Community Tookit MVVM

TABLE DES MATIERES

Ce document gratuit vous est offert par Olivier Dahan et E-Naxos via le premier blog français sur les technologies de développement Microsoft : Dot.Blog (www.e-naxos.com/blog).

N'hésitez pas à faire appel à Olivier pour tous vos besoin d'audit, conseil et développement.

1 MVVM RAPPEL

MVVM sont les initiales de Model / View / ViewModel, les trois tiers d'une application moderne qui en réalité utilise aussi d'autres paradigmes tels que les services, les micro-services, et d'autres patterns architecturaux. Toutefois MVVM, très proche de MVC ou d'autres patterns de ce type, s'applique principalement dans le champ d'application possédant une interface utilisateur sophistiquée. Dans le monde Microsoft c'est MVVM qui s'est imposé car ce pattern prend en compte la particularité du Binding de XAML et est mieux adapté à ce contexte.

Le Model représente les données utilisées par l'application, la View (vue) l'interface utilisateur graphique (GUI), le ViewModel étant le code permettant de présenter les données du Model à la View mais aussi de prendre en charge les commandes et autres interactions de l'utilisateur avec l'application (et, le plus souvent, de modifier les données du Model en conséquence).

Les avantages de MVVM et des patterns proches sont multiples :

- Meilleure réutilisation du code (DRY)
- Comportements *platform-agnostic*
- Meilleure testabilité
- Séparation claire entre le code GUI et le code applicatif, permettant une séparation plus nette entre Design et Développement
- Éviter le code spaghetti en proposant un cadre architectural qui a fait ses preuves

2 POURQUOI UN NOUVEAU TOOLKIT MAINTENANT ?

Depuis un moment certains toolkits très utilisés sont en perte non pas de vitesse mais de maintenance… Les concepteurs comme les équipes à l'origine de ces toolkits, comme nous, finissent par se lasser ou changer de poste, de vie quand on envisage des durées de plusieurs années. Et c'est un problème pour beaucoup de développeurs attachés à ses toolkits aussi bien que pour ceux qui envisageaient d'utiliser l'un de ces toolkits MVVM.

Par exemple Caliburn.Micro n'est plus maintenant. Bourré de bonnes idées, très complet, il n'est définitivement plus maintenu.

Mais c'est le cas aussi de MVVM Light de Laurent Bugnion. Et là c'est plus problématique car il s'agit du toolkit MVVM le plus utilisé dans les applications réalisées depuis des années : WPF, Silverlight, Windows Phone, UWP, Xamarin.Android, Xamarin.iOS, Xamarin.Forms… Tous les environnements C#/XAML Microsoft ont eu le droit à leur version de MVVM Light. Et ce à l'identique, sans confusion possible.

Certes Prism existe toujours et continue à être maintenu et pourrait être un excellent choix pour aiguiller les développeurs ayant perdu leur outil habituel.

Mais il faut noter plusieurs particularités de Prism qui interdisent une telle démarche, ou qui la rende moins simple et moins efficace qu'il pourrait y paraître.

Tout d'abord Prism n'est qu'un nom. Ce n'est pas un toolkit unique adapté à plusieurs environnements, c'est un code réécrit *from scratch* pour chaque environnement par des équipes

différentes. Ainsi, le Prism original, « le vrai », celui pour WPF que je conseillais à l'époque, n'a absolument rien à voir par exemple avec celui pour WinRT Windows 8 ou UWP. C'est très gênant ces changements d'API on en conviendra (et c'était justement la force de MVVM Light de rester le même partout). Il faut aussi noter que Prism est plus complexe, et pas forcément « meilleur » pour autant. Plus difficile de le faire adopter par des équipes de niveau hétérogène notamment. La documentation de Prism, ce que l'existence de ses différentes versions ne simplifie pas, est soit inexistante soit incomplète, soit non mise à jour. Il devient difficile d'utiliser un toolkit dans de telles conditions. Dernièrement, et de façon rédhibitoire pour nous, l'équipe Prism a annoncé après de longues tergiversations qu'elle renonçait à adapter son système de navigation pour le SHELL de Xamarin.Forms qui est le moyen officiel d'ajouter des menus aux apps.

Pour toutes ces raisons et d'autres encore, il a été décidé de recréer un remplaçant à MVVM Light, assez proche pour rendre les migrations assez faciles, mais résolument moderne, réécrit totalement pour être plus rapide et tirer profit des évolutions de C# et de l'environnement.

> *Ainsi, depuis 2020, la communauté .NET s'est lancée dans la conception de ce nouveau toolkit. Aidée au départ par Laurent Bugnion – qui considère d'ailleurs que ce dernier est la suite logique de MVVM Light et qu'en conséquence il conseille vivement.*

Le nouveau projet est bien entendu Open Source, utilise .NET Standard 2.0 pour la facilité de portage cross-plateforme, cross-technologie, il est optimisé pour une meilleure utilisation du CPU et de la mémoire.

Le Microsoft MVVM Toolkit est ainsi intégré au ***Windows Community Toolkit***, lui-même sous la houlette de la ***Dotnet Foundation***. Microsoft n'est ni l'auteur ni le propriétaire du projet, c'est un projet communautaire, même si bien sûr la société est très impliquée et fournit un appui non négligeable. Il se trouve désormais dans le paquet ***CommunityToolkit.Mvvv***.

3 L'ETAT D'AVANCEMENT

Ce papier arrive au bon moment (ou c'est parce que c'est le bon moment que ce papier arrive…) car en effet, après un temps de gestation compréhensible le Microsoft MMVM Toolkit arrive au state de *Release Candidat* et peut être utilisé dans les nouveaux projets (avec prudence, une RC n'est pas une version finale, mais elle est assez solide).
Pour se faire une idée, au début octobre, l'historique du projet est le suivant :

Version	Téléchargements	Mise à jour
7.1.1-rc2	23	Il y a un jour
7.1.0-rc1	473	Il y a 12 jours
7.1.0-preview1	420	Il y a un mois
7.0.2	35 511	Il y a 4 mois
7.0.1	13 246	Il y a 5 mois
7.0.0	13 562	Il y a 6 mois
…	Preview 3 à 5	
Preview 2	3 534	Le 8/11/2020
Finale 8.1.0+	**486 347**	**Octbre 2022**

Comme on peut le constater c'est un projet vivant qui évolue à un rythme soutenu et qui en un an est arrivé à la stabilité ce qui a permis avec confiance l'arrivée de la release officielle puis de ces évolutions.

4 LES CINQ GRANDS BLOCS

Le toolkit se partage en cinq grands blocs qui sont :

- **`ObservableObject`** (**`INotifyPropertyChanged`**)
- **`RelayCommand`** (**`ICommand`**)
- **`Messenger`** (Pub/Sub)
- Inversion de contrôle (**`IServiceProvider`**)
- La génération de code

Les objets « *observables* » c'est-à-dire capable de déclencher une notification de changement sont à la base de MVVM et du Data Binding de XAML, on retrouve ce concept dans tous les toolkits MVVM (sous XAML donc).

Les *commandes* sont des implémentations de l'interface **`ICommand`** facilement *interfaçable* avec le code du développeur. Elles jouent un rôle essentiel dans les *interactions* de l'utilisateur final avec l'application. On retrouve le concept là encore dans tous les toolkits.

MVVM ayant pour principe le **découplage** le plus strict possible il rend impossible (par principe) certaines communications par exemple un ViewModel ne peut pas connaître la Vue qui lui est attachée (en réalité aucun élément ni classe d'UI), un élément du Model ne peut pas connaître un ViewModel précis, etc. Deux ViewModels ne peuvent même pas se connaître. Entendre « se connaître » comme un équivalent à un « **`using xxx;`** ». De fait il faut trouver un moyen non pas d'autoriser (ce serait violer le pattern) mais de permettre ces communications qui souvent sont indispensables. C'est la *messagerie* qui joue ce rôle sous MVVM et c'est tout naturellement un élément qu'on retrouve dans tous les toolkits.

L'inversion de contrôle comme l'injection de dépendances sont un « plus », une aide précieuse pour le découplage. Cela permet, via des interfaces, au code d'appeler des services rendus par d'autres parties du code sans avoir à « se connaître » (au sens indiqué plus haut).

Au tout début des travaux sur le MS Toolkit il y avait une hésitation entre reprendre le **`SimpleIoC`** de MVVM Light, simple et très léger, ou utiliser le module existant de chez Microsoft, un peu plus lourd mais plus complet. L'adoption de l'interface **`IServiceProvider`** permet de régler le problème est rendant l'implémentation du moteur d'IoC indépendant du toolkit. Par défaut celui-ci utilise le moteur Microsoft désormais, mais chacun peut utiliser celui qu'il préfère.

5 AU-DELA DES GRANDS BLOCS

Les quatre grands blocs présentés ci-dessus ne sont qu'une approximation très grossière de ce que contient le toolkit. Bien que résolument simple, dans la lignée de MVVM Light ne l'oublions pas (*Light* = léger !), il contient tout de même certains ajouts qui avec le temps étaient devenus nécessaires.

Par exemple on va trouver des notifications liées aux tâches (**`Task`**) qui permettent notamment d'effectuer une notification INPC une fois qu'une certaine tâche liée à la modification d'une valeur de propriété sera terminée (afin que les utilisateurs de la valeur puissent voir uniquement la nouvelle valeur une fois qu'elle sera créée).

On trouve aussi des méthodes ou objets simplifiant l'implémentation des ViewModels, des propriétés et leur *setter*, voire même des ViewModels offrant l'interface **`INotifyDataErrorInfo`** pour la gestion des validations de saisie.

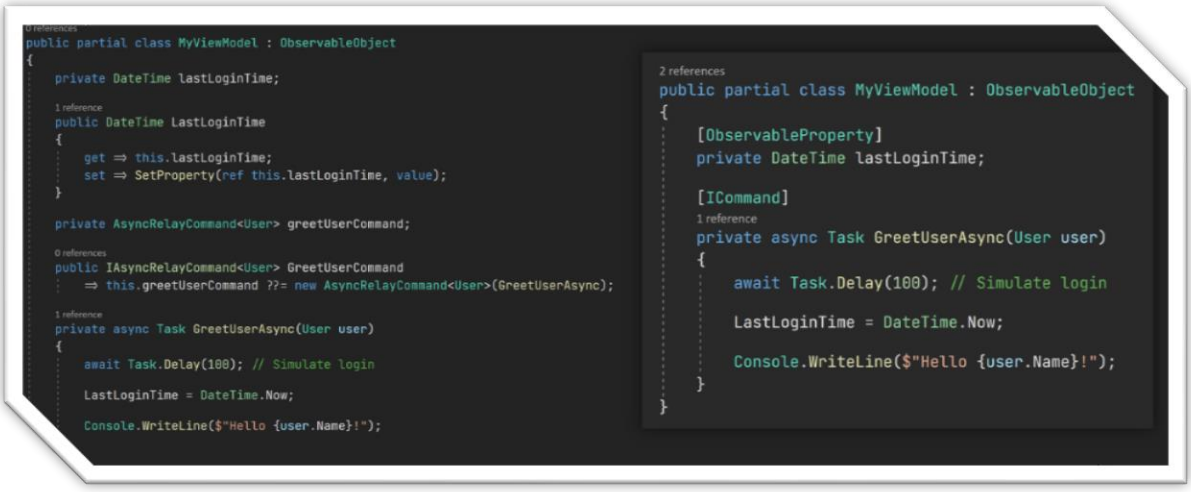
Il existe aussi deux nouvelles commandes qui prennent en charge *l'asynchronisme* avec annulation possible en cours de travail.

Tout le code ayant été réécrit dans le but d'une plus grande vitesse et d'une consommation mémoire maîtrisée, la messagerie utilise par exemple des **`WeakReference`**. Si cela prévient des fuites mémoire il s'agit d'une indirection qui coûte en vitesse d'exécution. Pour les cas (rares) où la rapidité d'exécution ou la consommation mémoire liée aux messages priment sur les éventuels memory leaks, il existe une seconde messagerie n'utilisant que des références fortes (*strong references*). Elle est bien plus rapide et autorise l'utilisation de la messagerie dans du code où la vitesse est cruciale.

6 GENERATION DE CODE AUTOMATIQUE

Parmi les avancées en cours, le toolkit a pour ambition de rendre encore plus simple son utilisation en exploitant la génération de code de **Roselyn**. En plaçant quelques *attributs* il sera donc possible d'écrire des propriétés complètes juste en partant d'une simple définition de champ privé par exemple. Le code sera généré à la volée et mis à jour en temps réel durant la frappe dans Visual Studio.

Un exemple est donné par la capture ci-dessous, à gauche le code tel qu'on l'écrit « classiquement » avec le toolkit, à droite le code qu'on peut écrire en tirant profit du générateur de code source :



```
public partial class MyViewModel : ObservableObject
{
    private DateTime lastLoginTime;

    public DateTime LastLoginTime
    {
        get => this.lastLoginTime;
        set => SetProperty(ref this.lastLoginTime, value);
    }

    private AsyncRelayCommand<User> greetUserCommand;

    public IAsyncRelayCommand<User> GreetUserCommand
        => this.greetUserCommand ??= new AsyncRelayCommand<User>(GreetUserAsync);

    private async Task GreetUserAsync(User user)
    {
        await Task.Delay(100); // Simulate login

        LastLoginTime = DateTime.Now;

        Console.WriteLine($"Hello {user.Name}!");
    }
}
```

```
public partial class MyViewModel : ObservableObject
{
    [ObservableProperty]
    private DateTime lastLoginTime;

    [ICommand]
    private async Task GreetUserAsync(User user)
    {
        await Task.Delay(100); // Simulate login

        LastLoginTime = DateTime.Now;

        Console.WriteLine($"Hello {user.Name}!");
    }
}
```

Il s'agit d'option, et chacun pourra choisir d'utiliser ou non cette extension. Les raisons principales de s'en servir : rapidité de frappe, moins de code, moins d'erreurs possibles, code plus concis, plus facilement lisible, programmation par intention, etc. La seule raison de ne pas s'en servir pourrait se cacher dans la pérennité. Tous les toolkits meurent un jour, le MS MMV toolkit est là pour nous le rappeler puisque même des stars comme MVVM Light ou Caliburn.Micro par leur disparition ont entraîné la création du nouveau toolkit. Ce dernier viendra forcément un jour à être remplacé ou à disparaître. Ce jour-là votre code sera toujours utilisable très certainement (le code C# et .NET ont survécu à toutes les évolutions jusqu'ici), mais si l'extension de génération ne fonctionne plus ou n'existe plus, vous ne récupérerez qu'un code avec de jolis attributs sans code fonctionnel derrière. Il faudra donc réécrire ce que le générateur avait écrit pour vous automatiquement.

Personnellement je n'aime pas que mon code soit lié à l'existence d'outils externes. Il doit être complet et fonctionnel et *surtout se suffire à lui-même* au moins sur l'essentiel.

Il existera donc des petits projets à durée vie courte pour lesquels la génération semble apporter plus d'avantages que d'inconvénients. Et des projets plus lourds ou s'inscrivant dans une durée plus longue pour lesquels il faudra, à mon avis, favoriser l'autonomie du code vis-à-vis des outils de développement et des toolkits.

Mais on peut déjà noter que ce type d'extension utilisant la génération de code de *Roselyn* marque plutôt le futur de ce que seront les outils dans les années à venir, sans compter sur l'IA qui fera son apparition bientôt pour aider à coder ou même se substituer au développeur dans certains cas… Nul n'est l'abris de l'IA ! L'informaticien de base est assis sur une branche que des informaticiens plus qualifiés sont en train de scier…

7 CROSS-PLATEFORME

Avant d'entrer dans le vif du sujet, et même si de nombreuses informations données jusqu'ici le laisse entendre, il semble essentiel de préciser que le Microsoft MVVM Toolkit est dès sa conception totalement cross-plateforme.

L'adoption de .NET Standard 2.0 comme seule dépendance permet d'utiliser le toolkit sur toutes les plateformes supportées par .NET, sachant – mais ce serait un autre sujet bien long à traiter – que la version 5 comme la 6 à venir ne sont que la suite des premières versions de .NET CORE. Le mot « **CORE** » (noyau) était utilisé à la base car *.NET CORE* se destinait plutôt aux serveurs. Un code .NET plus rapide à exécuter, plus facile à installer et à packager avec les applications, etc. Toutefois dans son aventure qui l'amène aujourd'hui à devenir le standard unificateur il a semblé, assez légitimement, à Microsoft que le mot « CORE », « noyau » donc, pouvait prêter à confusion plus qu'autre chose. Et depuis .NET CORE 5.0, on parle de .NET 5 tout court, tout comme la version de suivante s'appellera .NET 6 alors qu'il s'agira bien d'un .NET CORE 6. On peut facilement se tromper si l'on considère que la dernière version du « *framework .NET* » est la version 4.8 d'avril 2019, ce qui ferait de « .NET 5 » son successeur logique. Il s'agit d'une pure coïncidence, les versions de CORE et du framework ayant évolué chacune de leur côté depuis longtemps sans qu'une telle convergence n'ait été diaboliquement planifiée ! Mais parfois le hasard fait bien les choses…

Donc le toolkit MVVM a été conçu pour être cross-plateforme dès le départ, identique à lui-même (pas des versions différentes comme Prism), utilisable aussi bien sous UWP que WPF, Xamarin.Forms, MAUI et WinUI3.

8 REFERENCES

Dernier point avant d'aborder le toolkit lui-même, les références qui vous seront certainement utiles…

- Exemples et doc : https://tinyurl.com/yjkph6fc
- Le WCT (Windows Community Toolkit) : https://aka.ms/WCT
- Demo WCT et template : https://aka.ms/windowstoolkitapp
- Guide WCT : https://aka.sm/wct/wiki
- NameSpace MVVM : Microsoft.Toolkit.Mvvm
- Nuget MVVM : https://www.nuget.org/packages/Microsoft.Toolkit.Mvvm
- Migration depuis MVVM Light : https://tinyurl.com/yk6yyxf7
- MVVM Light dernier code publié sur github : https://tinyurl.com/ye836atc
- MVVM Light SimpleIOC : https://tinyurl.com/yk4myzvz
- CommonServiceLocator https://tinyurl.com/yzugqb2b
- Source Generator : https://tinyurl.com/yj4zvfut IA avec image classification https://tinyurl.com/yh2w542k

9 CLASSES ET SERVICES RENDUS PAR LE TOOLKIT

9.1 TOUR D'HORIZON

Il ne s'agira pas ici de faire un cours sur MVVM ni même sur l'utilisation fine du toolkit. Il s'agit plutôt d'un guide présentant les grandes lignes, les classes, leur utilité. Le tout en supposant que le lecteur possède déjà une connaissance du pattern MVVM et de certains toolkit comme MVVM Light, Prism, Caliburn.Micro, etc (ou au minimum le petit rappel en début d'article).

En suivant ce lien (https://www.e-naxos.com/Blog/search.aspx?q=mvvm) vous obtiendrez la liste de plus de 200 articles que j'ai écrit sur le sujet, le traitant de très près ou parfois de plus loin mais dont MVVM est toujours l'un des sujets principaux. Ne manquez pas aussi cette autre liste qui présente des papiers liées aux méthodologies, dont MVVM (avec un recoupement avec la liste précédente) : https://www.e-naxos.com/Blog/?tag=/methodologie ou https://www.e-naxos.com/Blog/?tag=/mvvm

Il reste que la meilleure façon de comprendre un toolkit est d'étudier son code source. Le Microsoft MVVM Toolkit étant Open Source (adresses données plus haut) je conseille au lecteur de télécharger le code source et de le regarder de près. Toutes les réponses, mêmes aux questions les plus incroyables s'y trouvent !

Bien entendu d'autres articles suivront pour montrer plus en détail l'utilisation des fonctionnalités du toolkit. Mais chaque chose en son temps !

9.2 INSTALLATION

Pour bénéficier des services du toolkit il suffit de l'ajouter au projet principal (UWP, Xamarin.Forms…)

La version courante au moment de l'écriture est la 8.1, préférez les versions stables pour vos mises en production, les RC ou Preview permettent de jouer avec les nouveautés mais ne sont pas validées pour de l'exploitation.

Dans Visual Studio c'est le package suivant qu'il faut ajouter : **CommunityToolkit.Mvvm**

9.3 CLASSE DE BASE OBSERVABLE

Il s'agit d'un ensemble de classes dont on peut hériter et dont le principe commun est de simplifier l'implémentation de **INPC** dans diverses situations.

- `ObservableObject`
  - Classe base implémentant `INotifiPropertyChanged` et offrant les méthodes `SetProperty`() avec diverses variantes.
  - On peut avec l'une des syntaxes rendre « INPC compatible » un modèle ou une classe qui n'hérite pas de `ObservableObject` ce qui peut s'avérer très pratique.
  - Les propriétés de type `Task<T>` sont supportées
- `ObservableValidator`
  - Classe qui implémente en plus de ce que `ObservableObject` propose l'interface `INotifyDataErrorInfo` pour la prise en compte de la **validation des données**.
  - INPC est implémenté dans sa version « ***changed*** » et « ***changing*** » pour plus de souplesse (requis le plus souvent pour la validation des données).
  - La classe propose un `TrySetProperty`() qui étend le `SetProperty` de base mais qui ne met à jour la propriété que si elle est validée par les mécanismes de validation mis en place.

  - On retrouve dans le même esprit un `ValidateProperty` ou `ValidateAllProperties`, `ClearAllErrors`, etc, dont le sens est assez clair pour qui s'est déjà frotté à la validation des données dans un ViewModel.
- `ObservableRecipient`
  - C'est une classe de base qui en plus des services des `ObservableObject` offre un accès intégré à la messagerie.
  - La propriété `IsActive` est aussi exposée avec la possibilité de répondre aux événements `OnActivated` et `OnDeactivated`
  - On trouve aussi les méthodes liées à la messagerie comme `BroadCast<T,T,string>`.
  - Cette classe est plutôt bien taillée pour devenir la *classe de base des ViewModels*.

9.4 LES COMMANDES

Les commandes sont à la base des interactions entre l'utilisateur et l'application. L'interface `ICommand` est le point de départ. Elle définit certaines méthodes à implémenter pour qu'une classe devienne une « commande » utilisable en tant que telle et puisse être liée à des éléments de l'interface visuelle.

L'aridité d'un codage systématique à la main de classes supportant `ICommand` a amené les toolkits MVVM à proposer assez rapidement des implémentations standard mais personnalisables. Le toolkit MVVM n'y échappe pas et reprend à la fois ce qu'on trouvait dans MVVM Light et Caliburn.Micro plus des nouveautés en ligne avec la programmation moderne comme l'asynchronisme.

- `RelayCommand`
  - La plus simple des commandes qui accepte un délégué qui sera exécuté immédiatement (une `Action`)
- `RelayCommand<T>`
  - En plus de `ICommand` la classe implémente `IRelayCommand<T>` qui expose `NotifyCanExecuteChanged`.
  - La classe accepte un délégué sous la forme d'une Action ou `Func<T>` ce qui permet de passer un paramètre typé à la commande.
- `AsyncRelayCommand` et `AsyncRelayCommand<T>` sont bâties sur le même principe à la différence qu'elles acceptent des délégués retournant une `Task` ouvrant la voie à des *commandes asynchrones* pouvant éventuellement être annulées (`CanBeCanceled`, `IsCancellationRequested`, `Cancel`).
  - Elles exposent une propriété `ExecutionTask` qui permet par exemple de monitorer l'état de la tâche exécutée par la commande.
  - On notera aussi le support de `IAsyncRelayCommmand` et `IAsyncRelayCommand<T>` ce qui permet aux ViewModels d'exposer des propriétés de ce type avec le minimum de couplage entre les types.
- `IRelayCommand` et `IRelayCommand<T>` qui prennent en charge la notification du changement de `CanExecute`

Comme on le voit le choix est assez vaste mais reste maîtrisable et se découpe en réalité en deux parties : les commandes classiques, avec ou sans paramètre typé, et les commandes asynchrones avec ou sans paramètre typé. Le choix d'un type de commande est donc assez simple, tout dépend si le code de la commande est une `Task` ou non et si elle utilise un paramètre ou non.

9.5 LA MESSAGERIE

La messagerie peut vite se transformer en enfer car les messages qui se « promènent » dans tous les sens *sont très difficiles à déboguer*. Pire que le code spaghetti, l'abus de messagerie crée un code spaghetti *invisible* ! Le ballet des messages est en effet insaisissable… Il est donc raisonnable de ne

pas abuser de cette facilité qu'on retrouve dans tous les toolkits MVVM. Et ce pour une bonne raison, c'est qu'une fois les réserves précédentes posées, la messagerie reste l'un des seuls moyens d'autoriser des classes à communiquer entre-elles sans se connaître et donc de garantir le *découplage fort propre à MVVM.*

Le Microsoft MVVM Toolkit va un cran plus loin en proposant deux messageries différentes. L'une optimisée pour éviter les fuites mémoires et l'autre optimisée pour la rapidité d'exécution. La première utilise des **`WeakReference`** alors que la seconde utilise le principe des *strong references*. On peut utiliser les deux messageries conjointement selon les besoins de l'application. Il est même possible d'utiliser plusieurs instances de ces messageries en même temps pour séparer des canaux de communication différents n'ayant aucune interaction. Sauf raisons solides à justifier je déconseille cet exercice de style dangereux…

Encore une fois sachez maîtriser vos ardeurs et je vous conseille de faire un choix clair entre l'une ou l'autre des messageries ou d'établir un véritable guide de conduite précisant les conditions ouvrant à l'utilisation de l'une ou de l'autre dans votre code. Sinon vous laissez à chaque développeur ce choix il est évident qu'aucun ne placera la barre à la même hauteur. Au final, un double code spaghetti virtuel, le pire du pire, le code spaghetti à la bolognaise virtuelle (avec de vrais morceaux de bogues fantomatiques dedans).

Une fois les précautions d'usage comprises, la messagerie n'est pas le diable non plus. Elle autorise des communications qui ne pourraient s'établir autrement en respectant MVVM. Ponctuellement elle a donc un rôle à jouer dans le film de votre application. Un indice toutefois : trop d'utilisation de la messagerie est le plus souvent le signe qu'il existe des défauts architecturaux graves ou des choix d'implémentation douteux dans l'application.

- **`IMessenger`**
  - C'est le contrat que toute messagerie doit supporter. Celles qui sont fournies le font bien entendu. Ce contrat stipule notamment comment les classes peuvent publier des messages ou s'abonner à la messagerie tout en s'ignorant (découplage). Vous pouvez, si cela se justifie, implémenter votre propre messagerie tant qu'elle supporte cette interface (le besoin semble plus hypothétique que réel il faut l'avouer).
- **`WeakReferenceMessenger`**
  - Cette implémentation de **`IMessenger`** est optimisée pour éviter les fuites mémoire et exploite en interne des référence faibles (*Weak References*). Il existe un léger surcoût en charge CPU pour le traitement des messages au profit d'une sécurisation contre les pertes mémoire. C'est en général la messagerie que vous utiliserez par défaut. Si vous respecter les avertissements plus haut, très peu de messages seront échangés et les millisecondes perdues dans une meilleure gestion de la mémoire seront vites rentabilisées.
- **`StrongReferenceMessenger`**
  - Cette autre implémentation de **`IMessenger`** fonctionne de façon opposée à la première : elle n'utilise que des références fortes (*strong references*). De telles références permettent une communication bien plus rapide (deux à trois fois) mais en créant des liens forts entre les classes participants au ballet des messages. Il faut absolument se désabonner « manuellement » au risque de fuites mémoire fatales à l'application. Il faut donc de très bonnes raisons et une bonne maitrise architecturale pour utiliser cette messagerie.
- **`IRecipient<TMessage>`**
  - Définit un contrat à respecter par les classes désirant s'abonner à un message particulier.
- **`MessageHandler<TRecipient,TMessage>`**

- Définit le type du délégué qui représente l'action à exécuter quand un message est reçu. Les exemples de code qui seront proposés dans une second temps permettront d'y voir plus clair, ne vous attardez pas trop sur ces définitions de types.

9.5.1 Les messages

La messagerie MVVM permet ainsi, à l'aide des outils présentés plus haut, de faire transiter des « messages » dans l'application. *Mais qu'est-ce qu'un message ? Comment est-il défini ?*

Le toolkit propose plusieurs classes ou définitions qui vont nous donner un bon aperçu de la nature des messages qui peuvent être utilisés et ce qu'on peut en faire. Pour ceux qui ne sont pas tout à fait à l'aise avec ces concepts MVVM, ne vous inquiétez pas, tout cela va prendre sens bien entendu dans les exemples à venir.

Sans trop spoiler la suite, on peut préciser que les outils de la messagerie proposent une méthode **`Send`** qu'on verra plus en détail dans les exemples. Cette méthode permet d'envoyer un message qui doit être de type **`TMessage`** comme on l'a vu dans la signature du **`MessageHandler`** ci-dessus par exemple. Le message lui-même peut être de toute nature, il s'agit toujours de transférer une ou plusieurs valeurs de type précis (les messages étant génériques).

Le Toolkit nous offre de base plusieurs types de message qui simplifient notre travail dans la majorité des cas usuels.

- **`PropertyChangedMessage<T>`**
  - Ce message permet de diffuser (*broadcast*) le changement d'état d'une propriété autrement que par un Binding. Souvent utilisé par des ViewModels pour se synchroniser entre eux sur des changements de valeur sans pour autant avoir à se « connaître » ou par les ViewModels et les Views pour échanger des informations sans violer le pattern MVVM. On retrouve ce message dans les *Setter* des propriétés le plus généralement. Les Modèles peuvent aussi tirer parti de ce mécanisme en prévenant des changements de leurs valeurs sans obliger qui que ce soit à écouter ses changements.
- **`RequestMessage<T>`**
  - Ce type de message est très utile car il permet de savoir si une réponse a été donnée et de prendre connaissance de celle-ci. La plupart des messages fonctionnent en mode « *Fire & Forget* » (selon le langage de l'aviation militaire à propos de certains missiles notamment), c'est-à-dire qu'on lance le message sans savoir s'il sera reçu, par qui, et surtout sans jamais obtenir de réponse (militairement cela permet de tirer et de se sauver le plus vite possible !). Les messageries les plus simples n'offrant pas l'équivalent du **`RequestMessage<T>`** obligent à utiliser deux messages différents ainsi que des variables d'état pour simuler le procédé de l'accusé de réception. Il s'agit donc ici d'un type de message précieux évitant beaucoup de code difficile à déboguer.
- **`AsyncRequestMessage<T>`**
  - Dans le même esprit et en se modernisant encore plus, le toolkit nous offre une variante du précédent type de message qui se base sur une réponse de type asynchrone. Plus subtile mais plus efficace ce type de message est à privilégier dès lors que la réponse à donner au message fait intervenir des I/O par exemple.
- **`CollectionRequestMessage<T>`**
  - Ce type de message permet tout simplement d'obtenir plus d'une réponse à la fois. Les réponses multiples étant transférées sous la forme de **`Collection`**.
- **`AsyncCollectionRequestMessage<T>`**
  - Comme on peut le supposer cette variante est à utiliser lorsque la liste des réponses à retourner à l'appelant dépend de traitements faisant intervenir des I/O (web, réseau, disques…) et ce afin de ne pas bloquer l'application.
- **`ValueChangedMessage<T>`**

- C'est un type de base qui permet d'indiquer qu'une « valeur » (quelle qu'elle soit) vient de changer. Tous les abonnés intéressés par ce changement seront ainsi prévenus.

9.6 L'INVERSION DE CONTROLE

Un modèle courant qui peut être utilisé pour augmenter la modularité dans la base de code d'une application utilisant le modèle MVVM consiste à utiliser l'inversion de contrôle sous une forme ou une autre. L'une des solutions les plus courantes consiste à utiliser l'injection de dépendances, qui consiste à créer un certain nombre de services qui sont injectés dans des classes backend (c'est-à-dire passées en paramètres aux constructeurs des ViewModel en général) - cela permet au code utilisant ces services de ne pas s'appuyer sur les détails d'implémentation de ces services, et il facilite également l'échange des implémentations concrètes de ces services. Ce modèle facilite également la mise à disposition de fonctionnalités spécifiques à la plate-forme pour le code backend, en les transformant en abstractions via un service qui est ensuite injecté là où cela est nécessaire. Une autre possibilité d'utilisation est le **Locator**, qui créer un point d'accès unique à l'ensemble des services qu'il va chercher dans le moteur d'IoC. Souvent plus pratique que le bricolage des constructeurs, les services s'utilisent alors naturellement depuis une classe Statique qui les expose à toute l'application. MVVM Light a fourni pendant longtemps une classe « **`ViewModelLocator`** » dans cet esprit.

Le Toolkit MVVM ne fournit pas d'API intégrées pour faciliter l'utilisation de ce modèle, car il existe déjà des bibliothèques dédiées spécifiquement pour cette tâche, tel que le paquet **`Microsoft.Extensions.DependencyInjection`**, qui fournit un ensemble d'API de DI complet et puissant, et agit comme un outil simple à configurer et qui utilise l'interface **`IServiceProvider`**.

On notera l'intelligence du Toolkit qui a évité à tout prix toute forme de dépendance forcée même pour l'IoC. A chacun de choisir son moteur préféré ou aucun. Même si les outils Microsoft sont préconisés, ils ne sont en aucun cas imposés. On retrouve cette même légèreté du toolkit dans son ignorance totale du système de navigation ce qui permet de l'adapter à toutes les situations à la différence de Prism par exemple.

10 EXEMPLES DE MISE EN ŒUVRE DU TOOLKIT

Cette partie reprend les principaux objets du Toolkit qui seront communément utilisés. D'autres objets existent, la plupart ont été présentés dans la première partie. Se référer à la doc Microsoft pour plus d'information à leur sujet. Ici je me limiterais aux types qui seront les plus utilisés – ou potentiellement les plus utilisés – notamment dans la transformation dite « Zéro Prism » de COBALT.

10.1 OBSERVABLEOBJECT

Classe de base pour les objets dont les propriétés doivent être observables (INPC).

ObservableObject est une classe de base pour les objets observables en implémentant les interfaces **`INotifyPropertyChanged`** et **`INotifyPropertyChanging`**. Il peut être utilisé comme point de départ pour toutes sortes d'objets qui doivent prendre en charge les notifications de modification de propriété.

`ObservableObject` principals fonctionnalités :

- Il fournit une implémentation de base pour **`INotifyPropertyChanged`** et **`INotifyPropertyChanging`**, exposant les événements **`PropertyChanged`** et **`PropertyChanging`**.
- Il fournit une série de méthodes **`SetProperty`** qui peuvent être utilisées pour définir facilement des valeurs de propriété à partir de types héritant de **`ObservableObject`**, et pour déclencher automatiquement les événements appropriés.
- Il fournit la méthode **`SetPropertyAndNotifyOnCompletion`**, qui est analogue à **`SetProperty`** mais avec la possibilité de définir les propriétés Task et de déclencher automatiquement les événements de notification lorsque les tâches assignées sont terminées.
- Il expose les méthodes **`OnPropertyChanged`** et **`OnPropertyChanging`**, qui peuvent être remplacées dans les types dérivés pour personnaliser la façon dont les événements de notification sont déclenchés.

10.1.1 Propriété simple

```
public class User : ObservableObject
{
    private string name;

    public string Name
    {
        get => name;
        set => SetProperty(ref name, value);
    }
}
```

La méthode **`SetProperty<T>(ref T, T, string)`** fournie vérifie la valeur actuelle de la propriété et la met à jour si elle est différente, puis déclenche automatiquement les événements pertinents. Le nom de la propriété est automatiquement capturé via l'utilisation de l'attribut **`[CallerMemberName]`**, il n'est donc pas nécessaire de spécifier manuellement quelle propriété est mise à jour.

10.1.2 Encapsuler un objet non observable

Un scénario courant, par exemple lorsque vous travaillez avec des éléments de base de données, consiste à créer un modèle d'encapsulation qui relaie les propriétés du modèle de base de données et déclenche les notifications de modification de propriété si nécessaire. Ceci est également nécessaire lorsque vous souhaitez injecter une prise en charge de notification aux modèles qui n'implémentent pas l'interface **`INotifyPropertyChanged`**. **`ObservableObject`** fournit une méthode dédiée pour rendre ce processus plus simple. Pour l'exemple suivant, **`User`** est un modèle mappant directement une table de base de données, sans hériter de **`ObservableObject`** :

```
public class ObservableUser : ObservableObject
{
    private readonly User user;

    public ObservableUser(User user) => this.user = user;

    public string Name
    {
        get => user.Name;
        set => SetProperty(user.Name, value, user, (u, n) => u.Name = n);
    }
}
```

Dans ce cas, nous utilisons la surcharge **`SetProperty<TModel, T>(T, T, TModel, Action<TModel, T>, string)`**. La signature est légèrement plus complexe que la précédente - cela est nécessaire pour que le code soit toujours extrêmement efficace même si nous n'avons pas accès à un champ de sauvegarde comme dans le scénario précédent. Nous pouvons parcourir chaque partie de cette signature de méthode en détail pour comprendre le rôle des différents composants:

- **`TModel`** est un argument de type, indiquant le type du modèle que nous enveloppons. Dans ce cas, il s'agira de notre classe **`User`**. Notez que nous n'avons pas besoin de le spécifier explicitement - le compilateur C# le déduira automatiquement par la façon dont nous appelons la méthode **`SetProperty`**.
- **`T`** est le type de propriété que nous voulons définir. Comme pour **`TModel`**, cela est déduit automatiquement.
- **`T oldValue`** est le premier paramètre, et dans ce cas, nous utilisons **`user. Nom`** pour transmettre la valeur actuelle de cette propriété que nous encapsulons.
- **`T newValue`** est la nouvelle valeur à définir sur la propriété, et ici nous passons **`value`**, qui est la valeur d'entrée dans le property setter.
- **`Model TModel`** est le modèle cible que nous enveloppons, dans ce cas, nous passons l'instance stockée dans le champ **`utilisateur`**.
- **`Action<TModel, T> callback`** est une fonction qui sera appelée si la nouvelle valeur de la propriété est différente de la valeur actuelle et que la propriété doit être définie. Cela se fera par cette fonction de rappel, qui reçoit en entrée le modèle cible et la nouvelle valeur de propriété à définir. Dans ce cas, nous attribuons simplement la valeur d'entrée (que nous avons appelée **`n`**) à la propriété **`Name`** (en faisant '**`u.Name = n`**'). Il est important ici d'éviter de capturer des valeurs de l'étendue actuelle et d'interagir uniquement avec celles données en entrée du rappel, car cela permet au compilateur C# de mettre en cache la fonction de rappel et d'effectuer un certain nombre d'améliorations de performances. C'est pour cette raison que nous n'accédons pas seulement directement au champ **`user`** ici ou au paramètre

value dans le setter, mais que nous n'utilisons que les paramètres d'entrée pour l'expression lambda.

La méthode **SetProperty<TModel, T>(T, TModel, Action<TModel, T>, string)** rend la création de ces propriétés d'encapsulation extrêmement simple, car elle prend en charge à la fois la récupération et la définition des propriétés cibles tout en fournissant une API extrêmement compacte.

10.1.3 Gérer les propriétés de type `Task<T>`

Si une propriété est une « tâche », il est nécessaire de déclencher également l'événement de notification une fois la tâche terminée, afin que les liaisons soient mises à jour au bon moment. Par exemple pour afficher un indicateur de chargement ou d'autres informations d'état sur l'opération représentée par la tâche. **ObservableObject** dispose d'une API pour ce scénario :

```
public class MyModel : ObservableObject
{
    private TaskNotifier<int>? requestTask;

    public Task<int>? RequestTask
    {
        get => requestTask;
        set => SetPropertyAndNotifyOnCompletion(ref requestTask, value);
    }

    public void RequestValue()
    {
        RequestTask = WebService.LoadMyValueAsync();
    }
}
```

Ici, la méthode **SetPropertyAndNotifyOnCompletion<T>(ref TaskNotifier<T>, Task<T>, string)** se chargera de mettre à jour le champ cible, de surveiller la nouvelle tâche, et de lever l'événement de notification lorsque cette tâche sera terminée. De cette façon, il est possible de simplement se lier à une propriété de tâche et d'être averti lorsque son état change. Le **TaskNotifier<T>** est un type spécial exposé par **ObservableObject** qui encapsule une instance **Task<T>** cible et active la logique de notification nécessaire pour cette méthode. Le type **TaskNotifier** est également disponible pour une utilisation directe si vous avez une **Task** générale uniquement.

10.2 ObservableRecipient

Le type **ObservableRecipient** est destiné à être utilisé comme base pour les ViewModels qui utilisent également les fonctionnalités **IMessenger**, car il fournit une prise en charge intégrée. En particulier:

- Il possède à la fois un constructeur sans paramètre et un constructeur qui prend une instance **IMessenger**, à utiliser avec l'injection de dépendance. Il expose également une propriété **Messenger** qui peut être utilisée pour envoyer et recevoir des messages dans le ViewModel. Si le constructeur sans paramètre est utilisé, l'instance **WeakReferenceMessenger.Default** est affectée à la propriété **Messenger**.
- Il expose une propriété **IsActive** pour activer/désactiver le ViewModel. Dans ce contexte, « activer » signifie qu'un ViewModel donné est marqué comme étant en cours d'utilisation,

de sorte que, par exemple. il commencera à écouter les messages enregistrés, effectuera d'autres opérations de configuration, etc. Il existe deux méthodes associées, **`OnActivated`** et **`OnDeactivated`**, qui sont appelées lorsque la propriété change de valeur. Par défaut, **`OnDeactivated`** annule automatiquement l'enregistrement de l'instance actuelle de tous les messages enregistrés. Pour de meilleurs résultats et éviter les fuites de mémoire, il est recommandé d'utiliser **`OnActivated`** pour enregistrer les messages et d'utiliser **`OnDeactivated`** pour effectuer des opérations de nettoyage. Ce modèle permet à un ViewModel d'être activé/désactivé plusieurs fois, tout en étant sûr de collecter ce qui doit l'être sans risque de fuites de mémoire chaque fois qu'il est désactivé. Par défaut, **`OnActived`** enregistre automatiquement tous les gestionnaires de messages définis via l'interface **`IRecipient<TMessage>`**.

- Il expose une méthode **`Broadcast<T>(T, T, string)`** qui envoie un message **`PropertyChangedMessage<T>`** via l'instance **`IMessenger`** disponible à partir de la propriété **`Messenger`**. Cela peut être utilisé pour diffuser facilement des modifications des propriétés d'un ViewModel sans avoir à récupérer manuellement une instance **`Messenger`** à utiliser. Cette méthode est utilisée par la surcharge des différentes méthodes **`SetProperty`**, qui ont une propriété de diffusion *bool* supplémentaire pour indiquer s'il faut ou non envoyer également un message.

Voici un exemple de ViewMode qui gère les messages de type (imaginaire) LoggedInUserRequestMessage :

```
public class MyViewModel : ObservableRecipient, IRecipient<LoggedInUserRequestMessage>
{
    public void Receive(LoggedInUserRequestMessage message)
    {
        // Handle the message here
    }
}
```

Dans l'exemple ci-dessus, **`OnActivated`** enregistre automatiquement l'instance en tant que destinataire pour les messages **`LoggedInUserRequestMessage`**, en utilisant cette méthode comme action à appeler. L'utilisation de l'interface **`IRecipient<TMessage>`** n'est pas obligatoire, et l'enregistrement peut également être effectué manuellement (même en utilisant uniquement une expression lambda en ligne) :

```
public class MyViewModel : ObservableRecipient
{
    protected override void OnActivated()
    {
        // Using a method group...
        Messenger.Register<MyViewModel, LoggedInUserRequestMessage>(this, (r, m) =>
r.Receive(m));

        // ...or a lambda expression
        Messenger.Register<MyViewModel, LoggedInUserRequestMessage>(this, (r, m) =>
        {
            // Handle the message here
        });
    }

    private void Receive(LoggedInUserRequestMessage message)
    {
        // Handle the message here
    }
}
```

10.3 ObservableValidator

ObservableValidator est une classe de base implémentant l'interface **INotifyDataErrorInfo**, qui prend en charge la validation des propriétés exposées à d'autres modules d'application. Il hérite également **d'ObservableObject**, il implémente donc **INotifyPropertyChanged** et **INotifyPropertyChanging**. Il peut être utilisé comme point de départ pour toutes sortes d'objets qui doivent prendre en charge à la fois les notifications de modification de propriété et la validation de propriété.

ObservableValidator possède les principales caractéristiques suivantes :

- Il fournit une implémentation de base pour **INotifyDataErrorInfo**, exposant l'événement **ErrorsChanged** et les autres API nécessaires.
- Il fournit une série de surcharges **SetProperty** supplémentaires (en plus de celles fournies par la classe **ObservableObject** de base), qui offrent la possibilité de valider automatiquement les propriétés et de lever les événements nécessaires avant de mettre à jour leurs valeurs.
- Il expose un certain nombre de surcharges **TrySetProperty**, qui sont similaires à **SetProperty** mais avec la possibilité de ne mettre à jour la propriété cible que si la validation réussit, et de renvoyer les erreurs générées (le cas échéant) pour une inspection plus approfondie.
- Il expose la méthode **ValidateProperty**, qui peut être utile pour déclencher manuellement la validation d'une propriété spécifique au cas où sa valeur n'a pas été mise à jour mais que sa validation dépend de la valeur d'une autre propriété qui a été mise à jour.
- Il expose la méthode **ValidateAllProperties**, qui exécute automatiquement la validation de toutes les propriétés d'instance publique dans l'instance actuelle, à condition qu'au moins un **[ValidationAttribute]** leur soit appliqué.
- Il expose une méthode **ClearAllErrors** qui peut être utile lors de la réinitialisation d'un modèle lié à un formulaire que l'utilisateur peut vouloir remplir à nouveau.
- Il propose un certain nombre de constructeurs qui permettent de passer différents paramètres pour initialiser l'instance **ValidationContext** qui sera utilisée pour valider les propriétés. Cela peut être particulièrement utile lors de l'utilisation d'attributs de validation personnalisés qui peuvent nécessiter des services ou des options supplémentaires pour fonctionner correctement.

Voici un exemple d'implémentation d'une propriété qui prend en charge à la fois les notifications de modification et la validation :

```
public class RegistrationForm : ObservableValidator
{
    private string name;

    [Required]
    [MinLength(2)]
    [MaxLength(100)]
    public string Name
    {
        get => name;
        set => SetProperty(ref name, value, true);
    }
}
```

Ici, nous appelons la méthode **SetProperty<T>(ref T, T, bool, string)** exposée par **ObservableValidator**, et ce paramètre *bool* supplémentaire défini sur **true** indique que nous voulons également valider la propriété lorsque sa valeur est mise à jour. **ObservableValidator** exécute automatiquement la validation sur chaque nouvelle valeur à l'aide de toutes les vérifications spécifiées avec les attributs appliqués à la propriété. D'autres composants (tels que les contrôles

d'interface utilisateur) peuvent ensuite interagir avec le ViewModel et modifier leur état pour refléter les erreurs actuellement présentes dans le ViewModel, en s'inscrivant dans ErrorsChanged et en utilisant la méthode **`GetErrors(string)`** pour récupérer la liste des erreurs pour chaque propriété qui a été modifiée.

10.3.1 Custom validation methods

Parfois, la validation d'une propriété nécessite qu'un ViewModel ait accès à des services, des données ou d'autres API supplémentaires. Il existe différentes façons d'ajouter une validation personnalisée à une propriété, en fonction du scénario et du niveau de flexibilité requis. Voici un exemple de la façon dont le type **`[CustomValidationAttribute]`** peut être utilisé pour indiquer qu'une méthode spécifique doit être appelée pour effectuer une validation supplémentaire d'une propriété :

```
public class RegistrationForm : ObservableValidator
{
    private readonly IFancyService service;

    public RegistrationForm(IFancyService service)
    {
        this.service = service;
    }

    private string name;

    [Required]
    [MinLength(2)]
    [MaxLength(100)]
    [CustomValidation(typeof(RegistrationForm), nameof(ValidateName))]
    public string Name
    {
        get => this.name;
        set => SetProperty(ref this.name, value, true);
    }

    public static ValidationResult ValidateName(string name, ValidationContext context)
    {
        RegistrationForm instance = (RegistrationForm)context.ObjectInstance;
        bool isValid = instance.service.Validate(name);

        if (isValid)
        {
            return ValidationResult.Success;
        }

        return new("The name was not validated by the fancy service");
    }
}
```

Dans ce cas, nous disposons d'une méthode **`ValidateName`** statique qui effectuera la validation sur la propriété **`Name`** via un service injecté dans notre ViewModel. Cette méthode reçoit la valeur de la propriété **`Name`** et l'instance **`ValidationContext`** en cours d'utilisation, qui contient des éléments tels que l'instance ViewModel, le nom de la propriété en cours de validation et, éventuellement, un fournisseur de services et des indicateurs personnalisés que nous pouvons utiliser ou définir. Dans ce cas, nous récupérons l'instance **`RegistrationForm`** à partir du contexte de validation, et à partir de là, nous utilisons le service injecté pour valider la propriété. Notez que cette validation sera exécutée à côté de celles spécifiées dans les autres attributs, nous sommes donc libres de combiner des méthodes de validation personnalisées et des attributs de validation existants comme bon nous semble.

10.3.2 Custom validation attributes

Une autre façon d'effectuer une validation personnalisée consiste à implémenter un **`[ValidationAttribute]`** personnalisé. puis insérer la logique de validation dans la méthode

IsValid remplacée. Cela permet une flexibilité supplémentaire par rapport à l'approche décrite ci-dessus, car il est très facile de réutiliser le même attribut à plusieurs endroits.

Supposons que nous voulions valider une propriété en fonction de sa valeur relative par rapport à une autre propriété dans le même modèle de vue. La première étape consisterait à définir une **[GreaterThanAttribute]** personnalisée, comme :

```
public sealed class GreaterThanAttribute : ValidationAttribute
{
    public GreaterThanAttribute(string propertyName)
    {
        PropertyName = propertyName;
    }

    public string PropertyName { get; }

    protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
    {
        object
            instance = validationContext.ObjectInstance,
            otherValue = instance.GetType().GetProperty(PropertyName).GetValue(instance);

        if (((IComparable)value).CompareTo(otherValue) > 0)
        {
            return ValidationResult.Success;
        }

        return new("The current value is smaller than the other one");
    }
}
```

Ensuite, nous pouvons ajouter cet attribut dans notre ViewModel:

```
public class ComparableModel : ObservableValidator
{
    private int a;

    [Range(10, 100)]
    [GreaterThan(nameof(B))]
    public int A
    {
        get => this.a;
        set => SetProperty(ref this.a, value, true);
    }

    private int b;

    [Range(20, 80)]
    public int B
    {
        get => this.b;
        set
        {
            SetProperty(ref this.b, value, true);
            ValidateProperty(A, nameof(A));
        }
    }
}
```

Dans ce cas, nous avons deux propriétés numériques qui doivent être dans une plage spécifique et avec une relation spécifique entre elles (A doit être supérieur à B). Nous avons ajouté le nouveau **[GreaterThanAttribute]** sur la première propriété, et nous avons également ajouté un appel à **ValidateProperty** dans le setter pour B, de sorte que A est validé à nouveau chaque fois que B

change (puisque son état de validation en dépend). Nous avons juste besoin de ces deux lignes de code dans notre ViewModel pour activer cette validation personnalisée, et nous avons également l'avantage d'avoir un attribut de validation personnalisé réutilisable qui pourrait également être utile dans d'autres ViewModels de notre application. Cette approche facilite également la modularisation du code, car la logique de validation est désormais complètement découplée de la définition du ViewModel lui-même.

10.4 COMMANDES

10.4.1 RelayCommand et RelayCommant<T>

`RelayCommand` et **`RelayCommand<T>`** ont les principales caractéristiques suivantes :

- Ils fournissent une implémentation de base de l'interface **`ICommand`**.
- Ils implémentent également l'interface **`IRelayCommand`** (et **`IRelayCommand<T>`**), qui expose une méthode **`NotifyCanExecuteChanged`** pour déclencher l'événement **`CanExecuteChanged`**.
- Ils exposent des constructeurs prenant des délégués comme **`Action`** et **`Func<T>`**, qui permettent l'encapsulation de méthodes standard et d'expressions lambda.

Utilisation d'une commande simple :

```
public class MyViewModel : ObservableObject
{
    public MyViewModel()
    {
        IncrementCounterCommand = new RelayCommand(IncrementCounter);
    }

    private int counter;

    public int Counter
    {
        get => counter;
        private set => SetProperty(ref counter, value);
    }

    public ICommand IncrementCounterCommand { get; }

    private void IncrementCounter() => Counter++;
}
```

Et voici l'UI XAML qui pourrait être utilisatrice de ce code :

```
<Page
    x:Class="MyApp.Views.MyPage"
    xmlns:viewModels="using:MyApp.ViewModels">
    <Page.DataContext>
        <viewModels:MyViewModel x:Name="ViewModel"/>
    </Page.DataContext>

    <StackPanel Spacing="8">
        <TextBlock Text="{x:Bind ViewModel.Counter, Mode=OneWay}"/>
        <Button
            Content="Click me!"
            Command="{x:Bind ViewModel.IncrementCounterCommand}"/>
    </StackPanel>
</Page>
```

Le Button se lie à **`ICommand`** dans le ViewModel, qui encapsule la méthode privée **`IncrementCounter`**. Le **`TextBlock`** affiche la valeur de la propriété **`Counter`** et est mis à jour chaque fois que la valeur de la propriété change.

10.4.2 AsyncRelayCommand and AsyncRelayCommand<T>

AsyncRelayCommand et **AsyncRelayCommand<T>** sont des implémentations **ICommand** qui étendent les fonctionnalités offertes par **RelayCommand**, avec prise en charge des opérations asynchrones.

AsyncRelayCommand et **AsyncRelayCommand<T>** ont les principales fonctionnalités suivantes :

- Ils étendent les fonctionnalités des commandes synchrones incluses dans la bibliothèque, avec la prise en charge des délégués de retour de tâche (**Task**).
- Ils peuvent encapsuler des fonctions asynchrones avec un paramètre **CancellationToken** supplémentaire pour prendre en charge l'annulation, et ils exposent des propriétés **CanBeCanceled** et **IsCancellationRequested**, ainsi qu'une méthode **Cancel**.
- Ils exposent une propriété **ExecutionTask** qui peut être utilisée pour surveiller la progression d'une opération en attente et un **IsRunning** qui peut être utilisé pour vérifier la fin d'une opération. Ceci est particulièrement utile pour lier une commande à des éléments d'interface utilisateur tels que des indicateurs de chargement.
- Ils implémentent les interfaces **IAsyncRelayCommand** et **IAsyncRelayCommand<T>**, ce qui signifie que le ViewModel peut facilement exposer des commandes à l'aide de celles-ci pour réduire le couplage étroit entre les types. Par exemple, cela facilite le remplacement d'une commande par une implémentation personnalisée exposant la même surface d'API publique, si nécessaire.

Utilisation de commandes asynchrones

Imaginons un scénario similaire à celui décrit dans l'exemple **RelayCommand**, mais une commande exécutant une opération asynchrone :

```
public class MyViewModel : ObservableObject
{
    public MyViewModel()
    {
        DownloadTextCommand = new AsyncRelayCommand(DownloadText);
    }

    public IAsyncRelayCommand DownloadTextCommand { get; }

    private Task<string> DownloadText()
    {
        return WebService.LoadMyTextAsync();
    }
}
```

Le code d'UI serait le suivant :

```
<Page
    x:Class="MyApp.Views.MyPage"
    xmlns:viewModels="using:MyApp.ViewModels"
    xmlns:converters="using:Microsoft.Toolkit.Uwp.UI.Converters">
    <Page.DataContext>
        <viewModels:MyViewModel x:Name="ViewModel"/>
    </Page.DataContext>
    <Page.Resources>
        <converters:TaskResultConverter x:Key="TaskResultConverter"/>
    </Page.Resources>

    <StackPanel Spacing="8" xml:space="default">
        <TextBlock>
            <Run Text="Task status:"/>
```

```
            <Run Text="{x:Bind ViewModel.DownloadTextCommand.ExecutionTask.Status,
Mode=OneWay}"/>
            <LineBreak/>
            <Run Text="Result:"/>
            <Run Text="{x:Bind ViewModel.DownloadTextCommand.ExecutionTask,
Converter={StaticResource TaskResultConverter}, Mode=OneWay}"/>
        </TextBlock>
        <Button
            Content="Click me!"
            Command="{x:Bind ViewModel.DownloadTextCommand}"/>
        <ProgressRing
            HorizontalAlignment="Left"
            IsActive="{x:Bind ViewModel.DownloadTextCommand.IsRunning, Mode=OneWay}"/>
    </StackPanel>
</Page>
```

Lorsque vous cliquez sur le bouton, la commande est appelée et la tâche d'exécution mise à jour. Une fois l'opération terminée, la propriété déclenche une notification qui est reflétée dans l'interface utilisateur. Dans ce cas, l'état de la tâche et le résultat actuel de la tâche sont affichés. Notez que pour afficher le résultat de la tâche, il est nécessaire d'utiliser la méthode **`TaskExtensions.GetResultOrDefault`** - cela permet d'accéder au résultat d'une tâche qui n'est pas encore terminée sans bloquer le thread (et éventuellement provoquer un blocage).

10.5 MESSAGERIE

L'interface **`IMessenger`** est un contrat pour les types qui peuvent être utilisés pour échanger des messages entre différents objets. Cela peut être utile pour découpler différents modules d'une application sans avoir à conserver des références fortes aux types référencés. Il est également possible d'envoyer des messages à des canaux spécifiques, identifiés de manière unique par un jeton, et d'avoir différents messagers dans différentes sections d'une application. MVVM Toolkit fournit deux implémentations prêts à l'emploi : **`WeakReferenceMessenger`** et **`StrongReferenceMessenger`** : le premier utilise des références faibles en interne, offrant une gestion automatique de la mémoire pour les destinataires, tandis que le second utilise des références fortes et oblige les développeurs à désabonner manuellement leurs destinataires lorsqu'ils ne sont plus nécessaires (vous trouverez plus de détails sur la façon de désinscrire les gestionnaires de messages ci-dessous), mais en échange de cela offre de meilleures performances et beaucoup moins d'utilisation de la mémoire.

> *Attention : Étant donné que le type **`WeakReferenceMessenger`** est plus simple à utiliser et correspond au comportement du type **`Messenger`** de la bibliothèque **MvvmLight**, il s'agit du type par défaut utilisé par le type **`ObservableRecipient`** dans MVVM Toolkit. Le **`StrongReferenceType`** peut toujours être utilisé en passant une instance au constructeur de cette classe.*

Les types implémentant **`IMessenger`** sont responsables de la maintenance des liens entre les destinataires (destinataires de messages) et leurs types de messages enregistrés, avec des gestionnaires de messages relatifs. Tout objet peut être enregistré en tant que destinataire pour un type de message donné à l'aide d'un gestionnaire de messages, qui sera appelé chaque fois que l'instance **`IMessenger`** est utilisée pour envoyer un message de ce type. Il est également possible d'envoyer des messages via des canaux de communication spécifiques (chacun identifié par un jeton unique), de sorte que plusieurs modules puissent échanger des messages du même type sans provoquer de conflits. Les messages envoyés sans jeton utilisent le canal partagé par défaut.

Il existe deux façons d'effectuer l'enregistrement des messages : soit via l'interface **`IRecipient<TMessage>`**, soit à l'aide d'un délégué **`MessageHandler<TRecipient, TMessage>`** agissant en tant que gestionnaire de messages. Le premier vous permet d'enregistrer tous les gestionnaires avec un seul appel à l'extension **`RegisterAll`**, qui enregistre

automatiquement les destinataires de tous les gestionnaires de messages déclarés, tandis que le second est utile lorsque vous avez besoin de plus de flexibilité ou lorsque vous souhaitez utiliser une expression lambda simple comme gestionnaire de messages.

`WeakReferenceMessenger` et **`StrongReferenceMessenger`** exposent également une propriété **`Default`** qui offre une implémentation *thread-safe* intégrée dans le package. Il est également possible de créer plusieurs instances de messagerie si nécessaire, par exemple si une autre est injectée avec un fournisseur de services DI dans un module différent de l'application (par exemple, plusieurs fenêtres s'exécutant dans le même processus). Ce type d'utilisation réclame une organisation planifiée de la messagerie stricte, documentée et pensée. Dans le cas contraire le logiciel risque de ne même pas pouvoir être débogué !

10.5.1 Envoi et réception de messages

Partons des éléments suivants pour construire un exemple simple :

```
// Create a message
public class LoggedInUserChangedMessage : ValueChangedMessage<User>
{
    public LoggedInUserChangedMessage(User user) : base(user)
    {
    }
}

// Register a message in some module
WeakReferenceMessenger.Default.Register<LoggedInUserChangedMessage>(this, (r, m) =>
{
    // Handle the message here, with r being the recipient and m being the
    // input message. Using the recipient passed as input makes it so that
    // the lambda expression doesn't capture "this", improving performance.
});

// Send a message from some other module
WeakReferenceMessenger.Default.Send(new LoggedInUserChangedMessage(user));
```

Imaginons que ce type de message soit utilisé dans une application de messagerie simple, qui affiche un en-tête avec le nom d'utilisateur et l'image de profil de l'utilisateur actuellement connecté, un panneau avec une liste de conversations et un autre panneau avec des messages de la conversation en cours, si une est sélectionnée. Supposons que ces trois sections sont prises en charge par les types **`HeaderViewModel`**, **`ConversationsListViewModel`** et **`ConversationViewModel`** respectivement. Dans ce scénario, le message **`LoggedInUserChangedMessage`** peut être envoyé par **`HeaderViewModel`** une fois l'opération de connexion terminée, et les deux autres ViewModels peuvent inscrire des gestionnaires pour celui-ci. Par exemple, **`ConversationsListViewModel`** chargera la liste des conversations pour le nouvel utilisateur, et **`ConversationViewModel`** fermera simplement la conversation en cours, le cas échéant.

L'instance **`IMessenger`** se charge de remettre les messages à tous les destinataires enregistrés. Notez qu'un destinataire peut s'abonner à des messages d'un type spécifique. Notez que les types de messages hérités ne sont pas enregistrés dans les implémentations **`IMessenger`** par défaut fournies par MVVM Toolkit. Il est donc nécessaire de les traiter comme si cet héritage n'existait pas.

Lorsqu'un destinataire n'est plus nécessaire, vous devez le désenregistrer afin qu'il cesse de recevoir des messages. Vous pouvez annuler l'inscription par type de message, par jeton d'enregistrement ou par destinataire :

```
// Unregisters the recipient from a message type
WeakReferenceMessenger.Default.Unregister<LoggedInUserChangedMessage>(this);

// Unregisters the recipient from a message type in a specified channel
```

```
WeakReferenceMessenger.Default.Unregister<LoggedInUserChangedMessage, int>(this, 42);

// Unregister the recipient from all messages, across all channels
WeakReferenceMessenger.Default.UnregisterAll(this);
```

> *Attention : Comme mentionné précédemment, le désabonnement n'est pas strictement nécessaire lors de l'utilisation du type* **`WeakReferenceMessenger`**, *car il utilise des références faibles pour suivre les destinataires, ce qui signifie que les destinataires inutilisés seront toujours éligibles pour le garbage collector même s'ils ont encore des gestionnaires de messages actifs. Il est malgré tout toujours bon de les désabonner, au moins pour améliorer les performances et faire en sorte qu'en cas de changement pour une messagerie à références fortes cela ne cause pas de fuite de mémoire. D'autre part, l'implémentation de* **`StrongReferenceMessenger`** *utilise des références fortes pour suivre les destinataires enregistrés. Ceci est fait pour des raisons de performances, et cela signifie que chaque destinataire enregistré doit être désinscrit manuellement pour éviter les fuites de mémoire. Autrement dit, tant qu'un destinataire est enregistré, l'instance* **`StrongReferenceMessenger`** *utilisée conservera une référence active à celle-ci, ce qui empêchera le garbage collector de pouvoir collecter cette instance. Vous pouvez soit gérer cela manuellement, soit hériter d'***`ObservableRecipient`**, qui par défaut se charge automatiquement de supprimer tous les enregistrements de messages pour le destinataire lorsqu'il est désactivé (voir la section sur* **`ObservableRecipient`** *pour plus d'informations à ce sujet).*

Il est également possible d'utiliser l'interface **`IRecipient<TMessage>`** pour enregistrer les gestionnaires de messages. Dans ce cas, chaque destinataire devra implémenter l'interface pour un type de message donné et fournir une méthode **`Receive(TMessage)`** qui sera appelée lors de la réception de messages, comme ceci :

```
// Create a message
public class MyRecipient : IRecipient<LoggedInUserChangedMessage>
{
    public void Receive(LoggedInUserChangedMessage message)
    {
        // Handle the message here...
    }
}

// Register that specific message...
WeakReferenceMessenger.Default.Register<LoggedInUserChangedMessage>(this);

// ...or alternatively, register all declared handlers
WeakReferenceMessenger.Default.RegisterAll(this);

// Send a message from some other module
WeakReferenceMessenger.Default.Send(new LoggedInUserChangedMessage(user));
```

10.5.2 Utilisation des messages de demande

Une autre caractéristique utile des instances de messagerie est qu'elles peuvent également être utilisées pour demander des valeurs d'un module à un autre. Pour ce faire, le package inclut une classe **`RequestMessage<T>`** de base, qui peut être utilisée comme suit :

```
// Create a message
public class LoggedInUserRequestMessage : RequestMessage<User>
{
}

// Register the receiver in a module
WeakReferenceMessenger.Default.Register<MyViewModel, LoggedInUserRequestMessage>(this, (r,
m) =>
{
```

```
    // Assume that "CurrentUser" is a private member in our viewmodel.
    // As before, we're accessing it through the recipient passed as
    // input to the handler, to avoid capturing "this" in the delegate.
    m.Reply(r.CurrentUser);
});

// Request the value from another module
User user = WeakReferenceMessenger.Default.Send<LoggedInUserRequestMessage>();
```

La classe **`RequestMessage<T>`** inclut un convertisseur implicite qui permet la conversion d'un Objet **`LoggedInUserRequestMessage`** vers son objet **`User`** contenu. Cela vérifiera également qu'une réponse a été reçue pour le message et lèvera une exception si ce n'est pas le cas. Il est également possible d'envoyer des messages de demande sans cette garantie de réponse obligatoire : il suffit de stocker le message renvoyé dans une variable locale, puis de vérifier manuellement si une valeur de réponse est disponible ou non. Cela ne déclenchera pas l'exception automatique si aucune réponse n'est reçue lors du retour de la méthode **`Send`**.

Le même espace de noms inclut également un message de demandes de base pour d'autres scénarios : **`AsyncRequestMessage<T>`**, **`CollectionRequestMessage<T>`** et **`AsyncCollectionRequestMessage<T>`**. Voici comment utiliser un message de demande asynchrone :

```
// Create a message
public class LoggedInUserRequestMessage : AsyncRequestMessage<User>
{
}

// Register the receiver in a module
WeakReferenceMessenger.Default.Register<MyViewModel, LoggedInUserRequestMessage>(this, (r,
m) =>
{
    m.Reply(r.GetCurrentUserAsync()); // We're replying with a Task<User>
});

// Request the value from another module (we can directly await on the request)
User user = await WeakReferenceMessenger.Default.Send<LoggedInUserRequestMessage>();
```

10.6 IoC

Comme déjà indiqué dans la première partie de ce document le MVVM Toolkit ne fournit pas de moteur IoC. C'est au développeur de choisir celui qu'il préfère, de le mettre en route (paramétrage) et de l'utiliser à sa guise. Il n'y a de ce fait aucun couplage avec un toolkit externe.

Mais l'utilisation de la DI ou d'un Locator ou autre émanation de l'IoC reste très simple comme je vais le montrer. Rappelons qu'ici je vais utiliser **`Microsoft.Extensions.DependencyInjection`** qui fournit un moteur de DI avec une API complète et qui expose l'interface **`IServiceProvider`**. Il est tout à fait possible de choisir un autre toolkit, celui-ci a l'avantage de la cohérence « Microsoft » avec le Toolkit MVVM, ce qui est subjectif et non technique.

Un dernier rappel : je vais ici montrer comment mettre en œuvre et utiliser simplement l'IoC avec le MVVM Toolkit, pour plus d'information sur le package utilisé se référer à la documentation Microsoft spécifique à celui-ci.

10.6.1 Configurer et résoudre les services

La première étape consiste à déclarer une instance **`IServiceProvider`** et à initialiser tous les services nécessaires, généralement au démarrage. Par exemple, sur UWP (mais une configuration similaire peut également être utilisée sur d'autres frameworks, notamment les Xamarin.Forms) :

```
public sealed partial class App : Application
```

```
{
    public App()
    {
        Services = ConfigureServices();

        this.InitializeComponent();
    }

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

    /// <summary>
    /// Gets the <see cref="IServiceProvider"/> instance to resolve application services.
    /// </summary>
    public IServiceProvider Services { get; }

    /// <summary>
    /// Configures the services for the application.
    /// </summary>
    private static IServiceProvider ConfigureServices()
    {
        var services = new ServiceCollection();

        services.AddSingleton<IFilesService, FilesService>();
        services.AddSingleton<ISettingsService, SettingsService>();
        services.AddSingleton<IClipboardService, ClipboardService>();
        services.AddSingleton<IShareService, ShareService>();
        services.AddSingleton<IEmailService, EmailService>();

        return services.BuildServiceProvider();
    }
}
```

Ici, la propriété **`Services`** est initialisée au démarrage et tous les services de l'application et les ViewModels sont enregistrés. Il existe également une nouvelle propriété **`Current`** qui peut être utilisée pour accéder facilement à la propriété **`Services`** à partir d'autres vues de l'application. Par exemple :

```
IFilesService filesService = App.Current.Services.GetService<IFilesService>();

// Use the files service here...
```

L'aspect clé ici est que chaque service peut très bien utiliser des API spécifiques à la plate-forme, mais comme celles-ci sont toutes abstraites via l'interface que notre code utilise, nous n'avons pas à nous soucier d'elles chaque fois que nous résolvons simplement une instance et l'utilisons pour effectuer des opérations.

10.6.2 Injection dans le constructeur

Une fonctionnalité puissante disponible est « l'injection dans le constructeur », ce qui signifie que le fournisseur de services DI est capable de résoudre automatiquement les dépendances indirectes entre les services inscrits lors de la création d'instances du type demandé. Considérez le service suivant :

```
public class FileLogger : IFileLogger
{
    private readonly IFilesService FileService;
    private readonly IConsoleService ConsoleService;

    public FileLogger(
        IFilesService fileService,
```

```
        IConsoleService consoleService)
    {
        FileService = fileService;
        ConsoleService = consoleService;
    }

    // Methods for the IFileLogger interface here...
}
```

Ici, nous avons un type **`FileLogger`** implémentant l'interface **`IFileLogger`** et nécessitant des instances **`IFilesService`** et **`IConsoleService`**. L'injection dans le constructeur signifie que le fournisseur de services DI rassemblera automatiquement tous les services nécessaires :

```
/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    services.AddSingleton<IFilesService, FilesService>();
    services.AddSingleton<IConsoleService, ConsoleService>();
    services.AddSingleton<IFileLogger, FileLogger>();

    return services.BuildServiceProvider();
}

// Retrieve a logger service with constructor injection
IFileLogger fileLogger = App.Current.Services.GetService<IFileLogger>();
```

Le fournisseur de services DI vérifiera automatiquement si tous les services nécessaires sont enregistrés, puis il les récupérera et appellera le constructeur pour le type concret **`IFileLogger`** enregistré, pour obtenir l'instance en retour.

10.6.3 Qu'en est-il des ViewModels ?

Un fournisseur de services a « service » dans son nom, mais il peut en fait être utilisé pour résoudre des instances de n'importe quelle classe, y compris les ViewModels ! Les mêmes concepts expliqués ci-dessus s'appliquent toujours, y compris l'injection dans le constructeur. Imaginez que nous ayons un type **`ContactsViewModel`**, utilisant un **`IContactsService`** et une instance **`IPhoneService`** via son constructeur. Nous pourrions avoir une méthode **`ConfigureServices`** comme celle-ci :

```
/// <summary>
/// Configures the services for the application.
/// </summary>
private static IServiceProvider ConfigureServices()
{
    var services = new ServiceCollection();

    // Services
    services.AddSingleton<IContactsService, ContactsService>();
    services.AddSingleton<IPhoneService, PhoneService>();

    // Viewmodels
    services.AddTransient<ContactsViewModel>();

    return services.BuildServiceProvider();
}
```

Et puis dans notre **`ContactsView`**, nous attribuerions le contexte de données comme suit :

```
Contacts publicsView()
{
    ceci. InitializeComponent();
```

```
    ceci. DataContext = App.Current.Services.GetService<ContactsViewModel>();
}
```