Le développement cross plateforme avec Xamarin

Module 4 – Conception de vues Xamarin.Forms

Objectifs

- Comprendre l'intérêt du pattern MVC dans un projet Xamarin
- Savoir développer une vue avec le langage XAML
- Savoir mettre en forme une vue à l'aide des concepts de pages et de layout

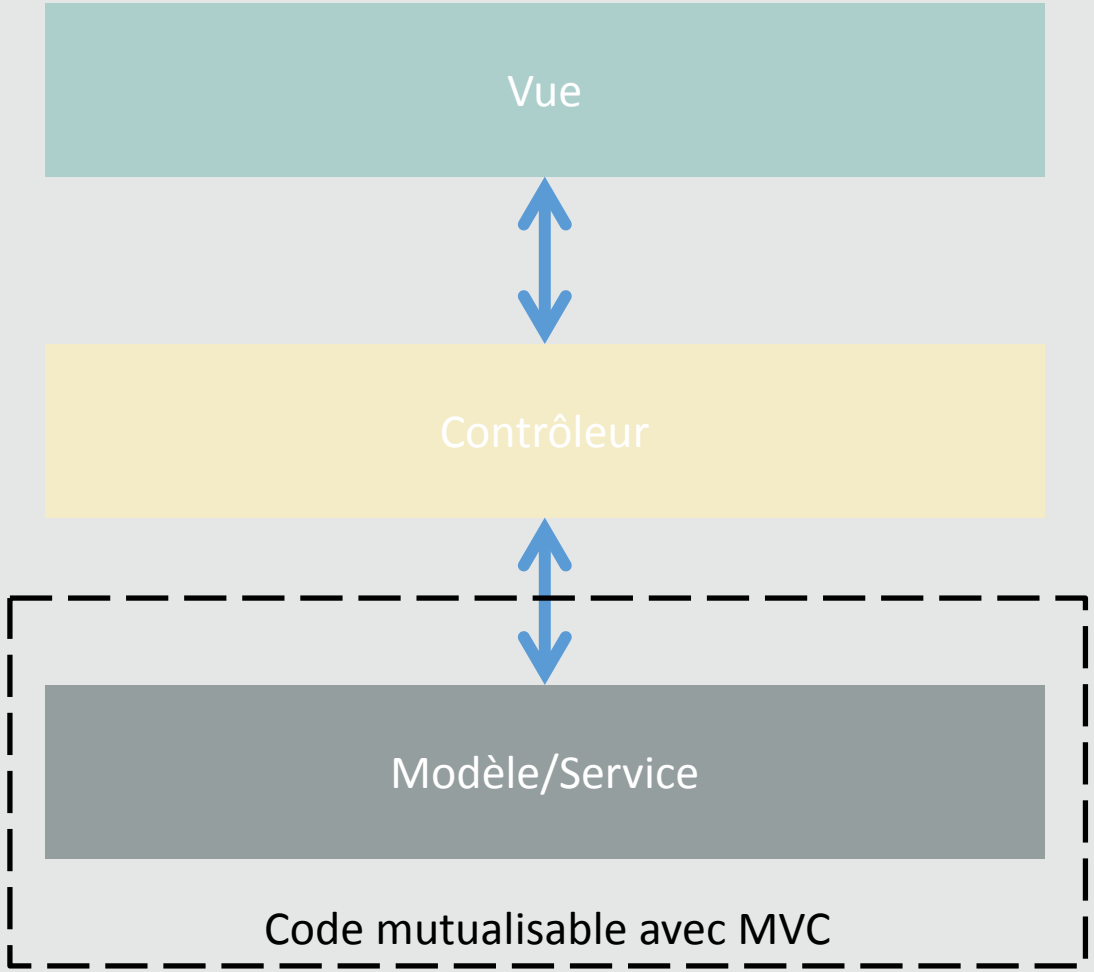
Rappels sur le pattern MVC

Le **pattern MVC** (Modèle Vue Contrôleur) permet de séparer le code source de la vue, du contrôleur et des services métier

Xamarin repose sur le fait que seul l'affichage (vue et contrôleur) doit être décliné sur les différentes plateformes

Le recours à ce type de pattern permet donc de séparer le code mutualisable du code spécifique

Rappels sur le pattern MVC



Rappels sur le pattern MVC

La **vue** rassemble tous les éléments visuels

La vue ne doit pas faire de calculs, de vérifications des données saisies, de requêtes en base de données ou sur un serveur

Elle se contente de mettre en forme les données qui lui sont fournies par le contrôleur et de transmettre les interactions au contrôleur

Rappels sur le pattern MVC

Le **contrôleur** vérifie les actions et les données utilisateurs puis prépare les données pour la vue

Le contrôleur ne prend pas de décision métier, ne fait pas de calculs et n'exécute pas de requêtes.

Il se contente d'effectuer des contrôles de surface et éventuellement de convertir des données à afficher.

Rappels sur le pattern MVC

Le service exécute les traitements métier, réalise des calculs ou exécute des requêtes en base de données ou vers un serveur distant

Le service ne vérifie pas la provenance des données et ne se soucie pas de savoir dans quel contexte les retours vont être utilisés.

Rappels sur le pattern MVC

L'utilisation du pattern MVC est indispensable pour isoler le code source du code métier

Avec Xamarin.Forms, les pages étant converties d'un métalangage vers des composants natifs, 100% du code est mutualisable avec MVC

Avec Xamarin native, il est obligatoire de développer une nouvelle vue tirant parti des composants natifs et un contrôleur chargé de piloter ses composants avec le SDK natif. Dans ce cas, 50% du code uniquement peut être mutualisé.

Développer une vue avec Xamarin.Forms

Il existe deux moyens de développer une vue avec Xamarin.Forms

Développer une vue avec Xamarin.Forms

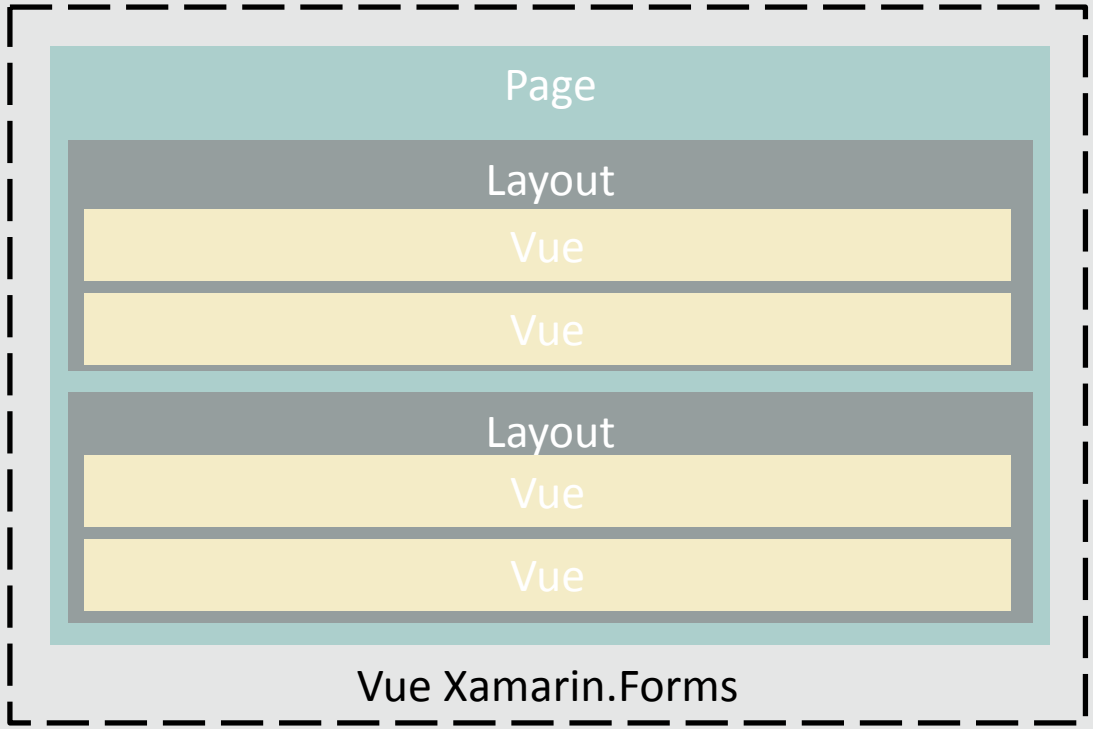
L'utilisation de C# ou XAML est une affaire de goût et de contraintes : en tous les cas, le code XAML sera converti en code C# au moment de la compilation

En C#, 100% de l'API Xamarin.Forms est accessible et il est plus aisé de lier un composant visuel avec une action dans le contrôleur ou des données

Avec XAML, une portion plus restreinte de l'API est disponible mais l'avantage est que la vue est bien différenciée du contrôleur

Développer une vue avec Xamarin.Forms

Quel que soit le langage utilisé pour développer une vue, la hiérarchie est toujours la même



Développer une vue avec Xamarin.Forms

Une page ne dispose pas d'un cycle de vie, à l'inverse des Activity Android ou des UIViewController iOS

Les différentes initialisations et appels aux autres composants se font donc à l'initialisation de la vue

Pour réagir à des évènements du cycle de vie, il faut nécessairement agir au niveau de l'application

Présentation de XAML

Le XAML est un dérivé du XML permettant de décrire les éléments présents dans une vue, leur hiérarchie et leurs propriétés

Il peut être comparé au principe de **layout** d'Android

Présentation de XAML

Le XAML n'a pas été créé pour Xamarin, mais est présent depuis plusieurs années dans l'écosystème .NET

Des composants sont implémentés par défaut mais la grande force de XAML réside dans le fait qu'il est possible de créer des composants personnalisés

Présentation de XAML

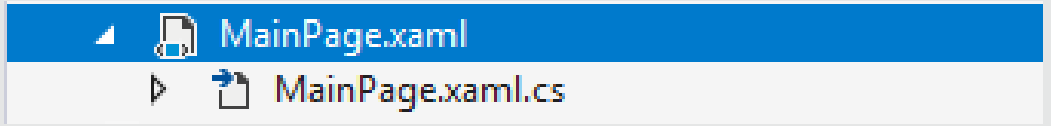
- Les avantages de XAML
  - Une meilleure lisibilité que le C# et une syntaxe plus compacte
  - La hiérarchie des composants est facilement mesurable
  - Le XAML peut être généré à partir d'outils graphiques dans Visual Studio (WYSIWYG)

Présentation de XAML

- Le XAML a aussi quelques inconvénients
  - Il ne peut pas contenir de code exécutable, de transformation de la donnée, etc.
  - Il ne peut pas utiliser de boucles ou de conditions pour insérer dynamiquement des composants
  - Il ne peut appeler des méthodes du contrôleur qu'avec une liste d'évènements restreinte

Présentation de XAML

- Un fichier XAML est structuré de la manière suivante :
  - Un fichier XML qui contient le code de la vue
  - Un fichier CS (appelé code behind) qui contient le code du contrôleur



Présentation de XAML

- Le fichier XAML affichant le message de bienvenue contient le code suivant

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:CoursXamarin"
             x:Class="CoursXamarin.MainPage">

    <StackLayout>
        <!-- Place new controls here -->
        <Label Text="Welcome to Xamarin.Forms!"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />
    </StackLayout>

</ContentPage>
```

Présentation de XAML

- La balise **ContentPage** représente la page elle-même et déclare les différents **namespaces** :
  - Le namespace par défaut qui référence des balises Xamarin.Forms
  - Le namespace **x:** qui référence le langage XAML
  - Le namespace **local:** qui référence le projet lui-même
- Les namespaces permettent de préciser dans quel contexte doit être interprété une balise, de manière à le transformer en code C#
- L'attribut **x:Class** indique qu'une classe doit être créée dans le **namespace** du projet pour cet écran

Présentation de XAML

- Les deux balises filles de la balise **ContentPage** sont des composants Xamarin.Forms
  - La première, **StackLayout**, indique quel est le type de Layout à utiliser (soit comment disposer les éléments les uns par rapport aux autres)
  - La deuxième, **Label**, indique qu'un texte doit être affiché

Présentation de XAML

- Le fichier C# jouant le rôle de contrôleur pour la page de bienvenue est composé du code suivant :

```
namespace CoursXamarin
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

Présentation de XAML

- Le fichier est composé d'une **classe**, faisant partie du **namespace CoursXamarin**
  - Le namespace est le même que celui renseigné dans le fichier XAML
  - Le nom de la classe est le même que celui renseigné dans le fichier XAML
- La classe hérite de la classe ContentPage
  - C'est le même type de page que celui qui est renseigné dans le fichier XAML
- Un constructeur appelle la méthode **InitializeComponent**, implémentée dans l'ancêtre **ContentPage** qui a pour rôle d'appeler le code C# généré à partir du fichier XAML et d'afficher les composants décrits dans celui-ci

Présentation de XAML

- La syntaxe d'une balise XAML est la suivante :
  - Le nom de la balise qui correspond au composant à intégrer dans la page (qui est lié à une classe C# dans le framework Xamarin.Forms)
  - Des attributs pour paramétrer ce composant

```
<Label Text="Welcome to Xamarin.Forms!"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand" />
```

Présentation de XAML

Le XAML permet de bien faire la séparation entre contrôleur et vue, de produire un code plus lisible et plus maintenable

Pour un développeur Android natif, l'utilisation du XAML à la place des layouts permet une transition plus douce vers Xamarin

Le XAML ne permet pas de tirer parti de 100% de l'API de Xamarin.Forms et nécessite de passer par le contrôleur pour l'alimentation, la transformation des données et la réaction à des évènements utilisateurs

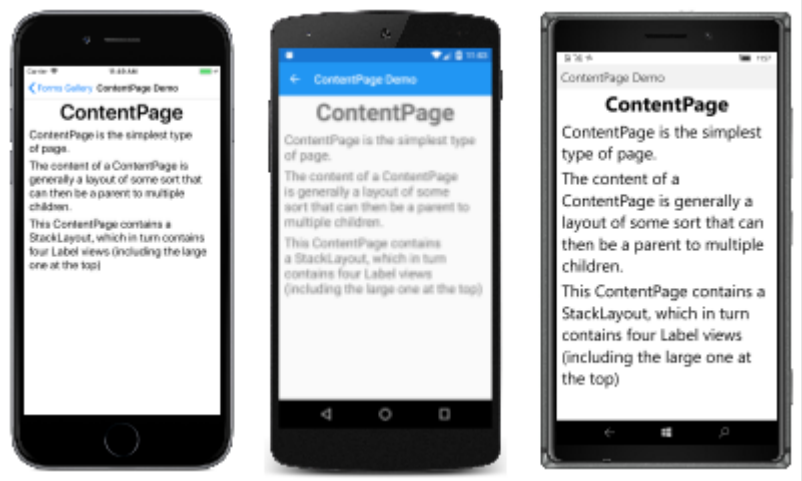
Les types de page Xamarin.Forms

Xamarin.Forms propose plusieurs types de pages pour traiter notamment les problèmes de navigation

Les types de page Xamarin.Forms

La page la plus régulièrement utilisée est la **ContentPage**, héritière de la **TemplatedPage**

Cette page permet d'afficher un ensemble de composants ordonnés dans des layouts



Les types de page Xamarin.Forms

La **MasterDetailPage** permet de disposer de deux zones d'affichage à la fois

Elle est principalement utilisée pour l'affichage d'un menu sur le côté

Elle peut aussi servir pour l'affichage 1/3 – 2/3 rencontré sur les tablettes

Les types de page Xamarin.Forms

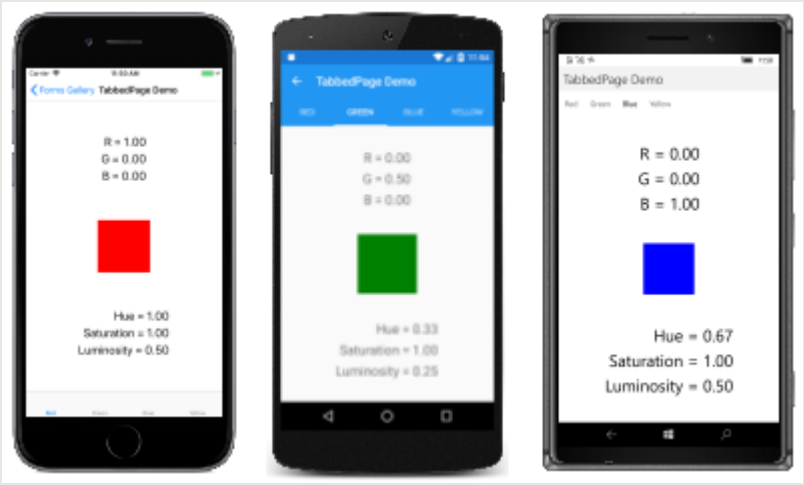
La **NavigationPage** ajoute une barre de navigation en haut de l'écran et permet d'afficher le titre de l'écran et de revenir à l'écran précédent

Page très utilisée car une application est rarement composée d'une seule page

Les types de page Xamarin.Forms

La **TabbedPage** ajoute des onglets pour permettre à l'utilisateur d'emprunter plusieurs chemins de navigation à la fois

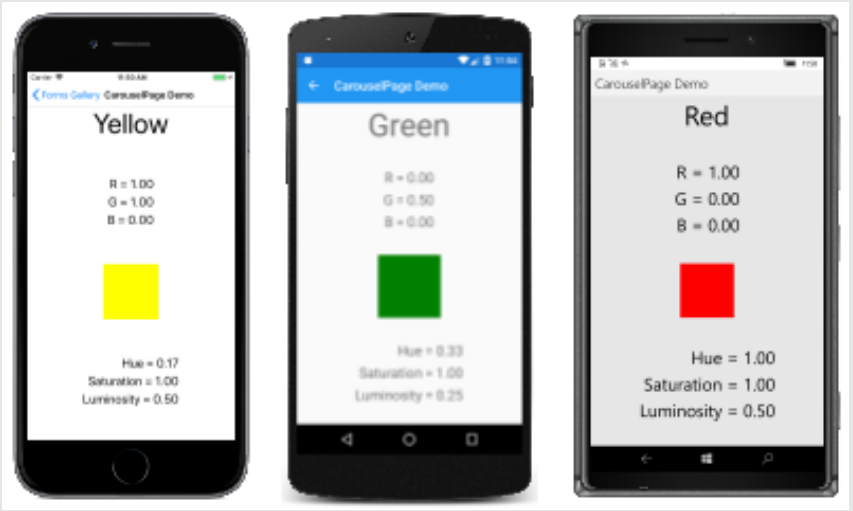
En fonction de la plateforme native, les onglets sont positionnés en haut ou en bas de l'écran



Les types de page Xamarin.Forms

La **CarouselPage** répond à un besoin très particulier qui est de faire défiler des images ou des écrans

Cela peut être très utile pour afficher un tutoriel ou faire défiler des images en plein écran



Les types de page Xamarin.Forms

Les différents types de page permettent de servir les besoins les plus courants pour développer une application mobile

Leur mise en place est immédiate et ne requiert aucun développement ou paramétrage

Pour des besoins particuliers, il reste nécessaire de développer son propre type de page ou d'avoir recours à des vues natives

Les layouts simples

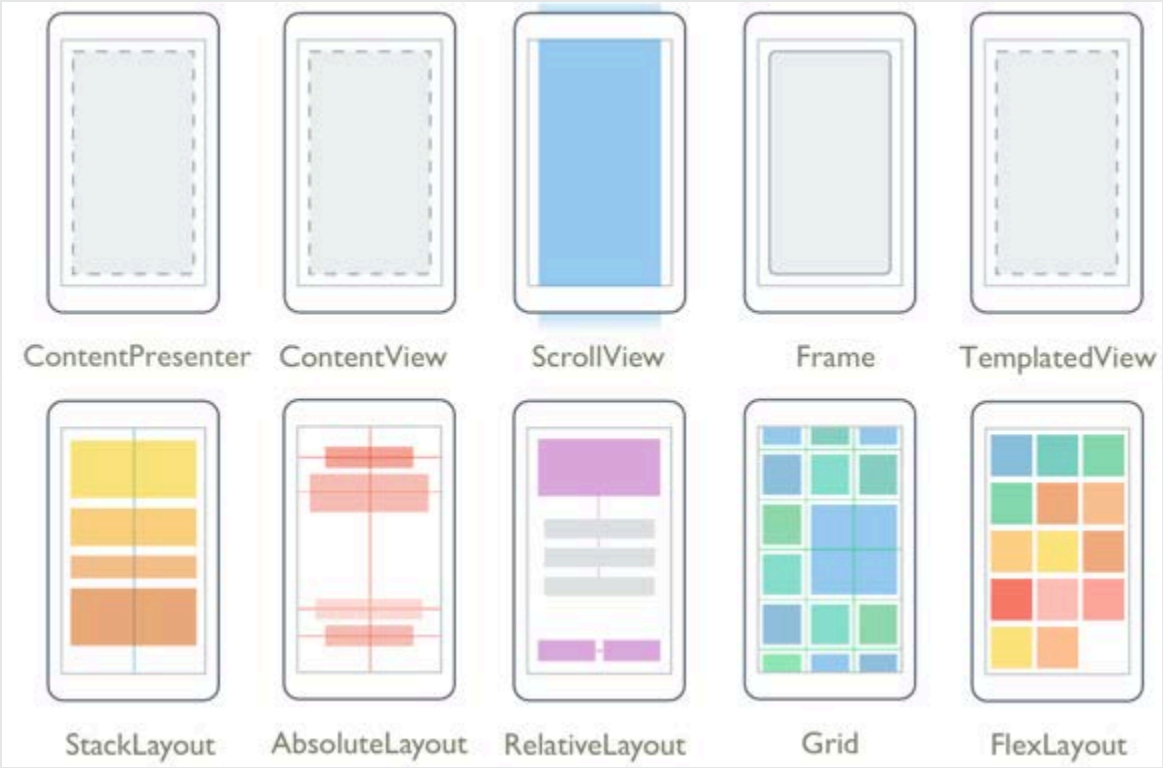
Les pages occupent tout l'écran et servent de base pour la vue : navigation, menu, carousel

La mise en page des composants à l'intérieur de la vue est réalisée grâce à des layouts

Comme pour le développement natif, il existe plusieurs layouts différents (grilles, positions absolues, relatives, etc.) et il convient d'utiliser le bon layout pour le bon besoin, et de ne pas s'acharner à utiliser toujours le même, détourné de son utilisation première

Conception de vues Xamarin.Forms

Les layouts simples



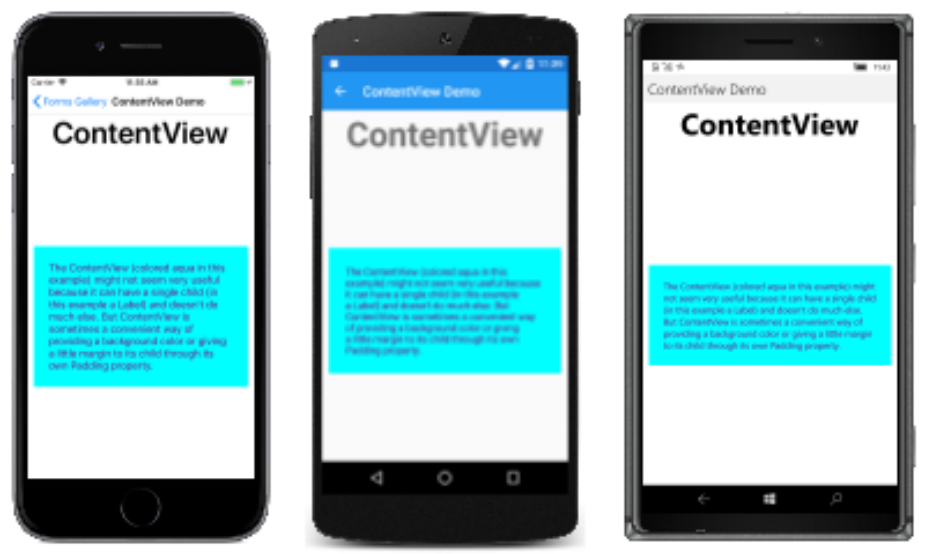
Les layouts simples

Il existe deux types de **layouts** : ceux qui ne peuvent contenir qu'un enfant, et ceux qui peuvent contenir un nombre illimité d'enfants

Pour les premiers, il s'agit généralement d'un **super-layout** qui a pour objectif de contenir un **layout** possédant plusieurs enfants

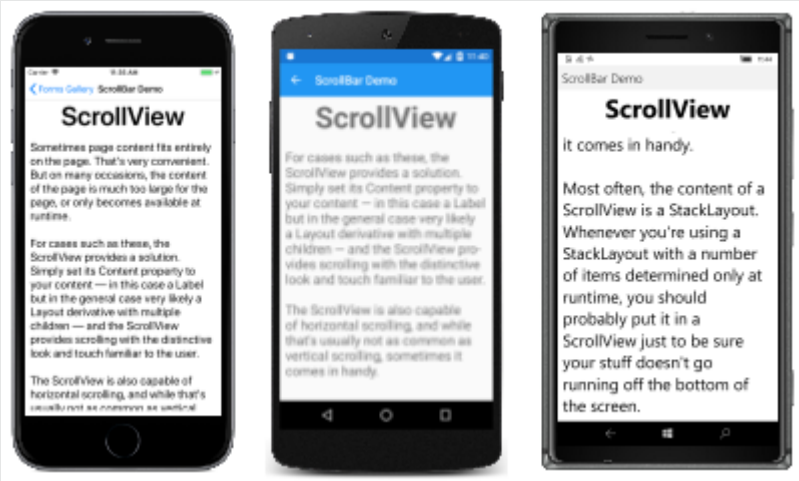
Les layouts simples

Les layouts **ContentView** et **Frame** ne peuvent contenir qu'un seul enfant.
Dans le cas de Frame, un cadre est dessiné autour du layout.



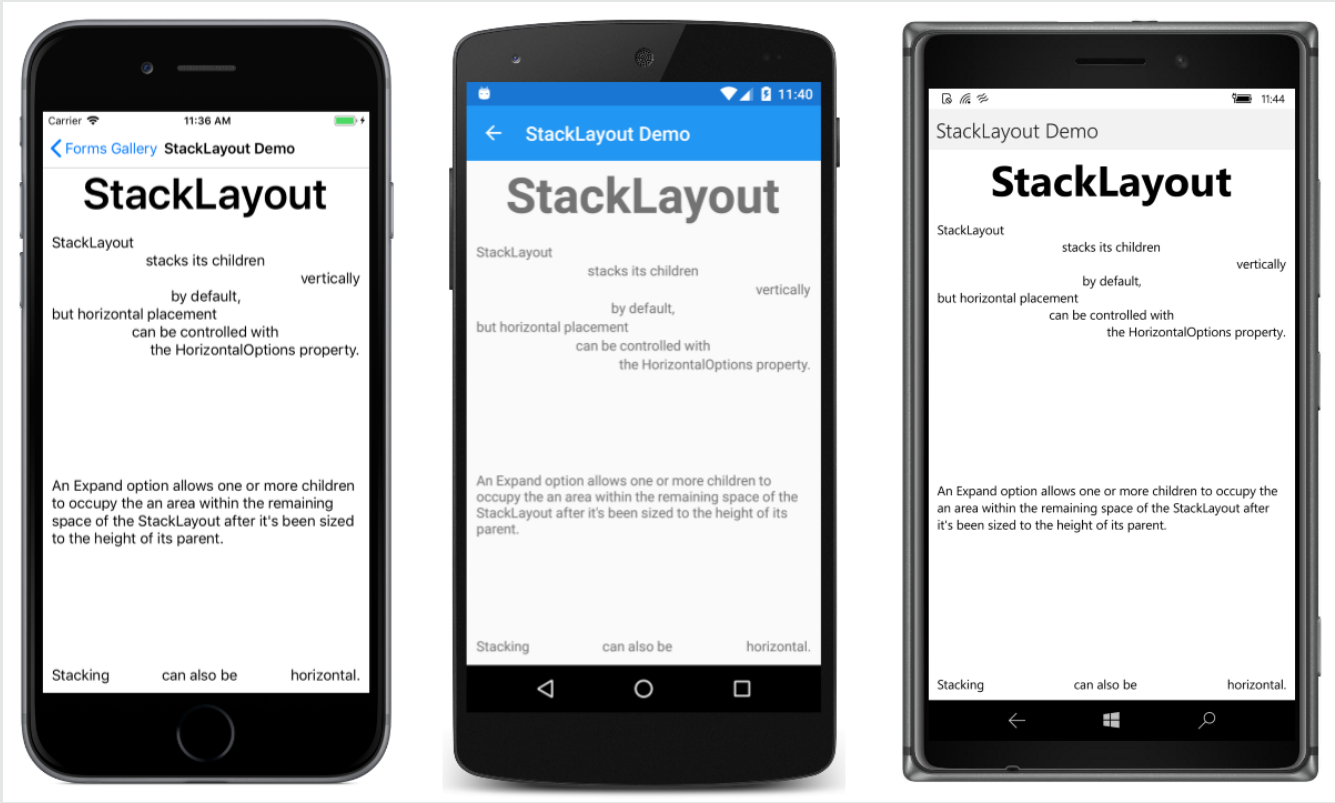
Les layouts simples

Le layout **ScrollView** contient également un seul enfant (généralement un autre type de layout) mais est capable de faire défiler son contenu (verticalement comme horizontalement) si celui-ci est plus haut ou plus large que l'écran



Le StackLayout

Le layout **StackLayout** est l'un des plus utilisés. Il permet de mettre les composants enfants les uns à côté des autres ou les uns en dessous des autres.



Le StackLayout

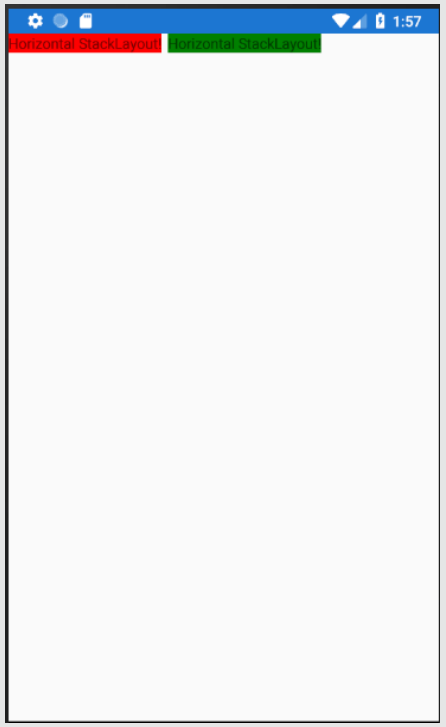
L'attribut **Orientation**, associé aux valeurs **horizontal** et **vertical**, permet de préciser si les éléments seront positionnés à côté les uns des autres ou les uns au-dessus des autres.

```
<StackLayout Orientation="Horizontal">
            <Label Text="Horizontal StackLayout!" BackgroundColor="Red"/>
            <Label Text="Horizontal StackLayout!" BackgroundColor="Green"/>
</StackLayout>

<StackLayout Orientation="Vertical">
            <Label Text="Horizontal StackLayout!" BackgroundColor="Purple"/>
            <Label Text="Horizontal StackLayout!" BackgroundColor="Gray"/>
</StackLayout>
```

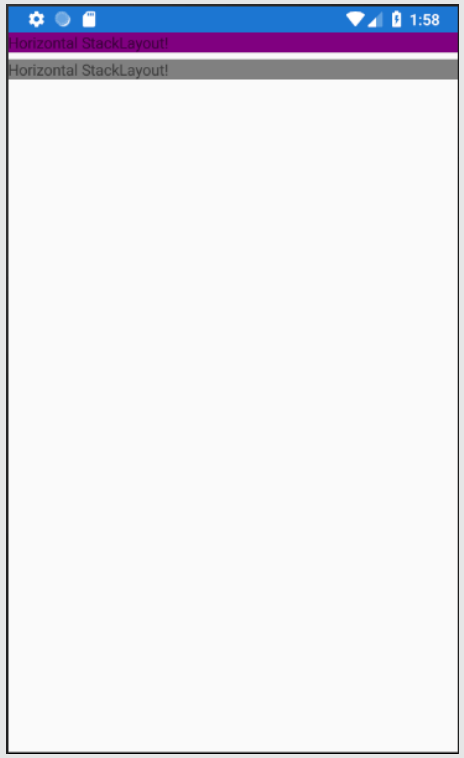
Le StackLayout

```
<StackLayout Orientation="Horizontal">
            <Label Text="Horizontal StackLayout!" BackgroundColor="Red"/>
            <Label Text="Horizontal StackLayout!" BackgroundColor="Green"/>
</StackLayout>
```



Le StackLayout

```
<StackLayout Orientation="Vertical">
            <Label Text="Horizontal StackLayout!" BackgroundColor="Purple"/>
            <Label Text="Horizontal StackLayout!" BackgroundColor="Gray"/>
</StackLayout>
```

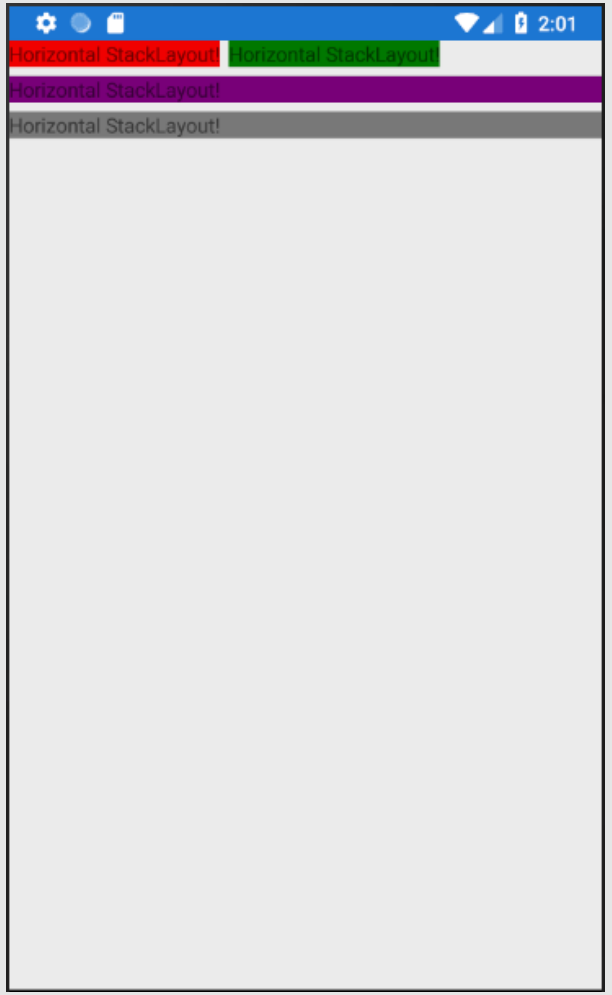


Le StackLayout

Il est bien sûr possible de combiner des StackLayout pour changer l'orientation d'une zone de l'écran

```
<StackLayout Orientation="Vertical">
        <StackLayout Orientation="Horizontal">
            <Label Text="Horizontal StackLayout!" BackgroundColor="Red"/>
            <Label Text="Horizontal StackLayout!" BackgroundColor="Green"/>
        </StackLayout>
        <StackLayout Orientation="Vertical">
            <Label Text="Horizontal StackLayout!" BackgroundColor="Purple"/>
            <Label Text="Horizontal StackLayout!" BackgroundColor="Gray"/>
        </StackLayout>
    </StackLayout>
```

Le StackLayout



Le StackLayout

La taille de chaque enfant dans le layout peut être précisée à travers les attributs **WidthRequest** et **HeightRequest**

Ces tailles sont spécifiées sans unités, car Xamarin.Forms utilise une unité spéciale, comparables au dp d'Android, au DIP d'iOS et au DIU de Windows

64 unités Xamarin.Forms correspondent à un centimètre sur l'écran, indépendamment de la résolution, de la densité de pixels ou de la taille de l'écran

```
<Label Text="Fixed StackLayout!" WidthRequest="50 " HeightRequest="10"/>
```

Le StackLayout

Les tailles fixes sont rarement utilisées, sauf pour des besoins spécifiques (icônes, logos, etc.)

Xamarin.Forms propose deux moyens pour aligner et définir la taille des éléments de manière dynamique

Xamarin.Forms détermine alors la position et la taille des composants les uns par rapport aux autres en répartissant l'espace libre sur la hauteur et/ou la largeur

Le StackLayout

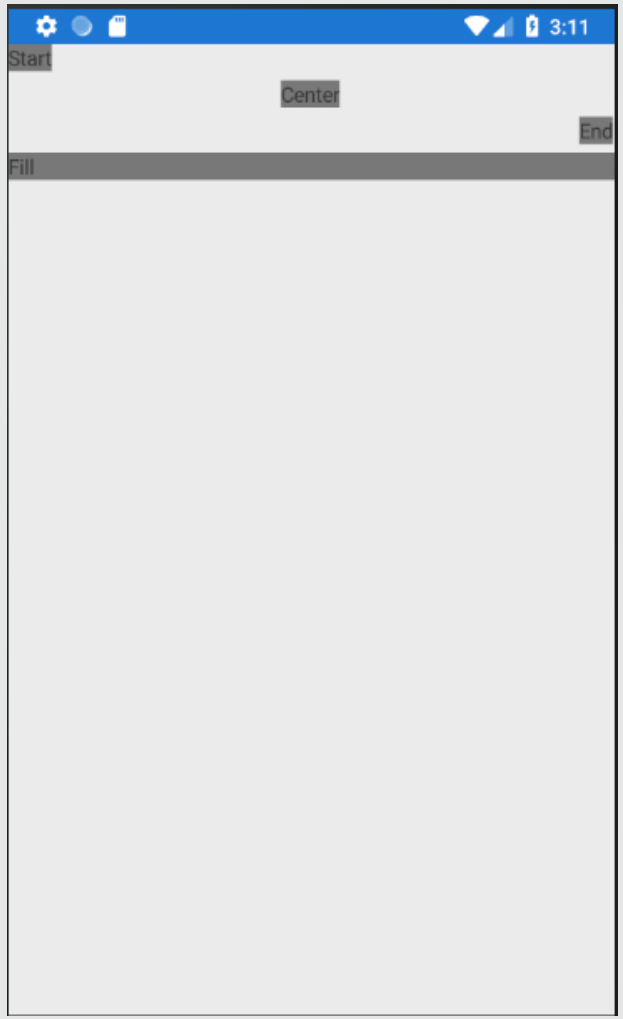
L'alignement et la taille sont définis grâce aux attributs **HorizontalOptions** et **VerticalOptions**

L'alignement peut recevoir les valeurs **Start**, **Center**, **End** et **Fill**

```
<StackLayout Orientation="Vertical">
        <Label Text="Start" BackgroundColor="Gray" HorizontalOptions="Start" />
        <Label Text="Center" BackgroundColor="Gray" HorizontalOptions="Center" />
        <Label Text="End" BackgroundColor="Gray" HorizontalOptions="End" />
        <Label Text="Fill" BackgroundColor="Gray" HorizontalOptions="Fill" />
    </StackLayout>
```

Conception de vues Xamarin.Forms

Le StackLayout



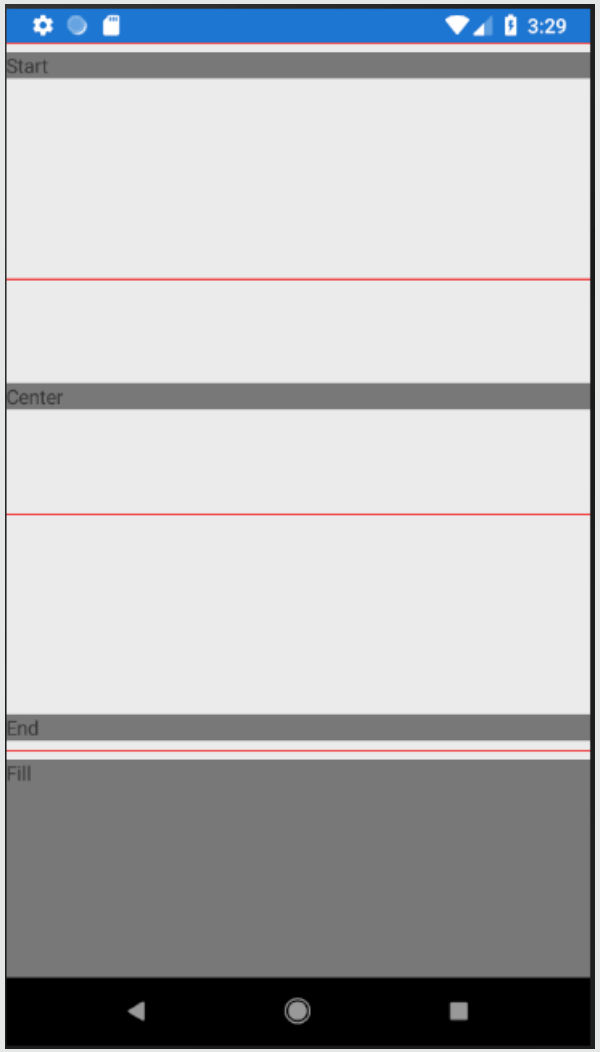
Le StackLayout

La taille peut être fixée, ajustée ou Xamarin.Forms peut répartir l'espace restant entre les composants qui le souhaitent.

Cette dernière opération est réalisée en ajoutant le suffixe **AndExpand** à l'alignement souhaité

```
<StackLayout Orientation="Vertical">
        <Label Text="Start" VerticalOptions="StartAndExpand" />
        <Label Text="Center" VerticalOptions ="CenterAndExpand" />
        <Label Text="End" VerticalOptions ="EndAndExpand" />
        <Label Text="Fill" VerticalOptions ="FillAndExpand" />
</StackLayout>
```

Le StackLayout



Le StackLayout

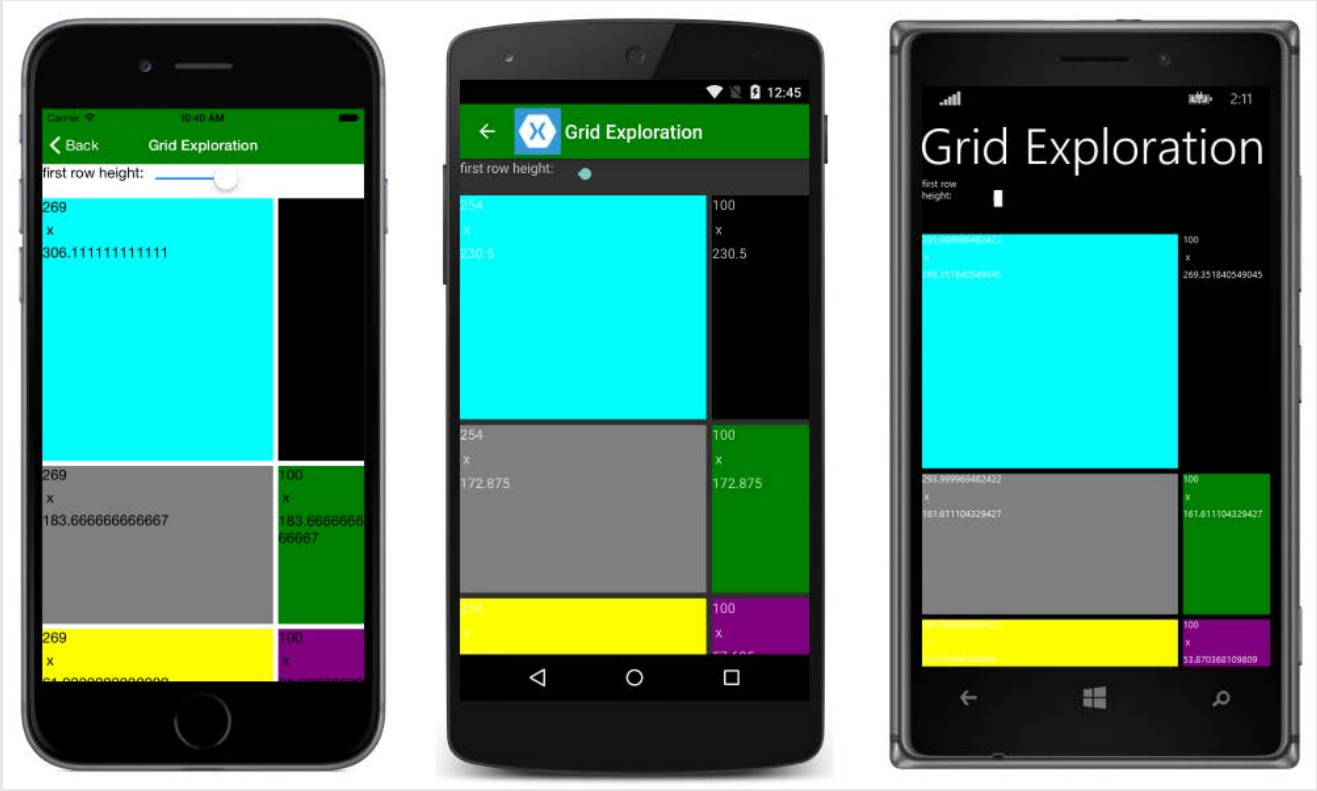
Le StackLayout permet de construire à lui seul la grande majorité des interfaces classiques : listes, formulaires, cartes de fil d'actualité

L'utilisation des propriétés d'alignement et de répartition de l'espace libre permettent de réaliser simplement des interfaces harmonieuses et responsives

Pour des besoins spécifiques comme l'affichage de tableaux ou le fait de positionner des éléments à un emplacement précis dans l'écran, le StackLayout n'est pas le plus adapté

Le GridLayout

Le layout **GridLayout** est utilisé principalement pour représenter des tableaux ou des formulaires



Le GridLayout

Le GridLayout repose, comme pour les tableaux HTML, sur un système de lignes et de colonnes

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition/>
            <RowDefinition/>
            <RowDefinition/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
</Grid>
```

Le GridLayout

- La taille des lignes et des colonnes peut être fixée suivant trois modes :
  - Automatique : la taille est adaptée au contenu
  - * (équivalent au **AndExpand**) : répartit de manière équitable l'espace entre les lignes et les colonnes qui bénéficient de cette taille
  - Absolue : une taille fixe est définie par le développeur

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
            <RowDefinition Height="200" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
</Grid>
```

Le GridLayout

Pour positionner du contenu dans le tableau, il suffit d'ajouter n'importe quel layout ou vue dans celui-ci, en précisant pour chacun à quelle colonne et à quelle ligne il doit être ajouté

```
<Grid>
        <Grid.RowDefinitions>
            …
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            …
        </Grid.ColumnDefinitions>
        <Label Text="Top Left" Grid.Row="0" Grid.Column="0" />
        <Label Text="Top Right" Grid.Row="0" Grid.Column="1" />
        <Label Text="Bottom Left" Grid.Row="1" Grid.Column="0" />
        <Label Text="Bottom Right" Grid.Row="1" Grid.Column="1" />
</Grid>
```

Le GridLayout

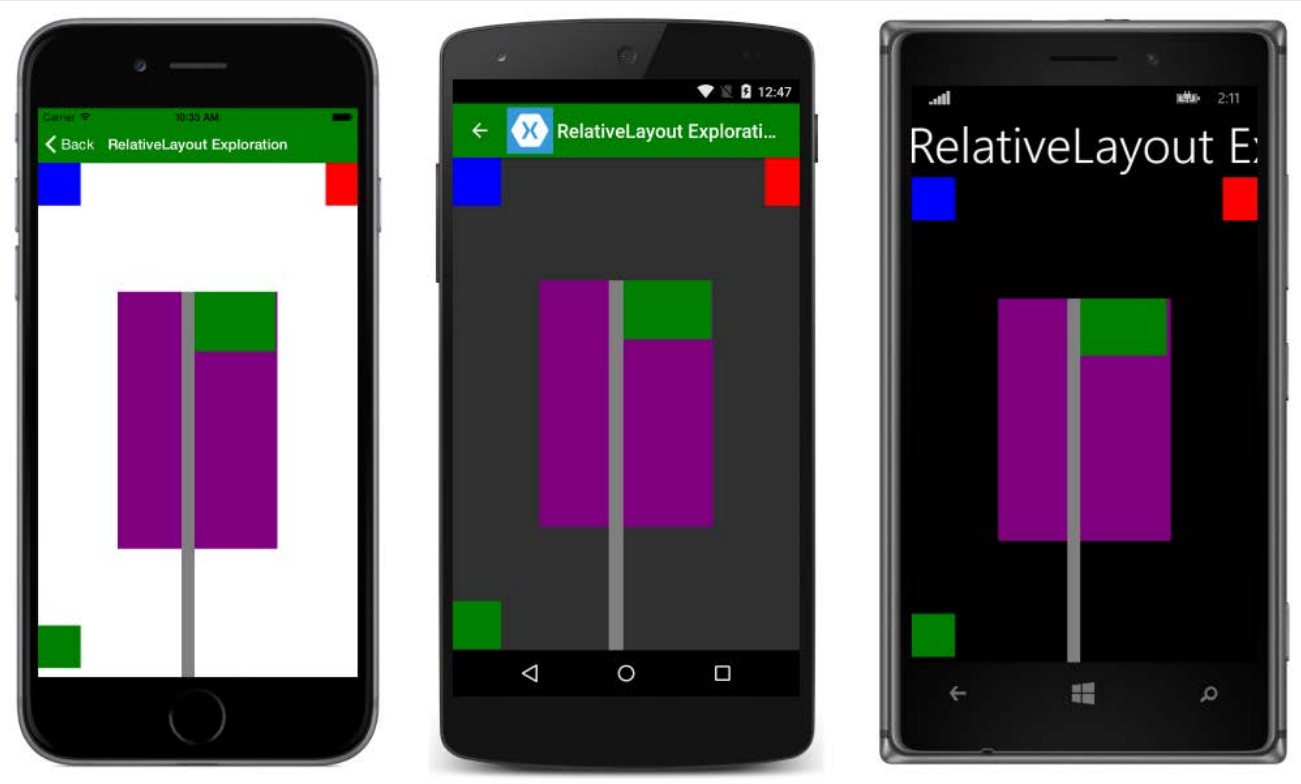
- Divers attributs permettent d'affiner l'affichage des tableaux
  - **ColumnSpacing** pour régler l'espacement entre colonnes
  - **RowSpacing** pour régler l'espacement entre lignes
  - **Grid.ColumnSpan** (sur une vue) pour que cette vue occupe plusieurs colonnes
  - **Grid.RowSpan** (sur une vue) pour que cette vue occupe plusieurs lignes

Le GridLayout

```
<Grid RowSpacing="1" ColumnSpacing="1">
        <Grid.RowDefinitions>
            <RowDefinition Height="150" />
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>
        <Label Text="0" Grid.Row="0 Grid.ColumnSpan="2" Grid.RowSpan="2" />
    </Grid>
```

Le RelativeLayout

Le layout RelativeLayout est utilisé lorsqu'il faut positionner des éléments les uns par rapport autres : typiquement pour des mises en forme complexes



Le RelativeLayout

Dans un RelativeLayout, chaque élément doit expliciter comment il est positionné par rapport à un autre élément ou par rapport au parent, sur l'axe X et sur l'axe Y

```
<BoxView Color="Green" WidthRequest="50" HeightRequest="50"
RelativeLayout.XConstraint = "{ConstraintExpression
       Type=RelativeToParent,
       Property=Width, Factor=0.5,
       Constant=-100}"
RelativeLayout.YConstraint = "{ConstraintExpression
       Type=RelativeToParent,
       Property=Height,
       Factor=0.5,
       Constant=-100}"
/>
```

Le RelativeLayout

- Plusieurs attributs permettent de paramétrer la position par rapport à un autre élément
  - **Type** : indique le type d'alignement : **RelativeToParent** ou **RelativeToView**
  - **ElementName** : désigne l'élément sur lequel s'aligner dans le cas d'un type **RelativeToView** (défini à l'aide de l'attribut **x:Name**)
  - **Property** : définit sur quelle propriété s'aligner : **Width** ou **Height**
  - **Factor** : indique si une proportion des propriétés doit être utilisée (de 0 à 1)
  - **Constant** : définit une valeur absolue à utiliser pour s'aligner avec l'objet de référence. La valeur peut être positive ou négative

Le RelativeLayout

```
<RelativeLayout>
        <BoxView Color="Red" x:Name="redBox"
                RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
                        Property=Height,Factor=.15,Constant=0}"
                RelativeLayout.WidthConstraint="{ConstraintExpression
                        Type=RelativeToParent,Property=Width,Factor=1,Constant=0}"
                RelativeLayout.HeightConstraint="{ConstraintExpression
                        Type=RelativeToParent,Property=Height,Factor=.8,Constant=0}" />
        <BoxView Color="Blue"
                RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToView,
                        ElementName=redBox,Property=Y,Factor=1,Constant=20}"
                RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToView,
                        ElementName=redBox,Property=X,Factor=1,Constant=20}"
                RelativeLayout.WidthConstraint="{ConstraintExpression
                        Type=RelativeToParent,Property=Width,Factor=.5,Constant=0}"
                RelativeLayout.HeightConstraint="{ConstraintExpression
                        Type=RelativeToParent,Property=Height,Factor=.5,Constant=0}" />
</RelativeLayout>
```

Démonstration

Conception de vues Xamarin.Forms

Manipuler les layouts

- Manipuler les StackLayout
  - Comprendre les différentes propriétés permettant de préciser la stratégie de répartition de l'espace vide et l'alignement des composants

Démonstration

Conception de vues Xamarin.Forms

Manipuler les layouts

- Manipuler les GridLayout
  - Savoir construire un tableau avec l'aide du Gridlayout

Conception de vues Xamarin.Forms

Construire un écran complexe grâce aux layouts

TP