Notes sur le module : *C# Avancé.*

Armel Pitelet

13 février 2022

Table des matières

1 Code Synchrone et Asynchrone

Remarque : *Sur les fenêtre wpf.*
Il y a un processus qui tourne sur le Thread UI qui s'occupe de gérer la partie visible d'une fenêtre.

Remarque : *Pour faire de l'asynchrone.*
On peut passer par la classe Thread (depreciated) qui est la pour des raisons historique pour déclarer une action à faire dans un nouveaux Thread.

Mot clef : **async**.
Authorise l'utilisation du **await**. Voir https://docs.microsoft.com/fr-fr/dotnet/csharp/programming-guide/concepts/async/.

Mot clef : **Task**.
Une task représente une action à effectuer de manière asynchrone (en générale mais pas obligatoirement). On ne peut pas mettre à jour un autre Thread que celui en cours dans une Task. (voir remarque sur le `Dispacher`).

Mot clef : **await**.
Tant que la ligne marquée await n'a pas finie son éxcécution, on attent avant de passer à la suite de la fonction.

Note :

On peut afficher les Threads en cours dans visual studio.

Remarque : *Sur la classe* ***Dispacher***.

Elle permet de faire discuter les Threads entre eux via `Dispacher.Invoke(une lambda qui fait un truc)`.

Note :

Si on a le choix, c'est plus performant d'utiliser de l'asynchrone que du synchrone.

Remarque : *Pour faire des fonctions asynchrone.*

Il faut utiliser le mot clef `async` et renvoyer void ou bien renvoyer une `Task<UnType>`. Il vaut mieux renvoyer une Task car elle à des propriétés que l'on peut vouloir récupérer (comme la Task est elle toujours en cours d'exécution ou pas). Le `return` de la fonction n'a cependant pas à renvoyer lui même une `Task` (à partir du moment ou on a un `await` quelque part dans la fonction).

Remarque : *Sur `Task.FromResult()`.*

Permet de créer une `Task` à partir de ce que l'on met dans le `Result(un_truc)`. Lorsque l'évènement est levé, la méthode abonner est appelée. On se désabonne ensuite via un $Event- = UnTruc$.

Remarque : *Sur la notion d'Asynchrone.*

Faire de l'asynchrone avec `async await` revient à écrire du code de manière synchrone (le code après un await est excécuté aprés la fin de la Task marqué en await) mais ce qui est éxcécuté en asynchrone est exécutée sur un Thread à part qui ne bloque pas l'exécution des autres Threads en cours.

Note :

On peut rendre une tache synchrone en utilisant `Task.GetAwaiter().GetResult()`. C'est complètement inutile sauf si on ne peut absolument pas renvoyer une `Task` (comme sur du code Legacy par exemple)

Remarque : *Sur la gestion de plusieurs Task en même temps.*

Task.WhenAll(task1,task2), WaitAll(task1,task2) permettent d'attendre que les Task données soient terminées. `WhenAny` permet d'éxcécuter ce qui suit une fois qu'au moins une des Task est terminée

Remarque : *Divers trucs sur les Tasks en générale.*

`Task.FromResult(résultat)` permet de renvoyer un résultat (synchrone ou non). Est équivalent à passer par le constructeur de Task (`new Task()`). `Task.FromException(new Exception())` permet de créer une Task à partir d'une exception.

Remarque : *Sur la gestion des exceptions dans les Task.*

Les exceptions (pasque c'est bien fait) peuvent remonter dans le Thread principale. A partir du moment ou la Task est asynchrone et lève une exception tt se passe (pour nous) comme si on était en synchrone, il n'y a pas de précautions particulières à prendre.

Remarque : *Sur le type d'exception qui remonte des Task.*

Ces Exceptions sont typé de la même manière que les classiques.

Remarque : *Sur l'`AggregateException`.*

Type particulier d'exception qui peut popper dans certain contexts (ex : Parallel) à la place de l'excpetion que l'on attend. C'est une exception qui aggrège (wrap) un ensemble d'exceptions et qui

contient d'autres exceptions. Ce type d'Exception arrive quand on est dans d'autres Thread que le principale. N'arrive pas si l'on fait bien les async await mais peut arriver facilement dans les systèmes multi-threadés et/ou parallélisés.

2 Les Events

Remarque : *Sur les Events.*
Dans visual Studio un symbol d'éclaire à coté d'un nom de variable imndique un évènement. Un évènement se déclare avec le mot clef `event` puis un type délégué Nom de l'event. Lorsque l'on fait un `Evenement += UnTruc`. On abonne la méthode `UnTruc` à l'évènement `Evenement`. La desinscription est obligatoire à un moment lors du programme. Attention, si l'on passe des fonction anonymes, il n'existe aucun moyen de se désabonner !

Remarque : *Sur les `Loaded` et `Unloaded` dans WPF.*
Il n'est pas nécessaire de se désabonner à ces évènement la puisqu'ilms sont détruits en même temps que la fenêtre.

Note :
En Angular, le système d'abonnement est identique. Cas particulier, les requêtes Http ne nécessite pas de désabonnement.

Note :
En générale on passe plus par du `async await` que du `EventHandler` pour faire de l'asynchrone. Mais c'est encore très présent en Wpf.

Remarque : *Sur le passage d'argument aux Events.*
Il est nécessaire de créer des classes qui hérite de `EventArgs` pour passer des arguments aux évènements.

Remarque : *Sur le `InitializeComponent` en WPF.*
Cette commande vient en fait transformer le xaml en une vrai classe .cs (d'ou le `partial` devant les deux types de classes)

Note :
Les EventQQChose sont en faite des classes de **délégué**

Note :
Les `Event` sont des propriétés sans Getter Setter !

Remarque : *Sur les méthodes qui s'abonne aux event.*
Il vaut mieux toujours vérifier que l'event existe (c'est à dire qu'il ait au moins un abonnement $\rightarrow$ d'où les `if(MonEvent!= null)`).

3 Le chemin critique : notion de threads partagés

Remarque : *Sur la notion de thread-safe.*
Une méthode que l'on ne peut pas partager entre plusieurs thread, ou plus simplement une ressource, est dites non thread-safe

Remarque : *Sur les sémaphores.*
C'est un type d'outil qui permet de rendre thread safe une ressource. Une autre des manière de faire (qui n'est pas un sémaphore) est d'utiliser des lockers (de type **object**). Une autre est de passer par la class **SemaphoreSlim**.

Mot clef : **lock**.
Tant que l'objet contenue dans le lock(monObjet) est maintenue, on ne peut plus accéder au block couvert par le lock. Cela crée ce que l'on appel un **chemin critique**. Le lock met en attente automatiquement les thread qui essaie d'accéder à la ressource si un thread occupe déjà ce block. Attention le lock ne fait que du verrou séquentielle (les threads le passeront un par par un uniquement).

Remarque : *Sur les environnements multi-threadés.*
Il est très conseiller d'utiliser une classe dédié à la gestion des threads lorsque l'on est dans un environnement gérant plusieurs threads. Il est bien d'isoler le code qui doit tourner dans un thread dans cette classe.

Remarque : *Sur la notion de watch-dog.*
C'est un système de surveillance qui track et détecte les changement. Le `dotnet watch` est un exemple d'un tel système.

Remarque : *Sur le **SemaphoreSlime**.*
Permet de filtrer l'accès à un trhead mais par paquet de X (SémaphoreSlim(X)) que l'on peut définir (on est plus limité à de l'envoie un par un mais on peut quand même limiter le nombre de thread qui travail simultanément au même endroit). Attention il faut Wait puis Release le sémaphore. Cela peut être critique si il y a une erreur entre le Wait et le Release ! (Si l'on veut cacth l'erreur il faut absoluement que le release du sémaphore soit dans un finaly).

Remarque : *Sur le **System.Collection.Concurrent**.*
Fournit une alternative aux classes de base non thread safe (comme la List) pour pouvoir les utiliser dans un environnement multithreadé (ce n'est pas vraiment). Si l'on utilise pas ce namespace, les Threads peuvent s'exécuter un peu dans n'importe quelle ordre, et un Thread peut prendre plus de tache que d'autre. Concurrent règle ce problème de *cannibalisme* entre Threads.

Remarque : *Le HttpClient et les accès base de données sont ThreadSafe..*

4 Collection avancée

Remarque : *Sur le **Stack**.*
Liste de type LIFO (Last In, First Out).

Remarque : *Sur le **Queue**.*
Liste de type FIFO (First in Fisrt Out).

Remarque : *Sur le **Lazy**.*
Permet de faire du thread safe de manière un peu magique. → A creuser. Ne crée une instance d'un objet qu'au tt dernier moment (à la première initialisation).

5 La “containerisation” des applis

Ou comment déployer dans un environnement spécifique.

Note :
On passe par **Docker**.

Remarque : *Sur la notion de container.*
Docker permet de lancer des *containers*. Le principe du container est d'avoir un environnement maitrisé (maitrisé ce qu'il y a dans l'environnement très exactement). Docker permet d'héberger des environnement différents de celui de la mchine *host*.

Note :
Lorsqu'on lance un projet Visual Studio on peut lui indiquer d'utiliser un container docker. Cela ajoute un fichier `Dockerfile`. A partir de ce fichier on peut installer via Docker des images d'environnement déjà préparé (On peut faire tourner du Dotnet En Docker sans avoir à installer dotnet directement sur la machine). Voix **DockerHub** pour une liste de ce qui existe déjà

Note :
Le gros intérêt de docker est de ne plus avoir aucune dépendance sur la machine locale et de tt déporter dans le conteneur docker lui même qui simule autant d'environnement différents que l'on veut. On peut voir les container comme de petites machines virtuelles light (mais ce ne sont pas des machines virtuelles !). C'est plus proche des navigateur qui alloue à chaque onglet un processus distinct séparé des autres.

Note :
On peut lancer plusieurs containers docker depuis la même machine.

Remarque : *Sur les images.*
Les images crées par docker sont conservées en local (après le téléchargement initial). On peut après sa réutiliser la même image dans plusieurs container différents.

Remarque : *Sur la communication inter-container.*
On peut indiquer à docker un port pour la communication à l'extérieur. Cela permet par exemple d'avoir une base de données sur un conteneur, et une ou plusieurs API dans d'autres conteneur qui se connecte à cette base. Pour faire ce gerne de chose on passe par un fichier Dockercompose (un .yml) qui configure ces connexions.

Remarque : *Pour créer un network.*
On passe par **docker-compose** up.

Note :
La bonne pratique est d'avoir un conteneur par composants (dans le même sens que les diagrammes de composants).

Remarque : *Sur l'alias latest.*
Attention à ne pas utiliser cette alias car il a le sens de “dernière version” dans docker (c'est un alias qui représente la dernière version disponible, qui peut changer d'une machine à l'autre !).

Remarque : *Sur les alternatives à Docker.*

Il y a **kubernetes** ou bien **tanzu**.

Remarque : *Pour ce mettre à docker.*
Go to https://docs.docker.com/get-started/.

6 Collections avancées

Ces différentes collections avancées répondent surtout à des problèmes de performances spécifiques. Il faut les choisir en fonciton des besoins. Liens principaux **System.Collection.Generic** et **System.Collection**.

Mot clef : **SortedList**.
Liste trié non typé. Ce n'est pas tt à fait une Liste mais plus un dictionnaire car système de clef valeur.Non typée mais on le déclare quand même avec un type <type_clef, type_valeur>, ou bien on peut juste ne pas la typer mais c'est prendre le risque de ne plus savoir ce que l'on a dedans. On peut récupérer une valeur par l'indexation entre crochet. Voir https://www.tutorialsteacher.com/csharp/csharp-sortedlist, https://www.geeksforgeeks.org/c-sharp-sortedlist-with-examples/ et https://fr.acervolima.com/c-sortedlist-avec-des-exemples/. Particularité, ce genre de collection réarange les valeurs en triant par clef dans un ordre croissant. On peut récupérer les valeur via `MaList[UneClef]` ou bien via `MaList.ElementAt(un_index)`

Note :
Attention à la différence entre **SortedList** (de System.Collection) et **SortedList<T1,T2>** (de SYstem.Collection.Generic).

Remarque : *Sur la différence entre dictionnaire et orted List.*
Le dictionnaire est plus performant en écriture alors que la list est plus rapide en lecture (car elle est trié). Le gros intêret de la SOrtedList c'est l'accès par index plus que l'accès par clef. Voir la note remaqrue suivante sur les performances.

Remarque : *Sur le **SortedDictionnary**.*
Dans l'utilisation on en fait la même chose. Ce qui change c'est les performances en fonction de l'utilisation. Voir https://stackoverflow.com/questions/935621/whats-the-difference-between-sortedlist-and-sor

Mot clef : **LinkedList**.
Liste doublement chainée dont le principale interet est le fait que les opérations d'insertion et suppresion sont en O(1). Liste double chainée au sens des pointeurs. N'a pas d'indexeurs mais un système de noeuds. Voir https://stackoverflow.com/questions/169973/when-should-i-use-a-list-vs-a-linkedlist concernant l'intrêt et les performances de ce genre de trucs.

Mot clef : **HashSet**.
Equivalent du Set classiqe aka une collection sans répétitions. Surtout utilisé pour de la comparaisons d'élément (de Set à Set). Le hashSet se base sur le hash de la valeur plutôt que sur la valeur en elle même. Il existe aussi le **SortedSet** qui maintiens un ordre en plus d'une uncicité (surtout utile car est thread safe par rapport au HashSet qui semble ne pas l'être). C'est Set sont surtout utiles et performants sur les opérations de comparaison comme, union, exclusion. Un peu plus de doc sur les HashSet https://www.c-sharpcorner.com/article/working-with-hashset-in-c-sharp/. Attention lors de l'utilisation d'objet, si l'on veut garantr l'unicité il faut se basé sur **IEquatable** (override du Equals) et **GetHashCode** pour faire de la comparaison. En théorie le HashCode est suffisant pour

les opérations sur les Set, le IEquatable est une bonne pratique qui permet de faire des comparaison non basé sur la référence d'un objet dans d'autre collections.

Remarque : *Sur l'interface* ***IEquatableComparer***.
C'est une interface qui force l'implémentation du Equals(object) **et** GetHashCode.

Remarque : *Sur la méthode* ***Equals*** *et l'override du* ==.
Pour override un opréteur, on fait comme en c++, et il faut que cette dernière renvoie un résultat basée sur l'implem du Equals.

Note :
Des liens sur la surcharge et l'override des opérateurs : https://docs.microsoft.com/fr-fr/dotnet/csharp/language-reference/operators/operator-overloading, https://www.tutorialspoint.com/csharp/csharp_operator_overloading.htm. Commencer par https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/operator-overloading#overloadable-operators.

Note :
Pour revoir le GetHashCode on peut passer par la structure **HashCode** qui fournit des méthodes de génération que l'on peut utiliser pour définir comment est calculé le hash d'un objet.

7 Les Métadonnées

Sur les métadonnées https://docs.microsoft.com/en-us/dotnet/standard/metadata-and-self-describing-comp
Pour la suite voir le schéma sur 08/02/22 1.
PLus de liens : https://docs.microsoft.com/fr-fr/dotnet/api/system.componentmodel.dataannotations.metadatatypeattribute?view=net-6.0 (doc microsoft), https://www.section.io/engineering-education/working-with-metadata-in-csharp/ (un article qui résume un peu).

Remarque : *Sur les* ***Assemblies***.
Regroupe toute les collections, version et ressources (toutes les données) que l'on utilise dans un projet.Le runtime prend en entrée une assembly pour la passer au *Common Language Runtime* (CLR). Dans visual studio : Affichage : Divers on peut avoir tt les infos sur l'assembly.

Remarque : *Sur les* ***dll***.
Est utilisé par une assembly comme une ressource au RunTime. C'est différent d'un .exe qui lui sera un véritable programme qui utilisera (voir embarquera) les dll pendant le runtime.

Définition : **Common Intermediate Language** .
Ou CIL est le language de compilation avant le CLR. C'est ce qui va utiliser les Métadonnées pour décrire les types, classes, ... et les passer au CLR.
Remarque : *Sur les* ***Métadonnée***.
Ce sont les *données des données*. On peut les retrouver dans l'assembly.

Note :
Lorsque l'on regarde dans les propréiéts d'un projet ou d'une solution on est en fait dans les Métadonnées de l'assembly/des assemblies.

Remarque : *Sur les* ***Attributs***.
Voir le guide https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/

pour utiliser les attributs et même en créer de custom. Voir le namespace **System.Attribute**.

Remarque : *POur faire fonctionner les Tests Unitaire avec des classes internal.*

On passe par l'attribut **friend** ! Il faut marquer la classe internal et référencer la friend assembly. Cette attribut se met au dessus du namespace. Une autre manière de faire est de passer par le csproj (dans lequel se trouve l'internal). Dans la section <ItemGroup>... il faut faire un <AssemblyAttribute Include>... avec Include = "System.Runtime.CompilerService.InternalToVisibleTo" et lui donner un <_ParameterX>MonProjetAmis</_ParameterX> (X est un nombre à donner et le underscore est complètement optionnel, c'est une simple convention). Cela permet de rendre l'entièretés des classes internal du projet visible par le projet Amis.

Remarque : *Sur la **réflexion**.*

C'est basé sur les métadonnées des classes. Principalement on utilise le `MonInstance.GetType().GetProp` ou bien `typeof(MaClasse).GetProperties()`pour récupérer les propriétées associées à une classe. Cela permet d'accéder aux champs privées, de bouclé sur tt les propriétées sans les nommer. Attention la réflexion est un mécanisme relativement lent. Pour plus de Doc : https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/reflection et https://docs.microsoft.com/fr-fr/dotnet/api/system.reflection.parameterinfo?view=net-6.0 pour quelques exemple. Sinon de manière générale voir tt la doc sur le namespace **System.Reflection**.

Remarque : *Sur les **Attibuts** en api.*

Voir aussi de manière générale la doc sur le namespace **System.componentModel.DataAnntotations**.

Note :

Voir ce genre de **truc** pour convertir du json en classe C# et inversement.

Note :

Sur une piste pour régler le problème du IngredientController : passer par les models binders https://docs.microsoft.com/en-us/aspnet/core/mvc/advanced/custom-model-binding?view=aspnetcore-6.0.

8 Divers

Remarque : *De la doc sur Docker et les API.*

https://github.com/dotnet/dotnet-docker/blob/main/samples/run-aspnetcore-https-development.md (repo .NET) et la doc associée https://docs.microsoft.com/fr-fr/aspnet/core/host-and-deploy/docker/building-net-docker-images?view=aspnetcore-6.0 (Doc Microsoft). On peut partir sur ce git pour s'entrainer à utiliser docker.

Remarque : *Sur la manière de deckerisez le programme.*

Soit on sépare la base (dans son environnement avec le sdk et mssql) et le prgramme de l'api dans leur propre container docker. A ce moment la il faudra les faire communiquer entre eux vai les port des container. Soit on lance la base en local et on fait communiquer l'api en contectant son conteneur avec le localhost.

Note :

On peut aussi faire de la communication avec le net (web) via docker.

Remarque : *Un peu plus de doc sur docker.*

Comment créer du multi container via visual studio : https://docs.microsoft.com/fr-fr/visualstudio/containers/tutorial-multicontainer?view=vs-2022. Comment lancer une base de donnée dans son propre conteneur : https://docs.docker.com/samples/aspnet-mssql-compose/.

Note :

Lorsque l'on crée un Projet VS on peut lui indiquer de prendre en compte Docker. Cela va en fait créer un Dockerfile adapté. Sur un pojet en cours il *suffit* de rajouter un Dockerfile adapté. Il faut alors faire attention aux noms.

Note :

Sur la construction d'un Dockerfile : https://devopssec.fr/article/creer-ses-propres-images-docker-dockerfile (blog) ou bien la doc officiel : https://docs.docker.com/engine/reference/builder/#usage.