Le développement cross plateforme avec Xamarin

Module 9 – Notions avancées de Xamarin.Forms

Objectifs

- Savoir mettre en places des listes avec Xamarin.Forms
- Utiliser la navigation pour créer un parcours utilisateur
- Interroger une API disponible sur Internet
- Comprendre l'intérêt du pattern MVVM face à MVC

Les listes

Une application mobile est composée de deux éléments fondamentaux : les listes et les formulaires

Les listes permettent d'afficher un fil d'actualité, une série d'options et de paramètres, des photos, des nouvelles, des récompenses gagnées dans un jeu, etc.

Les listes

Avec les technologies natives, les listes sont difficiles à mettre en œuvre et demande de faire appel à des notions et des composants difficiles d'accès

Grâce au XAML et à la surcharge native proposée par Xamarin.Forms, la création de liste est une étape rendue simple et efficace

Les listes

Xamarin.Forms repose sur trois outils pour restituer une liste :

- Le composant ListView
- Un template pour chaque cellule de la liste
- Une source de données

Les listes

Le composant ListView s'insère comme n'importe quelle autre balise XAML.

Un nom doit lui être associé pour injecter plus tard la source de données depuis le contrôleur.

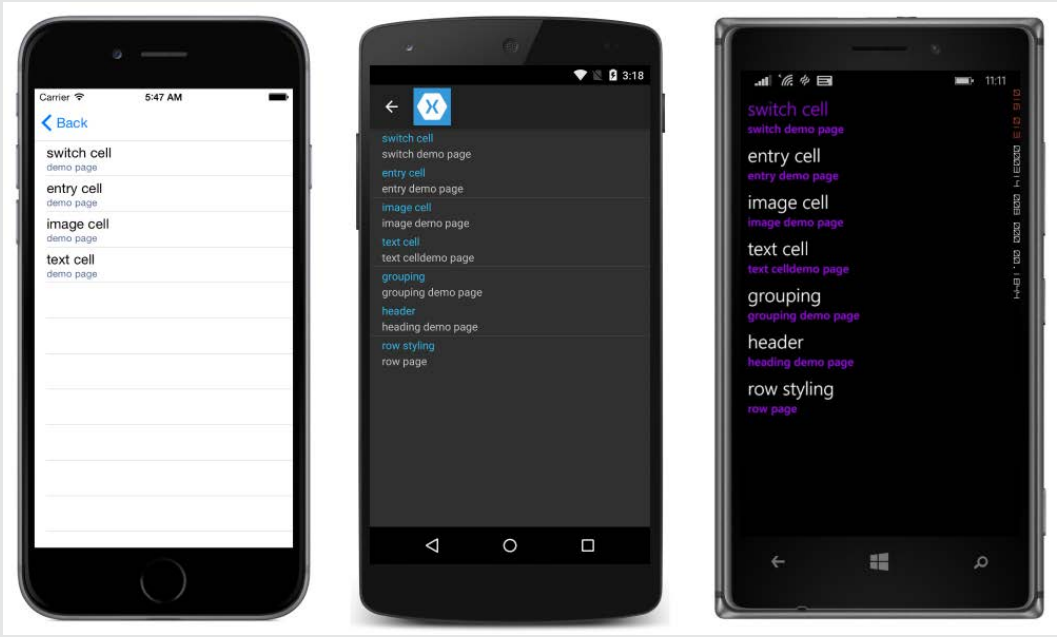
```
<ListView x:Name="MaListe">
        <ListView.ItemTemplate>
                <DataTemplate>

                </DataTemplate>
         </ListView.ItemTemplate>
</ListView>
```

Les listes

Le TextCell présente deux lignes : la première pour afficher un titre, la seconde pour afficher une description

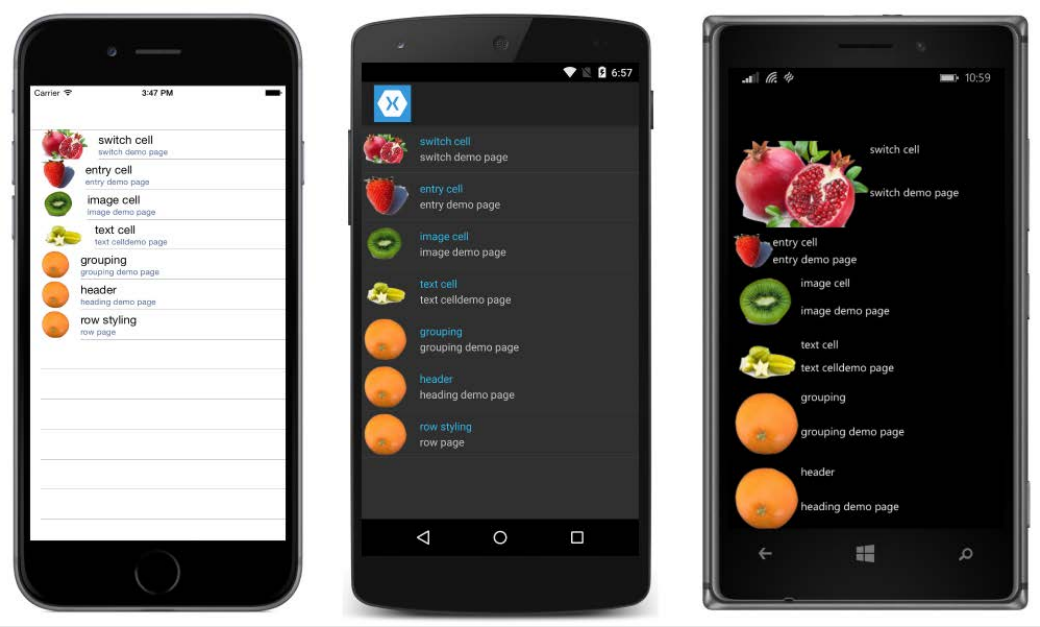
```
<TextCell Text="…" TextColor="…" Detail="…" DetailColor="…" />
```



Les listes

L'ImageCell est une extension du TextCell qui propose en complément d'afficher une image sur le côté gauche

```
<ImageCell Text="…" TextColor="…" Detail="…" DetailColor="…" ImageSource="…" />
```



Les listes

Pour alimenter la liste, une **Collection** doit être fournie comme source de données. Il est ensuite nécessaire de faire correspondre chaque attribut de la cellule avec un attribut de la source données.

Cette opération est réalisée grâce à un mécanisme de Binding proposé par Xamarin.Forms qui va permettre d'injecter une valeur en provenance d'un objet dans un attribut XAML, à travers un pattern **Observer**.

Les listes

La première étape consiste à créer une classe représentant la structure de données

```
namespace CoursXamarin
{
    public class Contact
    {
        public String Nom { get; set; }
        public String Prenom { get; set; }
        public String Avatar { get; set; }
    }
}
```

Les listes

La deuxième étape consiste à lier les attributs de la classe qui servira de source de données avec le template

```
<ListView x:Name="MaListe">
        <ListView.ItemTemplate>
            <DataTemplate>
                <ImageCell Text="{Binding Nom}" TextColor="Black"
                      Detail="{Binding Prenom}" DetailColor="Gray"
                      ImageSource="{Binding Avatar}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
```

Les listes

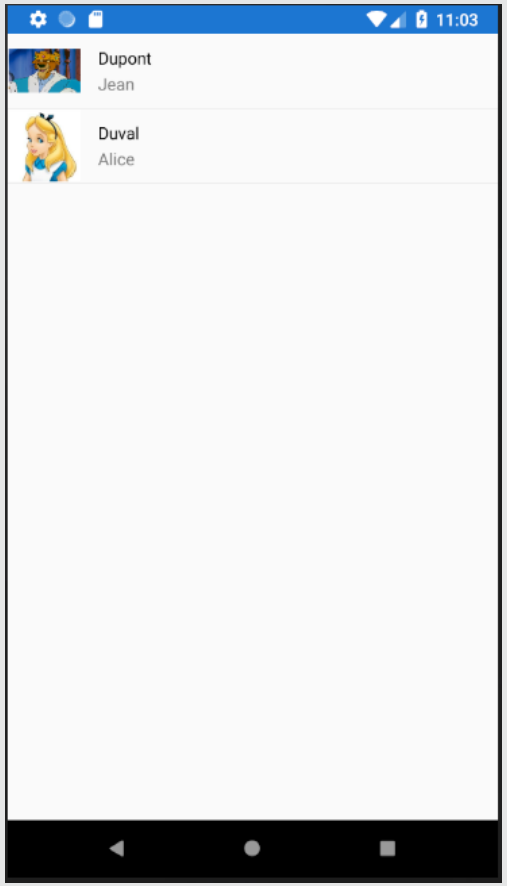
Pour terminer, le contrôleur doit injecter une source de données sous forme de **Collection** dans la **ListView**

```
public MainPage()
        {
            InitializeComponent();

            ObservableCollection<Contact> contacts = new ObservableCollection<Contact>();
            contacts.Add(new Contact { Nom = "Dupont", Prenom = "Jean",
               Avatar = "jean.jpg" });
            contacts.Add(new Contact { Nom = "Duval", Prenom = "Alice",
               Avatar = "alice.jpg" });
            this.MaListe.ItemsSource = contacts;
        }
```

Les listes

La liste s'affiche alors à l'écran



Les listes personnalisées

La plupart du temps, l'utilisation de listes de type **TextCell** ou **ImageCell** est insuffisant

Xamarin.Forms propose d'utiliser un template personnalisé pour déterminer l'affichage d'une ligne de la liste

Les listes personnalisées

Pour commencer à détailler un template personnalisé, il suffit d'utiliser le type de cellule **ViewCell**

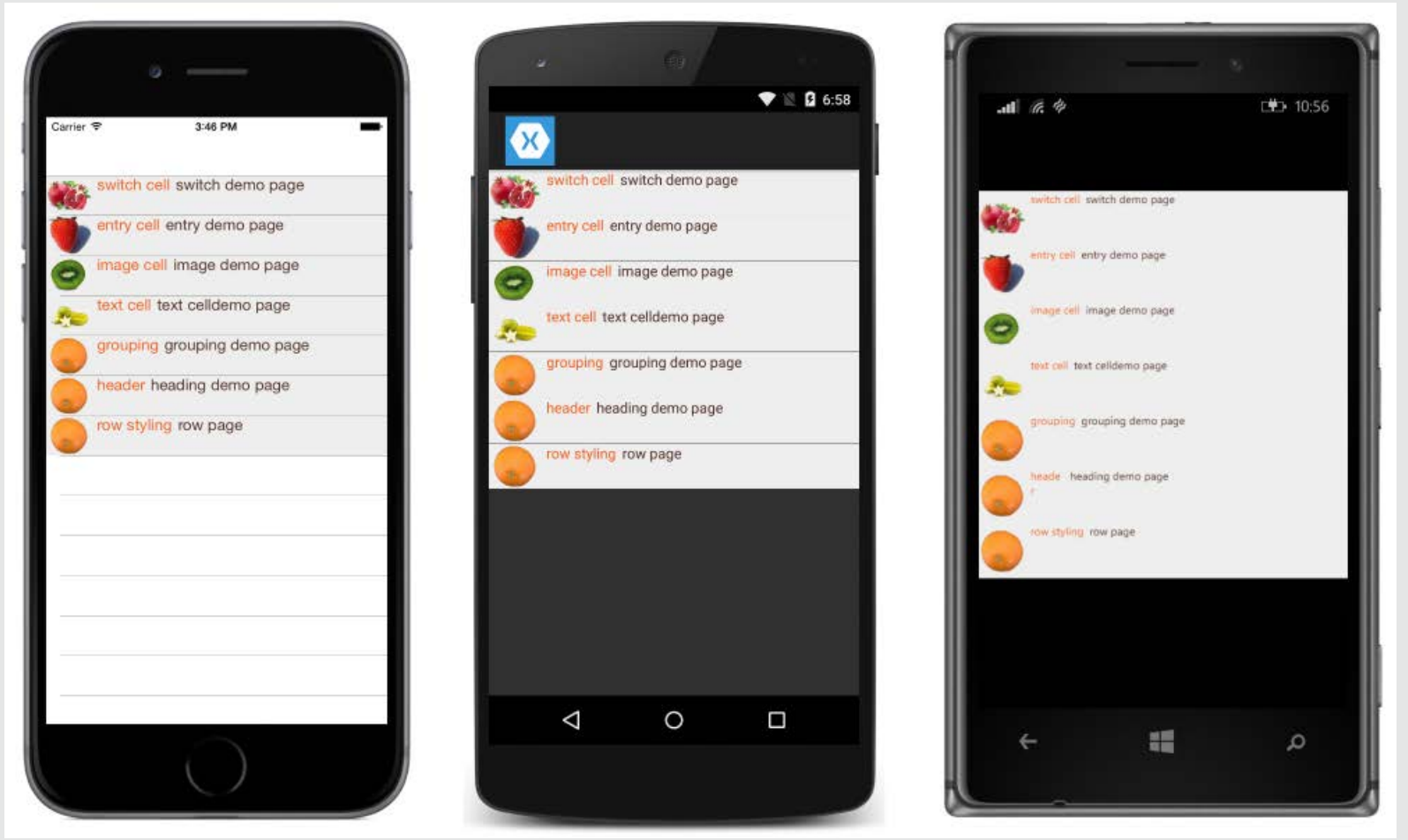
Attention, une **ViewCell** ne peut avoir qu'un seul enfant direct : on utilise alors un layout

Les listes personnalisées

```
<ListView x:Name="maListe">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <StackLayout BackgroundColor="#eee" Orientation="Vertical">
                    <StackLayout Orientation="Horizontal">
                        <Image Source="{Binding image}" />
                        <Label Text="{Binding title}" TextColor="#f35e20" />
                        <Label Text="{Binding subtitle}"
                               HorizontalOptions="EndAndExpand" TextColor="#503026" />
                    </StackLayout>
                </StackLayout>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

Notions avancées de Xamarin.Forms

Les listes personnalisées



Démonstration

Créer une liste de titres de musique

- Ajouter une **ListView** à la page
- Créer un modèle de données permettant de stocker des morceaux de musique
- Utiliser une **ImageCell** pour afficher les titres de musique
- Créer une **ViewCell** personnalisée pour afficher les titres de musique

La navigation

La navigation est gérée de manière très simple dans Xamarin.Forms

Après avoir lancé la page principale dans une page de navigation **NavigationPage**,
il suffit d'ajouter ou de supprimer des pages dans la pile de navigation

La navigation

La première étape pour activer la navigation est d'encapsuler la page lancée par défaut par une **NavigationPage** dans le fichier **App.xaml.cs**

```
public App()
        {
            InitializeComponent();

            MainPage = new NavigationPage(MainPage());
        }
```

La navigation

Par la suite, pour naviguer d'une page à l'autre, il suffit de l'instancier et demander au contrôleur de navigation de présenter la page à l'écran

```
async void OnBoutonSuivantClicked(object sender, EventArgs e) {
        await Navigation.PushAsync (new PageSuivante());
}
```

La navigation

Pour retourner à une page précédente, le contrôleur de navigation présente une méthode **PopAsync**

```
async void OnBoutonRetourClicked (object sender, EventArgs e) {
        await Navigation.PopAsync ();
}
```

La navigation

Il est également possible de revenir à la première page avec la méthode **PopToRoot** du contrôleur de navigation

```
async void OnBoutonAccueilClicked(object sender, EventArgs e) {
        await Navigation.PopToRootAsync ();
}
```

La navigation

S'il est nécessaire de passer une donnée à la page suivante lors de la navigation, il faut ajouter un paramètre au constructeur de la nouvelle page

```
public NouvellePage(string id) {
        InitializeComponent ();
        idArticle = id;
}


async void OnBoutonSuivantClicked(object sender, EventArgs e) {
        await Navigation.PushAsync (new NouvellePage("abcdefg"));
}
```

Démonstration

Créer une page de détails pour chaque musique

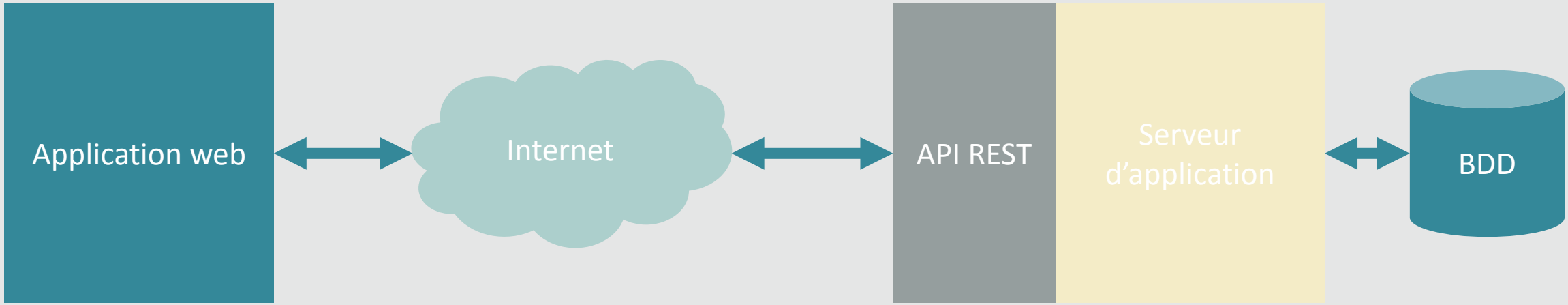
- Créer une nouvelle page depuis Visual Studio
- Intégrer la page principale dans un contrôleur de navigation
- Naviguer de la liste vers la page de détails
- Revenir vers la liste depuis la page de détails

Interroger une API

Une application mobile fonctionne rarement de manière autonome, elle fait généralement appel à un serveur distant

La tendance actuelle consiste à exposer une API REST qui permet de lire, créer, modifier et supprimer des ressources, en s'appuyant sur le protocole HTTP et le format JSON

Interroger une API



Interroger une API

Pour réaliser des appels à une API REST avec Xamarin, il suffit d'intégrer les modules Microsoft.Net.Http et Newtonsoft.Json via NuGet

Ces packages exposent des méthodes qui gèrent la communication avec le serveur, l'encodage et le décodage des informations

Interroger une API

Il faut également développer les classes qui permettront de décoder les informations renvoyées par l'API et les manipuler à travers un objet

```
public class RendezVous
        {
            public string Description { get; set; }
            public string Date { get; set; }
            public bool isAccepte { get; set; }
        }
```

Interroger une API

Une méthode **async** permet ensuite d'interroger l'API et de récupérer un objet en retour

```
public async Task<List<RendezVous>> GetRendezVousAsync()
        {
            HttpClient client = new HttpClient();
            var response = await client.GetStringAsync("http://localhost:5000/api/rdv");
            var todoItems = JsonConvert.DeserializeObject<List<RendezVous>>(response);
            return todoItems;
        }
```

Interroger une API

Une liste d'API accessibles librement et gratuitement est disponible sur GitHub

https://github.com/toddmotto/public-apis

Démonstration

Interroger une API pour récupérer des citations

- Intégrer les packages Microsoft.Net.Http et Newtonsoft.Json via NuGet
- Trouver une API de citations sur le dépôt GitHub qui référence les API publiques
- Modéliser la réponse de l'API sous forme de classes
- Interroger une API et afficher le résultat à l'écran

Le pattern MVVM

Le pattern MVC peut parfois montrer ses limites lorsqu'il est utilisé en contexte multiplateforme.

C'est par exemple le cas lorsqu'il faut bien isoler les responsabilités du service et du contrôleur.

Le contrôleur se vide alors au bénéfice (ou détriment) du service : le contrôleur ne sert alors plus que de relais entre le service et la vue

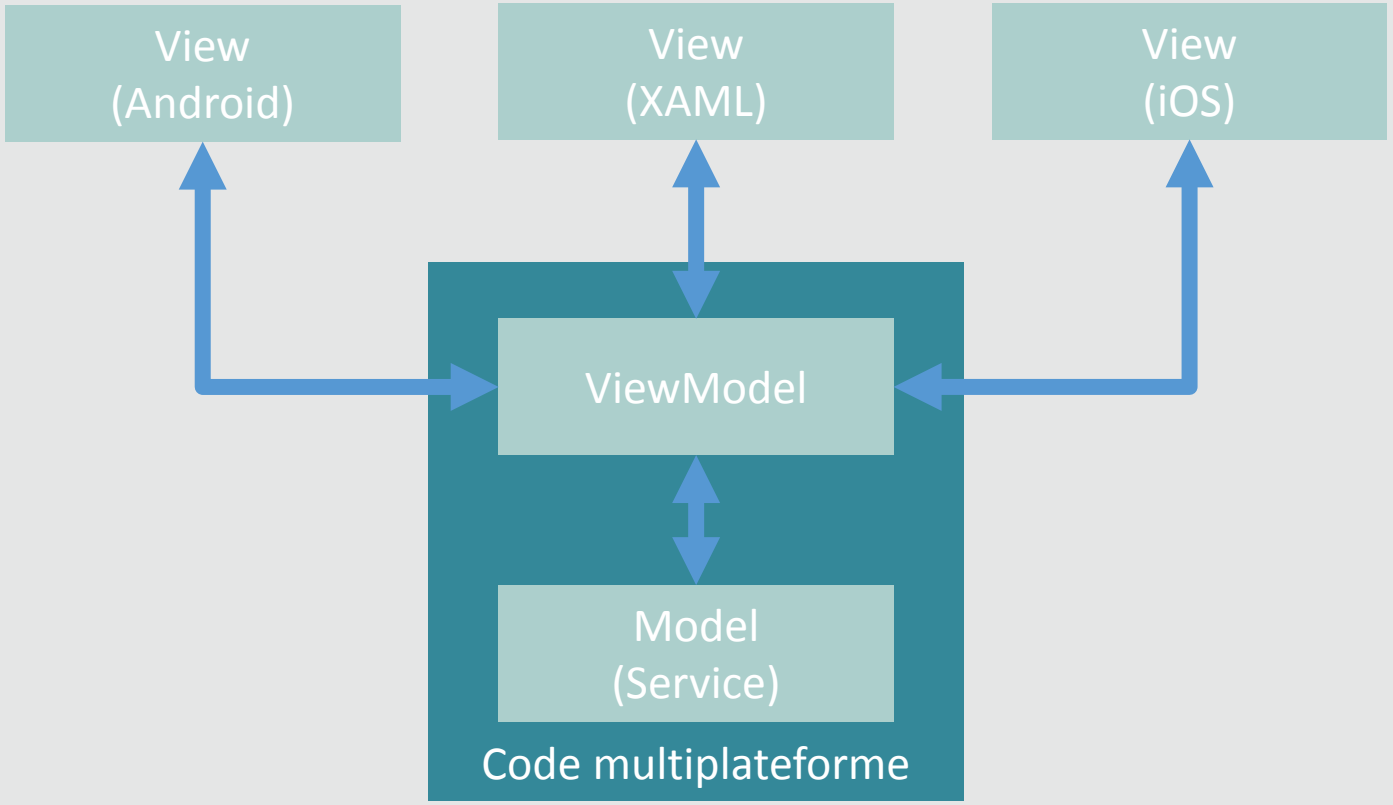
Le service se retrouve de son côté surchargé de méthodes métier, de contrôles de surface et de transformations de données

Le pattern MVVM

Le pattern MVVM propose de délaisser le contrôleur qui, dans le cas de Xamarin native, est redéveloppé pour chaque plateforme, et de le remplacer par un **ViewModel**.

Ce **ViewModel** a pour rôle de centraliser les réponses à des évènements et de mettre à disposition les données à afficher.

Le pattern MVVM



Le pattern MVVM

La révolution que propose MVVM est que la vue va s'alimenter automatiquement auprès du **ViewModel**.

Cette opération, appelée **Binding**, utilise le pattern **Observer** pour récupérer automatiquement les mises à jour qui seraient effectuées dans le **ViewModel**.

Le contrôleur n'a donc plus besoin de récupérer une référence vers chaque composant et de mettre à jour ses informations : la vue s'en charge elle-même.

Le pattern MVVM

Pour mettre en place MVVM, il faut donc utiliser le pattern **Observer**.

Le framework .NET propose des classes utilitaires pour gérer les abonnements (de la vue) et notifier des changements

Le pattern MVVM

```
class ClockViewModel : INotifyPropertyChanged {
        DateTime dateTime;
        public event PropertyChangedEventHandler PropertyChanged;
        public ClockViewModel() { this.DateTime = DateTime.Now; }

        public DateTime DateTime {
                set { if (dateTime != value) {
                                dateTime = value;
                                if (PropertyChanged != null) {
                                        PropertyChanged(this,
                                                new PropertyChangedEventArgs("DateTime"));
                                }
                        }
                }
                get { return dateTime; }
        }
}
```

Le pattern MVVM

Le **Binding** est en tout point similaire à celui utilisé pour la mise en place des listes.

Il suffit d'indiquer à la vue quelle instance de ViewModel lui est associée et, grâce au Binding, elle se chargera de récupérer les données et de contracter les abonnements qui l'informeront de changements dans le ViewModel.

Le pattern MVVM

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
        x:Class="XamlSamples.ClockPage"
        Title="Clock Page">

        <ContentPage.BindingContext>
                <local:ClockViewModel/>
        </ContentPage.BindingContext>

        <Label Text="{Binding DateTime, StringFormat='{0:T}’} »
                FontSize="Large" HorizontalOptions="Center" VerticalOptions="Center">
        </Label>
</ContentPage>
```

Le pattern MVVM

L'autre intérêt de MVVM est de pouvoir exécuter des commandes directement dans le **ViewModel**.

De cette manière, même les interactions utilisateur sont déportées dans le ViewModel et n'ont plus à être codées dans le contrôleur (potentiellement dupliqué pour chaque plateforme).

Le pattern MVVM

```
class ClockViewModel : INotifyPropertyChanged {

        public ICommand ResetHourCommand { protected set; get; }

         public ClockViewModel() {
                 ResetHourCommand = new Command(() => {
                                 this.DateTime = DateTime.Now;
                 });
}
```

Le pattern MVVM

Les commandes peuvent également recevoir des arguments, qui sont passés au moment du déclenchement de l'évènement.

```
class ClockViewModel : INotifyPropertyChanged {

        public ICommand SetHourCommand { protected set; get; }

         public ClockViewModel() {
                SetHourCommand = new Command<string>((hour) => {
                        …
                });
}
```

Le pattern MVVM

Au niveau de la vue **XAML**, la logique reste la même que pour l'affichage de données depuis le **ViewModel** grâce au **Binding**.

```
<Button Text="Reset" Command="{Binding ResetHourCommand}" />
<Button Text="Set 12:00" Command="{Binding SetHourCommand}"
        CommandParameter="12:00" />
```

Le pattern MVVM

Le pattern MVVM permet d'aller plus loin dans la mutualisation du code et d'introduire les fondements de la programmation réactive.

Son utilisation nécessite néanmoins :

- plus de technicité,
- la mise en place de frameworks externes dans le cas de projets Xamarin.Native
- de changer la manière du développeur de concevoir les applications par rapport à MVC.

Les applications Xamarin.Forms sont souvent réalisées à l'aide de MVVM

Les applications Xamarin.Native, les applications de type POC ou les petites applications reposent souvent sur MVC

Notions avancées de Xamarin.Forms

Construire une liste et utiliser la navigation

TP