

## Table des matières

1	Introduction.....	2
2	Application.....	2
3	Maquette.....	3
4	Installation de l'environnement.....	3
5	Création du projet .....	4
6	Emulateur .....	5
7	Environnement et première Activité.....	6
7.1	Les fenêtres de l'environnement .....	6
7.2	Interface utilisateur .....	9
7.3	Quelque widgets.....	10
7.4	Layouts .....	10
7.4.1	LinearLayout .....	10
7.4.2	RelativeLayout .....	11
7.4.3	GridLayout .....	11
8	Fichier de ressources.....	11
9	Cycle de vie.....	12
9.1	État .....	12
9.1.1	Active / Running (Activée).....	12
9.1.2	Paused (En pause) .....	12
9.1.3	Stopped (Arrêtée) / Backgrounded (En arrière-plan).....	12
9.1.4	Restarted (Redémarrée).....	12
9.2	Méthode.....	13
9.2.1	OnCreate .....	13
9.2.2	OnStart .....	13
9.2.3	OnResume .....	13
9.2.4	OnPause.....	14
9.2.5	OnStop.....	14
9.2.6	OnRestart.....	14
9.2.7	OnDestroy.....	14
10	Lien entre deux Activités .....	14
10.1	Création d'une deuxième Activité.....	14
10.2	Lié les Activités .....	16
11	Méthodes événementielles.....	17
12	Layout dynamique .....	17

13	ListView .....	18
13.1	ListView de base .....	18
13.2	ListView personnalisé .....	19
14	Persistence de données.....	22
14.1	Base de données.....	22
14.2	Accès aux données .....	23
15	Utilisation d'un capteur.....	25
15.1	Récupération des capteurs.....	25
15.2	Utilisation d'un capteur .....	25

## 1 Introduction

Ce document a pour objectif de mettre en évidence les différentes étapes dans la réalisation d'une application Android.

Les explications qui seront précisées dans ce document seront pour des personnes déjà familiarisées avec C#.

Même si MS annonce la fin de cette plateforme avec l'arrivée de .Net Maui, nous utiliserons l'environnement Visual Studio C# avec Xamarin.

## 2 Application

L'application sera un gestionnaire de tâches afin de passer par les différentes étapes de réalisation d'un projet sur mobile.

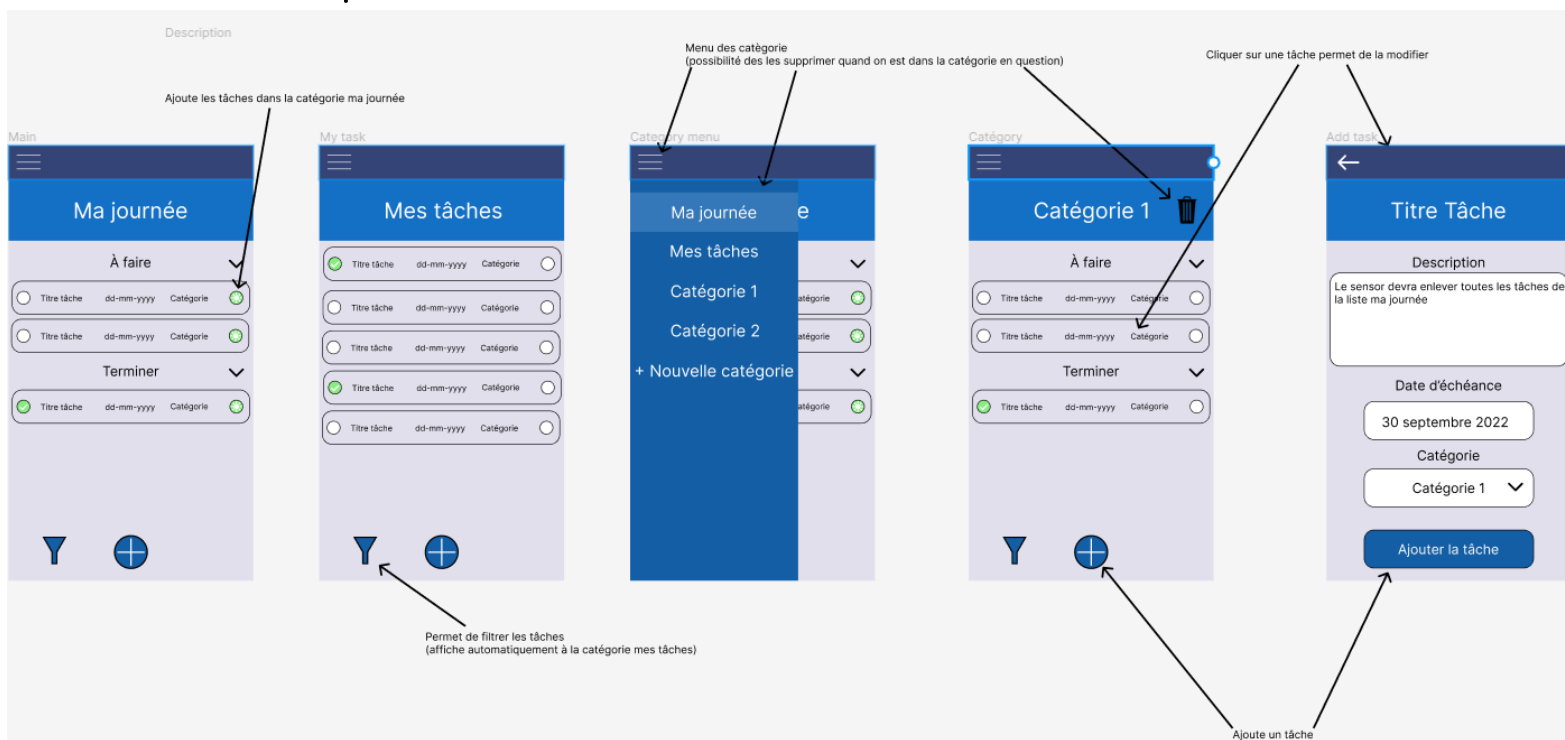
Cette application doit avoir au minimum les fonctionnalités suivantes :

- Pouvoir créer et visualiser des tâches.
- Une tâche est caractérisée au minimum par son titre, sa description et sa date d'échéance.
- Il doit être possible de mettre une tâche dans une catégorie.
- Il doit être possible de sélectionner des tâches pour en faire une liste « Ma journée », c'est-à-dire choisir les tâches à réaliser aujourd'hui.

De plus, cette application doit permettre d'aborder les thèmes suivants :

- Création d'une première Activité.
- Utilisation d'au moins un fichier de ressources, comme « Strings ».
- Lien entre deux Activités.
- Méthodes événementielles.
- Layout dynamique (c'est-à-dire création dans le code C#)
- Persistence de données.
- Utilisation d'un sensor.

### 3 Maquette



Deux types interfaces sont prévues. Lors du lancement de l'application, on tombera sur la première interface qui affiche les tâches à faire de la journée. On y ajoute des tâches choisies en cliquant sur le cercle droit de la tâche (le cercle gauche permet de finir la tâche).

En cliquant sur les trois barres en haut à gauche, l'utilisateur aura accès aux autres sections. Il sera possible de créer, renommer et supprimer les sections.

En cliquant sur le + en bas de l'écran, on peut créer une nouvelle tâche. Avec un titre, une description, une date d'échéance et une catégorie (la catégorie de base est « Mes tâches »). Il est aussi possible de modifier une tâche en cliquant sur la tâche en question.

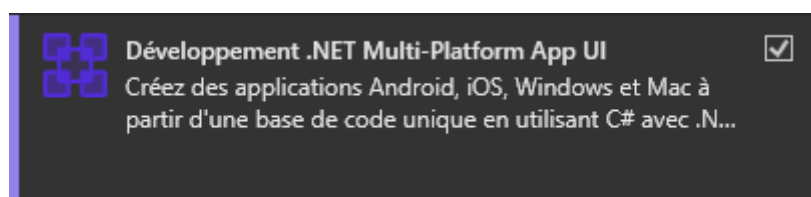
Si le temps le permet, on pourra ajouter permettant de trier par catégorie, date, terminée, non terminée.

### 4 Installation de l'environnement

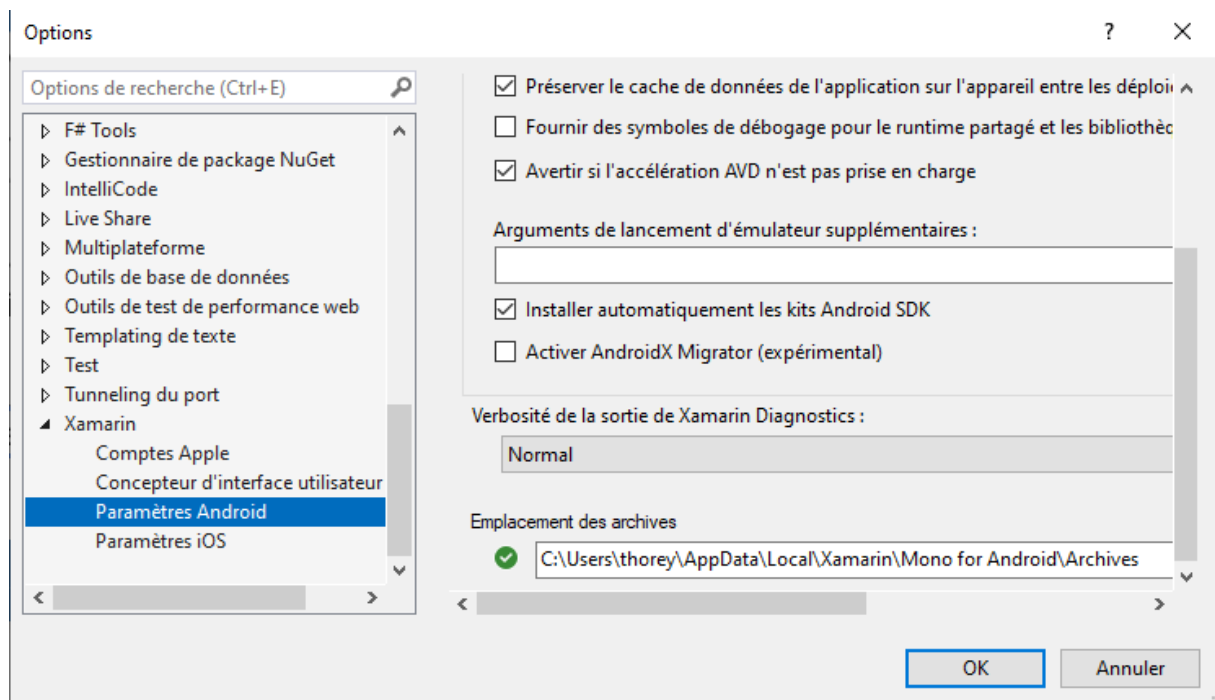
Une fois Visual Studio installé, il faut y intégrer l'environnement Xamarin. Pour cela, il faut lancer Visual Studio Installer et Modifier l'installation de Visual Studio 2022.



Le package « Développement mobile en .Net Multi-Platform App UI » doit être sélectionné. Attention : il faut le droit administrateur



Le développement pour Android reposant sur le SDK Java et sur le SDK Android, il est possible de le configurer dans Visual Studio. Menu Outils -> Options -> Xamarin -> Paramètres Android.



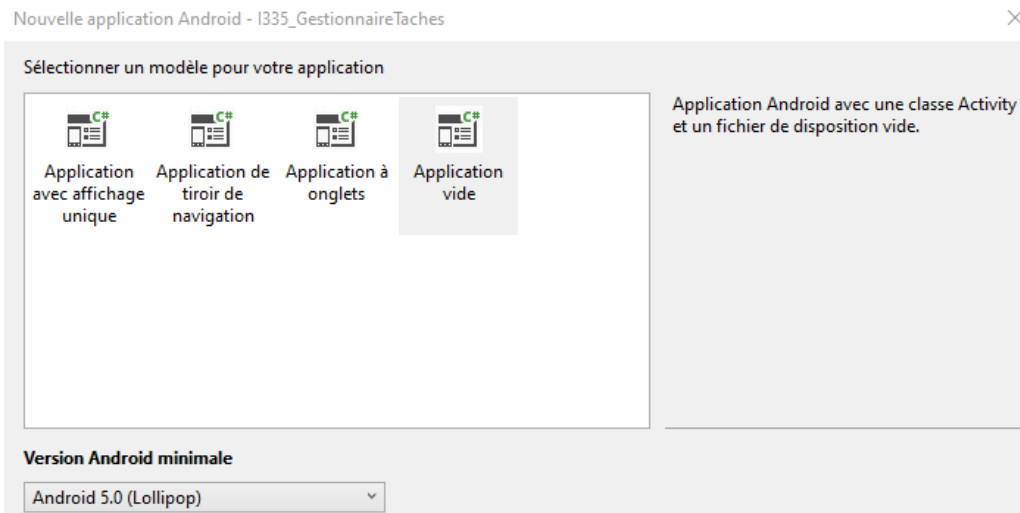
Après que le premier projet est créé, il faut aussi

## 5 Création du projet

Lors de la création d'un projet, une étape supplémentaire sera ajoutée comparé à la création d'un projet normal C#.

Il s'agit d'un menu de sélection du modèle à utiliser, dans notre cas nous choisirons « Application vide » pour répondre à la contrainte de l'explication de la création d'une première Activité.

Choisir la version d'Android minimale. Pour cette partie, plus nous choisissons une récente, plus nous aurons des fonctionnalités récentes à disposition. La solution semble être de choisir la version la plus récente. Cependant cela aura comme conséquence que notre application ne pourra tourner que sur une portion minimale des supports Android existants sur le marché. En effet peu de personnes ont un support Android de dernière génération. Il faut donc choisir la version selon le public cible.

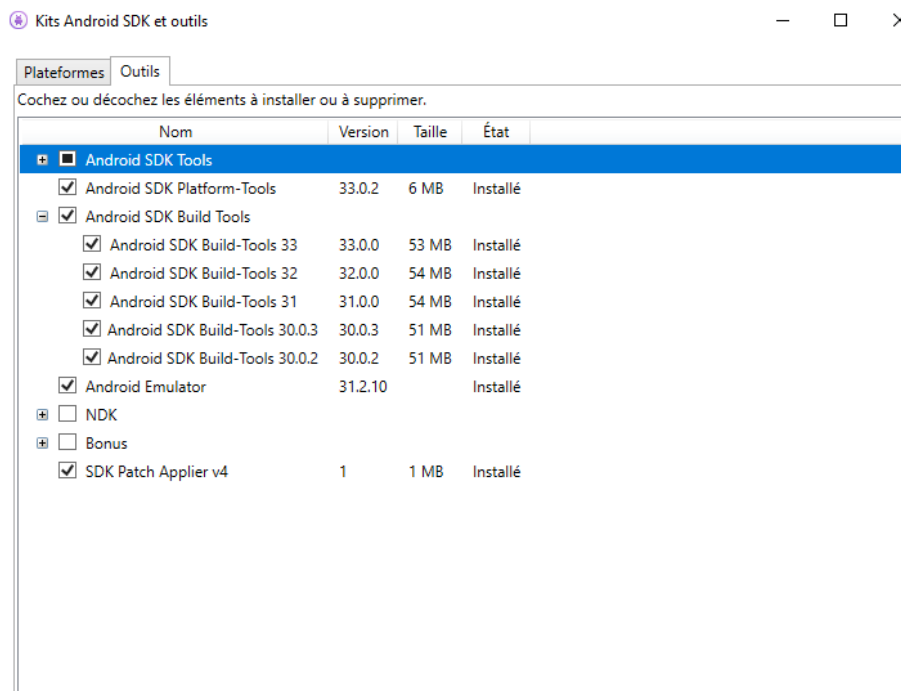



## 6 Emulateur

Avant de regarder l'environnement, il faut configurer l'émulateur pour qu'il puisse faire tourner notre application. Un émulateur est un outil qui permet de simuler un outil informatique comme un smartphone ou une console de jeux sur ordinateur. On peut aussi utiliser un smartphone physique.

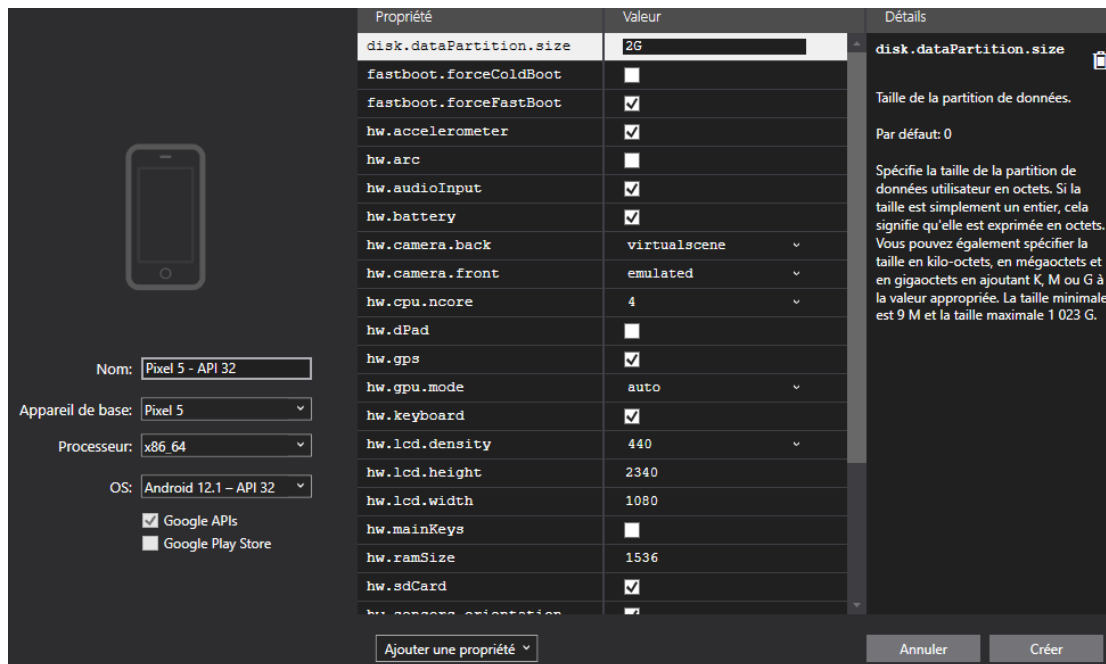
Ces deux méthodes ont leurs avantages et leurs inconvénients. Le support physique nous permet de mieux se rendre compte de l'utilisateur du sensor, s'il est trop sensible ou non. Alors que l'émulateur lui nous offre une multitude de version, de taille ou d'appareille pour notre application. Il est donc conseiller d'utiliser les deux méthodes.

Pour utiliser l'émulateur, il faut aller dans Outils->Android->Gestionnaire SDK Android... Puis aller dans Outils de la fenêtre qui vient de s'ouvrir et cocher « Android SDK Build Tools » et « Android Emulator ». Puis appliquer les modifications.



Une fois fait, il faut créer notre émulateur. Pour cela, on clique sur le bouton  ou aller dans Outils->Androide-> Gestionnaire d'appareils Android

Après cela, on clique sur « Nouveauté » et on décide de ses propriétés.



Propriété	Valeur
disk.dataPartition.size	2G
fastboot.forceColdBoot	<input type="checkbox"/>
fastboot.forceFastBoot	<input checked="" type="checkbox"/>
hw.accelerometer	<input checked="" type="checkbox"/>
hw.arc	<input type="checkbox"/>
hw.audioInput	<input checked="" type="checkbox"/>
hw.battery	<input checked="" type="checkbox"/>
hw.camera.back	virtualscene
hw.camera.front	emulated
hw.cpu.ncore	4
hw.dPad	<input type="checkbox"/>
hw.gps	<input checked="" type="checkbox"/>
hw.gpu.mode	auto
hw.keyboard	<input checked="" type="checkbox"/>
hw.lcd.density	440
hw.lcd.height	2340
hw.lcd.width	1080
hw.mainKeys	<input type="checkbox"/>
hw.ramSize	1536
hw.sdCard	<input checked="" type="checkbox"/>
hw.sensors.orientation	<input checked="" type="checkbox"/>

Détails

disk.dataPartition.size

Taille de la partition de données.

Par défaut: 0

Spécifie la taille de la partition de données utilisateur en octets. Si la taille est simplement un entier, cela signifie qu'elle est exprimée en octets. Vous pouvez également spécifier la taille en kilo-octets, en mégaoctets et en gigaoctets en ajoutant K, M ou G à la valeur appropriée. La taille minimale est 9 M et la taille maximale 1 023 G.

Annuler Créer

Et un fois cela fait, il ne manque plus que de le créer et de la démarrer.

## 7 Environnement et première Activité

En créant le projet, la première activité (vide) est créée. Celle-ci est composée de deux fichiers principaux :

- MainActivity.cs
- Activity\_main.xml

MainActivity.cs contient le code C# et les méthodes événementielles. Il s'agit de la partie C# de la première activité. C'est aussi l'activité principale puisque c'est celle qui est lancée à l'ouverture de l'application.

Activity\_main.xml contient les balises xml qui permettent de créer l'interface utilisateur qui sera affichée. C'est celle qui va contenir notre interface « ma journée ».

Une activité fonctionne pour elle-même, c'est-à-dire que quand on ouvre une activité, toutes les autres se ferment.

### 7.1 Les fenêtres de l'environnement

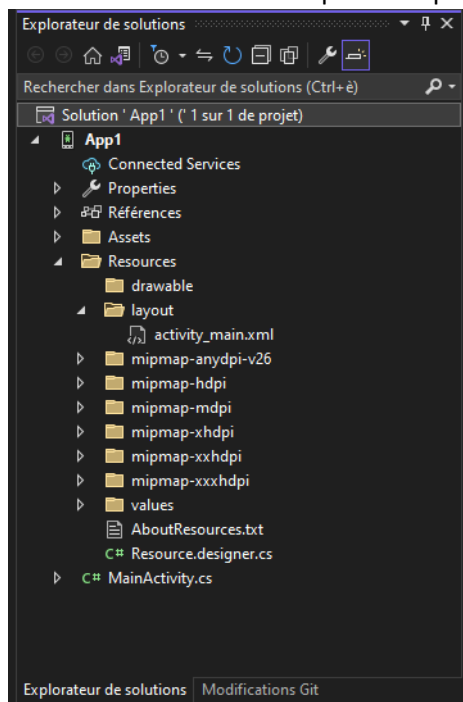
L'environnement s'ouvre avec différentes fenêtres à disposition.

La partie centrale contient la fenêtre de développement. C'est d'ailleurs « MainActivity.cs » qui est affiché et on peut voir que la liaison entre le MainActivity.cs et l'Activity\_main.xml grâce à ce bout de code :

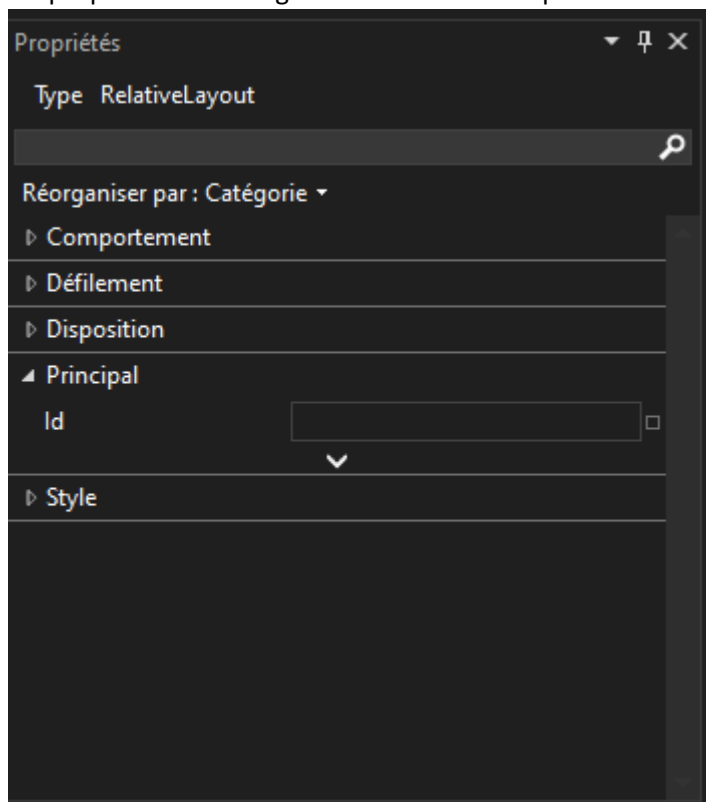
```
SetContentView(Resource.Layout.activity_main);
```



A droite, nous trouvons « l'explorateur de solutions » qui nous permettra de sélectionner les différents fichiers comme par exemple, le code C#, les balises xml ou les fichiers de ressources.



On peut trouver enfin la fenêtre des « propriétés ». Comme pour la fenêtre de la « Boîte à outils », les propriétés des Widgets s'afficheront lorsque nous sommes sur la partie interface utilisateur.

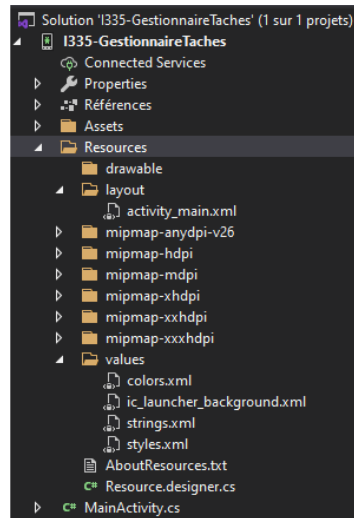


En somme, l'environnement ressemble à celui d'un projet Windows Form.

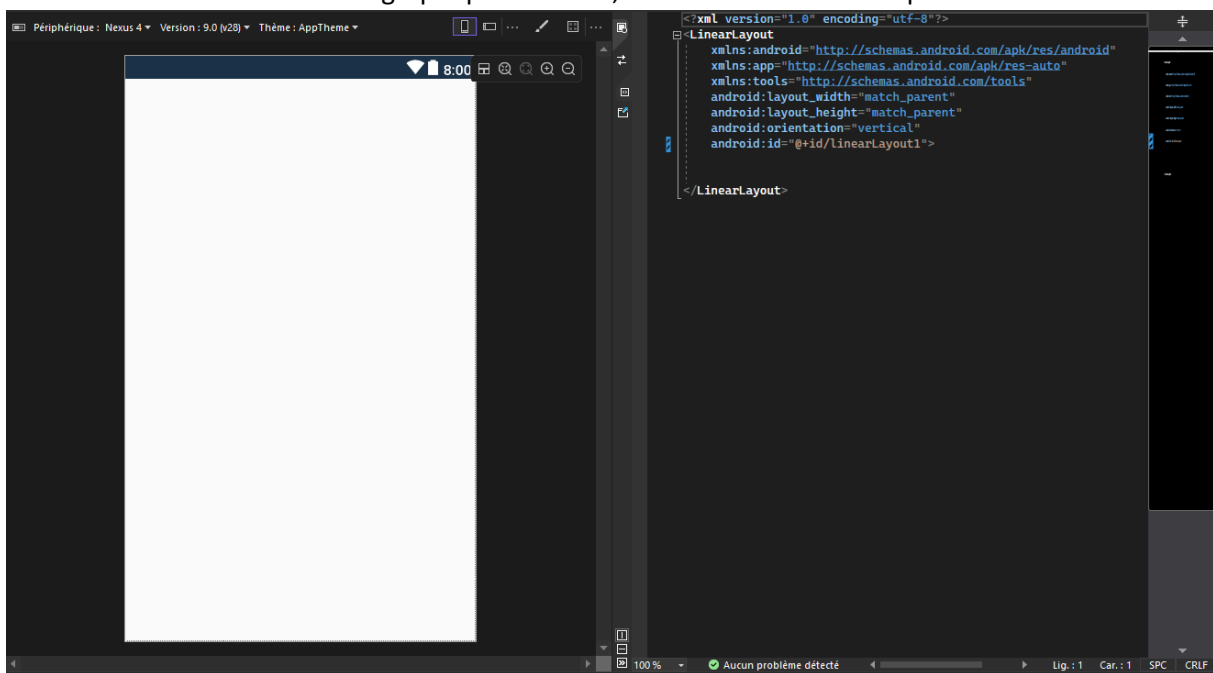


## 7.2 Interface utilisateur

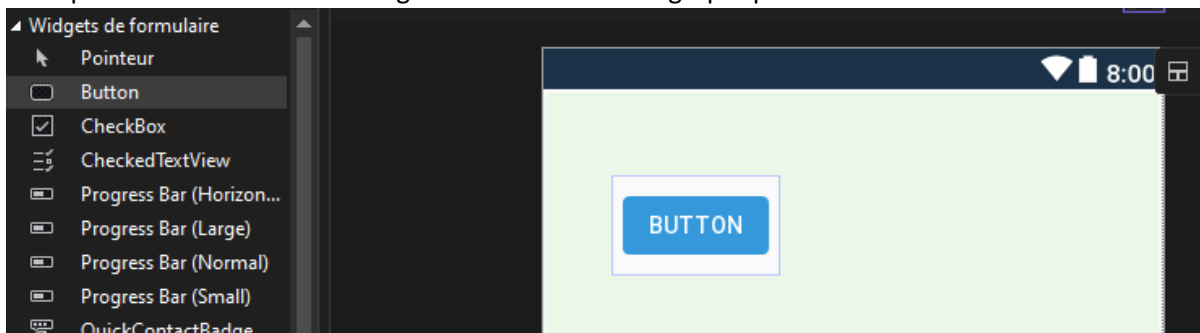
Pour mettre en place l'interface utilisateur, il faut ouvrir le fichier xml correspondant. Pour cette première Activité, il s'agit du fichier « Activity\_main.xml ». Pour ceci. Il faut aller dans « l'explorateur de solutions » et aller dans le répertoire « Resources » et « Layout ».



Lorsque le fichier est sélectionné, la fenêtre correspondante est partagée en deux. A gauche se trouve une fenêtre de création graphique. A droite, nous trouvons le correspondant en balises xml.



Il est possible de mettre des Widgets soit sur la fenêtre graphique soit sur la fenêtre balise.



Une fois cela fait, du code xml est ajouter avec des paramètres par défauts et l'élément déposer est afficher sur la partie graphique



### 7.3 Quelques widgets

Le paramètre text est, comme vous pouvez l'imaginer, le texte contenu dans le widget. Cependant, il n'est pas trop conseillé d'écrire le texte en dur (comme pour l'id). Il vaut mieux utiliser le fichier strings.xml du dossier Resources. On vous expliquera dans le chapitre suivants.

Le paramètre layout\_width/height permet d'initialiser la largeur/hauteur du widget. match\_parent permet de donner la valeur maximum qu'il peut prendre, c'est-à-dire la taille du widget parent. Le wrap\_content se contente de regarder le contenu du widget et de donner une taille minime.

Id permet de donner un « nom » au widget. C'est ce qui permet de le récupérer dans le côté CS.

Le backgroundTint permet de donner une couleur au fond du widget.

textColor permet de changer le couleur du texte du widget.

Visibility change la visibilité entre : visible, invisible et gone => le widget ne prend pas de place et est invisible.

### 7.4 Layouts

Si l'on regarde du côté des balises, on peut voir que le premier Widget est un Relative Layout.

Les layouts sont des boîtes qui permettent de mettre en communs plusieurs Widgets et les traiter ensemble.

Il existe différents types de Layout, chacun ayant ses spécificités.

#### 7.4.1 LinearLayout

Ce type de mise en page permet d'organiser une liste d'éléments de façon horizontale ou verticale. Chaque nouvel élément venant automatiquement se placer en dessous ou à droite de l'élément précédent.



#### 7.4.2 RelativeLayout

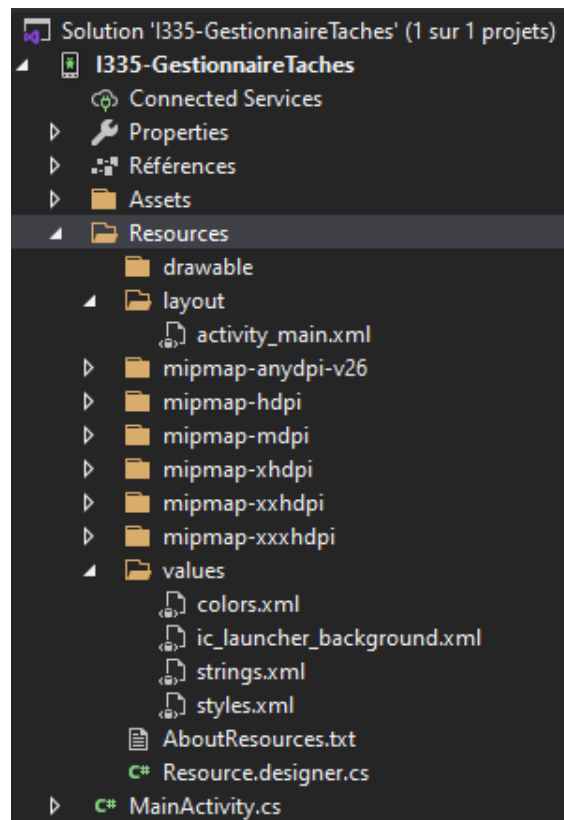
Ce type de mise en page permet d'organiser les éléments les uns en fonction des autres, donc de façon relative. Il est ainsi possible de spécifier qu'un champ de texte soit au-dessus d'un champ de saisie ou qu'une image soit centrée dans son composant parent.

#### 7.4.3 GridLayout

## 8 Fichier de ressources

Pour afficher les différents fichiers de références, il faut aller dans « l'explorateur de solutions » et aller dans le répertoire « Resources ».

Il y a trois dossiers importants dans le répertoire « Resources » : drawable, layout et values.



Le dossier « drawable » permet d'afficher des éléments qu'on ne peut pas reproduire dans les Layout, comme par exemple les images.

Le dossier « layout » est l'endroit où sera contenu tous les fichier xml qui serviront à l'affichage de l'interface.

Le dossier « values » quant à lui, sert à stocker toutes les données (constante) que nous allons utiliser pour les layouts ou la persistance des données. Les strings seront stockés dans strings.xml, les couleurs dans colors.xml et ainsi de suite. Pour faire appel à ces valeurs, il suffit de commencé par "@leNomDuFichier/leNomDeLaValeurs".

Exemple de constante : `<string name="app_name">X_335_ThomasRey_Projet</string>`

Exemple d'appelle : `@string/app_name`

Les fichiers ressources sont très pratique car il permet de les utiliser pour changer la langues ou montrer un affichage selon la taille de l'écran.

## 9 Cycle de vie

Une activité possède ce qu'on appelle un état (qui sont appliqué à partir de méthode) et il y en a plusieurs.

### 9.1 État

#### 9.1.1 Active / Running (Activée)

C'est l'état quand l'activité 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.

#### 9.1.2 Paused (En pause)

Il s'agit de l'état quand l'activité est partiellement masquée (ex : une autre activité transparente ou qui prend une partie de l'écran) ou quand le smartphone se met en veille. Dans cet état, l'os ne déchargera pas les ressources.

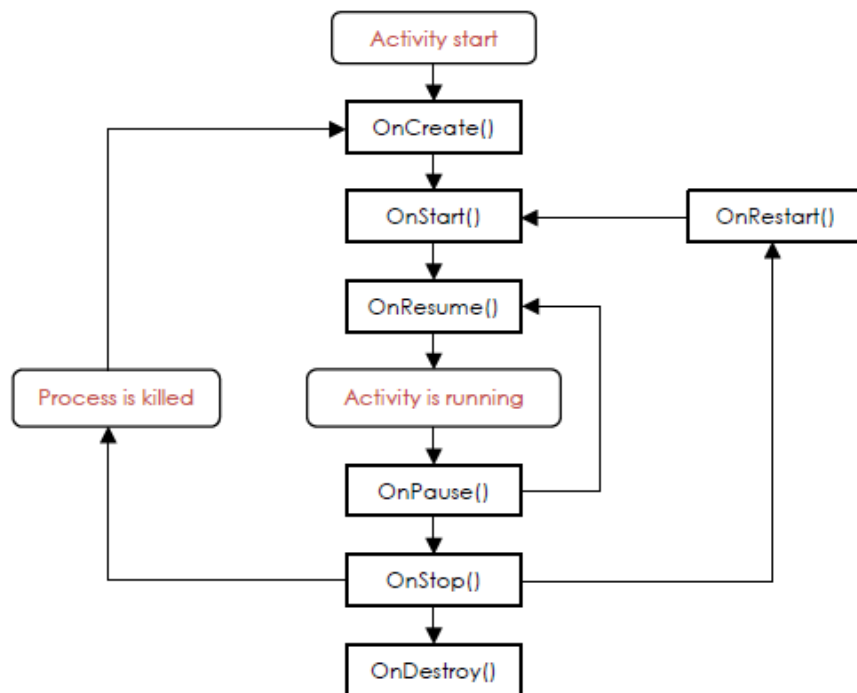
#### 9.1.3 Stopped (Arrêtée) / Backgrounded (En arrière-plan)

L'activité entre dans cet état quand elle est complètement masquée. À ce moment, deux possibilités se présente pour les ressources. S'ils n'entravent en rien le fonctionnement des activités activées et en pauses, l'OS va les maintenir. Sinon, ils seront libérés pour laisser de la place.

#### 9.1.4 Restarted (Redémarrée)

Quand l'activité a été effacée de la mémoire. L'OS considère qu'il doit recharger l'activité, si possible à partir de son précédent état, puis affichée à l'écran. Cet état est souvent utilisé lorsque l'utilisateur appuie sur le bouton Back.

## 9.2 Méthode



### 9.2.1 OnCreate

Il s'agit de la première méthode à être appelée lors de la création d'une activité. Elle 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);
}
```

### 9.2.2 OnStart

Il s'agit de la méthode qui fait référence à l'état **Active**. Elle est appelée lorsque l'activité démarre et devient visible à l'utilisateur.

```
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 !");
    };
}
```

### 9.2.3 OnResume

Après que la vue est affichée et que l'utilisateur peut interagir avec elle, la méthode OnResume est appelée. Il s'agit du meilleur moment pour initialiser les périphériques de l'appareil (GPS, camera, gyroscope...) ou d'afficher une animation. Si une activité redevient visible alors qu'elle était en pause, la méthode OnResume est à nouveau appelée.

#### 9.2.4 onPause

Comme pour son état du même nom (**Paused**), cette méthode permet de laisser le temps, si l'activité est partiellement masquée ou sur le point de passer en arrière-plan, 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.

#### 9.2.5 onStop

Cette méthode est appelée quand il faut faire passer l'activité en arrière-plan (parce qu'elle est remplacée ou entièrement masquée par un autre), On peut effectuer un nettoyage de la mémoire quand elle est appelée mais elle n'est pas forcément déclenchée. Si l'application manque de mémoire, elle peut détruire, de façon assez sévère, une activité en arrière-plan. Dans ces cas-là, elle ne passera pas par onStop mais par onDestroy. Elle fait référence à l'état **Stopped**.

#### 9.2.6 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. Une fois fini, elle appellera la méthode onStart.

#### 9.2.7 onDestroy

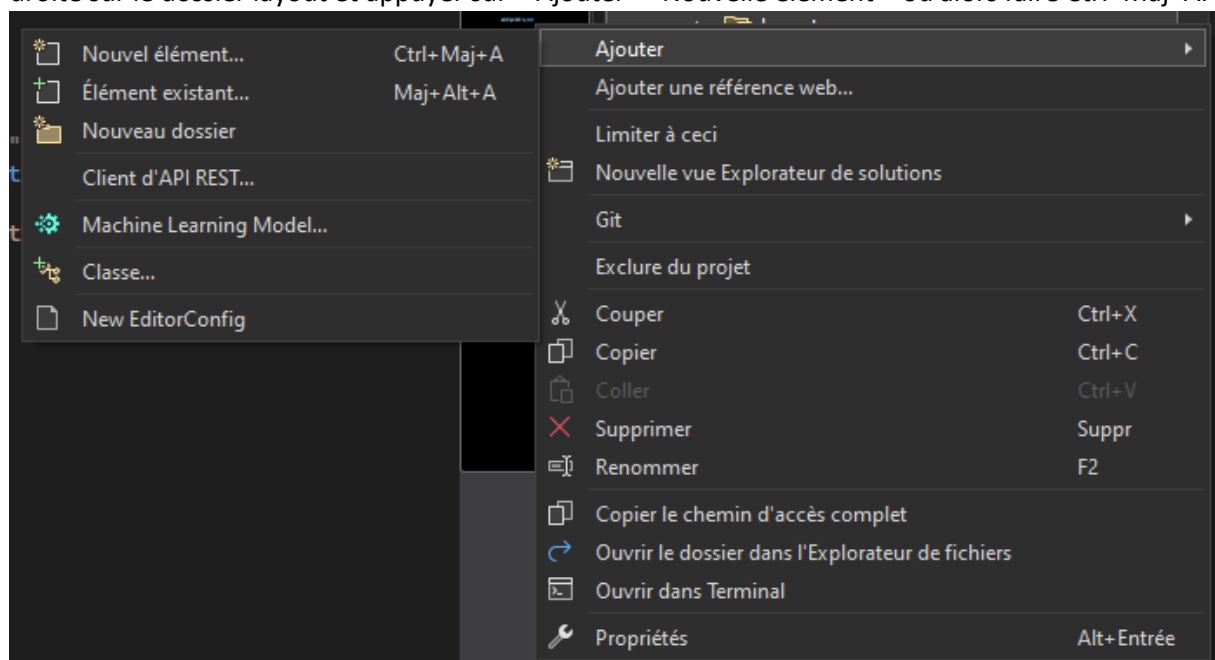
Cette méthode permet de supprimer l'activité de la mémoire. C'est au développeur de nettoyer tout ce qui pourrait rester en mémoire ou d'arrêter les tâches en arrière-plan.

## 10 Lien entre deux Activités

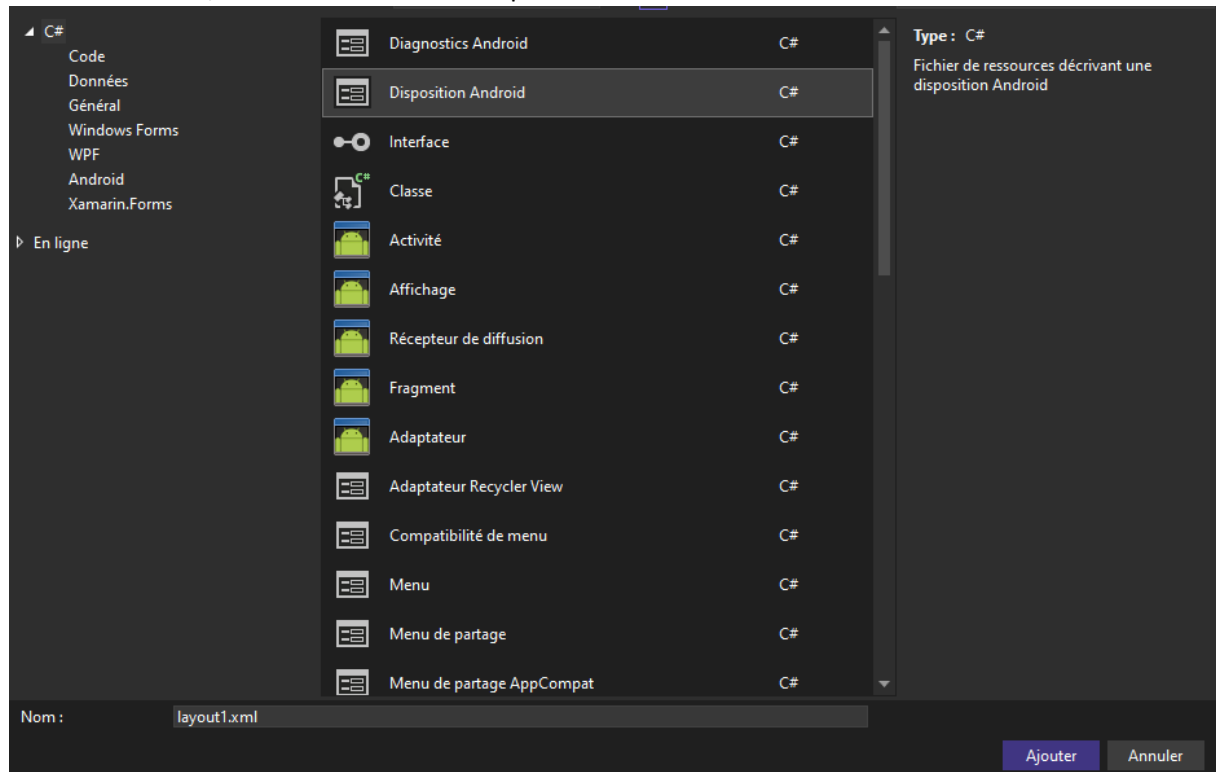
Pour créer un lien entre deux activités, il faut déjà en avoir deux, or nous en possédant qu'une qui est « Activity\_main.xml ». Il nous faut donc créer la deuxième.

### 10.1 Création d'une deuxième Activité

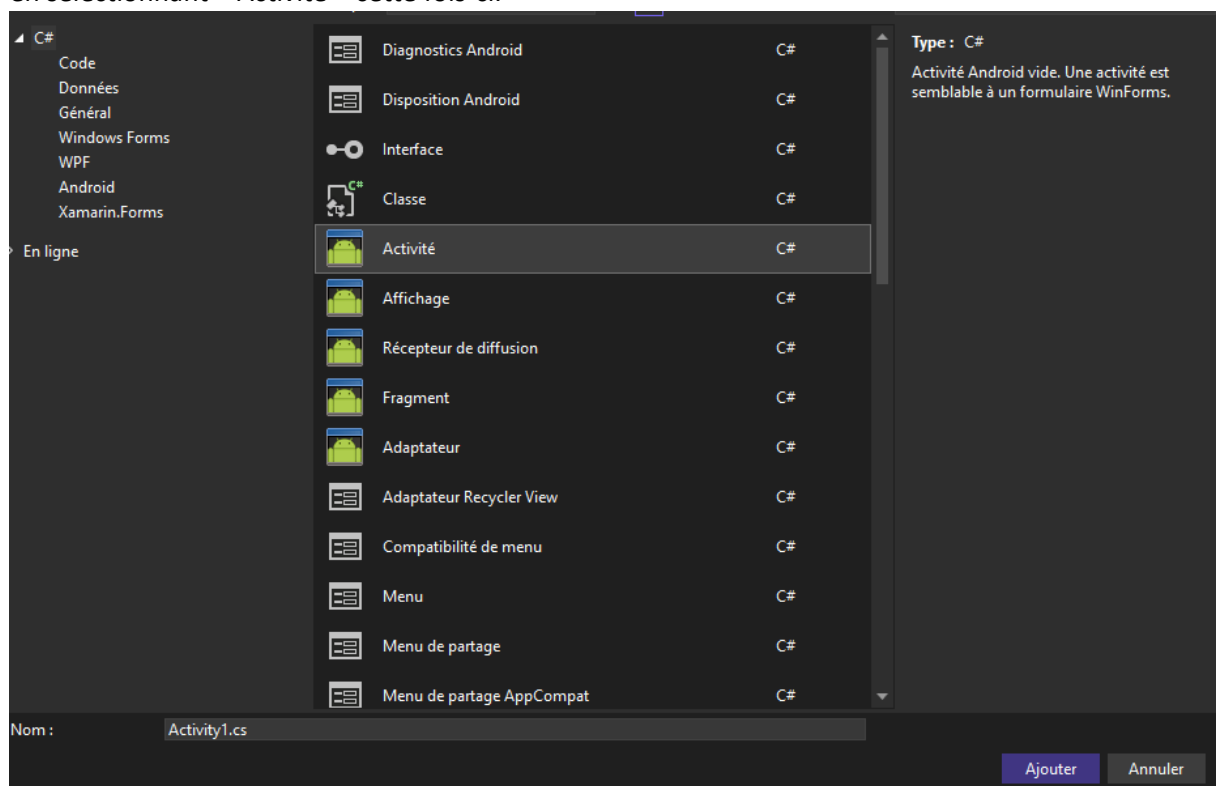
Pour Créer une nouvelle Activité, il faut aller dans l'Explorateur de solution puis faire une clique droite sur le dossier layout et appuyer sur « Ajouter -> Nouvelle élément » ou alors faire Ctrl+Maj+A.



Une fois cela fait, il faut sélectionner « Disposition Android » et nommé notre nouvelle Activité.



Une fois le fichier .xml créer, il faut faire la même chose mais sur la solution pour créer un fichier .cs en sélectionnant « Activité » cette fois-ci.



Le fichier .cs contiendra ceci :

```
[Activity(Label = "Activity1")]
0 références
public class Activity1 : Activity
{
    0 références
    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);

        // Create your application here
    }
}
```

Il ne manque plus qu'as lié le fichier .xml au .cs en ajoutant ce bout de code:

```
Xamarin.Essentials.Platform.Init(this, savedInstanceState);
SetContentView(Resource.Layout.activity_task);
```

## 10.2 Lié les Activités

Il y a deux manières pour lier des Activités entres elles. La première est la plus simple, il suffit d'appeler la méthode « StartActivity(typeof(MonActivité.cs)) ». De cette manière l'Activité appelée sera afficher sans problème :

```
StartActivity(typeof(AddTaskActivity));
```

La deuxième manière est plus longue mais permet de faire passer des données entre les Activités.

Il suffit de mettre dans une variable « Intent » l'Activité à afficher est les données à passer

```
Intent addActivity = new Intent(this, typeof(AddTaskActivity));
addActivity.PutExtra("Data", "Some data from MainActivity");
addActivity.PutExtra("Age", 30);
addActivity.PutStringArrayListExtra("Names",
new List<string> { "Jerôme", "Michel", "Paul" });
StartActivity(addActivity);
```

Une fois les données passées, il faut les récupérer dans la seconde Activité avec un TextView ou autre élément.

```
TextView dataTextView = FindViewById<TextView>(Resource.Id.textView1);
string data = Intent.GetStringExtra("Data");
string data2 = Intent.GetStringExtra("Age");
if (!string.IsNullOrEmpty(data))
{
    dataTextView.Text = data;
}
if (!string.IsNullOrEmpty(data2))
{
    dataTextView.Text = data;
}

Button myButton2 = new Button(this)
{
    Text = data2
};
```



## 11 Méthodes événementielles

Les méthodes événementielles sont des méthodes qui sont appelées quand l'utilisateur fait une certaine action dans l'interface. Par exemple quand on appuie sur un bouton.

Pour cela il faut créer une méthode avec des paramètres « object sender » et « EventArgs e » puis mettre le code que vous voulez dans celle-ci.


```
2 références
private void BackToMenu(object sender, EventArgs e)
{
    StartActivity(typeof(MainActivity));
}
```

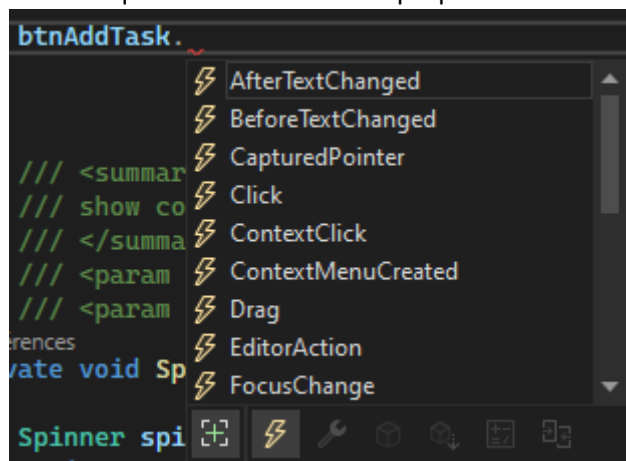
Puis lier cette méthode à un événement d'un élément.

```
Button btnAddTask = (Button)FindViewById(Resource.Id.btnAddTask);

btnAddTask.Click += BackToMenu;
```

Si on ne sait pas quel événement sélectionner il faut prendre pour une action, on peut appuyer sur

 lorsque VS nous donne des propositions.



## 12 Layout dynamique

Le layout dynamique consiste d'ajouter des éléments dans le xml depuis le code. Pour créer un nouvel élément il suffit de mettre la variable, son nom et ses propriétés.

```
Button myButton2 = new Button(this)
{
    Text = data2,
    Visibility = ViewStates.Invisible,
    Right = 60,
    ScaleX = 20
};
```

Le bouton sera ajouté automatiquement au chargement de l'application.

si on veut ajouter une propriété après avoir instancier l'élément, il suffit de faire comme ceci :

```
myButton2.ScaleY = 15;
```

Autre exemple :

```
LinearLayout layout = new LinearLayout(this)
{
    ScaleX = 80,
    ScaleY = 40,
    Enabled = false
};
layout.Top = 50;
```

Cependant, il faut faire attention. Certains paramètres sont des appels de méthode. C'est-à-dire qu'on ne pourra pas l'instancier ce paramètre pendant la création de l'élément mais après.

```
LinearLayout layout = new LinearLayout(this)
{
    ScaleX = 80,
    ScaleY = 40,
    Enabled = false,
    BackgroundColor
};
layout.Top = 50;
layout.SetBackgroundColor(Color.White);
```

Il ne manque plus qu'à afficher l'élément dans l'activité. Et pour ça, rien de plus simple. Déjà il faut récupérer l'activité en question.

```
LinearLayout mainLayout = FindViewById<LinearLayout>(Resource.Id.mainLayout);
```

Puis de lui passer l'élément avec : `mainLayout.AddView(myButton2);`

## 13 ListView

Les ListView sont des listes qui permettent d'afficher des données (Qu'elles soient persistantes ou non) selon ce que vous souhaitez afficher. Pour cela, nous aurons besoin d'un objet nommé « Adapteur ».

### 13.1 ListView de base

Cet adaptateur sera créé à partir de la méthode « CreateFromResource » qui possède comme paramètre : **l'environnement de l'application** => this, **l'identifiant du tableau à utiliser comme source de donnée** => Resource.Array.planets\_array (ici les données sont en dur) et **l'identifiant du Layout utilisé pour créer la view** => Android.Resource.Layout.SimpleSpinnerItem. C'est là qu'il y a des rendus de ListView pré-fait.

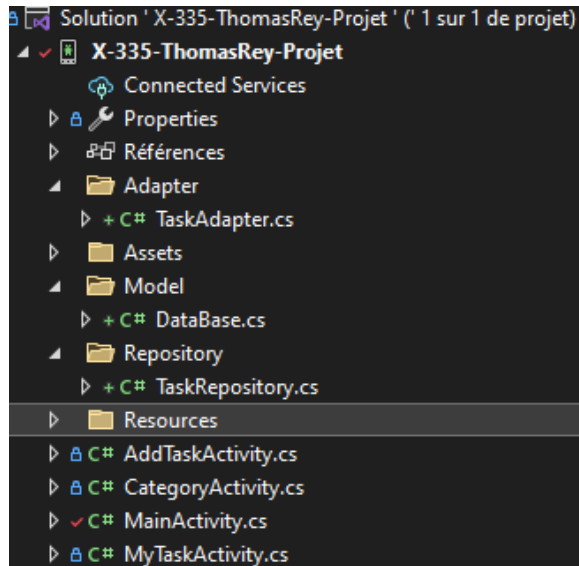
Une fois créé, il faut définir la ressource du Layout pour créer la view déroulante via la méthode SetDropDownViewResource. Elle aura comme un paramètre qui définit le dit Layout (Android.Resource.Layout.SimpleSpinnerItem).

```
//Adapter
var adapter1 = ArrayAdapter.CreateFromResource(this, Resource.Array.planets_array, Android.Resource.Layout.SimpleSpinnerItem);
adapter1.SetDropDownViewResource(Android.Resource.Layout.SimpleSpinnerDropDownItem);
listToDo.Adapter = adapter1;
```

Une fois l'adaptateur créé et initialisé, il suffit de l'assigner à la ListView (maListView.Adapter = monAdapteur).

## 13.2 ListView personnalisé

Il est aussi possible de créer son propre adaptateur pour afficher les données qu'on veut afficher. Pour cela on va d'abord devoir créer des dossiers pour se retrouver dans nos fichiers. L'image si dessous nous montre une possibilité de trier nos dossiers. Car pour chaque donnée, il faut un nouveau fichier. Ex : TaskAdapter.cs et CategoryAdapter.cs, TaskRepository.cs et CategoryRepository.cs, etc...



Le dossier adapter va contenir tous les adaptateurs différents.

Le dossier Repository (ou Service) va contenir toutes les méthodes qui vont interagir avec les données comme ajouter, supprimer, modifier une tâche, voir toutes les tâches, etc... On verra plus de détails au chapitre sur la persistance des données.

Le dossier Model va contenir toutes les données. À savoir que l'on peut créer un fichier pour chaque donnée ou créer un fichier appelé « DataBase.cs » et ajouter une nouvelle classe pour chaque donnée. À vous de voir. Les données dans les modèles seront contenues de cette manière.

```
10 références
public class Task
{
    1 référence
    public int Id { get; set; }
    2 références
    public string Name { get; set; }
    2 références
    public string Description { get; set; }
    3 références
    public Task (int id, string name, string description)
    {
        Id = id;
        Name = name;
        Description = description;
    }
}
```

Attention : Quand nous verrons la persistance de données, nous modifierons la manière de stocker les données.

Dans notre fichier TaskAdapter.cs, nous allons créer une classe qui va hériter d'une autre classe nommée « BaseAdapter<nomDuTypeDeDonnée> => BaseAdapter<Task> ». Cette classe doit contenir les variable List<> et Activity, un constructeur et de méthode prédéfinir.

Dans ce constructeur, on ajoute manuellement des tâches, mais cela devrait être une méthode à appeler dans le repository.

```
// Variable
List<Task> tasks;
Activity activity;

/// <summary>
/// Constructor: charge l'activité et la liste des tâche
/// </summary>
/// <param name="tasks"></param>
/// <param name="activity"></param>
2 références
public TaskAdapter(List<Task> tasks, Activity activity) : base()
{
    this.tasks = tasks;
    this.activity = activity;

    this.tasks.Add(new Task(1, "tache 1", "c'est la tache 1"));
    this.tasks.Add(new Task(2, "tache 2", "c'est la tache 2"));
    this.tasks.Add(new Task(3, "tache 3", "c'est la tache 3"));
}
```

Les méthodes si dessous, sont des méthodes obligatoires à ajouter. Ce sont des méthodes assez simples, la première permet d'obtenir l'id d'un item, dans ce contexte une tâche. Le deuxième nous retourne un item à l'aide d'un id. le troisième compte le nombre d'item dans la liste.

```
/// <summary>
/// retourne l'id d'une tâche
/// </summary>
/// <param name="position"></param>
/// <returns></returns>
0 références
public override long GetItemId(int position)
{
    return position;
}

/// <summary>
/// retourne une tâche
/// </summary>
/// <param name="position"></param>
/// <returns></returns>
0 références
public override Task this[int position]{
    get { return tasks[position]; }
}

/// <summary>
/// retourne le nombre de tâche
/// </summary>
0 références
public override int Count
{
    get { return tasks.Count; }
}
```

Le dernier est un peu spécial. Il s'agit de la même méthode que la deuxième mais avec du java.

```
/// <summary>
/// retourne une tâche
/// </summary>
/// <param name="position"></param>
/// <returns></returns>
0 références
public override Java.Lang.Object GetItem(int position)
{
    // Pour l'instant retourne rien
    return "";
}
```

Une fois ces méthodes obligatoires faites, il faut faire la méthode qui va permettre d'attribuer les données à la listview. Il suffit juste de choisir un modèle de Layout et de spécifier quelle valeur va où.

```
/// <summary>
/// Modèle de rendu des données
/// </summary>
/// <param name="position"></param>
/// <param name="convertView"></param>
/// <param name="parent"></param>
/// <returns></returns>
/// <exception cref="NotImplementedException"></exception>
0 références
public override View GetView(int position, View convertView, ViewGroup parent)
{
    var currentTask = tasks[position];

    View view = activity.LayoutInflater.Inflate(Android.Resource.Layout.SimpleListItem2, null);

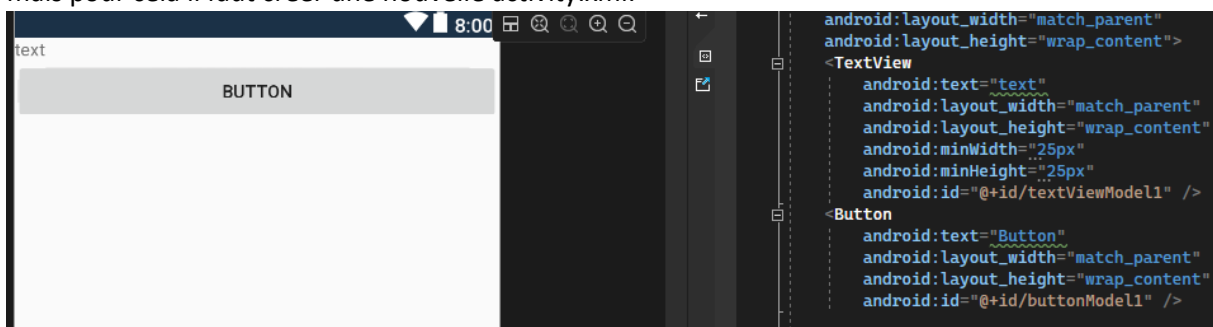
    view.FindViewById<TextView>(Android.Resource.Id.Text1).Text = currentTask.Name;
    view.FindViewById<TextView>(Android.Resource.Id.Text2).Text = currentTask.Description;

    return view;
}
```

Il est aussi possible de faire son propre modèle pour afficher les valeurs comme on le veut.

mais pour cela il faut créer une nouvelle activity.xml.

+ activity\_modelTask.xml



```
View view = activity.LayoutInflater.Inflate(Resource.Layout.activity_modelTask, null);

view.FindViewById<TextView>(Resource.Id.textViewModel1).Text = currentTask.Name;
view.FindViewById<Button>(Resource.Id.buttonModel1).Text = currentTask.Description;
```

Pour passer les données à la listView, il faut d'abord créer une liste avec les données voulues (Task) et la remplir avec.

```
List<Task> tasks = new List<Task>();

//Adapter
TaskAdapter adapter1 = new TaskAdapter(tasks, this);

listToDo.Adapter = adapter1;
```

## 14 Persistance de données

La persistance de données est l'équivalent de la base de données sur les smartphones. Pour faire une persistance de données, il faut reprendre les deux fichiers CS : DataBase.cs et TaskRepository.cs

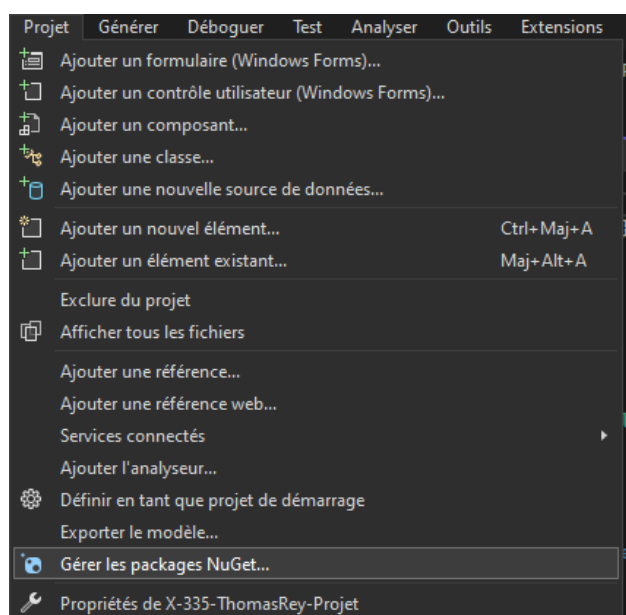
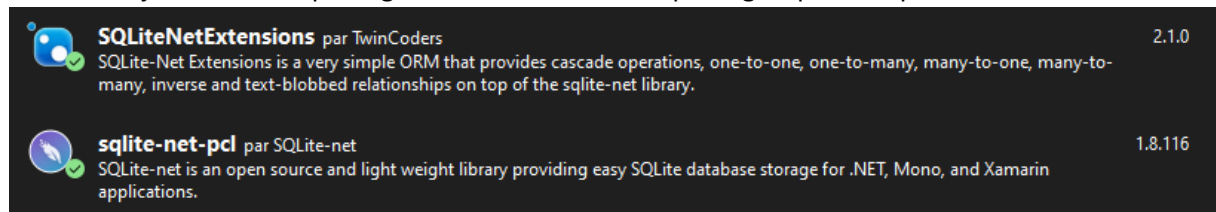
Pour les exemples qui vont suivre (et toujours en rapport avec notre projet), nous allons prendre les données pour les tâches.

### 14.1 Base de données

Comme expliqué précédemment, nous pouvons avoir plusieurs fichiers model pour chaque table ou avoir un seul fichier qui contient toutes les tables. Nous allons prendre la deuxième option mais il faut savoir que si on prend la première il faudra faire les modifications sur tous les fichiers.

Il y a plusieurs moyens de faire un DB, pour la suite nous allons utiliser SQLite car leur DB sont légers et ont très peu d'adhérence au système d'exploitation vu qu'il se base sur un système de fichiers. Cependant, les DB sont locales et ne permettent pas de créer une application partagée.

Pour utiliser SQLite, il faut installer un PCL (Portable Class Libraries) avec le gestionnaire de paquets NuGet. Projet->Gérer les packages NuGet et installer le package sqlite-net-pcl et SQLiteNetExtensions



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

Il faudra aussi utilisé les using suivant :

Le using SQLite permet d'utiliser ces mots clés : Table, PrimaryKey, AutoIncrément, Indexed, MaxLength, Unique et NotNull.

Le using SQLiteNetExtension permet de gérer les relations entre les tables.

Et le using Java.Sql permet d'ajouter des objets pour le SQL comme Date ou autre.

Pour créer un table, il faut ajouter en dessus du nom de la classe l'attribut Table(NomTable).

```
[Table("t_task")]
7 références
public class Task
{
```

Ensuite, il faut ajouter les différents champs que va contenir la table, il s'agit de simple variable que l'on modifie à l'aide d'attribut.

```
[PrimaryKey, AutoIncrement, NotNull, MaxLength(11), Unique]
1 référence
public int Id { get; set; }
[MaxLength(100)]
2 références
public string Name { get; set; }
[MaxLength(500)]
2 références
public string Description { get; set; }
1 référence
public Date DueDate { get; set; }
1 référence
public bool Daily { get; set; }
1 référence
public bool Done { get; set; }
```

Grace au using SQLiteNetExtentions, nous pouvons ajouter des clés étrangères.

```
[ForeignKey(typeof(Category))]
1 référence
public int CategoryId { get; set; }
[ManyToOne]
1 référence
public Category Category { get; set; }
```

## 14.2 Accès aux données

Pour manipuler ces données, il faut entrer dans le fichier repository qui nous permettra d'ajouter, supprimer, modifier une tâche et bien d'autre méthode encore.

Comme pour le fichier DataBase.cs, il faudra ajouter le `using SQLite;` au fichier repository. Car le fichier contiendra une variable SQLite spécifique à la plateforme. Comme pour chaque classe, il faut

commencer avec un constructeur.

```
1 référence
public class TaskRepository
{
    // Message de status
    0 références
    public string StatusMessage { get; set; }

    // Implémentation SQLite spécifique à la plateforme
    private SQLiteAsyncConnection _connection;

    0 références
    public TaskRepository(string dbPath) {
        // Récupération ou création de la connexion à la DB
        _connection = new SQLiteAsyncConnection(dbPath);

        // Création de la table Task
        _connection.CreateTableAsync<Task>();
    }
}
```

Pour faire une manipulation de la DB, la méthode doit être async. Il doit aussi contenir un try/catch car il arrive qu'il y ait des erreurs. Dans c'est cas-là, il faut renvoyer un message par le biais de la variable StatusMessage qu'on a implémenter juste en dessus.

```
// Ajout d'une tâche
1 référence
public async System.Threading.Tasks.Task AddNewTaskAsync (string name, string description, bool daily = false)
{
    var result = 0;

    try
    {
        // ajout un utilisateur
        result = await _connection.InsertAsync(new Task { Name = name, Description = description, Daily = daily });
        StatusMessage = $"{result} tâche ajouté : {name}";
    }
    catch (Exception ex)
    {
        // message en cas d'erreur
        StatusMessage = $"AddNewTask erreur {name}.\nErreur : {ex.Message}";
    }
}
```

Autre exemple :

```
// Récupération de tous les utilisateurs
0 références
public async Task<List<Task>> GetTasksAsync()
{
    try
    {
        // retourne la liste de toutes les tâches
        return await _connection.Table<Task>().ToListAsync();
    }
    catch (Exception ex)
    {
        // en cas d'erreur, message et retour d'une liste vide
        StatusMessage = $"GetTasks Impossible.\nErreur : {ex.Message}";
        return new List<Task>();
    }
}
```

Une fois cela fait, on peut utiliser le repository dans le code de l'activité que l'on veut.

```
// Instanciation du repository
TaskRepository taskRepository = new TaskRepository(_dbPath);
taskRepository.AddNewTaskAsync("ma tâche", "ma description");
```



Mais il faut d'abord préciser l'emplacement de la DB.

```
[Activity(Label = "AddTaskActivity")]
3 références
public class AddTaskActivity : Activity
{
    private string _dbPath = System.IO.Path.Combine(FileSystem.AppDataDirectory, "MaDB.db3");
}
```

## 15 Utilisation d'un capteur

### 15.1 Récupération des capteurs

Avant de commencer à utiliser les capteurs (sensor en anglais), il faut savoir qu'il y a différents capteurs, qu'il soit présent ou non sur l'appareil. Une application nécessite un capteur mais qu'il n'est pas disponible sur un appareil, ça posera un problème pour l'utilisateur. Il est donc préférable d'indiquer au Google Play Store que le smartphone doit disposer de ce capteur. Chaque capteur a sa propre manière d'utilisation et ses propres événements. Il est aussi possible d'avoir une liste qui est à notre disposition pour connaître les capteurs que possède le mobile.

Pour cela il faut utiliser un `SensorManager` qui va gérer tous les capteurs de notre appareil. Une fois le `SensorManager` créé, il faut ressortir les capteurs dans un `ICollection` de `Sensor`.

Attention : il ne faut pas oublier d'utiliser le `using Android.Hardware`

```
// Récupération d'un objet SensorManager
SensorManager sensorMgr = (SensorManager)SystemService(SensorService);

// Liste de tous les capteurs disponibles
ICollection<Sensor> sensors = sensorMgr.GetSensorList(SensorType.All);
```

Il suffit de l'afficher à l'aide d'un `foreach`.

```
foreach(Sensor sensor in sensors)
{
    textTmp.Text += $"{sensor.Name}";
}
```

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

### 15.2 Utilisation d'un capteur

Pour pouvoir utiliser un capteur ainsi que ses événements, il faut que la classe qui va l'utiliser le capteur hérite de l'interface `ISensorEventListener`.

```
public class MainActivity : AppCompatActivity, ISensorEventListener
```

Une erreur va alors se produire, il suffit de faire cliquer droit, « Actions rapides et refactorisations... » puis « Implémenter l'interface ». Cela créera une nouvelle méthode.

Pour choisir le capteur que l'on veut, il faut l'enregistrer avec la méthode `RegisterListener` du `SensorManager`. Pour notre exemple on va utiliser le capteur d'accéléromètre.

```
private SensorManager _sensorMgr;  
private Sensor _accelerometer;  
private float _xValue;
```

Il faut tout d'abord implémenter nos variables.

Puis on va instancier l'accéléromètre.

```
// Enregistrement de l'accéléromètre dans onCreate ou onResume  
_accelerometer = _sensorMgr.getDefaultSensor(SensorType.Accelerometer);  
_sensorMgr.RegisterListener(this, _accelerometer, SensorDelay.Normal);
```

Une fois cela fait, on va créer une méthode que l'on va appeler `OnSensorChanged` et qui va nous servir pour effectuer les tâches désirées (Exemple : détecter un mouvement horizontal).

```
/// <summary>  
/// détecte si l'utilisateur secoue le téléphone horizontalement  
/// </summary>  
/// <param name="e"></param>  
0 références  
public void OnSensorChanged(SensorEvent e)  
{  
    // Il s'agit de l'accéléromètre  
    if (e.Sensor.Equals(_accelerometer))  
    {  
        // 3 valeurs qui correspondent ici au 3 accélération en x, y et z  
        IList<float> valeurs = e.Values;  
  
        // Détecte un mouvement horizontal  
        if (Math.Abs(valeurs[0] - _xValue) > 5)  
        {  
            _textTmp.Text = "Effacer les éléments";  
        }  
        else  
        {  
            _xValue = valeurs[0];  
        }  
    }  
}
```

Quand l'activité est mise en pause, il ne faut pas oublier de désenregistrer le capteur.

```
0 références  
protected override void OnPause()  
{  
    base.OnPause();  
    // Libère l'accéléromètre  
    _sensorMgr.UnregisterListener(this);  
}
```