FileLib

Reuters’ files library for the Ric Generator

Abstract: FileLib is a Dynamic library written in F# and C# for the ETI Ric Generator software. It gives interfaces for the manipulation of financial files such as NDA, IDN and FM as well as Data representation for those file types.

*Revisions*

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Author | Section(s) | Comments |
| 11/10/2013 | Aurelien Souchet | All | Creation |
| 02/27/2014 | Aurelien Souchet | All | Edit |
| 03/06/2014 | Aurelien Souchet | All | Edit |
| 04/10/2014 | Aurelien Souchet | 2. 3. | Edit |
| 04/18/2014 | Aurelien Souchet | All | Edit |
|  |  |  |  |
|  |  |  |  |

Summary:

[1. Introduction 5](#_Toc385761113)

[a. Presentation 5](#_Toc385761114)

[b. Problem 5](#_Toc385761115)

[c. Solution 6](#_Toc385761116)

[2. File representation 7](#_Toc385761117)

[a. AFile abstract class 7](#_Toc385761118)

[i. File mode 7](#_Toc385761119)

[ii. File format 8](#_Toc385761120)

[iii. Rules 8](#_Toc385761121)

[Other 8](#_Toc385761122)

[b. Nda 8](#_Toc385761123)

[Content 9](#_Toc385761124)

[Default settings 9](#_Toc385761125)

[Rules 9](#_Toc385761126)

[Exceptions 9](#_Toc385761127)

[c. Idn 9](#_Toc385761128)

[Content 9](#_Toc385761129)

[Default settings 9](#_Toc385761130)

[Rules 9](#_Toc385761131)

[Exceptions 9](#_Toc385761132)

[d. Fm 10](#_Toc385761133)

[Content 10](#_Toc385761134)

[Default settings 10](#_Toc385761135)

[Rules 10](#_Toc385761136)

[Exceptions 10](#_Toc385761137)

[e. Other file 10](#_Toc385761138)

[Content 10](#_Toc385761139)

[Default settings 10](#_Toc385761140)

[Rules 10](#_Toc385761141)

[Exceptions 10](#_Toc385761142)

[3. Usage 12](#_Toc385761143)

[a. Creation 12](#_Toc385761144)

[b. Loadind 14](#_Toc385761145)

[i. From file 14](#_Toc385761146)

[ii. From template 14](#_Toc385761147)

[iii. Entry 15](#_Toc385761148)

[c. Saving 15](#_Toc385761149)

[d. Entry customization 15](#_Toc385761150)

[4. File formatting 17](#_Toc385761151)

[a. Horizontal format 17](#_Toc385761152)

[b. Vertical format 18](#_Toc385761153)

[c. Raw format 19](#_Toc385761154)

[d. AFormat abstract class 19](#_Toc385761155)

[5. Templates 19](#_Toc385761156)

[a. Variable handling 19](#_Toc385761157)

[b. Complex variable usage 20](#_Toc385761158)

[6. Lexing & parsing 21](#_Toc385761159)

[a. Lexing 21](#_Toc385761160)

[b. Parsing 21](#_Toc385761161)

[c. Ast reading 21](#_Toc385761162)

[7. About F# and functional programming 22](#_Toc385761163)

[a. About functional programming 22](#_Toc385761164)

[b. F# 22](#_Toc385761165)

[c. Why choose this technology to C# 22](#_Toc385761166)

[8. Cons 23](#_Toc385761167)

[a. Maintainability 23](#_Toc385761168)

[b. Doability 23](#_Toc385761169)

[c. Not fully tested 23](#_Toc385761170)

[10. Future & Improvements 24](#_Toc385761171)

[a. Template 24](#_Toc385761172)

[i. Features 24](#_Toc385761173)

[ii. Functions 24](#_Toc385761174)

[b. New file types 24](#_Toc385761175)

[c. User Interface integration 24](#_Toc385761176)

[d. Excel Add-in 24](#_Toc385761177)

# Introduction

## a. Presentation

ETI Ric generator is a software made by Thomson Reuters, it can run some automation “tasks" to help perform work that Business Analyst have to do every day but don't necessarily have the time to do. So they can win time efficiency on daily-task, but also do some that they otherwise not do because the effort/time ratio is not worth it.

A "task" is mainly consisted of 3 parts:

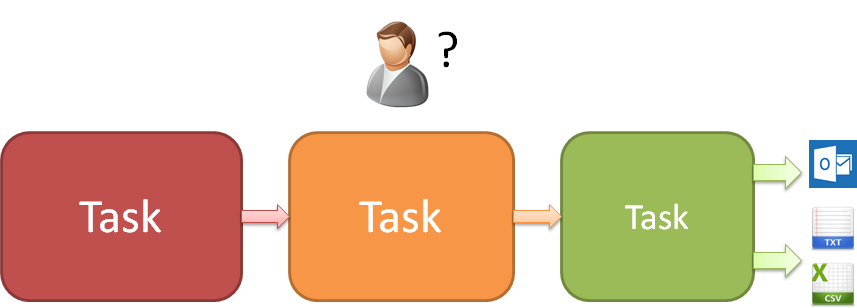


* Information collection: Can be from a website, from an Excel file, a FTP, an Email … That makes every task very specific. But if the data to collect comes from an Nda or a file with a known format, it’s not that different.
* Information formatting: Once the information collected, it needs to be formatted. Most of the time into a format such as IDN or NDA, here a pattern comes out, even if every market differs in some point, the main part is still in common.
* Distribution: The information now formatted, they need to be saved in .txt, .csv or .xls files and eventually need to be sent by mail. Here too, those actions are redundant and some of them are often the same.

## b. Problem

After describing the workflow of a task, we can see that a lot of parts are always the same, and every time we have a task to write the same code will be reused lots of time. And different developers will without even knowing it write the same functions.

Furthermore, analysts doesn’t have any control on the output, any slight modification will need changes from a developer.



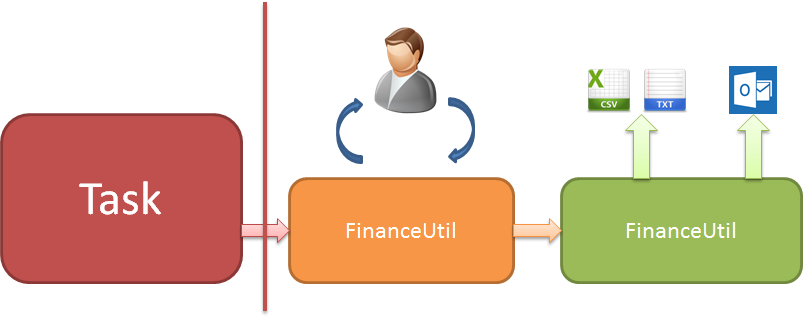
## c. Solution

Writing Interfaces to handle the main file type that Thomson Reuters’ business analysts are working with seems a good way to solve the problem. It can provide a better and more efficient way to create Fm or Nda file.

Using a common interface also makes code more readable, it avoids different developer to write the same code twice but with some differences in the usage and different names.

It will also ensure that the algorithms and the data structure used are the most adapted, and that the exception handling is done write, making every task less error prone.

And now analysts have more control on the result file as the developer may just do the data collection part then let the analyst write its own template for the file he wants to create.



# File representation

In this part will show how all the file classes available in the library are represented.

As every kind of file got a base in common, an abstract class AFile was created to regroup everything that files have in common and avoid boilerplate. It also serve as base for every one that would like to create a new class file representation.

You can find all the classes and code we talk about in this section in the project Ric.FileLib in the Ric Generator solution. Surprisingly most of the code is in this class, after all an NDA is just a normal file with some rules and a format, so that’s pretty much all you will find in the subclasses implementations.

First let’s see what’s in this abstract class, then we’ll go more in details about existing implementations.

## AFile abstract class

The AFile abstract class contains all the elements in common for every file type. Every one of them must inherit from AFile.

### File mode



The mode of the file will determine what the actions you can do with it are:

* **ReadWrite**

Can do everything

* **WriteOnly**

A file with WriteOnly mode cannot be iterated, it should be used when creating file from scratch or creating file from template.

This way the entire file loading is avoided and there is no conversion between format and file model, so performance can be improved.

* **ReadOnly**

On the opposite ReadOnly mode files are only for grabbing data, you cannot modify it.

I you want to be sure that no one can modify this file or if you write a data provider library that return a file inheriting from AFile, then it’s the way to go.

The default value depends the children type, most of the time you can set it in the constructor, but some children could choose to not let the user change it.

### File format

FileFormat represents the way the data will be formatted when written to a file, the basic formats are

Nda or Idn like. First raw is the titles then each row is an entry.

- **Vertical**

Fm like. 2 columns file, left one is title right one is value.

- **Raw**

Not formatted file.

All those formats inherit from AFormat (see section 6) and other can be implemented.

### Rules



TitleRule is bound to hold rules checking functions for titles.

EntryRule also holds checking functions but for entries.

Rules are functions that will check if the titles or entries respect the file format given rules.

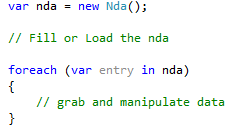
Each subclass defined their own rules and can let the user add some.

### Other

The Path of the file

## Nda

### Content

The content of an Nda file is represented as a List of NdaEntry object, Nda implements IEnumerable interface, so recommended way to grab data from Nda file is that way:

Or with Linq to object:



### Default settings

By default Nda is set with a horizontal format, ReadWrite Mode.

Basic entry NdaEntry which contains the following properties.

### Rules

### Exceptions

All errors

## Idn

### Content

### Default settings

### Rules

### Exceptions

## Fm

File maintenance files, or “FM”, got two different format possible. A vertical FM doesn’t have very strict rules and is often bound to be sent via mail to concerned people. Horizontal FM looks a lot like Nda files, but less restrictive.

### Content

### Default settings

### Rules

### Exceptions

## Other file

“File” represents all other kind of file, so no specific rules or titles, this class should be used when the analyst need a file that does not correspond to a particular type like the one cited before.

Be careful if your task also use System.IO the class “File” will have ambiguity with this one, so write the full name or create an alias:



### Content

### Default settings

By default its format is Raw, and FileMode is ReadWrite.

The Entry is set to null, so if you want to read or iterate it, you can use the dynamic entry, or set by yourself a custom entry type (should inherit AEntry)

### Rules

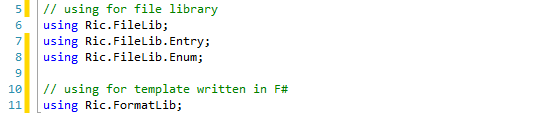
This kind of file doesn’t have any rules by default, this is to add a class that contains all the helper function to create and read file but gives you more liberty with less constraints.

### Exceptions

The basic exception type for this class is FileException, be aware of name ambiguity again.

As every other exception you can find it in the FileLib.Exception folder/namespace, should be used when rules want to throw exception, as this type is made to let you with more liberty, FileException is a sealed class.

# 3. Usage

To use the file library you need to put the following references:

Just the first one is mandatory, other depends the usage of the file you are doing.

All part 3 explanations and examples should work on all AFile subclasses, to see special rules or default behavior for a particular type see Part 2.

## a. Creation



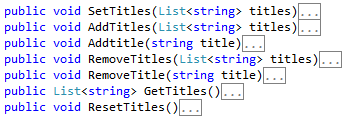
Nda's constructor takes an optional Format as parameters, by default it's Format.Horizontal, but you can always choose a different one.

The filemode is ReadWrite by default, but you can choose it on construction

Now the file is initialized, but is empty, you can either load it from a template or an existing file (next section) or fill it by yourself (see below).

First we need to set the titles, which are represented like that:

So this is just a list of string, but it's private because not any string can be set as a title, so some functions are provided to let you manage the titles and still ensure that the file will be correct:



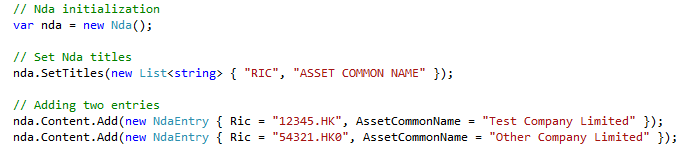
The function names are self-explanatory, just AddTitle(s) will check titles before adding them in the list;

The entries are like that:

Not all the parameters need to be filled, for example if your titles doesn't contain "LOT SIZE", you don't need to fill .LotSize and can let it with its default null value.

The AEntry object is just an abstact class. Every main type file have a default child class of AEntry that contains most often used file field, if those field are not enough for the file you want to create, you can specify another class to be taken instead of the basic entry (see section 3.d).

Full example



After saving the Nda to a Csv file this is the result:

*RIC,ASSET COMMON NAME*

*12345.HK,Test Company Limited*

*54321.HKO,Other Company Limited*

## b. Loading

### i. From file

Loading from an existing file is very simple, see this function:

The path is optional if you already set the path before

After calling this function, the file is loaded and you can iterate it to get entries as AEntry.

### ii. From template

To load a file from template, you need two parameters, the template and the properties:

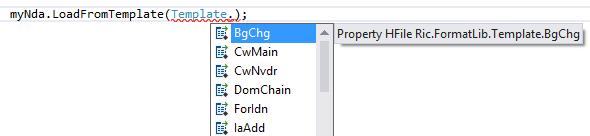
The first parameter is the path where the template file is, and the second a list of dictionaries containing the properties that the template needs to be generated (see section 5 for more information about template)

The FormatLib library will then load template file and give it back to the FileLib, it will assume that the file format and the template format are matching.

If you don't want to load the template from a file but from one directly written In F#, you can call this function:

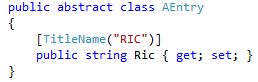
Be aware that trying to use a VFile with horizontal format or vice-versa will throw an exception, the format of the file and the template you want to use must match.

the first parameters is just a template, they are located in the "Template" module from FormatLib namespace, so to have the full list just type "Template." in Visual Studio and intellisense will show you the list of available templates. To be more specialized different modules were created for every types, so if you are writing an Fm template you should put in in the TemplateFm module this way template can be found more easily.



### iii. Entry

To know what the relation between Title and AEntry Property is, see the AEntry class, every property got a TitleName attribute:



Supported types are string, int and DateTime. See more about entries in section 3d.

## c. Saving

To save, just call this function.

Or if you already set the path before:

Path is where you want to save the file, the file will be saved in the correct format automatically depending the extension, .csv, .xls/.xlsx and .txt. If no extension is given, the textual format is used.

## d. Entry customization

If you want to load or create a file with more fields than the basic entry gives, you can create a new child class of AEntry to have one that suits you.

All entry classes are located in the FileLib project, under the Entry folder, and every new child of AEntry should be there.

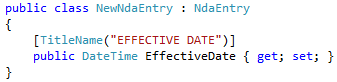
By default every file got a basic entry type

Nda -> NdaEntry

Idn –> IdnEntry

[Filetype] -> [Filetype]Entry

Every type entry class is shown in details in their respective part in Section 2

So let’s say you want to add an “EFFECTIVE DATE” field in your Nda file and the basic NdaEntry doesn’t have this field, you can create a new class NewNdaEntry that inherit NdaEntry.

Then you need to call a method to specify that you want to use a custom entry type



Little example:



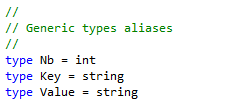
This method will check that the given Type is a child class of AEntry; if it's not the case the method won't have any effects. Even it’s possible to directly inherit from AEntry, it’s not recommended though. For logical reasons and clarity, always inherit from the basic entry type from the file type you are writing.

# 4. File formatting

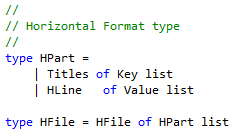
The formatting is the conversion between the file model and the raw values obtained by reading the file on the hard drive.

This part is also in charge of all the distributing and saving of result files and the parsing of template files if used.

Written in F#, this is the core of the Library and despite the fact that it should rarely be changed in this part I will explained how it works.

Some alias for string and int type were created to make type more readable:

## Horizontal format



Simple horizontal template example:

let DomChain =

        HFile([

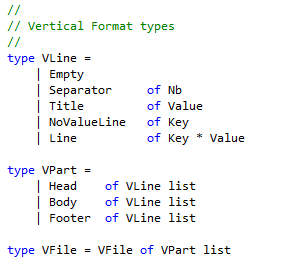
                Titles(["SYMBOL"; "DSPLY\_NAME"; "RIC"; "REF\_COUNT"; "LINK\_1"; "LINK\_2"; "EXL\_NAME"])

                HLine(["0#{ric}if[{ric}.LENGTH(11)]then[o.BK]else[ol.BK]"; "{display}"; "0#{ric}if[{ric}.LENGTH(11)]then[o.BK]else[ol.BK]"; "3"; "{ric}ol.BK"; "{ric}ol.BKd"; "SET\_EQLB\_W\_OL\_DOM\_CHAIN"])

    ])

## Vertical format

Vertical format files, although different depending on the market, the operation type or other variables, we can see that every a pattern comes out:

This is the main representation of a vertical format file in F#

So a vertical file is represented as a list of parts, each of them represented as a list of lines.

The three different vertical parts all have the same type, they are all a list of VLine, what differentiate them is in the file generation, the Head will be written first, then the Body and to finish the Footer.

If it’s a bulk file, only the Body will be written multiple times, notice also that multiple parts are not mandatory, you can create a file composed just of one Body or one Head that would works just fine.

Line representation

* Empty line: nothing written
* Separator : “-” repeated a X number of time to separate parts
* Title: Some text written
* NoValueLine: The line got a field but the value is empty
* Line: The most common line, one field and one value

Which can be represented in F# code in a type called VLine, see code below:

An example of a full file representation in F#:

let TwTemplate =

VFile([

                 Head([Title("PROFORMA - ADD")

                       Empty

                       NoValueLine("FM Serial Number")])

                 Body([Empty

                       Title("For AQS/TQS")

                       Separator(10)

                       Line("Effective Date", "{effectivedate}")

                       Line("Bond\_type", "CONVERTIBLE")])

                 Footer([Empty

                         Title("=== End of Proforma ===")])

         ])

Template can be directly written in F# for further use, or can be loaded automatically from a text or CSV file using the Load method provided with the FinanceFormat class

## Raw format

## AFormat abstract class

[<AbstractClass>]

type AFormat() =

    abstract member Save :          string      -> unit

    abstract member Load :          string      -> unit

    abstract member Generate :      unit        -> unit

    abstract member ParseMyValue :  string      -> string

    abstract member Send :          List<string> -> unit

abstract member AddProp :       Dictionary<string, string> -> unit

# 5. Templates

## a. Variable handling

A problem can be easily spotted, some values, like the date or the Ric number, changes with every Fm, so hardly write them in the template is not a good option.

Another solution is to have a Map in which we can find the value:

I see a field called “Ric” so I replace the value by ValueDictionary[“Ric”]

It seems a good idea but what if I have a field “Ta Ric” that is the same ric plus “ta” at the end, I have to make a new entry in the Map, same if I got a field with just the first four characters in the ric.

The given solution here is to let the developer set some variables, then let him use them as much as he wants, and let him do some changes.

To add a variable, just use the “AddProp” method from FmFormat.

In the template, the variables can be placed within brackets, if you added a “ric” variable with “12345” as value , “{ric}” will give “12345” This way if another field needs to have the ric plus “ta.TWO” ta the end, just write “{ric}ta.TWO” in the template, it will become “12345ta.TWO”

## b. Complex variable usage

The examples shown above are simple, but real situations are often more complex, you sometimes could need just the first two characters of the variable, or some conditions to pass to write it.

To handle those more complexes situations, this is the given solutions:

* Substring : if you need just a part of a string, you can use the “%” keyword
  + “{ric}%3” => “123”
  + “{ric}%2,4” => “234
* Functions:
  + {ric}.STARTSWITH(5)
  + {ric}.ENDSWITH(4)
  + {ric}.EQUALS(12345)
  + {ric}.EQUALS({ric2})
  + {ric}.LENGTH(3)
* Condition: they are to be used with Functions, something else will always be considered true
  + If[{ric}.STARTSWITH(5)]then[{ric}.CC]
    - Writes “”, would write “51234.CC” if ric was “51234”
  + If[{ric}.STARTSWITH(5)]then[{ric}.CC]else[{ric].CS]
    - Writes “12345.CS”, would write “51234.CC” if ric was “51234”
* Logical OR and AND
  + ||(Or)
    - If[{ric}.STARTSWITH(5)||{ric}.STARTSWITH](6)]then[{ric}.CC] <- Matches whether the ric starts with 5 or 6, if it doesn’t matches nothing is written
  + &&(AND)
    - If[{ric}.STARTSWITH(5)&&{ric}.ENDSWITH](6)]then[{ric}.CC] <- Matches only if ric starts with 5 and ends with 6

# 6. Lexing & parsing

## a. Lexing

During the lexing phase, the string is divided into “token”

Exemple:

{ric}%2,3 -> LSCOPE VAL RSCOPE PERCENT NB COMMA NB

## b. Parsing

The parsing phase will use a given grammar to match token with rules. It created an Abstract Syntax Tree.

## c. Ast reading

A recursive function is used to go through the abstract tree, the final value is returned

# 7. About F# and functional programming

## a. About functional programming

## b. F#

F# is basically the OCaml language, created by the INRIA in France, with .NET Framework.

As Ocaml, F# is a multi-paradigm language. And a big part of the Format lib is object oriented and .NET is also widely used, all that to help the library being very easily understood even for people that are not familiar with functional programming or F#.

Since the release of Visual Studio 2010, F# is a first class language like VB, C++ or C#

## c. Why choose this technology to C#

# 8. Cons

## a. Maintainability

We just explained why choose this technology and the advantages it gave for the library implementation, but It also have some cons, like maintainability. F# and functional programming are not widely known by developers who are more used to object oriented programming.

To solve this problem, writing a full documentation is very important, including some video presentations, and extended comments in the code. Furthermore the FormatLib library is not fully functional; there are a lot of objects so you the code is easily readable. It’s the back-end, when it’s totally stable the code should be changed very rarely, most often changes will need to be done to the FileLib which is written in C#.

Lot of the types used in the F# are .Net type, so it’s easier to understand and C# developers won’t be totally lost.

## b. Doability

Doing some representation classes for most of the Thomson Reuters financial files is a good idea, but most of the markets have their own rules, even if the library is very customizable with the file entries, fields, or applied rules. Because if every time a new task is written a new AEntry class must be written and need to add a lots of new rules, that destroys the idea of a general class.

Even if the generalization is not 100% achieved, time will be gained in task writing because of all the utility functions like saving or sending mail or just write simple files.

## c. Not fully tested

The library is still in an early stage, so some bugs could appear, mostly in the template parsing. A Unit test project is available in the solution, with example files.

In the conversion between Format and File there could be bugs too, some unit tests are also available for this part.

But if some bug appears it shouldn’t be big ones and should be solve rather quickly.

# 10. Future & Improvements

Here you can see the list of new features or improvements that should be made to the library in the future.

## a. Template

### i. Features

### ii. Functions

## b. New file types

Em File

Ema File

## c. User Interface integration

## d. Excel Add-in