

---

# **docxtemplater Documentation**

*Release*

**Edgar Hipp**

**Mar 13, 2017**



---

## Contents

---

|           |                                     |           |
|-----------|-------------------------------------|-----------|
| <b>1</b>  | <b>What is docxtemplater ?</b>      | <b>1</b>  |
| <b>2</b>  | <b>Installation</b>                 | <b>3</b>  |
| <b>3</b>  | <b>Generate a document</b>          | <b>5</b>  |
| <b>4</b>  | <b>Types of tags</b>                | <b>9</b>  |
| <b>5</b>  | <b>Configuration</b>                | <b>15</b> |
| <b>6</b>  | <b>Angular parser</b>               | <b>17</b> |
| <b>7</b>  | <b>Platform Support</b>             | <b>19</b> |
| <b>8</b>  | <b>Error handling</b>               | <b>21</b> |
| <b>9</b>  | <b>Command Line Interface (CLI)</b> | <b>23</b> |
| <b>10</b> | <b>API</b>                          | <b>25</b> |
| <b>11</b> | <b>Frequently asked questions</b>   | <b>27</b> |
| <b>12</b> | <b>Online Demos</b>                 | <b>31</b> |



---

## What is docxtemplater ?

---

docxtemplater is a mail merging tool that is used programmatically and handles conditions, loops, and can be extended to insert anything (tables, html, images).

docxtemplater uses JSON (Javascript objects) as data input, so it can also be used easily from other languages. It handles docx but also pptx templates.

It works in the same way as a templating engine.

There are a lot of solutions like `docx.js`, `docx4j`, ... that generate docx, but they require you to write specific code to create a title, an image, ...

In contrast, docxtemplater is based on the concepts of tags, and each type of tag exposes a feature to the user writing the template.

## Why you shouldn't write it from scratch

Docx is a zipped format that contains some xml. If you want to build a simple replace {tag} by value system, it can become complicated, because the {tag} is internally separated into `<w:t>{</w:t><w:t>tag</w:t><w:t>}</w:t>`.

If you want to have loops to iterate over an array, it will become even more complex.

docxtemplater provides a very simple API that gives you abstraction to deal with loops, conditions, ...

If you need additional features, you can either build your own module, or use one of the free or paid modules that you can find here :

- <https://github.com/open-xml-templating/docxtemplater-image-module>
- <https://modules.docxtemplater.com/>



### Node

npm is the easiest way to install docxtemplater

```
npm install docxtemplater
npm install jszip@2
```

**jszip version 2 is important !** It won't work with jszip version 3

If you want to use the command line interface, you should use the global flag:

```
npm install docxtemplater --global
```

### Browser

You can find `.js` and `.min.js` files for docxtemplater on this repository :

<https://github.com/open-xml-templating/docxtemplater-build/tree/master/build>

You will also need JSZip version 2.x, which you can download here : <https://github.com/Stuk/jszip/tree/v2.x/dist>

### Build it yourself

If you want to build docxtemplater for the browser yourself, here is how you should do :

```
git clone https://github.com/open-xml-templating/docxtemplater.git
cd docxtemplater
npm install
npm test
npm run compile
```

```
./node_modules/.bin/browserify -r "./js/docxtemplater.js" -s docxtemplater > "browser/  
↪docxtemplater.js"  
./node_modules/.bin/uglifyjs "browser/docxtemplater.js" > "browser/docxtemplater.min.  
↪js" --verbose --ascii-only
```

Docxtemplater will be exported to `window.docxtemplater` for easy usage.

The generated files of docxtemplater will be in `/browser` (minified and non minified).

## Bower

You can use bower to install docxtemplater

```
bower install --save docxtemplater
```



---

## Generate a document

---

### Node

```
var JSZip = require('jszip');
var Docxtemplater = require('docxtemplater');

var fs = require('fs');
var path = require('path');

//Load the docx file as a binary
var content = fs
  .readFileSync(path.resolve(__dirname, 'input.docx'), 'binary');

var zip = new JSZip(content);

var doc = new Docxtemplater();
doc.loadZip(zip);

//set the templateVariables
doc.setData({
  first_name: 'John',
  last_name: 'Doe',
  phone: '0652455478',
  description: 'New Website'
});

try {
  // render the document (replace all occurrences of {first_name} by John, {last_
  ↪name} by Doe, ...)
  doc.render()
}
catch (error) {
  var e = {
    message: error.message,
```

```

        name: error.name,
        stack: error.stack,
        properties: error.properties,
    }
    console.log(JSON.stringify({error: e}));
    // The error thrown here contains additional information when logged with JSON.
    ↪stringify (it contains a property object).
    throw error;
}

var buf = doc.getZip()
    .generate({type: 'nodebuffer'});

// buf is a nodejs buffer, you can either write it to a file or do anything else with ↪
    ↪it.
fs.writeFileSync(path.resolve(__dirname, 'output.docx'), buf);

```

You can download [input.docx](<https://github.com/open-xml-templating/docxtemplater/raw/master/examples/tag-example.docx>) and put it in the same folder than your script.

## Browser

```

<html>
  <script src="docxtemplater.js"></script>
  <script src="jszip.js"></script>
  <script src="vendor/file-saver.min.js"></script>
  <script src="vendor/jszip-utils.js"></script>
  <!--
  Mandatory in IE 6, 7, 8 and 9.
  -->
  <!--[if IE]>
    <script type="text/javascript" src="examples/vendor/jszip-utils-ie.js"></
    ↪script>
  <![endif]-->
  <script>
    function loadFile(url, callback) {
      JSZipUtils.getBinaryContent(url, callback);
    }
    loadFile("examples/tag-example.docx", function(error, content) {
      if (error) { throw error };
      var zip = new JSZip(content);
      var doc=new Docxtemplater().loadZip(zip)
      doc.setData({
        first_name: 'John',
        last_name: 'Doe',
        phone: '0652455478',
        description: 'New Website'
      });

      try {
        // render the document (replace all occurrences of {first_name} by John,
        ↪{last_name} by Doe, ...)
        doc.render()
      }
      catch (error) {

```

```
    var e = {
      message: error.message,
      name: error.name,
      stack: error.stack,
      properties: error.properties,
    }
    console.log(JSON.stringify({error: e}));
    // The error thrown here contains additional information when logged with
    ↪JSON.stringify (it contains a property object).
    throw error;
  }

  var out=doc.getZip().generate({
    type:"blob",
    ↪document",
    mimeType: "application/vnd.openxmlformats-officedocument.wordprocessingml.
  }) //Output the document using Data-URI
  saveAs(out,"output.docx")
})
</script>
</html>
```



# CHAPTER 4

---

## Types of tags

---

The syntax is inspired by [Mustache](#). The template is created in Microsoft Word or other software that can save a docx.

### Introduction

With this template (input.docx):

```
Hello {name} !
```

And given the following data (data.json):

```
{
  name: 'John'
}
```

docxtemplater will produce (output.docx):

```
Hello John !
```

### Conditions

Conditions start with a pound and end with a slash. That is `{#hasKitty}` starts a condition and `{/hasKitty}` ends it.

```
{#hasKitty}Cat's name: {kitty}{/hasKitty}
{#hasDog}Dog's name: {dog}{/hasDog}
```

and this data:

```
{
  "first_name": "Jane",
  "hasKitty": true,
```

```
"kitty": "Minie"  
"hasDog": false,  
"dog" :null  
}
```

renders the following:

```
Cat's name: Minie
```

For a more detailed explanation about Conditions, have a look at [Sections](#)

You can also have “else” blocks with [Inverted Sections](#)

## Loops

In docxtemplater, conditions and loops use the same syntax called Sections

The following template:

```
{#products}  
  {name}, {price} €  
{/products}
```

Given the following data:

```
{  
  "products": [  
    { name : "Windows", price: 100},  
    { name : "Mac OSX", price: 200},  
    { name : "Ubuntu", price: 0}  
  ]  
}
```

will result in :

```
Windows, 100 €  
Mac OSX, 200 €  
Ubuntu, 0€
```

## Sections

A section begins with a pound and ends with a slash. That is `{#person}` begins a “person” section while `{/person}` ends it.

The section behaves in the following way:

| Type of the value    | the section is shown      | scope            |
|----------------------|---------------------------|------------------|
| boolean              | once if true              | unchanged        |
| falsy or empty array | never                     |                  |
| non empty array      | for each element of array | element of array |
| object               | once                      | the object       |

This table shows for each type of value, what is the condition for the section to be changed and what is the scope of that section.

If the value is of type **boolean**, the section is shown **once if the value is true**, and the scope of the section is **unchanged**.

## Example 1

If we have the section

```
{#hasProduct}
  {price} €
{/hasProduct}
```

Given the following data:

```
{
  "hasProduct": true,
  "price" : 10
}
```

Since `hasProduct` is a boolean, the section is shown once if `hasProduct` is `true`. Since the scope is unchanged, the subsection `{price} €` will render as `10 €`

## Inverted Sections

An inverted section begins with a caret (hat) and ends with a slash. That is `{^person}` begins a “person” inverted section while `{/person}` ends it.

While sections can be used to render text one or more times based on the value of the key, inverted sections may render text once based on the inverse value of the key. That is, they will be rendered if the key doesn’t exist, is false, or is an empty list. The scope of an inverted section is unchanged.

Template:

```
{#repo}
  <b>{name}</b>
{/repo}
{^repo}
  No repos :(
{/repo}
```

Data:

```
{
  "repo": []
}
```

Output:

```
No repos :(
```

## Sections and newlines

New lines are kept inside sections, so the template :

```
{#repo}
  <b>{name}</b>
{/repo}
{^repo}
  No repos :(
{/repo}
```

Data:

```
{
  "repo": [{name: "John"}]
  "repo": [{name: "Jane"}]
}
```

Will actually render

```
NL
  <b>John</b>
NL
NL
  <b>Jane</b>
NL
```

(where NL represents an emptyline)

The way to make this work as expected is to not put unnecessary new lines after the start of the section and before the end of the section.

For our example , that would be :

```
{#repo} <b>{name}</b>
{/repo} {^repo} No repos :( {/repo}
```

## Raw XML syntax

It is possible to insert raw (unescaped) XML, for example to render a complex table, an equation, ...

With the `rawXML` syntax the whole current paragraph (`w:p`) is replaced by the XML passed in the value.

```
{@rawXml}
```

with this data:

```
{rawXml: '<w:p><w:pPr><w:rPr><w:color w:val="FF0000"/></w:rPr></w:pPr><w:r><w:rPr>
↪<w:color w:val="FF0000"/></w:rPr><w:t>My custom</w:t></w:r><w:r><w:rPr><w:color_
↪w:val="00FF00"/></w:rPr><w:t>XML</w:t></w:r></w:p>' }
```

This will loop over the first parent `<w:p>` tag

If you want to insert HTML styled input, you can also use the docxtemplater html module : <https://modules.docxtemplater.com/modules/html/>

## Dash syntax

When using sections, docxtemplater will try to find on what element to loop over by itself:



---

If between the two tags `{#tag}_____{/tag}`

- there is a tag `<w:tc>`, that means that your loop is inside a table, and it will loop over `<w:tr>` (table row).
- by default, it will loop over `<w:t>`, which is the default Text Tag

With the Dash syntax you can specify the tag you want to loop on: For example, if you want to loop on paragraphs (`w:p`), so that each of the loop creates a new paragraph, you can write :

```
{-w:p loop} {inner} {/loop}
```



You can configure docxtemplater with an options object by using the `setOptions` method.

```
var doc=new Docxtemplater();
doc.loadZip(zip);
doc.setOptions(options)
```

## Custom Parser

The name of this option is *parser* (function).

With a custom parser you can parse the tags to for example add operators like '+', '-', or whatever the way you want to parse expressions.

To enable this, you need to specify a custom parser. You need to create a parser function:

docxtemplater comes shipped with this parser:

```
parser: function(tag) {
  return {
    'get': function(scope) {
      if (tag === '.') {
        return scope;
      }
      else {
        return scope[tag];
      }
    }
  };
},
```

## Custom delimiters

You can set up your custom delimiters with this syntax:

```
new Docxtemplater()  
  .loadZip(zip)  
  .setOptions({delimiters:{start:'[[',end:']']}})
```

## nullGetter

You can customize the value that is shown whenever the parser returns 'null' or undefined. The nullGetter option is a function. By default the nullGetter is the following function

```
nullGetter(part) {  
  if (!part.module) {  
    return "undefined";  
  }  
  if (part.module === "rawxml") {  
    return "";  
  }  
  return "";  
},
```

This means that the default value for simple tags is to show "undefined". The default for rawTags (`{ @rawTag }`) is to drop the paragraph completely (you could enter any xml here).

## CHAPTER 6

---

### Angular parser

---

The angular-parser makes creating complex templates easier. You can for example now use :

```
{user.name}
```

To access the nested name property in the following data :

```
{
  user: {
    name: 'John'
  }
}
```

You can also use +, -, \*, /, >, < operators.

You also get access to filters :

```
expressions.filters.upper = function(input) {
  // This condition should be used to make sure that if your input is undefined,
  ↪ your output will be undefined as well and will not throw an error
  if(!input) return input;
  return input.toUpperCase();
}
```

Will make it possible to write the template {user.name | upper}, and have the resulting string be uppercased.

Here's a code sample for how to use the angularParser :

```
var expressions= require('angular-expressions');
// define your filter functions here, for example, to be able to write {clientname |
↪ lower}
expressions.filters.lower = function(input) {
  // This condition should be used to make sure that if your input is undefined,
  ↪ your output will be undefined as well and will not throw an error
  if(!input) return input;
  return input.toLowerCase();
}
```

```
}
var angularParser = function(tag) {
  return {
    get: tag === '.' ? function(s){ return s;} : expressions.compile(tag)
  };
}
new Docxtemplater().loadZip(zip).setOptions({parser:angularParser})
```

---

**Note:** The `require()` works in the browser if you include `vendor/angular-parser-browser.js`

---

See for a complete reference of all possibilities of angularjs parsing: <http://teropa.info/blog/2014/03/23/angularjs-expressions-cheatsheet.html>

## Conditions

With `angularParser`, you can also use conditions :

```
{#users.length>1}
  They are multiple users
{/}
```

Will render the section only if they are 2 users or more.

docxtemplater works with

- Node.js versions 0.10, 0.11, 0.12 ,4, 5 and 6, with iojs
- Chrome **tested** on version 26
- Firefox 3+ (**tested** on version 21, but should work with 3+)
- Safari **tested**
- IE9+ **tested**
- Android 4.2+ **tested**
- iPads and iPhones v8.1 **tested**

You can test if everything works fine on your browser by using the test runner: <http://javascript-ninja.fr/docxtemplater/v3/test/mocha.html>

## Dependencies

1. [jszip.js](<http://stuk.github.io/jszip/>) to zip and unzip the docx files
2. [xmldom](<https://github.com/jindw/xmldom>) to parse the files as xml





This section is about how to handle Docxtemplater errors.

To be able to see these errors, you need to catch them properly.

```
try {
  // render the document (replace all occurrences of {first_name} by John, {last_
  ↪name} by Doe, ...)
  doc.render()
}
catch (error) {
  var e = {
    message: error.message,
    name: error.name,
    stack: error.stack,
    properties: error.properties,
  }
  console.log(JSON.stringify(e));
  // Handle error
}
```

## Error Schema

All errors thrown by docxtemplater have the following schema:

```
{
  name: One of [GenericError, TemplateError, ScopeParserError, InternalError],
  message: The message of that error,
  properties : {
    explanation: An error that is user friendly (in english), explaining what_
  ↪failed exactly. This error could be shown as is to end users
    id: An identifier of the error that is unique for that type of Error
    ... : The other properties are specific to each type of error.
```

```
}  
}
```

## Error example

If the content of your template is `{user {name}}`, docxtemplater will throw the following error :

```
try {  
  doc.render()  
}  
catch (e) {  
  // All these expressions are true  
  e.name === "TemplateError"  
  e.message === "Unclosed tag"  
  e.properties.explanation === "The tag beginning with '{user ' is unclosed"  
  e.properties.id === "unclosed_tag"  
  e.properties.context === "{user {"  
  e.properties.xtag === "user "  
}
```

---

## Command Line Interface (CLI)

---

This section is about the commandline interface of docxtemplater.

The syntax is the following:

```
docxtemplater config.json
```

The full config.json should be the following:

```
{
  "config.inputFile": "input.docx",
  "config.outputFile": "output.docx",
  "config.debug": true,
  "first_name": "John",
  "last_name": "Smith",
  "age": 62
}
```

## Config.json Syntax

### Config properties:

These are the properties to configure docxtemplater:

```
{
  "config.docxFile": "input.docx", //The input file path
  "config.outputFile": "output.docx", //The output file path
  "config.debug": true //whether to show debug output or not
}
```

### Data properties:

To add data to your template, just use keys that don't start with "config."

```
{  
  "first_name": "John",  
  "last_name": "Smith",  
  "age": 62  
}
```

### Constructor

```
new Docxtemplater()
```

This **function** returns a **new** Docxtemplater Object

### Methods

```
loadZip(zip)
```

You have to pass a zip instance to that method, coming from jszip version 2

```
setData(Tags)
```

Tags:

Type: Object {tag\_name:tag\_replacement}

Object containing **for** each tag\_name, the replacement **for this** tag. For

↪example, **if** you want to replace firstName by David, your Object will be: {"firstName  
↪":"David"}

```
render()
```

This **function** replaces all template variables by their values

```
getZip()
```

This will **return** you the zip that represents the docx. You can then call `.

↪generate` on **this** to generate a buffer, string , ... (see [https://stuk.github.io/jszip/documentation/api\\_jszip/generate.html](https://stuk.github.io/jszip/documentation/api_jszip/generate.html))



---

## Frequently asked questions

---

### Inserting new lines

```
pre = '<w:p><w:r><w:t>';
post = '</w:t></w:r></w:p>';
lineBreak = '<w:br/>';
text = pre + 'testing line 1' + lineBreak + 'testing line 2' + post;
data = {text : text}
docx.setData(data)
```

then in your template, just put `{@text}` instead of the usual `{text}`

If you use the angular-parser, you can also write a filter like this:

```
angularexpressions.filters.raw = function (text) {
  var lines = text.split("\n");
  var pre = "<w:p><w:r><w:t>";
  var post = "</w:t></w:r></w:p>";
  var lineBreak = "<w:br/>";
  return pre + lines.join(lineBreak) + post;
}
```

and then have your docx as : `{@textraw}`

with

```
data = {text: "testing line 1 \n testing line 2"};
docx.setData(data)
```

### Insert HTML formatted text

It is possible to insert HTML formatted text using the [HTML pro module](#)

## Generate smaller docx using compression

The size of the docx output can be big, in the case where you generate the zip the following way:

```
docx.getZip().generate({ type: "nodebuffer" })
```

This is because the zip will not be compressed in that case. To force the compression (which could be slow because it is running in JS for files bigger than 10 MB)

```
var zip = docx.getZip().generate({
  type: "nodebuffer",
  compression: "DEFLATE"
});
```

## Writing if else

To write if/else, see the documentation on sections for if and inverted sections for else.

You can also have conditions with operators > and < using angular parser conditions.

## Conditional Formatting

I plan to develop a pro module to do conditional formatting <https://modules.docxtemplater.com/modules/styling/>. As a workaround, you could also use the [HTML pro module](#)

## Using data filters

You might want to be able to show data a bit differently for each template. For this, you can use the angular parser and the filters functionality.

For example, if a user wants to put something in uppercase, you could write in your template :

```
{ user.name | uppercase }
```

See angular parser for comprehensive documentation

## Performance

Docxtemplater is quite fast, for a pretty complex 50 page document, it can generate 250 output of those documents in 44 seconds, which is about 180ms per document

## Support for IE9 and lower

docxtemplater should work on almost all browsers as of version 1 : IE7 + . Safari, Chrome, Opera, Firefox.

The only 'problem' is to load the binary file into the browser. This is not in docxtemplater's scope, but here is the code that jszip's creator recommends to use to load the zip from the browser:



[https://stuk.github.io/jszip/documentation/howto/read\\_zip.html](https://stuk.github.io/jszip/documentation/howto/read_zip.html)

The following code should load the binary content on all browsers:

```
JSZipUtils.getBinaryContent('path/to/content.zip', function(err, data) {  
  if(err) {  
    throw err; // or handle err  
  }  
  
  var zip = new JSZip(data);  
});
```

## Pptx support

Docxtemplater now handles pptx automatically (since version 3.0.4).

It does so by detecting whether there is a file called “/word/document.xml”, if there is one, the file is “docx”, if not, it is pptx.



# CHAPTER 12

---

Online Demos

---

Including:

- Replace Variables
- Formating
- Angular Parsing
- Loops
- Loops and tables
- Lists
- Raw Xml Insertion



## A

Angular parser, 16

API, 24

## C

Command Line Interface (CLI), 22

Configuration, 13

## E

Errors, 19

## F

FAQ, 25

## G

Generate a Document, 4

Goals, 1

## I

Installation, 1

## P

platform\_support, 18

## T

Types of tags, 7