

## 0 Table des matières

<b>0</b>	<b>Table des matières</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Résumé du cahier des charges	4
1.1.1	Fonctionnalités de base	4
1.1.2	A réaliser au minimum	4
1.2	Description du projet	4
<b>2</b>	<b>Mise en œuvre du projet</b>	<b>4</b>
<b>3</b>	<b>Maquettes</b>	<b>5</b>
3.1	Menu	5
3.2	Tâches	6
3.2.1	To Do List	6
3.2.2	Ma journée	6
3.2.3	Ajout d'une tâche	6
3.2.4	Affichage d'une tâche	6
3.3	Catégories	7
3.3.1	Gérer les catégories	7
3.3.2	Ajouter une catégorie	7
<b>4</b>	<b>Environnement du projet</b>	<b>8</b>
4.1	Installation	8
4.1.1	VS 2019	8
4.1.2	VS 2022	9
4.2	Problèmes d'installation	9
<b>5</b>	<b>Création d'un projet</b>	<b>9</b>
<b>6</b>	<b>Explication de l'environnement</b>	<b>10</b>
6.1	Fichiers source	10
6.2	Fichiers de ressources	11
6.3	Modélisation	11
6.4	Code d'une activité	12
<b>7</b>	<b>Emulateur</b>	<b>12</b>
7.1	Création d'un émulateur	12
7.2	Utilité d'un émulateur	13
<b>8</b>	<b>Activités</b>	<b>14</b>
8.1	Layouts	14
8.1.1	LinearLayout	14
8.1.2	RelativeLayout	14
8.1.3	GridLayout	16

<b>8.2</b>	<b>Création de composants</b>	<b>17</b>
8.2.1	Dans le fichier XML	17
8.2.1.1	Balises	17
8.2.1.2	Boite à outils	17
8.2.1.3	Attributs	18
8.2.2	Dynamiquement	18
8.2.2.1	Attributs dans le corps du bouton	19
8.2.2.2	Attributs après la déclaration de la variable	19
8.2.2.3	Affichage de l'élément dans un layout	19
8.2.3	Lien entre un composant et le code	20
8.2.3.1	Créer des évènements	20
<b>8.3</b>	<b>Lier le fichier CS avec le fichier XML</b>	<b>21</b>
<b>8.4</b>	<b>Lien entre 2 activités</b>	<b>21</b>
8.4.1	Créer une seconde activité	21
8.4.2	Appeler une autre activité	22
8.4.2.1	Navigation vers une autre activité	22
8.4.2.2	Passage de données entre 2 activités	23
8.4.2.3	Récupération des données	23
<b>9</b>	<b>Cycle de vie d'une application</b>	<b>24</b>
<b>10</b>	<b>ListView</b>	<b>25</b>
10.1	ListView classique	25
10.2	ListView personnalisée	25
10.2.1	Model	26
10.2.2	Layout	27
10.2.3	Adaptateurs	27
10.2.4	Views	29
<b>11</b>	<b>Persistence des données</b>	<b>30</b>
11.1	Mise en place de la DB	30
11.2	Création de la DB	31
11.2.1	Créer une table	31
11.2.2	Créer des champs	31
11.3	Manipulation des données	32
<b>12</b>	<b>Mise en place d'un capteur (Sensor)</b>	<b>33</b>
12.1	Détecter la disponibilité des capteurs	33
12.2	Utiliser le capteur	34

# Application mobile Android - Xamarin



<b>Auteur</b>	Damien Loup
<b>Lieu</b>	ETML - Vennes
<b>Classe</b>	Cin-Cid3B
<b>Date</b>	12.10.2022
<b>Maître de projet</b>	Ferrari Roberto

# 1 Introduction

Ce document est fait pour des informaticiens et plus précisément des personnes ayant des notions en C#, celui-ci fait référence à toutes les étapes mises en œuvre pour créer un projet d'application mobile en C# avec **Microsoft Visual Studio Xamarin**.

Au final, l'application réalisée devra être exploitable et livrable, ainsi que la réalisation de la documentation du projet

## 1.1 Résumé du cahier des charges

### 1.1.1 Fonctionnalités de base

- Des tâches
  - Chaque tâche doit contenir :
    - Un titre
    - Un mini descriptif
    - Une date d'échéance
  - La possibilité à l'utilisateur d'en créer
  - La possibilité de sélectionner plusieurs tâches et de les ajouter à ma journée
- Des activités
  - Une activité comprenant toutes les tâches créées
    - Tri avec les catégories
  - Une activité comprenant toutes les tâche d'aujourd'hui
    - Tri avec les catégories
  - Une activité permettant d'ajouter une tâche
- Utiliser un sensor
  - Récupérer les données et les utiliser pour exécuter quelque chose
- Des catégories
  - Chaque catégorie doit au moins contenir un nom
  - Il doit être possible d'ajouter une tâche dans une catégorie

### 1.1.2 A réaliser au minimum

- Un lien entre 2 activités différentes
- Utiliser les ressources
- Utiliser des méthodes événementielles
- Faire un layout dynamique
  - Générer du code xml dans le code c#
- Stocker des informations en base de données
- Utiliser un sensor du téléphone

## 1.2 Description du projet

Réaliser une application mobile en **C#** avec Xamarin gérant des tâches. Des tâches peuvent être ajoutées ou retirées. Chaque tâche peut être ajouter à la liste de tâche à faire aujourd'hui et peuvent être retirées manuellement ou en secouant le téléphone.

# 2 Mise en œuvre du projet

Pour créer une application de base, il faut réaliser certaines étapes obligatoires pour son bon fonctionnement, qui est ici **un gestionnaire de tâches**.

Les fonctionnalités ont été citées dans le cahier des charges ci-dessus.

Dans ce document vont être énuméré chaque étape et chaque tâches important au bon déroulement selon les suivantes :

- Création des maquettes de l'application
- L'environnement de projet
- Création d'un projet
- Création d'une activité
- Utilisation d'un fichier de ressources
- Lien entre deux activités
- Les méthodes événementielles
- Les différents types de layout
- Les layouts dynamiques
- Persistance de données
- L'utilisation d'un sensor

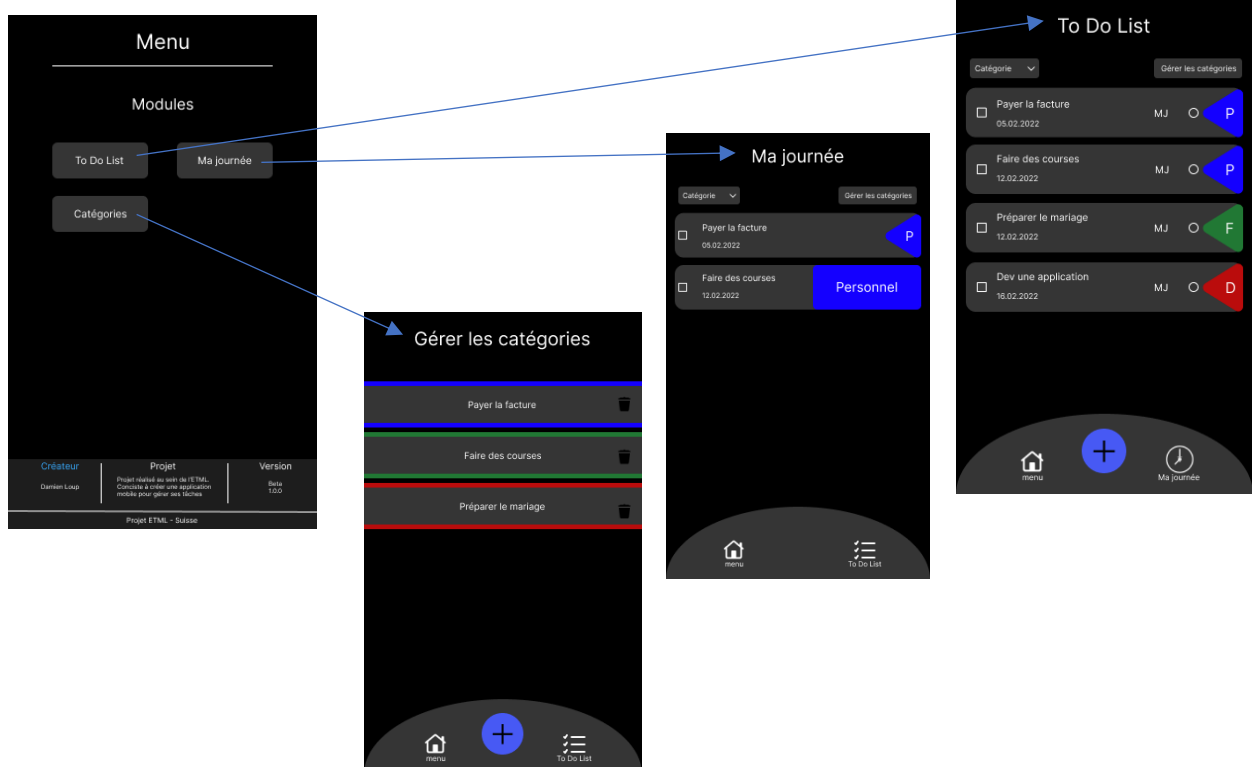
### 3 Maquettes

Tout d'abord, avant de commencer le projet, il faut y créer les maquettes de l'application.

Les maquettes ont été faites sur Figma à l'aide des outils à disposition sur leur application/site

#### 3.1 Menu

Le menu est la première activité à s'ouvrir au lancement de l'application, elle permet de se déplacer dans les différentes autres activités comme la liste de tâches, ma journée et les catégories.



## 3.2 Tâches

### 3.2.1 To Do List

La To Do List (Liste de tâches) permet d'afficher toutes les tâches créées dans l'application et entrées en base de données avec une liste scrollable. Il est possible de filtrer les tâches par catégorie ou de les gérer.

Il sera affiché le nom de la tâche, sa date d'échéance, un checkbox qui définit si la tâche est terminée ou non et un radio bouton qui définit si la tâche doit se trouver dans les tâches du jour.

Il est possible aussi de créer des tâches avec le bouton « + » en bas.

Il est possible de supprimer la tâche en appuyant longtemps sur elle et d'appuyer sur le bouton « - »

### 3.2.2 Ma journée

L'activité « Ma journée » permet de voir toutes les tâches ajoutées à « aujourd'hui » c'est-à-dire les tâches que l'utilisateur veut pouvoir faire le jour-même avec une liste scrollable ! Comme pour la To Do List, Il est possible de filtrer les tâches par catégorie ou de les gérer.

Il sera affiché le nom de la tâche, sa date d'échéance, un checkbox qui définit si la tâche est terminée ou non et un radio bouton qui définit si la tâche doit se trouver dans les tâches du jour.

Il est possible de supprimer la tâche pour le jour en appuyant longtemps sur elle et d'appuyer sur le bouton « - »

#### **C'est sur cette activité que sera utilisé le sensor :**

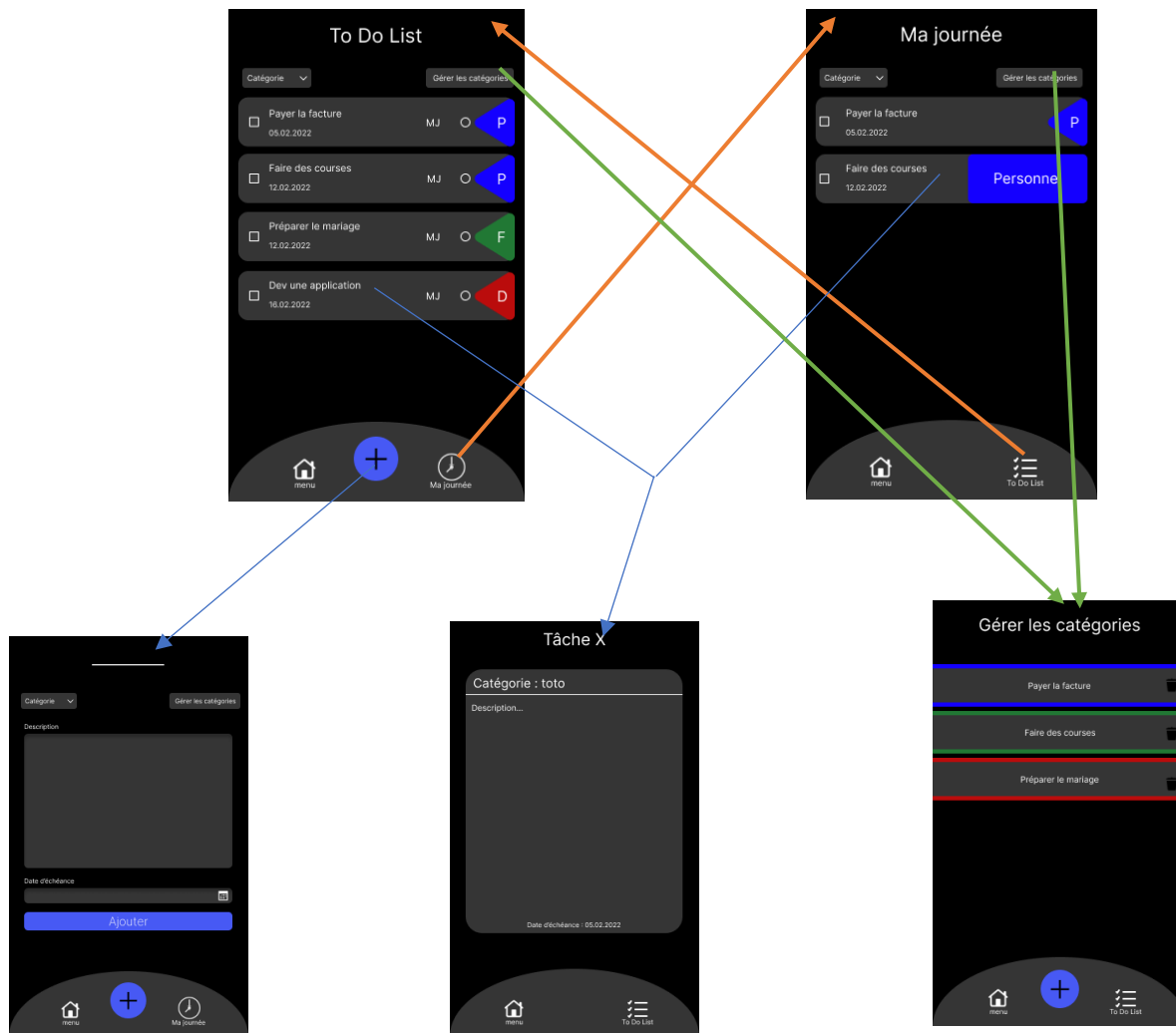
Le but est de faire que lorsque l'on secoue le téléphone, toutes les tâches de la journée actuelle se suppriment (Uniquement dans « aujourd'hui »).

### 3.2.3 Ajout d'une tâche

L'activité d'ajout d'une tâche permet d'ajouter une tâche avec un titre, une description, une catégorie et une date.

### 3.2.4 Affichage d'une tâche

L'activité d'affichage d'une tâche permet d'afficher son titre, sa description, sa catégorie et sa date d'échéance.



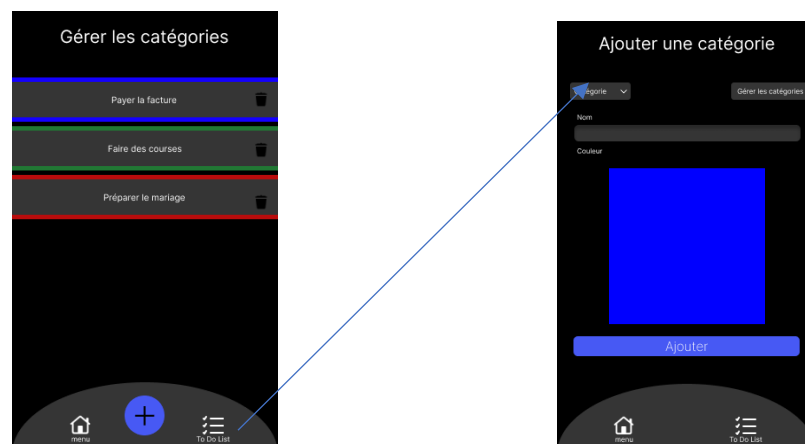
### 3.3 Catégories

#### 3.3.1 Gérer les catégories

Cette activité est le sous-menu des catégories, ce menu permet de voir toutes les catégories créées dans l'application, il permet aussi d'en créer

#### 3.3.2 Ajouter une catégorie

Cette activité permet de créer une catégorie avec un nom et une couleur donnée ""

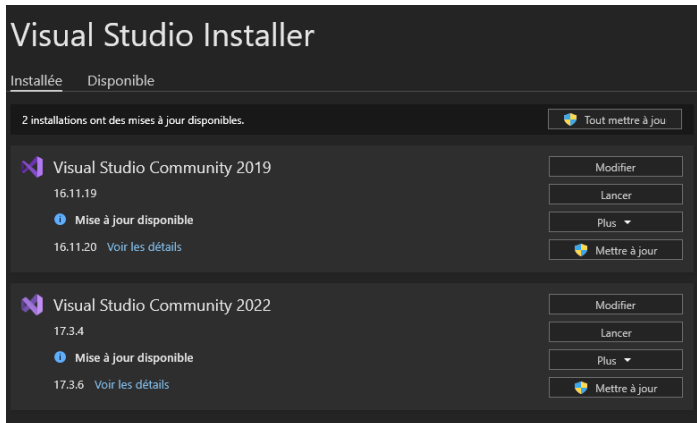


## 4 Environnement du projet

L'environnement de projet utilisé est **Xamarin**, pour l'utiliser, il faut installer l'application (IDE) Visual Studio.

### 4.1 Installation

En premier lieu il faut lancer « **Visual Studio Installer** »

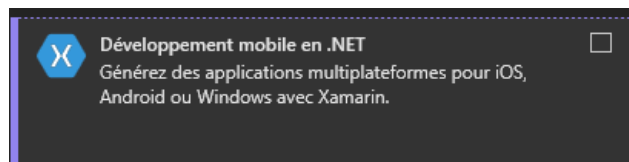


Une fois sur cette page, il faudra choisir (S'il y a plusieurs versions de Visual Studio) la version de Visual studio préférée.

Il faudra venir modifier celle-ci avec le bouton « **Modifier** » et trouver en fonction de la version :

#### 4.1.1 VS 2019

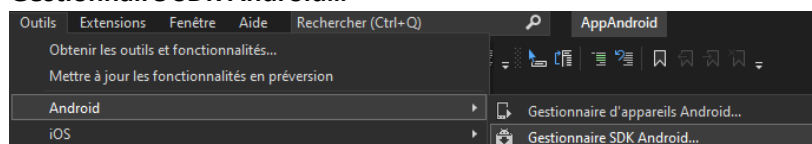
« Développement mobile en .NET » et cocher la checkbox en haut à droite et installer la modification.



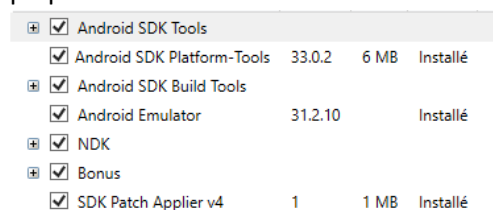
Et il faudra installer les packages SDK et un émulateur Android dont nous allons reparler plus tard.

Afin d'installer les packages SDK, il faut se rendre dans **outils -> Android ->**

**Gestionnaire SDK Android...**



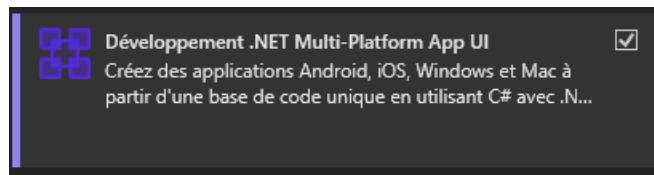
Une fois cette fenêtre ouverte, il ne reste plus qu'à installer dans les outils, les paquets donnés



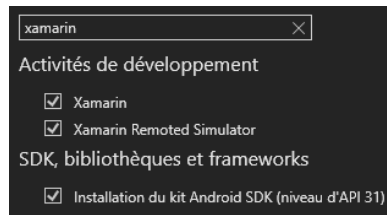


#### 4.1.2 VS 2022

« Développement .NET Multi-Platform App UI » et cocher la checkbox en haut à droite et installer la modification.



Et ensuite aller dans l'onglet « **Composants individuels** » et rechercher « **Xamarin** » et cocher ces 3 checkboxes.



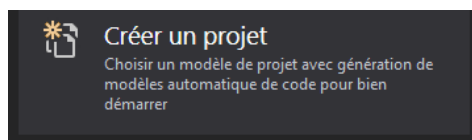
#### 4.2 Problèmes d'installation

Etant donné que Xamarin va bientôt être obsolète et remplacé par **Maui**, il peut y avoir des problèmes d'installation ou ne pas trouver Xamarin dans les dernières versions de Visual Studio.

Si cela se passe, il faudra aller chercher dans les versions antérieures aux versions qui ne disposent plus de cet environnement ou de suivre le point précédent en installant différemment l'environnement.

### 5 Création d'un projet

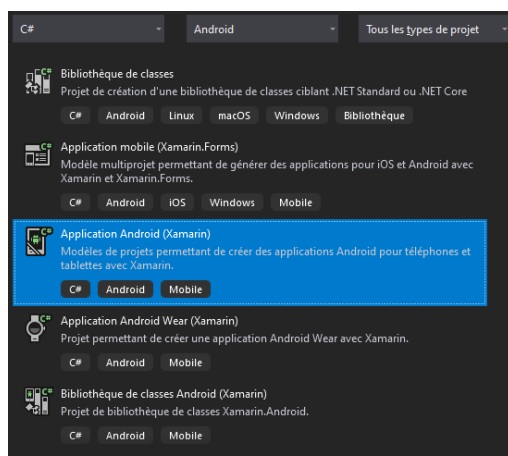
Au démarrage de Visual Studio il faut créer un projet.



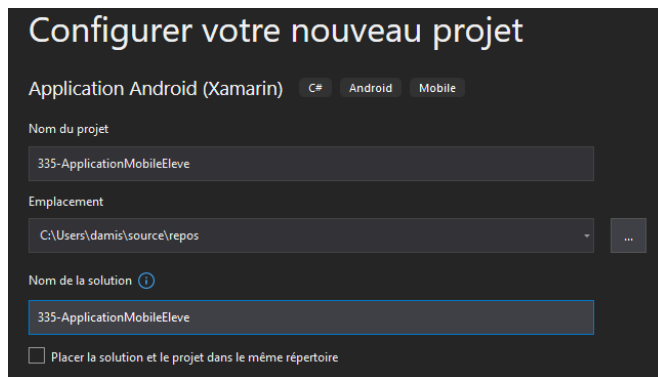
Une fois ce bouton cliqué, il faudra choisir le bon déroulé pour avoir accès au projet.

Il faut choisir les filtres séparément : **C#**, **Android** et **Tous les types de projet**.

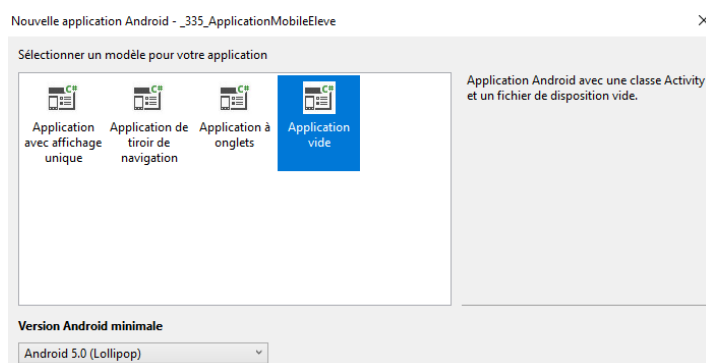
Le type de projet « **Application Android (Xamarin)** » est celui qui nous intéresse pour ce projet.



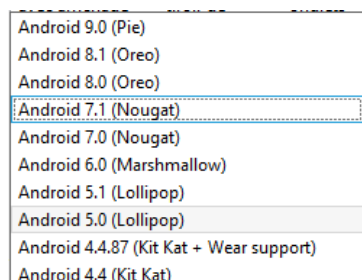
Ensuite il faudra venir donner un titre et choisir l'emplacement stocké du projet



Une fois cela fait et le bouton « **Créer** » appuyé, il faudra choisir le modèle de base du projet. Ici nous allons utiliser une application vide pour la faire de A à Z.



Vient ensuite, le choix de la version de notre application, ici nous allons choisir la ou l'une des plus anciennes versions d'Android, car plus la version est récente, plus il y aura de fonctionnalités à notre disposition, mais il y aura aussi plus de téléphones qui ne pourront pas l'utiliser et la supporter, dû à leur version d'Android.



## 6 Explication de l'environnement

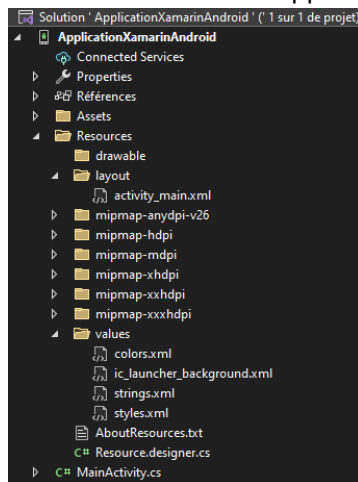
L'environnement ressemble à un projet C# classique, cependant, il y a quelques subtilités qui en font une différence considérable.

### 6.1 Fichiers source

Les fichiers sources s'apparentent à ceci.

- Les interface de l'application, avec les fichiers de modélisation en **XML** se trouvent dans « **layout** »
- Les images ou les éléments à ajouter dans la modélisation de l'application se trouveront dans « **drawable** »

- Les fichiers de ressources permettant de gérer les **styles**, les **couleurs** et des **chaines de caractères** de l'application sont dans « **values** »



## 6.2 Fichiers de ressources

Les fichiers de ressources se trouvant dans « **values** » sont considérés comme des fichiers contenant des constantes accessibles par tout l'environnement.

Ils peuvent permettre de changer la **couleur du thème**, changer des **couleurs** ou en **ajouter** autant que l'on veut.

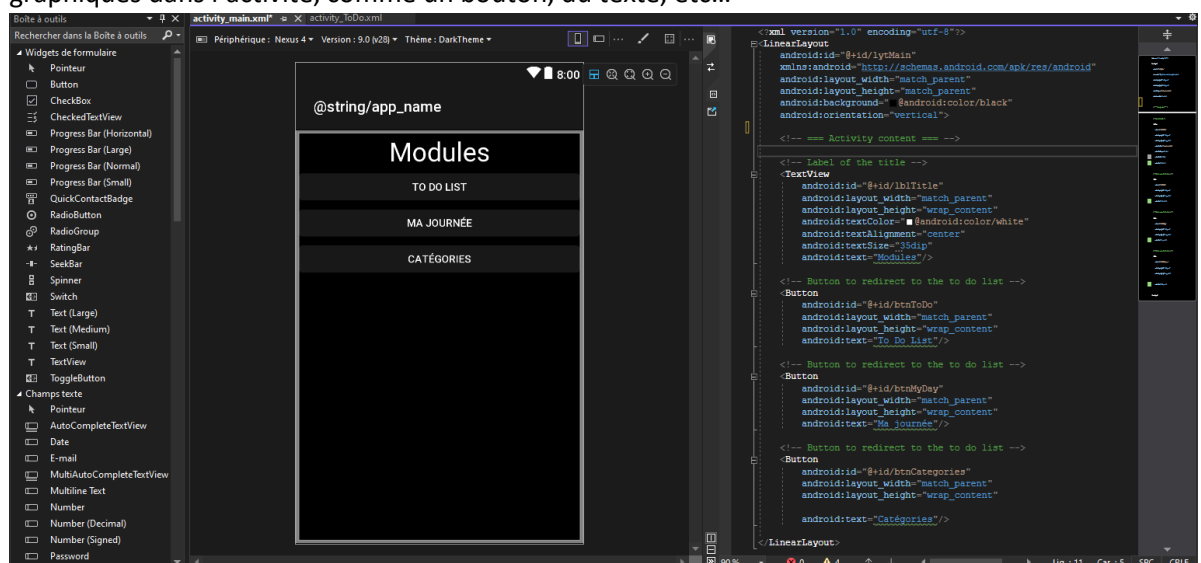
Les fichiers de ressources peuvent aussi être adaptés pour rendre l'application **multilingue** avec un fichier par langue différente. Ils peuvent aussi faire en sorte d'adapter l'application avec des fichiers différents en fonction de la taille de l'écran, cela permet de séparer le code source des éléments de design.

## 6.3 Modélisation

Le fichier de modélisation étant, dans ce cas-là, le fichier « **activity\_main.xml** »

En double cliquant sur le fichier de modélisation, l'interface ci-dessous apparaît, le code **XML** est à droite et la **visualisation de l'interface** de l'application à gauche.

Tout à gauche de l'interface, se trouve la **boîte à outils**, elle permet d'y placer des éléments graphiques dans l'activité, comme un bouton, du texte, etc...



## 6.4 Code d'une activité

Le code de base d'une activité s'apparente à une classe classique.

Quelques éléments à relever sont conçus pour l'environnement de Xamarin :

La ligne **En dessus** de la classe

```
[Activity(Label = "@strings/app_name", Theme = "@style/AppTheme", MainLauncher = true)]
```

permet de définir :

- **Label** -> Le nom à afficher par défaut dans l'activité.
- **Theme** -> Le thème à utiliser qui a été défini dans un des fichiers ressources (**styles.xml**).
- **MainLauncher** -> Défini si l'activité est la principale de l'application.

```
using Android.App;
using Android.OS;
using Android.Runtime;
using AndroidX.AppCompat.App;

namespace ApplicationXamarinAndroid
{
    [Activity(Label = "@string/app_name", Theme = "@style/AppTheme", MainLauncher = true)]
    public class MainActivity : AppCompatActivity
    {
        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);
            Xamarin.Essentials.Platform.Init(this, savedInstanceState);
            // Set our view from the "main" layout resource
            SetContentView(Resource.Layout.activity_main);
        }

        public override void OnRequestPermissionsResult(int requestCode, string[] permissions, [GeneratedEnum] Android.Content.PM.Permission[] grantResults)
        {
            Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions, grantResults);

            base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
        }
    }
}
```

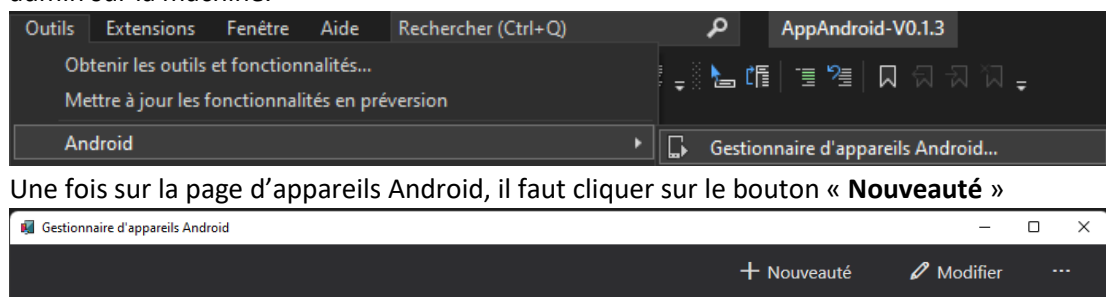
## 7 Emulateur

L'émulateur est un point important pour tester son application. Le but est de simuler un vrai téléphone sur le PC.

Il est possible aussi de tester son application en la déployant sur un élément physique comme un téléphone. Pour ceci, il faut aller dans **paramètres -> À propos du téléphone -> informations sur le logiciel**. Une fois ceci fait, il faut appuyer 7 fois sur **Numéro de version** pour activer le mode développeur pour ensuite se rendre dans **Options de développement** activer le débogage USB et connecter son téléphone en USB à votre PC et lancer le débogage dans visual studio.

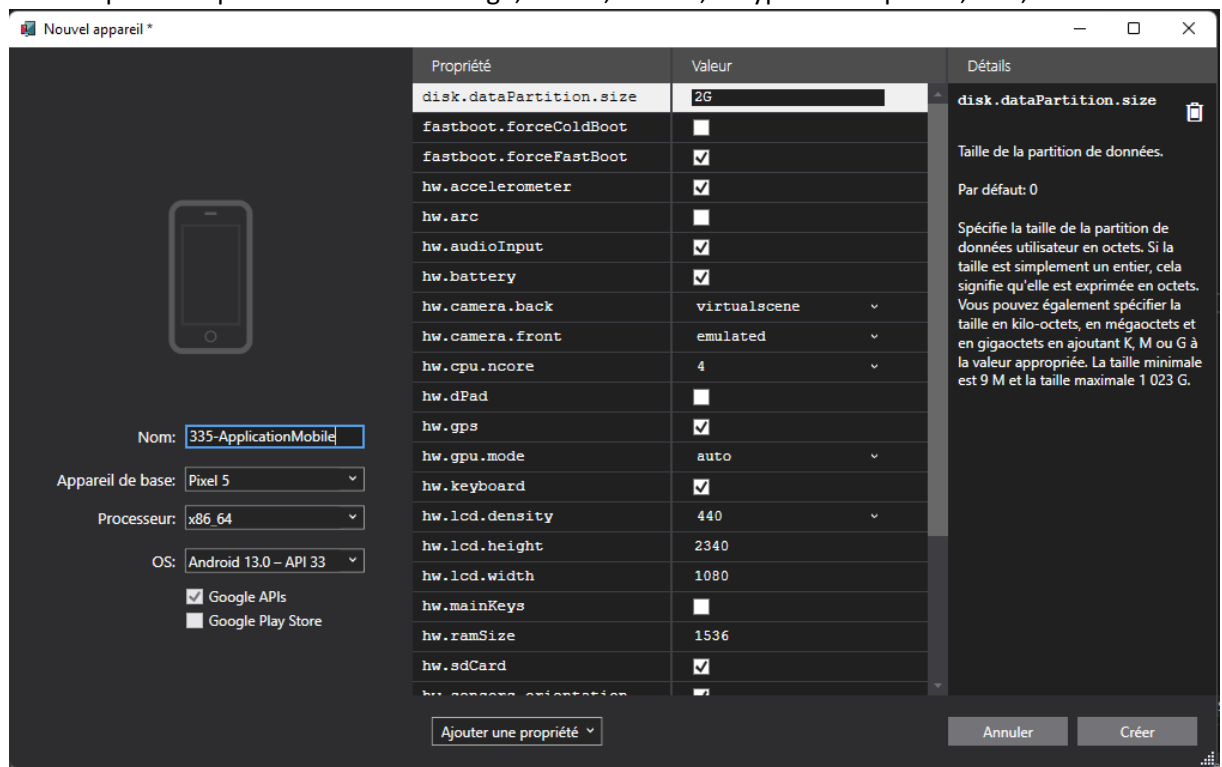
### 7.1 Création d'un émulateur

En premier lieu, il faut aller dans **outils -> android -> Gestionnaire d'appareils Android** et être admin sur la machine.

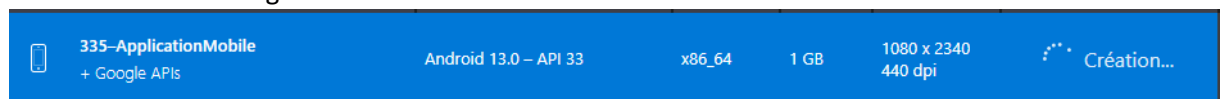


Une fois sur la page d'appareils Android, il faut cliquer sur le bouton « **Nouveauté** »

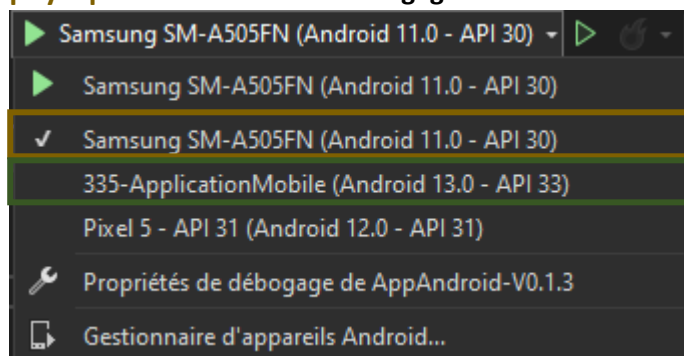
Après ceci, une page de configuration va s'afficher. Elle permet de configurer le téléphone comme par exemple : La taille de stockage, la ram, le nom, le type de téléphone, l'OS, etc...



Lorsque la configuration est effectuée, il faut appuyer sur « **Créer** » et notre émulateur va s'installer dans notre gestionnaire.



Une fois que l'émulateur est installé, il est possible de le démarrer et l'utiliser « **comme** » un vrai téléphone et **déboguer** le code effectué. Il suffit de choisir **l'émulateur** ou le **téléphone physique** dans le menu de **débugage**.





## 7.2 Utilité d'un émulateur

Un émulateur permet de tester l'application et de voir si elle fonctionne correctement sur d'autres types de téléphones sans les avoirs physiquement et de les simuler. Cela permet aussi de tester selon la version de l'OS étant donné que chaque version contient des fonctionnalités en plus ou en moins.

## 8 Activités

Au début du projet, de base une activité est créée. Une activité est composée de 2 fichiers différents :

-  MainActivity.cs
-  activity\_main.xml

Le fichier **MainActivity.cs** contient tout le code **C#** de l'activité. Il permet de créer des objets comme des boutons, etc... et d'en récupérer aussi. Il permet aussi de gérer les méthodes événementielles.

Le fichier Activity\_main.xml contient tout le code **XML** avec ses balises, c'est ce qui permet de créer l'interface de l'application.

### 8.1 Layouts

Un layout est un des composants principaux d'une activité, il est en quelque sorte, un conteneur qui contient une série d'éléments comme des **boutons**, des **textes** et même d'autres **layouts**.

Les layouts ne donnent pas la possibilité de scroller, c'est pourquoi s'il y a trop d'éléments il faudra rajouter un scrollView.

Il en existe plusieurs qui effectuent des tâches différentes, en voici une petite liste.

#### 8.1.1 LinearLayout

Le linear layout est un des layouts les plus utilisés. Il permet d'afficher, comme son nom l'indique, de manière **linéaire** tous les éléments qu'il contient. Il est possible de le paramétrer dans les **attributs** pour changer son **orientation**, afin d'afficher son contenu **verticalement** ou **horizontalement**.



Il est possible aussi de changer la couleur de fond, la taille et d'autre **attributs** dont l'**ID** qui est le plus important, afin de le récupérer dans le code plus tard.

```
<LinearLayout
    android:id="@+id/lytMain"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@android:color/black"
    android:orientation="vertical">
```

#### 8.1.2 RelativeLayout

Le relative layout est aussi un des layouts les plus utilisés et permet d'afficher des éléments aux positions exactes que l'on souhaite.

Les éléments se placent toujours de la même manière dans le code **XML**, mais ne se posent pas automatiquement en dessous de celui d'avant comme le **linear layout**.

Ici pour placer un élément, il faut se baser sur les autres ou l'élément **parent** qui peut être un autre layout.

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Ajouter des catégories"
    android:id="@+id/txtCategoryTitle"/>

<!-- Buttons layout -->
<LinearLayout
    android:id="@+id/lytToDoCategories"
    android:orientation="horizontal"
    android:layout_below="@id/txtCategoryTitle"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <!-- Button select category -->
    <Button
        android:id="@+id/btnCategory_categories"
        android:textColor="@android:color/white"
        android:text="Gérer les catégories"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:layout_marginRight="5dp"
        android:layout_marginLeft="10dp"
        android:textAlignment="center"/>

    <!-- Button manage categories -->
    <Button
        android:id="@+id/btnCategoryAdd"
        android:textColor="@android:color/white"
        android:text="Ajouter une catégorie"
        android:layout_height="match_parent"
        android:layout_marginRight="10dp"
        android:textAlignment="center"
        android:layout_weight="1" />
</LinearLayout>
```

Ici, on place un **TextView** contenant un texte « **Ajouter des catégories** » ce texte étant le premier élément, il est affiché au début.

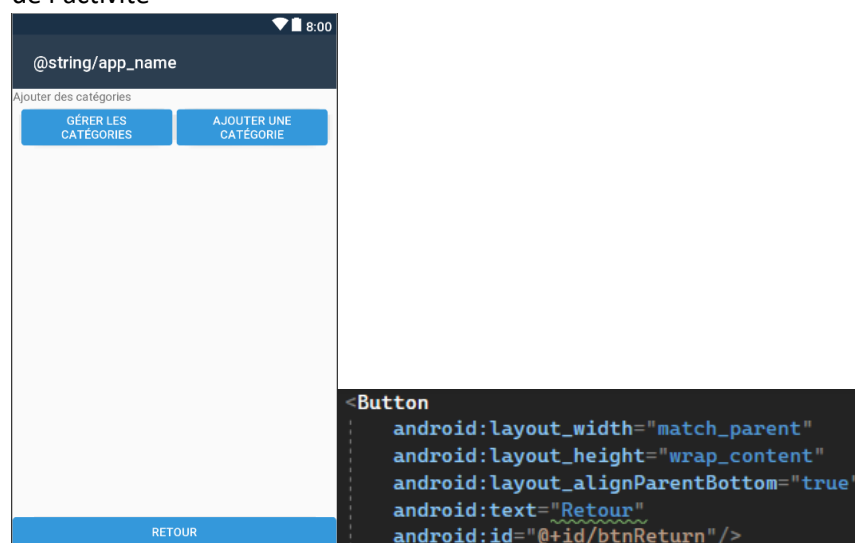
Ensuite, on place un **LinearLayout** en **horizontal** en spécifiant qu'il soit en dessous du texte placé juste avant à l'aide de l'attribut

« **android:layout\_below="@id/txtCategoryTitle"** »

Ensuite, on y place deux boutons avec des attributs qui permettent de gérer le style, ainsi que respectivement un poids de 1 pour qu'ils aient la même taille et se placent correctement sur l'**horizontal** comme défini dans le layout parent, ainsi que des marges.

Il y a aussi un bouton placé après le **LinearLayout** avec l'attribut

« **android:layout\_alignParentBottom="true"** » qui permet de le placer tout en bas de l'activité



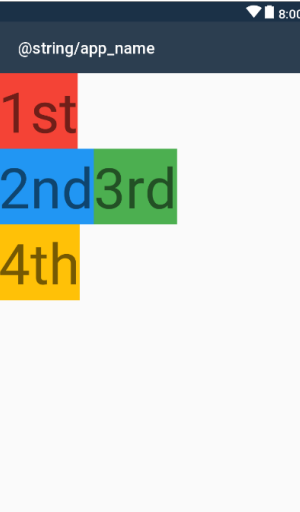
### 8.1.3 GridLayout

Le **GridLayout** permet de gérer son activité à l'aide d'une **grille** et de placer les éléments sur des **lignes** et des **colonnes**.

Tout d'abord, les premiers réglages se font dans la balise « **<GridLayout>** » et d'y placer les attributs habituels ainsi que le nombre de colonnes et de lignes de celui-ci à l'aide des attributs « **android:columnCount=" " " " et « android:rowCount=" " " " »**

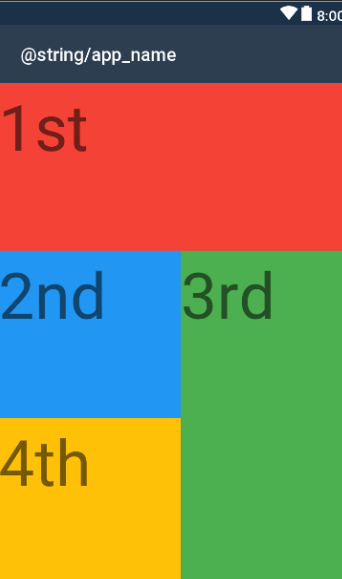
```
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:columnCount="2"
    android:rowCount="3">
```

Pour l'exemple, il y a 4 TextView placés l'un après l'autre. 3 de ces 4 éléments contiennent un attribut définissant combien de ligne il occupe  
« **android:layout\_rowSpan=" " " »**.



```
<TextView
    android:layout_width="auto"
    android:layout_height="auto"
    android:layout_columnSpan="2"
    android:background="#F44336"
    android:textSize="70sp"
    android:text="1st" />
<TextView
    android:layout_width="auto"
    android:layout_height="auto"
    android:background="#2196F3"
    android:textSize="70sp"
    android:text="2nd" />
<TextView
    android:layout_width="auto"
    android:layout_height="auto"
    android:layout_rowSpan="2"
    android:background="#4CAF50"
    android:textSize="70sp"
    android:text="3rd" />
<TextView
    android:layout_width="auto"
    android:layout_height="auto"
    android:background="#FFC107"
    android:textSize="70sp"
    android:text="4th" />
```

Il y a normalement un moyen de faire en sorte que chacun de ces éléments remplissent totalement avec les attributs « **android:layout\_rowWeight="1" " » et « android:layout\_columnWeight="1" " »**. Malheureusement dans la version actuelle, Visual Studio génère des erreurs et demande une unité qui désactive ces attributs.



```
<TextView
    android:layout_columnWeight="1"
    android:layout_rowWeight="1"
    android:layout_width="auto"
    android:layout_height="auto"
    android:layout_columnSpan="2"
    android:background="#F44336"
    android:textSize="70sp"
    android:text="1st" />
<TextView
    android:layout_width="auto"
    android:layout_height="auto"
    android:layout_columnWeight="1"
    android:layout_rowWeight="1"
    android:background="#2196F3"
    android:textSize="70sp"
    android:text="2nd" />
<TextView
    android:layout_width="auto"
    android:layout_height="auto"
    android:layout_columnWeight="1"
    android:layout_rowWeight="1"
    android:background="#4CAF50"
    android:textSize="70sp"
    android:text="3rd" />
<TextView
    android:layout_width="auto"
    android:layout_height="auto"
    android:layout_columnWeight="1"
    android:layout_rowWeight="1"
    android:background="#FFC107"
    android:textSize="70sp"
    android:text="4th" />
```



## 8.2 Création de composants

Un composant peut être créé de différentes manières, que ça soit dans le fichier **XML** en **dur** directement ou dans le code **dynamiquement**.

Il est important de bien différencier ces 2 manières de faire, car elles ne sont pas forcément utilisées pour le même but. Créer des éléments dynamiquement pourrait servir à créer plusieurs fois le même élément, mais avec une utilité différente.

### 8.2.1 Dans le fichier XML

Il y a 2 manières de base de placer un composant dans le fichier. Il est possible de directement les écrire via le clavier ou d'utiliser une boîte à outils qui placera automatiquement les balises et les attributs importants à l'intérieur.

#### 8.2.1.1 Balises

Pour créer un composant, il y a toujours besoin d'une balise **XML**. Le nom de la balise est équivalent au nom de la **classe C#**. Cela permet de définir le **type** d'élément et le récupérer dans le code.

Un composant peut avoir deux écritures de balise, mais dans tous les cas, il doit y avoir un **début** et une **fin**.

1.

```
<Button></Button>
```

2.

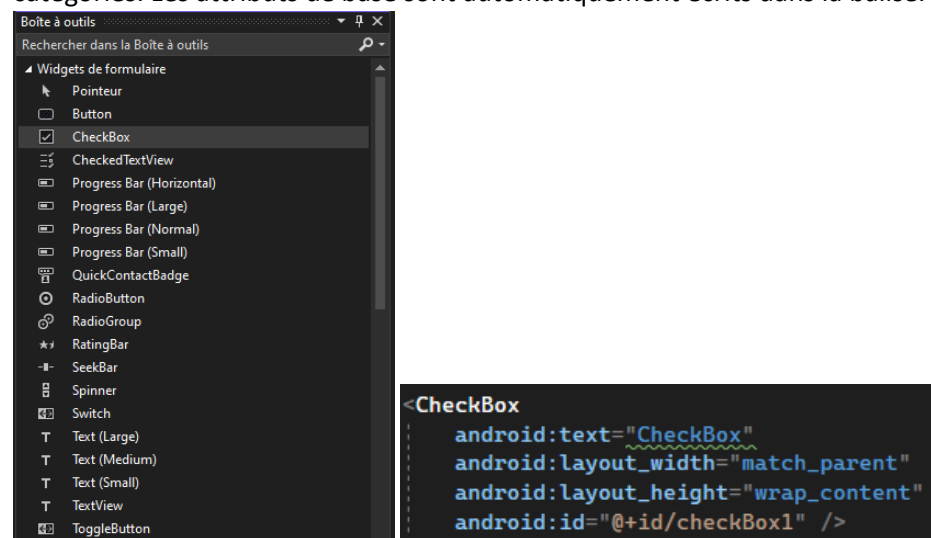
```
<Button />
```

Il existe plusieurs types d'éléments et chaque élément a ses propres balises :

- Button
- TextView
- LinearLayout
- etc...

#### 8.2.1.2 Boîte à outils

La boîte à outils permet directement de placer les balises dans le code **XML** en **double cliquant** sur un des éléments dans la liste. Depuis cet endroit, tous les éléments possibles sont affichés dans la liste et sont répertoriés dans des catégories. Les attributs de base sont automatiquement écrits dans la balise.



### 8.2.1.3 Attributs

Les attributs sont les bases d'un élément.

Ceux-ci permettent de donner le style de celui-ci : **changer la couleur du fond, du texte**, d'y inscrire un texte et d'y **changer son alignement**, sa **taille**, etc...

```
android:text="Modules"/>  
android:textColor="@android:color/white"
```

Ils permettent aussi d'attribuer un **ID** aux éléments, ce qui permet plus tard de les retrouver dans le code et de les répertorier dans le fichier de **ressources**.

Un **id** se déclare d'une certaine manière, il faut faire « **@+id/** », car il se crée automatiquement dans le fichier de **ressources**. Chaque **Id** est de type « **int** » et contient un nombre supérieur de 1 au dernier ID créé. Un ID n'est pas forcément ceux que nous avons créé, il peut être un **Id système** permettant le bon fonctionnement du programme interne.

Celui-ci contient un **nombre** et son **nom de variable** qui est placé après le « **/** ».

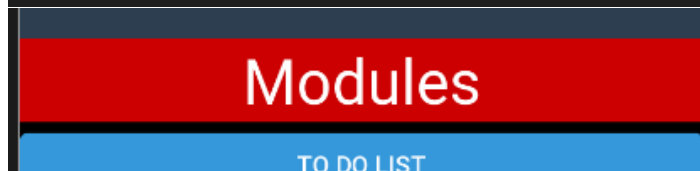
```
android:id="@+id/lblTitle"
```

Voici comment un ID est représenté dans le fichier de ressource en C#

```
// aapt resource value: 0x7F080085  
public const int largeLabel = 2131230853;  
  
// aapt resource value: 0x7F080086  
public const int lblTitle = 2131230854;
```

Devant un attribut, le mot clef « **android** » est placé pour faire référence à l'application qui est de type android.

```
<!-- Label of the title -->  
<TextView  
    android:id="@+id/lblTitle"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textColor="@android:color/white"  
    android:textAlignment="center"  
    android:textSize="35dip"  
    android:background="@android:color/holo_red_dark"  
    android:text="Modules"/>
```



## 8.2.2 Dynamiquement

Pour créer un élément **dynamiquement**, il suffit de le faire dans le code. Comme dans les points précédents, un élément est un « **objet** » du nom de sa balise. Ici pour créer un élément, cela requiert d'instancier une variable à l'aide de la classe appropriée.

Pour l'exemple, nous allons prendre des boutons. Afin de créer un bouton il faut commencer à l'aide de la classe « **Button** » et lui donner un nom de variable.

Il est aussi obligatoire d'ajouter le **contexte** dans lequel le bouton va s'afficher qui est le layout que l'on veut. Ici on y met le mot clef « this » afin que le contexte soit le layout actuel, étant donné que ce bouton a été déclaré dans la classe d'une activité.

```
// Create a button
Button button = new Button(this);
```

Il est aussi possible de créer des **layouts dynamiques** et donc d'ajouter des éléments dynamiques à l'intérieur ce layout-ci.

Désormais, il y a 2 manières de changer les attributs d'un élément.

#### 8.2.2.1 Attributs dans le corps du bouton

La première manière est d'instancier un bouton de la même manière que ci-dessus, mais en ajoutant des accolades avant le point-virgule.

En faisant ceci, nous pouvons ajouter certains attributs directement avec leur nom à l'intérieur du corps de « méthode » (qui ressemble à un corps de méthode) du bouton.

Nous ne pouvons pas définir tous les attributs dans le corps et il faudra les définir après la déclaration de la variable pour certains.

Ici on donne un texte au bouton et une visibilité qui fera en sorte que celui-ci sera invisible à l'affichage

```
// Button created dynamically visible with the text "appear"
Button btnAppear = new Button(this)
{
    Text = "appear",
    Visibility = Android.Views.ViewStates.Invisible
};
```

#### 8.2.2.2 Attributs après la déclaration de la variable

La deuxième manière d'ajouter des attributs à l'élément est d'utiliser le nom de variable suivi d'un point et de l'attribut que nous voulons définir

```
// Set the color of the text
btnAppear.SetTextColor(Color.Violet);
btnAppear.SetBackgroundColor(Color.Rgb(52, 152, 219));
```

Ici on définit la couleur du fond et du texte.

#### 8.2.2.3 Affichage de l'élément dans un layout

Désormais, nous pouvons ajouter notre élément à un layout. Pour ceci, il faut tout d'abord récupérer le layout auquel nous voulons ajouter l'élément avec la méthode « FindViewById » qui permet de récupérer tout type d'éléments dont les layouts.

```
// Get principal layout
LinearLayout layout = FindViewById<LinearLayout>(Resource.Id.lytMain);
```

Après ceci fait, nous pouvons ajouter notre élément au layout à l'aide de la variable déclarée qui récupère le layout principal de l'activité grâce à la méthode « AddView() » qui permet d'y entrer l'élément que nous venons de créer.

```
layout.AddView(btnAppear);
```

### 8.2.3 Lien entre un composant et le code

Lorsque l'on crée un élément dans le fichier XML, nous pouvons faire le lien entre le code et cet élément en le récupérant dans un objet dans le code. Prenons l'exemple de la création d'un bouton dans le XML.

```
<Button
    android:text="Faire apparaître le bouton 'Appare'"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/btnAppareVisible" />
```

FAIRE APPARAÎTRE LE BOUTON 'APPEAR'

Dans le code, nous pouvons désormais récupérer ce bouton à l'aide de la méthode « findViewById » en déclarant un objet de type « Button » étant donné que l'élément créé dans le XML est un bouton.

```
// Get btnAppareVisible
Button btnAppareVisible = findViewById<Button>(Resource.Id.btnAppareVisible);
```

#### 8.2.3.1 Créer des événements

Pour terminer, nous pouvons ajouter des méthodes événementielles à ce bouton en ajoutant « .click += » après la variable du bouton en spécifiant le nom de la méthode que nous voulons qu'elle s'exécute lorsque l'utilisateur appuie dessus.

```
btnAppareVisible.Click += BtnAppareVisible_Click;
```

Lorsque l'on clique sur ce bouton, la méthode événementielle liée s'exécute. Pour l'exemple, ici nous allons faire en sorte que le texte change et que le bouton créé plus tôt s'affiche. Pour ceci le bouton a besoin d'être instancié dans les variables de classes.

Maintenant, pour changer la visibilité du bouton, il suffit de reprendre l'objet du bouton créé plutôt.

Ensuite, il y a 2 éléments importants dans la signature de méthode qui sont le « sender » et le « e » ces deux éléments permettent de récupérer des informations en rapport au clic du bouton.

Le « sender » est l'élément qui a été cliqué, donc l'objet du bouton en lui-même, ce qui permet d'affecter des changements à celui-ci à l'aide d'attributs, ici on y change le texte.

Le « e » est l'événement de clic du bouton et donne des informations par rapport à l'événement qui a été effectué.

```
/// <summary>
/// Make appear the button btnAppare when clicked
/// </summary>
/// <param name="sender">Element clicked</param>
/// <param name="e">Event information</param>
I référence
private void BtnAppareVisible_Click(object sender, EventArgs e)
{
    // Make visible the btnAppare button
    _btnAppare.Visibility = Android.Views.ViewStates.Visible;

    // Set the text of the clicked button by "button was clicked"
    (sender as Button).Text = "Button was clicked";
}
```

Voici le résultat de cet événement.

[videoEvent.mp4](#)

### 8.3 Lier le fichier CS avec le fichier XML

La simple ligne de code « `SetContentView();` » permet d'associer le fichier .cs au fichier .xml comme ci-dessous.

```
0 références
protected override void OnCreate(Bundle savedInstanceState)
{
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState);
    // Set our view from the "main" layout resource
    SetContentView(Resource.Layout.activity_main);
}
```

À sa création, le fichier XML de l'activité s'inscrit dans les ressources à l'aide d'un ID et se retrouve à l'aide de « `Ressource.Layout.activity_name` »

### 8.4 Lien entre 2 activités

Le lien entre 2 activités et plus permet de voyager, ainsi que de passer des données entre elles. C'est pourquoi il est important d'assimiler le fait qu'à chaque fois que nous voulons faire le lien entre 2 activités, il faut que la seconde existe et de la même manière que la première créée automatiquement.

Ici il y a plusieurs moyens de faire le lien entre elles qui permettent soit de simplement naviguer ou d'envoyer des données et d'en récupérer.

#### 8.4.1 Créer une seconde activité

A la création du projet, la première activité a déjà été créée, désormais, nous devons en créer une seconde du nom de notre choix.

Une activité se crée en 3 étapes :

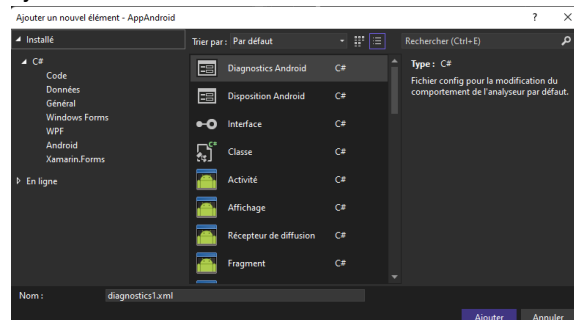
- Création du fichier C#
- Création du fichier XML
- Liaison des fichiers

En tout premier lieu, afin de créer le fichier C# qui contiendra le code de l'activité, il faut se rendre dans l'explorateur de solutions à l'endroit de votre choix où se trouveront toutes vos activités.

Pour se faire, il faut effectuer cette combinaison suivante :  **clic droit -> Ajouter ->**

**Nouvel élément.**

Une fois ceci fait, une fenêtre s'ouvre permettant de choisir quel type de fichier nous voulons ajouter.



Dans cette fenêtre, il faut rechercher « Activité » et lui donner un nom approprié.

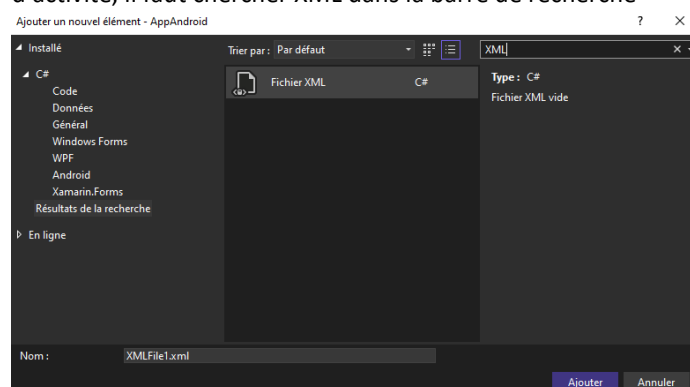


Une fois ceci fait la classe de cette activité a été créée et comporte la base de création d'une activité

```
namespace AppAndroid.Views
{
    [Activity(Label = "SecondActivity")]
    public class SecondActivity : Activity
    {
        protected override void OnCreate(Bundle savedInstanceState)
        {
            base.OnCreate(savedInstanceState);

            // Create your application here
        }
    }
}
```

Maintenant, il faut créer le fichier XML qui sera lié à cette activité. Comme pour la classe C#, cette fois-ci, il faut se rendre dans **Resources -> layout** et effectuer la même action qu'avant à un détail près. **Clic droit -> Ajouter -> Nouvel élément** et à la place de créer un fichier d'activité, il faut chercher XML dans la barre de recherche



Ici nous sélectionnons le seul élément affiché et le nommons comme nous le souhaitons, ici la nomenclature est d'écrire « activity\_<NomDeL'activité> » et souvent en référence au nom de classe C# que nous venons de créer.

**activity\_second.xml**

Une fois le fichier créé, libre à vous de faire le style à l'intérieur de celui-ci

A ce stade-là, il ne nous reste plus qu'à faire la liaison de ces deux fichiers. Comme au point 8.3, la ligne de code permettant d'associer les fichiers est « **SetContentView();** »

```
protected override void OnCreate(Bundle savedInstanceState)
{
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState);
    // Set our view from the "main" layout resource
    SetContentView(Resource.Layout.activity_second);
}
```

## 8.4.2 Appeler une autre activité

Dans ce chapitre, le but est de voir comment naviguer entre plusieurs activités

### 8.4.2.1 Navigation vers une autre activité

Afin de simplement appeler une seconde activité uniquement pour but de l'afficher se résume à une simple ligne de code.

Ici prenons l'exemple lors de l'événement de clic d'un bouton.

La méthode « **StartActivity()** » permet d'appeler l'activité choisie. Comme vu avant, une activité est composée de deux fichiers, un fichier XML et un fichier C# avec une classe héritant de la classe « **Activity** ». Cette classe permet

d'afficher l'activité et de la gérer, c'est pourquoi, il faut placer le nom de classe de l'activité dans cette méthode. Le « `typeof` », permet de spécifier que nous rentrons un type.

```
/// <summary>
/// Called when the buttons is clicked
/// </summary>
/// <param name="sender">Sender object</param>
/// <param name="e">EventArgs</param>
1 référence
private void btnToDoList_Click(object sender, EventArgs e)
{
    StartActivity(typeof(ToDoListActivity));
}
```

#### 8.4.2.2 Passage de données entre 2 activités

Android permet aussi de la possibilité de passer des données d'une activité à une autre. Pour faire ceci, cela utilise la classe **Intent** qui permet d'effectuer des actions bien précises à Android comme pour un appel téléphonique, l'ouverture d'une page internet et bien sûr d'afficher une autre activité.

C'est donc grâce à cette classe que nous allons pouvoir faire transiter des données entre les activités.

Pour ceci, il faut créer un nouvel objet de **Intent** et lui donner les données que nous souhaitons et démarrer la navigation grâce à la méthode utilisée pour le point précédent.

```
Intent activity = new Intent(this, typeof(SecondActivity));
activity.PutExtra("Data", "Datas to send to another activity");
activity.PutStringArrayListExtra("Names", new List<string> { "Jerôme", "Michel", "Paul" });
StartActivity(activity);
```

Comme on peut le voir dans le code, il y a 2 manières d'envoyer des données, soit des données simples, comme ici, du texte ou des collections génériques comme des tableaux ou des listes.

#### 8.4.2.3 Récupération des données

Afin de récupérer les données envoyées ci-dessus, il faut aller dans la classe concernée, ici **SecondActivity**.

La récupération de données se fait principalement dans la méthode **OnCreate()** qui est la première méthode à s'exécuter.

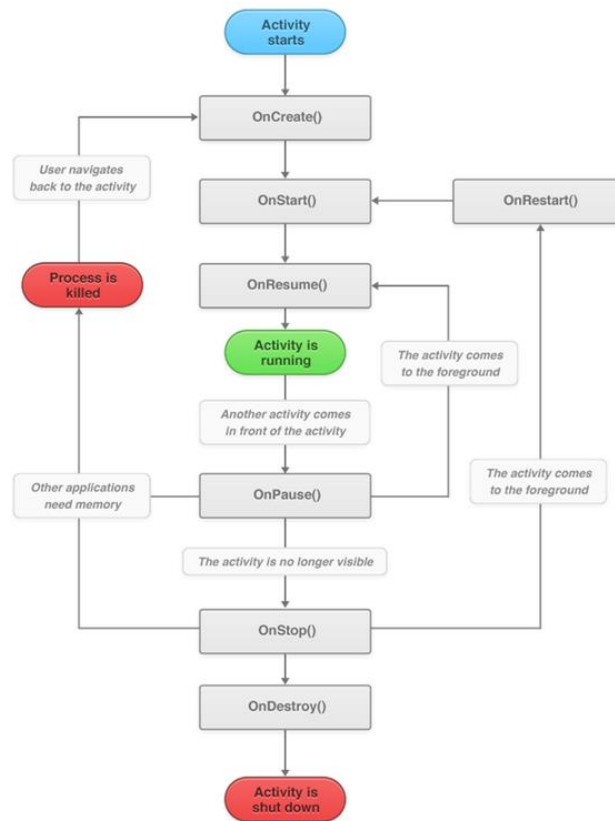
Ici, nous allons réutiliser la classe **Intent** et utiliser la méthode `Get<type>Extra("<nomDeDonnée>")` ici nous allons remplacer <nomDeDonnée> par **Data** et <type> par le type de variable adéquat qui peut être de tout type (int, float, boolean, string, ...) qui sera un **string**

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

Cette valeur peut ensuite être récupérée dans une variable !

## 9 Cycle de vie d'une application

Le cycle de vie d'une application contient plusieurs états qui permettent de créer, démarrer, afficher et gérer les activités. Chacun de ces états ont leur équivalent en méthodes dans le code qui sont automatiquement créées et appelées par le programme en fonction des actions de l'utilisateur sur l'application.



Les méthodes sont définies dans la classe « **Activity** », elles permettent d'effectuer ces états à chacune des activités dû au fait que toutes les classes de celles-ci héritent de « **Activity** ». Afin de pouvoir effectuer certains états à une activité, il suffit de déclarer ces méthodes dans la classe de l'activité voulue. Etant donné l'héritage, il faut réécrire ces méthodes avec le mot clef « **override** » et « **protected** ».

```

/// <summary>
/// On the creation of the activity
/// </summary>
/// <param name="savedInstanceState">Instance</param>
0 références
protected override void OnCreate(Bundle savedInstanceState)
{
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState);
    // Set our view from the "main" layout resource
    SetContentView(Resource.Layout.activity_menu);

    // Get the buttons and assigns events
    _btnToDoList = FindViewById<Button>(Resource.Id.btnToDo);
    _btnToDoList.Click += _btnToDoList_Click;

    _btnMyDay = FindViewById<Button>(Resource.Id.btnMyDay);
    _btnMyDay.Click += _btnMyDay_Click;

    _btnCategories = FindViewById<Button>(Resource.Id.btnCategories);
    _btnCategories.Click += _btnCategories_Click;
}

```



## 10 ListView

Les `ListView` sont des listes permettant d'y placer et afficher des données. Elles peuvent être créées dans le fichier XML ou dans le code C#. Tout type de données peuvent être affichées et de différentes manières.

Nous allons voir comment faire une `ListView` classique et une personnalisée et dans les deux cas, nous aurons besoin d'un élément que nous appelons **adaptateur**.

### 10.1 ListView classique

Afin de créer une `ListView` classique, l'**adaptateur** peut être créé à l'aide d'un modèle de base et des données écrites en dur.

Tout d'abord, il faut créer un objet de type **ArrayAdapter** à l'aide de la méthode statique **CreateFromResource** qui retourne **ArrayAdapter**.

Cette méthode a comme paramètres :

- Le contexte donné, qui ici est l'activité actuelle
- Le nom du tableau de données en dur à donner à l'adaptateur, qui est un tableau codé en dur dans le fichier de ressource **strings.xml**

```
<array name="array_tasks">
  <item>Faire les courses</item>
  <item>Apprendre le typescript</item>
  <item>Finir la doc Xamarin</item>
  <item>Préparer le mariage</item>
  <item>Prendre rendez-vous pour le stage de 3ème année</item>
</array>
```

- Le layout modèle à utiliser

```
// Create Adapter
var classicAdapter = ArrayAdapter.CreateFromResource(this, Resource.Array.array_tasks, Android.Resource.Layout.SimpleSpinnerItem);
```

Ensuite, afin de créer la liste déroulante, il faut maintenant utiliser la méthode **SetDropDownViewResource** avec comme paramètre le layout modèle pour l'affichage.

```
// Set a drop down view to the adapter
classicAdapter.SetDropDownViewResource(Android.Resource.Layout.SimpleSpinnerDropDownItem);
```

Pour finir, il faut ajouter notre adaptateur à la **ListView** concernée à l'aide de notre objet le contenant en attribuant notre **adaptateur** à la variable **Adapter** de la **ListView**.

```
// Add to the ListView
_listView = FindViewById<ListView>(Resource.Id.lstTasks);
_listView.Adapter = classicAdapter;
```

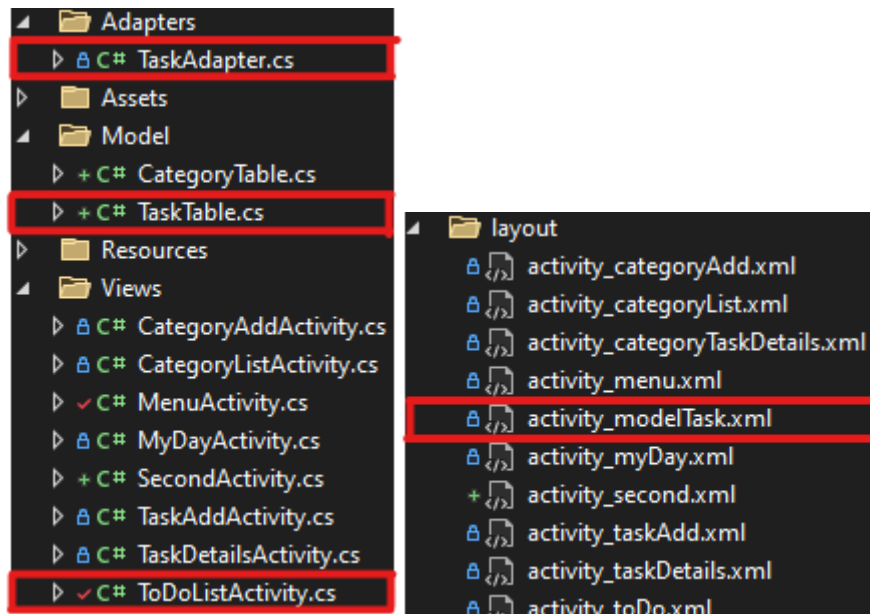
### 10.2 ListView personnalisée

Maintenant que nous savons faire une **ListView** classique, il est possible de créer la sienne de toute pièce, afin d'y afficher ce que l'on veut.

Par comparaison, la **ListView** classique ne permet que d'afficher du texte. Maintenant, personnalisée, elle permet d'afficher ce que l'on veut sur une des lignes de celle-ci. Ce qui veut dire qu'il peut y avoir plusieurs éléments pour une seule tâche (Un layout entier défini).

Pour ceci, nous allons commencer par définir une arborescence de fichier correcte afin de trier nos classes et nous permettre de se retrouver facilement.

Nous aurons besoin de créer ces classes et layout en **rouge** ci-dessous :



### 10.2.1 Model

Le dossier **Model** permet de contenir toutes les données de la même manière qu'une base de données. Ici nous pouvons créer un fichier par table (classe) ou toutes les tables (classes) dans le même fichier.

Voici comment se crée une classe contenant des données.

```

/// <summary>
/// Datas of a task
/// </summary>
1 référence
public class TaskTable
{
    /// <summary>
    /// Id of the task
    /// </summary>
    2 références
    public int ID { get; set; }

    /// <summary>
    /// Name of the task
    /// </summary>
    2 références
    public string Name { get; set; }

    /// <summary>
    /// Description of the task
    /// </summary>
    2 références
    public string Description { get; set; }

    /// <summary>
    /// End date of the task
    /// </summary>
    2 références
    public DateTime EndDate { get; set; }

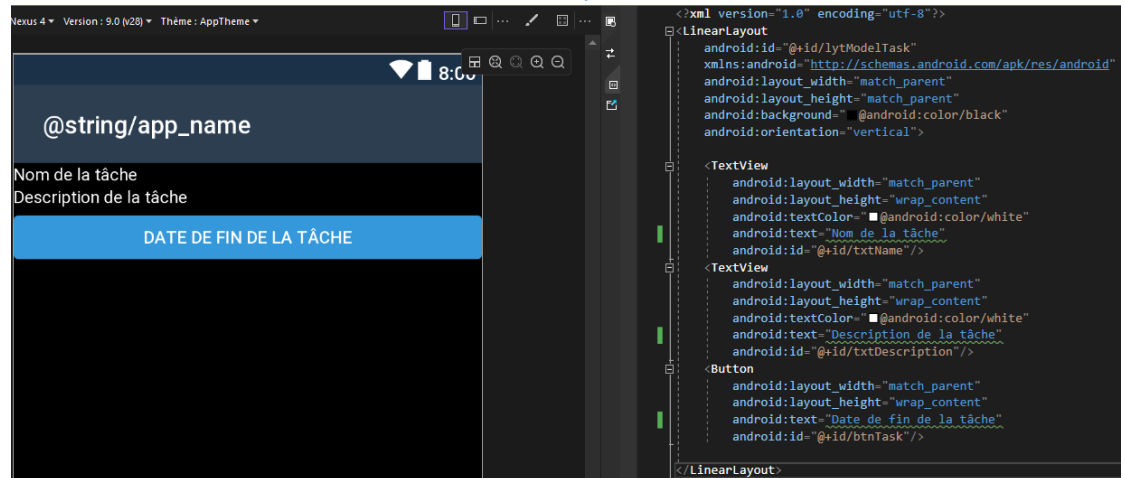
    /// <summary>
    /// Task constructor
    /// </summary>
    /// <param name="id">Id of the task</param>
    /// <param name="name">Name of the task</param>
    /// <param name="description">Description of the task</param>
    /// <param name="endDate">End date of the task</param>
    0 références
    public TaskTable(int id, string name, string description, DateTime endDate)
    {
        ID = id;
        Name = name;
        Description = description;
        EndDate = endDate;
    }
}

```

Etant donné que nous ne sommes pas encore passé par le chapitre de persistance de données, la classe n'est pas à sa version finale.

### 10.2.2 Layout

Le layout est un point important à la création d'une **ListView** personnalisée. Il permet d'afficher les données de la façon que l'on souhaite et pour ce faire, il faut créer un nouveau fichier XML comme décrit [ici \(ListView personnalisée\)](#) dans l'arborescence.



Une fois le layout terminé, il pourra être prêt à être utilisé dans **l'adaptateur**.

Evidemment, les textes affichés ici seront remplacés dans le futur par de véritables données, cela sert uniquement à voir le rendu.

### 10.2.3 Adaptateurs

Le dossier **Adapters** permet de contenir tous les **adaptateurs** différents que nous allons créer. Il existera autant d'**adaptateurs** que de modèles différents.

Dans notre fichier **TaskAdapter**, nous allons créer une classe qui va hériter de **BaseAdapter<TaskTable>** étant la base d'un adaptateur qui contient un type. Ce type est donc la classe **TaskTable** dont nous avons parlé précédemment.

Pour commencer, il nous faut 2 variables de classe qui sont :

- Une liste de tâche, qui va être remplie plus tard
- Une activité, qui est l'activité dans laquelle nous allons afficher ces données

```

/// <summary>
/// List of task
/// </summary>
3 références
public List<TaskTable> Tasks { get; }

/// <summary>
/// Actual activity
/// </summary>
public Activity Activity;

```

Ainsi qu'un **constructeur** qui dans ses paramètres contient l'équivalent des deux variables déclarées plus hautes.

```

/// <summary>
/// Task adapter constructor
/// </summary>
/// <param name="tasks">List of tasks</param>
/// <param name="activity">Actual activity</param>
1 référence
public TaskAdapter(List<TaskTable> tasks, Activity activity) : base()
{
    Tasks = tasks;
    Activity = activity;
}

```

Ensuite, votre **IDE** va retourner une erreur expliquant que la classe n'implémente pas les membres abstraits hérités de **BaseAdapter**.

```
class TaskAdapter : BaseAdapter<TaskTable>
{
    // class AppAndroid.Adapters.TaskAdapter
    // Adapter of tasks
    // CS0534: 'TaskAdapter' n'implémente pas le membre abstrait hérité 'BaseAdapter.GetView(int, View?, ViewGroup?)'
    // CS0534: 'TaskAdapter' n'implémente pas le membre abstrait hérité 'BaseAdapter.Count.get'
    // CS0534: 'TaskAdapter' n'implémente pas le membre abstrait hérité 'BaseAdapter.GetItemId(int)'
    // CS0534: 'TaskAdapter' n'implémente pas le membre abstrait hérité 'BaseAdapter<TaskTable>.this[int].get'
```

Ceci est tout à fait normal, il suffit juste de faire **clic droit** sur le nom de votre classe et d'appuyer sur **Actions rapides et refactorisations..**

Actions rapides et refactorisations... Ctrl+.

Ainsi qu'**Implémenter une classe abstraite**, ce qui va automatiquement écrire les méthodes obligatoires demandées par la classe **BaseAdapter**.

class TaskAdapter

Implémenter une classe abstraite

Une fois ceci fait, il faut modifier les éléments à l'intérieur de leurs corps.

Celle-ci permet de retourner une tâche à une position donnée

```
/// <summary>
/// Get a task on a specific position
/// </summary>
/// <param name="position">Position on the list of tasks</param>
/// <returns>Returns a task object on the position</returns>
0 références
public override TaskTable this[int position]
{
    get { return Tasks[position]; }
}
```

Celle-ci permet de retourner le nombre de tâches existantes dans la liste de tâches

```
/// <summary>
/// Get the number of tasks
/// </summary>
0 références
public override int Count
{
    get { return Tasks.Count; }
}
```

Et celle-ci permet de retourner l'id d'une tâche

```
/// <summary>
/// Get the id of a task
/// </summary>
/// <param name="position">Position of the task in the list</param>
/// <returns>Returns an id => INT</returns>
0 références
public override long GetItemId(int position)
{
    return position;
}
```

La dernière fait exactement la même chose que la première méthode, mais utilise du Java. Elle n'est pas utile ni obligatoire

```
/// <summary>
/// Get a task on a specific position (Not usefull method)
/// </summary>
/// <param name="positon">Position on the list of tasks</param>
/// <returns>Returns a task object on the position</returns>
0 références
public override Java.Lang.Object GetItem(int positon)
{
    // Return nothing
    return null;
}
```

Désormais, toutes les méthodes demandées ont été faites, passons maintenant à la plus importante de toutes.

Cette méthode permet d'affecter les données (tâches) au **layout modèle** et le retourner afin de l'ajouter à la **ListView**

Nous récupérons d'abord dans un objet la **tâche** actuelle.  
 Ensuite, nous récupérons le **layout du modèle** à l'aide de son **ID**.  
 On lui attribue les données : **Nom, Description, Date de fin**.  
 Et pour finir, on retourne la **view** afin de l'afficher dans la **ListView**

```

/// <summary>
/// Set the render model of ListView
/// </summary>
/// <param name="position">Position on the list of tasks</param>
/// <param name="convertView">View</param>
/// <param name="parent">parent View group</param>
/// <returns>Returns a view of datas</returns>
0 références
public override View GetView(int position, View convertView, ViewGroup parent)
{
    // Create a task
    TaskTable task = Tasks[position];

    // Get the view and set datas
    View v = Activity.LayoutInflater.Inflate(Resource.Layout.activity_modelTask, null);
    v.FindViewById<TextView>(Resource.Id.txtName).Text = task.Name;
    v.FindViewById<TextView>(Resource.Id.txtDescription).Text = task.Description;
    v.FindViewById<Button>(Resource.Id.btnTask).Text = task.EndDate.ToString();

    // Returns view
    return v;
}

```

Bien sûr, il est possible de ne pas personnaliser le layout et d'utiliser un **modèle**, mais il est toujours mieux de le faire quand même afin d'être sûr du rendu final.

#### 10.2.4 Views

Le dossier **Views** contient toutes les classe C# liées aux **activités**, c'est en quelque sorte le **front-end** de l'application et c'est donc ici que nous allons afficher notre **ListView** avec nos données.

C'est donc dans cette **ListView** que nous allons afficher nos données.

```

<!-- Tasks layout -->
<LinearLayout
    android:id="@+id/lytToDoTasks"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <ListView
        android:id="@+id/lstTasks"
        android:minWidth="25px"
        android:minHeight="25px"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>
</LinearLayout>

```

Le reste va se passer dans la classe C#.

Pour l'exemple, créons des tâches écrites en dur dans une nouvelle liste dans la classe **ToDoListActivity**.

```

List<TaskTable> _tasks;

// Create your application here
_tasks = new List<TaskTable>();
_tasks.Add(new TaskTable(0, "Apprendre le typescript", "Apprendre le typescript en Web pour créer des sites personnels", DateTime.Now));
_tasks.Add(new TaskTable(1, "Finir la doc Xamarin", "Finir la doc Xamarin pour mardi", DateTime.Now));
_tasks.Add(new TaskTable(2, "Faire les courses", "Prendre 25Kg de lait, 12L de farine et 7g d'oeufs", DateTime.Now));

```

Une fois ceci fait, créons un nouveau **TaskAdapter** en lui donnant la liste de tâche et le mot clef **this** étant l'activité actuelle.

```

// Get new task adapter
TaskAdapter taskAdapter = new TaskAdapter(_tasks, this);

```

Ensuite, nous devons récupérer la **ListView** de l'activité

```

ListView _listView;

// Get ListView
_listView = FindViewById<ListView>(Resource.Id.lstTasks);

```

Pour finir, nous devons impérativement ajouter notre **adaptateur** à notre **ListView**.

```
// Add adapter to the ListView
_listView.Adapter = taskAdapter;
```

Et que la magie opère.

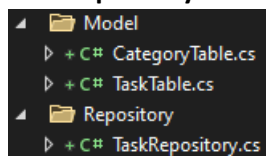
[videoListView-Adapter.mp4](#)

## 11 Persistance des données

La persistance des données est en soit la **base de données** de notre application sur notre téléphone. Ce n'est pas du SQL, mais du SQLite supporté par les smartphones pour gérer des bases de données internes au système.

Afin de mettre en place de la persistance des données, il nous faut utiliser et modifier les fichiers C# : **TaskTable** et **CategoryTable** et il faudra désormais ajouter un nouveau fichier qui va faire office de gérant de la base de données qui permettra de se connecter et d'exécuter des requêtes :

**TaskRepository.**



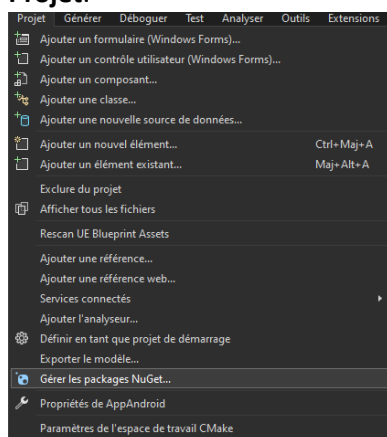
### 11.1 Mise en place de la DB

Afin de mettre sur pied notre **base de données** nous allons utiliser **SQLite** qui est un système simple d'utilisation et peu couteux en ressources.

Comme dit au-dessus, ce sont des bases de données locales et donc ne permettent pas de se connecter à un serveur afin de créer des applications connectées.

Maintenant que nous avons tous les fichiers que nous avons besoin, nous allons commencer par mettre en place **SQLite**. En premier lieu, il faut s'assurer d'avoir les modules **SQLite** d'installés qui sont **sqlite-net-pcl** et **SQLiteNetExtensions**.

Ces modules se trouvent dans le gestionnaire de package **NuGet** qui se trouve dans la barre **Projet**.



**sqlite-net-pcl** par SQLite-net, 9,87M téléchargements

1.8.116

SQLite-net is an open source and light weight library providing easy SQLite database storage for .NET, Mono, and Xamarin applications.



**SQLiteNetExtensions** par TwinCoders, 1,37M téléchargements

2.1.0

SQLite-Net Extensions is a very simple ORM that provides cascade operations, one-to-one, one-to-many, many-to-one, many-to-many, inverse and text-blobbed relationships on top of the sqlite-net library.

Il faudra ensuite utiliser, dans chacune des classes de votre **DB**, les using concernant **SQLite** :

- using SQLite ;
  - Mots clef: [Table/PrimaryKey/AutoIncrement/...]
- using Java.Sql ;
- using SQLiteNetExtensions.Attributes ;
  - Gérer les relations entre les tables

```
using SQLite;
using Java.Sql;
using SQLiteNetExtensions.Attributes;
```

## 11.2 Création de la DB

Maintenant que la **DB** et **SQLite** ont été configurés, nous pouvons passer à la création de celle-ci dans les classes respectives.

Pour créer notre **DB**, nous utilisons les mots clefs de **SQLite** et afin de mettre un mot clef, il faut toujours l'entourer de crochets.

### 11.2.1 Créer une table

Afin de créer notre table, nous devons mettre le mot clef **Table(<nomTable>)** en dessus de la déclaration de notre classe.

```
/// <summary>
/// Datas of a task
/// </summary>
[Table("t_task")]
11 références
public class TaskTable
{
    /// <summary>
    /// Datas of a category
    /// </summary>
    [Table("t_category")]
    1 référence
    public class CategoryTable
    {
```

Malgré le fait que cette classe s'appelle **TaskTable**, nous pouvons donner le nom que nous voulons à notre table.

### 11.2.2 Créer des champs

Maintenant que notre **table** est créée, nous pouvons passer aux **champs**. Le fonctionnement est exactement le même, cependant nous utilisons d'autres mots clefs en fonction du champ et des restrictions que nous voulons lui donner.

```
/// <summary>
/// Id of the task
/// </summary>
[PrimaryKey, AutoIncrement, NotNull, Unique, MaxLength(10)]
2 références
public int ID { get; set; }

/// <summary>
/// Name of the task
/// </summary>
[MaxLength(100)]
3 références
public string Name { get; set; }

/// <summary>
/// Description of the task
/// </summary>
[MaxLength(1000)]
3 références
public string Description { get; set; }

/// <summary>
/// End date of the task
/// </summary>
3 références
public DateTime EndDate { get; set; }

/// <summary>
/// Id of the task
/// </summary>
[PrimaryKey, AutoIncrement, NotNull, Unique, MaxLength(10)]
0 références
public int ID { get; set; }

/// <summary>
/// Name of the task
/// </summary>
[MaxLength(100)]
0 références
public string Name { get; set; }

/// <summary>
/// Color of the category
/// </summary>
0 références
public Color Color { get; set; }
```

Nous pouvons aussi créer des **clef étrangères** d'une autre table pour permettre les relations entre les deux **tables**.

L'argument **ForeignKey()** fait en sorte que la variable **FkCategory** soit une clef étrangère et **ManyToOne** contribue aussi à la relation. C'est l'équivalent d'une

relation à deux facteurs. Ici 0-N -> 0-1 et permet à ce qu'une tâche n'ait qu'une seule catégorie et qu'une catégorie puisse appartenir à plusieurs tâches.

```
/// <summary>
/// Foreign key of the category
/// </summary>
[ForeignKey(typeof(CategoryTable))]
0 références
public int FkCategory { get; set; }

/// <summary>
/// Category of the task
/// </summary>
[ManyToOne]
0 références
public CategoryTable Category { get; set; }
```

### 11.3 Manipulation des données

Pour terminer ce chapitre de base de données, nous allons voir comment manipuler la **DB** et cette fois-ci avec le nouveau fichier que nous avons créé (**TaskRepository**).

De la même façon que nos tables, nous aurons besoin du **using SQLite** pour effectuer la connexion à la base de données.

Tout d'abord, on instancie deux variables responsables des **messages de statuts** de la **DB** et de la **connexion** à cette dernière.

```
/// <summary>
/// Manage the database
/// </summary>
0 références
public class TaskRepository
{
    // Status message
    0 références
    public string StatusMessage { get; set; }

    // Connection SQLite variable
    private SQLiteAsyncConnection _connection;
}
```

Ensuite, nous devons déclarer le constructeur de la classe et nous connecter à la DB ainsi que créer la première table **TaskTable**.

```
/// <summary>
/// Task repository constructor
/// </summary>
/// <param name="path">Path of the database</param>
0 références
public TaskRepository(string path)
{
    //Get the DB connection
    _connection = new SQLiteAsyncConnection(path);

    // Creation of the task table
    _connection.CreateTableAsync<TaskTable>();
}
```

Afin de manipuler correctement les données, toutes les méthodes doivent être **asynchrones**. Il y a par exemple la méthode d'ajout d'une tâche à laquelle nous devons ajouter un **try/catch** en cas d'erreurs et gérer le message d'erreur dans une variable.

```
/// <summary>
/// Add a new task
/// </summary>
/// <param name="name">Name of the task</param>
/// <param name="description">Description of the task</param>
/// <returns></returns>
0 références
public async System.Threading.Tasks.Task AddTaskAsync(string name, string description)
{
    var result = 0;

    try
    {
        // Insert into task table
        result = await _connection.InsertAsync(new TaskTable { Name = name, Description = description });
        StatusMessage = $"tâche ajoutée : {name} - {result}";
    }
    catch (Exception ex)
    {
        // Error message
        StatusMessage = $"Error AddTaskAsync {name}. Error : {ex.Message}";
    }
}
```



Désormais, il suffit d'ajouter une tâche à la **DB** en créant un objet de type **TaskRepository** où l'on souhaite permettre l'ajout d'une tâche. Ici **TaskAddActivity**.

Mais avant toute chose, il faut d'abord donner l'emplacement de la **base de données**.

```
public class TaskAddActivity : Activity
{
    private string _path = System.IO.Path.Combine(FileSystem.AppDataDirectory, "db_335-AppMobile");

    // Create a new task
    TaskRepository taskRepository = new TaskRepository(_path);
    taskRepository.AddTaskAsync("Apprendre le typescript", "Apprendre le typescript en Web pour créer des sites personnels");
}
```

## 12 Mise en place d'un capteur (Sensor)

Un capteur permet d'exécuter un programme à chaque fois qu'il détecte un **événement extérieur** propre à son type, cependant, en fonction de **la version du système d'exploitation et le smartphone**, certains **capteurs** ne sont pas forcément présents sur l'appareil, c'est pourquoi il faut faire attention dans le cas où celui-ci est indispensable à l'application et de restreindre cette dernière aux appareils le possédant pas.

Il existe plusieurs types de capteurs, comme le **capteur de mouvements**, le **capteur de rotation**, le **capteur de luminosité**, etc...

Pour ceci nous devons utiliser des classes appartenant à la au package **android.hardware**.

```
using Android.Hardware;
```

Ici nous allons rester sur la base d'un capteur de mouvement et y affecter la tâche de supprimer toutes les tâches du jour lorsque l'on secoue son smartphone.

### 12.1 Détecter la disponibilité des capteurs

Comme dit précédemment, nous ne savons pas un appareil à un certain capteur et peut interagir avec. C'est pourquoi nous devons vérifier quels capteurs sont disponible avec le smartphone que nous testons.

En premier lieu, nous devons récupérer notre capteur en déclarant un objet de type **SensorManager** qui permet de gérer tous les capteurs disponibles, d'où son nom.

```
// Get the sensor manager
SensorManager sensorMgr = (SensorManager)GetSystemService(SensorService);
```

Ensuite, nous allons récupérer tous les capteurs disponibles dans une **ICollection** de **Sensor** à l'aide de la méthode **GetSensorList** sur le **SensorManager**.

```
// list of all the sensors
ICollection<Sensor> sensorList = sensorMgr.GetSensorList(SensorType.All);
```

Pour finir, nous allons afficher les noms de tous les capteurs dans un **TextView** afin de définir lesquels sont utilisables.

```
// Get the TextView
TextView txtSensors = FindViewById<TextView>(Resource.Id.txtSensors);

// Display the sensor names
foreach (Sensor sensor in sensorList)
{
    txtSensors.Text = $"{sensor.Name}";
}
```

Une fois ceci fait, voici un résultat que nous pouvons obtenir.

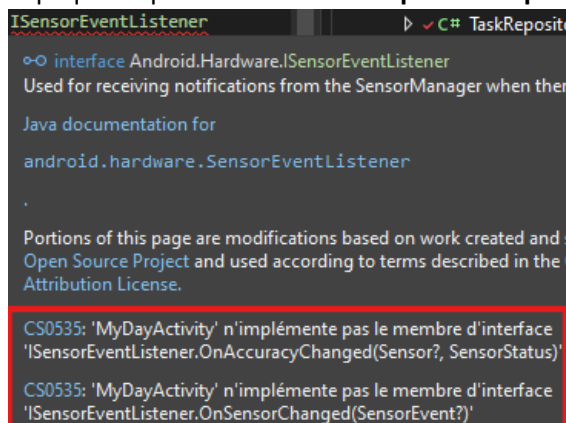
```
Goldfish 3-axis Accelerometer
Goldfish 3-axis Gyroscope
Goldfish 3-axis Magnetic field sensor
Goldfish Orientation sensor
Goldfish Ambient Temperature sensor
Goldfish Proximity sensor
Goldfish Light sensor
Goldfish Pressure sensor
Goldfish Humidity sensor
Goldfish 3-axis Magnetic field sensor (uncalibrated)
Game Rotation Vector Sensor
GeoMag Rotation Vector Sensor
Gravity Sensor
Linear Acceleration Sensor
Rotation Vector Sensor
Orientation Sensor
```

## 12.2 Utiliser le capteur

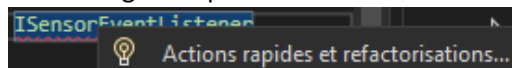
Désormais, nous savons quels capteurs nous pouvons utiliser, il est temps de mettre en place un événement pour lorsque le capteur que nous définirons changera d'état.

Afin de pouvoir utiliser ces fameux événements liés aux capteurs, il est impératif de faire **hériter** la classe dans laquelle nous voulons utiliser ce capteur par **ISensorEventListener** qui est une interface.

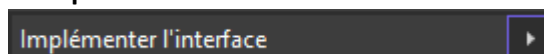
Une fois l'héritage de cette interface effectuée, Visual Studio, va nous retourner des erreurs expliquant que **notre classe n'implémente pas le membre d'interface**.



afin de régler ce problème : **clic droit sur l'interface -> Actions rapides et refactorisations...**



**Et Implémenter l'interface.**



Cela va automatiquement écrire les méthodes obligatoires et nous permettre d'avancer.

```
0 références
public void OnAccuracyChanged(Sensor sensor, [GeneratedEnum] SensorStatus accuracy)
{
    throw new NotImplementedException();
}

0 références
public void OnSensorChanged(SensorEvent e)
{
    throw new NotImplementedException();
}
```

Maintenant que nous avons ceci, nous pouvons déclarer les premières variables importantes au bon déroulement.

Pour plus de lisibilité, nous pouvons déclarer des variables de classe qui seront :

- **SensorManager** => Le manager des capteurs que nous avons vu avant.
- **Sensor** => Le capteur que nous allons utiliser.
- Un **float** => Il permettra de calculer la distance parcourue par le smartphone, afin de ne déclencher l'évènement que lors de mouvements minimums, afin que chaque minuscule mouvement ne le déclenche pas.

```
private SensorManager _sensorMgr; // Sensor manager
private Sensor _accelerometer; // Accelerometer sensor
private float _posX; // Pos x of the smartphone
```

Afin de d'enregistrer notre capteur, nous pouvons tout faire dans la méthode **OnCreate**

Tout d'abord nous allons affecter à notre variable **Sensor** un capteur à l'aide de la méthode **GetDefaultSensor** dans le **manager de capteurs**.

```
// Get the accelerometer
_accelerometer = _sensorMgr.GetDefaultSensor(SensorType.Accelerometer);
```

Comme dans le code précédant, les capteurs sont de type **Sensor**, maintenant afin de définir le capteur à utiliser, il faut utiliser la méthode **RegisterListener** dont nous allons affecter l'activité actuelle, l'accéléromètre créé précédemment et un délai normal.

```
// Register the accelerometer in the sensor manager
_sensorMgr.RegisterListener(this, _accelerometer, SensorDelay.Normal);
```

Maintenant que nos variables sont créées et que nous avons ajouté un **enregistrement** sur notre capteur, nous pouvons nous atteler à la méthode implémentée par l'interface **OnSensorChanged**.

A l'aide de cette méthode et de notre capteur sous **écoute**, à chaque fois que celui-ci détecte un mouvement, cette méthode va s'exécuter.

C'est pourquoi nous devons effectuer des calculs et des vérifications afin que chaque mouvement aussi petit qu'ils soit ne suppriment pas les tâches du jour.

En premier lieu, nous devons vérifier si le capteur est bel est bien un accéléromètre.

```
/// <summary>
/// Execute when sensor state changed
/// </summary>
/// <param name="e">Sensor event</param>
0 références
public void OnSensorChanged(SensorEvent e)
{
    // Is the sens an accelerometer
    if (e.Sensor.Equals(_accelerometer))
    {
        // ...
    }
}
```

Ensuite, nous allons récupérer les 3 valeurs qui correspondent aux 3 accélérations en **X**, **Y** et **Z** et de faire une dernière vérification pour regarder sur un mouvement **horizontal**, si la valeur du **X** du **départ** moins la valeur d'**arrivée** est plus grande que 5 pour laisser passer la suppression des tâches.

```
/// <summary>
/// Execute when sensor state changed
/// </summary>
/// <param name="e">Sensor event</param>
0 références
public void OnSensorChanged(SensorEvent e)
{
    // Is the sens an accelerometer
    if (e.Sensor.Equals(_accelerometer))
    {
        // Get the 3 value of X, Y and Z
        IList<float> values = e.Values;

        if (Math.Abs(values[0] - _posX) > 5)
        {
            // Delete the tasks
        }
        else
        {
            _posX = values[0];
        }
    }
}
```

Pour finir il faut toujours désenregistrer la capteur lorsque l'activité est en pause.

```
// Make free the sensor manager
_sensorMgr.UnregisterListener(this);
```