Project Realization Jetze Luyten SCK CEN

Contents

Introduction

The Critical Documents Synchronisation Tool is a program that is used by BR2 and some emergency rooms at SCK CEN in Mol. It is used to download important documents from Alexandria. Alexandria is a huge library full of documents, finding the ones you're looking for is difficult, that's why CDST is created.

CDST

There is a startrepo on DevOps. Using this repo, I followed the following steps:

- Copy the reop to Visual Studio 2022
- Checking the code until I understand what was written before
- Using a document which explains some of the bugs, fixing those bugs
- Creating a new project

Bugs

There are a few bugs that I had to fix to bring this project to a good end.

Default ‘yes’ for ‘mandatory’ attributes

You can select from where you can download documents and to what location, using this tool. To filter documents in Alexandria, we use attribute. These attributes have a checkbox ‘mandatory’. If this is off, then all documents get downloaded. If it is on, then all documents with that attribute and value get downloaded. I have solved this by using the ‘isEnabled’-element of a checkbox and setting it to ‘true’.

```
private void btnAddAttribute_Click(object sender, RoutedEventArgs e)
{
    AddAttribute addAtt = new AddAttribute(new Attribute())
    {
        Title = "Alexandria Critical Documents - Add Attribute"
    };
    addAtt.chkBoxMandatory.IsChecked = true;
    if (addAtt.ShowDialog() == true)
    {
        Attribute att = addAtt.Attribute;
        addAtt.TxtBlckAttribute.Text = "Add a new attribute to collection";
        var parents = ((Button)sender).GetParents();
        foreach (var item in parents)
        {
            if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
            {
                Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

                AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
                opt.Attributes.Add(att);

                RadGridViewDocuments.Rebind();
            }
        }
    }
}
```

Tool fails on nodes with a question mark in the name

Both Windows and Alexandria have special characters that you cannot use. In Alexandria this is ':' and in Windows there are many others. To solve this, I used regular expressions to remove the characters that Windows and Alexandria do not support.

```
public static string GetSafeFilename(string documentName)
{
    documentName = string.Concat(documentName.Split(Path.GetInvalidFileNameChars()));
    documentName = Regex.Replace(documentName, "[ \f\t\v]+$", "");
    documentName = Regex.Replace(documentName, "[ ]{2,}", " ");
    return documentName;
}
```

Make stopping automatic sync a bit more difficult

You can choose whether to sync manually or download it on a timer (every 24 hours, for example). If you go for the automatic sync, there is an option to stop it. The bug is that, when you press the button, it stops the automatic sync immediately. I fixed this by calling up a message box that asks the user if they are sure they want to stop the sync.

```
MessageBoxResult result = MessageBox.Show("Are you sure you want to stop the automatic download?", "Warning!", MessageBoxButton.YesNo, MessageBoxImage.Warning);
switch (result)
{
    case MessageBoxResult.Yes:
        DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " interrupted by user";
        OptionsContext context = new OptionsContext();

        context.UpdateOptions(new List<KeyValuePair<string, string>>
            {
                new KeyValuePair<string, string>("lastSuccessRun",
                    DocumentProcessing._lastSuccess)
            });
        SetLastSuccessfulRuns();
        Process.GetProcessById(_processId).Kill();
        break;
    case MessageBoxResult.No:
        break;
}
```

A new (test)project

The original project was created for .NET EntityFramework v4.7 and the new project is .NET Core v6.0

The test project was used to improve my knowledge of .NET Core and the changes between versions. I started by adding a textbox where I include the link to the otds server. I gave this a particular name to add events to it later. Then I added another textbox that I disable so no one can edit it and I gave it a certain name. Finally, I added a button to retrieve your otds ticket using the otds link. I displayed the ticket in my other textbox. In both textboxes I put a binding. This was the end of the frontend (for now).

In the code-behind, I made sure that if you press the button and the otds link is not empty, then the client tries to connect to the Alexandria server to see if that my link is correct. If the link is empty or Alexandria does not find that link, then an error message is displayed reporting that "Otds link does not exist!". If Alexandria finds the link, then the ticket is returned.

After getting the otds ticket, I started implementing MVVM. I did this by first creating 3 folders: Model, View and ViewModel. I placed my frontend to the View folder.

After I finished creating my "Options" model, I checked to see if Alexandria is reachable. I did this by going back to my View and adding some items. First, I added a textbox. In this textbox, a user can enter a link so they can connect to Alexandria. Next, I added a checkbox that is disabled. This checkbox was used to see if that Alexandria is reachable. In both components I put a binding.

Then in the backend I made sure that if you click the button (which we used to get the otds ticket), that the client tries to connect to Alexandria adhv the "AlexandriaUrl" textbox. If the textbox is empty or the link to Alexandria cannot be found, the client returns this error message "Connection to Alexandria has been refused due to incorrect url." And the checkbox is off. If you can connect to Alexandria, then the checkbox is on. This was the end of my test to connect to Alexandria.

After connecting to Alexandria, I had created some new Views: AddAttribute, AddCollection, MainWindow and OptionsWindow.

View

MainWindow.xaml

This is the first screen the user sees. This xaml consists of a few elements: Window, Grid, Textblock, Button and Stackpanel.

In the Window element we put all the information that our screen contains: the title of my screen and the height and width of my screen.

The Grid is the subdivision of the different elements. The Grid is divided into 5 rows to be defined. In this we also specify the height of each row.

Once we have defined the rows, then we add a Textblock. This is an element where you can work flexibly with. This element does not need multiple paragraphs of text. We put the Textblock on row 0 of our Grid. We give our element a name and make text automatically go to the next line. We also give our element a certain padding and margin.

The next element we add is a Button. A Button is an element that creates Events when you interact with it. The Button is on a certain row, has a certain margin and background color. There is an element Resources of the parent element Button. In this we put another element Style with the TargetType being Border. That is, we change the border style of our button. We change the CornerRadius to 2 with a Setter so that the button has a more rounded corner. The next element is a Button and we do exactly the same with it as the first Button, but we change the row, name and event. For the 3rd Button, we do the same thing, but we set the Visibility to Visible.

After the Button element, we create a StackPanel element. A StackPanel is used to stack elements on top of each other in a specific direction. The StackPanel we use has some properties: Name, Grid.Row, Orientation, Visibility and Margin. Name is the name of the element. Grid.Row is the location of our element in the Grid. Orientation is how we stack the elements, in our case this is on top of each other (you can also stack them horizontally). We set the Visibility to Collapsed and we have a Margin of 5 and 0.

In the StackPanel element we have some elements: TextBlock and ProgressBar. The TextBlock element is used to display which items are being downloaded. The ProgressBar is used to show that the files are downloading. The ProgressBar element has 1 special property: IsIndeterminate. This property is used to see an animation in which you see part of the bar go by.

The next (and last) element is a StackPanel with a vertical orientation. There are 4 TextBlock elements. The first and third are used for information of when the last (successful) run succeeded. The second and fourth TextBlock gives the date of when the last (successful) run succeeded.

MainWindow.xaml.cs

In de class maken we enkele private attributen: _otdsTicket, _inProgress, _processId, _lastRunText, In this class, we create some private attributes: _otdsTicket, _inProgress, _processId, _lastRunText, _lastSuccessRunText and _readWriteLock. _otdsTicket, _lastRunText and _lastSuccessRunText are strings. _inProgress is a bool. _processId is an int. _readWriteLock is a static. _lastRunText and _lastSuccessRunText are both empty strings. _readWriteLock is a constructor of ReaderWriterLockSlim.

In the MainWindow constructor, we change the look and feel of the View. We pass in 2 new methods: SetLastRuns() and SetServiceButton().

OnStateChanged(EventArgs e) causes us to hiden the window when you minimize it.

SetServiceButton() is a method where we change the btnService button depending on its state. We have 1 attribute: schedule which is a string. Here we are looking at the status of an option called schedule. If the schedule option is null or False, then we disable the btnService and set the content to "Automatic download not enabled". Otherwise, we enable the btnService button and we create a new Process[] that focuses on the process name "Alexandria CDST". If the length of that process is equal to 0 (when the process finishes), then we set the content of btnService button to "Start automatic download" and we set _processId to 0. Otherwise, we set the content of btnService button to "Stop automatic download".

The next method is an event when you press the btnOptions button. We give 2 arguments here: the object sender and RoutedEventArgs e. We use this button to go to another window (OptionsWindow) and we use the SetServiceButton() method.

The next method is SetLastRuns(). In this we use the constructor OptionsContext(). In this constructor we create 2 attributes of type string: last and lastSuccess. Last is the value of the last run. LastSuccess is the value of the last successful run. We put last in _lastRunText and lastSuccess in _lastSuccessRun. Then we empty LastRun.Text and put the date in bold with LastRun.Inlines. The same goes for LastSuccessRun.Text and LastSuccessRun.Inlines.

BtnService_Click is the next method. The parameters are the same as the btnOptions_Click. If the content of the btnService button is equal to "Start automatic download", then we create a new process in which we hiden our program. This process will be started. Otherwise, we stop the process. Finally, the thread waits half a second and we call the method SetServiceButton().

BtnForce_Click(object sender, RoutedEventArgs e): we change the visibility of btnForce to collapsed and the prgsDownload to visible. We disable both btnOptions and btnService. Next, we change the cursor to the wait icon. Next, we wait for the ProcessRequest task to finish running. After this, we call the SetLastRuns() method. We set the visibility of btnForce to visible and prgsDownload to collapsed. We enable btnOptions and btnService. Finally, we set the cursor to default.

The ProcessRequest() method is used to start the download of all files/documents and folders. In this method, we write "Start of forced download" to the log file. We update the options. We set _inProgress to true, we write "Getting Otds Ticker" to the log file and we call AlexandriaContext. When the client returns the otds ticket, the client gets a notification that the ticket returns. Next, the client checks that Alexandria is reachable. If it is, then the client gets a response back and a collection of containers is retrieved. Then the client tries to retrieve the container node for each item (from AlexandriaSourceOptions) in the OptionsContext. Then it checks if the list of files in that node is reachable and returns the node. After this, a backup is created and the CopyOldDirectory() method is called. It then transfers the files to the local folder specified by the user. This creates a new folder with the node and files by the BuildNewDirectory() method. Next, the options are updated. If there is a problem with any of these steps, the error message is written to the log file and the client displays the error message on the screen. The same applies if Alexandria is not reachable or if the otds ticket is not reachable. Finally, "End of forced download" is written in the log file and the _inProgress attribute is set to false.

The CopyOldDirectory() method has as a parameter the name of a directory. This parameter is of type string. In this method, 1 new attribute is added: retry, which is a bool. We set this attribute to true. As long as that retry is set to true, the client tries to retrieve the OptionsContext and we create 2 attributes: basePath and fullPath. They are both of type string. The basePath retrieves the value of the "localdirectory" TextBlock. Then we use the basePath + folderName + "/" to get our fullPath attribute. As long as that directory of fullPath exists, we're going to see if that doesn't already have a backup for it. If it does, then the directory will be created. If the backup containing the directory name exists, the directory is deleted. After this, we move the downloaded files to the backup/folder name directory. The attribute retry is set to false. If there is an error in this section it is reported with a message box.

The BuildNewDirectory() method has 2 parameters: folderName and list. folderName is of type string and list is of type DocumentList. In this method, a new directory is created. Then the files are transferred.

Buildfiles() is a method with 3 parameters: baseDirectory, doc and checkfolder (which is set to false). baseDirectory is of type string, doc is of type DocumentList and checkFolder is of type bool. In this method, we have an attribute retry of type bool that defaults to true. As long as that retry is set to true, the client tries to modify the file/folder name so that the file/folder can be downloaded. If a document contains multiple documents, we get the baseDirectory + the folder name + "/" and we put this in the parameter folderPath with the type string. If checkFolder is set to false, or baseDirectory is not equal to the document name, then a new directory is created with folderPath as the parameter. Otherwise, we set folderPath equal to baseDirectory. Then we build all documents in the folder at the same time. If a document contains no other document, then the client looks at the node's subType. If nodeSubType is equal to 140, then a shortcut is created that leads to the original document. If nodeSubType is equal to 749, then the file is downloaded (if Alexandria is reachable). At the end, we set retry to false. If there is an error, then the error is written to the log file and shows the client that something went wrong and what.

GetSafeFilename() is a method with parameter filename which has type string. In this method special characters are extracted from the file name using regular expressions.

WriteToUi() is a method with parameter message which is a string. In this method, the message is written to a TextBlock. We end with a method WriteToFile(), which contains as parameter message.

Window_Closing() is an event used when you close the program. If you close the program when it is downloading, the client asks if the user is sure they want to close the window. If the user selects "No," then the downloading continues. Otherwise, the program is closed.

WriteToFile() is a method with text as parameter, which has string as type. In this method, "DocSyncServiceLog.txt" is put into an attribute filename which contains the type string. The attribute basePath is registered with type string. Next, we extract the value from localDirectory and put it into basePath. We merge basePath and filename to get an attribute path with type string. There is also an attribute writeText that writes text to the file. Next, _readWriteLock is instantiated. After this, the client tries to get the date time written to the document. Finally we stop the lock and write the date and time.

OptionsWindow.xaml

The title of the window is "Alexandria Critical Documents - Options."

The first element we are going to discuss is the ScrollViewer. The ScrollViewer is an element that is used when a higher level element (which contains this element) contains a certain size, then a scrollbar is visible. We have an attribute HorizontalScrollBarVisibility that is used to make the horizontal bar visible. In this case, it happens automatically.

In the ScrollViewer, we have the Grid. This is an element that determines what position other elements are at.

Inside the Grid, there is a StackPanel with a vertical orientation. Inside the StackPanel there are 2 TextBlocks.

The next element is a TextBlock located in the Grid. The TextBlock is shown in bold and underlined.

In the Grid we have another element Grid in which we define the columns and rows.

In the 2nd Grid we have 4 elements: 2 Labels and 2 TextBoxes.

We go back up one level and create a TextBlock element that will be in bold and underlined.

The next element is a Grid with defined columns and rows. In this element are 4 other elements: 2 Labels, 1 Checkbox and 1 TextBox.

We put back a TextBlock element that puts words in bold and underlined.

Next is a Grid with columns and rows. In the Grid, we have a Labe and a TextBox.

After this we place a Button element with a CornerRadius of 2.

At the same height, we have a DataGrid element. In this element, a user cannot add rows with CanUserAddRows. We set the vertical scrollbar to auto.

The next element is DataGrid.Columns. In this element there are some columns. The first column is a type of text, its name is NodeName and the user cannot modify anything in it. The 2nd column is a type of text, the name is NodeId and the user can read all of them. The next element is a TemplateColumn with a CellTemplate and DataTemplate in it. In the DataTemplate we place a Button. We do this 2 more times.

After the DataGrid we have a Seperator. This is a line that distinguishes between parts of the frontend.

Finally, we have a Button with Grid.Column of 1, Grid.Row of 12, name is btnSaveSettings, Click is btnSaveSettings_Click and a blue Background. This Button has a CornerRadrius of 2.

OptionsWindow.xaml.cs

In the class library, we have 1 general attribute: newCollection, which is a list of AlexandriaSourceOptions.

In the constructor, we have an initialization of OptionsContext that we set to context. We fetch the value of the otdsURL and place it in the TextBox txtBoxOtdsUrl. Next, we fetch the value of the alexandriaURL and place it in the TextBox txtBoxAlexandriaUrl. If the value of the schedule option is true, then we set the CheckBox chkPeriodic property IsChecked to true. Otherwise, we set the chkPeriodic property IsChecked to false. After this, we fetch the value of the syncTime option and place it in the TextBox txtBoxSyncTime Text property. Next, we fetch the value of the localDirectory option and place it in the TextBox txtBoxDirectory Text property. Next, we retrieve the collection of containers and place it in the attribute newCollection. Finally, we place this attribute in the ItemsSource of ViewDocuments. This fills our DataGrid with containers.

The next item on the list is the method SaveOptions() which is a void. In this method we create an attribute that is a list of KeyValuePair<string, string>. This attribute is called options. We add the schedule and syncTime to this attribute. Next, we read the TextBox txtBoxDirectory property and remove the white-space characters and replace a '/' with '/'. We place this in a string attribute localdirectory. If the localdirectory is empty, we fill it with D:/. Otherwise if the last index of 'ith' is not the same as the length of localdirectory -1, then we add 'ith' to localdirectory. Then we remove the whitespaces before and after the TextBox txtBoxOtdsUrl Text property and place it in a string attribute otdsurl. If the last index of otdsurl '/' is not the same as the length of otdsurl -1, we add '/' to otdsurl. Then we remove the whitespaces before and after the TextBox txtBoxAlexandriaUrl Text property and place them in a string attribute axurl. If the last index of axurl '/' is not the same as the length of axurl -1, we add '/' to axurl. After this, we add the localdirectory, otdsurl and axurl to the attribute options. Next, we create a new attribute of OptionsContext(). Then we update the options with UpdateOptions(options). We update the collections with newCollection and place them in the ItemsSource of ViewDocument. Finally, we close the screen.

The button btnSaveSettings saves the options.

The btnDeleteCollection button uses OptionsContext to delete a row from newCollection and then update it. We place this in the ItemsSource of ViewDocuments.

btnAddCollection adds a container.

AddCollection.xaml

In our Grid, we define 4 columns and 4 rows. We set the width of the first column to 160.

Next, we have 2 TextBlocks. The first one sits on Row 0, Column 0 and ColumnSpan 4 of our Grid. The 2nd TextBlock has the Name error and sits on Row 0, Column 0 and ColumnSpan 4 of our Grid with a Margin of 0 14 0 0. Error has a red text color that is closed.

After this, there are 2 Labels and 2 TextBoxes. The first Label is on Row 1 and Column 0. This control has a Margin of 4. The next control is a Textbox named TxtBoxName. This control sits on Row 1, Column 1 and ColumnSpan 3. TxtBoxName has a Margin of 4. The 2nd Label sits on Row 2 and Column 0. This

control has a Margin of 4. The next control is a TextBox named TxtBoxNodeId. TxtBoxNodeId sits on Row 0, Column 1 and a ColumnSpan of 3. This control has a Margin of 4.

The last control is a Button named Savecoll that sits on Row 3, Column 0, has a Margin of 4 and has a dark blue background color. This Button has a CornerRadius of 2.

AddCollection.xaml.cs

We start with an attribute _Collection whose type is AlexandriaContext.

The first method is AddCollection() with a parameter options of type AlexandriaSourceOptions. In this method, we set _Collections equal to options. We fetch the Name of _Collection and place it in the TextBox TxtBoxName Text property. Next, we extract the DataId from _Collection, make it a string and place it in the TextBox TxtBoxNodeId Text property.

The next method is when you press the button. In this method, we have 2 attributes: context and result. The attribute context has type AlexandriaContext. The attribute retrieves the container and is of type NodeResult. If result is not null and Code of result is not 500 and there is no Message of result, then we check if the Text of TxtBoxName is not empty. If TxtBoxName.Text is not empty, then the Name of _Collection is equal to TxtBoxName.Text. Otherwise, we place the Name of result in Name of _Collection. If Attributes of _Collection is null, then _Collection.Attributes becomes a new list of Attributes. We convert the TxtBoxNodeId.Text to an integer and place it in _Collection.DataId. finally, we set DialogResult to true. Otherwise, an error message will be displayed.

Model

Options

I added a class library to my Model named "Options". I created an inheritance of INotifyPropertyChanged. Then I added some attributes: _otdsUrl, _otdsTicket, _alexandriaUrl and _nodeId. All attributes are a string except _nodeId, which is an int.

Alexandria

I added an item to my Model named "Alexandria". In it, my class inherits from INotifyPropertyChanged. Then I added some attributes: _nodeId, _message and _name. All attributes except _nodeId are strings, _nodeId is an int.

AlexandriaResult

Another new class being added in my Model. This time with the name "AlexandriaResult". In it there are 3 attributes: _code, _message and _alexandriaId. _code and _alexandriaId are both an int and _message is a string.

DocumentList

A class that is added in my Model. It has the name "DocumentList" containing 7 attributes: _nodeId, _nodeName, _version, _nodeSubType, _url, _lastModified and _children. _nodeId, _version and _nodeSubType are of type int. _nodeName and _url are of type string. _lastModified is of type DateTime. Finally, we have _children which represents a list of documents.

DocumentListResult

I added a new class named "DocumentListResult" to my Model. This class contained 3 attributes: _code, _message and _docList. _code is an int. _message is a string. _docList returns a list of documents.

Attribute

I added a new class named "Attribute" to my Model with in the class 4 attributes: _id, _name, _value and _mandatory. _id is an int. _name and _value are a string. _mandatory is a bool.

AlexandriaSourceOptions

I added a new class named "AlexandriaSourceOptions" to my Model containing 4 attributes: _id, _name, _dataId and _atttributes. _id and _dataId are both an int. _name is a string. _attributes is a list of the Model Attribute.

AliveResult

I added a new class to my Model named "AliveResult" in which there is 1 attribute: _isAlive. This attribute is a bool.

DocumentHistory

I added a new class to my Model named "DocumentHistory" in which there are 3 attributes: _id, _documentId and _lastModified. _id and _documentId are an int. _lastModified is a DateTime.

NodeResult

I added a new class to my Model named "NodeResult" in which there are 3 attributes: _code, _message and _name. _code is an int. _message and _name are both a string.

OTDSResult

I added a new class to my Model named "OTDSResult" in which there are 9 attributes:

- _token, _userId, _ticket, _resource_Id, _failureReason, _passwordExpirationTime, _continuation, _continuationContext, _continuationData. _token, _userId and _ticket is a string.
- _resourceId, _failureReason, _continuationContext and _continuationData are all an object.
- _passwordExpirationTime is an int.
- _continuation is a bool.

Changed project

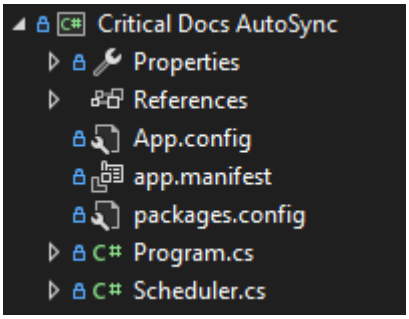
The original project contains classes that already existed and that I added. In this section, I'm going to explain them.

Critical Docs AutoSync project

This project provides automatic synchronization of documents and uses some packages. These packages are:

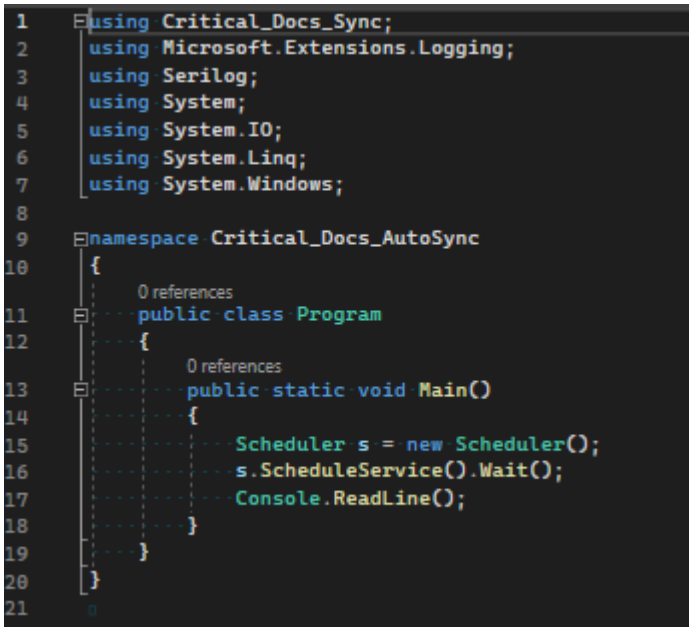
- EntityFramework
- Microsoft.Bcl.AsyncInterfaces
- Microsoft.Extensions.Configuration, Microsoft.Extensions.Configuration.Abstractions, Microsoft.Extensions.Configuration.Binder, Microsoft.Extensions.Configuration.FileExtensions, Microsoft.Extensions.Configuration.Json, Microsoft.Extensions.DependencyInjection, Microsoft.Extensions.DependencyInjection.Abstraction, Microsoft.Extensions.DependencyModel, Microsoft.Extensions.FileProviders.Abstraction, Microsft.Extensions.FileProviders.Physical, Microsoft.Extensions.FileSystemGlobbing, Microsoft.Extensions.Logging, Microsoft.Extensions.Logging.Abstraction, Microsoft.Extensions.Options, Microsoft.Extensions.Primitives
- Newtonsoft.Json
- PresentationFramework
- Serilog, Serilog.Settings.AppSetings, Serilog.Settings.Configuration, Serilog.Sinks.Email, Serilog.Sinks.File, Serilog.Sinks.PeriodicBatching, Serilog.Sinks.RollingFile
- System
- System.Buffers
- System.ComponentModel.Composition, System.ComponentModel.DataAnnotations
- System.Configuration
- System.Core
- System.Data, System.Data.DataSetExtensions, System.Data.SQLite, System.Data.SQLite.EF6, System.Data.SQLite.Linq
- System.Diagnostics.DiagnosticSource
- System.IO
- System.Memory
- System.Net, System.Net.Http
- System.Numerics

- System.Numerics.Vectors
- System.Runtime, System.Runtime.CompilerServices.Unsafe, System.Runtime.InteropServices.RuntimeInformation
- System.Security.Cryptography.Algorithms, System.Security.Cryptography.Encoding, System.Security.Cryptography.Primitives
- System.Text.Encodings.WebSystem.Text.Json
- System.Threading.Tasks.Extensions
- System.ValueTuple
- System.Windows.Forms,
- System.Xml, System.Xml.Linq.



Program.cs

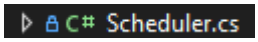
This class refers to Scheduler and creates a new instance 's'. In this instance we use the ScheduleService which downloads the documents and displays when the next synchronization starts.



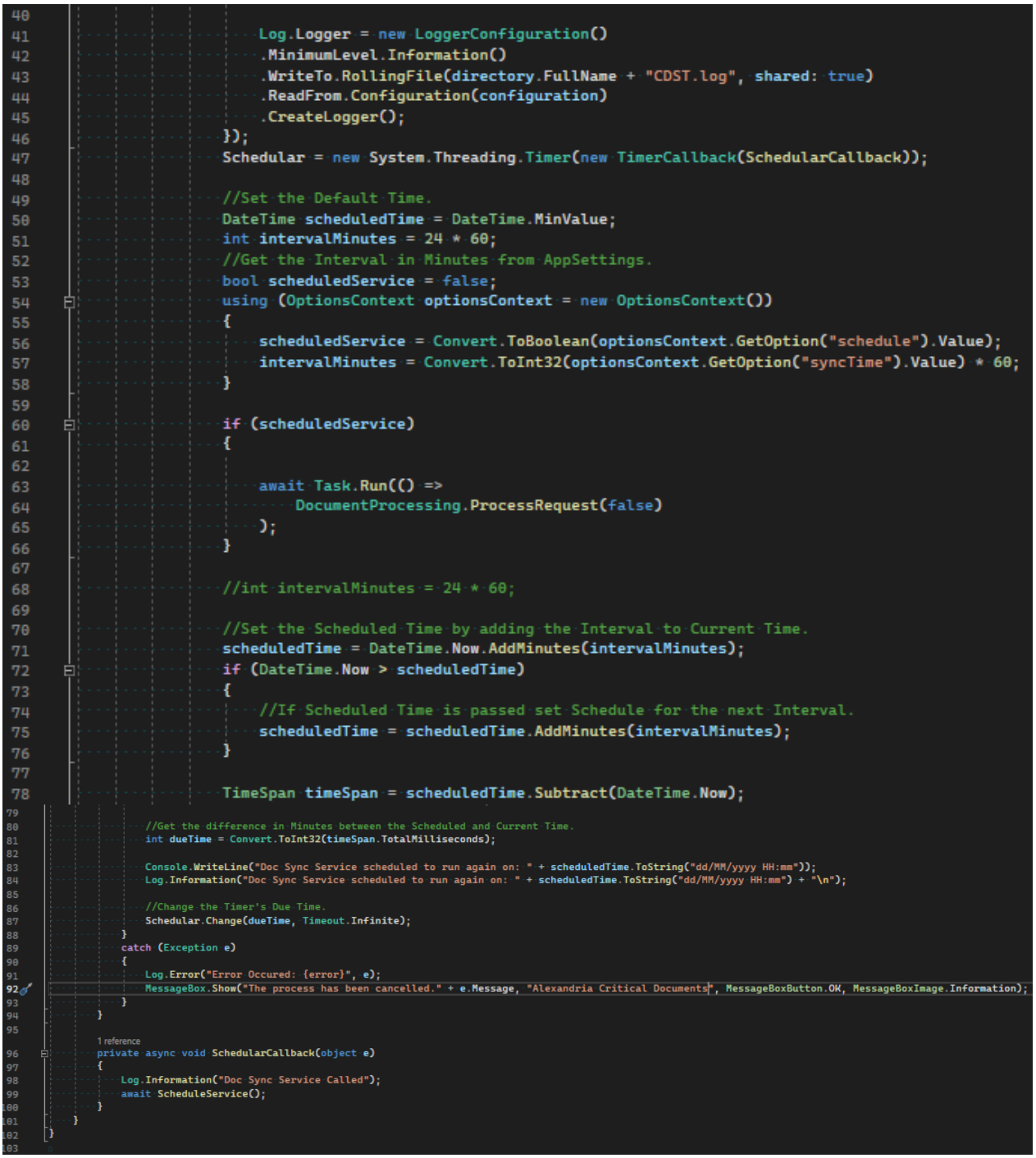
```
using Critical_Docs_Sync;
using Microsoft.Extensions.Logging;
using Serilog;
using System;
using System.IO;
using System.Linq;
using System.Windows;

namespace Critical_Docs_AutoSync
{
    0 references
    public class Program
    {
        0 references
        public static void Main()
        {
            Scheduler s = new Scheduler();
            s.ScheduleService().Wait();
            Console.ReadLine();
        }
    }
}
```

Scheduler.cs

This class automatically downloads documents.



```
using Critical.Docs.Business.Logic;
using Critical_Docs_Sync;
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using MessageBox = System.Windows.MessageBox;
using Serilog;
using Microsoft.Extensions.Logging;
using System.IO;
using System.Linq;
using Microsoft.Extensions.Configuration;

namespace Critical_Docs_AutoSync
{
    2 references
    public class Scheduler
    {
        private static Timer Schedular;
        2 references
        public async Task ScheduleService()
        {
            try
            {
                string localDirectory;
                using (OptionsContext context = new OptionsContext())
                {
                    localDirectory = context.GetOption("localDirectory").Value;
                }

                var configuration = new ConfigurationBuilder()
                .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
                .Build();
                Log.Information(configuration.ToString());

                ILoggerFactory loggerFactory = LoggerFactory.Create(builder =>
                {
                    var directory = new DirectoryInfo(localDirectory);
                    var myFile = directory.GetFiles()
                        .OrderByDescending(f => f.LastWriteTime)
                        .FirstOrDefault();
```



```
                Log.Logger = new LoggerConfiguration()
                    .MinimumLevel.Information()
                    .WriteTo.RollingFile(directory.FullName + "CDST.log", shared: true)
                    .ReadFrom.Configuration(configuration)
                    .CreateLogger();
            });
            Schedular = new System.Threading.Timer(new TimerCallback(SchedularCallback));

            //Set the Default Time.
            DateTime scheduledTime = DateTime.MinValue;
            int intervalMinutes = 24 * 60;
            //Get the Interval in Minutes from AppSettings.
            bool scheduledService = false;
            using (OptionsContext optionsContext = new OptionsContext())
            {
                scheduledService = Convert.ToBoolean(optionsContext.GetOption("schedule").Value);
                intervalMinutes = Convert.ToInt32(optionsContext.GetOption("syncTime").Value) * 60;
            }

            if (scheduledService)
            {

                await Task.Run(() =>
                    DocumentProcessing.ProcessRequest(false)
                );
            }

            //int intervalMinutes = 24 * 60;

            //Set the Scheduled Time by adding the Interval to Current Time.
            scheduledTime = DateTime.Now.AddMinutes(intervalMinutes);
            if (DateTime.Now > scheduledTime)
            {
                //If Scheduled Time is passed set Schedule for the next Interval.
                scheduledTime = scheduledTime.AddMinutes(intervalMinutes);
            }

            TimeSpan timeSpan = scheduledTime.Subtract(DateTime.Now);

            //Get the difference in Minutes between the Scheduled and Current Time.
            int dueTime = Convert.ToInt32(timeSpan.TotalMilliseconds);

            Console.WriteLine("Doc Sync Service scheduled to run again on: " + scheduledTime.ToString("dd/MM/yyyy HH:mm"));
            Log.Information("Doc Sync Service scheduled to run again on: " + scheduledTime.ToString("dd/MM/yyyy HH:mm") + "\n");

            //Change the Timer's Due Time.
            Schedular.Change(dueTime, Timeout.Infinite);
        }
        catch (Exception e)
        {
            Log.Error("Error Occured: {error}", e);
            MessageBox.Show("The process has been cancelled." + e.Message, "Alexandria Critical Documents", MessageBoxButton.OK, MessageBoxImage.Information);
        }
    }

    1 reference
    private async void SchedularCallback(object e)
    {
        Log.Information("Doc Sync Service Called");
        await ScheduleService();
    }
}
}
```

ScheduleService()

In this method, we set the default time and minutes. Then we retrieve the interval in minutes from OptionsContext. The items we retrieve is the value of schedule and syncTime.

If schedule is checked, then we add the interval to the instantaneous time and put it in scheduledTime. Then the program waits for all documents to synchronize.

If the instantaneous time is later than the scheduled time, then we change the scheduled time by the number of minutes.

Then we extract the difference in minutes between the scheduled time and the instantaneous time. After this, we change the time.

```
19          public async Task ScheduleService()
20          {
21              try
22              {
23                  string localDirectory;
24                  using (OptionsContext context = new OptionsContext())
25                  {
26                      localDirectory = context.GetOption("localDirectory").Value;
27                  }
28
29                  var configuration = new ConfigurationBuilder()
30                  .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
31                  .Build();
32                  Log.Information(configuration.ToString());
33
34                  ILoggerFactory loggerFactory = LoggerFactory.Create(builder =>
35                  {
36                      var directory = new DirectoryInfo(localDirectory);
37                      var myFile = directory.GetFiles()
38                           .OrderByDescending(f => f.LastWriteTime)
39                           .FirstOrDefault();
40
41                      Log.Logger = new LoggerConfiguration()
42                      .MinimumLevel.Information()
43                      .WriteTo.RollingFile(directory.FullName + "CDST.log", shared: true)
44                      .ReadFrom.Configuration(configuration)
45                      .CreateLogger();
46                  });
47                  Schedular = new System.Threading.Timer(new TimerCallback(SchedularCallback));
48
49                  //Set the Default Time.
50                  DateTime scheduledTime = DateTime.MinValue;
51                  int intervalMinutes = 24 * 60;
52                  //Get the Interval in Minutes from AppSettings.
53                  bool scheduledService = false;
54                  using (OptionsContext optionsContext = new OptionsContext())
55                  {
56                      scheduledService = Convert.ToBoolean(optionsContext.GetOption("schedule").Value);
57                      intervalMinutes = Convert.ToInt32(optionsContext.GetOption("syncTime").Value) * 60;
58                  }
59
60                  if (scheduledService)
61                  {
62
63                      await Task.Run(() =>
64                          DocumentProcessing.ProcessRequest(false)
65                      );
66                  }
67
68                  //int intervalMinutes = 24 * 60;
69
70                  //Set the Scheduled Time by adding the Interval to Current Time.
71                  scheduledTime = DateTime.Now.AddMinutes(intervalMinutes);
72                  if (DateTime.Now > scheduledTime)
73                  {
74                      //If Scheduled Time is passed set Schedule for the next Interval.
75                      scheduledTime = scheduledTime.AddMinutes(intervalMinutes);
76                  }
77
78                  TimeSpan timeSpan = scheduledTime.Subtract(DateTime.Now);
79
80                  //Get the difference in Minutes between the Scheduled and Current Time.
81                  int dueTime = Convert.ToInt32(timeSpan.TotalMilliseconds);
82
83                  Console.WriteLine("Doc Sync Service scheduled to run again on: " + scheduledTime.ToString("dd/MM/yyyy HH:mm"));
84                  Log.Information("Doc Sync Service scheduled to run again on: " + scheduledTime.ToString("dd/MM/yyyy HH:mm") + "\n");
85
86                  //Change the Timer's Due Time.
87                  Schedular.Change(dueTime, Timeout.Infinite);
88              }
89              catch (Exception e)
90              {
91                  Log.Error("Error Occured: {error}", e);
92                  MessageBox.Show("The process has been cancelled." + e.Message, "Alexandria Critical Documents", MessageBoxButton.OK, MessageBoxImage.Information);
93              }
94          }
95
```

SchedularCallback(object e)

We use this method to call the ScheduleService() method.

```
private async void SchedularCallback(object e)
{
    Log.Information("Doc Sync Service Called");
    await ScheduleService();
}
```

Critical Docs Options

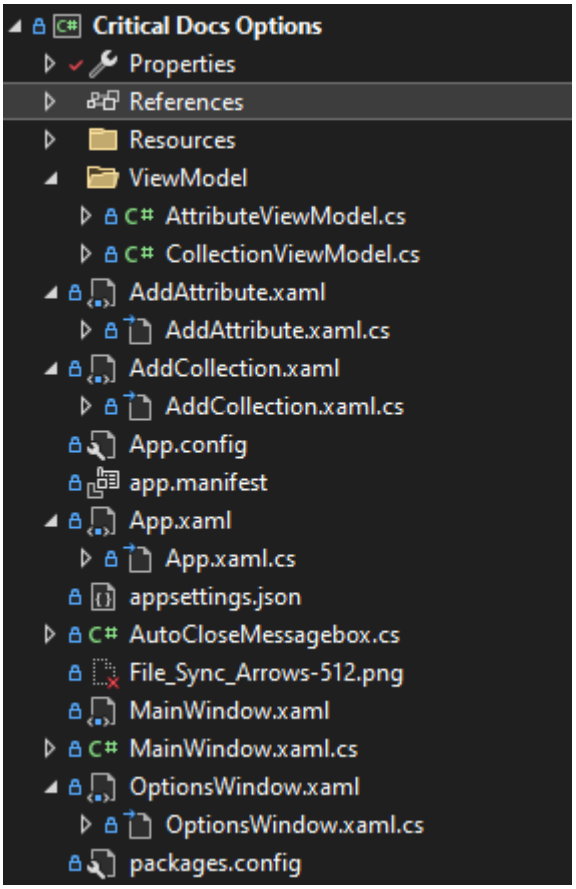
This project contains 2 components of MVVM: ViewModel and View. There are 4 views, 2 viewModels and 1 json. The 4 views are:

- AddAttribute
- AddCollection
- MainWindow
- OptionsWindow.

This project contains some packages:

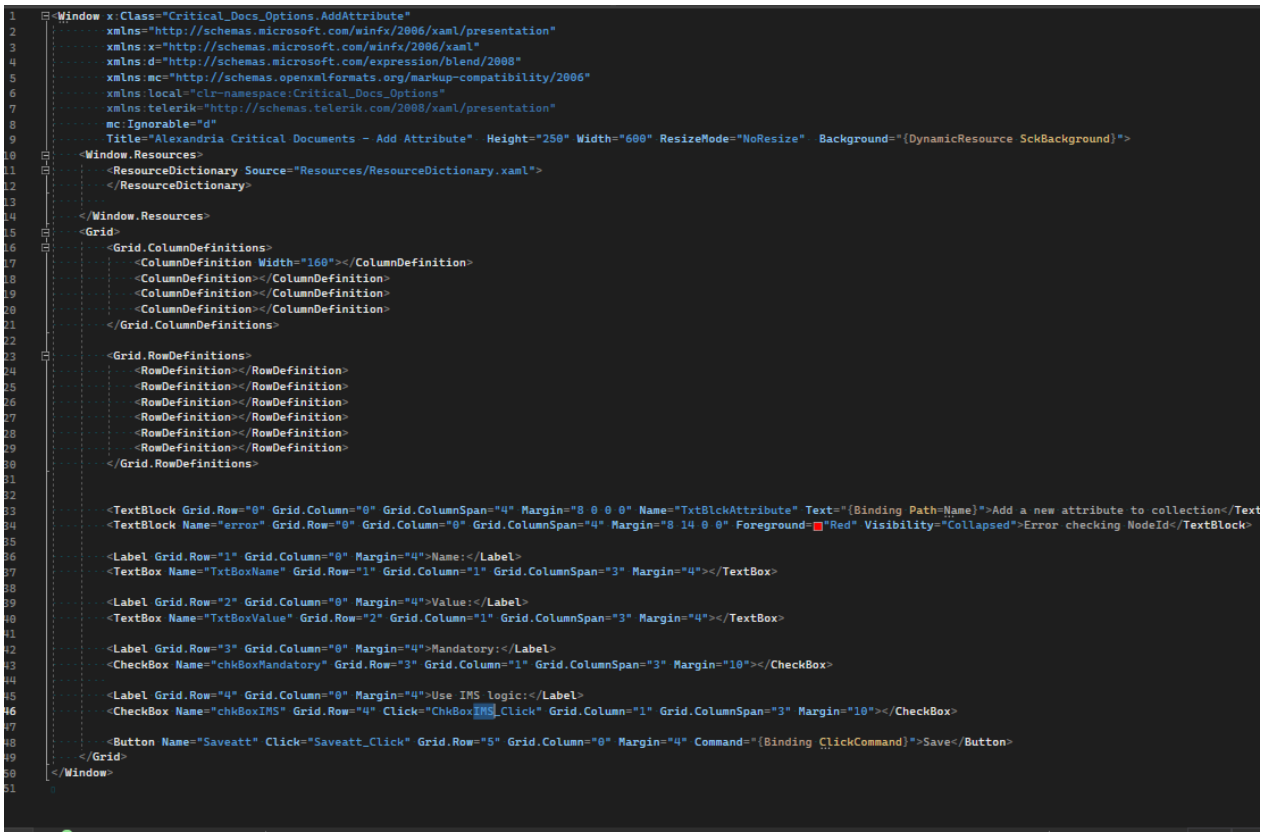
- CommonServiceLocator
- EntityFramework
- Microsoft.Bcl.AsyncInterfaces
- Microsoft.Extensions.Configuration, Microsoft.Extensions.Configuration.Abstractions, Microsoft.Extensions.Configuration.Binder, Microsoft.Extensions.Configuration.FileExtensions, Microsoft.Extensions.Configuration.Json, Microsoft.Extensions.DependencyInjection, Microsoft.Extensions.DependencyInjection.Abstractions, Microsoft.Extensions.DependencyModel, Microsoft.Extensions.FileProviders.Abstractions, Microsoft.Extensions.FileProviders.Physical, Microsoft.Extensions.FileSystemGlobbing, Microsoft.Extensions.Logging, Microsoft.Extensions.Logging.Abstractions, Microsoft.Extensions.Options, Microsoft.Extensions.Primitives
- Newtonsoft.Json
- PresentationCore
- PresentationFramework
- Serilog, Serilog.Settings.AppSettings, Serilog.Settings.Configuration, Serilog.Sinks.Email, Serilog.Sinks.File, Serilog.Sinks.PeriodicBatching, Serilog.Sinks.RollingFile
- System
- System.Buffers
- System.ComponentModel.Composition, System.ComponentModel.DataAnnotations
- System.Configuration
- System.Core
- System.Data, System.Data.DataSetExtensions
- System.Deployment
- System.Diagnostics.DiagnosticSource
- System.Drawing
- System.IO
- System.Management
- System.Memory
- System.Net, System.Net.Http
- System.Numerics, System.Numerics.Vectors
- System.Runtime, System.Runtime.CompilerServices.Unsafe, System.Runtime.InteropServices.RuntimeInformation

- System.Security.Cryptography.Algorithms, System.Security.Cryptography.Encoding, System.Security.Cryptography.Primitives
- System.Text.Encodings.Web, System.Text.Json
- System.Threading.Tasks.Extensions
- System.ValueTuple.



AddAttribute

This xaml and cs contains the logic and design to add an attribute. An attribute has some parts: id, name, values and mandatory. There are 2 textboxes and 2 checkboxes. The first textbox has information about the location of the attribute. The 2nd textbox has the value of that attribute. The first checkbox causes only documents with that attribute to be downloaded. The 2nd textbox ensures that only documents with IMS logic are downloaded. Finally, we have a button that saves the completed data. If you press the button and the attribute cannot be found, a textblock appears above the first textbox representing that the attribute cannot be found.



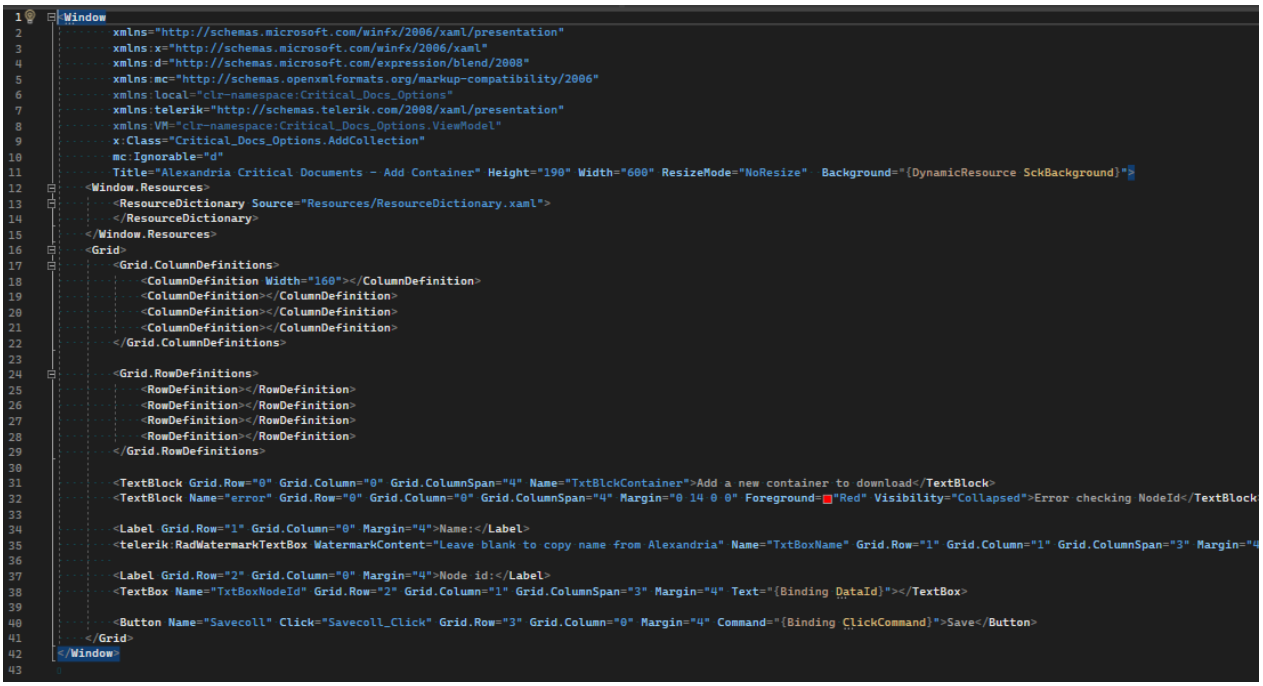
```xml
<Window x:Class="Critical_Docs_Options.AddAttribute"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Critical_Docs_Options"
        xmlns:telerik="http://schemas.telerik.com/2008/xaml/presentation"
        mc:Ignorable="d"
        Title="Alexandria Critical Documents - Add Attribute"  Height="250" Width="600" ResizeMode="NoResize"  Background="{DynamicResource SckBackground}">
    <Window.Resources>
        <ResourceDictionary Source="Resources/ResourceDictionary.xaml">
        </ResourceDictionary>

    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="160"></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>


        <TextBlock Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="4" Margin="8 0 0 0" Name="TxtBlckAttribute" Text="{Binding Path=Name}">Add a new attribute to collection</Text
        <TextBlock Name="error" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="4" Margin="8 14 0 0" Foreground="Red" Visibility="Collapsed">Error checking NodeId</TextBlock>

        <Label Grid.Row="1" Grid.Column="0" Margin="4">Name:</Label>
        <TextBox Name="TxtBoxName" Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3" Margin="4"></TextBox>

        <Label Grid.Row="2" Grid.Column="0" Margin="4">Value:</Label>
        <TextBox Name="TxtBoxValue" Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="3" Margin="4"></TextBox>

        <Label Grid.Row="3" Grid.Column="0" Margin="4">Mandatory:</Label>
        <CheckBox Name="chkBoxMandatory" Grid.Row="3" Grid.Column="1" Grid.ColumnSpan="3" Margin="10"></CheckBox>

        <Label Grid.Row="4" Grid.Column="0" Margin="4">Use IMS logic:</Label>
        <CheckBox Name="chkBoxIMS" Grid.Row="4" Click="ChkBoxIMS_Click" Grid.Column="1" Grid.ColumnSpan="3" Margin="10"></CheckBox>

        <Button Name="Saveatt" Click="Saveatt_Click" Grid.Row="5" Grid.Column="0" Margin="4" Command="{Binding ClickCommand}">Save</Button>
    </Grid>
</Window>
```

```
using Critical_Docs_Options.ViewModel;
using Critical.Docs.Sync.Alexandria;
using Serilog;
using System;
using System.Windows;
using Attribute = Critical_Docs_Sync.Model.Attribute;

namespace Critical_Docs_Options
{
    /// <summary>
    /// Interaction logic for AddAttribute.xaml
    /// </summary>
    6 references
    public partial class AddAttribute : Window
    {
        12 references
        public Attribute attribute { get; set; }

        2 references
        public Attribute Attribute
        {
            get { return attribute; }
        }
        2 references
        public AddAttribute(Attribute att)
        {

            InitializeComponent();
            /*var viewModel = new AttributeViewModel(att);
            viewModel.Name = att.Name;
            viewModel.Value = att.Value;
            this.DataContext = viewModel;*/
            attribute = att;
            TxtBoxName.Text = attribute.Name;
            TxtBoxValue.Text = attribute.Value;
            chkBoxMandatory.IsChecked = attribute.Mandatory;
            if (attribute.Name == "IMS Logic" && attribute.Value == "True" && attribute.Mandatory == true)
            {
                chkBoxIMS.IsChecked = true;
                ChkBoxIMS_Click(chkBoxIMS, new RoutedEventArgs());
            }
        }

        1 reference
        private async void Saveatt_Click(object sender, RoutedEventArgs e)
        {
            attribute.Name = TxtBoxName.Text;
            attribute.Value = TxtBoxValue.Text;
            attribute.Mandatory = Convert.ToBoolean(chkBoxMandatory.IsChecked);

            if (chkBoxIMS.IsChecked != true)
            {
                try
                {
                    string result = await AlexandriaWorkflow.CheckCategory(attribute);
```

```
                if (result != null)
                {
                    DialogResult = true;
                }
                else
                {
                    MessageBox.Show(result, "Error", MessageBoxButton.OK, MessageBoxImage.Error);
                }
            }
            catch (Exception ex)
            {
                Log.Error("Attribute not found: {error}", ex);
                MessageBox.Show("Attribute not found in Alexandria", "Error", MessageBoxButton.OK, MessageBoxImage.Error);
            }
        }
        else
        {
            DialogResult = true;
        }
    }

    2 references
    private void ChkBoxIMS_Click(object sender, RoutedEventArgs e)
    {
        if (chkBoxIMS.IsChecked == true)
        {
            TxtBoxName.Text = "IMS Logic";
            TxtBoxName.IsEnabled = false;
            TxtBoxValue.Text = "True";
            TxtBoxValue.IsEnabled = false;
            chkBoxMandatory.IsChecked = true;
            chkBoxMandatory.IsEnabled = false;
        }
        else
        {
            TxtBoxName.Text = "";
            TxtBoxName.IsEnabled = true;
            TxtBoxValue.Text = "";
            TxtBoxValue.IsEnabled = true;
            chkBoxMandatory.IsChecked = false;
            chkBoxMandatory.IsEnabled = true;
        }
    }
}
}
```

AddCollection

This xaml and cs contains the logic and design to add a container (from where you download the documents) the container uses a name and nodeid. There are 2 textboxes: name and nodeid. The name uses a watermark textbox. If you don't fill this in and the nodeid exists, then the name (in Alexandria) with that nodeid is used. The next textbox is used to search if that nodeid exists in Alexandria. Next, we have a button that stores the container. If you press the button and the nodeid does not exist, then a textblock is displayed that tells the user that the id does not exist.



```
<Window
        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:Critical_Docs_Options"
        xmlns:telerik="http://schemas.telerik.com/2008/xaml/presentation"
        xmlns:VM="clr-namespace:Critical_Docs_Options.ViewModel"
        x:Class="Critical_Docs_Options.AddCollection"
        mc:Ignorable="d"
        Title="Alexandria Critical Documents - Add Container" Height="190" Width="600" ResizeMode="NoResize"  Background="{DynamicResource SckBackground}">
    <Window.Resources>
        <ResourceDictionary Source="Resources/ResourceDictionary.xaml">
        </ResourceDictionary>
    </Window.Resources>
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="160"></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>

        <TextBlock Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="4" Name="TxtBlckContainer">Add a new container to download</TextBlock>
        <TextBlock Name="error" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="4" Margin="0 14 0 0" Foreground="Red" Visibility="Collapsed">Error checking NodeId</TextBlock

        <Label Grid.Row="1" Grid.Column="0" Margin="4">Name:</Label>
        <telerik:RadWatermarkTextBox WatermarkContent="Leave blank to copy name from Alexandria" Name="TxtBoxName" Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="3" Margin="4

        <Label Grid.Row="2" Grid.Column="0" Margin="4">Node id:</Label>
        <TextBox Name="TxtBoxNodeId" Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="3" Margin="4" Text="{Binding DataId}"></TextBox>

        <Button Name="Savecoll" Click="Savecoll_Click" Grid.Row="3" Grid.Column="0" Margin="4" Command="{Binding ClickCommand}">Save</Button>
    </Grid>
</Window>
```

```
using Critical_Docs_Sync.Model;
using System;
using System.Windows;
using Critical.Docs.Business.Logic;
using Critical.Docs.Sync.Alexandria;
using Critical_Docs_Options.ViewModel;
using Critical_Docs_Sync;
using System.IO;
using System.Collections.Generic;
using System.Linq;

namespace Critical_Docs_Options
{
    /// <summary>
    /// Interaction logic for AddCollection.xaml
    /// </summary>
    0 references
    public partial class AddCollection : Window
    {
        OptionsWindow optionsWindow = new OptionsWindow();
        9 references
        public AlexandriaSourceOptions _Collection { get; set; }

        1 reference
        public AlexandriaSourceOptions Collection
        {
            get { return _Collection; }
        }

        2 references
        public AddCollection(AlexandriaSourceOptions options)
        {

            InitializeComponent();
            var viewModel = new CollectionViewModel(options);
            viewModel.Name = options.Name;
            viewModel.DataId = options.DataId;
            viewModel.Attributes = options.Attributes;
            this.DataContext = viewModel;
            _Collection = options;
            TxtBoxName.Text = viewModel.Name;
            TxtBoxName.Text = DocumentName.GetSafeFilename(TxtBoxName.Text);
            TxtBoxNodeId.Text = viewModel.DataId.ToString();
        }

        1 reference
        private async void Savecoll_Click(object sender, RoutedEventArgs e)
        {
            OptionsWindow options = new OptionsWindow();
            NodeResult result = await AlexandriaWorkflow.GetNodeItem(TxtBoxNodeId.Text);
            if (result != null && result.Code != 500 && result.Message != "false")
            {
                if (TxtBoxName.Text != "")
                {
                    var test = optionsWindow.newCollection.FirstOrDefault(x => x.Name == TxtBoxName.Text);
```

```
                    if (test == null)
                    {
                        TxtBoxName.Text = DocumentName.GetSafeFilename(TxtBoxName.Text);

                        _Collection.Name = TxtBoxName.Text;
                        OptionsContext context = new OptionsContext();
                    }
                    else
                    {

                    }

                }
                else
                {
                    _Collection.Name= result.Name;
                    string finalResult = DocumentName.GetSafeFilename(result.Name);
                    _Collection.Name = finalResult;
                }
                if (_Collection.Attributes == null)
                {
                    _Collection.Attributes = new System.Collections.Generic.List<Critical_Docs_Sync.Model.Attribute>();
                }
                _Collection.DataId = Convert.ToInt32(TxtBoxNodeId.Text);
                DialogResult = true;
            }
            else
            {
                error.Visibility = Visibility.Visible;
            }
        }
    }
}

```

Appsettings.json

This file contains configuration information about Serilog.

```
{
  "Serilog": {
    "Using": ["Serilog.Sinks.Email" ],
    "WriteTo": [
      {
        "Name": "Email",
        "Args": {
          "RestrictedToMinimumLevel": "Error",
          "FromEmail": "no-reply@sckcen.be",
          "ToEmail": "Jetze.Luyten@sckcen.be",
          "MailServer": "smtp.sckcen.be",
          "MailSubject": "Error log"
        }
      }
    ]
  }
}
```

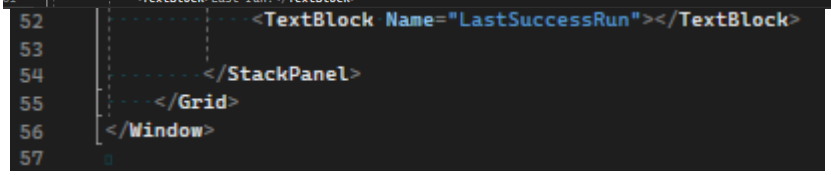
Using is used to select which packages you implement. WriteTo is used to write out logging. The name Email means that logging is written to this. This takes some arguments:

- The minimum level that is in the log
- Where the email comes from
- To where that the email is sent
- Along which server the email is sent
- The title of the email

MainWindow

This xaml and cs is the starting point of the solution. There are 3 buttons: one where the user configures the different options, one where the user can start and stop the automatic download (if a certain checkbox is checked) and finally we have a button where the user can start the forced download.

```xaml
<Window x:Name="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"
        x:Class="Critical_Docs_Options.MainWindow"
        mc:Ignorable="d"
        Title="Alexandria Critical Documents" Height="400" Width="550" Closing="Window_Closing" Background="{DynamicResource SckBackground}" >
    <Window.Resources>
        <ResourceDictionary Source="Resources/ResourceDictionary.xaml">
        </ResourceDictionary>
    </Window.Resources>

        <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="3*"></RowDefinition>
            <RowDefinition Height="2*"></RowDefinition>
            <RowDefinition Height="2*"></RowDefinition>
            <RowDefinition Height="2*"></RowDefinition>
            <RowDefinition Height="3*"></RowDefinition>
        </Grid.RowDefinitions>
        <TextBlock Grid.Row="0" Name="txtBlockIntroText" TextWrapping="Wrap" Margin="5" Padding="0,10,0,0">This tool will download folders and documents from Alexandria to a
        <Button Grid.Row="1" Name="btnOptions" Click="btnOptions_Click" Margin="5">Change options</Button>
        <Button Grid.Row="2" Name="btnService" Click="btnService_Click" Margin="5">Start/Stop</Button>
        <Button Grid.Row="3" Name="btnForce" Click="btnForce_Click" Visibility="Visible" Margin="5">Force download</Button>
        <StackPanel  Name="prgsDownload" Grid.Row="3" Orientation="Vertical" Visibility="Collapsed" Margin="5,0">
            <TextBlock Name="txtBlckInfo" >
                <TextBlock.Triggers>
                    <EventTrigger RoutedEvent="TextBlock.Loaded">
                        <BeginStoryboard>
                            <Storyboard>
                                <ObjectAnimationUsingKeyFrames Storyboard.TargetProperty="Text"
                                                               Duration="00:00:02.0"
                                                               RepeatBehavior="Forever">
                                    <DiscreteObjectKeyFrame KeyTime="00:00:00.00" Value="Downloading"/>
                                    <DiscreteObjectKeyFrame KeyTime="00:00:00.50" Value="Downloading."/>
                                    <DiscreteObjectKeyFrame KeyTime="00:00:01.00" Value="Downloading.."/>
                                    <DiscreteObjectKeyFrame KeyTime="00:00:01.50" Value="Downloading..."/>
                                </ObjectAnimationUsingKeyFrames>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                </TextBlock.Triggers></TextBlock>
            <ProgressBar IsIndeterminate="True" Height="20"/>

        </StackPanel>
        <StackPanel  Name="LastRuns" Grid.Row="4" Orientation="Vertical" Visibility="Visible" Margin="5,0,5,5">

            <TextBlock>Last run started:</TextBlock>
            <TextBlock Name="LastRun"></TextBlock>
            <TextBlock>Last run:</TextBlock>
```



```
using Critical.Docs.Business.Logic;
using Critical.Docs.Sync.Alexandria;
using Critical_Docs_Sync;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using Serilog;
using System;
using System.Collections.Generic;
using System.Deployment.Application;
using System.Diagnostics;
using System.Drawing;
using System.Reflection;
using System.Runtime.Remoting.Contexts;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Documents;
using System.Windows.Forms;
using AppUI = System.Windows.Application;
using Cursors = System.Windows.Input.Cursors;
using MessageBox = System.Windows.MessageBox;

namespace Critical_Docs_Options
{
    /// <inheritdoc cref="Window" />
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    // ReSharper disable once UnusedMember.Global
    // ReSharper disable once RedundantExtendsListEntry
    4 references
    public partial class MainWindow : Window
    {
        private int _processId;
        private string _lastRunText = "";
        private NotifyIcon ni = new NotifyIcon
        {
            Icon = new Icon(AppUI.GetResourceStream(new Uri("pack://application:,,,/Resources/favicon (2).ico"))
                                ?.Stream ?? throw new InvalidOperationException()),
            Visible = true
        };
        private bool interrupted;


        // ReSharper disable once FieldCanBeMadeReadOnly.Local

        0 references
        public MainWindow()
        {
            InitializeComponent();

            mainWindow.Title = "Alexandria Critical Documents - v" + getRunningVersion();

```

```
53
54
55              ni.DoubleClick +=
56              delegate
57              {
58                  Show();
59                  WindowState = WindowState.Normal;
60                  SetLastRuns();
61              };
62
63              var contextMenu = new ContextMenu();
64              var menuItemOpen = new MenuItem();
65              var menuItemExit = new MenuItem();
66              var menuItemLastRun = new MenuItem();
67              var menuItemLastSuccessRun = new MenuItem();
68
69              menuItemLastRun.Index = 0;
70              menuItemLastRun.Text = _lastRunText;
71              contextMenu.MenuItems.Add(menuItemLastRun);
72
73              menuItemLastSuccessRun.Index = 1;
74              menuItemLastSuccessRun.Text = DocumentProcessing._lastSuccess;
75              contextMenu.MenuItems.Add(menuItemLastSuccessRun);
76
77              contextMenu.MenuItems.Add(new MenuItem("-"));
78
79              menuItemOpen.Index = 3;
80              menuItemOpen.Text = @"Open Critical Docs Tool";
81              menuItemOpen.Click += delegate
82              {
83                  Show();
84                  WindowState = WindowState.Normal;
85                  SetLastRuns();
86              };
87              contextMenu.MenuItems.Add(menuItemOpen);
88              contextMenu.MenuItems.Add(new MenuItem("-"));
89
90              menuItemExit.Index = 5;
91              menuItemExit.Text = @"Exit Critical Docs Tool";
92              menuItemExit.Click += delegate
93              {
94                  Close();
95
96              };
97
98              contextMenu.MenuItems.Add(menuItemExit);
99
100             contextMenu.Popup += delegate
101             {
102                 SetLastRuns();
103                 menuItemLastRun.Text = _lastRunText;
```

```
104                 menuItemLastSuccessRun.Text = DocumentProcessing._lastSuccess;
105             };
106
107             using (OptionsContext context = new OptionsContext())
108             {
109                 DocumentProcessing._lastSuccess = context.GetOption("lastSuccessRun").Value;
110             }
111
112             ni.ContextMenu = contextMenu;
113
114             // Set up how the form should be displayed.
115             SetLastRuns();
116             SetLastSuccessfulRuns();
117
118             SetServiceButton();
119         }
120
            0 references
121         protected override void OnStateChanged(EventArgs e)
122         {
123             if (WindowState == WindowState.Minimized)
124                 Hide();
125
126             base.OnStateChanged(e);
127         }
128
            3 references
129         private void SetServiceButton()
130         {
131             string schedule;
132
133             using (OptionsContext context = new OptionsContext())
134             {
135                 schedule = context.GetOption("schedule").Value;
136             }
137             if (schedule == null || schedule == "False")
138             {
139                 btnService.IsEnabled = false;
140                 btnService.Content = "Automatic download not enabled";
141             }
142             else
143             {
144                 btnService.IsEnabled = true;
145                 Process[] processName = Process.GetProcessesByName("Alexandria CDST");
146                 if (processName.Length == 0)
147                 {
148                     btnService.Content = "Start automatic download";
149                     _processId = 0;
150                 }
151                 else
152                 {
153                     btnService.Content = "Stop automatic download";
154                     _processId = processName[0].Id;
```

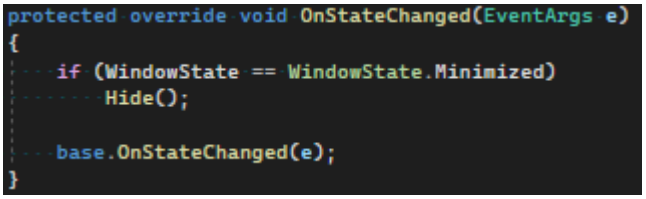
```
155                 }
156             }
157         }
158
            1 reference
159         private void btnOptions_Click(object sender, RoutedEventArgs e)
160         {
161             OptionsWindow options = new OptionsWindow();
162             options.ShowDialog();
163             SetServiceButton();
164         }
165
            4 references
166         private void SetLastRuns()
167         {
168             using (OptionsContext context = new OptionsContext())
169             {
170                 string last = context.GetOption("lastRun").Value;
171                 _lastRunText = $"Last run started: {last}";
172
173                 LastRun.Text = "";
174                 LastRun.Inlines.Add(new Run(last) { FontWeight = FontWeights.Bold });
175             }
176         }
177
            2 references
178         private void SetRun()
179         {
180             using (OptionsContext context = new OptionsContext())
181             {
182                 string last = context.GetOption("lastRun").Value;
183                 _lastRunText = $"Last run started: {last}";
184
185                 LastRun.Text = "";
186                 LastRun.Inlines.Add(new Run(DateTime.Now.ToString("dd-MM-yyyy HH:mm")) { FontWeight = FontWeights.Bold });
187             }
188         }
189
            3 references
190         private void SetLastSuccessfulRuns()
191         {
192             using (OptionsContext context = new OptionsContext())
193             {
194                 if (DownloadDocument.countErrors > 0)
195                 {
196                     DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";
197
198                     context.UpdateOptions(new List<KeyValuePair<string, string>>
199                             {
200                                 new KeyValuePair<string, string>("lastSuccessRun",
201                                     DocumentProcessing._lastSuccess)
202                             });
203                 }
204
```

```
                LastSuccessRun.Text = "";
                LastSuccessRun.Inlines.Add(new Run(DocumentProcessing._lastSuccess) { FontWeight = FontWeights.Bold });

            }
        }

        1 reference
        private void btnService_Click(object sender, RoutedEventArgs e)
        {

            if (btnService.Content.ToString() == "Start automatic download")
            {
                Process p = new Process
                {
                    StartInfo = { FileName = "Alexandria CDST", WindowStyle = ProcessWindowStyle.Hidden }
                };
                p.Start();
                SetRun();
            }
            else
            {
                MessageBoxResult result = MessageBox.Show("Are you sure you want to stop the automatic download?", "Warning!", MessageBoxButton.YesNo, MessageBoxImage.Warnir
                switch (result)
                {
                    case MessageBoxResult.Yes:
                        DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " interrupted by user";
                        OptionsContext context = new OptionsContext();

                        context.UpdateOptions(new List<KeyValuePair<string, string>>
                            {
                                new KeyValuePair<string, string>("lastSuccessRun",
                                    DocumentProcessing._lastSuccess)
                            });
                        SetLastSuccessfulRuns();
                        Process.GetProcessById(_processId).Kill();
                        break;
                    case MessageBoxResult.No:
                        break;
                }
            }
            Thread.Sleep(500);

            SetServiceButton();
        }

        1 reference
        private async void btnForce_Click(object sender, RoutedEventArgs e)
        {
            SetRun();
            var configuration = new ConfigurationBuilder()
                .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
                .Build();
```

```
255             ILoggerFactory loggerFactory = LoggerFactory.Create(builder =>
256             {
257                 string localDirectory;
258                 using (OptionsContext optionsContext = new OptionsContext())
259                 {
260                     localDirectory = optionsContext.GetOption("localDirectory").Value;
261                 }
262                 Log.Logger = new LoggerConfiguration()
263                     .MinimumLevel.Information()
264                     .WriteTo.RollingFile(localDirectory + "\\CDST.log", shared: true)
265                     .ReadFrom.Configuration(configuration)
266                     .CreateLogger();
267             });
268
269             btnForce.Visibility = Visibility.Collapsed;
270             prgsDownload.Visibility = Visibility.Visible;
271             btnOptions.IsEnabled = false;
272             btnService.IsEnabled = false;
273             Cursor = Cursors.Wait;
274             DocumentProcessing._inProgress = true;
275             await Task.Run(() => DocumentProcessing.ProcessRequest(true));
276
277
278             btnForce.Visibility = Visibility.Visible;
279             prgsDownload.Visibility = Visibility.Collapsed;
280             btnOptions.IsEnabled = true;
281             btnService.IsEnabled = true;
282             Cursor = null;
283             DocumentProcessing._inProgress = false;
284
285             SetLastSuccessfulRuns();
286             WriteToUi("Download finished successfully");
287         }
288
289         1 reference
289         public void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
290         {
291             Log.Information("Closing application");
292             ni.Visible = false;
293             if (DocumentProcessing._inProgress)
294             {
295                 MessageBoxResult boxResult = MessageBox.Show("An operation is still running, Are you certain you want to close this window? This will end the current runnin
296                 if (boxResult == MessageBoxResult.No)
297                 {
298
299                     e.Cancel = true;
300                 }
301                 else
302                 {
303                     Process.GetCurrentProcess().Kill();
304                 }
305             }
306             else
307             {
308                 Process.GetCurrentProcess().Kill();
309             }
310
311         }
312
313         1 reference
313         public void WriteToUi(string message)
314         {
315             AppUI.Current.Dispatcher.Invoke(
316                 () =>
317                 {
318                     MessageBox.Show(message, "Alexandria CDST");
319                 });
320         }
321
322         1 reference
322         private Version getRunningVersion()
323         {
324             try
325             {
326                 return ApplicationDeployment.CurrentDeployment.CurrentVersion;
327             }
328             catch (Exception)
329             {
330                 return Assembly.GetExecutingAssembly().GetName().Version;
331             }
332         }
333     }
334 }
335
```

OnStateChanged(EventArgs e)

This class is used to hide a window that's minimized for the user.



```
protected override void OnStateChanged(EventArgs e)
{
    if (WindowState == WindowState.Minimized)
        Hide();

    base.OnStateChanged(e);
}
```

SetServiceButton

This class is used to enable the status of the btnService. We create a variable "schedule" of type string. Then we retrieve the information from the "schedule" option, which can be found in the database under Options. If schedule is null or False, then we disable the btnService button and change the content to "Automatic download not enabled". Otherwise, we enable the btnService button and retrieve a list of processes named "Alexandria CDST". If the list of processes is empty, then we change the content of btnService to "Start automatic download" and set _processId to 0. Otherwise, we set the content to "Stop automatic download" and set the _processId to the Id of the first process of the process list.

```
private void SetServiceButton()
{
    string schedule;

    using (OptionsContext context = new OptionsContext())
    {
        schedule = context.GetOption("schedule").Value;
    }
    if (schedule == null || schedule == "False")
    {
        btnService.IsEnabled = false;
        btnService.Content = "Automatic download not enabled";
    }
    else
    {
        btnService.IsEnabled = true;
        Process[] processName = Process.GetProcessesByName("Alexandria CDST");
        if (processName.Length == 0)
        {
            btnService.Content = "Start automatic download";
            _processId = 0;
        }
        else
        {
            btnService.Content = "Stop automatic download";
            _processId = processName[0].Id;
        }
    }
}
```

BtnOptions_Click

If the user clicks on the btnOptions button, then OptionsWindow.xaml opens.

```
private void btnOptions_Click(object sender, RoutedEventArgs e)
{
    OptionsWindow options = new OptionsWindow();
    options.ShowDialog();
    SetServiceButton();
}
```

SetLastRuns

This class is used to display information about the last run. Here we use the database and retrieve the value of the last run and place it in a variable with type string. Then we put that information in _lastRunText and show it in bold to the user.

```
private void SetLastRuns()
{
    using (OptionsContext context = new OptionsContext())
    {
        string last = context.GetOption("lastRun").Value;
        _lastRunText = $"Last run started: {last}";

        LastRun.Text = "";
        LastRun.Inlines.Add(new Run(last) { FontWeight = FontWeights.Bold });
    }
}
```

SetRun

SetRun is the same as SetLastRuns, but the date and time of that moment gets shown.

```
private void SetRun()
{
    using (OptionsContext context = new OptionsContext())
    {
        string last = context.GetOption("lastRun").Value;
        _lastRunText = $"Last run started: {last}";

        LastRun.Text = "";
        LastRun.Inlines.Add(new Run(DateTime.Now.ToString("dd-MM-yyyy HH:mm")) { FontWeight = FontWeights.Bold });
    }
}
```

SetLastSucessRuns

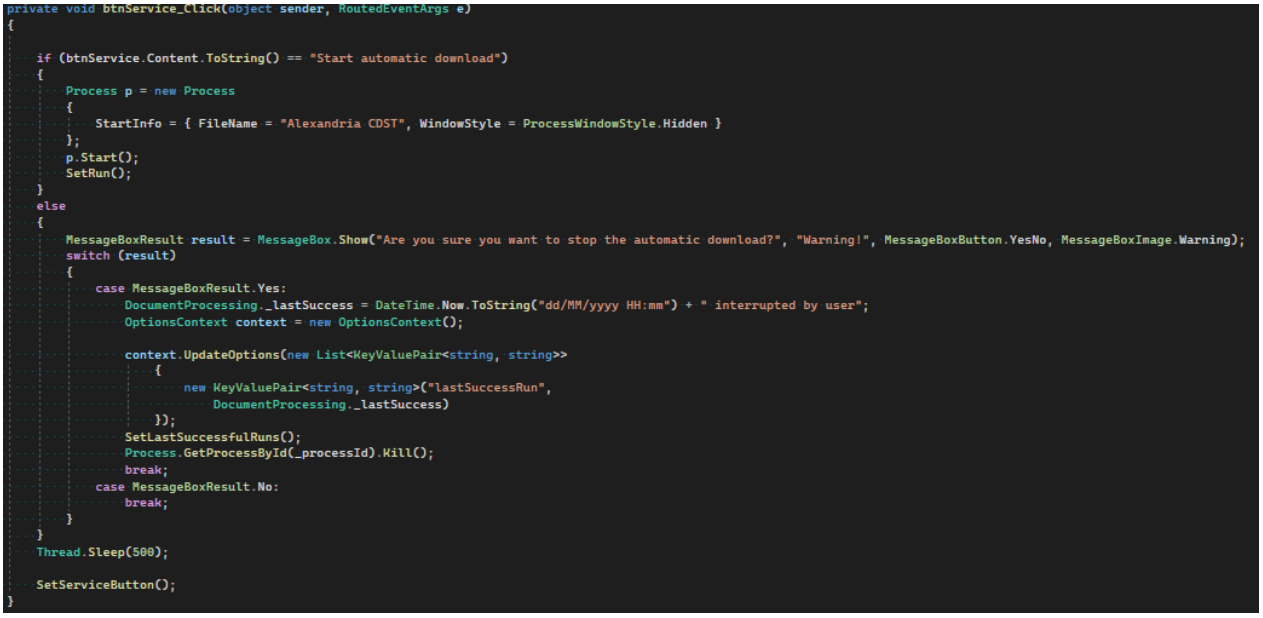
This method is used to show whether the result is finished, with or without errors, or interrupted. If the number of errors is greater than 0, you get the date and time of that moment plus "finished with errors".

```
private void SetLastSuccessfulRuns()
{
    using (OptionsContext context = new OptionsContext())
    {
        if (DownloadDocument.countErrors > 0)
        {
            DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";

            context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                        DocumentProcessing._lastSuccess)
                    });
        }

        LastSuccessRun.Text = "";
        LastSuccessRun.Inlines.Add(new Run(DocumentProcessing._lastSuccess) { FontWeight = FontWeights.Bold });

    }
}
```

BtnService_Click

If the user presses the button and the text on btnService is "Start automatic download", then we start a new process of "Alexandria CDST" and set the runs. Otherwise, the user gets a message box asking if the user is sure they want to stop the download. If they choose no, then the download continues. If yes, then we set the last successful run to the date and time of that moment plus "interrupted by user". Then we

wait 0.5 seconds and change the SetServiceButton.



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

    if (btnService.Content.ToString() == "Start automatic download")
    {
        Process p = new Process
        {
            StartInfo = { FileName = "Alexandria CDST", WindowStyle = ProcessWindowStyle.Hidden }
        };
        p.Start();
        SetRun();
    }
    else
    {
        MessageBoxResult result = MessageBox.Show("Are you sure you want to stop the automatic download?", "Warning!", MessageBoxButton.YesNo, MessageBoxImage.Warning);
        switch (result)
        {
            case MessageBoxResult.Yes:
                DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " interrupted by user";
                OptionsContext context = new OptionsContext();

                context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            DocumentProcessing._lastSuccess)
                    });
                SetLastSuccessfulRuns();
                Process.GetProcessById(_processId).Kill();
                break;
            case MessageBoxResult.No:
                break;
        }
    }
    Thread.Sleep(500);

    SetServiceButton();
}
```

BtnForce_Click

When the user presses the button, we set the start of the last run to this moment. Then we use the information from appsettings.json to log in a file. This file contains logging with minimum level of Information. After this we collapsed btnForce, show prgsDownload, disable btnOptions and btnService, change the cursor to the wait sign, change _inprogress (what is in DocumentProcessing) to true and go to the workflow with forceDownload true as force. When the download is done then we show btnForce back and collapse prgsDownload. Then we enable btnOptions and btnService, reset the cursor and set _inProgress to false. Finally, we set the last successful run at this time and let the user know that the download is finished.

```
private async void btnForce_Click(object sender, RoutedEventArgs e)
{
    SetRun();
    var configuration = new ConfigurationBuilder()
        .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
        .Build();
    ILoggerFactory loggerFactory = LoggerFactory.Create(builder =>
    {
        string localDirectory;
        using (OptionsContext optionsContext = new OptionsContext())
        {
            localDirectory = optionsContext.GetOption("localDirectory").Value;
        }
        Log.Logger = new LoggerConfiguration()
            .MinimumLevel.Information()
            .WriteTo.RollingFile(localDirectory + "\\CDST.log", shared: true)
            .ReadFrom.Configuration(configuration)
            .CreateLogger();
    });

    btnForce.Visibility = Visibility.Collapsed;
    prgsDownload.Visibility = Visibility.Visible;
    btnOptions.IsEnabled = false;
    btnService.IsEnabled = false;
    Cursor = Cursors.Wait;
    DocumentProcessing._inProgress = true;
    await Task.Run(() => DocumentProcessing.ProcessRequest(true));


    btnForce.Visibility = Visibility.Visible;
    prgsDownload.Visibility = Visibility.Collapsed;
    btnOptions.IsEnabled = true;
    btnService.IsEnabled = true;
    Cursor = null;
    DocumentProcessing._inProgress = false;

    SetLastSuccessfulRuns();
    WriteToUi("Download finished successfully");
}
```

Window_Closing

If the user closes the application, then this is logged and we make the NotifyIcon invisible. If the download is in progress, then the user is asked if they are sure they want to stop the process. If no, then nothing happens. If yes, then we stop the instantaneous process. If no, then we stop the instantaneous process.

```
public void Window_Closing(object sender, System.ComponentModel.CancelEventArgs e)
{
    Log.Information("Closing application");
    ni.Visible = false;
    if (DocumentProcessing._inProgress)
    {
        MessageBoxResult boxResult = MessageBox.Show("An operation is still running, Are you certain you want to close this window? This will end the current running process.", "Operation in progress", MessageBoxButton.YesNo, MessageBoxImage.Stop);
        if (boxResult == MessageBoxResult.No)
        {

            e.Cancel = true;
        }
        else
        {
            Process.GetCurrentProcess().Kill();
        }
    }
    else
    {
        Process.GetCurrentProcess().Kill();
    }
}
```

WriteToUi(string message)

This method is used to show information to the user using a messagebox.

```
public void WriteToUi(string message)
{
    AppUI.Current.Dispatcher.Invoke(
        () =>
        {
            MessageBox.Show(message, "Alexandria CDST");
        });
}
```

GetRunningVersion

This method is used to show the version of the application to the user.

```
private Version getRunningVersion()
{
    try
    {
        return ApplicationDeployment.CurrentDeployment.CurrentVersion;
    }
    catch (Exception)
    {
        return Assembly.GetExecutingAssembly().GetName().Version;
    }
}
```

OptionsWindow.xaml

This xaml and cs contains the logic to configure the different options. The xaml contains 3 parts: the connection configuration, the synchronization configuration and the document configuration. There is a table with information about the container and attributes of that container. There is also a button that adds a container and takes the user to the AddCollection.xaml go. Finally, there is a save button that saves all configurations.

Connectie-configuratie

This section has 2 textboxes. The first textbox contains the information to get an OTDS ticket. The 2nd textbox contains the information to see if Alexandria is down or not.

Synchronisatie-configuratie

This section has a checkbox and a textbox. The checkbox looks at the switching of the "automatic download" button. The textbox has information about the interval in hours used between each automatic download.

Document-configuratie

This component has a textbox. The textbox contains information about the location where that container is placed and all documents are downloaded.

```xaml
<Window x:Class="Critical_Docs_Options.OptionsWindow"
        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:telerik="http://schemas.telerik.com/2008/xaml/presentation"
        mc:Ignorable="d"
        Title="Alexandria Critical Documents - Options" Height="500" Width="800"  Background="{DynamicResource SckBackground}">
    <Window.Resources>
        <ResourceDictionary Source="Resources/ResourceDictionary.xaml">
        </ResourceDictionary>
    </Window.Resources>
        <ScrollViewer HorizontalScrollBarVisibility="Auto">
        <Grid>
            <Grid.Resources>
                <DataTemplate x:Key="RowDetailsTemplate">
                    <telerik:RadGridView Name="AttributeGrid"
                                         ItemsSource="{Binding Attributes}"
                                         AutoGenerateColumns="False"
                                         IsReadOnly="True"
                                         ShowGroupPanel="False"
                                         RowIndicatorVisibility="Collapsed"
                                         IsFilteringAllowed="False">

                        <telerik:RadGridView.Columns>
                            <telerik:GridViewDataColumn Header="Attribute Name" DataMemberBinding="{Binding Name}"/>
                            <telerik:GridViewDataColumn Header="Attribute Value" DataMemberBinding="{Binding Value}"/>
                            <telerik:GridViewDataColumn Header="Mandatory" DataMemberBinding="{Binding Mandatory}"/>
                            <telerik:GridViewColumn Header="Edit" >
                                <telerik:GridViewColumn.CellTemplate>
                                    <DataTemplate>
                                        <Button Click="btnEditAttribute_Click">Edit</Button>
                                    </DataTemplate>
                                </telerik:GridViewColumn.CellTemplate>
                            </telerik:GridViewColumn>
                            <telerik:GridViewColumn Header="Delete" >
                                <telerik:GridViewColumn.CellTemplate>
                                    <DataTemplate>
                                        <Button Click="btnDeleteAttribute_Click">Delete</Button>
                                    </DataTemplate>
                                </telerik:GridViewColumn.CellTemplate>
                            </telerik:GridViewColumn>
                        </telerik:RadGridView.Columns>
                    </telerik:RadGridView>
                </DataTemplate>
            </Grid.Resources>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="10"></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
```

```
            <ColumnDefinition Width="10"></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="10"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="auto "></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="5"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="20"></RowDefinition>
            <RowDefinition Height="auto"></RowDefinition>
            <RowDefinition Height="10"></RowDefinition>
        </Grid.RowDefinitions>
        <StackPanel Orientation="Vertical" Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="4" Margin="0 0 0 10" >
            <TextBlock Text="Configuration of your Critical Documents Syncronisation Tool."></TextBlock>
            <TextBlock Text="Settings are stored locally under '[your personal folder]/AppData/Roaming/AlexandriaCriticalDocs/'"></TextBlock>
        </StackPanel>

        <TextBlock Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="4" Margin="0 0 0 10" >
            <TextBlock.Inlines>
                <Run FontWeight="Bold" TextDecorations="Underline" Text="Connection setings" />
            </TextBlock.Inlines>
        </TextBlock>

        <Grid Grid.Row="3" Grid.Column="1" Grid.ColumnSpan="4">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="150"></ColumnDefinition>
                <ColumnDefinition></ColumnDefinition>
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="auto"></RowDefinition>
                <RowDefinition Height="auto"></RowDefinition>
                <RowDefinition></RowDefinition>
            </Grid.RowDefinitions>

            <Label Grid.Row="0" Grid.Column="0" Margin="0 0 10 10">OTDS Base URL :</Label>
            <TextBox Grid.Row="0" Grid.Column="1" Margin="0 0 0 10" Name="txtBoxOtdsUrl"></TextBox>

            <Label Grid.Row="1" Grid.Column="0" Margin="0,0,5,10" Width="NaN">Alexandria Base URL :</Label>
            <TextBox Grid.Row="1" Grid.Column="1" Margin="0 0 0 10" Name="txtBoxAlexandriaUrl"></TextBox>
        </Grid>

        <TextBlock Grid.Row="4" Grid.Column="1" Grid.ColumnSpan="4" Margin="0 0 0 10" >
            <TextBlock.Inlines>
                <Run FontWeight="Bold" TextDecorations="Underline" Text="Synchronisation settings" />
            </TextBlock.Inlines>
```

```
104            </TextBlock>
105
106            <Grid Grid.Row="5" Grid.Column="1" Grid.ColumnSpan="4">
107                <Grid.ColumnDefinitions>
108                    <ColumnDefinition Width="150"></ColumnDefinition>
109                    <ColumnDefinition></ColumnDefinition>
110                </Grid.ColumnDefinitions>
111                <Grid.RowDefinitions>
112                    <RowDefinition Height="auto"></RowDefinition>
113                    <RowDefinition Height="auto"></RowDefinition>
114                </Grid.RowDefinitions>
115
116                <Label Grid.Row="0" Grid.Column="0" Margin="0 0 10 10">Periodic download :</Label>
117                <CheckBox Grid.Row="0" Grid.Column="1" Margin="0 5 0 5" Name="chkPeriodic"></CheckBox>
118
119                <Label Grid.Row="1" Grid.Column="0" Margin="0 0 10 10">Sync interval period (h) :</Label>
120                <TextBox Grid.Row="1" Grid.Column="1" Margin="0 0 0 10" Name="txtBoxSyncTime" PreviewTextInput="NumberValidationTextBox"></TextBox>
121            </Grid>
122
123            <TextBlock Grid.Row="6" Grid.Column="1" Grid.ColumnSpan="4" Margin="0 0 0 10" >
124                <TextBlock.Inlines>
125                    <Run FontWeight="Bold" TextDecorations="Underline" Text="Document settings" />
126                </TextBlock.Inlines>
127            </TextBlock>
128
129            <Grid Grid.Row="7" Grid.Column="1" Grid.ColumnSpan="4">
130                <Grid.ColumnDefinitions>
131                    <ColumnDefinition Width="150"></ColumnDefinition>
132                    <ColumnDefinition></ColumnDefinition>
133                </Grid.ColumnDefinitions>
134                <Grid.RowDefinitions>
135                    <RowDefinition Height="auto"></RowDefinition>
136                </Grid.RowDefinitions>
137
138                <Label Grid.Row="0" Grid.Column="0" Margin="0 0 10 10">Local directory :</Label>
139                <TextBox Grid.Row="0" Grid.Column="1" Margin="0 0 0 10" Name="txtBoxDirectory"></TextBox>
140            </Grid>
141
142
143            <telerik:RadGridView Grid.Column="1" Grid.Row="8"
144                                 Grid.ColumnSpan="4"
145                                 RowIndicatorVisibility="Hidden"
146                                 Name="RadGridViewDocuments"
147                                 AutoGenerateColumns="False"
148                                 RowDetailsTemplate="{StaticResource RowDetailsTemplate}"
149                                 ShowGroupPanel="False" IsFilteringAllowed="False">
150
151                <telerik:RadGridView.Columns>
152                    <telerik:GridViewToggleRowDetailsColumn />
153                    <telerik:GridViewDataColumn IsReadOnly="True" DataMemberBinding="{Binding Name}" Header="Name" Name="colName"/>
154                    <telerik:GridViewDataColumn IsReadOnly="True" DataMemberBinding="{Binding DataId}" Header="Alexandria Id" />
```

```
155                    <telerik:GridViewColumn Header="Add attribute" Width="130">
156                        <telerik:GridViewColumn.CellTemplate>
157                            <DataTemplate>
158                                <Button Click="btnAddAttribute_Click" Margin="5">Add attribute</Button>
159                            </DataTemplate>
160                        </telerik:GridViewColumn.CellTemplate>
161                    </telerik:GridViewColumn>
162                    <telerik:GridViewColumn Header="Edit"  Width="100">
163                        <telerik:GridViewColumn.CellTemplate>
164                            <DataTemplate>
165                                <Button Click="btnEditCollection_Click"  Margin="5">Edit</Button>
166                            </DataTemplate>
167                        </telerik:GridViewColumn.CellTemplate>
168                    </telerik:GridViewColumn>
169                    <telerik:GridViewColumn Header="Delete"  Width="100">
170                        <telerik:GridViewColumn.CellTemplate>
171                            <DataTemplate>
172                                <Button Click="btnDeleteCollection_Click"  Margin="5">Delete</Button>
173                            </DataTemplate>
174                        </telerik:GridViewColumn.CellTemplate>
175                    </telerik:GridViewColumn>
176                </telerik:RadGridView.Columns>
177            </telerik:RadGridView>
178
179            <Button Grid.Column="1" Grid.Row="10" Name="btnAddCollection" Click="btnAddCollection_Click">Add Container</Button>
180
181            <Separator Grid.Row="11" Grid.Column="1" Grid.ColumnSpan="4"></Separator>
182
183            <Button Grid.Column="1" Grid.Row="12" Name="btnSaveSettings" Click="btnSaveSettings_Click">Save settings</Button>
184        </Grid>
185    </ScrollViewer>
186</Window>
187
```

OptionsWindow.cs

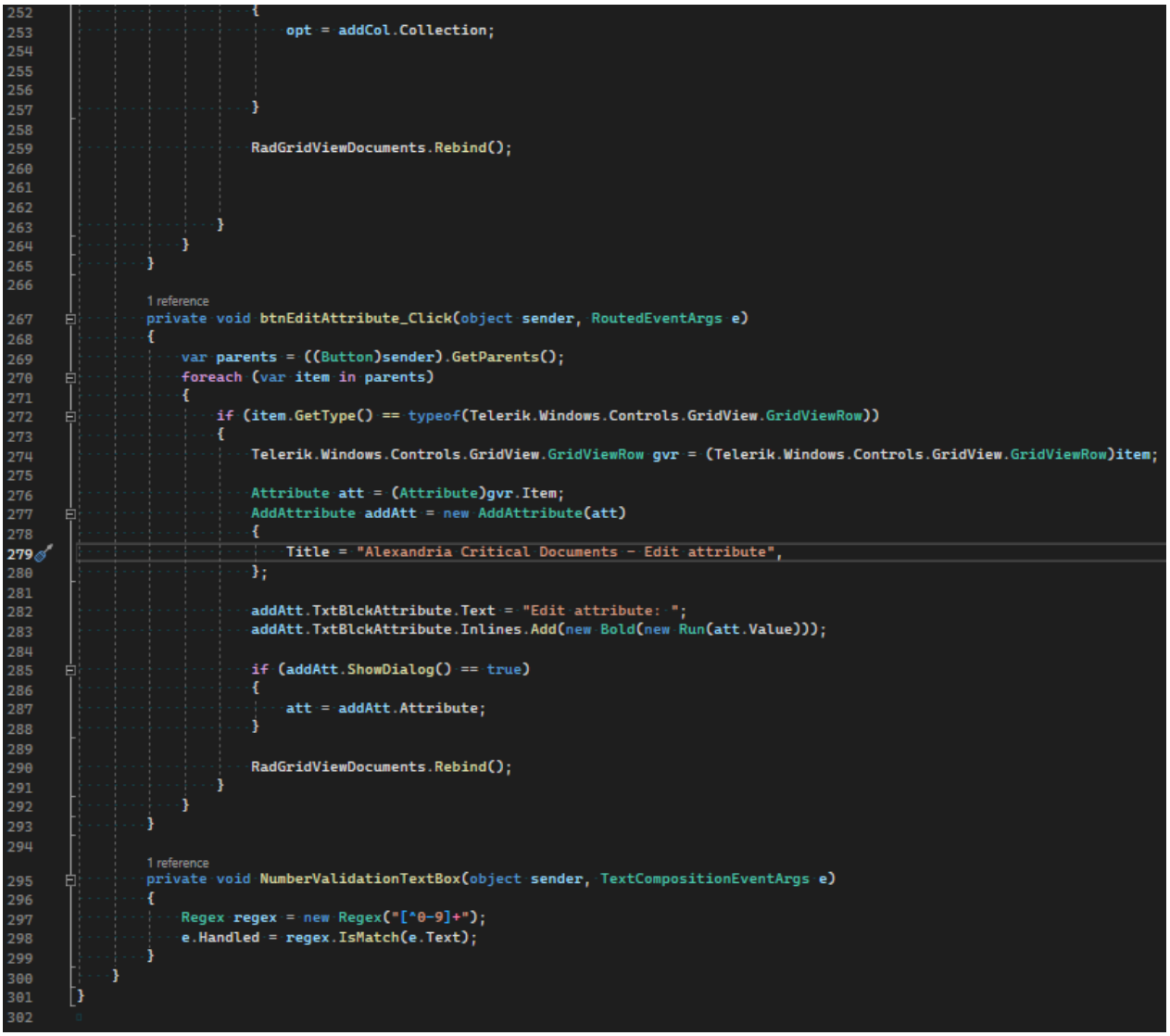
```
using Critical_Docs_Sync;
using Critical_Docs_Sync.Model;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Serilog;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text.RegularExpressions;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;
using System.Windows.Input;
using Telerik.Windows.Controls;

namespace Critical_Docs_Options
{
    /// <summary>
    /// Interaction logic for OptionsWindow.xaml
    /// </summary>
    8 references
    public partial class OptionsWindow : Window
    {
        public readonly List<AlexandriaSourceOptions> newCollection;
        public readonly List<AlexandriaSourceOptions> originalCollection;
        public AlexandriaSourceOptions collection;
        3 references
        public OptionsWindow()
        {
            InitializeComponent();

            OptionsContext context = new OptionsContext();

            txtBoxOtdsUrl.Text = context.GetOption("otdsURL").Value;
            txtBoxAlexandriaUrl.Text = context.GetOption("alexandriaURL").Value;

            if (context.GetOption("schedule").Value == "True")
            {
                chkPeriodic.IsChecked = true;
            }
            else
            {
                chkPeriodic.IsChecked = false;
            }

            txtBoxSyncTime.Text = context.GetOption("syncTime").Value;
            txtBoxDirectory.Text = context.GetOption("localDirectory").Value;

            newCollection = context.GetCollections();
            originalCollection = context.GetCollections();
```

```
53              RadGridViewDocuments.ItemsSource = newCollection;
54          }
55
            1 reference
56          public void SaveOptions()
57          {
58              List<KeyValuePair<string, string>> options = new List<KeyValuePair<string, string>>
59              {
60                  new KeyValuePair<string, string>("schedule", chkPeriodic.IsChecked.ToString()),
61                  new KeyValuePair<string, string>("syncTime", txtBoxSyncTime.Text)
62              };
63
64              string localdirectory = txtBoxDirectory.Text.Trim().Replace('/', '\\');
65              if (localdirectory == "")
66              {
67                  localdirectory = "D:\\";
68              }
69              else
70              {
71                  if (localdirectory.LastIndexOf('\\') != localdirectory.Length - 1)
72                  {
73                      localdirectory += "\\";
74                  }
75              }
76
77              string otdsurl = txtBoxOtdsUrl.Text.Trim();
78
79              if (otdsurl.LastIndexOf('/') != otdsurl.Length - 1)
80              {
81                  otdsurl += "/";
82              }
83
84              string axurl = txtBoxAlexandriaUrl.Text.Trim();
85
86              if (axurl.LastIndexOf('/') != axurl.Length - 1)
87              {
88                  axurl += "/";
89              }
90
91              options.Add(new KeyValuePair<string, string>("localDirectory", localdirectory));
92              options.Add(new KeyValuePair<string, string>("otdsURL", otdsurl));
93              options.Add(new KeyValuePair<string, string>("alexandriaURL", axurl));
94              OptionsContext context = new OptionsContext();
95              context.UpdateOptions(options);
96
97              foreach (var item in newCollection)
98              {
99                  var names = originalCollection.Where(x => x.DataId == item.DataId && x.Name != item.Name);
100                 if (names != null)
101                 {
102                     foreach (var original in originalCollection)
```

```
                    {
                        if (Directory.Exists(txtBoxDirectory.Text + original.Name) && txtBoxDirectory + original.Name != txtBoxDirectory + item.Name)
                        {
                            Directory.Move(txtBoxDirectory.Text + original.Name, txtBoxDirectory.Text + item.Name);
                        }

                    }
                }
                collection = item;
            }

            var containerId = newCollection.Where(x => x.DataId == collection.DataId).Count() > 1;
            var containerName = newCollection.Where(x => x.Name == collection.Name).Count() > 1;
            if (containerId)
            {
                MessageBox.Show("Container with ID: '" + collection.DataId + "' already exists in collection");
            }
            else if (containerName)
            {
                MessageBox.Show("Container with Name: '" + collection.Name + "' already exist in collection");
            }
            else if (chkPeriodic.IsChecked == true && txtBoxSyncTime.Text == "")
            {
                MessageBox.Show("Synchronisation time cannot be empty!");
            }
            else
            {
                context.UpdateCollections(newCollection);
                Close();
            }

        }

        1 reference
        private void btnSaveSettings_Click(object sender, RoutedEventArgs e)
        {
            SaveOptions();
        }

        1 reference
        private void btnDeleteCollection_Click(object sender, RoutedEventArgs e)
        {
            var parents = ((Button)sender).GetParents();

            foreach (var item in parents)
            {
                if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
                {
                    Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;
                    AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
                    MessageBoxResult result = MessageBox.Show("Are you sure you want to delete this container?", "Warning", MessageBoxButton.YesNo, MessageBoxImage.Information);

```

```
                switch (result)
                {
                    case MessageBoxResult.Yes:
                        newCollection.Remove((AlexandriaSourceOptions)gvr.Item);
                        if (Directory.Exists(txtBoxDirectory.Text + opt.Name))
                        {
                            Directory.Delete(txtBoxDirectory.Text + opt.Name, true);
                        }

                        break;
                    case MessageBoxResult.No:
                        break;
                }

                RadGridViewDocuments.Rebind();
            }
        }
    }

    1 reference
    private void btnDeleteAttribute_Click(object sender, RoutedEventArgs e)
    {
        var parents = ((Button)sender).GetParents();
        foreach (var item in parents)
        {
            if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
            {
                Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

                foreach (var col in newCollection)
                {
                    col.Attributes.Remove((Attribute)gvr.Item);
                }
                RadGridViewDocuments.Rebind();
            }
        }
    }

    1 reference
    private void btnAddCollection_Click(object sender, RoutedEventArgs e)
    {
        AddCollection addCol = new AddCollection(new AlexandriaSourceOptions())
        {
            Title = "Alexandria Critical Documents - Add Container"
        };
        addCol.TxtBlckContainer.Text = "Add a new container to download";
        if (addCol.ShowDialog() == true)
        {
            AlexandriaSourceOptions coll = addCol._Collection;

```

```
                newCollection.Add(coll);

                RadGridViewDocuments.Rebind();
            }
        }

        1 reference
        private void btnAddAttribute_Click(object sender, RoutedEventArgs e)
        {
            AddAttribute addAtt = new AddAttribute(new Attribute())
            {
                Title = "Alexandria Critical Documents - Add Attribute"
            };
            addAtt.chkBoxMandatory.IsChecked = true;
            if (addAtt.ShowDialog() == true)
            {
                Attribute att = addAtt.Attribute;
                addAtt.TxtBlckAttribute.Text = "Add a new attribute to collection";
                var parents = ((Button)sender).GetParents();
                foreach (var item in parents)
                {
                    if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
                    {
                        Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

                        AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
                        opt.Attributes.Add(att);

                        RadGridViewDocuments.Rebind();
                    }
                }
            }
        }

        1 reference
        private void btnEditCollection_Click(object sender, RoutedEventArgs e)
        {
            var parents = ((Button)sender).GetParents();
            foreach (var item in parents)
            {
                if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
                {
                    Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

                    AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
                    AddCollection addCol = new AddCollection(opt)
                    {
                        Title = "Alexandria Critical Documents - Edit Container"
                    };
                    addCol.TxtBlckContainer.Text = "Edit container: ";
                    addCol.TxtBlckContainer.Inlines.Add(new Bold(new Run(opt.Name)));
                    if (addCol.ShowDialog() == true)
```



```
                {
                    opt = addCol.Collection;



                }

                RadGridViewDocuments.Rebind();



            }
        }
    }

    1 reference
    private void btnEditAttribute_Click(object sender, RoutedEventArgs e)
    {
        var parents = ((Button)sender).GetParents();
        foreach (var item in parents)
        {
            if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
            {
                Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

                Attribute att = (Attribute)gvr.Item;
                AddAttribute addAtt = new AddAttribute(att)
                {
                    Title = "Alexandria Critical Documents - Edit attribute",
                };

                addAtt.TxtBlckAttribute.Text = "Edit attribute: ";
                addAtt.TxtBlckAttribute.Inlines.Add(new Bold(new Run(att.Value)));

                if (addAtt.ShowDialog() == true)
                {
                    att = addAtt.Attribute;
                }

                RadGridViewDocuments.Rebind();
            }
        }
    }

    1 reference
    private void NumberValidationTextBox(object sender, TextCompositionEventArgs e)
    {
        Regex regex = new Regex("[^0-9]+");
        e.Handled = regex.IsMatch(e.Text);
    }
}
}
```

SaveOptions

This method succeeds all kinds of options. It succeeds on whether chkPeriodic is checked and it succeeds the txtBoxSyncTime.

Next, we change '/' to '/' and put it in a variable localdirectory with type string. If localdirectory is empty, then we set localdirectory to "D:/". Otherwise, we check to see if the last index of 'ע' is not at the end of localdirectory. If it is, then we'll add 'ith' at the end.

After this, we retrieve the trimmed information from txtBoxOtdsUrl and place it in a variable otdsurl with type string. If '/' is not at the end of otdsurl, that is added. Next, we retrieve the trimmed information from txtBoxAlexandriaUrl and place it in a variable axurl with type string. If '/' is not at the end of axurl, then that is added. The next step is to add localdirectory, otdsurl, axurl and update the options table in the database.

For each item in the new collection, we look for the dataId and name, if the dataId is the same, we put this into a variable names. If the names are found, we look for each item in the original collection. If a folder exists that has the text of txtBoxDirectory + the name of the item in the original list, and the text of txtBoxDirectory + the name of the item in the original list is not the same as the text of txtBoxDirectory + the name of the item in the new list. Next, we set collection to item.

We look at the dataIds that are the same and appear more than 1 time in the new list and put them in a variable called containerId. We count the names that are the same and occur more than 1 time in the new list and place them in a variable named containerName. If the dataId occurs multiple times, the user gets a notification that the dataId already appears in the list. Otherwise, if the names occurs multiple times, the user gets a notification that the name already appears in the list. Otherwise, if chkPeriodic is checked and the text of txtBoxSyncTime is empty, then the user gets a notification that that textbox should not be empty. Otherwise, update the collection and close the screen.

```
public void SaveOptions()
{
    List<KeyValuePair<string, string>> options = new List<KeyValuePair<string, string>>
    {
        new KeyValuePair<string, string>("schedule", chkPeriodic.IsChecked.ToString()),
        new KeyValuePair<string, string>("syncTime", txtBoxSyncTime.Text)
    };

    string localdirectory = txtBoxDirectory.Text.Trim().Replace('/', '\\');
    if (localdirectory == "")
    {
        localdirectory = "D:\\";
    }
    else
    {
        if (localdirectory.LastIndexOf('\\') != localdirectory.Length - 1)
        {
            localdirectory += "\\";
        }
    }

    string otdsurl = txtBoxOtdsUrl.Text.Trim();

    if (otdsurl.LastIndexOf('/') != otdsurl.Length - 1)
    {
        otdsurl += "/";
    }

    string axurl = txtBoxAlexandriaUrl.Text.Trim();

    if (axurl.LastIndexOf('/') != axurl.Length - 1)
    {
        axurl += "/";
    }

    options.Add(new KeyValuePair<string, string>("localDirectory", localdirectory));
    options.Add(new KeyValuePair<string, string>("otdsURL", otdsurl));
    options.Add(new KeyValuePair<string, string>("alexandriaURL", axurl));
    OptionsContext context = new OptionsContext();
    context.UpdateOptions(options);

    foreach (var item in newCollection)
    {
        var names = originalCollection.Where(x => x.DataId == item.DataId && x.Name != item.Name);
        if (names != null)
        {
            foreach (var original in originalCollection)
            {
                if (Directory.Exists(txtBoxDirectory.Text + original.Name) && txtBoxDirectory + original.Name != txtBoxDirectory + item.Name)
                {
                    Directory.Move(txtBoxDirectory.Text + original.Name, txtBoxDirectory.Text + item.Name);
                }
```

```
                    }
                }
                collection = item;
            }

            var containerId = newCollection.Where(x => x.DataId == collection.DataId).Count() > 1;
            var containerName = newCollection.Where(x => x.Name == collection.Name).Count() > 1;
            if (containerId)
            {
                MessageBox.Show("Container with ID: '" + collection.DataId + "' already exists in collection");
            }
            else if (containerName)
            {
                MessageBox.Show("Container with Name: '" + collection.Name + "' already exist in collection");
            }
            else if (chkPeriodic.IsChecked == true && txtBoxSyncTime.Text == "")
            {
                MessageBox.Show("Synchronisation time cannot be empty!");
            }
            else
            {
                context.UpdateCollections(newCollection);
                Close();
            }

        }
```

BtnSaveSettings_Click

If the user clicks on btnSaveSettings, then the program saves the options.

```
private void btnSaveSettings_Click(object sender, RoutedEventArgs e)
{
    SaveOptions();
}
```

BtnDeleteCollection_Click

When the user presses button, the user is asked if they are sure they want to delete the container. If yes, then the container is deleted locally and FileHistory is emptied.

```
private void btnDeleteCollection_Click(object sender, RoutedEventArgs e)
{
    var parents = ((Button)sender).GetParents();

    foreach (var item in parents)
    {
        if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
        {
            Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;
            AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
            MessageBoxResult result = MessageBox.Show("Are you sure you want to delete this container?", "Warning", MessageBoxButton.YesNo, MessageBoxImage.Information);

            // Process message box results
            switch (result)
            {
                case MessageBoxResult.Yes:
                    newCollection.Remove((AlexandriaSourceOptions)gvr.Item);
                    if (Directory.Exists(txtBoxDirectory.Text + opt.Name))
                    {
                        Directory.Delete(txtBoxDirectory.Text + opt.Name, true);
                    }

                    break;
                case MessageBoxResult.No:
                    break;
            }

            RadGridViewDocuments.Rebind();
        }
    }
}
```

BtnDeleteAttribute_Click

When the user presses the button, then the attribute of that container gets removed out of the database.

```
private void btnDeleteAttribute_Click(object sender, RoutedEventArgs e)
{
    var parents = ((Button)sender).GetParents();
    foreach (var item in parents)
    {
        if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
        {
            Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

            foreach (var col in newCollection)
            {
                col.Attributes.Remove((Attribute)gvr.Item);
            }
            RadGridViewDocuments.Rebind();
        }
    }
}
```

BtnAddCollection_Click

When the user presses the button, the title of AddCollection.xaml changes and we set the text of TxtBlckContainer to "Add a new container to download." The new collection is added to a grid.

```
private void btnAddCollection_Click(object sender, RoutedEventArgs e)
{
    AddCollection addCol = new AddCollection(new AlexandriaSourceOptions())
    {
        Title = "Alexandria Critical Documents - Add Container"
    };
    addCol.TxtBlckContainer.Text = "Add a new container to download";
    if (addCol.ShowDialog() == true)
    {
        AlexandriaSourceOptions coll = addCol._Collection;

        newCollection.Add(coll);

        RadGridViewDocuments.Rebind();
    }
}
```

BtnAddAttribute_Click

When the user presses the button, the title of AddAttribute.xaml changes and we set the text of TxtBlckAttribute to "Add a new attribute to collection". The attribute is added to the grid under the collection.

```
private void btnAddAttribute_Click(object sender, RoutedEventArgs e)
{
    AddAttribute addAtt = new AddAttribute(new Attribute())
    {
        Title = "Alexandria Critical Documents - Add Attribute"
    };
    addAtt.chkBoxMandatory.IsChecked = true;
    if (addAtt.ShowDialog() == true)
    {
        Attribute att = addAtt.Attribute;
        addAtt.TxtBlckAttribute.Text = "Add a new attribute to collection";
        var parents = ((Button)sender).GetParents();
        foreach (var item in parents)
        {
            if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
            {
                Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

                AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
                opt.Attributes.Add(att);

                RadGridViewDocuments.Rebind();
            }
        }
    }
}
```

BtnEditCollection_Click

When the user presses the button, the user sees which container they want to change. The container is modified.

```
private void btnEditCollection_Click(object sender, RoutedEventArgs e)
{
    var parents = ((Button)sender).GetParents();
    foreach (var item in parents)
    {
        if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
        {
            Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

            AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
            AddCollection addCol = new AddCollection(opt)
            {
                Title = "Alexandria Critical Documents - Edit Container"
            };
            addCol.TxtBlckContainer.Text = "Edit container: ";
            addCol.TxtBlckContainer.Inlines.Add(new Bold(new Run(opt.Name)));
            if (addCol.ShowDialog() == true)
            {
                opt = addCol.Collection;



            }

            RadGridViewDocuments.Rebind();



        }
    }
}
```

BtnEditAttribute_Click

When the user presses the button, then the attribute gets updated.

```
private void btnEditAttribute_Click(object sender, RoutedEventArgs e)
{
    var parents = ((Button)sender).GetParents();
    foreach (var item in parents)
    {
        if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
        {
            Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

            Attribute att = (Attribute)gvr.Item;
            AddAttribute addAtt = new AddAttribute(att)
            {
                Title = "Alexandria Critical Documentsro - Edit attribute",
            };

            addAtt.TxtBlckAttribute.Text = "Edit attribute: ";
            addAtt.TxtBlckAttribute.Inlines.Add(new Bold(new Run(att.Value)));

            if (addAtt.ShowDialog() == true)
            {
                att = addAtt.Attribute;
            }

            RadGridViewDocuments.Rebind();
        }
    }
}
```

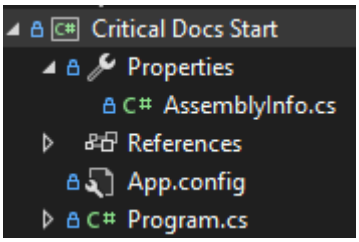
NumberValidationTextBox

This method makes sure that only numbers get filled in a textbox, in this case its only used in txtBoxSyncTime.

```
private void NumberValidationTextBox(object sender, TextCompositionEventArgs e)
{
    Regex regex = new Regex("[^0-9]+");
    e.Handled = regex.IsMatch(e.Text);
}
```

Critical Docs Start

This project is used to start a process.



AssemblyInfo

This class only has a Main, in this method we start the “Alexandria CDST” process.

```
using System.Diagnostics;

namespace Critical_Docs_Start
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            var p = new Process {StartInfo = {FileName = "Alexandria CDST", WindowStyle = ProcessWindowStyle.Hidden}};
            p.Start();
        }
    }
}
```

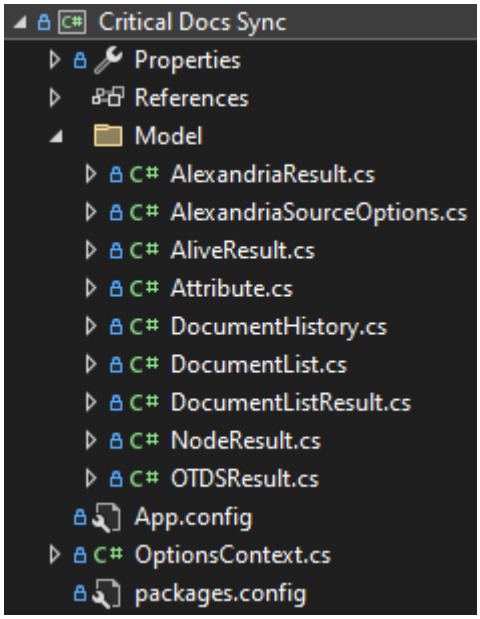
Critical Docs Sync

This project contains a component of MVVM (model) and uses a database. The model has 9 classes:

- AlexandriaResult
- AlexandriaSourceOptions
- AliveResult
- Attribute
- DocumentHistory
- DocumentList
- DocumentListResult
- NodeResult
- OTDSResult.

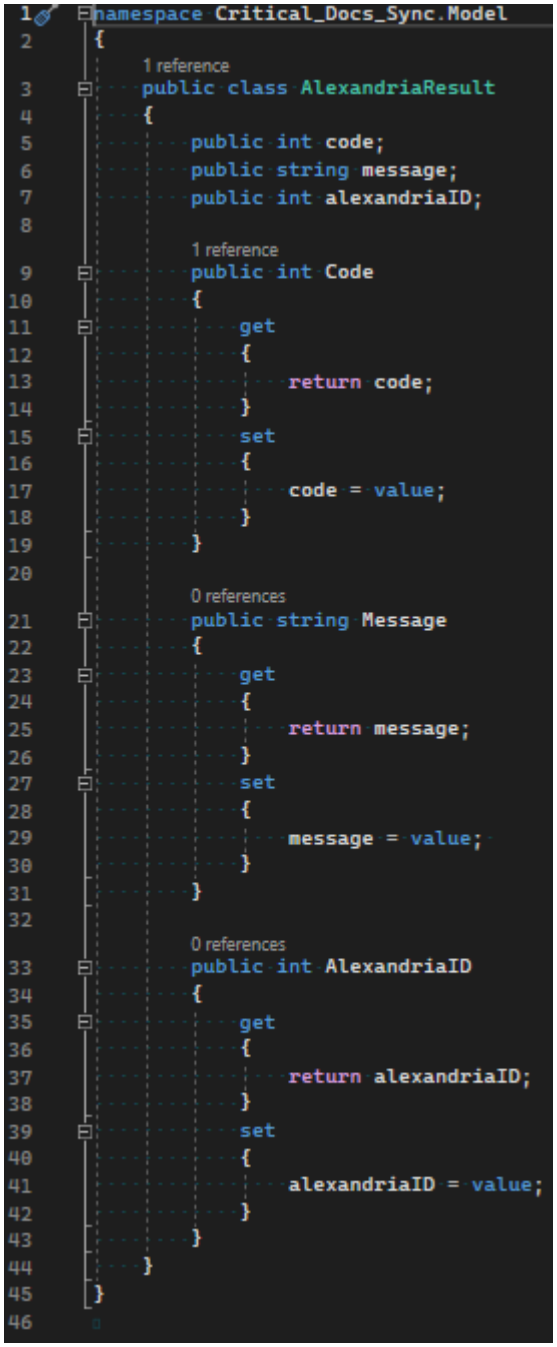
This project contains 8 packages:

- EntityFramework
- Newtonsoft.Json, Serilog
- SQLite.System.Data.SQLite, System.Data.SQLite.Core, System.Data.SQLite.EF6 and System.Data.SQLite.Linq.



AlexandriaResult

This model checks the accessibility of Alexandria.



```
namespace Critical_Docs_Sync.Model
{
    1 reference
    public class AlexandriaResult
    {
        public int code;
        public string message;
        public int alexandriaID;

        1 reference
        public int Code
        {
            get
            {
                return code;
            }
            set
            {
                code = value;
            }
        }

        0 references
        public string Message
        {
            get
            {
                return message;
            }
            set
            {
                message = value;
            }
        }

        0 references
        public int AlexandriaID
        {
            get
            {
                return alexandriaID;
            }
            set
            {
                alexandriaID = value;
            }
        }
    }
}
```

AlexandriaSourceOptions

This model has all information of a container: the objectId, the name and the attributes.

```
using System.Collections.Generic;
using System.ComponentModel;
using System.Text.RegularExpressions;

namespace Critical_Docs_Sync.Model
{
    34 references
    public class AlexandriaSourceOptions
    {
        public int id;
        public string name;
        public int dataId;
        public List<Attribute> attributes;

        11 references
        public int Id
        {
            get
            {
                return id;
            }
            set
            {
                id = value;
            }
        }

        29 references
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
                OnPropertyChanged(nameof(Name));
            }
        }

        17 references
        public int DataId
        {
            get
            {
                return dataId;
            }
            set
            {
                dataId = value;
                OnPropertyChanged(nameof(DataId));
            }
        }
```

```
51
                12 references
52              public List<Attribute> Attributes
53              {
54                  get
55                  {
56                      return attributes;
57                  }
58                  set
59                  {
60                      attributes = value;
61                      OnPropertyChanged(nameof(Attributes));
62                  }
63              }
64
65              public event PropertyChangedEventHandler PropertyChanged;
                3 references
66              private void OnPropertyChanged(string propertyName)
67              {
68                  if (PropertyChanged != null)
69                  {
70                      PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
71                  }
72              }
73          }
74      }
75
```

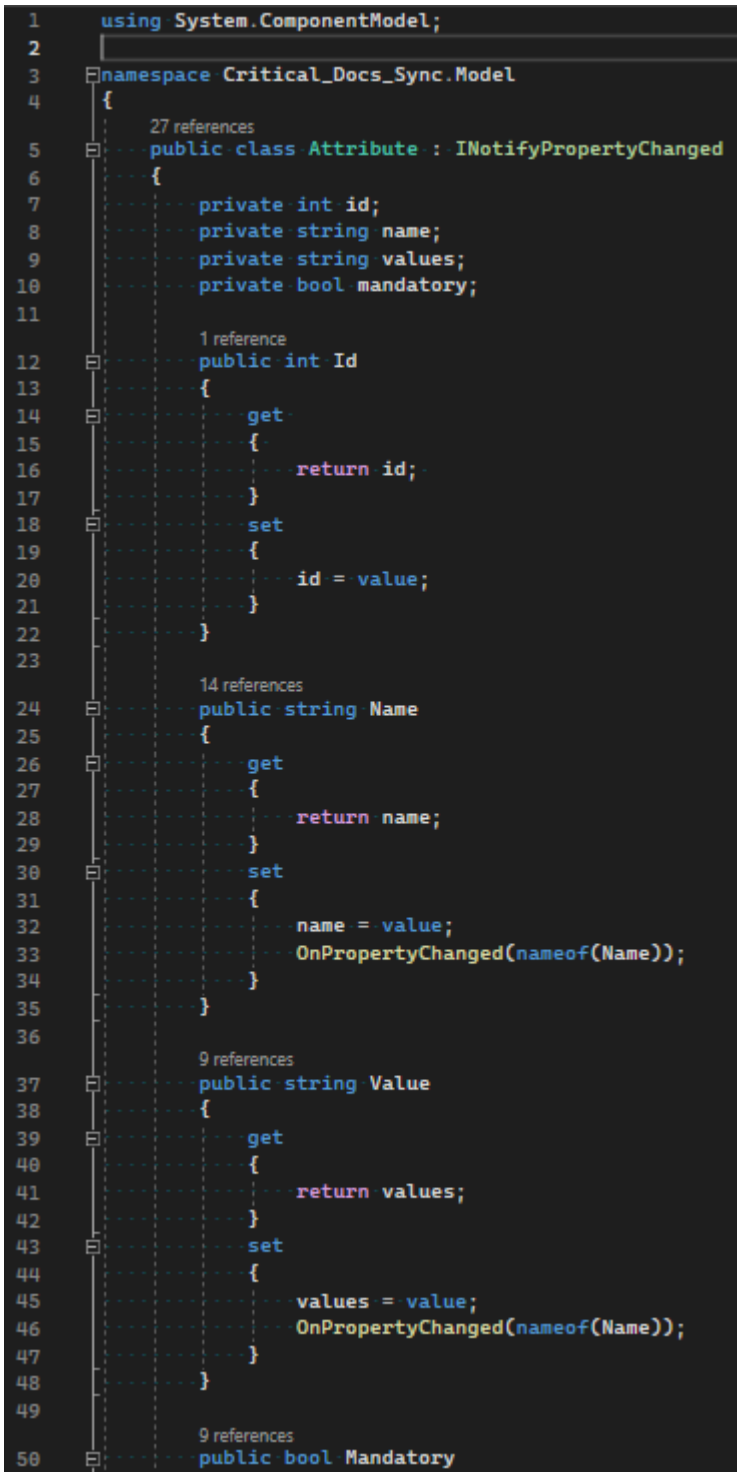
AliveResult

This model is used to check the availability of Alexandria.

```
1   namespace Critical_Docs_Sync.Model
2   {
        1 reference
3       public class AliveResult
4       {
5           public bool isAlive;
6
            0 references
7           public bool IsAlive
8           {
9               get
10              {
11                  return isAlive;
12              }
13              set
14              {
15                  isAlive = value;
16              }
17          }
18      }
19  }
20
```

Attribute

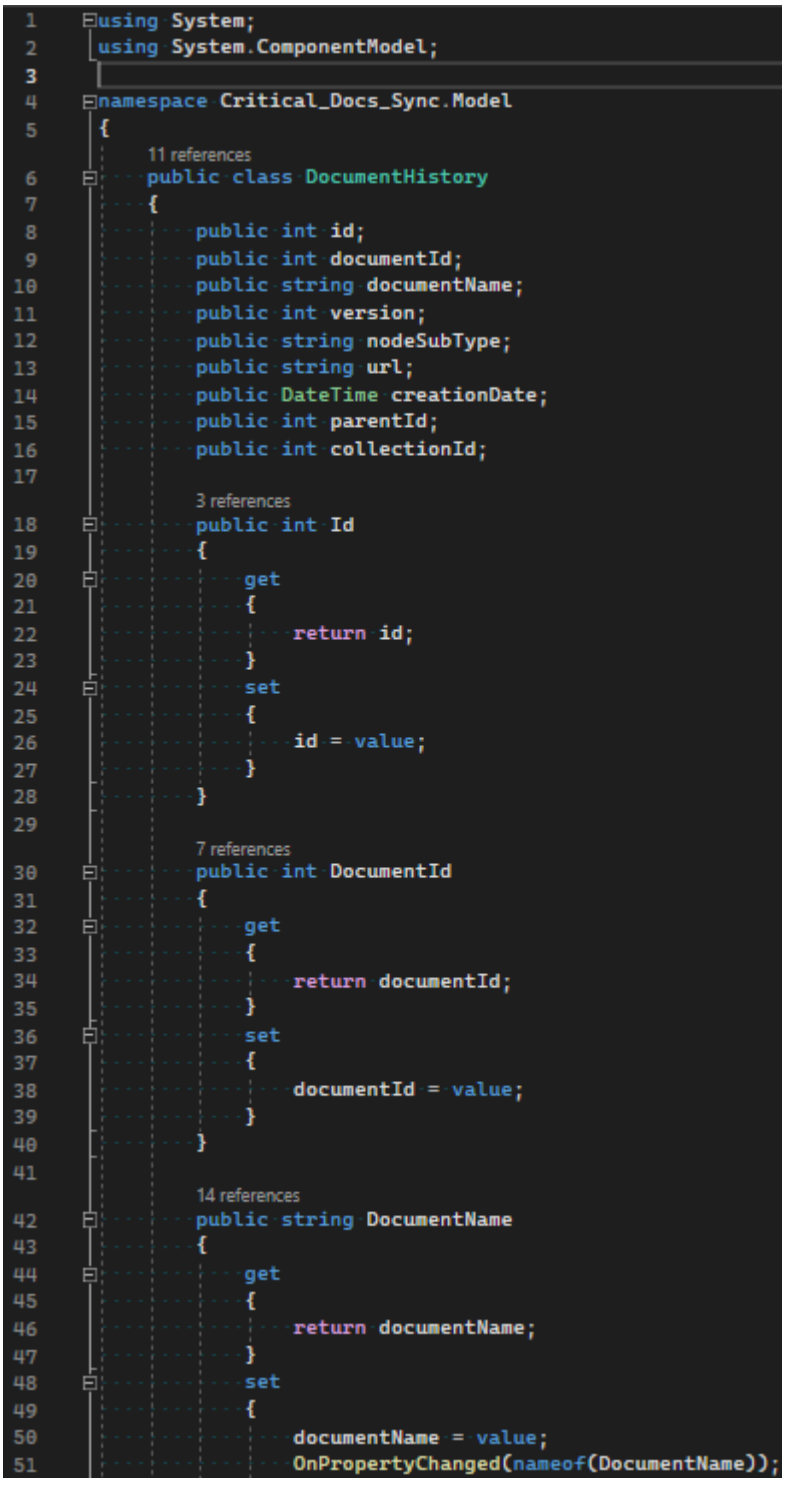
This model contains the filters for the downloaded documents. This model contains the name and value of the attribute and if that attribute is required.



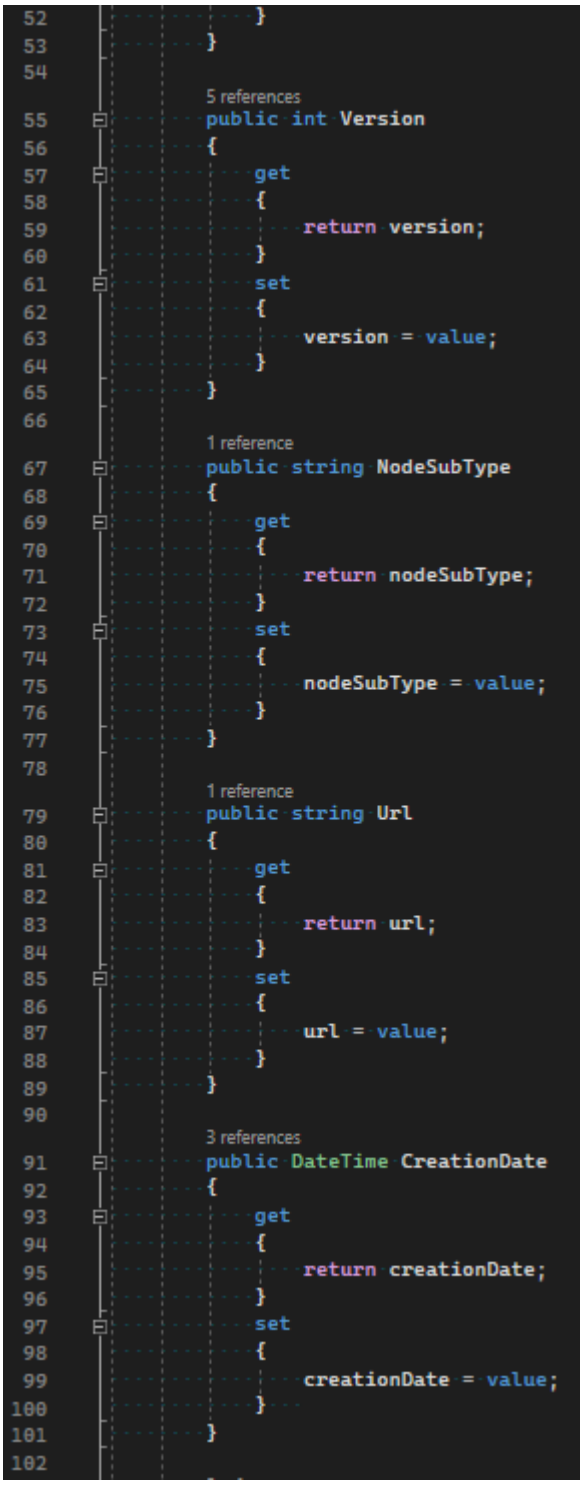
```
        {
            get
            {
                return mandatory;
            }
            set
            {
                mandatory = value;
                OnPropertyChanged(nameof(Mandatory));
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

DocumentHistory

This model is part of a table in the database. This model contains the documentId, the name of the document, the version of the document, the type of document, the link to the document, the creation date of the document, and what folder that document is in (unless the document is a container, in which case there is no folder).



```
using System;
using System.ComponentModel;

namespace Critical_Docs_Sync.Model
{
    11 references
    public class DocumentHistory
    {
        public int id;
        public int documentId;
        public string documentName;
        public int version;
        public string nodeSubType;
        public string url;
        public DateTime creationDate;
        public int parentId;
        public int collectionId;

        3 references
        public int Id
        {
            get
            {
                return id;
            }
            set
            {
                id = value;
            }
        }

        7 references
        public int DocumentId
        {
            get
            {
                return documentId;
            }
            set
            {
                documentId = value;
            }
        }

        14 references
        public string DocumentName
        {
            get
            {
                return documentName;
            }
            set
            {
                documentName = value;
                OnPropertyChanged(nameof(DocumentName));
```

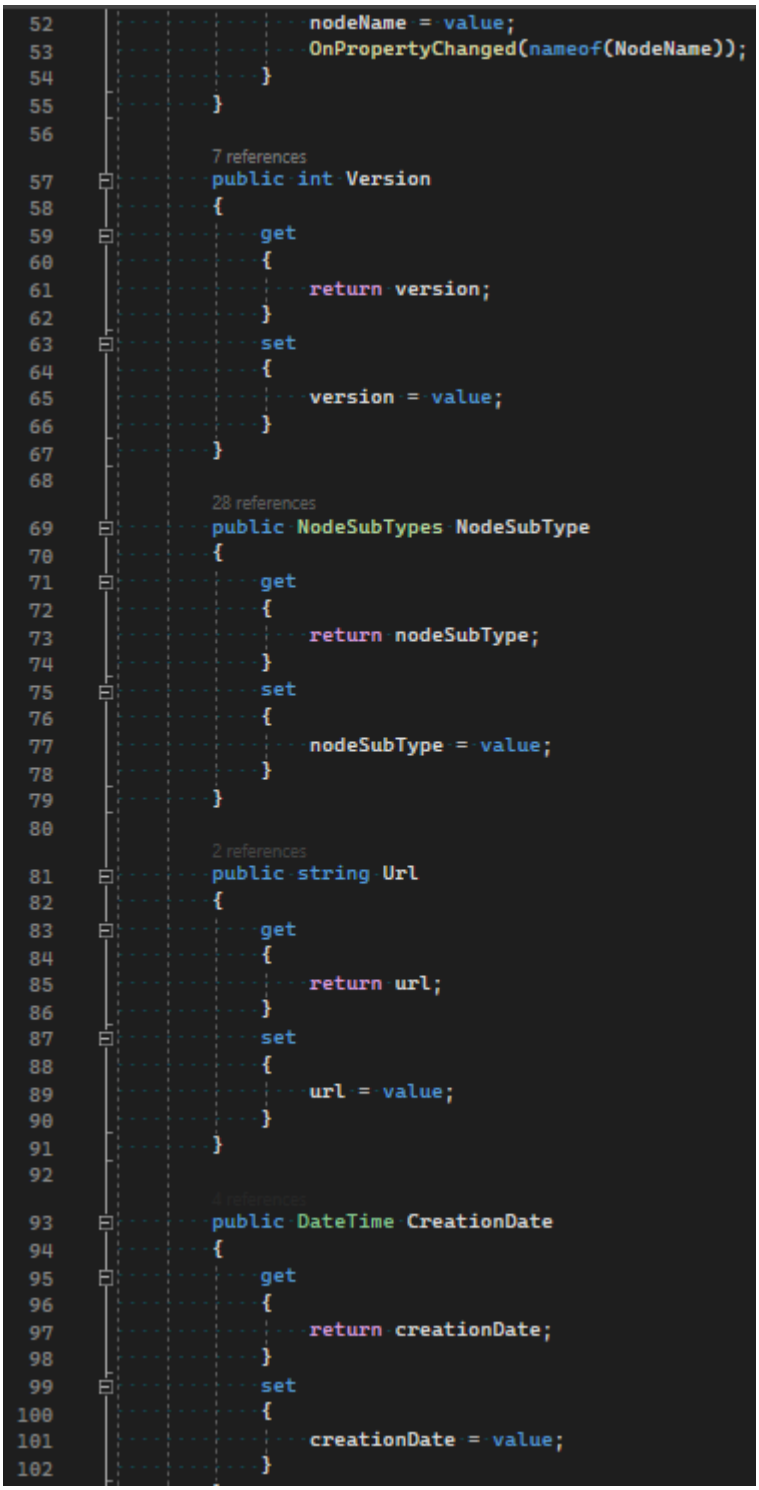


```
103         public int ParentId
104         {
105             get
106             {
107                 return parentId;
108             }
109             set
110             {
111                 parentId = value;
112             }
113         }
114
            0 references
115         public int CollectionId
116         {
117             get
118             {
119                 return collectionId;
120             }
121             set
122             {
123                 collectionId = value;
124             }
125         }
126
127         public event PropertyChangedEventHandler PropertyChanged;
            1 reference
128         private void OnPropertyChanged(string propertyName)
129         {
130             if (PropertyChanged != null)
131             {
132                 PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
133             }
134         }
135     }
136 }
137
```

DocumentList

This model is part of the data in Alexandria. This model contains the documentId, the name of the document, the version of the document, the type of document, the link to the document, the creation date of the document, and what folder that document is in (unless the document is a container, in which case there is no folder).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;

namespace Critical_Docs_Sync.Model
{
    14 references
    public class DocumentList
    {
        public int id;
        public int nodeId;
        public string nodeName;
        public int version;
        public NodeSubTypes nodeSubType;
        public string url;
        public DateTime creationDate;
        public List<DocumentList> children;
        public int parentId;
        public int collectionId;

        0 references
        public int Id
        {
            get
            {
                return id;
            }
            set
            {
                id = value;
            }
        }

        32 references
        public int NodeId
        {
            get
            {
                return nodeId;
            }
            set
            {
                nodeId = value;
            }
        }

        66 references
        public string NodeName
        {
            get
            {
                return nodeName;
            }
            set
            {
```

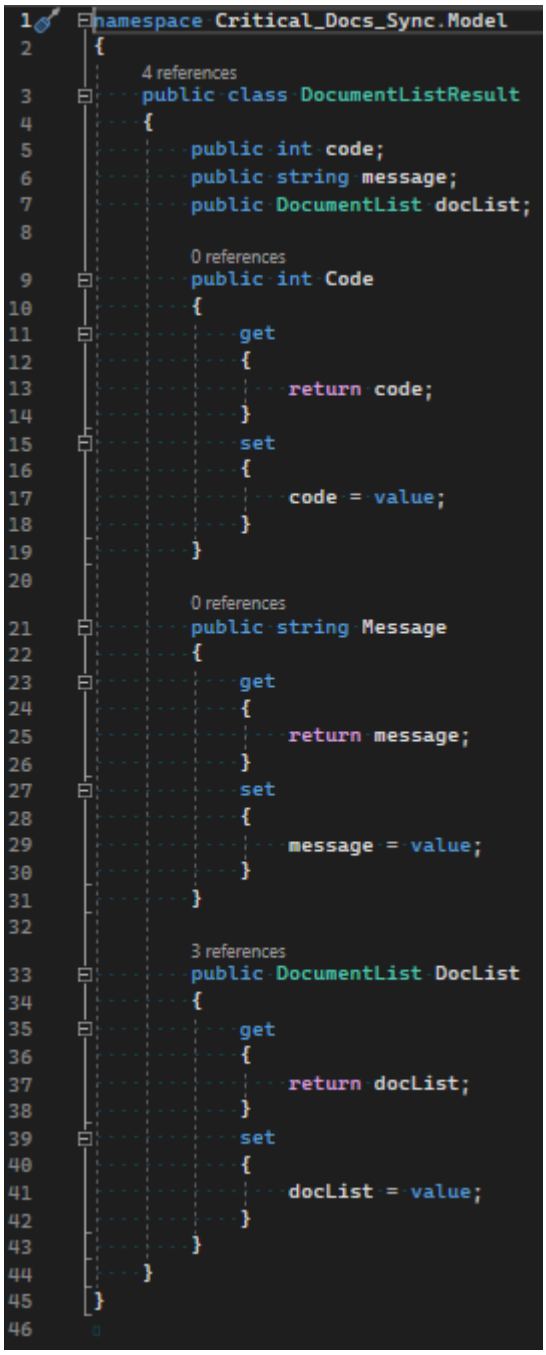


```
103             }
104
                10 references
105             public List<DocumentList> Children
106             {
107                 get
108                 {
109                     return children;
110                 }
111                 set
112                 {
113                     children = value;
114                 }
115             }
116
                15 references
117             public int ParentId
118             {
119                 get
120                 {
121
122                     return parentId;
123
124                 }
125                 set
126                 {
127                     parentId = value;
128                 }
129             }
130
131             public int CollectionId
132             {
133                 get
134                 {
135                     return collectionId;
136                 }
137                 set
138                 {
139                     collectionId = value;
140                 }
141             }
142
143             public event PropertyChangedEventHandler PropertyChanged;
144             private void OnPropertyChanged(string propertyName)
145             {
146                 if (PropertyChanged != null)
147                 {
148                     PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
149                 }
150             }
151         }
152     }
153
```

```
public enum NodeSubTypes
{
    Folder = 0,
    Shortcut = 1,
    Generation = 2,
    CompoundDocument = 136,
    URL = 140,
    Document = 144,
    TextDocument = 145,
    Collection = 298,
    Email = 749,
    EmailFolder = 751,
    Binder = 31066,
    WBSElement = 31350,
    IMSProcessDescription = 31351,
    EmployeeFile = 31352,
    CustomerContract = 31353,
    CEKTraining = 31354,
    CustomerQuotation = 31355,
    IDPBWFcase = 31356,
    TMB = 31357,
    IDPBWEcase = 31358,
    IDPBWHSEcase = 31359,
    IDPBWScase = 31360,
    IDPBWTcase = 31361,
    MYRRHAFEEDRFI = 31362,
    MYRRHAFEEDVO = 31363,
    Stageairefile = 31364,
    PUROverheidsopdrachten = 31365,
    CTSProjects = 31366,
    Interimfile = 31367,
    IMSProjectdescription = 31368,
    COMprojects = 31369,
    BinderVolume = 310660
}
```

DocumentListResult

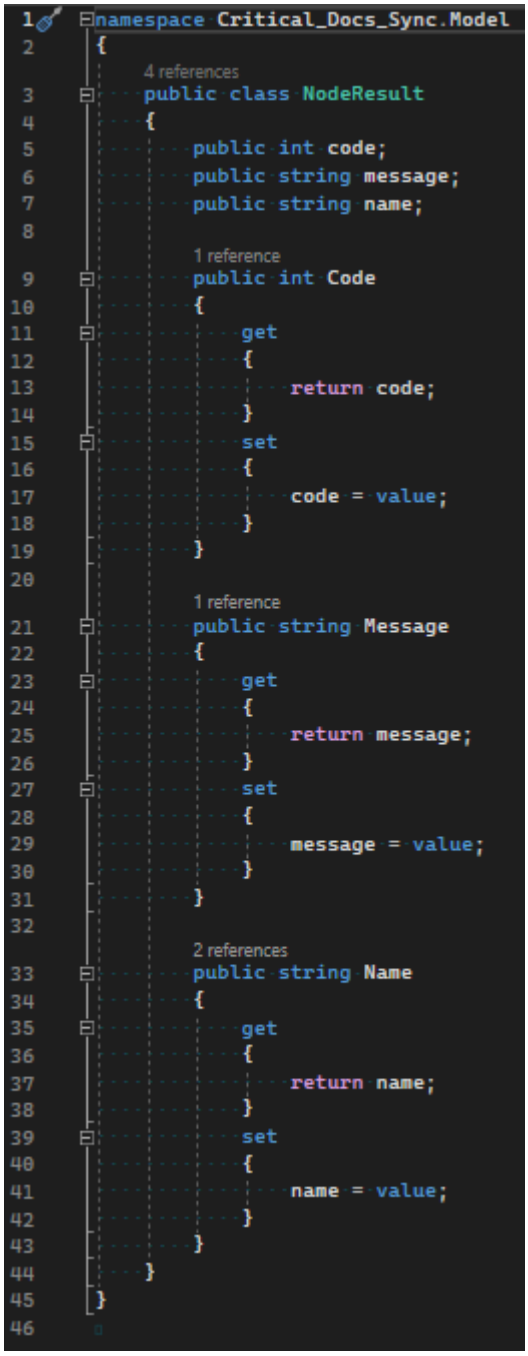
This model returns the list of documents to download.



```
namespace Critical_Docs_Sync.Model
{
    4 references
    public class DocumentListResult
    {
        public int code;
        public string message;
        public DocumentList docList;

        0 references
        public int Code
        {
            get
            {
                return code;
            }
            set
            {
                code = value;
            }
        }

        0 references
        public string Message
        {
            get
            {
                return message;
            }
            set
            {
                message = value;
            }
        }

        3 references
        public DocumentList DocList
        {
            get
            {
                return docList;
            }
            set
            {
                docList = value;
            }
        }
    }
}
```

NodeResult

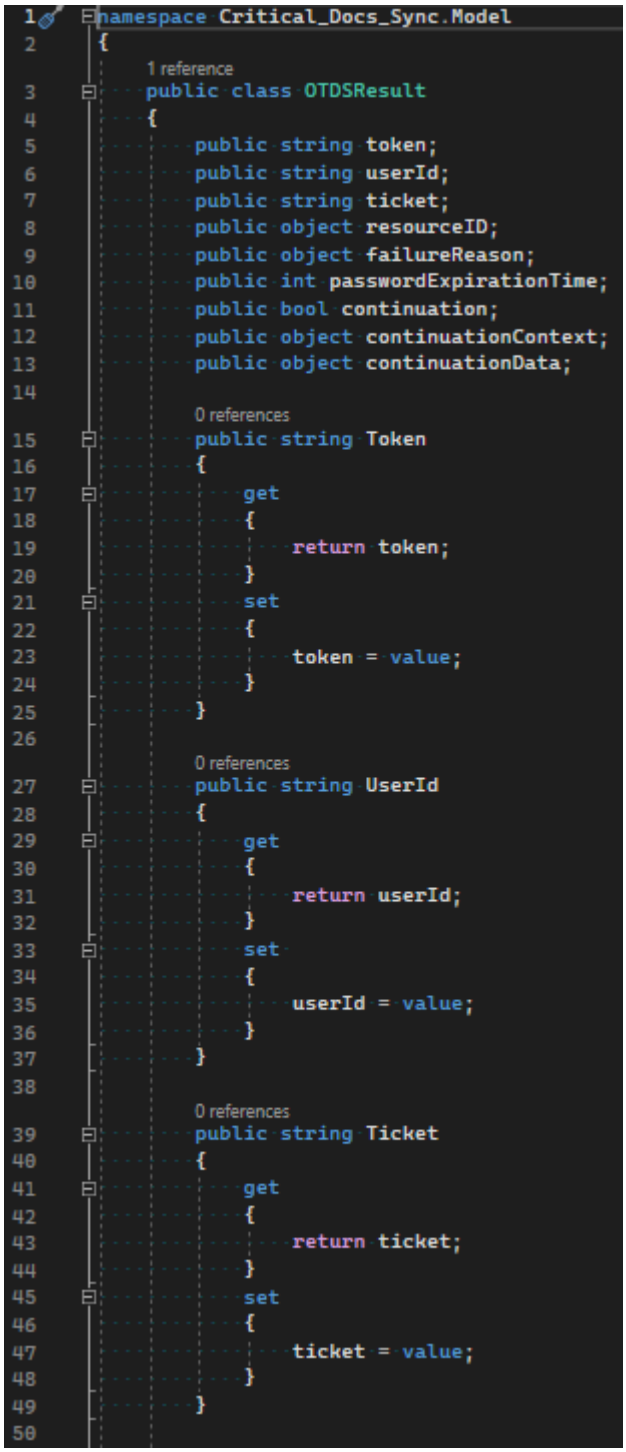
This model checks if the container is accessible.



```
namespace Critical_Docs_Sync.Model
{
    4 references
    public class NodeResult
    {
        public int code;
        public string message;
        public string name;

        1 reference
        public int Code
        {
            get
            {
                return code;
            }
            set
            {
                code = value;
            }
        }

        1 reference
        public string Message
        {
            get
            {
                return message;
            }
            set
            {
                message = value;
            }
        }

        2 references
        public string Name
        {
            get
            {
                return name;
            }
            set
            {
                name = value;
            }
        }
    }
}
```

OTDSResult

This model checks if the otds-ticket is available.



```
namespace Critical_Docs_Sync.Model
{
    1 reference
    public class OTDSResult
    {
        public string token;
        public string userId;
        public string ticket;
        public object resourceID;
        public object failureReason;
        public int passwordExpirationTime;
        public bool continuation;
        public object continuationContext;
        public object continuationData;

        0 references
        public string Token
        {
            get
            {
                return token;
            }
            set
            {
                token = value;
            }
        }

        0 references
        public string UserId
        {
            get
            {
                return userId;
            }
            set
            {
                userId = value;
            }
        }

        0 references
        public string Ticket
        {
            get
            {
                return ticket;
            }
            set
            {
                ticket = value;
            }
        }

```

```
51          public object ResourceID
52          {
53              get
54              {
55                  return resourceID;
56              }
57              set
58              {
59                  resourceID = value;
60              }
61          }
62
                0 references
63          public object FailureReason
64          {
65              get
66              {
67                  return failureReason;
68              }
69              set
70              {
71                  failureReason = value;
72              }
73          }
74
                0 references
75          public int PasswordExpirationTime
76          {
77              get
78              {
79                  return passwordExpirationTime;
80              }
81              set
82              {
83                  passwordExpirationTime = value;
84              }
85          }
86
                0 references
87          public bool Continuation
88          {
89              get
90              {
91                  return continuation;
92              }
93              set
94              {
95                  continuation = value;
96              }
97          }
98
                0 references
99          public object ContinuationContext
100         {
```

```
        get
        {
            return continuationContext;
        }
        set
        {
            continuationContext = value;
        }
    }

    0 references
    public object ContinuationData
    {
        get
        {
            return continuationData;
        }
        set
        {
            continuationData = value;
        }
    }
}
}
```

OptionsContext

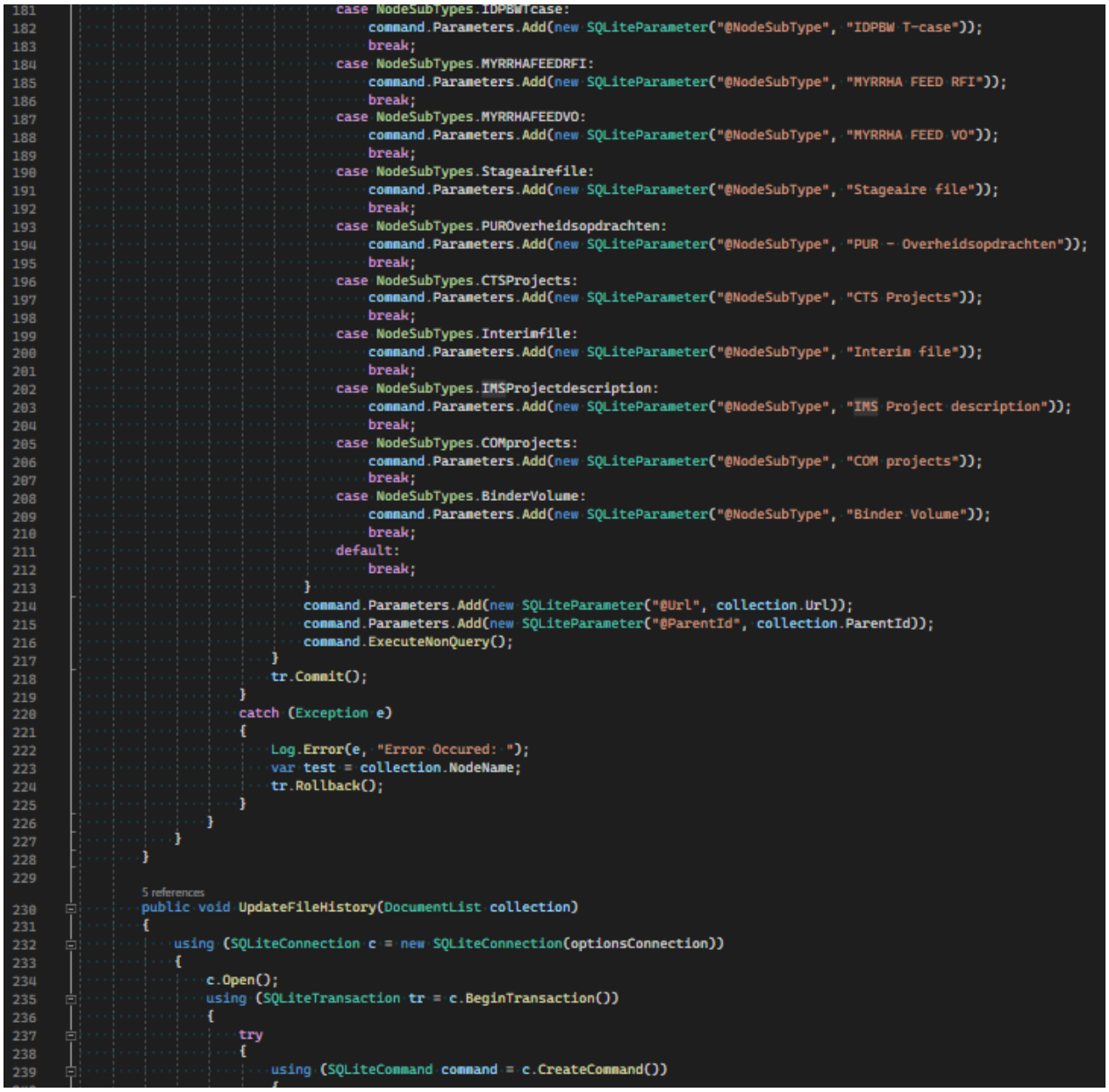
This class connects to the database and keeps all kinds of data. The tables are as follows:

- Option
- Collection
- Attribute
- FileHistory

```
using Critical_Docs_Sync.Model;
using Serilog;
using System;
using System.Collections.Generic;
using System.Data.SQLite;
using System.IO;
using System.Linq;
using System.Windows.Forms;

namespace Critical_Docs_Sync
{
    57 references
    public class OptionsContext : IDisposable
    {
        readonly string optionsConnection = "Data Source=" + Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "/AlexandriaCriticalDocs/optionsv2.0.sqlite;Version=3;";

        28 references
        public OptionsContext()
        {
            if (!File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "/AlexandriaCriticalDocs/optionsv2.0.sqlite"))
            {
                if (!Directory.Exists(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "/AlexandriaCriticalDocs"))
                {
                    Directory.CreateDirectory(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "/AlexandriaCriticalDocs");
                }
                SQLiteConnection.CreateFile(Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "/AlexandriaCriticalDocs/optionsv2.0.sqlite");

                //Create options table

                using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
                {
                    c.Open();
                    using (SQLiteCommand cmd = new SQLiteCommand("CREATE TABLE Option (Name VARCHAR(255), OptionValue VARCHAR(255))", c))
                    {
                        cmd.ExecuteNonQuery();
                    }
                    using (SQLiteCommand cmd = new SQLiteCommand("CREATE TABLE Collection (Id INTEGER PRIMARY KEY AUTOINCREMENT, Name VARCHAR(255), CollectionId VARCHAR(255))", c))
                    {
                        cmd.ExecuteNonQuery();
                    }
                    using (SQLiteCommand cmd = new SQLiteCommand("CREATE TABLE Attribute (Id INTEGER PRIMARY KEY AUTOINCREMENT, CollectionId VARCHAR(255), AttributeName VARCHAR(255), AttributeValue VARCHAR(255), Mandatory BIT)", c))
                    {
                        cmd.ExecuteNonQuery();
                    }
                    using (SQLiteCommand cmd = new SQLiteCommand("CREATE TABLE FileHistory (Id INTEGER PRIMARY KEY AUTOINCREMENT, DocumentId INTEGER, DocumentName VARCHAR(1), Version VARCHAR(255), NodeSubType VARCHAR, Url VARCHAR(255), CreationDate DATETIME, ParentId INTEGER, unique (DocumentId,ParentId))", c))
                    {
                        cmd.ExecuteNonQuery();
                    }
                    using (SQLiteCommand cmd = new SQLiteCommand(
                        "INSERT INTO Option (Name, OptionValue) " +
                        "values " +
                        "('errorMails', 'example@sckcen.be'), " +
                        "('syncTime', '24'), " +
                        "('schedule', 'False'), " +
                        "('otdsURL', ''), " +
                        "('localDirectory', '" + Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) + "\\'), " +
                        "('lastRun', 'none'), " +
                        "('lastSuccessRun', 'none'), " +
                        "('alexandriaURL', '')", c))
                    {
                        cmd.ExecuteNonQuery();
                    }
```

```
            }
        }
    }

    2 references
    public List<DocumentHistory> GetFileHistory()
    {
        List<DocumentHistory> history = new List<DocumentHistory>();
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteCommand cmd = new SQLiteCommand($"select * from FileHistory", c))
            {
                using (SQLiteDataReader rdr = cmd.ExecuteReader())
                {
                    while (rdr.Read())
                    {
                        history.Add(new DocumentHistory
                        {
                            Id = Convert.ToInt32(rdr["Id"].ToString()),
                            DocumentId = Convert.ToInt32(rdr["DocumentId"].ToString()),
                            DocumentName = rdr["DocumentName"].ToString(),
                            Version = Convert.ToInt32(rdr["Version"].ToString()),
                            NodeSubType = rdr["NodeSubType"].ToString(),
                            Url = rdr["Url"].ToString(),
                            CreationDate = Convert.ToDateTime(rdr["CreationDate"].ToString()),
                            ParentId = Convert.ToInt32(rdr["ParentId"].ToString())
                        });
                    }
                }
            }
        }
        return history;
    }

    6 references
    public void CreateFileHistory(DocumentList collection)
    {
        List<AlexandriaSourceOptions> sourceOptions = new List<AlexandriaSourceOptions>();
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteTransaction tr = c.BeginTransaction())
            {
                try
                {
                    using (SQLiteCommand command = c.CreateCommand())
                    {
                        command.Transaction = tr;
                        command.CommandText = $"INSERT or IGNORE INTO FileHistory('DocumentId', 'DocumentName', 'CreationDate', 'Version', 'NodeSubType', 'Url', 'ParentId') VALUES (@DocumentId, @DocumentName, @CreationDate, @Version, @NodeSubType, @Url, @ParentId)";
                        command.Parameters.Add(new SQLiteParameter("@DocumentId", collection.NodeId));
                        command.Parameters.Add(new SQLiteParameter("@DocumentName", collection.NodeName));
                        command.Parameters.Add(new SQLiteParameter("@CreationDate", collection.CreationDate));
                        command.Parameters.Add(new SQLiteParameter("@Version", collection.Version));
                        switch (collection.NodeSubType)
                        {
                            case NodeSubTypes.Folder:
                                command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Folder"));
                                break;
                            case NodeSubTypes.Shortcut:
                                command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Shortcut"));
```

```
                    break;
                case NodeSubTypes.Generation:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Generation"));
                    break;
                case NodeSubTypes.CompoundDocument:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Compound Document"));
                    break;
                case NodeSubTypes.URL:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "URL"));
                    break;
                case NodeSubTypes.Document:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Document"));
                    break;
                case NodeSubTypes.TextDocument:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Text Document"));
                    break;
                case NodeSubTypes.Collection:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Collection"));
                    break;
                case NodeSubTypes.Email:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Email"));
                    break;
                case NodeSubTypes.EmailFolder:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Email Folder"));
                    break;
                case NodeSubTypes.Binder:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Binder"));
                    break;
                case NodeSubTypes.WBSElement:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "WBS Element"));
                    break;
                case NodeSubTypes.IMSProcessDescription:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IMS Process Description"));
                    break;
                case NodeSubTypes.EmployeeFile:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Employee File"));
                    break;
                case NodeSubTypes.CustomerContract:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Customer Contract"));
                    break;
                case NodeSubTypes.CEKTraining:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "CEK Training"));
                    break;
                case NodeSubTypes.CustomerQuotation:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Customer Quotation"));
                    break;
                case NodeSubTypes.IDPBWFcase:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW F-case"));
                    break;
                case NodeSubTypes.TMB:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "TMB"));
                    break;
                case NodeSubTypes.IDPBWEcase:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW E-case"));
                    break;
                case NodeSubTypes.IDPBWHSEcase:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW HSE-case"));
                    break;
                case NodeSubTypes.IDPBWScase:
                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW S-case"));
                    break;
```



```
                                case NodeSubTypes.IDPBWTcase:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW T-case"));
                                    break;
                                case NodeSubTypes.MYRRHAFEEDRFI:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "MYRRHA FEED RFI"));
                                    break;
                                case NodeSubTypes.MYRRHAFEEDVO:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "MYRRHA FEED VO"));
                                    break;
                                case NodeSubTypes.Stageairefile:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Stageaire file"));
                                    break;
                                case NodeSubTypes.PUROverheidsopdrachten:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "PUR - Overheidsopdrachten"));
                                    break;
                                case NodeSubTypes.CTSProjects:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "CTS Projects"));
                                    break;
                                case NodeSubTypes.Interimfile:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Interim file"));
                                    break;
                                case NodeSubTypes.IMSProjectdescription:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IMS Project description"));
                                    break;
                                case NodeSubTypes.COMprojects:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "COM projects"));
                                    break;
                                case NodeSubTypes.BinderVolume:
                                    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Binder Volume"));
                                    break;
                                default:
                                    break;
                            }
                            command.Parameters.Add(new SQLiteParameter("@Url", collection.Url));
                            command.Parameters.Add(new SQLiteParameter("@ParentId", collection.ParentId));
                            command.ExecuteNonQuery();
                        }
                        tr.Commit();
                    }
                    catch (Exception e)
                    {
                        Log.Error(e, "Error Occured: ");
                        var test = collection.NodeName;
                        tr.Rollback();
                    }
                }
            }
        }

        5 references
        public void UpdateFileHistory(DocumentList collection)
        {
            using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
            {
                c.Open();
                using (SQLiteTransaction tr = c.BeginTransaction())
                {
                    try
                    {
                        using (SQLiteCommand command = c.CreateCommand())
```

```
                    {
                        command.Transaction = tr;
                        command.CommandText = $"update FileHistory set DocumentName = @DocumentName, CreationDate = @CreationDate, Version = @Version where DocumentId = " + collection.NodeId;
                        command.Parameters.Add(new SQLiteParameter("@DocumentName", collection.NodeName));
                        command.Parameters.Add(new SQLiteParameter("@CreationDate", collection.CreationDate));
                        command.Parameters.Add(new SQLiteParameter("@Version", collection.Version));
                        command.Parameters.Add(new SQLiteParameter("@Url", collection.Url));
                        command.ExecuteNonQuery();
                    }
                    tr.Commit();
                }
                catch (Exception)
                {
                    tr.Rollback();
                }
            }
        }
    }

    2 references
    public void DeleteFileHistory(int historyId)
    {
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteTransaction tr = c.BeginTransaction())
            {
                try
                {
                    using (SQLiteCommand command = c.CreateCommand())
                    {
                        command.Transaction = tr;
                        command.CommandText = $"delete from FileHistory where Id = @ItemId";
                        command.Parameters.Add(new SQLiteParameter("@ItemId", historyId));
                        command.ExecuteNonQuery();
                    }
                    tr.Commit();
                }
                catch (Exception)
                {
                    tr.Rollback();
                    throw;
                }
            }
        }
    }

    18 references
    public KeyValuePair<string, string> GetOption(string optionName)
    {
        KeyValuePair<string, string> option = new KeyValuePair<string, string>();
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteCommand cmd = new SQLiteCommand($"select * from Option where Name = @Name", c))
            {
                cmd.Parameters.Add(new SQLiteParameter("@Name", optionName));
                using (SQLiteDataReader rdr = cmd.ExecuteReader())
                {

                    if (rdr.Read())
```

```
                    {
                        option = new KeyValuePair<string, string>(rdr["Name"].ToString(), rdr["OptionValue"].ToString());
                    }
                }
            }
        }
        return option;
    }

    0 references
    public List<KeyValuePair<string, string>> GetOptions()
    {
        List<KeyValuePair<string, string>> options = new List<KeyValuePair<string, string>>();
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteCommand cmd = new SQLiteCommand($"select * from Option", c))
            {
                using (SQLiteDataReader rdr = cmd.ExecuteReader())
                {
                    while (rdr.Read())
                    {
                        options.Add(new KeyValuePair<string, string>(rdr["Name"].ToString(), rdr["OptionValue"].ToString()));
                    }
                }
            }
        }
        return options;
    }

    4 references
    public List<AlexandriaSourceOptions> GetCollections()
    {
        List<AlexandriaSourceOptions> alexandriaSourceOptions = new List<AlexandriaSourceOptions>();
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteCommand cmd = new SQLiteCommand($"select * from Collection", c))
            {
                using (SQLiteDataReader rdr = cmd.ExecuteReader())
                {
                    while (rdr.Read())
                    {
                        alexandriaSourceOptions.Add(new AlexandriaSourceOptions()
                        {
                            Id = Convert.ToInt32(rdr["Id"].ToString()),
                            Name = rdr["Name"].ToString(),
                            DataId = Convert.ToInt32(rdr["CollectionId"].ToString()),
                            Attributes = new List<Model.Attribute>()
                        });
                    }
                }
            }
        }

        foreach (AlexandriaSourceOptions item in alexandriaSourceOptions)
        {
            using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
            {
                c.Open();
                using (SQLiteCommand cmd = new SQLiteCommand($"select * from Attribute WHERE CollectionId = @ItemId", c))
```

```
                {
                    cmd.Parameters.Add(new SQLiteParameter("@ItemId", item.Id));

                    using (SQLiteDataReader rdr = cmd.ExecuteReader())
                    {
                        while (rdr.Read())
                        {
                            Model.Attribute attribute = new Model.Attribute()
                            {
                                Id = Convert.ToInt32(rdr["Id"].ToString()),
                                Name = rdr["AttributeName"].ToString(),
                                Value = rdr["AttributeValue"].ToString(),
                                Mandatory = Convert.ToBoolean(rdr["Mandatory"].ToString())
                            };
                            item.Attributes.Add(attribute);
                        }
                    }
                }
            }
        }
        return alexandriaSourceOptions;
    }

    12 references
    public void UpdateOptions(List<KeyValuePair<string, string>> options)
    {
        foreach (KeyValuePair<string, string> item in options)
        {
            using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
            {
                c.Open();
                using (SQLiteCommand cmd = new SQLiteCommand($"UPDATE Option SET OptionValue = @Value where Name = @Name", c))
                {
                    cmd.Parameters.Add(new SQLiteParameter("@Value", item.Value));
                    cmd.Parameters.Add(new SQLiteParameter("@Name", item.Key));
                    cmd.ExecuteNonQuery();
                }
            }
        }
    }

    1 reference
    public void UpdateCollections(List<AlexandriaSourceOptions> collection)
    {
        List<AlexandriaSourceOptions> existingCollection = GetCollections();
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteTransaction tr = c.BeginTransaction())
            {
                try
                {
                    using (SQLiteCommand command = c.CreateCommand())
                    {
                        command.Transaction = tr;

                        foreach (AlexandriaSourceOptions item in existingCollection)
                        {
                            var check = collection.Find(x => x.Id == item.Id && x.DataId == item.DataId);
                            var checkNames = existingCollection.Where(x => x.Name == item.Name).Count() > 1;
                            if (check == null)
```

```
                {
                    command.CommandText = $"DELETE FROM Collection where Id = @ID AND CollectionId = @CollID";
                    command.Parameters.Add(new SQLiteParameter("@ID", item.Id));
                    command.Parameters.Add(new SQLiteParameter("@CollID", item.DataId));
                    command.ExecuteNonQuery();

                    command.CommandText = $"delete from FileHistory";
                    command.ExecuteNonQuery();

                    command.CommandText = $"DELETE FROM Attribute where CollectionId = @CollId";
                    command.Parameters.Add(new SQLiteParameter("@CollID", item.Id));
                    command.ExecuteNonQuery();
                }
                else if (!checkNames)
                {
                    command.CommandText = $"UPDATE Collection SET Name = @Name  WHERE Id = @ID AND CollectionId = @CollID";
                    command.Parameters.Add(new SQLiteParameter("@Name", check.Name));
                    command.Parameters.Add(new SQLiteParameter("@ID", item.Id));
                    command.Parameters.Add(new SQLiteParameter("@CollID", item.DataId));
                    command.ExecuteNonQuery();

                    command.CommandText = $"DELETE FROM Attribute where CollectionId = @CollId";
                    command.Parameters.Add(new SQLiteParameter("@CollID", item.Id));
                    command.ExecuteNonQuery();


                    foreach (Model.Attribute att in check.Attributes)
                    {
                        command.CommandText = $"INSERT INTO Attribute('CollectionId','AttributeName','AttributeValue', 'Mandatory') VALUES (@CollectionId, @AttributeName, @AttributeValue, @Mandatory)";
                        command.Parameters.Add(new SQLiteParameter("@CollectionId", item.Id));
                        command.Parameters.Add(new SQLiteParameter("@AttributeName", att.Name));
                        command.Parameters.Add(new SQLiteParameter("@AttributeValue", att.Value));
                        command.Parameters.Add(new SQLiteParameter("@Mandatory", att.Mandatory));
                        command.ExecuteNonQuery();
                    }
                    collection.Remove(check);
                }
            }

            foreach (AlexandriaSourceOptions item in collection)
            {
                command.CommandText = $"INSERT INTO Collection('Name', 'CollectionId') VALUES (@Name , @CollectionId)";
                command.Parameters.Add(new SQLiteParameter("@Name", item.Name));
                command.Parameters.Add(new SQLiteParameter("@CollectionId", item.DataId));
                command.ExecuteNonQuery();

                item.Id = Convert.ToInt32(c.LastInsertRowId);

                foreach (Model.Attribute att in item.Attributes)
                {
                    command.CommandText = $"INSERT INTO Attribute('CollectionId','AttributeName','AttributeValue', 'Mandatory') VALUES (@CollectionId, @AttributeName, @AttributeValue, @Mandatory)";
                    command.Parameters.Add(new SQLiteParameter("@CollectionId", item.Id));
                    command.Parameters.Add(new SQLiteParameter("@AttributeName", att.Name));
                    command.Parameters.Add(new SQLiteParameter("@AttributeValue", att.Value));
                    command.Parameters.Add(new SQLiteParameter("@Mandatory", att.Mandatory));
                    command.ExecuteNonQuery();
                }
            }
        }
        tr.Commit();
```

```
                }
                catch (Exception)
                {
                    tr.Rollback();
                }
            }
        }

        0 references
        public void Dispose()
        {

        }
    }
}
```

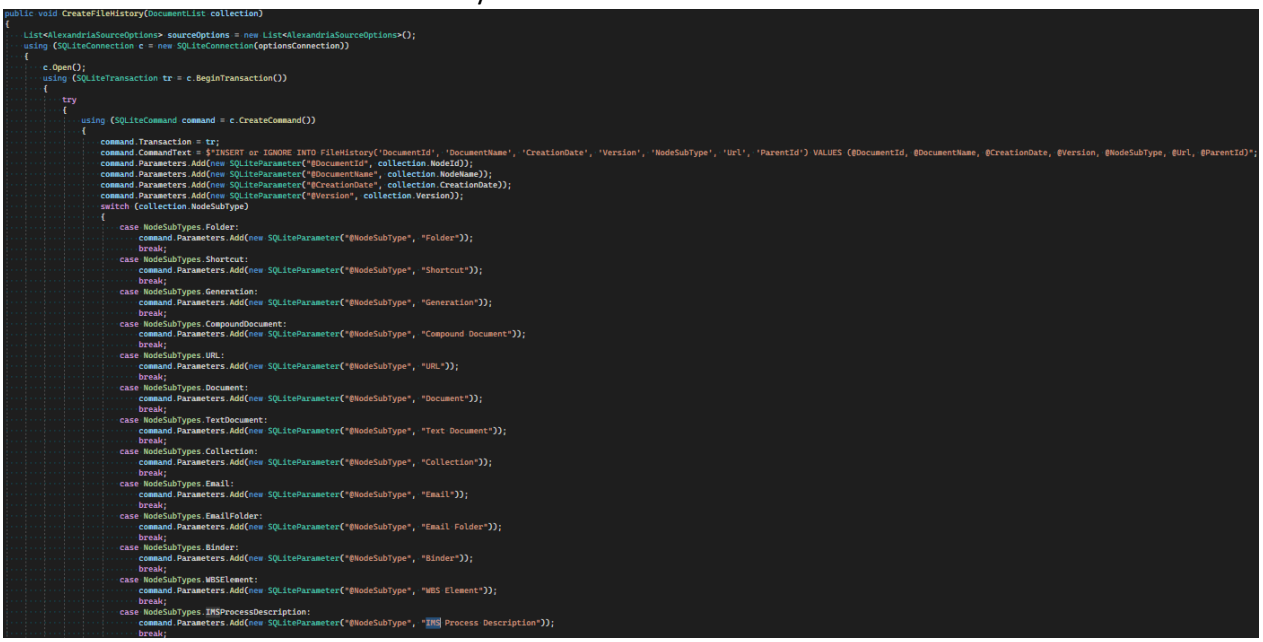
GetFileHistory

This method is used to extract items from the FileHistory table and insert them into the DocumentHistory list.

```
public List<DocumentHistory> GetFileHistory()
{
    List<DocumentHistory> history = new List<DocumentHistory>();
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteCommand cmd = new SQLiteCommand($"select * from FileHistory", c))
        {
            using (SQLiteDataReader rdr = cmd.ExecuteReader())
            {
                while (rdr.Read())
                {
                    history.Add(new DocumentHistory
                    {
                        Id = Convert.ToInt32(rdr["Id"].ToString()),
                        DocumentId = Convert.ToInt32(rdr["DocumentId"].ToString()),
                        DocumentName = rdr["DocumentName"].ToString(),
                        Version = Convert.ToInt32(rdr["Version"].ToString()),
                        NodeSubType = rdr["NodeSubType"].ToString(),
                        Url = rdr["URL"].ToString(),
                        CreationDate = Convert.ToDateTime(rdr["CreationDate"].ToString()),
                        ParentId = Convert.ToInt32(rdr["ParentId"].ToString())
                    });
                }
            }
        }
    }
    return history;
}
```

CreateFileHistory(DocumentList collection)

This method is used to fill the FileHistory tables.



```
public void CreateFileHistory(DocumentList collection)
{
    List<AlexandriaSourceOptions> sourceOptions = new List<AlexandriaSourceOptions>();
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteTransaction tr = c.BeginTransaction())
        {
            try
            {
                using (SQLiteCommand command = c.CreateCommand())
                {
                    command.Transaction = tr;
                    command.CommandText = $"INSERT or IGNORE INTO FileHistory('DocumentId', 'DocumentName', 'CreationDate', 'Version', 'NodeSubType', 'Url', 'ParentId') VALUES (@DocumentId, @DocumentName, @CreationDate, @Version, @NodeSubType, @Url, @ParentId)";
                    command.Parameters.Add(new SQLiteParameter("@DocumentId", collection.NodeId));
                    command.Parameters.Add(new SQLiteParameter("@DocumentName", collection.NodeName));
                    command.Parameters.Add(new SQLiteParameter("@CreationDate", collection.CreationDate));
                    command.Parameters.Add(new SQLiteParameter("@Version", collection.Version));
                    switch (collection.NodeSubType)
                    {
                        case NodeSubTypes.Folder:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Folder"));
                            break;
                        case NodeSubTypes.Shortcut:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Shortcut"));
                            break;
                        case NodeSubTypes.Generation:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Generation"));
                            break;
                        case NodeSubTypes.CompoundDocument:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Compound Document"));
                            break;
                        case NodeSubTypes.URL:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "URL"));
                            break;
                        case NodeSubTypes.Document:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Document"));
                            break;
                        case NodeSubTypes.TextDocument:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Text Document"));
                            break;
                        case NodeSubTypes.Collection:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Collection"));
                            break;
                        case NodeSubTypes.Email:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Email"));
                            break;
                        case NodeSubTypes.EmailFolder:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Email Folder"));
                            break;
                        case NodeSubTypes.Binder:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Binder"));
                            break;
                        case NodeSubTypes.WBSElement:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "WBS Element"));
                            break;
                        case NodeSubTypes.IMSProcessDescription:
                            command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IMS Process Description"));
                            break;
```

```
case NodeSubTypes.EmployeeFile:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Employee File"));
    break;
case NodeSubTypes.CustomerContract:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Customer Contract"));
    break;
case NodeSubTypes.CEKTraining:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "CEK Training"));
    break;
case NodeSubTypes.CustomerQuotation:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Customer Quotation"));
    break;
case NodeSubTypes.IDPBWFcase:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW F-case"));
    break;
case NodeSubTypes.TMB:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "TMB"));
    break;
case NodeSubTypes.IDPBWEcase:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW E-case"));
    break;
case NodeSubTypes.IDPBWHSEcase:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW HSE-case"));
    break;
case NodeSubTypes.IDPBWScase:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW S-case"));
    break;
case NodeSubTypes.IDPBWTcase:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IDPBW T-case"));
    break;
case NodeSubTypes.MYRRHAFEEDRFI:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "MYRRHA FEED RFI"));
    break;
case NodeSubTypes.MYRRHAFEEDVO:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "MYRRHA FEED VO"));
    break;
case NodeSubTypes.Stageairefile:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Stageaire file"));
    break;
case NodeSubTypes.PUROverheidsopdrachten:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "PUR - Overheidsopdrachten"));
    break;
case NodeSubTypes.CTSProjects:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "CTS Projects"));
    break;
case NodeSubTypes.Interimfile:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Interim file"));
    break;
case NodeSubTypes.IMSProjectdescription:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "IMS Project description"));
    break;
case NodeSubTypes.COMprojects:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "COM projects"));
    break;
case NodeSubTypes.BinderVolume:
    command.Parameters.Add(new SQLiteParameter("@NodeSubType", "Binder Volume"));
    break;
default:
    break;
```

```
                }
                command.Parameters.Add(new SQLiteParameter("@Url", collection.Url));
                command.Parameters.Add(new SQLiteParameter("@ParentId", collection.ParentId));
                command.ExecuteNonQuery();
            }
            tr.Commit();
        }
        catch (Exception e)
        {
            Log.Error(e, "Error Occured: ");
            var test = collection.NodeName;
            tr.Rollback();
        }
    }
  }
}
```

UpdateFileHistory(DocumentList collection)

This method is used to update documents.

```
public void UpdateFileHistory(DocumentList collection)
{
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteTransaction tr = c.BeginTransaction())
        {
            try
            {
                using (SQLiteCommand command = c.CreateCommand())
                {
                    command.Transaction = tr;
                    command.CommandText = $"update FileHistory set DocumentName = @DocumentName, CreationDate = @CreationDate, Version = @Version where DocumentId = " + collection.NodeId;
                    command.Parameters.Add(new SQLiteParameter("@DocumentName", collection.NodeName));
                    command.Parameters.Add(new SQLiteParameter("@CreationDate", collection.CreationDate));
                    command.Parameters.Add(new SQLiteParameter("@Version", collection.Version));
                    command.Parameters.Add(new SQLiteParameter("@Url", collection.Url));
                    command.ExecuteNonQuery();
                }
                tr.Commit();
            }
            catch (Exception)
            {
                tr.Rollback();
            }
        }
    }
}
```

DeleteFileHistory(int historyId)

This method is used if a document is removed from Alexandria and you want to download from Alexandria again.

```
public void DeleteFileHistory(int historyId)
{
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteTransaction tr = c.BeginTransaction())
        {
            try
            {
                using (SQLiteCommand command = c.CreateCommand())
                {
                    command.Transaction = tr;
                    command.CommandText = $"delete from FileHistory where Id = @ItemId";
                    command.Parameters.Add(new SQLiteParameter("@ItemId", historyId));
                    command.ExecuteNonQuery();
                }
                tr.Commit();
            }
            catch (Exception)
            {
                tr.Rollback();
                throw;
            }
        }
    }
}
```

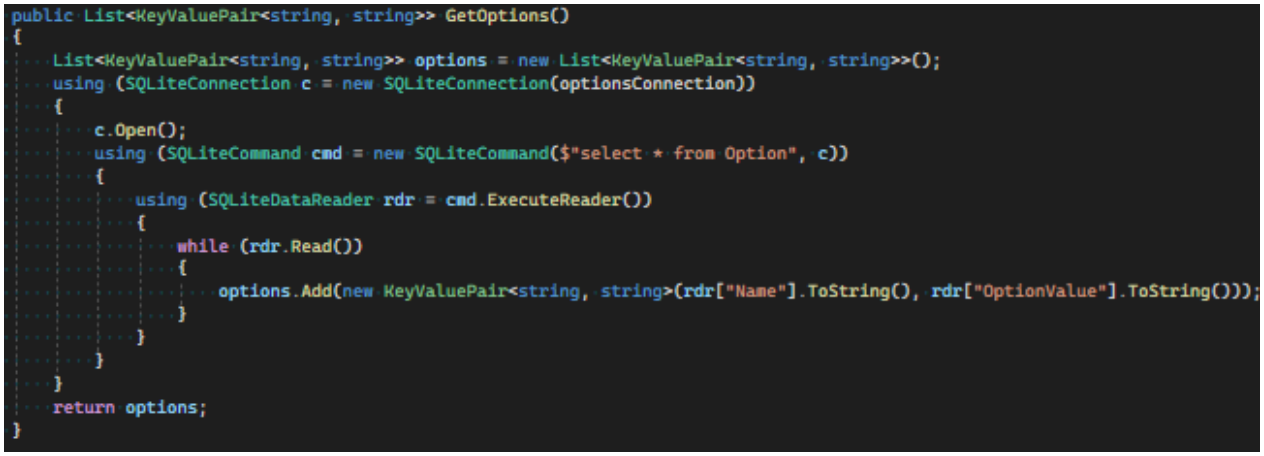
GetOption(string optionName)

This method is used to get an option with the same name.

```
public KeyValuePair<string, string> GetOption(string optionName)
{
    KeyValuePair<string, string> option = new KeyValuePair<string, string>();
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteCommand cmd = new SQLiteCommand($"select * from Option where Name = @Name", c))
        {
            cmd.Parameters.Add(new SQLiteParameter("@Name", optionName));
            using (SQLiteDataReader rdr = cmd.ExecuteReader())
            {

                if (rdr.Read())
                {
                    option = new KeyValuePair<string, string>(rdr["Name"].ToString(), rdr["OptionValue"].ToString());
                }
            }
        }
    }
    return option;
}
```

GetOptions

This method gets the whole Options table.



```
public List<KeyValuePair<string, string>> GetOptions()
{
    List<KeyValuePair<string, string>> options = new List<KeyValuePair<string, string>>();
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteCommand cmd = new SQLiteCommand($"select * from Option", c))
        {
            using (SQLiteDataReader rdr = cmd.ExecuteReader())
            {
                while (rdr.Read())
                {
                    options.Add(new KeyValuePair<string, string>(rdr["Name"].ToString(), rdr["OptionValue"].ToString()));
                }
            }
        }
    }
    return options;
}
```

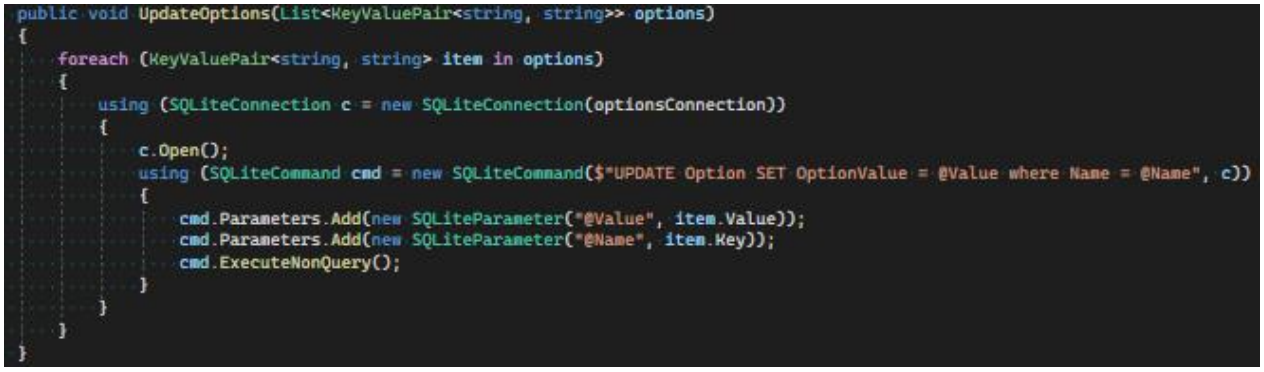
GetCollection

This method is used to fetch containers and their attributes.

```
public List<AlexandriaSourceOptions> GetCollections()
{
    List<AlexandriaSourceOptions> alexandriaSourceOptions = new List<AlexandriaSourceOptions>();
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteCommand cmd = new SQLiteCommand($"select * from Collection", c))
        {
            using (SQLiteDataReader rdr = cmd.ExecuteReader())
            {
                while (rdr.Read())
                {
                    alexandriaSourceOptions.Add(new AlexandriaSourceOptions()
                    {
                        Id = Convert.ToInt32(rdr["Id"].ToString()),
                        Name = rdr["Name"].ToString(),
                        DataId = Convert.ToInt32(rdr["CollectionId"].ToString()),
                        Attributes = new List<Model.Attribute>()
                    });
                }
            }
        }
    }

    foreach (AlexandriaSourceOptions item in alexandriaSourceOptions)
    {
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteCommand cmd = new SQLiteCommand($"select * from Attribute WHERE CollectionId = @ItemId", c))
            {
                cmd.Parameters.Add(new SQLiteParameter("@ItemId", item.Id));

                using (SQLiteDataReader rdr = cmd.ExecuteReader())
                {
                    while (rdr.Read())
                    {
                        Model.Attribute attribute = new Model.Attribute()
                        {
                            Id = Convert.ToInt32(rdr["Id"].ToString()),
                            Name = rdr["AttributeName"].ToString(),
                            Value = rdr["AttributeValue"].ToString(),
                            Mandatory = Convert.ToBoolean(rdr["Mandatory"].ToString())
                        };
                        item.Attributes.Add(attribute);
                    }
                }
            }
        }
    }
    return alexandriaSourceOptions;
}
```

UpdateOptions(List<KeyValuePair<string, string>> options)

This method is used to change an option.



```
public void UpdateOptions(List<KeyValuePair<string, string>> options)
{
    foreach (KeyValuePair<string, string> item in options)
    {
        using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
        {
            c.Open();
            using (SQLiteCommand cmd = new SQLiteCommand($"UPDATE Option SET OptionValue = @Value where Name = @Name", c))
            {
                cmd.Parameters.Add(new SQLiteParameter("@Value", item.Value));
                cmd.Parameters.Add(new SQLiteParameter("@Name", item.Key));
                cmd.ExecuteNonQuery();
            }
        }
    }
}
```

UpdateCollections(List<AlexandriaSourceOptions> collection)

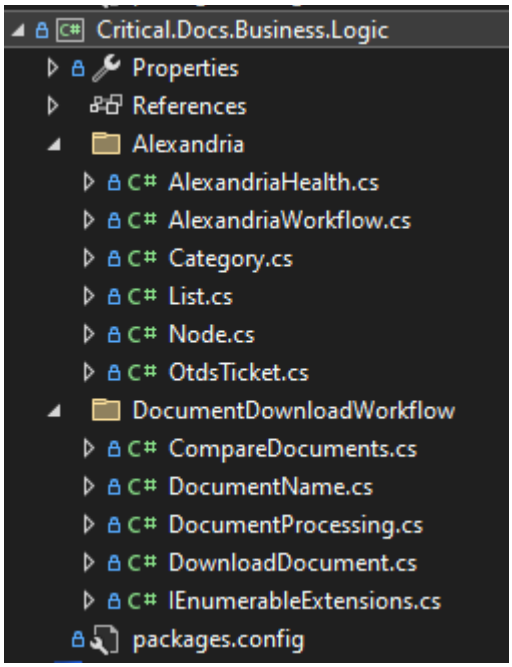
This method is used to update containers and their attributes.

```
public void UpdateCollections(List<AlexandriaSourceOptions> collection)
{
    List<AlexandriaSourceOptions> existingCollection = GetCollections();
    using (SQLiteConnection c = new SQLiteConnection(optionsConnection))
    {
        c.Open();
        using (SQLiteTransaction tr = c.BeginTransaction())
        {
            try
            {
                using (SQLiteCommand command = c.CreateCommand())
                {
                    command.Transaction = tr;

                    foreach (AlexandriaSourceOptions item in existingCollection)
                    {
                        var check = collection.Find(x => x.Id == item.Id && x.DataId == item.DataId);
                        var checkNames = existingCollection.Where(x => x.Name == item.Name).Count() > 1;
                        if (check == null)
                        {
                            command.CommandText = $"DELETE FROM Collection where Id = @ID AND CollectionId = @CollID";
                            command.Parameters.Add(new SQLiteParameter("@ID", item.Id));
                            command.Parameters.Add(new SQLiteParameter("@CollID", item.DataId));
                            command.ExecuteNonQuery();

                            command.CommandText = $"delete from FileHistory";
                            command.ExecuteNonQuery();

                            command.CommandText = $"DELETE FROM Attribute where CollectionId = @CollId";
                            command.Parameters.Add(new SQLiteParameter("@CollID", item.Id));
                            command.ExecuteNonQuery();
                        }
                        else if (!checkNames)
                        {
                            command.CommandText = $"UPDATE Collection SET Name = @Name  WHERE Id = @ID AND CollectionId = @CollID";
                            command.Parameters.Add(new SQLiteParameter("@Name", check.Name));
                            command.Parameters.Add(new SQLiteParameter("@ID", item.Id));
                            command.Parameters.Add(new SQLiteParameter("@CollID", item.DataId));
                            command.ExecuteNonQuery();

                            command.CommandText = $"DELETE FROM Attribute where CollectionId = @CollId";
                            command.Parameters.Add(new SQLiteParameter("@CollID", item.Id));
                            command.ExecuteNonQuery();


                            foreach (Model.Attribute att in check.Attributes)
                            {
                                command.CommandText = $"INSERT INTO Attribute('CollectionId','AttributeName','AttributeValue', 'Mandatory') VALUES (@CollectionId, @AttributeName, @AttributeValue, @Mandatory)";
                                command.Parameters.Add(new SQLiteParameter("@CollectionId", item.Id));
                                command.Parameters.Add(new SQLiteParameter("@AttributeName", att.Name));
                                command.Parameters.Add(new SQLiteParameter("@AttributeValue", att.Value));
                                command.Parameters.Add(new SQLiteParameter("@Mandatory", att.Mandatory));
                                command.ExecuteNonQuery();
                            }
                            collection.Remove(check);
                        }
                    }

                    foreach (AlexandriaSourceOptions item in collection)
                    {
                        command.CommandText = $"INSERT INTO Collection('Name', 'CollectionId') VALUES (@Name , @CollectionId)";
                        command.Parameters.Add(new SQLiteParameter("@Name", item.Name));
                        command.Parameters.Add(new SQLiteParameter("@CollectionId", item.DataId));
                        command.ExecuteNonQuery();

                        item.Id = Convert.ToInt32(c.LastInsertRowId);

                        foreach (Model.Attribute att in item.Attributes)
                        {
                            command.CommandText = $"INSERT INTO Attribute('CollectionId','AttributeName','AttributeValue', 'Mandatory') VALUES (@CollectionId, @AttributeName, @AttributeValue, @Mandatory)";
                            command.Parameters.Add(new SQLiteParameter("@CollectionId", item.Id));
                            command.Parameters.Add(new SQLiteParameter("@AttributeName", att.Name));
                            command.Parameters.Add(new SQLiteParameter("@AttributeValue", att.Value));
                            command.Parameters.Add(new SQLiteParameter("@Mandatory", att.Mandatory));
                            command.ExecuteNonQuery();
                        }
                    }
                }
                tr.Commit();
            }
            catch (Exception)
            {
                tr.Rollback();
            }
        }
    }
}
```

Alexandria.Docs.Business.Logic

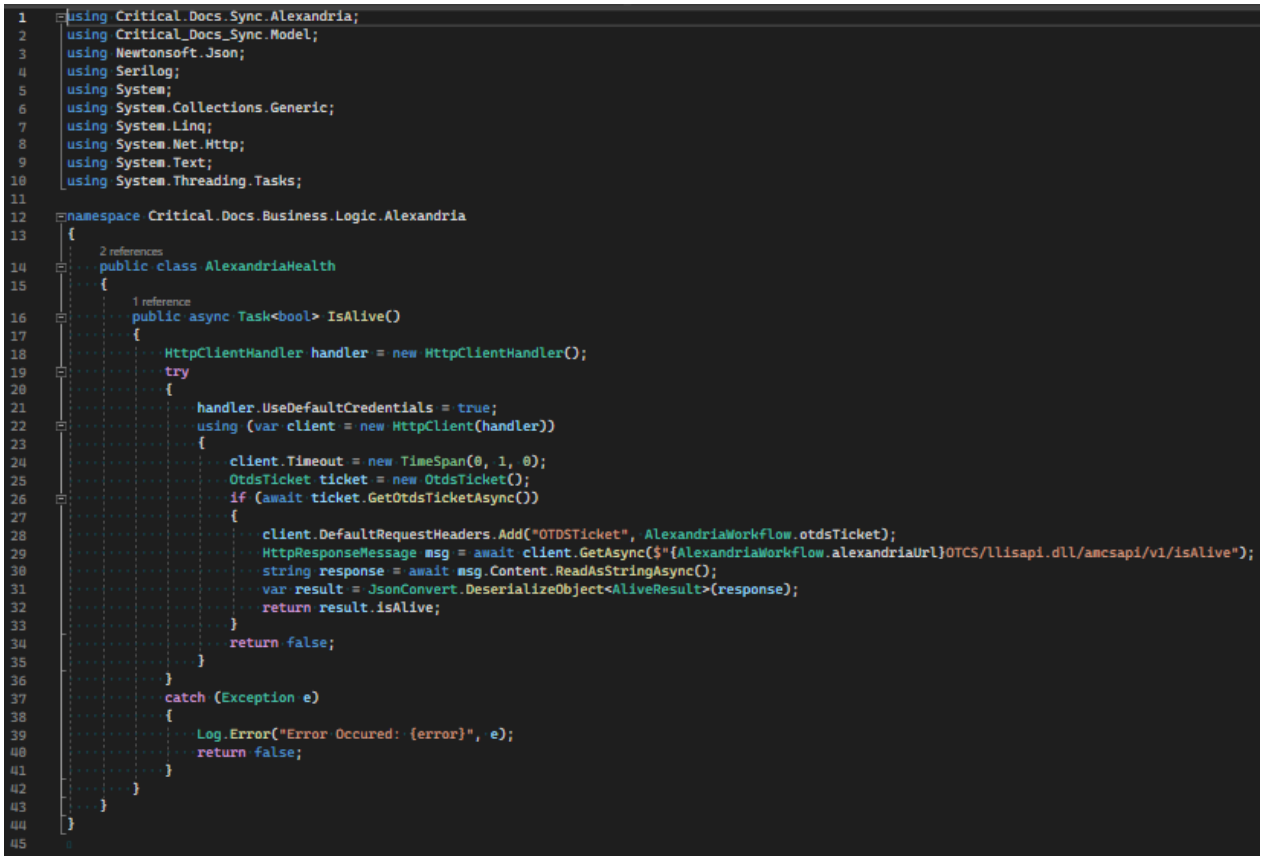
This project contains the business logic to download documents and work with Alexandria's workflow.
This project contains 3 packages:

- NewtonSoft.Json,
- Serilog and Serilog.Sinks.File



AlexandriaHealth

This class checks if Alexandria is accessible using an otds-ticket.

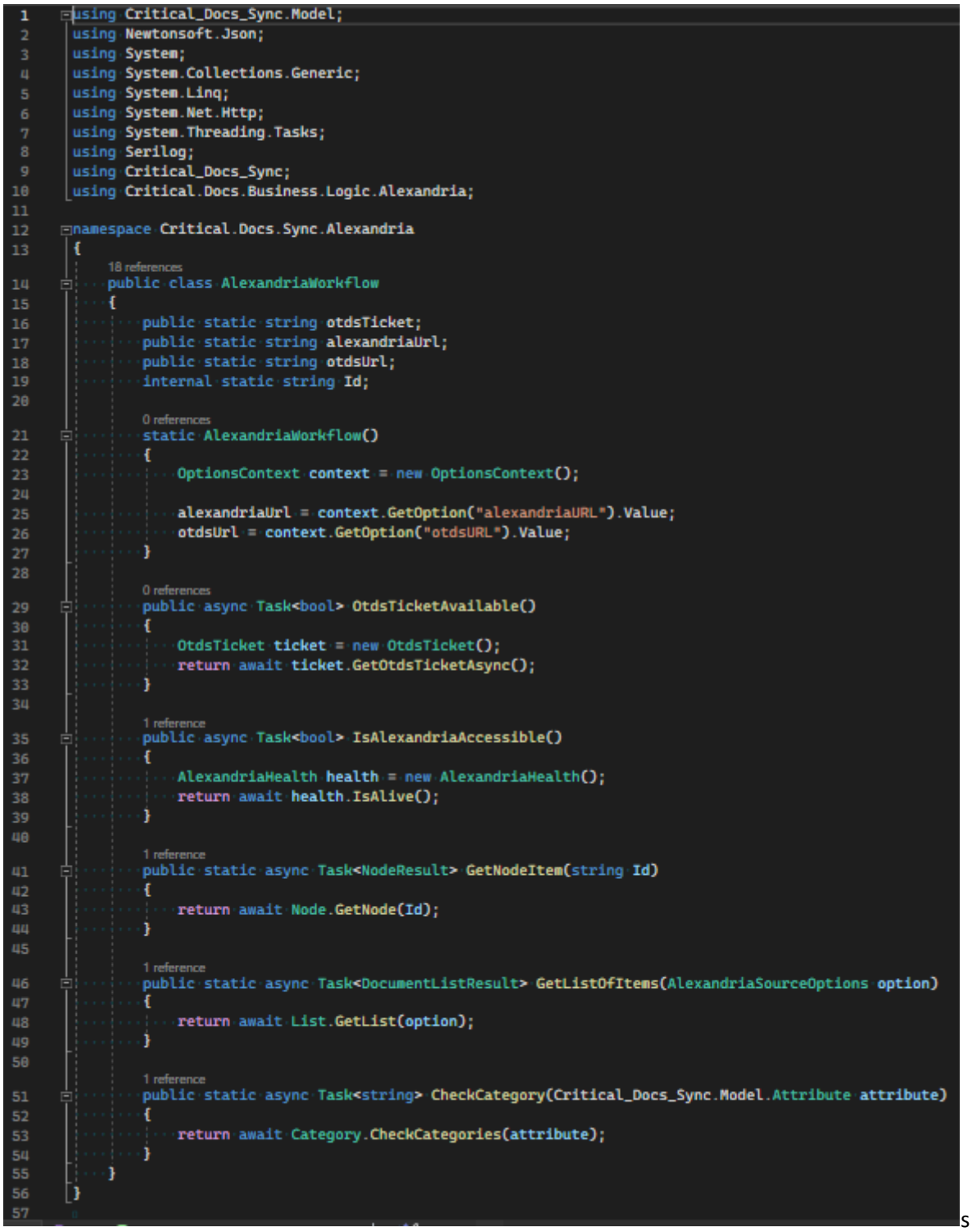


```
using Critical.Docs.Sync.Alexandria;
using Critical_Docs_Sync.Model;
using Newtonsoft.Json;
using Serilog;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace Critical.Docs.Business.Logic.Alexandria
{
    2 references
    public class AlexandriaHealth
    {
        1 reference
        public async Task<bool> IsAlive()
        {
            HttpClientHandler handler = new HttpClientHandler();
            try
            {
                handler.UseDefaultCredentials = true;
                using (var client = new HttpClient(handler))
                {
                    client.Timeout = new TimeSpan(0, 1, 0);
                    OtdsTicket ticket = new OtdsTicket();
                    if (await ticket.GetOtdsTicketAsync())
                    {
                        client.DefaultRequestHeaders.Add("OTDSTicket", AlexandriaWorkflow.otdsTicket);
                        HttpResponseMessage msg = await client.GetAsync($"{AlexandriaWorkflow.alexandriaUrl}OTCS/llisapi.dll/amcsapi/v1/isAlive");
                        string response = await msg.Content.ReadAsStringAsync();
                        var result = JsonConvert.DeserializeObject<AliveResult>(response);
                        return result.isAlive;
                    }
                    return false;
                }
            }
            catch (Exception e)
            {
                Log.Error("Error Occured: {error}", e);
                return false;
            }
        }
    }
}
```

AlexandriaWorkflow

This class is the ‘Hub’ to get to the other classes:

- AlexandriaHealth
- Category
- List
- Node
- OtdsTicket.

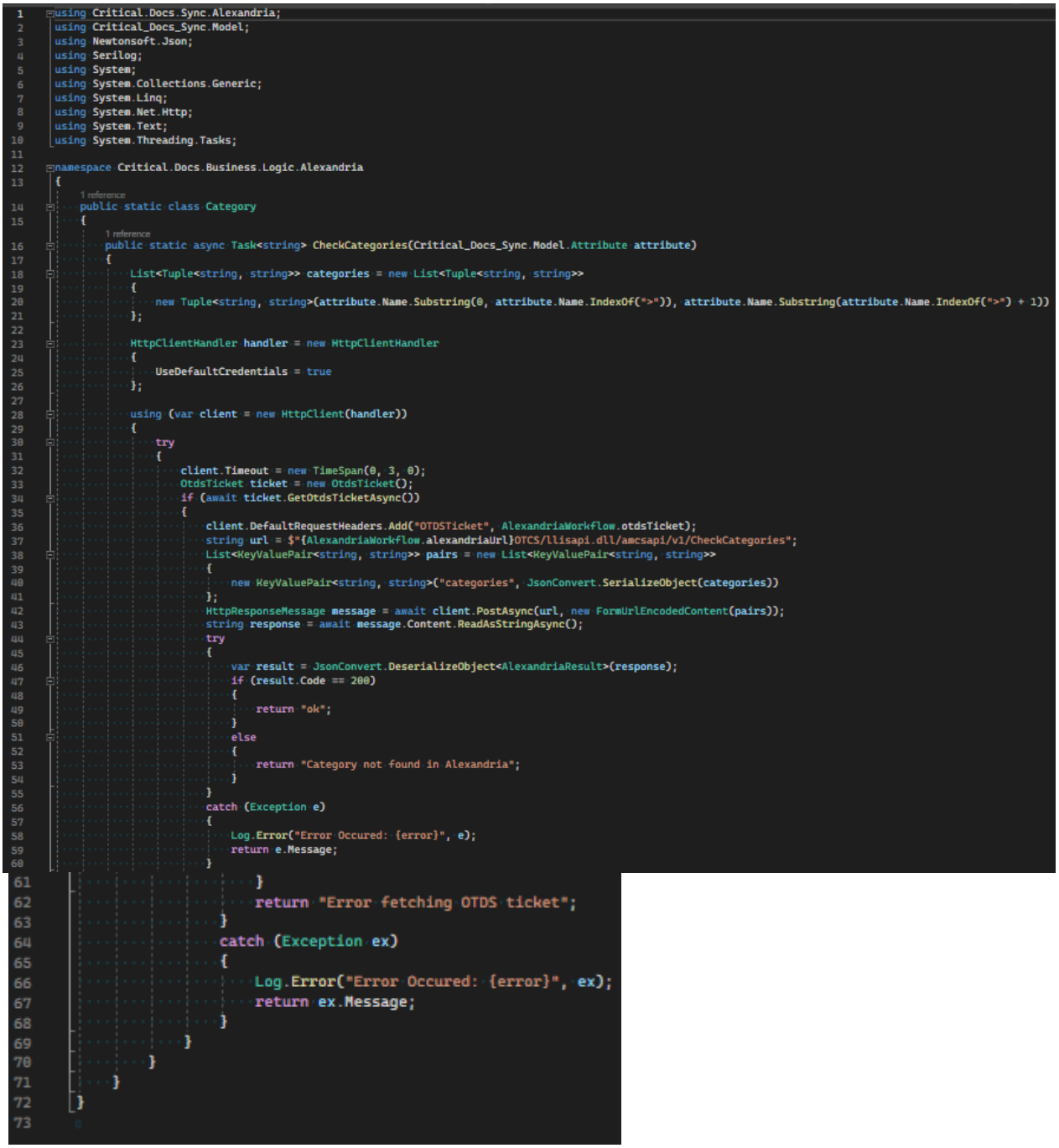


```
using Critical_Docs_Sync.Model;
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks;
using Serilog;
using Critical_Docs_Sync;
using Critical.Docs.Business.Logic.Alexandria;

namespace Critical.Docs.Sync.Alexandria
{
    18 references
    public class AlexandriaWorkflow
    {
        public static string otdsTicket;
        public static string alexandriaUrl;
        public static string otdsUrl;
        internal static string Id;

        0 references
        static AlexandriaWorkflow()
        {
            OptionsContext context = new OptionsContext();

            alexandriaUrl = context.GetOption("alexandriaURL").Value;
            otdsUrl = context.GetOption("otdsURL").Value;
        }

        0 references
        public async Task<bool> OtdsTicketAvailable()
        {
            OtdsTicket ticket = new OtdsTicket();
            return await ticket.GetOtdsTicketAsync();
        }

        1 reference
        public async Task<bool> IsAlexandriaAccessible()
        {
            AlexandriaHealth health = new AlexandriaHealth();
            return await health.IsAlive();
        }

        1 reference
        public static async Task<NodeResult> GetNodeItem(string Id)
        {
            return await Node.GetNode(Id);
        }

        1 reference
        public static async Task<DocumentListResult> GetListOfItems(AlexandriaSourceOptions option)
        {
            return await List.GetList(option);
        }

        1 reference
        public static async Task<string> CheckCategory(Critical_Docs_Sync.Model.Attribute attribute)
        {
            return await Category.CheckCategories(attribute);
        }
    }
}
```

S

Category

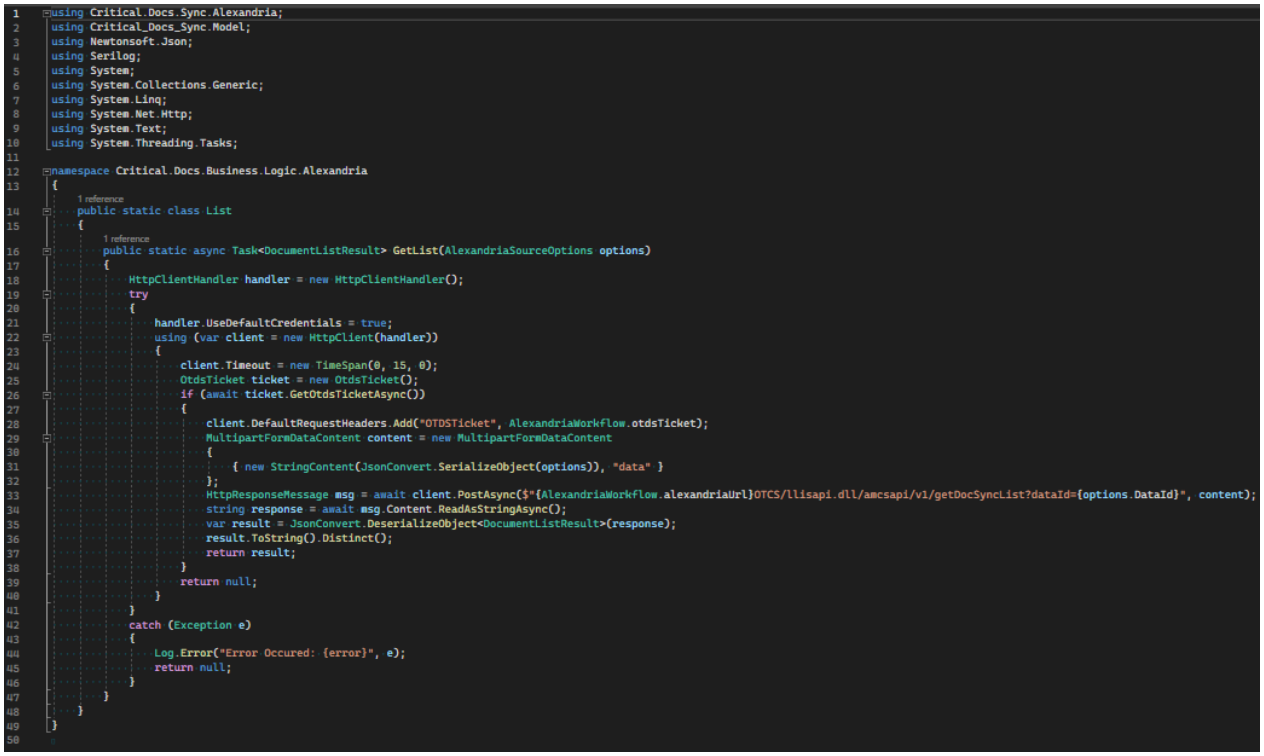
This class checks if the attribute exists in Alexandria.



```
using Critical.Docs.Sync.Alexandria;
using Critical_Docs_Sync.Model;
using Newtonsoft.Json;
using Serilog;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace Critical.Docs.Business.Logic.Alexandria
{
    1 reference
    public static class Category
    {
        1 reference
        public static async Task<string> CheckCategories(Critical_Docs_Sync.Model.Attribute attribute)
        {
            List<Tuple<string, string>> categories = new List<Tuple<string, string>>
            {
                new Tuple<string, string>(attribute.Name.Substring(0, attribute.Name.IndexOf(">")), attribute.Name.Substring(attribute.Name.IndexOf(">") + 1))
            };

            HttpClientHandler handler = new HttpClientHandler
            {
                UseDefaultCredentials = true
            };

            using (var client = new HttpClient(handler))
            {
                try
                {
                    client.Timeout = new TimeSpan(0, 3, 0);
                    OtdsTicket ticket = new OtdsTicket();
                    if (await ticket.GetOtdsTicketAsync())
                    {
                        client.DefaultRequestHeaders.Add("OTDSTicket", AlexandriaWorkflow.otdsTicket);
                        string url = $"{AlexandriaWorkflow.alexandriaUrl}OTCS/llisapi.dll/amcsapi/v1/CheckCategories";
                        List<KeyValuePair<string, string>> pairs = new List<KeyValuePair<string, string>>
                        {
                            new KeyValuePair<string, string>("categories", JsonConvert.SerializeObject(categories))
                        };
                        HttpResponseMessage message = await client.PostAsync(url, new FormUrlEncodedContent(pairs));
                        string response = await message.Content.ReadAsStringAsync();
                        try
                        {
                            var result = JsonConvert.DeserializeObject<AlexandriaResult>(response);
                            if (result.Code == 200)
                            {
                                return "ok";
                            }
                            else
                            {
                                return "Category not found in Alexandria";
                            }
                        }
                        catch (Exception e)
                        {
                            Log.Error("Error Occured: {error}", e);
                            return e.Message;
                        }
                    }
                    return "Error fetching OTDS ticket";
                }
                catch (Exception ex)
                {
                    Log.Error("Error Occured: {error}", ex);
                    return ex.Message;
                }
            }
        }
    }
}
```

List

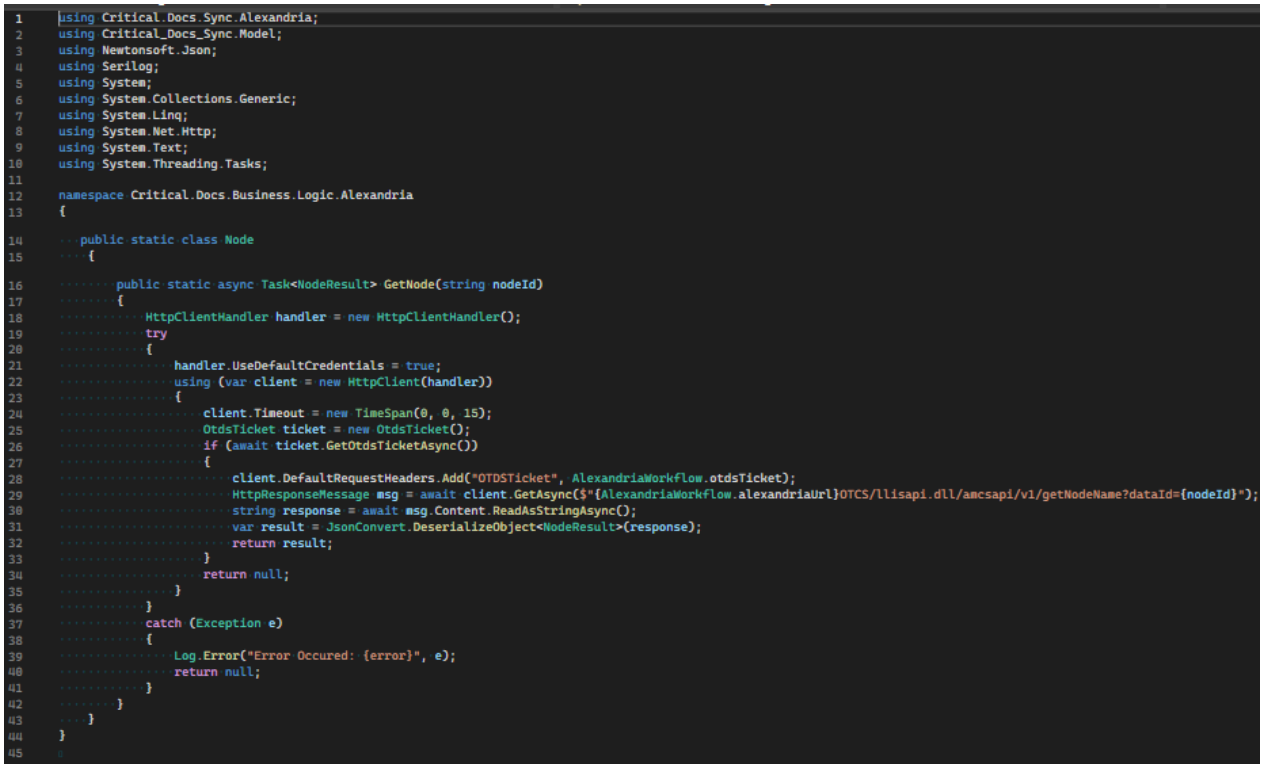
This class fetches the list of downloaded documents from Alexandria depending on the container.



```
using Critical.Docs.Sync.Alexandria;
using Critical_Docs_Sync.Model;
using Newtonsoft.Json;
using Serilog;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace Critical.Docs.Business.Logic.Alexandria
{
    1 reference
    public static class List
    {
        1 reference
        public static async Task<DocumentListResult> GetList(AlexandriaSourceOptions options)
        {
            HttpClientHandler handler = new HttpClientHandler();
            try
            {
                handler.UseDefaultCredentials = true;
                using (var client = new HttpClient(handler))
                {
                    client.Timeout = new TimeSpan(0, 15, 0);
                    OtdsTicket ticket = new OtdsTicket();
                    if (await ticket.GetOtdsTicketAsync())
                    {
                        client.DefaultRequestHeaders.Add("OTDSTicket", AlexandriaWorkflow.otdsTicket);
                        MultipartFormDataContent content = new MultipartFormDataContent
                        {
                            { new StringContent(JsonConvert.SerializeObject(options)), "data" }
                        };
                        HttpResponseMessage msg = await client.PostAsync($"{AlexandriaWorkflow.alexandriaUrl}OTCS/llisapi.dll/amcsapi/v1/getDocSyncList?dataId={options.DataId}", content);
                        string response = await msg.Content.ReadAsStringAsync();
                        var result = JsonConvert.DeserializeObject<DocumentListResult>(response);
                        result.ToString().Distinct();
                        return result;
                    }
                    return null;
                }
            }
            catch (Exception e)
            {
                Log.Error("Error Occured: {error}", e);
                return null;
            }
        }
    }
}
```

Node

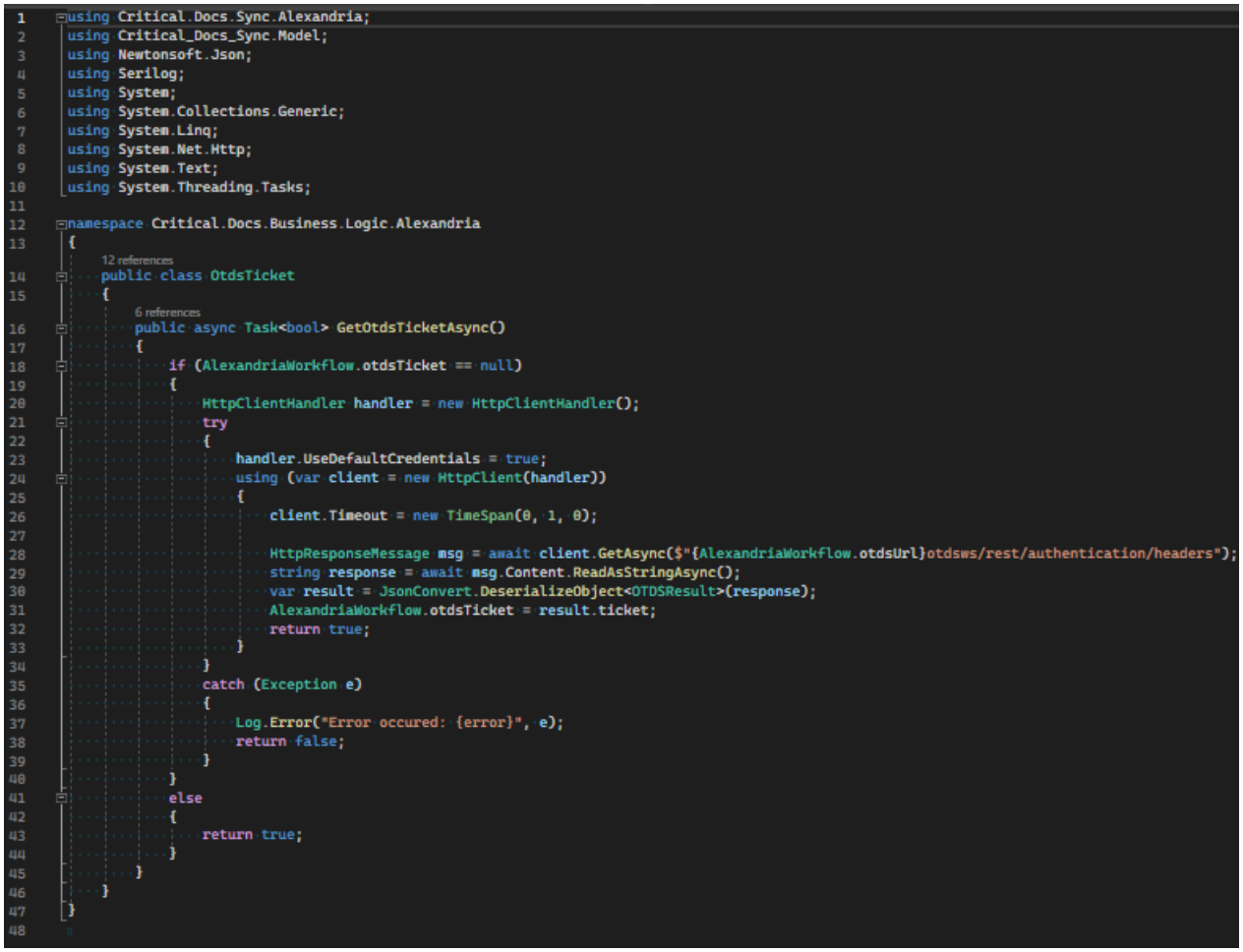
This class fetches the node (container) from Alexandria.



```
using Critical.Docs.Sync.Alexandria;
using Critical_Docs_Sync.Model;
using Newtonsoft.Json;
using Serilog;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace Critical.Docs.Business.Logic.Alexandria
{
    public static class Node
    {
        public static async Task<NodeResult> GetNode(string nodeId)
        {
            HttpClientHandler handler = new HttpClientHandler();
            try
            {
                handler.UseDefaultCredentials = true;
                using (var client = new HttpClient(handler))
                {
                    client.Timeout = new TimeSpan(0, 0, 15);
                    OtdsTicket ticket = new OtdsTicket();
                    if (await ticket.GetOtdsTicketAsync())
                    {
                        client.DefaultRequestHeaders.Add("OTDSTicket", AlexandriaWorkflow.otdsTicket);
                        HttpResponseMessage msg = await client.GetAsync($"{AlexandriaWorkflow.alexandriaUrl}OTCS/llisapi.dll/amcsapi/v1/getNodeName?dataId={nodeId}");
                        string response = await msg.Content.ReadAsStringAsync();
                        var result = JsonConvert.DeserializeObject<NodeResult>(response);
                        return result;
                    }
                    return null;
                }
            }
            catch (Exception e)
            {
                Log.Error("Error Occured: {error}", e);
                return null;
            }
        }
    }
}
```

OtdsTicket

This class fetches the otds-ticket to connect to Alexandria.



```
using Critical.Docs.Sync.Alexandria;
using Critical_Docs_Sync.Model;
using Newtonsoft.Json;
using Serilog;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace Critical.Docs.Business.Logic.Alexandria
{
    12 references
    public class OtdsTicket
    {
        6 references
        public async Task<bool> GetOtdsTicketAsync()
        {
            if (AlexandriaWorkflow.otdsTicket == null)
            {
                HttpClientHandler handler = new HttpClientHandler();
                try
                {
                    handler.UseDefaultCredentials = true;
                    using (var client = new HttpClient(handler))
                    {
                        client.Timeout = new TimeSpan(0, 1, 0);

                        HttpResponseMessage msg = await client.GetAsync($"{AlexandriaWorkflow.otdsUrl}otdsws/rest/authentication/headers");
                        string response = await msg.Content.ReadAsStringAsync();
                        var result = JsonConvert.DeserializeObject<OTDSResult>(response);
                        AlexandriaWorkflow.otdsTicket = result.ticket;
                        return true;
                    }
                }
                catch (Exception e)
                {
                    Log.Error("Error occured: {error}", e);
                    return false;
                }
            }
            else
            {
                return true;
            }
        }
    }
}
```

CompareDocuments

This class compares the documents between Alexandria and the database, the database and locally, locally and Alexandria.

```
using Critical_Docs_Sync.Model;
using Critical_Docs_Sync;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using Serilog;
using File = System.IO.File;
using System.Windows.Controls;
using System.Runtime.Remoting.Contexts;

namespace Critical.Docs.Business.Logic
{
    1 reference
    public static class CompareDocuments
    {
        2 references
        public static void Compare(DocumentList list, string source, List<DocumentList> flattened, List<DocumentHistory> fileHistory, string[] localFiles)
        {
            try
            {
                OptionsContext context = new OptionsContext();
                DirectoryInfo dir = new DirectoryInfo(source);
                var localDirectories = Directory.GetDirectories(source, "*", SearchOption.AllDirectories);

                foreach (var child in list.Children)
                {
                    foreach (var fileHist in fileHistory)
                    {
                        var alexandriaDocumentExistsInDatabase = flattened.FirstOrDefault(x => x.NodeId == fileHist.DocumentId);
                        if (alexandriaDocumentExistsInDatabase == null)
                        {
                            context.UpdateFileHistory(child);
                        }
                        else if (alexandriaDocumentExistsInDatabase != null && child.NodeName == fileHist.DocumentName)
                        {

                        }
                    }

                }

                foreach (var fileHist in fileHistory)
                {
                    foreach (var localFile in localFiles)
                    {
                        var local = new DirectoryInfo(localFile).Name;
                        var fileExistsInDatabase = flattened.FirstOrDefault(x => x.NodeId == fileHist.DocumentId);
                        if (fileExistsInDatabase == null)
                        {
                            if (local == fileHist.DocumentName)
                            {
                                context.DeleteFileHistory(fileHist.Id);
                                File.Delete(localFile);
                                Log.Information("File deleted: '" + localFile + "'");
                            }
                        }
                    }

                    foreach (var localDirectory in localDirectories)
                    {
                        var local = new DirectoryInfo(localDirectory).Name;
```

```
61                  var fileExistsInDatabase = flattened.FirstOrDefault(x => x.NodeId == fileHist.DocumentId);
62                  if (fileExistsInDatabase == null)
63                  {
64                      foreach (var child in list.Children)
65                      {
66                          if (local == fileHist.DocumentName && !Directory.Exists(source + "/" + local + "/"))
67                          {
68                              context.DeleteFileHistory(fileHist.Id);
69                              Directory.Delete(localDirectory, true);
70                              Log.Information("Folder deleted: '" + localDirectory + "'");
71                          }
72                      }
73
74                  }
75              }
76              // Hier morgen verder werken
77          }
78
79
80          foreach (var localDirectory in localDirectories)
81          {
82              foreach (var child in list.Children)
83              {
84                  var folder = new DirectoryInfo(localDirectory).Name;
85                  var folderExistsInDatabase = flattened.FirstOrDefault(y => y.NodeName == folder);
86                  if (folderExistsInDatabase == null && !Directory.Exists(source + "/" + folder + "/"))
87                  {
88                      Directory.Delete(source + "/" + folder + "/", true);
89                      Log.Information("Folder deleted: '" + folder + "'");
90                  }
91                  else if (folderExistsInDatabase != null && Directory.Exists(source + "/(" + child.NodeId + ") " + child.NodeName + "/"))
92                  {
93                      Directory.Delete(source + "/" + child.NodeName + "/", true);
94                      Log.Information("Folder deleted: " + child.NodeName);
95                  }
96              }
97              foreach (var localFile in localFiles)
98              {
99                  foreach (var child in list.Children)
100                 {
101                     var local = new FileInfo(localFile).Name;
102                     var folder = new DirectoryInfo(localDirectory).Name;
103                     var fileExistsInAlexandria = flattened.FirstOrDefault(y => y.NodeName == local);
104                     if (fileExistsInAlexandria == null && File.Exists(source + "/" + folder + "/" + local))
105                     {
106                         File.Delete(source + "/" + folder + "/" + local);
107                         Log.Information("File deleted: '" + local + "'");
108                         context.UpdateFileHistory(list);
109                     }
110                 }
111             }
112         }
113
114         foreach (var localFile in localFiles)
115         {
116             foreach (var child in list.Children)
117             {
118                 var local = new FileInfo(localFile).Name;
119                 var alexandriaFileExists = flattened.FirstOrDefault(y => y.NodeName ==  local);
120                 var fileExists = flattened.FirstOrDefault(x => "(" + x.NodeId + ") " + x.NodeName == local);
```

```
                    if (alexandriaFileExists == null && fileExists == null)
                    {
                        File.Delete(source + "/" + local);
                        Log.Information("File deleted: '" + local + "'");
                    }
                }
            }

            foreach (var child in list.Children)
            {
                if (child.NodeSubType == NodeSubTypes.Folder)
                {
                    Compare(child, source, flattened, fileHistory, localFiles);
                }
            }
        }
        catch (DirectoryNotFoundException dnfe)
        {

        }
        catch (Exception e)
        {
            DownloadDocument.countErrors++;
            DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";
            OptionsContext context = new OptionsContext();
            context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            DocumentProcessing._lastSuccess)
                    });
            Log.Error("Error Occured: {error}", e);
        }
    }
  }
}
```

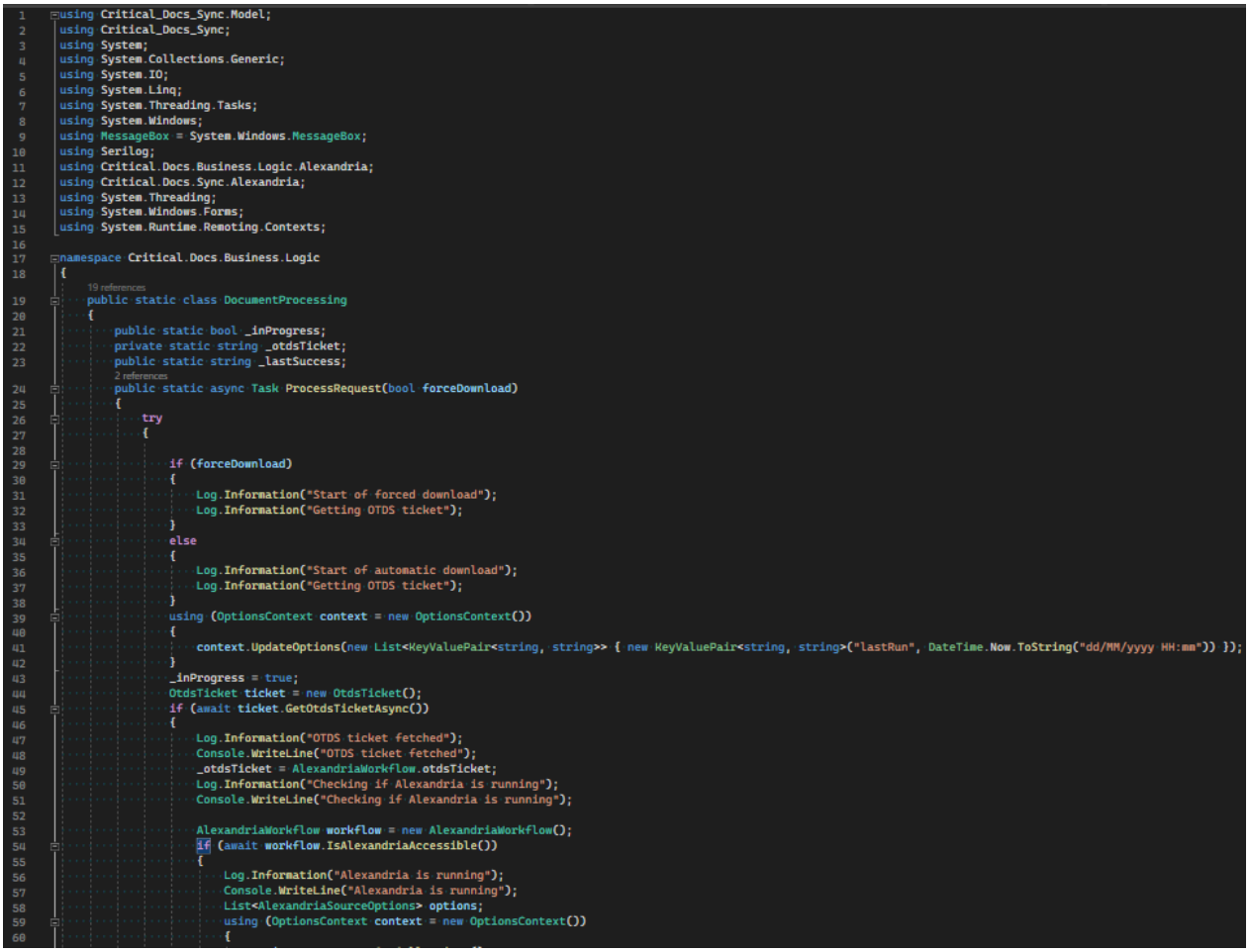
DocumentName

This class fetches the name of the document and removes all illegal characters (according to Windows) and removes too many spaces.

```
using System.IO;
using System.Text.RegularExpressions;

namespace Critical.Docs.Business.Logic
{
    5 references
    public class DocumentName
    {
        5 references
        public static string GetSafeFilename(string documentName)
        {
            documentName = string.Concat(documentName.Split(Path.GetInvalidFileNameChars()));
            documentName = Regex.Replace(documentName, "[ \f\t\v]+$", "");
            documentName = Regex.Replace(documentName, "[ ]{2,}", " ");
            return documentName;
        }
    }
}
```

DocumentProcessing

This class is the "Hub" regarding document downloading. First it checks if the OTDS ticket is reachable. Then it checks if Alexandria is reachable. If Alexandria is reachable, then the node is retrieved and downloaded. Then all the documents are compared.



```
using Critical_Docs_Sync.Model;
using Critical_Docs_Sync;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;
using MessageBox = System.Windows.MessageBox;
using Serilog;
using Critical.Docs.Business.Logic.Alexandria;
using Critical.Docs.Sync.Alexandria;
using System.Threading;
using System.Windows.Forms;
using System.Runtime.Remoting.Contexts;

namespace Critical.Docs.Business.Logic
{
    19 references
    public static class DocumentProcessing
    {
        public static bool _inProgress;
        private static string _otdsTicket;
        public static string _lastSuccess;
        2 references
        public static async Task ProcessRequest(bool forceDownload)
        {
            try
            {

                if (forceDownload)
                {
                    Log.Information("Start of forced download");
                    Log.Information("Getting OTDS ticket");
                }
                else
                {
                    Log.Information("Start of automatic download");
                    Log.Information("Getting OTDS ticket");
                }
                using (OptionsContext context = new OptionsContext())
                {
                    context.UpdateOptions(new List<KeyValuePair<string, string>> { new KeyValuePair<string, string>("lastRun", DateTime.Now.ToString("dd/MM/yyyy HH:mm")) });
                }
                _inProgress = true;
                OtdsTicket ticket = new OtdsTicket();
                if (await ticket.GetOtdsTicketAsync())
                {
                    Log.Information("OTDS ticket fetched");
                    Console.WriteLine("OTDS ticket fetched");
                    _otdsTicket = AlexandriaWorkflow.otdsTicket;
                    Log.Information("Checking if Alexandria is running");
                    Console.WriteLine("Checking if Alexandria is running");

                    AlexandriaWorkflow workflow = new AlexandriaWorkflow();
                    if (await workflow.IsAlexandriaAccessible())
                    {
                        Log.Information("Alexandria is running");
                        Console.WriteLine("Alexandria is running");
                        List<AlexandriaSourceOptions> options;
                        using (OptionsContext context = new OptionsContext())
                        {
```

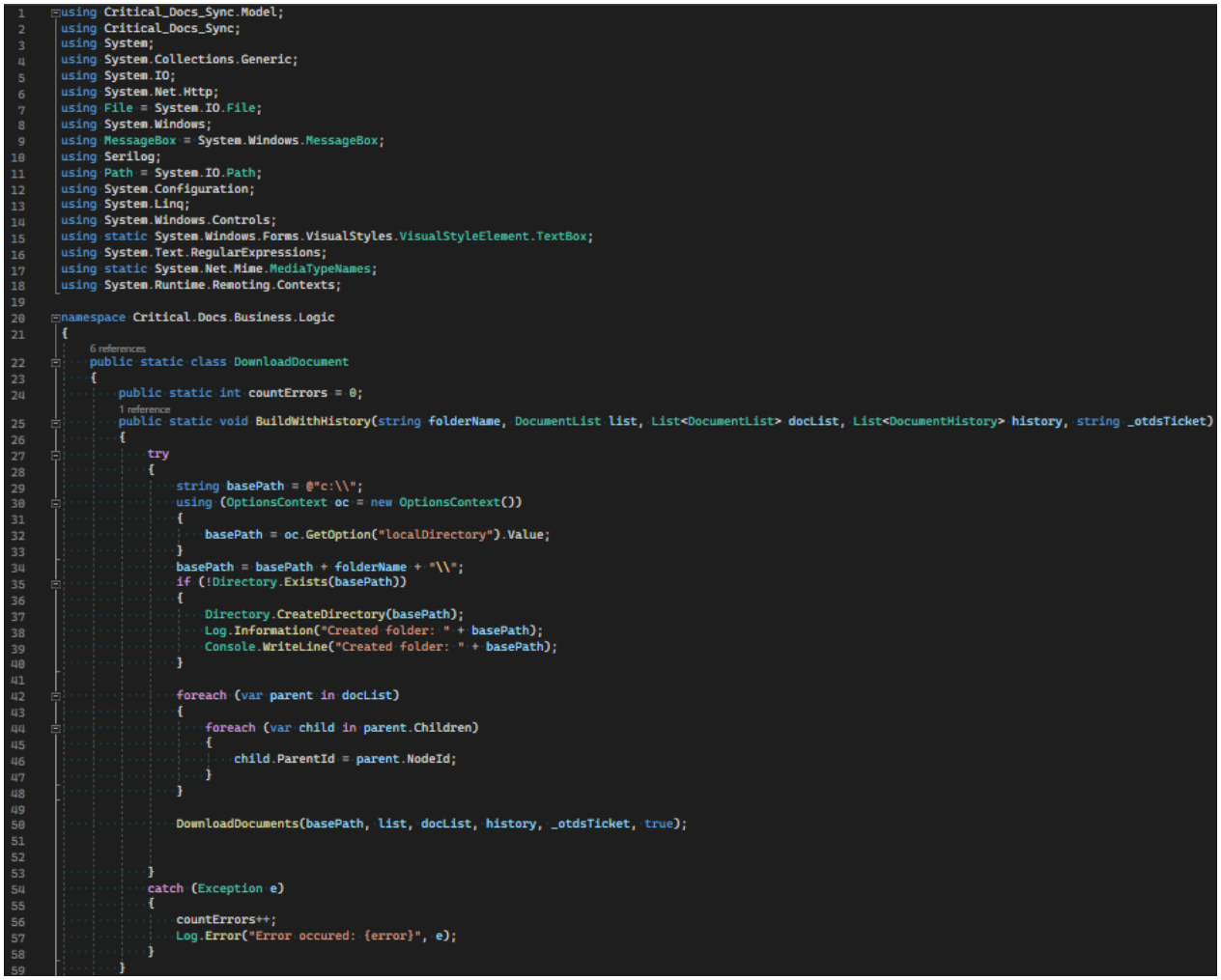
```
                    options = context.GetCollections();
                }
                foreach (AlexandriaSourceOptions item in options)
                {
                    //Get document list with last modified dates
                    List<DocumentHistory> history;
                    OptionsContext context = new OptionsContext();
                    history = context.GetFileHistory();
                    string sourceDirectory = context.GetOption("localDirectory").Value + item.Name;
                    Log.Information("Getting list for node: {nodeName}", item.Name);

                    DocumentListResult docListRes = await AlexandriaWorkflow.GetListOfItems(item);
                    var flattened = docListRes.DocList.Children.RecursiveSelector(x => x.Children).ToList();
                    Log.Information("Downloading documents");
                    var fileHistory = context.GetFileHistory();

                    DownloadDocument.BuildWithHistory(item.Name, docListRes.DocList, flattened,  history, _otdsTicket);
                    Log.Information("Comparing documents");
                    Console.WriteLine("Comparing documents");

                    var localFiles = Directory.GetFiles(sourceDirectory, "*", SearchOption.AllDirectories);
                    CompareDocuments.Compare(docListRes.DocList, sourceDirectory, flattened, fileHistory, localFiles);
                }
                Log.Information("Doc Sync Service Completed Successfully");
                Console.WriteLine("Doc Sync Service Completed Successfully");
                _lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished successfully";
                using (OptionsContext context = new OptionsContext())
                {
                    context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            _lastSuccess)
                    });
                }
            }
            else
            {
                DownloadDocument.countErrors++;
                Log.Error("The connection to Alexandria has been refused");
                MessageBox.Show("The connection to Alexandria has been refused.", "Alexandria Critical Documents|| exception:", MessageBoxButton.OK, MessageBoxImage.Information);
                _lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";
                using (OptionsContext context = new OptionsContext())
                {
                    context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            _lastSuccess)
                    });
                }
            }
        }
        else
        {

            DownloadDocument.countErrors++;
            Log.Error("Cannot connect to OTDS server");
            MessageBox.Show("Cannot connect to OTDS server.", "Alexandria Critical Documents|| exception:", MessageBoxButton.OK, MessageBoxImage.Information);
            _lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";
            using (OptionsContext context = new OptionsContext())
            {
                context.UpdateOptions(new List<KeyValuePair<string, string>>
```

```
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            _lastSuccess)
                    });
                }
            }
            if (forceDownload)
            {
                Log.Information("End of forced download \n");
            }
            else
            {
                Log.Information("End of automatic download");
            }
            _inProgress = false;
            _lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished successfully";
            using (OptionsContext context = new OptionsContext())
            {
                context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            _lastSuccess)
                    });
            }
        }
        catch (Exception e)
        {
            DownloadDocument.countErrors++;
            Log.Error("Error Occured: {error}", e);
            MessageBox.Show("The process has been cancelled.", "Alexandria Critical Documents|| exception:" + e.Message, MessageBoxButton.OK, MessageBoxImage.Information);
            _lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";
            using (OptionsContext context = new OptionsContext())
            {
                context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            _lastSuccess)
                    });
            }
        }
    }
}
```

DownloadDocument

This class is used to download the documents. First, we check if the container already exists locally. If it does not exist, then the container gets added. Otherwise, we check if the documents exist locally.

There are 2 different documents (locally): files and folders. First, we check if a file/folder with that name already exists. If it does not, then we download that document. If it does exist, then we skip over it. If there are 2 documents with the same name (when stripped off illegal characters), the documentId goes in front of the name. If a file has a different version or creation date (or both), then that file gets updated.



```
using Critical_Docs_Sync.Model;
using Critical_Docs_Sync;
using System;
using System.Collections.Generic;
using System.IO;
using System.Net.Http;
using File = System.IO.File;
using System.Windows;
using MessageBox = System.Windows.MessageBox;
using Serilog;
using Path = System.IO.Path;
using System.Configuration;
using System.Linq;
using System.Windows.Controls;
using static System.Windows.Forms.VisualStyles.VisualStyleElement.TextBox;
using System.Text.RegularExpressions;
using static System.Net.Mime.MediaTypeNames;
using System.Runtime.Remoting.Contexts;

namespace Critical.Docs.Business.Logic
{
    6 references
    public static class DownloadDocument
    {
        public static int countErrors = 0;
        1 reference
        public static void BuildWithHistory(string folderName, DocumentList list, List<DocumentList> docList, List<DocumentHistory> history, string _otdsTicket)
        {
            try
            {
                string basePath = @"c:\\";
                using (OptionsContext oc = new OptionsContext())
                {
                    basePath = oc.GetOption("localDirectory").Value;
                }
                basePath = basePath + folderName + "\\";
                if (!Directory.Exists(basePath))
                {
                    Directory.CreateDirectory(basePath);
                    Log.Information("Created folder: " + basePath);
                    Console.WriteLine("Created folder: " + basePath);
                }

                foreach (var parent in docList)
                {
                    foreach (var child in parent.Children)
                    {
                        child.ParentId = parent.NodeId;
                    }
                }

                DownloadDocuments(basePath, list, docList, history, _otdsTicket, true);


            }
            catch (Exception e)
            {
                countErrors++;
                Log.Error("Error occured: {error}", e);
            }
        }
```

```
2 references
public static void DownloadDocuments(string baseDirectory, DocumentList doc, List<DocumentList> docList, List<DocumentHistory> history, string _otdsTicket, bool initialLoad = false)
{
    bool retry = true;

    while (retry)
    {
        try
        {
            OptionsContext oc = new OptionsContext();
            string alxUrl =
                $"{oc.GetOption("alexandriaURL").Value}OTCS/llisapi.dll/amcsapi/v1/getDocument?dataId={doc.NodeId}&version={doc.Version}";
            var localDirectories = Directory.GetDirectories(baseDirectory, "*", SearchOption.AllDirectories);
            OptionsContext context = new OptionsContext();
            DocumentHistory docHis = history.Find(x => x.DocumentId == doc.NodeId && x.ParentId == doc.ParentId);


            doc.NodeName = DocumentName.GetSafeFilename(doc.NodeName);
            var localFiles = Directory.GetFiles(baseDirectory, "*", SearchOption.AllDirectories);

            if (doc.NodeSubType == NodeSubTypes.Folder || doc.NodeSubType == NodeSubTypes.CompoundDocument || doc.NodeSubType == NodeSubTypes.Collection || doc.NodeSubType == NodeSubTypes.Binder || doc.NodeSubType == NodeSubType
            {
                //is folder
                if (doc.NodeName.Length > 248)
                {
                    doc.NodeName = doc.NodeName.Remove(248);
                }

                string folderPath = baseDirectory + doc.NodeName + "/";

                if (folderPath.Length > 248)
                {
                    folderPath = folderPath.Remove(248);
                }

                if (!initialLoad)
                {
                    //Test Folder does not exist, need to fix that. It does create Test Folder with NodeId
                    var checkForFoldersWithSameNodeId = docList.Where(x => x.NodeId == doc.NodeId && x.NodeName == doc.NodeName && x.ParentId == doc.ParentId).Count() > 1;
                    var checkForDoubles = docList.Where(x => x.NodeName == doc.NodeName && x.ParentId == doc.ParentId).Count() > 1;
                    if (checkForDoubles && !Directory.Exists(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName + "/") && !checkForFoldersWithSameNodeId || doc.NodeName == "")
                    {
                        folderPath = baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName + "/";
                        Directory.CreateDirectory(folderPath);
                        Log.Information("Folder created: {file}", "(" + doc.NodeId + ") " + doc.NodeName);

                    }

                    else if (Directory.Exists(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName + "/") && Directory.Exists(baseDirectory + doc.NodeName + "/"))
                    {

                    }
                    else if (docHis != null)
                    {
                        if (docHis.DocumentName == "")
                        {
                            folderPath = baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName + "/";
                            Directory.CreateDirectory(folderPath);
                            Log.Information("Folder created: {file}", "(" + doc.NodeId + ") " + doc.NodeName);
                        }
                        else if (docHis.Version != doc.Version || docHis.Version != doc.Version && docHis.CreationDate != doc.CreationDate || docHis.DocumentId == doc.NodeId && docHis.DocumentName != doc.NodeName)
                        {
                            if (Directory.Exists(baseDirectory + docHis.DocumentName))
                            {
                                Directory.Move(baseDirectory + docHis.DocumentName, baseDirectory + doc.NodeName);
                                Log.Information("Folder: '" + docHis.DocumentName + "' changed to: '" + doc.NodeName + "'");
                            }

                        }
                        else
                        {
                            Directory.CreateDirectory(folderPath);
                            Log.Information("Folder created: {file}", doc.NodeName);
                        }

                    }

                    else
                    {
                        Directory.CreateDirectory(folderPath);
                        Log.Information("Folder created: {file}", doc.NodeName);
                    }

                    foreach (var item in docList)
                    {

                        item.NodeName = DocumentName.GetSafeFilename(item.NodeName);
                        var test = item;
                        if (Directory.Exists(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName + "/") && !Directory.Exists(baseDirectory + doc.NodeName + "/"))
                        {

                        }

                    }

                    context.CreateFileHistory(doc);
                    context.UpdateFileHistory(doc);
                }

                else
                {
                    folderPath = baseDirectory;
                    context.CreateFileHistory(doc);
                    context.UpdateFileHistory(doc);
                }

                foreach (var child in doc.Children)
                {
                    //child.ParentId = doc.NodeId;
                    DownloadDocuments(folderPath, child, docList, history, _otdsTicket);
                }
            }
            else
            {
                if (doc.NodeName.Length > 260)
                {
                    doc.NodeName = doc.NodeName.Remove(260);
                }
                string folderPath = baseDirectory + doc.NodeName;
```

```
            if (folderPath.Length > 260)
            {
                folderPath = folderPath.Remove(260);
            }
            //is no directory

            DocumentHistory histories = new DocumentHistory();
            List<DocumentList> documents = new List<DocumentList>();

            if (docHis == null)
            {


                switch (doc.NodeSubType)
                {
                    case NodeSubTypes.Shortcut:
                    case NodeSubTypes.Generation:
                    case NodeSubTypes.URL:
                    case NodeSubTypes.Document:
                    case NodeSubTypes.TextDocument:
                    case NodeSubTypes.Email:

                        using (var client = new HttpClient())
                        {
                            client.DefaultRequestHeaders.Add("OTDSTicket", _otdsTicket);
                            var newResponse = client.GetAsync(alxUrl).Result;

                            var content = newResponse.Content;

                            var checkForSameNodeId = docList.Where(x => x.NodeId == doc.NodeId && x.NodeName == doc.NodeName && x.ParentId == doc.ParentId).Count() > 1;
                            var checkForDoubles = docList.Where(x => x.NodeName == doc.NodeName && x.ParentId == doc.ParentId).Count() > 1;
                            if (checkForDoubles && !File.Exists(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName) && !checkForSameNodeId)
                            {
                                using (var file = File.Create(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName))
                                {
                                    context.CreateFileHistory(doc);
                                    var contentStream = content.ReadAsStreamAsync().Result;
                                    contentStream.CopyTo(file);
                                    file.Flush();
                                }
                                Log.Information("File created: {file}", "(" + doc.NodeId + ") " + doc.NodeName);
                            }
                            else
                            {
                                using (var file = File.Create(baseDirectory + doc.NodeName))
                                {
                                    context.CreateFileHistory(doc);
                                    var contentStream = content.ReadAsStreamAsync().Result;
                                    contentStream.CopyTo(file);
                                    file.Flush();
                                }
                                Log.Information("File created: {file}", doc.NodeName);
                            }



                            //Log.Information("File created: {file}", doc.NodeName);
                        }
                        break;
                    default:
                        break;
```

```
                }
            }
            else if (docHis.Version != doc.Version || docHis.Version != doc.Version && docHis.CreationDate != doc.CreationDate || docHis.DocumentId == doc.NodeId && docHis.DocumentName != doc.NodeName)
            {
                if (File.Exists(baseDirectory + docHis.DocumentName))
                {
                    File.Move(baseDirectory + docHis.DocumentName, baseDirectory + doc.NodeName);
                    Log.Information("File: '" + docHis.DocumentName + "' changed to: '" + doc.NodeName + "'");
                }

                //docHis.DocumentName = doc.NodeName;
                context.UpdateFileHistory(doc);
            }
            else if (!File.Exists(baseDirectory + doc.NodeName))
            {
                switch (doc.NodeSubType)
                {
                    case NodeSubTypes.Shortcut:
                    case NodeSubTypes.Generation:
                    case NodeSubTypes.URL:
                    case NodeSubTypes.Document:
                    case NodeSubTypes.TextDocument:
                    case NodeSubTypes.Email:

                        using (var client = new HttpClient())
                        {
                            client.DefaultRequestHeaders.Add("OTDSTicket", _otdsTicket);
                            var newResponse = client.GetAsync(alxUrl).Result;

                            var content = newResponse.Content;

                            var checkForSameNodeId = docList.Where(x => x.NodeId == doc.NodeId && x.NodeName == doc.NodeName && x.ParentId == doc.ParentId).Count() > 1;
                            var checkForDoubles = docList.Where(x => x.NodeName == doc.NodeName && x.ParentId == doc.ParentId).Count() > 1;
                            if (checkForDoubles && !File.Exists(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName) && !checkForSameNodeId)
                            {
                                using (var file = File.Create(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName))
                                {
                                    context.CreateFileHistory(doc);
                                    var contentStream = content.ReadAsStreamAsync().Result;
                                    contentStream.CopyTo(file);
                                    file.Flush();
                                }
                                Log.Information("File created: {file}", "(" + doc.NodeId + ") " + doc.NodeName);
                            }
                            else if (File.Exists(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName))
                            {

                            }
                            else
                            {
                                using (var file = File.Create(baseDirectory + doc.NodeName))
                                {
                                    context.CreateFileHistory(doc);
                                    var contentStream = content.ReadAsStreamAsync().Result;
                                    contentStream.CopyTo(file);
                                    file.Flush();
                                }
                                Log.Information("File created: {file}", doc.NodeName);
                            }

                            //Log.Information("File created: {file}" + doc.NodeName);
```

```
                                }
                                break;
                            default:
                                break;
                        }
                    }


                    //else if (!File.Exists(baseDirectory + "(" + doc.NodeId + ") " + doc.NodeName) && docHis.DocumentName != doc.NodeName)
                    //{
                    //    switch (doc.NodeSubType)
                    //    {
                    //        case NodeSubTypes.Shortcut:
                    //        case NodeSubTypes.Generation:
                    //        case NodeSubTypes.URL:
                    //        case NodeSubTypes.Document:
                    //        case NodeSubTypes.TextDocument:
                    //        case NodeSubTypes.Email:
                    //            OptionsContext oc = new OptionsContext();
                    //            string alxUrl =
                    //                $"{oc.GetOption("alexandriaURL").Value}OTCS/llisapi.dll/amcsapi/v1/getDocument?dataId={doc.NodeId}&version={doc.Version}";

                    //            using (var client = new HttpClient())
                    //            {
                    //                client.DefaultRequestHeaders.Add("OTDSTicket", _otdsTicket);
                    //                var newResponse = client.GetAsync(alxUrl).Result;

                    //                var content = newResponse.Content;
                    //                foreach (var localFile in localFiles)
                    //                {
                    //                    var local = new DirectoryInfo(localFile).Name;
                    //                    var folder = history.Find(x => x.DocumentName == local);

                    //                    if (local == doc.NodeName)
                    //                    {
                    //                        doc.NodeName = "(" + doc.NodeId + ") " + doc.NodeName;
                    //                    }
                    //                }

                    //                using (var file = File.Create(baseDirectory + doc.NodeName))
                    //                {
                    //                    context.CreateFileHistory(doc);
                    //                    var contentStream = content.ReadAsStreamAsync().Result;
                    //                    contentStream.CopyTo(file);
                    //                    file.Flush();
                    //                }
                    //                Log.Information("File created: " + doc.NodeName);
                    //            }
                    //            break;
                    //        default:
                    //            break;
                    //    }
                    //}
                }
                retry = false;
            }
            catch (Exception e)
            {
                DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";
                OptionsContext options = new OptionsContext();
```

```
                options.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            DocumentProcessing._lastSuccess)
                    });
                Log.Error("Error Occured: {error}", e);

                MessageBoxResult result = MessageBox.Show("Something went wrong.\n To retry please close all folders and documents.", "Alexandria Critical Documents", MessageBoxButton.OKCancel, MessageBoxImage.Error);

                // Process message box results
                switch (result)
                {
                    case MessageBoxResult.OK:
                        retry = true;
                        break;
                    case MessageBoxResult.Cancel:
                        DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " finished with errors";
                        using (OptionsContext context = new OptionsContext())
                        {
                            context.UpdateOptions(new List<KeyValuePair<string, string>>
                            {
                            new KeyValuePair<string, string>("lastSuccessRun",
                                DocumentProcessing._lastSuccess)
                            });
                        }
                        throw;
                }
            }
        }
    }
}
```

Bugfixes and Quality of Life

The project has undergone some bug fixes. In this section I am going to discuss them and show what I have added/changed. I have improved error handling by using Serilog to create a file with bugs and non-bugs. If there are bugs in it, those bugs are sent to a person via email. I fixed the bug "crash with unclear error message when syncing shortcut without permissions on original document" by not creating a shortcut.

Documentnme with illegal characters – original vs changed

The original method took some characters out of the document name, but these are not illegal characters in Windows. I solved it by taking those characters out, then checking to see if there is a whitespace and if there are 2 or more spaces. I solved the space problem by making it one space and removing the whitespace.

```
public string GetSafeFilename(string filename)
{
    filename = Regex.Replace(filename, "[^\\w\\._ ()-]", "");

    return filename;

}
```

```
public static string GetSafeFilename(string documentName)
{
    documentName = string.Concat(documentName.Split(Path.GetInvalidFileNameChars()));
    documentName = Regex.Replace(documentName, "[ \f\t\v]+$", "");
    documentName = Regex.Replace(documentName, "[ ]{2,}", " ");
    return documentName;
}
```

Default yes for ‘mandatory’ attributes – original vs changed

In the original, mandatory does not default to yes when you see addAttribute. I changed this by setting addAtt.chkBoxMandatory.IsChecked to true. Then I gave clarification for the user to the view by changing the title so the user knows if that view changes or adds attributes.

```
private void btnEditAttribute_Click(object sender, RoutedEventArgs e)
{
    var parents = ((Button)sender).GetParents();
    foreach (var item in parents)
    {
        if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
        {
            Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;


            Attribute att = (Attribute)gvr.Item;
            AddAttribute addAtt = new AddAttribute(att);
            if (addAtt.ShowDialog() == true)
            {
                att = addAtt.Attribute;
            }

            RadGridViewDocuments.Rebind();
        }
    }
}
private void btnAddAttribute_Click(object sender, RoutedEventArgs e)
{
    AddAttribute addAtt = new AddAttribute(new Attribute());
    if (addAtt.ShowDialog() == true)
    {
        Attribute att = addAtt.Attribute;

        var parents = ((Button)sender).GetParents();
        foreach (var item in parents)
        {
            if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
            {
                Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;


                AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
                opt.Attributes.Add(att);

                RadGridViewDocuments.Rebind();
            }
        }
    }
}
```

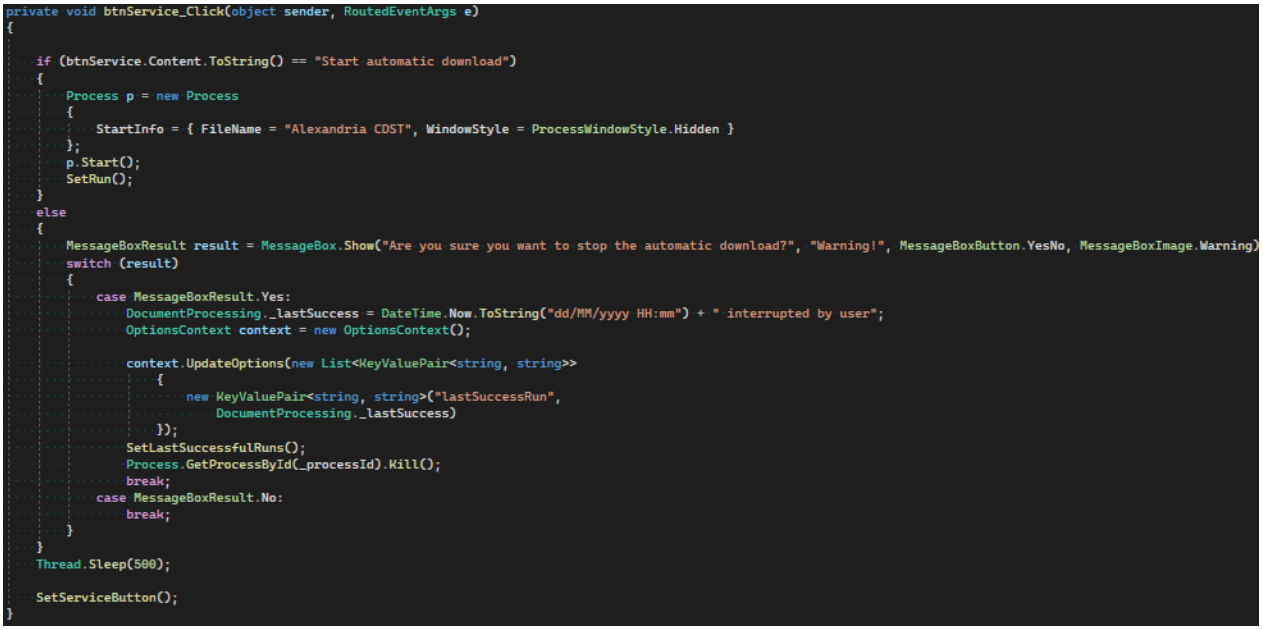
```
private void btnAddAttribute_Click(object sender, RoutedEventArgs e)
{
    AddAttribute addAtt = new AddAttribute(new Attribute())
    {
        Title = "Alexandria Critical Documents - Add Attribute"
    };
    addAtt.chkBoxMandatory.IsChecked = true;
    if (addAtt.ShowDialog() == true)
    {
        Attribute att = addAtt.Attribute;
        addAtt.TxtBlckAttribute.Text = "Add a new attribute to collection";
        var parents = ((Button)sender).GetParents();
        foreach (var item in parents)
        {
            if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
            {
                Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

                AlexandriaSourceOptions opt = (AlexandriaSourceOptions)gvr.Item;
                opt.Attributes.Add(att);

                RadGridViewDocuments.Rebind();
            }
        }
    }
}
```

```
private void btnEditAttribute_Click(object sender, RoutedEventArgs e)
{
    var parents = ((Button)sender).GetParents();
    foreach (var item in parents)
    {
        if (item.GetType() == typeof(Telerik.Windows.Controls.GridView.GridViewRow))
        {
            Telerik.Windows.Controls.GridView.GridViewRow gvr = (Telerik.Windows.Controls.GridView.GridViewRow)item;

            Attribute att = (Attribute)gvr.Item;
            AddAttribute addAtt = new AddAttribute(att)
            {
                Title = "Alexandria Critical Documentsro - Edit attribute",
            };

            addAtt.TxtBlckAttribute.Text = "Edit attribute: ";
            addAtt.TxtBlckAttribute.Inlines.Add(new Bold(new Run(att.Value)));

            if (addAtt.ShowDialog() == true)
            {
                att = addAtt.Attribute;
            }

            RadGridViewDocuments.Rebind();
        }
    }
}
```

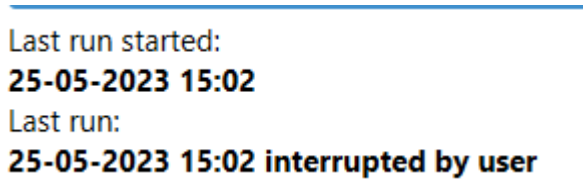
Make stopping automatic sync a little more difficult - original vs changed

The original stops the process immediately when the user clicks on btnService (only if the content is "Stop automatic download"). I solved this by using a message box to ask the user if they want to stop it. If they want to stop it, the process stops, and the user gets the information "*date* interrupted by user.

```
private void btnService_Click(object sender, RoutedEventArgs e)
{
    if (btnService.Content.ToString() == "Start automatic download")
    {
        Process p = new Process
        {
            StartInfo = { FileName = "Alexandria CDST", WindowStyle = ProcessWindowStyle.Hidden }
        };
        p.Start();

    }
    else
    {
        Process.GetProcessById(_processId).Kill();
    }
    Thread.Sleep(500);
    SetServiceButton();
}
```



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

    if (btnService.Content.ToString() == "Start automatic download")
    {
        Process p = new Process
        {
            StartInfo = { FileName = "Alexandria CDST", WindowStyle = ProcessWindowStyle.Hidden }
        };
        p.Start();
        SetRun();
    }
    else
    {
        MessageBoxResult result = MessageBox.Show("Are you sure you want to stop the automatic download?", "Warning!", MessageBoxButton.YesNo, MessageBoxImage.Warning);
        switch (result)
        {
            case MessageBoxResult.Yes:
                DocumentProcessing._lastSuccess = DateTime.Now.ToString("dd/MM/yyyy HH:mm") + " interrupted by user";
                OptionsContext context = new OptionsContext();

                context.UpdateOptions(new List<KeyValuePair<string, string>>
                    {
                        new KeyValuePair<string, string>("lastSuccessRun",
                            DocumentProcessing._lastSuccess)
                    });
                SetLastSuccessfulRuns();
                Process.GetProcessById(_processId).Kill();
                break;
            case MessageBoxResult.No:
                break;
        }
    }
    Thread.Sleep(500);

    SetServiceButton();
}
```



Different possible ends of download

There are some options for ending downloads. I have added these for clarity to the user:

- Finished successfully: this is when the documents finish downloading correctly.
- Interrupted by user: this is when the user stops the automatic download.
- Finished with errors: this is when 1 or more errors occur during the download.

Last run started:
25-05-2023 15:06
Last run:
25-05-2023 15:06 finished successfully

Last run started:
25-05-2023 15:07
Last run:
25-05-2023 15:07 interrupted by user

Last run started:
25-05-2023 15:06
Last run:
25-05-2023 15:06 finished with errors

Only numbers in SyncTime

One of the QoL features I added is a problem that can occur when the user enters characters other than numbers in the SyncTime textbox. I solved this by adding a "PreviewTextInput" named "NumberValidationTextBox. Then I created a method with that name. In this method, I used regex to use only numbers in the textbox.

```
<TextBox Grid.Row="1" Grid.Column="1" Margin="0 0 0 10" Name="txtBoxSyncTime" PreviewTextInput="NumberValidationTextBox"></TextBox>
```

```
private void NumberValidationTextBox(object sender, TextCompositionEventArgs e)
{
    Regex regex = new Regex("[^0-9]+");
    e.Handled = regex.IsMatch(e.Text);
}
```

Sync interval period 24