

# Notes sur le module : *C# Avancé.*

Armel Pitelet

4 février 2022

## Table des matières

<b>1</b>	<b>Code Synchrone et Asynchrone</b>	<b>1</b>
<b>2</b>	<b>Les Events</b>	<b>3</b>
<b>3</b>	<b>Le chemin critique : notion de threads partagés</b>	<b>3</b>
<b>4</b>	<b>Collection avancée</b>	<b>4</b>
<b>5</b>	<b>La “containerisation” des applis</b>	<b>5</b>
<b>6</b>	<b>Divers</b>	<b>6</b>

## 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 Dispatcher).

Mot clef : **await**.

Tant que la ligne marquée await n'a pas finie son exécution, on attend avant de passer à la suite de la fonction.

Note :

On peut afficher les Threads en cours dans visual studio.

Remarque : Sur la classe **Dispatcher**.

Elle permet de faire discuter les Threads entre eux via `Dispatcher.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 a 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 où 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 `EventUnTruc`.

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 exécuté après la fin de la `Task` marqué en `await`) mais ce qui est exécuté en asynchrone est exécuté sur un `Thread` à part qui ne bloque pas l'exécution des autres `Threads` en cours.

Note :

On peut rendre une tâche 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'exécuter 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. À partir du moment où 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'exception 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 indique un évènement. Un évènement se déclare avec le mot clef `event` puis un type délégué Nom de l'événement. Lorsque l'on fait un `Evenement += UnTruc`. On abonne la méthode `UnTruc` à l'évènement `Evenement`. La désinscription 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 évènements la puisqu'ils 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 vraie classe .cs (d'où le `partial` devant les deux types de classes)

Note :

Les `EventArgs` sont en fait des classes de **délégué**

Note :

Les `EventArgs` sont des propriétés sans Getter Setter !

Remarque : *Sur les méthodes qui s'abonne aux event.*

Il vaut mieux toujours vérifier que l'événement existe (c'est à dire qu'il ait au moins un abonnement → 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 dite non thread-safe

Remarque : *Sur les sémaphores.*

C'est un type d'outil qui permet de rendre thread safe une ressource. Une autre des manières 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 appelle 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 un par un uniquement).

Remarque : *Sur les environnements multi-threadés.*

Il est très conseillé d'utiliser une classe dédiée 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 changements. Le dotnet watch est un exemple d'un tel système.

Remarque : *Sur le **SemaphoreSlim**.*

Permet de filtrer l'accès à un thread mais par paquet de  $X$  ( $\text{SemaphoreSlim}(X)$ ) que l'on peut définir (on est plus limité à l'envoi un par un mais on peut quand même limiter le nombre de thread qui travaillent 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 catch l'erreur il faut absolument que le release du sémaphore soit dans un finally).

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 quel ordre, et un Thread peut prendre plus de tâche 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 First 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 tout 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 maîtrisé (maîtrisé ce qu’il y a dans l’environnement très exactement). Docker permet d’héberger des environnement différents de celui de la machine *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). Voir **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 tout 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 créées 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 genre de chose on passe par un fichier `Dockercompose` (un `.yaml`) 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 Divers