Table des matières

Introduction

Dans un premier temps, les différentes parties essentielles pour la réalisation d'une application Android, seront réalisées de manière dirigée et tous ensemble.

Xamarin est un produit qui met à la disposition des développeurs une implémentation du Framework .NET sur des environnements divers et variés comme Android, iOS ou macOS (alors qu'initialement, la plateforme .NET était uniquement utilisable dans les environnements Microsoft).

Xamarin permet d'écrire du code C# et d'utiliser le Framework .NET sur iOS et Android avec 100 % des API iOS et 100 % des API Android ! Autrement dit, tout ce que vous pouvez faire en Swift/Objective-C pour iOS et Java pour Android est réalisable avec .NET/C#.

Installation de l'environnement

Intégration avec Visual Studio

Le développement pour Android reposant sur le SDK Java et sur le SDK Android, il est possible de le configurer dans Visual Studio.

Menu Tools → Options → Xamarin → Android Settings.

Emulateurs Android

Il existe un très grand nombre de périphériques exécutant Android, que ce soit des téléphones, des tablettes, des « phablettes », etc. Donc un maximum de tests serait le mieux pour valider l'application.

La fragmentation d'Android est telle qu'il est peu concevable d'acheter un modèle de chaque version d'Android (avec, pour chaque version, un modèle/constructeur différent). Le mieux est d'utiliser des émulateurs qui permettront de simuler les différents environnements.

Il existe actuellement trois types d'émulateurs différents qui peuvent être utilisés pour lancer un système Android sur sa machine :

- Visual Studio Emulator pour Android qui repose sur la technologie Hyper-V.
- Google Android SDK Emulator qui repose sur HAXM.
- Genymotion qui repose sur Virtual Box.

Certains conflits peuvent apparaître lorsque vous utilisez plusieurs de ces solutions lors de vos développements, il est donc recommandé de n'utiliser qu'une seule technologie sur sa machine.

Android SDK Manager

Vous serez amené très souvent à modifier les API que vous souhaitez cibler dans vos applications. Celles-ci changeant assez souvent, il est nécessaire de les installer, de les mettre à jour ou simplement les supprimer lorsqu'elles sont obsolètes. Android fournit un logiciel appelé Android SDK Manager qui permet de faire cela très simplement.

Depuis la version 25.2.3 des Android SDK Tools, l'application Android SDK Manager de Google est déprécié à la faveur de la version Xamarin Android SDK Manager. C'est celle qui est installée dans notre environnement.

Menu Tools → Android → Android SDK Manager.

Compilation

Lorsque l'on compile une application en C#, le compilateur transforme le code C# en code **MSIL** (MicroSoft Intermediate Language). Puis, en fonction des plateformes, ce code est encore transformé pour créer un exécutable qui fonctionne sur la plateforme ciblée.

Spécificité iOS

Apple a interdit l'exécution de code généré dynamiquement sur le périphérique. Le code MSIL n'est donc pas transformé dynamiquement en code natif. Xamarin iOS ajoute une étape de compilation appelée AOT (Ahead-of-Time) qui compile ce code MSIL en code natif. Le binaire qui en résulte est très proche d'un binaire que l'on obtiendrait en compilant une application écrite en Objective C ou Swift.

Spécificité Android

Ici, le modèle de compilation n'est pas de type AOT, mais de type JIT (Just-In-Time), ce qui permet, sur le papier, de gagner en performance. La grosse différence entre les deux types est que l'AOT compile le programme avant qu'il ne soit exécuté alors que JIT compile le programme pendant qu'il s'exécute.

Il est néanmoins possible d'activer la compilation AOT pour Android pour générer son application en Release. Cela se fait au détriment de la taille du package généré, mais d'un lancement plus rapide de l'application au démarrage.

Différence entre Debug et Release

L'application peut être compilée en deux modes :

- Debug : mode destiné à tester votre application.
- Release : mode destiné au déploiement et à l'exécution finale.

En mode Debug, vous exécutez en réalité du code IL (Intermediate Language), qui est traduit par le compilateur JIT (Just-In-Time) en code natif. La compilation et le déploiement sont alors plus rapides et les diagnostics de votre application sont simplifiés.

Quand vous utilisez le mode Release, c'est la chaîne de compilation native qui s'exécute. Le binaire compilé ne contient alors pas les bibliothèques du Framework NET. Le mode Release optimise votre code et peut dès lors empêcher certains scénarios de débogage.

Il est important tout de même de tester régulièrement votre application dans ce mode Release, car c'est celui qui sera exécuté par vos utilisateurs finaux.

MVVM

L'architecture MVVM (Modèle Vue-Vue Modèle) est l'architecture idéale pour partager du code entre applications cross-plateformes. Une architecture MVVM est composée de trois parties :

- Le Model (Modèle) qui correspond aux données que l'on manipule et affiche.
- La View (Vue) qui correspond à l'interface d'affichage.
- Le ViewModel (Vue-Modèle) qui sert d'interface entre le modèle et la vue en affichant les données du modèle grâce au Binding et en interprétant les interactions faites depuis la vue.

Le pattern MVVM permet le découpage des responsabilités des différentes couches applicatives, la liaison de données avec la vue ou la possibilité de tester facilement le code contenu dans les ViewModel.

Dans le cadre de Xamarin, cette testabilité est un véritable atout dès lors que nous plaçons un maximum de code commun dans ces ViewModel.

Paramètres utilisateur

Comme pour le système de fichiers, le système de paramètres utilisateur est différent selon le système d'exploitation. De ce fait, il y a plusieurs approches pour accéder aux fichiers.

Android

Android permet d'utiliser la classe SharedPreferences (Préférences partagées) afin de stocker des données de manière persistante sous forme de clé-valeur. Elle ne gère que les types primitifs comme bool, string, int, float et long.

Les données ainsi sauvegardées sont persistées même lorsque l'application est complètement quittée.

Tout comme pour la partie système de fichiers, il est possible d'utiliser du code unifié pour accéder aux paramètres de l'utilisateur fonctionnant aussi bien sur Android que sur iOS.

Pour cela, un composant Xamarin existe : Settings Plugin.

Utilisation d'une base de données locale

Le besoin d'une base de données apparaît dès lors que la volumétrie de données devient importante. Une autre raison pouvant conduire à l'utilisation d'une base de données est le besoin d'architecturer selon un modèle relationnel ses données.

SQLite

Plusieurs moteurs de base de données existent avec chacun leurs avantages et inconvénients. Celui qui colle au mieux au besoin d'une application mobile est SQLite, car il a l'avantage d'être extrêmement léger avec également peu d'adhérence au système d'exploitation utilisé puisqu'il se base sur un système de fichiers.

Il est donc très facile à intégrer aux développements Xamarin et de nombreux composants existent pour permettre cette intégration.

Autre avantage non négligeable : sa réputation n'étant plus à faire, une large documentation est disponible sur le Web.

La Base de Données reste cependant locale et ne permet pas de créer une application partagée.

PCL SQLite-Net

Pour pouvoir utiliser le code sur plusieurs plateformes, le mieux est d'installer une PCL (Portable Class Libraries).

Pour ajouter la PCL **SQLite-Net**, il faut utiliser le gestionnaire de paquets NuGet.

Dans la bibliothèque partagée, il faut alors ajouter le paquet SQLite.Net.Async-PCL. Son acquisition comporte également le paquet NuGet SQLite.Net.Core-PCL dépendant.

Classe de données

Les tables d'une base de données sont générées en se basant sur des classes de données.

Voici une classe représentant un type de donnée basique :

```
// pour pouvoir utiliser les mots clés PrimaryKey et AutoIncrement
using SQLite.Net.Attributes;

public class DataItem {
   [PrimaryKey, AutoIncrement]
   public int ID { get; set; }
   public string Label { get; set; }
   public string Description { get; set; }

   public override string ToString(){
      return $"{{ID={ID}, Label={Label}, Description={Description}}}";
   }
}
```

[PrimaryKey] et [AutoIncrement] permettent de définit la clé primaire et son incrémentation automatique.

D'autres attributs sont disponibles tels que [Ignore] qui permet d'indiquer qu'une propriété déclarée dans la classe C# ne doit pas être interprétée par SQLite et donc pas ajoutée à la table associée ou [Indexed], [MaxLength] et [NotNull].

Service d'accès aux données

Il faut créer une interface entre la couche métier et la base de données. Celle-ci peut être mise dans un projet à part. Cette interface se matérialise par un Repository nommé ici **DataItemRepository**.

```
using System.Collections.Generic;
using System.Threading.Tasks;
using SQLite.Net.Async;
using SQLite.Net.Interop;

// classe utilisée pour définir les actions disponibles sur la table DataItem.
public class DataItemRepository
{
   // propriété qui correspond à l'implémentation SQLite spécifique à la plateforme.
   private SQLiteAsyncConnection _connection;

   // méthode d'initialisation prenant en paramètre le chemin vers la base de
   // données et l'implémentation SQLite spécifique.
   public async Task InitializeAsync(string path,ISQLitePlatform sqlitePlatform)
   {
      _connection = SQLiteDatabase.GetConnection(path, sqlitePlatform);

      // Création de la base de données et table
      await _connection.CreateTableAsync<DataItem>();
   }

   // quatre méthodes principales d'accès aux données
   // ***********************************************
   // Récupération des éléments d'une table
   public async Task<IEnumerable<DataItem>> GetAllAsync()
   {
      var entities =
         await _connection.Table<DataItem>().OrderBy(m => m.Label).ToListAsync();
      return entities;
   }

   // récupération d'un élément spécifique
   public async Task<DataItem> GetAsync(int id)
   {
      var entity = await _connection.GetAsync<DataItem>(id);
      return entity;
   }

   // Ajout d'une entrée à la base de données
   public async Task<DataItem> CreateAsync(string label, string description)
   {
      var entity = new DataItem()
      {
         Label = label,
         Description = description
      };
      var count = await _connection.InsertAsync(entity);
      return (count == 1) ? entity : null;
   }

   // suppression d'une entrée à la base de données
   public async Task<bool> DeleteAsync(int id)
   {
      var count = await _connection.DeleteAsync<DataItem>(id);
      return (count == 1);
   }
}
```

Implémentation SQLite

La couche d'accès aux données est créée. Il faut ensuite l'implémenter pour chaque plateforme !

Il faut expliquer à Xamarin comment faire la liaison entre notre couche de service et l'implémentation réelle sur les plateformes.

Il faut commencer par ajouter le paquet NuGet SQLite-Net au projet.

Pour Android, il n'y a rien de spécifique à faire, car SQLite est installé par défaut. Il suffit d'implémenter la récupération du chemin de la base de données ainsi que l'initialisation du repository en s'appuyant, cette fois-ci, sur la classe SQLite.Net.Platform.XamarinAndroid.SQLitePlatformAndroid.

```
// récupération du fichier local de base de données pour Android
private async Task<string> GetDbPathAsync()
{
   var folder =
      System.Environment.GetFolderPath(System.Environment.SpecialFolder.Personal);
   var path = Path.Combine(folder, "SQLiteDb.db3");
   if (!File.Exists(path))
   {
      File.Create(path);
   }

   return path;
}

// implémentation ISQLite d'accès à la connexion SQLite pour Android
private DataItemRepository _dataItemRepository;

private async Task InitializeRepositoryAsync()
{
   _dataItemRepository = new DataItemRepository();

   var path = await GetDbPathAsync();

   await _dataItemRepository.InitializeAsync(path,
      new SQLite.Net.Platform.XamarinAndroid.SQLitePlatformAndroid());

   var dataItems = await GetAllDataItemsAsync();
   foreach (var dataItem in dataItems)
   {
      DataItems.Add(dataItem);
   }
}
```

Une fois cette implémentation initialisée, nous pouvons utiliser l'instance de repository pour effectuer des actions sur la base de données.

```
// récupération des éléments de la table depuis le projet Android
private async Task<IEnumerable<DataItem>> GetAllDataItemsAsync()
{
   var dataItems = await _dataItemRepository.GetAllAsync();
   return dataItems;
}

// ajout d'un élément à la table depuis le projet Android
private async Task CreateDataItemASync(string label, string description)
{
   var dataItem = await _dataItemRepository.CreateAsync(label, description);
   DataItems.Add(dataItem);
}
```

Utilisation de Webservices

Il est possible de devoir utiliser des données distantes. Il s'agit en fait d'applications clientes consommant des services web. Ces services peuvent avoir différentes topologies et utiliser différentes technologies.

Verbes HTTP

REST (Representational State Transfer) est un protocole basé sur des requêtes effectuées via le protocole HTTP et les verbes associés :

GET, POST, PUT, DELETE.

Création de requêtes

API

Depuis la sortie de Windows 10, une nouvelle API est disponible : Windows.Web.Http.HttpClient. Celle-ci est fortement recommandée par Microsoft en lieu et place de System.Net.Http.HttpClient.

L'ancienne API, System.Net.Http, est toujours disponible et de nombreuses documentations en ligne, dont la documentation officielle de Xamarin, l'utilisent toujours. La suite du document se basera sur son usage.

Création de requêtes

On peut utiliser la classe HttpClient pour créer et recevoir des requêtes via le protocole HTTP.

```
// création d'un objet HttpClient
var httpClient = new System.Net.Http.HttpClient();
```

Une fois le client créé, il convient d'indiquer l'URI (Uniform Resource Identifier) sur lequel la requête doit être interrogée pour récupérer les données. Il s'agit en fait d'une adresse réseau pointant vers une ressource web, respectant une syntaxe particulière.

```
// création d'un URI de ressource service web
var restUrl = "<adresse web>";
var uri = new Uri(restUrl);
```

Il reste à créer la requête correspondante.

```
// création d'une requête HTTP GET
var result = await httpClient.GetAsync(uri);
```

Plusieurs méthodes de la classe HttpClient sont disponibles pour requêter les différents verbes, par exemple :

GetAsync (requête GET), PostAsync, PutAsync et DeleteAsync.

Headers

Dans certains cas, des en-têtes de requêtes sont demandés par le service.

Il peut s'agir d'informations sur le format des données envoyées ou sur le format de données à recevoir.

```
// ajout d'un header à une requête HTTP
httpClient.DefaultRequestHeaders.TryAddWithoutValidation("Content
-Type", "application/x-www-form-urlencoded");
```

Contenu de requête

Dans le cas des actions POST et PUT, le serveur s'attendra à recevoir des données concernant les entités à créer ou à mettre à jour. Ces données sont contenues dans des objets de type HttpContent.

Il existe plusieurs classes héritant de HttpContent et permettant d'encapsuler des données formalisées sous forme de formulaires, de chaînes de caractères brutes, de tableau d'octets ou de contenu multipart. Pour créer un contenu de requête, il faut donc instancier un de ces types de classes.

```
//Création d'un contenu HttpContent
var parameters = new List<KeyValuePair<string, string>>
{
   new KeyValuePair<string, string>("clef_1", "valeur_1"),
   new KeyValuePair<string, string>("clef_2", "valeur_2"),
```

```
};
HttpContent body = new FormUrlEncodedContent(parameters);
```

L'envoi de ce contenu se fait alors par l'utilisation du paramètre content de la méthode PostAsync ou PutAsync.

```
// envoi d'une requête POST avec contenu
var result = await _httpClient.PostAsync(new Uri(url), body);
```

Réponses

Statut de réponse

Le format des réponses http fournies par un service web est réglementé. Ainsi, une action indisponible, une action réalisée avec succès ou avec erreur doivent être encapsulées dans un message contenant un code de statut. Il existe plusieurs dizaines de codes, par exemple :

- 200 : succès de la requête.
- 301 et 302 : redirection, respectivement permanente et temporaire.
- 401 : utilisateur non authentifié.
- 403 : accès refusé.
- 404 : ressource non trouvée.
- 500 et 503 : erreur serveur.

Le retour des méthodes de requête est de type HttpResponseMessage. Ce dernier comporte une propriété HttpStatusCode contenant le code de statut de la réponse HTTP.

Il est dès lors possible de vérifier qu'une requête s'est bien déroulée via ce code.

```
// vérification du code de statut d'une réponse http
if (result.StatusCode != System.Net.HttpStatusCode.OK)
{
    // Gestion d'erreur
}
```

Afin de ne pas avoir à gérer tous les statuts d'erreur ou de succès, une autre propriété est disponible : **IsSuccessStatusCode**, qui renvoie vrai si le code est compris entre 200 et 299.

Il est également possible d'utiliser la méthode **EnsureSuccessStatusCode** qui lèvera une exception si la propriété IsSuccessSatusCode est fausse.

```
result.EnsureSuccessStatusCode();
```

Lecture de la réponse

Une fois que le statut controllé, il faut alors lire le contenu de la réponse. Ce contenu est encapsulé dans la propriété **Content** de la réponse du type **HttpContent**.

Plusieurs méthodes sont disponibles dans cette classe pour sérialiser le contenu :

- **ReadAsStringAsync**() : retourne le contenu de la réponse comme une chaîne de caractères.
- **ReadAsStreamAsync**() : retourne le contenu sous la forme d'un flux.
- **ReadAsByteArrayAsync**() : retourne le contenu sous forme d'un tableau d'octets.

```
// lecture d'une réponse HTTP sous forme de chaîne de caractères
var httpResp = await result.Content.ReadAsStringAsync();
```

Cycle de vie

Introduction

Chacune des plateformes possède un cycle de vie pour une application qui lui est propre. Cela ne veut pas dire qu'ils sont tous complètement différents, bien au contraire. On va, par exemple,

retrouver le même concept de suspension (ou mise en pause) d'une application (ou d'une partie) lorsque celle-ci n'est plus utilisée pour pouvoir la recharger une fois que l'utilisateur souhaite la réutiliser. Il est important de comprendre le cycle de vie d'une application car il existe des spécificités qui impliquent directement la manière de développer son application.

Pour Android, le maître-mot est **Activity**. Tout s'articule autour des activités, une application en est d'ailleurs constituée, la plupart du temps, de plusieurs. Une activité correspond à un écran ou une partie, mais peut aussi être une fenêtre utilisateur (popup). Chacune d'elle a son propre cycle de vie avec un état qui lui est propre.

Une activité est toujours rattachée à une interface utilisateur qui va permettre à l'utilisateur d'interagir avec, et à un comportement : le code à proprement parler. Contrairement à ce que l'on peut trouver dans la plupart des applications, il n'y a pas de méthode dite de « démarrage », la fameuse méthode statique Main. Chaque activité de l'application peut être définie en tant qu'activité principale et sera donc lancée au démarrage de l'application. Cela se traduit dans le code de votre activité par l'utilisation de la propriété MainLauncher dans l'attribut Activity.

```
[Activity(Label = "Sample.Droid", MainLauncher = true, Icon = "@drawable/icon")]
public class MainActivity : Activity
{
    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);
        SetContentView(Resource.Layout.Main);
    }
}
```

Lors d'un changement d'état (initialisation, suspension, destruction…), l'activité va appeler une méthode précise avant que celui-ci se fasse réellement afin de laisser la possibilité de gérer ces changements dans son application.

Les principaux changements d'état sont dus à la mise en arrière-plan des applications ce qui a pour incidence, pour le système d'exploitation, de savoir qu'elles ne seront plus utilisées tant qu'elles ne reviendront pas au premier plan.

Les différents états d'une activité

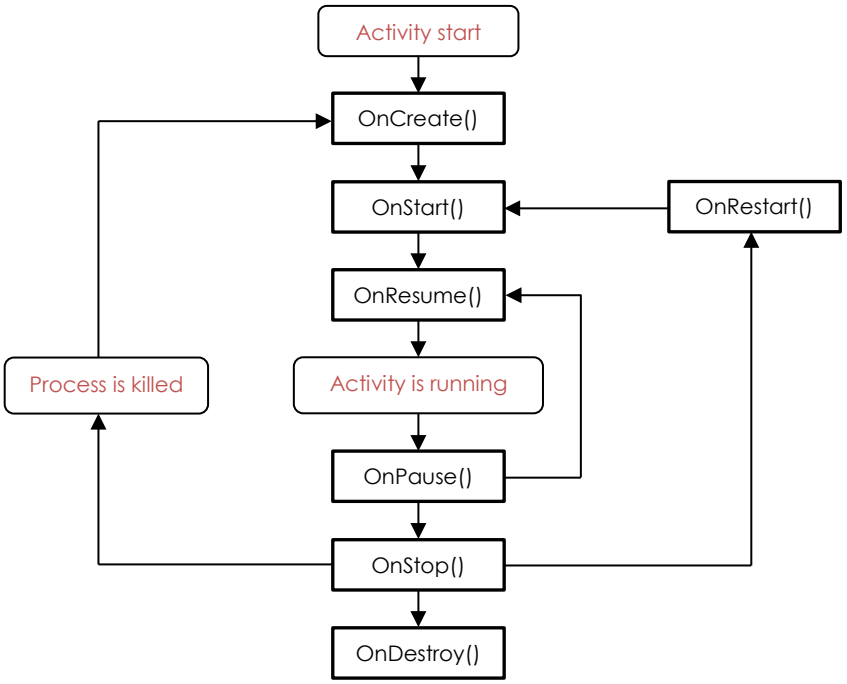
On peut ressortir quatre états principaux :

- **Activée** (Active) / **En cours d'exécution** (Running) : l'activité entre dans cet état lorsqu'elle est au premier plan, c'est donc l'activité principale que l'utilisateur peut voir et avec laquelle il va interagir directement. L'OS considère alors que celle-ci est prioritaire sur toutes les autres et ne sera fermée que dans des cas très rares comme une consommation de mémoire trop importante.
- **En pause** (Paused) : l'activité entre dans cet état lorsqu'elle devient partiellement masquée (par une autre activité non « plein écran » ou transparente) ou que l'appareil se met en veille. L'OS considère alors que l'activité ne doit pas être suspendue, c'est-à-dire que ses ressources ne doivent pas être déchargées, puisqu'elle est encore utilisée, mais de façon secondaire.
- **Arrêtée** (Stopped) / **En arrière-plan** (Backgrounded) : l'activité entre dans cet état lorsqu'elle devient totalement masquée (par une autre activité « plein écran »). L'OS considère alors qu'il peut maintenir les ressources de cette activité tant que cela n'entrave pas le fonctionnement de l'activité principale (Running) et des activités secondaires (Paused). Dans le cas contraire, les ressources seront libérées pour laisser la place aux activités prioritaires.
- **Redémarrée** (Restarted) : l'activité entre dans cet état lorsqu'elle a été effacée de la mémoire. L'OS considère alors qu'il doit recharger l'activité, si possible à partir de son précédent état, puis affichée à l'écran. Cela arrive souvent lorsque l'utilisateur appuie sur le bouton Back.

Les méthodes du cycle de vie

Du côté des développeurs, il convient de s'appuyer sur le Framework Xamarin en surchargeant simplement certaines méthodes de la classe Activity.

Illustration



OnCreate

C'est la première méthode à être appelée lors de la création d'une activité et est généralement utilisée pour l'initialisation des variables ainsi que pour définir les vues.

```
protected override void OnCreate(Bundle bundle)
{
   base.OnCreate(bundle);
   SetContentView(Resource.Layout.Main);
   if (bundle != null)
   {
      var myParam = bundle.GetString("MyParam");
   }

   SetContentView(Resource.Layout.Main);
}
```

OnStart

Elle est appelée lorsque l'activité démarre et devient visible à l'utilisateur. Il est ici possible d'interagir avec les éléments composant la vue.

```
protected override void OnStart()
{
   base.OnStart();
   var button = FindViewById<Button>(Resource.Id.MyButton);
   button. Background = GetDrawable(Resource.Drawable.Icon);
   button. Click += (e, args) =>
   {
      Console.WriteLine("Button clicked !");
   };
}
```

OnResume

Une fois la vue parfaitement affichée et que l'utilisateur peut interagir avec elle, la méthode OnResume est appelée. C'est le meilleur endroit pour initialiser les périphériques de l'appareil (GPS, camera, gyroscope…) ou de lancer une animation. Lorsque l'activité qui était en pause redevient visible, la méthode OnResume est à nouveau appelée.

OnPause

Si l'activité est partiellement masquée ou sur le point de passer en arrière-plan, le système va appeler la méthode OnPause afin de laisser le temps de sauvegarder des données en mémoire de façon persistante afin de les retrouver plus tard ou bien de supprimer les données volumineuses ou non utilisées de la mémoire.

OnStop

C'est la méthode qui est appelée lorsque l'activité passe en arrière-plan, car une autre la remplace ou, qu'elle est entièrement masquée par une autre. On peut être tenté d'effectuer des opérations de nettoyage de la mémoire à cet endroit cependant, elle n'est pas forcément déclenchée tout le temps. Lorsque l'application manque de mémoire et qu'elle a besoin d'en libérer rapidement, elle peut détruire une activité en arrière-plan de façon assez sévère. Elle ne passera alors jamais par la case OnStop, mais immédiatement par OnDestroy.

OnRestart

Elle est appelée après l'arrêt de l'activité qui a été arrêtée, avant de redémarrer, afin de pouvoir recharger le contexte de celle-ci lorsque l'utilisateur l'a quittée. À noter qu'une fois cette méthode terminée, elle déclenchera la méthode OnStart immédiatement après.

OnDestroy

C'est le dernier appel avant que l'activité ne soir supprimée de la mémoire. Elle laisse au développeur le soin de nettoyer tout ce qui pourrait rester en mémoire ou d'arrêter les tâches lancées en arrière-plan.

Les changements de configuration

Android introduit quelques exceptions possibles dans ce cycle de vie notamment via les changements de configuration (Configuration Changes). Ces changements interviennent lorsque l'affichage passe du mode portrait au mode paysage, lorsque le clavier virtuel apparaît, etc. Cela a pour conséquence la destruction de l'activité pour immédiatement la reconstruire avec une configuration différente. Bien entendu, vu que les changements s'appliquent à une activité en cours d'exécution, la gestion de la reconstruction doit être le plus rapide possible afin que cela ne se ressente pas sur l'affichage. Android met donc à disposition une API spécifique afin de pouvoir persister l'état de son activité lors d'un changement de configuration.

Navigation

Introduction

Dans la plupart des applications, il est nécessaire de pouvoir naviguer de page en page afin d'afficher du contenu, car tout ne peut pas être affiché sur une seule page.

Un cas classique que l'on rencontre dans la grande majorité des applications est l'affichage d'éléments sous forme de liste ou de grille. Lorsque l'utilisateur va appuyer sur un de ces éléments, l'application va changer de page, pour afficher une vue qui détaille l'élément sélectionné.

Le point d'entrée d'une application est toujours lié à une Activity (Activité). Celle-ci permet à l'utilisateur d'interagir avec l'application, car c'est elle qui se charge de l'affichage d'une fenêtre où sont placés les éléments graphiques (images, champs de formulaire, vidéos…). Bien souvent,

une activité représente une page en plein écran, mais il est aussi possible de créer une activité imbriquée dans une autre, on parlera d'ActivityGroup (Groupe d'activités) ainsi que des fenêtres dites « flottantes » qui auront un rendu de style popup grâce à un thème spécifique.

L'Activity

Pour définir une activité, la première étape est de créer une classe qui hérite de la classe Activity. Cette classe fournit une implémentation de base pour une activité et gère son cycle de vie.

```
public class MyActivity : Activity
```

La deuxième étape est d'ajouter l'attribut Activity à cette classe afin qu'il ajoute l'activité au fichier AndroidManifest.xml avec l'ensemble de ses propriétés.

```
[Activity(Label = "Sample.Droid", MainLauncher = true, Icon = "@drawable/icon")]
```

La troisième étape consiste à créer ce que l'utilisateur doit voir lorsque l'activité est lancée. Il s'agit du Layout. L'ensemble des Layout se trouvent dans le répertoire Resources/Layout à la racine du projet.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <Button
      android:id="@+id/MyButton"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="Ceci est la page principale" />
</LinearLayout>
```

La quatrième étape concerne la définition de la ressource à utiliser pour afficher du contenu pour notre activité. Cela se fait en appelant la méthode SetContentView(). Au niveau du cycle de vie de l'activité, nous l'appellerons dans la méthode surchargée OnCreate() qui est lancée lors de la création de l'activité.

```
protected override void OnCreate(Bundle bundle)
{
   base.OnCreate(bundle);
   SetContentView(Resource.Layout.Main); // id's dans la classe statique Resource
}
```

Navigation vers une autre activité

Pour naviguer d'une activité à une autre, il suffit d'appeler la méthode StartActivity() et lui passer en paramètre le type d'activité vers lequel on souhaite naviguer.

```
StartActivity(typeof(MainActivity));
```

Passage de données entre activités

Il peut être nécessaire de pouvoir passer des données d'une activité vers une autre afin de garder un contexte. Pour faire ceci, Android s'appuie sur la classe **Intent** qui sert à effectuer des actions spécifiques à Android comme le lancement d'un appel téléphonique, l'ouverture d'une page internet ou plus simplement d'une nouvelle page (Activity). Cette classe a le gros avantage de pouvoir transmettre des données à l'opération qu'elle souhaite faire, ce qui est très utile pour faire passer des données d'une activité à une autre.

Création de la navigation

Il faut créer un nouveau **Intent** en lui affectant les données souhaitées puis lancer la navigation grâce à la méthode **StartActivity**().

```
var activity = new Intent(this, typeof(SecondaryActivity));
activity.PutExtra("Data", "Some data from MainActivity");
StartActivity(activity);
```

Récupération des données

La récupération des données se fait généralement dans la méthode OnCreate() en utilisant les méthodes GetXXXExtra(string name) où XXX peut prendre différents types de données (string, float, int…).

```
Intent.GetStringExtra("Data");
```

Exemple de navigation (à faire faire aux apprentis)

MainActivity.cs

```
[Activity(Label = "PAGE PRINCIPALE", MainLauncher = true,Icon = "@drawable/icon")]
public class MainActivity : Activity
{
   protected override void OnCreate(Bundle bundle)
   {
      base.OnCreate(bundle);
      SetContentView(Resource.Layout.Main);

      // lien sur les boutons et leur methode evenement du click
      var btn2Act2 = FindViewById<Button>(Resource.Id.Btn2Act2);
      btn2Act2.Click += OnButtonClick;
      var btn2Act2WithData = FindViewById<Button>(Resource.Id.Btn2Act2WithData);
      btn2Act2WithData.Click += OnButtonWithDataClick;
   }
   // lancement de la deuxieme activity en cliquant sur le premier bouton
   private void OnButtonClick(object sender, System.EventArgs e)
   {
      StartActivity(typeof(SecondaryActivity));
   }
   // lancement de la deuxieme activity en passant des donnees
   private void OnButtonWithDataClick(object sender, System.EventArgs e)
   {
      var secondaryActivity = new Intent(this, typeof(SecondaryActivity));
      secondaryActivity.PutExtra("Data", "Some data from MainActivity");
      secondaryActivity.PutExtra("Age", 30);
      secondaryActivity.PutStringArrayListExtra("Names",
         new List<string> {"Jerôme", "Michel", "Paul"});
      StartActivity(secondaryActivity);
   }
}
```

Main.axml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <Button
      android:id="@+id/Btn2Act2"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="NAVIGUER VERS LA PAGE SECONDAIRE (sans données)" />
```

```
    <Button
        android:id="@+id/Btn2Act2WithData"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="NAVIGUER VERS LA PAGE SECONDAIRE (avec données)" />
</LinearLayout>
```

SecondaryActivity.cs

```
[Activity(Label = "PAGE SEC.", MainLauncher = false, Icon = "@drawable/icon")]
public class SecondaryActivity : Activity
{
   protected override void OnCreate(Bundle bundle)
   {
      base.OnCreate(bundle);
      SetContentView(Resource.Layout.Secondary);

      var btn2Act1 = FindViewById<Button>(Resource.Id.Btn2Act1);
      btn2Act1.Click += OnClick;

      // recuperation des donnees (si existe)
      var dataTextView = FindViewById<TextView>(Resource.Id.Data2Recover);
      string data = Intent.GetStringExtra("Data");
      // teste si la donnee existe
      if (!string.IsNullOrEmpty(data))
      {
         dataTextView.Text = data;
      }
   }

   private void OnClick(object sender, System.EventArgs e)
   {
      // retour a la premiere activity
      StartActivity(typeof(MainActivity));
   }
}
```

Secondary.axml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent" >
   <Button
      android:id="@+id/Btn2Act1"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="NAVIGUER VERS LA PAGE PRINCIPALE" />
   <TextView
      android:id="@+id/Data2Recover"
      android:text="No data"
      android:layout_width="match_parent"
      android:layout_height="match_parent" />
</LinearLayout>
```

Persistance des données

Durant le cycle de vie d'une activité, il se peut que celle-ci soit détruite ou stoppée pour ensuite être relancée. Lors des phases de destruction, il est possible de sauvegarder son état de différentes façons afin de le restaurer, plus tard, lorsque l'activité sera lancée à nouveau.

La manière la plus simple d'enregistrer des états est d'utiliser le dictionnaire d'objets dans le bundle de l'activité. Les données stockées dans ce dictionnaire sont sérialisées de façon simple et rapide, il est donc préférable de ne pas ajouter des objets complexes qui pourraient ralentir le chargement de la page lors de la dé-sérialisation.

Il est très facile de mettre en œuvre ce genre de scénario en s'appuyant sur les méthodes de base d'une activité comme **OnCreate**(), **OnSaveInstanceState**() ou **OnRestoreInstanceState**().

Enregistrement de l'état

Lors de la destruction ou de l'arrêt de l'activité, le système appelle automatiquement la méthode **OnSaveInstanceState**() afin de pouvoir enregistrer l'état si nécessaire.

```
protected override void OnSaveInstanceState(Bundle outState)
{
   outState.PutInt("MonEtatASauvegarder", 69);
   base.OnSaveInstanceState(outState);
}
```

Récupération de l'état

Quand une activité se lance ou se relance, le système appelle automatiquement la méthode **OnRestoreInstanceState**() afin de pouvoir récupérer un précédent état sauvegardé dans le bundle.

```
protected override void OnRestoreInstanceState(Bundle savedInstanceState)
{
   base.OnRestoreInstanceState(savedInstanceState);

   int myState = savedInstanceState.GetInt("MyState2Save");
}
```

Exemple avec compteur (à faire faire aux apprentis)

Exemple qui permet d'afficher le nombre de fois que l'on navigue vers l'activité grâce à un compteur qui est enregistré dans le bundle de manière persistante.

PersistantStateActivity.cs

```
[Activity(Label = "PERSISTANT STATE PAGE")]
public class PersistantStateActivity : Activity
{
   private int _activityDisplayCounter;
   private const string COUNTER_KEY = "Counter";
   private TextView _counterTextView;

   protected override void OnCreate(Bundle bundle)
   {
      base.OnCreate(bundle);
      SetContentView(Resource.Layout.PersistantState);

      // pour pouvoir afficher le compteur
      _counterTextView = FindViewById<TextView>(Resource.Id.CounterTextView);

      var myButton = FindViewById<Button>(Resource.Id.MyButton);
      myButton.Click += OnClick;

      HandleState(bundle);
   }

   private void OnClick(object sender, System.EventArgs e)
   {
      StartActivity(typeof(MainActivity));
   }

   protected override void OnSaveInstanceState(Bundle outState)
   {
      outState.PutInt(COUNTER_KEY, _activityDisplayCounter);
      base.OnSaveInstanceState(outState);
   }

   protected override void OnRestoreInstanceState(Bundle savedInstanceState)
   {
      base.OnRestoreInstanceState(savedInstanceState);
      HandleState(savedInstanceState);
```

```
    }

    private void HandleState(Bundle bundle)
    {
        if (bundle != null)
        {
            _activityDisplayCounter = bundle.GetInt(COUNTER_KEY, 0);
        }

        _activityDisplayCounter++;

        _counterTextView.Text = string.Format("Page affichée {0} fois.",
            _activityDisplayCounter);
    }
}
```

PersistantState.axml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/MyButton"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="NAVIGUER VERS LA PAGE PRINCIPALE" />
    <TextView
        android:id="@+id/CounterTextView"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

User Interface (UI)

Vues

Chaque élément graphique dépend des classes View ou ViewGroup qui sont les classes de base permettant d'afficher un élément ou un groupe d'éléments à l'écran et d'interagir avec.

Les ViewGroup ne sont que des containers invisibles à l'écran composés de plusieurs enfants de type ViewGroup ou View formant donc une hiérarchie.

Mise en place

Via AXML

Dans la grande majorité des cas, il est préférable de définir son interface grâce à un fichier permettant une lecture plus facilement lisible grâce à sa structure qui se base sur le XML.

Chaque nœud représente un composant graphique avec l'ensemble de ses propriétés définies via les attributs ; l'ensemble permet de composer une page entière, une vue ou un composant.

Lorsqu'une ressource est chargée, Xamarin parcourt chaque nœud pour créer un objet à la volée que l'on peut modifier par la suite.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent">
   <Button
      android:id="@+id/MyButton"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="@string/Hello" />
</LinearLayout>
```

Il est important de noter qu'Android utilise l'extension de fichier **.xml** lorsque Xamarin.Android utilise **.axml** pour définir son UI. Attention donc à utiliser la bonne extension (.axml) sans quoi les fichiers ne seront pas reconnus correctement.

En utilisant Visual Studio, il est possible de composer son interface à partir d'un designer spécifique permettant de visualiser le rendu à l'écran des composants ainsi que de changer leurs propriétés.

Les fichiers sont à placer dans le dossier Resources/Layout et par convention, on lui donnera le nom de l'activité à laquelle il est rattaché. Par exemple, l'activité MainActivity.cs aura pour fichier de ressource Main.axml.

Du côté de l'activité, il suffit d'utiliser la fonction SetContentView()dans la méthode OnCreate().

```
protected override void OnCreate(Bundle bundle)
{
   base.OnCreate(bundle);
   SetContentView(Resource.Layout.Main);
}
```

La classe statique Resource.Layout répertorie automatiquement l'ensemble des fichiers inclus dans le répertoire Resources/Layout de l'application afin d'y accéder plus facilement.

Via C#

Il est tout à fait possible de créer des composants ou sous-composants dynamiquement à partir du code, notamment dans celui du constructeur de celui-ci.

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

   // creation d'un bouton
   Button mySecondaryButton = new Button(){
      Name = "MySecondaryButton",
      Content = "Secondary Button" };

   // ajout du bouton a la grille principale
   this.MainGrid.Children.Add(mySecondaryButton);
}
```

Système de mise en page

Présentation

Android dispose de trois types de mise en page principale qui se base sur des principes fondamentalement différents. Il est souvent nécessaire d'imbriquer plusieurs d'entre eux afin d'obtenir le résultat souhaité. Chacun de ceux-ci est un ViewGroup à qui il est possible d'ajouter plusieurs enfants à chaque fois.

LinearLayout

Le premier type de mise en page permet d'organiser une liste d'éléments de façon horizontale ou verticale.

Les composants enfants peuvent être de hauteurs et de largeurs différentes, la taille finale du parent, si elle n'est pas définie par une valeur fixe, s'adaptera automatiquement au contenu. Si le contenu est trop grand par rapport à la zone d'affichage, une barre de défilement (scrollbar) s'affichera afin d'accéder à l'ensemble des éléments non visibles.

Les attributs **layout_width** et **layout_height** permettent de définir la largeur et la hauteur de chacun des composants.

Ceux-ci peuvent prendre une valeur fixe : android:layout_height="1dp".

Il est aussi possible de donner deux autres modes permettant d'adapter automatiquement la taille du composant :

- match_parent : la taille du composant s'adaptera en fonction de celle de son parent.
- wrap_content : la taille du composant s'adaptera en fonction de son contenu.

Il est aussi possible de donner un poids d'affichage à chaque élément afin que la place qui lui est attribuée soit plus grande ou plus petite. Ceci se fait à l'aide de l'attribut **layout_weight**.

Par défaut, la valeur de cet attribut est à zéro. Si dans un même layout, on définit l'attribut pour un premier élément à 2 et pour un deuxième élément à 1, le premier élément prendra deux fois plus de place que le deuxième.

A noter que pour utiliser facilement l'attribut layout_weight, il est conseillé de définir les attributs layout_width et/ou layout_height à « 0dp ».


```
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:padding="5dp">
```

```
<!-- titre -->
<TextView
   android:id="@+id/textView1"
   android:text="Conditions d'utilisation"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:textSize="44dip" />

<!-- separateur -->
<LinearLayout
   android:id="@+id/linearLayout2"
   android:orientation="horizontal"
   android:minWidth="25px"
   android:minHeight="25px"
   android:layout_width="match_parent"
   android:layout_height="1dp"
   android:background="#FF3333FF"
   android:layout_marginLeft="10dp"
   android:layout_marginRight="10dp"
   android:layout_marginBottom="5dp"
   android:layout_marginTop="15dp" />

<!-- contenu -->
<TextView
   android:id="@+id/textView3"
   android:text="Lorem ipsum dolor bla bla bla."
   android:layout_width="match_parent"
   android:layout_height="wrap_content" />

<!-- separateur -->
<LinearLayout
   android:id="@+id/linearLayout2"
   android:orientation="horizontal"
   android:minWidth="25px"
   android:minHeight="25px"
   android:layout_width="match_parent"
   android:layout_height="1dp"
   android:background="#FF3333FF"
   android:layout_marginLeft="10dp"
   android:layout_marginRight="10dp"
   android:layout_marginBottom="5dp"
   android:layout_marginTop="15dp" />

<!-- formulaire -->
<LinearLayout
   android:id="@+id/linearLayout1"
   android:orientation="horizontal"
   android:layout_width="match_parent"
   android:layout_height="wrap_content">

   <!-- Label -->
   <TextView
      android:id="@+id/textView2"
      android:text="Acceptez-vous les conditions d'utilisation ?"
      android:layout_width="0dp"
      android:layout_height="match_parent"
      android:layout_weight="2"
      android:gravity="center_vertical" />
   <!-- CheckBox -->
   <CheckBox
      android:id="@+id/checkBox1"
      android:text="J'accepte"
      android:layout_width="0px"
      android:layout_height="match_parent"
      android:layout_marginRight="10dp"
      android:layout_weight="1" />
</LinearLayout>
```

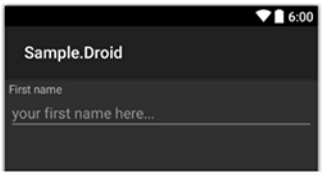
```
    <!-- Button -->
    <Button
       android:id="@+id/MyButton"
       android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="Valider" />
</LinearLayout>
```

RelativeLayout

Le second type de mise en page permet d'organiser les éléments les uns en fonction des autres, donc de façon relative. Il est ainsi possible de spécifier qu'un champ de texte soit au-dessus d'un champ de saisie ou qu'une image soit centrée dans son composant parent.

Positionnement entre éléments enfants

- **layout_above** permet d'aligner le bord inférieur d'un élément avec le bord supérieur d'un autre élément.
- **layout_below** permet d'aligner le bord supérieur d'un élément avec le bord inférieur d'un autre élément.
- **layout_endOf** permet d'aligner l'extrémité inférieure d'un élément avec l'extrémité supérieure d'un autre élément.
- **layout_startOf** permet d'aligner l'extrémité supérieure d'un élément avec l'extrémité inférieure d'un autre élément.
- **layout_toLeftOf** permet d'aligner le bord droit d'un élément avec le bord gauche d'un autre élément.
- **layout_toRightOf** permet d'aligner le bord gauche d'un élément avec le bord droit d'un autre élément.

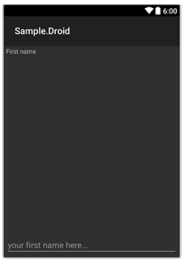



```
<RelativeLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:layout_margin="5dp">
   <TextView
      android:id="@+id/label"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:text="First name" />
   <EditText
      android:id="@+id/result"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:layout_below="@id/label"
      android:hint="your first name here..." />
</RelativeLayout>
```

Positionnement par rapport au parent

- **layout_alignParentBottom** permet d'aligner le bord inférieur d'un élément avec le bord inférieur de son parent.
- **layout_alignParentEnd** permet d'aligner l'extrémité inférieure d'un élément avec l'extrémité inférieure de son parent.
- **layout_alignParentTop** permet d'aligner le bord supérieur d'un élément avec le bord supérieur de son parent.
- **layout_alignParentStart** permet d'aligner l'extrémité supérieure d'un élément avec l'extrémité supérieure de son parent.

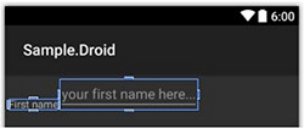
- **layout_alignParentLeft** permet d'aligner le bord gauche d'un élément avec le bord gauche de son parent.
- **layout_alignParentRight** permet d'aligner le bord droit d'un élément avec le bord droit de son parent.




```
<RelativeLayout
   … >
   <TextView
      android:id="@+id/label"
      … />
   <EditText
      …
      android:layout_alignParentBottom="true"
      android:hint="your first name here..." />
</RelativeLayout>
```

Alignement entre éléments enfants

- **layout_alignBottom** permet d'aligner le bord inférieur d'un enfant avec le bord inférieur d'un autre enfant.
- **layout_alignEnd** permet d'aligner l'extrémité inférieure d'un enfant avec l'extrémité inférieure d'un autre enfant.
- **layout_alignTop** permet d'aligner le bord inférieur d'un enfant avec le bord inférieur d'un autre enfant.
- **layout_alignStart** permet d'aligner l'extrémité supérieure d'un enfant avec l'extrémité supérieure d'un autre enfant.
- **layout_alignLeft** permet d'aligner le bord gauche d'un enfant avec le bord gauche d'un autre enfant.
- **layout_alignRight** permet d'aligner le bord droit d'un enfant avec le bord droit d'un autre enfant.




```
<RelativeLayout
   … >
   <TextView
      android:id="@+id/label"
      …
      android:layout_alignBottom="@id/result" />
     <EditText
        android:id="@+id/result"
        android:layout_toRightOf="@id/label" />
</RelativeLayout>
```

Centrage d'un élément

- **layout_centerInParent** permet de centrer verticalement et horizontalement l'élément à l'intérieur de son parent.
- **layout_centerVertical** permet de centrer verticalement l'élément dans son parent.
- **layout_centerHorizontal** permet de centrer horizontalement l'élément dans son parent.


```
<RelativeLayout
   … >
   <TextView
      android:id="@+id/label"
      … />
   <EditText
      android:id="@+id/result"
      …
      android:hint="your first name here..."
      android:layout_centerInParent="true" />
</RelativeLayout>
```

TableLayout

Lorsque l'on souhaite afficher des éléments sous la forme d'un tableau, avec lignes et colonnes, il faut utiliser la classe TableLayout. Elle se compose d'une liste de lignes (TableRow) pouvant avoir une ou plusieurs cellules associées.

L'ordre des cellules dépend, par défaut, de l'ordre dans lequel elles apparaissent dans le fichier .AXML ou dans l'ordre dans lequel elles sont ajoutées dans le code. La première colonne correspond à l'index 0, la deuxième à l'index 1 et ainsi de suite. Pour spécifier la colonne dans laquelle une cellule doit s'afficher, il faut utiliser la propriété **layout_column**.

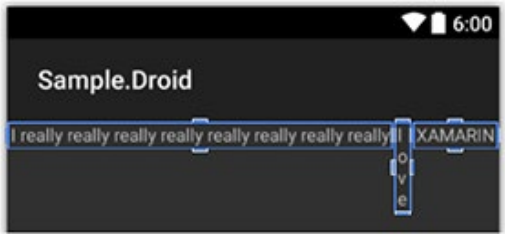
Concernant l'affichage, les bordures du tableau (lignes et colonnes) ne sont pas affichées, il vous faut donc gérer cela vous-même si vous le souhaitez.

Il n'est pas possible de définir une largeur spécifique avec la propriété layout_width pour un enfant d'un composant TableLayout, celle-ci est toujours définie sur fill_parent.

Les colonnes

La manipulation de l'affichage et de la largeur des colonnes se fait à partir de la classe TableLayout et des propriétés :

- **stretchColumns** permet de spécifier les colonnes pouvant s'étirer pour prendre toute la place disponible.
- **shrinkColumns** permet de spécifier les colonnes qui peuvent se rétracter pour laisser de la place aux autres colonnes du tableau.
- **collapsedColumns** permet de spécifier les colonnes à ne pas afficher.



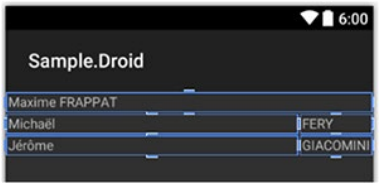

```
<TableLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
```

```
   android:stretchColumns="0"      <!-- colonne étirable -->
   android:collapseColumns="1"     <!-- colonne masquée -->
   android:shrinkColumns="2"       <!-- colonne peut être réduite -->
>
   <TableRow>
      <TextView
         android:text="I really really really really really really really really"
         android:padding="3dip" />
      <TextView
         android:text="hate"
         android:padding="3dip" />
      <TextView
         android:text="love"
         android:padding="3dip" />
      <TextView
         android:text="XAMARIN"
         android:padding="3dip" />
   </TableRow>
</TableLayout>
```

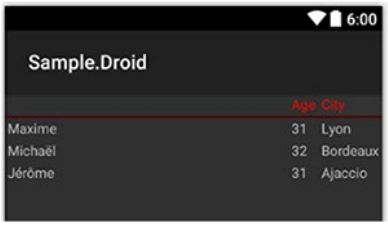
Les lignes

Les lignes (TableRow) possèdent deux propriétés principales permettant de modifier l'affichage des cellules :

- **layout_span** permet de définir le nombre de colonnes à regrouper.
- **layout_column** permet de définir la colonne où doit s'afficher cette cellule.




```
<TableLayout
   … >
   <TableRow>
      <TextView
         android:text="Maxime FRAPPAT"
         android:layout_span="2" /> <!--groupe les colonnes 1 et 2 de la 1e ligne-->
   </TableRow>
   <TableRow>
      <TextView
         android:text="Michaël" />
      <TextView
         android:text="FERY" />
   </TableRow>
   <TableRow>
      <TextView
         android:text="Jérôme" />
      <TextView
         android:text="GIACOMINI" />
   </TableRow>
</TableLayout>
```



Dans l'exemple suivant, on spécifie la valeur de la colonne à 1 pour la cellule affichant le texte "Age". Automatiquement, la prochaine cellule "City" voit sa colonne modifiée elle aussi pour avoir la valeur « 2 ».

Les lignes suivantes commencent bien dès la première colonne, car rien n'est spécifié pour celles-ci, la propriété layout_column prend donc la valeur 0.

```
<TableLayout
   …
>
   <TableRow>
      <TextView
         android:layout_column="1" <!-- s'affichera à la colonne 1 (2e) -->
         android:text="Age"  />
      <TextView
         <!-- layout_column pas spécifié prend la valeur 2 -->
         android:text="City" />
   </TableRow>
   <View … />
   <TableRow>
      <TextView
         android:text="Maxime" />
      <TextView
         android:text="31" />
      <TextView
         android:text="Lyon" />
   </TableRow>
   <TableRow>
      <TextView
         android:text="Michaël" />
      <TextView
         android:text="32" />
      <TextView
         android:text="Bordeaux" />
   </TableRow>
   <TableRow>
      <TextView
         android:text="Jérôme" />
      <TextView
         android:text="31" />
      <TextView
         android:text="Ajaccio" />
   </TableRow>
</TableLayout>
```

Les formulaires

Les libellés

Pour afficher du texte dans une application Android, il faut s'appuyer sur la classe **TextView**.

Elle possède plusieurs propriétés dont Text qui permet de spécifier le texte que l'on souhaite afficher ou textColor pui permet de modifier la couleur.

```
<TextView
   android:id="@+id/myTextView"
   android:text="Merci Xamarin !"              <!-- specification du texte -->
   android:textColor="@android:color/white" /> <!-- modification de la couleur -->
```

OU

```
// recuperation du layout principal
_rootLayout = FindViewById<LinearLayout>(Resource.Id.rootLayout);
// creation du TextView
_myTextView = new TextView(this);
_myTextView.Text = "Une autre TextView";       // specification du texte
_myTextView.SetTextColor(Color.OrangeRed);     // modification de la couleur
// ajout au layout principal
_rootLayout.AddView(_textView);
```

Pour modifier la taille du texte, il faut utiliser la propriété textSize. Par défaut, l'unité utilisée est le scaled pixel. Cette unité se base sur la densité de pixel ainsi que sur les préférences utilisateurs en termes de taille de police afin d'ajuster la taille réelle à l'écran.

```
android:textSize="16sp"
```

Par défaut, la classe TextView n'affiche qu'une seule ligne. Afin de pouvoir afficher un texte sur plusieurs lignes, il faut passer la propriété singleLine à false.

```
android:singleLine="false"
```

Saisir un texte, un nombre

Pour afficher du texte dans une application Android, il faut s'appuyer sur la classe **EditText**.

Elle permet de gérer plusieurs formats de texte de façon native (nombre, adresse mail, numéro de téléphone...) à l'aide de la propriété inputType. On peut aussi lui dire de mettre en majuscule chaque caractère (textCapCharacters), la première lettre de chaque mot (textCapWords) ou encore la première lettre de chaque phrase (textCapSentences).

Le clavier virtuel sera affecté par ce champ pour s'adapter au type de texte qu'il attend.

```
<EditText
   android:id="@+id/myEditText"
   android:text="123456789"
   android:inputType="number" />
```

La propriété Text permet d'obtenir ou de définir le texte de la zone de texte modifiable.

A noter que l'on peut surveiller le changement de valeur de la zone de texte via l'évènement TextChanged.

```
_editText = FindViewById<EditText>(Resource.Id.myEditText);
_editText.TextChanged += OnTextChanged;
...
private void OnTextChanged(object sender,
Android.Text.TextChangedEventArgs e)
{
    Console.WriteLine("Le texte a changé : {0}", e.Text);
}
```

Les boutons

Pour afficher du texte dans une application Android, il faut s'appuyer sur la classe **Button**.

Cela permet d'exécuter un code personnalisé en réponse à une interaction de l'utilisateur.

```
<Button
   android:id="@+id/myButton"
   android:text="Mon Bouton"       <!-- affiche du texte dans le bouton -->
   android:textColor="#ff33bbff"   <!-- couleur du texte -->
   android:background="#ffeeee"    <!-- couleur du fond -->
   android:enabled="false" />      <!-- rend inactif le bouton -->
```

Pour être notifié lorsque l'utilisateur clique sur un bouton, il faut s'abonner à l'évènement.

```
_button = FindViewById<Button>(Resource.Id.myButton);
_button.Click += OnClick;
...
private void OnClick(object sender, EventArgs e)
{
    Console.WriteLine("Le bouton a été déclenché");
}
```

Les cases à cocher

Sur Android, il existe deux choix de case à cocher : la **CheckBox** ou le **ToggleButton**.

En général, on utilise un ToggleButton si l'on souhaite effectuer un changement immédiat, comme activer ou désactiver une préférence utilisateur, sans une validation ultérieure.


```
<CheckBox
   android:id="@+id/myCheckbox"
   android:checked=”false”                          <!-- checkBox désactivée  -->
   android:text="Choisir un état (CheckBox):" /> <!-- texte de la checkBox  -->
```


```
<ToggleButton
   android:id="@+id/myToggleButton"
   android:checked=”false”                        <!-- bouton désactivé  -->
   android:textOff="Désactivé (ToggleButton)"   <!-- texte du bouton désactif -->
   android:textOn="Activé (ToggleButton)" />    <!-- texte du bouton actif -->
```

Le changement d'état, donc le passage d'un état activé ou désactivé et inversement, s'effectue pour les deux contrôles à partir de la propriété checked.

Pour être notifié lorsque l'état du contrôle change, il faut s'abonner à l'évènement CheckedChange.

```
_checkBox = FindViewById<CheckBox>(Resource.Id.myCheckbox);
_checkBox.CheckedChange += OnCheckedChange;

...
private void OnCheckedChange(object sender,
CompoundButton.CheckedChangeEventArgs e)
{
    Console.WriteLine("Je suis {0}", e.IsChecked ? "on" : "off");
}
```

Les curseurs

Le curseur (ou réglette) est un contrôle visuel qui permet de faire saisir à un utilisateur une valeur comprise entre deux bornes.

Pour afficher un curseur dans une application Android, il faut s'appuyer sur la classe **SeekBar**.

```
<SeekBar
    android:id="@+id/mySeekBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content" />
```

Pour être notifié lorsqu'il y a un changement de la valeur du curseur, il faut s'abonner à l'évènement ProgressChanged.

```
_seekBar = FindViewById<SeekBar>(Resource.Id.mySeekBar);
_seekBar.ProgressChanged += OnProgressChanged;
...
private void OnProgressChanged(object sender, SeekBar.ProgressChangedEventArgs e)
{
   Console.WriteLine("La valeur du curseur est de : {0}", e.Progress);
}
```

Hélas, on ne peut pas spécifier de borne minimale, celle-ci est toujours égale à 0. Le pas (ou step) est lui toujours égal à 1. Ceci contraint le développeur à faire des calculs dans le code.

Afficher un changement

Lorsqu'une application fait un traitement, par exemple enregistrer une donnée, il faut indiquer à l'utilisateur que l'application travaille. Sans quoi il peut penser que l'application ne fonctionne plus.

Pour afficher un changement dans une application Android, on peut s'appuyer sur deux classes, **ProgressBar** et **ProgressDialog**.

ProgressBar permet d'afficher une barre de chargement qui peut être ou non quantifiable.

```
<ProgressBar
   android:id="@+id/myProgressBar"
   android:indeterminate="false" <!-- spécifie si le chargement est quantifiable -->
   android:progress="25" /> <!-- niveau de chargement (si indeterminate=false)-->
```

ProgressDialog permet d'afficher une fenêtre au-dessus de la fenêtre actuelle, indiquant à l'utilisateur un chargement. Elle ne peut pas être instanciée directement dans la vue, il faut le faire de manière dynamique.

```
// ouverture d'une fenêtre de dialogue
var myProgressDialog = ProgressDialog.Show(this, "Chargement",
"Apprentissage de Xamarin en cours", true);
…
// fermeture de la fenêtre de dialogue
myProgressDialog.Dismiss();
```

Afficher des alertes

Les alertes permettent de transmettre des informations importantes à l'utilisateur. Elles se composent d'un titre, d'un message, et d'un ou plusieurs boutons permettant à l'utilisateur de faire un choix.

Pour afficher une alerte dans une application Android, on peut s'appuyer sur deux classes, **AlertDialog** (types d'alertes prédéfinies) et **DialogFragment** (alertes personnalisées).