Sheng

Thank you for using Sheng, our Ruby gem for data merging Word documents! We’ve designed Sheng to be intuitive (both for developers and document template authors), but there are some guidelines you need to follow to make the best use of its powerful features.

This reference is written for template authors, so you won’t need any Ruby programming knowledge to use it. All you’ll need is a copy of Microsoft Word. You’ll be creating Word MergeFields, but we’ll give you instructions on how to do so.

Note that any instructions on how specifically to use certain commands in Word are based on the version of Word that comes with Microsoft Office 2011 for Mac, but they should be the same or similar in recent versions on either platform.

Table of Contents

The Data Set 3

An Example JSON Data Set 3

A JSON Reference Data Set 5

Accessing the Data Set: Finding Variable Names 5

Creating MergeFields in Microsoft Word 6

Editing an Existing MergeField 6

Basic MergeField Substitution 8

Inline Basic MergeFields 8

Filters on String Values 8

Filters on Numeric Values 9

Basic Arithmetic Operations 9

Checkboxes 9

Sequences 11

Arrays of Primitives 12

Inline Sequences 12

Comma-Separated Series 12

Sequences in Tables 12

Embedded Sequences 13

Conditional Blocks 14

# The Data Set

Before we get started on how Sheng substitutes values into Word document templates, let’s talk about where Sheng **gets** these values: the data set. The data set is basically a list of all of the possible dynamic variables you can use in your templates, and will be provided by your developers. The data set your developers will provide you will have meaningful placeholder values for each dynamic variable, and soon you’ll learn more about those placeholder values. But for now, let’s look at something a bit simpler.

## An Example JSON Data Set

This is an example of an **actual** data set, with real values instead of placeholder values, that Sheng would use to populate a document template:

{  
 “last\_user”: {  
 “first\_name”: “Pat”,  
 “last\_name”: “Shoeshine”,  
 “phone\_numbers”: [  
 {  
 “number”: “555-123-1234”,  
 “is\_mobile”: true  
 },  
 {  
 “number”: “555-456-4567”,  
 “is\_mobile”: false  
 }  
 ],  
 “favorite\_colors”: [“red”, “white”, “green”]  
 },  
 “website\_language”: “English”,  
 “daily\_visits”: 10,  
 “percent\_of\_clicks”: 83.14,  
 ...  
}

The format you see above is called JSON (JavaScript Object Notation). It’s commonly used to send data between different services on the internet, so it’s designed to be easy for a computer to generate and parse, but it’s also designed to be easy for humans to read and write.

A block of JSON text contains, at the root level, a single object. Objects, which are always surrounded by curly braces, are a set of key-value pairs. The root level object above has three key-value pairs – the keys are “last\_user”, “website\_language”, and “visits”. Each of those keys is followed by a colon, then the actual value. A value can be any of the following:

* A string (in double quotes)
* A number (integer or decimal)
* A “Boolean” value (true or false, not in quotes)
* An object (surrounded by curly braces, { and })
* An array of any of the above values (surrounded by square brackets, [ and ])
* The word null, which represents missing or unknown data

In the example above, the value for “last\_user” is another object, and that object in turn has key-value pairs for the user’s attributes. The “first\_name” and “last\_name” values are strings, so they’re in double quotes.

The value for “favorite\_colors” is a bit more complicated – because a user can have more than one favorite color, this value is an array (which basically just means an ordered list). It’s a group of values separated by commas, just like you’d expect to write a list in plain language, and the whole list is surrounded by square brackets ([ and ]).

The next key-value pair, “phone\_numbers”, is even more complicated, but follows the same principles. A user can have more than one phone number, so this value is also an array, but instead of just being an array of strings or numbers, it’s an array of objects. This is because each phone number has a few traits of its own – the number itself, and whether or not it’s a mobile phone. So each of the two phone number “values”, separated by a comma, is an object with two key-value pairs. Notice that the second key-value pair in each phone number object has a “Boolean” value (true or false), and it’s not in quotes. You can use this data type for things that are black or white, yes or no questions – in this case, whether the phone number is mobile or not. Other examples might be “agreed\_to\_terms”, “over\_18”, or “opted\_in\_to\_newsletter”. In Sheng, these are mostly useful for automatically checking a checkbox according to the value in the data set, but they’ll also display as the words “true” or “false” if not used with a checkbox.

Finally, there are two other values at the “root” level: “website\_language” and “visits”. The former is simply another string. The latter, “visits”, is an integer, so it doesn’t need to be in double quotes.

And that’s it! Now that you know what a sample data set looks like, let’s look at what an actual data set reference might look like from your development team.

## A JSON Reference Data Set

The example above is great, but your developers will probably send you something a bit more “abstract” – a reference data set that reveals, by definition rather than example, what values you’ll be working with. Here’s a sample reference data set that correlates to the example we’ve already seen:

{  
 “last\_user”: {  
 “first\_name”: “string”,  
 “last\_name”: “string”,  
 “phone\_numbers”: [  
 {  
 “number”: “string”,  
 “is\_mobile”: “Boolean”  
 }  
 ],  
 “favorite\_colors”: [“string”]  
 },  
 “website\_language”: “string”,  
 “daily\_visits”: “integer”,  
 “percent\_of\_clicks”: “decimal”,  
 ...  
}

That looks familiar, right? All the keys are the same, but the values are different. Those values are placeholders that describe what **kind** of values you can expect to receive for that given key. They’re mostly intuitive, but here’s a quick guide:

|  |  |  |
| --- | --- | --- |
| Placeholder | Description | Example |
| string | An arbitrary string | “Arthur von Blasterface” |
| integer | A whole number (no decimal or fraction) | 18493 |
| decimal | A number with a decimal component | 6.143 |
| Boolean | A “Boolean” value (true or false, not in quotes) | true |

Note that arrays each only contain one item in the data set reference, but that single item indicates what each value should look like.

Compare the data set reference above with the first example JSON given, and see how the example adheres to the data types shown in the reference file.

Phew, that’s enough staring at JSON. Well, almost.. we still need to talk about how to access these values when creating your templates.

## Accessing the Data Set: Finding Variable Names

When you want to access a certain value, you need to tell Sheng how to find it in the data set. For key-value pairs at the root level, this is easy – you just use the key itself. So, for “visits” in the example above, you use just that – the string “visits”.

However, this gets complicated once you get beyond the root level. How, for example, do you tell Sheng you want the “first\_name” of the “last\_user” object?

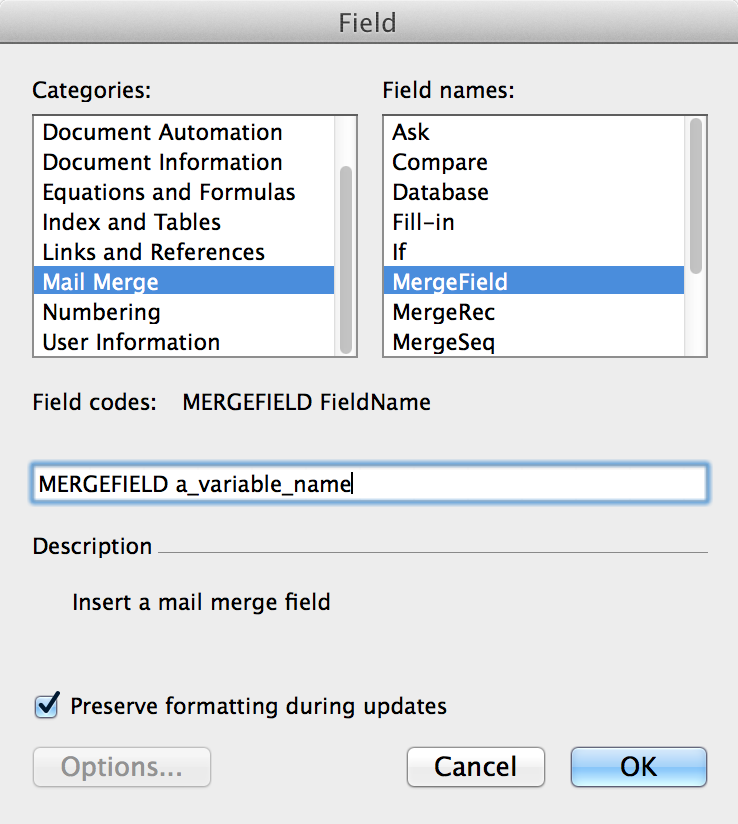
To do so, you have to combine the hierarchy you want to traverse into a single string, and Sheng uses something we call “dot notation” for this. In dot notation, a single period (.) is an instruction to move down the hierarchy one step. Thus, to get the “first\_name” from the “last\_user” object, you would create the variable name “last\_user.first\_name”, and Sheng would find the string “Pat”.

What about accessing something inside an array, like the number of the last\_user’s first phone number? With Sheng templating, you wouldn’t access that value directly – you’d do it using sequences, which you’ll learn about soon.

So that’s data sets. Now you’re ready to start creating document templates!

# Creating MergeFields in Microsoft Word

Let’s get acquainted with mergefields in Microsoft Word, since you’ll be using these to indicate places in your document template where you want data from the data set to be inserted (replacing the mergefield).

Before inserting a mergefield, look through the data set reference provided by your developers, and find the variable name that corresponds to the value you want inserted.

To insert a mergefield into your document, place the cursor where you want the new mergefield to appear, then select *Insert > Field…* from the menu. You’ll get a dialog box that looks something like the one on the right.

Make sure “Mail Merge” is selected from the “Categories:” column on the left, and then select “MergeField” from the “Field names:” column on the right. The “Field codes” text field will change to read “MERGEFIELD”. Put the variable name you’ve selected at the end (ensure there is a space between). Click “OK”, and you’ll see the variable appear. That’s it!

## Editing an Existing MergeField

Unfortunately, editing the variable for a mergefield is a bit more complex. Microsoft Word stores two pieces of text when you create a mergefield – one is the label shown in the document template, and the other is the actual mergefield variable, which is hidden by default. When you first create the mergefield, these are the same, but if you try to edit the label shown on the screen (between the « and » characters), you won’t actually succeed in changing the variable used for substitution.

You have two options here – 1) the brute force, but simpler, option; or 2) the more surgical one.

**Option 1:**

Just delete the existing mergefield, then recreate it using the steps above. Make sure, when you do this, to select everything including the « and » characters, but nothing more, and then just hit Delete.

**Option 2:**

Place your cursor within the mergefield label, then hit Shift + F9. The mergefield will change from something like this:

«my\_merge\_field»

To something like this:

**{** MERGEFIELD my\_merge\_field \\* MERGEFORMAT **}**

Now you can edit the label. **Do not** change anything outside of the label itself – leave everything else as is. So, for example, if you’re changing *my\_merge\_field* to *another\_merge\_field*, after you’re done editing it should look like this:

**{** MERGEFIELD another\_merge\_field \\* MERGEFORMAT **}**

Once you’re finished, make sure your cursor is within the label, and hit Shift + F9 again. Hang on, what happened?! It looks the same as it did before! That’s because you changed the important part (the variable), but the label shown in the template is still what it used to be. To force this to update, hit F9 (without the Shift key this time) while your cursor is in the label, and it will automatically update itself to look like this:

«another\_merge\_field»

And that’s it! You’re probably wishing you used **Option 1** now, aren’t you?

# Basic MergeField Substitution

Given a single primitive value (a string, integer, Boolean, or floating point number) in the data set, just create a mergefield with that value’s variable name, and it will be substituted with the value.

### Examples:

«a\_basic\_string»

«a\_basic\_integer»

«a\_basic\_boolean»

«a\_basic\_float»

## Inline Basic MergeFields

Basic mergefields can be placed in the middle of a paragraph of text, and will substitute without breaking the flow of the text.

### Example:

The haberdasher, a gentle fellow by the name of «a\_basic\_string», meticulously pinned «a\_basic\_integer» hats.

## Filters on String Values

There are some built-in filters available to modify the value of a string variable (e.g. to display a value as fully uppercase regardless of its case in the data set). These filters, if used on non-string values, will have no effect. The filters are:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Description | Example Input | Example Output |
| upcase | Uppercases entire string | Doctor special | DOCTOR SPECIAL |
| downcase | Downcases entire string | What Is Going ON? | what is going on? |
| capitalize | Capitalizes only first letter, downcases rest | how many KIDS do you HAVE? | How many kids do you have? |
| titleize | Capitalizes first letter of each word | there is no excuse for BADNESS | There Is No Excuse For Badness |
| reverse | Reverses characters in string | This is not a palindrome. | .emordnilap a ton si sihT |

Filters can also be chained, and will be applied in which they appear (from left to right).

### Examples:

«a\_basic\_string|upcase»

«a\_basic\_string|downcase»

«a\_basic\_string|capitalize»

«a\_basic\_string|titleize»

«a\_basic\_string|reverse»

«a\_basic\_string|capitalize|reverse»

«a\_basic\_integer|downcase»

## Filters on Numeric Values

There are also built-in filters available for modifying the output of numeric values (e.g. to round a number or display it as currency). These filters, if used on non-numeric values, will have no effect. Strings that can be read as numeric values (like "145.2") will work with these filters. Unlike the string filters, these numeric filters can accept arguments, to clarify what the filter should do. The filters are:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Example Input | Example Command | Example Output |
| round(n) | Rounds number to *n* decimal places | 2360.7853 | round(2) | 2360.79 |
| floor | Truncates decimal portion of number | 2360.7853 | floor | 2360 |
| currency(x) | Formats number as currency (2 fixed decimals, thousands separator), prepending optional argument *x* | 149020.5 | currency($) | $149,020.50 |

Filters can also be chained, and will be applied in which they appear (from left to right).

### Examples:

«a\_basic\_float|floor»

«a\_basic\_float|currency($)»

## Basic Arithmetic Operations

You can perform basic arithmetic operations within a mergefield, using either hardcoded numeric values or variable names. The operators allowed are + (addition), - (subtraction), \* (multiplication), and / (division). The usual order of precedence applies with these operators, and you can use parenthesis to control that precedence.

Note, however, that adding formatted numbers (like "1,400.50" and "24,000.20") will result in an unformatted number ("25400.7"), so if you want currency formatting or rounding, you'll want to use the numeric formatting methods shown above. You can combine arithmetic with filters to accomplish this, as seen in the last example below.

### Examples:

«25 \* 6»

«8.3 + 5.232»

«(2.1 + 5) \* 18»

«(hair.length \* 2) + head.height»

«principal + interest | currency($)»

# Checkboxes

To create a checkbox:

1. Go to the Developer tab in the Ribbon (you may have to turn it on under *Preferences > Ribbon*), and click on the "Check Box" button.
2. When the checkbox appears on the page, double click it, or right-click it and choose "Properties".
3. Locate the relevant variable name from the data set, which should be a Boolean (i.e. have a true or false value, without double quotes).
4. Find the "Bookmark" field under Field Settings, and enter the variable name in that field.
5. Click "OK" to close the dialog box.

Checkboxes with a missing "Bookmark" value, or with one that doesn't match a variable from the data set, will be left alone.

### Examples:

This checkbox will be checked

This checkbox will be unchecked

This checkbox will be left unchecked

This checkbox will be ignored completely

# Sequences

Sheng’s most powerful feature is the sequence. Sequences are used when you have a an array of objects in the data set that you want to iterate over, and repeat a certain block of text (or other element) once for each member of that array. Each member of the array should have key-value pairs itself, to be used for substitution *within* the repeating section.

A sequence is created by wrapping the repeating elements with special “bookend” mergefields. These bookend mergefields should be named with “*start:*” and “*end:*” followed by the name of the variable pointing to the desired array. So if your variable is named “birds”, your bookend mergefields will look like *«start:birds»* and *«end:birds»*.

The repeated block itself can have mergefields in it; these mergefields will be populated using the each member’s values. Arrays of basic primitive values (like strings, integers, Booleans, etc) are also allowed; see “Arrays of Primitives” below.

Within the repeated block, the variable names you concoct should pretend that the member object itself is at the root level. So if you have a data set that looks like this...

{  
 “ghosts”: [  
 {  
 “name”: “string”,  
 “age”: “integer”,  
 “favorites”: {  
 “snack”: “string”,  
 “haunts”: [“string”]  
 }  
 }  
 ]  
}

... and you’re creating a sequence of ghosts, you’d use *«start:ghosts»* and *«end:ghosts»* as your bookend mergefields, but in between, you’d just use «name» instead of «ghosts.name». So your mergefields might look like this:

«start:ghosts»  
 Name: «name»  
 Age: «age»  
 Favorite Snack: «favorites.snack»  
 Favorite haunts: «start:favorites.haunts»«item», «end:favorites.haunts»  
«end:ghosts»

Wait, we snuck in something new there, didn’t we? That array of haunts – where did the “item” variable come from when trying to iterate on an array of strings? Well, that’s coming up in the next section.

### Examples:

«start:classroom.students»

Please welcome «first\_name» «last\_name» to our institution. «pronoun|capitalize» will be a wonderful addition!

«end:classroom.students»

## Arrays of Primitives

An array does not have to contain objects with key-value pairs; it can simply be a array of primitives (like strings, integers, etc). When using a sequence like this, some variable name must be used within the repeating block to represent the member; by default this will be “item”, but this can be overridden using a special “as(*variable*)” filter.

### Examples:

«start:adjectives»

«item»

«end:adjectives»

«start:adjectives|as(an\_adjective)»

«an\_adjective»

«end:adjectives»

## Inline Sequences

Sequences can be placed in the middle of a paragraph of text, and will iterate and substitute without breaking the flow of the text.

### Example:

The chemist was thrilled when she produced a «start:adjectives»«item» «end:adjectives»mammoth from her test tube.

## Comma-Separated Series

A special filter is also available on sequences that will automatically turn the repeated block into a member of a comma-separated series, complete with conjunction. The conjunction defaults to the English “and”, but can be customized with an argument. Note that currently, an Oxford comma will always be used.

### Example:

The chemist and the haberdasher partnered to create hats for the newly spawned mammoth: «start:adjectives|series\_with\_commas|as(hat\_type)»a «hat\_type» one«end:adjectives».

El químico y la mercería se asociaron para crear sombreros para el mamut recién engendrado: «start:español.adjetivos|series\_with\_commas(y)»uno «item»«end:español.adjetivos».

## Sequences in Tables

Using a sequence in a table is easy, but you must follow some specific rules for how to set up your table. The “bookend” markers must be on rows by themselves, bracketing the row(s) to be repeated, in the first column of their row (the other column(s) should remain empty). Your member substitution mergefields can appear anywhere in the row(s) between, even separated onto multiple rows.

### Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| Last Name | First Name | Favorite Juice | Smart? |
| «start:classroom.students» |  |  |  |
| «last\_name|upcase» | **«first\_name»** | «fruit» juice |  |
| «end:classroom.students» |  |  |  |
| *All students above are special, even if not smart.* | | | |

## Embedded Sequences

Any block in Sheng can be nested in another block, theoretically as deeply as you want. Therefore, if you have members of a array who, themselves, contain arrays, you can embed a sequence within another sequence and the result should be intuitive.

### Examples:

The 2016 Student List:

* «start:classroom.students»
* «first\_name» «last\_name»  
  «pronoun|capitalize» grades:
  + «start:classes»
  + «name»: «grade»
  + «end:classes»
* «end:classroom.students»

# Conditional Blocks

If you have a section in your document that should or should not display based on the existence of a certain variable, you can use conditional blocks. These come in two flavors – *if:* and *unless:* – and are used as expected, given their names.

Anything within an *if:* block will only be included if the given variable exists (and is not an empty array).

The *unless:* block is the inverse – if the given variable exists, the block will **not** be included.

The bookend mergefields for conditional blocks are similar to those for sequences, but instead of *start:* and *end:*, you use *if:* and *end\_if:* (or *unless:* and *end\_unless:* for an unless block).

As seen in the example below, sequences can be nested within conditional blocks, and vice versa, creating a powerful way to tailor your document based on the data.

### Examples:

«if:classroom.students»

We have a bunch of students! Their names are «start:classroom.students|series\_with\_commas»«first\_name» «if:nickname»“«nickname»” «end\_if:nickname»«last\_name»«end:classroom.students».

«end\_if:classroom.students»

«unless:classroom.students»

Oh dear. We have no students.

«end\_unless:classroom.students»

«if:classroom.teachers»

We have a bunch of teachers! Their names are «start:classroom.teachers|series\_with\_commas»«first\_name» «if:nickname»“«nickname»” «end\_if:nickname»«last\_name»«end:classroom.teachers».

«end\_if:classroom.teachers»

«unless:classroom.teachers»

Oh dear. We have no teachers.

«end\_unless:classroom.teachers»