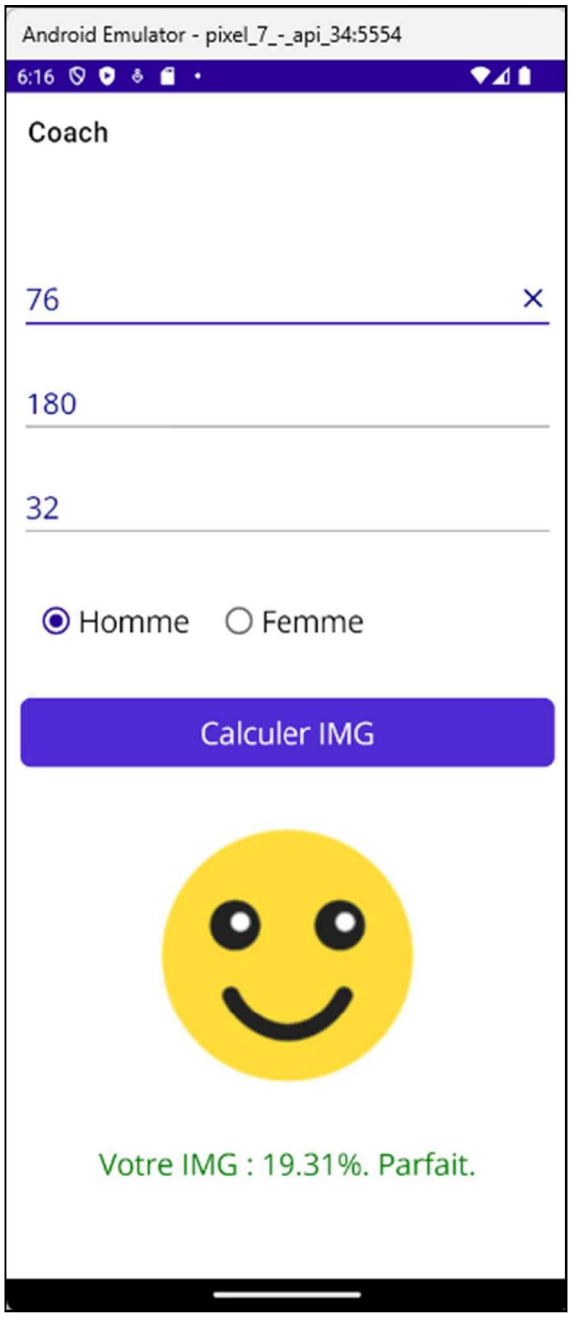
Application Android « Coach »

Dans ce TP, vous allez générer votre première application .NET MAUI (.NET Multi-platform App UI) dans Visual Studio et acquérir les fondements du développement d'applications Android.

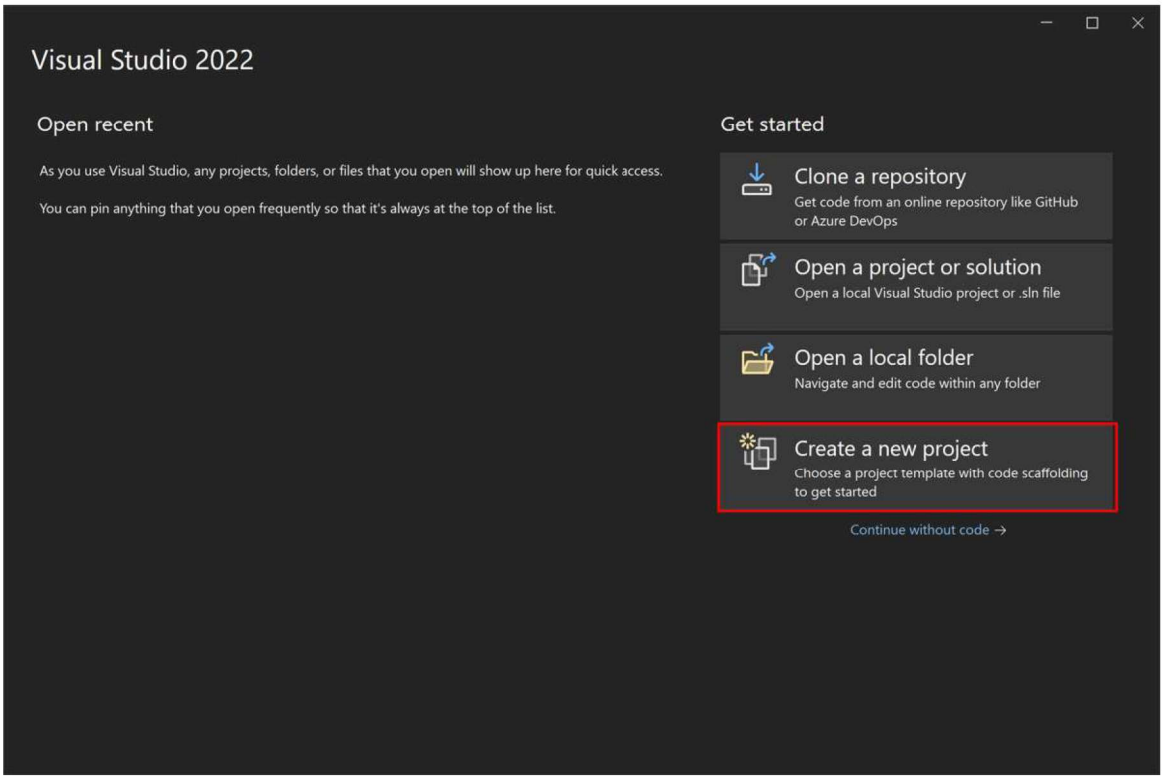
Vous allez créer une application qui permet de calculer et de mémoriser l'Indice de Masse Grasse (IMG) de l'utilisateur. L'application finale ressemble à ceci :



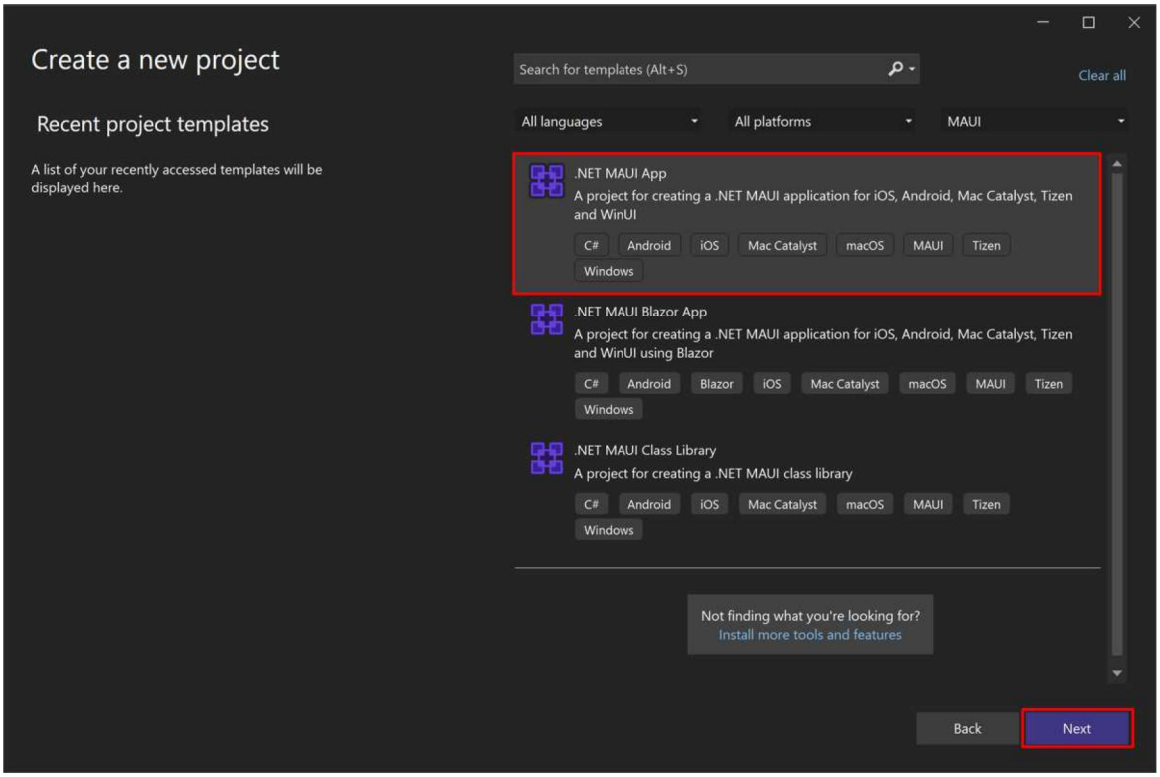
Créer votre première application mobile avec .NET MAUI

Dans cette étape, vous allez créer votre première application .NET MAUI dans Visual Studio et l'exécuter sur un émulateur Android :

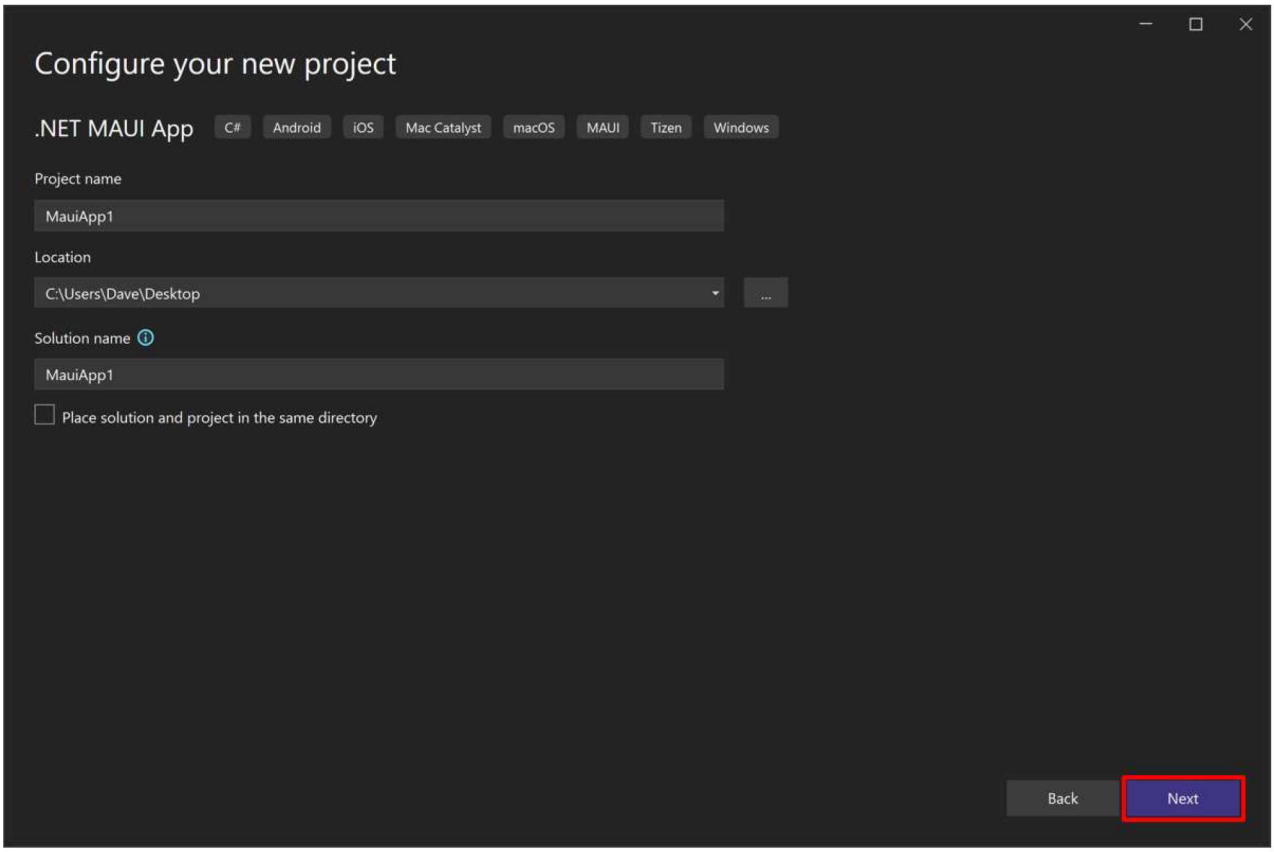
1. Lancez Visual Studio. Dans la fenêtre de démarrage, cliquez sur **Create a new project** pour créer un projet :



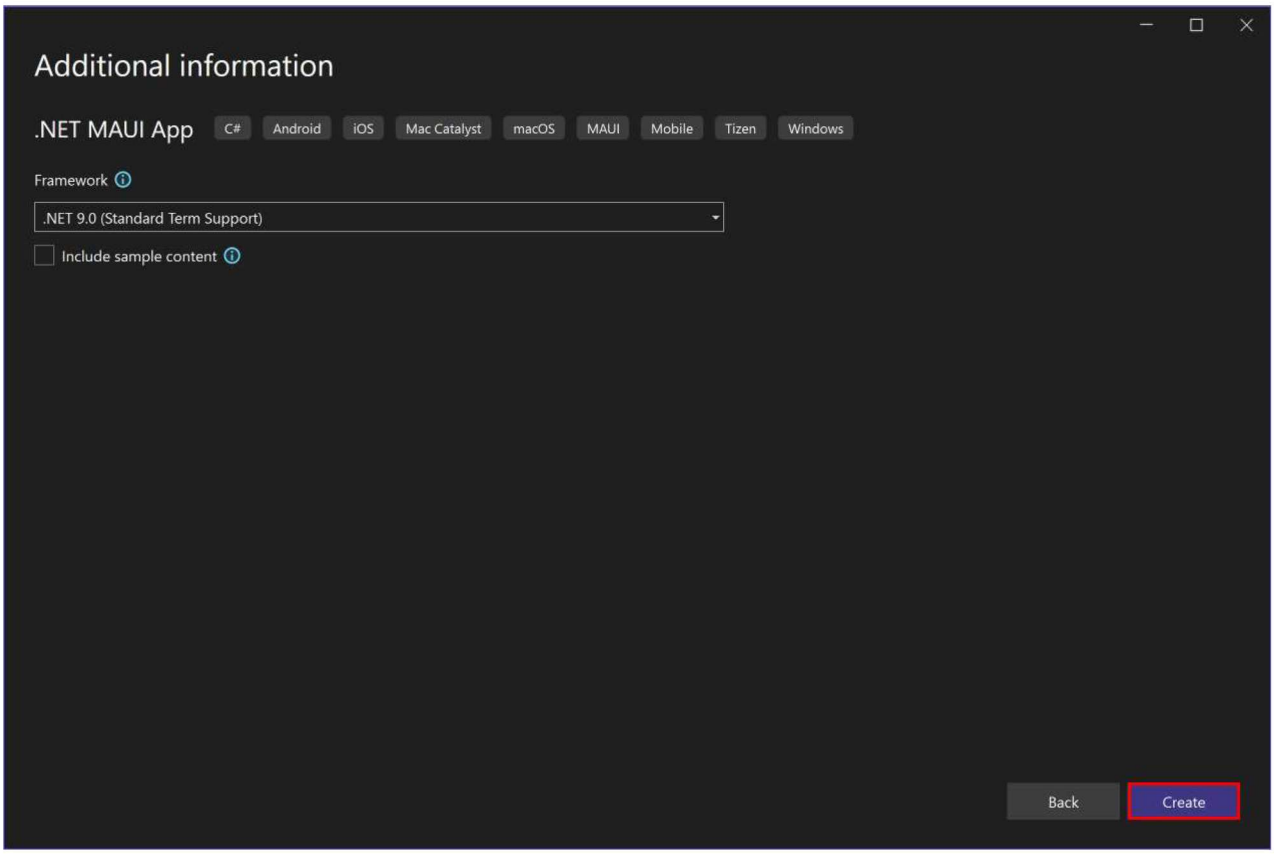
2. Dans la fenêtre **Create a new project**, sélectionnez **MAUI** dans la liste déroulante **All project types**, sélectionnez le modèle **.NET MAUI App**, puis cliquez sur le bouton **Next** :



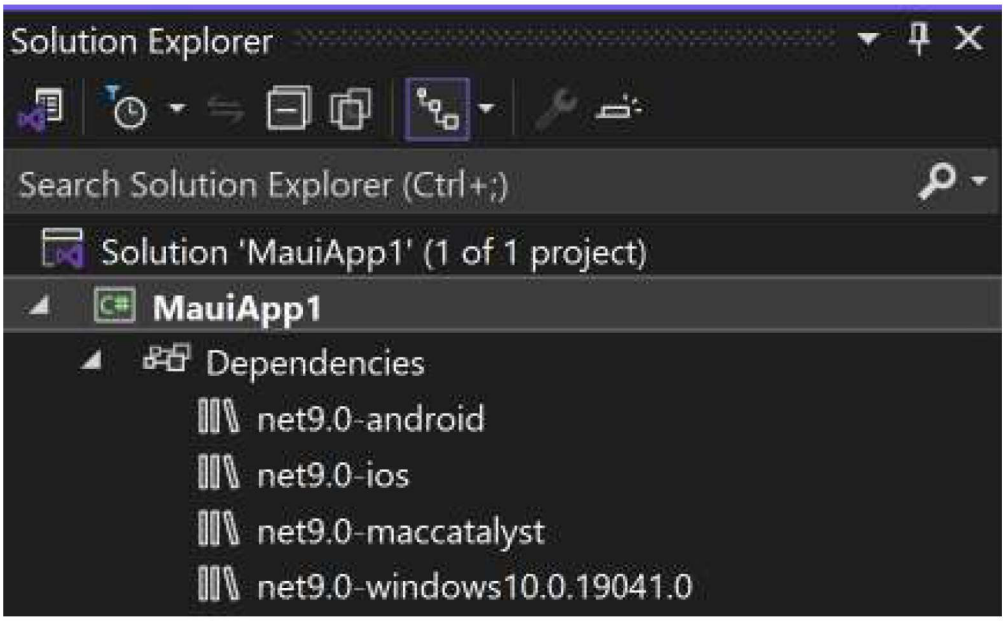
3. Dans la fenêtre **Configure your new project**, nommez votre projet « **MauiAppCoach** », choisissez un emplacement approprié et cliquez sur le bouton **Next** :



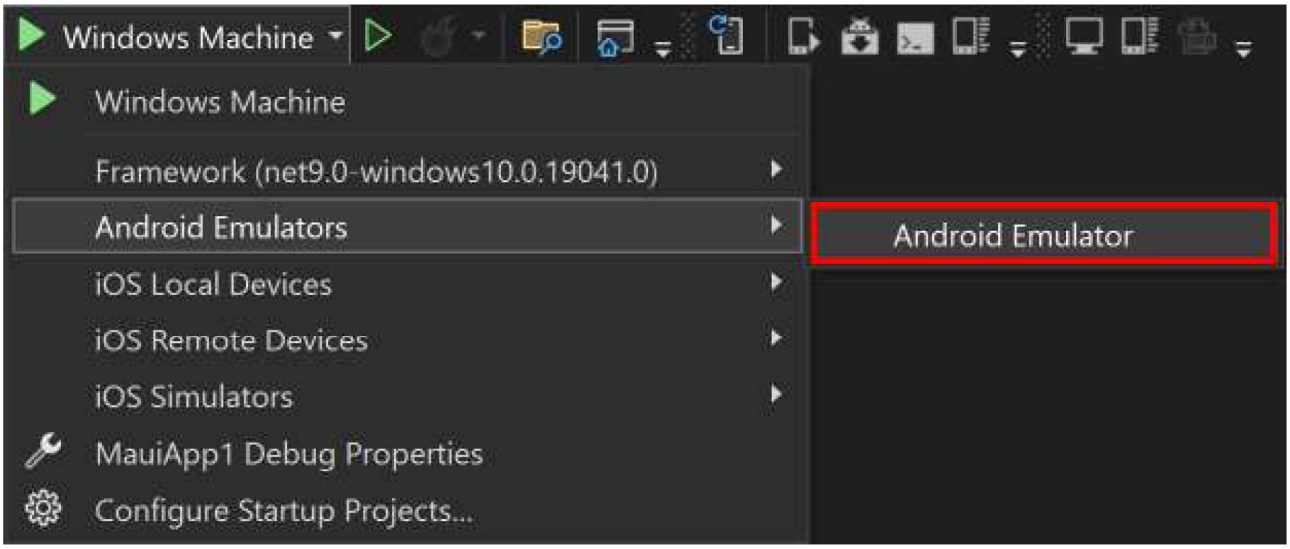
4. Dans la fenêtre **Additional information**, choisissez la version de .NET que vous souhaitez cibler, puis cliquez sur le bouton **Create** :



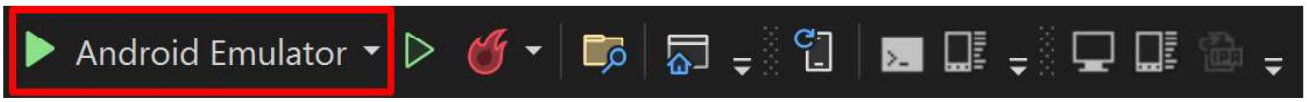
5. Attendez que le projet soit créé et que ses dépendances soient restaurées :



6. Dans la barre d'outils Visual Studio, utilisez la liste déroulante **Debug Target** pour sélectionner **Android Emulators**, puis l'entrée **Android Emulator** :

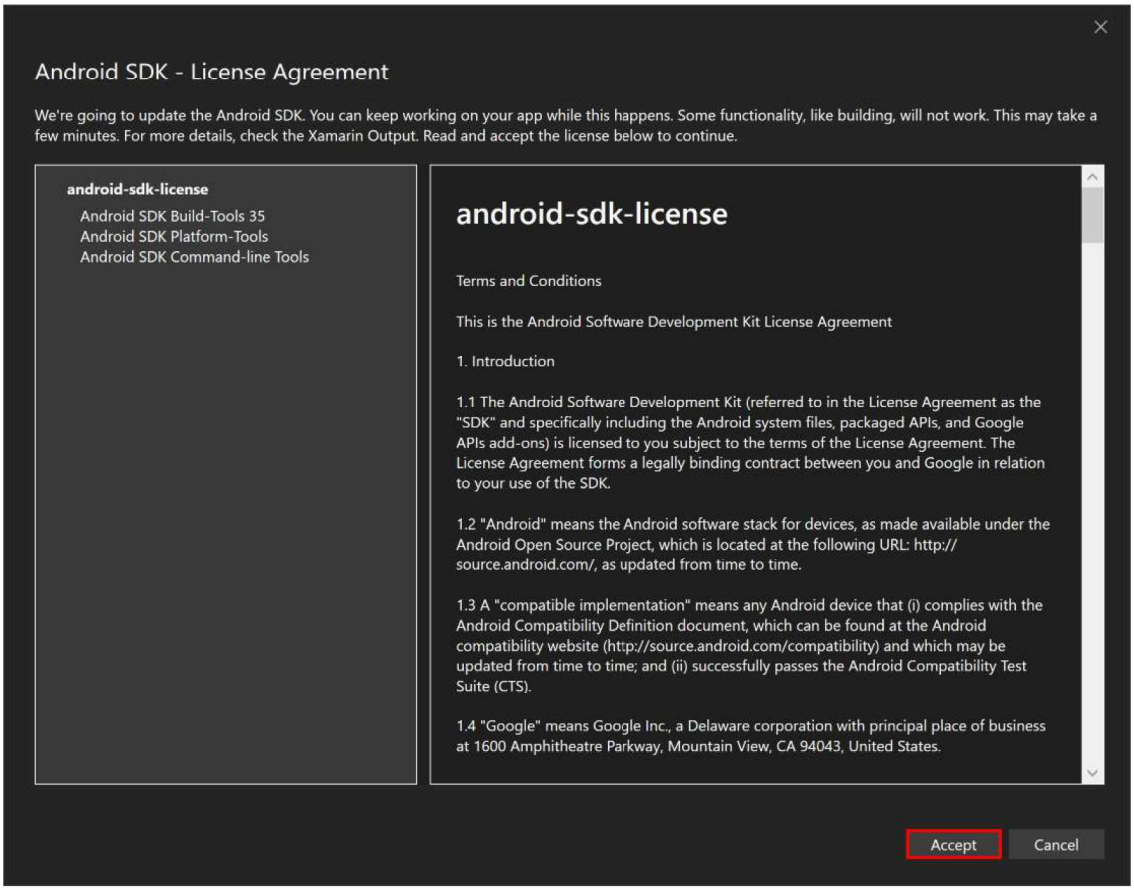


7. Dans la barre d'outils Visual Studio, appuyez sur le bouton **Android Emulator** :



Visual Studio démarre le processus d'installation du SDK Android et de l'émulateur Android par défaut.

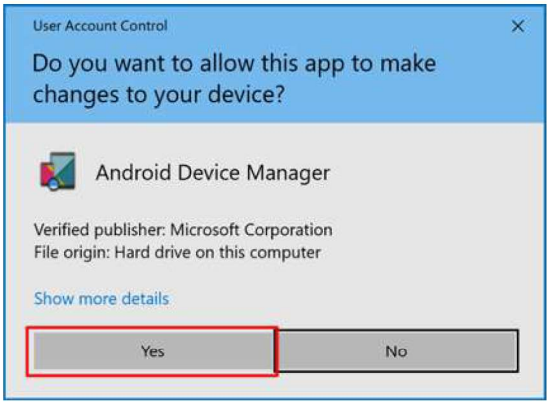
8. Dans la fenêtre **Android SDK - License Agreement**, appuyez sur le bouton **Accept** :



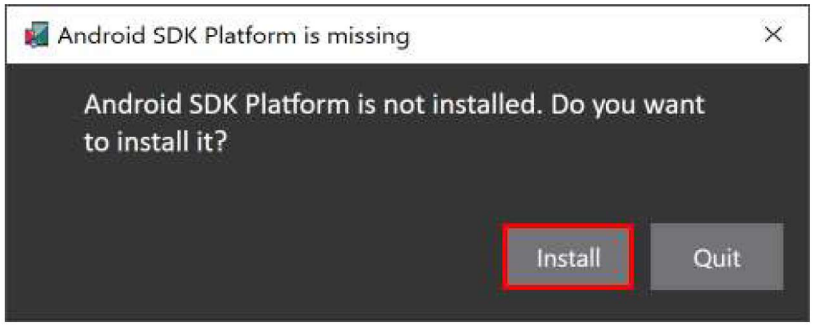
9. Dans la boîte de dialogue **User Account Control**, appuyez sur le bouton **Yes** :

Attendez que Visual Studio télécharge le SDK Android et l'émulateur Android par défaut.

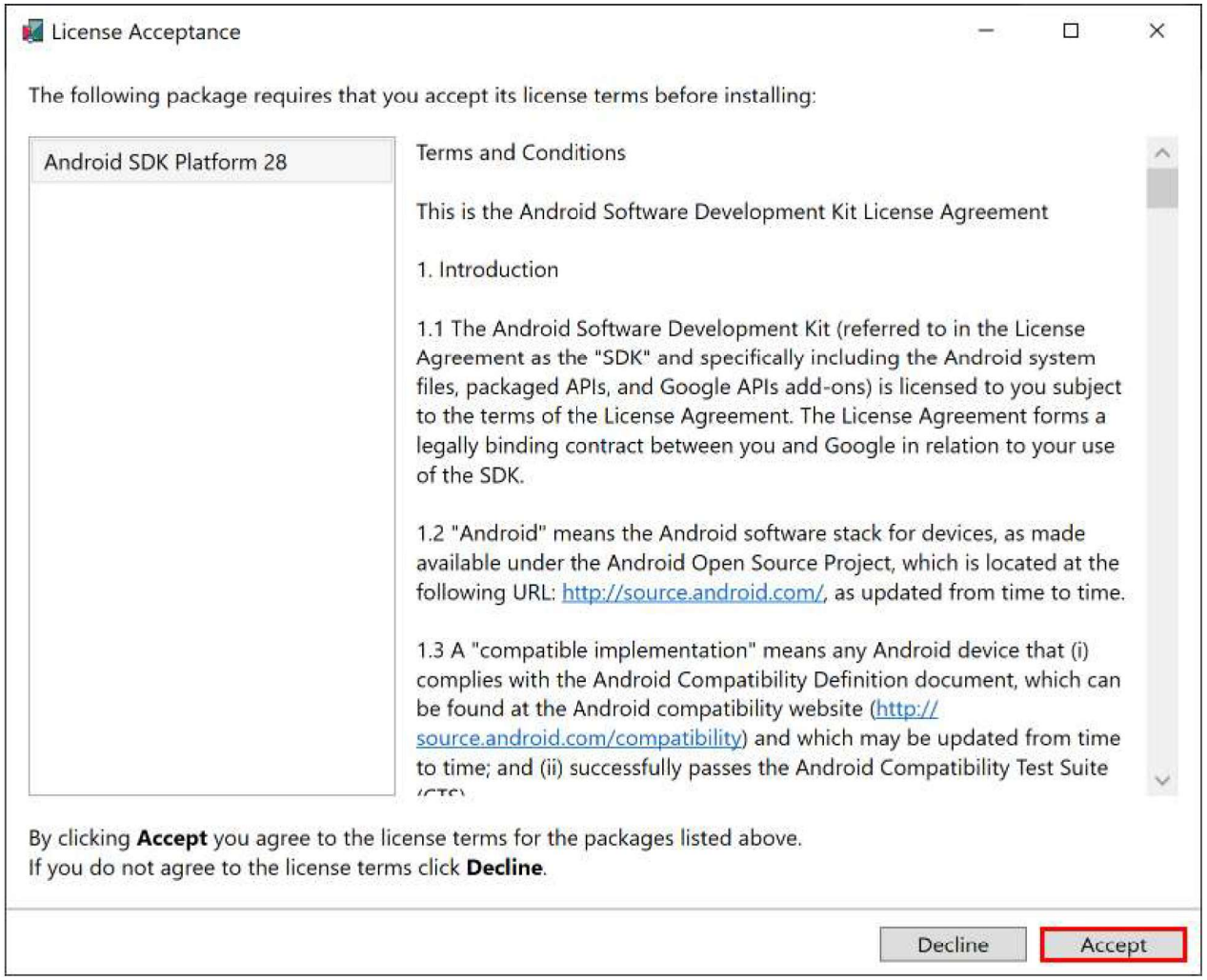
10. Dans la boîte de dialogue **User Account Control**, appuyez sur le bouton **Yes** :



11. Dans la boîte de dialogue **Android SDK Platform is missing**, appuyez sur le bouton **Install** :

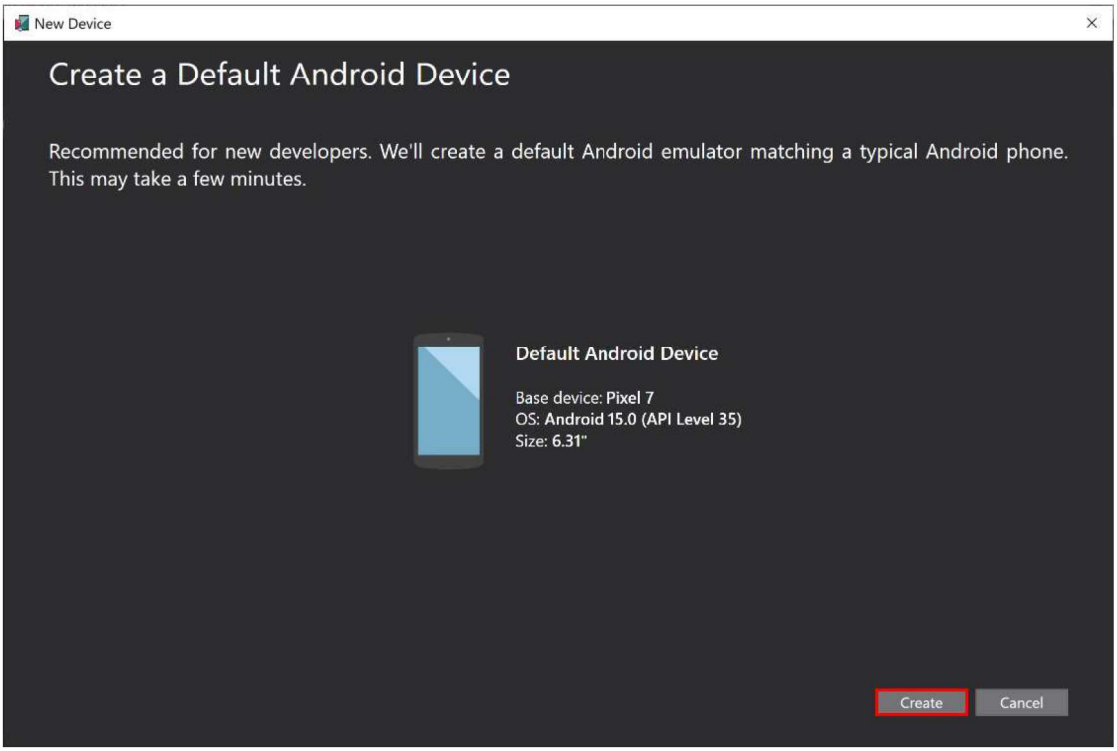


12. Dans la fenêtre **Android SDK - License Agreement**, appuyez sur le bouton **Accept** :



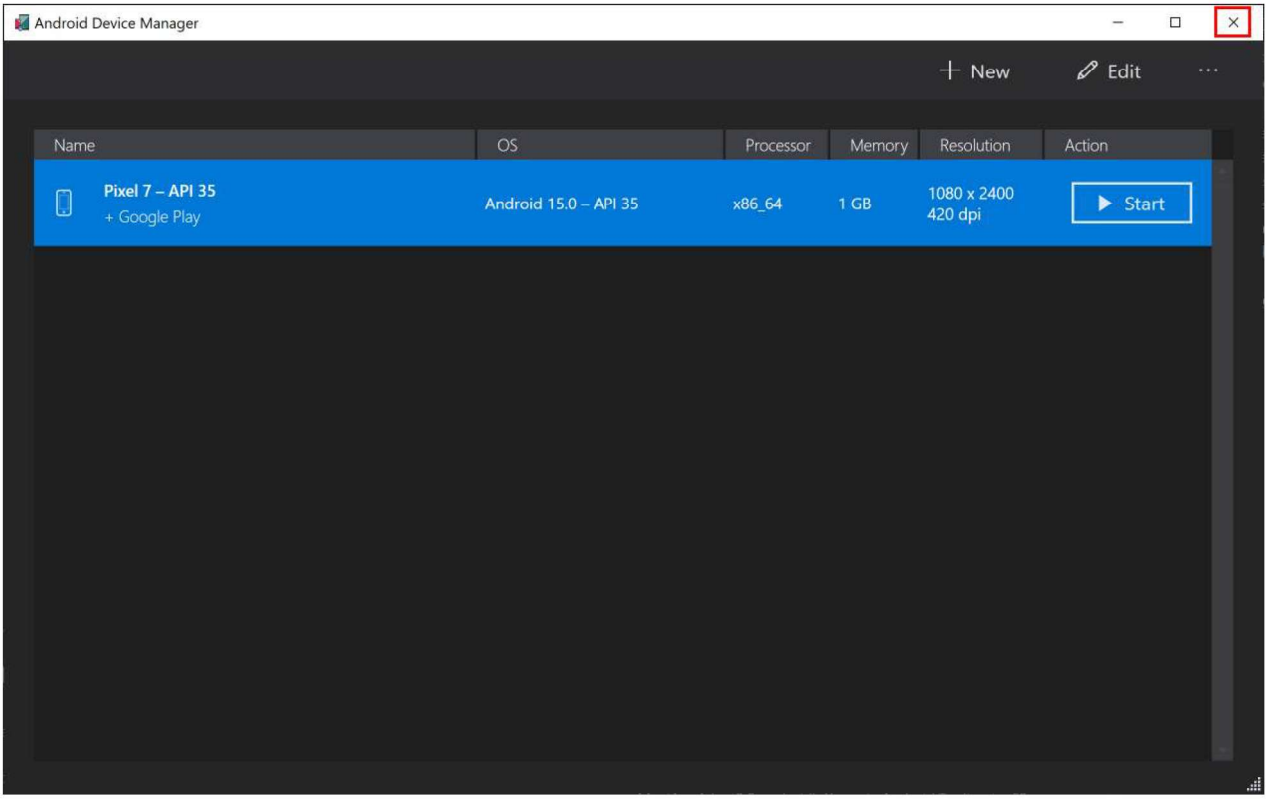
Attendez que Visual Studio installe les composants du SDK Android.

13. Dans la fenêtre **New Device**, appuyez sur le bouton **Create** :

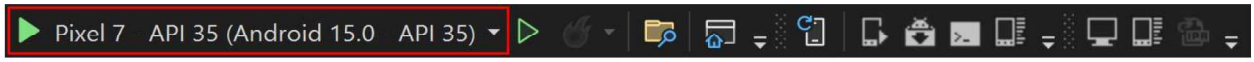


Attendez que Visual Studio télécharge, décompresse et crée un émulateur Android.

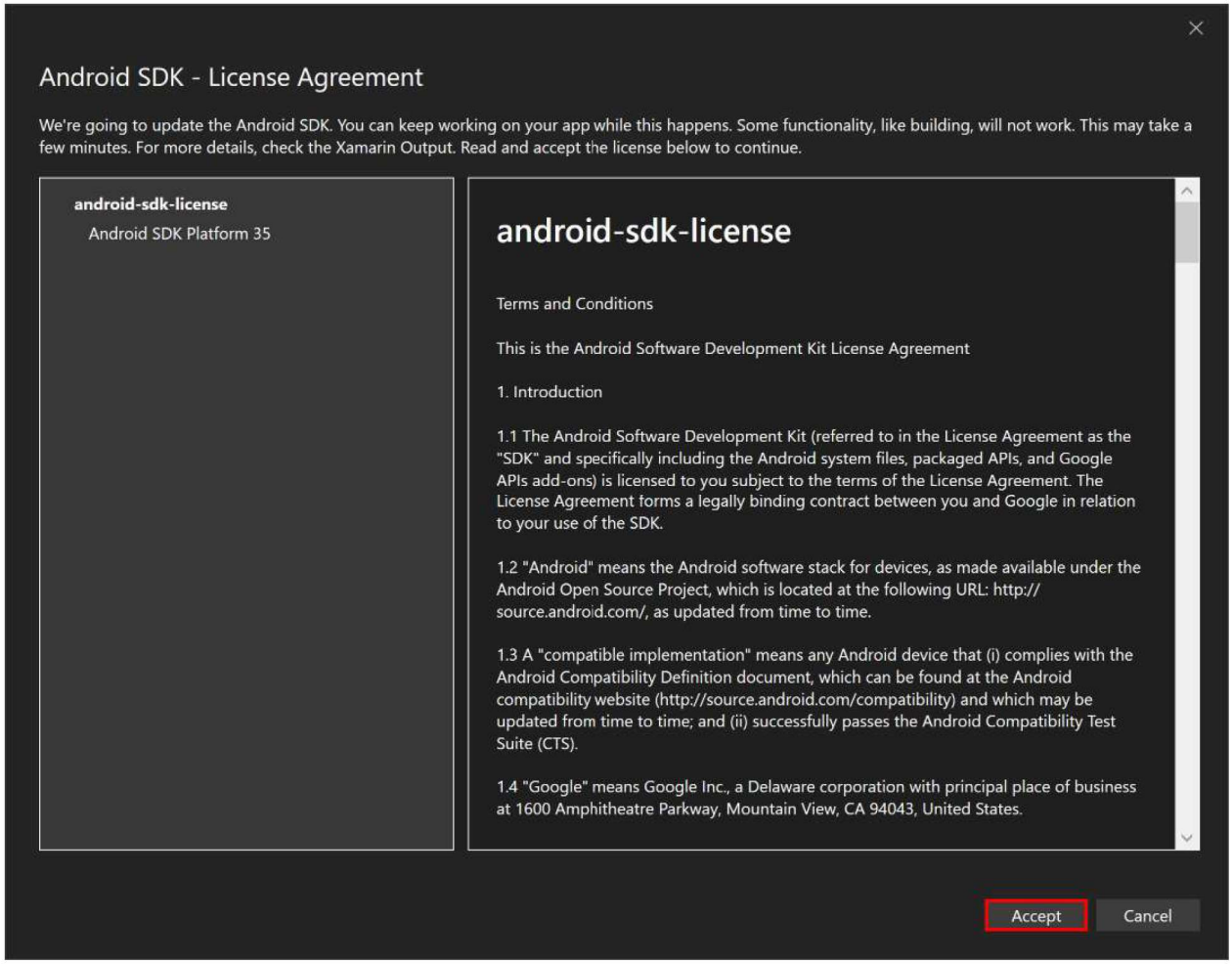
14. Fermez la fenêtre **Android Device Manager** :



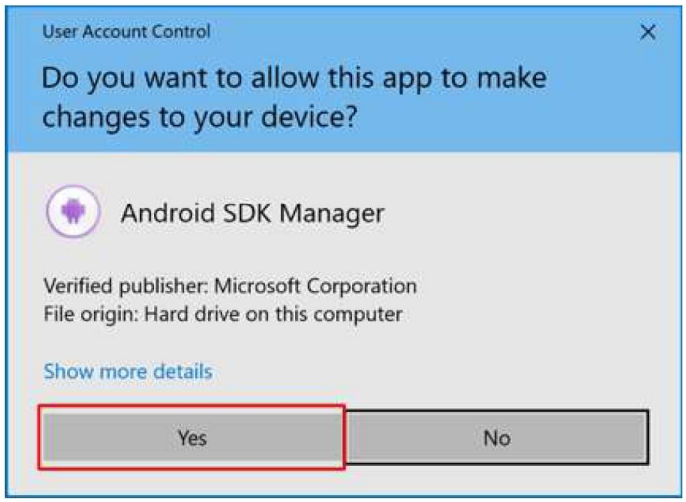
15. Dans la barre d'outils Visual Studio, appuyez sur le bouton **Pixel 7 - API 35 (Android 15.0 - API 35)** :



16. Dans la fenêtre **Android SDK - License Agreement**, appuyez sur le bouton **Accept** :

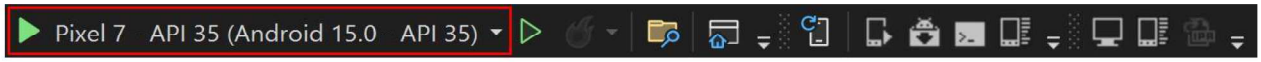


17. Dans la boîte de dialogue **User Account Control**, appuyez sur le bouton **Yes** :



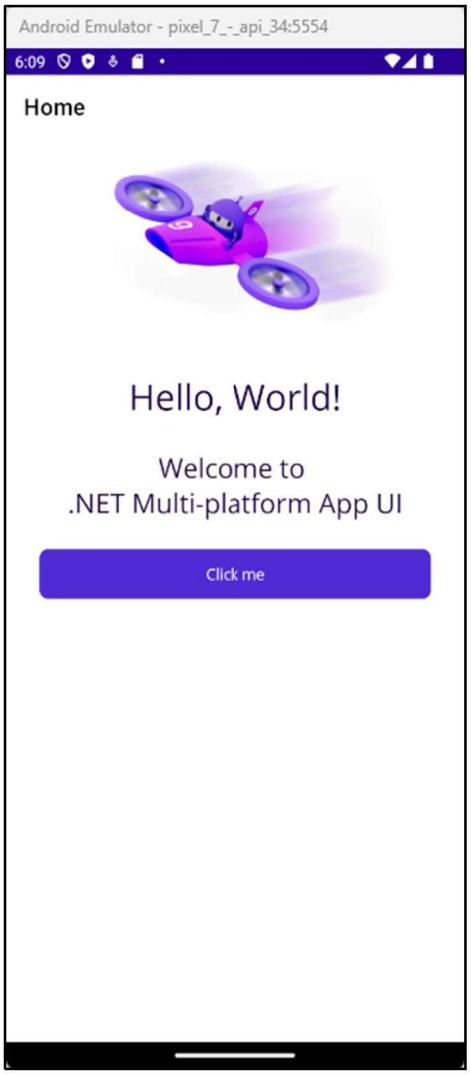
Attendez que Visual Studio soit installé pour télécharger le SDK Android.

18. Dans la barre d'outils Visual Studio, appuyez sur le bouton **Pixel 7 - API 35 (Android 15.0 - API 35)** pour générer et exécuter l'application :



Visual Studio démarre l'émulateur Android, génère l'application et déploie l'application sur l'émulateur.

Dans l'application en cours d'exécution dans l'émulateur Android, appuyez plusieurs fois sur le bouton **Click me** et observez l'incrémentation du nombre de clics du bouton.

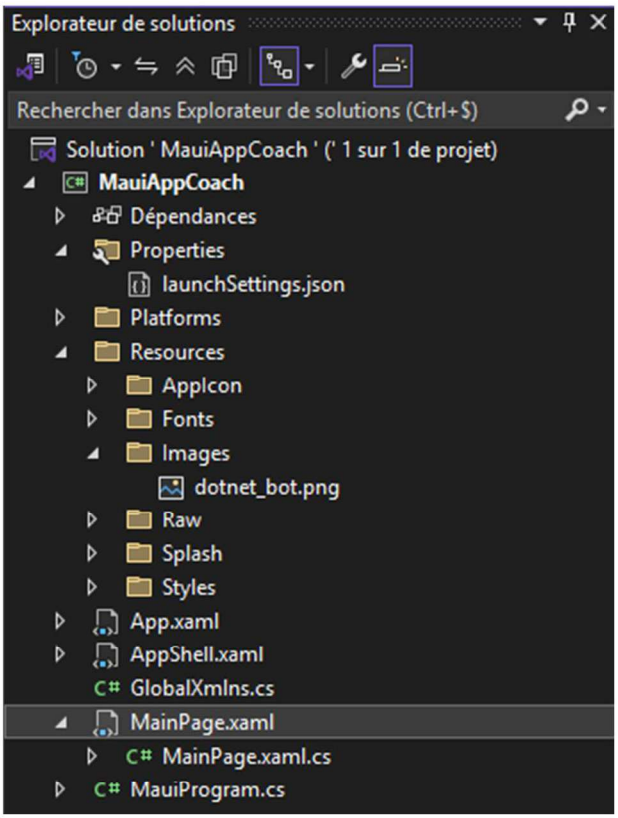


Remarque : L'accélération matérielle doit être activée pour optimiser les performances de l'émulateur Android. À défaut, l'émulateur s'exécutera très lentement.
Pour plus d'informations, consultez l'article Comment activer l'accélération matérielle avec des émulateurs Android (Hyper-V & AEHD) à l'adresse suivante :
https://learn.microsoft.com/fr-fr/dotnet/maui/android/emulator/hardware-acceleration?view=net-maui-10.0

Comprendre le fonctionnement de base

Avant de commencer à coder, nous allons découvrir le fonctionnement d'une application .NET MAUI à partir de l'exemple fourni par Visual studio.

Commencez par ouvrir l'explorateur de solutions, sur la partie droite de Visual Studio :



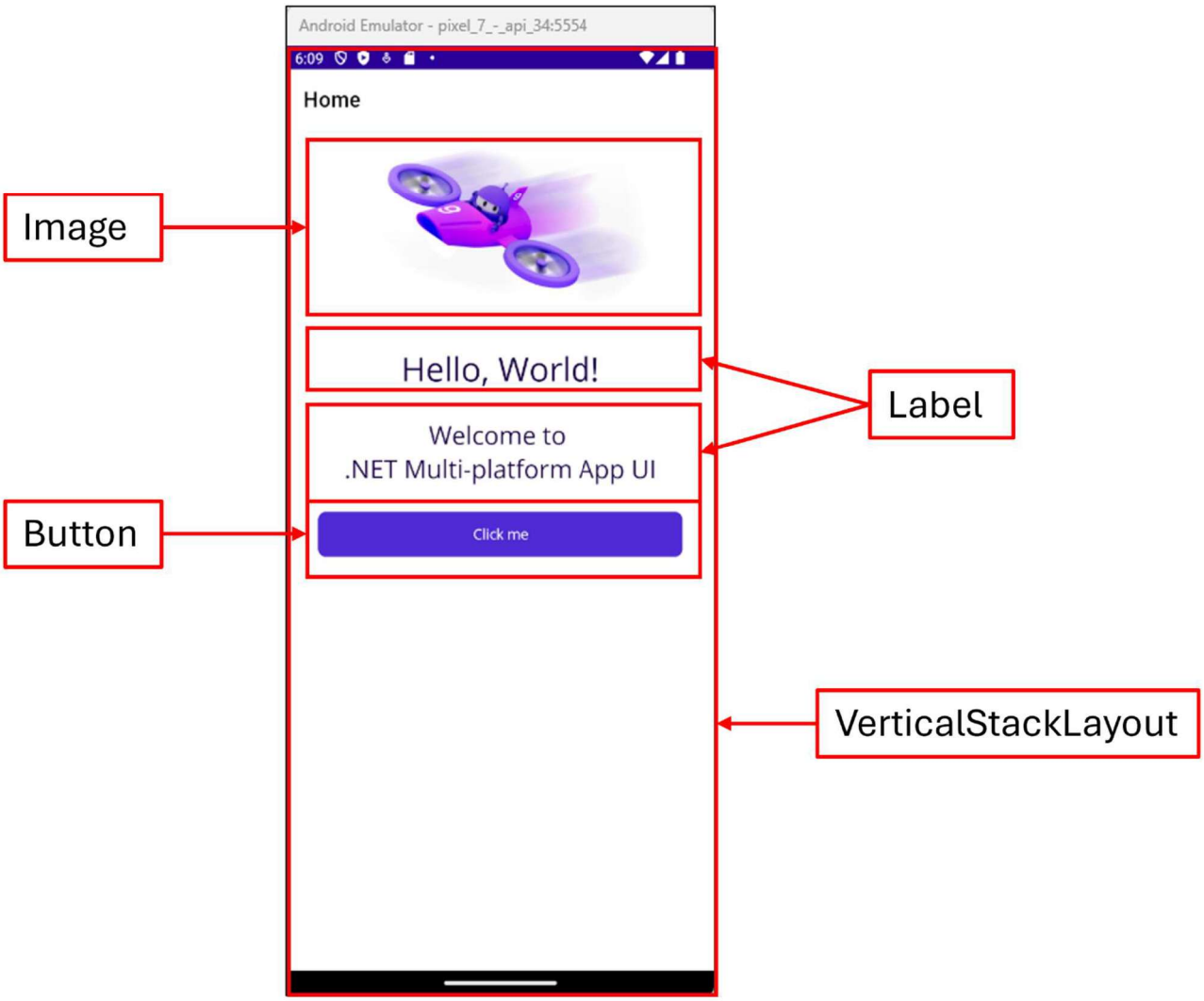
Le répertoire « **Resources** » est destiné à contenir des éléments utilisés par l'application. Ici il contient l'image utilisée par l'application (« dotnet_bot.png »).

Le fichier « **Mainpage.xaml** » permet de définir tous les éléments de l'interface graphique de la page principale de notre application :

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MauiAppCoach.MainPage">
    <ScrollView>
        <VerticalStackLayout
            Padding="30,0"
            Spacing="25">
            <Image
                Source="dotnet_bot.png"
                HeightRequest="185"
                Aspect="AspectFit"
                SemanticProperties.Description="dot net bot in a hovercraft number nine" />
            <Label
                Text="Hello, World!"
```

```
            Style="{StaticResource Headline}"
            SemanticProperties.HeadingLevel="Level1" />
        <Label
            Text="Welcome to &#10;.NET Multi-platform App UI"
            Style="{StaticResource SubHeadline}"
            SemanticProperties.HeadingLevel="Level2"
            SemanticProperties.Description="Welcome to dot net Multi platform App U I" />
        <Button
            x:Name="CounterBtn"
            Text="Click me"
            SemanticProperties.Hint="Counts the number of times you click"
            Clicked="OnCounterClicked"
            HorizontalOptions="Fill" />
    </VerticalStackLayout>
  </ScrollView>
</ContentPage>
```

L'illustration suivante permet de visualiser les différents éléments qui sont affichés sur la page principale de notre application :



Le fichier « **Mainpage.xaml.cs** » est le code associé à la page principale :

```
namespace MauiAppCoach
{
    public partial class MainPage : ContentPage
    {
        int count = 0;
        public MainPage()
        {
            InitializeComponent();
        }

        private void OnCounterClicked(object? sender, EventArgs e)
        {
            count++;
            if (count == 1)
                CounterBtn.Text = $"Clicked {count} time";
            else
                CounterBtn.Text = $"Clicked {count} times";

            SemanticScreenReader.Announce(CounterBtn.Text);
        }
    }
}
```

Ici, la méthode **`OnCounterClicked()`** est déclenchée lors de l'appui sur le bouton **`CounterBtn`**. La variable count est incrémentée puis affichée sur le bouton (propriété **`CounterBtn.Text`**).

TRAVAIL A FAIRE POUR LES ETUDIANTS :

- Modifiez les différents éléments du fichier « MainPage.xaml » pour vous familiariser avec la création d'interfaces graphiques et observez les résultats.
- Améliorez l'application pour afficher le nombre de clics dans un nouveau label (placé par exemple sous le bouton « Click me ».
- Trouvez le fichier qui permet de changer le titre de l'application et remplacez-le par « test SIO ».

Application « Coach » : création de l'interface graphique

1. Nettoyage de l'application de test

Commencez par modifier le titre de l'application : remplacez « **Test SIO** » par « **Coach** ».

Allez dans le fichier « **MainPage.xaml** » et supprimez tous les éléments inutiles de l'interface graphique pour ne garder que la balise « **ContentPage** » :

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MauiAppCoach.MainPage">


</ContentPage>
```

Supprimez ensuite le code inutile dans le fichier « MainPage.xaml.cs » :

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

2. Ajout des nouveaux éléments dans le fichier « MainPage.xaml » :

Nous allons commencer par modifier notre page principale « **MainPage.xaml** » en ajoutant tous les éléments de notre interface graphique.

Par exemple, le code suivant permet d'ajouter une zone de saisie de données, en autorisant la saisie d'une valeur numérique :

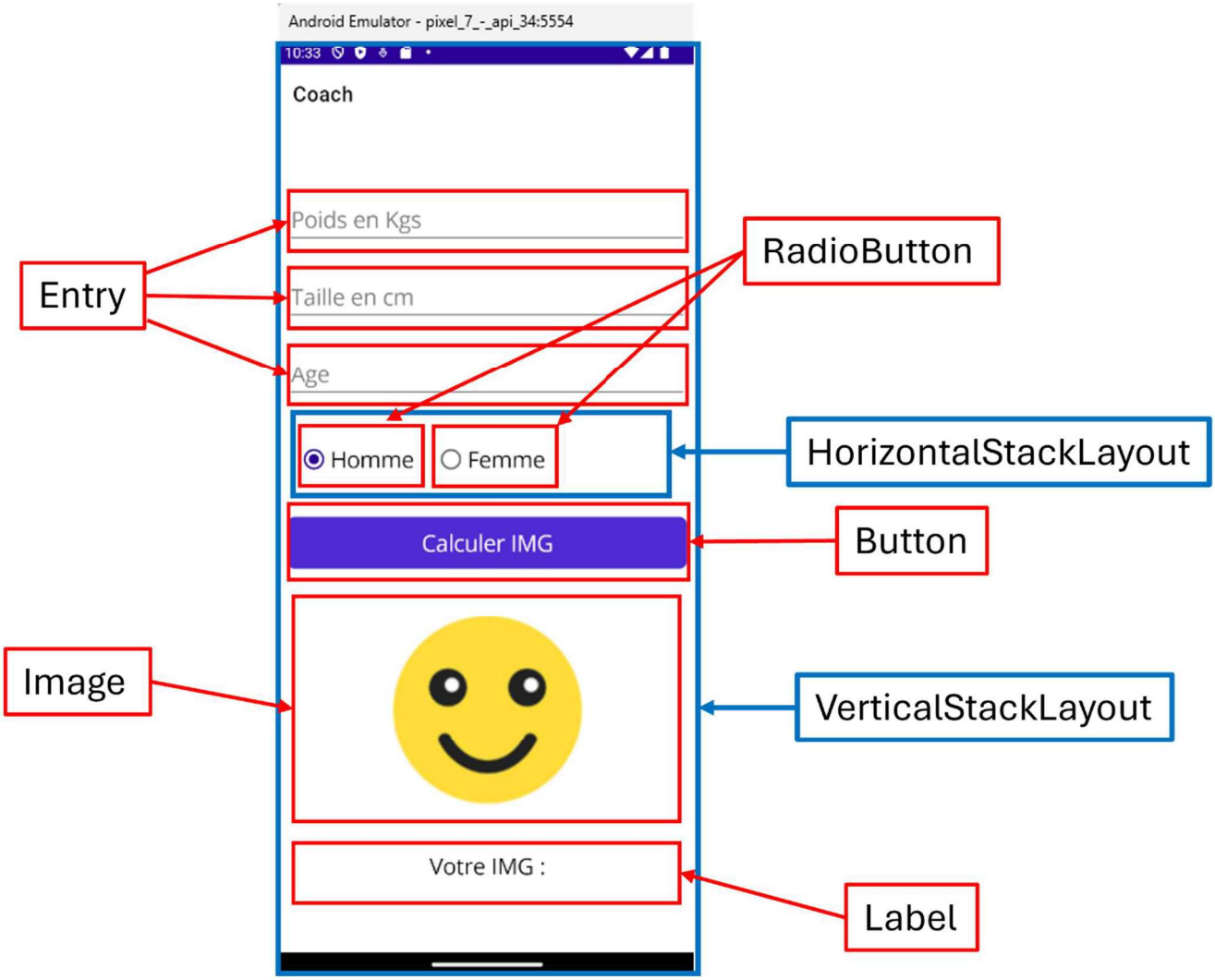
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MauiAppCoachV1.MainPage">


    <VerticalStackLayout VerticalOptions="Center" Spacing="25" Margin="10">


        <Entry
            TextColor="DarkBlue"
            FontSize="22"
            x:Name="entPoids"
            Placeholder="Poids en Kgs"
```

```
            PlaceholderColor="#7d7c7c"
            ClearButtonVisibility="WhileEditing"
            Keyboard="Numeric"
            VerticalOptions="Center">
        </Entry>
</ContentPage>
```

Voici l'interface graphique à réaliser pour notre application « Coach » :



Remarques :

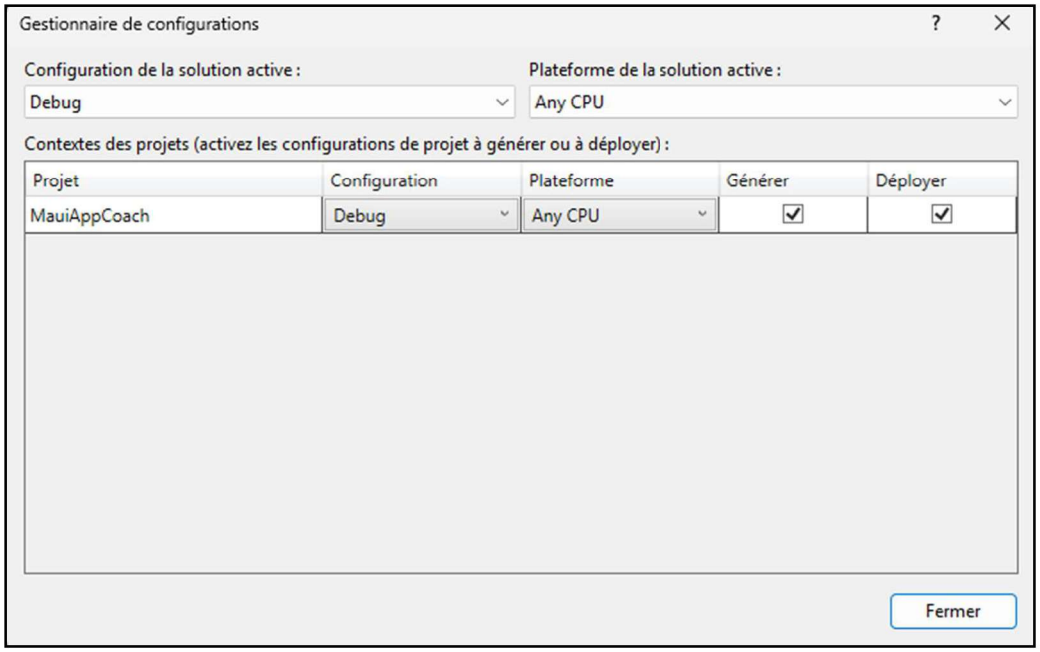
- N'oubliez pas de nommer chaque élément que vous devrez utiliser plus tard dans votre code (propriété « **x:Name** »)
- Choisissez un « RadioButton » sélectionné par défaut (Homme ou Femme).
- N'ajoutez pas le déclencheur d'évènement dans votre élément « Button » pour l'instant (propriété « Clicked »).
- N'hésitez pas à consulter la documentation en ligne, par exemple pour découvrir comment fonctionnent les « Layouts » de .NET MAUI : https://learn.microsoft.com/en-us/dotnet/maui/user-interface/layouts/?view=net-maui-10.0

TRAVAIL A FAIRE POUR LES ETUDIANTS :

- Modifiez les différents éléments du fichier « MainPage.xaml » pour créer l'interface graphique de l'application « Coach ».
- Changez le titre de l'application pour afficher « Coach » sur la page principale.

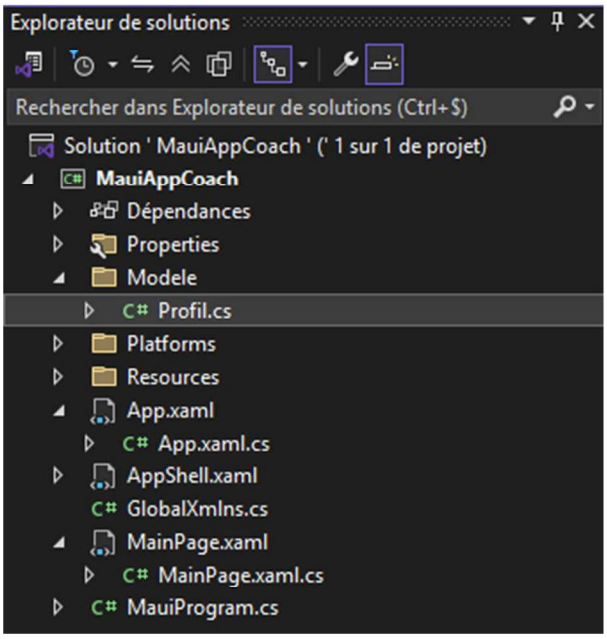
Attention : avant de générer et exécuter votre application, vérifiez dans le gestionnaire de configurations que la case « **Déployer** » est bien cochée.

Faites un clic-droit sur le nom de votre solution puis cliquez sur « **Gestionnaire de configurations …** » :



3. Création de la classe métier « Profil » :

Commencez par ajouter un dossier **« Modele »** à la racine de l'application puis créez une nouvelle classe « **Profil** » dans ce dossier.



La classe « **Profil** » permet de mémoriser les données (sexe, poids, taille, âge) et réaliser tous les calculs nécessaires.

Pour chaque objet de la classe profil, on aura les propriétés suivantes :

- sexe `// 0 pour une Femme et 1 pour un homme`
- poids `// poids en Kg`
- taille `// taille en cm`
- age `// âge en années`
- img `// Indice de Masse Grasse`
- message `// Annonce du résultat avec commentaire`

Calcul de l'IMG et rédaction du message destiné à l'utilisateur :

L'indice de masse grasse (IMG) est un indice, exprimé en pourcentage, qui permet de juger de la proportion de tissus adipeux d'une personne adulte. Il prend en compte la disproportion entre la masse de graisse et celles des muscles (source : Wikipédia).

Comment calculer l'IMG ?
IMG = (1,2 * Poids / Taille ^{2}) + (0,23 * Age) – (10,83 * S) – 5,4
Avec S = 0 pour une femme, 1 pour un homme.
Poids en Kg, Taille en mètres et âge en années.

Comment interpréter l'IMG ?
Pour les femmes :

- Si l'IMG est < 25 % : Trop maigre.
- Si l'IMG est compris entre 25 et 30 % : Parfait.
- Si l'IMG est supérieur à 30% : Surpoids.

Pour les hommes :

- Si l'IMG est < 15 % : Trop maigre.
- Si l'IMG est compris entre 15 et 20 % : Parfait.
- Si l'IMG est supérieur à 20% : Surpoids.

TRAVAIL A FAIRE POUR LES ETUDIANTS :

- Créez la **classe publique « Profil »** avec toutes ses propriétés privées dans le dossier « Modele ».
- Créez la méthode privée **CalculIMG()** qui effectue le calcul de l'IMG et qui valorise la propriété **img**.
- Créez la méthode privée **ResultatIMG()** qui interprète le résultat et qui valorise la propriété **message** (la propriété message peut recevoir les valeurs suivantes : "Parfait.", "Trop maigre.", "Surpoids.").
- Ajoutez un appel à ces deux méthodes **dans le constructeur** pour valoriser les propriétés **img** et **message** dès l'instanciation d'un objet de la classe profil.
- Créez les getters.
- Effectuez les **tests unitaires** pour vérifier que les méthodes **CalculIMG()** et **ResultatIMG()** fonctionnent correctement.

Remarque importante : n'oubliez pas de commenter toutes vos méthodes (« /// » sous Visual Studio) !

Calcul et affichage du résultat dans la page principale

Nous allons commencer par ajouter le déclencheur d'évènement dans votre élément **« Button »** (propriété **« Clicked »**).

Ouvrez le fichier « MainPage.xaml » et placez-vous au niveau du contrôle « Button ».

Attention, le nom du déclencheur est basé sur la propriété « **x :Name** ».

Renommez votre bouton « **btCalculer** » puis placez-vous à la fin du contrôle « Button » (juste avant la fermeture de la balise />).

Commencez à taper « **Clicked=** » puis cliquez sur **<Nouveau gestionnaire d'évènement>** :

```
<Button
    x:Name="btCalculer"
    FontSize="22"
    Text="Calculer IMG"
    SemanticProperties.Hint="Calculate IMG"
    HorizontalOptions="Fill"
    Clicked=" = "/>
                <Nouveau gestionnaire d'événements>
```

Remarque : vous pouvez également vous placer entre les guillemets de la propriété **Clicked " "** et appuyer sur « **Ctrl + Espace** » pour créer le déclencher et générer le code.

Observez le résultat dans le fichier « **MainPage.xaml** » :

```
<Button
     x:Name="btCalculer"
     FontSize="22"
     Text="Calculer IMG"
     SemanticProperties.Hint="Calculate IMG"
     HorizontalOptions="Fill"
     Clicked="btCalculer_Clicked"/>
```

Et dans le fichier « MainPage.xaml.cs » :

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

        private void btCalculer_Clicked(object sender, EventArgs e)
        {

        }
```

```
    }
}
```

Un nouvel évènement est créé dans le code de la page principale. C'est là que nous allons pouvoir calculer notre IMG et affichage le résultat !

TRAVAIL A FAIRE POUR LES ETUDIANTS :

- Modifiez la méthode « **btCalculer_Clicked()** » pour calculer l'IMG et afficher le résultat.

Remarques importantes :

- Vous devez commencer par récupérer les valeurs saisies par l'utilisateur puis **instancier un nouvel objet de type « Profil »**. **Attention**, la classe « **Profil** » se trouve dans le dossier « **Modele** ». Il faut ajouter `using MauiAppCoach.Modele;` au début du fichier « **MainPage.xaml.cs** » pour pouvoir utiliser la classe « Profil » dans le code…
- En fonction du contenu de la propriété « **message** » de notre nouvel objet, vous devrez afficher le **résultat** ainsi que l'**image associée** (smiley trop maigre, smiley parfait ou smiley surpoids).
- Vous pouvez accéder à la valeur d'un contrôle à partir de son nom (propriété **x :Name**). Par exemple, **entPoids.Text** correspond à la valeur saisie par l'utilisateur dans le contrôle de type « **Entry** » avec « **x:Name= "entPoids"** ».
- Vous pouvez modifier l'image à afficher avec la propriété « **source** » du contrôle « **Button** ». **Attention**, le fichier doit se trouver dans le dossier « **Resources/Images** » du projet et l'extension ne doit pas figurer dans la propriété « **source** » (par exemple **img.Source="smiley_parfait"** pour le fichier « **Resources/Images/smiley_parfait.png** ».
- N'oubliez pas d'entourer votre code d'un « **try … catch** » pour éviter le plantage de l'application en cas de saisie incorrecte, par exemple :

```
private void btCalculer_Clicked(object sender, EventArgs e)
{
     try
     {
           //Votre code
     }
     catch (Exception ex)
     {
         DisplayAlertAsync("Erreur", "Saisie incorrecte", "Ok");
     }
 }
```

Persistance des données

Etudions maintenant les méthodes qui permettent de sauvegarder et de restaurer les données saisies par l'utilisateur.

Pour garder une trace de chaque exercice, je vous conseille de mettre de côté votre application Coach (par exemple en la nommant **CoachV1**), et de repartir de cette première version pour réaliser les versions suivantes (CoachV2 pour la persistance des données qui utilise la sérialisation, CoachV3 pour la base de données locale et CoachV4 pour la base de données distante).

1. Sérialisation

Ici, nous allons enregistrer les données dans un fichier local (sur le téléphone), au format binaire.

Faites un clic droit sur l'application Coach dans l'explorateur de solutions et créez un nouveau dossier nommé « **outils** » (à la racine de votre application).

Pour commencer, nous allons créer une nouvelle classe abstraite nommée « **Serializer** » dans ce nouveau dossier (clic droit sur le dossier « **outils** » puis Ajouter – Classe).

Pour rappel, une classe abstraite est une classe qui n'est pas instanciable (on ne peut pas utiliser « new » pour créer un nouvel objet).

Pour indiquer au compilateur qu'il s'agit bien d'une classe abstraite, ajoutez simplement le mot-clé `abstract` devant le nom de la classe.

Dans la classe « **Serializer »**, nous allons créer deux méthodes statiques (à portée de classe) : « **serialize()** » pour enregistrer les données et « **deserialize()** » pour charger les données.

```
abstract class Serializer
{
    public static void serialize(string nomFichier, Profil profil)
    {
        string fichier =
        Path.Combine(System.Environment.GetFolderPath(System.Environment.SpecialFolder.
        LocalApplicationData), nomFichier);

        if (File.Exists(fichier))
        {
            File.Delete(fichier);
        }
        try
        {
            //Sérialisation des objets de la collection
            string jsonString = JsonSerializer.Serialize(profil);

           //WriteAllText tronque et remplace le fichier cible s'il existe déjà.
            File.WriteAllText(fichier, jsonString);
        }
```

```
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine("Erreur lors de la sérialisation");
            }
        }

        public static Profil deserialize(string nomFichier)
        {
            Profil profil = null;

            string fichier =
            Path.Combine(System.Environment.GetFolderPath(System.Environment.SpecialFolder.
            LocalApplicationData), nomFichier);

            if (File.Exists(fichier))
            {
                try
                {
                    string jsonString = File.ReadAllText(fichier);

                    //Récupération des données sérialisées
                    profil = JsonSerializer.Deserialize<Profil>(jsonString);
                }
                catch (Exception ex)
                {
                     System.Diagnostics.Debug.WriteLine("Erreur lors de la desérialisation");
                }
            }
            return profil;
        }
    }
```

Remarque : cette classe **Serializer** peut être utilisée dans n'importe quelle application C# !

L'étape suivante consiste à indiquer au compilateur quels sont les objets qui sont sérialisables. Ici, nous voulons sérialiser les objets de la classe **profil**.

Pour cela, il suffit d'ajouter `[Serializable]` juste avant la déclaration de la classe profil :

```
[Serializable]
class Profil
{
        ....
}
```

Attention, le processus de sérialisation nécessite d'adapter la classe métier « **Profil** » :

Commencez par créer un nouveau constructeur sans paramètres :

```
public Profil()
{
    sexe = 0;
    poids = 0;
    taille = 0;
    age = 0;
    img = 0;
    message="";
}
```

Modifiez les getters pour utiliser le format suivant (**Nom de la propriété avec une majuscule** et ajout d'un setter) :

```
public int Poids
{
    get { return poids; }
    set { poids = value; }
}
```

Attention, il faudra obligatoirement adapter le code de votre méthode « **btCalculer_Clicked()** » dans le fichier « **MainPage.xaml.cs** ».

Par exemple :

```
if (unProfil.GetMessage() == "Parfait.") //utilisation de l'ancien Getter GetMessage()
```

Devient

```
if (unProfil.Message == "Parfait.") //utilisation du nouveau Getter Message
```

Une fois cette étape réalisée, nous allons modifier le code du fichier « **MainPage.xaml.cs** » pour enregistrer et charger nos profils en utilisant la sérialisation Json.

Dans un premier temps, nous allons créer une propriété privée « readonly » de type string nommée **nomFichier** pour stocker le nom du fichier utilisé pour la sérialisation :

```
public partial class MainPage : ContentPage
{
    private readonly string nomFichier = "saveprofil"; //fichier pour sérialisation
    …
}
```

Il nous reste maintenant à procéder à la sérialisation et à la désérialisation au bon moment. Par exemple, nous pouvons décider de sérialiser (enregistrer les données) au moment de la création du profil, dans la méthode « **btCalculer_Clicked()** » :

```
private void btCalculer_Clicked(object sender, EventArgs e)
{
    try
    {
        var poids = int.Parse(entPoids.Text);
        var taille = int.Parse(entTaille.Text);
        var age = int.Parse(entAge.Text);
        var sexe = 0;

        if (rbHomme.IsChecked)
        {
            sexe = 1;
        }

        if (rbFemme.IsChecked)
        {
            sexe = 0;
        }

        Profil unProfil = new Profil(poids, taille, age, sexe);

        //Enregistrement du profil (serialisation Json)

        Serializer.serialize(nomFichier, unProfil);

        …
```

Remarque : la classe « **Serializer** » est une classe abstraite, on peut accéder aux méthodes de la classe en tapant simplement le nom de la classe suivie de « **.** ». Ici, on passe en paramètre le nom du fichier binaire de sérialisation et l'objet sérialisé (**profil**).

Pour récupérer les données, nous allons créer une nouvelle méthode statique « **RecupProfil()** », toujours dans le fichier « **MainPage.xaml.cs** » :

```
private void RecupProfil()
{
    try
    {
        Profil unProfil = null;
        unProfil = Serializer.deserialize(nomFichier);

        if (unProfil != null)
        {
            entPoids.Text = unProfil.Poids.ToString();
            entTaille.Text = unProfil.Taille.ToString();
            entAge.Text = unProfil.Age.ToString();
```

```
            if (unProfil.Sexe == 1)
            {
                rbHomme.IsChecked = true;
            }
            else
            {
                rbFemme.IsChecked = true;
            }

            if (unProfil.Message == "Parfait.")
            {
                imgResult.Source = "smiley_parfait";
                lblImg.TextColor = Colors.Green;
            }
            else if (unProfil.Message == "Trop maigre.")
            {
                imgResult.Source = "smiley_tropmaigre";
                lblImg.TextColor = Colors.DarkRed;
            }
            else
            {
                imgResult.Source = "smiley_surpoids";
                lblImg.TextColor = Colors.DarkRed;
            }
            lblImg.Text = "Votre IMG : " + unProfil.Img.ToString("F2") + "%. " +
             unProfil.Message;
        }
    }
    catch (Exception ex)
    {
        DisplayAlertAsync("Erreur", "Erreur lors de la récupération des données", "Ok");
    }
}
```

Il nous reste à appeler la méthode « **RecupProfil()** » au démarrage de la page principale :

```
public partial class MainPage : ContentPage
{
    private readonly string nomFichier = "saveprofil"; //fichier pour sérialisation
    public MainPage()
    {
        InitializeComponent();
        RecupProfil();
    }
    …
```

TRAVAIL A FAIRE POUR LES ETUDIANTS :

- Dans le fichier « MainPage.xaml.cs », l'affichage du résultat (image et message) est codé 2 fois ! Refactorisez le code pour éviter ce doublon.
- Trouvez une solution sur le terminal Android pour remettre à zéro les données utilisées par l'application.

2. Base de données locale (SQLite)

Pour améliorer notre application, nous allons maintenant essayer d'enregistrer plusieurs profils dans un base de données locale.

Nous allons commencer par faire une copie de notre application **CoachV2** et la renommer **CoachV3**.

Avant de travailler sur notre base SQLite, nous allons devoir nettoyer notre code.

TRAVAIL A FAIRE POUR LES ETUDIANTS :

Dans le fichier « MainPage.xaml.cs »

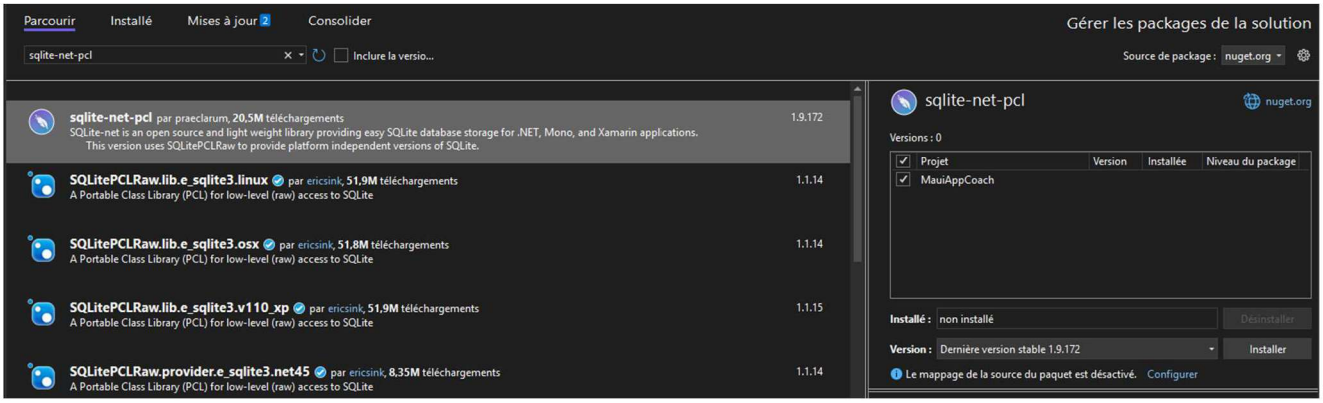
- Supprimez la creation de la variable « nomFichier ».
- Supprimez la méthode « RecupProfil() ».
- Supprimez l'appel à la fonction « RecupProfil() » dans « MainPage() ».
- Supprimez la ligne « Serializer.serialize(nomFichier, unProfil); » dans la méthode « btCalculer_Clicked() ».

Vérifiez que l'application fonctionne correctement, sans enregistrer le profil. Si tout va bien, l'application est prête pour passer à la suite !

La première étape consiste à installer le package NuGet « **sqlite-net-pcl** ».

Allez dans le menu « **Outils** » puis « **Gestionnaire de package NuGet** » et cliquez sur « **Gérer les packages NuGet pour la solution** ».

Recherchez puis sélectionnez le package « **sqlite-net-pcl** » et cochez « **Projet** » (partie droite) puis cliquez sur le bouton « **Installer** » :



Remarque : pour simplifier les interactions avec la base de données SQLite, nous allons utiliser la sérialisation JSON pour transmettre les données aux requêtes. De la même façon, nous utiliserons la sérialisation pour transformer le résultat des requêtes en objets de type « **Profil** ».

Plusieurs modifications sont à apporter à notre classe « **Profil** » pour enregistrer nos données dans une base de données SQLite.

Commencez par importer la bibliothèque « **SQLite** » : `using SQLite;`

Nous allons ensuite créer une propriété « **id** », que nous allons utiliser comme clé primaire lors de l'enregistrement d'un profil.

Ajoutez une propriété « **datemesure** » de type « **DateTimeOffset** » dans la classe « **Profil** ». Cela nous permettra d'enregistrer la date et l'heure de la création du profil.

Modifiez le constructeur pour y ajouter la valorisation des nouvelles propriétés « id » et « **dateMesure** ».

Enfin, créez un getter et un setter pour la propriété « **datemesure** » :

```
[Serializable]
public class Profil
{
    private readonly Nullable<int> id; //ajout id pour la BDD
    private DateTimeOffset datemesure; //ajout pour historique
    private int sexe; //0 pour une Femme et 1 pour un homme
    private int poids;
    private int taille;
    private int age;
    private double img;
    private string message;

    public Profil(Nullable<int> unId, DateTimeOffset uneDate, int unPoids, int uneTaille,
    int unAge, int unSexe)
    {
        id = unId;
        datemesure = uneDate;
        sexe = unSexe;
        poids = unPoids;
        taille = uneTaille;
        age = unAge;
        CalculIMG();
        ResultatIMG();
    }

    public Profil()
    {
        datemesure = new DateTimeOffset();
        sexe = 0;
        poids = 0;
        taille = 0;
        age = 0;
        img = 0;
```

```
        message="";
    }

    [PrimaryKey, AutoIncrement] //Clé primaire et auto-incrément pour la BDD
    public int Id { get; set; }

    public DateTimeOffset Datemesure
    {
        get { return datemesure; }
        set { datemesure = value; }
    }

    …
```

Notez l'ajout de `[PrimaryKey, AutoIncrement]` avant la méthode `public int Id { get; set; }`. C'est indispensable pour indiquer à SQLite que cette propriété est la clé primaire et qu'il faut utiliser l'auto-incrément.

Ouvrez ensuite le fichier « **MainPage.xaml.cs** » et modifiez la méthode « **BtnCalculer_Clicked()** » pour tenir compte de l'ajout de la propriété « **datemesure** » dans le constructeur de « **Profil** ».

Remarque : nous ne connaissons évidemment pas la valeur à indiquer pour la propriété « **id** » (c'est une valeur auto-incrémentée). Nous utiliserons ici le fait que la propriété soit « **Nullable** » et passerons « **null** » en paramètre.

La date de la mesure ne doit pas être saisie par l'utilisateur mais ajoutée automatiquement. Pour cela, nous utiliserons l'instruction « **DateTimeOffset.Now** ».

```
private void BtnCalculer_Clicked(object? sender, EventArgs e)
{
    try
    {
        DateTimeOffset date = DateTimeOffset.Now;
        var poids = int.Parse(entPoids.Text);
        var taille = int.Parse(entTaille.Text);
        var age = int.Parse(entAge.Text);
        var sexe = 0;

        if (rbHomme.IsChecked )
        {
            sexe = 1;
        }

        if (rbFemme.IsChecked)
        {
            sexe = 0;
        }
```

```
        Profil unProfil = new Profil(null, date, poids, taille, age, sexe);
        //id=null pour auto-increment Bdd
        …
```

Implémentation de l'accès à SQLite :

Dans le dossier « **Outils** », créez une nouvelle classe nommée « **SQLiteDb** ».

Ouvrez le fichier « **SQLiteDb.cs** » et ajoutez le code suivant :

```
class MySQLiteDb
{
      public const string DatabaseFilename = "dbcoach.db3";

      public static string DatabasePath = Path.Combine(FileSystem.AppDataDirectory,
      DatabaseFilename);

      public const SQLiteOpenFlags Flags = SQLiteOpenFlags.Create |
      SQLiteOpenFlags.ReadWrite | SQLiteOpenFlags.SharedCache;

      private SQLiteAsyncConnection connection;
}
```

Ces variables et la constante « **SQLiteOpenFlags** » vont nous permettre de créer une connexion vers une base de données SQLite qui sera stockée sur votre terminal Android dans le fichier « **dbcoach.db3** ». Cette connexion va permettre la création de tables, la lecture et l'écriture de données dans la base.

Nous allons utiliser la programmation asynchrone pour effectuer toutes les opérations de notre application qui font appel à la base de données. Cela permettra à l'utilisateur de garder le contrôle sur l'application, y compris quand l'accès aux données prend du temps.

Toujours dans le fichier « **SQLiteDb.cs** », nous allons commençons par créer une méthode asynchrone **« Initialize()** » de type « **Task** », c'est-à-dire qu'elle retourne une tâche (promise) et pas de valeur :

```
private async Task Initialize()
{
        if (connection is not null)
            return;

        connection = new(DatabasePath);

        await connection.CreateTableAsync<Profil>();
}
```

Si notre connexion à la base de données n'existe pas encore, on crée une nouvelle connexion avec la base de données SQLite. On crée ensuite la table qui correspond au modèle « **Profil** » (notre classe métier), là aussi si elle n'existe pas encore.

On peut maintenant ajouter une première méthode qui va enregistrer un profil dans la base de données :

```
public async Task<int> SaveItemAsync(Profil unProfil)
{
        await Initialize();

        if (unProfil.Id != 0)
        {
            return await connection.UpdateAsync(unProfil);
        }
        else
        {
            return await connection.InsertAsync(unProfil);
        }
    }
}
```

Ajoutons ensuite la méthode « **GetLastItemQueryAsync()** » qui va récupérer le dernier profil enregistré dans la base de données :

```
public async Task<Profil> GetLastItemQueryAsync()
{
        await Initialize();
        return await connection.FindWithQueryAsync<Profil>("SELECT * FROM Profil ORDER BY
        dateMesure DESC");
}
```

Il nous reste une dernière méthode à créer, qui sera bien utile par la suite. Ajoutez la méthode « **QueryGetAllItemsAsync()** » qui va récupérer tous les profils enregistrés dans la base de données, triés par date d'ajout :

```
public async Task<List<Profil>> QueryGetAllItemsAsync()
    {
        await Initialize();
        return await connection.QueryAsync<Profil>("SELECT * FROM Profil ORDER BY id DESC");
    }
```

Modifions maintenant le fichier « **MainPage.xaml.cs** » pour récupérer le dernier profil dans la base SQLite. Remarquez l'utilisation, là aussi, d'une **méthode asynchrone** :

```
private Profil unProfil = null;
private readonly SQLiteDb SQLiteDbCoach = null;
public MainPage()
{
     InitializeComponent();
     SQLiteDbCoach = new SQLiteDb();
     SqliteSelect();
}
```

```
private async void SqliteSelect()
{
     unProfil = await SQLiteDbCoach.GetLastItemQueryAsync();

     if (unProfil != null)
     {
         entPoids.Text = unProfil.Poids.ToString();
         entTaille.Text = unProfil.Taille.ToString();
         entAge.Text = unProfil.Age.ToString();

         if (unProfil.Sexe == 1)
         {
             rbHomme.IsChecked = true;
         }
         else
         {
             rbFemme.IsChecked = true;
         }
         //Affichage du résultat
         AfficheResultat(unProfil);
     }
}

…
```

Enfin, toujours dans « **MainPage.xaml.cs** », ajoutez le code nécessaire pour enregistrer le profil dans la base SQLite.

On commence par ajouter la méthode asynchrone « **SqliteInsert()** » :

```
private async void SqliteInsert(Profil unProfil)
{
    int id = await SQLiteDbCoach.SaveItemAsync(unProfil);
}
```

Puis on ajoute l'instruction `SqliteInsert(unProfil); //Enregistrement du profil dans la bdd` après la création du profil dans la méthode « **btCalculer_Clicked()** » :

```
private void btCalculer_Clicked(object sender, EventArgs e)
{
     try
     {
         DateTimeOffset date = DateTimeOffset.Now;
         var poids = int.Parse(entPoids.Text);
         var taille = int.Parse(entTaille.Text);
         var age = int.Parse(entAge.Text);
         var sexe = 0;
```

```
        if (rbHomme.IsChecked)
        {
            sexe = 1;
        }

        if (rbFemme.IsChecked)
        {
            sexe = 0;
        }

        Profil unProfil = new Profil(null, date, poids, taille, age, sexe);

        SqliteInsert(unProfil); //Enregistrement du profil dans la bdd

        AfficheResultat(unProfil); //Affichage du résultat
    }
    catch (Exception ex)
    {
        DisplayAlertAsync("Erreur", "Saisie incorrecte", "Ok");
    }
}
```

Testez votre application et vérifiez que l'enregistrement et la récupération des données dans la base locale SQLite fonctionnent correctement.

Si tout va bien, vous pouvez passer à la suite …