Intro to scripting in Fast video cataloger

Introduction

Scripting is one of the most powerful features of Fast video cataloger (FVC); it lets you automate tasks, extend the program and even build complete solutions on top of the software.

You write FVC scripts in C#.

This document is not an introduction to C# programming. There are plenty of C# tutorials online and books if you are serious about learning C#.

However, this document will help you get started with scripting in FVC, even if you are not a programmer.

The latest version of FVC can be downloaded from https://videocataloger.com/download/ and includes samples and API documentation.

Now let's get started.

Structure of a script

Open the console window.

The upper part of the console script window contains a script. The lower part of the windows leaves space for output from scripts.

The console window has a load button to load script files. For now, we will write the script straight into the window.

There is a help button to open the script API documentation. A text field to give input to the script and a run button to run the script.

By default, the script window has a simple sample script loaded.

Let's start with this sample script and remove everything between the {} right at the comment *"Enter script here…"*.

After your edits, the script window should have the following text:

```
using System.Runtime;
using System.Collections.Generic;
using VideoCataloger;
using VideoCataloger.RemoteCatalogService;

class Script
{
  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )
  { // Enter script here...
  }
}
```

The above is a bare minimal FVC C# script that runs without errors but does nothing.

The "using" lines at the top tell the script what namespaces you want access to.

Next, the script needs to be inside of a class like this:

```
class Script
{
}
```

Inside the class definition, the script always needs to have a static function named Run that looks like the sample below. The Run function is the entry point of the script and where execution always starts when you click the run button:

```
  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )
```

In the arguments to the script, **IScripting** is the root interface. The IScripting interface is how you communicate with the Fast video cataloger program from your script.

The second argument is a string containing whatever is in the text box in the console window.

These components are always the same for all scripts.

Hello World, the first script

Now let's make a hello world example as a script in Fast video cataloger.

We write the code inside the console window.

First, we need to get hold of the interface to the console window,

In the scripting documentation (open with the help button), you can click on the "Video Cataloger" page to get all interfaces available.

Using the passed-in **IScripting** interface, we want to get the **IConsole** interface. We can do this with the following line:

```
IConsole console = scripting.GetConsole();
```

The console variable is of the IConsole type, i.e., you can call all IConsole functions from the console variable. = is to assign the value, and to the right of the =, we call GetConsole from the scripting interface. GetConsole returns an object that implements the interface. And, as always in C#, we need to end the statement with a semicolon character.

Again, looking in the documentation for **IConsole,** we see it has a member function **WriteLine**.

To write text, we call the function like this:

```
console->WriteLine("Hello World");
```

and the complete script is now:

```
using System.Runtime;

using System.Collections.Generic;

using VideoCataloger;

using VideoCataloger.RemoteCatalogService;

class Script

{

  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )

  {

     IConsole console = scripting.GetConsole();

     console->WriteLine("Hello World");

  }

}
```

Let's instead put the text in a popup window.

The scripting environment has access to the *.net* framework. The .net framework has tons of functions for string manipulations, OS access, and anything else you might need for desktop application development.

You do not need to do anything special to bring in the .net framework. We will use the Message box that is in the System.Windows namespace. At the following code to your run function

```
System.Windows.MessageBox.Show("Hello World");
```

And click the Run button to run the script. This will show a popup window with the *Hello world* text.

Now finally, let's use the argument that gets passed into the run function **string arguments**. To combine two strings in C#, we can use the + operator. Write the following:

```
System.Windows.MessageBox.Show("Hello" + arguments);
```

Write your name in the text box next to Run and click the run button. You should now see a text box showing "Hello [Your name]".

Getting user input

Next, we will look at how we can get the current selection state into our script. Commonly, we want to create a script that inputs what the user has selected for videos, thumbnails, etc.

We get a selection from the ISelection interface. As with all FVC interfaces, we get this from the IScripting interface.

```
ISelection selection = scripting.GetSelection();
```

If we look at the documentation for ISelection we can see it has many Get and Set functions. Use the **get** functions to get the current selection from the video, actor, bins, playlist, and thumbnails window. Use the **Set** function to change the current selection.

GetSelectedVideos will return the currently selected videos. We will get a list of all the selected videos.

```
List<long> selected_videos = selection.GetSelectedVideos();
```

Each of the selected videos is represented with a unique ID.

In FVC, you can see the number in the ID column. The concept is the same for the selection of thumbnails, actors, and playlists.

Reading Video properties

As mentioned before, each video in FVC has a unique ID. From this id, we can access and change data about the video.

Now we need to access the actual video catalog interface. As before, we can get this interface from the scripting interface, and the interface we want is `IVideoCatalogService`.

```
IVideoCatalogService catalog = scripting.GetVideoCatalogService();
```

The *IVideoCatalogService* interface has a lot of functions. It has its own section in the documentation. I highly recommend you browse it to get an idea of the functionality available.

This interface is the same as is used by the Fast video cataloger. The FVC software user interface is a separate executable that communicates with the video database through this interface. The actual catalog is running in a separate process. If you are running with a server, it might even run on a different computer.

Now that we have the catalog interface let us see what we can find out about a video. We need to use the `GetVideoFileEntry()` function.

```
    VideoFileEntry entry = catalog.GetVideoFileEntry(1);
```

The *1* argument is the videoid, and we get back an entry class with info about the video. If the video id does not exist in the catalog, you will get back a null value.

Look in the documentation for the **VideoFileEntry** class for all the information available about the video through this interface. One of the properties is the FilePath, i.e., the path to the video file.

Let's try to print the path to the video in the console window. From the previous section, we know how to get the console interface where we can print text.

```
scripting.GetConsole().WriteLine( entry.FilePath );
```

We also had a list of selected video ids.

To loop over a list in C#, we can use a *for each* loop. For each will execute the content inside the loop once for each entry in the list

```
foreach (long VideoID in selected_videos)
{

// Executed once for each entry in the selected_videos list

}
```

Inside the loop, we have a *VideoID*. Any line that starts with // is a comment, and all text between the // and the end of the line is skipped.

So, let's put the lines from above inside the loop, and we will print the path of each selected video.

```
   foreach (long VideoID in selected_videos)
    {
      VideoFileEntry entry = catalog.GetVideoFileEntry( VideoID );
      scripting.GetConsole().WriteLine( entry.FilePath );
    }
```

Special folders

Fast video cataloger has support for something we call special folders. These are path mappings where you can specify a variable as a root to a path-dependent on the system where the catalog is running. You can specify these special folders in the preference of the program. One of these special folders is *[LOCAL]*, local gets used when you add a video below the catalog file's folder. Special folders make it possible to move around a folder with a catalog and videos in subfolders without breaking paths. So, what does this have to do with scripting? In the previous example, your printed paths perhaps looked something like this:

[LOCAL]\my-videos\test.mp4

If we want to use this as an absolute path, we need to convert it. Functions to do that are available in the *Utilities* interface. This interface is how you convert this to an absolute path.

```
   IUtilities utils = scripting.GetUtilities();

   string path = utils.ConvertToLocalPath(entry.FilePath);
```

ConvertToLocal will take a path and convert it to a local system path. If the path is already absolute, it will simply return the exact string that was input.

If we combine all we have gone through, here is the complete program to print the path of the currently selected videos

```
using System.Runtime;
using System.Collections.Generic;
using VideoCataloger;
using VideoCataloger.RemoteCatalogService;

class Script
{
  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )
  {

     ISelection selection = scripting.GetSelection();
    List<long> selected_videos = selection.GetSelectedVideos();
    IVideoCatalogService catalog = scripting.GetVideoCatalogService();
    IUtilities utils = scripting.GetUtilities();

    foreach (long VideoID in selected_videos)
    {
      VideoFileEntry entry = catalog.GetVideoFileEntry( VideoID );
      string path = utils.ConvertToLocalPath(entry.FilePath);
      scripting.GetConsole().WriteLine( path );
    }
  }
}
```

Updating Video Properties

In the previous section, we read one of the video properties, path. A video has several basic properties like *Title, Link, Description, FilePath, and Rating*. You can view these properties at the top of the video details window.

When writing video properties, you need to be very careful as the changes you make are permanent. Always make sure to have a backup of your video catalog, especially when creating new scripts that alter the catalog.

When a video is indexed, it can read metadata for titles (video indexer tab in preferences). If you read a video title from metadata and that metadata is nonsense, it is possible to reset this to be the video file's name with the help of a script. Let's continue to build on our previous example and make a script that takes the video path, gets the filename from that path, and sets that as the video's title.

First, we need a bit of string manipulation to get the filename from the path.

LastIndexOf searches a string from the end for a given character and returns the 0 based index of the found character (or -1 if it was not present in the string).

```
int filename_start = path.LastIndexOf("\\");
```

We search from the end to find the \ character. In C#, the \ is special and denotes the beginning of a special character sequence. That is why we need to write two \. Once to start a special sequence and second to get the character.

Next, we grab a substring of the path starting at one character after the value of the filename_start variable.

```
string filename = path.Substring( filename_start+1 );
```

To set the title of a video we use this function on the scripting interface

```
void SetVideoProperty(
        long video_file_id,
        string property_name,
        string value
)
```

The first argument is the id of the video, the second is a name string of the property we want to change, and the third is the property's new value.

```
   catalog.SetVideoProperty( VideoID, "Title", filename );
```

If we combine it with the previous example, the whole program should now look like this:

```
using System.Runtime;
using System.Collections.Generic;
using VideoCataloger;
using VideoCataloger.RemoteCatalogService;

class Script
{
  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )
  {
      ISelection selection = scripting.GetSelection();
    List<long> selected_videos = selection.GetSelectedVideos();
    IVideoCatalogService catalog = scripting.GetVideoCatalogService();
    IUtilities utils = scripting.GetUtilities();

    foreach (long VideoID in selected_videos)
    {
      VideoFileEntry entry = catalog.GetVideoFileEntry( VideoID );
      string path = utils.ConvertToLocalPath(entry.FilePath);
      int filename_start = path.LastIndexOf("\\");
      string filename = path.Substring( filename_start+1 );
      catalog.SetVideoProperty( VideoID, "Title", filename );
      scripting.GetConsole().WriteLine( filename  );
    }
  }
}
```

Running the script will update the title of the selected video to be the filename. But you will not see the change unless you search or deselect and then select the video again. Since we access the catalog directly, the user interface is not aware of the update.

We can tell the user interface to refresh. We get the user interface API from *GetGUI()* in scripting and use the Refresh function like this,

```
scripting.GetGUI().Refresh("");
```

Put the refresh call at the end of the script.

Extended properties

The properties we discussed in the previous section, like path and title, are available for all videos. FVC also lets you define your video properties, and we call these extended properties.

Extended properties are used to store extracted metadata, including XMP data. The extended properties are stored as separate entries that you need to read explicitly. In FVC, you can view extended properties in a list at the bottom of the video detail window.

`ExtendedProperty[] GetVideoFileExtendedProperty( int video_file_id )`

Give you all extended properties for a given video. [] is how you define an array in C#, so we get back a zero-based array of extended properties.

An extended property class is a key value store with a Property and a Value. Here is how you would get an extended property from a video:

```
ExtendedProperty[] props = catalog.GetVideoFileExtendedProperty((int)
VideoID);
```

Note that the interface wants an int VideoID. VideoIDs are always zero-based positive numbers, but some APIs use a negative value to indicate an invalid ID and need a signed number. Sometimes you need a VideoID as a signed integer and sometimes as an unsigned. You can convert between them with the cast operators, (int) will cast to a signed integer id.

Now to print all extended properties for a video we can again use a foreach loop like this:

```
ExtendedProperty[] props = catalog.GetVideoFileExtendedProperty( (int)
VideoID );
      foreach (ExtendedProperty p in props)
      {
        scripting.GetConsole().WriteLine( p.Property + " - " + p.Value  );
      }
```

And here is the complete program to print all extended properties for the currently selected videos:

```
using System.Runtime;
using System.Collections.Generic;
using VideoCataloger;
using VideoCataloger.RemoteCatalogService;

class Script
{
  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )
  {
    ISelection selection = scripting.GetSelection();
    List<long> selected_videos = selection.GetSelectedVideos();
    IVideoCatalogService catalog = scripting.GetVideoCatalogService();
    IUtilities utils = scripting.GetUtilities();

    foreach (long VideoID in selected_videos)
    {
      VideoFileEntry entry = catalog.GetVideoFileEntry( VideoID );
      string path = utils.ConvertToLocalPath(entry.FilePath);
      scripting.GetConsole().WriteLine( path  );

      ExtendedProperty[] props =
catalog.GetVideoFileExtendedProperty((int) VideoID);
      foreach (ExtendedProperty p in props)
      {
        scripting.GetConsole().WriteLine( p.Property + " - " + p.Value  );
      }
    }
  }
}
```

If you want to set a property use **SetVideoFileExtendedProperty**, with the printed property above.

```
void SetVideoFileExtendedProperty(
        int video_file_id,
        string property,
        string value
)
```

Downloading files

If you want to open a web page in the web browser, we first need to get the browser interface. As with the other Interfaces, we get this from IScripting using the `GetBrowser()`. You will get a ChromiumWebBrowser interface. This interface is a standard interface for the Chromium web browser. Read more about the chromium web browser at https://cefsharp.github.io/ and Google for the official documentation (ChromiumWebBrowser Class (cefsharp.github.io)).

The `WebBrowser` property is of type IWebBrowser and has a function Load to load a web page into the browser.

Here is a short snippet to open a web page in the web window.

```
using System.Runtime;
using System.Collections.Generic;
using VideoCataloger;
using VideoCataloger.RemoteCatalogService;

class Script
{
  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )
  {
    var browser = scripting.GetBrowser();
    browser.WebBrowser.Load("https://videocataloger.com");
  }
}
```

An alternative to using the integrated browser is to use the network functions in .net. You can find these functions in the System.net namespace. For more information about these functions, look at the Microsoft documentation, Network Programming in the .NET Framework | Microsoft Docs

Here is a simple sample to download a file.

```
using System.Runtime;
using System.Collections.Generic;
using VideoCataloger;
using VideoCataloger.RemoteCatalogService;
using System.Net;

class Script
{
  static public async System.Threading.Tasks.Task Run ( IScripting
scripting, string arguments )
  {
    using (var client = new WebClient())
    {
        client.Credentials = new NetworkCredential("username",
"password");

        try
        {
            client.DownloadFile("https://complete_url_to_the_file_you_want
_to_download/", "c:\\complete_path_to_where_to_save_the_file");
        }
        catch (WebException ex)
        {
            scripting.GetConsole().WriteLine( ex.Message );
        }
    }
  }
}
```