

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 deux API de web service REST :

- .Net Web API basée sur ASP.Net (Framework .Net) fonctionnant sur IIS.
- ASP.NET Core Web API basée sur ASP.Net Core (Framework .Net Core) fonctionnant dans n'importe quel serveur web. C'est ce framework que nous utiliserons.

Ces deux API sont particulièrement recommandées dans le cadre d'interactions avec des mobiles. Elles sont en outre open source, simples à mettre en œuvre, légères et très robustes (reposent sur le framework MVC).

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.

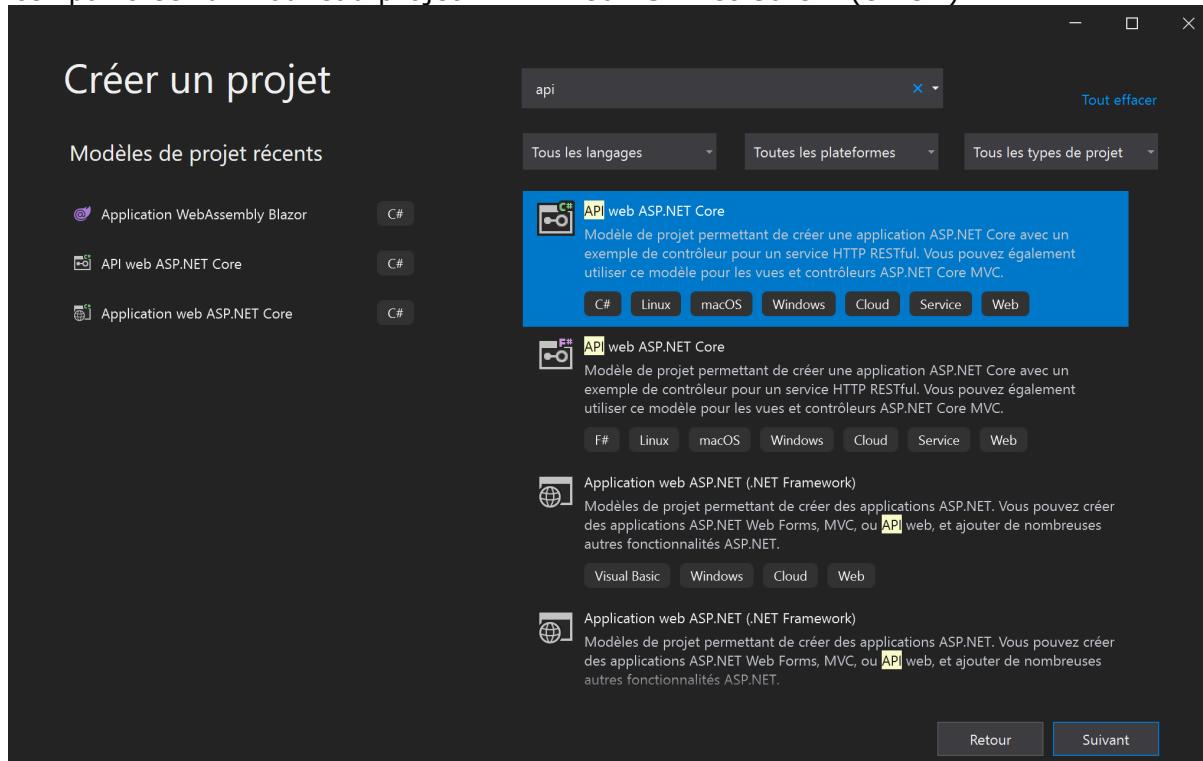
Un WS REST est orienté ressources (et non traitement) par défaut (i.e. obtention de données au format JSON, XML, ATOM, etc.). Il s'agit donc ici de gérer des listes de devises avec leur taux de conversion. Une application cliente simple (Universal Windows) permettra de convertir un montant saisi en euros 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.

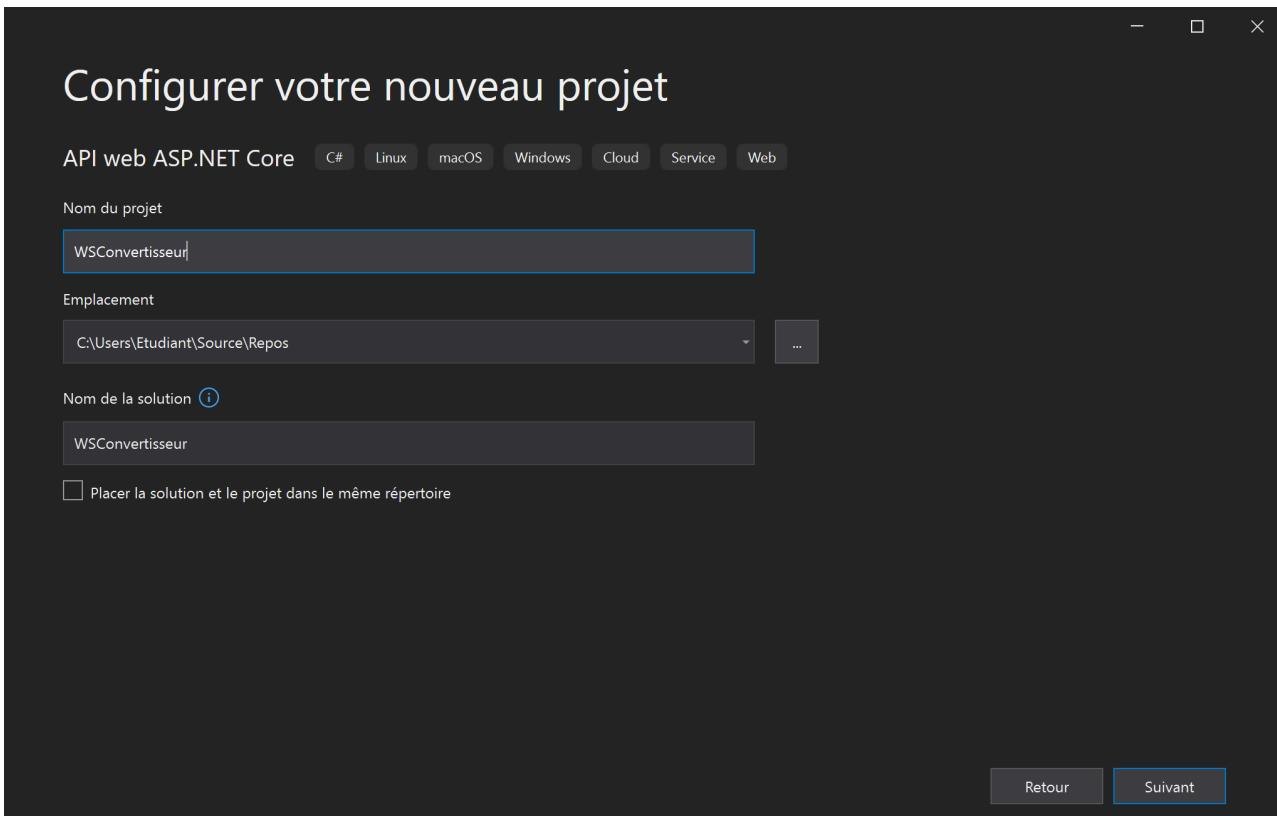
1. Création du WS REST

1.1. Création du projet

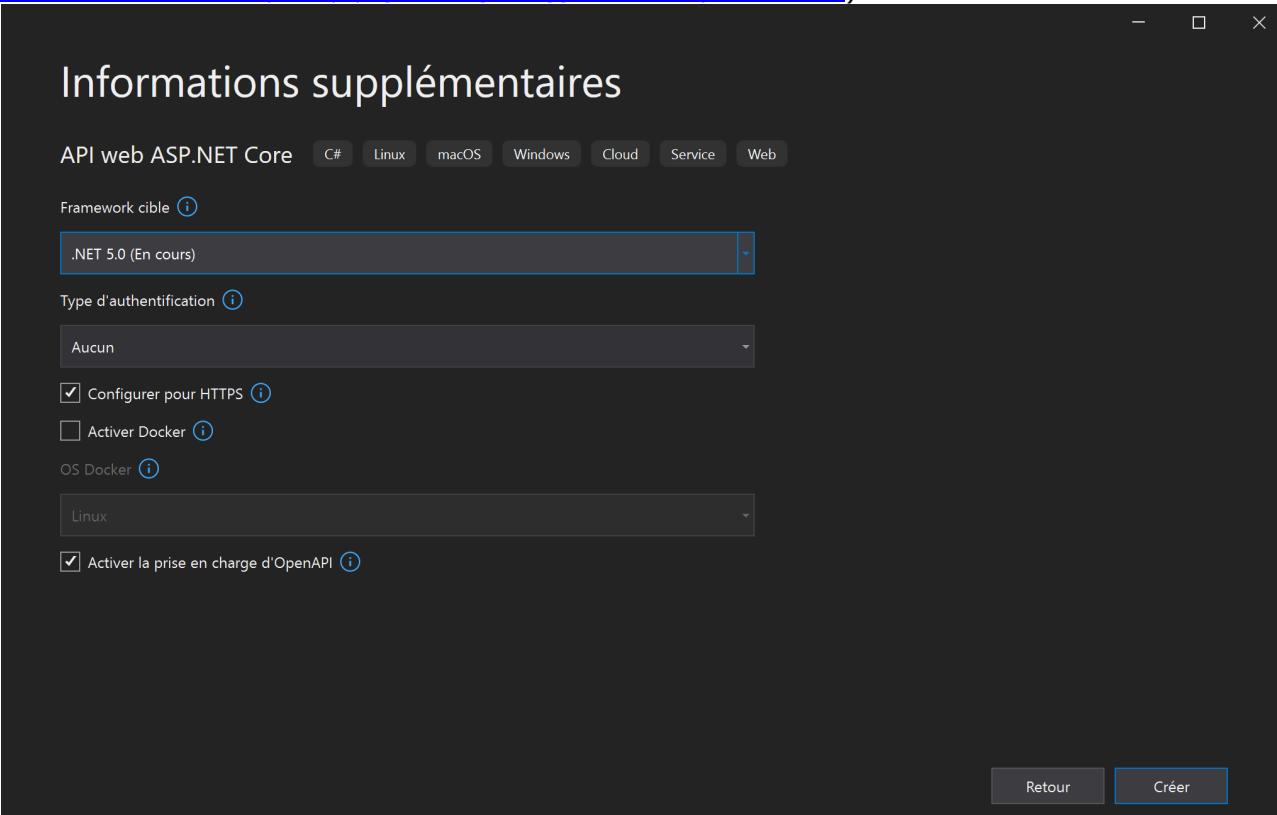
Lancez Visual Studio 2019.

Commencez par créer un nouveau projet « API Web ASP.Net Core » (en C#).





Utiliser le framework **.NET Core 5** (version non LTS). Cocher « Activer la prise en charge d'OpenAPI » qui permet de générer la documentation et tester l'API via Swagger (<https://docs.microsoft.com/fr-fr/aspnet/core/tutorials/web-api-help-pages-using-swagger?view=aspnetcore-5.0>).

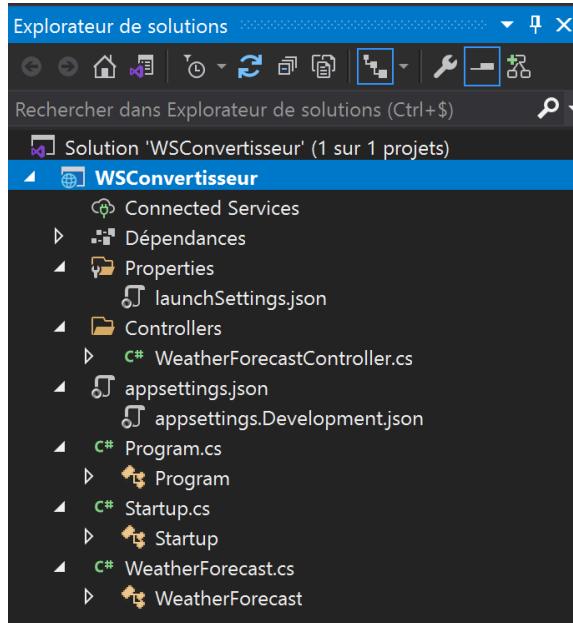


Valider.

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

Fichiers générés :

Principaux fichiers :



- Program.cs : ce fichier contient la méthode Main qui utilise Startup.cs pour configurer l'application.
- Startup.cs : ce fichier contient toute la logique de bootstrapping
- WeatherForecastController.cs : le contrôleur d'API par défaut (exemple d'application) utilisant la classe métier WeatherForecast.cs

Program.cs contient le point d'entrée de notre application (la méthode Main) et l'instanciation du host web :

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

La principale méthode est Host.CreateDefaultBuilder. L'appel de cette méthode initialise une nouvelle instance de la classe HostBuilder avec la configuration par défaut. L'appel de la méthode UseStartup <Startup> indique à l'instance de HostBuilder d'utiliser l'instance de démarrage lors de la configuration d'exécution. L'instance de démarrage est utilisée lors de l'exécution pour configurer différents services et configurer le pipeline de requêtes http.

Startup.cs enregistre les services nécessaires à l'API :

```
public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
```

```

        services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "WSConvertisseur", Version = "v1" });
    });

    // This method gets called by the runtime. Use this method to configure the HTTP request
    pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseSwagger();
            app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json", "WSConvertisseur
v1"));
        }

        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}

```

Dans cette classe, les commentaires par défaut fournis sont explicites.

La méthode `ConfigureServices` est l'endroit où l'on peut ajouter de nouveaux services au conteneur d'injection de dépendance (DI), comme par exemple le service de prise en charge des contrôleurs et des fonctionnalités liées à l'API (`services.AddControllers()` : nouvelle méthode d'inscription de service MVC sans vue de .NET Core). Plus de détails ici :

<https://docs.microsoft.com/fr-fr/dotnet/api/microsoft.extensions.dependencyinjection.mvcservicecollectionextensions.addcontrollers?view=aspnetcore-5.0>

Comme nous avons coché l'option « Activer la prise en charge d'OpenAPI », le service Swagger de description et de test du WS est également ajouté au conteneur de DI.

Il est bien sûr possible de modifier le titre de l'API :

```

services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "API convertisseur", Version = "v1" });
});

```

Remarque : si l'option OpenAPI n'a pas été cochée, voir Annexe en fin de document et suivre la procédure.

Rappel : l'injection de dépendance correspond au D de SOLID, que tout développeur de programmation objet doit connaître.

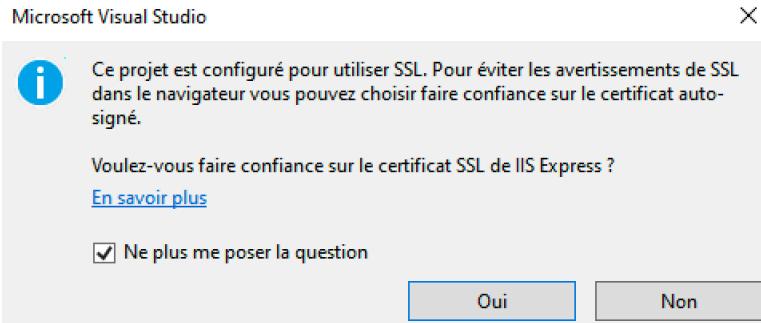
Pour plus de détails :

- Injection de dépendance : <https://www.tinci.fr/blog/injection-dependance-csharp/>
- SOLID : [https://fr.wikipedia.org/wiki/SOLID_\(informatique\)](https://fr.wikipedia.org/wiki/SOLID_(informatique))

La méthode `Configure` est utilisée pour configurer le pipeline de requêtes HTTP. Par défaut, le routing lié aux contrôleurs est ajouté au pipeline (`UseRouting`), de même que la gestion de l'authentification (`UseAuthorization`) et la redirection automatique vers le protocole HTTPS `UseHttpsRedirection` (car nous avons laissé cochée l'option Configurer pour HTTPS).

Cette classe est régie par une convention et les noms de méthode doivent être exactement les mêmes pour que le runtime les appelle.

Exécuter le web service avec IIS Express (seulement disponible sous Windows) pour le tester



Par défaut la documentation de l'API s'affiche (interface Swagger). Il suffit alors de cliquer sur la méthode GET puis sur les boutons « Try it out » et « Execute » pour l'exécuter.

A screenshot of the Swagger UI interface. The URL in the browser is "localhost:44356/swagger/index.html". The top navigation bar shows "Swagger" and "WSConvertisseur v1". Below the navigation, the title is "API convertisseur" with "v1 OAS3" buttons. Under "WeatherForecast", there is a "GET /WeatherForecast" button. In the "Schemas" section, the "WeatherForecast" schema is displayed as a JSON object:

```
WeatherForecast <pre>{<br>    date: string($date-time)<br>    temperatureC: integer($int32)<br>    temperatureF: integer($int32)<br>    summary: string<br>    readOnly: true<br>    nullable: true<br>}</pre>
```

Un ensemble (tableau) d'enregistrements JSON est affiché en guise de résultat :

```

curl -X GET "https://localhost:44356/WeatherForecast" -H "accept: text/plain"

```

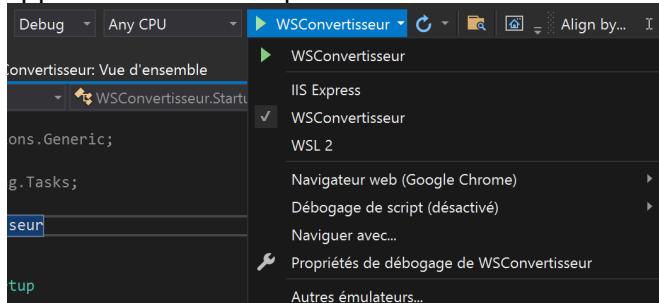
Request URL
https://localhost:44356/WeatherForecast

Server response

Code	Details
200	<p>Response body</p> <pre>[{ "date": "2021-09-20T15:37:00.1852221+02:00", "temperatureC": 41, "temperatureF": 105, "summary": "cool" }, { "date": "2021-09-21T15:37:00.18876+02:00", "temperatureC": 48, "temperatureF": 118, "summary": "Hot" }, { "date": "2021-09-22T15:37:00.1887805+02:00", "temperatureC": -12, "temperatureF": 11, "summary": "Hot" }, { "date": "2021-09-23T15:37:00.1887812+02:00", "temperatureC": 52, "temperatureF": 125, "summary": "Mild" }, { "date": "2021-09-24T15:37:00.1887817+02:00", "temperatureC": -10, "temperatureF": 14 }]</pre> <p>Download</p> <p>Response headers</p> <pre> content-length: 493 content-type: application/json; charset=utf-8 date: Sun, 19 Sep 2021 13:37:00 GMT server: Microsoft-IIS/10.0 x-powered-by: ASP.NET </pre>

En cas de problème d'installation du certificat, il suffit de mettre en commentaire la ligne app.UseHttpsRedirection(); afin de désactiver la redirection vers HTTPS.

On peut également lancer l'application en tant qu'hôte autonome :



Vous devrez aussi installer le certificat.

Ce lancement est beaucoup plus simple et consomme moins de mémoire.

```
C:\Users\Etudiant\Source\Repos\WSConvertisseur\WSConvertisseur\bin\Debug\net5.0\WSConvertisseur.exe
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: C:\Users\Etudiant\Source\Repos\WSConvertisseur\WSConvertisseur
```

Swagger UI

localhost:5001/swagger/index.html

```
curl -X GET "https://localhost:5001/WeatherForecast" -H "accept: text/plain"
```

Request URL

https://localhost:5001/WeatherForecast

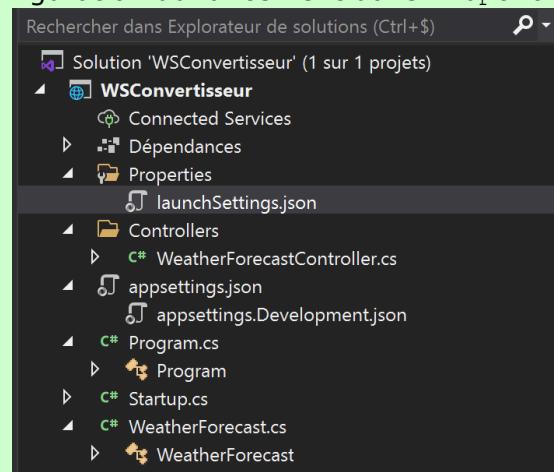
Server response

Code Details

200 Response body

```
[{"date": "2021-09-20T15:41:22.9653081+02:00", "temperatureC": 50, "temperatureF": 121, "summary": "Chilly"}, {"date": "2021-09-21T15:41:23.0219845+02:00", "temperatureC": 34, "temperatureF": 93, "summary": "Mild"}]
```

On peut voir et modifier la configuration du lancement dans Properties/launchSettings.json :



```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:37636",
      "sslPort": 44356
    }
  },
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

```

        "ASPNETCORE_ENVIRONMENT": "Development"
    },
},
"WSConvertisseur": {
    "commandName": "Project",
    "launchBrowser": true,
    "launchUrl": "swagger",
    "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
    },
    "dotnetRunMessages": "true",
    "applicationUrl": "https://localhost:5001;http://localhost:5000"
},
"WSL 2": {
    "commandName": "WSL2",
    "launchBrowser": true,
    "launchUrl": "https://localhost:5001/swagger",
    "environmentVariables": {
        "ASPNETCORE_URLS": "https://localhost:5001;http://localhost:5000",
        "ASPNETCORE_ENVIRONMENT": "Development"
    },
    "distributionName": ""
}
}
}

```

Par exemple, si l'on veut lancer directement le web service (sans passer par swagger) :

```

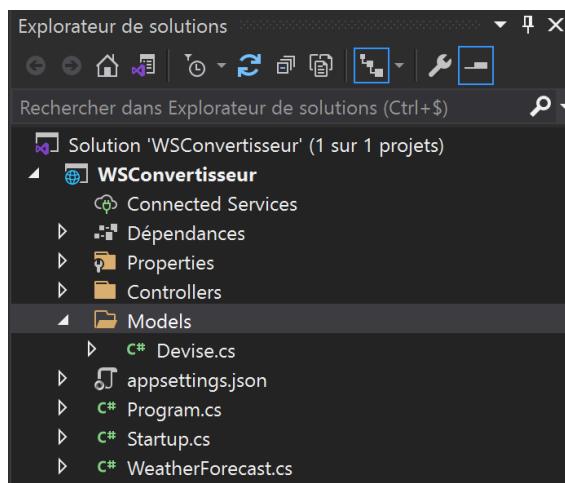
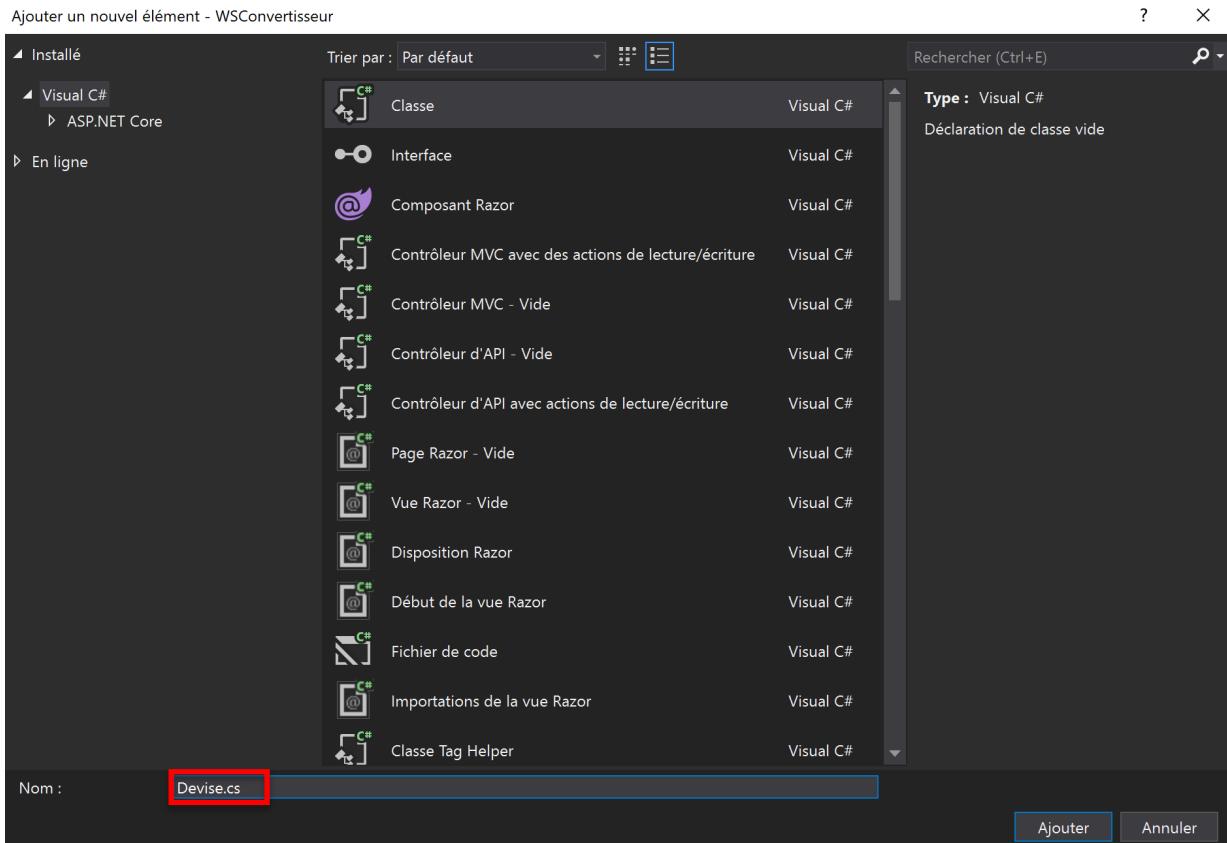
"WSConvertisseur": {
    "commandName": "Project",
    "launchBrowser": true,
    "launchUrl": "weatherforecast",
    "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
    },
    "dotnetRunMessages": "true",
    "applicationUrl": "https://localhost:5001;http://localhost:5000"
},

```

1.2. Création de la couche M

Ajouter un dossier `Models` au projet `WSConvertisseur` : bouton droit de la souris sur le projet puis *Ajouter > Nouveau dossier*.

Dans le dossier `Models`, ajouter la classe `Model Devise`. Pour cela, dans l'explorateur de la solution puis dossier `Models`, ajouter 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)

Pour faciliter la construction des properties, saisir prop (snippet) puis tabulation 2 fois :

```

0 références
public class Devise
{
    prop
    prop
    propdp
    propfull
    propg
}

```

prop
Extrait de code pour une propriété implémentée automatiquement
Version de langage : C# 3.0 ou supérieure
Remarque : appuyez deux fois sur la touche Tab pour insérer l'extrait de code 'prop'.

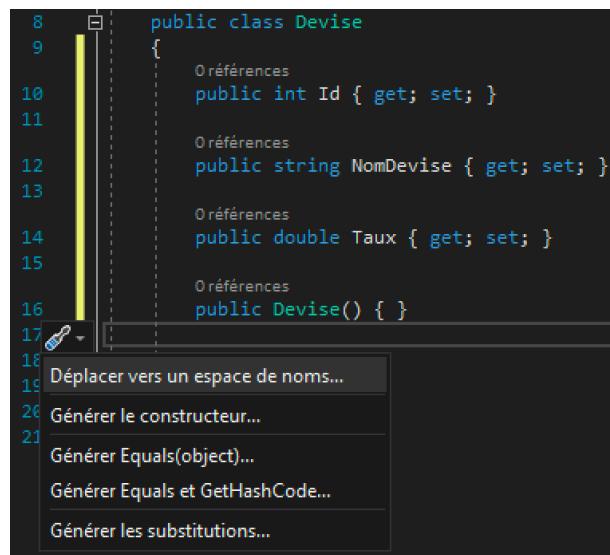
Génère :

0 références | 0 exceptions
public int MyProperty { get; set; }

Modifier ensuite le nom et éventuellement le type de la property.

Ajouter un constructeur paramétré et un constructeur sans paramètre.

Pour générer le constructeur, utiliser le snippet `ctor` puis tab puis tab ou utiliser l'aide à la génération (à droite du numéro de ligne) :



```
public class Devise
{
    O références
    public int Id { get; set; }

    O références
    public string NomDevise { get; set; }

    O références
    public double Taux { get; set; }

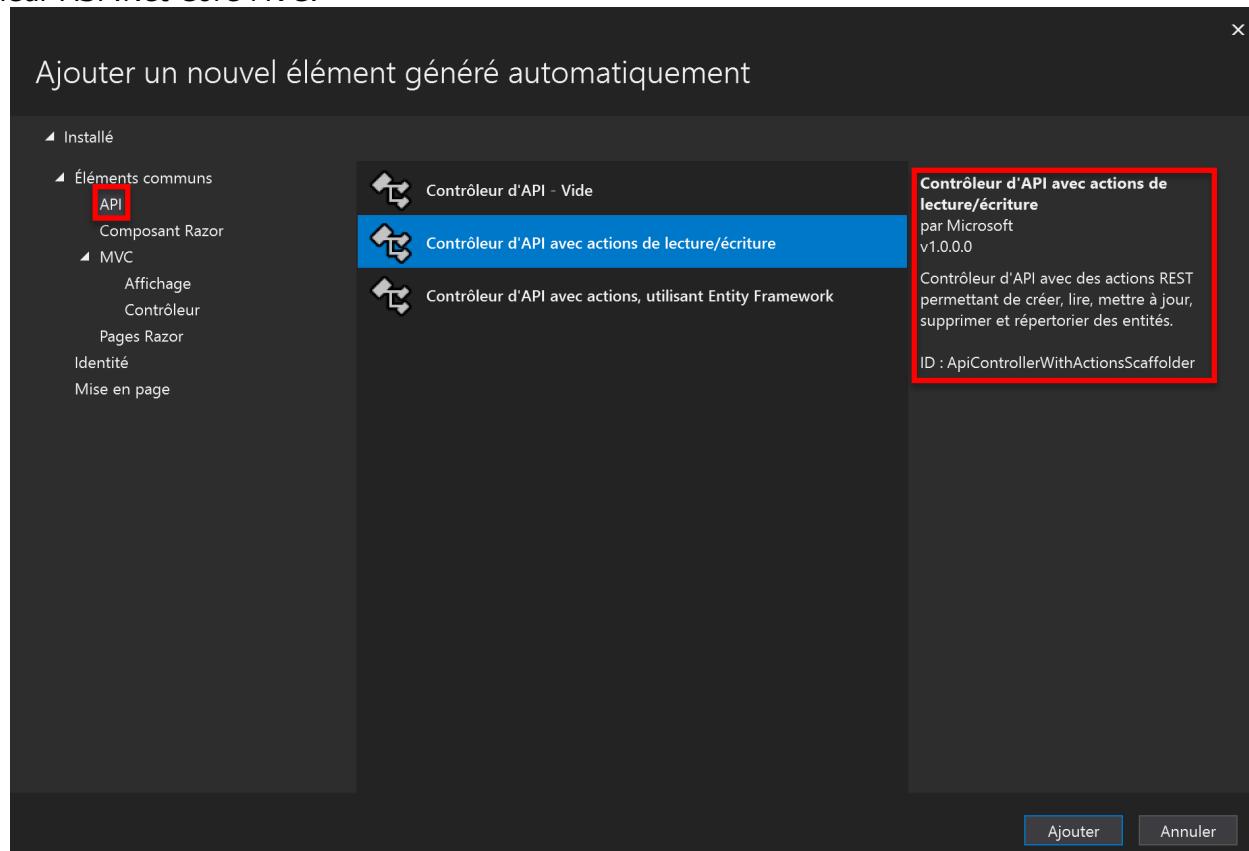
    O références
    public Devise() { }

    18 Déplacer vers un espace de noms...
    19 Générer le constructeur...
    20 Générer Equals(object)...
    21 Générer Equals et GetHashCode...
    22 Générer les substitutions...
```

1.3. Création de la couche C

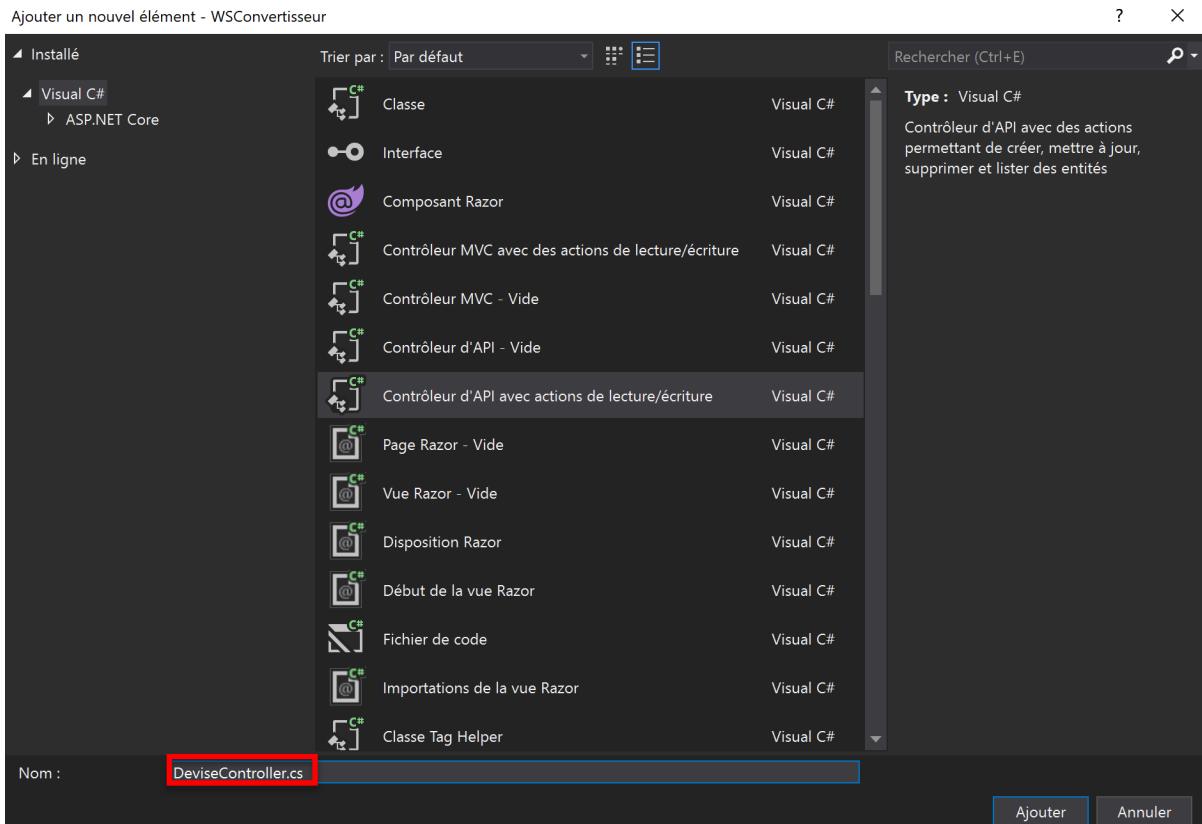
Un *contrôleur* est un objet qui gère les requêtes HTTP et crée la réponse HTTP.

Dans le dossier `Controllers` de la solution, ajouter un nouveau contrôleur (*Ajouter > Contrôleur*). Choisir « Contrôleur d'API avec actions de lecture/écriture » car il s'agit d'un contrôleur d'API REST et non d'un contrôleur ASP.NET Core MVC.



Remarque : sous Mac, il faudra utiliser *Ajouter > Nouvelle génération de modèles automatique*.

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



En sélectionnant le modèle « Contrôleur d'API avec actions de lecture/écriture », Visual Studio génère le squelette de code correspondant aux méthodes GET (2 méthodes), PUT, POST et DELETE.

Cette technique de génération est appelée scaffolding.

```

 9  namespace WSConvertisseur.Controllers
10 {
11     [Route("api/[controller]")]
12     [ApiController]
13     public class DeviseController : ControllerBase
14     {
15         // GET: api/<DeviseController>
16         [HttpGet]
17         public IEnumerable<string> Get()
18         {
19             return new string[] { "value1", "value2" };
20         }
21
22         // GET api/<DeviseController>/5
23         [HttpGet("{id}")]
24         public string Get(int id)
25         {
26             return "value";
27         }
28
29         // POST api/<DeviseController>
30         [HttpPost]
31         public void Post([FromBody] string value)
32         {
33         }
34
35         // PUT api/<DeviseController>/5
36         [HttpPut("{id}")]
37         public void Put(int id, [FromBody] string value)
38         {
39         }
40
41         // DELETE api/<DeviseController>/5
42         [HttpDelete("{id}")]
43         public void Delete(int id)
44         {
45         }
46     }
47 }
```

En entête de classe, on peut voir les annotations indiquant l'accès au contrôleur (`Route`) ainsi que le type de contrôleur (`ApiController`) :

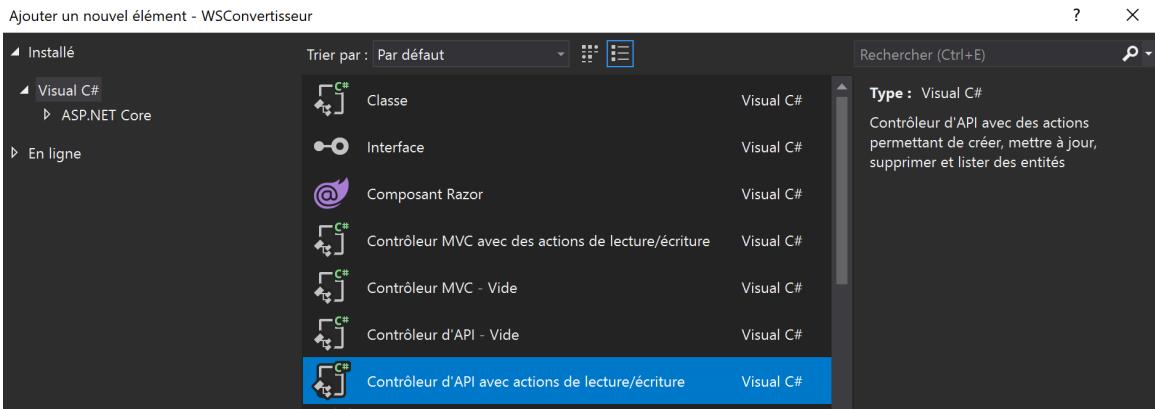
```
[Route("api/[controller]")]
[ApiController]
```

Explications :

- Le format de réponse est par défaut Json. On peut imposer ce format ou un autre en utilisant l'attribut `[Produces]` (par ex. `[Produces("application/json")]`). Par défaut, la Web API .NET Core prend en charge seulement Json : même si un autre format est spécifié dans le header `Accept` lors de l'appel, le résultat retourné est donc toujours au format JSON. Vous verrez comment ajouter d'autres formateurs, comme XML, en fin de TP.
- `[ApiController]`. Cette annotation (attribut) n'est pas obligatoire mais permet d'activer les consignes strictes liées aux comportements spécifiques aux API : exigence du routage d'attribut, réponses HTTP 400 automatiques, utilisation des paramètres de source de liaison (`FromBody`, `FromRoute`, etc.), gestion des codes d'erreurs (404, etc.), etc.
- `[Route("api/[controller]")]` peut être remplacé par `[Route("api/Devise")]`.
- Un contrôleur d'API (i.e. contrôleur sans vue) hérite (normalement toujours, cf. remarque suivante) de la classe `ControllerBase` (`Microsoft.AspNetCore.Mvc.ControllerBase`) : <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase?view=aspnetcore-5.0>

Remarques :

- On peut également ajouter un contrôleur d'API en utilisant Ajouter > Nouvel élément puis choisir « Contrôleur d'API avec actions de lecture/écriture » :



- **ATTENTION à ne pas choisir « Contrôleur MVC... » utilisé pour créer des applications Web ASP.NET avec vue razor. Dans ce cas, la classe utilisée sera Microsoft.AspNetCore.Mvc.Controller et non ControllerBase.**

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()

Renommer la 1^{ère} méthode en `GetAll`.

En entête de méthode, on peut voir `[HttpGet]`. Cet attribut désigne une méthode qui répond à une requête HTTP GET. Le chemin d'URL (route) pour chaque méthode est construit ainsi :

- Prend le modèle de chaîne dans l'attribut `Route` du contrôleur (`[Route("api/Devise")]`).
- Si l'attribut `[HttpGet]` a un modèle de route (comme `[HttpGet("/produits")]`), il sera ajouté au chemin. Cet exemple n'utilise pas de modèle. Plus de détails ici :

<https://docs.microsoft.com/fr-fr/aspnet/core/mvc/controllers/routing?view=aspnetcore-5.0#attribute-routing-with-httpverb-attributes>

Modifier la méthode (`public IEnumerable<string> GetAll()`) afin qu'elle retourne 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` :
 - <https://www.codeproject.com/Articles/832189>List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v>
 - <http://stackoverflow.com/questions/4455428/difference-between-iqueryable-icollection-ilist-idictionary-interface>

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

Exécuter l'application. Vous pourrez utiliser swagger ou modifier la configuration de `launchSettings.json` pour appeler le contrôleur `Devise` par défaut (`"launchUrl": "api/devise"`).

```
[{"id":1,"nomDevise":"Dollar","taux":1.08}, {"id":2,"nomDevise":"Franc Suisse","taux":1.07}, {"id":3,"nomDevise":"Yen","taux":120}]
```

Noter le port utilisé. Ici : 44360 pour IIS ; 5001 pour le hosting environment en https / 5000 pour http.

Ne pas fermer le navigateur.

Nous allons tester l'appel au WS (hébergé localement) dans l'outil « Postman ».
Lancer Postman disponible dans les applications Windows de la machine virtuelle (sinon l'installer : <https://www.getpostman.com>).

Saisir l'URL locale du WS, choisir la méthode « GET ». Il n'est pas nécessaire d'ajouter un header Accept car le format de retour est obligatoirement Json (même si vous spécifiez un autre format).
Cliquer sur « Send ».

Postman pose parfois des problèmes pour tester des WS https.

Si vous avez une version récente de Postman, il suffit de cliquer sur « Disable SSL Verification » lors de l'exécution :

SSL Error: Unable to verify the first certificate | [Disable SSL Verification](#)

Sinon :

- Pour résoudre le problème sur une version ancienne de Postman : <https://blog.getpostman.com/2019/07/17/self-signed-ssl-certificate-troubleshooting/>
- Ou mettre en commentaire app.UseHttpsRedirection() puis tester l'appel sur le port 5000 (en http).
- Ou utiliser swagger.

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

The screenshot shows the Postman interface with a successful API call. The URL is https://localhost:5001/api/devise. The response status is 200 OK. The response body is a JSON array of currency objects:

```
[{"id": 1, "nomDevise": "Dollar", "taux": 1.08}, {"id": 2, "nomDevise": "Franc Suisse", "taux": 1.07}, {"id": 3, "nomDevise": "Yen", "taux": 120}]
```

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

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

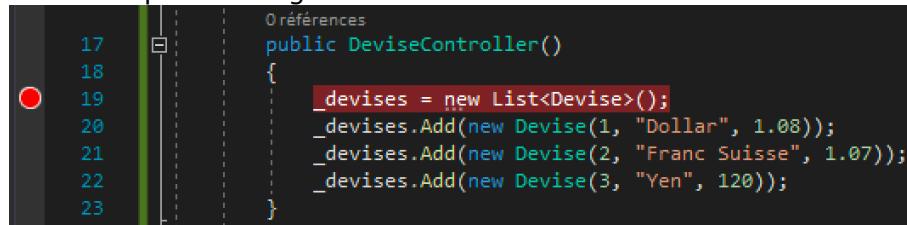
```
[{"Id": 1, "nomDevise": "Dollar", "taux": 1.08}, {"id": 2,
```

```

        "nomDevise": "Franc Suisse",
        "taux": 1.07
    },
{
    "Id": 3,
    "nomDevise": "Yen",
    "taux": 120
}
]

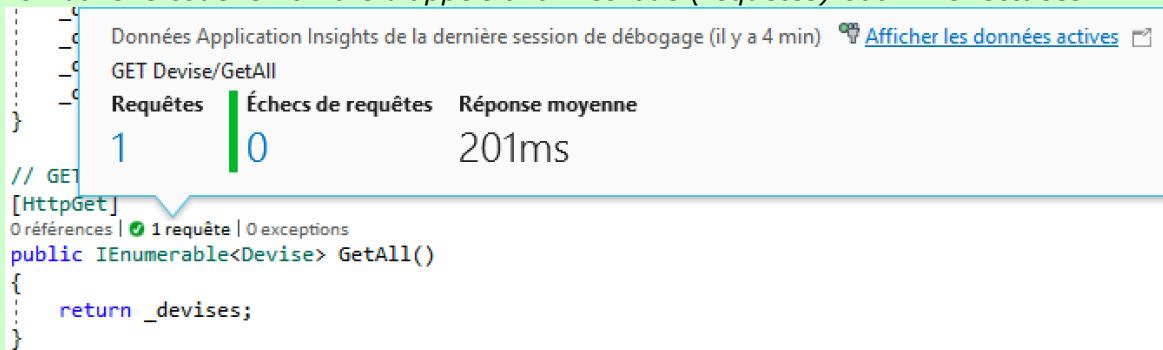
```

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



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.

On peut voir dans le code le nombre d'appels à la méthode (requêtes) GetAll effectuées.



Méthode Get(id)

Renommer la méthode en GetById.

```
[HttpGet("{id}", Name = "GetDevise")]
```

Name = "GetDevise" crée un itinéraire nommé. Les itinéraires nommés permettent à l'application de créer une liaison HTTP à l'aide du nom de l'itinéraire. Nous verrons ultérieurement à quoi sert un itinéraire nommé.

Dans cette méthode, 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> devise =
            from d in _devises
            where d.Id == id
            select d;
            return devise.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;
```

- En lambda expression :

- Devise devise = _devises.FirstOrDefault(d => d.Id == id);

- First/FirstOrDefault :

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

La méthode GetById récupérant une seule devise est maintenant fonctionnelle et se consomme de cette façon : GET /api/devise/id, par exemple : /api/devise/2

Le mode par défaut pour récupérer le paramètre est [FromRoute] : public Devise GetById(int id) ⇔ public Devise GetById([FromRoute] int id). Cela signifie que l'ID va être récupéré à partir du chemin (Uri) d'appel du WS.

Résultat :

The screenshot shows the Postman interface. At the top, the URL is set to `https://localhost:5001/api/devise/2`. Below the URL, the method is selected as `GET`. The `Params` tab is active, showing a single entry: `Key` under `KEY` and `Value` under `Value`. In the `Body` section, the response is displayed as a JSON object:

```

1
2   "id": 2,
3   "nomDevise": "Franc Suisse",
4   "taux": 1.07
5

```

The status bar at the bottom indicates a `200 OK` response with `257 ms` and `195 B`.

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

Tester avec un ID inexistant. On récupère une erreur `204 No Content`, ce qui n'est pas un retour standard. Ce devrait être une erreur http `404 Not Found`.

https://localhost:5001/api/devise/20

GET https://localhost:5001/api/devise/20 Send

Params Authorization Headers (6) Body Pre-request Script Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (3) Test Results 204 No Content 123 ms 100 B Save Response

Pretty Raw Preview Visualize Text

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

```
[HttpGet("{id}", Name = "GetDevise")]
public IActionResult GetById(int id)
{
    Devise devise = _devises.FirstOrDefault(d => d.Id == id);
    if (devise == null)
    {
        return NotFound();
    }
    return Ok(devise);
}
```

- L'interface `IActionResult` définit une commande qui crée de manière asynchrone un message http de réponse contenant une représentation (Json) de la devise renvoyée ou un message d'erreur (*404 Not Found*).
Le type de retour `IActionResult` est utilisé quand plusieurs types de retour `ActionResult` sont possibles dans une action. Les types `ActionResult` représentent différents codes d'état HTTP. Certains types de retour courants relevant de cette catégorie sont `BadRequestResult` (400), `NotFoundResult` (404) et `OkObjectResult`(200).
 - Si un objet de type `Devise` a été trouvé, c'est la méthode `Ok(T)` de la classe `Controller` qui est appelée. Le message http finalement retourné contient le statut `StatusCodes.Status200OK` (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`.
La méthode `Ok` est appelée comme forme abrégée de `return new OkObjectResult(devise);`.
Classe `Controller` :
<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controller?view=aspnetcore-5.0>

<code>Ok(Object)</code>	Creates an <code>OkObjectResult</code> object that produces an <code>Status200OK</code> response.
-------------------------	---
 - Si aucun objet `Devise` n'a été trouvé, c'est la méthode `NotFound()` de la classe `Controller` qui est appelée. Elle générera une erreur 404 (`StatusCodes.Status404NotFound`).

<code>NotFound()</code>	Creates an <code>NotFoundResult</code> that produces a <code>Status404NotFound</code> response.
-------------------------	---

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

The screenshot shows the Postman interface. At the top, the URL is set to `https://localhost:5001/api/devise/20`. Below the URL, there's a dropdown menu showing "GET". The "Params" tab is selected. In the "Query Params" table, there is one row with "Key" and "Value" columns both empty. The "Body" tab is selected. The response section shows a status of "404 Not Found" with a red box around it. The response body is a JSON object:

```

1
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
3   "title": "Not Found",
4   "status": 404,
5   "traceId": "00-7bcbbcf8cef04847b7321b6be69773b0-9677eb8f7f4fbb43-00"
6

```

Méthode Post

```
[HttpPost]
public IActionResult Post([FromBody]Devise devise)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
    _devises.Add(devise);
    return CreatedAtRoute("GetDevise", new { id = devise.Id }, devise);
}
```

- Prend un objet `Devise` en paramètre et retourne un `IActionResult` contenant une représentation Json de la devise créée (choix de développement) ou une erreur 400 Bad Request en cas d'échec de la validation du modèle.
- Le code précédent est une méthode HTTP POST, comme indiqué par l'attribut `[HttpPost]`. L'attribut `[FromBody]` indique qu'il faut obtenir la valeur de la devise à partir du corps (body) de la requête HTTP. Autres attributs possibles en fonction des méthodes : `FromQuery`, `FromRoute`, `FromHeader`, `FromForm`
<https://www.dotnetcurry.com/aspnet/1390/aspnet-core-web-api-attributes>,
<https://docs.microsoft.com/fr-fr/aspnet/core/mvc/models/model-binding?view=aspnetcore-5.0>

A partir de .NET Core 2.1, il n'est pas nécessaire de préciser `[FromBody]` puisqu'il s'agit du mode par défaut pour le POST et le PUT.

- `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)
{
    return BadRequest(ModelState);
}

```

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

```

if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}

_dевise.Add(_devise);
return CreatedAtAction("GetDevise", new { id = _devise.Id });

```

Affichage : une erreur **400 Bad Request** est générée par la méthode `BadRequest()` de la classe `Controller` avec le `ModelState` en paramètre.

https://localhost:5001/api/devise

POST https://localhost:5001/api/devise

Body (JSON)

```

1 {
2     "id": 4,
3     "nomDevise": "Rouble",
4     "taux": "AAA"
5 }

```

Body Cookies Headers (4) Test Results 400 Bad Request 182 ms 471 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2     "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3     "title": "One or more validation errors occurred.",
4     "status": 400,
5     "traceId": "00-82d7b1f51a197248b1395beb9cb8ced6-bb563710f292d149-00",
6     "errors": {
7         "$.taux": [
8             "'A' is an invalid start of a value. Path: $.taux | LineNumber: 3 | BytePositionInLine: 10."
9         ]
10    }
}

```

De même, si vous passez le fichier JSON Suivant, une erreur **400 Bad Request** sera levée.

```

{
    "id": 4,
    "nomDevise": "Rouble",
    "taux": "AAA"
}

```

The screenshot shows a POST request to <https://localhost:5001/api/devise>. The request body is:

```

1 {
2   "id": 4,
3   "nomDevise": "Rouble",
4   "taux": "AAA"
5 }

```

The response is a 400 Bad Request with the following validation errors:

```

2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3   "title": "One or more validation errors occurred.",
4   "status": 400,
5   "traceId": "00-8dee111ebdeab34aa062daf4a66ddfae-3b23d5e031920943-00",
6   "errors": {
7     "$.taux": [
8       "The JSON value could not be converted to System.Double. Path: $.taux | LineNumber: 3 |
9         BytePositionInLine: 15."
10    ]
11  }

```

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

`CreatedAtRoute(String, Object, Object)`

Creates a [CreatedAtRouteResult](#) object that produces a [Status201Created](#) response.

```
C#
[Microsoft.AspNetCore.Mvc.NonAction]
public virtual Microsoft.AspNetCore.Mvc.CreatedAtRouteResult CreatedAtRoute (string
routeName, object routeValues, object value);
```

Parameters

routeName `String`

The name of the route to use for generating the URL.

routeValues `Object`

The route data to use for generating the URL.

value `Object`

The content value to format in the entity body.

Returns

[CreatedAtRouteResult](#)

The created [CreatedAtRouteResult](#) for the response.

- Retourne une réponse 201. HTTP 201 est la réponse standard d'une méthode HTTP POST qui crée une ressource sur le serveur.

- Ajoute un en-tête Location à la réponse. L'en-tête Location spécifie l'URI de l'élément d'action qui vient d'être créé.
- Utilise l'itinéraire nommé « GetDevise » pour créer l'URL. Il est défini dans la méthode GetById : [HttpGet("{id}", Name = "GetDevise")].
- Il est aussi possible d'appeler directement l'action (la méthode) de la façon suivante : CreatedAtAction("GetById", new { id = devise.id }, devise)

CreatedAtAction(String, Object, Object)

Creates a [CreatedAtActionResult](#) object that produces a [Status201Created](#) response.

```
C# Copy
[Microsoft.AspNetCore.Mvc.NonAction]
public virtual Microsoft.AspNetCore.Mvc.CreatedActionResult CreatedAtAction (string
actionName, object routeValues, object value);
```

Parameters

actionName `String`

The name of the action to use for generating the URL.

routeValues `Object`

The route data to use for generating the URL.

value `Object`

The content value to format in the entity body.

Returns

[CreatedAtActionResult](#)

The created [CreatedAtActionResult](#) for the response.

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

Test d'insertion d'une devise valide :

The screenshot shows the Postman interface with the following details:

- Method:** POST
- URL:** https://localhost:5001/api/devise
- Body (JSON):**

```

1
2   "id": 4,
3   "nomDevise": "Rouble",
4   "taux": 1510.5
5

```
- Response Status:** 201 Created
- Response Body:**

```

1
2   "id": 4,
3   "nomDevise": "Rouble",
4   "taux": 1510.5
5

```

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

Exécuter ensuite la méthode GET 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 Core.

Test d'insertion de données manquantes : par défaut, on peut créer une devise sans nom (ou sans ID ou sans taux).

The screenshot shows the Postman interface with the following details:

- Method: POST
- URL: http://localhost:5000/api/devise
- Body tab selected, showing JSON content:

```
1 {  
2   "id": 4,  
3   "taux": 1510.5  
4 }
```
- Response status: Status: 201 Created Time: 776ms Size: 238 B

Modifier la classe Devise ainsi :

```
[Required]  
public string NomDevise { get; set; }
```

Rajouter le namespace using System.ComponentModel.DataAnnotations;

Réexécuter le POST précédent. Vous obtiendrez l'erreur suivante :

The screenshot shows the Postman interface. At the top, the URL is set to `https://localhost:5001/api/devise`. The method is selected as `POST`. The `Body` tab is active, showing a JSON payload:

```

1 {
2   "id": 4,
3   "taux": 1510.5
4 }

```

Below the body, the status bar indicates a `400 Bad Request` with `208 ms` and `416 B` response time. The response body is displayed in JSON format:

```

1 {
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3   "title": "One or more validation errors occurred.",
4   "status": 400,
5   "traceId": "00-cefe33cdd6d925429da8c36ddfb4d154-9bdc26f21a0b6e48-00",
6   "errors": [
7     "NomDevise": [
8       "The NomDevise field is required."
9     ]
10  ]
11 }

```

On peut rajouter de nombreuses annotations au modèle, spécifiant notamment des contraintes, comme `[Phone]`, `[Url]`, `[EmailAddress]`, `[RegularExpression]`, etc. (Cf. TP n°3).

Méthode Delete

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

- Même type de retour que les méthodes précédentes.
- 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)` (On peut aussi renvoyer un `NoContent()`)

La méthode `Delete` se consomme de cette façon : `DELETE /api/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

```

public IActionResult Put(int id, [FromBody] 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 NoContent();
}

```

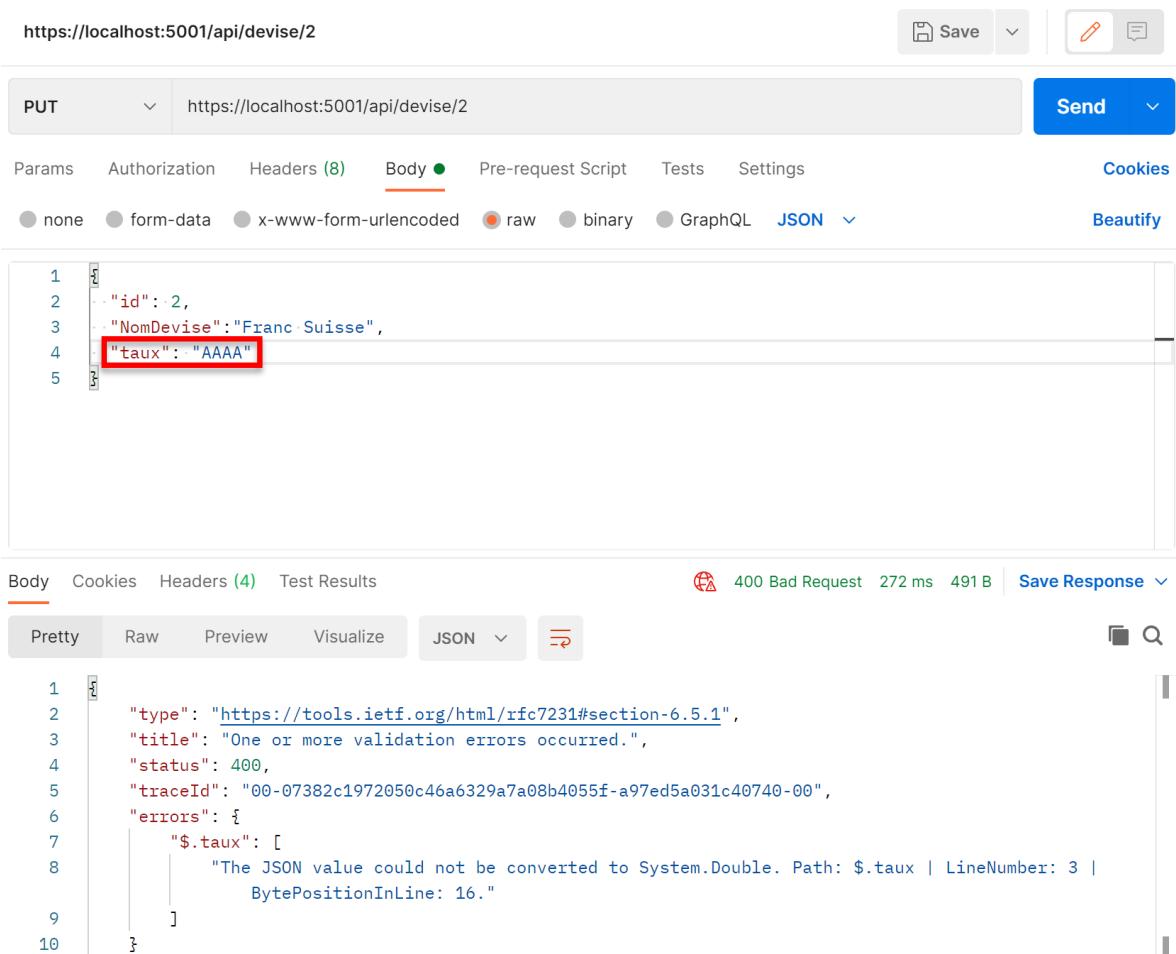
- Ici, nous avons fait le choix de ne pas retourner l'objet Devise modifié (`return Ok<devise>`), mais juste un statut de réponse http *204 No content*, sans contenu JSON.

D'après la spécification HTTP, une requête PUT nécessite que le client envoie toute l'entité mise à jour, et pas seulement les différences. Pour prendre en charge les mises à jour partielles, il faut utiliser HTTP PATCH (nous le ferons en TP3) :

<https://www.infoworld.com/article/3206264/application-development/how-to-perform-partial-updates-to-rest-web-api-resources.html>
<https://www.youtube.com/watch?v=oVzD74AJL4E>.

La méthode PUT permettant de mettre à jour une devise se consomme de cette façon : `PUT /api/devise/id` en lui passant une représentation Json.

Tests de modifications non valides :



The screenshot shows a POSTMAN interface with the following details:

- Method:** PUT
- URL:** `https://localhost:5001/api/devise/2`
- Body:** JSON (highlighted in red)
- Request Body Content:**

```

1
2   "id": 2,
3   "NomDevise":"Franc Suisse",
4   "taux": "AAAA"
5

```
- Response Status:** 400 Bad Request
- Response Headers:** 272 ms, 491 B
- Response Body (Pretty JSON):**

```

1
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3   "title": "One or more validation errors occurred.",
4   "status": 400,
5   "traceId": "00-07382c1972050c46a6329a7a08b4055f-a97ed5a031c40740-00",
6   "errors": {
7     "$.taux": [
8       "The JSON value could not be converted to System.Double. Path: $.taux | LineNumber: 3 |
9       BytePositionInLine: 16."
10    ]
}

```

https://localhost:5001/api/devise/1

PUT https://localhost:5001/api/devise/1

Params Authorization Headers (8) Body **JSON** Pre-request Script Tests Settings Cookies Beautify

```

1 {
2   "id": 2,
3   "NomDevise": "Franc Suisse",
4   "taux": 1.8
5 }
```

Body Cookies Headers (4) Test Results 400 Bad Request 157 ms 328 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
3   "title": "Bad Request",
4   "status": 400,
5   "traceId": "00-0172c417d3000743a6002b473dc723d2-0a2a6fa854d4d447-00"
6 }
```

https://localhost:5001/api/devise/50

PUT https://localhost:5001/api/devise/50

Params Authorization Headers (8) Body **JSON** Pre-request Script Tests Settings Cookies Beautify

```

1 {
2   "id": 50,
3   "NomDevise": "Franc Suisse",
4   "taux": 1.8
5 }
```

Body Cookies Headers (4) Test Results 404 Not Found 197 ms 324 B Save Response

Pretty Raw Preview Visualize JSON

```

1 {
2   "type": "https://tools.ietf.org/html/rfc7231#section-6.5.4",
3   "title": "Not Found",
4   "status": 404,
5   "traceId": "00-3134a2df76e9b349914b31be8162e995-87410f51eab62647-00"
6 }
```

Modification valide :

Tester en mettant un point d'arrêt ou un espion pour vérifier que la devise est bien mise à jour dans la liste.

The screenshot shows the Postman interface. At the top, the URL is set to `https://localhost:5001/api/devise/2`. The method is selected as `PUT`. The `Body` tab is active, showing a JSON payload:

```

1 {
2   "id": 2,
3   "NomDevise": "Franc Suisse",
4   "taux": 1.8
5

```

Below the body, the response status is `204 No Content`, with `102 ms` and `81 B` response times. The `Body` tab is also active here.

1.4. Documentation de l'API

Une API (OpenAPI) doit toujours fournir une documentation afin que les développeurs sachent comment la consommer.

La documentation doit contenir la liste des actions disponibles (URL, méthode), leurs paramètres et leur réponse (code de statut http, corps).

Nous allons réaliser la documentation à partir du code réalisé et de commentaires XML. Celle-ci sera visualisable dans Swagger.

1. Ajouter des commentaires XML (documentation) en entête de chaque action du contrôleur.

Commentaires XML :

- o `<summary></summary>` : Description de l'action.
- o `<remarks></remarks>` : Information supplémentaire, exemple.
- o `<returns><returns>` : Description de la valeur de retour
- o `<param name="id"></param>` : Description d'un paramètre d'entrée
- o `<response code="201"></response>` : Description de la valeur de retour quand status code = 201.

Exemple :

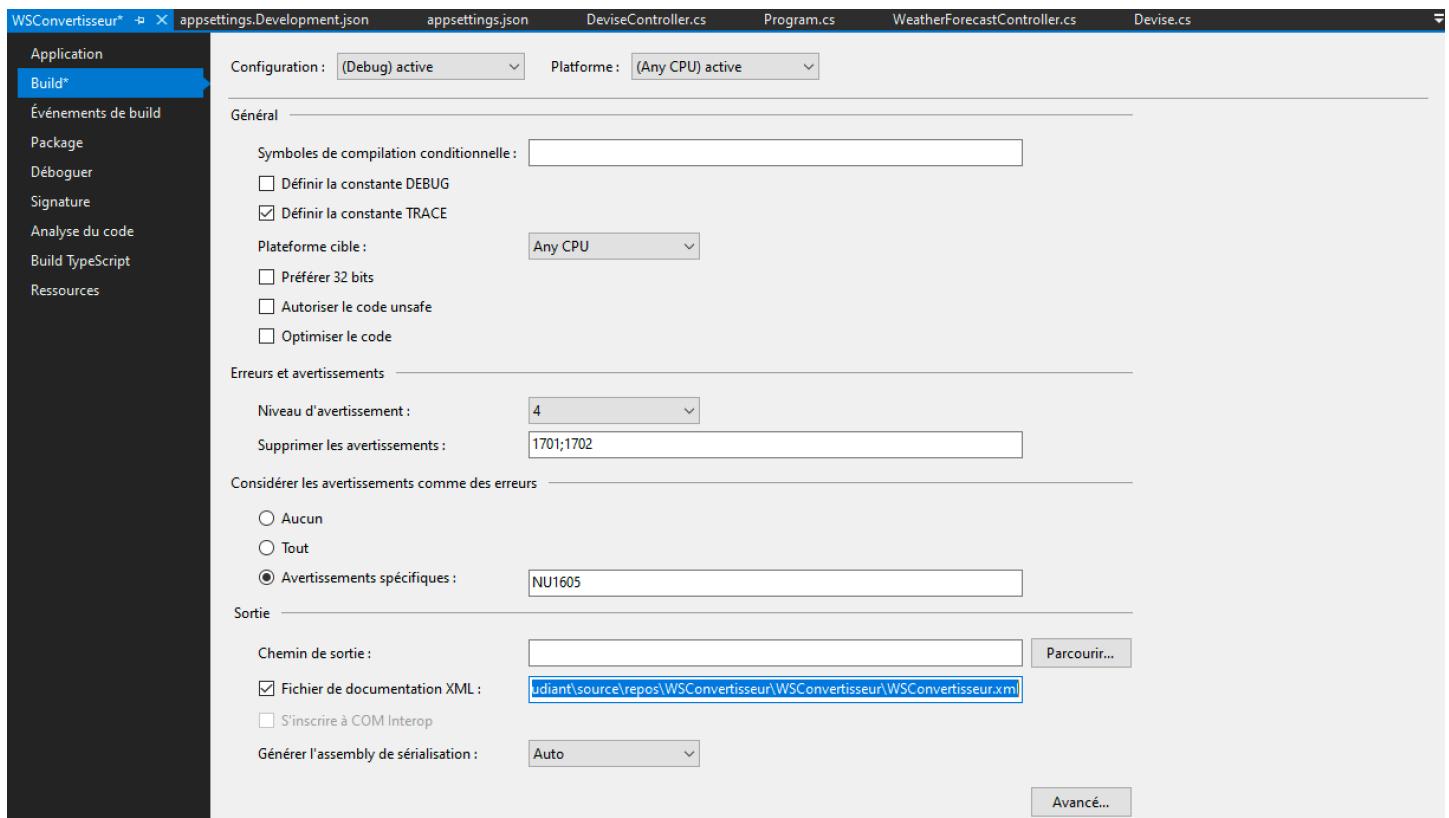
```

/// <summary>
/// Get a single currency.
/// </summary>
/// <returns>Http response</returns>
/// <param name="id">The id of the currency</param>
/// <response code="200">When the currency id is found</response>
/// <response code="404">When the currency id is not found</response>
// GET: api/Devise/5
[HttpGet("{id}", Name = "GetDevise")]
public IActionResult GetById(int id)

```

Vous pouvez également commenter la classe *Model*.

2. Dans les propriétés du projet (bouton droit de la souris sur le nom du projet puis *Propriétés*), cocher « Fichier de documentation XML » dans l'onglet « Build ». Laisser le chemin par défaut. Sauvegarder.



Toute méthode non commentée sera maintenant soulignée en vert :

```
// POST api/<DeviseController>
[HttpPost]
0 références
public IActionResult Post([FromBody] Devise devise)
{
    if (!ModelState.IsValid)
    {
        return BadRequest();
    }
    _devises.Add(devise);
    return CreatedAtRoute("GetDevise", new { id = devise.Id }, devise);
}
```

Avertissement CS1591 : Commentaire XML manquant pour le type ou le membre visible publiquement 'DeviseController.Post(Devise)'.

Des avertissements seront également générés :

Liste d'erreurs				
	Code	Description	Projet	Fichier
CS1591	CS1591	Commentaire XML manquant pour le type ou le membre visible publiquement 'DeviseController'	WSConvertisseur	DeviseController.cs
CS1591	CS1591	Commentaire XML manquant pour le type ou le membre visible publiquement 'DeviseController.DeviseController()'	WSConvertisseur	DeviseController.cs
CS1591	CS1591	Commentaire XML manquant pour le type ou le membre visible publiquement 'DeviseController.GetAll()'	WSConvertisseur	DeviseController.cs
CS1591	CS1591	Commentaire XML manquant pour le type ou le membre visible publiquement 'DeviseController.Post(Devise)'	WSConvertisseur	DeviseController.cs
CS1591	CS1591	Commentaire XML manquant pour le type ou le membre visible publiquement 'DeviseController.Put(int Devise)'.	WSConvertisseur	DeviseController.cs

3. Indiquer à Swagger où récupérer la documentation XML générée (fichier startup.cs) :

```
// Configure Swagger to use the xml documentation file
var xmlFile = Path.ChangeExtension(typeof(Startup).Assembly.Location, ".xml");
c.IncludeXmlComments(xmlFile);

// This method gets called by the runtime. Use this method to add services to the container.
0 références
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "API convertisseur", Version = "v1" });

        // Configure Swagger to use the xml documentation file
        var xmlFile = Path.ChangeExtension(typeof(Startup).Assembly.Location, ".xml");
        c.IncludeXmlComments(xmlFile);
    });
}
```

4. Ajouter des attributs [ProducesResponseType] au niveau des actions indiquant les types de retour (en remplacement ou en plus de la balise XML <response>)

[ProducesResponseType] génère des détails descriptifs de la réponse pour les pages d'aide de l'API. Il indique les types connus et les codes d'état HTTP que l'action doit retourner.

Exemples :

- o Pour GetAll : [ProducesResponseType(200)] ou mieux :
[ProducesResponseType(typeof(IEnumerable<Devise>), 200)]
- o Pour GetById : [ProducesResponseType(200)] ou mieux :
[ProducesResponseType(typeof(Devise), 200)]
[ProducesResponseType(404)]
- o Pour Post : [ProducesResponseType(201)] ou mieux
[ProducesResponseType(typeof(Devise), 201)]
[ProducesResponseType(400)]
- o Etc.

Exemple :

```
/// <summary>
/// Get a single currency.
/// </summary>
/// <returns>Http response</returns>
/// <param name="id">The id of the currency</param>
/// <response code="200">When the currency id is found</response>
/// <response code="404">When the currency id is not found</response>
// GET: api/Devise/5
[HttpGet("{id}", Name = "GetDevise")]
[ProducesResponseType(typeof(Devise), 200)]
[ProducesResponseType(404)]
public IActionResult GetById(int id)
```

5. Exécuter l'API. Pour avoir accès à la documentation Swagger : <http://<host>:<port>/swagger/>

The screenshot shows the Swagger UI interface for the 'WSConvertisseur v1' definition. At the top, there's a navigation bar with the Swagger logo and the title 'Select a definition' set to 'WSConvertisseur v1'. Below the title, the page header reads 'API convertisseur v1 OAS3' with a link to '/swagger/v1/swagger.json'. The main content area is divided into sections by category. The first section, 'Devise', contains five operations: a blue 'GET /api/Devise' button, a green 'POST /api/Devise' button, a blue 'GET /api/Devise/{id}' button with a description 'Get a single currency.', an orange 'PUT /api/Devise/{id}' button, and a red 'DELETE /api/Devise/{id}' button. The second section, 'WeatherForecast', has a single entry with a right-pointing arrow. At the bottom, under 'Schemas', there's a detailed description of the 'Devise' schema:

```
Devise ▶ {
    id          integer($int32)
    nomDevise*  string
    taux        number($double)
}
```

GET :

Devises

GET /api/Devises

POST /api/Devises

GET /api/Devises/{id} Get a single currency.

Parameters

Name Description

id * required integer(\$int32) The id of the currency
(path)

id - The id of the currency

Responses

Code	Description	Links
200	When the currency id is found	No links
	Media type	
	text/plain	
	Controls Accept header.	
	Example Value Schema	
	{ "id": 0, "nomDevises": "string", "taux": 0 }	
404	When the currency id is not found	No links
	Media type	
	text/plain	

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

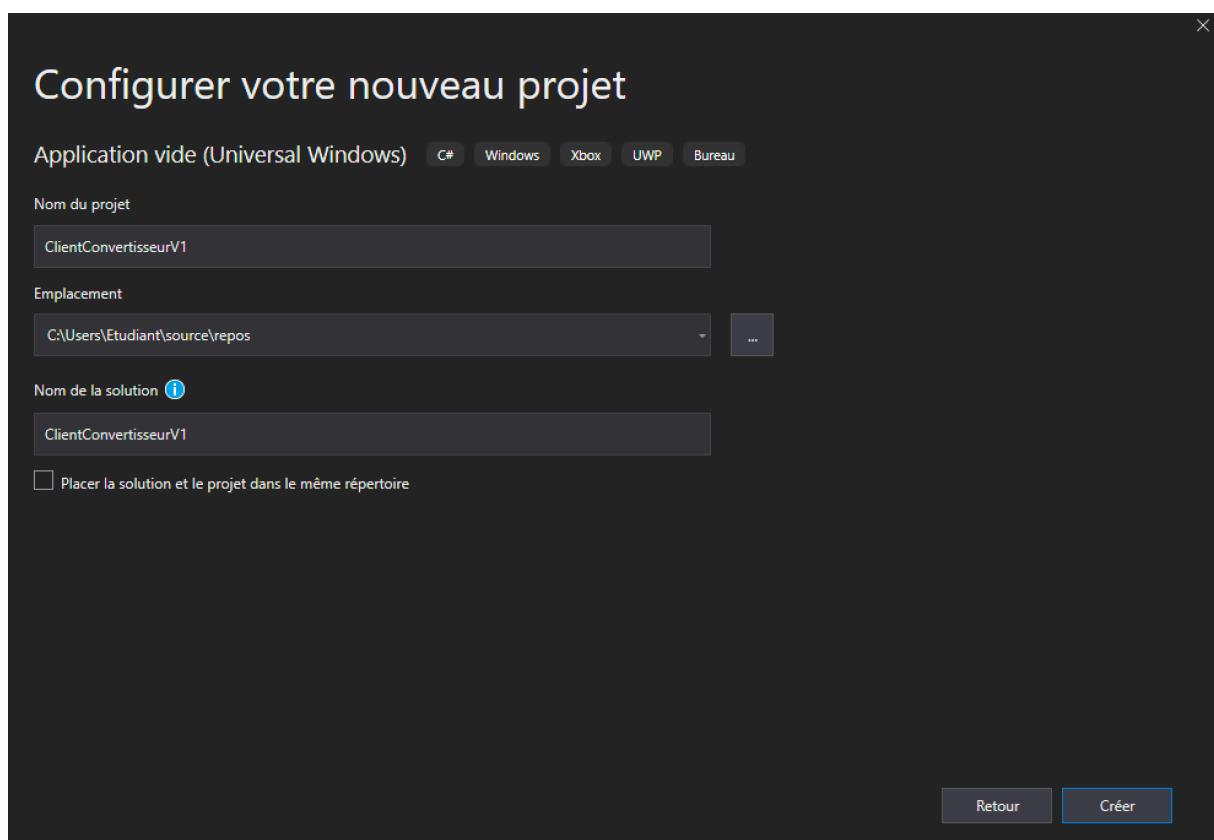
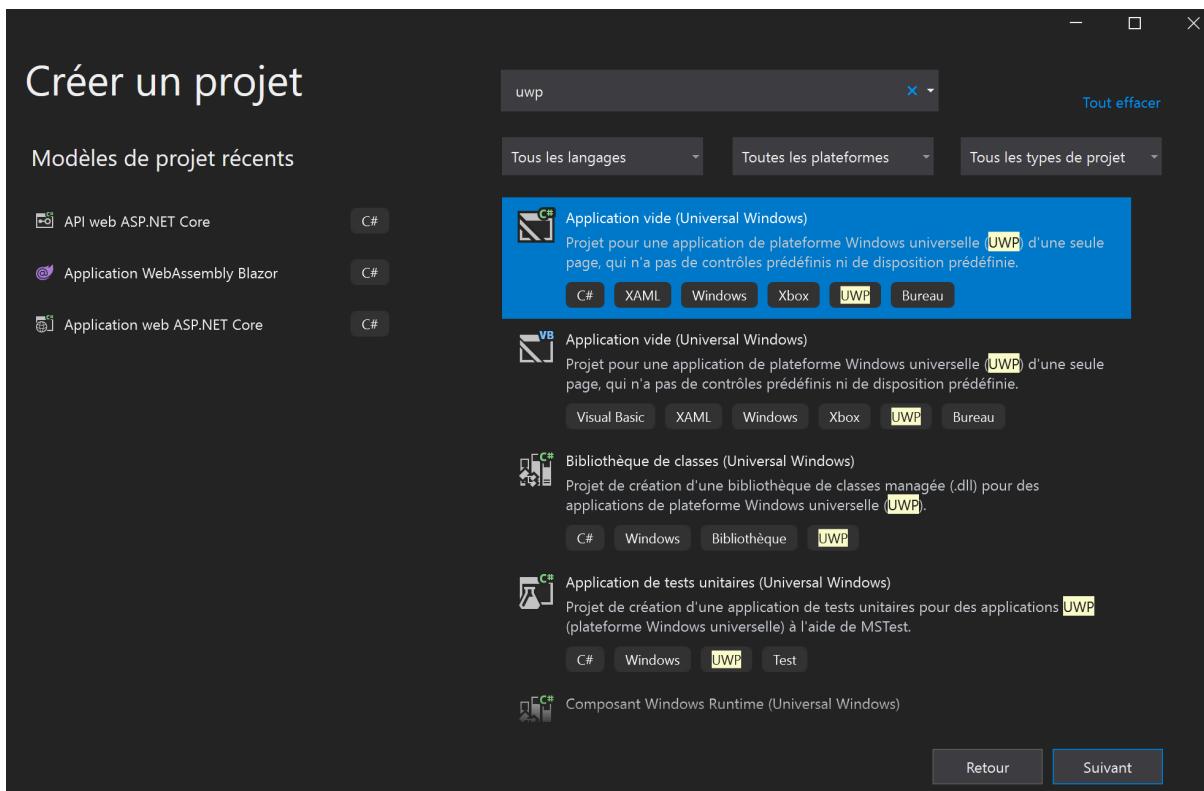
Une application UWP doit être développée sous Windows.

2.1. Client V1 (version simple sans MVVM)

Si vous connaissez le MVVM, vous pouvez directement coder le client en V2 (section 2.2).

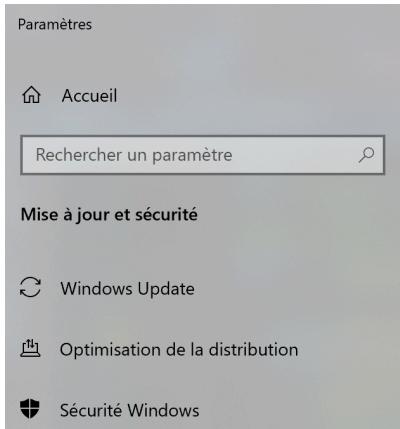
Lancer une nouvelle instance de Visual Studio.

Créer un projet « Application vide (Windows universel) » dans une nouvelle solution. Vous pourrez le nommer ClientConvertisseurV1.



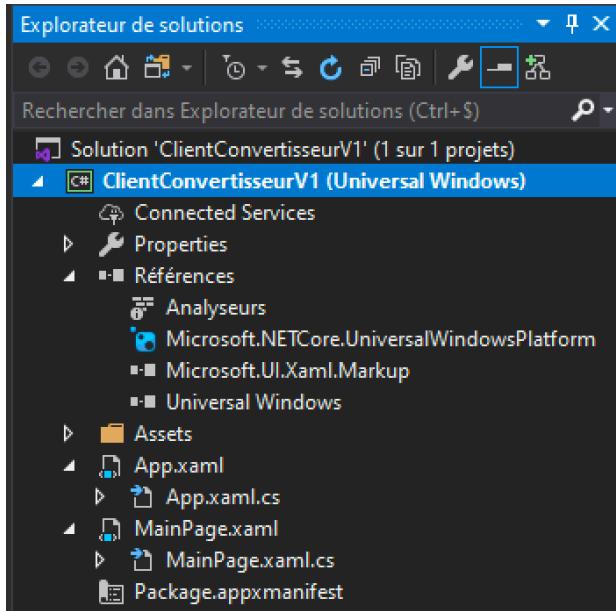
Valider le message indiquant les versions d'OS Windows 10 cibles.

Remarque : si cela n'est pas fait, vous devrez activer le mode développeur de Windows :



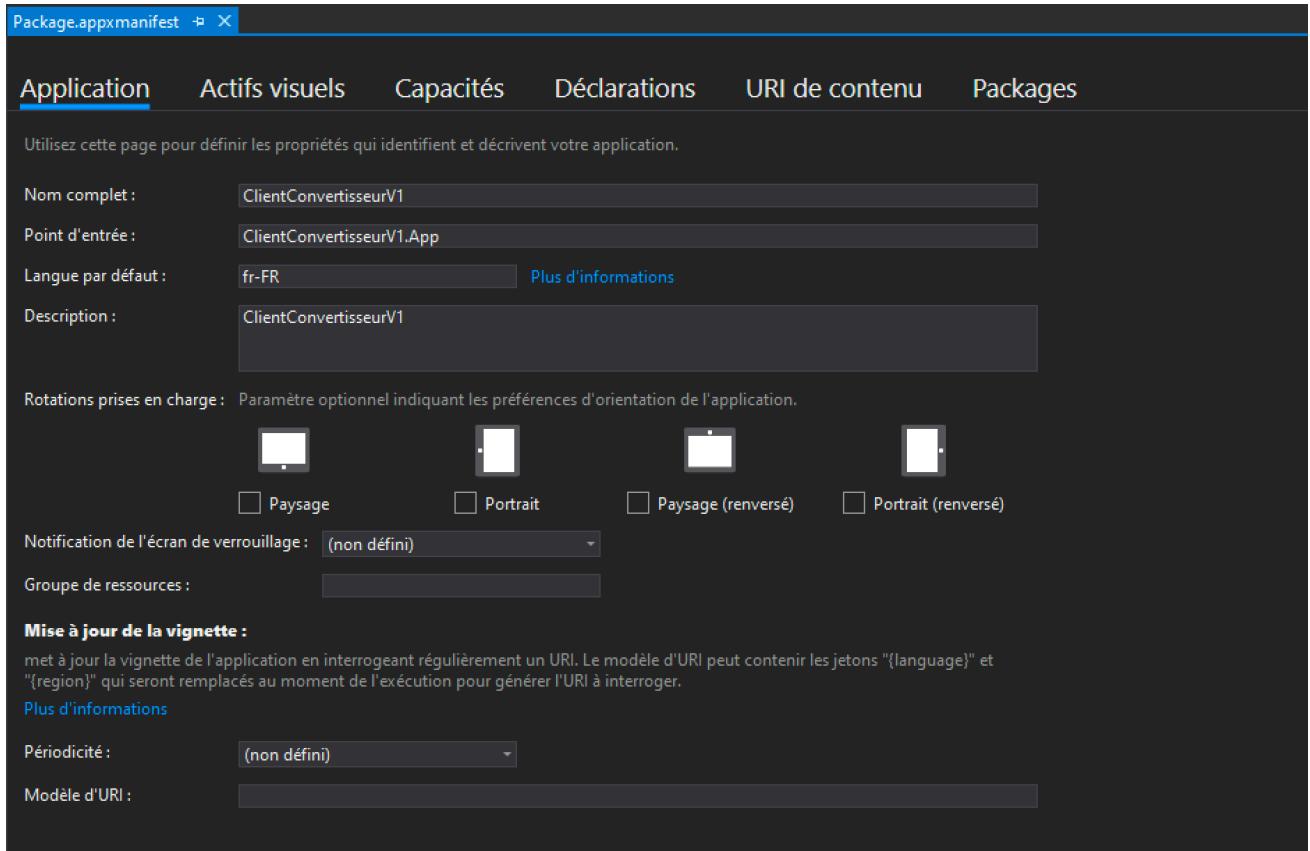
Le modèle *Application vide* crée une application du Windows Store vide qui se compile et s'exécute, mais qui ne contient aucun contrôle d'interface utilisateur ni aucune donnée. Vous ajouterez des contrôles et des données à l'application par la suite.

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. **Regarder l'ensemble des onglets.**



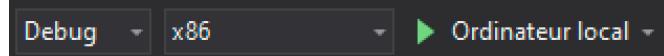
Remarque : La première page à s'afficher lorsqu'on démarre une application est la page MainPage.xaml. Ceci est modifiable dans le fichier App.xaml.cs :

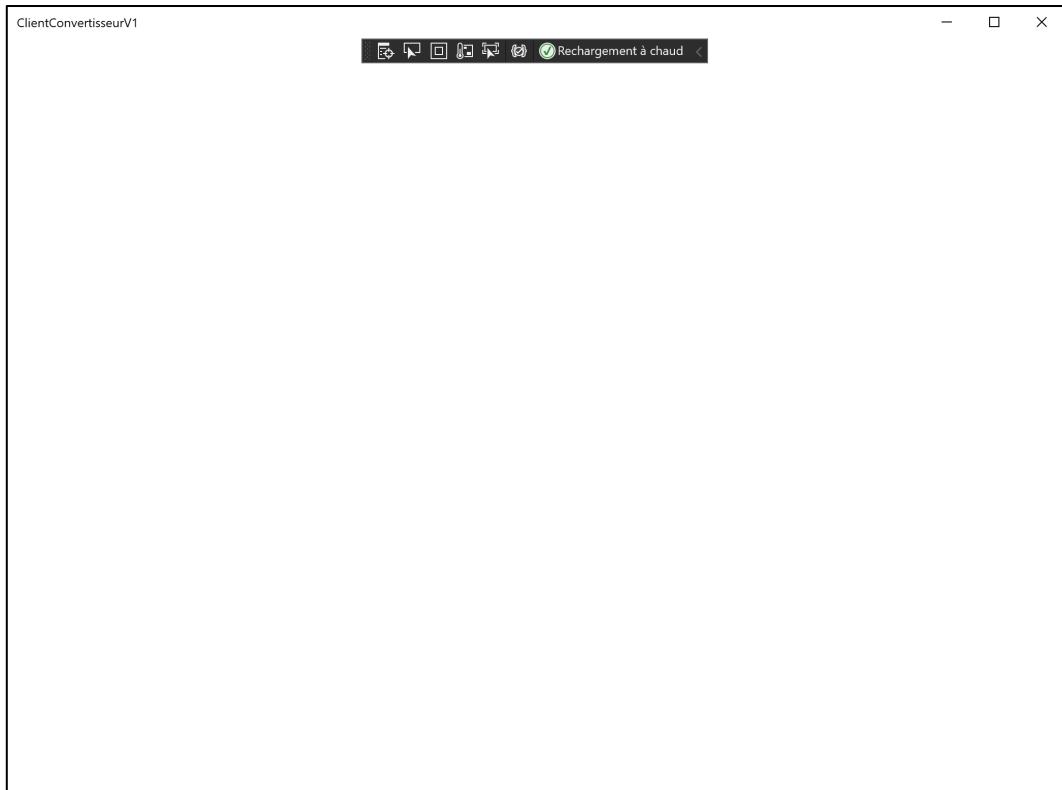
- Un ensemble d'images de logos de petit et grand format à afficher (dans le dossier Assets).
- Les fichiers XAML et de code de l'application (App.xaml et App.xaml.cs).
- Une page de démarrage (MainPage.xaml) et un fichier de code associé (MainPage.xaml.cs) qui s'exécute au démarrage de votre application.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

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



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, en partie, cela par la suite...

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).

```
<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

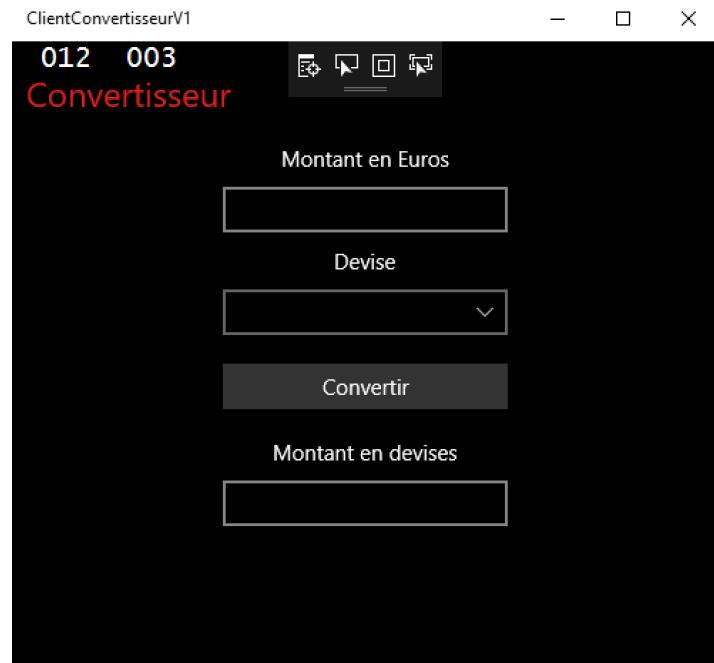
MainPage.xaml.cs est la page code-behind de MainPage.xaml. C'est ici que vous ajoutez la logique de votre application et les gestionnaires d'événements. Le modèle « Application vide (Windows Universel) **Visual C#** » comporte deux méthodes dans lesquelles vous pouvez enregistrer et charger l'état de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

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

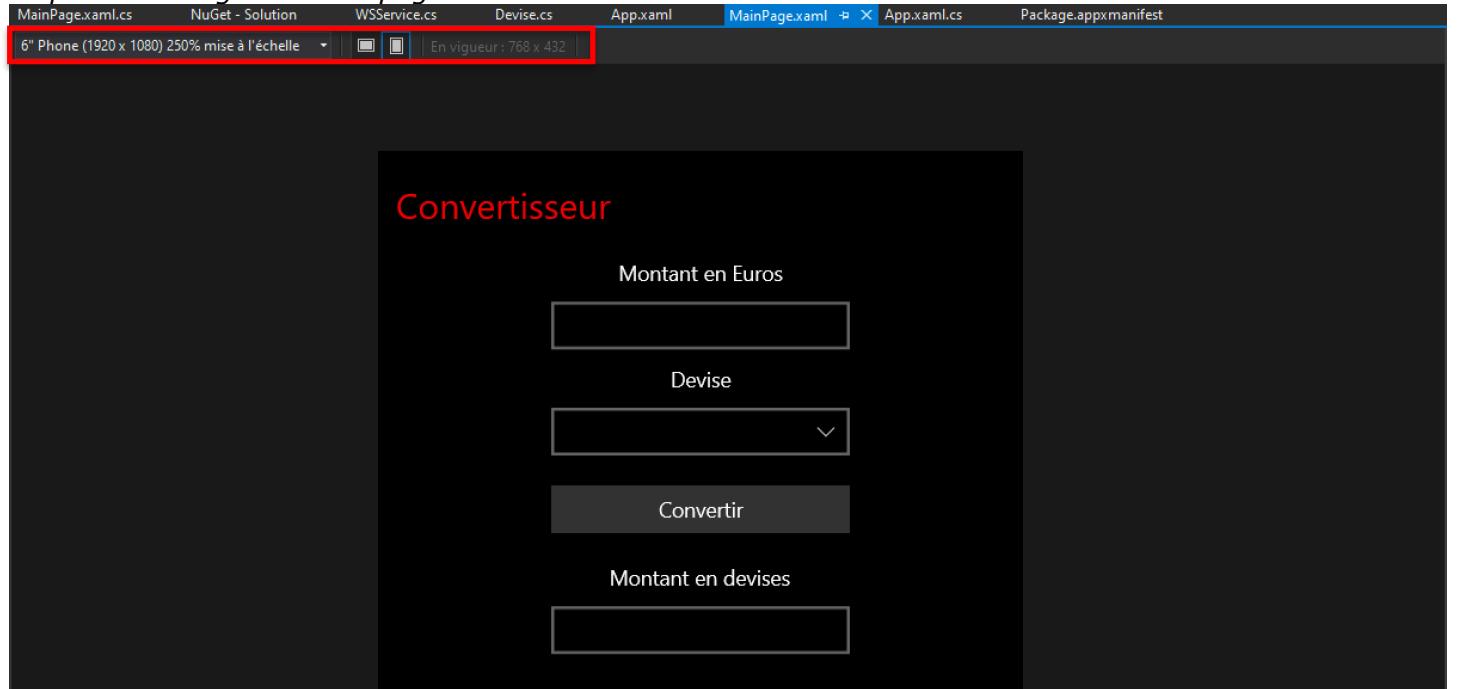
- Remplacer la grille par un RelativePanel
- Dans le panel précédent, ajouter :
 - o 3 TextBlock
 - o 1 ComboBox
 - o 2 TextBox dont un en lecture seule.
 - o 1 Button

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

Nous avons défini des marges (attribut Margin) pour placer les différents contrôles.



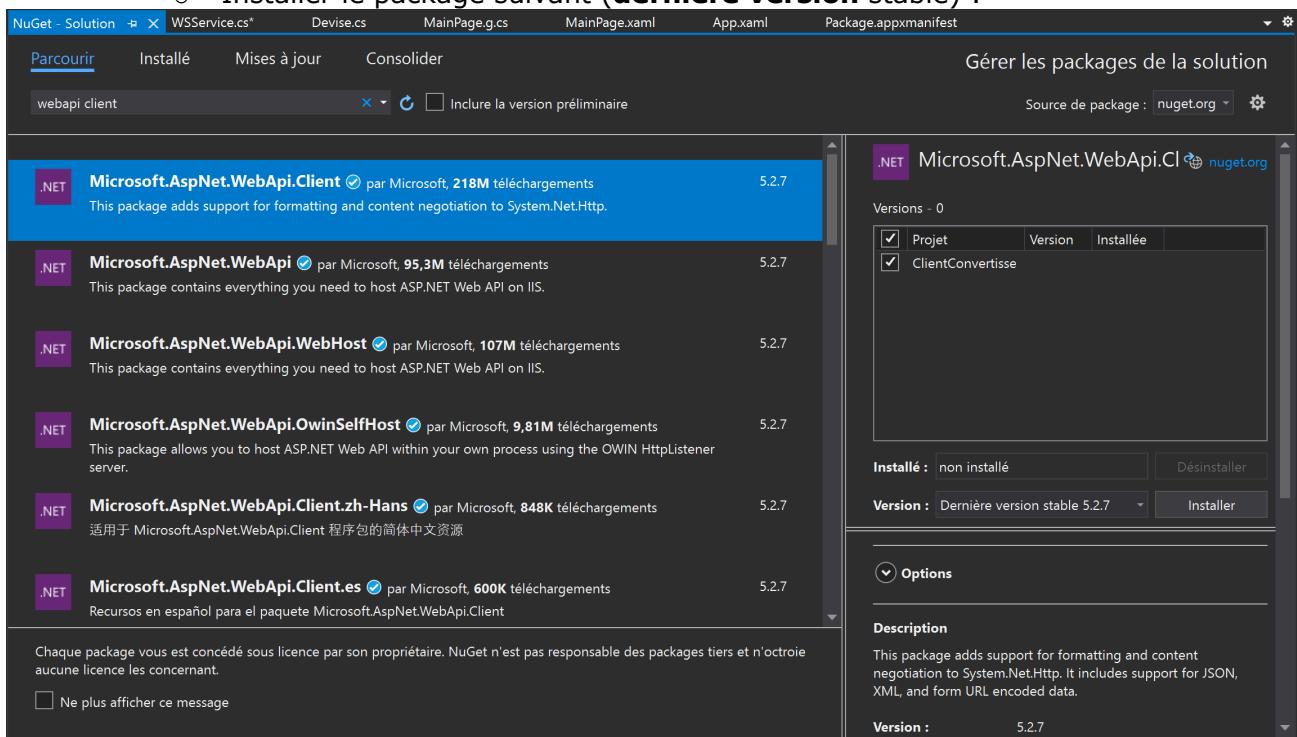
Remarque : Vous pourrez configurer différentes résolutions et/ou modes d'affichage pour tester le responsive design de votre page.



Codage de l'alimentation de la ComboBox

- Créer un dossier `Model` et y ajouter la classe `Devise` correspondant à celle du WS (seules les properties sans les annotations 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 asynchrone `public async Task<List<Devise>> getAllDevisesAsync()` retournant la liste des devises.
 - Plus de détails sur `async/await` et les `tasks` `asynchrones` : <http://fdorin.developpez.com/tutoriels/csharp/threadpool/part3/>
 - Utiliser la classe `httpClient` :
 - Exemple de code ici (l'initialisation de l'`httpClient` est à faire dans le constructeur de la classe et non dans une méthode `RunAsync`) :
 <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 (**dernière version stable**) :



- L'appel du WS est : <https://localhost:PORT/api/devise>. Donc l'Uri sera <https://localhost:PORT/api/> (ne pas oublier le / en fin d'URI). Exemple : <https://localhost:5001/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()
{
    this.InitializeComponent();
    ActionGetData();
}

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

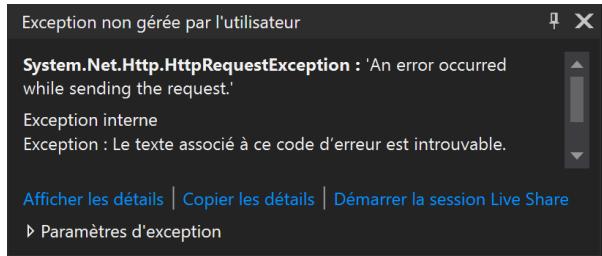
Remarque : A la place du binding XAML, on peut bien sûr écrire le code suivant dans le code-behind :

```
this.cbxDevise.ItemsSource = devises;
this.cbxDevise.SelectedValuePath = "Id";
this.cbxDevise.DisplayMemberPath = "NomDevise";
```

Codage du calcul

A vous de jouer...

Lancer l'application cliente (ne pas oublier d'exécuter le WS).
Vous devriez obtenir l'erreur suivante :



Après affichage du détail, l'erreur est due au certificat :

The screenshot shows the Visual Studio Watch window with the expression "\$exception" selected. The watch list displays the following information:

Nom	Valeur	Type
\$exception	["An error occurred while sending the request."]	System.Net.Http.HttpRequestException
Data	[System.Collections.ListDictionaryInternal]	System.Collections.IDictionary
HRESULT	-2147012851	int
HelpLink	null	string
InnerException	("Le texte associé à ce code d'erreur est introuvable.\r\n\r\nL'autorité de certification n'est pas valide ou correcte\r\n")	System.Exception
Message	"An error occurred while sending the request."	string
Source	"System.Net.Http"	string
StackTrace	" at System.Net.Http.HttpClientHandler.<SendAsync>d__113.MoveNext()\r\n à System.Runtime.ExceptionServices.ExceptionDispatchInfo..."	System.Reflection.MethodBase
TargetSite	(Void MoveNext())	
Membres statiques		
Membres non publics		

Modifier le manifeste de l'application cliente : cocher « Certificats utilisateurs partagés » des Capacités.

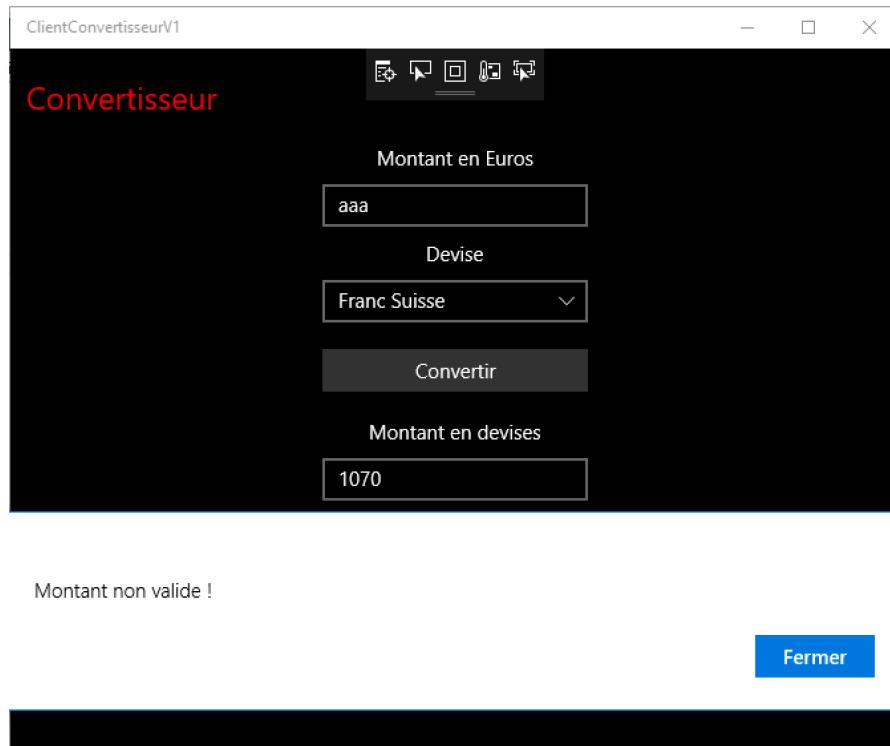
The screenshot shows the Windows App Manifest editor with the "Capacités" tab selected. The "Capacités :" section contains several checkboxes, and the "Certificats utilisateurs partagés" checkbox is checked. The "Description :" section provides a detailed explanation of what this capability does, mentioning card reader access and user certificate validation.

Résultat :

The screenshot shows the "ClientConvertisseurV1" application window titled "Convertisseur". It has a dark theme. The interface includes fields for "Montant en Euros" (1000), "Devise" (Franc Suisse), a "Convertir" button, and a result field "Montant en devises" (1070).

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...

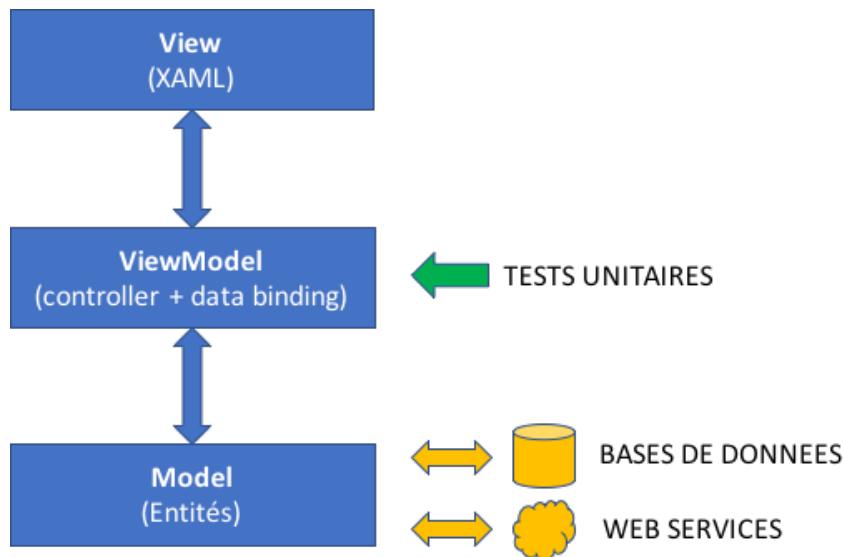


2.2. Client V2 (version avec MVVM)

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.



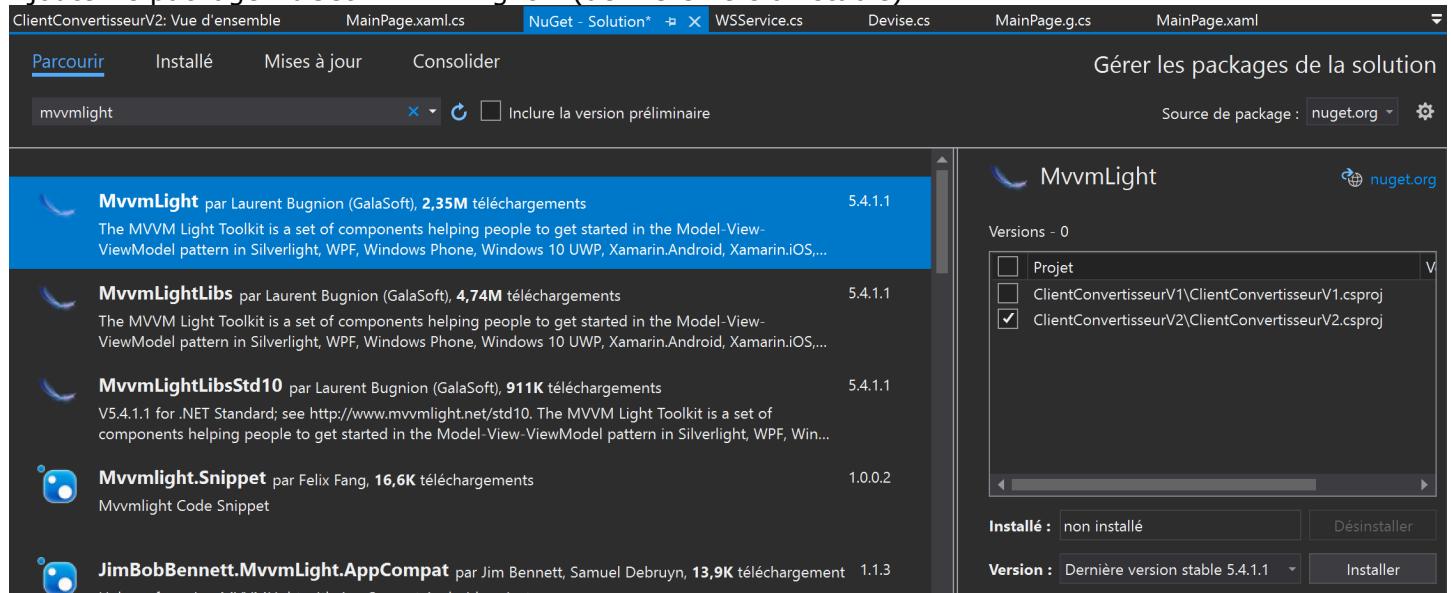
Le but de MVVM est de faire en sorte que la vue n'effectue aucun traitement : elle ne doit faire qu'afficher les données présentées par le ViewModel. C'est le ViewModel qui est chargé de réaliser les traitements et d'accéder au modèle.

Création du projet et installation des packages NuGet

Ajouter un nouveau projet nommé ClientConvertisseurV2 de type « Application vide (Windows universel) » 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 le package NuGet « MvvmLight » (dernière version stable).



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 » comme précédemment.

Couches Model et Service

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

Couches ViewModel et View

Supprimer la page MainPage.xaml.

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

```
1  using GalaSoft.MvvmLight;
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.ViewModel
9  {
10    public class ConvertisseurDeviseViewModel : ViewModelBase
11    {
12    }
13}
14
```

C'est le ViewModel de notre page ConvertisseurDevisePage.xaml (que nous allons créer). 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 créer la page ConvertisseurDevisePage.xaml.

Modifier également le fichier App.xaml.cs permettant de lancer la page ConvertisseurDevisePage.

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

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. Dans un projet MVVM, la coordination entre la vue (view) et le.viewmodel est assurée par une classe spécifique nommée 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 CommonServiceLocator;
using GalaSoft.MvvmLight.Ioc;
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(() => SimpleLoc.Default);
        SimpleLoc.Default.Register<ConvertisseurDeviseViewModel>();
    }

    /// <summary>
    /// Gets the Main property.
    /// </summary>
    public ConvertisseurDeviseViewModel ConvertisseurDevise =>
ServiceLocator.Current.GetInstance<ConvertisseurDeviseViewModel>();

}

}

```

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

Nous utilisons ici SimpleLoc. C'est l'injection de dépendance fournie par MVVM Light. Plus d'infos ici : <https://docs.microsoft.com/en-us/archive/msdn-magazine/2013/february/mvvm-locator-containers-and-mvvm>

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.

La classe ViewModelLocator a maintenant pour key Locator. Nous utiliserons cette clé par la suite.

Ajout du Binding dans la vue ConvertisseurDevisePage.xaml :

```

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

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

```

- ConvertisseurDevise fait référence à la ligne suivante du fichier ViewModelLocator et permet de lier la vue (page) au viewmodel ConvertisseurDeviseViewModel :


```
public ConvertisseurDeviseViewModel ConvertisseurDevise =>
ServiceLocator.Current.GetInstance<ConvertisseurDeviseViewModel>();
```
- Locator fait référence à la key du fichier App.xaml et renvoie au ViewModelLocator.

```

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 `ConvertisseurDeviseViewModel.cs` :

- Créer une property (utiliser le snippet `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`.

- <https://docs.microsoft.com/fr-fr/dotnet/api/system.collections.objectmodel.observablecollection-1?view=netcore-5.0>
- <https://channel9.msdn.com/Series/Windows-Phone-8-1-Development-for-Absolute-Beginners/Part-18-Understanding-MVVM-ObservableCollection-T-and-INotifyPropertyChanged>

- Créer le constructeur et ajouter la méthode `ActionGetData()`

```

public ConvertisseurDeviseViewModel()
{
    ActionGetData();
}

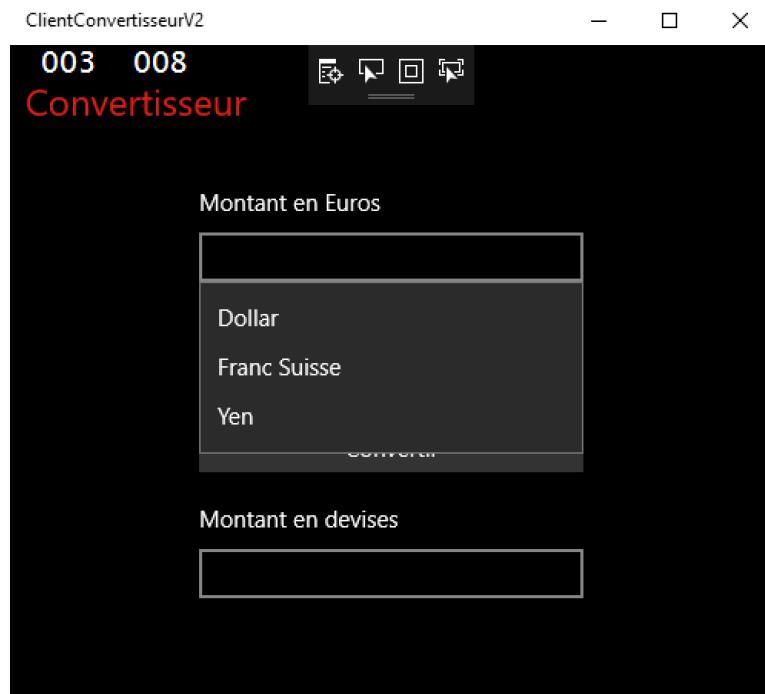
private async void ActionGetData()
{
    var result = await ... GetAllDevisesAsync();
    ComboBoxDevises = new ObservableCollection<Devise>(result);
}

```

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

Terminer le codage de l'alimentation de la combobox.

Permettre la prise en compte des certificats. Lancer l'application (définir le projet en tant que projet de démarrage à l'aide du bouton droit de la souris). 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.aspx>). Par exemple, le contrôle `Button` possède (par héritage) une propriété `Command` du type `ICommand` (<https://docs.microsoft.com/en-us/uwp/api/windows.ui.xaml.input.icommand>) 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 `ConvertisseurDeviseViewModel`, ajouter le code suivant permettant de gérer le bouton :

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

```
public ConvertisseurDeviseViewModel()
{
    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.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 :

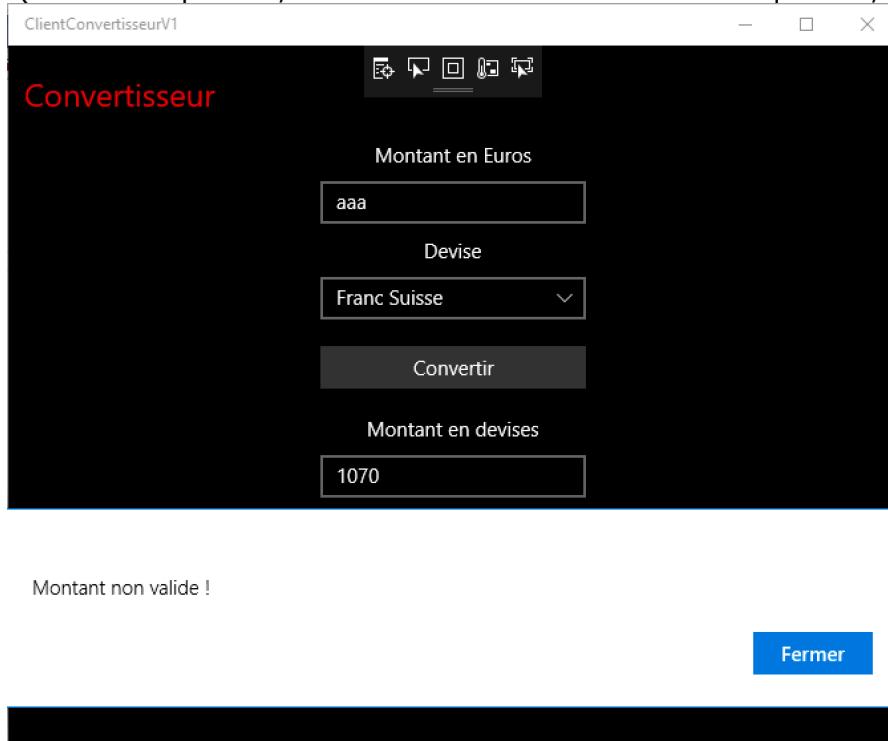
- <ComboBox x:Name="cbxDevise" ... SelectedItem="{Binding ComboBoxDeviseItem, Mode=TwoWay}"/>
- Créer une property nommée ComboBoxDeviseItem (type : Devise).

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.

Vous pourrez utiliser float.Parse() ou float.TryParse() pour convertir une chaîne de caractères en float.

Exceptions

Gérer les exceptions (WS non disponible, valeurs saisies non valides ou manquantes, etc.).



2.3. Ajout d'une nouvelle page

Dans le dossier View, ajouter une nouvelle page (Page vierge XAML) 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(ConvertisseurDevisepage), e.Arguments);
```

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 ConvertisseurDevisepage.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 de code...

3. Améliorations du WS : tests unitaires

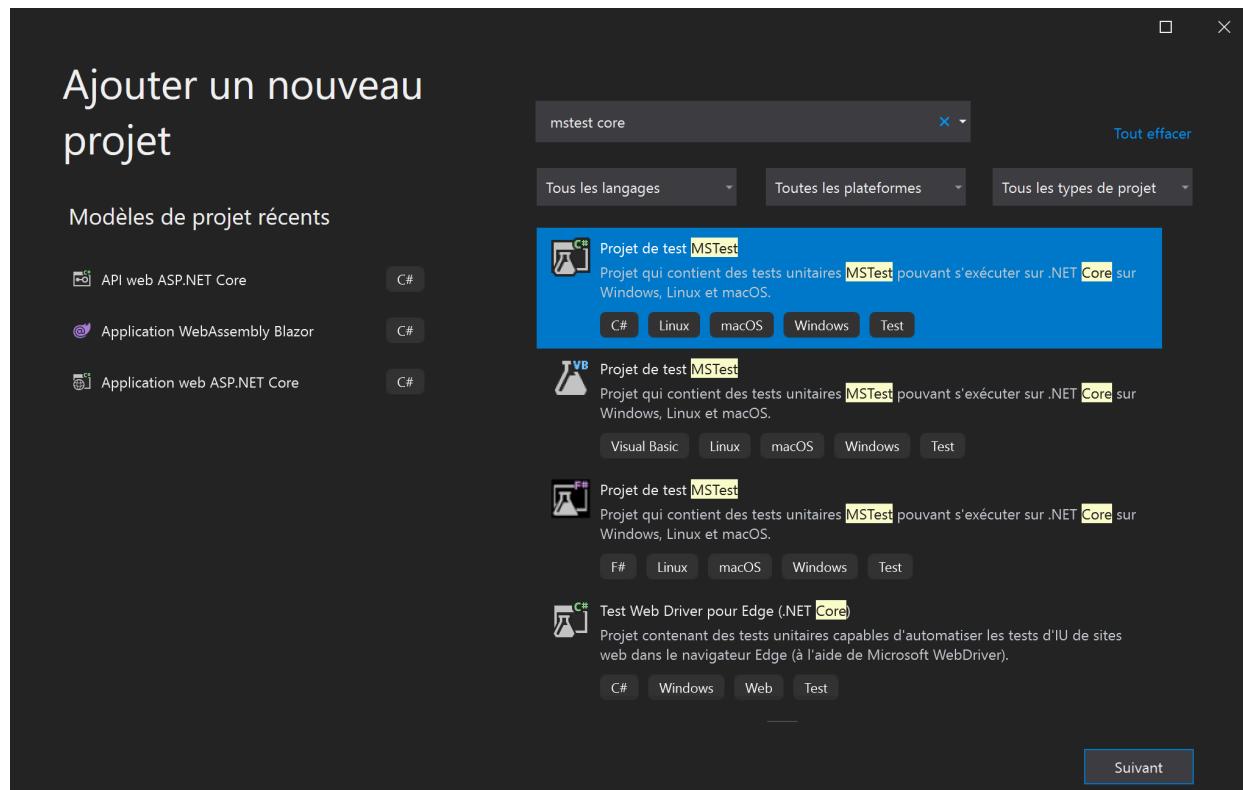
Une API doit toujours être testée : vous devez au minimum réaliser des tests unitaires.

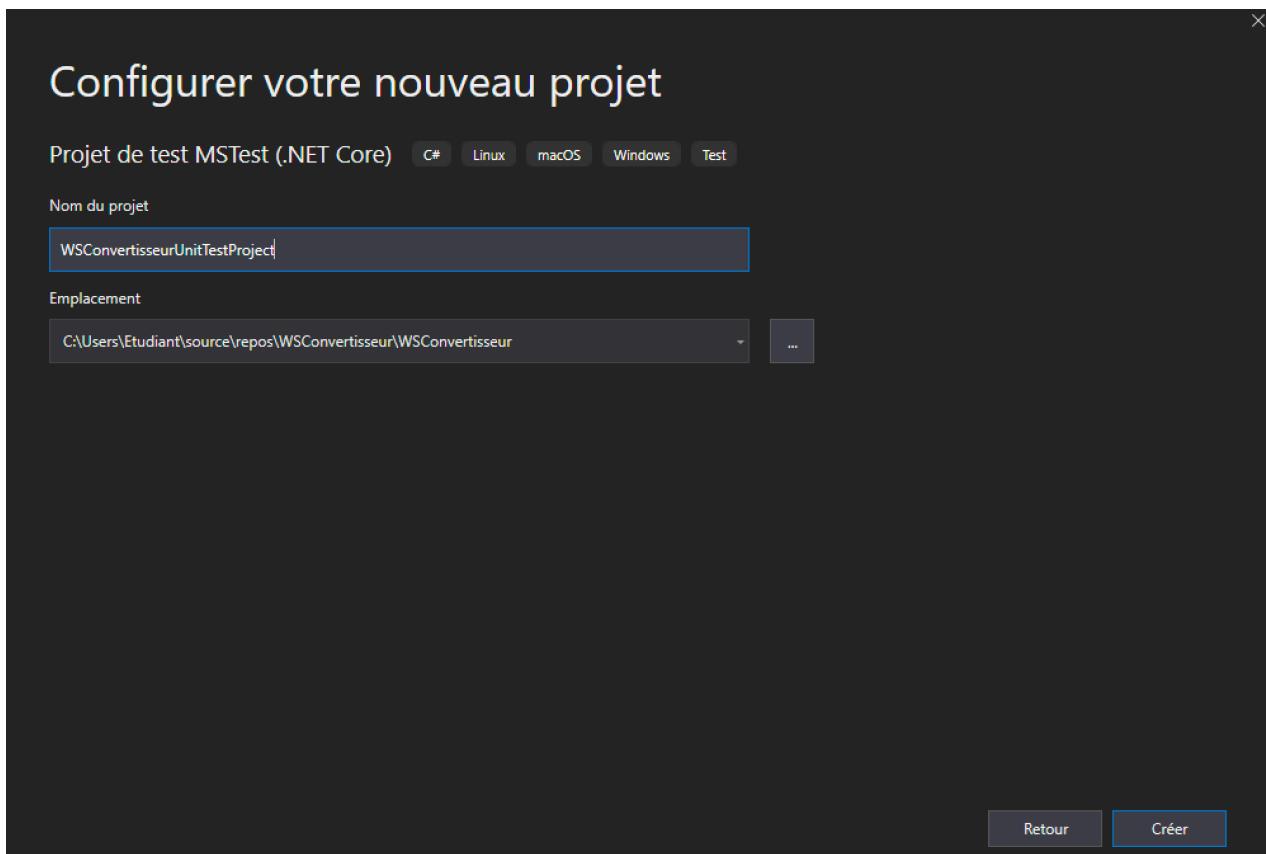
Dans notre cas, il n'est pas utile de tester la couche *Model* car elle ne contient qu'une classe métier simple. Remarquez, que dans le cas d'une couche *Model* générée via un ORM tel que Entity Framework ou Entity Framework Core, on ne crée pas non plus de test.

Nous allons donc uniquement tester le contrôleur DeviseController.

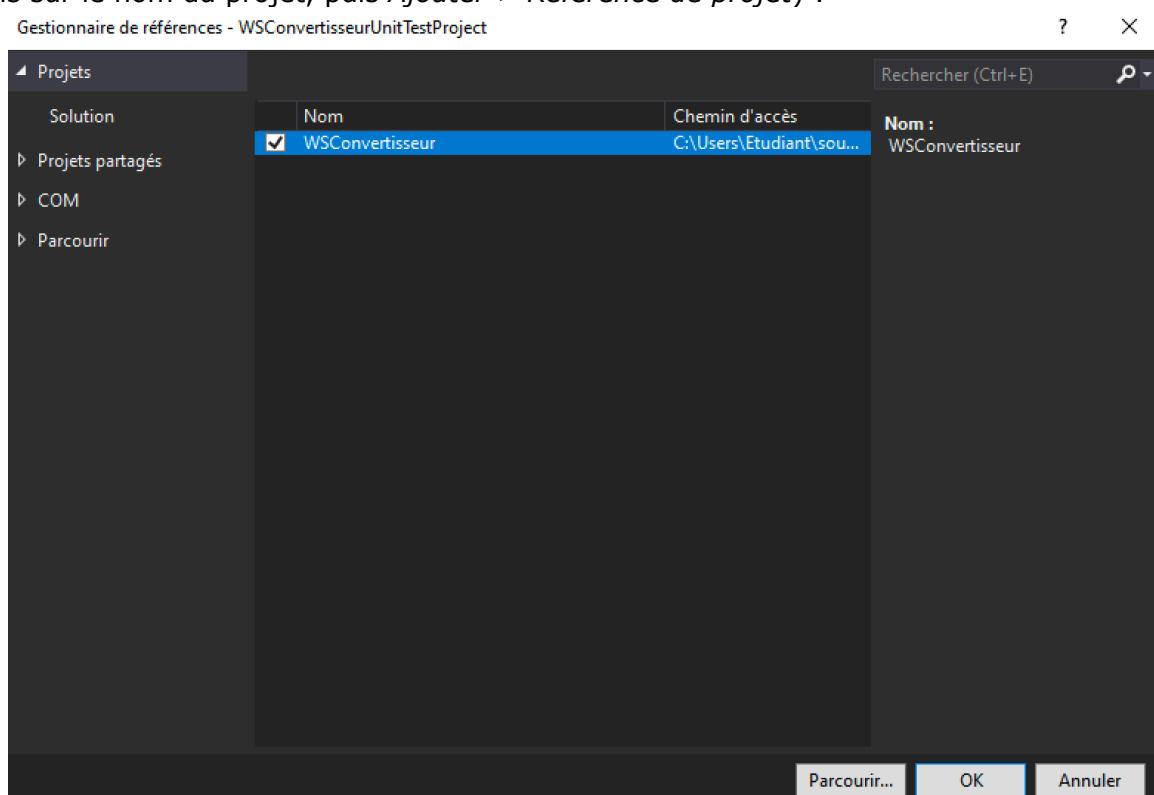
Dans la solution du WS, ajouter un nouveau projet de type « Projet de test MSTest (.NET Core) » :

MSTest est plus simple à utiliser, mais si vous préférez, vous pouvez créer des tests unitaires avec xUnit.





Ajouter une référence vers le projet WSConvertisseur pour lequel nous allons coder les tests (bouton droit de la souris sur le nom du projet, puis *Ajouter > Référence de projet*) :



Exemple : test de la méthode `GetById` :

Il faut (au moins) créer un test par type de retour (erreur 404, Devise). Ici, nous allons en réaliser 3 :

```
[TestMethod]
public void GetById_ExistingIdPassed_ReturnsOkObjectResult()
{
    // Arrange
    var _controller = new DeviseController();
```

```

    // Act
    var result = _controller.GetById(1);

    // Assert
    Assert.IsInstanceOfType(result, typeof(OkObjectResult), "Pas un OkObjectResult");
}

```

Explications :

- Le dernier argument de Assert.xxx est un message optionnel.
- La méthode `Ok()` du contrôleur retourne un `OkObjectResult`, c'est donc ce type que nous testons. Cf. documentation de la classe `ControllerBase` :

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.controllerbase.OkObject>

`Ok(Object)` Creates an `OkObjectResult` object that produces an `Status200OK` response.

```

[TestMethod]
public void GetById_ExistingIdPassed_ReturnsRightItem()
{
    // Arrange
    var _controller = new DeviseController();

    // Act
    var result = _controller.GetById(1) as OkObjectResult;

    // Assert
    Assert.IsInstanceOfType(result.Value, typeof(Devise), "Pas une Devise");
    Assert.AreEqual(new Devise(1, "Dollar", 1.08), (Devise)result.Value, "Devises pas identiques");
}

```

```

[TestMethod]
public void GetById_UncorrectIdPassed_ReturnsNotFoundResult()
{
    // Arrange
    var _controller = new DeviseController();

    // Act
    var result = _controller.GetById(20);

    // Assert
    Assert.IsInstanceOfType(result, typeof(NotFoundResult), "Pas un NotFoundResult");
}

```

Plus de détail sur le Framework MSTest v2 :

<https://docs.microsoft.com/fr-fr/dotnet/core/testing/unit-testing-with-mstest>

Classe `Assert` :

[https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert\(v=vs.140\).aspx](https://msdn.microsoft.com/en-us/library/microsoft.visualstudio.testtools.unittesting.assert(v=vs.140).aspx)

Comme la partie « Arrange » est dupliquée, refactoriser le code en créant un constructeur.

Si nécessaire on peut aussi ajouter des méthodes `[TestInitialize]` et `[TestCleanup]`.

```

[TestInitialize]
public void InitialisationDesTests()
{
    // Rajouter les initialisations exécutées avant chaque test
}

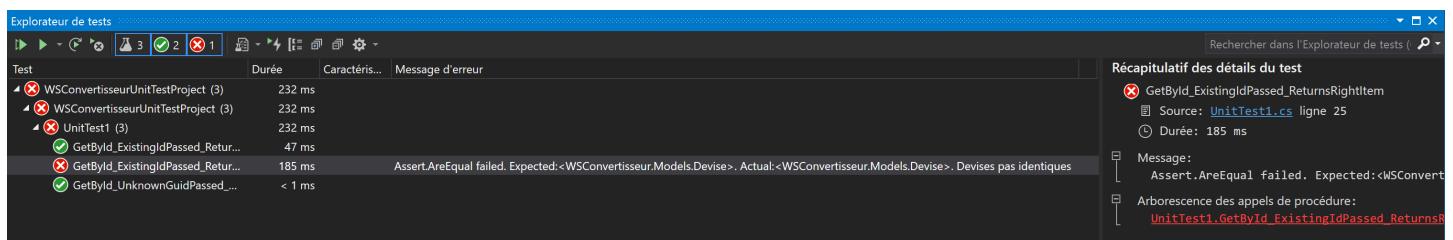
[TestMethod]
public void MonTest()
{
    // test à faire
}

```

```
[TestCleanup]
public void NettoyageDesTests()
{
    // Nettoyer les variables, ... après chaque test
}
```

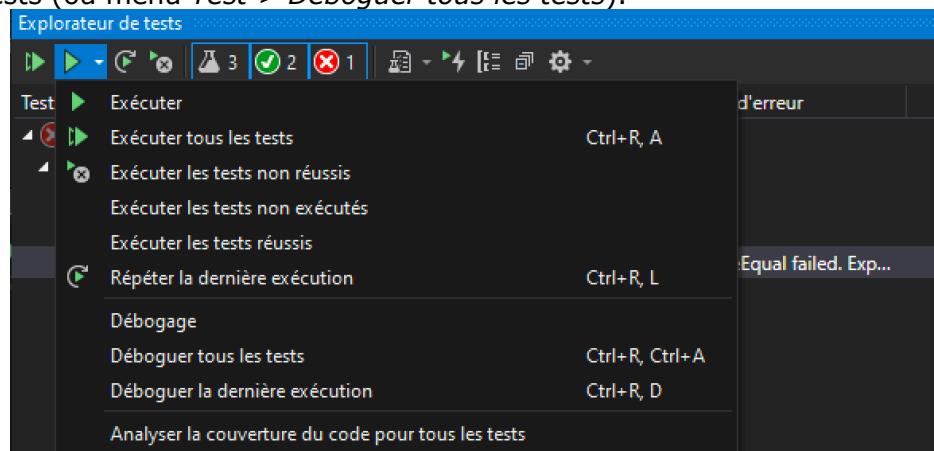
Attention ces méthodes sont exécutées lors de chaque test.

Exécuter tous les tests (menu *Test*) :

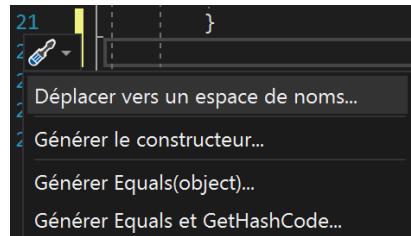


La 2^{nde} méthode de test n'est pas fonctionnelle. Plus exactement, c'est la ligne suivante qui ne l'est pas :
`Assert.AreEqual(new Devise(1, "Dollar", 1.08), (Devise)result.Value, "Devises pas identiques");`

On peut le vérifier en débuguant les tests : mettre un point d'arrêt puis en lançant le débogage à partir de l'explorateur de tests (ou menu *Test > Déboguer tous les tests*).



En effet, vous devez coder la méthode `Equals` de la classe `Devise`. Vous pouvez aussi la générer (*Générer Equals et GetHashCode*) :



Résultat :

Test	Durée	Caractéris...	Message d'erreur
WSConvertisseurUnitTestProject (3)	59 ms		
WSConvertisseurUnitTestProject (3)	59 ms		
UnitTest1 (3)	59 ms		
GetById_ExistingIdPassed_Return...	58 ms		
GetById_ExistingIdPassed_Return...	1 ms		
GetById_UnknownGuidIdPassed_...	< 1 ms		

Couverture de code (menu *Test > Analyser la couverture de code pour tous les tests*) :

Résultats de la couverture du code				
Etudiant_DESKTOP-35UT1OG 2021-09-19 2	Non couverts (blocs)	Non couverts (% blocs)	Couverts (blocs)	Couverts (% blocs)
Hiérarchie				
Etudiant_DESKTOP-35UT1OG 2021-09-19 ...	134	68,37 %	62	31,63 %
wsconvertisseur.dll	134	75,71 %	43	24,29 %
WSConvertisseur	60	100,00 %	0	0,00 %
WSConvertisseur.Controllers	65	77,38 %	19	22,62 %
WSConvertisseur.Models	9	27,27 %	24	72,73 %
wsconvertisseurunittestproject.dll	0	0,00 %	19	100,00 %

Remarque : la partie suivante sur Live Unit Testing ne peut être réalisée que si vous avez Visual Studio Enterprise. C'est cette version qui est disponible sur la VM.

Dans le menu *Test > Live Unit Testing*, cliquer sur « Démarrer ». Le live unit testing permet d'avoir des indicateurs en temps réel sur la couverture de code ainsi que son statut. Ces indicateurs sont représentés par des petits sigles sur le côté gauche du code. Ainsi, au fur et à mesure que vous allez écrire votre code de test, celui-ci sera vérifié et exécuté.

```

30 [TestMethod]
31 0 références | 0 exceptions
32 public void GetById_ExistingIdPassed_ReturnsRightItem()
33 {
34     // Act
35     var result = _controller.GetById(1) as OkObjectResult;
36 
37     // Assert
38     Assert.IsInstanceOfType(result.Value, typeof(Devise), "Pas une Devise");
39     Assert.AreEqual(new Devise(1, "Dollar", 1.08), (Devise)result.Value, "Devises pas identiques");
}

```

Si vous modifiez votre méthode de test, on peut voir en temps réel si elle réussit ou est en échec.

```

30 [TestMethod]
31 0 références | 0 exceptions
32 public void GetById_ExistingIdPassed_ReturnsRightItem()
33 {
34     // Act
35     var result = _controller.GetById(2) as OkObjectResult;
36 
37     // Assert
38     Assert.IsInstanceOfType(result.Value, typeof(Devise), "Pas une Devise");
39     Assert.AreEqual(new Devise(1, "Dollar", 1.08), (Devise)result.Value, "Devises pas identiques");
}

```

Coder les tests des autres méthodes de l'API.

Pour les tests de la méthode `GetAll`, vous pourrez utiliser la classe `CollectionAssert` (<https://msdn.microsoft.com/fr-fr/library/microsoft.visualstudio.testtools.unittesting.collectionassert.aspx>).

Vous devez obtenir une couverture de 100% sur le contrôleur Devise :

Résultats de la couverture du code				
Etudiant_DESKTOP-35UT1OG 2021-09-19 2	Non couverts (blocs)	Non couverts (% blocs)	Couverts (blocs)	Couverts (% blocs)
Hiérarchie				
Etudiant_DESKTOP-35UT1OG 2021-09-19 ...	88	34,92 %	164	65,08 %
wsconvertisseur.dll	88	49,72 %	89	50,28 %
WSConvertisseur	60	100,00 %	0	0,00 %
WSConvertisseur.Controllers	19	22,62 %	65	77,38 %
DevisController	0	0,00 %	59	100,00 %
DevisController.<>c__DisplayC...	0	0,00 %	2	100,00 %
DevisController.<>c__DisplayC...	0	0,00 %	2	100,00 %
DevisController.<>c__DisplayC...	0	0,00 %	2	100,00 %
WeatherForecastController	10	100,00 %	0	0,00 %
WeatherForecastController.<>c...	9	100,00 %	0	0,00 %

4. Pour les plus rapides

4.1. Client : création d'un menu hamburger

Dans le dossier View de la version 2, ajouter une page nommée `RootPage` contenant un contrôle `SplitView`.

Pour le contenu du fichier XAML, cf.

<https://ivan.vlaevski.com/hamburger-menu-on-windows-10-universal-app/>

<https://www.c-sharpcorner.com/UploadFile/5c2d70/windows-10-split-view-unleashing-hamburger-menu/>

Police de caractères à utiliser pour les boutons : <http://modernicons.io/segoe-mdl2/cheatsheet/>



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(ConvertisseurDevisePage));
}
```

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

- Code du bouton Hamburger (premier bouton) :

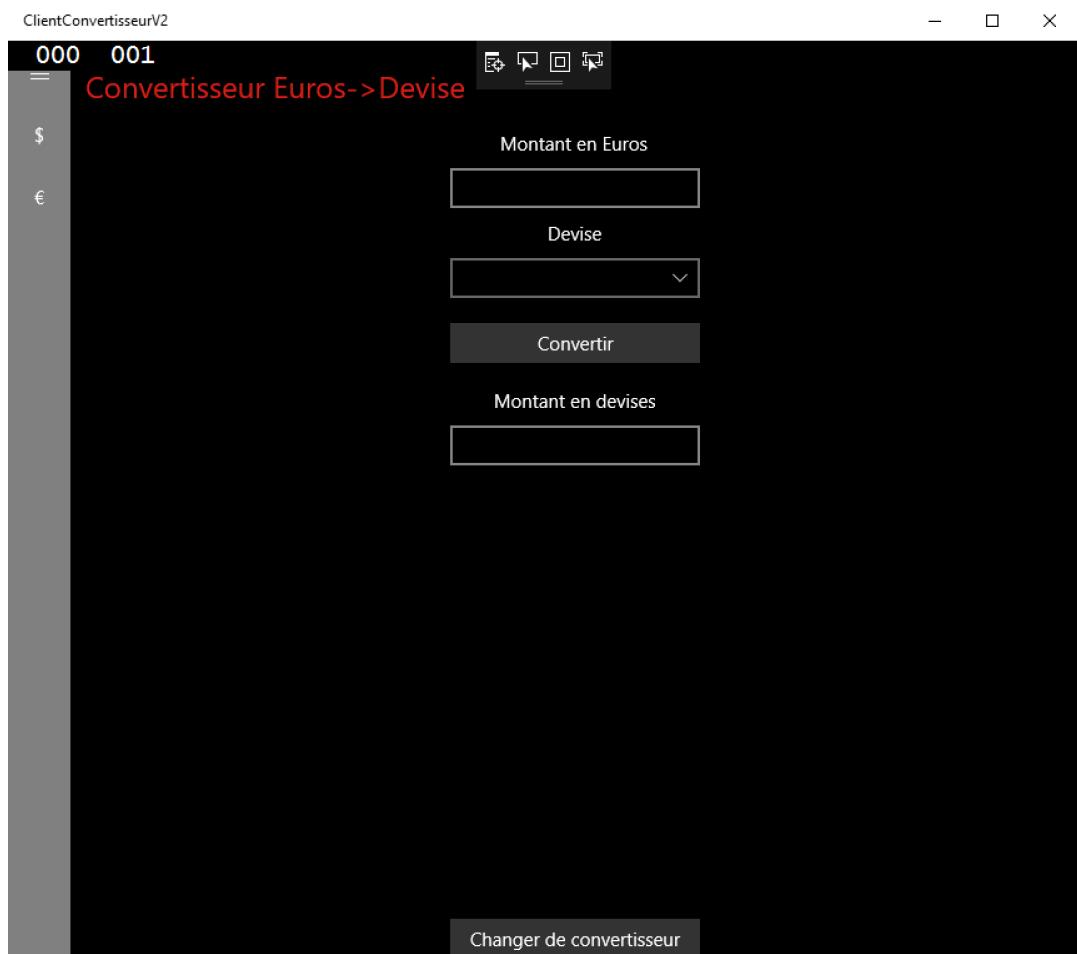
```
MySplitView.IsPaneOpen = !MySplitView.IsPaneOpen;
```

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

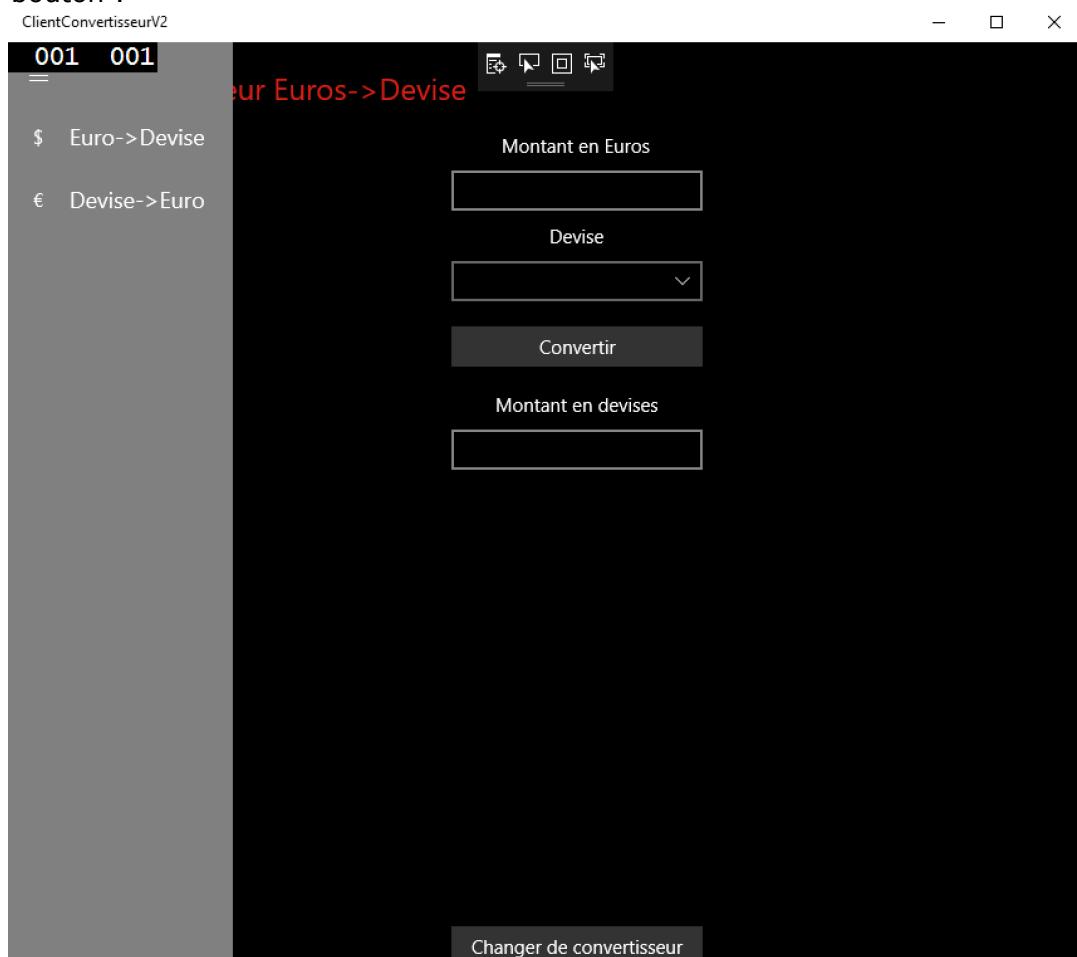
Modifier le code suivant dans le fichier `App.xaml.cs` :

```
Window.Current.Content = new RootPage(rootFrame);
...
rootFrame.Navigate(typeof(RootPage), e.Arguments);
```

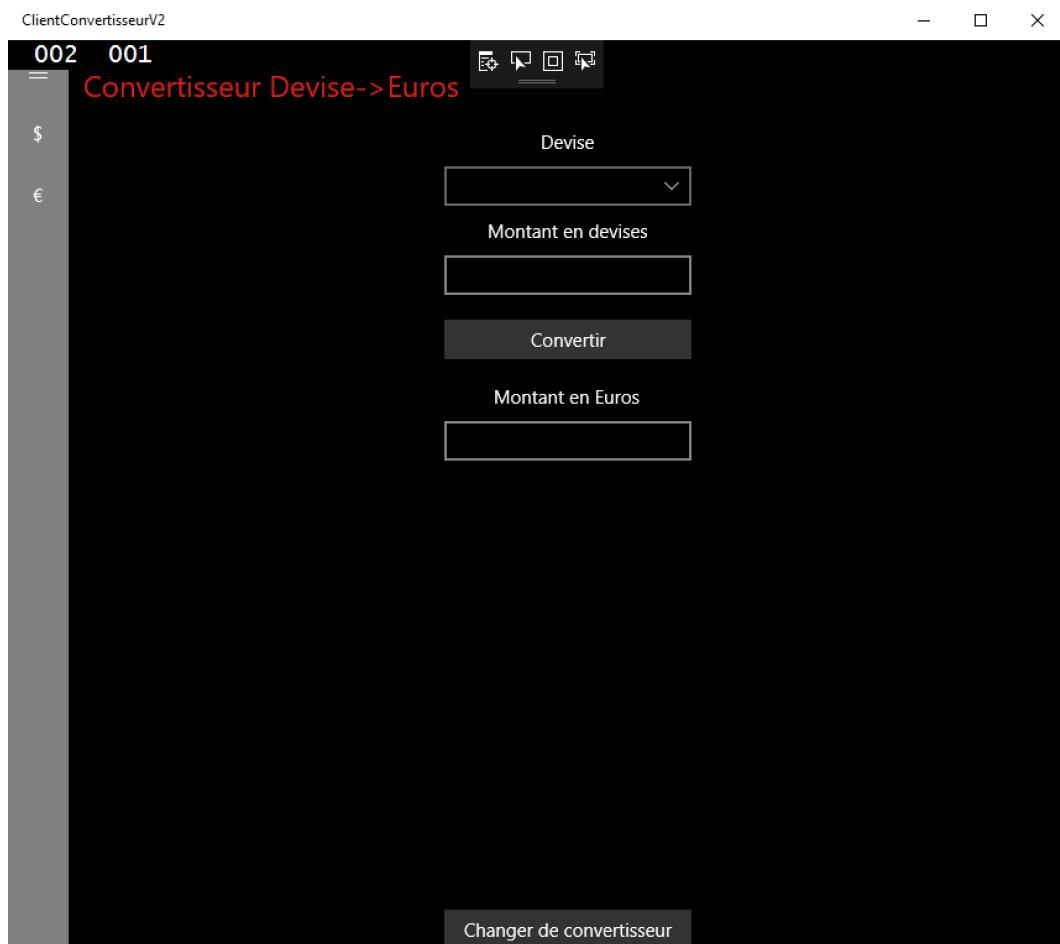
Résultat :



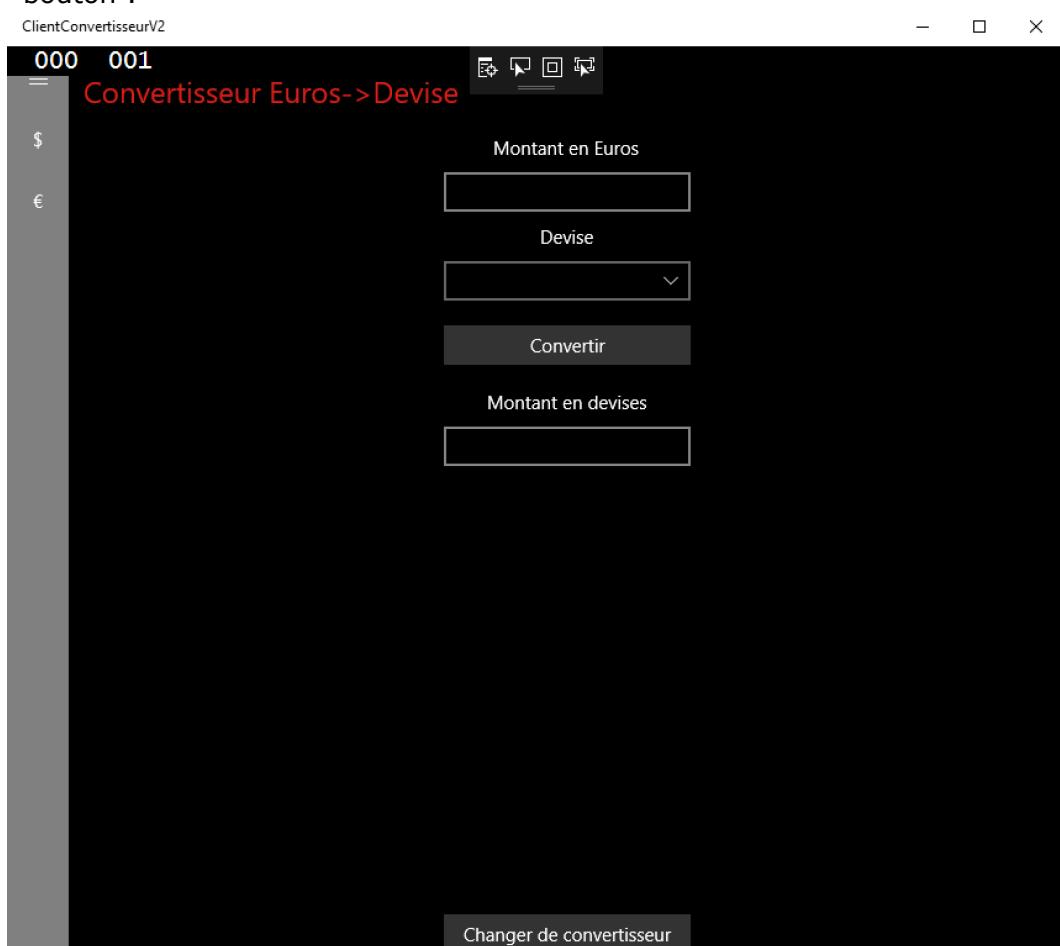
Clic sur le 1^{er} bouton :



Clic sur le 3ème bouton :



Clic sur le 2ème 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(ConvertisseurEuroPage));
}
```

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 à la faire sur de la navigation entre pages. Le MVVM a par contre été respecté sur les pages de code métier.

4.2. API : Formateur XML

Ajouter l'attribut `[Produces]` en entête de l'API : `[Produces("application/xml")]`. Lancer l'API. Vous obtiendrez une erreur « 406 Not Acceptable » :

The screenshot shows the Postman interface. At the top, it says "GET" and the URL "https://localhost:5001/api/devise/1". Below the URL, there are tabs for "Params", "Authorization", "Headers (8)", "Body", "Pre-request Script", "Tests", "Settings", and "Cookies". The "Headers" tab is selected. Under "Headers", there is a table titled "Query Params" with one row: "Key" and "Value". In the "Body" tab, there is a table with columns "KEY", "VALUE", "DESCRIPTION", and "Bulk Edit". The "Value" column has a single entry: "Description". At the bottom, the status bar shows "406 Not Acceptable 626 ms 104 B Save Response". Below the status bar, there are buttons for "Pretty", "Raw", "Preview", "Visualize", "Text", and "JSON".

En clair, on impose par l'attribut `[Produces]` d'utiliser le formateur XML, mais celui-ci n'est pas disponible.

Dans ce cas, nous devons ajouter un package NuGet permettant la gestion du format XML et configurer le modèle MVC pour le prendre en charge. Il existe des formateurs distincts pour les entrées et pour les sorties. Les formateurs d'entrée sont utilisés par la liaison de modèle (`content`) ; les formateurs de sortie sont utilisés pour mettre en forme les réponses. Il est également possible de créer des formateurs personnalisés.

Dans notre cas, nous allons uniquement configurer un nouveau formateur de sortie à titre d'exemple.

Installer le package NuGet `Microsoft.AspNetCore.Mvc.Formatters.Xml`.

Configurer XML comme formateur de sortie dans le fichier `Startup.cs` :

```
services.AddControllers(options =>
{
    options.OutputFormatters.Add(new XmlSerializerOutputFormatter());
});
```

Pour configurer XML comme formateur d'entrée :

```
services.AddControllers()
    .AddXmlSerializerFormatters();
```

Test (pas besoin de spécifier un header `Accept`) :

The screenshot shows the Postman interface with a successful API call. The URL is `https://localhost:5001/api/devise/1`. The Headers tab shows 8 items. The Body tab displays a JSON response:

```

1   <Devise xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2     <Id>1</Id>
3     <NomDevise>Dollar</NomDevise>
4     <Taux>1.08</Taux>
5   </Devise>

```

Nous obtenons une réponse au format imposé, XML.

Nous souhaiterions finalement que notre application gère le format XML en plus du format par défaut Json. Mettre en commentaire l'attribut `[Produces]` de l'API. Ainsi, le format de sortie sera celui spécifié grâce à l'entête `Accept` de l'appel.

Test avec le header `Accept = application/xml` :

The screenshot shows the Postman interface with a successful API call. The URL is `https://localhost:5001/api/devise/1`. The Headers tab shows 9 items, including `Accept: application/xml`. The Body tab displays the same XML response as before:

```

1   <Devise xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2     <Id>1</Id>
3     <NomDevise>Dollar</NomDevise>
4     <Taux>1.08</Taux>
5   </Devise>

```

Test avec `Accept = application/json` :

The screenshot shows the Postman interface with a successful API call. The URL is `https://localhost:5001/api/devise/1`. The Headers tab is selected, showing two entries: `Accept: application/json` and `Content-Type: application/json`. The Body tab shows a JSON response with the following structure:

```

1  {
2   "id": 1,
3   "nomDevise": "Dollar",
4   "taux": 1.08
5 }

```

The response status is 200 OK, with 143 ms latency and 189 B size.

Test sans header :

The screenshot shows the Postman interface with a successful API call. The URL is `https://localhost:5001/api/devise/1`. The Headers tab is selected, showing no entries. The Body tab shows a plain text response with the following content:

```

1  {
2   "id": 1,
3   "nomDevise": "Dollar",
4   "taux": 1.08
5 }

```

The response status is 200 OK, with 63 ms latency and 189 B size.

Json est bien le format par défaut.

5. ANNEXE : procédure à suivre pour installer Swagger (si OpenAPI n'a pas été coché)

1. Ajouter la **dernière version** du package NuGet `Swashbuckle.AspNetCore` (menu *Outils > Gestionnaire de package NuGet > Gérer les packages NuGet pour la solution*) :
2. Modifier le fichier `Startup.cs` pour enregistrer Swagger.

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();

    // Configure Swagger
    services.AddSwaggerGen(c =>

```

```

        {
            c.SwaggerDoc("v1", new OpenApiInfo { Title = "WSConvertisseur", Version =
"v1" });
        });
    }
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();
    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "TP1 API Documentation");
    });

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}

```