

L'objectif des applications UWP est de fonctionner avec des Web services REST, que ces web services soient fournis par des tiers (Bing Maps, Google Maps, etc.) ou développés pour les propres besoins de votre application. Microsoft a développé une API de web service REST nommée WEB API.NET, basée sur ASP.NET MVC. Cette API est particulièrement recommandée dans le cadre d'interactions avec des mobiles. Elle en outre simple à mettre en œuvre, légère et très robuste.

Nous allons réaliser un premier WS REST simple sans état (i.e. sans base de données). Ce WS va nous permettre de convertir des euros en différentes devises.

Un WS REST est orienté ressources (et non traitement) par défaut (i.e. obtention de données au format JSON, XML, ATOM, etc.). Il s'agit donc ici de gérer des listes de devises avec leur taux de conversion. Une application cliente simple (Universal Windows) permettra de convertir un montant saisi en euros vers la devise sélectionnée (la liste de devises sera alimentée par le WS). La même application sera par la suite redéveloppée en appliquant le pattern MVVM.

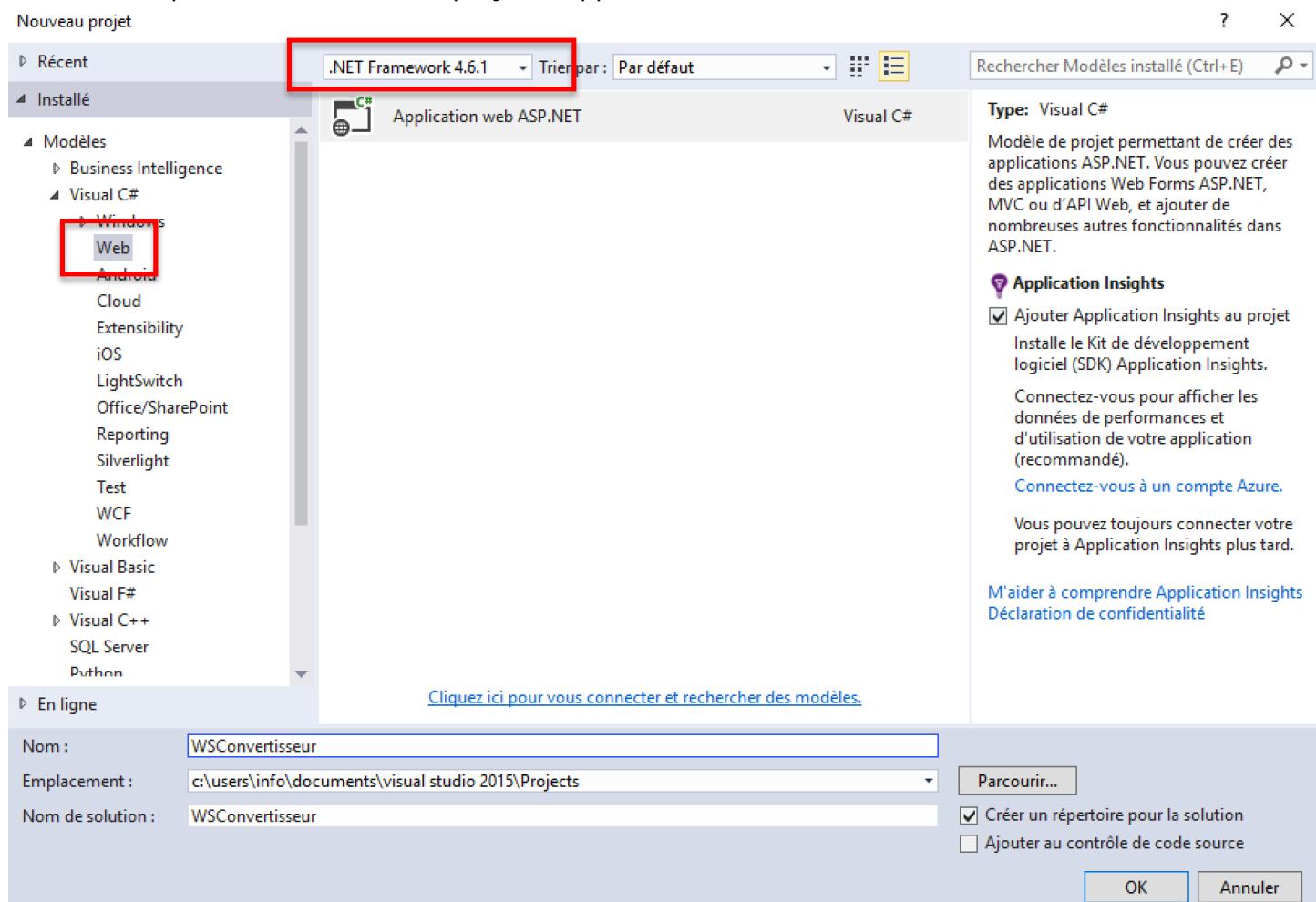
CREER L'ENSEMBLE DE VOS PROJETS ET SOLUTIONS SUR D:\

1. Création du WS REST

1.1. Création du projet

Lancez Visual Studio 2015.

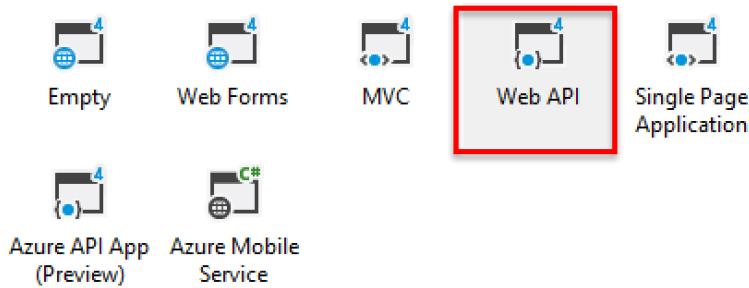
Commencez par créer un nouveau projet « Application Web ASP.NET » :



De type Web API :

Ajouter également un projet de tests unitaires. Décocher l'hébergement dans le cloud Azure.

Sélectionner un modèle :

Modèles ASP.NET 4.6.1**Modèles ASP.NET 5**

Ajoutez des dossiers et des références de base pour :

 Web Forms MVC Web API Ajouter des tests unitaires

Nom du projet de test : WSConvertisseur.Tests

Modèle de projet pour la création de services RESTful HTTP pouvant atteindre un grand nombre de clients, notamment des navigateurs et des téléphones mobiles.

[En savoir plus](#)**Modifier l'authentification**Authentification : **Comptes d'utilisateur individuels**

Microsoft Azure

 Host in the cloud

Web App

OK

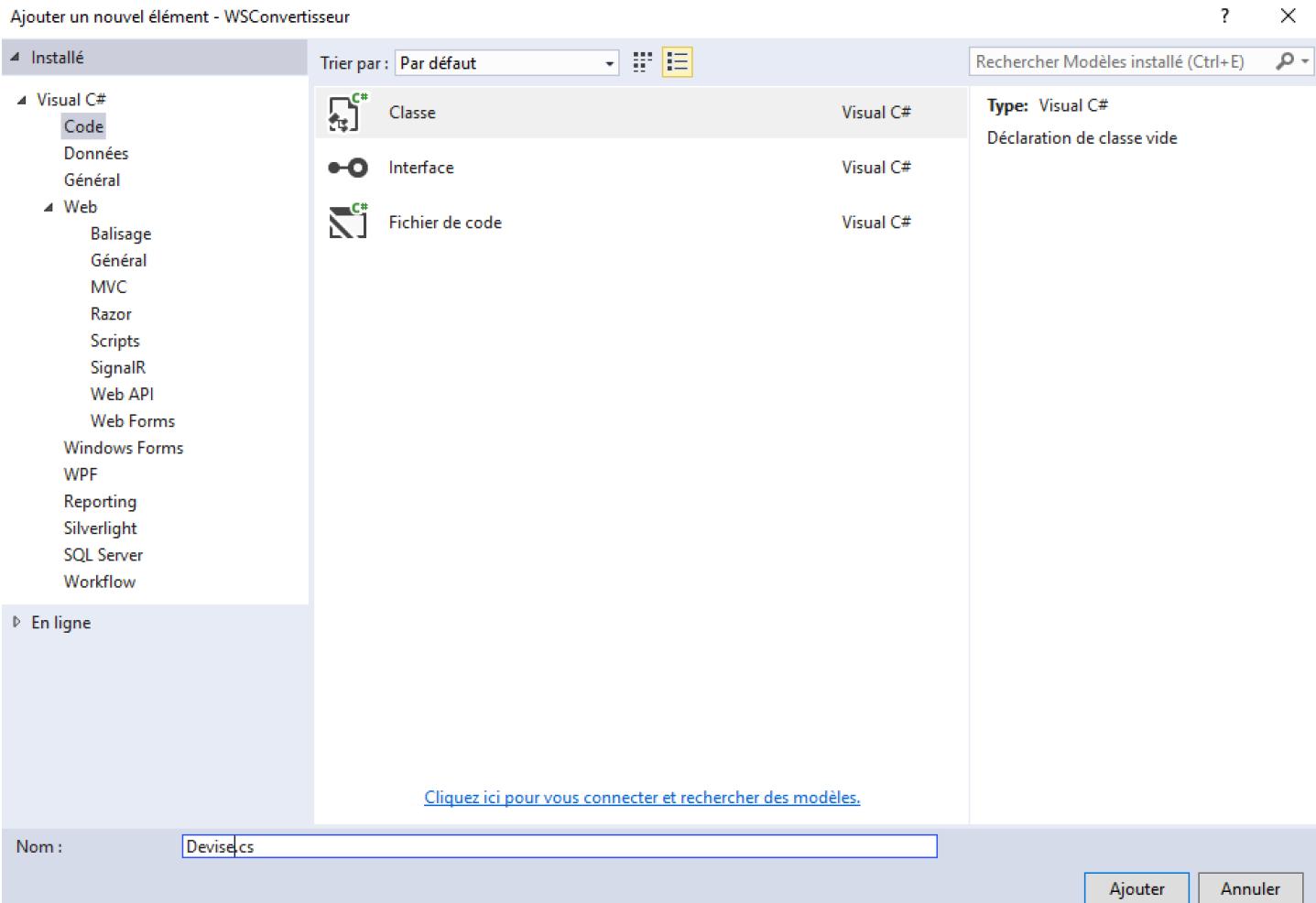
Annuler

Valider.

L'architecture est basée sur un modèle MVC (Modèle-Vue-Contrôleur). Vous y retrouverez donc des contrôleurs et des modèles. Il n'y aura cependant pas de vues (view) car la vue correspondra à l'application cliente qui accèdera au WS.

1.2. Création de la couche M

Nous allons ajouter une classe *Model*. Pour cela, dans l'explorateur de la solution puis dossier *Models*, ajoutez une classe (*Ajouter > Nouvel élément*).



Créer 3 properties dans la classe :

- Une pour l'Id (int)
- Une pour le Nom de la devise (String)
- Une pour le Taux (double)

Ajouter un constructeur paramétré et un constructeur sans paramètre.

1.3. Création de la couche C

Dans le dossier *Controllers* de la solution, ajouter un nouveau contrôleur (*Ajouter > Contrôleur*). Choisir « Contrôleur Web API 2 avec actions de lecture/écriture » car il s'agit d'un WS WebAPI.Net.

Ajouter un modèle automatique

X

Installé

Commun
Contrôleur

-  Contrôleur MVC 5 - Vide
-  Contrôleur MVC 5 avec des actions de lecture/écriture
-  Contrôleur MVC 5 avec vues, utilisant Entity Framework
-  Contrôleur Web API 2 – Vide
-  Contrôleur Web API 2 avec actions de lecture/écriture
-  Contrôleur Web API 2 avec actions, utilisant Entity Framework
-  Web API 2 OData v3 Controller with actions, using Entity Framework
-  Web API 2 OData v3 Controller with read/write actions

Contrôleur Web API 2 avec actions de lecture/écriture
par Microsoft
v2.0.0.0

Un contrôleur Web API avec des actions REST pour créer, lire, mettre à jour, supprimer et répertorier des entités.

Id : ApiControllerWithActionsScaffolder

[Cliquez ici pour accéder à Internet et rechercher d'autres extensions de génération de modèles automatique.](#)

Ajouter

Annuler

Vous pouvez renommer le nom du contrôleur en **DeviseController**.

Ajouter un contrôleur

X

Nom du contrôleur :

DeviseController

Ajouter

Annuler

En sélectionnant le modèle « Contrôleur Web API 2 avec actions de lecture/écriture », Visual Studio génère le squelette de code correspondant aux méthodes Get, Put, Post et Delete.

```

namespace WSConvertisseur.Controllers
{
    public class DeviseController : ApiController
    {
        // GET: api/Devise
        public IEnumerable<string> Get()
        {
            return new string[] { "value1", "value2" };
        }

        // GET: api/Devise/5
        public string Get(int id)
        {
            return "value";
        }

        // POST: api/Devise
        public void Post([FromBody]string value)
        {
        }

        // PUT: api/Devise/5
        public void Put(int id, [FromBody]string value)
        {
        }

        // DELETE: api/Devise/5
        public void Delete(int id)
        {
        }
    }
}

```

On contrôleur hérite toujours de la classe ApiController : [https://msdn.microsoft.com/en-us/library/system.web.http.apicontroller\(v=vs.118\).aspx](https://msdn.microsoft.com/en-us/library/system.web.http.apicontroller(v=vs.118).aspx)

Dans le contrôleur (constructeur), créer une liste de devises (type Devise) contenant les données suivantes :

1	Dollar	1.08
2	Franc Suisse	1.07
3	Yen	120

Méthode Get()

Modifier la première méthode Get (public IEnumerable<string> Get()) permettant de retourner la liste des devises (type de retour : IEnumerable<Devise>).

Remarques :

- Il est possible, mais pas nécessaire, d'appeler la méthode AsEnumerable() sur la liste.
- Le type de retour IQueryable<Devise> est également possible (meilleures performances si nombreuses données). Dans ce cas, il faudra appeler la méthode AsQueryable() sur la liste.
- Différence entre IEnumerable et IQueryable :
 - o <http://stackoverflow.com/questions/4455428/difference-between-iqueryable-icollection-ilist-idictionary-interface>
 - o <https://www.codeproject.com/Articles/832189>List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v>

La méthode Get récupérant toutes les devises est maintenant fonctionnelle et se consomme de cette façon : GET /Devise

Exécuter l'application. Un navigateur s'ouvre alors. Vous pourrez voir l'API du WS en cliquant sur le lien API.

The screenshot shows a browser window titled "ASP.NET Web API Help". The address bar displays "localhost:2824/Help". The main content area is titled "Nom de l'application" and includes navigation links for "Accueil" and "API". Below this, there is a table listing various API endpoints:

API	Description
GET api/Account/ExternalLogins?returnUrl={returnUrl}&generateState={generateState}	No documentation available.
POST api/Account/Register	No documentation available.
POST api/Account/RegisterExternal	No documentation available.

Below this section, there is a heading "Values" followed by another table:

API	Description
GET api/Values	No documentation available.
GET api/Values/{id}	No documentation available.
POST api/Values	No documentation available.
PUT api/Values/{id}	No documentation available.
DELETE api/Values/{id}	No documentation available.

Finally, there is a heading "Devise" followed by a third table:

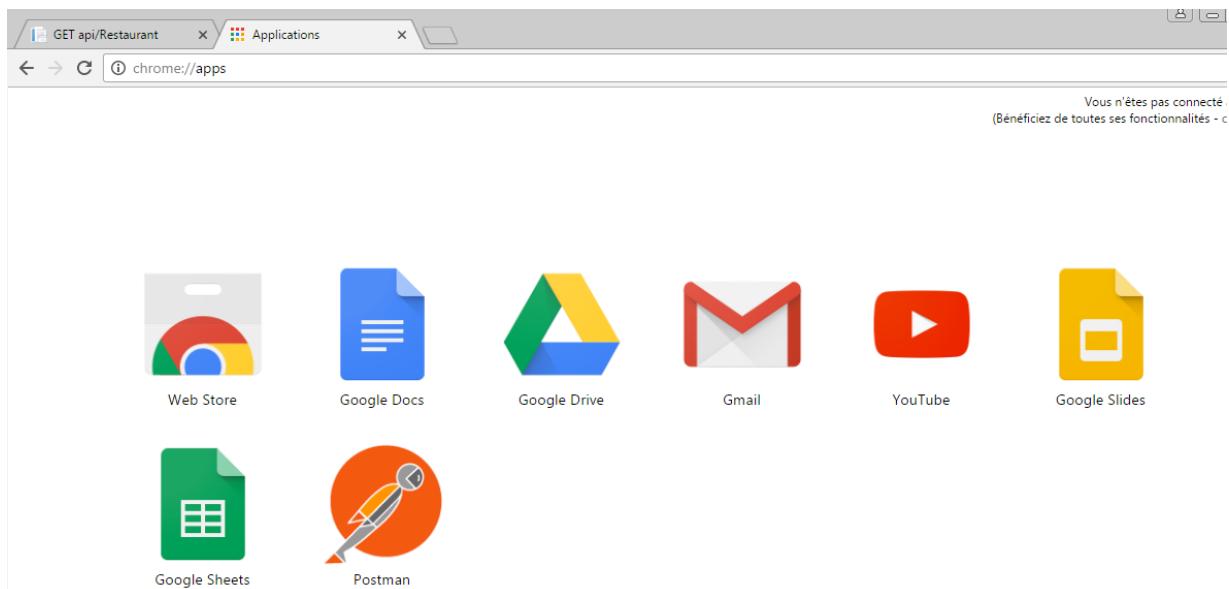
API	Description
GET api/Devise	No documentation available.
GET api/Devise/{id}	No documentation available.
POST api/Devise	No documentation available.
PUT api/Devise/{id}	No documentation available.
DELETE api/Devise/{id}	No documentation available.

© 2017 - My ASP.NET Application

Noter le port utilisé (ici : 2824).

Ne pas fermer le navigateur.

Nous allons tester l'appel au WS (hébergé localement) dans l'outil « Postman ». Ouvrir Chrome. Saisir chrome://apps dans la barre d'adresse de Chrome puis cliquer sur Postman.



Remarques :

- Si Postman n'est pas installé, l'installer (pour cela rechercher Postman dans Google avec Chrome).
- Vous pouvez tester vos services web REST avec d'autres applications telles que Fiddler, Curl ou un autre plugin Chrome comme Advanced Rest Client.

Saisir l'URL **locale** du WS, choisir la méthode « GET » et ajouter un header (Accept = application/json). Cliquer sur « Send ».

On obtient la liste de toutes les devises au format JSON.

The screenshot shows the Postman interface with the following details:

- Request URL:** http://localhost:2824/api/Devise
- Method:** GET
- Headers:** Accept: application/json
- Body:** (Pretty) JSON response:

```

1 [ ]
2   [
3     {
4       "Id": 1,
5       "NomDevise": "Dollar",
6       "Taux": 1.08
7     },
8     {
9       "Id": 2,
10      "NomDevise": "Franc Suisse",
11      "Taux": 1.07
12    },
13    {
14      "Id": 3,
15      "NomDevise": "Yen",
16      "Taux": 120
17    }
18  ]

```

Noter le statut (**200 OK**).

Vous remarquerez les [] dans le JSON, indiquant que l'on obtient une collection (tableau).

```
[ ]
{
  "Id": 1,
  "NomDevise": "Dollar",
  "Taux": 1.08
},
{
  "Id": 2,
  "NomDevise": "Franc Suisse",
  "Taux": 1.07
}
```

```

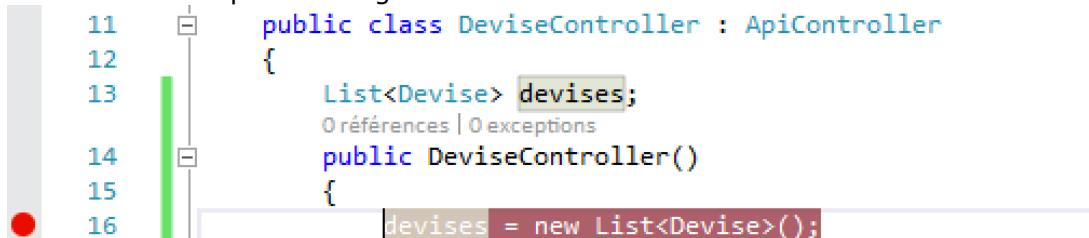
        },
        {
            "Id": 3,
            "NomDevise": "Yen",
            "Taux": 120
        }
    ]
]
```

Vous pouvez tester la récupération en XML en modifiant le header `Accept`.

```

1 <ArrayOfDevise xmlns:i="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://schemas.datacontract.org/2004/07/WSConvertisseur.Models">
2     <Devise>
3         <Id>1</Id>
4         <NomDevise>Dollar</NomDevise>
5         <Taux>1.08</Taux>
6     </Devise>
7     <Devise>
8         <Id>2</Id>
9         <NomDevise>Franc Suisse</NomDevise>
10        <Taux>1.07</Taux>
11    </Devise>
12    <Devise>
13        <Id>3</Id>
14        <NomDevise>Yen</NomDevise>
15        <Taux>120</Taux>
16    </Devise>
17 </ArrayOfDevise>
```

Mettre un point d'arrêt sur la première ligne de code du constructeur.

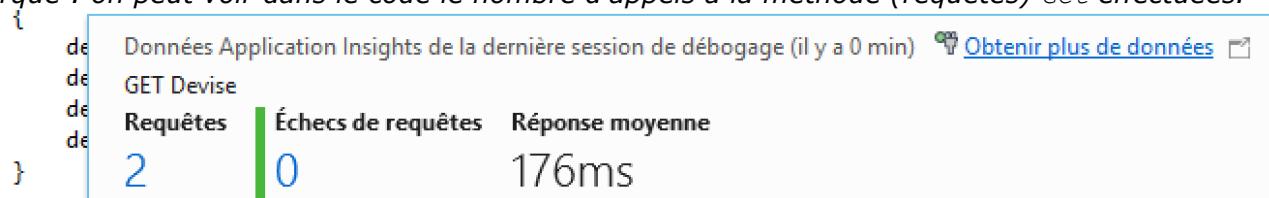


```

11     public class DeviseController : ApiController
12     {
13         List<Devise> devises;
14         public DeviseController()
15         {
16             devises = new List<Devise>();
```

Ré-exécuter l'appel GET dans Postman (vous pouvez le faire plusieurs fois). On remarque que le constructeur est exécuté à chaque appel. Autrement dit, un nouvel objet est instancié à chaque appel. Ainsi, la liste des devises est recréée à chaque fois.

Remarque : on peut voir dans le code le nombre d'appels à la méthode `Get` effectuées.



```

} } Données Application Insights de la dernière session de débogage (il y a 0 min) Obtenir plus de données
GET Devise
Requêtes Échecs de requêtes Réponse moyenne
2 0 176ms

// GET: api/Devise
public IEnumerable<Devise> Get()
{
    return devises;
}
```

Méthode `Get(id)`

Il s'agit ici de parcourir la liste de devises et de renvoyer la bonne devise (type de retour : `Devise`). Pour parcourir la liste, plusieurs possibilités :

- Faire une boucle `foreach` (pas terrible !)
- Créer une instruction LINQ <https://msdn.microsoft.com/en-us/library/bb397678.aspx> :
 - o En pseudo SQL, on récupère une liste d'éléments correspondant à la clause where


```
IEnumerable<Devise> dev =
from d in devises
where d.Id == id
select d;
return dev.ToList()[0];
```

OU MIEUX (car le code précédent plante si la requête ne renvoie rien) :

```
Devise devise =  
    (from d in devises  
     where d.Id == id  
     select d).FirstOrDefault();  
return devise;
```

- o En lambda expression :

```
Devise devise = devises.FirstOrDefault(d => d.Id == id);
```

- o First/FirstOrDefault : <http://stackoverflow.com/questions/1024559/when-to-use-first-and-when-to-use-firstOrDefault-with-linq>

La méthode Get(id) récupérant une seule devise est maintenant fonctionnelle et se consomme de cette façon : GET /Devise/id

Résultat :

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:2824/api/Devise/2`. The response body is a JSON object:

```
1 [ {  
2   "Id": 2,  
3   "NomDevise": "Franc Suisse",  
4   "Taux": 1.07  
5 } ]
```

On remarque cette fois-ci qu'il ne s'agit pas d'une collection (pas de []).

Tester avec un ID inexistant. On récupère un *null*, ce qui n'est pas un retour standard. Ce devrait être une erreur http *404 Not Found*.

The screenshot shows the Postman interface with a failed API call. The URL is `http://localhost:2824/api/Devise/10`. The response body is `null`:

```
1 null
```

Nous allons améliorer le code afin de gérer l'erreur *404 Not Found*.

```
using System.Web.Http.Description; // Pour [ResponseType...]
```

```
[ResponseType(typeof(Devise))]  
public IHttpActionResult Get(int id)  
{  
    Devise devise = devises.FirstOrDefault(d => d.Id == id);  
    if (devise == null)  
    {
```

```

        return NotFound();
    }
    return Ok(devise);
}

```

- Si un objet de type `Devise` a été trouvé, c'est la méthode `Ok(T)` de la classe `ApiController` qui est appelée. Le message http finalement retourné contient le statut `HttpStatusCodes.OK` (correspondant au code `200 OK`) et la représentation de l'objet (au format JSON ou XML, par exemple) passée en paramètre de la méthode `Ok`.

Classe `ApiController` : [https://msdn.microsoft.com/en-us/library/system.web.http.apicontroller\(v=vs.118\).aspx](https://msdn.microsoft.com/en-us/library/system.web.http.apicontroller(v=vs.118).aspx)

- Si aucun objet `Devise` n'a été trouvé, c'est la méthode `NotFound()` de la classe `ApiController` qui est appelée. Elle générera une erreur 404.

	<code>NotFound()</code>	Creates a <code>NotFoundResult</code> .
--	-------------------------	---

- L'interface `IHttpActionResult` définit une commande qui crée de manière asynchrone un `HttpResponseMessage`. Le message http de réponse créé contiendra une représentation de la devise retournée ou un message d'erreur (*404 Not Found*).

Le type de retour `IHttpActionResult` est associé au type de la réponse `[ResponseType(typeof(Devise))]` qui permet d'indiquer le type d'entité retourné (il s'agit juste d'une description ; ce n'est donc pas obligatoire).

`[ResponseType(typeof(Devise))]`
ResponseAttribute.ResponseAttribute(Type responseType)
Permet de spécifier le type d'entité retournée par une action quand le type de retour déclaré est `HttpResponseMessage` ou `IHttpActionResult`. Le `ResponseType` est lu par l'`ApiExplorer` lors de la génération de `ApiDescription`.

Tester le WS avec un ID existant et avec un ID inconnu. Dans ce 2nd cas, une erreur 404 est maintenant retournée.

http://localhost:2824/ + No Environment Send Save

GET http://localhost:2824/api/Devise/10 Params Send Save

Authorization Headers (1) Body Pre-request Script Tests Code

Accept application/json Bulk Edit Presets

key value

Body Cookies Headers (9) Tests Status: 404 Not Found Time: 95 ms

Remarque : il est également possible d'appeler la méthode `Get(id)` de la façon suivante : `GET /Devise?id=2`

http://localhost:2824/ + No Environment Send

GET http://localhost:2824/api/Devise?id=2 Params Send

Authorization Headers (1) Body Pre-request Script Tests

Type No Auth

Body Cookies Headers (10) Tests Status: 200 OK

Pretty Raw Preview JSON JSON

```

1  {
2      "Id": 2,
3      "NomDevise": "Franc Suisse",
4      "Taux": 1.07
5  }

```

Méthode Post

`[ResponseType(typeof(Devise))]`

```

public IHttpActionResult Post(Devise devise)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    devises.Add(devise);
    return CreatedAtRoute("DefaultApi", new { id = devise.Id }, devise);
}

```

- Prend un objet Devise en paramètre et retourne un `IHttpActionResult` de description `[ResponseType(typeof(Devise))]`.
- `ModelState.IsValid` indique si une (ou plusieurs) erreur de modèle (classe métier `Devise`) ont été ajoutées à `ModelState`. Des erreurs peuvent être ajoutées dans le cas de problèmes d'encodage ou de conversion (par exemple, du texte, alors qu'un `int` ou `float` est attendu). Exemple, une erreur sera levée si vous passez le fichier JSON suivant :

```

{
    "Id": 4,
    "NomDevise": "Rouble",
    "Taux": AAA
}

if (!ModelState.IsValid)
{
    // ...
}

```

The screenshot shows the Visual Studio debugger's watch window. It displays the `ModelState` variable as a `System.Web.Http.ModelBinding.ModelStateDictionary`. The `IsValid` property is shown as `false`. Expanding the `Keys` collection reveals a single entry for `"devise.Taux"`. A tooltip for this entry states: "L'erreur d'encodage est ensuite retournée au programme appelant : Unexpected character encountered while parsing value: A. Path 'Taux', line 4, position 11.".

L'erreur d'encodage est ensuite retournée au programme appelant :

```

return BadRequest(ModelState);

```

The screenshot shows the `ModelState` variable expanded to show its `Values` collection. For the key `"devise.Taux"`, there are two entries in the `Errors` collection. Each entry has an `ErrorMessage` property containing the string `"Unexpected character encountered while parsing value: A. Path 'Taux', line 4, position 11."`.

Affichage. Une erreur `400 Bad Request` est générée par la méthode `BadRequest()` de la classe `ApiController` avec le `ModelState` en paramètre.

The screenshot shows the Fiddler interface. The status bar at the top right indicates `Status: 400 Bad Request`. The `Body` tab is selected, showing the JSON response. The response body is:

```

1 <pre>{
2     "Message": "La demande n'est pas valide.",
3     "ModelState": {
4         "devise.Taux": [
5             "Unexpected character encountered while parsing value: A. Path 'Taux', line 4, position 11.",
6             "Unexpected character encountered while parsing value: A. Path 'Taux', line 4, position 11."
7         ]
8     }
9 }</pre>

```

A red box highlights the validation errors for the `'Taux'` field.

De même si vous passez le fichier JSON Suivant, une erreur (de conversion cette fois) `400 Bad Request` sera levée.

```

{
    "Id": 4,
    "NomDevise": "Rouble",
    "Taux": "AAA"
}

```

Body Cookies Headers (10) Tests Status: 400 Bad Request

Pretty Raw Preview JSON ↴

```

1 [{}]
2   "Message": "La demande n'est pas valide."
3   "ModelState": {
4     "devise.Taux": [
5       "Error converting value \"AAAA\" to type 'System.Double'. Path 'Taux', line 4, position 17."
6     ]
7   }
8 ]

```

- Le méthode `CreatedAtRoute("DefaultApi", new { id = devise.id }, devise)` de `ApiController` génère l'URL suivante `/api/devise/IdInséré.`

ApiController.CreatedAtRoute<T>, méthode (String, Object, T)

Crée un `CreatedAtRouteNegotiatedContentResult<T>` (201 Crée) avec les valeurs spécifiées.

Espace de noms : `System.Web.Http`
Assembly : `System.Web.Http` (dans `System.Web.Http.dll`)

Syntaxe

C# C++ F# JScript VB

```

protected internal CreatedAtRouteNegotiatedContentResult<T> CreatedAtRoute<T>(
    string routeName,
    Object routeValues,
    T content
)

```

La `DefaultApi` est configurée dans le fichier `WebApiConfig.cs`

RouteConfig.cs WebApiConfig.cs DeviseController.cs Devise.cs

WSConvertisseur WSConvertisseur.WebApiConfig Register(HT

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Net.Http;
5  using System.Web.Http;
6  using Microsoft.Owin.Security.OAuth;
7  using Newtonsoft.Json.Serialization;
8
9  namespace WSConvertisseur
10 {
11     public static class WebApiConfig
12     {
13         public static void Register(HttpConfiguration config)
14         {
15             // Configuration et services de l'API Web
16             // Configurer l'API Web pour utiliser uniquement l'authentification de jeton du porteur.
17             config.SuppressDefaultHostAuthentication();
18             config.Filters.Add(new HostAuthenticationFilter(OAuthDefaults.AuthenticationType));
19
20             // Itinéraires de l'API Web
21             config.MapHttpAttributeRoutes();
22
23             config.Routes.MapHttpRoute(
24                 name: "DefaultApi",
25                 routeTemplate: "api/{controller}/{id}",
26                 defaults: new { id = RouteParameter.Optional }
27             );
28         }
29     }
30 }
31

```

La méthode Post permettant d'insérer une devise est maintenant fonctionnelle et se consomme de cette façon : POST /Devise en lui passant une représentation JSON.

Test d'insertion d'une devise valide :

- Modifier la méthode d'appel (POST) et ajouter un header content-type :

http://localhost:2824/ POST http://localhost:2824/api/Devise

Headers (2)

key	value
Accept	application/json
Content-Type	application/json

- Ajouter en Body une représentation JSON :

http://localhost:2824/ POST http://localhost:2824/api/Devise

Body (raw) JSON (application/json)

```
1 {  
2   "Id": 4,  
3   "NomDevise": "Rouble",  
4   "Taux": 1510.5  
5 }
```

Cliquer sur « Send ».

On remarque le statut « 201 Created » et que le nouvel enregistrement est retourné.

http://localhost:2824/ POST http://localhost:2824/api/Devise

Status: 201 Created Time: 118 ms

Pretty Raw Preview JSON

```
1 {  
2   "Id": 4,  
3   "NomDevise": "Rouble",  
4   "Taux": 1510.5  
5 }
```

Test d'insertion d'une devise non valide :

The screenshot shows the Postman interface. At the top, it says "No Environment". Below that, it shows a "POST" request to "http://localhost:2824/api/Devise". The "Body" tab is selected, showing a JSON payload:

```

1 [{}]
2   "Id": 5,
3   "NomDevise": "Rouble",
4   "Taux": "AAAA"
5 ]

```

Below the body, the status is "Status: 400 Bad Request" and the time is "Time: 192 ms". The "Body" tab is also selected here.

Exécuter ensuite la méthode GET (<http://localhost:2824/api/Devise/>) permettant de récupérer toutes les devises.

On remarque que la devise 4 n'est pas dans le JSON renvoyé et donc absente de la liste des devises. Pourtant, elle a bien été ajoutée à la liste (vous pouvez le vérifier en mettant un point d'arrêt ou un espion dans la méthode `Post`). En effet, comme nous l'avons vu précédemment la liste de devises est recréée à chaque appel au WS. Le WS réalisé est sans état. Une base de données est donc nécessaire afin de sauvegarder les modifications de devises (ajout, MaJ, suppression). Celle-ci est à gérer dans la couche *Model* via un CRUD créé manuellement ou via l'ORM Entity Framework.

Méthode Delete

Coder la méthode `Delete` ressemblant à la méthode `Get(int Id)` :

- Même type de retour
- Utiliser une expression LINQ permettant de récupérer dans la liste la devise dont l'`Id` est passé en paramètre.
- Si la devise est inexistante, renvoyer une erreur *Not Found*.
- Utiliser la méthode `Remove` de la liste pour supprimer la devise
- Retour : `return Ok(devise);`

La méthode `Delete` se consomme de cette façon : `DELETE /Devise/id`.

Tester en mettant un point d'arrêt ou un espion pour vérifier que la devise est bien supprimée de la liste.

Méthode Put

```

[ResponseType(typeof(void))]
public IHttpActionResult Put(int id, Devise devise)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    if (id != devise.Id)
    {
        return BadRequest();
    }
}

```

```

        int index = devises.FindIndex((d) => d.Id == id);
        if (index < 0)
        {
            return NotFound();
        }
        devises[index] = devise;
        return StatusCode(HttpStatusCode.NoContent);
    }
}

```

- Ici, nous avons fait le choix de ne pas retourner l'objet Devise modifié (return Ok<devise>), mais juste un statut de réponse http 204 *No content*, sans contenu JSON.

La méthode Put permettant de mettre à jour une devise se consomme de cette façon : PUT /Devise/id en lui passant une représentation JSON.

Tests de modifications non valides :

The screenshot shows a Postman request configuration for a PUT request to `http://localhost:2824/api/Devise/2`. The Body tab is selected, showing a JSON payload:

```

1 {
2     "Id": 2,
3     "NomDevise": "Franc Suisse",
4     "Taux": "AAA"
5 }

```

The response status is 400 Bad Request, and the error message is displayed in the JSON tab:

```

1 {
2     "Message": "La demande n'est pas valide.",
3     "ModelState": {
4         "devise.Taux": [
5             "Error converting value \"AAA\" to type 'System.Double'. Path 'Taux', line 4, position 16."
6         ]
7     }
8 }

```

The screenshot shows two separate API requests made via Postman:

- Request 1 (Successful):** PUT /api/Devise/1. The JSON body is:


```

1 {
2   "Id": 2,
3   "NomDevise": "Franc Suisse",
4   "Taux": 3
5 }
```

 Response status: 200 OK
- Request 2 (Unsuccessful):** PUT /api/Devise/50. The JSON body is:


```

1 {
2   "Id": 50,
3   "NomDevise": "Franc Suisse",
4   "Taux": 3
5 }
```

 Response status: 404 Not Found

Modification valide :
Tester en mettant un point d'arrêt ou un espion pour vérifier que la devise est bien remplacée dans la liste.

PUT <http://localhost:2824/api/Devise/2>

Body (2)

```

1 [
2   "Id": 2,
3   "NomDevise": "Franc Suisse",
4   "Taux": 3
5 ]

```

Params

Send

Authorization Headers (2) Body Pre-request Script Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

Body Cookies Headers (8) Tests Status: 204 No Content

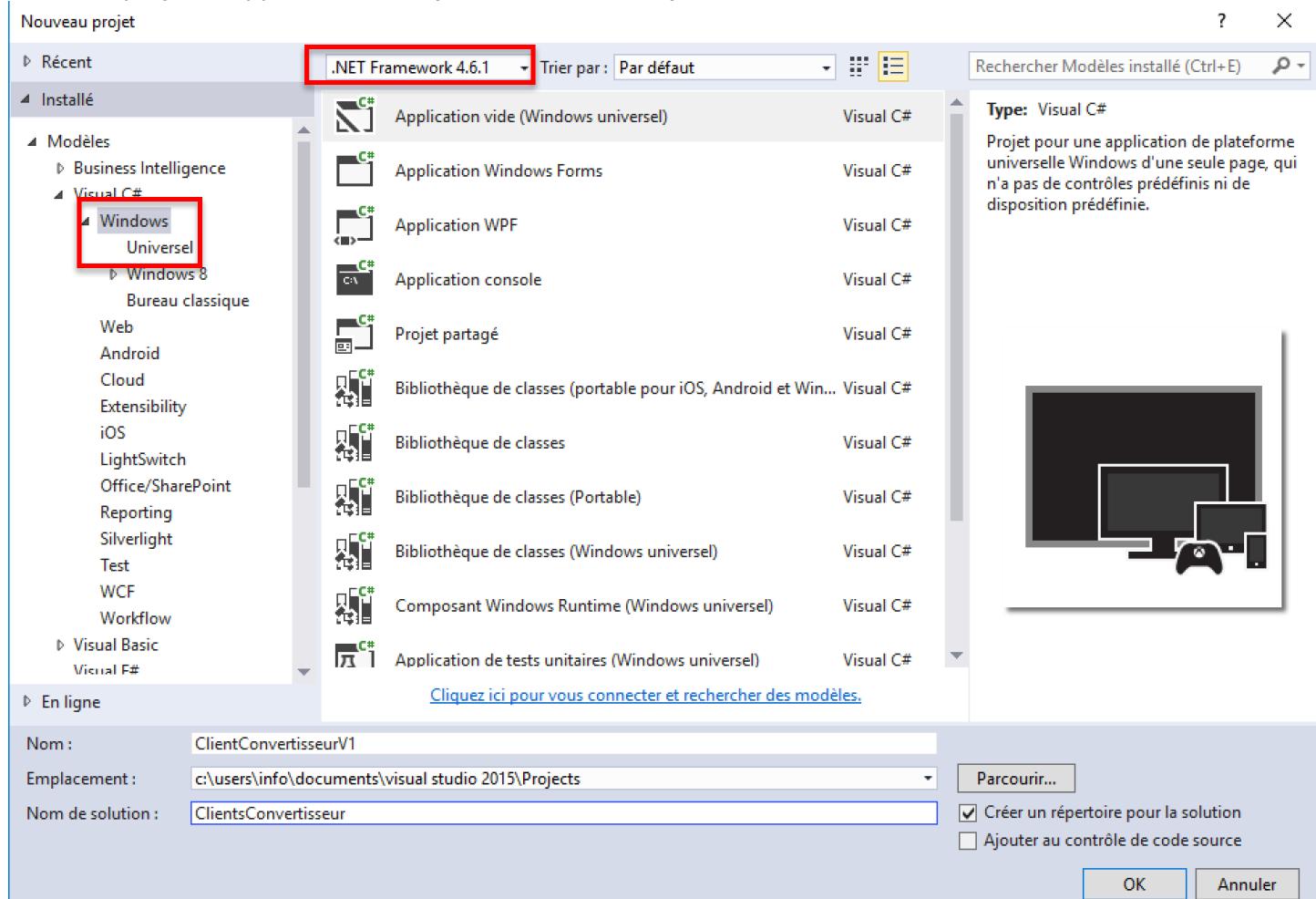
Pretty Raw Preview Text

2. Création de l'application cliente lourde (Windows Universel)

2.1. Client V1 (version simple)

Lancer une nouvelle instance de Visual Studio.

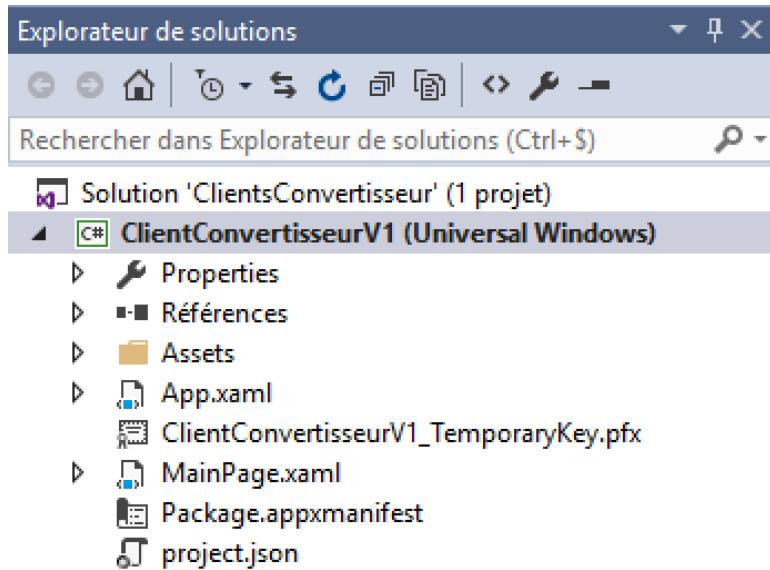
Créer un projet « Application vide (Windows universel) » dans une nouvelle solution.



Valider le message indiquant les versions d'OS Windows 10 cibles.

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 UWP 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. **Regardez l'ensemble des onglets.**

`Package.appxmanifest` `App.xaml.cs`

Les propriétés du package de déploiement pour votre application sont contenues dans le fichier manifeste de l'application. Vous pouvez utiliser le concepteur du manifeste pour définir ou modifier une ou plusieurs propriétés.

Application	Actifs visuels	Capacités	Déclarations	URI de contenu	Packages
Utilisez cette page pour définir les propriétés qui identifient et décrivent votre application.					
Nom complet : <input type="text" value="ClientConvertisseurV1"/>					
Point d'entrée : <input type="text" value="ClientConvertisseurV1.App"/>					
Langue par défaut : <input type="text" value="fr-FR"/> Plus d'informations					
Description : <input type="text" value="ClientConvertisseurV1"/>					
Rotations prises en charge : Paramètre optionnel indiquant les préférences d'orientation de l'application.					
<input type="checkbox"/> Paysage <input type="checkbox"/> Portrait <input type="checkbox"/> Paysage (renversé) <input type="checkbox"/> Portrait (renversé)					
Notification de l'écran de verrouillage : <input type="text" value="(non défini)"/>					
Groupe de ressources : <input type="text"/>					
Mise à jour de la vignette :					
met à jour la vignette de l'application en interrogeant régulièrement un URI. Le modèle d'URI peut contenir les jetons "{language}" et "{region}" qui seront remplacés au moment de l'exécution pour générer l'URI à interroger.					
Plus d'informations					
Périodicité : <input type="text" value="(non défini)"/>					
Modèle d'URI : <input type="text"/>					

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` : `rootFrame.Navigate(typeof(MainPage), e.Arguments)` ;

- 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

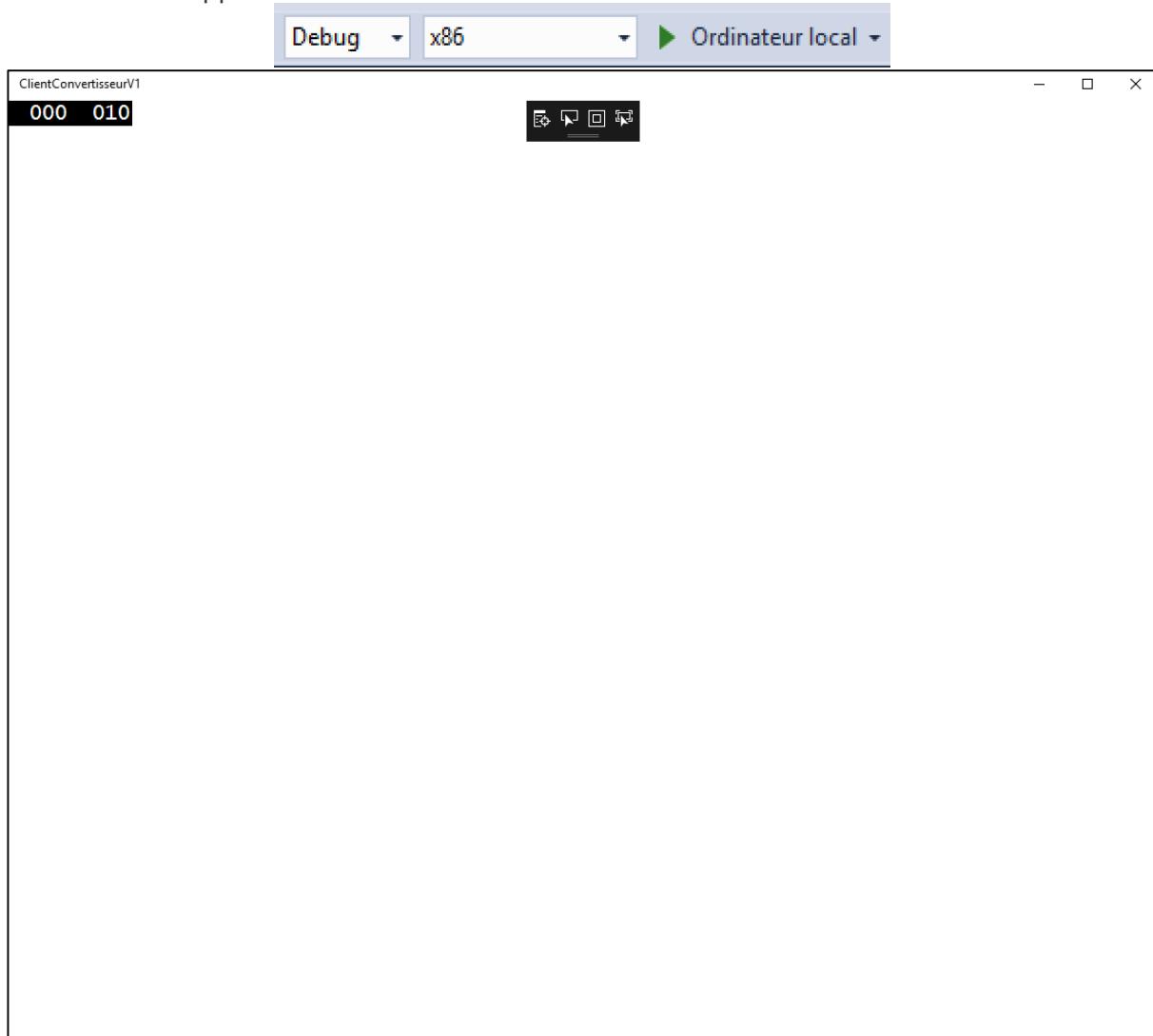
s'exécute au démarrage de votre application.

- Un fichier `.pfx` utilisé pour le déploiement ClickOnce (manifeste signé par un certificat).
- Un fichier `.json` qui indique les dépendances et le framework utilisés. On peut noter la dépendance vers le package `Microsoft.NETCore.UniversalWindowsPlatform` du .NET Core.

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

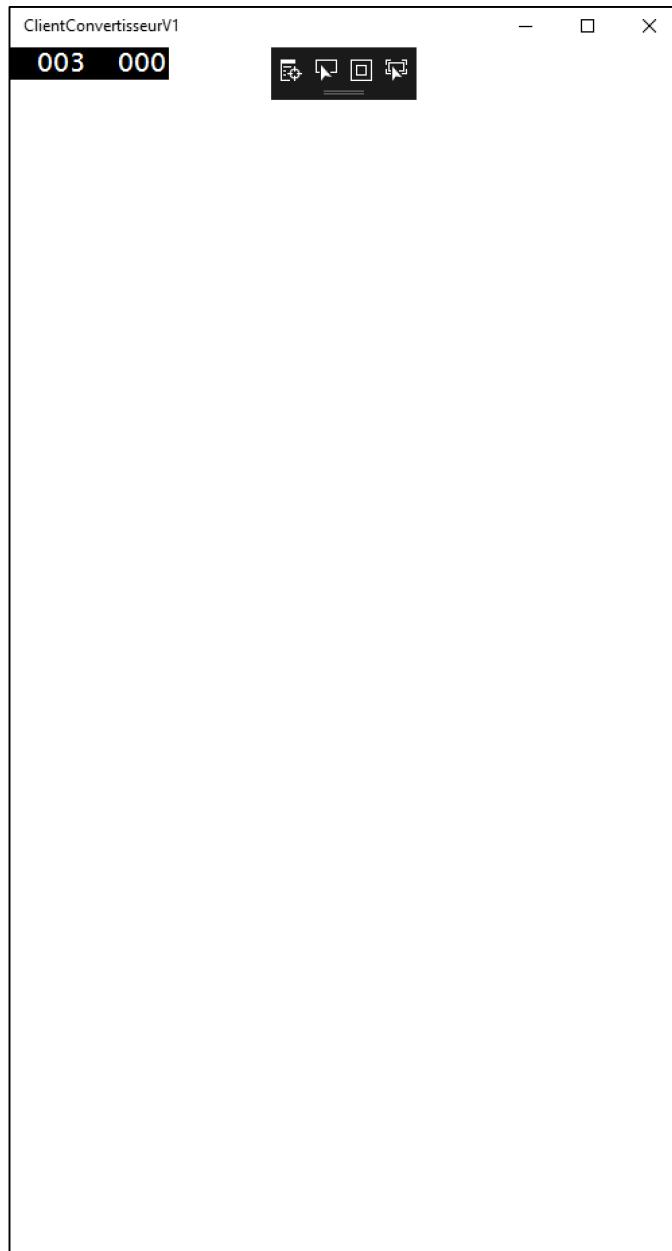
Quel que soit le périphérique utilisé (smartphone, Xbox, tablette, PC), le code de l'application sera le même. Ainsi, c'est Windows qui adaptera l'application au périphérique utilisé (résolution, etc.).

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



Dans ce cas, il s'agit d'une application UWP « classique » (i.e. pour PC). Les chiffres en haut à gauche sont liés à la performance de l'application.

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

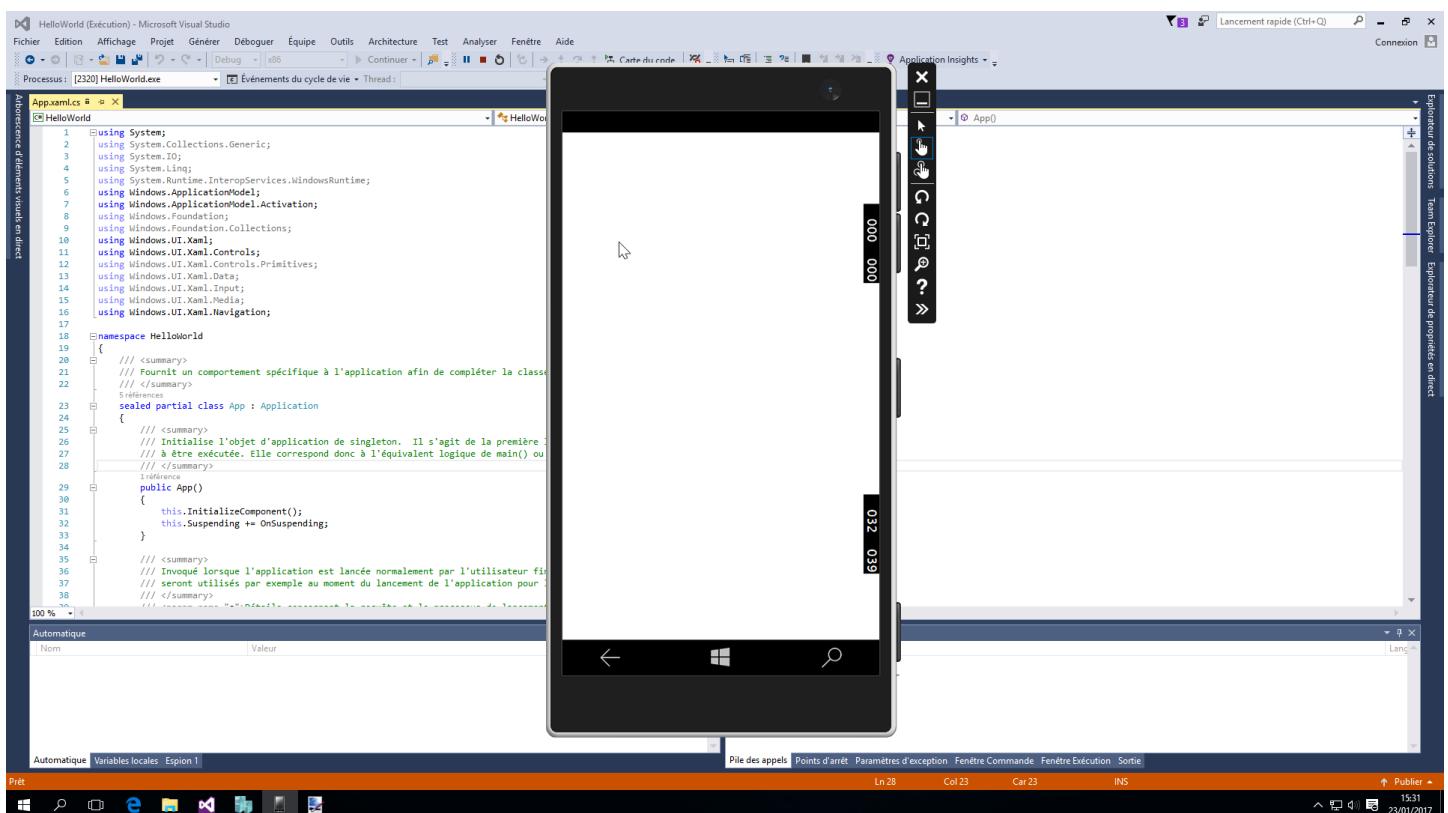
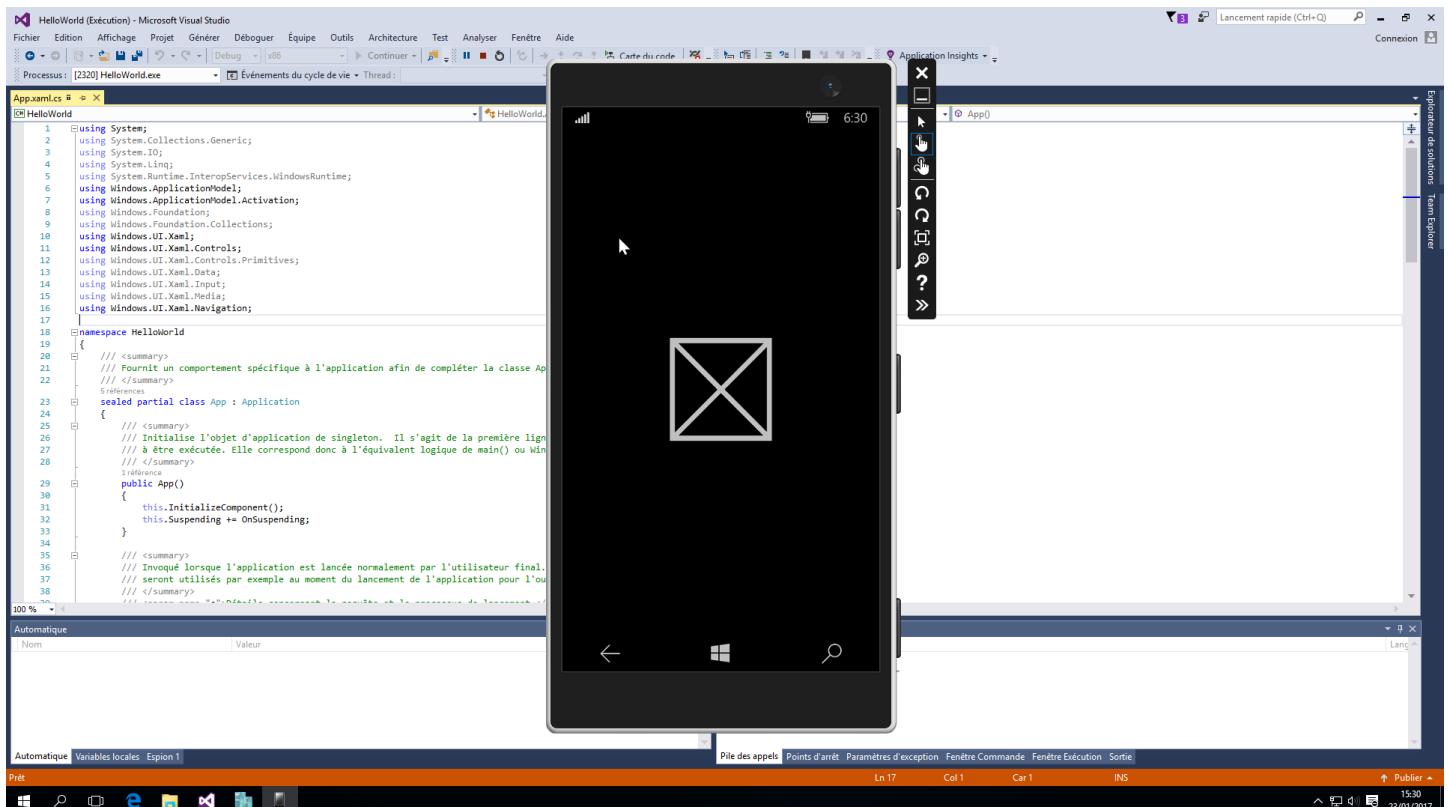


ATTENTION LE TEST SUR L'EMULATEUR NE FONCTIONNE PAS A L'IUT.

Exécuter ensuite l'application sur un émulateur Windows 10 mobile, par exemple :

Debug ▾ x86 ▾ ➤ Mobile Emulator 10.0.14393.0 WVGA 4 inch 512MB ▾

Si elle ne se lance pas automatiquement, lancer l'application *ClientConvertisseurV1* disponible dans les Apps du mobile.



Dans le modèle de projet Application vide, *MainPage* est basé sur le modèle *Page vierge*. Il contient le minimum de XAML et de code pour instancier une *Page*. Cependant, lorsque vous créez une application pour Windows Store, vous devrez ajouter du code supplémentaire. Par exemple, même une simple application d'une page doit s'adapter à différentes dispositions et vues, enregistrer son état si elle est suspendue et restaurer son état lorsqu'elle reprend. Nous reverrons cela par la suite...

Explication des fichiers

Lorsque vous créez un projet à partir du modèle **Application vide**, Visual Studio crée une application qui contient quelques 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).

```
<Application  
    x:Class="ClientConvertisseurV1.App"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:local="using:ClientConvertisseurV1"  
    RequestedTheme ="Dark">  
  
</Application>
```

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 des méthodes destinées à gérer l'activation et la suspension de l'application.

MainPage.xaml

Le fichier MainPage.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 (Windows Universel) **Visual C#** » crée une nouvelle classe appelée MainPage qui hérite de Page (<https://msdn.microsoft.com/fr-fr/library/windows/apps/windows.ui.xaml.controls.page.aspx>). De plus, il comporte du contenu simple, comme un bouton Précédent et fournit des méthodes de navigation et de gestion d'état.

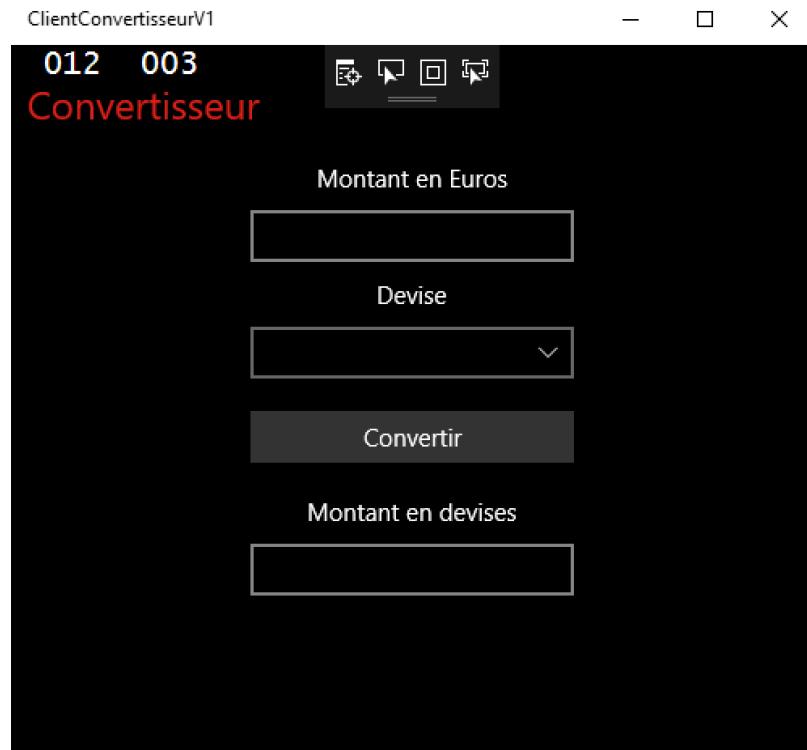
MainPage.xaml.cs

MainPage.xaml.cs est la page code-behind de MainPage.xaml. C'est ici que vous ajoutez la logique de votre application et les gestionnaires d'événements. Le modèle « Application vide (Windows Universel) **Visual C#** » comporte deux méthodes dans lesquelles vous pouvez enregistrer et charger l'état de page.

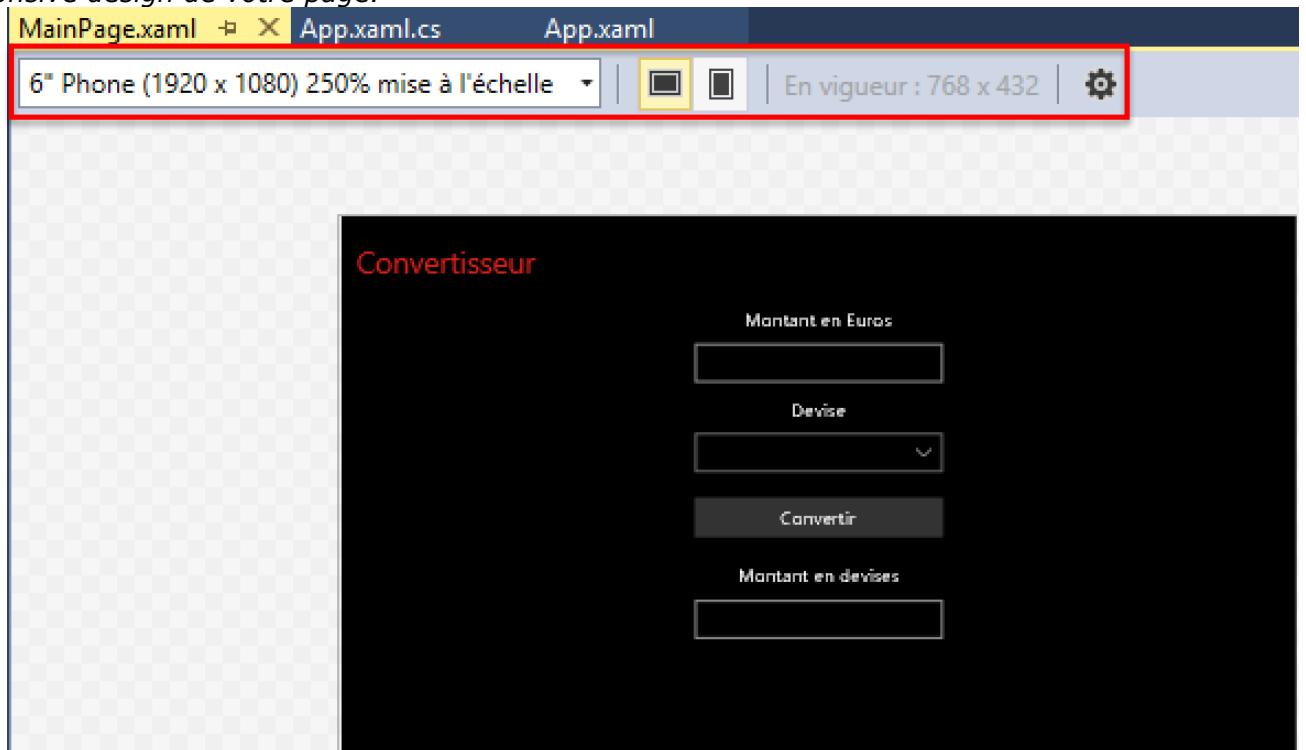
Dans le fichier MainPage.xaml, ajouter les contrôles suivants :

- Remplacer la grille par un RelativePanel
- Dans le panel précédent, ajouter :
 - o 3 TextBlock
 - o 1 ComboBox
 - o 2 TextBox dont un en lecture seule.
 - o 1 Button

Pour placer les éléments, vous pouvez notamment utiliser les attributs RelativePanel.AlignHorizontalCenterWithPanel et RelativePanel.Below.



Remarque : Vous pourrez configurer différentes résolutions et/ou modes d'affichage pour tester le responsive design de votre page.

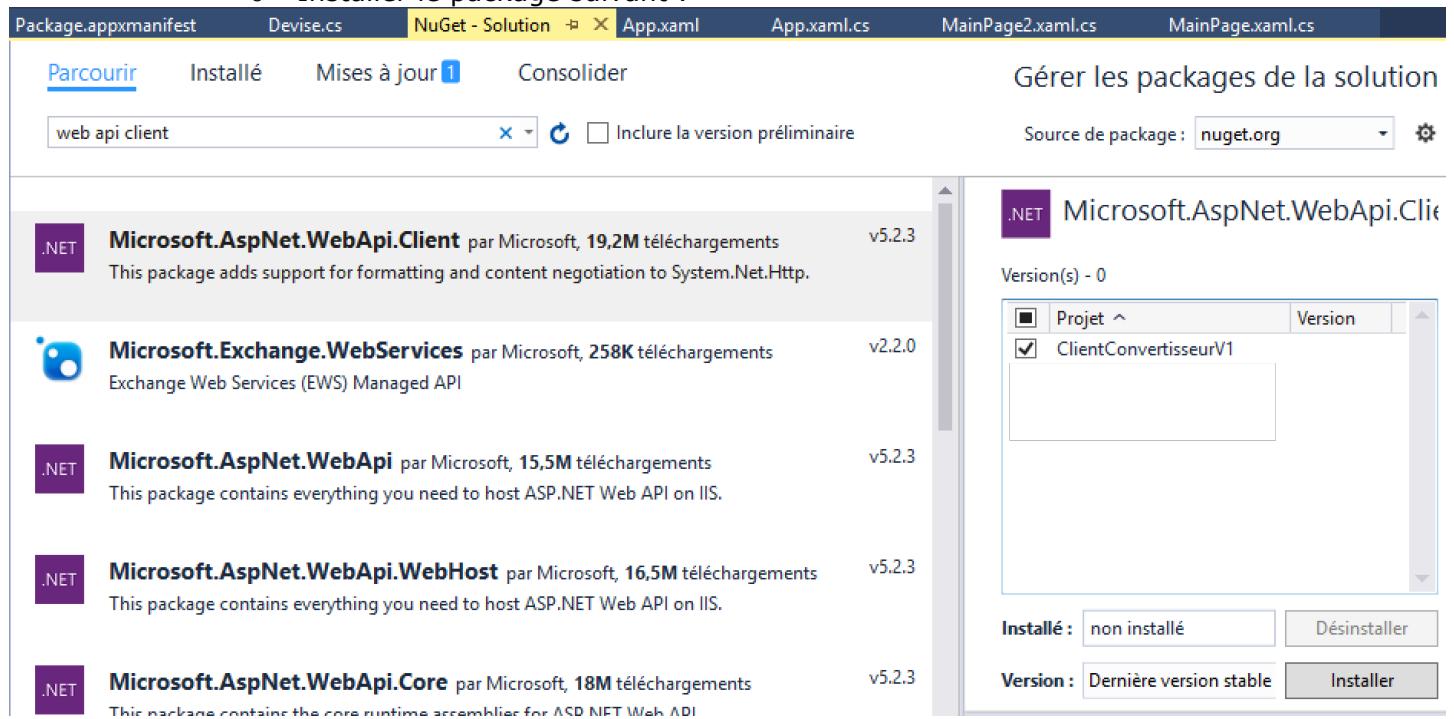


Codage de l'alimentation de la ComboBox

- Créer un dossier `Model` et y ajouter la classe `Devise` correspondant à celle du WS (seuls les properties et éventuellement les attributs sont nécessaires).
- Créer un dossier `Service` et une classe `WSService`. Dans cette classe, ajouter le code permettant de se connecter au WS et créer la méthode `public async Task<List<Devise>> getAllDevisesAsync()` retournant la liste des devises.
 - Utiliser la classe `httpClient` : [https://msdn.microsoft.com/fr-fr/library/system.net.http.httpclient\(v=vs.118\).aspx](https://msdn.microsoft.com/fr-fr/library/system.net.http.httpclient(v=vs.118).aspx)
 - Exemple de code ici :

<https://www.asp.net/web-api/overview/advanced/calling-a-web-api-from-a-net-client>

- Vous devez installer le package NuGet « WebAPI.Client »
 - Aller dans le menu *Outils > Gestionnaire de package NuGet > Gérer les packages NuGet pour la solution.*
 - Installer le package suivant :



- L'appel du WS est : <http://localhost:PORT/api/devise>. Donc l'Uri sera <http://localhost:PORT/api/>. La méthode GetAsync prend en paramètre un String correspondant au nom du controller à appeler, ici "devise".
- Bonne pratique : appliquer le pattern Singleton de façon que cette classe ne soit instanciée qu'une seule fois.

c. Code du fichier MainPage.xaml

- Utilisation du binding du contrôle ComboBox :

```
<ComboBox x:Name="cbxDevise" ... ItemsSource="{Binding}" SelectedValuePath="Id" DisplayMemberPath="NomDevise"/>
```

d. Code du fichier MainPage.xaml.cs

```
public MainPage()
{
    InitializeComponent();
    ...
    ActionGetData();
}
private async void ActionGetData()
{
    var result = await Devise.GetAllDevisesAsync();
    this.cbxDevise.DataContext = new List<Devise>(result);
}
```

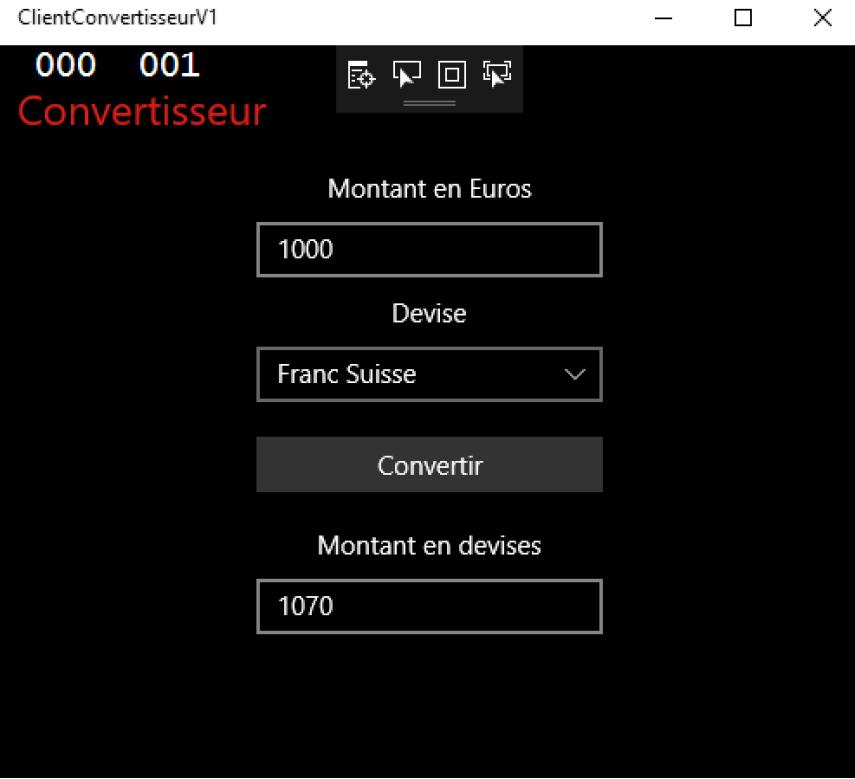
Remarques :

- A la place du binding XAML, on peut bien sûr écrire le code suivant dans le code-behind :

```
this.cbxDevise.ItemsSource = devises;
this.cbxDevise.SelectedValuePath = "Id";
this.cbxDevise.DisplayMemberPath = "NomDevise";
```
- Plus de détails sur `async/await` et les tasks asynchrones, ici : <http://fdorin.developpez.com/tutoriels/csharp/threadpool/part3/>

Codage du calcul

A vous de jouer...

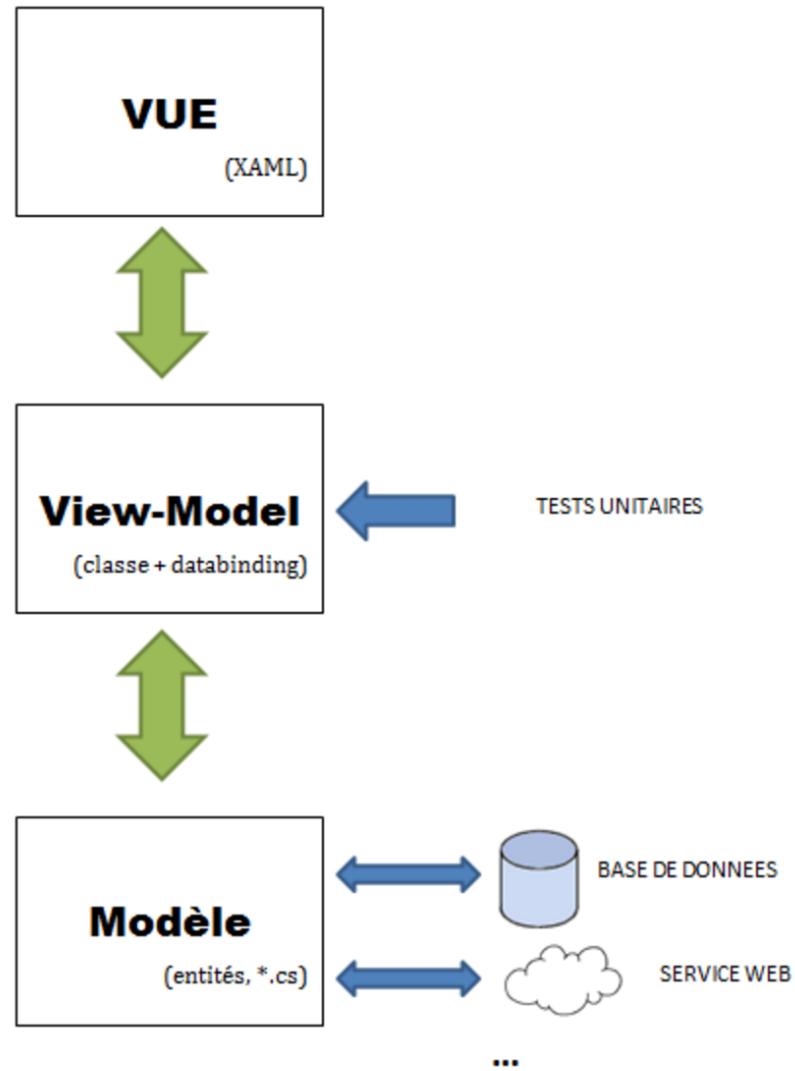


2.2. Client V2 (version MVVM)

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 patron de conception MVVM

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 a en charge de faire les traitements et d'accéder au modèle.

Création du projet et installation des packages NuGet

Ajouter un nouveau projet nommé ClientConvertisseurV2 de type « Application vide (Windows universel) » à la solution ClientsConvertisseur. **Bien utiliser le framework 4.6.1.**
Définir ce projet comme projet de démarrage.

Ajouter le package NuGet « MVVM Light ».

The screenshot shows the NuGet package manager interface. The search bar at the top contains the text 'Mvvm'. Below the search bar, there are tabs for 'Parcourir', 'Installé', 'Mises à jour', and 'Consolidier'. On the right side, there's a section titled 'Gérer les packages de la solution' with a dropdown menu set to 'nuget.org' and a gear icon.

Nom du package	Auteur	Téléchargements	Version
MvvmLight	par Laurent Bugnion (GalaSoft)	547K téléchargements	v5.3.0
The MVVM Light Toolkit is a set of components helping people to get started in the Model-View-ViewModel pattern in Silverlight, WPF, Windows Phone, Windows 10 UWP, Xamarin.Android, Xamarin.iOS...			
MvvmLightLibs	par Laurent Bugnion (GalaSoft)	961K téléchargements	v5.3.0
The MVVM Light Toolkit is a set of components helping people to get started in the Model-View-ViewModel pattern in Silverlight, WPF, Windows Phone, Windows 10 UWP, Xamarin.Android, Xamarin.iOS...			
knockoutjs	par Steven Sanderson	5,62M téléchargements	v3.4.1
A JavaScript MVVM library to help you create rich, dynamic user interfaces with clean maintainable code			
Mvvm	par alexanderpo.com	1,12K téléchargements	v1.0.4
The mvvm pattern for wpf application.			
MvvmFx-MVVM-Light-WinForms	par Tiago Freitas Leal	1,4K téléchargements	v2.0.2
Supports the creation of MVVM Windows Forms applications using a (partial) port of the MVVM Light			

On the right side, a detailed view for the 'MvvmLight' package is shown. It lists 'Version(s) - 0' and shows two projects selected: 'ClientConvertisseurV1' and 'ClientConvertisseurV2'. It also includes 'Installé : non installé' and 'Désinstaller' buttons, and 'Version : Dernière version stable 5.3.0' with an 'Installer' button.

MVVM Light est un framework qui va nous aider à mettre en place le pattern MVVM. Il fournit notamment une architecture de projet compatible MVVM.

Lire cet excellent article sur MVVM Light : <http://blog.soat.fr/2015/06/mvvm-light-toolkit/>

Remarques : il existe d'autres frameworks MVVM tels que Prism, Okra, Caliburn.Micro, etc.

Installer le package NuGet « WebAPI.Client ».

Couche Model

Créer les dossiers « Model » et « Service ». Y créer les mêmes classes que pour le client V1. Modifier les namespace.

Couches ViewModel et View

Créer un dossier `ViewModel`. Ajouter une classe nommée `MainViewModel` héritant de `ViewModelBase`.

```
using GalaSoft.MvvmLight;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClientConvertisseurV2.ViewModel
{
    public class MainViewModel : ViewModelBase
    {
    }
}
```

C'est le ViewModel de notre page `MainPage.xaml`. La philosophie du MVVM demande de limiter au maximum le code dans le fichier code behind de la page (voire aucun code du tout !). C'est donc ce ViewModel qui coordonnera les échanges entre la page et le modèle.

Créer un dossier `View`. Y déplacer la page `MainPage.xaml`.

Modifier le namespace :

- Fichier `cs` : namespace `ClientConvertisseurV2.View`
- Fichier `xaml` :

```
x:Class="ClientConvertisseurV2.View.MainPage"
xmlns:local="using:ClientConvertisseurV2.View"
```

Modifier également le fichier App.xaml.cs.

Copier-coller le code XAML du client V1 dans la page MainPage.xaml du client V2. Supprimer le code événementiel (ex. click="...", etc.).

Exécuter l'application pour voir si la page s'affiche. Pour le moment, la page est toujours chargée grâce à l'appel réalisé dans le fichier App.xaml.cs. Dans un projet MVVM, la coordination est assurée par une classe spécifique ViewModelLocator. C'est cette classe qui va permettre à la page de trouver son ViewModel.

Créer la classe ViewModelLocator dans le dossier ViewModel. Ajouter le code suivant (attention au namespace si votre projet n'a pas le même nom).

```
using GalaSoft.MvvmLight.Ioc;
using Microsoft.Practices.ServiceLocation;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ClientConvertisseurV2.ViewModel
{
    /// <summary>
    /// This class contains static references to all the view models in the
    /// application and provides an entry point for the bindings.
    /// <para>
    /// See http://www.mvvmlight.net
    /// </para>
    /// </summary>
    public class ViewModelLocator
    {
        static ViewModelLocator()
        {
            ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
            SimpleIoc.Default.Register<MainViewModel>();
        }

        /// <summary>
        /// Gets the Main property.
        /// </summary>
        public MainViewModel Main => ServiceLocator.Current.GetInstance<MainViewModel>();
    }
}
```

Remarque : MainViewModel correspond à la classe que nous avons précédemment créée.

Nous utilisons ici SimpleIoc. C'est l'injection de dépendance fournie avec MVVM Light. De manière générale, l'injection de dépendance correspond au D de SOLID, que tout développeur de programmation objet doit connaître.

Pour plus de détails :

- Injection de dépendance :
 - DI : <https://www.tinci.fr/blog/injection-dependance-csharp/>
 - SimpleIoc : <http://www.guruumeditation.net/fr/ioc-et-di-avec-mvvm-light/>
- SOLID : [https://fr.wikipedia.org/wiki/SOLID_\(informatique\)](https://fr.wikipedia.org/wiki/SOLID_(informatique))

Modifier le fichier App.xaml. Ajouter le code suivant :

```
<Application
    x:Class="ClientConvertisseurV2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:ClientConvertisseurV2"
xmlns:vm="using:ClientConvertisseurV2.ViewModel"
RequestedTheme="Dark">

<Application.Resources>
    <!--Global View Model Locator-->
    <vm:ViewModelLocator x:Key="Locator"/>
</Application.Resources>
</Application>

```

S'il vous indique qu'il ne connaît pas `ViewModelLocator`, c'est qu'il est nécessaire de générer l'application. Exécuter l'application.

Ajout du Binding dans la vue `MainPage.xaml` :

```

<Page
    x:Class="ClientConvertisseurV2.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ClientConvertisseurV2"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    DataContext="{Binding Main, Source={StaticResource Locator}}">

    ...
        <ComboBox x:Name="cbxDevise" ItemsSource="{Binding ComboBoxDevises}" ...>
    ...
</Page>

```

- Main fait référence à la ligne suivante du fichier `ViewModelLocator` et permet de lier la vue au viewmodel `MainViewModel` :

```
public MainViewModel Main => ServiceLocator.Current.GetInstance<MainViewModel>();
```

- Locator fait référence à la key du fichier `App.xaml`.

```

<Application
    x:Class="ClientConvertisseurV2.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ClientConvertisseurV2"
    xmlns:vm="using:ClientConvertisseurV2.ViewModel"
    RequestedTheme="Dark">

    <Application.Resources>
        <!--Global View Model Locator-->
        <vm:ViewModelLocator x:Key="Locator"/>
    </Application.Resources>
</Application>

```

- Nous reviendrons sur la property `ComboBoxDevises` (sur laquelle le binding est réalisé) dans la partie suivante.

Ajout du code dans le fichier `MainViewModel.cs` :

- a. Créer une property (*propfull*) `ComboBoxDevises` et ajouter le code suivant :

```

private ObservableCollection<Devise> _comboBoxDevises;

public ObservableCollection<Devise> ComboBoxDevises
{
    get { return _comboBoxDevises; }
    set
    {
        _comboBoxDevises = value;
        RaisePropertyChanged(); // Pour notifier de la modification de ses données
    }
}

```

```
    }  
}
```

On ne peut utiliser une List pour assurer le binding. Nous sommes obligés d'utiliser une ObservableCollection.

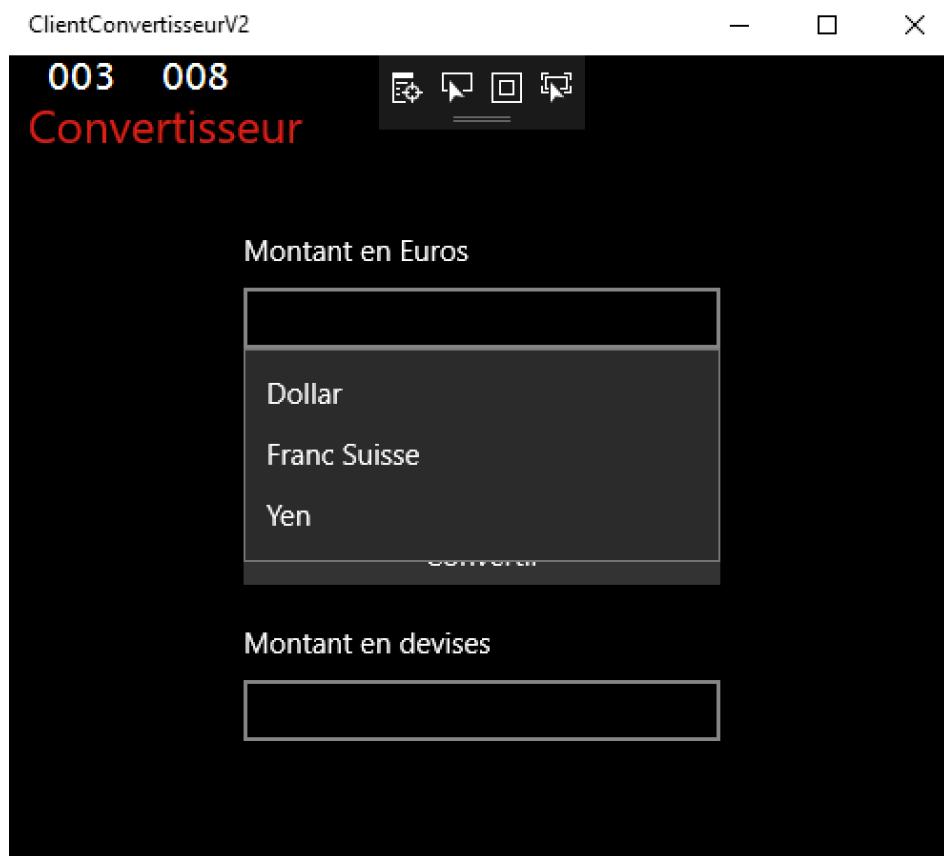
- <http://www.supinfo.com/articles/single/472-wpf-mvvm-data-binding>
- <https://channel9.msdn.com/Series/Windows-Phone-8-1-Development-for-Absolute-Beginners/Part-18-Understanding-MVVM-ObservableCollection-T-and-INotifyPropertyChanged>

b. Créer le constructeur et ajouter la méthode ActionGetData()

```
public MainViewModel()  
{  
    ActionGetData();  
}  
  
private async void ActionGetData()  
{  
    var result = await ....getAllDevisesAsync();  
    ComboBoxDevises = new ObservableCollection<Devise>(result);  
}
```

Modifier le code de l'appel à la méthode getAllDevisesAsync du WS.

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. Le XAML dispose d'un mécanisme simple de gestion de commandes via l'interface ICommand (<https://msdn.microsoft.com/fr-fr/library/system.windows.input.icommand%28v=vs.95%29.aspx>). Par exemple, le contrôle Button possède (par héritage) une propriété Command du type ICommand (<https://msdn.microsoft.com/fr-fr/library/system.windows.controls.button%28v=vs.95%29.aspx>).

fr/library/system.windows.controls.primitives.buttonbase.command%28v=vs.95%29.aspx) 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 (<http://blog.soat.fr/2015/06/mvvm-light-toolkit/>).

- Dans le fichier XAML, ajouter le code suivant :

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

- Dans le fichier MainViewModel, ajouter le code suivant permettant de gérer le bouton :

```
public ICommand BtnSetConversion { get; private set; }
```

```
public MainViewModel()
{
    ActionGetData();
    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, il va falloir récupérer la valeur saisie dans le champ « Montant en euros », ainsi que la devise sélectionnée dans la ComboBox.

Nous allons voir comment récupérer la valeur du TextBox.

- Ajouter un binding sur la propriété Text du contrôle TextBox

```
<TextBox x:Name="txtMontantEuros" ... Text="{Binding MontantEuros, Mode=TwoWay}" />
```

Ici, nous avons défini la propriété Binding.Mode ([https://msdn.microsoft.com/fr-fr/library/system.windows.data.binding.mode\(v=vs.110\).aspx](https://msdn.microsoft.com/fr-fr/library/system.windows.data.binding.mode(v=vs.110).aspx)) à TwoWay afin de pouvoir récupérer la valeur de la vue dans le ViewModel. Le mode par défaut est OneWay en UWP (ViewModel vers View).

- Définir la property MontantEuros sur laquelle porte le binding dans le ViewModel :

```
private string _montantEuros;
```

```
public string MontantEuros
{
    get { return _montantEuros; }
    set {
        _montantEuros = value;
        RaisePropertyChanged();
    }
}
```

Faire de même pour la propriété SelectedItem de la ComboBox afin de récupérer l'objet Devise sélectionné :

- <ComboBox x:Name="cbxDevise" ... SelectedItem="{Binding ComboBoxDeviseItem, Mode=TwoWay}" />

- Créer une property nommée ComboBoxDeviseItem (type : Devise).

Coder ensuite le calcul et l'affichage dans le TextBox « Montant en devises ». Vous utiliserez également le binding (mode : OneWay) et devrait créer une property.

Exceptions

Gérer les exceptions (WS non disponible, valeurs saisies non valides ou manquantes, etc.). Vous pourrez utiliser la classe Windows.UI.Popups.MessageDialog permettant d'afficher des fenêtres popup. Vous utiliserez ShowAsync(), méthode asynchrone à utiliser avec le couple async/await. Penser à écrire du code générique...

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 ne figure dans le fichier `MainPage.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. Peu importe si vous ne respectez pas parfaitement MVVM, le principe de ce pattern est de vous aider dans la réalisation de votre application et surtout dans sa maintenabilité.

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

3. Modification

3.1. Ajout d'une nouvelle page

Dans le dossier `View`, ajouter une nouvelle page permettant de convertir un montant en devise en un montant en euros.

Modifier le code du `ViewModelLocator` pour prendre en charge cette seconde page. 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(MainPage), e.Arguments);
```

3.2. Création d'un menu Hamburger

Dans le dossier `View`, ajouter une page nommée `RootPage` contenant un contrôle `SplitView`.

Pour le contenu du fichier XAML, cf. https://blogs.msdn.microsoft.com/quick_thoughts/2015/06/01/windows-10-splitview-build-your-first-hamburger-menu/



Cf. point suivant pour coder

Dans le fichier code-behind :

- Constructeur :

```
public RootPage(Frame frame)
{
    this.InitializeComponent();
    this.MySplitView.Content = frame;
    (MySplitView.Content as Frame).Navigate(typeof(MainPage));
}
```

Le constructeur prend en paramètre la page à charger dans le Content du SplitView. Par défaut, nous chargeons ici la page `MainPage` correspondant au convertisseur Euros -> Devise.

- Code du bouton Hamburger (premier bouton) :

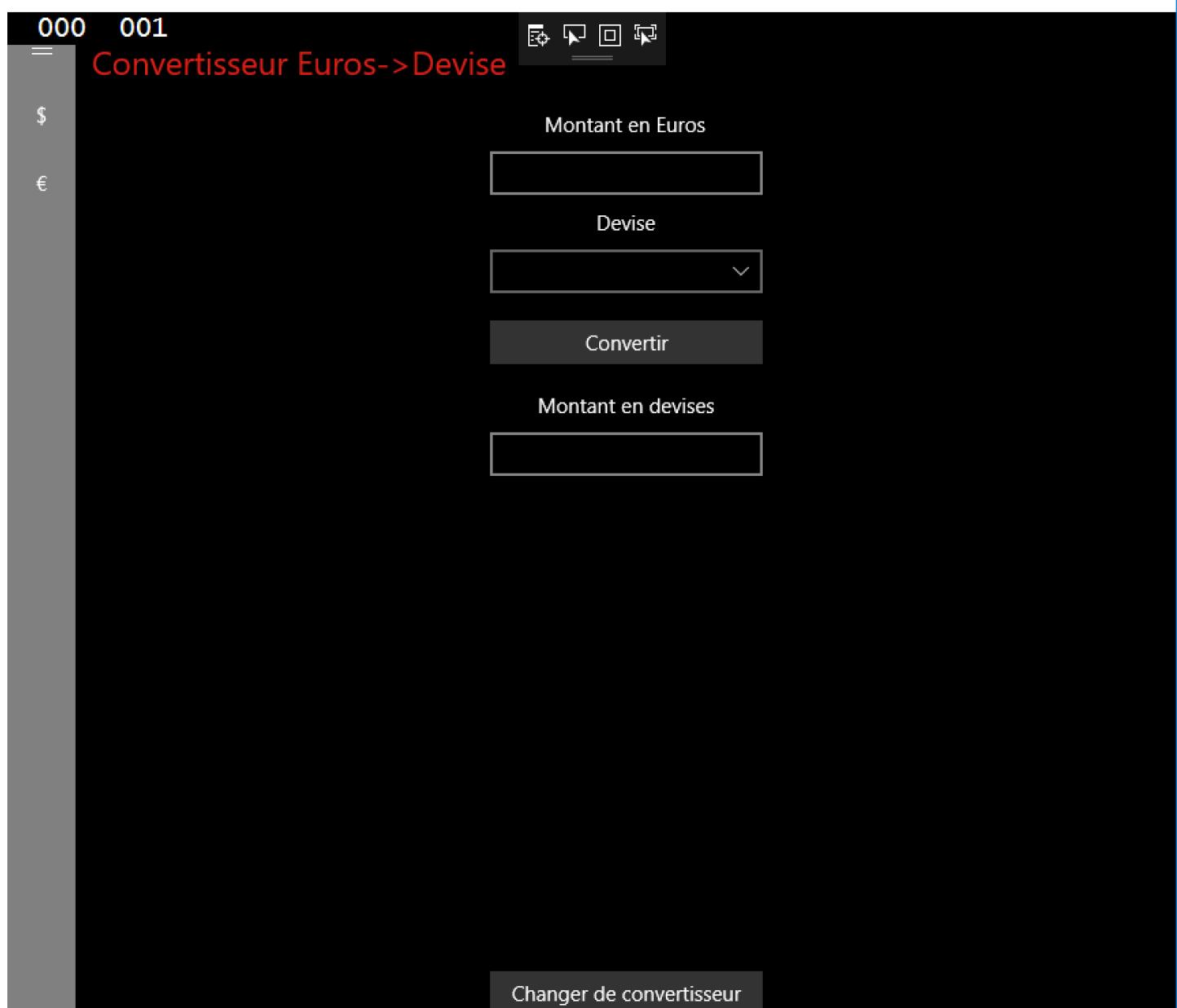
```
MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen;
```

- Ajouter le code des deux autres boutons. Il s'agit juste de naviguer vers la bonne page.

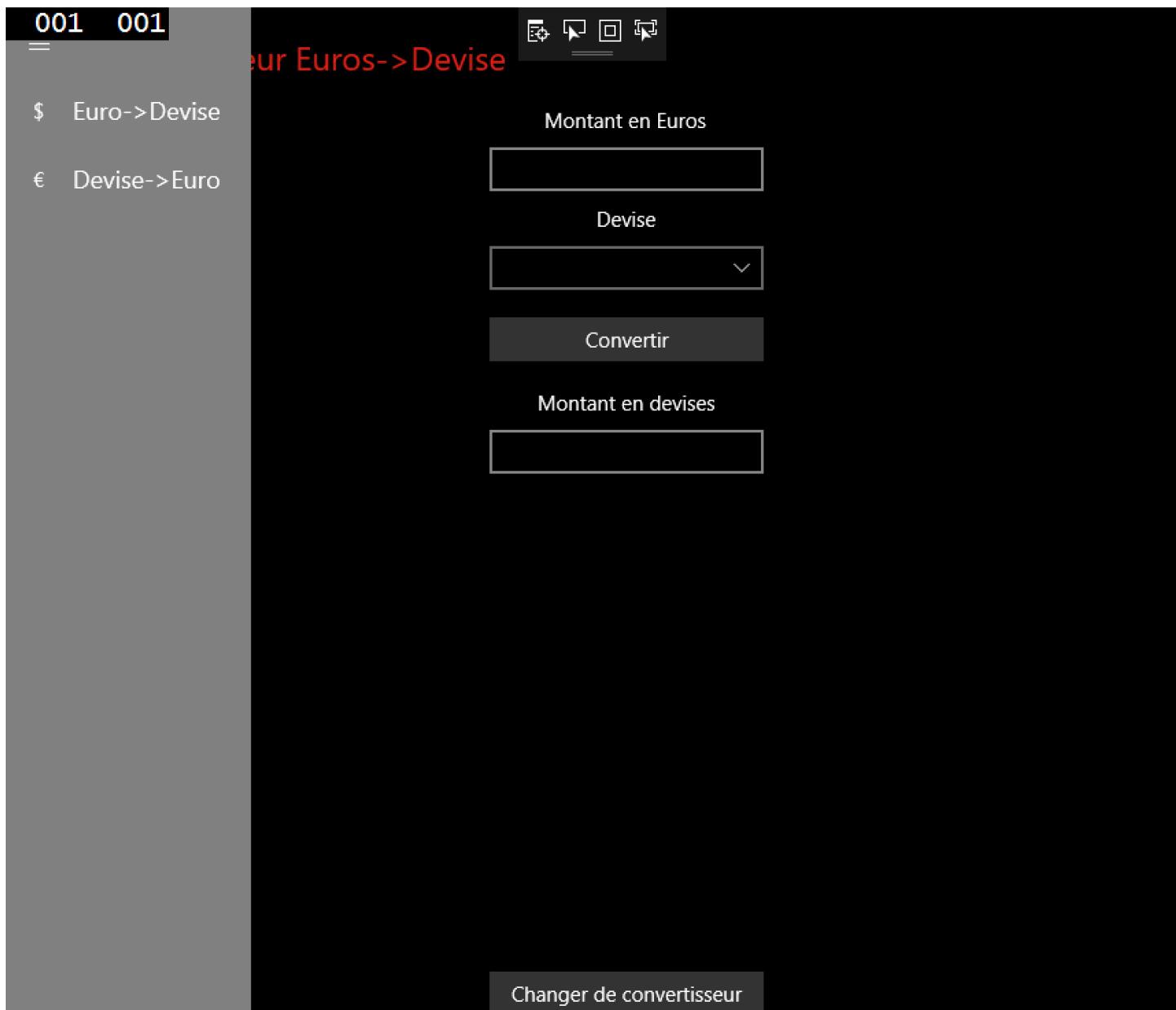
Modifier le code suivant dans le fichier `App.xaml.cs` : `Window.Current.Content = new RootPage(rootFrame);`

Résultat :

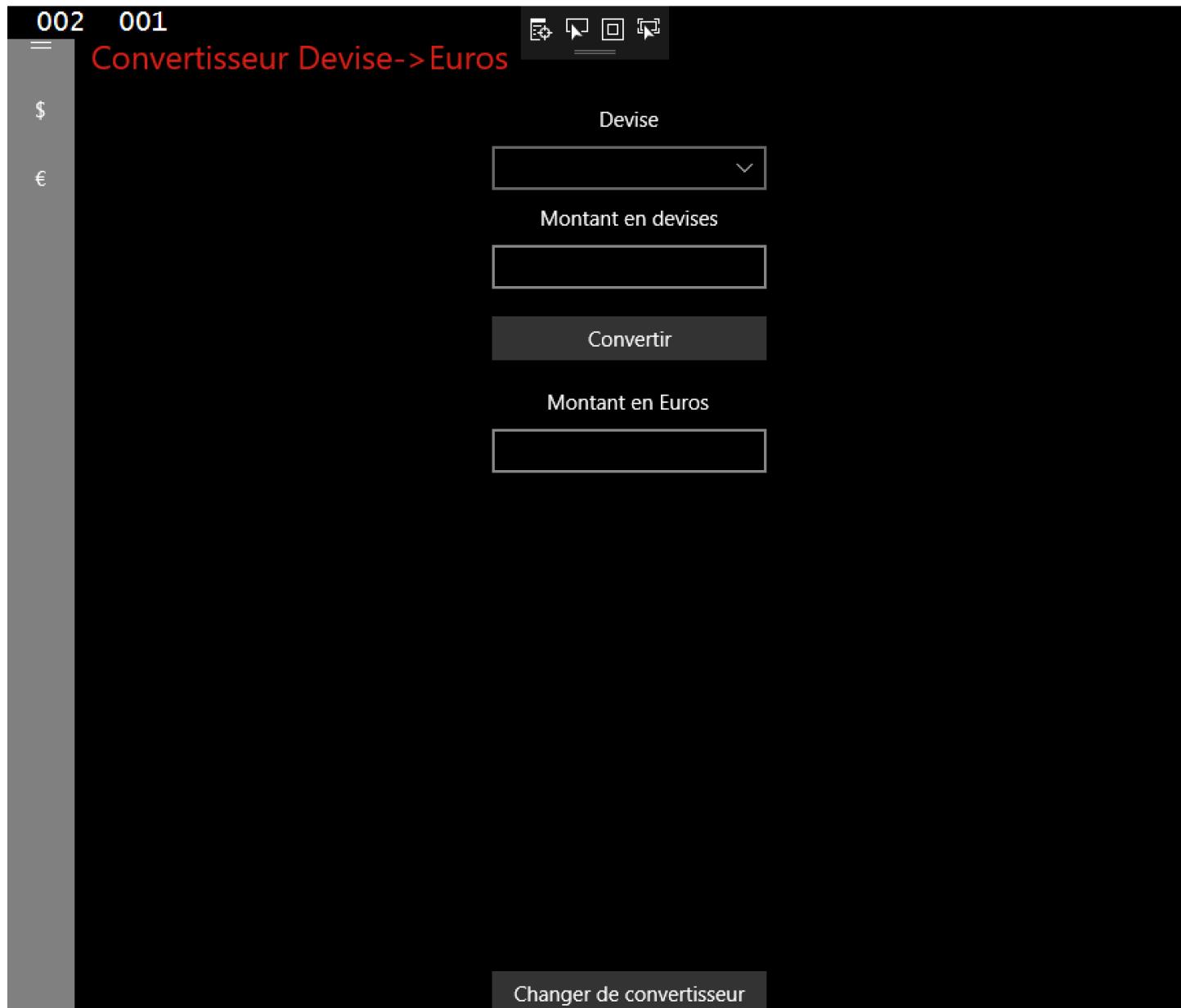
ClientConvertisseurV2



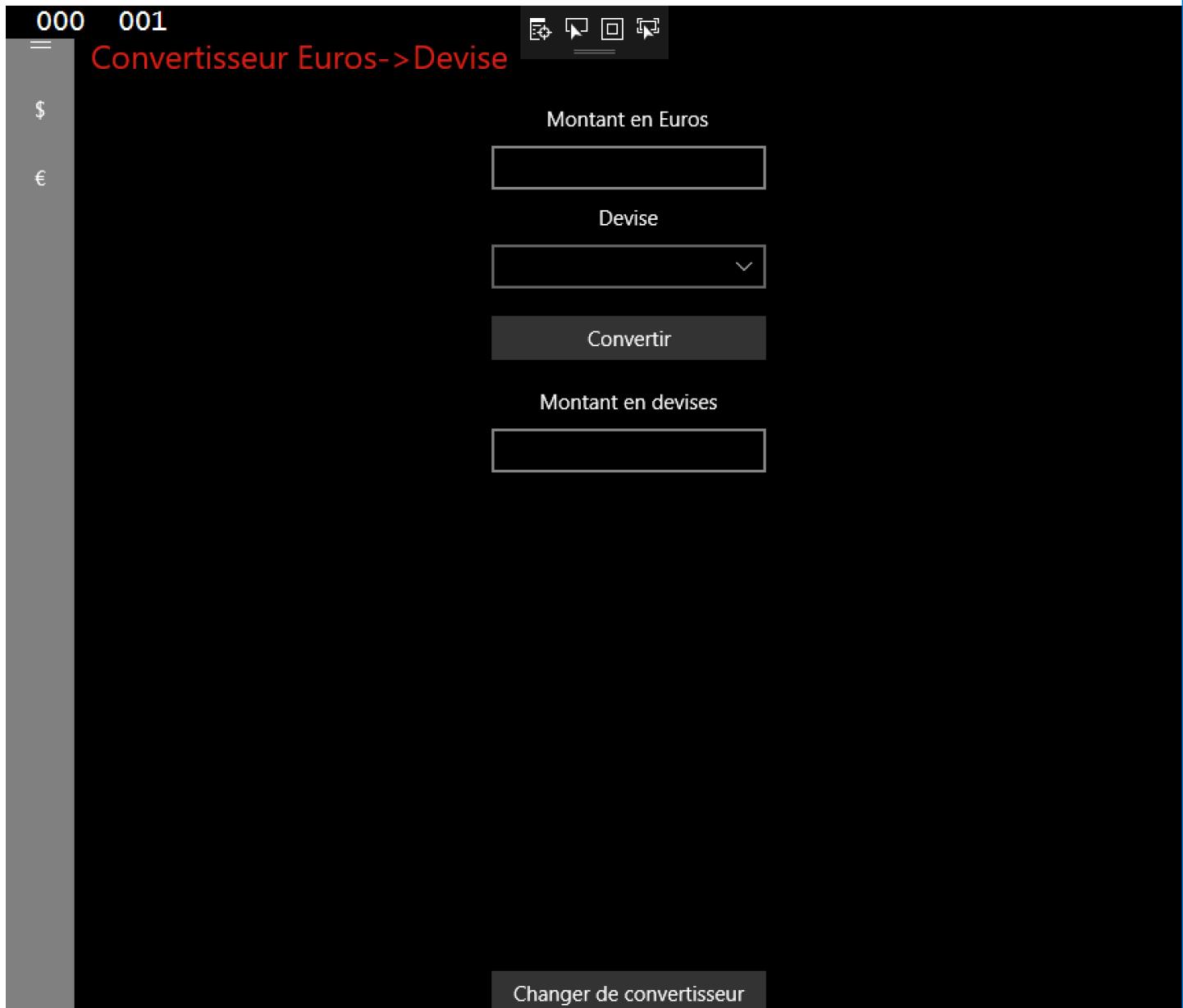
Clic sur le 1^{er} bouton :



Clic sur le 3^{ème} bouton :



Clic sur le 2^{ème} bouton :



Remarque : un bouton permettant de changer de convertisseur a été ajouté en bas du *RelativePanel*, juste à des fins pédagogiques. Le code (à intégrer dans le *ViewModel*) :

```
private void ActionChangeConvertisseur()
{
    RootPage r = (RootPage)Window.Current.Content;
    SplitView sv = (SplitView)(r.Content);
    (sv.Content as Frame).Navigate(typeof(MainPage));
}
```

Ici, nous n'avons pas respecté le pattern MVVM sur la page d'accueil. Il est bien sûr possible de le respecter, mais il y a peu d'intérêt pour ce type de page. Le MVVM a par contre été respecté sur les pages de code.

4. Bonnes pratiques (pour les plus rapides)

4.1. Test de la couche Controller du Web Service

Dans notre cas, il n'est pas utile de tester la couche *Model* car elle ne contient qu'une classe métier simple. Remarquez, que dans le cas d'une couche *Model* générée via un ORM tel que Entity Framework, on ne crée pas non plus de test.

Nous allons donc uniquement tester le contrôleur *DeviseController*.

Regarder le code du fichier ValuesControllerTest.cs. Il ne vous reste plus qu'à faire de même... Vous pourrez utiliser la classe Assert (<https://msdn.microsoft.com/fr-fr/library/microsoft.visualstudio.testtools.unittesting.assert.aspx>) et/ou CollectionAssert (<https://msdn.microsoft.com/fr-fr/library/microsoft.visualstudio.testtools.unittesting.collectionassert.aspx>).

Ne pas oublier de coder la méthode Equals de Devise.

Exemple de code ici :

<https://docs.microsoft.com/en-us/aspnet/web-api/overview/testing-and-debugging/unit-testing-with-aspnet-web-api>

4.2. Test de la couche ViewModel de l'application UWP (PLUS DIFFICILE !!!)

Réaliser les tests de la couche ViewModel. Pour cela, ajouter un projet de tests à la solution.

- https://www.tutorialspoint.com/mvvm/mvvm_unit_testing.htm
- <https://msdn.microsoft.com/en-us/magazine/dn463790.aspx>
- <http://stackoverflow.com/questions/1441534/wpf-mvvm-unit-tests-for-the-viewmodel> (le test d'un projet UWP est identique au test d'un projet WPF – Windows Platform Foundation).