

Vous allez développer une application cliente simple (WinUI) qui permettra de convertir un montant saisi en euros vers la devise sélectionnée (la liste de devises sera alimentée par le WS réalisé en partie 1). La même application sera par la suite redéveloppée en appliquant le pattern MVVM.

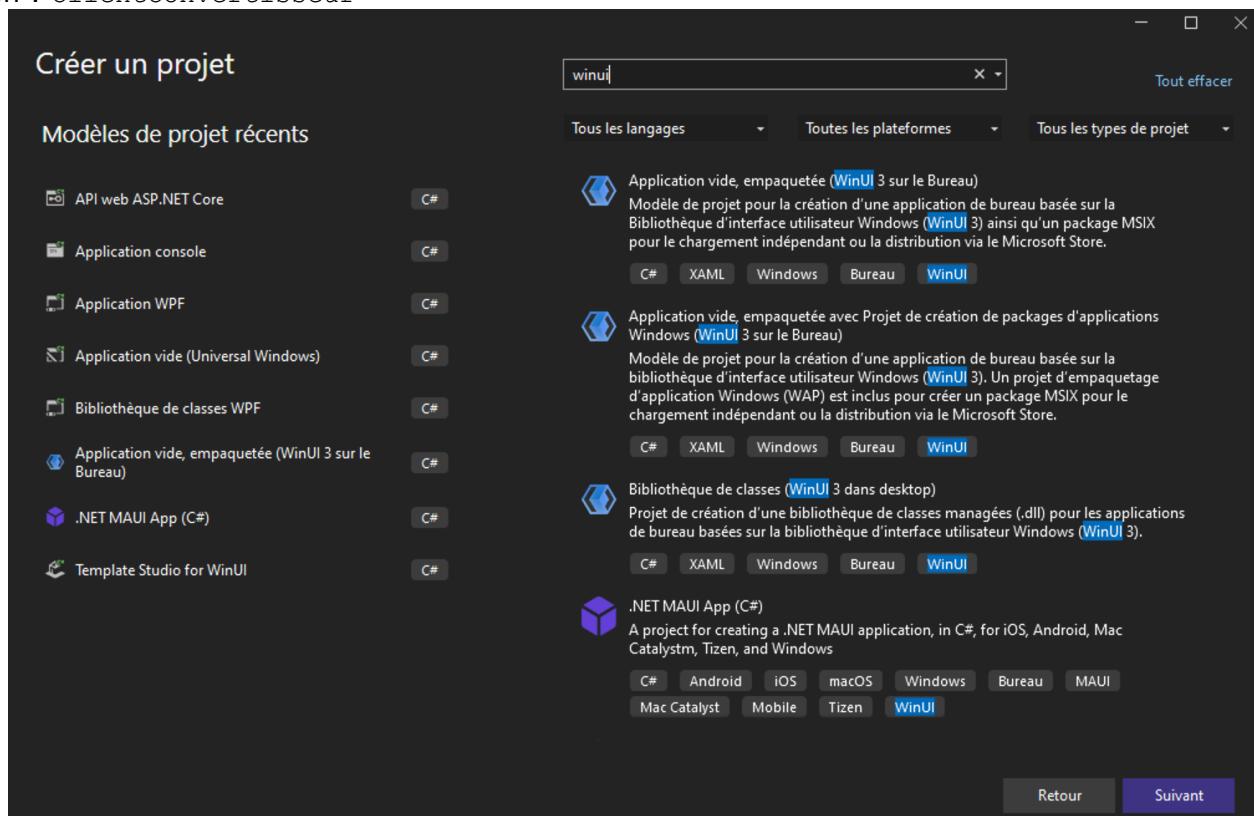
### 1. Création de l'application cliente lourde (WinUI) – V1 (version simple sans MVVM)

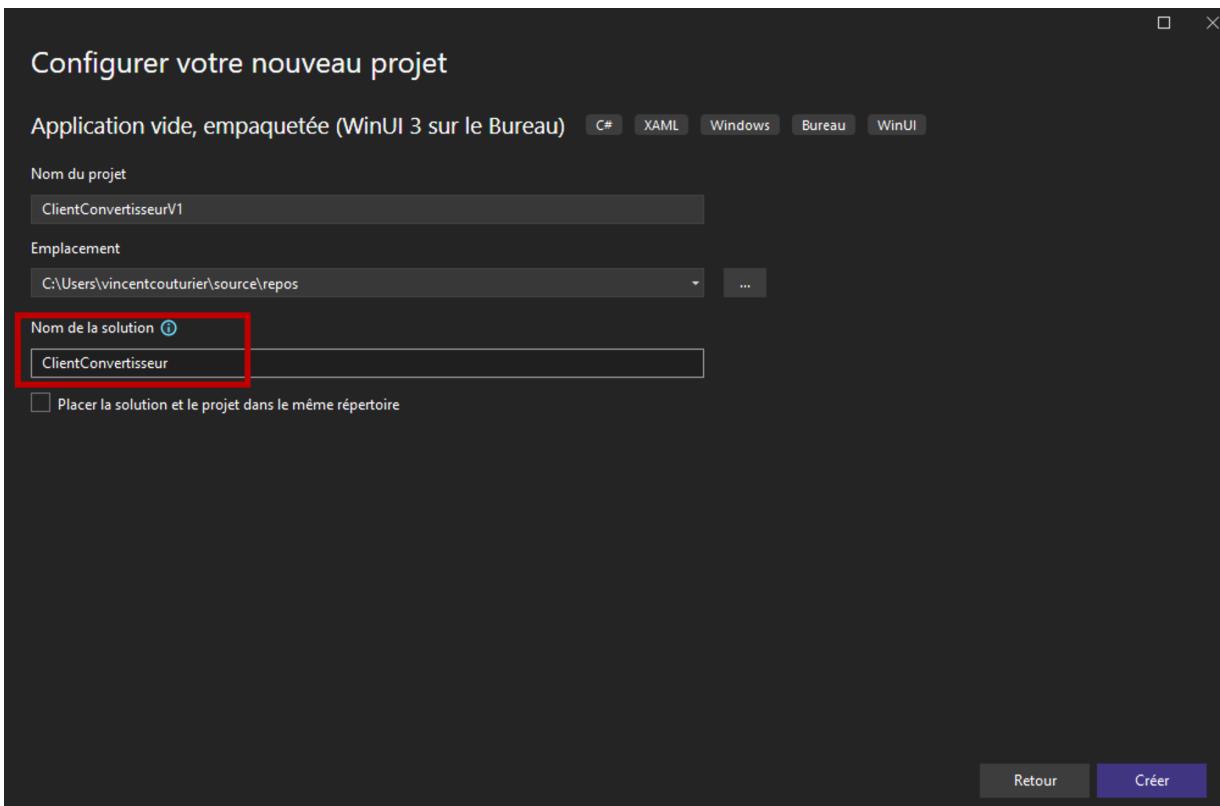
Une application WinUI doit être développée sous Windows.

Lancer une nouvelle instance de Visual Studio.

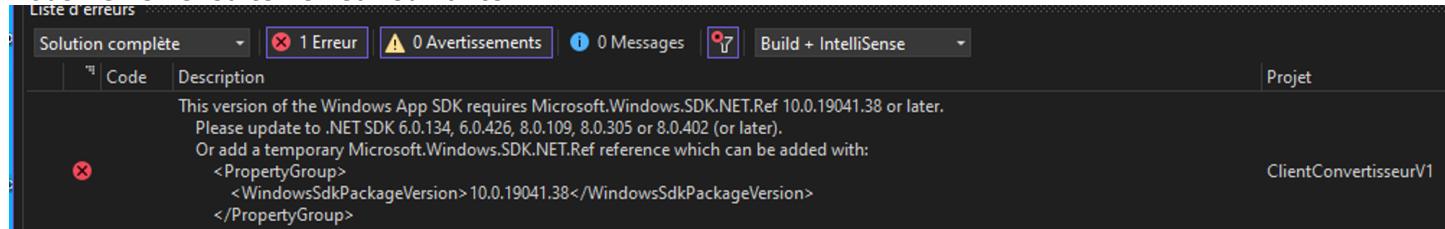
Créer un projet « Application vide empaquetée (WinUI 3 sur le Bureau) » dans une nouvelle solution **sur le bureau**. Vous pourrez le nommer ClientConvertisseurV1.

Solution : ClientConvertisseur





Vous verrez ensuite l'erreur suivante :



Double-cliquer sur le nom du projet. Ajouter la balise suivante :

```
<WindowsSdkPackageVersion>10.0.19041.38</WindowsSdkPackageVersion>
```

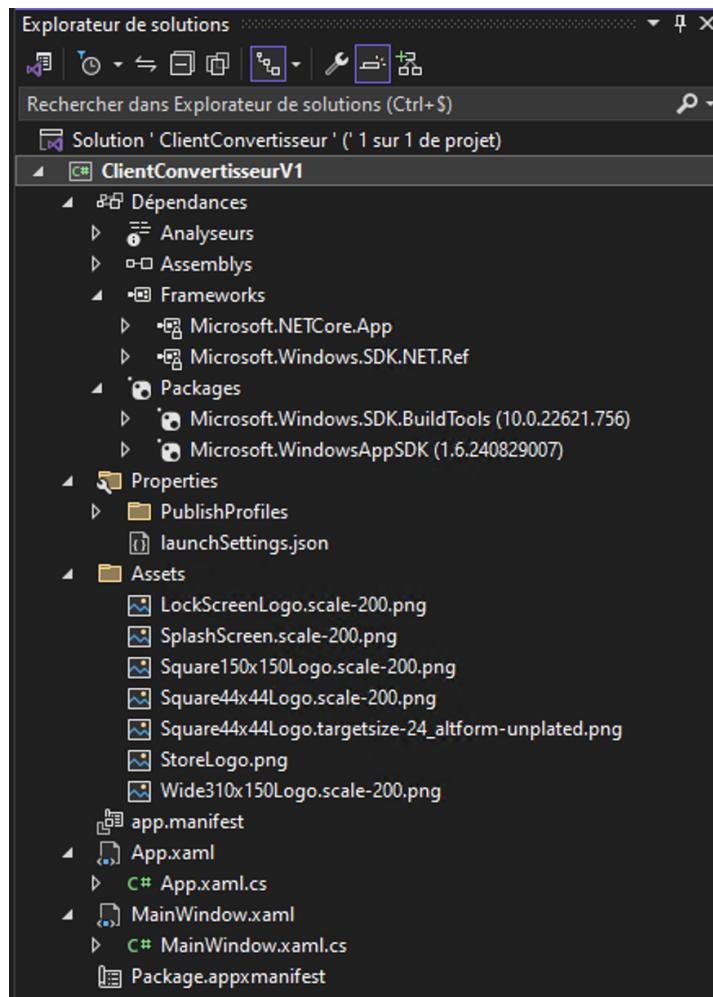
```
<PropertyGroup>
  <WindowsSdkPackageVersion>10.0.19041.38</WindowsSdkPackageVersion>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net8.0-windows10.0.19041.0</TargetFramework>
  <TargetPlatformMinVersion>10.0.17763.0</TargetPlatformMinVersion>
  <RootNamespace>App1</RootNamespace>
  <ApplicationManifest>app.manifest</ApplicationManifest>
  <Platforms>x86;x64;ARM64</Platforms>
  <RuntimeIdentifiers Condition="$(MSBuild)::GetTargetFrameworkVersion('$
```

Exécuter le projet pour charger le package. Si une erreur indiquant que le package n'a pas été chargé, le réexécuter.

## Configurer un nouveau repos GitHub.

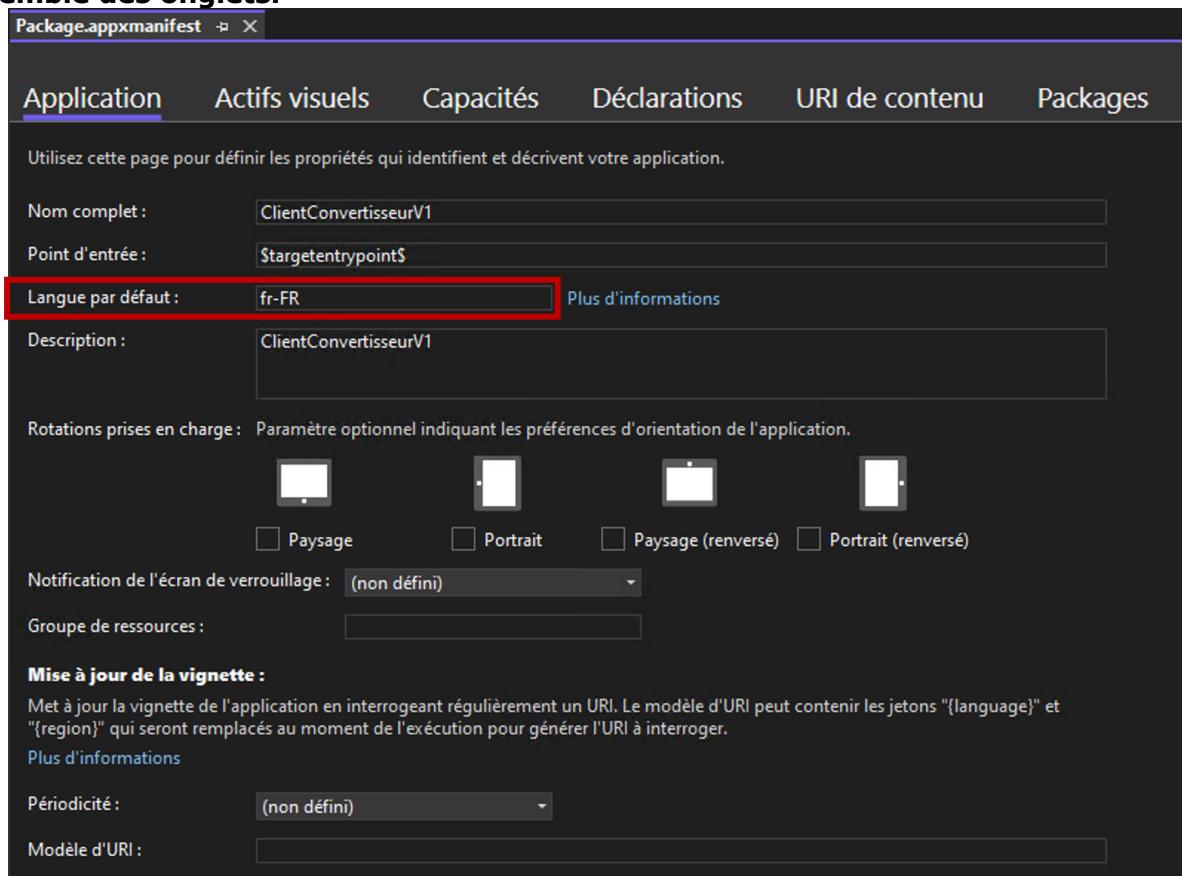
Le modèle *Application vide* crée une application du Windows Store vide qui se compile et s'exécute, mais qui ne contient aucun contrôle d'interface utilisateur ni aucune donnée. Vous ajouterez des contrôles et des données à l'application par la suite.

Solution générée :



Même vide, un projet WinUI contient les fichiers suivants :

- Un fichier manifeste (`Package.appxmanifest`) qui décrit l'application (nom, description, vignette, page de démarrage, etc.) et répertorie les fichiers contenus dans l'application. **Regarder l'ensemble des onglets.**



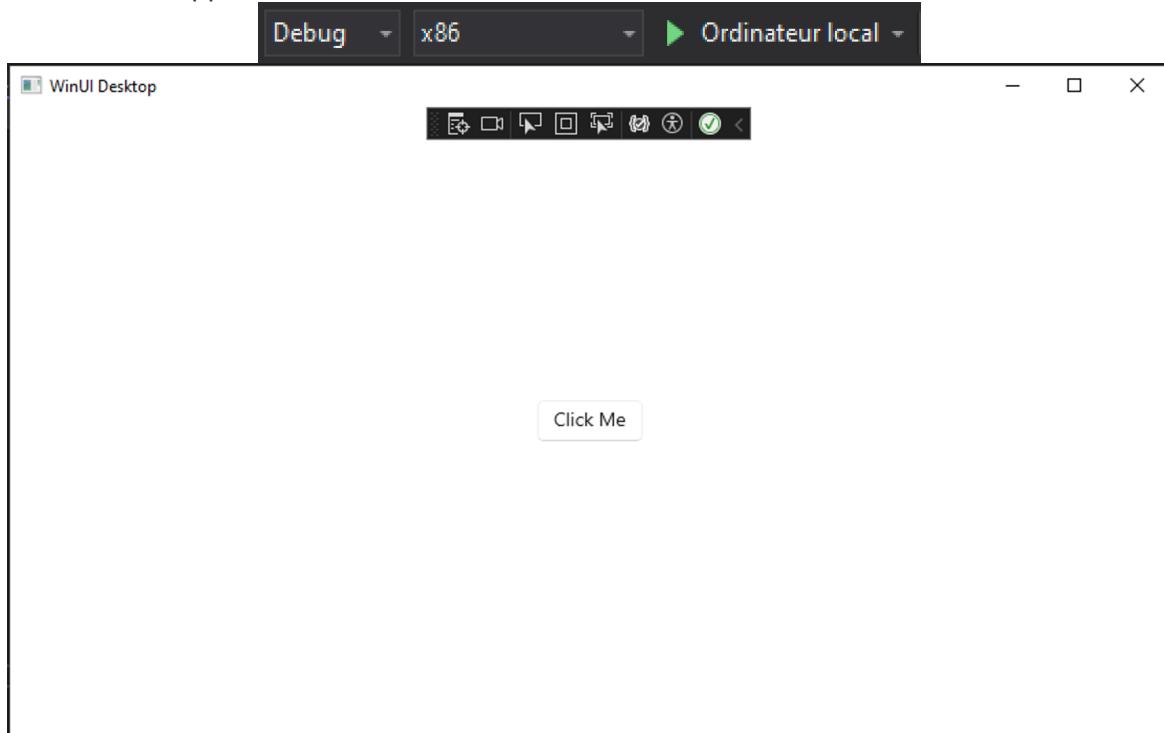
*Remarque : La première page à s'afficher lorsqu'on démarre une application est la page MainPage.xaml. Ceci est modifiable dans le fichier App.xaml.cs.*

- Un ensemble d'images de logos de petit et grand format à afficher (dans le dossier Assets).
- Les fichiers XAML et de code de l'application (App.xaml et App.xaml.cs).
- Une page de démarrage (MainPage.xaml) et un fichier de code associé (MainPage.xaml.cs) qui s'exécute au démarrage de votre application.
- Le dossier Dépendances qui intègre les dépendances et le framework utilisés.

Ces fichiers sont essentiels à toutes les applications de type Windows qui doivent se retrouver dans le Windows Store.

Quel que soit le périphérique utilisé (smartphone, Xbox, tablette, PC), le code de l'application sera le même. Ainsi, c'est Windows qui adaptera l'application au périphérique utilisé (résolution, etc.).

Nous allons exécuter l'application. L'exécuter d'abord sur l'ordinateur local.



Dans ce cas, il s'agit d'une application WinUI « classique » (i.e. pour PC).

Il sera possible de redimensionner le fenêtre pour voir comment se positionnent les différents contrôles.



## ***Explication des fichiers***

Bien regarder le contenu de chaque fichier.

App.xaml

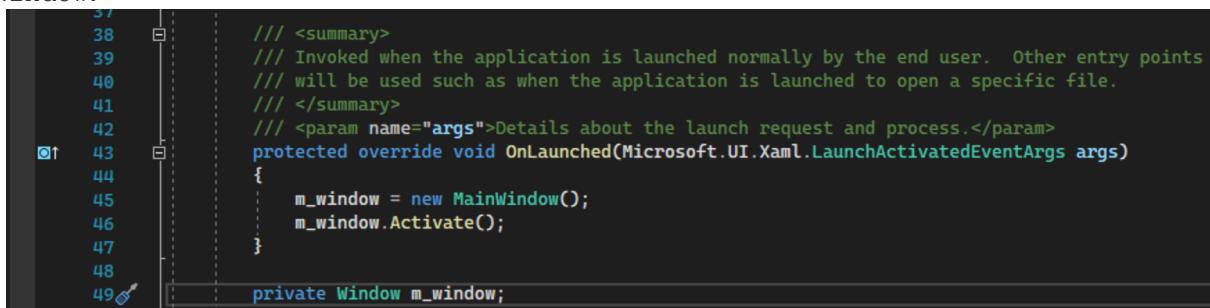
App.xaml est le fichier dans lequel vous déclarez les ressources utilisées dans l'application. Il est possible de changer le thème (Dark vs Light) en ajoutant la balise RequestedTheme.

```
App.xaml* ➔ X
Application
1  ↳ <Application
2    ↳ x:Class="ClientConvertisseurV1.App"
3    ↳ xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
4    ↳ xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
5    ↳ xmlns:local="using:ClientConvertisseurV1"
6    ↳ RequestedTheme="Dark">
7      ↳ <Application.Resources>
8        ↳ <ResourceDictionary>
9          ↳ <ResourceDictionary.MergedDictionaries>
10         ↳ <XamlControlsResources xmlns="using:Microsoft.UI.Xaml.Controls" />
11         ↳ <!-- Other merged dictionaries here -->
12       ↳ </ResourceDictionary.MergedDictionaries>
13       ↳ <!-- Other app resources here -->
14     ↳ </ResourceDictionary>
15   ↳ </Application.Resources>
16 </Application>
17
```

App.xaml.cs

App.xaml.cs est le fichier code-behind de App.xaml. Le code-behind est le code joint à la classe partielle de la page XAML. Ensemble, la page XAML et le code-behind forment une classe complète. Comme toutes les pages code-behind, elle contient un constructeur qui appelle la méthode InitializeComponent. Cette

méthode est générée par Visual Studio et vise essentiellement à initialiser les éléments déclarés dans le fichier XAML. App.xaml.cs contient par ailleurs une méthode destinée à lancer la 1<sup>ère</sup> page de l'application, ici MainWindow.



```
37
38     /// <summary>
39     /// Invoked when the application is launched normally by the end user. Other entry points
40     /// will be used such as when the application is launched to open a specific file.
41     /// </summary>
42     /// <param name="args">Details about the launch request and process.</param>
43     protected override void OnLaunched(Microsoft.UI.Xaml.LaunchActivatedEventArgs args)
44     {
45         m_window = new MainWindow();
46         m_window.Activate();
47     }
48
49     private Window m_window;
```

MainWindow.xaml

Le fichier MainWindow.xaml contient l'interface utilisateur de l'application. Vous pouvez ajouter des éléments directement en utilisant du balisage XAML ou les outils de conception fournis avec Visual Studio (contrôles de la boîte à outils). Le modèle « Application vide » crée une nouvelle classe appelée MainPage qui hérite de Window.

MainWindow.xaml.cs

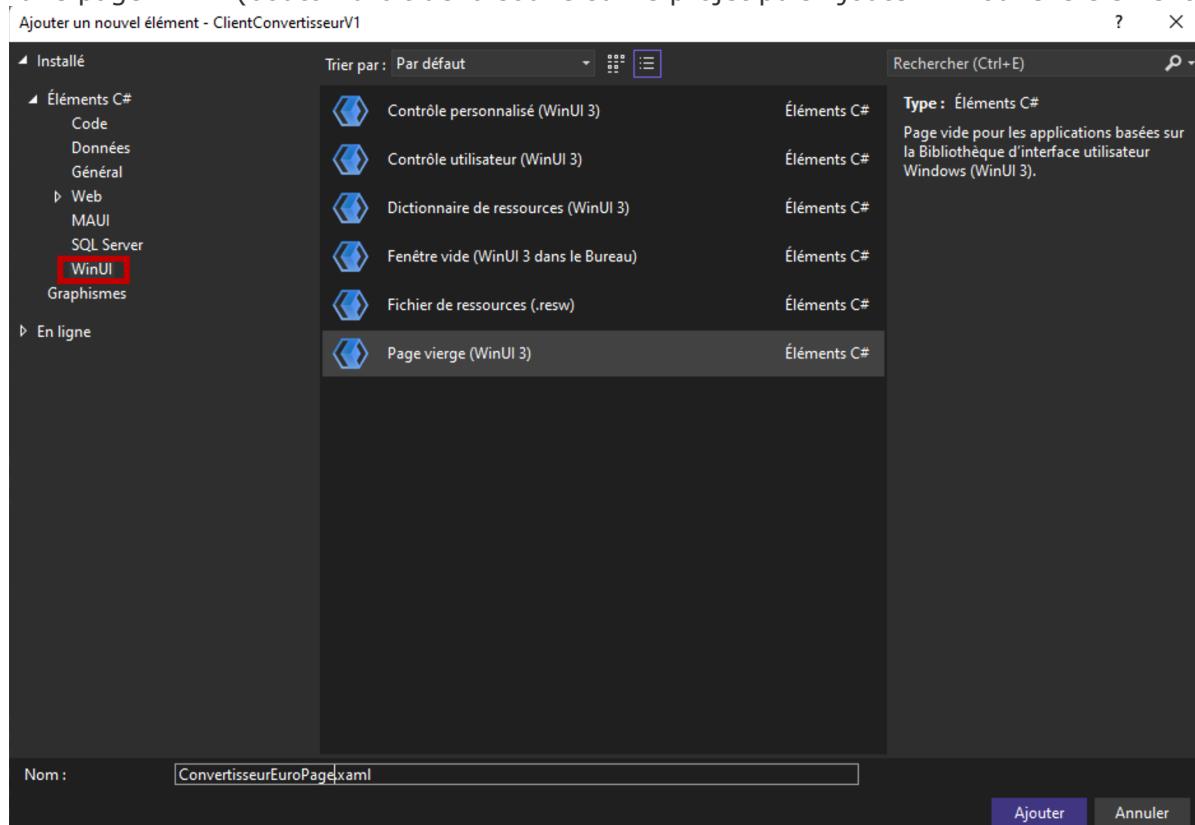
MainWindow.xaml.cs est la page code-behind de MainWindow.xaml. C'est ici que vous ajoutez la logique de votre application et les gestionnaires d'événements. Il existe plusieurs moyens de gérer des applications de type MDI (interfaces multi-documents) :

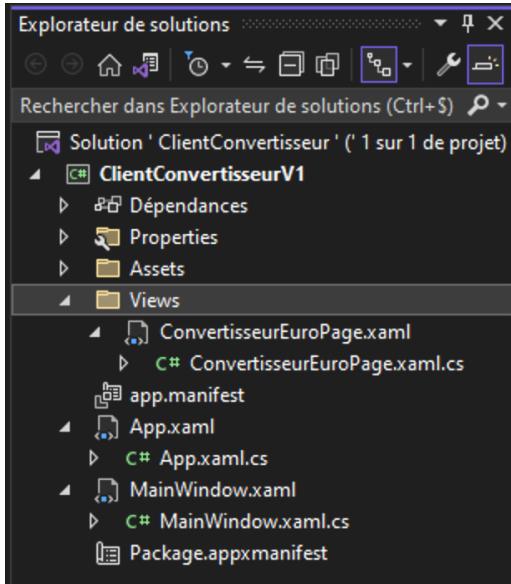
1. Création de plusieurs fenêtres (de type Window). Chaque Window venant se positionner l'une au-dessus de l'autre.
2. Création de pages au sein d'une seule Window.

Nous allons procéder de la seconde manière.

Créer un dossier Views dans l'application (bouton droit de la souris sur le projet puis *Ajouter > Nouveau dossier*).

Y ajouter une page XAML (bouton droit de la souris sur le projet puis *Ajouter > Nouvelle élément*) :





Pour charger la page XAML créée, le code de la méthode `OnLaunched` du fichier `App.xaml.cs` devient :

```

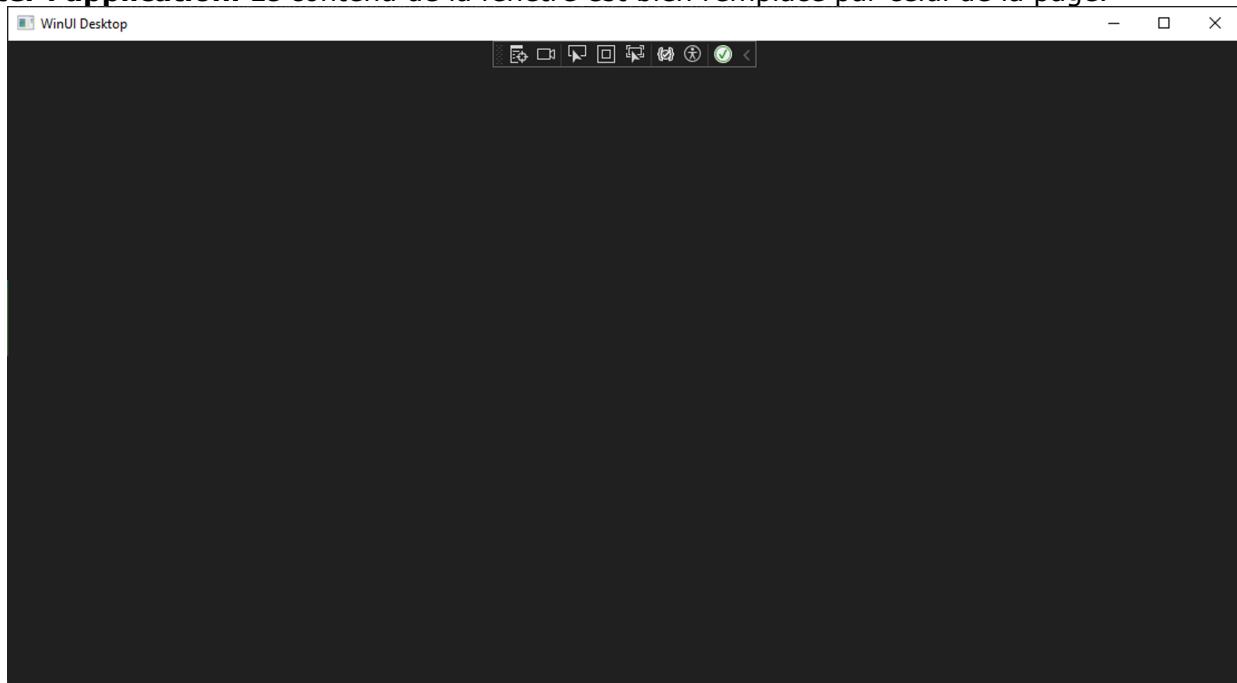
/// <summary>
/// Invoked when the application is launched normally by the end user. Other entry points
/// will be used such as when the application is launched to open a specific file.
/// </summary>
/// <param name="args">Details about the launch request and process.</param>
protected override void OnLaunched(Microsoft.UI.Xaml.LaunchActivatedEventArgs args)
{
    m_window = new MainWindow();
    // Create a Frame to act as the navigation context and navigate to the first page
    Frame rootFrame = new Frame();
    // Place the frame in the current Window
    this.m_window.Content = rootFrame;
    // Ensure the current window is active
    m_window.Activate();
    //Navigate to the first page
    rootFrame.Navigate(typeof(ConvertisseurEuroPage));
}

private Window m_window;

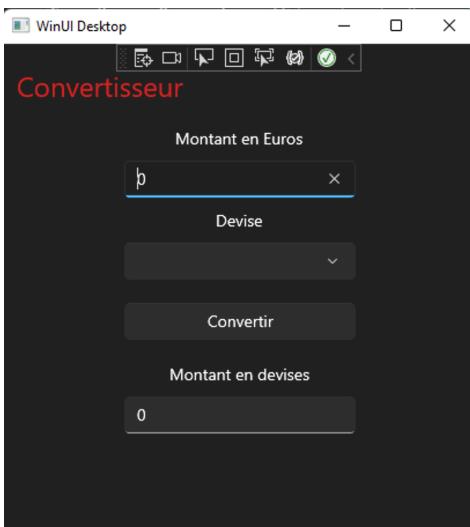
```

Nous verrons comment améliorer ce code de lancement ultérieurement.

**Exécuter l'application.** Le contenu de la fenêtre est bien remplacé par celui de la page.



## Visuel de l'application



Nous réaliserons cette interface plus tard...

## Codage de la classe d'accès à l'API REST

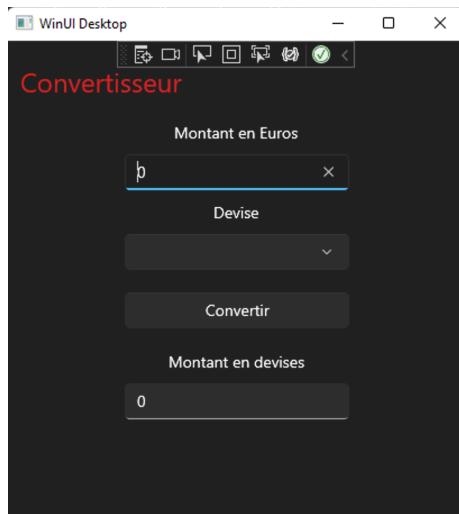
1. Créer un dossier `Models` et y ajouter la classe `Devise` correspondant à celle du WS (supprimer l'annotation `[Required]`).
2. Créer un dossier `Services`. Dans ce dossier :
  - o Créer l'interface `IService`. L'interface ne contiendra qu'une méthode permettant de récupérer les devises : `Task<List<Devise>> GetDevisesAsync(string nomControleur);`  
*Remarque : Plus de détails sur `async/await` et les tasks asynchrones : <http://fdorin.developpez.com/tutoriels/csharp/threadpool/part3/>*
  - o Créer une classe `WSService` implémentant `IService` :
    - Créer le constructeur de la classe `WSService`
    - Dans le constructeur de la classe, instancier un objet de type `HttpClient`, par exemple nommé `client` et le configurer (`BaseAddress`, etc.). Exemple de code ici (récupérer le code contenu dans la méthode `RunAsync` et l'intégrer dans le constructeur) :  
<https://learn.microsoft.com/fr-fr/aspnet/web-api/overview/advanced/calling-a-web-api-from-a-net-client>

```
C#                                     Copier

static async Task RunAsync()
{
    // Update port # in the following line.
    client.BaseAddress = new Uri("http://localhost:64195/");
    client.DefaultRequestHeaders.Accept.Clear();
    client.DefaultRequestHeaders.Accept.Add(
        new MediaTypeWithQualityHeaderValue("application/json"));
```

- L'adresse à indiquer devra être celle de votre API, construite de la façon suivante : <https://localhost:PORT/api/> Ex. : <https://localhost:7223/api/>
- Ne pas indiquer le nom du contrôleur dans l'Uri.
- Classe `HttpClient` :  
<https://docs.microsoft.com/fr-fr/dotnet/api/system.net.http.httpclient>
- Améliorer le code en passant l'Uri (de type `string`) en paramètre du constructeur (constructeur paramétré).
- Coder la méthode `public async Task<List<Devise>> GetDevisesAsync(string nomControleur)` :  
`public async Task<List<Devise>> GetDevisesAsync(string nomControleur)
{
 try
 {
 return await httpClient.GetFromJsonAsync<List<Devise>>(nomControleur);
 }
}`





### Codage de l'alimentation de la ComboBox

Procéder comme en R3.04 (data binding).

*N'oubliez pas que le mode par défaut du Binding en WinUI est OneWay (de la source vers la cible, c'est-à-dire du code-behind vers la vue). Et n'oubliez pas d'indiquer à la vue qu'elle se mette à jour...*

La création d'un objet `WSService` et l'appel de la méthode `GetDevisesAsync` se fera au chargement de la page :

```
public ConvertisseurEuroPage()
{
    this.InitializeComponent();
    this.DataContext = this;
    GetDataOnLoadAsync();
}

private async void GetDataOnLoadAsync()
{
    WSService service = new WSService("https://localhost:7243/api/");
    List<Devise> result = await service.GetDevisesAsync("devises");
    if (result == null)
        MessageAsync("API non disponible !", "Erreur");
    else
        Devises = new ObservableCollection<Devise>(result);
}
```

Pour la gestion des erreurs (appel de la méthode `MessageAsync`, voir page suivante).

Lancer l'application cliente pour tester le chargement de la ComboBox (ne pas oublier d'exécuter le WS).

### Codage du calcul

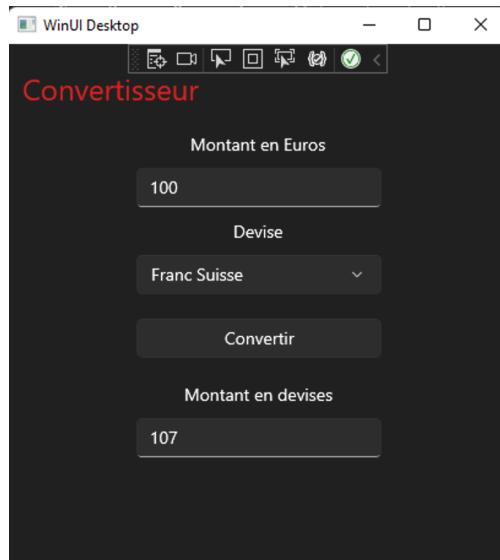
Comme il n'y a plus de mode *design*, pour ajouter un événement sur un bouton, ajouter la balise `Click` et le nom de l'évènement, puis cliquer sur `<Nouveau gestionnaire d'évènements>` :

```
20 <Button x:Name="BtnConvertir" Content="Convertir" Click="BtnConvertir_Click" Width="200" Margin="10, 10, 10, 10>
Liez l'événement à une nouvelle méthode nommée 'BtnConvertir_Click'. Utilisez 'Atteindre la définition' <Nouveau gestionnaire d'évènements>
pour accéder à la méthode qui vient d'être créée.
```

A vous de jouer pour le codage...

Lancer l'application cliente (ne pas oublier d'exécuter le WS).

Résultat :



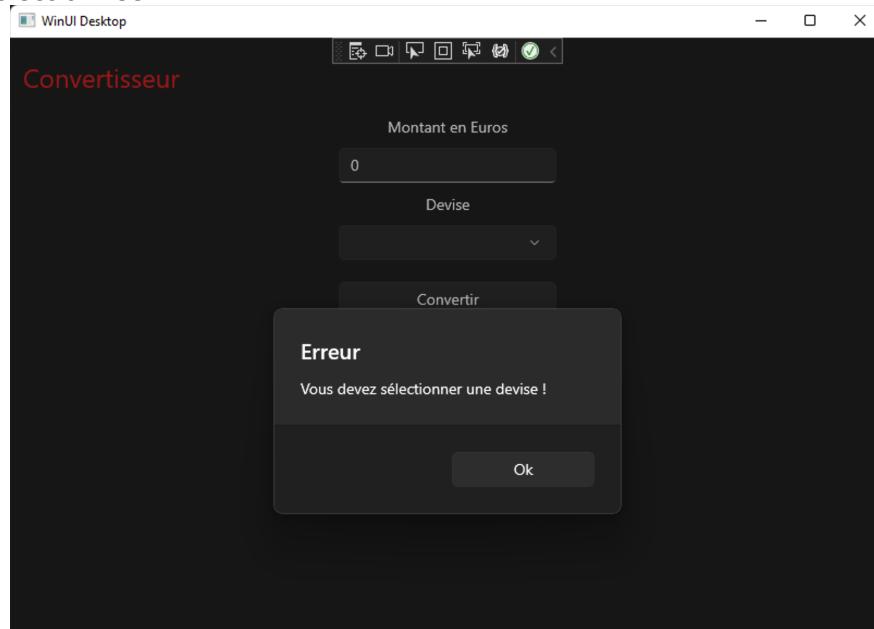
Vérifier le responsive design de l'application.

### Gestion des erreurs

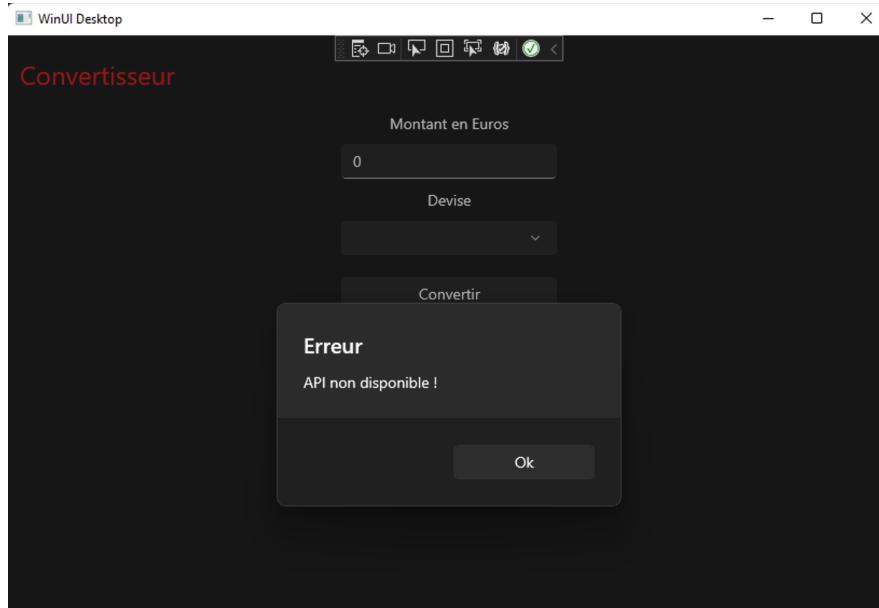
Gérer les erreurs (devise non saisie, API non disponible). Vous pourrez utiliser la classe `Windows.UI.Xaml.Controls.ContentDialog` permettant d'afficher des fenêtres popup. Vous créerez la méthode `ShowAsync()`, méthode asynchrone à utiliser avec le couple `async/await`. Penser à écrire du code générique...

Exemple de code ici : <https://learn.microsoft.com/fr-fr/windows/apps/design/controls/dialogs-and-flyouts/dialogs>  
ATTENTION, comme le message est lancé à partir d'une frame et non de la fenêtre, il est nécessaire d'ajouter le code suivant avant d'afficher le message (c'est-à-dire avant d'appeler la méthode `ShowAsync`) :  
`maContentDialog.XamlRoot = this.Content.XamlRoot;`

Erreur Devise non sélectionnée :



Erreur API non disponible (si la liste de devises renvoyée à l'interface graphique est null) :

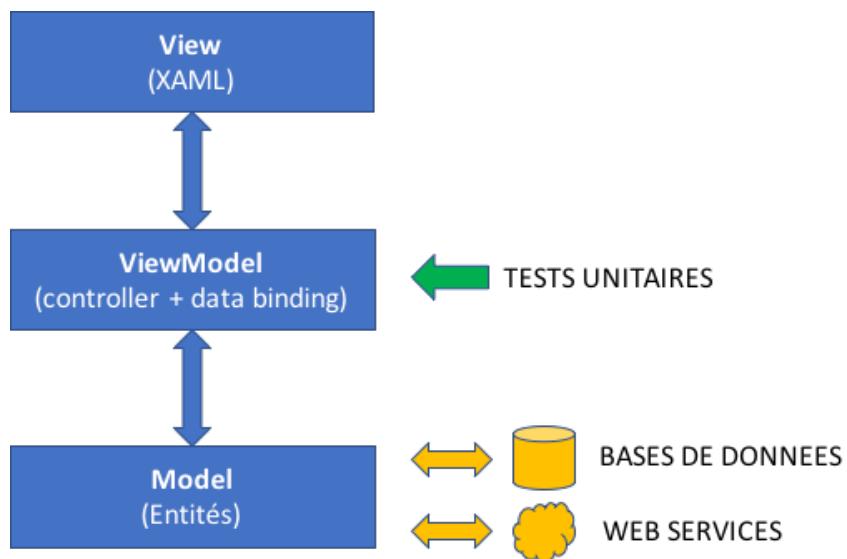


## **2. Création de l'application cliente lourde – V2 avec MVVM**

### **MVVM ?**

MVVM signifie *Model-View-ViewModel* :

- *Model* correspond aux données. Il s'agit en général de plusieurs classes qui permettent d'accéder aux données, comme une classe *Client*, une classe *Commande*, etc. Peu importe la façon dont on remplit ces données (base de données, service web,...), c'est ce modèle qui est manipulé pour accéder aux données.
- *View* correspond à tout ce qui sera affiché, comme la page, les boutons, etc. En pratique, il s'agit du fichier *.xaml*.
- *ViewModel*, que l'on peut traduire en « modèle de vue », constitue la colle entre le modèle et la vue. Il s'agit d'une classe qui fournit une abstraction de la vue. Ce modèle de vue s'appuie sur la puissance du binding pour mettre à disposition de la vue les données du modèle. Il s'occupe également de gérer les commandes (actions événementielles) que nous verrons un peu plus loin.



Le but de MVVM est de faire en sorte que la vue n'effectue aucun traitement : elle ne doit faire qu'afficher les données présentées par le ViewModel. C'est le ViewModel qui est chargé de réaliser les traitements et d'accéder au modèle.

### ***Création du projet et installation des packages NuGet***

Vincent COUTURIER

Ajouter un nouveau projet nommé ClientConvertisseurV2 de type « Application vide empaquetée (WinUI 3) » à la solution (bouton droit de la souris sur le nom de la solution puis *Ajouter* puis *Nouveau projet*). Définir ce projet comme projet de démarrage (bouton droit de la souris sur le projet puis *Définir comme projet de démarrage*).

Ajouter la balise <WindowsSdkPackageVersion>10.0.19041.38</WindowsSdkPackageVersion> au fichier XML projet.

Ajouter le package NuGet « CommunityToolkit.Mvvm » (**dernière version stable**) au projet (Menu *Outils* > *Gestionnaire de packages NuGet* > *Gérer les packages NuGet pour la solution*).

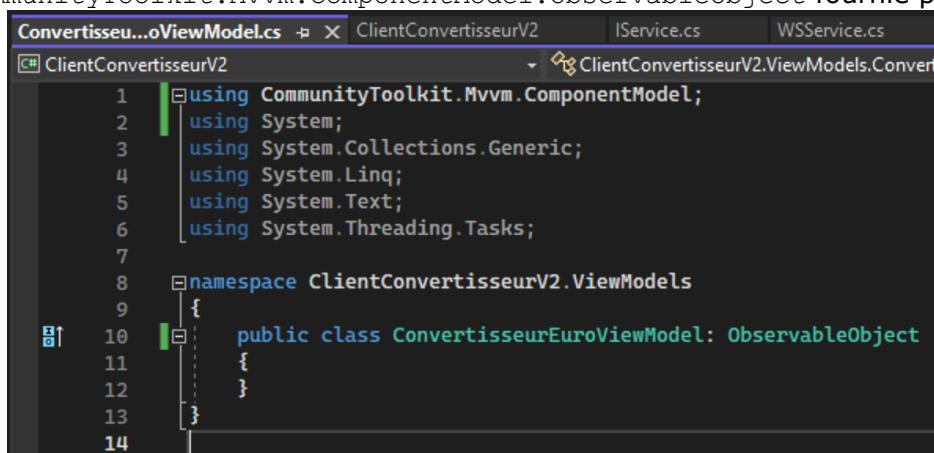
*CommunityToolkit.Mvvm* (qui remplace *Microsoft.Toolkit.Mvvm*) est un framework qui va nous aider à mettre en place le pattern MVVM. Il fournit notamment une architecture de projet compatible MVVM.

## Couches Model et Service

Créer les dossiers « Models » et « Services ». Y créer les mêmes classes et interfaces que pour le client V1. Modifier si nécessaire les namespace.

## Couches ViewModel et View

Créer un dossier *ViewModels*. Ajouter une classe nommée *ConvertisseurEuroViewModel* héritant de la classe de base *CommunityToolkit.Mvvm.ComponentModel.ObservableObject* fournie par MVVM Toolkit :



```
Convertisseur...oViewModel.cs  X | ClientConvertisseurV2           IService.cs      WSService.cs
C# ClientConvertisseurV2
1  using CommunityToolkit.Mvvm.ComponentModel;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using System.Threading.Tasks;
7
8  namespace ClientConvertisseurV2.ViewModels
9  {
10    public class ConvertisseurEuroViewModel: ObservableObject
11    {
12    }
13
14 }
```

Comme on peut le voir dans la documentation, cette classe implémente l'interface *INotifyPropertyChanged*. Nous n'aurons donc plus besoin de cette interface (comme vous pouvez le voir sur la copie d'écran ci-dessus), ni de l'évènement *PropertyChangedEventHandler*, ni de l'implémentation de la méthode *void OnPropertyChanged(string name)*:

<https://learn.microsoft.com/fr-fr/windows/communitytoolkit/mvvm/observableobject>

# ObservableObject

Article • 14/03/2024 • 2 contributeurs

[Commentaires](#)

## Dans cet article

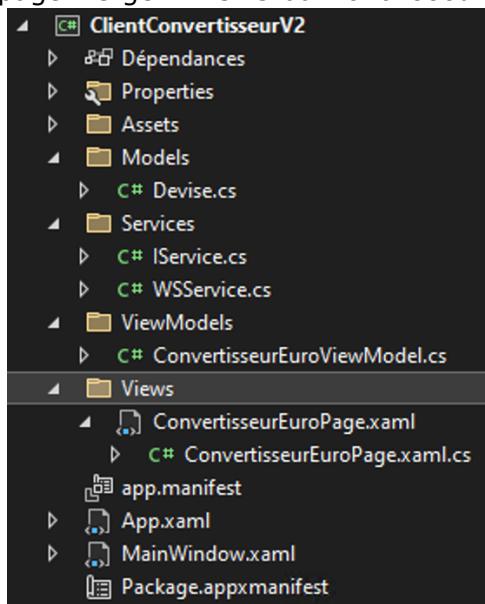
- [Fonctionnement](#)
- [Propriété simple](#)
- [Enveloppement d'un modèle non observable](#)
- [Gestion des propriétés Task<T>](#)
- [Exemples](#)

ObservableObject est une classe de base pour les objets qui sont observables en implémentant les interfaces [INotifyPropertyChanged](#) et [INotifyPropertyChanging](#). Elle peut servir de point de départ pour tous les types d'objets qui doivent prendre en charge les notifications de modification de propriété.

API de plateforme : ObservableObject, TaskNotifier, TaskNotifier<T>

ConvertisseurEuroViewModel est le ViewModel de notre page ConvertisseurEuroPage.xaml (que nous allons créer). La philosophie du MVVM demande de limiter au maximum le code dans le fichier code behind de la page (seul le code de navigation entre les pages est en général conservé dans le code behind, même s'il est possible de la gérer en code-behind). C'est donc ce ViewModel qui coordonnera les échanges entre la page et le modèle.

Créer un dossier Views. Y créer la page vierge WinUI 3 ConvertisseurEuroPage.xaml.



Modifier également le fichier App.xaml.cs permettant de lancer la page ConvertisseurEuroPage. Copier-coller le code XAML du client V1 dans la page ConvertisseurEuroPage.xaml du client V2. Supprimer le code événementiel (ex. `click="..."`).

Exécuter l'application pour voir si la page s'affiche. La page est toujours chargée grâce à l'appel réalisé dans le fichier App.xaml.cs, mais pour le moment aucun.viewmodel n'est associé à cette page et donc aucun code n'est exécuté.

Rajouter maintenant le code suivant dans le fichier .cs de la vue :

```

20  namespace ClientConvertisseurV2.Views
21  {
22      /// <summary>
23      /// An empty page that can be used on its own or navigated to within a Frame.
24      /// </summary>
25      public sealed partial class ConvertisseurEuroPage : Page
26      {
27          public ConvertisseurEuroPage()
28          {
29              this.InitializeComponent();
30              ConvertisseurEuroViewModel convertisseurEuroViewModel = new ConvertisseurEuroViewModel();
31              DataContext = convertisseurEuroViewModel;
32          }
33      }
34  }

```

Ce code permet de lier la vue à son ViewModel et de définir le DataContext qui est le ViewModel créé. Nous verrons une façon beaucoup plus propre d'écrire ce code plus loin.

Ajout du code dans le fichier `ConvertisseurEuroViewModel.cs` : récupérer le code des properties et de la méthode `GetDataOnLoadAsync` de la V1 et le coller dans le ViewModel. Créer le constructeur du ViewModel et appeler la méthode `GetDataOnLoadAsync`. Supprimer le code C# (en erreur) lié à la gestion des erreurs. Dans le cas du MVVM (et plus exactement de la classe `ObservableObject`), on n'est pas obligé de spécifier le nom de la property lors de l'appel de la méthode  `OnPropertyChanged()`, donc :

`OnPropertyChanged(nameof(MaProperty)); -> OnPropertyChanged();`

Lancer l'application. Normalement, la liste des devises devrait s'afficher (ne pas oublier d'exécuter le WS).

Codage de l'action sur le bouton :

Nous allons gérer une commande sur le bouton. En effet, avec le découpage View / ViewModel, le ViewModel n'est pas au courant d'une action sur l'interface, car c'est un fichier à part. Il n'est donc pas directement possible de réaliser une action dans le ViewModel lors d'un clic sur le bouton. Les commandes correspondent à des actions faites sur la vue, comme un clic sur un bouton. Le XAML dispose d'un mécanisme simple de gestion de commandes via l'interface `IRelayCommand` (<https://learn.microsoft.com/fr-fr/dotnet/api/microsoft.toolkit.mvvm.input irelaycommand>). Par exemple, le contrôle `Button` possède (par héritage) une propriété `Command` du type `IRelayCommand` (<https://learn.microsoft.com/fr-fr/windows/communitytoolkit/mvvm/relaycommand>) permettant d'invoquer une commande lorsque le bouton est appuyé.

La classe `RelayCommand` permet ensuite de lier une commande à une action, i.e. une méthode.

1. Dans le fichier XAML de la vue, ajouter le code suivant :

`<Button Content="Convertir" ... Command="{Binding BtnSetConversion}" .../>`

2. Dans le ViewModel, ajouter le code suivant permettant de gérer le bouton :

`public IRelayCommand BtnSetConversion { get; }`

```

public ConvertisseurEuroViewModel()
{
    ...
    //Boutons
    BtnSetConversion = new RelayCommand(ActionSetConversion);
}

private void ActionSetConversion()
{
    //Code du calcul à écrire
}

```

Le bouton est maintenant créé. Il appelle une méthode nommée `ActionSetConversion` à coder.

Dans la méthode `ActionSetConversion`, ajouter le code permettant de réaliser le calcul (même code que dans la V1, si vous avez créé les mêmes properties). Ne pas gérer les erreurs pour le moment.

## Erreurs

Gérer les erreurs comme en V1.

La ligne de code suivante ne fonctionnera plus car le code d'affichage de la boîte de dialogue n'est plus dans la page XAML : maContentDialog.XamlRoot = **this.Content.XamlRoot**;

Il faut donc procéder différemment :

- Dans **App.xaml.cs**, définir une property de type **FrameworkElement**  
**public static FrameworkElement MainRoot { get; private set; }**
- Le code devient :  
**maContentDialog.XamlRoot = this.Content.XamlRoot; -> maContentDialog.XamlRoot = App.MainRoot.XamlRoot;**
- Il reste à initialiser la property **MainRoot** dans la méthode **OnLaunched** de **App.xaml.cs** :  
**MainRoot = m\_window.Content as FrameworkElement;**

Exemple ici : <https://xamlbrewer.wordpress.com/2022/03/09/a-dialog-service-for-winui-3/>

## Injection de dépendance

1. Installer le package Nuget **Microsoft.Extensions.DependencyInjection**.
2. Ajouter le code suivant dans le constructeur du programme principal **App.xaml.cs** :

```
/// <summary>
/// Gets the instance to resolve application services.
/// </summary>
public ServiceProvider Services { get; }

/// <summary>
/// Initializes the singleton application object. This is the first line of authored code
/// executed, and as such is the logical equivalent of main() or WinMain().
/// </summary>
public App()
{
    this.InitializeComponent();

    /// <summary>
    /// Configures the services for the application.
    /// </summary>
    ServiceCollection services = new ServiceCollection();

    //ViewModels
    services.AddTransient<ConvertisseurEuroViewModel>();

    Services = services.BuildServiceProvider();
}

/// <summary>
/// Gets the current app instance in use
/// </summary>
public new static App Current => (App)Application.Current;
```

Ce code permet d'enregistrer le "service" **ConvertisseurEuroViewModel** (notre classe View Model) dans le container de services au démarrage de l'application. Un container d'IoC est une sorte de cache sensé stocker tous les services (comme, par exemple, des classes) utiles et utilisables par la suite.

*Remarque : ici nous avons utilisé la méthode **AddTransient**. Autres méthodes possibles : **AddScoped** et **AddSingleton**. Différences :*

**Singleton** which creates a single instance throughout the application. It creates the instance for the first time and reuses the same object in all calls.

**Scoped** lifetime services are created once per request within the scope. It is equivalent to a singleton in the current scope. For example, in MVC it creates one instance for each HTTP request, but it uses the same instance in the other calls within the same web request.

**Transient** lifetime services are created each time they are requested. This lifetime works best for lightweight, stateless services.

3. Modifier le DataContext dans le code behind de la vue :

```
24  namespace ClientConvertisseurV2.Views
25  {
26      /// <summary>
27      /// An empty page that can be used on its own or navigated to within a Frame.
28      /// </summary>
29      public sealed partial class ConvertisseurEuroPage : Page
30      {
31          public ConvertisseurEuroPage()
32          {
33              this.InitializeComponent();
34              DataContext = App.Current.Services.GetService<ConvertisseurEuroViewModel>();
35          }
36      }
37  }
```

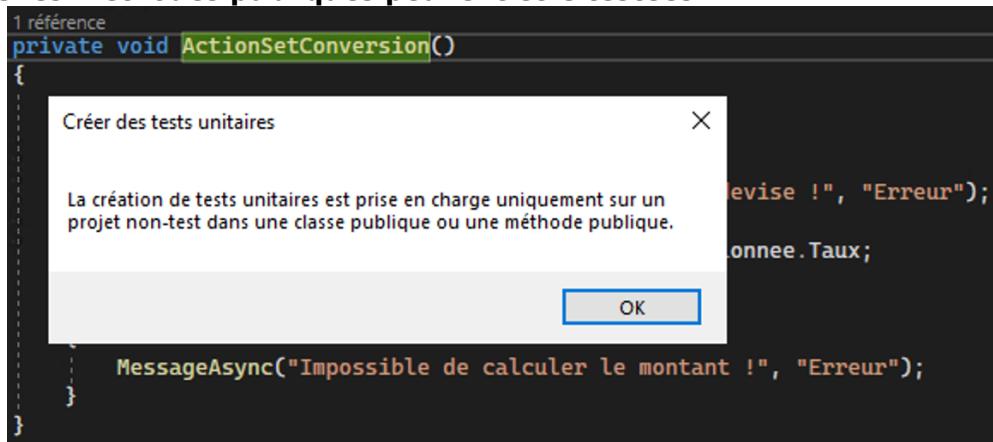
Lancer l'application et la tester.

**Améliorer le code.**

### **3. Tests unitaires de l'application cliente**

#### **3.1. Tests unitaires du ViewModel**

**Rappel : seules les méthodes publiques peuvent être testées.**



Définir en `public` les méthodes `GetDataOnLoadAsync` et `ActionSetConversion`.

Cliquer sur une des méthodes **publiques** de `ConvertisseurEuroViewModel` avec le bouton droit de la souris, puis choisir « Crée des tests unitaires » pour générer le projet de test MSTestv2.

Ajouter la balise `<WindowsSdkPackageVersion>10.0.19041.38</WindowsSdkPackageVersion>` au fichier XML projet.

Méthodes de test à ajouter (**A MODIFIER EN FONCTION DE VOTRE CODE**) :

- Test du constructeur :

```
/// <summary>
/// Test constructeur.
/// </summary>
[TestMethod()]
public void ConvertisseurEuroViewModelTest()
{
    ConvertisseurEuroViewModel convertisseurEuro = new ConvertisseurEuroViewModel();
    Assert.IsNotNull(convertisseurEuro);
}
```

Et oui, un constructeur étant public, se teste aussi !!!!

Tout comme les properties d'ailleurs, mais c'est rare qu'elles le soient. Si elles contiennent du code spécifique (vérification, etc.), il faudra les tester.

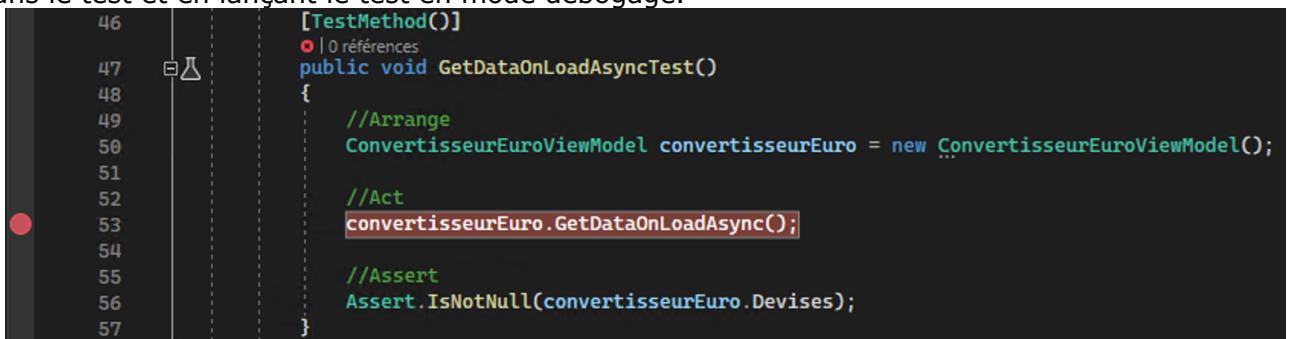
- Test de la méthode GetDataOnLoadAsyncTest() :

```
///<summary>
/// Test GetDataOnLoadAsyncTest OK
/// </summary>
[TestMethod()]
public void GetDataOnLoadAsyncTest_OK()
{
    //Arrange
    ConvertisseurEuroViewModel convertisseurEuro = new ConvertisseurEuroViewModel();

    //Act
    convertisseurEuro.GetDataOnLoadAsync();

    //Assert
    Assert.IsNotNull(convertisseurEuro.Devises);
}
```

Lancer l'exécution des tests. Vous verrez que le test ci-dessus est en erreur. En effet, comme l'appel au WS est asynchrone, le test s'exécute sans attendre le résultat de l'appel à la méthode asynchrone GetDevisesAsync (dans la méthode GetDataOnLoadAsync). Vous pouvez le vérifier en mettant un point d'arrêt dans le test et en lançant le test en mode débogage.

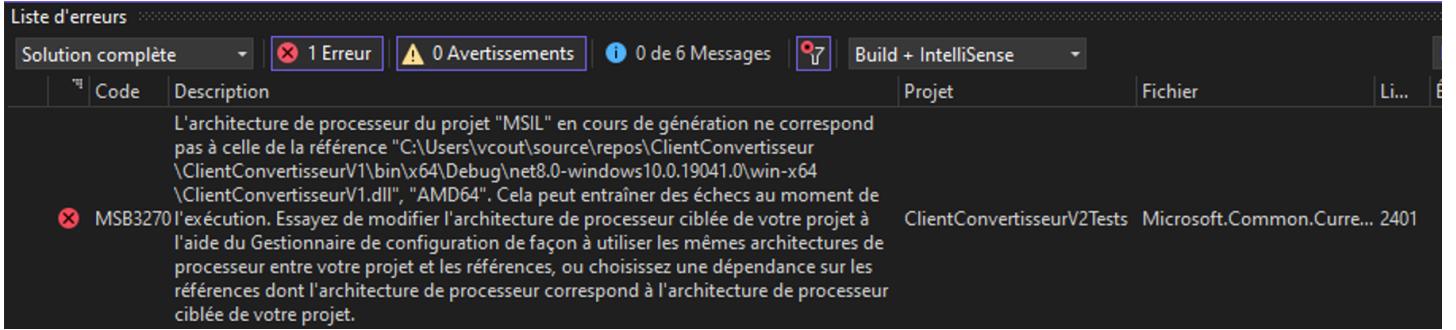


Une des solutions est de forcer le test à attendre. Pour cela, il est nécessaire d'ajouter la ligne suivante suite à l'appel à convertisseurEuro.GetDataOnLoadAsync(); : Thread.Sleep(1000);

Cette fois le test devrait passer, à condition que l'API soit bien sûr lancée (ce qui fait que ce test n'est pas optimal).

Améliorer le test unitaire : ajouter une assertion vérifiant que Devises contient bien les 3 devises prévues (utiliser la classe CollectionAssert). Vous pourrez ajouter un constructeur paramétré dans la classe Devise et devrez coder la méthode Equals.

***Si au lancement des tests, vous avez l'erreur suivante :***



**Vérifier dans l'onglet build du projet de l'application cliente et dans celui du test (bouton droit de la souris sur le projet puis Propriétés) que les informations sont identiques :**

The screenshots show the 'Properties' window for two projects: 'ClientConvertisseurV2' and 'ClientConvertisseurV2Tests'. Both windows have the 'Build' tab selected. In the 'Plateforme cible' dropdown, the value 'x86' is selected and highlighted with a red box.

- Test de la méthode ActionSetConversion (à écrire) :

```

/// <summary>
/// Test conversion OK
/// </summary>
[TestMethod()]
public void ActionSetConversionTest()
{
    //Arrange
    //Création d'un objet de type ConvertisseurEuroViewModel

    //Property MontantEuro = 100 (par exemple)
}

```

```

//Création d'un objet Devise, dont Taux=1.07
//Property DeviseSelectionnee = objet Devise instancié

//Act
//Appel de la méthode ActionSetConversion

//Assert
//Assertion : MontantDevise est égal à la valeur espérée 107
}

```

Vous ne pourrez pas tester la méthode permettant d'afficher un message car son résultat consiste en l'affichage d'une popup.

Si vous lancez l'analyse de la couverture du code (menu *Test > Analyser la couverture de code pour tous les tests*), vous verrez que le code du ViewModel est en majorité testé, même s'il ne l'est pas à 100% :

Résultats de la couverture du code		Couverts (blocs)	Non couverts (blocs)	Couvertes (Lignes)	Partiellement couvertes (Li)	Non
vcout_D101-S01_2024-09-09.23_00_08.cover						
Hierarchie						
{ } ClientConvertisseurV2.ViewModels		38	18	31	0	17
ConvertisseurEuroViewModel		29	5	25	0	6
get_MontantEuro()		2	0	1	0	0
set_MontantEuro(double)		1	0	1	0	0
get_MontantDevise()		2	0	1	0	0
set_MontantDevise(double)		2	0	4	0	0
get_DeviseSelectionnee()		2	0	1	0	0
set_DeviseSelectionnee(ClientConvertisseurV2.Models.Devise)		1	0	1	0	0
get_Devises()		2	0	1	0	0
set_Devises(System.Collections.ObjectModel.ObservableCollection<ClientConvertisseurV2.Models.Devise>)		2	0	4	0	0
get_BtnSetConversion()		0	1	0	0	1
ConvertisseurEuroViewModel()		5	0	5	0	0
ActionSetConversion()		10	4	6	0	5
ConvertisseurEuroViewModel.<GetDataOnLoadAsync>d__20		9	2	6	0	1
ConvertisseurEuroViewModel.<MessageAsync>d__22		0	11	0	0	10
{ } ClientConvertisseurV2.Services		17	1	11	0	3

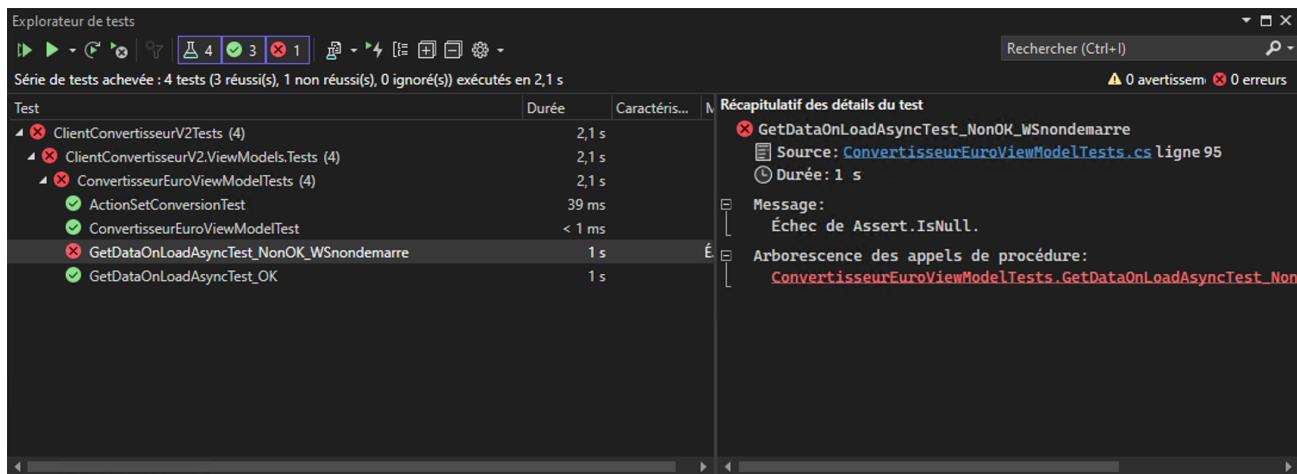
Le constructeur est testé à 100%. La gestion des erreurs de ActionSetConversion et GetDataOnLoadAsync n'est pas testée, d'où les blocs (ou lignes) non couverts.

Rajouter un test qui réussit si le WS n'est pas démarré.

API non lancée :

Explorateur de tests			Récapitulatif des détails du test		
4  3  1			0 avertissement  0 erreurs		
Série de tests achevée : 4 tests (3 réussi(s), 1 non réussi(s), 0 ignoré(s)) exécutés en 2,1 s					
Test	Durée	Caractéris...			
✗ ClientConvertisseurV2Tests (4)	2,1 s		GetDataOnLoadAsyncTest_NonOK_WSnondemarre		
✗ ClientConvertisseurV2.ViewModels.Tests (4)	2,1 s		Source: <a href="#">ConvertisseurEuroViewModelTests.cs</a> ligne 95		
✗ ConvertisseurEuroViewModelTests (4)	2,1 s		⌚ Durée: 1 s		
✓ ActionSetConversionTest	38 ms				
✓ ConvertisseurEuroViewModelTest	< 1 ms				
✓ GetDataOnLoadAsyncTest_NonOK_WSnondemarre	1 s				
✗ GetDataOnLoadAsyncTest_OK	1 s				

API lancée :



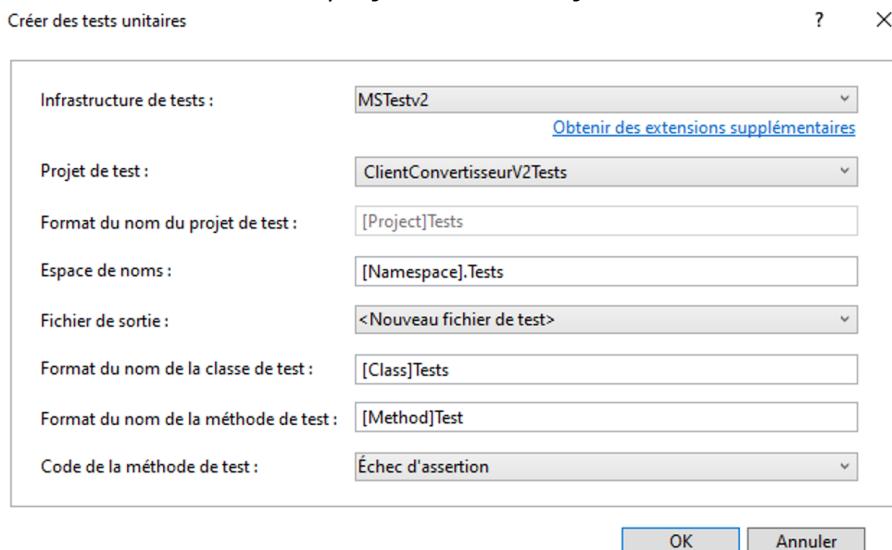
**Normalement les tests unitaires doivent toujours réussir. Notre fonctionnement actuel n'est donc pas optimal.**

### 3.2. Tests unitaires de la classe WSService

- Tester le constructeur.
- Tester la méthode GetDevisesAsync.

*Indications :*

- Ajouter des tests unitaires au projet de tests déjà créé :

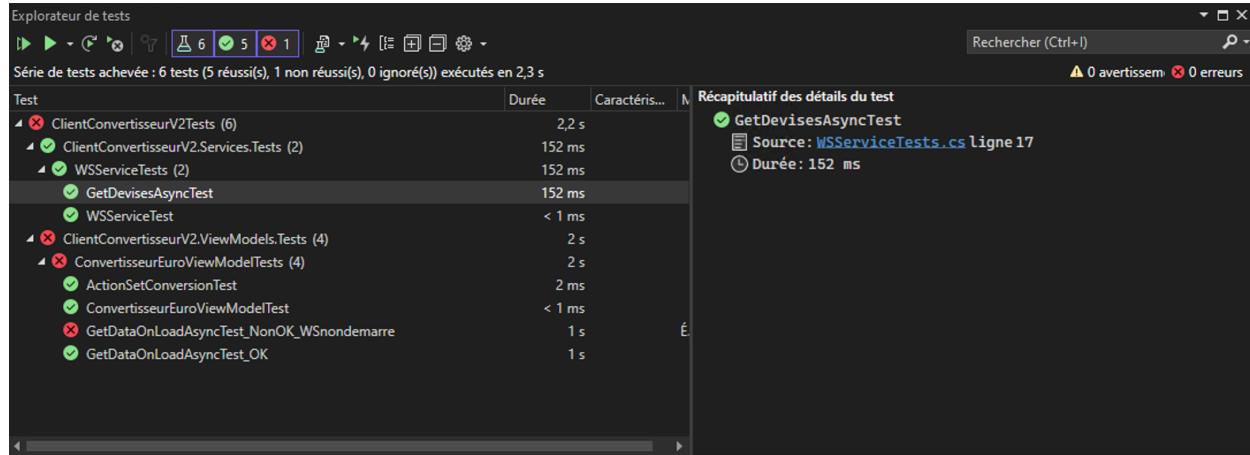


- Procéder de la même façon que pour `GetDataOnLoadAsync`
- Comme la méthode asynchrone `GetDevisesAsync` retourne un résultat, il sera possible de la bloquer dans l'attente de ce résultat (Cf. CM1) :
 

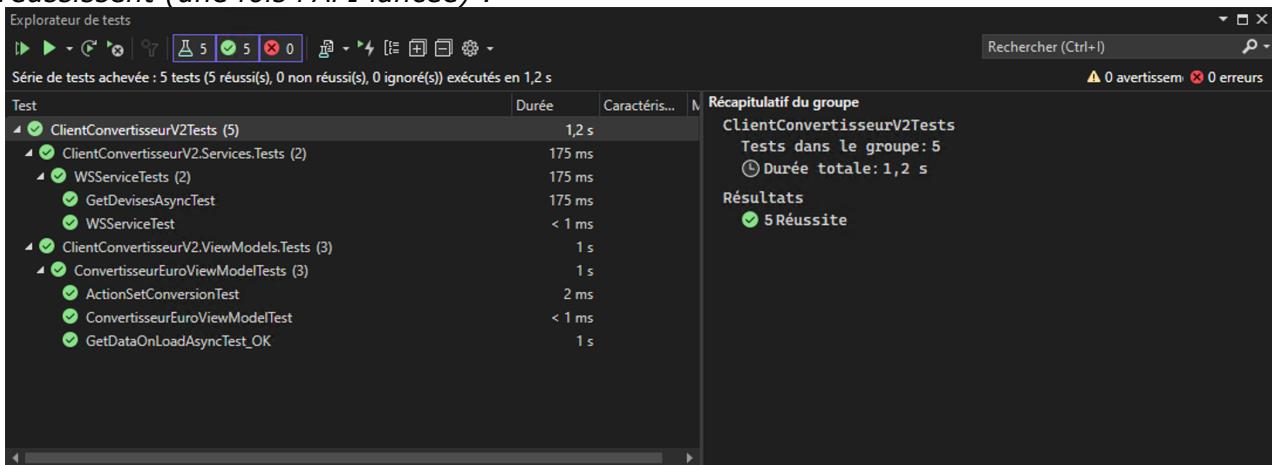
```
var result = service.GetDevisesAsync("devises").Result;
```

 Ainsi l'appel devient synchrone.
- Coder 3 assertions : le résultat n'est pas nul, le résultat est du type espéré, le résultat contient bien les 3 devises espérées.

Une fois l'API lancée :



Remarque : vous pouvez commenter le test `GetDataOnLoadAsyncTest_NonOK_WSnondemarre` afin que tous les tests réussissent (une fois l'API lancée) :



#### **4. Ajout d'une nouvelle page**

Dans le dossier `Views`, ajouter une nouvelle page (Page vierge XAML) permettant de convertir un montant en devise en un montant en euros.

Vous pourrez réutiliser une partie importante du code déjà créé. **Penser réutilisation et refactoring !!!!!!**

Tester la page après avoir modifié la ligne suivante du fichier `App.xaml.cs` :

```
rootFrame.Navigate(typeof(ConvertisseurEuroPage));
```

#### **RAPPEL - Etapes pour appliquer le patron d'architecture MVVM :**

1. Installer le package NuGet `CommunityToolkit.Mvvm`
2. Créer une vue (fenêtre ou page)
3. Créer une classe `View Model` héritant de `ObservableObject`. Y créer les properties et les `IRelayCommand` nécessaires (1 par bouton). Les properties pourront appeler la méthode `OnPropertyChanged` en cas de mise à jour de la vue.
4. Ajouter le binding sur les contrôles, y compris les boutons, dans la vue WinUI.
5. Dans le constructeur du fichier `App.xaml.cs` enregistrer la classe du View Model créé en étape 3 dans le conteneur de services :

```
public App()
{
    /// <summary>
    /// Configures the services for the application.
    /// </summary>
    ServiceCollection services = new ServiceCollection();

    //ViewModels
    services.AddTransient<ConvertisseurEuroViewModel>();
    services.AddTransient<MonNouveauViewModel>();
}
```

```
        Services = services.BuildServiceProvider();
    }

6. Dans le constructeur (code behind) de la vue, ajouter le DataContext :
DataContext = App.Current.Services.GetService<MonNouveauViewModel>();
```

## Améliorer le code.

## Coder les tests unitaires.

### Bilan

Lorsque l'on développe des petites applications, respecter parfaitement le patron de conception MVVM est peut-être un peu démesuré. Dans notre cas, nous l'avons pleinement appliqué car aucun code fonctionnel ne figure dans le code-behind de la vue (`ConvertisseurEuroPage.xaml.cs`). Le but premier de MVVM est de séparer les responsabilités, notamment en séparant les données de la vue. Cela facilite les opérations de maintenance en limitant l'impact d'éventuelles corrections sur un autre morceau de code.

L'intérêt également est qu'il devient possible de faire des tests unitaires sur le ViewModel, sans avoir besoin de réaliser des tests end-to-end. Cela permet de tester chaque fonctionnalité, dans un processus automatisé. Ce qui est un atout considérable pour éviter les régressions de code...

## 5. Pour les plus rapides

Ajouter un menu hamburger :

<https://learn.microsoft.com/fr-fr/windows/apps/design/controls/navigationview>

<https://xamlbrewer.wordpress.com/2021/07/06/navigating-in-a-winui-3-desktop-application/>

La navigation entre les pages sera réalisée dans le code-behind (pas dans le ViewModel) pour simplifier.