Cloud (DWS4) Template Guide

Version DWS4

April 2024

docmosis

TABLE OF CONTENTS

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.docx`	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...
ODT	The information only applies to ODT format templates created with LibreOffice Writer.
DOCX	The information only applies to DOCX format templates created with Microsoft Word.

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 Docmosis Resources website (https://resources.docmosis.com) has tutorials and template examples to help find answers quickly.

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, 'presentational' features can be developed separately from application code, 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.

2.3. How Does Document Generation Work?

In the simplest terms, Docmosis merges the data provided by a software application with the fields in the template to generate 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 the PDF documents or inaccurate page references may occur 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 the 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. Fields

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

`<<name>>`
`<<first-name>>`
`<<{[first-name]}>>`
	Replace this field by the data referenced by "name" or "first-name". Hyphenated names are supported and need to be enclosed in [and] when used in expressions.

`<<## 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.
`<<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.
`<<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"

Element	Description
`<<html:name>>`	Lookup "name" in the data and inject the data as HTML content into the document at this location
`<<pageBreak>>`	Insert a page break at this point. This is an alternative to inserting an actual page break in the template.
`<<pageBreakNotLast>>`	Insert a page break at this point unless in a repeating section and at the last iteration of the repeat.

`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.
`<<barcode:name:…>>`	Provide information for a barcode image in the template. e.g. <<barcode:dispatchLabel:code128>> defines image "dispatchLabel" as a code 128 barcode.
`<<ref:sub1.docx>>`	Insert the template named "sub1.docx" at this location. Template coordination: include "sub1.doc".
`<<refLookup:name>>` `<<refLookupOp:name>>`	Lookup "name" in the data to get the name of the template to insert at this location. Template coordination: include the template identified by the data item "name". The "Op" form does not raise an error if no name is resolved.

Element	Description
`<<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.
`<<coordinator:>>`	Mark this template as a coordinator of other templates. This template will not be part of the output, but instead specifies other templates to be rendered.
`<<coordinator:padToEvenPage>>` `<<coordinator:padToOddPage>>`	Before then next referenced template, if necessary, insert a blank page to make the output document an even or odd number of pages. Currently this only applies to Docmosis Cloud and Tornado when it combines PDF output into a single PDF document.
`<<coordinator:newFile>>`	At this point in processing, create a new file for the following content. This means that, when creating PDF output multiple templates may be combined into a collection of different PDF documents. Currently this only applies to Docmosis Cloud and Tornado when it combines PDF output into a single PDF document.

2.4.3.2. Repeating

Element	Description	Closing Element

`<<rs_items>>`
`<<rs_$abc>>`
	Content between the opening element and closing element is repeated whilst there is data associated with “items” or the variable “abc”.	`<<es_items>>`
		`<<es_$abc>>`
		or
		`<<es_>>` for any match

`<<list:reset>>`	To be used in a repeating section to force a numbered list to restart numbering (rather than continue numbering from previous repeat).	

`<<rr_items>>`
`<<rr_$abc>>`
	In a table, the rows between the opening element row and the closing element row are repeated whilst there is data associated with “items” or the variable “abc”.	`<<er_items>>`
		`<<er_$abc>>`
		or
		`<<er_>>` for any match

`<<noTableRowAlternate>>`	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.	

2.4.3.3. Repeating With Stepping

Element	Description

`<<rs_items:step2>>`
`<<rs_items:step2down>>`
 	
`<<rr_items:step2>>`
`<<rr_items:step2down>>`
	For the "items" data, repeat the content following until the matching closing element. "rs_" applies to general content, "rr_" applies to table rows.
	"stepN" indicates that the data ("items") should be iterated in steps of N size. When stepping is used, the variables $i1, $i2,...$iN are created automatically so items can be referenced in each step. This allows an array of data to be presented across a rows of N columns.
	"stepNdown" indicates that the data ("items") should be iterated in steps of N size and data should be presented in a "down"-ward (columnar) manner. Variables $i1, $i2,... $iN are created automatically. This allows an array of data to be presented in rows of N across, and values are placed filling column 1 first, then filling column 2 etc.

2.4.3.4. Repeating With Filters

Element	Description

`<<rs_persons:filter(ID<10)>>`
`<<rs_product:filter(startsWith(Code, 'AWA'))>>`
 	
`<<rr_persons:filter(ID<10)>>`
`<<rr_product:filter(startsWith(Code, 'AWA'))>>`
	Repeat over the "persons" data after filtering each item by the specified expression.

2.4.3.5. Repeating With Sort

Element	Description
	Repeat over the given data after sorting by the given data-field or expression.
`<<rs_persons:sort(name)>>`	Sort by “name” alphanumerically
`<<rs_persons:sort(DESC, name)>>`	Sort by “name” descending
`<<rs_persons:sort(DESC, CASE_SENSITIVE, NULLS_LAST, name)>>`	Sort by “name” descending, case-sensitive, nulls last
`<<rs_persons:sortStr(name)>>`	Sort by “name” as a string (not numerically smart)
`<<rs_persons:sortNum(ID)>>`	Sort by “ID” as a number
`<<rs_persons:sortDate(DOB)>>`	Sort by “DOB” as a date using defaults
`<<rs_persons:sortDate(DOB, ‘dd/MM/yyyy’, ‘German’, false)>>`	Sort by “DOB” using with format specifiers
`<<rr_persons:sort(name)>>` `<<rr_persons:sort(DESC, name)>>`	Repeat over table rows using sort directives. Sort types: sort – alphanumeric sort sortStr – simple string-based sort sortNum – sort as numeric data sortDate – sort as data data Sort directives: ASC | DESC – ascending or descending (default ASC) CASE_SENSITIVE | CASE_INSENSITIVE (default sensitive) NULLS_FIRST | NULLS_LAST

2.4.3.6. Repeating With Grouping

Element	Description
`<<rs_animals:group(species)>>`	Repeat over “animals” grouped by “species”.
`<<rs_persons:group(dateFormat(DOB,'MM','dd/MM/yyyy'))>>`	Repeat over “persons” grouped by “DOB”.
`<<$groupKey>>`	Inside the repeat, $groupKey is the current grouped term.
`<<rs_$groupItems>>`	Inside the group repeat, repeat over the items in the current group.

`<<rs_animals:group(species)>>`
`Species: <<$groupKey>>`
`<<rs_$groupItems>>`
`  Animal is <<name>>`
`<<es_>>`
`<<es_>>`
	Typical use pattern for grouped data.

`<<rr_animals:group(species)>>`
`Species: <<$groupKey>>`
`<<rr_$groupItems>>`
`  Animal is <<name>>`
`<<er_>>`
`<<er_>>`
	Typical use pattern across rows of a table.

2.4.3.7. Conditional

Element	Description	Closing Element

`<<cs_name>>`
`<<cs_{expr}>>`
`<<cs_$abc>>`
	Content between the opening element and the closing element is included or excluded depending on the value associated with “name” or the expression “expr” or the variable “abc”. The end tag must match exactly, or may be anonymous: <<es_>>.	`<<es_name>>`
		`<<es_{expr}>>`
		`<<es_$abc>>`
		`<<es_>>`

`<<cr_name>>`
`<<cr_{expr}>>`
`<<cr_$abc>>`
	Include the following table rows depending on the value associated with “name” or expression “expr” or the variable “abc”.	`<<er_name>>`
		`<<er_{expr}>>`
		`<<er_$abc>>`
		`<<er_>>`

Element	Description	Closing Element

`<<else_name>>`
`<<else_{expr}>>`
`<<else>>`
	This is the "else" tag related to a <<cs_>> tag to provide the "else" and "else if" options to a condition.	

`<<cc_name>>`
`<<cc_{expr}>>`
`<<cc_$abc>>`
	Include or exclude the table column containing this field depending on the value associated with "name" or the expression "expr" or the variable "abc".	

2.4.3.8. 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.

2.4.3.9. 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
`<<hotel[0]>>`	The first hotel (indexing starts at zero)
`<<hotel[F]>>`	The first hotel (equivalent to index zero)
`<<hotel[L]>>`	The last hotel
`<<hotel[*]>>`	All hotels
`<<hotel[F3]>>`	The first 3 hotels

Element	Description
`<<hotel[L3]>>`	The Last 3 hotels
`<<hotel[1,2,4]>>`	The hotels at indexes 1,2 and 4
`<<hotel[1-3,L2]>>`	The hotels at indexes 1 to 3 inclusive and the last 2
`<<hotel[0-L2]>>`	All but the last 2 hotels
`<<hotel[3].floor[L].room[0].name>>`	The name of the first room of the last floor of the hotel at index 3

2.4.3.10. Built-In Variables

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

Variable	Description
`<<$top>> or <<$root>>`	The root of the data regardless of the current position or context in the template.
`<<$this>> or <<$current>>`	The current source of data in the current position in the template. This allows for anonymous data lookups from arrays or collections such as `<<$current[0]>>`.
`<<$parent>>`	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.
`<<$nl>>`	A simple newline character
`<<$nowMS>>`	Current UTC time in milliseconds since 1/1/1970
`<<$nowUTC>>`	Current UTC time as in ISO 8601 format
`<<$nowUTCFormat>>`	The format used for $nowUTC which is: `yyyy-MM-dd'T'HH:mm:ssX`
`<<$quot>>`	The single-quote character
`<<$templateName>>`	The name of the current template being rendered
`<<$templateFolder>>`	The name of the folder containing the template being rendered

Variable	Description
`<<$templatePath>>`	The full path to the template being rendered (the combination of the name and folder)

2.4.3.11. Built-In Variables When Repeating

These variables available in *Repeating Sections* and *Repeating Rows*:

Variable	Description

`<<$idx>>`
 	
`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 `$itemidx`, however if repeating over a range of values that doesn't start at zero (e.g. `<<rs_names[3-5]>>`), the `$idx` values into the data would be 3,4,5.

`<<$itemidx>>`
 	
`Index in the iteration`
	The current index into an iteration, starting from zero.
	
	This is unaffected by the ranges of the data specified so the `$itemidx` values for `<<rs_names[3-5]>>` is 0,1,2.

`<<$num>>`	The same as `$idx` but starting from one.
`<<$itemnum>>`	The same as `$itemidx` but starting from one.
`<<$size>>`	The size of the current repeating data set. For example, if repeating over all hotels, `$size` would be the number of hotels.
`<<$rownum>>`	The current row number (starting at 1) when repeating (either repeating rows or repeating sections). This is most useful when using the "stepping" directives and the `$itemnum` is not suitable. For more information about the use of "steps of N" see *3.11.1 Stepping Across in Repeating Sections* and *3.11.2 Stepping Down in Repeating Sections*.
`<<$rowidx>>`	The current row number (starting at 0) when repeating (either repeating rows or repeating sections). This is most useful when using the "stepping" directives and the `$itemidx` is not suitable. For more information about the use of "steps of N" see *3.11.1 Stepping Across in Repeating Sections* and *3.11.2 Stepping Down in Repeating Sections*.

2.4.3.12. Built-In Variables When Stepping

The following variables available when “stepping” through repeating data:

Variable	Description
`<<$i1>>,<<$i2>>, … <<$iN>>`	References to the Nth item when repeating data in "steps of N". For example `<<rs_people:step3>>` steps through the people in "steps of 3" and Docmosis automatically creates variables `$i1`, `$i2` and `$i3` to access each element in the step. For more information about the use of "steps of N" see *3.11.1 Stepping Across in Repeating Sections* and *3.11.2 Stepping Down in Repeating Sections*.
`<<$idx1>>, … <<$idxN>>`	Shorthand for `$i1.$idx, … $iN.$idx`
`<<$num1>>, … <<$numN>>`	Shorthand for `$i1.$num, … $iN.$num`
`<<$itemidx1>>, … <$itemidxN>>`	Shorthand for `$i1.$itemidx, … $iN.$itemidx`
`<<$itemnum1>>, … <$itemnumN>>`	Shorthand for `$i1.$itemnum, … $iN.$itemnum`

2.4.3.13. Built-In Variables When Grouping

These variables available when iterating over “grouped” data:

Variable	Description
`<<$groupKey>>`	The key used for the current group of data. Eg if grouping by ProductCode, the key might be “123”.
`<<rs_$groupItems>>` `<<rr_$groupItems>>`	A repeating section or row containing the data within the current group. Eg if grouping by ProductCode and the $groupKey is “123” then this would be a list of every product with the ProductCode “123”.

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
`<<{expr}>>`	Replace this field with the results of the given expression.
`<<{10 * 3.0}>>`	Calculate 10 multiplied by 3.0
`<<{amount * qty}>>`	Lookup data elements "amount" and "qty" and multiply them together.
`<<{round(item/10)}>>`	Lookup data element "item", divide it by 10 then round the result.
`<<cs_{a<10}>>`	Lookup data element "a" and see if it is **less than** 10 numerically. If "a" is not numeric, a string comparison is performed automatically.
`<<cs_{a='fred'}>>`	Lookup data element "a" and see if it is **equal to** the String literal "fred".
`<<cs_{$a!=10}>>`	Lookup the variable "a" and see if it is **not equal to** the numeric value 10. If variable "a" does not resolve to a numeric value, a String comparison is performed.
`<<cs_{a=null}>>`	Lookup the data element "a" and determine if it's value is null
`<<cs_{$a}>>`	Determine if the value of the template variable $a is true

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

Function	Synopsis

`map`	A function to map one value to another.
	`map(key, test1, replace1 [,test2, replace2 ...] [,default])`
	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:
	<<{map(gender, 'M', 'Male', 'F', 'Female', 'Other')}>>
	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).
	`mapi(key, test1, replace1 [,test2, replace2 ...] [,default])`
	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:
	<<{mapi(gender, 'M', 'Male', 'F', 'Female', 'Other')}>>
	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`	Returns the character based on a given code
	`charAt (code)`
	where:
	`code = the character code. Examples below.`
	For Example, all the following insert the copyright symbol:
	`Decimal <<{char(169)}>>`
	`Hex <<{char('0xa9')}>>`
	`Html decimal <<{char('&#169')}>>`
	`Html hex <<{char('&#xa9')}>>`
	`Java/Json format <<{char('\ua9')}>>`
	All return the copyright symbol ©

`charAt`	Returns the character at the requested position in the source string.
	`charAt (string, position)`
	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:
	<<{charAt('abcdefg',3)}>> returns the character "d"
	<<{charAt(idNumber,6)}>>
	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.
	`endsWith (mainString, subString)`
	where:
	`mainString = the string to check`
	`subString = the string to look for at the end of mainString`
	
	For Example:
	<<{endsWith('The first string', 'ing')}>> returns the value "true"
	Useful when creating a conditional section. For example, this conditional section will only display the "serialNum" field if it ends with "ZZZ".
	<<cs_{endsWith(serialNum, 'ZZZ')}>> <<serialNum>> <<es_>>

`equalsIgnoreCase`	Compares to strings, regardless of case.
	`equalsIgnoreCase (string1, string2)`
	where:
	`string1 = the first string`
	`string2 = the second to compare to the first string`
	
	For Example:
	<<{equalsIgnoreCase ('Bob', 'bob')}>> returns the value "true"

`indexOf`	Returns the starting index of one string inside another.
	`indexOf (string, find [, startIdx])`
	where:
	`string = the string to scan`
	`find = the string to find`
	`startIdx = an optional search starting index`
	
	For Example:
	<<{indexOf('Bob Mathews', 'Mat')}>>
	returns "4.0"

Function	Synopsis

`length`	Returns the length of a string.
	`length (string)`
	where:
	`string = the string to check the length of`
	
	For Example:
	<<{length('Bob')}>> 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.
	<<cs_{length(refNo)>0}>> Ref Num : <<refNo>> <<es_>>

`replace`	Replaces characters in the source string with new characters.
	`replace (string, oldChar, newChar)`
	where:
	`string = the string`
	`oldChar = the character to find in the string`
	`newChar = the character to use in place of the oldChar`
	
	For Example:
	<<{replace(customerVIN,'o','0')}>>
	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"

Function	Synopsis

`replaceStr`	Replaces strings in the source string with the new string.
	`replaceStr(string, searchFor, replaceWith [, ignoreCase])`
	where:
	`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).`
	For Example:
	<<{replaceStr(address, 'street','St.', true)}>>
	If address="Matheson street", returns "Matheson St."

`replaceFirst`	Replaces the first occurrence of source string with the new string.
	`replaceFirst(string, searchFor, replaceWith [, ignoreCase])`
	where:
	`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).`
	For Example:
	<<{replaceFirst('Two times Two', 'two', '2', true)}>>
	Results in '2 time Two'.

Function	Synopsis

`countStr`	Count occurrences of a string within another string.
	`countStr (string, searchFor [, ignoreCase])`
	where:
	`string = the string`
	`searchFor = the character(s) to find in the string`
	`ignoreCase = true|false to ignore the case when searching. Defaults to false (ie case-sensitive).`
	For Example:
	<<{countStr("Bob", 'b', true)}>>
	Returns "2".

`split`	Split a string into parts that can be displayed separately.
	`split (string, splitChar, index)`
	where:
	`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.`
	For Example:
	<<{split('John|Mathews|47|Approved' , '|' , 1)}>>
	returns "Mathews"
	<<{split(cityStateZIPCountry , ';' , 1)}>> with
	cityStateZIPCountry = "Charleston;West Virginia;29402;United States"
	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.
	`squote(string)`
	where:
	`string = the string in which to replace double-quotes`
	For Example:
	<<{squote('This is Amy"s.')}>> returns "This is Amy's."

Function	Synopsis

`startsWith`	Checks to see if a string starts with a given string.
	`startsWith (mainString, subString)`
	where:
	`mainString = the string to check`
	`subString = the string to look for at the start of mainString`
	For Example:
	<<{startsWith('The first string', 'The')}>> 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".
	<<cs_{startsWith(VIN, '1VW')}>> <<VIN>> <<es_>>

`substring`	Display a subsection of a string given starting and finishing indexes.
	`substring(string, start, finish)`
	where:
	`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.`
	For Example:
	<<{substring('0123456' , 2 , 5)}>>
	returns "234"
	<<{substring(LatLong , 0 , 6)}>> with
	LatLong = "31.9088983S115.8049265E"
	will return "31.908"

Function	Synopsis

`left`	Display a subsection of a string given the finishing index.
	`left(string, finish)`
	where:
	`string = the string`
	`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.`
	For Example:
	<<{left('The quick brown fox jumps over the lazy dog.', 5)}>>
	returns "The q"

`right`	Display a subsection of a string given starting index.
	`right(string, start)`
	where:
	`string = the string`
	`start = the position in the string that will now become the first character. Indexing starts at 0.`
	For Example:
	<<{right('The quick brown fox jumps over the lazy dog.', 4)}>>
	returns "dog."

`titleCase`	Changes the string so that the first character of each word is a capital letter.
	`titleCase (string)`
	where:
	`string = the string to convert`
	For Example:
	<<{ titleCase ('bob mathews')}>>
	returns "Bob Mathews"
	<<{titleCase (firstName+ ' ' + lastName)}>>
	with data of firstName = "bob" and lastName = "MATHEWS"
	also returns "Bob Mathews"

Function	Synopsis

`toAlpha`	A function to convert a given number to a letter in the sequence:
	a, b, c, … z, aa, bb, cc, … zz, aaa, bbb, etc.
	`toAlpha(key)`
	where:
	`key = the data value`
	For Example:
	<<{toAlpha(index)}>>
	when index = “3”, returns “c”.
	when index = “28", returns “bb”.

`toAlpha2`	A function to convert a given number to a letter in the sequence:
	a, b, c, … z, aa, ab, ac … az, ba, bb, etc.
	`toAlpha2(key)`
	where:
	`key = the data value`
	For Example:
	<<{toAlpha2(index)}>>
	when index = “3”, returns “c”.
	when index = “28", returns “ab”.

`toLowerCase`	Returns the string using all lower case characters.
	`toLowerCase (string)`
	where:
	`string = the string to convert`
	For Example:
	<<{toLowerCase(‘Bob Mathews’)}>>
	returns “bob mathews”

`toUpperCase`	Returns the string using all upper case characters.
	`toUpperCase (string)`
	where:
	`string = the string to convert`
	For Example:
	<<{toUpperCase(‘Bob Mathews’)}>>
	returns “BOB MATHEWS”

Function	Synopsis

`toRoman`	A function to convert a given number to a roman numeral
	`toRoman(key)`
	where:
	`key = the data value`
	For Example:
	<<{toRoman(index)}>>
	when index = “2”, returns “ii”.
	when index = “29", returns “xxix”.

`toSentence`	Adjusts the given string converting it to sentence case.
	`toSentence(string)`
	where:
	`string = the string to convert`
	For Example:
	<<{toSentence(‘a little. ditty’)}>> returns “A little. Ditty”

`trim`	Removes leading and trailing spaces from a string.
	`trim (string)`
	where:
	`string = the string`
	For Example:
	<<{trim(productID)}>>
	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. `abs (number)` For Example: <<{abs(-153.57)}>> returns “153.57”. <<{abs(temp)}>> If the data has: temp = “-273.15” returns “273.15”
`ceil`	Returns the next largest whole number. `ceil (number)` For Example: <<{ceil(153.57)}>> returns “154.0”
`floor`	Returns the next smallest whole number. `floor (number)` For Example: <<{floor(153.57)}>> returns “153.0”
`isNumber`	Returns true if the given parameter is numeric: `isNumber (value1)` For Example: <<{isNumber(53)}>> returns “true” <<{isNumber(‘53.5111’)}>> returns “true” <<{isNumber(‘-1.0e20’)}>> returns “true”
`max`	Returns the larger of the two numbers. `max (number1, number2)` For Example: <<{max(53.5,23.1)}>> returns “53.5”

Function	Synopsis
`min`	Returns the smaller of two numbers. `min (number1, number2)` For Example: <<{min(53.5,23.1)}>> returns “23.1”

`numFormat`	Format a number based on the format provided and the locale.
	`numFormat (value, format [, locale [,applyLocaleToInput [, formatIsLocalized]]])`
	where:
	`value = the number to format`
	`format = the format to apply. E.g.: ‘#,###.00’`
	`locale = optional locale to use. Country or language name or code. E.g.: ‘GERMAN’, ‘USA’.`
	`applyLocaleToInput = 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.`
	`formatIsLocalized = 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 thousands-separator value in the FRENCH locale, by default the format uses a non-breaking space (Unicode \u00A0) to represent the separator. When this is false, ‘,’ is used to separate numbers into thousands.`
	See section *4.1 Formatting Numbers* for full formatting syntax.

Function	Synopsis

`numToDollars`	Format a number based on the format provided and the locale.
	`numToDollars(value)`
	where:
	`value = the number to format`
	If the number is an integer, just the dollar amount is written. If the number has a fractional component, the number with 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:
	`<<{numToDollars(100)}>>` returns "one hundred dollars"
	`<<{numToDollars(123.457)}>>` returns "one hundred twenty three dollars and forty six cents"

`numToText`	Write out the given number using English words
	`numToText (value [, form])`
	where:
	`value = the number to format`
	`form = 'andLast' - (default) write the last "and" in the text`
	`form = 'andAlways' - write the "and" in all places in the text`
	`form = 'andNone' - brief - don't write the "and" at all.`
	Supports numbers up to 9.2 quintillion.
	For Example:
	<<{numToText(123)}>> returns "one hundred and twenty three"
	<<{numToText(123, 'andNone')}>> returns "one hundred twenty three"

Function	Synopsis

`ordinal`	Write out the given number as an ordinal using either numbers or English words.
	`Ordinal (value [, form])`
	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:
	<<{ordinal(123)}>> returns "123rd"
	<<{ordinal(123, 'suffix')}>> returns "rd"
	<<{ordinal(123, 'long')}>> returns "one hundred and twenty third"
	<<{ordinal(123, 'longNoAnds')}>> returns "one hundred twenty third"

`pow`	Returns the power of two numbers. `pow (number1, number2)` For Example: <<{pow(7,2)}>> returns 7 to the power of 2, so "49.0"
`random`	Returns a random number between 0 and 1. `random()` For Example: <<{round(random()*100)}>> returns a random number between 0 and 100.

Function	Synopsis

`round`	Rounds the number to the specified number of places.
	`round (number [, places])`
	where:
	`number = the number to round.`
	`places = the number of decimal places required. If not specified then round to zero decimal places.`
	For Example:
	<<{round(153.75)}>>
	returns “154”
	<<{round(153.73455,2)}>>
	returns “153.73”

`sqrt`	Returns the square root of a number `sqrt (number)` For Example: <<{sqrt(81.0)}>> returns “9.0”

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.
	`dateAdd (date, amt, units [,outputFormat [,inputFormat]])`
	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.
	`dateDiff (date1, date2, units [, inputFormat])`
	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.
	`dateFormat (value [, outputFormat [, inputFormat [, outputLocale [, inputLocale [, outputFormatLocalized [, inputFormatLocalized]]]]]])`
	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 the data 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.
	`locale([spec])`
	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.
	`localeInfo([spec])`
	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"

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

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.

2.6. Useful Diagnostics

Docmosis offers a method to view the data it has received using the special Docmosis field `<<dump:...>>`. This can be used to output the entire set of data being processed, or just a subset of the data. This is useful when diagnosing template problems and for seeing exactly what data values are available in the template.

Docmosis will attempt to display the data in the format it was provided in. For example, json format when the data supplied is json.

To dump the entire contents sent to Docmosis use `<<dump:$top>>` or `<<dump:$root>>`. Note that this could result is a lot of data being output.

To dump a subsection of the data, reference the element name, eg `<<dump:address>>` would just output the address object of the data.

The dump field is aware of the current "context" and will render data relative to that context. For example:

```
<<rs_InvoiceItems>>
<<dump:$this>>
<<es_>>
```

Will dump the current InvoiceItem each time the loop is iterated.

2.6.1. Diagnostic tools

The following tools can be used to help diagnose template issues:

Element	Description
`<<dump:name>>` `<<dump:$builtInVariable>>`	Dump the data as it is understood into the document for diagnostics purposes. Lookup “name” in the data and dump the data object into the document at this location. Built in variables can also be used, eg: <dump:$top>> <dump:$parent>> <dump:$this>>

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

Docmosis "fields" can be added 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 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, 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 the template already includes merge fields or input fields.
- Text can be displayed 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 accidently lost (replaced with the underlying lookup value) if the fields in the document are "updated".
- depending upon the version of the chosen word processor users 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 users need to either type the field codes manually, or copy a merge field from another document then edit it to reference the data item required.

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. The text that is displayed can be adjusted without changing the name of the merge field. Simply by editing the text that appears between the angle brackets. Please note that if the field codes are later auto-updated 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 lookup 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 it was selected and typed over 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 users 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. Simple Field Name Syntax

Simple fields that look up an item of data have the following naming rules:

1. Must start with a letter
2. Can include letters, numbers, underscores and hyphens
3. Can be surrounded by square brackets [and]

For example, the following are valid fields:

```
<<personName>>
<<personName24>>
<<person-name>>
<<[person-name]>>
```

Note that when using a hyphenated name, like "person-name", the hyphen becomes ambiguous in expressions (it could be intended to be the subtraction operator). In this case, the surrounding [and] brackets are required to identify the fields. For example:

```
<<{[last-value]-[first-value]>>
```

In simple fields that are not expressions, the [and] are optional because it assumed to be a hyphenated field name.

3.4.2. 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.

3.4.3. 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

Variables can created in 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". To have a variable to retain its value outside of a loop, use it once before the 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 *2.4.3.10 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.

3.5.1. Check if a variable has been set

There are scenarios where the need to reference a variable without knowing if it has already been initialized, for example in a sub template called by various master templates. If the variable `<<$company>>` was used as the first line in a sub template but it had not been initialized previously, then Docmosis would return an error.

In this type of situation Docmosis offers a less-strict use of a variable, called a forgiving lookup: `<<$?...>>`. In the example the field could instead be `<<$?company>>` which would return a blank if not set (rather than an error).

Combining a forgiving variable lookup with a conditional section and the `isBlank` function would be a common pattern for working with a variable that might not have been set:

```
<<cs_{isBlank($?company)}>>
N/A
<<else>>
<<$company>>
<<es_>>
```

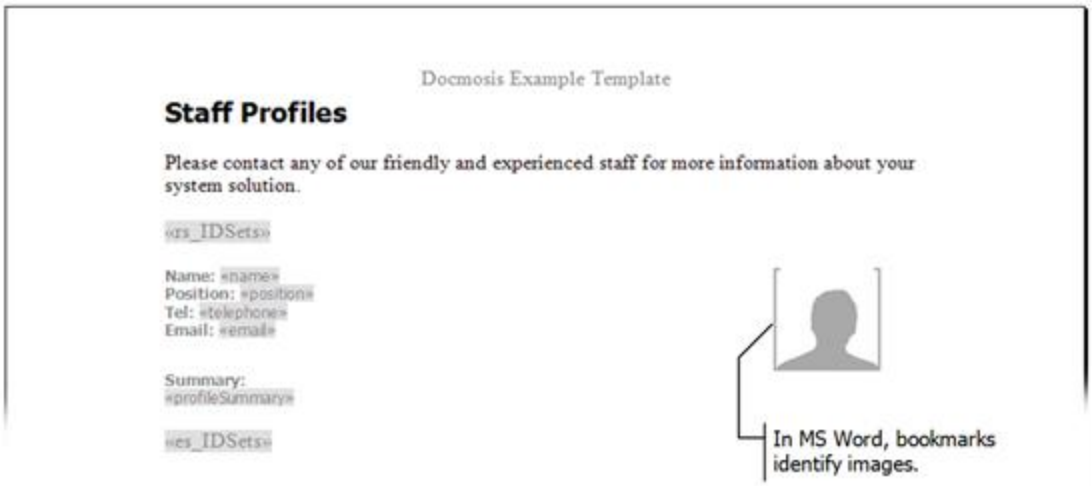
3.6. Using Images

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.

DOCX

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 positioning 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 the 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 a placeholder image is inserted, the image is embedded in the template. 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, 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 the Docmosis application and enable users to use the bookmarking and naming features for other items that aren't part of a document generation.

The prefixes available 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.
`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 a template since both Microsoft Word and LibreOffice Writer force the name to be unique. If a requirement is to reference the same image data in the template multiple times, different names will have to be specified 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 the template, since both Word and Writer force the name to be unique. To reference the same image in the template multiple times, different names will have to be used 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 the 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
- IMB
- QR Codes are also supported (see section *3.8 Creating QR Codes below*).

3.7.2. Typical Barcode Example

As a typical example, put a placeholder image into the template and "mark" it (with a name or bookmark as per section *3.6* above):

`Image marked as 'imgfit_dispatchLabel'.`

The 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 barcode. 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 the 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
docmosis Placeholder	"dispatchLabel":"1234567: code39"	A code39 barcode image with value 123456. The data provided the barcode type and value.
docmosis Placeholder `<<barcode:dispatchLabel:code39>>`	"dispatchLabel":"1234567"	A code39 barcode image with value 123456. The template specified the barcode type. The data provided the barcode value.

In the Template the placeholder image is marked "img_dispatchLabel" and the template may also include a "<<barcode:" field	Data (JSON example)	Result
docmosis Placeholder	"dispatchLabel":"1234567: code39:dpi=1200"	A code39 barcode image with value 123456 and resolution 1200 dpi. The data has specified all configuration.
docmosis Placeholder `<<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.
docmosis Placeholder `<<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 the 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. Use the "imgstretch" to force the barcode to match the placeholder precisely but this will likely reduce the accuracy of the barcode and may make it harder to scan. 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 a barcode is generated 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`	False2

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`	
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, 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
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 QR Codes

Docmosis can generate QR codes and insert them into output documents. QR code insertion is very similar to image insertion and the barcodes described above using a placeholder image and optionally a <<qrcode:>> field to set defaults in the template for the related QR code.

3.8.1. Typical QR Code Example

As a typical example, put a placeholder image into the template and "mark" it (with a name or bookmark as per section *3.6* above):

`Image marked as 'qrcode_dispatchLabel'.`

At the time of rendering, the **data** supplies the QR code value:

```
"dispatchLabel":"https://url.to.com/my/dispatchLabel/1233"
```

Docmosis then renders the barcode into the size and placing of the QR code:

3.8.2. Using a "qrcode" Field to Specify Default Settings

In the above example, the placeholder image determines the size, position and name for the resulting QR code. All other settings are defaulted (such as resolution and error correction level).

Default settings can be adjusted for a given QR code image by using an optional <<qrcode:>> field. The field can set specific settings described in the following sections.

Continuing the above example, the template could specify QR Code dpi setting like this:

```
<<qrcode:dispatchLabel:dpi=200>>
```

This indicates that dispatchLabel should use a resolution of 200 dpi instead of the default.

The `<<qrcode:…>>` field can appear anywhere in the template, either before or after the placeholder image. It links to the correct placeholder by name.

3.8.3. QR Code Settings in Detail

Anything about a QR code can be specified in the template with a qrcode field, including the actual value:

```
<<qrcode:dispatchLabel:https://my.domain.com/:dpi=200:ec=M>>
```

Settings can also be overridden by the data supplied during a render call. For example, the DPI resolution can be changed dynamically by the data (e.g. in JSON format):

```
"dispatchLabel":" https://my.domain.com/:dpi=800"
```

The data provided at render time will override any settings specified in a qrcode field in the template and then system defaults.

The following settings are supported for QR codes.

Common Settings				
Name	**Shorthand**	**Description**	**Example**	**Default Value**

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, use the minimum that suits the use of the barcode (e.g. taking into account the printer quality).	`dpi=1200`	600 (1200 max)
errorCorrection	`ec`	The Error Correction level L, M, Q or H.	`ec=M`	M
encoding	`en`	The encoding to use	`en=UTF-8`	UTF-8

3.9. Creating Active Hyperlinks

Docmosis allows hyperlink to be inserted dynamically into documents.

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 the data.

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

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

That would create this: `http://www.example.com` in the finished document.

The text to be displayed for the hyperlink may be different from the URL of the link. This is achieved by using the pipe (|) symbol in the data to separate the display name from the hyperlink value.

For example, 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 in the finished document.

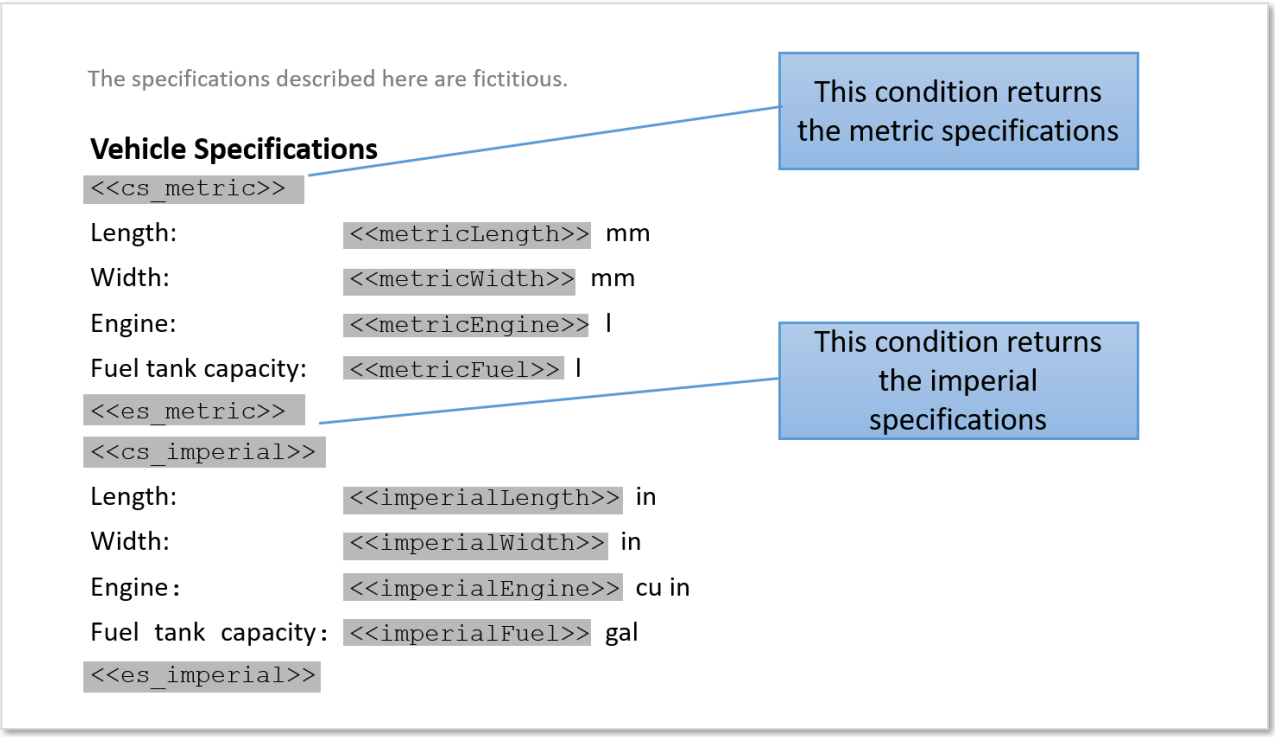
Note that in all cases:

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

3.10. 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 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 required.

Conditional sections also allow for alternative conditions to be tested using an *if-then-else* construct. Most simply, a conditional section can test something and else can be used to embody all non-matches to the conditional section test. For example:

```
<<cs_{name='Jill'}>>
This is Jill's content
<<else>>
This is anyone but Jill's content
<<es_>>
```

The above sequence will display "This is Jill's content" if name is "Jill", otherwise it will display "This is anyone but Jill's content".

The `else` field can also test data elements and use expressions the same way as the `cs_` field. For example:

```
<<cs_{name='Jill'}>>
```

```
This is Jill's content
<<else_{name='Jen'}>>
This Jen's content
<<else>>
This is anyone else's content
<<es_>>
```

The above tests name against "Jill" then "Jen" displaying the appropriate information if either of those conditions are true. If name is any other value, the else will be used to display the appropriate content.

3.11. 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.

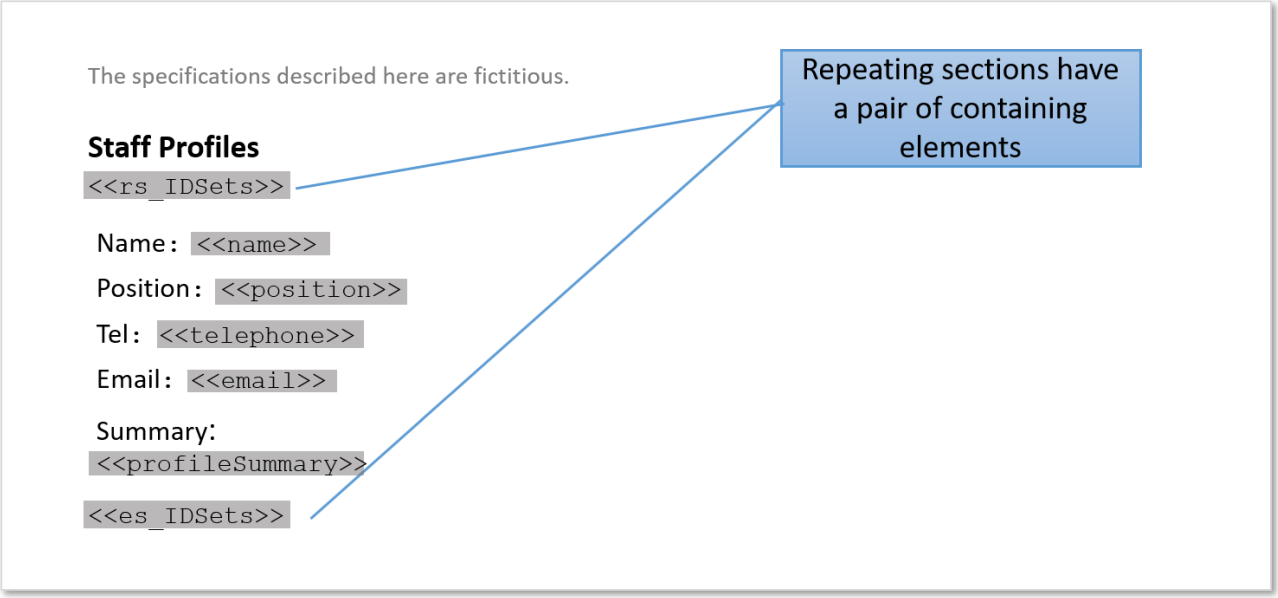
Tables and lists are special forms of repeating sections. They are discussed after this section. This section deals specifically with block-level repeating sections.

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 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

3.11.1. Stepping Across in Repeating Sections

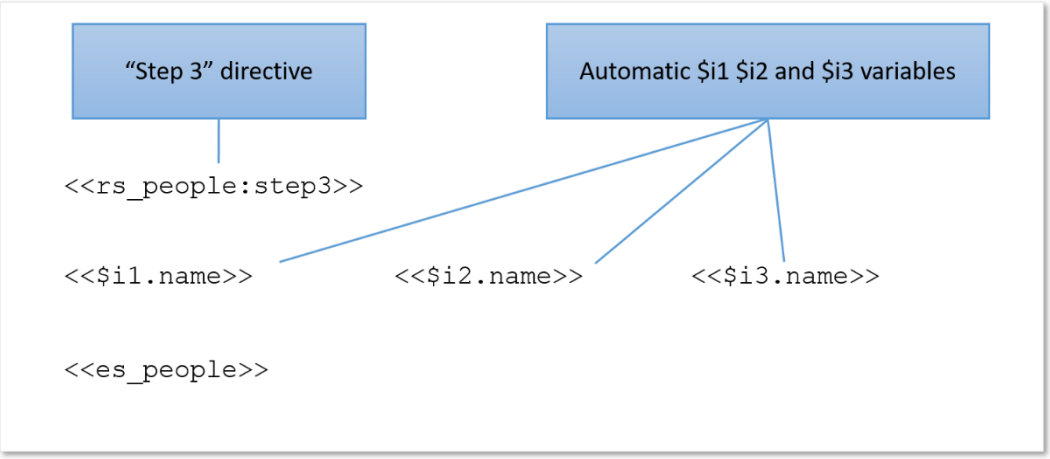
Docmosis supports the concept of repeating in "steps". Each "step" is a smaller subset of the complete array of data. For example, consider a simple array of "people" objects as follow:

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

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

James	Jenny	Julie
Katie	Kim	Kerry

The "stepping" is done 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 a 4-across layout is required instead, this is easily changed in the template by using "`:step4`" and adding the 4th column in the template to layout as required:

```
<<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...).

3.11.2. Stepping Down in Repeating Sections

In the same way that Docmosis can present the 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 wish to show the data down the columns, rather than across, the "`:step4down`" directive can be used 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.

3.11.3. Sorting in Repeating Sections

Docmosis supports sorting of data in a repeating section. For example, consider a simple array of "people" objects with the following names:

`"Chloe", "Sam", "Ben", "Catherine", "Andy", "Andrew"`

Docmosis is able to sort by these names before displaying the data. In a template use the `:sort` (or `:sortStr`) directive as follows:

```
<<rs_people:sort(name)>>

<<name>>

<<es_people:sort(name)>>
```

In the generated document, notice that the list of names is now in sorted order:

Andrew

Andy

Ben

Catherine

Chloe

Sam

To instead display the above names in sorted descending order use: "`:sort(DESC, name)`". The default order is ascending, which can explicitly be stated with `ASC`, eg "`:sort(ASC, name)`".

This example is sorted by a field in the data, but sorting by an expression is also possible. For example to sort by a person's full name, use the following sort "`:sort(lastName + ', ' + firstName)`" to combine the first and last names before performing the sort.

Docmosis can also sort in case sensitive or case insensitive mode. Take, for example, the following simple array of "business" objects with the following names:

```
"ECM Electronics", "eCentral", "aXcelerate", "Accendant", "ACME"
```

In this example the case used is important for the order of these objects. By default, Docmosis will sort in `CASE_INSENSITIVE` mode, it is possible to sort in `CASE_SENSITIVE` mode instead, which can be done with the following syntax: "`:sort(CASE_SENSITIVE, name)`".

Another consideration to make is how to handle missing data. Given a list of "business" objects with the following websites:

```
"ecm.org", null, "axcelerate.com.au", "accendant.biz", null
```

Where will the missing values end up? By default, Docmosis will put missing data ("nulls") up front, if sorting ascending (`ASC`); and it will put missing data last, if sorting descending (`DESC`). But this can be controlled using the keywords `NULLS_FIRST` or `NULLS_LAST`, eg "`:sort(NULLS_LAST, website)`".

Note that the above keywords can also be combined in a single sort, for example: "`:sort(DESC, CASE_SENSITIVE, NULLS_FIRST, name)`".

The `:sort` keyword is an alphanumeric sort, interpreting the text and numeric parts of data. This sort understands that "abc11" comes after "abc2". There are three other sort types: string, numeric and date.

A simple string sort is identified using `:sortStr`. String sorting is simple but if unlike alphanumeric sort above, it would sort "abc11" before "abc2". eg "`:sortStr(name)`".

The directive `:sortNum` can be used to explicitly sort numerical data. This is especially useful when the data is a mixture of number types. For example, take the following list of numbers:

```
"2.0", "-1", "0", "2e10", "-1e-10", "1E-10", "0.1"
```

Let's compare the result of a `:sortNum` to the regular alphanumeric `:sort`, take the following template:

<<rs_list:sortNum(number)>>	<<rs_list:sort(number)>>
<<number>>	<<number>>
<<es_list:sortNum(number)>>	<<es_list:sort(number)>>

The data above would result in the following output:

-1	-1
-1e-10	-1e-10
0	0
1E-10	0.1
0.1	1E-10
2.0	2.0
2e10	2e10

Notably there is a mistake in the regular sort result on the right. But the numerical sort, on the left, is in the correct order.

Docmosis also supports sorting by dates, with the keyword `:sortDate`. Consider a list of "people" objects with the following dates of birth:

```
"20/Feb/1986", "09/Dec/1972", "01/Oct/1999", "19/Mar/1987",
"19/Feb/1986", "29/Nov/1970"
```

It is possible to sort on these dates in a template. Note that Docmosis needs to be instructed about what `inputFormat` the dates are in, for more info on date formatting please see *4.2 Formatting Dates*. Take the following template:

```
<<rs_people:sortDate(dob, 'dd/MMM/yyyy')>>

<<dob>>

<<es_people:sortDate(dob, 'dd/MMM/yyyy')>>
```

Using the data above the result is the following sorted output:

```
29/Nov/1970

09/Dec/1972

19/Feb/1986

20/Feb/1986

19/Mar/1987

01/Oct/1999
```

The date sorter also allows specification of `inputLocale` and `inputFormatLocalized`.

`inputLocale` is used to specify the locale used for interpreting the input date (eg 'italian'), and `inputFormatLocalized` is used to indicate whether or not to interpret the `inputFormat` string using the specified `inputLocale`. For example, to apply a German locale use the following syntax `":sortDate(date, 'yyyy-mm-dd', 'german', true)"`.

Finally, multiple sorts can be specified to sort by one value, then by another. For example:

```
<<rs_people:sort(category):sort(lastName)>>
```

3.11.4. Filtering in Repeating Sections

Docmosis supports filtering data in a repeating section using an expression. This is similar to using a conditional section to only display the data that meets a conditional expression, see *3.10 Using Conditional Sections*. Consider a simple list of grocery "product", each containing a field of "name" and "type":

Name:	"Apple",	"Lettuce",	"Orange",	"Carrot",	"Pear"
Type:	"Fruit",	"Vegetable",	"Fruit",	"Vegetable",	"Fruit"

To just display the Fruit items, add filtering to products whose type equals "Fruit":

```
<<rs_product:filter(type='Fruit')>>

<<name>>

<<es_product:filter(type='Fruit')>>
```

And this would result in just the "Fruit" items being output:

Apple

Orange

Pear

3.11.5. Grouping in Repeating Sections

"Grouping" data is supported in Docmosis. To do so, a grouping expression needs to be specified to tell Docmosis what element of the data to group by. Take the previous example of grocery "product":

Name:	"Apple",	"Lettuce",	"Orange",	"Carrot",	"Pear"
Type:	"Fruit",	"Vegetable",	"Fruit",	"Vegetable",	"Fruit"

Similar products can be grouped together using the field “type”, eg `":group(type) "`. Once the data has been grouped together, use the special keyword `$groupItems` to repeat over the items within the group. Docmosis also sets the variable `$groupKey` to identify the key used to form the current group of items (the current “type” in the above example).

In a template it would look like this:

```
<<rs_product:group(type)>>

Key: <<$groupKey>>

<<rs_$groupItems>>

Name: <<name>>

<<es_$groupItems>>

<<es_product:group(type)>>
```

In this example, the Docmosis tag, `"rs_product:group(type)"`, is used to repeat over the “product” data, grouped by the “type” of product. Two groups are formed, one for the type “Fruit” and one for type “Vegetable” and hence the repeat will loop twice.

Inside the repeat, `<<$groupKey>>` is used to output the key used for the current group(one of the values of “type”). Next, the template repeats over the items in each group using the `<<rs_$groupItems">>` variable and displays the “name” of each product within the group. The output will look like this:

***Key:* Fruit**

Name: Apple

Name: Orange

Name: Pear

***Key:* Vegetable**

Name: Lettuce

Name: Carrot

"Grouping" can also use expressions to form the groups. This can be a very powerful tool. For example, with a simple list of "people" objects, group together people with the same birth month with the following expression:

```
":group(dateFormat(DOB, 'MM', 'dd/MMM/yyyy'))"
```

3.11.6. Combining Repeating Section Directives

Many of the repeating section directives above can be combined together, including:

```
:sort():filter()
:sort():group()
:filter():group()
:sort():filter():group()
:sort():stepN[down]
:filter():stepN[down]
:group():stepN[down]
:sort():filter():stepN[down]
:sort():group():stepN[down]
```

```
:filter():group():stepN[down]
```

```
:sort():filter():group():stepN[down]
```

These can also be combined with range specifiers, see Section *2.4.3.9 Range Specifiers*.

3.12. Using Tables

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

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

3.12.1. 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	
<<cr_hasFriends>>	
Jimmy has some friends	
<<er_hasFriends>>	

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 feilds `<<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_>>`.

3.12.2. 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
<<rr_friends>>	
Jimmy has a friend called <<friend>>	<<friendJob>>
<<er_friends>>	

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

3.12.3. 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 *3.11.1*) and Stepping Down in Repeating Sections (section *3.11.2*) also apply to data displayed in a table using Repeating Rows.

3.12.4. Sorting in Repeating Rows

Docmosis supports sorting of data in repeating rows. For example, taking a simple list of “people” objects, "`rr_people:sort(name)`" will sort the list of people by the name field.

The examples above of *Sorting in Repeating Sections* (section *3.11.3*) also apply to data displayed in a table using Repeating Rows.

3.12.5. Filtering in Repeating Rows

Docmosis supports filtering data in repeating rows using an expression. This is similar to using a conditional section to only display the data that meets a conditional expression, see *3.10 Using Conditional Sections*. For example, taking a simple list of “animal” objects, "`rr_animal:filter(type='bird')`" will filter a list of animals to just the “bird” type.

The examples above of *Filtering in Repeating Sections* (section *3.11.4*) also apply to data displayed in a table using Repeating Rows.

3.12.6. Grouping in Repeating Rows

Docmosis supports "grouping" data together in repeating rows using a field or expression. For example, to group together a simple list of "animal" objects by "type" might look like this:

<<rr_animal:group(type)>>
***Key:* <<$groupKey>>**
<<rr_$groupItems>>
Name: <<name>>
<<er_$groupItems>>
<<er_animal:group(type)>>

The examples above of *Grouping in Repeating Sections* (section *3.11.5*) also apply to data displayed in a table using Repeating Rows.

3.12.7. Combining Repeating Row Directives

Many of the repeating row directives above can be combined together, including:

`:sort():filter()`

`:sort():group()`

`:filter():group()`

`:sort():filter():group()`

`:sort():stepN[down]`

`:filter():stepN[down]`

`:group():stepN[down]`

`:sort():filter():stepN[down]`

`:sort():group():stepN[down]`

```
:filter():group():stepN[down]
```

```
:sort():filter():group():stepN[down]
```

These can also be combined with range specifiers, see Section *2.4.3.3 Range Specifiers*.

3.12.8. 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 *3.12.11 Advanced Table Structures*.

3.12.9. 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.

3.12.10. 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	<<cc_showJobs>>	
Friend	Job	Contact
<<rr_friends>>		
Jimmy has a friend called <<friend>>	<<friendJob>>	<<friendContact>>
<<er_friends>>		

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. Examining the `Trial 2` and `Trial 3` columns shows 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

3.12.11. 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>>

3.13. 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
```

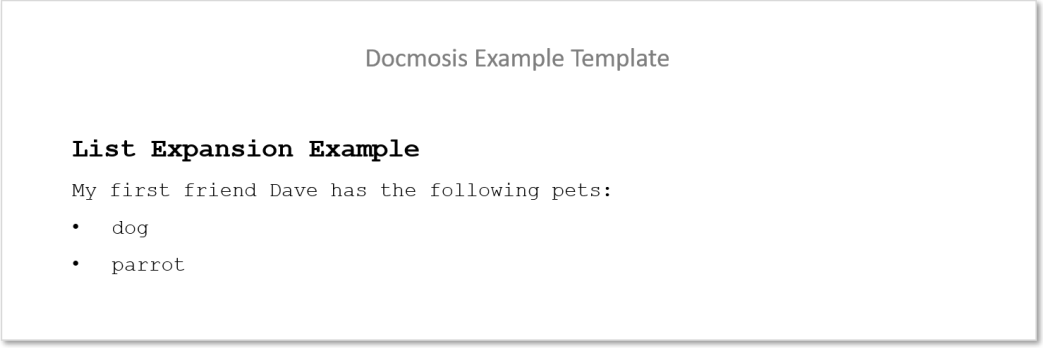
As another example of how the field is split, consider the following template:

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.

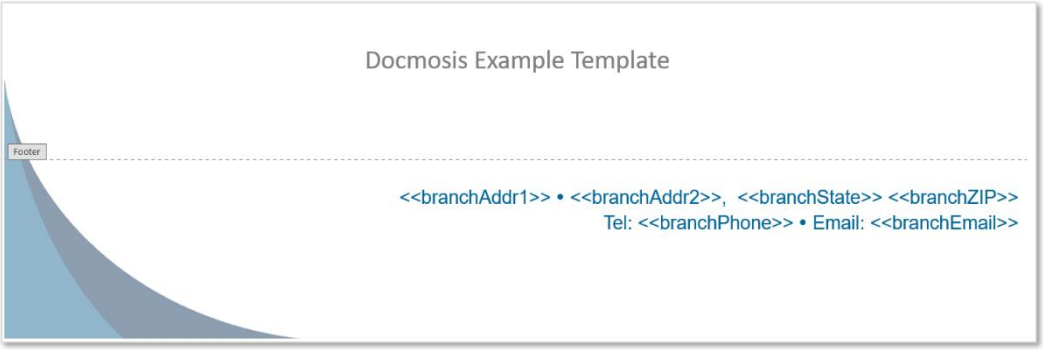
To create a list:

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.

3.14. Using Headers and Footers

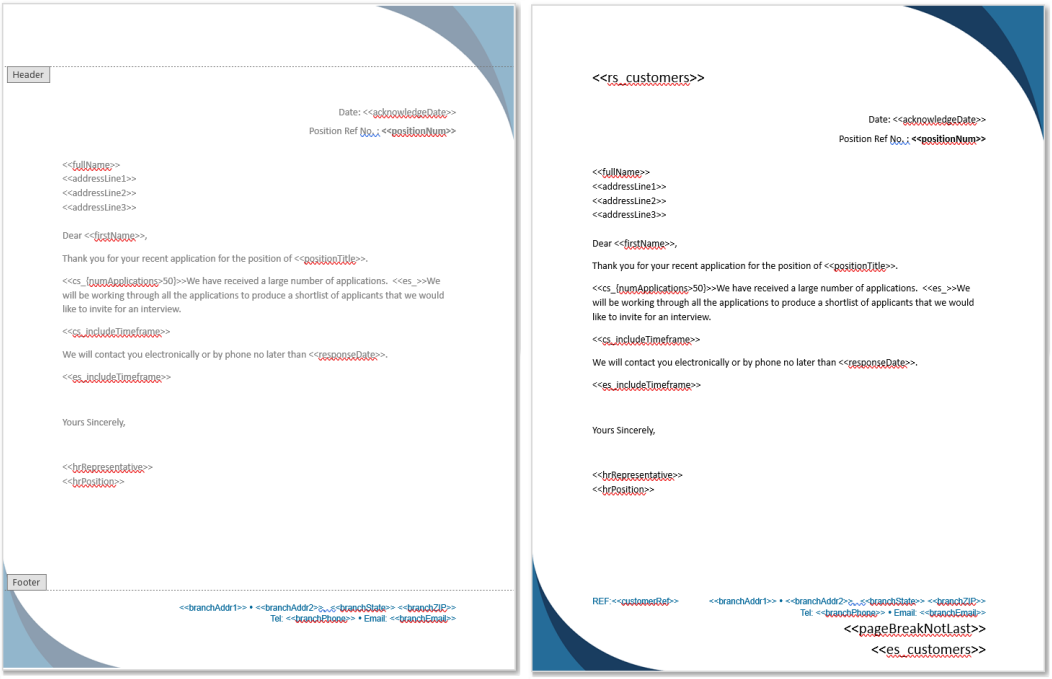
It is possible to use the headers and footers features of Microsoft Word or LibreOffice when creating templates. Any static text, images anchored in the header/footer or word processor features (such as page numbering) can be used in headers and footers and will appear in the finished document.

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. Specific parts of the data used in a header and footer will not change from page to page in a way related to the current page content.

One option to create a document that appears to align headers with page content is to remove the header and footer and 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 the content to be positioned where a traditional header/footer would be placed. Use a repeating section around the whole page, and use fields that do include page specific information at the top or bottom of the 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.

Another template feature that may help is to use Coordinator templates (see below).

3.15. 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 users may need to comment out a section of a template which itself contains other comments. For example, 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. ##>>
```

3.16. Merging Templates Together

Docmosis provides three methods for combining templates:

1. Merging/Embedding – pulling the contents of another template into the current template. A "master" template typically determines the main content and overall style (such as headers and footers) and it references other templates from which to pull content.
2. Coordinating – rendering multiple templates independently in a single call and possibly combining into a single PDF output. A "coordinating" template refers to other templates to be rendered in order but the coordination template is not part of the result.
3. Multiple template references in API calls – multiple templates can be specified in the Docmosis API when executing document generation. This is discussed in the product-specific developer / API guide, not in this guide.

Merging and Coordinating templates primarily involves the use of <<ref:xxx>>, <<refLookup:xxx>> and <<refLookupOp:xxx>> fields to refer to other templates.

Coordinating and Merging templates can refer to other templates, including ones that merge in others. A coordinator template however cannot be used by other templates and must always be the “main” template rendered.

3.16.1. Combining Templates Using Merging/Embedding

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.

This template **merging** is primarily for pulling in pieces of content which exist in other templates into a single output. In this case a “master” template refers to other templates from which to include content. The master typically determines the overall style, including headers and footers for the final document.

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 the 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.docx` that references two sub-templates `process1.docx` and `process2.docx`, Docmosis will insert `process1.docx` and `process2.docx` into `MainProcess.docx` as it processes `MainProcess.docx`.

For example, a master template that looks like this:

```
Hello <<name>>,
Today is a lovely day.
<<ref:signoff.docx>>
```

Will create a document that like this:

```
Hello Jamie,
Today is a lovely day.
Best Wishes,
Admin.
```

where the main content comes from the master template, "Jamie" comes from data and the best wishes comes from the signoff.docx template.

Docmosis supports two ways of referencing templates; directly and indirectly. Each of these is explained further a little later in this section.

3.16.2. Combining Templates Using Coordination

Template **coordinating** works as if rendering each template separately. A "coordinator" template exists to specify the other templates to be rendered. The coordinator template itself is not part of the resulting document, its sole purpose is to coordinate the rendering of other templates. The results of a coordinated render are one or more files correlating to each of the templates rendered in the order determined by the coordinator template.

Coordination results in one or more files as follows:

- PDF output – a zip file is created with the individual PDF files for each template. Docmosis Cloud and Docmosis Tornado can also create a combined single PDF automatically.
- Other output formats – a zip file is created with the individual files rendered for each template.

To create a coordinator template, create a document which contains the marker field:

```
<<coordinator:>>
```

somewhere in the template. Subsequent template references will be processed in order. A simple coordinator template may look like this:

```
<<coordinator:>>
<<ref:coverPage.docx>>
<<ref:mainOffer.docx>>
<<ref:annexA.docx>>
<<ref:annexB.docx>>
```

when the template is rendered, Docmosis will render the templates coverPage.docx, then mainOffer.docx, annexA.docx and finally annexB.docx into documents and return the combined result. As mentioned previously, the combined result will typically be a zip file, except in specific cases where Docmosis will automatically combine the results into a single PDF document.

Coordinator templates can use almost any Docmosis template features such as conditions, repeating, and variable setting. The template variables created in the coordinator template

are available to the templates being referenced. Further template variables created in the referenced template are visible to the coordinator and to subsequent referenced templates. Coordinator templates can even use templates that merge (but not coordinate) other templates.

So, the simple coordinator example above might optionally include annexB depending on some other condition:

```
<<coordinator:>>
<<ref:coverPage.docx>>
<<ref:mainOffer.docx>>
<<ref:annexA.docx>>
<<cs_includeAllAnnexes>>
<<ref:annexB.docx>>
<<es_>>
```

Docmosis supports two ways of referencing templates; directly and indirectly. Each of these is explained further a little later in this section.

3.16.3. Coordination-Specific Features

There are some features specific to direct the coordinator process. They are:

<<coordinator:padToEvenPage>> - if necessary, insert a blank page to make the output an even number of pages at this point.

<<coordinator:padToOddPage>> - if necessary, insert a blank page to make the output an odd number of pages at this point.

<<coordinator:newFile[:new name]>> - switch to a new file (with an optional new name) for the subsequent templates. This implies a zip file will be created instead of a single PDF. This allows coordination of a number of templates into a specific number of output PDFs.

Each of the above features take effect in Docmosis Cloud and Docmosis Tornado when automatically joining templates into a single output PDF. In Docmosis Java, the information is made available in the API response that these features were used, but Docmosis Java does not combine PDFs automatically.

3.16.4. Advantages and Disadvantages of Merging Features

Template merging and template coordination are two features for different purposes. Subtle advantages and disadvantages are present:

Advantages of Merging Templates:

- Suited for pulling common **pieces** of content into a main template
- Can be nested arbitrarily as required
- The master generally determines the main style, page numbering, headers, footers.
- Merging results in a combined single file and works for any output format.

Disadvantages of Merging Templates:

- The style of the sub-templates (referenced templates) can be overridden by the master template (this is also an advantage)
- Sub templates can't have their own headers and footers
- Have a performance impact (noticeably slower)

Advantages of Coordinator Templates:

- Make it easy to bring a set of templates into a single render call
- Can create a combined PDF (Tornado and Cloud only)
- Each document is rendered as if it is was rendered itself (ie headers/footers, page numbering, styles) but using the same data from a single call to Docmosis
- Efficient to render (low overhead and less latency)
- Can share template-variables and data between coordinated templates.

Disadvantages of Coordinator Templates:

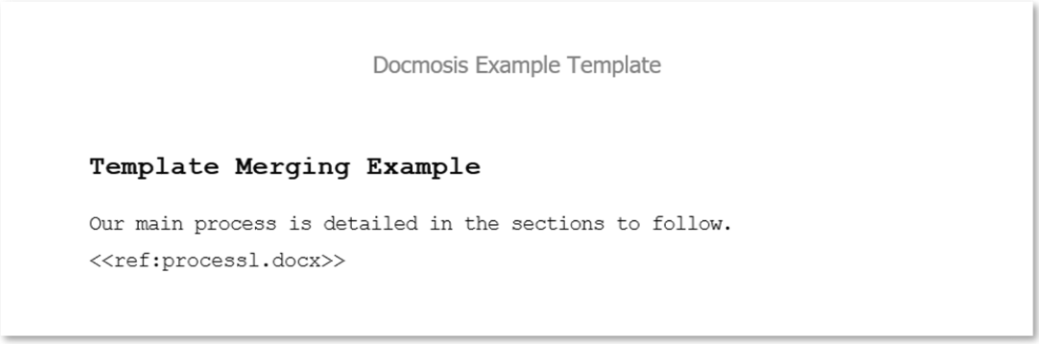
- Cannot create a single, combined output document (other than PDF using Tornado or Cloud).

3.16.5. Direct Referencing (ref:)

Direct referencing is simple - the name of the sub-template is literally placed in the master template.

The way to create a direct reference is to use a field prefixed with "`ref:`".

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



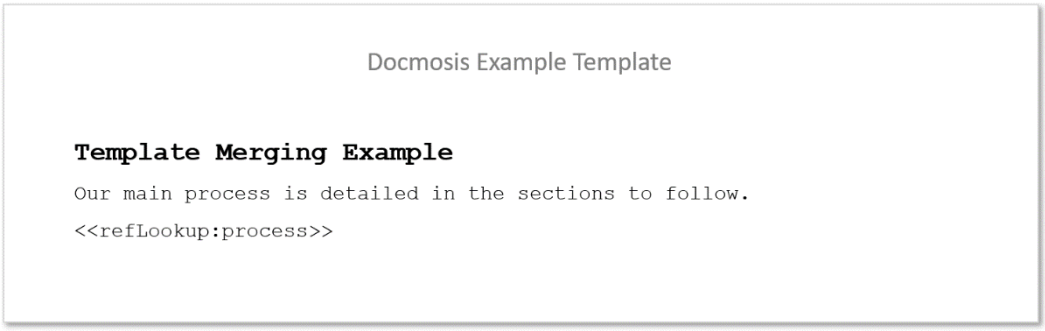
The direct reference indicates to Docmosis template merging to insert the content of the sub-template into the master at the point of the reference. It informs Docmosis template coordination that this is the next template to be rendered into a document.

3.16.6. Indirect Referencing (refLookup:)

An indirect reference serves the same purpose as a direct reference, except it obtains the name of the template to use via a lookup of the data at the time of the render.

To create an indirect reference, the field prefix "`refLookup:`" or "`refLookupOp:`" is used.

For example, creating 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 might look:



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

```
"process":"process1.docx"
```

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

If the given lookup ("process" in the above example) resolves to a blank, refLookup will treat this as an error whereas refLookupOp will consider this normal with nothing to do.

3.16.7. Templates in Different Locations

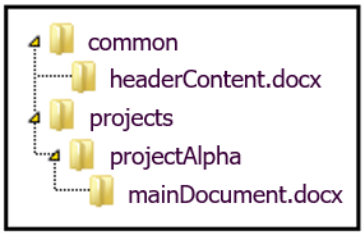
By default, Docmosis assumes the referenced sub-templates are stored in the same location as the referring template. In the scenario above, `process1.docx` must exist in the same location as the master-template.

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`".

To include a common header in templates separate the header layout/content into a separate sub-template called `headerContent.docx`. This new sub-template applies to other projects and is not exclusive to projectAlpha.

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



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

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

Note the leading "/" above. Or use the field:

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

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

3.16.8. 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.docx>> <<es_>>
```

3.16.9. 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, Docmosis can instructed to join the numbered lists together when the sub-template is inserted by using the `<<list:continue>>` directive.

This only applies to **merging** (not **coordinating**) since coordinated renders are largely independent of each other.

For example, if the master template looked like this:

Master Legal Document

1) First Legal Clause
2) Second Legal Clause

<<ref:commonClauses.docx>>

And the 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.).

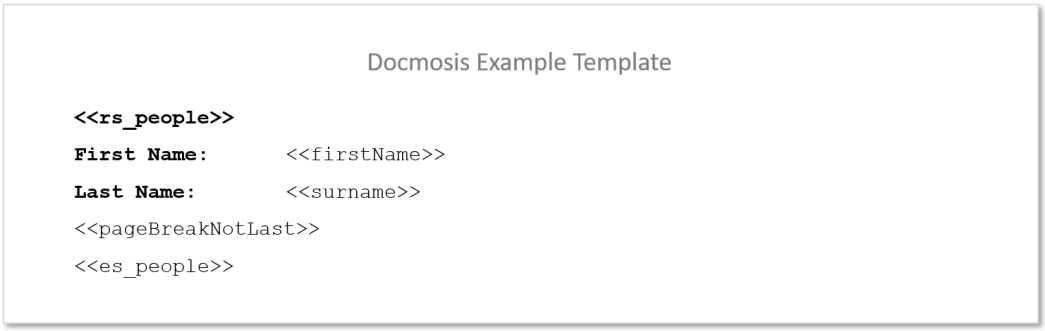
3.17. Page Breaks and Other Breaks

Docmosis templates may contain several types of break including page breaks, column breaks and section breaks. If the break is in the template it will appear in the rendered documents unless it is conditioned out with a conditional section. If the break is inside a repeating section it will be repeated each time the 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 at the last item in the current repeating section. This is only valid within a repeating section.
`<<columnBreakNotLast>>`	Insert a column break at this location in the document at the last item in 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:



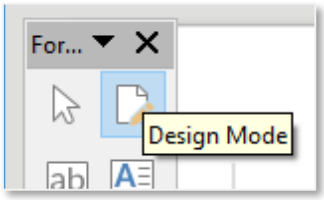
3.18. Creating Pre-filled PDF Forms

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.

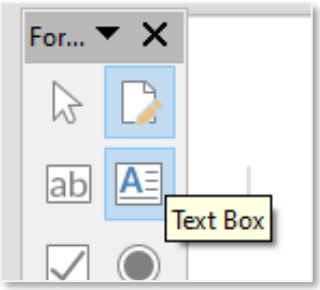
To Create a PDF form, start with a LibreOffice Writer document for the template. Make sure the Form Controls are visible: **View > Toolbars > Form Controls**. On the **Form Controls** toolbar, click the "Design Mode" button to be able to add form fields to the template:



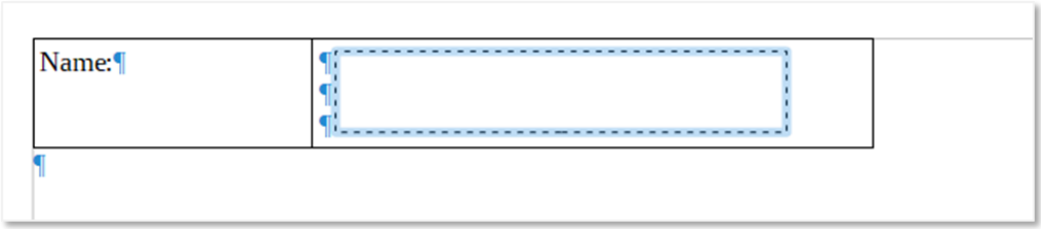
Once in design mode, all the controls are enabled and can be added to the template.

3.18.1. Adding a Text Field

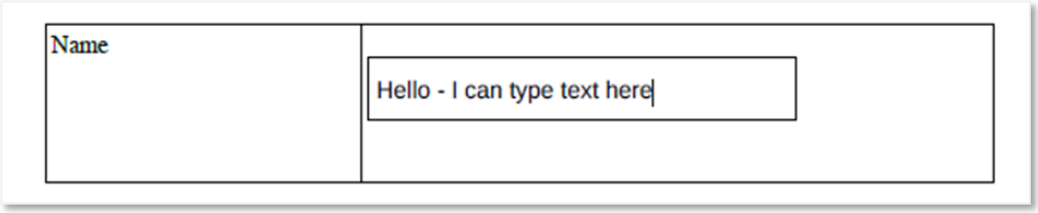
1. To add text field, click the text box field:



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

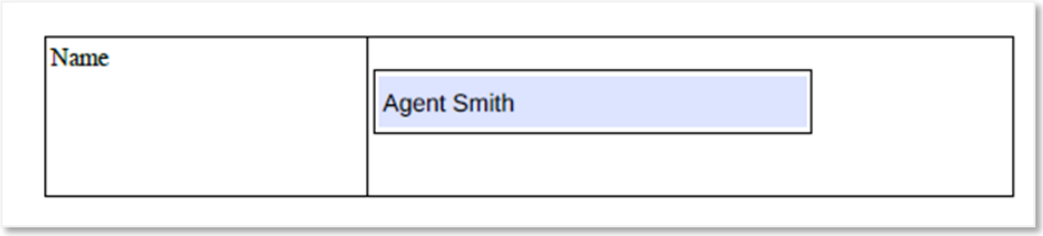


3. When the above template is rendered to PDF, there is a name field in which the user can enter text:



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

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

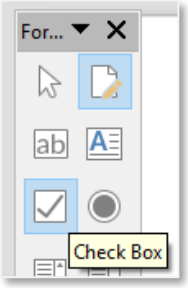


The PDF is now a pre-filled form, but can still be edited and adjusted as required.

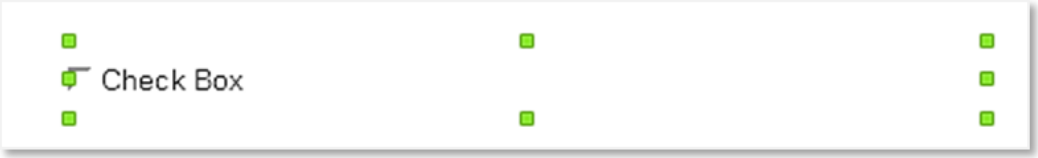
3.18.2. Adding a Checkbox

When working with checkboxes, the same process applies.

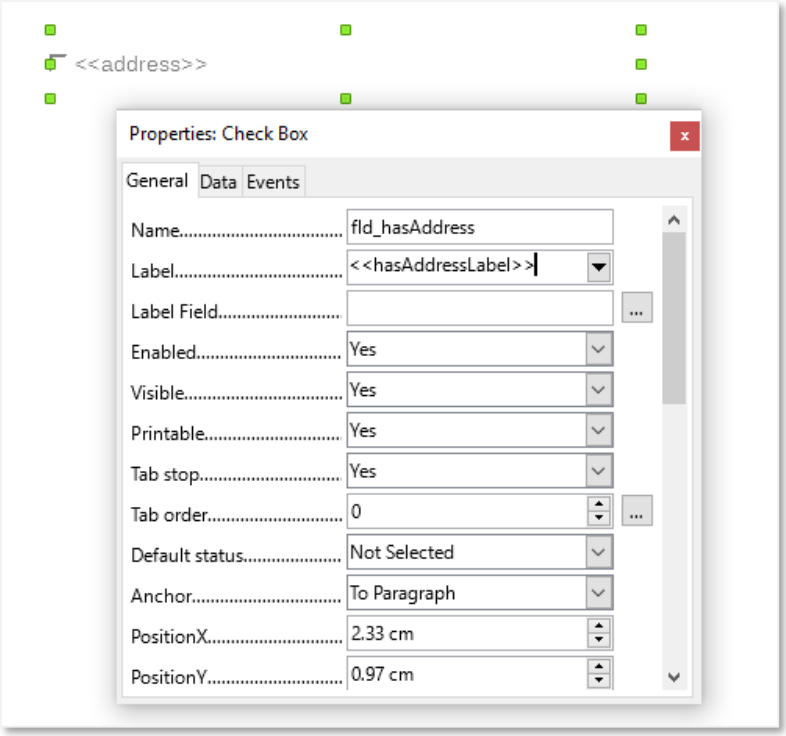
1. Select the **Check Box** button:



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



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



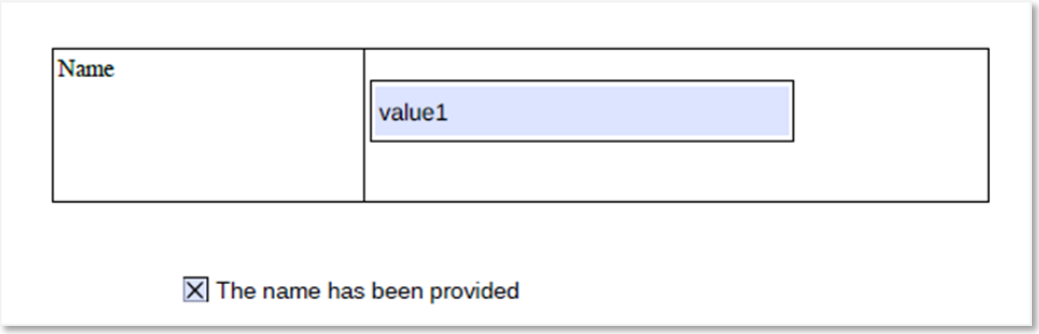
The "`fld_hasAddress`" name will cause Docmosis to look up the data for the value of "`hasAddress`" and use the value to tick or untick the box when the document is generated. The "`fld_`" prefix is what tells Docmosis to look up the value dynamically.

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:



4. FORMATTING DATA

Formatted or unformatted data may be passed to Docmosis. Docmosis has the ability to format data for presentation in finished documents, such as formatting numbers, currency or dates and displaying symbols for true or false boolean values. This section discusses the advantages and disadvantages of both options.

For example, if the finished document is 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 the data must be formatted before passing it to Docmosis.

Alternatively, raw data can be used, in which case the formatting can be applied 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 the template authors. Formatting can be applied using the functions that are part of the Docmosis expression processing (Functions *2.4.4.5 Date Functions*).

4.1. Formatting Numbers

Given numeric data (even from a text source like XML), Docmosis can be instructed to format it by providing a formatting "string". The string describes how the number should 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
`<<{numFormat(myVal,'0.0')}>>`	12.3
`<<{numFormat(myVal,'$0.00')}>>`	$12.34
`<<{numFormat(myVal,'0.0E0')}>>`	1.2E1
`<<{numFormat(myVal,'###.##')}>>`	12.34
`<<{numFormat(myVal,'000.0000')}>>`	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
`<<{numFormat("1234,5",'#.###,#','DE')}>>`	Data is in German format and output is using German format.	1.234,5
`<<{numFormat("1234.5",'#,###.#','US')}>>`	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 the `'¤'` character is used in the formatting string, Docmosis will replace this with the currency sign for the locale being used.

For example:

Example	Result
`<<{numFormat('1234.5', '¤#,###.00', 'GBR')}>>`	£1,234.50
`<<{numFormat('1234,5', ' #.###,00¤', 'DE')}>>`	1.234,50€
`<<{numFormat('1234.5', '¤#,###.00', 'USA')}>>`	$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
`<<{numFormat('12.345','#.###,0','DE','true')}>>`	12.345,0
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;	
`<<{numFormat('12.345','#.###,0','DE','false')}>>`	12,3

Example	Result
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;	

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 thousands separator is a non-breaking space rather that the common space. In a template, this can be inserted using ctrl-shift-space. In data such as JSON, it can be encoded as \u00A0.

4.2. Formatting Dates

Given time and date data Docmosis can format it by providing a formatting "string". The string describes how the time and/or date should 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
`<<{dateFormat(myDate,'dd/MM/yyyy')}>>`	27/05/2020
`<<{dateFormat(myDate,'MMM dd, yyyy')}>>`	May 27, 2020
`<<{dateFormat(myDate,'EEE, dd MMM yyyy')}>>`	Wed, 27 May 2020
`<<{dateFormat(myDate,'yyyy')}>>`	2020
`<<{dateFormat(myDate,'EEEE, dd 'of' MMM')}>>`	Wednesday 27 of May

`<<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','US')}>>`
`<<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','ITALY')}>>`
`<<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','GERMANY')}>>`
	Wednesday
	mercoledì
	Mittwoch

`<<{dateFormat(myDate,' EEEE tt/MMM/uuuu','dd-MMM-yyyy','GERMANY', 'US', true, true)}>>`	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
`<<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','US','de')}>>`	Wednesday
`<<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','ITALY','de')}>>`	mercoledì
`<<{dateFormat(myDate,'EEEE','dd-MMM-yyyy','GERMANY','de')}>>`	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
`<<{dateFormat(myDate)}>>`	No output string so `'dd MM yyyy'` is used.	25 May 2020
`<<{dateFormat(myDate,'MMM_dd,_yyyy')}>>`	Underscores are replaced with spaces.	May 25, 2020
`<<{dateFormat(myDate,'EEEE, dd 'of' MMM')}>>`	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, to write out the date as typical for the German locale (again using data "myDate":"25-May-2020"):

Example	Description	Result
`<<{dateFormat(myDate, null, null, 'German')}>>`	The null values for the output and input format mean use defaults for the format.	27 Mai 2020
`<<{dateFormat(myDate, 'EEEE', null, 'de')}>>`	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
`<<{dateFormat(myDate, null, null, 'de', 'de')}>>`	The null values for the output and input format mean use defaults for the format.	27 Mai 2020
`<<{dateFormat(myDate, null, null, 'US', 'de')}>>`	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.

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

Country			Language		
Code	ISO Code	Name	Code	ISO Code	Name
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
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

Country			Language		
Code	ISO Code	Name	Code	ISO Code	Name
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
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

Country			Language		
Code	ISO Code	Name	Code	ISO Code	Name
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 – 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

Docmosis Pty Ltd

Address
Suite 8 / 5 Hasler Road,
Osborne Park,
WA 6017 Australia

Website
https://www.docmosis.com

Resources
https://resources.docmosis.com

docmosis