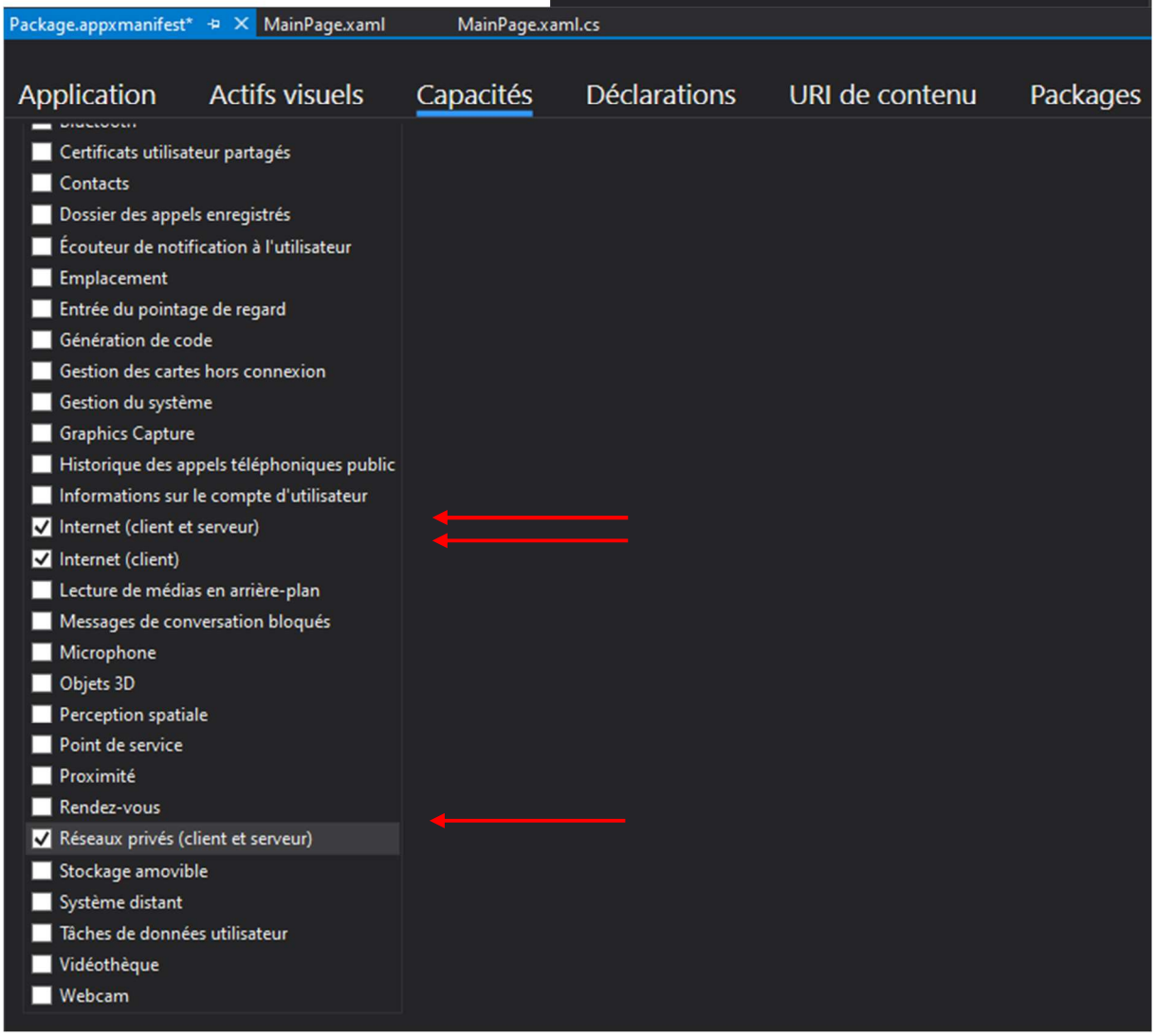
Sur la fenêtre suivante, il est question de choisir le style de page (décrit plus loin dans le document) pour la MainPage qui est la premiere page à s'ouvrir lors de l'exécution l'application, il est recommandé choisir Blank (page vide). Puis dessous on a le choix, des plateformes que l'on veut viser avec notre application. Attention il n'est pas possible de rajouter une plateforme une fois cette fenêtre passée.



Pour donner à un projet Windows l'accès à Internet, nécessaire pour les web services, il faut ouvrir le Package.appxmanifest accessible depuis l'explorateur de solution dans le projet Windows. Il faut ensuite choisir l'onglet Capacités puis cocher les cases :

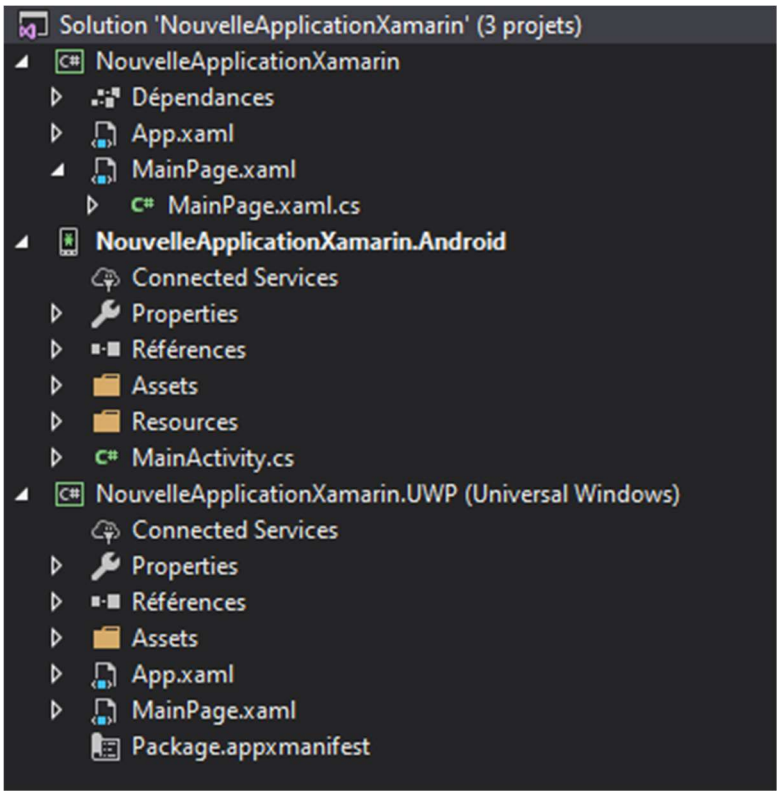
- Internet (client et serveur)
- Internet (client)
- Réseaux Privés





Structure

Suite à la création de notre projet, Visual Studio va créer une solution contenant plusieurs projets, une solution générique et une solution pour chaque plateforme choisie. Ces projets seront nommés *NomDuProjet.NomPlateforme* et le projet générique *NomDuProjet*.



Exemple de la structure d'un projet Xamarin à sa création, qui a pour cible Android et Windows.

C'est dans ce dernier que nous écrirons le code de l'application, dans la suite de ce document il sera nommé projet source. Les projets spécifiques aux plateformes sont générés en fonction du projet source, il n'est donc pas nécessaire de modifier le code sur ces projets. Cependant il peut y avoir quelques spécificités en fonction de la plateforme ; par exemple, la gestion des images.

MVVM

Explication

Le modèle-vue-vue modèle (en abrégé MVVM, de l'anglais Model View ViewModel) est une architecture et une méthode de conception, adaptée pour le développement des applications basées sur les technologies Windows Presentation Foundation(WPF) et Silverlight. De ce fait le MVVM est fortement recommandé pour le développement d'application Xamarin. C'est la méthode de conception que j'ai utilisée lors de mes tests.

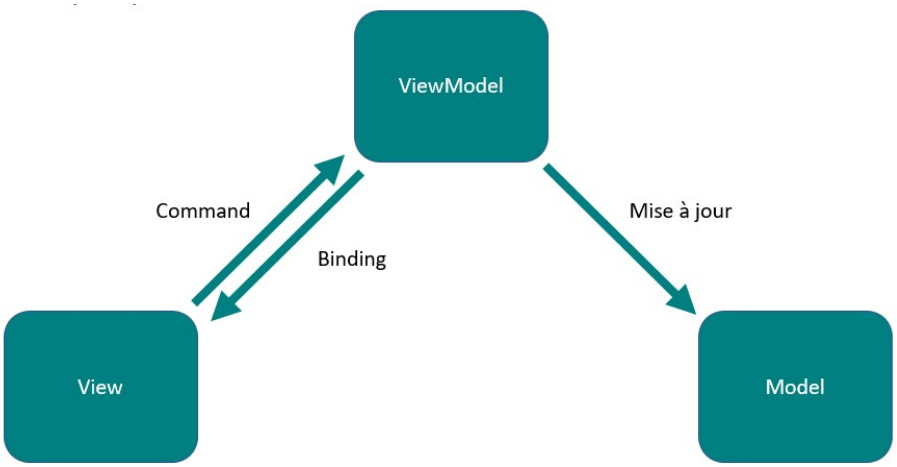


Schéma du design pattern MVVM

Décomposée en trois parties, Models, View, et ViewModel, chaque partie a son rôle. Premièrement, nous avons les Models qui sont les classes objets et ne contiennent que la définition des objets (avec leurs attributs) et la redéfinition de méthodes tel que *Equals*.

Ensuite, vient la View, il s'agit là du lieu où on réalise l'interface graphique de notre application ainsi que toute la partie de navigation entre les pages de l'application. Les Views comme dans les Windows.Forms se décomposent en deux parties. Cependant la structure n'est pas là même en Xamarin car notre View est écrite en XAML (détaillée plus loin) ce qui nous donne deux fichiers : *NomDeLaView.xaml* et le sous fichier *NomDeLaView.xaml.cs*

Dernière partie, il s'agit du ViewModel. Il a pour but de traiter les données fournies par les Models avant de les transmettre à la View. C'est ici, donc, que se trouve tout le traitement de l'application.

Binding

Le Binding va nous permettre de lier le ViewModel avec la View. Ainsi on pourra placer les données traitées par notre ViewModel à un endroit précis de notre View. Pour mettre en place le Binding, il faut, dans le ViewModel, déclarer l'objet à lier et créer les propriétés pour sa modification et son accès.

Source : https://docs.microsoft.com/fr-fr/xamarin/get-started/quickstarts/deepdive?pivots=windows#data-binding

Du côté de la View, il faut tout d'abord renseigner le BindingContext, il s'agit d'une propriété où il faut renseigner notre ViewModel. On le déclare généralement dans le début de la page car, selon les bonnes pratiques, on doit avoir un ViewModel par Model.

```
DetailTraitementViewModels.cs
Xamame
Xamame.ViewMode
75
76          // ---------------------- FamilleEvenement ----------------------
77
78          private List<familleEvent> familles;
79          public List<familleEvent> Familles {
80              get { return familles; }
81              set { SetProperty(ref familles, value); }
82          }
83
84          private familleEvent familleChoisie;
85          public familleEvent FamilleChoisie
86          {
87              get { return familleChoisie; }
88              set { SetProperty(ref familleChoisie, value);
89                  majComboEvenement();}
90          }
```

Dans notre ViewModel on déclare l'objet ainsi que ses propriétés. Par exemple, *List<familleEvent> familles*.

OngletTraitement.xaml

```
<Picker x:Name="Picker_ChoixFamilleEvenement"
        ItemsSource="{Binding Familles}"
        ItemDisplayBinding="{Binding libelle}"
        SelectedItem="{Binding FamilleChoisie}"/>
```

Puis dans la View, (fichier .xaml) on lie l'objet à une propriété d'un contrôle.

Ex : l'objet *Familles* est lié à la propriété *ItemSource* du control Picker.

INPC

INotifyPropertyChanged abrégé INPC est l'outil qui rend le MVVM utilisable. Vu que la View ne connait pas le ViewModel, il n'est pas possible d'écrire *TextBox.Text = variableDuViewModel* dans le code (XAML ou C#) d'une View. Cela nécessiterait que la View connaisse le ViewModel. Comme vu précédemment, le binding nous permet l'affichage. De plus, si la valeur de l'objet doit être modifié, il faut le signaler dans toute la View. C'est là où entre en jeu INPC qui est une interface que les objets ViewModel doivent implémenter. Cela permet, via la méthode *setProperty,* de comparer la nouvelle valeur avec l'ancienne valeur. Si elles sont égales ça ne fait rien, si elles sont différentes, on stocke la nouvelle valeur et actualise la View avec la nouvelle valeur.

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace NomProjet.ViewModels
{
    public class NomClasseViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected bool SetProperty<T> (ref T storage, T value, [CallerMemberName] string propertyname = null)
        {
            if (Object.Equals(storage, value)) return false;

            storage = value;
            OnPropertyChanged(propertyname);
            return true;
        }

        protected void OnPropertyChanged(string propertyName)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

Implémentation de INPC

```
private eventRef evenementChoisi;
public eventRef EvenementChoisi
{
    get { return evenementChoisi; }
    set { SetProperty(ref evenementChoisi, value); }
}
```

L'utilisation de INPC se résume à l'appel de *SetProperty* à l'intérieur du *set* de l'objet modifiable

XAML

Présentation

Il s'agit du langage à balises qui permet de concevoir les Views (interface graphique et la navigation entre les pages).

MainPage.xaml | MainPage.xaml.cs

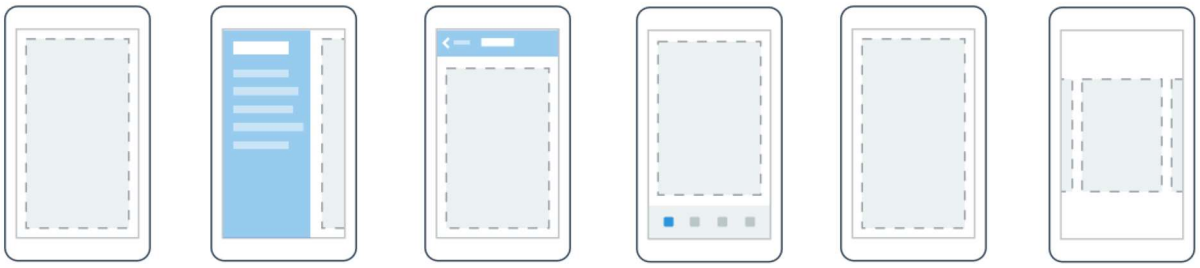
ContentPage

```
1   <?xml version="1.0" encoding="utf-8" ?>
2   <ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
3                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
4                xmlns:local="clr-namespace:NouvelleApplicationXamarin"
5                x:Class="NouvelleApplicationXamarin.MainPage">
6
7       <StackLayout>
8           <!-- Une ContentPage affichant Hello world -->
9           <Label Text="Hello world"
10            HorizontalOptions="Center"
11            VerticalOptions="CenterAndExpand" />
12      </StackLayout>
13
14  </ContentPage>
15
```

Exemple d'un Hello world en Xamarin

Page

Voici toutes les différentes pages disponibles nativement dans Xamarin



Source : https://docs.microsoft.com/fr-fr/xamarin/xamarin-forms/user-interface/controls/pages

ContentPage

Une page simple, vierge de tout contenu et de navigation native (il est toujours possible de la faire mais il est nécessaire de la coder). Elle peut être contenue dans d'autres types de pages.

MasterDetailPage

Cette page va découper en deux parties, le Master qui sera le menu latéral et le Detail qui sera le contenu de la page. Dans les bonnes pratiques Xamarin, on indique aux propriétés Master et Detail

directement des pages au lieu d'écrire directement le code dans les propriétés, ce qui transforme cette page en un conteneur de pages.

NavigationPage

Une page assez classique mais qui a un bandeau avec un bouton permettant de revenir à la page précédente.

TabbedPage

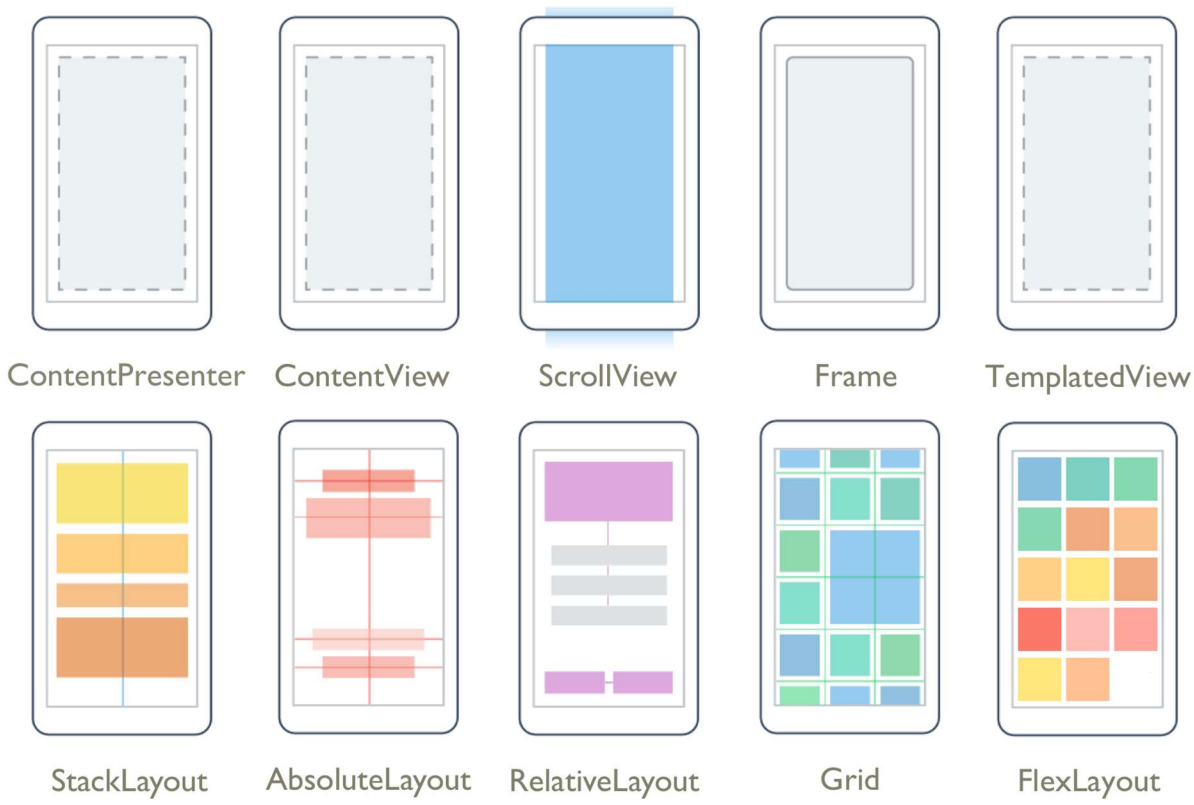
C'est une page qui comme le MasterDetailPage va être dans les bonnes pratiques une page conteneur de pages. Les pages seront organisées sous forme d'onglet.

CarouselPage

Assez proche de la TabbedPage, la CarouselPage va afficher les pages qu'elle contient les unes à côté des autres mais n'aura pas de bandeau affichant les autres pages.

Layout

Voici tous les Layouts disponibles nativement dans Xamarin. Ces derniers ont, pour la plupart du temps, une organisation qui leur est propre.



Source : https://docs.microsoft.com/fr-fr/xamarin/xamarin-forms/user-interface/controls/layouts

StackLayout

Il s'agit là du Layout le plus simple, les éléments sont mis les uns à la suite des autres.

AbsoluteLayout

Un Layout qui positionne des éléments enfants à des emplacements spécifiques par rapport à son parent. La position d'un enfant est indiquée à l'aide des propriétés jointes LayoutBounds et LayoutFlags.

RelativeLayout

Un Layout qui positionne des éléments enfants par rapport à RelativeLayout lui-même ou à leurs frères.

ScrollView

Un Layout qui va permettre de « scroll » lorsque nécessaire.

Frame

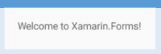
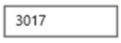
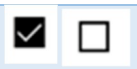
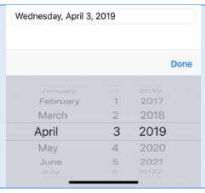
Un élément principalement visuel qui n'a pas forcement de propriété d'organisation.

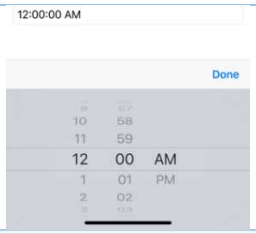
Grid

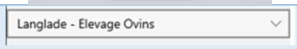
Ici, le Layout est découpé selon une grille au format souhaité et qui permet de mettre un control à un emplacement défini.

Contrôles

Le Contrôle est le plus petit élément d'une interface. Voici une liste des contrôles les plus fréquents et nécessaire pour développer une application Xamarin. Comme il n'existe pas (encore) d'éditeur graphique pour le XAML, il faut définir toutes les propriétés d'un contrôle en XAML.

Contrôle	Équivalent Win Form	Description	Exemple	Illustration
Label	Label	Permet d'afficher du texte Peut afficher une variable grâce au Binding	`<Label Text="Welcome to Xamarin.Forms!"/>`	
Button	Button	Bouton avec un texte à l'intérieur	`<Button Command="{Binding ActionBouton}" Text="Click Me!"/>`	
ImageButton	ImageButton	Bouton de la taille d'une image donnée	`<ImageButton Source="logoXamarin.png" Command="{Binding ActionBouton}"/>`	
Entry	TextBox	Champ de saisie La saisie peut être récupérer grâce au Binding. Le clavier peut être défini grâce à la propriété *Keyboard*	`<Entry x:Name="Entry_NumeroAnimal" Placeholder="NumeroAnimal" Text="{Binding NumAnimal.Value}" Keyboard="Numeric" Margin="5">`	
CheckBox	Checkbox	Case à cocher qui a deux états possible vrai ou faux	`<CheckBox x:Name="Checkbox_TraitementNouveau" Color="Black"/>`	
Switch		Interrupteur qui a deux états possible vrai ou faux	`<Switch x:Name="Switch_EtatSombre" HorizontalOptions="End" Toggled="Switch_EtatSombre_Toggled"/>`	
DatePicker		DateTimePicker(WForm) a été divisé en deux contrôles distinct, celui-ci s'occupe de la date	`<DatePicker Format="D"/>`	

TimePicker		DateTimePicker(WForm) a été divisé en deux contrôles distinct, celui-ci s'occupe de l'heure	`<TimePicker Format="T"/>`	

Picker	ComboBox	Liste déroulante
		Peut être remplie grâce au Binding
			`<Picker x:Name="Picker_ChoixPointAdministration"`
			`ItemsSource="{Binding PointAdms}"`
			`ItemDisplayBinding="{Binding libelle}"`
			`SelectedItem="{Binding PointAdmChoisi}"/>`
				

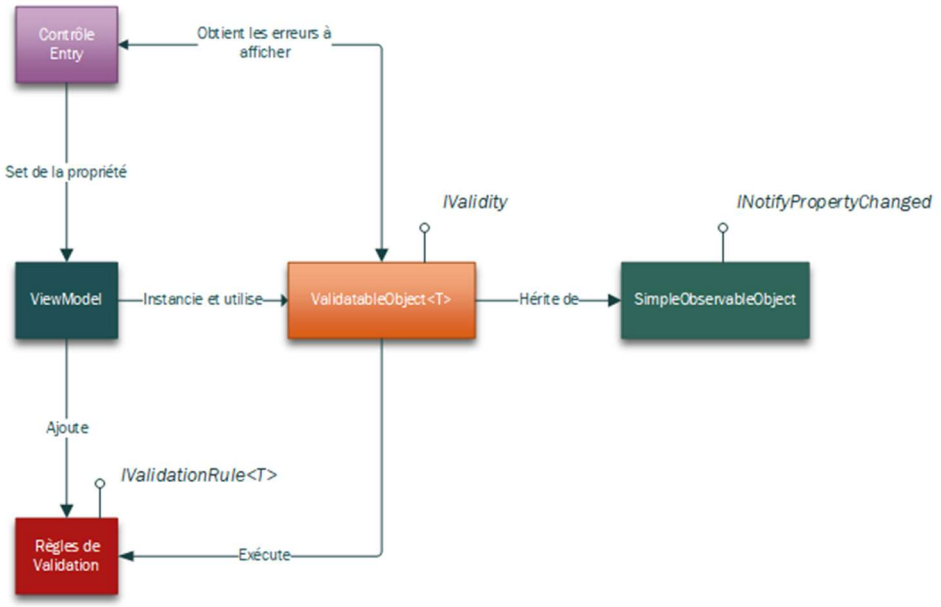
La liste entière des contrôles est disponible à l'adresse suivante : https://docs.microsoft.com/fr-fr/xamarin/xamarin-forms/xaml/xaml-controls

De plus, il est possible de créer ses propres Contrôles grâce à un dictionnaire de ressources.

Validation

Explication

Il s'agit d'un procédé qui permet de vérifier si une donnée respecte des règles données. C'est un principe qui peut être appliqué quel que soit le type de données saisies et quel que soit le contrôle où est saisie la donnée.



Source et code exemple : https://www.e-naxos.com/Blog/post/XamarinForms-Validation-des-donnees-et-Mvvm-Partie-1.aspx

Mise en place

On va utiliser une classe générique : ValidatableObject qui va venir remplacer le type de notre donnée. Par exemple, au lieu d'avoir un String on aura un ValidatableObject<String>. ValidatableObject contient plusieurs éléments dans sa définition, mais il faut surtout retenir qu'il y a : la valeur, un booléen qui indique si la valeur est acceptable, une liste de ValidationRule et une liste d'erreur. La puissance de ValidatableObject réside donc dans le fait que l'objet contient la valeur, sait s'autoévaluer et donner l'erreur associé s'il y en a une.

Rule

Ajouté dans le ValidatableObject, il existe les ValidationRule qui sont les définitions des règles permettant de dire si la valeur est acceptable. Il s'agit, encore une fois, d'un objet générique qui contient deux éléments le ValidationMessage qui est le message si la valeur ne respecte pas la règle et la méthode Check qui contient la règle à proprement parlé et renvoie un booléen en fonction du respect de la règle.

```
public class IsNotNullOrEmptyRule<T> : IValidationRule<T>
{
    public string ValidationMessage { get; set; }

    public bool Check(T value)
    {
        if (value == null) return false;
        return !string.IsNullOrWhiteSpace(value as string);
    }
}
```

Exemple de règle

```
public interface IValidationRule<in T>
{
    string ValidationMessage { get; set; }
    bool Check(T value);
}
```

Définition de l'interface IValidationRule

L'avantage des ValidationRule est le fait qu'elles peuvent être réutilisées sur plusieurs ValidatableObject et même réutilisées sur différents projets.

Utilisation

Dans notre ViewModel, on va créer notre propriété qui sera de type ValidatableObject *<TypeNecessaire>*, puis dans le constructeur de notre ViewModel on va ajouter des règles dans la liste de ValidationRule de l'objet ValidatableObject. Ensuite dans la View, on utilisera le ValidatableObject grâce à un Binding en faisant attention de lier la valeur de l'objet, de la manière suivante *NomValidatableObject.Value.* Et pour afficher s'il y a une erreur, on utilise la propriété FirstError en Binding dans un label pour afficher la première erreur, qu'il y en ait une ou plusieurs.

```
private ValidatableObject<String> numAnimal;
public ValidatableObject<String> NumAnimal
{
    get { return numAnimal; }
    set { SetProperty(ref numAnimal, value);}
}
```

Déclaration de l'objet de type ValidatableObject dans le ViewModel

```
NumAnimal = new ValidatableObject<string>(true);
NumAnimal.Value = "";
IValidationRule<string> r11 = new IsNotNullOrEmptyRule<string> { ValidationMessage = "Le numéro animal ne doit pas être vide." };
IValidationRule<string> r12 = new IsNumAnimalValideRule<string> { ValidationMessage = "Le numéro animal n'est pas correct." };
NumAnimal.Validations.Add(r11);
NumAnimal.Validations.Add(r12);
```

Instanciation de l'objet et attribution des règles dans le constructeur du VM

```
<Entry x:Name="Entry_NumeroAnimal"
       Placeholder="NumeroAnimal"
       Text="{Binding NumAnimal.Value}"
       Keyboard="Numeric"
       Margin="5">
<Label
    Text="{Binding NumAnimal.FirstError}"
    Style="{StaticResource ValidationErrorLabelStyle}" />
```

Utilisation du ValidatableObject dans la View

Command

Explication

Étant donné la structure du MVVM, on ne peut se servir des évènements (exemple : l'évènement Clicked d'un bouton) que pour la navigation dans l'application. Ainsi pour lancer l'exécution d'un traitement, on utilise les *Command*. Il s'agit d'une sorte de Binding. Les commandes possèdent l'avantage d'avoir une condition d'exécution, grisant automatiquement le contrôle dans lequel est situé la commande si la condition n'est pas respectée. Les commandes sont donc décomposées en deux parties : le traitement et la condition.

Exemple

```
public ICommand RechercheCommand { private set; get; }
```

Déclaration de la commande

```
RechercheCommand = new Command(RechercheAnimal,CanExecuteRecherche);
```

Instanciation avec attribution du traitement(RechercheAnimal) et de la condition(CanExecuteRecherche) sur RechercheCommand

Placé dans le constructeur du ViewModel

```
public bool CanExecuteRecherche()
{
    return (SiteChoisi != null && Animaux.Count() > 0);
}
```

Définition de la condition

ViewModel.cs

```
public void RechercheAnimal()
{
    //Corps du traitement
}
```

Définition du traitement

```
<Button
Command="{Binding RechercheCommand}"
Text="Valider"
Margin="5"/>
```

Utilisation de la commande dans un bouton

View.xaml

Retour d'expérience

Xamarin a beau être le successeur de Windows Form, il n'en est pas pour autant une version 2 de Windows Form. La transition de Windows Form vers Xamarin n'impliquera pas de grand changement au niveau du traitement de l'application et encore moins au niveau de la structure des données. Mais le XAML impliquera de refaire une interface de zéro. C'est l'occasion de repenser les interfaces des applications existantes. Car faire un copier-coller des interfaces impliquerait des structures de pages complexe et ne profitant pas des nouvelles possibilités qu'offre Xamarin. Même si cela doit changer les habitudes des utilisateurs cela serait au bénéfice de l'évolution qu'apporte Xamarin sinon il n'est pas intéressant de changer de technologie.

Référence

Configurer un appareil Android en débogage USB : https://docs.microsoft.com/fr-fr/xamarin/android/get-started/installation/set-up-device-for-development

Liaison de données : https://docs.microsoft.com/fr-fr/xamarin/get-started/quickstarts/deepdive

Liste des styles de pages : https://docs.microsoft.com/fr-fr/xamarin/xamarin-forms/user-interface/controls/pages

Liste des styles de Layouts : https://docs.microsoft.com/fr-fr/xamarin/xamarin-forms/user-interface/controls/layouts

Liste des contrôles : https://docs.microsoft.com/fr-fr/xamarin/xamarin-forms/xaml/xaml-controls

Validation : https://www.e-naxos.com/Blog/post/XamarinForms-Validation-des-donnees-et-Mvvm-Partie-1.aspx