Livrable 3

EasySave 3.0

Chef de groupe : Milan Sangare

Eugénie Hariniaina

Alexandre De Jesus Correia

Pierre Grossin

Date : 13/12/2022

Ecole d'ingénieur du CESI

Etudiant en Première année du cycle ingénieur informatique

93 Bd de la Seine, 92000 Nanterre

CESI
ÉCOLE D'INGÉNIEURS

Sommaire

Introduction

Après notre version 2.0, l'entreprise nous demande de faire évoluer le logiciel. Plusieurs évolutions sont suggérées : la parallélisation des travaux de sauvegarde, permettre à l'utilisateur de créer une priorité de fichiers à sauvegarder, éviter la saturation de la bande passante (définir une limite), rendre l'application mono-instance.

Sauvegarde en parallèle

Dans les trois dernières versions de notre projet, le type de fonctionnement de la sauvegarde était soit en mono sauvegarde soit en séquentielle.

Pour cette version 3.0, le client a exigé que ces deux modes de sauvegarde soient annulés. La sauvegarde en parallèle sera donc effective dans cette version.

Qu'est-ce qu'une sauvegarde en parallèle ?

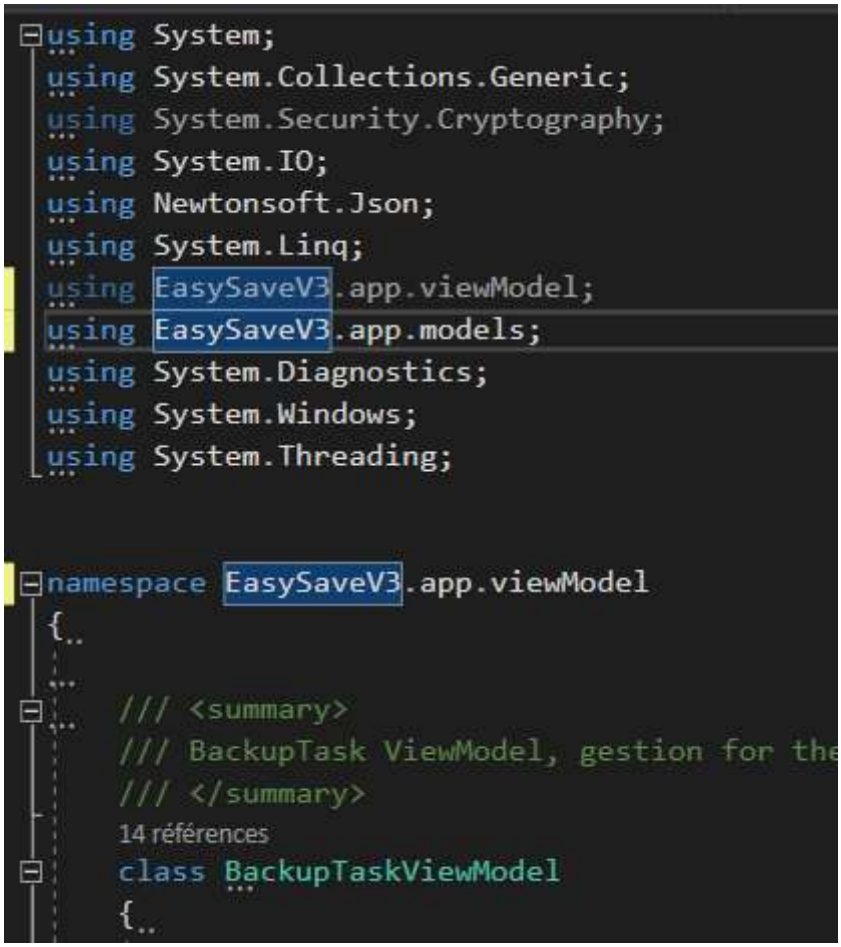
Contrairement au mode séquentiel, le mode de sauvegarde en parallèle permettra à un utilisateur de faire plusieurs sauvegardes en même temps et donc de gagner du temps.
En C#, la programmation d'un tel système est l'utilisation du multithreading.

Qu'est ce que le multithreading ?

Le multithreading en C# est un processus dans lequel plusieurs threads fonctionnent simultanément. C'est un processus pour atteindre le multitâche. Cela permet de gagner du temps car plusieurs tâches sont exécutées à la fois. Pour créer une application multithread en C#, nous devons utiliser le namespace **System.Threading**.

```
/// <summary>
/// function to start the save in multithreading
/// </summary>
/// <param name="backupTask"> (BackupTask) BackupTask to start </param>
/// <param name="backupsJsonMutex"> (Mutex) mutex to use</param>
///
2 références
public static void startThreadBackupTask(BackupTask backupTask,Mutex backupsJsonMutex)
{
    Trace.WriteLine("startThreadBackupTask : " + backupTask.name);

    processStartBackup bt = new processStartBackup(backupTask, backupsJsonMutex);
    System.Threading.Thread thrd = new System.Threading.Thread(bt.startBackupTask);
    thrd.Start();
}
```

En appelant ce namespace, tous les backups, donc sauvegardes lancées le seront en parallèle.
Cette méthode a été créée dans la partie ViewModel de notre architecture qui appelle les 2 autres qui sont View et Model :



Gestion des fichiers prioritaires

Pour cette dernière version, on souhaiterait pouvoir sauvegarder certains types de fichiers en priorité. Pour cela, le plus efficace semble être de faire une liste des extensions des fichiers que l'on souhaite sauvegarder en priorité :


```
<ListBox x:Name="ChoixPrio" Width="200" HorizontalAlignment="Left" BorderBrush="Black"
    BorderThickness="0,0,1,1" Margin="314,88,0,363" SelectionMode="Extended"
    Background="White" AutomationProperties.ItemType="string">
    <ListBox.Resources>
        <Style TargetType="{x:Type Border}">
            <Setter Property="CornerRadius" Value="10"/>
        </Style>
    </ListBox.Resources>
</ListBox>
```

De plus, pour pouvoir ajouter ou enlever des extensions de noms de fichiers dans cette liste, on peut rajouter les blocs suivants :

La méthode suivante permet d'ajouter une extension prioritaire :

```
private void AddChoixPrio_Click(object sender, RoutedEventArgs e)
{

    PriofExt = ChoixPrio.Items.Cast<string>().ToList();
    PriofExt.Add(InputPrio.Text);
    ChoixPrio.ItemsSource = PriofExt;
    InputPrio.Text = ".";

}
```

Cette méthode permet d'en supprimer une :

```
private void SupChoixPrio_Click(object sender, RoutedEventArgs e)
{
    if (ChoixPrio.SelectedItems.Count == 1)
    {
        PriofExt = ChoixPrio.Items.Cast<string>().ToList();
        PriofExt.Remove(ChoixPrio.SelectedItem.ToString());
        ChoixPrio.ItemsSource = PriofExt;
    }
}
```

On sauvegardera les éléments ajoutés avec la méthode AddChoixPrio_Click en les mettant dans la newList PriofExt :

```
List<string> PriofExt = new List<string>();
```

Pour permettre la mise en pause des fichiers non-prioritaires par rapport aux autres, on utilise la méthode SaveConfig_Click_2 qui se déclenche lorsqu'on clique sur le bouton "Save configuration" :

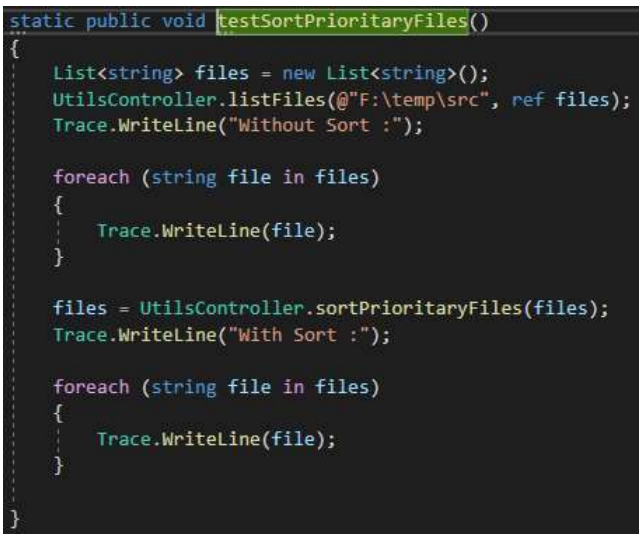
```
private void SaveConfig_Click_2(object sender, RoutedEventArgs e)
{
    if (String.IsNullOrEmpty(choice_key.Text) || String.IsNullOrEmpty(workJob.Text) || String.IsNullOrEmpty(numberThreads.Text) || String.IsNullOrEmpty(maxSizeFile.Text))
    {

        string message = EasySaveV2.app.models.Language.getInstance().getSlug("slugEmptyError");
        string title = EasySaveV2.app.models.Language.getInstance().getSlug("slugError");
        MessageBox.Show(message, title);
    }
    else
    {
        chifExt = ChoixCryptage.Items.Cast<string>().ToList();
        PriofExt = ChoixPrio.Items.Cast<string>().ToList();
        UtilsController.writeConfFile(chifExt, PriofExt, workJob.Text, Convert.ToInt32(maxSizeFile.Text), Convert.ToInt32(numberThreads.Text), choice_key.Text, Convert.ToInt32(blockSize.Text));
    }
}
```

La liste PriofExt est alors enregistrée avec la méthode writeConfFile et mise dans la List extensionPriority :

```
public List<string> extensionPriority { get; set; }
```

Ensuite, si la liste n'est pas vide, la méthode sortPrioritaryFiles est activée :

```
if (config.extensionPriority != null)
{
    files = UtilsController.sortPrioritaryFiles(files);
}
```



```
static public void testSortPrioritaryFiles()
{
    List<string> files = new List<string>();
    UtilsController.listFiles(@"F:\temp\src", ref files);
    Trace.WriteLine("Without Sort :");

    foreach (string file in files)
    {
        Trace.WriteLine(file);
    }

    files = UtilsController.sortPrioritaryFiles(files);
    Trace.WriteLine("With Sort :");

    foreach (string file in files)
    {
        Trace.WriteLine(file);
    }

}
```

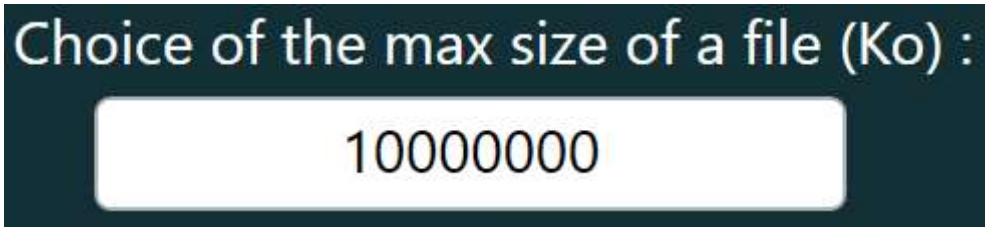
Cette méthode permet alors de sauvegarder d'abord les fichiers ayant des extensions contenues dans la List extensionPriority.

Interdiction de transfert en simultané de fichiers trop volumineux

On souhaiterait maintenant pouvoir interdire automatiquement le transfert simultané de fichiers ayant une taille supérieure à nKo, cette limite pouvant être modifiée dans les paramètres.

On commence donc par créer dans le fichier Setting.xaml un TextBox permettant de modifier la valeur de taille limite :

```
<TextBox Name="maxSizeFile" FontSize="20" FontFamily="helvetica" HorizontalAlignment="Left" Margin="628,64,0,0"
         PreviewTextInput="NumberValidationTextBox" Text="1000" TextWrapping="Wrap" VerticalAlignment="Top"
         Width="250" Height="38" TextAlignment="Center" VerticalContentAlignment="Center">
    <TextBox.Resources>
        <Style TargetType="{x:Type Border}">
            <Setter Property="CornerRadius" Value="5"/>
        </Style>
    </TextBox.Resources>
</TextBox>
```



La valeur entrée dans le TextBox est alors contenue dans la variable maxSizeFile.Text qui, lors de la sauvegarde des paramètres, est convertie en valeur entière à 32 bits grâce à la méthode Convert.ToInt32:

```
maxSizeFile.Text = readConf.maxSize.ToString();
```

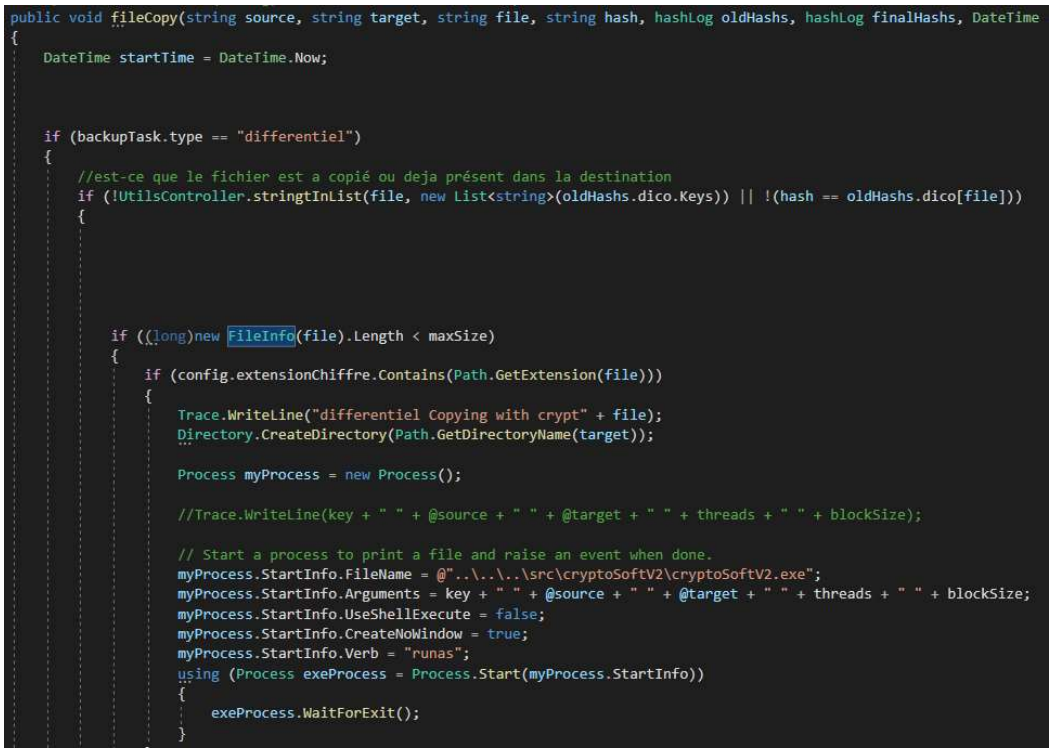
```
UtilsController.writeConfFile(chifExt, PriofExt, workJob.Text, Convert.ToInt32(maxSizeFile.Text),
```

Ensuite, à l'aide de la méthode writeconfFile, la valeur numérique est mise dans la variable maxSize:

```
public static void writeConfFile(List<string> extensionsChiffre, List<string> extensionsPrio, string process, long maxSize, int threads, string key, long blockSize)
{
    configFile config = new configFile();
    config.extensionChiffre = extensionsChiffre;
    config.process = process;
    config.extensionPriority = extensionsPrio;
    config.maxSize = maxSize;
    config.threads = threads;
    config.blockSize = blockSize;
    config.key = key;
    var json = JsonConvert.SerializeObject(config);
    File.WriteAllText("../../../src/config/config.json", json);
}
```

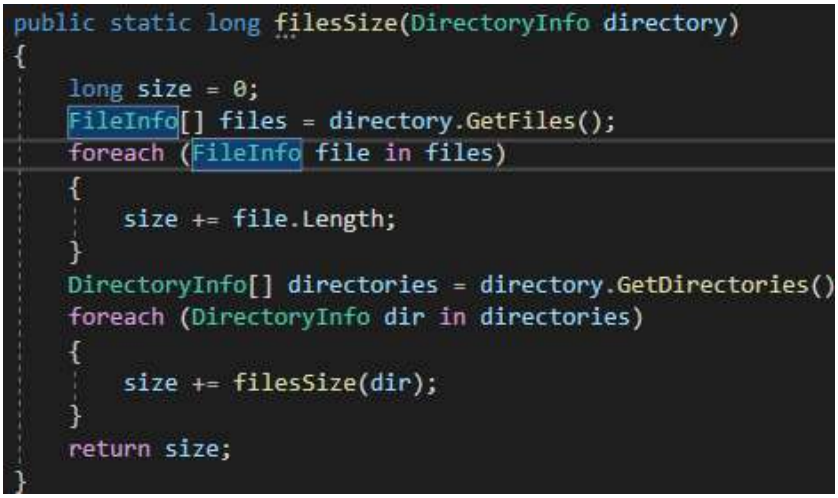
```
private long maxSize = config.maxSize;
```

Cette dernière est alors utilisée lors de la méthode FileCopy qui réalise les sauvegardes avec le processus exeProcess.WaitForExit uniquement lorsque la somme de la taille des fichiers est inférieur à la taille maximale autorisée (maxSize) :



```
public void fileCopy(string source, string target, string file, string hash, hashLog oldHashs, hashLog finalHashs, DateTime
{
    DateTime startTime = DateTime.Now;


    if (backupTask.type == "differentiel")
    {
        //est-ce que le fichier est a copié ou deja présent dans la destination
        if (!UtilsController.stringtInList(file, new List<string>(oldHashs.dico.Keys)) || !(hash == oldHashs.dico[file]))
        {




            if ((long)new FileInfo(file).Length < maxSize)
            {
                if (config.extensionChiffre.Contains(Path.GetExtension(file)))
                {
                    Trace.WriteLine("differentiel Copying with crypt" + file);
                    Directory.CreateDirectory(Path.GetDirectoryName(target));

                    Process myProcess = new Process();

                    //Trace.WriteLine(key + " " + @source + " " + @target + " " + threads + " " + blockSize);

                    // Start a process to print a file and raise an event when done.
                    myProcess.StartInfo.FileName = @"..\..\..\src\cryptoSoftV2\cryptoSoftV2.exe";
                    myProcess.StartInfo.Arguments = key + " " + @source + " " + @target + " " + threads + " " + blockSize;
                    myProcess.StartInfo.UseShellExecute = false;
                    myProcess.StartInfo.CreateNoWindow = true;
                    myProcess.StartInfo.Verb = "runas";
                    using (Process exeProcess = Process.Start(myProcess.StartInfo))
                    {
                        exeProcess.WaitForExit();
                    }
```

La somme de la taille des fichiers est calculée à l'aide de la méthode filesSize qui incrémente la variable size dans FileInfo. La taille de chaque fichiers est ici récupérée avec la propriété String.Lenght qui récupère le nombre de caractère d'un fichier (ici, file), la valeur de longueur renvoyée étant de type Int32, d'où l'utilisation de la méthode Convert.ToInt32 précédemment :



```
public static long filesSize(DirectoryInfo directory)
{
    long size = 0;
    FileInfo[] files = directory.GetFiles();
    foreach (FileInfo file in files)
    {
        size += file.Length;
    }
    DirectoryInfo[] directories = directory.GetDirectories()
    foreach (DirectoryInfo dir in directories)
    {
        size += filesSize(dir);
    }
    return size;
}
```

Interaction en temps réel avec chaque travail ou l'ensemble des travaux

Pour chaque travail de sauvegarde (ou l'ensemble des travaux), l'utilisateur doit pouvoir :

- Mettre sur pause : le logiciel doit afficher un bouton pour permettre une pause effective après le transfert du fichier en cours

```
/// <summary>
/// function that stop a backupTask
/// </summary>
/// <param name="backupTask"> (BackupTask) BackupTask to stop </param>
0 références
public static void stopBackupTask(SaveTask backupTask)
{

    backupTask.progress.state = "PAUSE";
    SaveTaskViewModel.updateBackupTask(backupTask.uuid,backupTask.name,backupTask

}
```

- Mettre sur Play : le logiciel doit afficher un bouton de démarrage ou reprise d'une pause

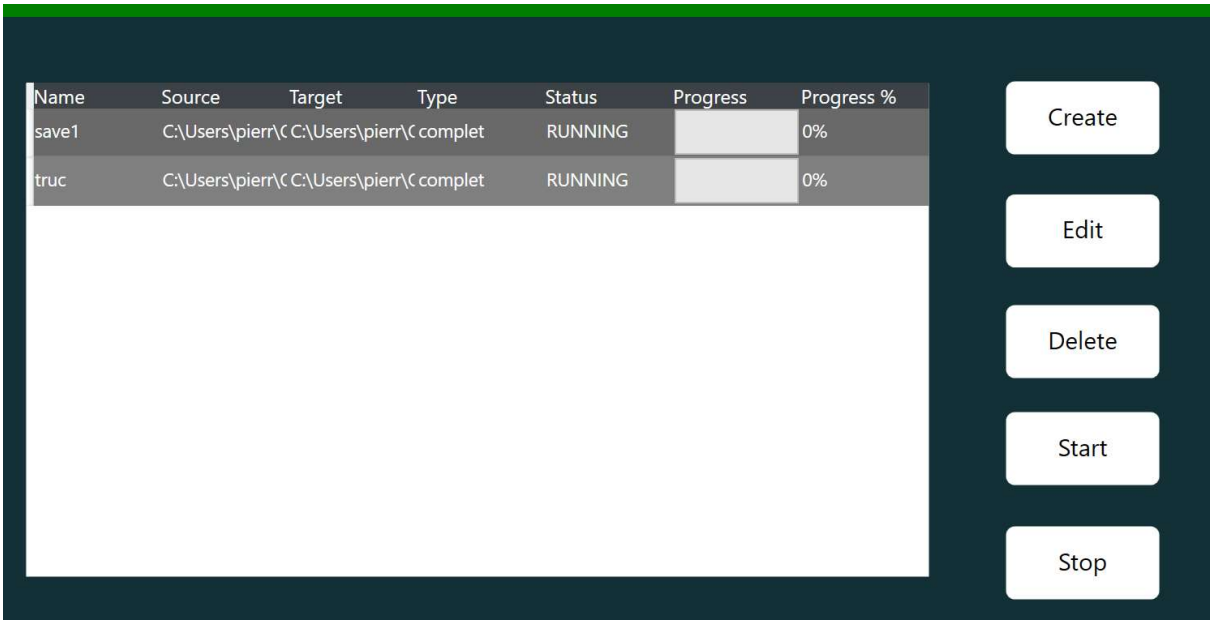
```
public static void startThreadBackupTask(SaveTask backupTask,Mutex backupsJsonMutex)
{
    Trace.WriteLine("startThreadBackupTask : " + backupTask.name);

    processStartBackup bt = new processStartBackup(backupTask, backupsJsonMutex);
    System.Threading.Thread thrd = new System.Threading.Thread(bt.startBackupTask);

    //dicoThrd.Add(backupTask.uuid,thrd);
    thrd.Start();
}
```

- Mettre sur Stop : le logiciel doit afficher un bouton d'arrêt immédiat du travail et de la tâche en cours

```
/// <summary>
/// function that stop a backupTask
/// </summary>
/// <param name="backupTask"> (BackupTask) BackupTask to stop </param>
0 références
public static void stopBackupTask(SaveTask backupTask)
{

    backupTask.progress.state = "PAUSE";
    SaveTaskViewModel.updateBackupTask(backupTask.uuid,backupTask.name,backupTask

}
```

Comme vous pouvez le voir, on appelle donc la même fonction pour stopper et mettre en pause un travail de sauvegarde.

Côté interface cela donnera ceci :



Start : Pour démarrer ou reprendre après une pause

Stop : pour faire pause ou arrêter le travail en cours

L'utilisateur doit pouvoir suivre en temps réel l'état d'avancement de chaque travail (au minimum avec un pourcentage de progression).

Comme vous pouvez le voir sur la capture d'écran de l'interface ci-dessus, il y a les différentes caractéristiques d'un ou de plusieurs travaux de sauvegarde :

- Name : Le nom du travail de sauvegarde
- Source : Le fichier source où se trouve les données à sauvegarder
- Target : Le fichier de destination où se trouve les données sauvegardées
- Type : Le type de sauvegarde si c'est complet ou différentiel
- Status : Le statut du travail de sauvegarde
- Progress : Là où se trouvera la barre de progression visible en temps réel
- Progress % : Là où sera affiché le pourcentage de progression du travail en cours

Pour l'affichage des travaux existants, les méthodes qui seront appelées sont les suivantes :

La méthode **readBackupTask()** pour afficher la liste des travaux de sauvegarde :

```
/// <summary>
/// Function that return all the Backup existing in the savefile
/// </summary>
/// <returns>return a BackupTaskList (list of BackupTask object)</returns>
3 références
static public SaveTaskList readBackupTasks()
{
    string json = File.ReadAllText("../../../src/config/backups.json");
    SaveTaskList backupTaskList = JsonConvert.DeserializeObject<SaveTaskList>(json);
    return backupTaskList;
}
```

La méthode **updateBackupTask()** pour afficher la liste mise à jour après des ajouts :

```
static public SaveTask updateBackupTask(string uuidInput, string nameInput, string sourceI
{
    string json = File.ReadAllText("../../../src/config/backups.json");

    SaveTaskList backupTaskList = JsonConvert.DeserializeObject<SaveTaskList>(json);

    for (int x = 0; x < backupTaskList.taskList.Count; x++)
    {
        if (backupTaskList.taskList[x].uuid == uuidInput)
        {
            //modify
            backupTaskList.taskList[x].name = nameInput;
            backupTaskList.taskList[x].source = sourceInput;
            backupTaskList.taskList[x].target = targetInput;
            backupTaskList.taskList[x].type = typeInput;
            backupTaskList.taskList[x].progress = progressInput;



            //Ecrit la sauvegarde
            string backupTaskListJson = JsonConvert.SerializeObject(backupTaskList);
            File.WriteAllText("../../../src/config/backups.json", backupTaskListJson);

            //return object
```

Pause temporaire en cas de fonctionnement d'un logiciel métier

Le logiciel doit obligatoirement mettre en pause le transfert de fichiers si celui-ci détecte le fonctionnement d'un logiciel métier.

Pour l'illustration, l'application "calculatrice" représentera notre logiciel métier. Ainsi, si elle est lancée, toutes les tâches de notre logiciel doivent être mises en pause.

La méthode détecte si le logiciel métier est en cours d'exécution :

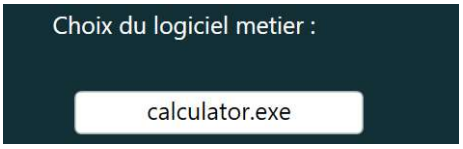
```
1 reference
public static bool jobProcessIsRunning()
{
    configFile config = readConfFile();
    string getProcess = config.process;
    string process = getProcess.Remove(getProcess.Length - 4);
    Process[] pname = Process.GetProcessesByName(process);
    if (pname.Length == 0)
        return false;
    else
        return true;
}
```

On peut alors appeler la méthode effectuant la "pause" du travail de sauvegarde :

```
public static void stopBackupTask(BackupTask backupTask)
{..
    backupTask.progress.state = "PAUSE";
    BackupTaskViewModel.updateBackupTask(backupTask.uuid,backupTask.name,backupTask.source,backupTask.target,backupTask.type,
```

On lit le nouveau choix du logiciel métier (ici, arbitraire, choisi dans l'interface graphique) :

```
private void workJob_TextChanged(object sender, TextChangedEventArgs e)
{
    Regex regex = new Regex(".exe");
    if (regex.IsMatch(workJob.Text))
    {

    }
    else
    {
        workJob.Text += ".exe";
    }
}
```



Console déportée

Pour permettre de suivre en temps réel l'avancement des sauvegardes sur un poste quelconque, une interface homme-machine (IHM) permettra à un utilisateur de suivre, sur un poste distant, l'évolution des travaux de sauvegarde, mais aussi d'agir sur ces travaux.

Cette dernière est réalisée avec WPF et FrameWork .Net Core. La communication est assurée par des Sockets.

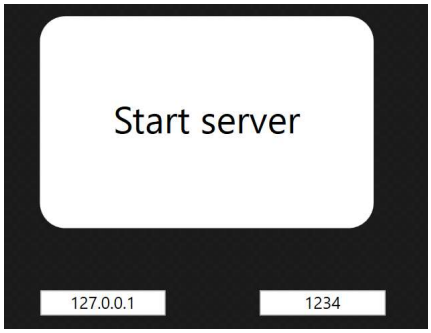
La requête du client exige donc une communication entre 2 poste distant en utilisant des socket.

Architecture et Protocole

Pour utiliser les socket nous allons réaliser un architecture client serveur, server lourd et client léger. Il faut aussi déterminer le type de protocole à utiliser. En effet, il ya plusieur type de protocole que nous pouvons utiliser avec les socket et le plus adapté à notre situation est le TCP.Le protocole adapté car les informations seront bien arrivées au complet, aucun changement dans le contenu et dans l'ordre. TCP par rapport à UDP : garanti de l'exactitude et l'ordre des données ainsi que la gestion de session, mais c'est plus long. Le TCP permet aussi d'assurer le temps réel puisque dans le développement doit être en dessous de la ms.

Connexion

Côté Serveur :
Une méthode qui écoute le client.
Il y a aussi un Socket client côté serveur. La différence n'est pas dans leur utilisation mais dans leur instanciation. Il utilise le même socket que côté client mais ne le crée pas de la même manière. Client => socket.connect() et Serveur => socket.accept()



Ici on permet à l'utilisateur de créer son EndPoint et de lancer le serveur.

```
while (true)
{
    Socket accept = client.Accept();
    int bytesRec = accept.Receive(bytes);
    datas = Encoding.ASCII.GetString(bytes, 0, bytesRec);
    string[] data = datas.Split(" ");


    if (data[0].Equals("progress"))
    {
        try
        {
            BackupTask save = BackupTaskViewModel.readBackupTask(data[1]);
            byte[] msg = Encoding.ASCII.GetBytes(save.progress.pourcentage.ToString());
            accept.Send(msg);
        }
        catch { }
    }
```

Ici, accept.send(msg), on envoie la progression d'une sauvegarde à la demande du client, de la même manière on implémente les autres fonctionnalités.

Application Mono-instance

Pour réaliser la mono-instance de l'application, nous allons utiliser les classes Process ainsi que Path et Assembly.

Process

Créer un tableau de nouveaux composants Process et les associe à toutes les ressources de processus sur l'ordinateur distant qui partagent le nom de processus spécifié.

Path

Renvoie le nom de fichier sans l'extension d'un chemin d'accès de fichier, représenté par une étendue de caractères en lecture seule.

Assembly

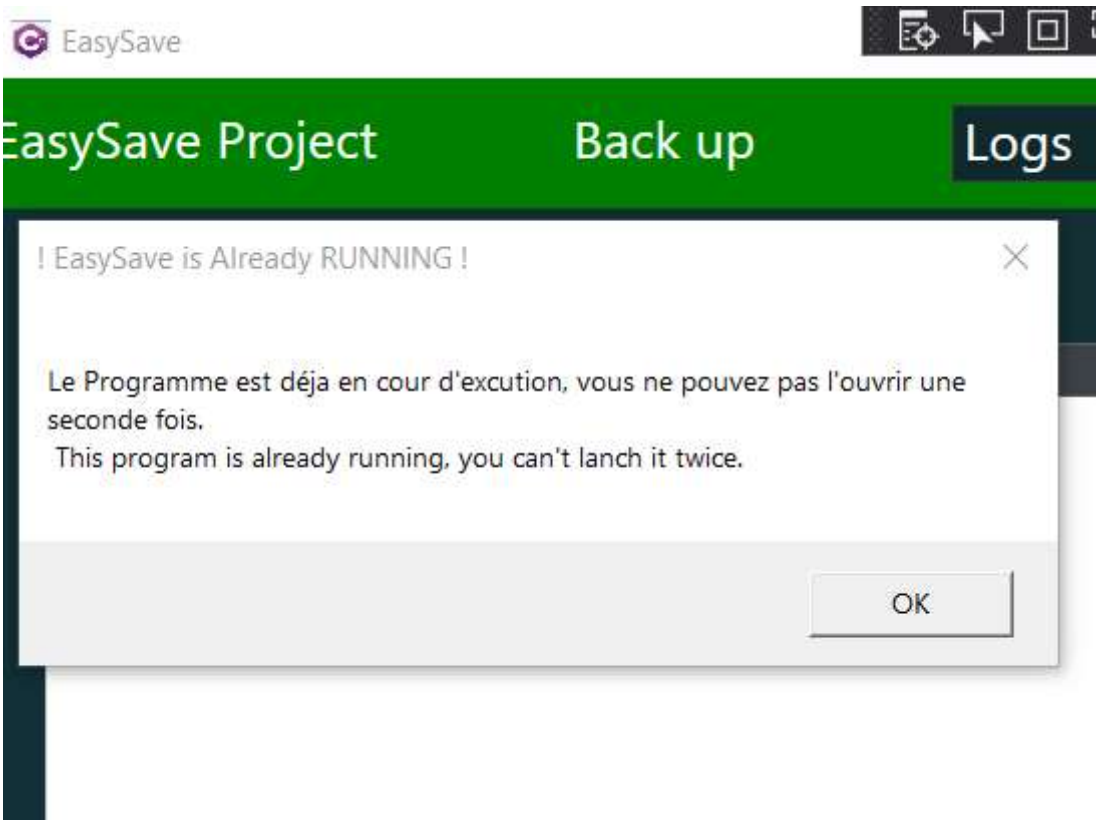
Obtient l'exécutable du processus dans le domaine d'application par défaut.

Condition d'instanciation

Pour avoir une application en mono-instance, on vérifie, au moment de l'initialisation, si notre exécutable est déjà en cours d'exécution.
Pour se faire, avec assembly, on récupère la localisation de notre exécutable, puis avec path, on ne retient que le nom de notre exécutable, puis avec process on fait un tableau des tous les processus répondant au nom de l'exécutable.
La condition est donc évidente : si il y a plus de 1 processus dans le tableau, qui ne recense que les processus du nom de notre exécutable, il faut notifier l'utilisateur que le programme a déjà été lancé et fermer la nouvelle tentative de lancement.

Voici la réalisation en C# :

```
// Checks if the app is already running,

if (Process.GetProcessesByName(System.IO.Path.GetFileNameWithoutExtension(Assembly.GetEntryAssembly().Location)).Length > 1)
{
    //The application is already running therefore we're noticing the user and closing the new easysave
    MessageBox.Show("Le Programme est déja en cour d'excution, vous ne pouvez pas l'ouvrir une seconde fois. \n This program
    System.Windows.Application.Current.Shutdown();
}
```



Evolution

Comme piste d'évolution, on pourrait, au lieu de notifier à l'utilisateur que l'application est déjà lancée, afficher la fenêtre qui est en cours d'exécution.

Conclusion

On constate alors que l'objectif du livrable est de rendre le logiciel plus performant et commercial, avec l'ajout d'une console déportée des interactions avec les travaux, possibilité de priorisation de traitement, effectué plusieur sauvegarde en même temps. La version 3.0 de Easy Save est donc un produit fonctionel et commercialisable grâce à l'ajout des dernières fonctionnalités.

Bilan personnel

Milan Sangare : Pendant la réalisation de ce livrable, je me suis principalement occupé de la mono instance, ainsi que de la mise en place d'une console déportée mais j'ai aussi apporté mon soutien lors de la réalisation de autre partie.

Pierre Grossin : Pour ce livrable, je me suis chargé de la partie sur la gestion des fichiers prioritaires ainsi que la partie sur le transfert en simultané et la taille maximale des fichiers autorisée.

Alexandre De Jesus Correia : Durant ce livrable, je me suis chargé des parties concernant la mise en pause (temporaire) en cas de fonctionnement d'un logiciel métier et de la console déportée. De plus, j'ai aidé au mieux mes collègues sur leurs parties respectives.

Eugénie Hariniaina : Durant la mise en place de la dernière version de notre projet, j'étais chargée d'assurer la partie sauvegarde parallèle et celle sur les intéractions en temps réel avec chaque travail ou l'ensemble des travaux.

CESI
ÉCOLE D'INGÉNIEURS

Bibliographie

Sauvegarde en parallèle :
Threads et threading | Microsoft Learn
Multithreading en C# - WayToLearnX

Mon-instance :
https://learn.microsoft.com/en-us/dotnet/api/system.threading.mutex.-ctor?redirectedfrom=MSDN&view=net-7.0#System_Threading_Mutex__ctor_System_Boolean_System_String_System_Boolean__
https://www.developpez.net/forums/d1386617/dotnet/langages/csharp/instance-unique-d-application/
console déportée :
http://csharp.net-informations.com/communications/csharp-socket-programming.htm
https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/sockets/socket-services