WPF : ASYNC ET await POUR DISPATCHER LES MESSAGES

Auteur : Hassan KANDIL

Date de publication :11/02/2008

Revisitée : 01/12/2016

Il est courant dans des applications n-tier d'avoir des appels SQL qui occupent le processeur et qui donnent l'impression d'un plantage de l'application où aucun message de paint ou même de click ne sera traité à temps. Et parfois un calcul assez prenant pourrait avoir le même effet.

Prenons l'exemple suivant qui occupe l'application et qui bizarrement ne fait pas ce que l'on lui demande de faire.

MainWindow.XAML

```
<Window x:Class="Dispatch.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Dispatch"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Button Content="Button" HorizontalAlignment="Left" Margin="120,65,0,0" VerticalAlignment="Top"
Width="110" Height="42" Click="Button_Click"/>
        <TextBlock Name="tbExemple" HorizontalAlignment="Left" Margin="180,190,0,0" TextWrapping="Wrap"
Text="TextBlock" VerticalAlignment="Top" Height="30" Width="160"/>

    </Grid>
</Window>
```

Code behind

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.ComponentModel;
using System.Threading;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Messaging;
using System.IO;

namespace Dispatch
{
    /// <summary>
    /// Logique d'interaction pour MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
```

```
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private  void Button_Click(object sender, RoutedEventArgs e)
        {
            Compteur();
        }
        public  void Compteur()
        {
            long m = 0;
            for(long i=0;i<100000000;i++)
            {
                tbExemple.Text = m++.ToString();
            }
        }
   }
}
```

Lancer le programme, cliquez sur Button et vous verrez immédiatement un écran figé, il est même impossible de bouger la fenêtre ou de la fermer avant la fin de la boucle. la valeur du compteur s'affiche une seule fois à la fin.

Plusieurs méthodes permettent de régler ce problème et de rendre fluide une boucle aussi prenante que celle là.

L'appel à la DLL user32

```
        [DllImport("user32.dll")]
        static extern sbyte GetMessage(ref long lpMsg, IntPtr hWnd, uint wMsgFilterMin, uint
wMsgFilterMax);
        [DllImport("user32.dll")]
        static extern bool TranslateMessage( ref long lpMsg);
        [DllImport("user32.dll")]
        static extern IntPtr DispatchMessage( ref long lpmsg);
        static long msg = 0;

        public MainWindow()
        {
            InitializeComponent();
        }
        public void MessageHandler()
        {
            GetMessage(ref msg, IntPtr.Zero, 0, 0);
            TranslateMessage(ref msg);
            DispatchMessage(ref msg);
        }

        private  void Button_Click(object sender, RoutedEventArgs e)
        {
             Compteur();
        }

       public  void  Compteur()
        {
            long m = 0;
            for(long i=0;i<100000000;i++)
            {
                tbExemple.Text = m++.ToString();
                MessageHandler();

            }
```

l'utilisation de la vieille méthode GetMessage, Translate et Dispatch pourrait remédier à cette situation, oublie de traiter le message de fin ce qui impose que l'on le traite dans la méthode MessageHandler.

Remarquez quand même la définition de message comme un long (de toute façon ce n'est qu'un pointeur qui contiendra une adresse transmise comme telle aux fonctions Translate et dispatch).

Ceci permettra à notre application de respirer, et de rester vivante ...

Mais il y aura toujours des petits effets de bord avec une telle solution, car WPF n'est pas fait pour se baser sur la bibliothèque user32.

D'où une autre solution qui se dégage de async et await (conseillés avec WPF)

async et await

Le compteur doit être défini en async .. Dans ce cas on peut le lancer avec BeginInvoke et l'arrêter avec EndInvoke, sauf que ces fonctions ne permettent aucune action sur des objets graphiques propriétés des autres. On aura donc des difficultés, surmontables à afficher dans tbExemple.

La méthode la plus simple c'est de faire appel à await Task.delay(10) qui ralentira légèrement l'exécution mais donnera la vie à l'application.

```
using System;
using System.Threading.Tasks;
using System.Windows;
using System.Diagnostics;
using System.Runtime.InteropServices;
using System.Messaging;
using System.IO;

namespace Dispatch
{
    /// <summary>
    /// Logique d'interaction pour MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
        private  void Button_Click(object sender, RoutedEventArgs e)
        {
            Compteur();
        }
        public async  void  Compteur()
        {
            long m = 0;
            for(long i=0;i<100000000;i++)
            {
                tbExemple.Text = m++.ToString();
                await Task.Delay(10);
            }
        }
    }
}
```

Cette méthode courte pourrait être une solution à moindre coût.

D'autres méthodes seront exposées dans d'autres articles.