# Scripting Utilities

This document describes a set of useful scripting functions for OpenText Capture Center along with sample profiles and sample documents.

Version of the Scripting Utilities: 1.7

OCC version required for the profiles: 16.4.1

Contents

[Scripting Utilities 1](#_Toc22200592)

[Introduction 3](#_Toc22200593)

[Using the samples 3](#_Toc22200594)

[Using the assemblies 3](#_Toc22200595)

[Modifying the assemblies 5](#_Toc22200596)

[Imprinting 6](#_Toc22200597)

[Redaction 7](#_Toc22200598)

[Get Snippet 9](#_Toc22200599)

[Blank Page Removal 10](#_Toc22200600)

[Email Rendition to PDF 11](#_Toc22200601)

[Join Documents 13](#_Toc22200602)

[PDF Compression 14](#_Toc22200603)

[Export Attachments 16](#_Toc22200604)

[Render Attachments 18](#_Toc22200605)

[Delete Uncertain Rows 19](#_Toc22200606)

[SQL data base validation 20](#_Toc22200607)

[Auto-skip validation 22](#_Toc22200608)

[Sort documents based on class name 23](#_Toc22200609)

[Notification for validation 24](#_Toc22200610)

[Highlight all fields on the document 25](#_Toc22200611)

[Extract table column-wise 26](#_Toc22200612)

[Lookup list updater 27](#_Toc22200613)

[How to use 27](#_Toc22200614)

[Important note 27](#_Toc22200615)

[Extract Image Metadata 29](#_Toc22200616)

[Invoice Totals Field Handling 30](#_Toc22200617)

[License Counting per Profile 32](#_Toc22200618)

[Save Table as CSV file 34](#_Toc22200619)

[Use multiple profile specific custom extensions 35](#_Toc22200620)

[Prevent broken batches 37](#_Toc22200621)

[Get OCR details 39](#_Toc22200622)

[Separate on Value Changes 41](#_Toc22200623)

[Maintaining Zones in Validation for Forms Applications 43](#_Toc22200624)

[Classify with rules on second page 44](#_Toc22200625)

# Introduction

OCC offers two different ways to change the behavior of a profile by .net code: providing a custom DLL and adding scripting code. Both serve a different purpose:

* A **custom DLL** is created using Visual Studio and allows to implement larger chunks of code using advanced debugging tools. A custom DLL typically adds bigger pieces of functionality to a profile.
* **Scripting code** is entered into an editing window with very limited editing support and no debugging capability. Scripting code will typically be used to add minor functions.

This document and the corresponding C# solution introduce a combined approach. A custom DLL adds functionality to a profile and scripting code is used to trigger and to configure this functionality. This approach became possible when OCC 16 introduced the possibility to add a custom DLL to the scripting interface (in addition to globally registered custom DLLs).

With this scheme, an experienced programmer will implement the custom DLL and the OCC profile developer will use these functions through scripting. No real programming skills are required to use the programmed functionality in a given profile.

OpenText has created a sample C# solution that contain some interesting functionalities and corresponding profiles that show how to use it through scripting. The samples are provided on an as-is basis, no maintenance or bug fixing is provided (although you are welcome to report them). You may freely use and change the sample to your needs.

## Using the samples

The profiles located in the folder *SampleProfiles* should run out of the box unless otherwise noted. The required DLLs are already embedded in the profiles. If an export destination is configured in the profiles it usually is a file system export to C:\temp; you can of course change it.

The samples are designed to work with sample images. You find these in the folder *SampleDocuments*. Each sample lists the profile to be used as well as the sample documents to be used in a section titled “Resources”.

## Using the assemblies

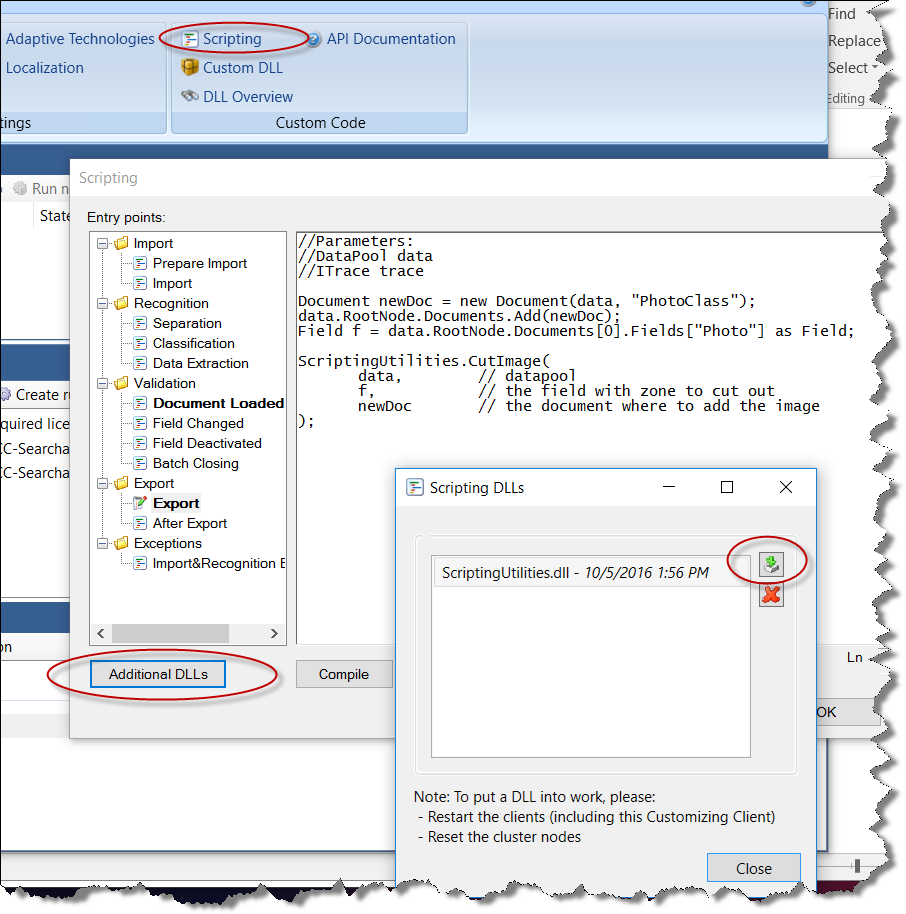
There are two assemblies needed throughout the examples. When you want to use the functionality in your profile you need to load these assemblies into you profile (see screen shot on how to do that).

**ScriptingUtilities.dll**: This assembly contains all functions that can be called from the scripting code as well as lower level classes that supply the functionality needed. It must be loaded using the Scripting->AdditionalDLLs dialog. You need to add “ScriptingUtilities” in the *Namespace* dialog too.

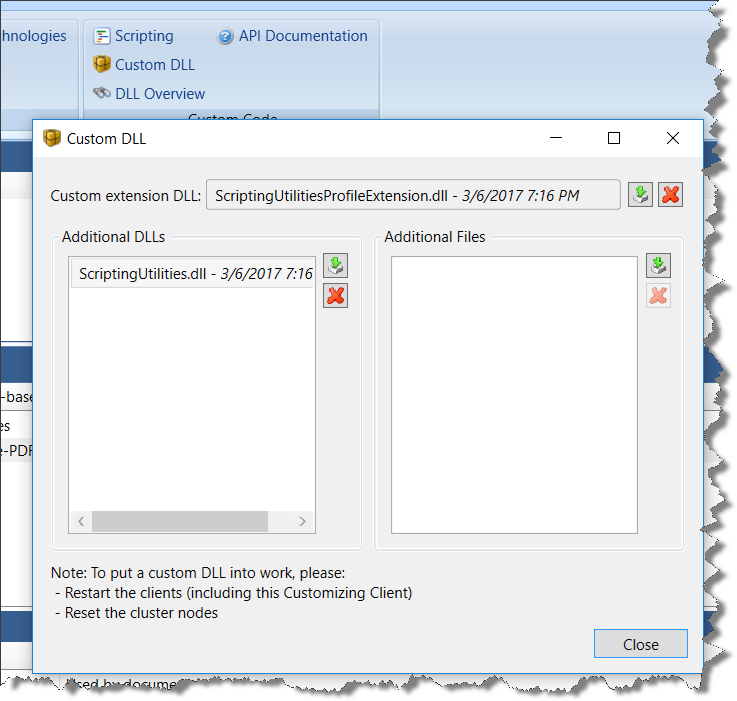
**ScriptingUtilitiesProfileExtension.dll:** This is a profile extension class and contains functions that are called by OCC directly. It must be loaded using the Custom DLL dialog. Please note that ScriptingUtitlities.dll is required as additional DLL in this dialog too.

You will also need to add “ScriptingUtilities” in the Namespace dialog. Some samples also require other system DLLs to be named in the namespace dialog, most notable System.Collections.Generic.





1. Loading ScriptingUtilities.dll



2. Loading ScriptingUtilitiesProfileExtension.dll together with ScriptingUtlitities.

## Modifying the assemblies

In order to modify the assemblies, you need to load the Visual Studio solution. Make sure you replace the references to OCC DLLs with references to your installation folder. Make also sure you adapt the build destination to your OCC installation folder. Compile the solution to create both assemblies.

Please note that the ScriptingUtlities.dll cannot be debugged by attaching to the executing recognition node. The way to develop the DLL is to load a datapool from the OCC repository as can be seen from the unit tests. (These unit tests are actually used as launcher to execute the function in the ScriptingUtilities.dll.)

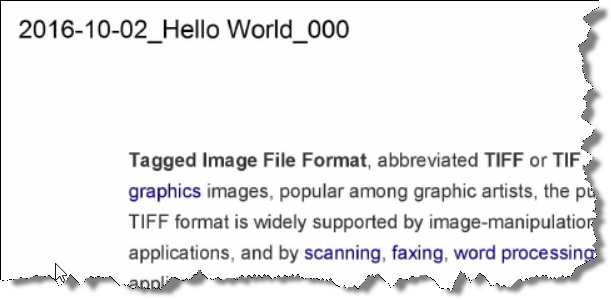
For ScriptingUtlitiesProfileExtension.dll see the API documentation for OCC, chapter “Debugging a Registered Profile-Specific Custom Extension Class”.

Please note that some part of the code is only compiled conditionally, e.g. EmailToPdf or PdfCompression. If no conditional compilation symbols are defined a dummy function will be included. In doing so, no external libraries are required by default.

The solutions assume that there is a local OCC installed at default locations. The Reference Path in Visual Studio is set to “C:\Program Files (x86)\OpenText\Capture Center 16.2\”. Compilation copies all DLLs into a subfolder “subin” in the solution folder and copies DLLs and PDB files to "C:\ProgramData\Open Text\DOKuStar Dispatch\Data\Config\CustomCode". The latter will update the profiles that use these DLLs.

# Imprinting

An imprint is sometimes called watermark. It is a piece of text writing onto each page of a document to show that it has been processed. High volume scanners provide the option to physically print something on the physical paper during scanning. The imprinting function here simulates that behavior within OCC. The picture below shows the result of imprinting. (“2016-10-02\_Hello World\_000”).



Below is the scripting code that has been added to the export scripting step. It calls the Imprint function in the ScriptingUtilities class and stores its result, which is the string imprinted on the first page of the document, in the field “Imprint”.

string result = ScriptingUtilities.Imprint(

data, // datapool

"<Date>\_<:Field1>\_<PageNumber>", // imprint string

true, // all pages

3, // number of digits for pagenumber

true, // input image

true, // OCR image

2, // Orientation: 0 = horizontal, 1 = vertical

0, // Corner: 0 = northeast, 1, 2, 3 = clockwise

30, // X-ofset in pixel

30 // Y-offset in pixel

);

data.RootNode.Documents[0].Fields["Imprint"].Value = result;

The meaning of the parameters should be obvious except the imprint string specification. It is a special syntax that contains constant parts (in this case the two ‘\_’ characters) and variable parts (enclosed in ‘<’ and ‘>’) that are substituted per page. The supported placeholders are:

* PageNumber: current page number starting with 0
* Host: name of the machine executing the code
* Date, Time: Current date and time
* UniqueId: Some unique 11 characters string
* Fieldname: A colon indicates the name of a field in the document

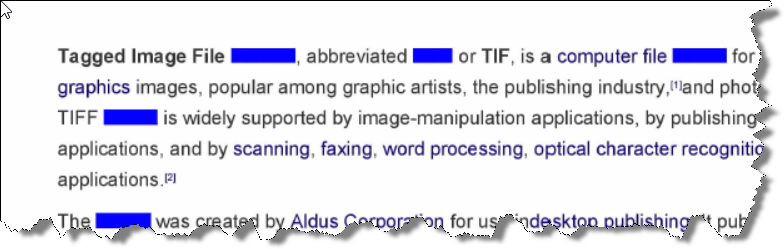
**Resources**

|  |  |
| --- | --- |
| Profile | Imprint.zip |
| Sample document | TestDocument\_Imaging.pdf |
| DLLs used | ScriptingUtilities.dll (Additional DLL) |

# Redaction

Redaction means to make certain information on a document unreadable. Often this is necessary when documents need to be published, e.g. for legal reasons, but personal information like credit card numbers should still be protected.

The below image shows that certain words have been obscured by blue color. Specifically, the first occurrence of the word “TIFF” has been removed as well as all occurrences of the word “format”.



The example profile contains two fields automated by rules based automation and a non-automated field. The first is a regular field, the second is a new field type, the list field. It can contain several values and behaves a bit like a table with one column. In the example profile is detects all occurrences of the world “format”.

The scripting code is in two places. In the export scripting step, the redaction is triggered. Here’s the code.

// Redact all fields

// ScriptingUtilities.Redact(data, "green");

// Redact only two fields

List<string> fl = new List<string>() {

"RedactF.\*",

"ManualField"

};

ScriptingUtilities.Redact(data, "#FFDFD991", fl);

The function ScriptingUtilities.Redact is called in two ways. First it is called with two parameters in which case all fields will be redacted. In the second call it has three parameters and the additional parameter is a list of field names that should be redacted[[1]](#footnote-1). The color can be either specified as a [color name](https://msdn.microsoft.com/de-de/library/system.windows.media.colors(v=vs.110).aspx) or by a hex value.

There is also a script in the Validation -> Document Loaded scripting step. Its purpose is to influence the behavior of Single Click Entry (SCE). Normally the zone created by SCE is the bounding box around the zone the user has drawn. So the final zone is typically smaller than the one the user has drawn. But for redaction one would like to give the user full control of the zone; consider cases where the OCR didn’t find the sensible information because some character was not found by the OCR.

Here’s the code:

//ScriptingUtilities.AddFieldAnnotation(doc, SCERawZone");

doc.Fields["ManualField"].Annotations.Add(new Annotation(doc.OwnerDataPool, "SCERawZone"));

­In order to have Single Click Entry behave different for a certain field one has to add a specific annotation to that field. There two variants in the code. The foreach loop adds the annotation to all fields. The second variant adds the annotation to one specific field.

There is one more important aspect with redaction. If you create a PDF as output, you need to make sure that this (searchable) PDF does not contain sensible information. You achieve this by enabling the “High fidelity PDF” option in the export destination. In doing so the text for the searchable PDF is created from the modified document pages.

**Resources**

|  |  |
| --- | --- |
| Profile | Redact.zip |
| Sample document | TestDocument\_Imaging.pdf |
| DLLs used | ScriptingUtilities.dll (Additional DLL) ScriptingUtilitiesProfileExtension.dll (Custom DLL)[[2]](#footnote-2) |

# Get Snippet

This example shows how to cut out a piece of a page and create a new document based on this snippet. A typical use case is in onboarding solutions when e.g. a person’s picture or signature is to be provided as a separate document.

The sample profile contains two document classes. “DocumentClass” contains one field called “Photo”. In the export scripting step, the zone of the Photo field is used to cut out a piece of the page and to create a new document of class “Photo” from it. This new document is added as a second document to the batch. Here’s the code:

Document newDoc = new Document(data, "PhotoClass");

data.RootNode.Documents.Add(newDoc);

Field f = data.RootNode.Documents[0].Fields["Photo"] as Field;

ScriptingUtilities.GetSnippet(

data, // datapool

f, // the field with zone to cut out

newDoc // the document where to add the image

);

The first two lines create a new document and add it at the end of the batch. The third line gets a reference to the source field from which the image will be cut. The last statement cuts the image and inserts it into the new document. Please note that this example assumes that there is just one document in the batch from which a snippet should be extracted.

For validation the Single Click Entry behavior is modified to provide the user’s original zone as has been described in the Redact example.

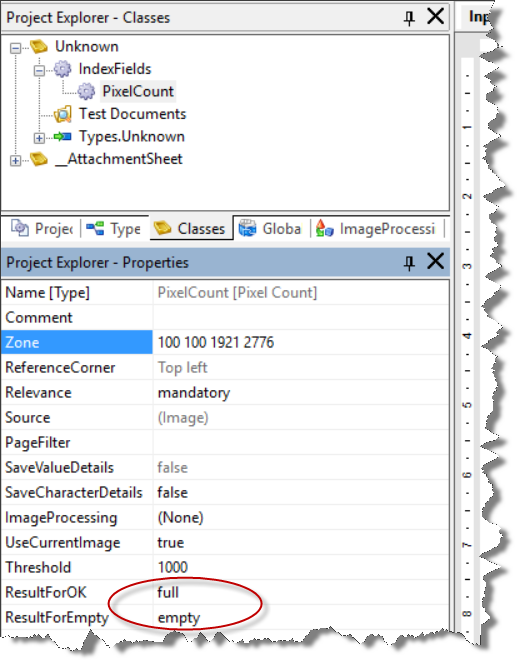
**Resources**

|  |  |
| --- | --- |
| Profile | GetSnippet.zip |
| Sample document | TestDocument\_Imaging.pdf |
| DLLs used | ScriptingUtilities.dll (Additional DLL) ScriptingUtilitiesProfileExtension.dll (Custom DLL)[[3]](#footnote-3) |

# Blank Page Removal

Removing blank pages from documents consists of two steps: Detection of blank pages and their removal.

Detecting blank pages is done by defining an OCC field that is automated by a rule. Typically, the rule will be a pixel counter covering the whole page. However, more sophisticated rules may be required. For example, an images specific preprocessing may be required to distinguish between a page with some dirt or gray shade from a page containing only a few lines of text.

At the end of the recognition step the field automated by a pixel counter will have a result in for the first page and a field alternative for each additional page. The pixel counter provides a setting to define values for checked and unchecked as you can see from the screen shot. These values will be used to decide on the removal of the page.

This is the code that needs to be placed into the “Data Extraction” scripting point:

ScriptingUtilities.RemoveBlankPage(data, "BlankPage");

//ScriptingUtilities.RemoveBlankPage(data, "BlankPage", "empty");

Besides the data pool the function has two parameters:

* fieldname (here “BlankPage”): This is the name of the OCC field holding the results of the pixel counter field.
* valueForEmpty (here “empty”): This is the value the signals that the page should be removed. This parameter is optional and defaults to “empty”.

**Resources**

|  |  |
| --- | --- |
| Profile | BlankPage.zip |
| Sample document | BlankPage.pdf |
| DLLs used | ScriptingUtilities.dll (Additional DLL) |

# Email Rendition to PDF

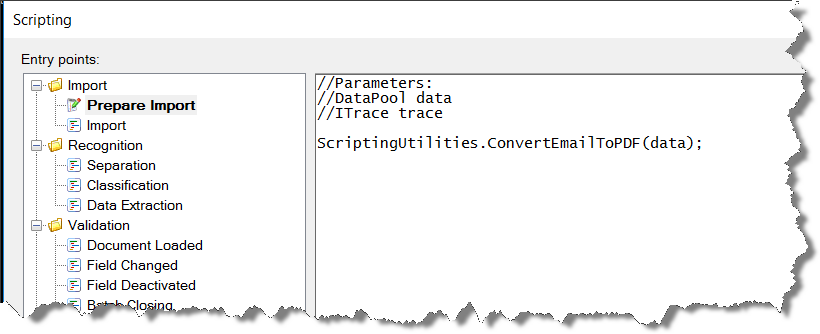
OCC renders the email body to tiff using a very simple algorithm that can deviate heavily from the original. There are tools available that try to preserve the look of a HTML page when creating a PDF. Using existing programming options in OCC one can convert the email body into a PDF before the recognition sees it. Converting into PDF has the additional benefit of preserving the true text. OCC will ignore its OCR result and use the strings from the HTML instead.

The source file “EmailToPdf.cs” implements the conversion[[4]](#footnote-4). It uses an online conversion service from [PdfCrowd](https://pdfcrowd.com/). However, this is neither a recommendation or endorsement for this software. You are encouraged to pick a package you feel is best. You only need to change the source file accordingly.

In order to use the web service from PdfCrowd you need to register on their website. There is a free registration for 100 or so conversions for private use. After registration you receive an api\_key that the software needs. The conversion code expects the user name and the api\_key to sit in the registry (see actual code to the location). But you may prefer to hard code the values into the code when playing around.

In order to compile the solution with the “PdfCrowd” compiler flag you need to download the .net client library from their web site (<https://pdfcrowd.com/web-html-to-pdf-net/>) and copy the files “Client.cs” and “Error.cs” into the PdfCrowd subfolder.

"Using the function is straight forward. You call the method “ConvertEmailToPDF as you can see below. There is an additional second parameter of type integer that specified the location in the sequence of documents where the new document should go. Zero mean “first document”, 1, 2, and so forth stands for the respective position and -1 (default) means “add at the end”.



**Resources**

|  |  |
| --- | --- |
| Profile | Email2pdf.zip |
| Sample document | None, read an email |
| DLLs used | ScriptingUtilities.dll (Additional DLL) |
| Conditional compilation | PdfCrowd (ScriptingUtlilities.csproj) |

# Join Documents

This function combines several documents into one. It is especially useful in conjunction with the email rendering function in the previous section. If you turn on header document creation in the email hotspot and convert the body into a document of its own you will get two documents. You may prefer to have one instead.

JoinDocuments will typically be called in in the separation scripting event. The function has a variable number of parameters. The first parameter (after the data pool) is the index of the master document, the document where the pages of the other documents will be added to. The remaining parameters are the indexes of the pages that should be combined with the master document. If only a master document is given, all document in the batch are combined with the master document.

ScriptingUtilities.JoinDocuments(data, 0, 1); // combine document 0 and document 1

ScriptingUtilities.JoinDocuments(data, 2, 0, 2, 3); // combine docs 0, 1, 2 and 3

ScriptingUtilities.JoinDocuments(data, 0); // combine all documents into one

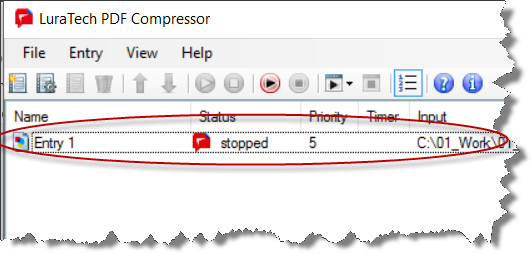
**Resources**

|  |  |
| --- | --- |
| Profile | Email2pdf.zip |
| Sample document | None |
| DLLs used | ScriptingUtilities.dll (Additional DLL) |

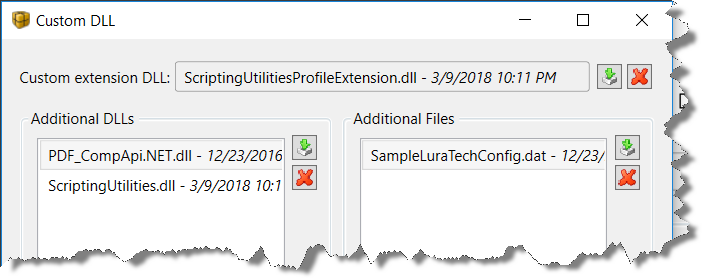
# PDF Compression

OCC converts the single pages of a document to images, either black and white image or color image and creates a PDF from these images. Black and white images are compressed using Group 4 lossless compression, color images are compressed using the well-known Wavelet method (JPEG). There are algorithms available that can compress color images even further, especially for business type documents. OCC does not provide these algorithms but they can be integrated and used.

The ScriptingUtlilities contain a sample that uses the [Luratech](https://www.luratech.com/en/) compressor. Other compressors can be used in a similar fashio. In order to use the Luratech compressor one needs to install Luratech’s software on each machine where recognition nodes are executed[[5]](#footnote-5). Luratech’s compressor uses so called jobs as units of configuration. In such a job one can define the details of how the compressor works. You need to create such a configuration and store them into a file (extension .dat).



In order to use the PDF compressor, you need to add several files to the “Custom DLL” dialog:



First you need to add the ScriptingUtilitiesProfileExtension DLL. This contains code executed whenever a PDF has been produced by OCC. It will call the Luratech DLL that you also need to add: PDF\_CompApi.Net.dll.

**The Luratech assembly *PDF\_CompApi\_32.dll* needs to be in the OCC installation folder**. You find the assembly in Luratech’s installation folder in the subfolder “api”. Finally, you need to add the Luratech configuration file created above, the .dat file in the additional file section.

The scripting code is simple and needs to be added to the **Export scripting section**. It will add an annotation to each document in the batch which will mark it for PDF compression[[6]](#footnote-6). The content of the annotation is the name of the file containing the Luratech configuration.

ScriptingUtilities.AddDocumentAnnotation(

data,

"SU\_PdfCompression",

“SampleLuraTechConfig.dat"

);

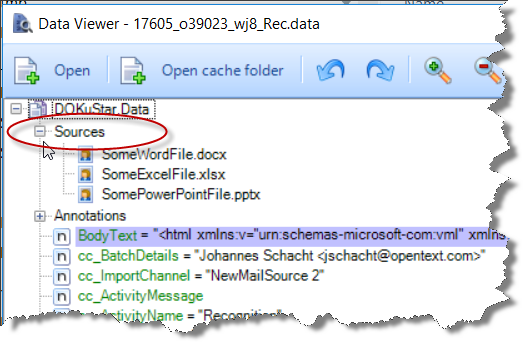
If you want to use another PDF compressor you will need to change the code in the PdfCompressor project in the ScriptingUtilities solution.

**Resources**

|  |  |
| --- | --- |
| Profile | PdfCompression |
| Sample document | ColorImage.jpeg |
| DLLs used | ScriptingUtilities.DLL (Additional DLL)  ScriptingUtilitiesProfileExtension.dll (Custom DLL)  PDF\_CompApi.Net.DLL (Custom: Additional DLL)  PDF\_CompApi.32.DLL (OCC installation folder)  SampleLuraTechConfig.dat (Custom: Additional File) |
| Conditional compilation | LuraTech (ScriptingUtlilitiesProfileExtension.csproj) |

# Export Attachments

OCC only processes images (TIFF, JPG, PNG, …) and PDF. Other document types cannot be processed by the engine. Sometimes however such document types are part of a batch. An example would be an email that has some office documents attached to it. The email hotspot connector makes these attachments part of the batch (data pool) as sources at the batch.



The export will ignore these files. In order to export them one needs to create an OCC-document for each source and add the source file to it. The following script in the Export scripting event does exactly that.

//ScriptingUtilities.ExportAttachments(data);

//ScriptingUtilities.ExportAttachments(data, 0);

ScriptingUtilities.ExportAttachments(data, 0, "DocumentClass2");

It takes one of the documents in the data pool and clones it for each attachment. It has two optional parameters. The number indicates the document after which the new documents will be inserted. It defaults to -1 which indicates to add them at the end of the batch. The second parameter is the class name of the newly created document. It defaults to “Unkown”.



**Please note: When exporting to file system the file names will end in “.pdf”.**

**Resources**

|  |  |
| --- | --- |
| Profile | EmailAttachments.zip  The profile has code in the “Data Extraction” scripting point that adds a page to the Source collection at the root. The code is commented out. |
| Sample document | None, create a mail hotspot or uncomment the above mentioned code. |
| DLLs used | ScriptingUtilities.DLL (Additional DLL) |
| Conditional compilation | None |

# Render Attachments

OCC only processes images (TIFF, JPG, PNG, …) and PDF, as described above. You may want to extract data from these types of documents too. This is possible by rendering these documents to PDF. This script in PrepareImport does the magic.

ScriptingUtilities.RenderAttachments(data,

@"\\occ-3.cloudapp.net\xChange\Default",

0,

"DocumentClass2");

The function RenderAttachments takes three parameters.

* The first parameter provides the exchange folder for OpenText’s Blazon engine. This is the folder has to have the input-job-output subfolders that are created by Blazon.
* The second parameter (zero in the above example) is the index of the document after which the new documents should be added. If the parameter is omitted it defaults to -1 which adds the attachments at the end of the batch.
* The third parameter is the name of the document class the newly created documents should get. It is optional too and defaults to “Unknown”.

**Blazon**

OpenText’s Blazon engine converts various input formats into various output formats. The default output format is PDF which can then be processed by OCC. If, for example, a Microsoft Word file is converted, the resulting PDF will contain the text as strings and with OCC’s e-text modes these strings will be taken instead of OCR’ing bitmap images. The result is error-free input to the document analysis process.

**Resources**

|  |  |
| --- | --- |
| Profile | RenderAttachments.zip |
| Sample document | None, create a mail hotspot |
| DLLs used | ScriptingUtilities.DLL (Additional DLL) |
| Conditional compilation | None |

# Delete Uncertain Rows

When automating tabular data extraction with the rules based engine (Document Extraction, a.k.a. DOKuStar Extraction) one typically gets several additional rows above and below the real table. These rows are marked as “uncertain” by the recognition. Usually you want to remove them from the result. You can use the scripting function RemoveUncertainRows for that purpose. You need to a this script to the Data Extraction scripting point.

ScriptingUtilities.UncertainRowsDefinition ucrd = new

ScriptingUtilities.UncertainRowsDefinition()

{

Top = true,

TopFields = new List<string>() { "Price", "Qty" },

Bottom = true,

BottomFields = new List<string>() { "Total" },

};

ScriptingUtilities.RemoveUncertainRows(data, "Items", ucrd);

Besides the data pool the function takes two parameters of which the second is optional. The first parameter, “Items” in the above case, is the name of the table field. With the second parameter you can specify some details of the operation:

* Top/Bottom specifies whether rows should be removed above/below the table. All uncertain rows above/below the first/last valid row are removed. Default values are Top=false, Bottom=true.
* TopFields/BottomFields lets you specify that rows from the top should **\*not\*** be removed if one of the fields specified contain data. The lists are empty by default.

You only need to assign values to those elements that differ from the default. Thus a definition like **new ScriptingUtilities.UncertainRowsDefinition() { Top=true }** is totally valid. Please note, that when you use TopFields or BottomFields you need to add the windows library System.Collections.Generic to the list of namespaces in the scripting dialog.

**Resources**

|  |  |
| --- | --- |
| Profile | UncertainRows.zip |
| Sample document | Invoice.pdf |
| DLLs used | ScriptingUtilities.DLL (Additional DLL) |
| Conditional compilation | None |

# SQL data base validation

A repeating requirement in capture project is the need to validate data against a database. OCC will extract a set of fields with which a search in a database should be made. If a record is found that matches the field values (and if \*only\* one record is found), one can conclude first that the data is correct and second one can set other fields for the current document based on values in other columns.

The ScriptingUtilities provide two functions for this. One function is used to specify an SQL verification. It is typically used in the Prepare Import step. This function registers information that is used later when executing the validation.

SqlTableSpec tableSpec = new SqlTableSpec(

@"jschachtGXY3JC2\DOKUSTAR",

"OCCExport", "CustomerDB",

"sa", "#DOKuStar#");

List<SqlFieldMapping> fieldMap = new List<SqlFieldMapping>()

{

new SqlFieldMapping("Firstname", "Firstname"),

new SqlFieldMapping("Lastname", "Lastname"),

new SqlFieldMapping("Street", "Street"),

};

ScriptingUtilities.AddSqlVerificationSpec(data,

new SqlVerificationSepc("Test", tableSpec, "DocumentClass", fieldMap));

The function AddSqlVerificationSpec takes one parameter besides the canonical data pool. The parameter is an object of type SqlVerificationSepc which has four elements:

* Name: The name of this specification that is refeneced when the verification is triggered
* Sql table specification: Specifies the connection to thhe SQL data base. This structure in turn contains these elements
  + Database name
  + Catalog name
  + Table name
  + Username
  + Password

Username and password are optiona. In that case the verification function tries to connect to the database to using single sign on.

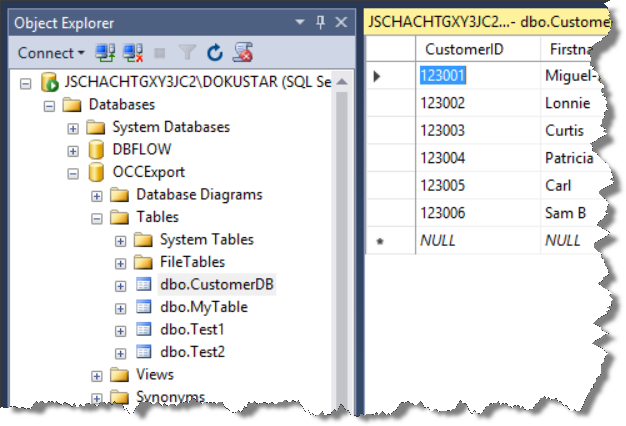
* Document class name: Only for this document class the Sql verification should be done.
* Field mapping: This is a list of string pairs, specifying which data base column name (first pair element) is connected with which OCC field (second pair element).

Once these definitions have been stored the Sql verification can be triggered by a call to SqlVerify. The function takes the name of the SqlVerificationSepc as parameter.

ScriptingUtilities.SqlVerify(data, "Test");

This function can be used either at the Data Extraction, Field Changed or Field Deactivated scripting step.

**Note:** The profile does not work out of the box because it relies on a database table. In the sample a new database had been created in OCC’s SQL server as shown below. The data can be found under the test documents folder (CustomerDB.txt).



**Implementation note**: AddSqlVerificationSpec writes the information to the data pool as annotation. Also SqlVerify only adds an annotation. The ScriptingUtilitiesProfileExtension actually executes the verification.

**Resources**

|  |  |
| --- | --- |
| Profile | SqlVerification.zip |
| Sample document | Any |
| DLLs used | ScriptingUtilities.DLL (Additional DLL)  ScriptingUtilitiesProfileExtension.DLL (Custom DLL)[[7]](#footnote-7) |
| Conditional compilation | None |

# Auto-skip validation

Using the OCC profile settings, you can set “Skip validation” on the validation tab. This will bypass the validation step for all batches. However, sometimes you want to only bypass those batches that are in error. The AutoSkipValidation function does exactly that. The function needs to be called in the “Data Extraction” scripting step.

ScriptingUtilities.AutoSkipValidation(data);

The definition for a batch to be in error is simple: If any field of any document is in state error the batch will show up in validation.

The sample profile *AutoSkipValidation.zip* contains several fields whose values are set by having default values. If you remove the value for one of the mandatory fields or a wrong value to the Amount field (e.g. “xxx”) the batch will show up in validation

**Resources**

|  |  |
| --- | --- |
| Profile | AutoSkipValidation.zip |
| Sample document | Any |
| DLLs used | ScriptingUtilities.DLL (Additional DLL) |
| Conditional compilation | None |

# Sort documents based on class name

This example is fairly straight. One wishes to have the documents in the batch in a certain order, given by document class name. Say you have a mixture of invoices, purchase orders and delivery notes and you want to have them sorted like first all purchase orders in the batch, next all delivery notes and finally all invoices. The function to use is *SortDocuments*, most probably in the “Data Export” step.

List<string> order = new List<string>() { "Class5", "Class3"};

ScriptingUtilities.SortDocuments(data, order);

The functions sorts all documents alphabetically by class name if given no parameter. If the list of names is given as parameter, these specify the sorting order, in above case it means to first put all documents of *Class5* into the batch and then all of *Class3*. All documents of classes not in the list are sorted behind them, again alphabetically.

In the sample profile there are seven document classes (*Class1* to *Class7*). Before sorting all documents in the batch are assigned class names with decreasing numbers so the effect of sorting can be seen.

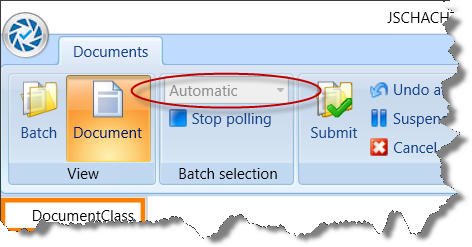
**Resources**

|  |  |
| --- | --- |
| Profile | SortDocuments.zip |
| Sample document | Any, you need several |
| DLLs used | ScriptingUtilities.DLL (Additional DLL) |
| Conditional compilation | None |

# Notification for validation

Sometimes users do only occasionally need to validate documents. In that situation they would like to get some kind of notification when new batches are ready for validation. If the user only needs to watch for one profile, there is a simple scripting solution.

The user needs to run OCC Validation in the automatic mode:



In this mode, Validation will automatically load the next batch if the previous has been submitted by the user. If no batch is available Validation goes into polling mode, regularly checking the OCC server for new batches.

The following script in the “Document loaded” step will show a notification if a new batch arrives and the OCC validation is minimized. **Please note: It will only show a notification when the Icon is set. Point to a valid icon on your system.**

System.Windows.WindowState x = System.Windows.Application.Current.MainWindow.WindowState;

if (x == System.Windows.WindowState.Minimized)

{

NotifyIcon ni = new NotifyIcon ();

//ni.Icon = new Icon(@"some\_folder\some\_icon.ico");

ni.Visible = true;

ni.ShowBalloonTip(5000, "OCC says:", "New work arrived", ToolTipIcon.Info);

}

**Resources**

|  |  |
| --- | --- |
| Profile | None |
| Sample document | Any |
| DLLs used | None |
| Conditional compilation | None |

# Highlight all fields on the document

OCC only displays the current field on a document. Sometimes it is helpful to see all extracted fields at once on the document. That way a user can easily verify whether all required fields have been found on the right spots. The image on the left shows an example. (Note the for invoices this view is not particularly helpful though.)

The “Show All” view can be achieved by (miss-)using a lookup list field. A lookup list field highlights all its subfields on the document. The approach we take is this:

1. Create a lookup list field with enough subfields, i.e. as many as there are other fields and subfields in the document. The lookup list needs a database file, a dummy file with no data and one column will suffice. You can map all sub fields of the lookup list to the very same column. You should make all subfields invisible.
2. You call “ShowAllFields“ in the Data Extraction scripting point:

ScriptingUtilities.ShowAllFields(data, "ShowAll");

The parameter (“ShowAll”) is the name of the above mentioned lookup list field. The function will copy all zone from all fields to the lookup list. Some fields will have subfields, in the sample profile is an US\_InvoiceTotals field. Each of its seven subfields will show up in the lookup list field if it is not empty

1. This is optional. In Validation -> Document loaded you can set the lookup list to error state. That way the user will stop right there. Actually you need to set a subfield in the lookup list to state error. In the sample profile the first lookup list sub field was named “Document”.

helper.SetError(

doc.Fields["ShowAll"].Fields["Document"], "Verify all fields");

1. This is optional too. It may be nice, if the error on the field disappears as soon as the user leaves the field. You achieve this by the below script in Validation -> Field deactivated.

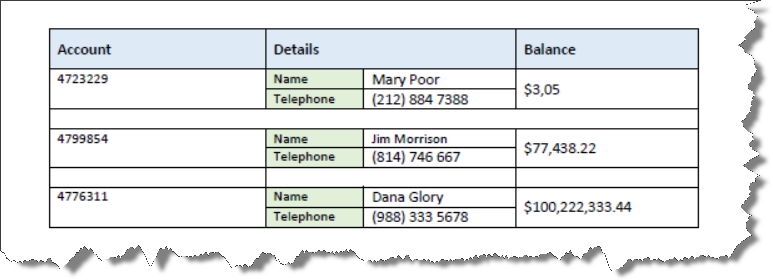
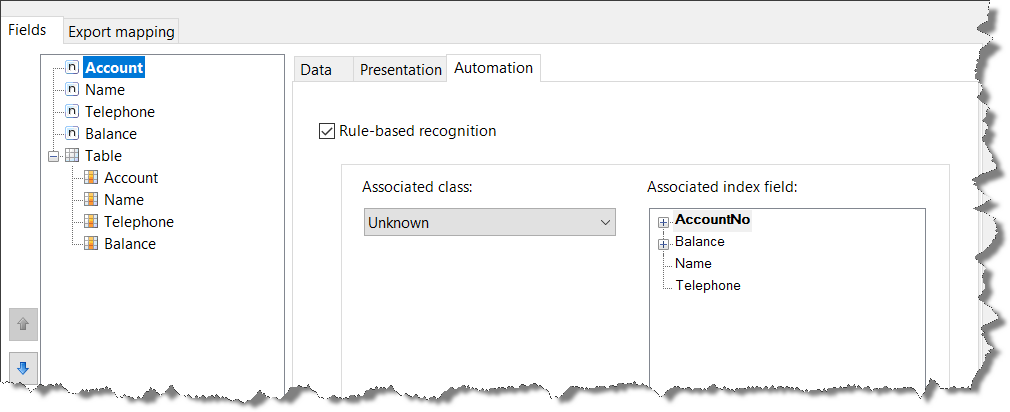
if (field.Name == "Document")

field.State = DataState.Ok;

**Resources**

|  |  |
| --- | --- |
| Profile | ShowAllFields.zip |
| Sample document | Invoice.pdf |
| DLLs used | ScriptingUtilities.DLL (Additional DLL) |
| Other | DummyLookupList.txt |
| Conditional compilation | None |

# Extract table column-wise

Sometimes the table operator in the rules engine (Capture Document Extraction “DOKuStar”) is not suited to extract tabular data. This is basically if the data is not arranged as a table. In the sample shown here for example has two columns condensed in one layout column. An approach that sometimes can be taken is to use separate fields for each column. In the rules engine definition for this profile there is one field for each column that has set the “Occurrences all” option set. That way each of these fields holds the values for one column. It uses key-value fields within key-value fields, something you might not have been aware of.

There is a table field defined that has no automation method assigned. Scripting code is used to assign the content of the automated fields into the table column. The assumption is that the name of the automated fields is identical to the column names in the table. In order to trigger the copying, one needs to call the below scripting function in the DataExtraction scripting point.

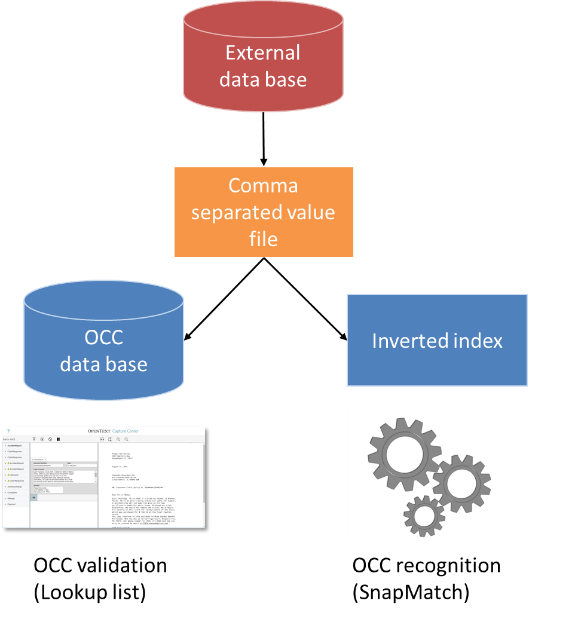
ScriptingUtilities.TableColumnWise(data, "Table");

“Table” is the name of the table field.

**Resources**

|  |  |
| --- | --- |
| Profile | TableColumnWise.zip |
| Sample document | TableColumnWise.pdf |
| DLLs used | ScriptingUtilities.DLL (Additional DLL)  ScriptingUtilitiesProfileExtension.DLL (Custom DLL)[[8]](#footnote-8) |
| Conditional compilation | None |

# Lookup list updater

OCC’s lookup list field requires to load a data base table as comma separated value file (CVS). Typically, there is an external data source which holds the data. This data source has to be converted into a CVS file and loaded into OCC. OCC creates a data base table in its own data base from it and create a so-called inverted index that is used the SnapMatch algorithm during the recognition process.

Creating the inverted index is a computational involved process and needs to happen before the recognitions starts. The SnapMatch algorithm cannot access an external data base directly. This is the main reason no life data is used.

In order to get close to life data, the CVS file should be loaded regularly e.g. daily into OCC. A windows command file is available to achieve this.

The lookup list update tool helps to automate the whole process under the assumption that the external data base is a Microsoft SQL data base. It consists of two components.

* **The LookupListUpdater tool.** This Windows program allows you to specify one or several data base tables that should be used by OCC. These definitions are called projects and are stored as a XML files. The tool also allows you to export the source table as CVS file for import into OCC, e.g. when you create your OCC profile initially. And it allows you to update the data base in an OCC profile.
* **A HotSpot connector**. This connector has just two parameters, the project definition as created by the LookupListUpdater and the profile name that should be updated. This hotspot will not import any documents into OCC but will update the data base.

How to use

1. Start the LookupListUpdater tool, set the definitions for the source data base table and press “Store data table as CSV file”.
2. Use the created file to define the lookup list in OCC.
3. Set “OCC profile” and “OCC table name” and press “Update OCC profile”; this should verify that updating the OCC profile works.
4. Store the project as XML file.
5. Create a HotSpot and use the project file as parameter.

Important note:

The LookupListUpdater tool should be copied into the OCC installation folder and executed from there. If – for debugging purposes for example – you want to execute it from another location, you need to set the OCC installation folder in the Configuration section of the toolbar.

**Resources**

|  |  |
| --- | --- |
| DLLs used | DOKuStar.Runtime.HotFolders.Connectors.LookupListUpdateConnectors.dll  LookupListUpdaterModel.dll |
| Conditional compilation | None |

The two DLLs need to be copied into the OCC installation folder.

# Extract Image Metadata

Images taken by a camera or a mobile phone contain plenty of metadata, like the exposure time, the time the picture was taken, or in some cases the GPS data from where the picture was taken. In some uses one may need to use these data as index data.

Extraction those metadata is easy using a proper library. This sample uses a library called metadata-extraction from Drewnoakes. You may download from their website (<https://www.drewnoakes.com/code/exif>), or from Github (<https://github.com/drewnoakes/metadata-extractor>). For Visual Studio users it is also available as Nuget package. In order to use the sample need to include the package in the solution and compile the ScriptingUtilities solution with the compile switch “Drewnoakes”.

Image metadata is organized in groups, e.g. “Exif SubIFD" for camera data and tag like "Exposure Time" that contain the values. There are websites available that allow you to analyze a given image e.g. <http://exifdata.com/exif.php>.

The sample here makes several assumptions. First of all, it assumes that the file loaded into OCC is an image file and it also assumes there is just one images file associated with each document. If OCC would combine several input images into one document, it would be unclear from which of them one wants to extract the metadata. (Actually, the implementation will pick the first of the image sources.

The below script in the “Data Extraction” step shows how to use the function. It extracts the exposure time for each document.

foreach (Document doc in data.RootNode.Documents)

{

doc.Fields["ExposureTime"].Value =

ScriptingUtilities.GetImageMetaData(doc, "Exif SubIFD", "Exposure Time");

}

**Resources**

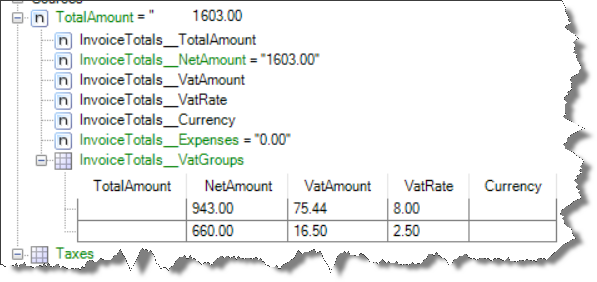
|  |  |
| --- | --- |
| Profile | ImageMetaData.zip |
| Sample document | ExifData.JPG |
| DLLs used | ScriptingUtilities.DLL (Additional DLL)  MetadataExtractor.dll (Additional DLL)  XmpCore.dll (Additional DLL) |
| Conditional compilation | Drewnoakes |

# Invoice Totals Field Handling

The Invoice\_Totals operator in OCC’s rules engine (“DOKuStar”) extracts quite a bunch of data, like total amount, net amount vat amount and so forth. Some invoices have several value added tax positions like below. These are also extracted by the operator.



Because of these many data the result of an Invoice\_Totals operator is quite complex. Is contains seven subfields where one of them actually is a table of vat records. Below such a result is shown from OCC’s data viewer.

s

The below script in the “Data Extraction” step shows how to use the functions. GetInvoiceTotalsField copies one subfield (third parameter) of the result to the OCC field given as fourth parameter. If the OCC field name is omitted it is assumed to be the same as the subfield name.

GetVatRates copies the Vat section of the result into an OCC table. The parameters of the function are the name of the result field for the InvoiceTotals field, the name of the table in OCC and a mapping list that maps column names between the result and the table. Not mentioned columns are expected to be identical.

ScriptingUtilities.GetInvoiceTotalsField(data, "InvoiceTotals", "TotalAmount", "Total");

ScriptingUtilities.GetInvoiceTotalsField(data, "InvoiceTotals", "NetAmount", "Net");

ScriptingUtilities.GetInvoiceTotalsField(data, "InvoiceTotals", "VatAmount", "Vat");

ScriptingUtilities.GetInvoiceTotalsField(data, "InvoiceTotals", "VatRate");

ScriptingUtilities.GetInvoiceTotalsField(data, "InvoiceTotals", "Expenses");

ScriptingUtilities.GetInvoiceTotalsField(data, "InvoiceTotals", "Currency");

Dictionary<string,string> map = new Dictionary<string, string>() {

{ "TotalAmount","Total" },{"NetAmount","Net" },{"VatAmount","Vat" } };

ScriptingUtilities.GetVatRates(data, "InvoiceTotals", "Taxes", map);

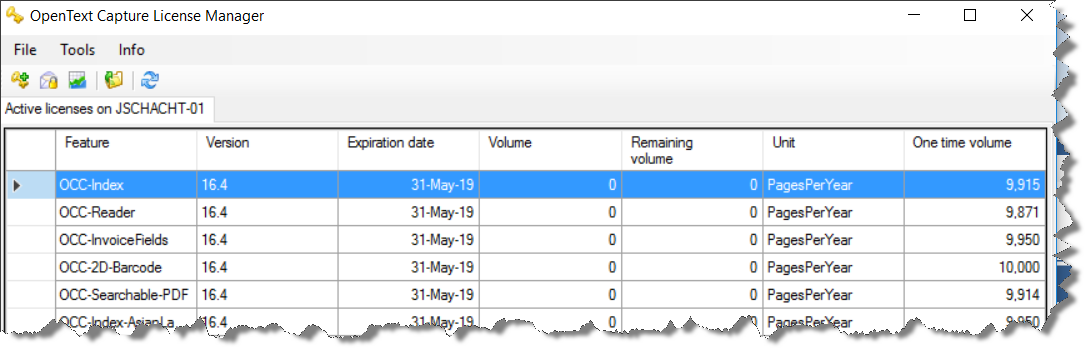
**Implementation note**: Basically the functions runs as an additional DLL (“ScriptingUtitlities.DLL”). However, this function needs to add rows to the OCC table and this in turns needs to have the schema attached to the data pool. The schema file in turn can only be determined by a profile extension DLL. Therefore ScriptingUtitlitiesProfileExtention.DLL is also needed in the profile. It writes the name of the schema into an annotation in the PrepareImport step.

**Resources**

|  |  |
| --- | --- |
| Profile | InvoiceTotals.zip |
| Sample document | InvoiceTotals.PDF |
| DLLs used | ScriptingUtilities.DLL (Additional DLL **and** Custom DLL)[[9]](#footnote-9)  ScriptingUtilitiesProfileExtension.dll (Custom DLL) |

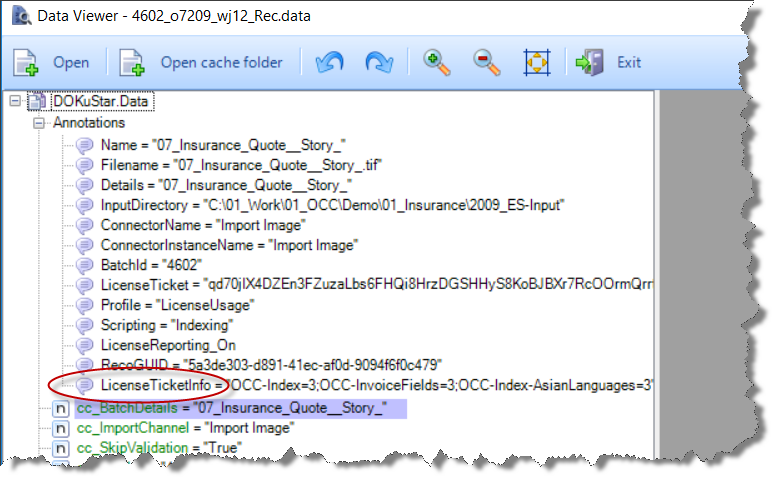
# License Counting per Profile

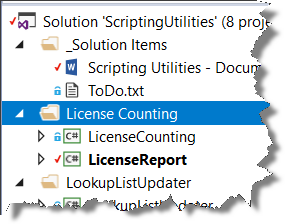
License are maintained by OCC’s license manager. There one can see how many licenses have been used of which type.



However, this report does not show, how many licenses are used for which profile. The projects under “License Counting” are doing that.

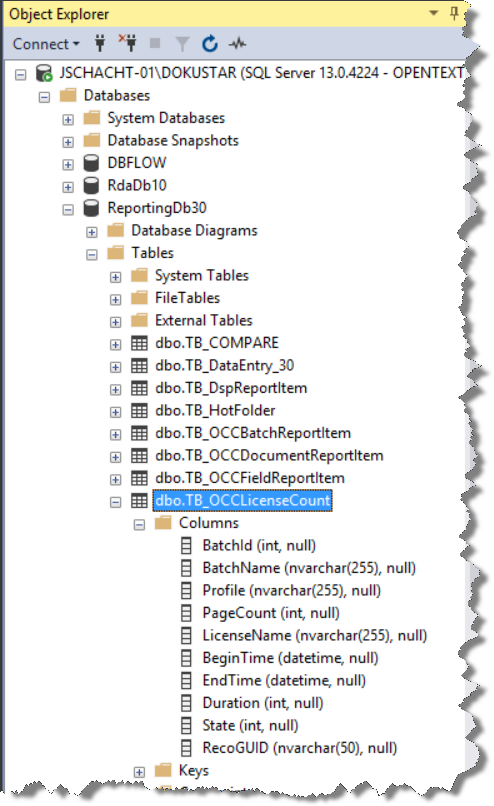
The example depends on a feature of OCC that will become available with Version 16.4.6 in April 2019. As of this writing the feature is not generally available. With that feature, OCC recognition adds an annotation to the RootNode of the data pool that contains the licenses used.



There are two projects for License Counting:

* LicenseCounting is a profile extension DLL that writes the information from the LicenseTicketInfo a reporting database
* LicenseReport creates a report as Excel sheet

The solution assumes that there is a certain table in OCC’s reporting database. You can create that table either manually or use the SQL statement that is embedded in the code (LicenseCountingProfileExtension.cs, region “Table definition”). This is the table:

Ideally, the profile extension would store the license information into the database during the recognition step, but unluckily the annotation is only set after the latest scripting point. Therefore, license information is stored during the export step.

Due to validation or export retries the licenses will potentially be stored multiple times. Therefore, the profile extension adds a GUID to each data pool in the recognition step and will store that GUID along the license information itself.

If the batch is reset to recognition a new GUID is set which is correct as in this case license are also counted twice.

**Resources**

|  |  |
| --- | --- |
| Profile | Any |
| Sample document | Any |
| DLLs used | LicenseCountingProfileExtension.dll (Custom DLL) |
| Programs | LicenseTool.exe |

# Save Table as CSV file

OCC’s file system export allows to store documents as PDF or Tiff and data as XML or CSV. There are situations where you may want to store other information. This example writes a special CSV file with tabular data. (The default CSV file does not contain tabular data.)

The function “StoreTableAsCVS” is called in the AfterExport scripting point.

string tableFieldName = "Items";

List<string> columnNames = new List<string>()

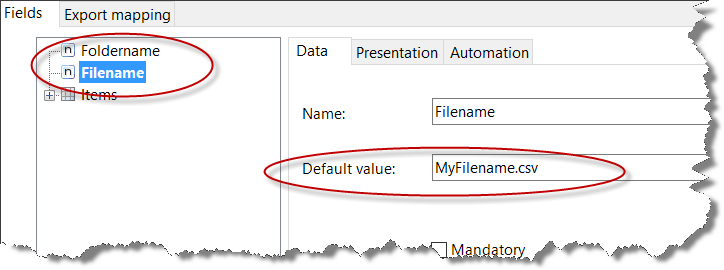
{"Qty", "UoM", "Description", "Price", "Total"};

ScriptingUtilities.StoreTableAsCVS(data,

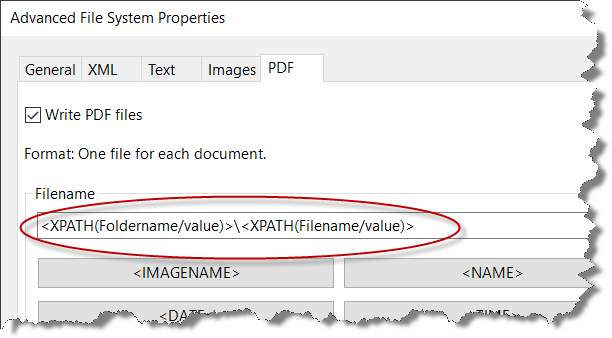
"Foldername", "Filename", tableFieldName, columnNames);

Care must be taken to write into the same folder as the document and maybe other files have been written by the file system export. The base location that has been configured by the file system export is available through the annotation “FileOutputPath” when you are in the AfterExport event.

In the example, the target folder and the target file name are taken from two OCC fields: Foldername and Filename. They have static values but could of course be set during extraction to implement dynamic folder and file name determination.



In the export extension the fields are configured using an XPath.



**Resources**

|  |  |
| --- | --- |
| Profile | SaveTableAsCSV.zip |
| Sample document | Invoice.PDF |
| DLLs used | ScriptingUtilities.DLL (Additional DLL)  Namespace: System.IO, System.Collections.Generic |

# Use multiple profile specific custom extensions

Profile specific custom extension have almost only benefits compared to global custom extension. There is one deficiency though: You can only add one custom extension DLL to a profile. However, there are situation you have several custom extension DLLs that you want to combine. For example, you may have developed a custom extension DLL and the profile already has a DLL installed or you want to work as a team developing functionality independently.

In all these cases you would like to combine existing custom extensions into one. This chapter proposes a solution to this problem. Basically, you create another customer extension that allows to easily combine several other custom extensions. This “dispatcher custom extension” allows other extensions to register. If OCC calls out to the profile specific custom DLL the dispatcher DLL will call call all registered custom DLL’s function sequentially. For validation functions the dispatcher also takes care of the “handled” parameter by aborting the sequence if needed. It also takes care of return codes if the call back function has one.

The dispatcher custom extension is work in progress. Not all OCC call back functions are implemented. If you miss a function you can easily implement the missing function by copying an modifying an existing one.

In order to use the dispatching functionality, you need to create your custom extension DLLs and an additional dispatching DLL. In the folder ScriptingUtilities\CustomExtensionDispatcher you find examples for that:

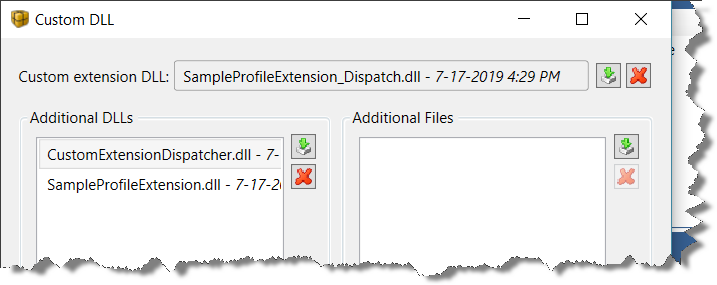
* SampleProfileExtension contains to custom extensions (Extension1 and Extension2)
* SampleProfileExtension\_Dispatch contains the corresponding dispatcher

The profile “CustomExtensionDispatcherTest” uses the extensions to achieve the following behavior. Extension1 sets field F1 during the indexing step and Extension2 sets field F2. During validation Extension1 sets the content of a validation field to upper case characters and Extension2 appends the number of characters in a string at the end of the string. So, after recognition field F1 contains “Hello”, after leaving the field without value change it will become “HELLO\_5”.

The dispatcher itself is implemented in CustomExtensionDispatcher.DLL which exposes two classes:

* CustomExtensionDispatcher is the base class for the dispatcher (e.g. SampleProfileExtension\_Dispatch)
* ProfileExtension\_Transform is a utility class, SampleProfileExtension is derived from that class, more on that later. But note that SampleProfileExtension could have been equally well derived directly from BaseCustomProfileExtension.

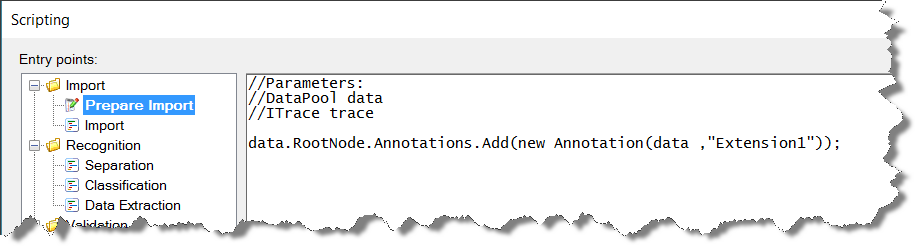
The below screen shot shows, how the DLLs are added in the profile.



The SampleProfileExtension\_Dispatch.cs is straight forward. It registers the two custom extensions



The registration call takes two parameters: The type of the extension class and the name of an annotation that enables the extension. The second parameter is optional and if omitted, the extension will always be active. To activate an extension, one has to add the annotation in the batch at the Prepare Import step as you can see below. Please note, that the mainRecognitionExceptionNamemethod in transform will always be called as the annotations are not available there.



ProfileExtension\_Transform is a utility class that “spells out” the transform method. Instead of figuring out which processing step you are called for, you only implement the target function. You can see the use in the sample profile extensions.

**Resources**

|  |  |
| --- | --- |
| Required version | OCC 16.4.1 Patch 1 |
| Profile | CustomExtensionDispatcherTest.zip |
| Sample document | any |
| DLLs used | SampleProfileExtension\_Dispatch.dll (Custom extension DLL)  CustomExtensionDispatcher.dll (Additional DLL)  SampleProfileExtension.dll (Additional DLL) |

# Prevent broken batches

Broken batches are a nuisance, especially when you want to run OCC basically unattended. This chapter proposes a solution that avoids broken batches for most cases.

Batches may go broken for a number of reasons: Exhaustive processing time (= timeout), scripting errors, product errors or non-sensible input (e.g. a zip file or an executable) . OCC makes a number of retries for broken batch under the assumption that a new or another cluster node may succeed, e.g. if some resource shortage caused the error, or if a required resource (scripting, network) was temporarily not available.

The proposed solution lets the batch run through for the first time unchanged. If the batch shows up a second time in recognition, it is known that the batch has failed once already. In this case the solution does different thing depending on whether document splitting was successful.

Splitting is a processing step in OCC that happens between the PrepareImport scripting point and the Import scripting point. Thus, when the Import scripting point is reached, splitting was successful. Splitting takes each input document and splits it into single page TIFFs. Once that is done, OCC can validate the batch and export the batch. Thus, even if recognition fails, the rest of the process will still work.

Therefore: If splitting fails, the proposed solution will divert the batch to some external (with respect to OCC) exception processing. If splitting succeeds the second time the batch enters recognition it the solution will set all pages to “SkipOCR”, thus trying to bring the batch to manual validation without any data. If even that fails, the solution will also divert the batch to some external exception processing.

So: Each batch will either run through normally, run through without OCR results (manual processing) or will be diverted to external exception processing. A broken batch in recognition will be avoided altogether.

In order for all this to happen a batch history must be maintained. The solution stores the batch history in a file in the repository alongside all other batch documents (images, results etc.).

The sample profile “Exception.zip” will separate the three-page input document into two documents and extract some value from the second document. If all that succeeds an artificial exception is raised (test-exception 1).

if (data.RootNode.Documents.Count == 2 && data.RootNode.Documents[1].Fields["F1"].Value == "Page 3")

// Separation and Classification worked --> Simuulate error

throw new Exception("test-exception 1");

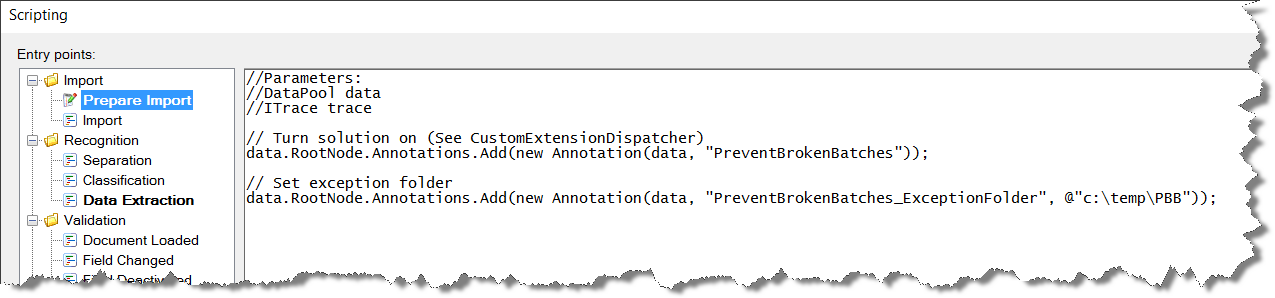
else

// Separation and Classification failed because no OCR results created --> Simulate another error

throw new Exception("test-exception 2");

If you either wait 10 minutes for OCC’s auto-retry or reset the batch to recognition manually, it will pass recognition without OCR, thus no separation, no classification, no data extraction. In this case the above script in the Indexing scripting step will raise “test-exception 2”.

On the next retry, the batch will be diverted to external exception processing. This is: The solution will store all batch data in a sub-folder named after the batch-id. The base location can be set as an annotation to the batch.

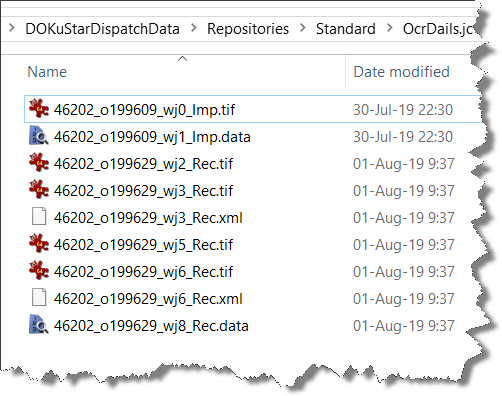


**Resources**

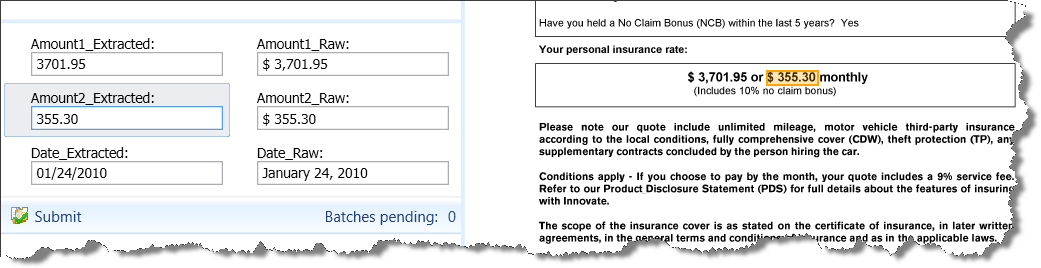
|  |  |
| --- | --- |
| Required version | OCC 16.4.1 Patch 1 |
| Profile | Exception.zip |
| Sample document | 3Pages.PDF |
| DLLs used | PreventBrokenBatchesProfileExtension\_Dispatch.dll (Custom extension DLL)  PreventBrokenBatchesProfileExtension.dll (Additional DLL)  CustomExtensionDispatcher.dll (Additional DLL) |

# Get OCR details

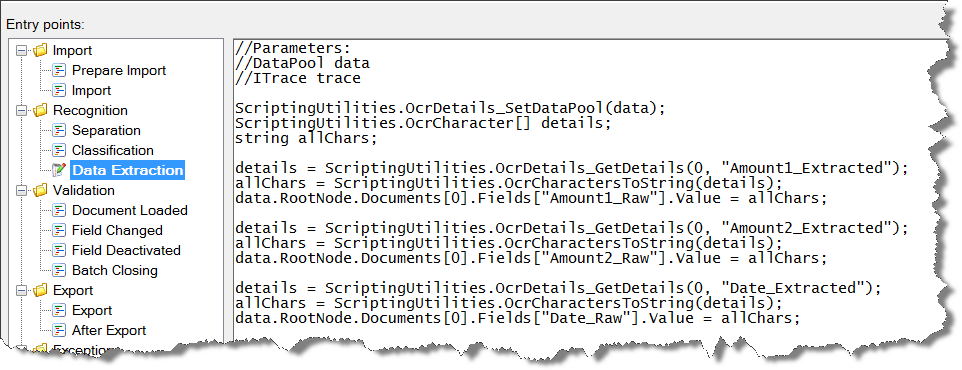
In general, OCC extracts data from documents, but in particular it provides strings and the bounding boxes on one of the pages of a document. But it is not even said that the strings are on the document. If “January 24, 2010” is written on the document the result may be “01/24/2010”. Where is the text that has been on the document?

One step of the extraction process is OCR proper, it converts the pixels of the input images into characters[[10]](#footnote-10),[[11]](#footnote-11). The results of the OCR step are stored in one file for each page. The screen shot shows the repository content for sample document (two pages). The two xml files contain the OCR data for the respective pages.

The sample profile reads three pieces of data from the document (two amounts and one date) and stores the associated OCR strings in three other fields (“xy\_Raw”). The below screen shot shows the result in the validation client.



Here is the scripting code used to make things happen:



Obviously, all the magic happens in a few functions in the ScriptingUtilities.DLL.

* OcrDetails\_SetDataPool(data): This function is the required initialization. It needs to be called before any of the other functions.
* OcrDetails\_GetDetails(0, "Amount1\_Extracted"): This function extracts the OCR details for the zone of a given field (second parameter) of a given document (first parameter) of the result. It actually reads all OCR character in the OCR result file and figures out whether this character falls into the bounding box of the zone of the OCC result.
* OcrCharactersToString(details): The return value from OcrDetails\_GetDetails is an array of OcrCharacter. Each element describes one character by its bounding box and one or more “choices”, each consisting of a character itself and the character confidence. This function reduces the whole set into a simple string. It also calculates whether a space needs to be inserted between two characters, using a simple heuristic.

Using the character details many more functions can be implemented. For this you would probably extend the functions in the ScriptingUtilities DLL.

**Resources**

|  |  |
| --- | --- |
| Profile | OcrDetails.zip |
| Sample document | 07\_Insurance\_Quote\_\_Story\_.tif |
| DLLs used | ScriptingUtlilities.dll (Additional DLL) |

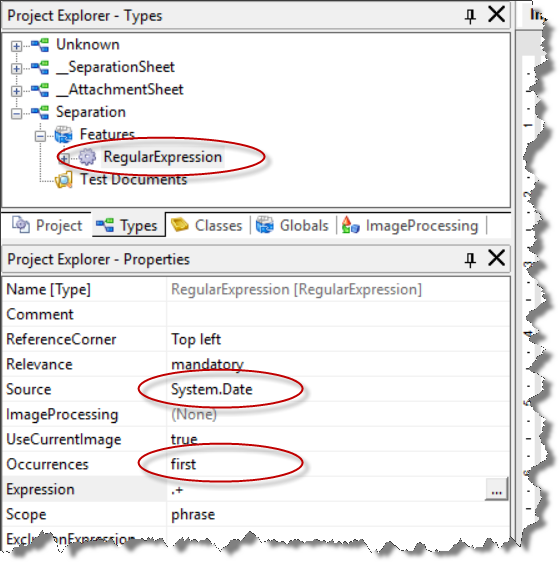
# Separate on Value Changes

OCC’s rule-based separation allows to start a new document whenever the page type changes as determined by a separation rule. This works fine if the first page of a document has unique features. However, there are situations where this scheme is not sufficient. A typical example is where a document can only be determined by a common feature on each page of the document.

In the document below each page (except for two) has a barcode. Pages with the same barcode belong together. The two pages without barcode can be seen as pages where the barcode cannot be recognized. They belong to the “current” document. The horizontal lines mark, the three documents. 

The solution implements a separation rule via scripting that splits a stream of pages whenever the value of a certain field changes its value. It can be easily extended to implement more complex rules.

The solution has a rule-based separation that does two things:

* The incoming documents are split such that each page is an individual document with one page only.

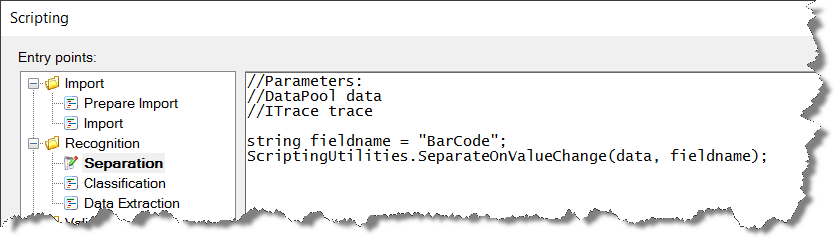
(Classifying each page as being a certain document type uses a common trick. A feature of type RegularExpression is added. Source for that field “System.Date”, which is some multi-character string. The regular expression is just “.+” meaning “accept anything”. The result is a feature that is true on each page.)

* The barcode is extracted into a field named “BarCode”.

The result are seven documents each having one field “BarCode” with these values:

* Document One
* Document One
* Document Two
* <Empty>
* Document Two
* Document Three
* <Empty>

In the document scripting point “Separation” the function **ScriptingUtilities.SeparateOnValueChange(data,"BarCode")** is called. This function looks at each page in turn and whenever the value of the given field changes, a new document is started, treating an empty field as a continuation of the current document.



**Resources**

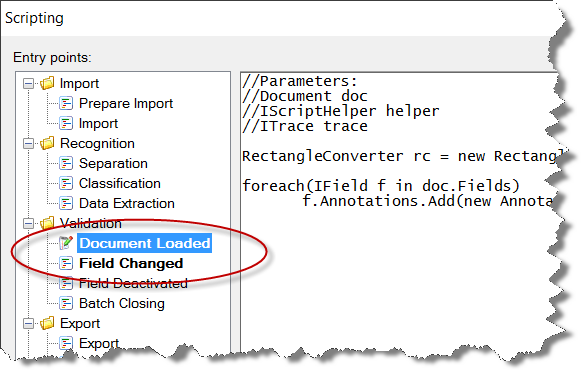
|  |  |
| --- | --- |
| Profile | Separation\_by\_Barcode.zip |
| Sample document | Separation by Barcode.pdf |
| DLLs used | ScriptingUtlilities.dll (Additional DLL) |

# Maintaining Zones in Validation for Forms Applications

There is a difference between forms applications and free forms application, in the way they should handle zones in validation.

* Free Forms Application: When the field is cleared (e.g. because data was wrong) the zone should be deactivated too. The new value is probably coming from a different area on the page. With Single Click Entry this happens smoothly.
* Forms Application: The zone is sticky. If the field is emptied, still the zone should be displayed, because it is the zone where the data has to come from.

OCC’s validation has been designed for free forms applications. If you have a forms application and you want a sticky zone in validation, you can achieve that by two simple scripts that you can put into the Scripting editor.



In the Document Loaded event we store all zones in an annotation at each field.

RectangleConverter rc = new RectangleConverter();

foreach(IField f in doc.Fields)

f.Annotations.Add(new Annotation(doc.OwnerDataPool, "ZoneBackup", rc.ConvertToString(f.Zone)));

In the Field Changed event we forcibly set the zone back to its original value. We have added a check as to whether the annotation exists. By not setting the annotation in Document Loaded on selected fields one could customize the form behavior for certain fields.

RectangleConverter rc = new RectangleConverter();

Annotation a = field.Annotations["ZoneBackup"];

if(a != null)

field.Zone = (Rectangle)rc.ConvertFromString(a.Value);

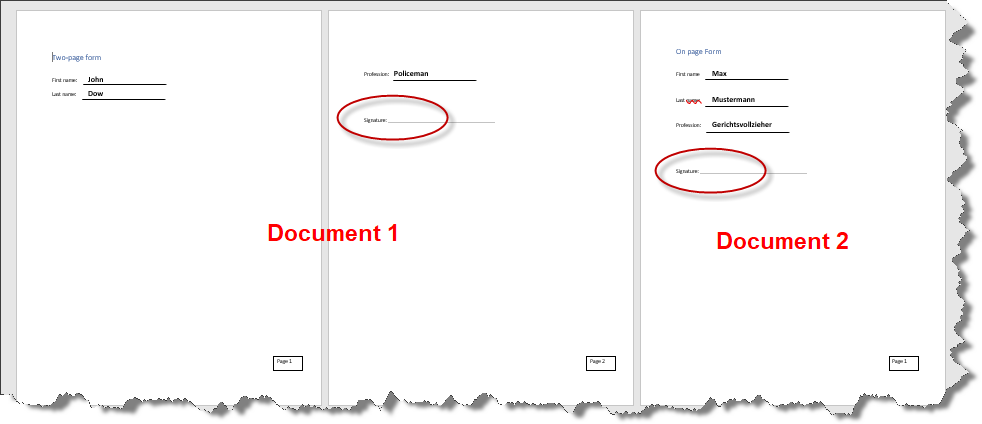
return false;

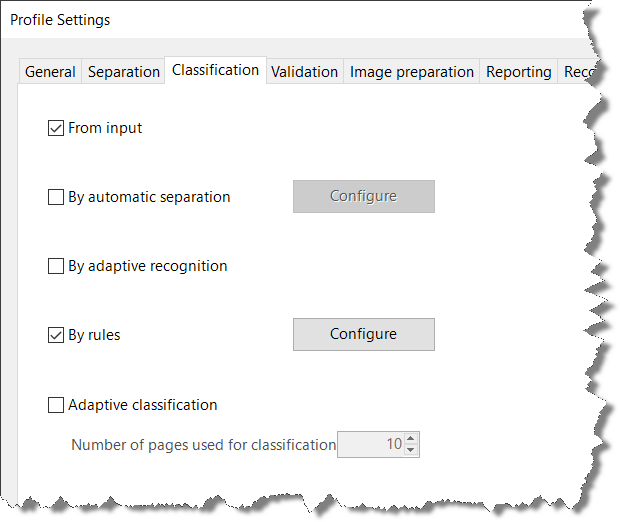
**Resources**

|  |  |
| --- | --- |
| Profile | FormsAppAndZones.zip |
| Sample document | 07\_Insurance\_Quote\_\_Story\_.tif |
| DLLs used | None, scripting code only |

# Classify with rules on second page

Rules-based classification in OCC will only consider the first page of a document. If the deciding feature that identifies a document types is on the second or third page rule based classification will fail. It’s not difficult to consider the other pages too using scripting code. This is shown in the example.

The sample input file contains two documents, the first two pages are document one and the third page is document two. I rule-based classification we define a rule that classifies a page as “Signature” if the word “Signature” is found on the page. Pages 2 and 3 contain that word.

When classified, the first document will be unknown[[12]](#footnote-12), the second will be classified correctly.

In order to understand the scripting code, one should know how OCC’s classification works. OCC has several classification algorithms, e.g. ACT, ART or by rule. Each of these algorithms, if activated, will provide a proposal. In a final step, these proposals are consolidated to a final result. Basically, the order in the classification determines the priority, i.e. “From input” takes precedence over all other options.

After all algorithm and the final vote scripting is called. The scripting code can examine all individual results and over-rule the final decision.

In our example the “By rules” sub-result is wrong for the first document. Its state is “Error” and its value is “Unknown”. However, as all pages are classified by the rules engine, there are alternatives to the rules result.



The below code in the *Classification* scripting point will loop through all alternatives from the rules-based classification and will make the first valid one the final classification result.

foreach (Document doc in data.RootNode.Documents)

{

IField ruleResult = doc.GetClassificationResultRules();

if (ruleResult.State == DataState.Error)

foreach (IDataNode node in ruleResult.Alternatives)

{

Field f = (Field)node;

if (f.State == DataState.Ok)

{

doc.SetClassDecisionField(f.Value);

break;

}

}

}

**Resources**

|  |  |
| --- | --- |
| Profile | FeatureOnSecondPage.zip |
| Sample document | FeatureOnSecondPage.pdf |
| DLLs used | Not needed. (The sample code is also in the ScriptingUtilitiesProfileExtension and can be used from there.) |

1. Actually you can provide a regular expression instead of a field name, in which case all fields matching that name are redacted; a field name happens to be a regular expression [↑](#footnote-ref-1)
2. ScriptingUtilitiesProfileExtension.dll also needs ScriptingUtilities.dll as „additional DLL“. It will also need System.Collections.Generic mentioned in the Namespaces. [↑](#footnote-ref-2)
3. ScriptingUtilitiesProfileExtension.dll also needs ScriptingUtilities.dll as „additional DLL“. [↑](#footnote-ref-3)
4. Actually, EmailToPdf.cf contains two implementations: The real one and another that simply creates an arbitrary PDF and can be used as test case. [↑](#footnote-ref-4)
5. Often this will be just one server. However, if you run multiple OCC recognition servers you may want to install Luratech’s software on one server only. In this case you need to configure the recognition cluster such that export processes run on this one server. See “Defining exclusive Load Manager operation types for a runtime node” in the administrator’s guide for details. [↑](#footnote-ref-5)
6. If you do not want to compress all files, you may mark only certain documents by implementing some code that choses some of the documents in the batch and adds the annotation to them. [↑](#footnote-ref-6)
7. ScriptingUtilitiesProfileExtension.dll also needs ScriptingUtilities.dll as „additional DLL“. It will also need System.Collections.Generic mentioned in the Namespaces. [↑](#footnote-ref-7)
8. ScriptingUtilitiesProfileExtension.dll also needs ScriptingUtilities.dll as „additional DLL“. [↑](#footnote-ref-8)
9. Also needs System.Collections.Generic in the Namespace section of the Scripting editor. [↑](#footnote-ref-9)
10. Even for non-image documents, e.g. PDFs OCR details will be generated. [↑](#footnote-ref-10)
11. Full page OCR is an option, it may be turned off; but as default it is on. [↑](#footnote-ref-11)
12. Please note that when opened in validation the first document will still be shown as “Form” because that is the default in this profile. The attached warning symbol indicates that the classification has failed. [↑](#footnote-ref-12)