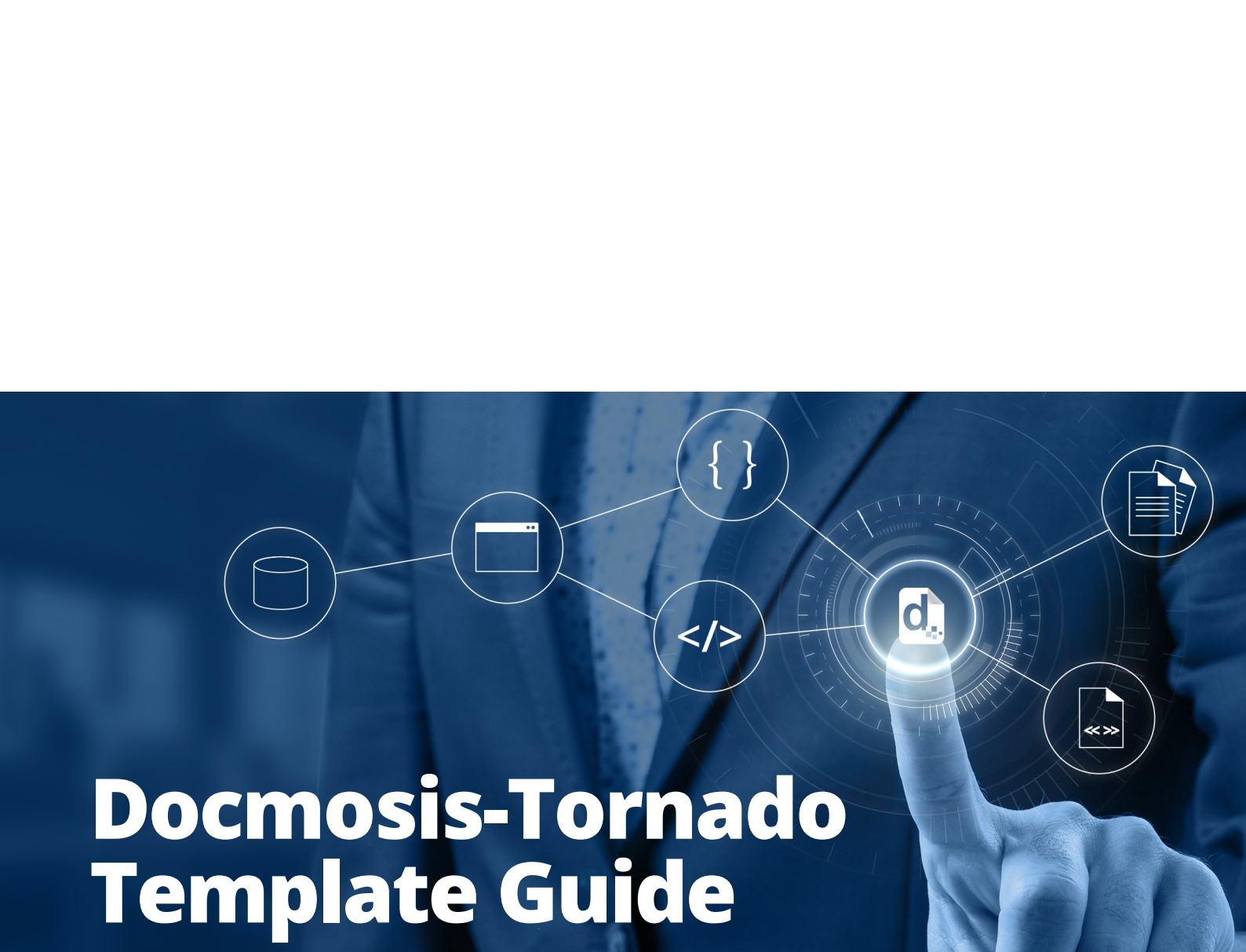




Docmosis-Tornado Template Guide

Version 2.8
Dec 2020



Template Guide

Copyrights

© 2020 Docmosis Pty Ltd

This document and all human-readable contents of the Docmosis distribution are the copyright of Docmosis Pty Ltd. You may not reproduce or distribute any of this material without the written permission of Docmosis.

<https://www.docmosis.com>

The placeholder image provided in the Docmosis distribution is intended for use in document templates and is not restricted by the terms above. You may use the image for the creation of document templates and distribute it as required.

Trademarks

Microsoft Word and Microsoft Windows are registered trademarks of the Microsoft Corporation.

<http://office.microsoft.com/en-us/default.aspx>

<http://www.microsoft.com/windows/>

Adobe® PDF is a trademark of the Adobe Corporation.

<http://www.adobe.com/products/acrobat/adobepdf.html>

LibreOffice Writer is a trademark of LibreOffice contributors and/or their affiliates

<http://www.libreoffice.org>

Table of Contents

1 Introduction.....	6
1.1 Using this Guide.....	6
1.1.1 Terminology and Conventions Used in this Document.....	6
1.2 Troubleshooting.....	7
2 Templates Overview.....	8
2.1 Separating Content from Presentation.....	8
2.2 What Are Templates?.....	8
2.3 How Does Document Generation Work?.....	9
2.4 Template Features.....	9
2.4.1 General Features.....	10
2.4.2 Advanced Features.....	10
2.4.3 Docmosis Elements.....	10
2.4.4 Expressions and Functions.....	16
2.5 Error Handling.....	36
3 Creating Docmosis Templates.....	37
3.1 Incorporating Docmosis Elements.....	37
3.2 Using Plain Text Fields.....	38
3.3 Using the Built-in Word Processor Fields.....	38
3.3.1 Creating a Field Using Microsoft Word Merge Fields.....	39
3.3.2 Inserting a Field Using LibreOffice Writer Input Fields.....	39
3.4 Using Text Substitution.....	40
3.4.1 Optional Paragraph Fields.....	40
3.4.2 HTML.....	42
3.5 Using Docmosis Variables.....	42
3.6 Using Images.....	43
3.7 Creating Barcodes.....	48
3.7.1 Supported Barcode Formats.....	48
3.7.2 Typical Barcode Example.....	48
3.7.3 Using a “barcode” Field to Specify Barcode Settings.....	49
3.7.4 Barcode Tips.....	50
3.7.5 Barcode Settings in Detail.....	51

3.8 Creating Active Hyperlinks.....	53
3.9 Using Conditional Sections.....	54
3.10 Repeating Sections.....	55
3.10.1 Stepping Across in Repeating Sections.....	57
3.10.2 Stepping Down in Repeating Sections.....	59
3.11 Using Tables.....	60
3.11.1 Conditional Rows.....	60
3.11.2 Repeating Rows.....	61
3.11.3 Stepping in Repeating Rows.....	62
3.11.4 Alternating Row Colours and Border Controls.....	62
3.11.5 Disabling Row Alternating.....	63
3.11.6 Conditional Columns.....	64
3.11.7 Advanced Table Structures.....	66
3.12 Using Lists.....	67
3.13 Using Headers and Footers.....	68
3.14 Using Comments in Templates.....	70
3.15 Merging Templates Together.....	70
3.15.1 Direct Referencing.....	71
3.15.2 Indirect Referencing.....	71
3.15.3 Templates in Different Locations.....	72
3.15.4 When a Template Cannot be Found.....	73
3.15.5 Continuing Numbered Lists Across Templates.....	73
3.15.6 Limitations.....	75
3.15.7 Page Breaks and Other Breaks.....	75
3.16 Creating Pre-filled PDF Forms.....	76
3.16.1 Adding a Text Field.....	77
3.16.2 Adding a Checkbox.....	78
4 Formatting Data.....	80
4.1 Formatting Numbers.....	81
4.1.1 The Number Formatting String.....	81
4.1.2 Locale-Specific Formatting.....	82
4.1.3 Using a Number Renderer.....	84
4.2 Formatting Dates.....	84
4.2.1 The Date Formatting String.....	85
4.2.2 Using a Date Renderer.....	87
4.3 Formatting Booleans.....	88
4.3.1 Built-in Boolean Renderer.....	88
4.3.2 Windings Boolean Renderer.....	89

5 Appendices.....91

Appendix 1 - Number Formatting Codes.....91

Appendix 2 – Date and Number Formatting Locales.....92

Appendix 3 - Date Formatting Codes.....95



1 Introduction

Welcome to the Docmosis Template Guide. This manual is intended for template authors who will create the richly formatted template layouts using the special Docmosis template syntax.

Docmosis is an easy-to-use document generation engine. It integrates with other software to provide the ability to generate documents and reports by merging data supplied from the software with the templates.

1.1 Using this Guide

This Template Guide provides information on creating the templates (in either Microsoft Word or LibreOffice Writer) that will be used to generate the documents. This guide assumes a level of competence in using one of those word processors and is not a complete reference manual for either.



In general, the activities to create the templates are the same for both tools, but where there are differences between the two, this document highlights them and describes the activities for each word processor.

1.1.1 Terminology and Conventions Used in this Document

This document uses typographical conventions that highlight significant parts of the text to distinguish it from normal text.

Text that looks like this...	Means this...
<<fieldname>>	A field in the document template that will be replaced with data.
template.doc	A file name, a file extension or a web site address.

Additionally, some parts of the document are written specifically for one of the word processors mentioned. When this is the case, the paragraph has the respective icon in the left margin.

This icon...	Means this...
	The information applies to LibreOffice Writer only.



This icon...	Means this...
	The information applies to Microsoft Word only.

The following terms are used in this document to identify elements of a template.

Term	Description
Field	A placeholder that is used by Docmosis to substitute data or to control document flow.
Boilerplate	Graphical and textual content that is added to a template as reusable content to avoid having the template author recreate the content for each document. Docmosis uses boilerplate components.
Header and footer	Elements of a printed document that repeat on every page. Information in these elements is usually administrative information about the document.

1.2 Troubleshooting

The FAQ section of the Docmosis Resources website (<https://resources.docmosis.com>) may help with troubleshooting problems within your templates.



2 Templates Overview

This chapter provides information about the main features of a Docmosis template.

2.1 Separating Content from Presentation

Developing applications that contain document presentation logic means that when an organisation's documents change (such as a new company logo, different corporate font) or the content of the documents change (following legal review or statutory changes), so must all of the documents generated by that application.

Using Docmosis, all of your 'presentational' features can be developed separately from your application, by using commonly used word processors. This has two distinct benefits:

- The initial creation of the templates can be assigned to those who are experts in that field and they can be created using commonly used word processors; and
- Branding and content changes do not require software development support, which can be time consuming and expensive.

In addition to these benefits, Docmosis is fast: the core document generation engine can generate hundreds of documents in minutes in the most popular formats, which is a great improvement on other systems currently available.

2.2 What Are Templates?

As far as Docmosis is concerned, templates are standard Microsoft Word or LibreOffice Writer documents that may also contain Docmosis "fields". Docmosis looks for the special Docmosis fields to determine where to insert data and to determine the start and end of content to be included, removed or repeated.

Docmosis does not require any custom plugins or special additions to those word processors in order to be able to create the fields. This guide will describe the different types of fields recognised by Docmosis, and how to use them.

As well as using fields to drive Docmosis, templates in Microsoft Word and LibreOffice Writer give authors control over aspects such as:

- page size, margins, and columns;
- information in headers and footers;
- typographic characteristics that describe paragraph and character styles; and
- boilerplate text, graphics and embedded field codes.

Documents generated using a template retain these features and settings when they are generated. Once the document is generated, it has no connection to the template from which



it came. Templates can be modified as required without any concern to the documents that have been previously generated.



In addition, as the paragraph and character patterns for the output are described in simple formatting terms, there is no real need to use paragraph and character styles. However, if you are comfortable using styles, it might be advantageous to implement them for future template updates.

2.3 How Does Document Generation Work?

In the simplest terms, Docmosis merges the data provided by your software application with the fields in the template to generate the documents.

Apart from simple field/data substitution, some of the fields may be instructions to Docmosis to add or remove content; repeat over blocks of data; inject images, html, hyperlinks and so on.

The generated documents may be:

- stored electronically, printed, viewed or any combination of these; and
- published in several document formats.

If the template includes an index, table of contents, page numbering or cross-references, Docmosis will automatically update these references in the resulting document, as part of the document generation process.

Any fonts used in a template (i.e. fonts installed on the template author's computer) should also be available on the server where Docmosis is running. If a font is used in a template, that is not available on the server where the document is being generated, then an unexpected font with similar shape/size may be automatically substituted in your PDF documents or you may see inaccurate page references when using indexes or tables of content.

2.4 Template Features

Modern word processors enable the creation of documents with support for high-quality typesetting and layouts incorporating inline images. By automatically inheriting these features, Docmosis provides the template author with a powerful automatic document generation capability.

Aspects of the template that are relevant to Docmosis are detailed in this section, including:

- **General features.** Information about the general word processing features that can be used to deliver high-quality layouts
- **Advanced features.** Details on the Docmosis features that can be incorporated into your template



- **Docmosis elements.** Details of the Docmosis elements that interact with the document generation process.
- **Expressions.** Details of the Docmosis math, logic and functions available within the template.

2.4.1 General Features

Many document features are achieved simply by using well-known word-processing documentation techniques. Docmosis recognizes and preserves common word processing conventions such as:

- Specifying page size and orientation
- Setting font, font-size, bold, italic, underline, text colour
- Numbered and/or bullet lists
- Headers and footers
- Page numbering and table of contents
- Using static tables and images

There is no need to learn new techniques to use these features in Docmosis templates.

2.4.2 Advanced Features

To generate sophisticated documents a variety of field types are recognised and interpreted by Docmosis. These fields can direct Docmosis by controlling:

- Insertion of text or image data into the body, headers and footers and tables
- Inclusion or exclusion of static or dynamic content
- Hyperlink Insertion
- Repeating of content
- Table row repetition or exclusion
- Table column removal
- Numbered and bullet list expansion
- Template merging.

This guide provides instructions and examples for including these advanced fields.

2.4.3 Docmosis Elements

All Docmosis “elements” are controlled by fields, except for image insertion which is controlled by bookmarks or image properties. Each element is discussed in detail later in this guide. In general, elements may be singular (such as a text insertion) or may be paired, having a start and end marker.



2.4.3.1 Field Reference

The following table provides a quick reference to the elements and their field syntax. The names of the fields must match exactly for the document generation to succeed.

Element	Description	Closing Element
<<name>>	Replace this field by the data referenced by "name".	
<<op:name>>	Replace this field by the data referenced by "name". If name is blank, the entire paragraph is stripped (including any other content). This makes the entire paragraph optional.	
<<{expr}>>	Replace this field with the results of the given expression.	
<<link:name>>	Insert a hyperlink at this location, using the URL from the data referenced by "name". The data can optionally specify display text by using the form: <text> <url> e.g.: "example https://www.example.com"	
<<\$abc=name>> <<\$abc=10.2>> <<\$abc='Fred'>> <<\$abc=true>> <<\$abc=null>>	Lookup the data associated with "name" and assign it to the variable "abc". Assign the number 10.2 to variable \$abc Assign the string "Fred" to variable \$abc Assign the boolean true to variable \$abc Assign the value null to variable \$abc	
<<\$abc>>	Lookup the variable "abc" and render its value	
<<cs_name>> <<cs_{expr}>> <<cs_\$abc>>	Content between the opening element and the closing element is included or excluded depending on the value associated with "name" or the expression "expr" or the variable "abc". The end field must match the start field exactly, or may be anonymous: <<es_>>.	<<es_name>> <<es_{expr}>> <<es_\$abc>> <<es_>>
<<else_name>> <<else_{expr}>> <<else>>	This is the "else" field related to a <<cs_>> field to provide the "else" and "else if" options in a conditional section.	



Element	Description	Closing Element
<code><<rs_name>></code> <code><<rs_\$abc>></code> <code><<rs_name:step2>></code> <code><<rs_name:step2down>></code>	Content between the opening element and closing element is repeated whilst there is data associated with “name” or the variable “abc”. “stepN” indicates that the data (“name”) should be iterated in steps of N size, effectively breaking the data into rows of N items. When stepping is used, the variables \$i1, \$i2,...\$iN are created automatically so you can reference the items available in each step. “stepNdown” indicates that the data (“name”) should be iterated in steps of N size and data should be presented in a “down”-ward (columnar) manner. The data is broken into rows of N elements, but are not presented across the row but rather down the column. Variables \$i1, \$i2,... \$iN are created automatically.	<code><<es_name>></code> <code><<es_\$abc>></code> <code><<es_name:step2>></code> <code><<es_>></code>
<code><<cr_name>></code> <code><<cr_{expr}>></code> <code><<cr_\$abc>></code>	Include the following table rows depending on the value associated with “name” or expression “expr” or the variable “abc”.	<code><<er_name>></code> <code><<er_{expr}>></code> <code><<er_\$abc>></code> <code><<er_>></code>
<code><<rr_name>></code> <code><<rr_\$abc>></code> <code><<rr_name:step2>></code> <code><<rr_name:step2down>></code>	The rows between the opening element row and the closing element row are repeated whilst there is data associated with “name” or the variable “abc”. “stepN” indicates that the data (“name”) should be iterated in steps of N size. When stepping is used, the variables \$i1, \$i2,...\$iN are created automatically so you can reference the items available in each step. “stepNdown” indicates that the data (“name”) should be iterated in steps of N size and data should be presented in a “down”-ward manner. Variables \$i1, \$i2,... \$iN are created automatically.	<code><<er_name>></code> <code><<er_\$abc>></code> <code><<er_>></code>
<code><<noTableRowAlternate>></code>	Disable automatic alternate-colouring of table rows. This can appear in a table to disable for the table or appear in the document body to disable for all following tables.	
<code><<cc_name>></code> <code><<cc_{expr}>></code> <code><<cc_\$abc>></code>	Include or exclude the table column containing this field depending on the value associated with “name” or the expression “expr” or the variable “abc”.	



Element	Description	Closing Element
Image Microsoft Word: bookmarked with label "img_name" LibreOffice Writer: image named "img_name"	Replace an image in the template with the image data associated with "name" using the default scaling settings (which is stretch).	
Image stretched bookmarked with label or named "imgstretch_name"	Replace an image in the template with the image data associated with "name" and stretch the new image to match the template image placeholder.	
Image scaled to fit bookmarked with label or named "imgfit_name"	Replace an image in the template with the image data associated with "name" and fit the new image into the template image placeholder preserving the new image aspect ratio.	
<<ref:sub1.doc>>	Insert the template named "sub1.doc" at this location.	
<<refLookup:name>>	Lookup "name" in the data to get the name of the template to insert at this location.	
<<list:continue>>	To be used inside a sub-template numbered list. Specifies that numbering should be continued on from an existing numbered list when inserted.	
<<list:reset>>	To be used for a numbered list inside a repeating section where the list numbering should not be continued between repeats of the section.	
<<html:name>>	Lookup "name" in the data and inject the data as HTML content into the document at this location	
<<barcode:name:...>>	Provide information for a barcode image in the template. e.g. <<barcode:dispatchLabel:code128>> defines image "dispatchLabel" as a code 128 barcode.	
<<## and ##>> <</* and */>>	Template-comments are delimited by the matching open and closing sequences. Content inside comments is not processed and is removed when creating documents.	

2.4.3.2 Nesting

Elements can use a "dot-notation" to look up nested / hierarchical data. The period "." character represents the delimiter between one level of data and the next.



For example, `<<hotel.floor>>` typically would refer to the floor within a hotel object.

2.4.3.3 Range Specifiers

Data elements can also be referenced by **ranges** of values Docmosis should look up. This provides a fair amount of power within the template to select the values of interest. It depends on the context of the element as to whether it is allowed to generate multiple values (and Docmosis will flag errors where inappropriate use is made). For example, a repeating section is expected to generate multiple values, but a simple lookup field is not.

The following table details the types of range specifier available.

Element	Description
<code><<hotel[0]>></code>	The first hotel (indexing starts at zero)
<code><<hotel[F]>></code>	The first hotel (equivalent to index zero)
<code><<hotel[L]>></code>	The last hotel
<code><<hotel[*]>></code>	All hotels
<code><<hotel[F3]>></code>	The first 3 hotels
<code><<hotel[L3]>></code>	The Last 3 hotels
<code><<hotel[1,2,4]>></code>	The hotels at indexes 1,2 and 4
<code><<hotel[1-3,L2]>></code>	The hotels at indexes 1 to 3 inclusive and the last 2
<code><<hotel[0-L2]>></code>	All but the last 2 hotels
<code><<hotel[3].floor[L].room[0].name>></code>	The name of the first room of the last floor of the hotel at index 3

2.4.3.4 Built-In Variables

Docmosis provides some built-in variables to assist with common data lookup requirements.

Variable	Description
<code><<\$top>></code> or <code><<\$root>></code>	The root of the data regardless of the current position or context in the template.



Variable	Description
<code><<\$this>></code> or <code><<\$current>></code>	The current source of data in the current position in the template. This allows for anonymous data lookups from arrays or collections such as <code><<\$current[0]>></code> .
<code><<\$parent>></code>	The parent or container of data in the current context of the template. Allows data lookup in the current “hotel” when the current context is a “floor” for example.
<code><<\$nl>></code>	A simple newline character
<code><<\$nowMS>></code>	Current UTC time in milliseconds since 1/1/1970
<code><<\$nowUTC>></code>	Current UTC time as in ISO 8601 format
<code><<\$nowUTCFormat>></code>	The format used for \$nowUTC which is: <code>yyyy-MM-dd'T'HH:mm:ssX</code>
<code><<\$quot>></code>	The single-quote character

Variables available in Repeating Sections and Repeating Rows:

Variable	Description
<code><<\$idx>></code> Index into data	The current index into the source data, starting from a zero offset from the beginning of the data range. This is typically the same as <code>\$itemidx</code> , however if repeating over a range of values that doesn't start at zero (e.g. <code><<rs_names[3-5]>></code>), the <code>\$idx</code> values into the data would be 3,4,5.
<code><<\$itemidx>></code> Index in our iteration	The current index into an iteration, starting from zero. This is unaffected by the ranges of the data specified so the <code>\$itemidx</code> values for <code><<rs_names[3-5]>></code> is 0,1,2.
<code><<\$num>></code>	The same as <code>\$idx</code> but starting from one.
<code><<\$itemnum>></code>	The same as <code>\$itemidx</code> but starting from one.
<code><<\$size>></code>	The size of the current repeating data set. For example, if we are repeating over all hotels, <code>\$size</code> would be the number of hotels.



Variable	Description
<code><<\$rownum>></code>	The current row number (starting at 1) when repeating (either repeating rows or repeating sections). This is most useful when using the "stepping" directives and the <code>\$itemnum</code> is not suitable. For more information about the use of "steps of N" see 3.10.1 Stepping Across in Repeating Sections and 3.10.2 Stepping Down in Repeating Sections.
<code><<\$rowidx>></code>	The current row number (starting at 0) when repeating (either repeating rows or repeating sections). This is most useful when using the "stepping" directives and the <code>\$itemidx</code> is not suitable. For more information about the use of "steps of N" see 3.10.1 Stepping Across in Repeating Sections and 3.10.2 Stepping Down in Repeating Sections.

Further variables available when "stepping" through repeating data:

Variable	Description
<code><<\$i1>>, <<\$i2>>, ... <<\$iN>></code>	References to the Nth item when repeating data in "steps of N". For example <code><<rs_people:step3>></code> steps through the people in "steps of 3" and Docmosis automatically creates variables <code>\$i1</code>, <code>\$i2</code> and <code>\$i3</code> to access each element in the step. For more information about the use of "steps of N" see 3.10.1 Stepping Across in Repeating Sections and 3.10.2 Stepping Down in Repeating Sections.
<code><<\$idx1>>, ... <<\$idxN>></code>	Shorthand for <code>\$i1.\$idx</code> , ... <code>\$iN.\$idx</code>
<code><<\$num1>>, ... <<\$numN>></code>	Shorthand for <code>\$i1.\$num</code> , ... <code>\$iN.\$num</code>
<code><<\$itemidx1>>, ... <<\$itemidxN>></code>	Shorthand for <code>\$i1.\$itemidx</code> , ... <code>\$iN.\$itemidx</code>
<code><<\$itemnum1>>, ... <<\$itemnumN>></code>	Shorthand for <code>\$i1.\$itemnum</code> , ... <code>\$iN.\$itemnum</code>

2.4.4 Expressions and Functions

Docmosis uses { and } to delimit an expression to be evaluated. Expressions are a powerful way of retrieving and manipulating data within the template.

The syntax supports:

- Operators (e.g. + to add numbers and strings, * to multiply numbers)



- Functions (e.g. `titleCase(name)`)
- Data lookup (get data by name)
- Literals (e.g. `'abc'` or `123`; additional spaces using `' '`)

Expressions can be used for simple data insertion:

```
<<{ 'Ms. ' + lastName }>>
```

and in conditional sections:

```
<<cs_{itemCount < 10}>>
```

and where template-variables are set:

```
<<$myVar={ 'Ms. ' + lastName }>>
```

The following table shows some examples of expressions in use. The sections to follow detail the operators and functions available.

Element	Description
<code><<{10 * 3.0}>></code>	Display the result of 10 multiplied by 3.0
<code><<{amount * qty}>></code>	Lookup data elements “amount” and “qty” and multiply them together and display the result.
<code><<{round(item/10)}>></code>	Lookup data element “item”, divide it by 10 then round the result.
<code><<cs_{a<10}>></code>	Lookup data element “a” and see if it is less than 10 numerically. If “a” is not numeric, a string comparison is performed automatically. If the value is less than 10 the expression is true and the conditional section will be included
<code><<cs_{a='fred'}>></code>	Lookup data element “a” and see if it is equal to the String literal “fred” and if so the conditional section will be included
<code><<cs_{\$a!=10}>></code>	Lookup the variable “a” and see if it is not equal to the numeric value 10. If variable “a” does not resolve to a numeric value, a String comparison is performed. If \$a is not equal to the value 10, the conditional section will be included.
<code><<cs_{a=null}>></code>	Lookup the data element “a” and determine if its value is null and include the conditional section if so.
<code><<cs_{\$a}>></code>	Determine if the value of the template variable \$a is true and if so, include the conditional section.



2.4.4.1 Expression Operators

The following operators are supported by the Docmosis expression syntax:

Operator	Description
(	open parentheses
)	close parentheses
+	addition (for numbers and strings)
-	subtraction
*	multiplication
/	division
%	modulus
+	unary plus
-	unary minus
=	equal (for numbers and strings)
==	equal (for numbers and strings)
!=	not equal (for numbers and strings)
<	less than (for numbers and strings)
<=	less than or equal (for numbers and strings)
>	greater than (for numbers and strings)
>=	greater than or equal (for numbers and strings)
&&	boolean and
	boolean or
!	boolean not

Typical “Operator precedence” rules apply to determine the order of processing (highest to lowest):

- (open parentheses,) close parentheses
- + unary plus, - unary minus, ! boolean not
- * multiplication, / division, % modulus



- + addition, - subtraction
- < less than, <= less than or equal, > greater than, >= greater than or equal
- = equal, != not equal
- && Boolean, and
- || boolean or

2.4.4.2 Logic and Transform Functions

The following general functions are supported by the Docmosis expression syntax:

Function	Synopsis
ifBlank	A function to use a default value if the given element is null or empty. <pre>ifBlank(key, default)</pre> where: <pre>key = the data value</pre> <pre>default = the value to use if key is blank</pre> For Example: <pre><<{ifBlank(name, 'Not Specified')}>></pre> Will lookup "name" in the data if null or empty it will return "Not Specified".
isBlank	A function to determine if the given element is null or empty. <pre>isBlank(key)</pre> where: <pre>key = the data value</pre> For Example: <pre><<{isBlank(name)}>></pre> Will lookup "name" in the data and return true if null or empty, otherwise false. This can be useful for conditional sections: <pre><<cs_{isBlank(address)}>></pre> <pre>There is no address</pre> <pre><<es_>></pre>



Function	Synopsis
map	A function to map one value to another. <pre>map(key, test1, replace1 [,test2, replace2 ...] [,default])</pre> where: key = the data value test1 = the first value to compare with the key replace1 = the value to use if test1 matches the key test2 = the second value to compare with the key replace2 = the value to use if test2 matches the key ... default = the value to use if no matches are made For Example: <pre><<{map(gender, 'M', 'Male', 'F', 'Female', 'Other')}>></pre> Will lookup “gender” in the data and if it equals “M” the value “Male” will be used.
mapi	A function to map one value to another ignoring the case of the key and test values (case insensitive). <pre>mapi(key, test1, replace1 [,test2, replace2 ...] [,default])</pre> where: key = the data value test1 = the first value to compare with the key replace1 = the value to use if test1 matches the key test2 = the second value to compare with the key replace2 = the value to use if test2 matches the key ... default = the value to use if no matches are made For Example: <pre><<{mapi(gender, 'M', 'Male', 'F', 'Female', 'Other')}>></pre> Will lookup “gender” in the data and if it equals “M” or “m” the value “Male” will be used.



2.4.4.3 Text Functions

The following text functions are supported by the expression syntax:

Function	Synopsis
char	Generate a character from the given character code <pre>char (code)</pre> where: code = the character code in decimal or hex form For Example: <pre><<{char(169)}>></pre> <pre><<{char("0xA9")}>></pre> <pre><<{char('&#169')}>></pre> <pre><<{char('&#xA9')}>></pre> <pre><<{char('\u0A9')}>></pre> All return the copyright symbol ©
charAt	Returns the character at the requested position in the source string. <pre>charAt (string, position)</pre> where: string = the string to lookup the character in key = the position of the required character, starting from 0 for the first position. For Example: <pre><<{charAt('abcdefg',3)}>></pre> returns the character "d" <pre><<{charAt(idNumber,6)}>></pre> will lookup "idNumber" in the data. If idNumber= "ID474-K234" then the character returned will be "K".



Function	Synopsis
endsWith	Checks to see if a string ends with a given string. <pre>endsWith (mainString, subString)</pre> where: <pre>mainString = the string to check subString = the string to look for at the end of mainString</pre> For Example: <pre><<{endsWith('The first string', 'ing')}}>> returns the value "true"</pre> Useful when creating a conditional section. For example, this conditional section will only display the "serialNum" field if it ends with "ZZZ". <pre><<cs_{endsWith(serialNum, 'ZZZ')}}>> <<serialNum>> <<es_>></pre>
equalsIgnoreCase	Compares to strings, regardless of case. <pre>equalsIgnoreCase (string1, string2)</pre> where: <pre>string1 = the first string string2 = the second to compare to the first string</pre> For Example: <pre><<{equalsIgnoreCase('Bob', 'bob')}}>> returns the value "true"</pre>
indexOf	Returns the starting index of one string inside another. <pre>indexOf (string, find [, startIdx])</pre> where: <pre>string = the string to scan find = the string to find startIdx = an optional search starting index</pre> For Example: <pre><<{indexOf('Bob Mathews', 'Mat')}}>> returns "4.0"</pre>



Function	Synopsis
length	Returns the length of a string. <pre>length (string)</pre> where: <pre>string = the string to check the length of</pre> For Example: <pre><<{length('Bob')}>></pre> returns the number "3.0" Useful when creating a conditional section. For example, this conditional section will only display the text if "refNo" is set. <pre><<cs_{length(refNo)>0}>> Ref Num : <<refNo>> <<es_>></pre>
replace	Replaces characters in the source string with new characters. <pre>replace (string, oldChar, newChar)</pre> where: <pre>string = the string oldChar = the character to find in the string newChar = the character to use in place of the oldChar</pre> For Example: <pre><<{replace(customerVIN,'o','0')}>></pre> If the data contains customerVIN = "JHMAB5227EC8oo65o" Then the replace function will turn all the letter "o" chars to the number "0" so the result looks like this : "JHMAB5227EC800650"
replaceStr	Replaces strings in the source string with the new string. <pre>replaceStr(string, searchFor, replaceWith [, ignoreCase])</pre> where: <pre>string = the string searchFor = the character(s) to find in the string replaceWith = the character(s) to use in place of the searchFor ignoreCase = true false to ignore the case when searching. Defaults to false (ie case-sensitive).</pre> For Example: <pre><<{replaceStr(address,'street','St.', true)}>></pre> If address="Matheson street", returns "Matheson St."



Function	Synopsis
replaceFirst	Replaces the first occurrence of source string with the new string. <pre>replaceFirst(string, searchFor, replaceWith [, ignoreCase])</pre> where: <pre>string = the string searchFor = the character(s) to find in the string replaceWith = the character(s) to use in place of the searchFor ignoreCase = true false to ignore the case when searching. Defaults to false (ie case-sensitive).</pre> For Example: <pre><<{replaceFirst('Two times Two','two','2', true)}>></pre> Results in '2 time Two'.
split	Split a string into parts that can be displayed separately. <pre>split (string, splitChar, index)</pre> where: <pre>string = the string splitChar = the character to use as a delimiter index = once split into parts, index identifies the part to be used - counting from 0.</pre> For Example: <pre><<{split('John Mathews 47 Approved',' ', 1)}>></pre> returns "Mathews" <pre><<{split(cityStateZIPCountry, ';', 1)}>> with cityStateZIPCountry = "Charleston;West Virginia;29402;United States"</pre> will return "West Virginia"
squote	Replace all double-quote characters in the given string with single-quotes. All forms of double-quotes are replaced. This is handy since the templates use single quotes for delimiters. <pre>squote(string)</pre> where: <pre>string = the string in which to replace double-quotes</pre> For Example: <pre><<{squote('This is Amy"s.')}>> returns "This is Amy's."</pre>



Function	Synopsis
startsWith	Checks to see if a string starts with a given string. <pre>startsWith (mainString, subString)</pre> where: <pre>mainString = the string to check subString = the string to look for at the start of mainString</pre> For Example: <pre><<{startsWith('The first string', 'The')}>></pre> returns the value "true" Useful when creating a conditional section. For example, this conditional section will only display the "VIN" field if it starts with "1VW". <pre><<cs_{startsWith(VIN, '1VW')}>> <<VIN>> <<es_>></pre>
substring	Display a subsection of a string given starting and finishing indexes. <pre>substring(string, start, finish)</pre> where: <pre>string = the string start = the position in the string that will now become the first character. Indexing starts at 0. finish = the position in the string that marks where to cut the string. The character before the cut makes it in to the substring. The finish character doesn't.</pre> For Example: <pre><<{substring('0123456' , 2 , 5)}>></pre> returns "234" <pre><<{substring(LatLong , 0 , 6)}>></pre> with <pre>LatLong = "31.9088983S115.8049265E"</pre> will return "31.908"



Function	Synopsis
titleCase	Changes the string so that the first character of each word is a capital letter. <pre>titleCase (string)</pre> where: <pre>string = the string to convert</pre> For Example: <pre><<{ titleCase ('bob mathews')}>></pre> returns "Bob Mathews" <pre><<{titleCase (firstName+ ' ' + lastName)}>></pre> with data of firstName = "bob" and lastName = "MATHEWS" also returns "Bob Mathews"
toAlpha	A function to convert a given number to a letter in the sequence: <i>a, b, c, ... z, aa, bb, cc, ... zz, aaa, bbb, etc.</i> <pre>toAlpha (key)</pre> where: <pre>key = the data value</pre> For Example: <pre><<{toAlpha(index)}>></pre> when index = "3", returns "c". when index = "28", returns "bb".
toAlpha2	A function to convert a given number to a letter in the sequence: <i>a, b, c, ... z, aa, ab, ac ... az, ba, bb, etc.</i> <pre>toAlpha2 (key)</pre> where: <pre>key = the data value</pre> For Example: <pre><<{toAlpha2(index)}>></pre> when index = "3", returns "c". when index = "28", returns "ab".



Function	Synopsis
toLowerCase	Returns the string using all lower case characters. <code>toLowerCase (string)</code> where: <code>string = the string to convert</code> For Example: <code><<{toLowerCase('Bob Mathews')}>></code> returns "bob mathews"
toUpperCase	Returns the string using all upper case characters. <code>toUpperCase (string)</code> where: <code>string = the string to convert</code> For Example: <code><<{toUpperCase('Bob Mathews')}>></code> returns "BOB MATHEWS"
toRoman	A function to convert a given number to a roman numeral <code>toRoman(key)</code> where: <code>key = the data value</code> For Example: <code><<{toRoman(index)}>></code> when index = "2", returns "ii". when index = "29", returns "xxix".
toSentence	Adjusts the given string converting it to sentence case. <code>toSentence(string)</code> where: <code>string = the string to convert</code> For Example: <code><<{toSentence('a little. ditty')}>></code> returns "A little. Ditty"



Function	Synopsis
trim	Removes leading and trailing spaces from a string. <pre>trim (string)</pre> where: <pre>string = the string</pre> For Example: <pre><<{trim(productID)}>></pre> Where productID = " 12CVCV123-454 " returns "12CVCV123-454"

2.4.4.4 Numeric Functions

The following numeric functions are supported by the expression syntax.

Any of the number literals (e.g.:"153.57") in the examples below could be replaced with a "name" that Docmosis will look for in the data.

Function	Synopsis
abs	Returns the absolute value of the number. <pre>abs (number)</pre> For Example: <pre><<{abs(-153.57)}>></pre> returns "153.57". <pre><<{abs(temp)}>></pre> If the data has: <pre>temp = "-273.15"</pre> returns "273.15"
ceil	Returns the next largest whole number. <pre>ceil (number)</pre> For Example: <pre><<{ceil(153.57)}>> returns "154.0"</pre>
floor	Returns the next smallest whole number. <pre>floor (number)</pre> For Example: <pre><<{floor(153.57)}>> returns "153.0"</pre>



Function	Synopsis
isNumber	Returns true if the given parameter is numeric: <code>isNumber (value1)</code> For Example: <code><<{isNumber(53)}>></code> returns "true" <code><<{isNumber('53.5111')}>></code> returns "true" <code><<{isNumber('-1.0e20')}>></code> returns "true"
max	Returns the larger of the two numbers. <code>max (number1, number2)</code> For Example: <code><<{max(53.5,23.1)}>></code> returns "53.5"
min	Returns the smaller of two numbers. <code>min (number1, number2)</code> For Example: <code><<{min(53.5,23.1)}>></code> returns "23.1"
numFormat	Format a number based on the format provided and the locale. <code>numFormat (value, format [, locale [,applyLocaleToInput [, formatIsLocalized]])</code> where: <code>value</code> = the number to format <code>format</code> = the format to apply. E.g.: '#,###.00' <code>locale</code> = optional locale to use. Country or language name or code. E.g.: 'GERMAN', 'USA'. <code>applyLocaleToInput</code> = whether to apply the locale to the input value. Default is true. Set to false when value is numeric data or data that is not parseable in the given locale. <code>formatIsLocalized</code> = whether to apply the locale-specific interpretation to the given format. Default is true. Set to false when the format is specified in no localized format. For example, when specifying the year value in the FRENCH locale, by default the format uses 'a' to represent the year. When this is false, 'y' is expected regardless of the locale. See section 4.1 Formatting Numbers for full formatting syntax.



Function	Synopsis
numToDollars	Format a number based on the format provided and the locale. <pre>numToDollars (value)</pre> where: <pre>value = the number to format</pre> If the number is an integer, just the dollar amount is written. If the number has a fractional component, the number will be rounded to the nearest even 2 decimal places and the “cents” value will be written. If the number is text with the \$ symbol embedded the \$ symbol is ignored. Supports numbers up to 9.2 quintillion. For Example: <pre><<{numToDollars(100)}>> returns “one hundred dollars”</pre> <pre><<{numToDollars(123.457)}>> returns “one hundred twenty three dollars and forty six cents”</pre>
numToText	Write out the given number using English words <pre>numToText (value [, form])</pre> where: <pre>value = the number to format</pre> <pre>form = 'andLast' - (default) write the last “and” in the text</pre> <pre>form = 'andAlways' - write the “and” in all places in the text</pre> <pre>form = 'andNone' - brief - don't write the “and” at all.</pre> Supports numbers up to 9.2 quintillion. For Example: <pre><<{numToText(123)}>> returns “one hundred and twenty three”</pre> <pre><<{numToText(123, 'andNone')}>> returns “one hundred twenty three”</pre>



Function	Synopsis
ordinal	Write out the given number as an ordinal using either numbers or English words. <pre>Ordinal (value [, form])</pre> where: value = the number to format form = 'short' - (default) write as numeric digits plus suffix (eg "123" as "123rd") form = 'suffix' - write only the suffix text ("st", "rd", "th") form = 'long' - write out the number in words and include the "and" word in the final text form = 'longNoAnds' - write out the words but don't use "and" to join any part of the text form = 'longAllAnds' - write out the words and use "and" to join in all locations. Supports numbers up to 9.2 quintillion. For Example: <pre><<{ordinal(123)}>> returns "123rd"</pre> <pre><<{ordinal(123, 'suffix')}>> returns "rd"</pre> <pre><<{ordinal(123, 'long')}>> returns "one hundred and twenty third"</pre> <pre><<{ordinal(123, 'longNoAnds')}>> returns "one hundred twenty third"</pre>
pow	Returns the power of two numbers. <pre>pow (number1, number2)</pre> For Example: <pre><<{pow(7,2)}>> returns 7 to the power of 2, so "49.0"</pre>
random	Returns a random number between 0 and 1. <pre>random()</pre> For Example: <pre><<{round(random()*100)}>></pre> returns a random number between 0 and 100.



Function	Synopsis
round	Rounds the number to the specified number of places. <pre>round (number [, places])</pre> where: number = the number to round. places = the number of decimal places required. If not specified then round to zero decimal places. For Example: <pre><<{round(153.75)}>></pre> returns "154" <pre><<{round(153.73455,2)}>></pre> returns "153.73"
sqrt	Returns the square root of a number <pre>sqrt (number)</pre> For Example: <pre><<{sqrt(81.0)}>> returns "9.0"</pre>



2.4.4.5 Date Functions

The following date functions are supported by the expression syntax:

Function	Synopsis
dateAdd	Add (or subtract) an amount from a given date. <pre>dateAdd (date, amt, units [,outputFormat [,inputFormat]])</pre> where: date = the starting date value amt = the amount to adjust by (may be negative) units = the units of amt (milli/millis, second/seconds, minute/minutes, hour/hours, day/days, week/weeks, month/months, year/years) outputFormat = optional - the output format to display the result. inputFormat = optional - the format used to decode the input data value. Docmosis will attempt to decode the given date using various formats if this parameter is not specified. For Example: <<{dateAdd('10 Jul 2020', 5, 'day')}>> returns "15 Jul 2020" <<{dateAdd('10 Jul 2020', -2, 'months')}>> returns "10 May 2020" <<{dateAdd('10 Jul 2020', 1, 'month', 'MMMM')}>> returns "August" <<{dateAdd('07-10-2020', 2, 'year', 'yyyy', 'MM-dd-yyyy')}>> returns "2022"
dateDiff	Calculates the difference between two dates in the requested units. <pre>dateDiff (date1, date2, units [, inputFormat])</pre> where: date1 = the starting date/time date2 = the finishing date/time units = the units in which to report the result (milli/millis, second/seconds, minute/minutes, hour/hours, day/days, week/weeks, month/months, year/years) inputFormat = optional - the format used to decode the input dates. Docmosis will attempt to decode them using various formats if this parameter is not specified. For Example: <<{dateDiff('10 Jul 2020', '18 Jul 2020')}>> returns "8"



Function	Synopsis
dateFormat	Format the value based on the output and input Formatting strings. <pre>dateFormat (value [, outputFormat [, inputFormat [, outputLocale [, inputLocale [, outputFormatLocalized [, inputFormatLocalized]]]]])</pre> where: value = the data value to format outputFormat = optional - the output format to apply inputFormat = optional - the format used to decode the input data value outputLocale = optional - the locale to use for rendering the date inputLocale = optional - the locale to use for parsing the input date outputFormatLocalized = optional - indicates whether the outputFormat string should be interpreted using characters for the specified outputLocale. Default=false. inputFormatLocalized = optional - indicates whether the inputFormat string should be interpreted using characters for the specified inputLocale. Default=false. With regards to localized formats, if using the FRENCH locale, then the *FormatLocalized flag should be set to true if your format specifies, for example, the year using 'a' rather than 'y'. See section 4.2 Formatting Dates for full formatting syntax.



2.4.4.6 Locale Functions

The following locale-related functions are supported by the expression syntax:

Function	Synopsis
locale	Get the specific ID for a given locale string. This is most useful during development and testing and can be used to determine specifically what Locale Docmosis will use when a language or country code is specified as a locale parameter. <pre>locale([spec])</pre> where: spec = the language or country name or code to lookup For Example: <<{locale()}>> returns the default locale in use, typically "en_US" (English, United States) <<{locale('GERMANY')}>> returns "de_DE"
localeInfo	Get the info for a locale obtained by the given locale string. This is most useful during development and testing and can be used to determine specifically what Locale Docmosis will use when a language or country code is specified as a locale parameter, and what attributes it has. <pre>localeInfo([spec])</pre> where: spec = the language or country name or code to lookup For Example: <<{locale()}>> returns info about the default locale in use: "Locale:[default] country=United States, lang=English, variant=, id=en_US" <<{locale('GERMANY')}>> returns "Locale:[GERMANY] country=Germany, lang=German, variant=, id=de_DE"



Function	Synopsis
localeDatePattern	For the given pattern and locale, display the “localized” version of the pattern. This can help diagnose issues with locale-specific patterns. <pre>localeInfo(pattern [, locale])</pre> where: <pre>pattern = the non-localized pattern: eg dd-MMMM-YYYY locale = the locale for which to display the localized pattern</pre> For Example: <pre><<{localeDatePattern('dd-MMMM-yyyy', 'GERMANY')}>> returns tt-MMMM-uuuu</pre>

2.5 Error Handling

Docmosis offers two ways to deal with errors encountered in templates during the document generation process:

1. **Dev Mode.** Write the error in to the resulting document - errors are highlighted in red and where possible footnotes are added for each error, to offer details and suggestions on how to resolve the error.
2. **Prod Mode.** Return an error and abort document generation. No document is produced and the error will contain details of the first problem that was found.

This behaviour is not controlled by the template, rather specified at the time the document is being generated, since it is expected to be related to the type of environment in which Docmosis is being used.



3 Creating Docmosis Templates

The basic steps for creating a template are:

1. create the layout, boilerplate/static content and typesetting characteristics of a document;
2. add the Docmosis elements (fields).

The boilerplate content can include sophisticated word processing structures using headings, lists, tables, static images, and headers and footers.

This chapter provides instructions for the inclusion of the supported fields. It is divided into sections that discuss the basic aspects through to some advanced techniques. In general, the information does not cover typesetting of documents but does provide information where necessary. Most of the information in this chapter is relevant to both word processors: where they differ, information is provided for each case.



All the procedures in this chapter assume that you understand the techniques required for the particular word processor and that you have a document open in the word processor on which you can perform the procedure.

In addition, the procedures use menu-based instructions for consistency.

3.1 Incorporating Docmosis Elements

You can add Docmosis "fields" at any location in a document template. Each field must have an appropriate and unique name that associates it with an element of the data that will be supplied to Docmosis.

During document generation, Docmosis expects the data to have values and logical structures that match the names and structure of the elements in the template. For example: nested items in the template should also be nested in the data structures.

Consideration should be given to how the content of the template will "move" during the document generation process. It is best practice to keep all text, images, tables, etc. in-line, so that the content moves and changes in a predictable way, just as if the content was cut from the template or pasted into the template.

Docmosis recognizes fields created using:

- plain text fields (i.e.: created just by typing into the template)
- merge fields in Microsoft Word
- input fields in LibreOffice Writer

Plain text fields are the simplest to use since there are no dialogs to interact with and what you see is what you get.



3.2 Using Plain Text Fields

Plain text fields are the easiest method of creating fields in Docmosis templates. By default, the start of a field is denoted by << [less-than less-than] and the end of a field by >> [greater-than greater-than].



The default start of field prefix '<<' and the end suffix '>>' can be changed to any pattern you start or end pattern you prefer.

The remainder of this document assumes the default prefix and suffix are being used.

To create a field that looks up "personName" in the data, you would simply type <<personName>> into the template.

To keep the text of a template simple, Docmosis is strict about identifying plain text fields and will ignore invalid fields, assuming it is just static text in the template. For example, <<personName> will be ignored because a closing ">" character is missing. A **single** space between the << and the name, or the name and >> is allowed, but more spaces will also mean the field is not recognised.

The following table shows typical errors that will result in a field not being recognised.

Example Field	Valid	Problem
<<personName>>	YES	Correct field. Docmosis will identify and substitute.
<<personName>	NO	Missing trailing >
<personName>>	NO	Missing leading <
<< personName>>	NO	2 spaces after leading <<
<<personName >>	NO	2 spaces before trailing >>
< <personName>>	NO	Space after leading <
<<personName> >	NO	Space before trailing >

3.3 Using the Built-in Word Processor Fields

Docmosis also supports the use of the "document fields" supplied by Microsoft Word (*merge* fields) and LibreOffice Writer (*input* fields).

The advantages of using these document fields include:

- less work if you already have a template that includes merge fields or input fields.
- you can display text that is different from the actual name of the data item used by Docmosis. For example, the following field can appear in the template as:



«friends»

but may in fact be referencing the following piece of data:

«friends[0].lookupName»

This means the field can appear smaller or more succinct in the template than if a plain text field was used.

The disadvantages of using document fields include:

- a field can be confusing or misleading because its true lookup value is hidden.
- more effort is required to work with these fields via popup dialogs or switching field codes on and off.
- with Word merge fields, the "display" value can be accidentally lost (replaced with the underlying lookup value) if the fields in the document are "updated".
- depending upon the version of your chosen word processor you may not be able to create fields with spaces or special characters in the name.
- it can be difficult to simply insert a merge field as Microsoft Word attempts to guide the user to link the field to a data source – which is not required. This means you need to either type the field codes manually, or copy a merge field from another document then edit it to reference the data item you require.

3.3.1 Creating a Field Using Microsoft Word Merge Fields

It is generally simpler to use plain text fields as described in section 3.2 above.

Microsoft Word has a built-in mail merge feature that uses MergeFields as the placeholders for the dynamic content. Docmosis will recognize a MergeField and will use the "name" of the MergeField to look up the data item.

By default, Microsoft Word displays the name of the MergeField in the template. You may change the text that is displayed without changing the name of the merge field. Simply edit the text that appears between the angle brackets. Please note that if you later update the field codes in the document, the "display" name will be reverted back to the "real" name.



In Microsoft Word, to view the data item that the merge field is referencing, instead of its display value, place the cursor in the merge field and press Alt-F9.

3.3.2 Inserting a Field Using LibreOffice Writer Input Fields

It is generally simpler to use plain text fields as described in section 3.2 above.

Docmosis will recognize an Input Field and will use the "Reference" value of the Input Field to look up the data item.



LibreOffice Writer does NOT show a pair of angled brackets (« ... ») around the Input Field.

To change the text displayed in the template you can type directly in to an input field. Right click on an Input Field and select Edit Fields to change the data item that the Input Field is referencing.

3.4 Using Text Substitution

The simplest (and often most useful) fields in a Docmosis template are the ones that look up data and place it into the document (essentially, this is a one-to-one match). Wherever a field occurs, Docmosis will inject the actual data value into that location in the finished document.

The inserted data inherits all the typesetting characteristics that are applied to the field, such as font settings (bold, underline, etc.) and paragraph settings (before/after spacing, etc.). Docmosis will replace the field with the data supplied as if you had selected and typed over the field by hand. If the lookup data contains new-line characters, Docmosis will create new paragraphs in the resulting document.

If no data item is found matching the field name, the field is removed.

Docmosis has a powerful expression processing engine allowing you to create fields that perform calculations within the template. The expression syntax supports literals, data-lookups, operators and functions. The result of evaluating an expression will be displayed, instead of simply the piece of data. Expression processing is enabled by using curly brackets, like this "<<{" and this "}>>", inside the normal field delimiters. See 2.4.4 Expressions and Functions for a complete reference of all supported operators and functions.

3.4.1 Optional Paragraph Fields

Optional paragraph fields operate like the fields described earlier, except if there is no data for the value, the entire paragraph containing the field is removed.



HINT: Turn on hidden formatting symbols, like the paragraph marker (¶), to see where a paragraph starts and finishes.

Optional paragraphs are specified with the prefix "op:", for example:

```
<<op:addressLine2>>
```

Optional paragraph fields are useful for condensing output (not leaving behind blank lines) when populating data. Consider a typical address block:



```
<<name>>  
<<addr1>>  
<<addr2>>  
<<city>>, <<country>>
```

If there is no value for `addr2`, the above sequence would produce output that looks like:

```
My Company  
123 The Boulevarde  
  
San Francisco, USA
```

The blank line in the middle of the above output is possibly undesirable.

Using an optional paragraph field with `<<addr2>>` would look like this:

```
<<name>>  
<<addr1>>  
<<op:addr2>>  
<<city>>, <<country>>
```

Which means that in the output the blank line will not appear:

```
My Company  
123 The Boulevarde  
San Francisco, USA
```

Optional paragraph fields are also useful for removing paragraphs from numbered or bullet lists. Consider:

1. I have one <<item1>>
2. I have one <<item2>>
3. I have one <<item3>>

With data `"item2" = "orange"` and `"item3" = "banana"`, this would result in:

1. I have one
2. I have one orange
3. I have one banana

Clearly point #1 above is incomplete because there is no `item1` data. Changing to optional paragraph fields resolves this:

1. I have one <<op:item1>>
2. I have one <<op:item2>>



3. I have one <<op:item3>>

With data "item2" = "orange" and "item3" = "banana", this would result in:

1. I have one orange
2. I have one banana



Optional Paragraphs will strip the entire paragraph containing the "op:" instruction – from the end of the last paragraph marker (¶) to the next paragraph marker (¶). If you have other content in the paragraph, it will also be removed.

3.4.2 HTML

Docmosis supports the insertion of HTML content. The following field:

```
<<html:myHtmlData>>
```

Will cause `myHtmlData` to be fetched from the data and injected as HTML. For example, if `myHtmlData` contained:

```
<h1>My Heading</h1>
```

Then the text "My Heading" will appear as Heading 1 in the output document.



HTML can be arbitrarily complex and not all HTML will be rendered into a document as well as it would in a browser. Typically, using inline styles (rather than style declarations) will produce good results. The intention is to allow simple HTML "snippets" to be inserted via data where this is advantageous to the application using Docmosis.

3.5 Using Docmosis Variables

You can create variables in your templates to hold the value of a single piece of data or as a reference to an object of data by using the \$ (dollar) symbol in front of a variable name.

This can be useful to store the value of a field with a long name, in a shorter variable name, for example:

```
<<$n=longField.ThatIsTooBigToFit.InaSmallSpace.name>>  
<<$n>>
```

In the example above the entire first line will be removed from the document because it is on a line by itself and its only purpose is to set the variable `$n`. Only the second line will produce output in the finished document which will be the content of `$n`.



Variables can be used to store the results of a calculation. Note the {} curly brackets around the expression:

```
<<$amount={quantity*unitPrice}>>
```

Once a variable is set, it is visible at the “level” at which it is set. This means that if a variable is only used for the first time inside a repeating section or a set of repeating rows, then it will only be available while inside that “loop”. If you want a variable to retain its value outside of a loop, you should use it once before your start repeating field, like `$subTotal` in this example:

```
<<$subTotal=0>>
<<rs_items>>
    The cost of item <<$num>> is <<cost>>.
<<$subTotal={$subTotal+cost}>>
<<es_>>
    The cost of all the items is <<$subTotal>>.
```

Note, `$num` in the example above is one of the built-in variables discussed in section 2.4.3.4 Built-In Variables.

A variable can be used to store an object, and once it is used, regardless of where in the template, it will provide visibility to the contents of the object that it referenced. For example, a variable named `$firstPerson`, could be used to store the first person in the `person` data. Then from deep inside a repeating section the name of the first person is retrieved using the `$firstPerson` variable.

```
<<$firstPerson=people[0]>>
<<rs_people>>
    Current person = <<name>>. First = <<$firstPerson.name>>
<<es_people>>
```



Variables can also be referenced using `var_` instead of `$`. This means `<<$name>>` is equivalent to `<<var_name>>`. This is particularly useful for bookmarking images using variables in Microsoft Word, where you cannot use the `$` symbol in the bookmark name.

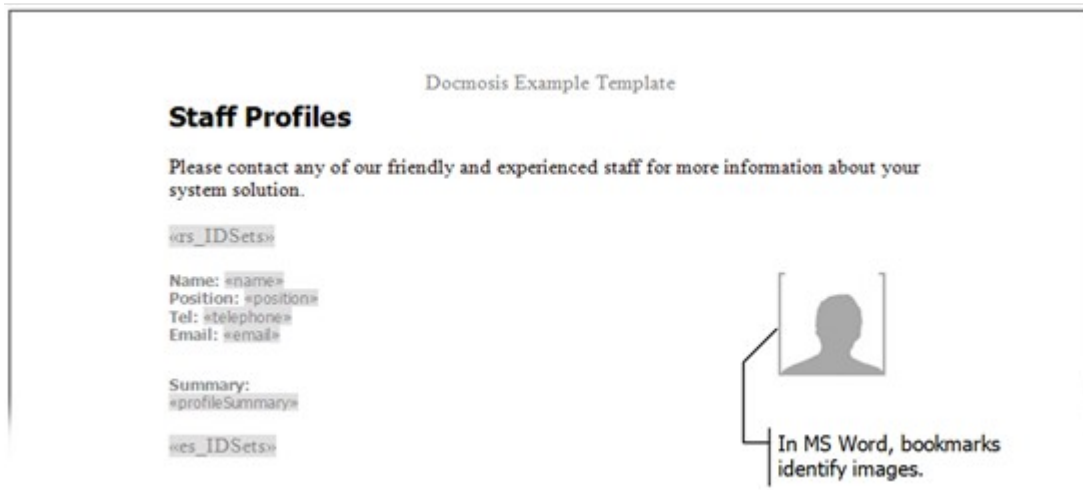
3.6 Using Images

Docmosis can insert images at arbitrary locations in documents. Instead of using fields to identify the location for an image substitution, Docmosis uses the word processor’s image handling features. By handling images this way, the template can precisely define how the image will be placed and bordered within the resulting document.



As each word processor works slightly differently, there are specific methods for setting up the Docmosis code element:

- In Microsoft Word, Docmosis uses the Bookmarks feature to identify a name for an image; and
- LibreOffice Writer supports the identification of images directly, using a Name property.



In the preceding example, a borderless table is used for layout purposes.



This activity doesn't discuss the actual images that you will publish, only the placeholder image. You may create and use your own image but for your convenience, a placeholder image is provided as part of the Docmosis distribution.

You are free to use it without restriction.



Use Only Inline Images in Microsoft Word

Docmosis cannot support floating images in Microsoft Word because it uses the bookmarks feature to assign a name to an image placeholder. When you position an image using the floating position settings, Microsoft Word removes the bookmark. There are other limitations to how Docmosis supports images, particularly in terms of overlapping with text and other images. This will not cause issues for typical documents.

Using images that sit inline with your text means that at the time a document is generated any surrounding text or fields will move in a more predictable way.

Image File Size

When you insert a placeholder image, you will embed the image in the document. This means there is a copy of the image at every location in which it is placed (not simply a single, referenced copy). To limit the overall size of the template file and to improve the performance of a document generation, you should use relatively simple and small placeholder images to identify the locations without compromising on print quality if the document is to be printed.

Image Placeholder Naming Convention

Image placeholder names are identified using special prefixes. These prefixes are a useful way to distinguish those items that are specific to your Docmosis application and enable you to use the bookmarking and naming features for other items that aren't part of a document generation.

The prefixes you can use are:

Prefix	Example	Effect
img_	img_image1	The image is substituted with the supplied image1 and default scaling is applied. The default scaling is "stretch" and may be changed by Docmosis properties or by parameters when rendering the document.
imgstretch_	imgstretch_image1	The image is substituted with the supplied image1 and stretch scaling is always applied. The image is stretched to be the same size and shape as the place holder image in the template.



Prefix	Example	Effect
imgfit_	imgfit_image1	The image is substituted with the supplied image1 and the image will be scaled to fit the template placeholder whilst preserving image1's aspect ratio.



Inserting an Image Element in Microsoft Word

1. Position the insertion point at the location of the image.
2. Select **Insert > Pictures**.
3. In the **Insert Picture** dialog box, navigate to the location of the placeholder image and select it in the list of files.
4. Click **Insert**.
5. When the image appears in the template, select it and use the reshaping handles to adjust the dimensions of the image.
6. Make sure that the image is selected and click **Insert>Bookmark**.
7. In the **Bookmark** dialog box, type `img_image1` into the **Bookmark Name** field.
8. Click **Add**.

Both bookmark names and image names must be unique in your template since both Microsoft Word and LibreOffice Writer force the name to be unique. If you wish to reference the same image data in your template multiple times, you will have to provide different names by which the image can be referenced. This can be achieved by creating unique template-variables to reference the same image data, then bookmark using the unique template-variable names. For example:

```
<<$pic1=photo>>
```

```
<<$pic2=photo>>
```

creates two template-variables referring to the “photo” data. The image bookmarks can then refer to the unique \$pic1 (var_pic1) and \$pic2 (var_pic2) names as required.



Microsoft Word wraps the content of the bookmark in light coloured, square brackets. To see the bookmark in place, set the Bookmarks option in the Microsoft Word Options dialog box.



Microsoft Word Bookmark names can't contain "\$" characters. To use a Docmosis variable in a bookmark name, use "var_" instead of "\$". For example, use "var_myVar" instead of "\$myVar".

Also, MS Word Bookmarks names can't contain "." characters, so it cannot directly use "nested" lookups (e.g. `person[0].photo`). You can use Docmosis variables to overcome this in conjunction with the tip above about referencing variables in bookmarks.

For example, use:

```
<<$myImage=person[0].photo>>
```

in your template body to set the variable, and

```
img_var_myImage
```

as the bookmark name to link the image data to the template image.



Inserting an Image Element in LibreOffice Writer

1. Position the insertion point at the location of the image.
2. Select **Insert > Image**.
3. In the **Insert Picture** dialog box, navigate to the location of the placeholder image and select it in the list of files.
4. Click **Open**.
5. When the image appears in the template, select it and use the reshaping handles to adjust the dimensions of the image.
6. Make sure that the image is selected and click **Format > Image > Properties**.
7. In the **Picture** dialog box, select the **Options** tab.
8. Type `img_image1` into the **Name** field.
9. Click **OK**.

Both bookmark names and image names must be unique in your template, since both Word and Writer force the name to be unique. If you wish to reference the same image in your template multiple times, you will need to provide different names by which the image can be referenced. This can be achieved by creating unique template-variables to reference the same image data, then bookmark using the unique template-variable names. For example:



```
<<$pic1=photo>>
```

```
<<$pic2=photo>>
```

creates two template-variables referring to the “photo” data. The image names can then refer to the unique \$pic1 and \$pic2 names as required.

3.7 Creating Barcodes

Docmosis can generate barcodes and insert them into your output document. Barcode insertion is the same as image insertion except extra information is provided to specify the type of barcode, resolution etc.

3.7.1 Supported Barcode Formats

The following barcode formats are supported:

- Code39
- Code128
- ITF14

3.7.2 Typical Barcode Example

As a typical example, you firstly put a placeholder image into your template and “mark” it (with a name or bookmark as per section 3.5 above):



Image marked as `'imgfit_dispatchLabel'`.

Your data should supply the barcode value followed by the barcode type, separated by a colon (e.g. in JSON format):

```
"dispatchLabel": "1234567:code128"
```

By including the barcode type “code128” as part of the data, Docmosis would then assume this data item is a barcode and generate a code 128 barcode with the value “1234567”:





3.7.3 Using a “barcode” Field to Specify Barcode Settings

In the above example, the placeholder image determines the size, position and name for the bookmark. The rest of the information is provided by the data at render-time.

It is also possible to provide barcode information with a “barcode” field in the template. A barcode field starts with “barcode:”.

Continuing the above example, the template could specify barcode setting like this:

```
<<barcode:dispatchLabel:code128>>
```

This indicates that dispatchLabel will be a code 128 barcode. The data provided at render-time could then simply be the value (e.g. in JSON format):

```
"dispatchLabel": "1234567"
```

The <<barcode:...>> settings field can appear anywhere in your template, either before or after the placeholder image.

In the Template the placeholder image is marked "img_dispatchLabel" and the template may also include a "<<barcode:" field	Data (JSON example)	Result
	"dispatchLabel": "1234567:code39"	A code39 barcode image with value 123456. The data provided the barcode type and value.
 <<barcode:dispatchLabel:code39>>	"dispatchLabel": "1234567"	A code39 barcode image with value 123456. The template specified the barcode type. The data provided the barcode value.
	"dispatchLabel": "1234567:code39:dpi=1200"	A code39 barcode image with value 123456 and resolution 1200 dpi. The data has specified all configuration.



In the Template the placeholder image is marked "img_dispatchLabel" and the template may also include a "<<barcode:" field	Data (JSON example)	Result
 <<barcode:dispatchLabel:code39:dpi=1200>>	"dispatchLabel": "1234567"	A code39 barcode image with value 123456 and resolution 1200 dpi. The template has specified the barcode type and resolution. The data has specified only the value.
 <<barcode:dispatchLabel:1234567:code128>>		A code128 barcode image with value 1234567. The template has specified the barcode value, type and resolution. This means the barcode is valid without any data and is always the same unless overridden by data.

3.7.4 Barcode Tips

When trying to work out the settings for the barcode, this is the recommended process:

1. Position the placeholder image in the template using the size and orientation that works for your template – the bigger the better for reliable scanning.
2. Use "imgfit_" to mark the placeholder image (e.g. `imgfit_dispatchLabel`) to preserve the aspect ratio of the generated barcode. You can use the "imgstretch" to force the barcode to match your placeholder precisely but this will likely reduce the accuracy of the barcode and may make it harder to scan. You can make the barcodes very small at a high resolution but this may impact the ability to be scanned.
3. Let Docmosis apply the default settings first and see if that produces a good result. If not, then start experimenting. The height, module width and wide factor are settings that change the width of the resulting barcode.
4. The DPI setting typically should be 200 or higher. If you generate a barcode at below 100 dpi the quality is typically too low to scan. The default is 600.
5. Test scanning the output barcode to ensure size/DPI settings produce a machine readable barcode.



3.7.5 Barcode Settings in Detail

Anything about a barcode can be specified in the template with a barcode field, including the value:

```
<<barcode:dispatchLabel:1234567:code128>>
```

Other settings can be appended. For example:

```
<<barcode:dispatchLabel:1234567:code128:dpi=800:orientation=90>>
```

As mentioned previously, any barcode settings in the template can be overridden by the data supplied on a 'per-render' basis. For example, the DPI resolution can be changed dynamically by the data (e.g. in JSON format):

```
"dispatchLabel": "1234567:dpi=1200"
```



The data provided at render time will override any settings specified in a barcode field in the template.

The following settings are common to the supported barcodes.

Common Settings				
Name	Shorthand	Description	Example	Default Value
moduleWidth	mw	Barcode module width as a double value. This defines the width of the narrow bars of the barcode. Typical values are in the range 1.0 – 3.0	mw=2.2	0.19 1.10 for ITF14
doQuietZone	dqz	Whether or not the quiet zone will be displayed around the barcode	dqz=true	False
quietZoneWidth	qzw	The width of the quiet zone in mm.	qzw=2.0	12.0 for ITF14
quietZoneHeight	qzh	The height of the quiet zone in mm.	qzh=2.0	



Common Settings				
height	h	The height of the barcode in mm. Depending on the type of barcode, the barcode value, the module width and other settings, the height influences also the width of the resulting barcode.	h=30.0	10.0 40.0 for ITF14
orientation	o	The orientation of the barcode in degrees, 0 is horizontal. Values allowed are 0, 90, -90, 180, -180, 270, -270.	o=90	0
fontSize	fs	The size of the font for the displayed barcode value. Zero will remove the display of the value.	fs=0	
dpi	dpi	The number of dots per inch (resolution) of the barcode. The higher the resolution the bigger the resulting document and processing time. Typically, you would use the minimum that suits the use of the barcode (e.g. taking into account the printer quality).	dpi=1200	600

Code 128 Specific Settings				
Name	Shorthand	Description	Example	Default Value
none				

ITF 14 Specific Settings				
Name	Shorthand	Description	Example	Default Value
wideFactor	wf	Barcode wide factor as a double value. This defines the factor that wide bars are wider than narrow bars. Typical values are in the range 1.0 – 3.0	wf=2.0	2.5



ITF 14 Specific Settings				
Name	Shorthand	Description	Example	Default Value
bearerBarWidth	bbw	The width of the bearer bar in mm.	bbw=2.0	1.0
displayChecksum	dc	Whether or not a checksum should be displayed in the human-readable part of the barcode.	dc=true	false
checksumMode	cm	The code 39 checksum mode: add, auto, check or ignore	cm=add	

3.8 Creating Active Hyperlinks

Docmosis allows you to insert a hyperlink dynamically into your document. In your template, a naming convention identifies fields you would like to work as hyperlinks.

To create a hyperlink, insert a field starting with "link:" (or "link_"). For example, the following field:

```
<<link:myWebSpace>>
```

will create a hyperlink by looking up the value for the hyperlink using `myWebSpace` in your data.

If your data provides the hyperlink address (eg: in JSON format):

```
"myWebSpace": "http://www.example.com"
```

That would create this: <http://www.example.com> in your finished document.

You may also wish to make the displayed text for your hyperlink different from the URL of the link. You do this in your data by using the pipe (|) symbol to separate the display name from the hyperlink value.



If data included the: display text, a pipe symbol (|) and then the hyperlink (eg: in JSON format):

```
"myWebSpace": "Visit our website|http://www.example.com"
```

That would create this: [Visit our website](http://www.example.com) in your finished document.



Note that in all cases:

- the field name identifies the data item
- the data provides the hyperlink, and optionally includes the display text.

3.9 Using Conditional Sections

Conditional content is content that will appear, or be removed, in the final document depending upon values in the data. If the specified condition is met, the content within the matching conditional section will appear in the document.

An example of the application of conditional content might be in a product description such as that for a motor vehicle in the following illustration.

The specifications described here are fictitious.

Vehicle Specifications

Length:	<<metricLength>> mm
Width:	<<metricWidth>> mm
Engine:	<<metricEngine>> l
Fuel tank capacity:	<<metricFuel>> l

<<es_metric>>

Length:	<<imperialLength>> in
Width:	<<imperialWidth>> in
Engine:	<<imperialEngine>> cu in
Fuel tank capacity:	<<imperialFuel>> gal

<<es_imperial>>

This condition returns the metric specifications

This condition returns the imperial specifications

The conditional sections will display the data that is appropriate for each condition. Metric data will be displayed in the Metric section and Imperial data will be displayed in the Imperial



section. Each conditional section is defined using a pair of fields: a start field and an end field. The general syntax for a conditional section is:

```
<<cs_conditionName>>
```

```
[The text and elements of the conditional section.]
```

```
<<es_conditionName>> or simply <<es_>>
```

The conditional start and end fields are removed from the resulting document and if each field is on a line by itself, the entire line will be removed.

Conditional sections can use expressions, variables and range specifiers. For example:

```
<<cs_{(someValue[0]+someValue[1])>10}>>
```

```
<<cs_{$phoneNum!=null&&$phoneNum!=' '}>>
```

See the tables in *2.4.3 Docmosis Elements* and *2.4.4 Expressions and Functions* for more information.



Note that the { } brackets are applied to the expression only – not including the “cs_”. The expression should evaluate to true or false.

To create a conditional section:

1. Position the insertion point in an empty paragraph at the starting location of the conditional section.
2. Insert the opening condition element into the empty paragraph.
3. Add the boilerplate content and other Docmosis elements into the subsequent paragraphs in the document.
4. Insert the closing condition element into an empty paragraph following the conditional content.
5. Repeat steps 1 through 4 for as many conditions as there are in your application.

3.10 Repeating Sections

In a document, a repeating section is a group of elements in succession where the content changes each time but the layout and format is the same. Docmosis supports several forms of repeating sections: block-level, tables and lists.



Tables and lists are special forms of repeating sections. They are discussed after this section. This section deals specifically with block-level repeating sections.

The most common form of repeating content in Docmosis templates is the “block-level” repeat and it uses the Docmosis repeating section elements. Repeating sections can contain any content desired, and it will be repeated whilst there is data to be displayed.

The general syntax for a repeating section is:

```
<<rs_repeating-section-name>>
```

```
[ The text and elements of the repeating section. ]
```

```
<<es_repeating-section-name>> or simply <<es_>>
```

The example below shows a repeating section named `IDSets`, and it contains a “block” that is a mixture of static text (“Name:”, “Position”, etc.) and fields (`<<name>>`, `<<position>>`, etc.). This block will be repeated as many times as there is data associated with `IDSets`.

The specifications described here are fictitious.

Staff Profiles

```
<<rs_IDSets>>
```

Name: `<<name>>`

Position: `<<position>>`

Tel: `<<telephone>>`

Email: `<<email>>`

Summary:
`<<profileSummary>>`

```
<<es_IDSets>>
```

Repeating sections have a pair of containing elements

The repeating start and end fields are removed from the resulting document and if each field is on a line by itself, the entire line will be removed.

Repeating sections can be “nested” inside other repeating sections to any depth desired. Repeating sections can use variables and range specifiers as appropriate. See the section *2.4.3 Docmosis Elements* for more information on “else” fields, range specifiers and nesting.

To create a repeating section:



1. Position the insertion point in an empty paragraph at the starting location of the repeating section.
2. Insert the opening repeating section element into the empty paragraph.
3. Add the boilerplate content and other Docmosis elements into the subsequent paragraphs in the template.
4. Insert the closing element into an empty paragraph following the repeated content.

When looping over a repeating sections Docmosis will automatically create some built-in variables, such as `$$itemidx` and `$$itemnum`. The value of `$$itemidx` changes each time the data is repeated and the variable will contain the number of times the loop has been repeated, starting from zero. The value of `$$itemnum` contains the number of times the loop has been repeated, starting from one.

For example, given 3 "names" in a "people" array, `$$itemidx` can be used as follows:

```
<<rs_people>>
<<$$itemidx>> - <<name>>
<<es_people>>
```

To generate output that might look like this:

```
0 - James
1 - Jenny
2 - Julie
```

whereas using `$$itemnum` would result in:

```
1 - James
2 - Jenny
3 - Julie
```

3.10.1 Stepping Across in Repeating Sections

Docmosis supports the concept of repeating in "steps". Each "step" is a smaller subset of the complete array of data. For example, consider a simple array of "people" objects as follow:

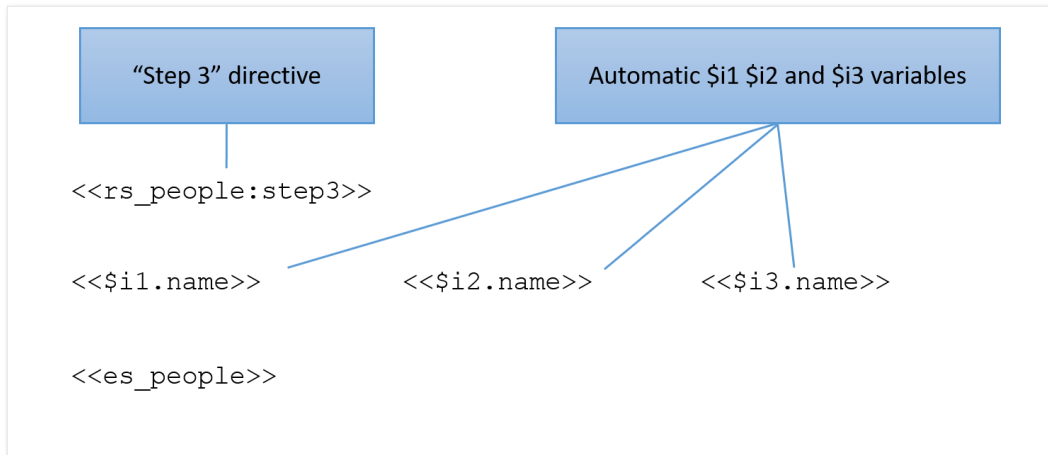
```
"James", "Jenny", "Julie", "Katie", "Kim", "Kerry"
```

in your data. Docmosis can place them on the page in a 3-across layout like this:



James	Jenny	Julie
Katie	Kim	Kerry

The “stepping” allows you to do this in the template as follows:



The `":step3"` directive tells Docmosis to move through the "people" data in groups of 3. Docmosis automatically creates the `$i1`, `$i2` and `$i3` variables to correspond to the first, second and third elements. For the second row, `$i1`, `$i2` and `$i3` will correspond to the fourth, fifth and sixth elements, and so on.

Docmosis will automatically create the `$i...` variables, corresponding to the size of the step being used. For example: for a `":step10"`, variables `$i1`, `$i2`, up to `$i10` will be created.

If you need a 4-across layout instead, this is easily changed in the template by using `":step4"` and adding the 4th column in your template to layout as you require:

```
<<rs_people:step4>>

<<$i1.name>>      <<$i2.name>>      <<$i3.name>>      <<$i4.name>>

<<es_people>>
```

And the resulting document (using the same data) would appear as follows:



James	Jenny	Julie	Katie
Kim	Kerry		

Just as Docmosis automatically creates the `$itemidx` and `$itemnum` variables in a normal repeating section, when stepping Docmosis will automatically create: `$itemidx1`, `$itemidx2...` and `$itemnum1`, `$itemnum2...` variables which relate to the `$i1`, `$i2...` variables. The `$itemidx1..` variables start at zero (i.e. 0,1,2,3...). The `$itemnum1...` variables start at one (i.e. 1,2,3...).

3.10.2 Stepping Down in Repeating Sections

In the same way that Docmosis can present your data in smaller subsets “across” a page, it can also arrange those subsets but moving “down” the page instead.

This means that Docmosis will repeat the items down column1 first, then down column 2, column3 and so on. Docmosis will automatically balance the data into the right number of rows based on the number of steps.

Given the same data as before:

`"James", "Jenny", "Julie", "Katie", "Kim", "Kerry"`

but we wish to show the data down the columns, rather than across, we would use the `step4down` directive in the template like this:

```
<<rs_people:step4down>>

<<$i1.name>>    <<$i2.name>>    <<$i3.name>>    <<$i4.name>>

<<es_people>>
```

In the generated document, notice that the first two item, James and Jenny, appear in column one. The third item, Julie, is displayed at the top of column 2 and so on:



James	Julie	Kim	Kerry
Jenny	Katie		

The “step” functions allow the template to control more of the presentation options given the same set of data.



The above examples use the repeating sections (`<<rs_...>>`) notation, but stepping can also be used with repeating rows (`<<rr_...>>`) below.

3.11 Using Tables

Using fairly simple table syntax, you can create sophisticated table layouts in your output documents. In addition to being able to insert text and images into table cells, using the methods already described, you can use the table-specific Docmosis elements to control:

- including or excluding of groups of rows;
- repeating groups of rows;
- removing columns.

3.11.1 Conditional Rows

A row, or a group of consecutive rows, can be removed from a table using conditional row elements. The following example uses the `<<cr_hasFriends>>` and `<<er_hasFriends>>` fields to identify a single row in a table. The row will appear or be removed depending on the value of `hasFriends`.

Docmosis Example Template	
Circle of Friends	
<code><<cr_hasFriends>></code>	
Jimmy has some friends	
<code><<er_hasFriends>></code>	



In this case, if the data indicates that `hasFriends` is `true`, then the row containing “Jimmy has some friends” would appear in the finished document. In all cases, the rows containing the fields `<<cr_hasFriends>>` and `<<er_hasFriends>>` will be removed:

Docmosis Example Template

Circle of Friends	
Jimmy has some friends	



End markers for conditional rows can also be defined without the name. In the above example, the field `<<er_hasFriends>>` could also be simplified to `<<er_>>`.

3.11.2 Repeating Rows

Rows of a table can be repeated whilst there is data to repeat. The following example will list all of the friends of Jimmy using one row for each friend, showing their name in one column and job in another.

Docmosis Example Template

Circle of Friends	
Friend	Job
<code><<rr_friends>></code>	
Jimmy has a friend called <code><<friend>></code>	<code><<friendJob>></code>
<code><<er_friends>></code>	

In this case, while the data can supply information for `friends`, the row containing the “lookup friend” information will be rendered. In all cases, the rows containing the markers `<<rr_friends>>` and `<<er_friends>>` will be removed.



Docmosis Example Template

Circle of Friends	
Friend	Job
Jimmy has a friend called Dave	Marketing Manager
Jimmy has a friend called Dee	C++ Developer
Jimmy has a friend called Pete	Roof Tiler

3.11.3 Stepping in Repeating Rows

Docmosis supports the concept of repeating in "steps". Each "step" is a smaller subset of the complete array of data.

For example, the `":step3"` directive can be used with a repeating row field as follows:

`<<rr_friends:step3>>`, tells Docmosis to move through the data in groups of 3.

Docmosis automatically creates the `$i1`, `$i2` and `$i3` variables to correspond to the first, second and third elements. For the second row, `$i1`, `$i2` and `$i3` will correspond to the fourth, fifth and sixth elements, and so on.

The examples above of Stepping Across in Repeating Sections (section 3.10.1) and Stepping Down in Repeating Sections (section 3.10.2) also apply to data displayed in a table using Repeating Rows.

3.11.4 Alternating Row Colours and Border Controls

The template for repeating rows also provides some tricks for colour and borders. The rules are as follows:

1. If a cell of a row inside a set of repeating rows has a background colour different to that of the corresponding cell of the starting row (the row with the `<<rr_xxx>>` element), then the background colour for that cell will alternate between that of the starter row and its own background colour. This allows everything from plain tables, to alternating rows to ad-hoc alternating patterns.
2. The starting row (the row with the `<<rr_xxx>>` element) determines the top border of the first repeating row. The ending row (the row with the `<<er_xxx>>` element) determines the bottom border of the last row to be rendered. This applies on a cell-by-cell basis as for the background colouring. This allows for highly configurable borders to be specified.



The following example creates a bounding border encapsulating all the repeating rows (including the marker rows, and alternates for the background colour).

Docmosis Example Template

Circle of Friends	
Friend	Job
<<rr_friends>>	
Jimmy has a friend called <<friend>> <<friendJob>>	
<<er_friends>>	

Notice in the result below the alternating background colours and the border wraps all cells collectively.

Docmosis Example Template

Circle of Friends	
Friend	Job
Jimmy has a friend called Dave	Marketing Manager
Jimmy has a friend called Dee	C++ Developer
Jimmy has a friend called Pete	Roof Tiler



End markers for repeating rows can also be defined without the name. In the above example, the field <<er_friends>> could also be simplified to <<er_>>.

More advanced examples are provided in section 3.11.7 *Advanced Table Structures*.

3.11.5 Disabling Row Alternating

Sometimes alternating row colouring is not desirable. In this case, Docmosis templates can disable the row colouring by using the <<noTableRowAlternate>> directive. The following rules apply the scope of effect of the directive:

1. if <<noTableRowAlternate>> appears anywhere in a table, the alternating colouring is disabled for that table.



2. If `<<noTableRowAlternate>>` appears in the body text of the template (outside of any table) all following tables will have no alternating colouring.

3.11.6 Conditional Columns

A template may indicate columns in a table that are to be conditionally removed. The width of the table remains as fixed in the template and the space recovered by the removal of the column is spread across the remaining columns. The following example shows a Docmosis conditional column element (`<<cc_showJobs>>`) at the top of the second column.

Docmosis Example Template

Circle of Friends	<code><<cc_showJobs>></code>	
Friend	Job	Contact
<code><<rr_friends>></code>		
Jimmy has a friend called <code><<friend>></code>	<code><<friendJob>></code>	<code><<friendContact>></code>
<code><<er_friends>></code>		

When rendered, this removes the column entirely where the data indicates that `showJobs` is false.

Docmosis Example Template

Circle of Friends	
Friend	Contact
Jimmy has a friend called Dave	After hours
Jimmy has a friend called Dee	Anytime
Jimmy has a friend called Pete	Anytime



The following example uses expressions to conditionally remove two columns from the table. If you examine the `Trial 2` and `Trial 3` columns, you will see the expressions in the conditional column fields `<<cc_{nt}>1>>` and `<<cc_{nt}>2>>`, where “nt” will contain a number representing the “number of trials”.

Lab Results	Trial 1	Trial 2 <<cc_{nt}>1>>	Trial 3 <<cc_{nt}>2>>	All Passed
<<rr_specimens>>				
<<name>>	<<results1>>	<<results2>>	<<results3>>	<<passed>>
<<er_>>				

If the incoming data has `nt = 1` then the conditions for the `Trial 2` and `Trial 3` columns will be false and the columns will be removed. Note that the table width remains the same. This is shown in the following example output:

Lab Results	Trial 1	All Passed
SP1A-2474	47.241	No
SP1A-2524	23.442	Yes
SP1A-3211	13.672	No
SP1A-3213	53.342	No

If the incoming data has `nt = 2`, then the `Trial 2` column will remain in the resulting document, but the `Trial 3` column is removed:

Lab Results	Trial 1	Trial 2	All Passed
SP1A-2474	47.241	32.533	No
SP1A-2524	23.442	44.123	Yes
SP1A-3211	13.672	61.153	No
SP1A-3213	53.342	12.223	No



3.11.7 Advanced Table Structures

Docmosis supports the nesting of repeating and conditional content in table structures.

The following example shows three levels of nested data, namely “hotel”, “floor” and “room” to print out the room details for each floor for each hotel.

<<rr_hotels>>		
Hotel <<hotel>>	Location <<location>>	
<<rr_floor>>		
Floor <<floor>>		
Room #	#Beds	Other Features
<<rr_rooms>>		
<<roomNo>>	<nBeds>	• <<roomFeature>>
<<er_rooms>>		
<<er_floorRow>>		
<<er_hotelRow>>		
Footer		

The example above is somewhat extreme and it would often be more natural to represent the structure in a combination of repeating sections and tables with repeating rows. The following template is equivalent but is not providing the entire structure within a single table:

```
<<rs_hotels>>
Hotel <<hotel>> in <<location>>
<<rs_floors>>
Floor <<floor>>
```

Room #	#Beds	Other Features
<<rr_rooms>>		
<<roomNo>>	<nBeds>	• <<roomFeature>>
<<er_rooms>>		

```
<<es_floors>>
<<es_hotels>>
```



3.12 Using Lists

Docmosis infers repetition when there are one or more elements in paragraphs formatted as a list using the ‘bullets and numbering’ features. As long as there is data to populate, the list will be filled with items.

In the following example, the field uses the `[*]` notation. The field is formatted as a numbered list, which will be automatically expanded.

```
Docmosis Example Template

List Expansion Example
These are my friends:
1. <<friends[*].friend>>
```

Docmosis interprets the field in two parts: a repeating component and a lookup component. The repeating component contains a range specifier that specifies multiple values. In this example, the `friends` data item is using the `[*]` range specifier to use “all” friends. The second part of the field, `friend`, is the name used to lookup the data to display.

```
Docmosis Example Template

List Expansion Example
These are my friends:
1. Dave
2. Dee
3. Pete
```

As another example of how the field is split, consider the following template:

```
Docmosis Example Template

List Expansion Example
My first friend <<friends[F].friend>> has the following pets:
• <<friends[F].pets[*].type>>
```



The element now is in a bullet list style rather than numbered. It has a repeating component `friends[F].pets[*]` meaning all pets of the first friend and a lookup component `type`. The resulting document is shown below where the friend has a dog and a parrot.



Docmosis only allows a single component of the element to be a multi-valued range. For example, Docmosis would not allow an element `friends[*].pets[*]` since this would repeat at multiple stages and typically would be a mistake.

To create a list:

1. Position the insertion point at the location of the first list item.
2. Format the paragraph as a list item (bulleted or numbered).
3. Add the Docmosis element that will render the data into the list paragraph.

3.13 Using Headers and Footers

It is possible to use the headers and footers features of Microsoft Word or LibreOffice when creating templates. Any static text, images anchored in the header/footer or word processor features (such as page numbering) can be used in headers and footers and will appear in the finished document.

For example, a document that had a branch name and address on the bottom of every page, could use the header/footer feature as follows:

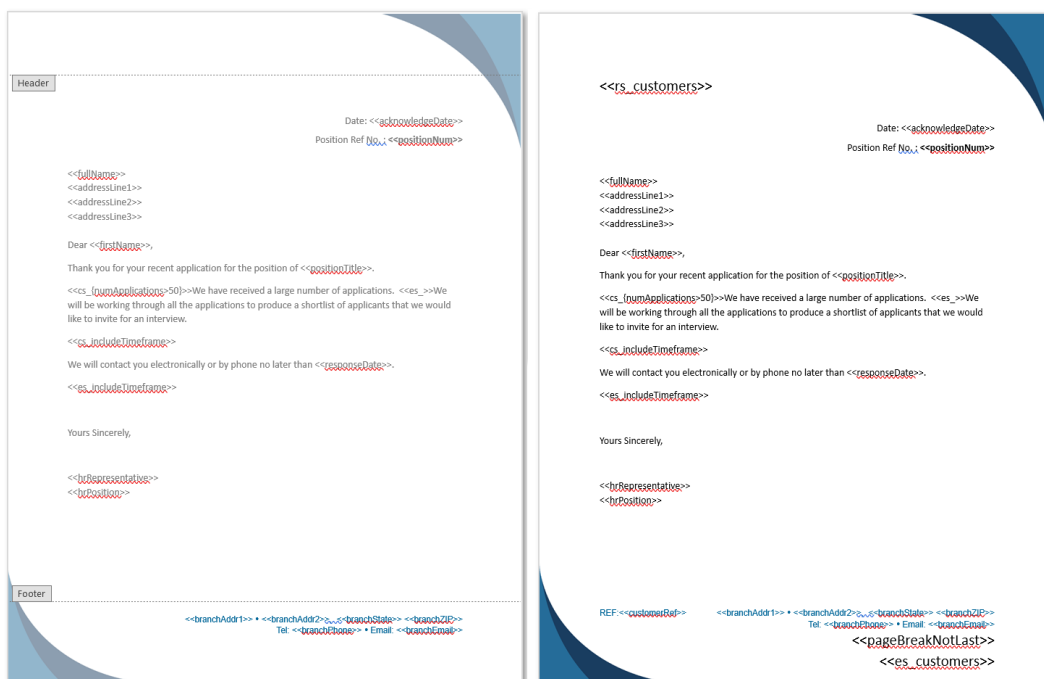




Any fields in the headers and footers are populated outside of the page-by-page flow of the document. This means there is no connection between the fields in the headers and footers and the data being populated on any particular page. You cannot use specific parts of your data in a header and footer and expect it to change from page to page.

If you had page specific information that you want to include at the top or bottom of a page, the header/footer feature may not be needed. For example, if you are trying to generate one letter, per customer, per page, where there is a customer reference number in the footer, you could not reference the details about a particular customer in the header/footer for the corresponding page.

To achieve this type of result, you would extend the margins for the body of the page so they extended in to and cover the header/footer area of the page. This then allows you to position content where a traditional header/footer would be placed. You can also use a repeating section around the whole page, and use fields that do include page specific information at the top or bottom of your page, similar to below:



The example template on the left uses the word processor header/footer feature. Whereas the template on the right “simulates” a header/footer, using extended body margins, and placing content in the header/footer positions.



3.14 Using Comments in Templates

Docmosis supports comments in templates. Comments are sections of the templates that are ignored by document processing and never appear in the output document. If a comment appears on a line by itself, the entire line is removed.

Comments are useful for:

- Creating permanent notes in the template that are helpful to template authors.
- Disabling sections of templates temporarily to assist with development and maintenance.

Due to the two distinct requirements for comments above, there are times where you may need to comment out a section of a template which itself contains other comments. For example, you are temporarily disabling a section of the template which happens to contain comments that are permanent. To support this, Docmosis provides two distinct comment delimiters as shown in the following table. Different delimiters may be used to nest comments inside other comments.

Start Delimiter	End Delimiter
<<##	##>>
<</*	*/>>

Comments can span multiple lines and they are always included as plain text in the document. Merge fields and Document fields cannot be used to create comments.

As an example, a template comment may look as follows:

```
<<## This is a comment block and will not be displayed. ##>>
```

3.15 Merging Templates Together

Docmosis has the ability to combine multiple templates into one resulting document. This gives template authors the ability to separate "common content" out of templates and into a "shared" or common sub-template. The common information then only needs to be maintained in one template and all referencing templates will automatically use the new information.

Examples of use include company information including name/address/logos in a header, standard contractual clauses or signature blocks/images for the body, or possibly copyright notices or contact details at the bottom of each page used in a footer.

Templates used in this way can include all typical content including styled text, headings, tables, images etc. Docmosis will populate the sub-templates as per normal using the data



that applies at the point of insertion, as if it were content in the main template. Any number of sub-templates can be included, and included sub-templates may include other sub-templates.

The way to control this is to insert a reference in your master template to another sub-template. The referenced sub-template will be populated with data and then inserted at the referenced location into the master template. For example, given a master template `MainProcess.doc` that references two sub-templates `process1.doc` and `process2.doc`, Docmosis will insert `process1.doc` and `process2.doc` into `MainProcess.doc` as it processes `MainProcess.doc`.

Docmosis supports two ways of referencing templates; directly and indirectly. Each of these is explained in the following sections.

3.15.1 Direct Referencing

Direct referencing is very simple. The name of the sub-template is known when the master template is being created.

The name of the sub-template is literally used in the reference. The way to create a direct reference is to use a field prefixed with “`ref:`”.

For example `<<ref:process1.doc>>` will cause the template `process1.doc` to be inserted. The following example shows how this would look in a template:

Docmosis Example Template

Template Merging Example

Our main process is detailed in the sections to follow.

`<<ref:process1.doc>>`

3.15.2 Indirect Referencing

With an indirect reference, the name of the sub-template will be provided by the data.

To create an indirect reference, the field prefix “`refLookup:`” is used.

For example, if we created a field `<<refLookup:process>>`, Docmosis will use the value of the data item “`process`” as the name of the template to insert into the document. The following example shows how this would:



Docmosis Example Template

Template Merging Example

Our main process is detailed in the sections to follow.
<<refLookup:process>>

If used with the following data (eg: in JSON format):

```
"process": "process1.doc"
```

this would cause the sub-template `process1.doc` to be inserted.

3.15.3 Templates in Different Locations

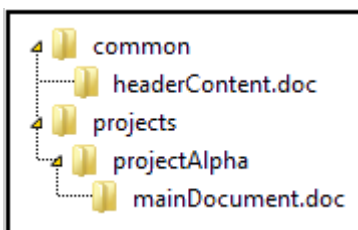
By default, Docmosis assumes the referenced sub-templates are stored in the same location as the referring template. In our scenario above, `process1.doc` must exist in the same location as the master-template.

Docmosis also allows the sub-templates to be referenced in any context using the familiar path notation.

For example, consider a project, `projectAlpha`, which has its own templates that are stored in the location `"projects/projectAlpha"`.

We want to include a common header in our templates, and so we separate the header layout/content into a separate sub-template called `headerContent.doc`. We realise that this new sub-template applies to other projects and is not exclusive to `projectAlpha`.

We decide to create an area `"common"` to store the sub-templates that are common to all projects. This results in a structure like this:



Templates like `mainDocument.doc` can reference the sub-template `headerContent.doc` using the field:

```
<<ref:/common/headerContent.doc>>
```

Note the leading `"/"` above. Or you can use the field:

```
<<ref:../..common/headerContent.doc>>
```




Which is using the common `..` notation to “go up one level” from where the master-template is located.

The examples just listed use the direct `ref:` field prefix; however the same result can be achieved using the indirect `refLookup:` field prefix if the data provides the appropriate value.

Here are some example template references and what they mean.

Field	Description
<code><<ref:template1.doc>></code>	template1.doc is expected to be in the same location as the calling template.
<code><<ref:/template1.doc>></code>	template1.doc is expected to be in "root" context. The root context is the parent of all other contexts.
<code><<ref:/common/template1.doc>></code>	template1.doc is expected to be in the "common" context one down from the root context
<code><<ref:../template1.doc>></code>	template1.doc is expected to be in the parent context of the calling template.

3.15.4 When a Template Cannot be Found

Docmosis treats a missing template as an error. During the rendering of a document, if a reference to a sub-template is encountered and the sub-template cannot be found then an error will be raised. Depending on Docmosis configuration this will either write the error into the resulting document, or generate no document at all and return an error (see section 2.5 *Error Handling*).



In some cases, it is possible for a template to reference a sub-template that doesn't exist and for processing to occur with no errors. This can happen if the reference to the sub-template was inside a conditional section that gets skipped. For example, if `includeSub` is false in this case:

```
<<cs_includeSub>> <<ref:nonExistantSubTemplate.doc>> <<es_>>
```

3.15.5 Continuing Numbered Lists Across Templates

When inserting a sub-template containing part of a numbered list in to a master template that also has a numbered list, you can instruct Docmosis to join the numbered lists together when the sub-template is inserted by using the `<<list:continue>>` directive.



For example, if your master template looked like this:

```
Master Legal Document

1) First Legal Clause
2) Second Legal Clause

<<ref:commonClauses.docx>>
```

And your sub-template (`commonClauses.docx`) contained a numbered list, where the first numbered item has the `<<list:continue>>` field, like this:

```
And in the case where <<reason>> applies:

1) <<list:continue>>Another Legal Clause
2) And Another Legal Clause
```

Docmosis will generate this output, where the numbered items from the sub-template will continue the sequence from the master template:

```
Master Legal Document

1) First Legal Clause
2) Second Legal Clause

And in the case where [use sub list] applies:

3) Another Legal Clause
4) And Another Legal Clause
```

Without the `<<list:continue>>` field, Docmosis would treat the list in the sub-template as a new list and numbering would start from 1) again.

The `<<list:continue>>` field can also be used to continue multi-level level lists (such as: 1, 1.1, 1.1.1, 1.1.2, 1.2, 2, 2.1, etc.).



3.15.6 Limitations

There are some practical limitations to the ability to include sub-templates. The following sections cover the fundamental limitations.

3.15.6.1 Headers and Footers

The master-template defines the headers and footers that will be used throughout the generated document. Any headers and footers in the sub-templates will be ignored. This doesn't mean the content of headers and footers can't be determined by included templates, but the presence and overall properties are controlled by the main template.

3.15.6.2 Performance

The flexibility and maintainability provided by the Docmosis template merging feature has a processing overhead at runtime. Since significantly more work needs to be done to generate the final combined document it is not surprising that there is a runtime cost. The impact will vary depending upon the numbers and sizes of included documents.

3.15.6.3 Styling Limitations

There are some tricks to learn about styling with regards to including sub-templates. One example is where the included template has a Heading style for the first line. Depending on the versions of LibreOffice Writer in use, the Heading style may get dropped unless there is a leading blank line first. This is fairly minor in practice, but something to be aware of. Most of the time, the behaviour will be as expected and so experimentation will only occasionally be required.

3.15.7 Page Breaks and Other Breaks

Docmosis templates may contain several types of break including page breaks, column breaks and section breaks. If the break is in your template it will appear in your rendered documents unless you condition it out with a conditional section. If the break is inside a repeating section it will be repeated each time your repeating section is displayed.

To allow templates to be more expressive, Docmosis provides several fields that can be used to render a break, but without having to place the break literally into the template:

Field	Description
<<pageBreak>>	Insert a page break at this location in the document.
<<columnBreak>>	Insert a column break at this location in the document. This only applies to templates that have a multi-column page layout.
<<pageBreakNotLast>>	Insert a page break at this location in the document unless we have finished repeating the current repeating section. This is



Field	Description
	only valid within a repeating section.
<<columnBreakNotLast>>	Insert a column break at this location in the document unless we have finished repeating the current repeating section. This is only valid within a repeating section and within a page layout that is multi-column.

For example, a template section repeating person details and desiring to put each person on a separate page could look like this:

```
Docmosis Example Template

<<rs_people>>
First Name:    <<firstName>>
Last Name:     <<surname>>
<<pageBreakNotLast>>
<<es_people>>
```

3.16 Creating Pre-filled PDF Forms

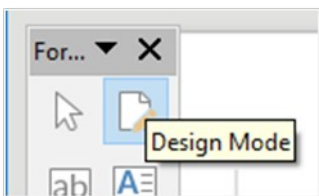
Docmosis supports the creation of PDF forms, optionally pre-filled with data. PDF forms can be useful for allowing customers to fill out information.

Docmosis can inject text into PDF form text fields, text areas and checkbox labels. Docmosis can also check or uncheck the checkboxes.



Only ODT (LibreOffice Writer) templates are supported for PDF form creation.

To Create a PDF form, start with a LibreOffice Writer document for your template. You need to make sure the Form Controls are visible: **View > Toolbars > Form Controls**. On the **Form Controls** toolbar, click the “Design Mode” button to be able to add form fields to your template:

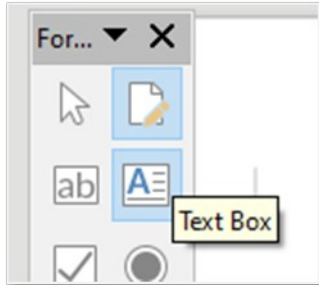


Once you are in design mode, all the controls are enabled and can be added to your template.



3.16.1 Adding a Text Field

1. To add text field, click the text box field:



2. Click and drag the area in your template to create the field. The following example shows creating a field to collect a name in a table:

Name:	
-------	---

3. When the above template is rendered to PDF, there is a name field in which the user can enter text:

Name	Hello - I can type text here
------	--

4. To prompt Docmosis to pre-fill the name, add the `<<name>>` field into the new text box in the template:

Name	<code><<name>></code>
------	---

The next time the template is rendered, if name data exists, it will be pre-populated into the form:



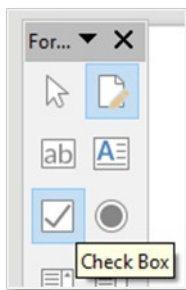
Name	Agent Smith
------	-------------

The PDF is now a pre-filled form, but can still be edited and adjusted as required.

3.16.2 Adding a Checkbox

When working with checkboxes, the same process applies.

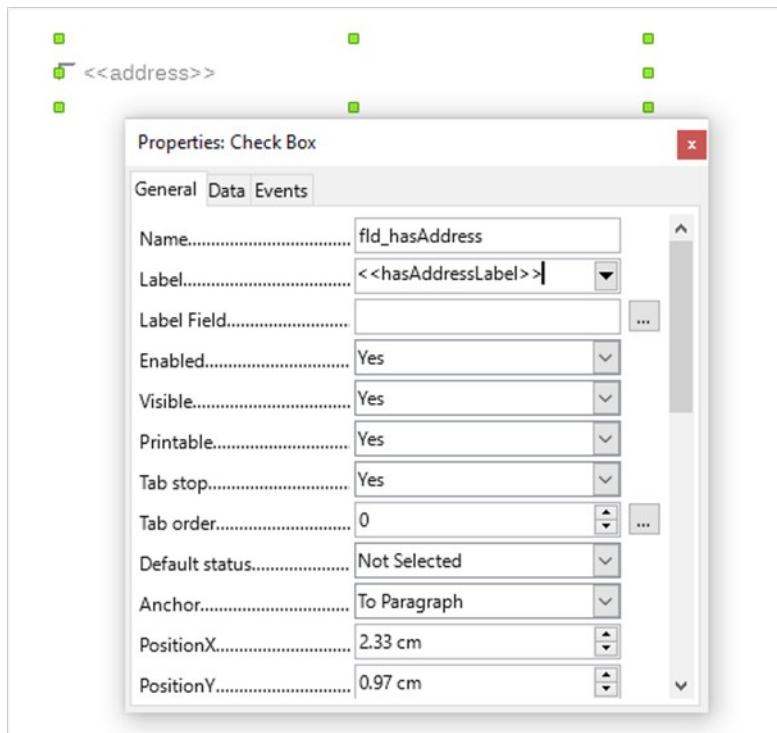
1. Select the **Check Box** button:



2. Click and drag an area on your template to create the space for the checkbox and its label:

☐		☐
☒ Check Box	☐	☐
☐	☐	☐

3. Double click on the field to access its properties. In the example below, we have set the name to "fld_hasAddress" and the label to <<hasAddressLabel>>:



The “fld_hasAddress” name will cause Docmosis to look up the data for the value of “hasAddress” and use the value to tick or untick the box when the document is generated. The “fld_” prefix is what tells Docmosis to look up the value dynamically.

The Label looks like a Docmosis field and will be dynamically populated when the document is rendered. If static text for the label is required, simply type the text and omit the << and >> delimiters.

The above example, when rendered with data:

```
hasAddress=true, and
```

```
hasAddressLabel="The name has been provided"
```

creates a pre-filled PDF form with the checkbox ticked and the label set to the data-provided text:

Name	value1
------	--------

☒ The name has been provided



4 Formatting Data

You may pass either formatted data or unformatted data to Docmosis. Docmosis has the ability to format data for presentation in your finished documents, such as formatting numbers, currency or dates and displaying symbols for true or false boolean values. This section will discuss the advantages and disadvantages of both options.

For example, in your finished document you may wish to display the following:

```
Rental Terms  
Weekly Rent: $1,200.00
```

Data could be formatted before passing to Docmosis: “\$1,200.00”, or unformatted: “1200”, and then the formatting can be controlled in the template.

The advantage of passing formatted data is that the template syntax can be very simple:

```
Rental Terms  
Weekly Rent: <<rent>>
```

However, the means you must do the work to format the data before passing it to Docmosis.

Alternatively, you may have to pass the raw data, in which case you can apply formatting in the template:

```
Rental Terms  
Weekly Rent: <<{numFormat(rent, '$#,###.00' )}>>
```

The advantage with this approach is that future changes to the formatting of the output can be made by adjusting the template, however the syntax in the template is more complex, which may not suit your template authors. Formatting can be applied using the functions that are part of the Docmosis expression processing (Functions 2.4.4.5 Date Functions) or by using Docmosis “renderers”.



Where possible, it is recommended to use the Date Functions (Section 2.4.4.5) available as part of the Docmosis Expression processing, in place of “renderers”. This is discussed further below.

4.1 Formatting Numbers

Given numeric data (even from a text source like XML), you can instruct Docmosis to format it by providing a formatting “string”. The string describes how you want the number to appear.

Below are examples of using the `numFormat` function (2.4.4.4 Numeric Functions), with the following data (eg: in JSON format):

```
"myVal": "12.345"
```

Example	Result
<code><<{numFormat(myVal, '0.0')}>></code>	12.3
<code><<{numFormat(myVal, '\$0.00')}>></code>	\$12.34
<code><<{numFormat(myVal, '0.0E0')}>></code>	1.2E1
<code><<{numFormat(myVal, '###.##')}>></code>	12.34
<code><<{numFormat(myVal, '000.0000')}>></code>	012.3450

4.1.1 The Number Formatting String

The second parameter to the `numFormat` function is the formatting “string” that should be constructed using the characters shown in the table in Appendix 1 - Number Formatting Codes.

The placement and meaning of each character within the string will determine how the input data will be formatted and displayed in the finished document.

Example 1

The formatting string: ``#, ##0.00'`, will be interpreted as follows:

- The ``.`` point character is used to indicate the position of the decimal point.
- The two ``0'` characters to the right of the point indicate that a digit should always be displayed in those positions. In the case where the number being formatted is an integer or only has one decimal place, then the result will be padded with trailing zeroes to the right.



- The single `'0'` to the left of the point indicates that at a minimum of one digit should always be displayed to the left of the point. In the case where the number being formatted is less than one, then a leading zero will be used in the result. (e.g.: An input of `".12"` will display as `"0.12"`)
- The output number will always use as many leading digits as is needed to correctly encode the number.
- There are three characters between the `' , '` comma and the `' . '` point. They are two `'#'` and one `'0'`. The `'#'` indicates: if there is a digit to display at this position then display it, otherwise display nothing. These three characters will cause the digits to the left of the decimal point will be grouped in blocks of three, effectively creating comma separated blocks for the thousands.

Example 2

The formatting string: `'0000'`, will be interpreted as follows:

- There is no `' . '` point character so all numbers will be output as integers.
- There will always be at least four digits as the `'0'` character is used four times. If the input number only has 1, 2 or 3 digits – then the output will be padded with leading zeroes.
- If the input contains more than four digits then it will expand so that the whole number is displayed.



If Docmosis cannot decode or encode the data because there is a mismatch between the data and the formatting string, Docmosis will deal with the error as discussed in section 2.5 Error Handling.

4.1.2 Locale-Specific Formatting

Numbers may be displayed or interpreted differently depending upon the rules and conventions of a specific geographic location or locale.

For example

- In the USA, `' . '` is used for the decimal point and a comma `' , '` for the thousand's separator.
- In Germany it is the other way around. The thousand's separator is a `' . '` And the comma `' , '` is used for the decimal.

The third, and optional, parameter to the `numFormat` function is a code representing a `locale`. If `locale` is not specified then by default the locale of the current environment will be used.

The table in Appendix 2 – Date and Number Formatting Locales lists all the values that may be used to specify the `locale` using a string to represent either country, language or code.

By default, the `locale` is considered when:



- interpreting or decoding data.
- formatting the output.

Below are some examples:

Example	Description	Result
<code><<{numFormat("1234,5", "#.###,#", 'DE')}>></code>	Data is in German format and output is using German format.	1.234,5
<code><<{numFormat("1234.5", "#.###.#", 'US')}>></code>	Data is in US format and output is using US format.	1,234.5

The formatting “string” can also include locale-specific features such as the ‘¤’ character. If you use the ‘¤’ character in your formatting string, Docmosis will replace this with the currency sign for the locale being used.

For example:

Example	Result
<code><<{numFormat('1234.5', '¤#,###.00', 'GBR')}>></code>	£1,234.50
<code><<{numFormat('1234,5', '#.###,00¤', 'DE')}>></code>	1.234,50€
<code><<{numFormat('1234.5', '¤#,###.00', 'USA')}>></code>	\$1,234.5

The fourth, optional, parameter to `numFormat` is a boolean to indicate if the incoming data should be interpreted using the `locale`. This value defaults to ‘true’, so in all the above examples the data was displayed using the locale specified AND was decoded using the `locale`.

For example:

Example	Result
<code><<{numFormat('12.345', '#.###,0', 'DE', 'true')}>></code> The data is expected to be provided in German format; The value of this number is: “twelve thousand, three hundred and forty-five”; The output was kept to one decimal place, in German format;	12.345,0
<code><<{numFormat('12.345', '#.###,0', 'DE', 'false')}>></code> The data is decoded using the platform format (in this case US); The value of this number is hence “twelve point three four five”; The output was kept to one decimal place, in German format;	12,3



The final parameter to `numFormat` is a boolean to indicate if the given format string should be interpreted using “localized” characters of the given `locale`. This value defaults to `'true'`, so by default the format is expected to be specified in localized characters.

For example, when using the FRENCH locale, the format character used to specify the year is ‘a’ rather than the common character ‘y’.

4.1.3 Using a Number Renderer

Prior to the `numFormat` function, Docmosis used a different syntax to format numbers, called a “renderer”.

The parameters to the built-in number renderer are identical to the `numFormat` function described above, however the syntax in the template looks different.

The equivalent syntax for:

```
<<{numFormat(value,format[,locale[,applyLocaleToInput]])}>>
```

Using a renderer would be:

```
<<value{renderer=number(format[,locale[,applyLocaleToInput])}>>
```

4.2 Formatting Dates

Given time and date data you can instruct Docmosis to format it by providing a formatting “string”. The string describes how you want the time and/or date to appear in the finished document.

Below are examples of using the `dateFormat` function (2.4.4.5 Date Functions), with the following data (eg: in JSON format):

```
"myDate": "27-May-2020"
```

Example Field	Result
<code><<{dateFormat(myDate,'dd/MM/yyyy')}>></code>	27/05/2020
<code><<{dateFormat(myDate,'MMM dd, yyyy')}>></code>	May 27, 2020
<code><<{dateFormat(myDate,'EEE, dd MMM yyyy')}>></code>	Wed, 27 May 2020
<code><<{dateFormat(myDate,'yyyy')}>></code>	2020
<code><<{dateFormat(myDate,'EEEE, dd 'of' MMM')}>></code>	Wednesday 27 of May
<code><<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','US')}>></code> <code><<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','ITALY')}>></code>	Wednesday mercoledì



Example Field	Result
<code><<{dateFormat(myDate, 'EEEE', 'dd-MMM-yyyy', 'GERMANY')}>></code>	Mittwoch
<code><<{dateFormat(myDate, 'EEEE tt/MMM/yyyy', 'dd-MMM-yyyy', 'GERMANY', 'US', true, true)}>></code>	Mittwoch 20/Mai/2020 (‘t’ is the day specifier and ‘u’ is the year)

Below are examples of using the `dateFormat` function (2.4.4.5 Date Functions), with the following data (eg: in JSON format):

```
"myDate": "27-Mai-2020"
```

Example Field	Result
<code><<{dateFormat(myDate, 'EEEE', 'dd-MMM-yyyy', 'US', 'de')}>></code>	Wednesday
<code><<{dateFormat(myDate, 'EEEE', 'dd-MMM-yyyy', 'ITALY', 'de')}>></code>	mercoledì
<code><<{dateFormat(myDate, 'EEEE', 'dd-MMM-yyyy', 'GERMANY', 'de')}>></code>	Mittwoch

The final parameter specifies the German ('de') input format, so the date string 27-Mai-2020 can be successfully processed.

4.2.1 The Date Formatting String

The `dateFormat` function accepts four optional parameters. The first two optional parameters specify the output format "string" and the input format "string". The strings should be constructed using the characters shown in the table in Appendix 3 - Date Formatting Codes.

The placement and meaning of each character within the string will determine how the input data will be formatted and displayed in the finished document.

- If no output formatting "string" is provided – Docmosis will use the default format: `'dd MM yyyy'`, hence the result in the first row of the table below.
- If no input formatting "string" is provided – Docmosis will attempt to use a set of common date formats to decode the data.
- It is possible to include characters in the formatting string that "survive" the formatting process and are not interpreted or replaced by Docmosis, by surrounding the characters in two apostrophes: `'`. These appear vertical and they are different to open and closing single quotes ``` and `'`, that slope left and right. Many word processors will convert apostrophes to single quotes – so often the easiest way to add an apostrophe is to cut/paste it in to the string.



- To allow for formats that include a space, the underscore character (_) can be used and this will be converted by Docmosis in to a space. To include an underscore, the backslash character can be used to indicate that the underscore should be left as an underscore (_).

Below are further examples of using the `dateFormat` function with the following data (eg: in JSON format):

```
"myDate": "25-May-2020"
```

Example	Description	Result
<code><<{dateFormat(myDate)}>></code>	No output string so 'dd MM yyyy' is used.	25 May 2020
<code><<{dateFormat(myDate, 'MMM_dd,_yyyy')}>></code>	Underscores are replaced with spaces.	May 25, 2020
<code><<{dateFormat(myDate, 'EEEE, dd 'of' MMM')}>></code>	The word 'of' is between two apostrophes.	Monday, 25 of May



If Docmosis cannot decode or encode the data because there is a mismatch between the data and the formatting string, Docmosis will deal with the error as discussed in section 2.5 Error Handling.

The `dateFormat` function also accepts a further two optional parameters. The first can be used to specify the locale of the output format. This will change the language that the date is rendered into. For example, we can write out the date as typical for the German locale (again using data "myDate": "25-May-2020"):

Example	Description	Result
<code><<{dateFormat(myDate, null, null, 'German')}>></code>	The null values for the output and input format mean use defaults for the format.	27 Mai 2020
<code><<{dateFormat(myDate, 'EEEE', null, 'de')}>></code>	The 'EEEE' format in the output prints the long day name.	Mittwoch



The table in Appendix 2 – Date and Number Formatting Locales details how the locales can be specified.

If “myDate” was specified using a German string, then the 4th optional parameter can be specified to process the input value:

```
"myDate": "27 Mai 2020"
```

Example	Description	Result
<code><<{dateFormat(myDate, null, null, 'de', 'de')}>></code>	The null values for the output and input format mean use defaults for the format.	27 Mai 2020
<code><<{dateFormat(myDate, null, null, 'US', 'de')}>></code>	Process myDate in German and write out for the US locale.	27 May 2020

The `dateFormat` function's final two parameters specify whether the format string is to be interpreted using the Locale-specific characters of the locale specified. Normally, the year is specified with the 'y' character. However if, for example, the locale FRENCH is being used, then it may be the 'a' character is being used for the year in the format string. In this case, the final two parameters can be used to specify whether the `outputFormat` and `inputFormat` (respectively) should be interpreted using pattern characters specific to the locale. By default, both parameters are false.

4.2.2 Using a Date Renderer

Prior to the `dateFormat` function, Docmosis used a different syntax to format dates, called a “renderer”.

The parameters to the built-in date renderer are identical to the `dateFormat` function described above, however the syntax in the template looks different.

The equivalent syntax for:

```
<<dateFormat(value[,outputFormat[,inputFormat]])>>
```

Using a renderer would be:

```
<<value{renderer=date([outputFormat[,inputFormat]])}>>
```



4.3 Formatting Booleans

Docmosis includes some built-in renderers for formatting: Booleans. If your data contains true/false values, in your finished document you can use a renderer to display Yes/No, Y/N or even a tick/cross symbol.

4.3.1 Built-in Boolean Renderer

Docmosis can format boolean (true/false) data into several presentational styles using the built-in "boolean" renderer. This renderer exists since true/false can often be better displayed in a document by Yes/No, Y/N, tick/cross etc.

The built-in Boolean render takes a single parameter indicating the way the true and false values should be displayed. The following table lists the built-in values that may be passed as a parameter.

Parameter	Effect
No parameter	true is rendered as "true" and false as "false"
"yn"	true is rendered as "Y" and false as "N"
"ynlc"	true is rendered as "y" and false as "n" (lower case)
"yesno"	true is rendered as "Yes" and false as "No" (mixed case)
"yesnolc"	true is rendered as "yes" and false as "no" (lower case)
"yesnouc"	true is rendered as "YES" and false as "NO" (upper case)
"wingdings1"	true is rendered as a wingdings tick and false as a cross
"wingdings2"	true is rendered as a wingdings checkbox ticked and false as unticked
"dingbats1"	true is rendered as a dingbats light tick and false as a light cross
"dingbats2"	true is rendered as a dingbats heavy tick and false as a heavy cross

For example, to use the `yn` Boolean renderer, a field may attach a renderer named "boolean" and give it the 'yn' parameter:

```
<<isRetired{renderer=boolean('yn')}>>
```

which will render true as Y and false as N.

The Boolean renderer will also try to parse textual data into a Boolean value to allow a renderer to control the way it displays. For example, t, y, yes, and 1 are all considered "true".



Using specific fonts may cause issues if you then migrate your system to another platform. You should always confirm the operation of your template in all its intended system environments.

4.3.2.1 Dingbats Boolean Renderer

The Boolean renderer also provides the ability to render Dingbats character replacements. Using Dingbats does not require any changes to fonts in the templates as is required for the Wingdings formatter.

You may find that Word does not understand the Dingbats characters, so if you choose Word as the output document type, typically you would not use Dingbats character renderers. The Dingbat Boolean renderer can be used with Microsoft Word or LibreOffice Writer templates, the effect is only an issue in the output document.



5 Appendices

Appendix 1 - Number Formatting Codes

The string used to describe the format of a number (section 4.1 Formatting Numbers) should be constructed using the specific characters shown in the table below.

Character	Location	Localized?	Meaning
0	Number	Yes	Digit
#	Number	Yes	Digit, zero shows as absent
.	Number	Yes	Decimal separator or monetary decimal separator
-	Number	Yes	Minus sign
,	Number	Yes	Grouping separator
E	Number	Yes	Separates mantissa and exponent in scientific notation. Need not be quoted in prefix or suffix.
;	Subpattern boundary	Yes	Separates positive and negative subpatterns
%	Prefix or suffix	Yes	Multiply by 100 and show as percentage
‰ (\u2030)	Prefix or suffix	Yes	Multiply by 1000 and show as per mille value
¤ (\u00A4)	Prefix or suffix	No	Currency sign, replaced by currency symbol. If doubled, replaced by international currency symbol. If present in a pattern, the monetary decimal separator is used instead of the decimal separator.
'	Prefix or suffix	No	Used to quote special characters in a prefix or suffix, for example, "'#'" formats 123 to "#123". To create a single quote itself, use two in a row: "'o'clock'".



Appendix 2 – Date and Number Formatting Locales

The following table lists all the values that may be used to specify the locale by country, language or code. These codes apply using locales in number and date formatting functions (see sections 4.1 Formatting Numbers and 4.2 Formatting Dates).

The locale may be specified as follows:

- Precisely using the Language Code and the Country Code combined. For example 'en_US' or 'en,US' and 'de_CH' or 'de,CH'
- Using the country 'US', 'USA' or 'United States'
- Using the language 'EN', 'Eng', or 'English'

Since using just the language or country might be ambiguous, the following order is applied when matching the locale:

1. First is Country– by code, iso code then name
2. Next is Language – by code, by iso code then name.

Country			Language		
Code	ISO Code	Name	Code	ISO Code	Name
AL	ALB	Albania	sq	sqi	Albanian
AE	ARE	United Arab Emirates	ar	ara	Arabic
AR	ARG	Argentina	es	spa	Spanish
AU	AUS	Australia	en	eng	English
AT	AUT	Austria	de	deu	German
BE	BEL	Belgium	nl	nld	Dutch
BG	BGR	Bulgaria	bg	bul	Bulgarian
BH	BHR	Bahrain	ar	ara	Arabic
BY	BLR	Belarus	be	bel	Byelorussian
BO	BOL	Bolivia	es	spa	Spanish
BR	BRA	Brazil	pt	por	Portuguese
CA	CAN	Canada	fr	fra	French
CH	CHE	Switzerland	it	ita	Italian
CL	CHL	Chile	es	spa	Spanish
CN	CHN	China	zh	zho	Chinese
CO	COL	Colombia	es	spa	Spanish
CR	CRI	Costa Rica	es	spa	Spanish
CZ	CZE	Czech Republic	cs	ces	Czech
DE	DEU	Germany	de	deu	German
DK	DNK	Denmark	da	dan	Danish
DO	DOM	Dominican Republic	es	spa	Spanish
DZ	DZA	Algeria	ar	ara	Arabic



Country			Language		
Code	ISO Code	Name	Code	ISO Code	Name
EC	ECU	Ecuador	es	spa	Spanish
EG	EGY	Egypt	ar	ara	Arabic
ES	ESP	Spain	ca	cat	Catalan
ES	ESP	Spain	es	spa	Spanish
EE	EST	Estonia	et	est	Estonian
FI	FIN	Finland	fi	fin	Finnish
FR	FRA	France	fr	fra	French
GB	GBR	United Kingdom	en	eng	English
GR	GRC	Greece	el	ell	Greek
GT	GTM	Guatemala	es	spa	Spanish
HK	HKG	Hong Kong	zh	zho	Chinese
HN	HND	Honduras	es	spa	Spanish
HR	HRV	Croatia	hr	hrv	Croatian
HU	HUN	Hungary	hu	hun	Hungarian
IN	IND	India	en	eng	English
IN	IND	India	hi	hin	Hindi
IE	IRL	Ireland	en	eng	English
IQ	IRQ	Iraq	ar	ara	Arabic
IS	ISL	Iceland	is	isl	Icelandic
IL	ISR	Israel	iw	heb	Hebrew
IT	ITA	Italy	it	ita	Italian
JO	JOR	Jordan	ar	ara	Arabic
JP	JPN	Japan	ja	jpn	Japanese
KR	KOR	South Korea	ko	kor	Korean
KW	KWT	Kuwait	ar	ara	Arabic
LB	LBN	Lebanon	ar	ara	Arabic
LY	LBY	Libya	ar	ara	Arabic
LT	LTU	Lithuania	lt	lit	Lithuanian
LU	LUX	Luxembourg	de	deu	German
LU	LUX	Luxembourg	fr	fra	French
LV	LVA	Latvia	lv	lav	Latvian (Lettish)
MA	MAR	Morocco	ar	ara	Arabic
MX	MEX	Mexico	es	spa	Spanish
MK	MKD	Macedonia	mk	mkd	Macedonian
NI	NIC	Nicaragua	es	spa	Spanish
NL	NLD	Netherlands	nl	nld	Dutch
NO	NOR	Norway	no	nor	Norwegian
NZ	NZL	New Zealand	en	eng	English
OM	OMN	Oman	ar	ara	Arabic
PA	PAN	Panama	es	spa	Spanish



Country			Language		
Code	ISO Code	Name	Code	ISO Code	Name
PE	PER	Peru	es	spa	Spanish
PL	POL	Poland	pl	pol	Polish
PR	PRI	Puerto Rico	es	spa	Spanish
PT	PRT	Portugal	pt	por	Portuguese
PY	PRY	Paraguay	es	spa	Spanish
QA	QAT	Qatar	ar	ara	Arabic
RO	ROM	Romania	ro	ron	Romanian
RU	RUS	Russia	ru	rus	Russian
SA	SAU	Saudi Arabia	ar	ara	Arabic
SD	SDN	Sudan	ar	ara	Arabic
SV	SLV	El Salvador	es	spa	Spanish
SK	SVK	Slovakia	sk	slk	Slovak
SI	SVN	Slovenia	sl	slv	Slovenian
SE	SWE	Sweden	sv	swe	Swedish
SY	SYR	Syria	ar	ara	Arabic
TH	THA	Thailand	th	tha	Thai
TN	TUN	Tunisia	ar	ara	Arabic
TR	TUR	Turkey	tr	tur	Turkish
TW	TWN	Taiwan	zh	zho	Chinese
UA	UKR	Ukraine	uk	ukr	Ukrainian
UY	URY	Uruguay	es	spa	Spanish
US	USA	United States	en	eng	English
VE	VEN	Venezuela	es	spa	Spanish
YE	YEM	Yemen	ar	ara	Arabic
YU	YUG	Yugoslavia	sh	srp	Serbo-Croatian
YU	YUG	Yugoslavia	sr	srp	Serbian
ZA	ZAF	South Africa	en	eng	English



Appendix 3 - Date Formatting Codes

The input and output date formats (see section 4.2 Formatting Dates) can be created by using combinations of the letters from the table below. Note, these are not “localized” pattern characters – if you wish to use localized pattern characters, please contact Docmosis Support.

Letter	Date or Time Component	Presentation	Examples
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800