

---

# **docxtemplater Documentation**

**Edgar Hipp**

**Jan 07, 2020**



---

## 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>21</b>
<b>7</b>	<b>Asynchronous generation</b>	<b>25</b>
<b>8</b>	<b>Platform Support</b>	<b>27</b>
<b>9</b>	<b>Error handling</b>	<b>29</b>
<b>10</b>	<b>Command Line Interface (CLI)</b>	<b>33</b>
<b>11</b>	<b>API</b>	<b>35</b>
<b>12</b>	<b>Frequently asked questions</b>	<b>37</b>
<b>13</b>	<b>Testing</b>	<b>45</b>
<b>14</b>	<b>Online Demos</b>	<b>49</b>
	<b>Index</b>	<b>51</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.

Many solutions like `docx.js`, `docx4j`, `python-docx` can 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.

### 1.1 Why you shouldn't write a similar library from scratch

Docx is a zipped format that contains some xml. If you want to build a simple replace {tag} by value system, it could easily be challenging, because the {tag} is internally separated into :

```
<w:t>{</w:t>
<w:t>tag</w:t>
<w:t>}</w:t>
```

The fact that the tags can be splitted into multiple xml tags makes the code challenging to write. I had to rewrite most of the parsing engine between version 2 and version 3 of docxtemplater to make the code more straightforward : See the migration here : <https://github.com/open-xml-templating/docxtemplater/commit/59af93bd281932da4586175bb2428d28298d1e65>.

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

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

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 at <https://docxtemplater.com/>



### 2.1 Node

npm is the easiest way to install docxtemplater

```
npm install docxtemplater pizzip
```

### 2.2 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 Pizzip, which you can download here : <https://unpkg.com/pizzip@3.0.6/dist/pizzip.js>

### 2.3 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).

## 2.4 Minifying the build

On Browsers that have `window.XMLSerializer` and `window.DOMParser` (all browsers normally have it), you can use that as a replacement for the `xmldom` dependency.

As an example, if you use webpack, you can do the following in your `webpack.config.js` :

```
module.exports = {
  // ...
  // ...
  resolve: {
    alias: {
      xmldom: path.resolve("./node_modules/docxtemplater/es6/browser-versions/
↔xmldom.js"),
    },
  },
  // ...
  // ...
}
```

## 2.5 Bower

You can use bower to install docxtemplater

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



---

## Generate a document

---

### 3.1 Node

```
var PizZip = require('pizzip');
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 PizZip(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) {
  // The error thrown here contains additional information when logged with JSON.
  ↪stringify (it contains a properties object).
```

(continues on next page)

```

var e = {
  message: error.message,
  name: error.name,
  stack: error.stack,
  properties: error.properties,
}
console.log(JSON.stringify({error: e}));
if (error.properties && error.properties.errors instanceof Array) {
  const errorMessages = error.properties.errors.map(function (error) {
    return error.properties.explanation;
  }).join("\n");
  console.log('errorMessages', errorMessages);
  // errorMessages is a humanly readable message looking like this :
  // 'The tag beginning with "foobar" is unopened'
}
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` and put it in the same folder than your JS file.

## 3.2 Browser

```

<html>
  <body>
    <button onclick="generate()">Generate document</button>
  </body>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/docxtemplater/3.14.0/
↳ docxtemplater.js"></script>
  <script src="https://unpkg.com/pizzip@3.0.6/dist/pizzip.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/FileSaver.js/1.3.8/FileSaver.
↳ js"></script>
  <script src="https://unpkg.com/pizzip@3.0.6/dist/pizzip-utils.js"></script>
  <!--
  Mandatory in IE 6, 7, 8 and 9.
  -->
  <!--[if IE]>
    <script type="text/javascript" src="https://unpkg.com/pizzip@3.0.6/dist/
↳ pizzip-utils-ie.js"></script>
  <![endif]-->
  <script>
    function loadFile(url, callback) {
      PizZipUtils.getBinaryContent(url, callback);
    }
    function generate() {
      loadFile("https://docxtemplater.com/tag-example.docx", function(error, content) {
        if (error) { throw error };
        var zip = new PizZip(content);

```

(continues on next page)

(continued from previous page)

```

var doc=new window.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) {
  // The error thrown here contains additional information when logged
↪with JSON.stringify (it contains a properties object).
  var e = {
    message: error.message,
    name: error.name,
    stack: error.stack,
    properties: error.properties,
  }
  console.log(JSON.stringify({error: e}));
  if (error.properties && error.properties.errors instanceof Array) {
    const errorMessages = error.properties.errors.map(function
↪(error) {
      return error.properties.explanation;
    }).join("\n");
    console.log('errorMessages', errorMessages);
    // errorMessages is a humanly readable message looking like this :
    // 'The tag beginning with "foobar" is unopened'
  }
  throw error;
}
var out=doc.getZip().generate({
  type:"blob",
  mimeType: "application/vnd.openxmlformats-officedocument.
↪wordprocessingml.document",
}) //Output the document using Data-URI
saveAs(out,"output.docx")
})
}
</script>
</html>

```

Please note that if you want to load a docx from your filesystem, you will need a webserver or you will be blocked by CORS policy.

*Access to XMLHttpRequest at file.docx from origin 'null' has been blocked by CORS policy*

It is also possible to read the docx from an `<input type="file" id="doc">`, by using the following :

```

var docs = document.getElementById('doc');
function generate() {
  var reader = new FileReader();
  if (docs.files.length === 0) {
    alert("No files selected")
  }
}

```

(continues on next page)

(continued from previous page)

```
reader.readAsBinaryString(docs.files.item(0));

reader.onerror = function (evt) {
  console.log("error reading file", evt);
  alert("error reading file" + evt)
}
reader.onload = function (evt) {
  const content = evt.target.result;
  var zip = new PizZip(content);
  // Same code as in the main HTML example.
}
}
```

# 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.

### 4.1 Introduction

With this template (input.docx):

```
Hello {name} !
```

And given the following data (data.json):

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

docxtemplater will produce (output.docx):

```
Hello John !
```

### 4.2 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*

## 4.3 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€
```

To loop over an array containing primitive data (ex: string):

```
{
  "products": [
    "Windows",
    "Mac OSX",
    "Ubuntu"
  ]
}
```

```
{#products} {.} {/products}
```

Will result in :

---

```
Windows Mac OSX Ubuntu
```

## 4.4 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
falsy or empty array	never	
non empty array	for each element of array	element of array
object	once	the object
other truthy value	once	unchanged

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**.

### 4.4.1 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 €`

## 4.5 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 :(
```

## 4.6 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}
```



## 4.7 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://docxtemplater.com/modules/html/>

## 4.8 Set Delimiter

Set Delimiter tags start and end with an equal sign and change the tag delimiters from `{` and `}` to custom strings.

Consider the following contrived example:

```
* {default_tags}
{=<% %>=}
* <% erb_style_tags %>
<%= { } =%>
* { default_tags_again }
```

Here we have a list with three items. The first item uses the default tag style, the second uses erb style as defined by the Set Delimiter tag, and the third returns to the default style after yet another Set Delimiter declaration.

Custom delimiters may not contain whitespace or the equals sign.

It is also possible to [change the delimiters from docxtemplater.setOptions](#).

## 4.9 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)
```

## 5.1 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 even create a Domain Specific Language to specify your tag values.

To enable this, you need to specify a custom parser.

To understand this option better, it is good to understand how docxtemplater manages the scope.

Whenever docxtemplater needs to render any tag, for example `{name}`, docxtemplater will delegate the retrieval of the value to the scopemanager.

The scopemanager does the following :

- it compiles the tag, by calling `parser('name')` where 'name' is the string representing what is inside the docxtemplater tag. For loop tags, if the tag is `{#condition}`, the passed string is only *condition* (it does not contain the #).

The compilation of that tag should return an object containing a function at the *get* property.

- whenever the tag needs to be rendered, docxtemplater calls `parser('name').get({name: 'John'})`, if `{name: 'John'}` is the current scope.

When inside a loop, for example : `{#users}{name}/{users}`, there are several "scopes" in which it is possible to evaluate the `{name}` property. The "deepest" scope is always evaluated first, so if the data is : `{users: [{name: "John"}], name: "Mary"}`, the parser calls the function `parser('name').get({name: "John"})`. Now if the returned value from the `.get` method is `null` or `undefined`, docxtemplater will call the same parser one level up, until it reaches the end of the scope.

If the root scope also returns *null* or *undefined* for the *.get* call, then the value from the *nullGetter* is used.

As a second argument to the *parser()* call, you receive more meta data about the tag of the document (and you could check if it is a loop tag for example).

As a second argument to the *get()* call, you receive more meta data about the scope, including the full *scopeList*.

Lets take an example, If your template is :

```
Hello {user}
```

And we call *doc.setData({user: "John"})*

docxtemplater uses by default the following parser :

```
doc.setOptions({
  parser: function(tag) {
    // tag is "user"
    return {
      'get': function(scope) {
        // scope will be {user: "John"}
        if (tag === '.') {
          return scope;
        }
        else {
          // Here we return the property "user" of the object {user: "John"}
          return scope[tag];
        }
      }
    };
  },
});
```

A very useful parser is the *angular-expressions* parser, which has implemented useful features.

See [angular parser](#) for comprehensive documentation

The parser function is given two arguments,

For the template

```
Hello {#users}{.}{/}
```

With the data :

```
{users: ['Mary', 'John']}
```

```
function parser(scope, context) {
  console.log(scope);
  console.log(context);
}
```

For the tag *.* in the first iteration, the arguments will be :

```
scope = { "name": "Jane" }
context = {
  "num": 1, // This corresponds to the level of the nesting, the {#users} tag is
  ↳ level 0, the {.} is level 1
  "scopeList": [
    {
```

(continues on next page)

(continued from previous page)

```

    "users": [
      {
        "name": "Jane"
      },
      {
        "name": "Mary"
      }
    ]
  },
  {
    "name": "Jane"
  }
],
"scopePath": [
  "users"
],
"scopePathItem": [
  0
]
// Together, scopePath and scopePathItem describe where we are in the data, in this
←case, we are in the tag users[0] (the first user)
}

```

Here is an example parser that allows you to lowercase or uppercase the data if writing your tag as `{user[lower]}` or `{user[upper]}`:

```

doc.setOptions({
  parser: function(tag) {
    // tag is "foo[lower]"
    let changeCase = false;
    if(tag.endsWith("[lower]")) {
      changeCase = "lower";
    }
    if(tag.endsWith("[upper]")) {
      changeCase = "upper";
    }
    return {
      'get': function(scope) {
        let result = null;
        // scope will be {user: "John"}
        if (tag === '.') {
          result = scope;
        }
        else {
          // Here we use the property "user" of the object {user: "John"}
          result = scope[tag];
        }

        if (typeof result === "string") {
          if(changeCase === "upper") {
            return result.toUpperCase();
          }
          else if(changeCase === "lower") {
            return result.toLowerCase();
          }
        }
      }
    }
  }
}

```

(continues on next page)

```
        return result;
    }
};
},
});
```

As an other example, it is possible to use the `{$index}` tag inside a loop by using following parser :

```
function parser(tag) {
    return {
        get(scope, context) {
            if (tag === "$index") {
                const indexes = context.scopePathItem;
                return indexes[indexes.length - 1];
            }
            if (tag === "$isLast") {
                const totalLength =
                    context.scopePathLength[context.scopePathLength.length - 1];
                const index =
                    context.scopePathItem[context.scopePathItem.length - 1];
                return index === totalLength - 1;
            }
            if (tag === "$isFirst") {
                const index =
                    context.scopePathItem[context.scopePathItem.length - 1];
                return index === 0;
            }
            return scope[tag];
        },
    };
}
```

## 5.2 Custom delimiters

You can set up your custom delimiters with this syntax:

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

## 5.3 paragraphLoop

The `paragraphLoop` option has been added in version 3.2.0.

It is recommended to turn that option on, since it makes the rendering a little bit easier to reason about.

However since it breaks backwards-compatibility, it is turned off by default.

```
new Docxtemplater()
    .loadZip(zip)
    .setOptions({paragraphLoop:true})
```

It allows to loop around paragraphs without having additional spacing.

When you write the following template

```
The users list is :
{#users}
{name}
{/users}
End of users list
```

Most users of the library would expect to have no spaces between the different names.

The output without the option is as follows :

```
The users list is :

John

Jane

Mary

End of users list
```

With the paragraphLoop option turned on, the output becomes :

```
The users list is :
John
Jane
Mary
End of users list
```

The rule is quite simple :

If the opening loop (`{#users}`) and the closing loop (`{/users}`) are both on separate paragraphs, treat the loop as a paragraph loop (eg create one new paragraph for each loop) where you remove the first and last paragraphs (the ones containing the loop tags).

## 5.4 nullGetter

You can customize the value that is shown whenever the parser (documented above) returns ‘null’ or undefined. By default the nullGetter is the following function

```
nullGetter(part, scopeManager) {
  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).

The scopeManager variable contains some meta information about the tag, for example, if the template is : `{#users}{name}{/users}` and the tag name is undefined, `scopeManager.scopePath === ["users", "name"]`

## 5.5 linebreaks

You can enable linebreaks, eg if your data contains newlines, those will be shown as linebreaks in the document

```
doc.setOptions({linebreaks:true});  
doc.setData({text: "My text,\nmultiline"});  
doc.render();
```



## 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 :

It is possible to write the template {user.name | upper}, and have the resulting string be uppercased.

```
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();
}
```

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

```
var expressions = require('angular-expressions');
var merge = require("lodash/merge");
// 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;
```

(continues on next page)

(continued from previous page)

```

    return input.toLowerCase();
}
function angularParser(tag) {
  if (tag === '.') {
    return {
      get: function(s){ return s;}
    };
  }
  const expr = expressions.compile(tag.replace(/'|"|`|\/g, "\'"));
  return {
    get: function(scope, context) {
      let obj = {};
      const scopeList = context.scopeList;
      const num = context.num;
      for (let i = 0, len = num + 1; i < len; i++) {
        obj = merge(obj, scopeList[i]);
      }
      return expr(scope, obj);
    }
  };
}
new Docxtemplater().loadZip(zip).setOptions({parser:angularParser})

```

**Note:** The require() will not work in a browser, you have to use a module bundler like [webpack](http://webpack.github.io/) or [browserify](http://browserify.org/). Alternatively, you can download an outdated version at <https://raw.githubusercontent.com/open-xml-templating/docxtemplater/6c8c76210d555fd0f6b3dbc927522a3805f17469/vendor/angular-parse-browser.js>

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

## 6.1 Conditions

With the angularParser option set, you can also use conditions :

```

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

```

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

It also handles the boolean operators AND &&, OR ||, +, -, the ternary operator a ? b : c, operator precedence with parenthesis (a && b) || c, and many other javascript features.

For example, it is possible to write the following template :

```

{#generalCondition}
{#cond1 || cond2}
Paragraph 1
{/}
{#cond2 && cond3}
Paragraph 2
{/}

```

(continues on next page)

(continued from previous page)

```
{#cond4 ? users : usersWithAdminRights}  
Paragraph 3  
{/  
They are {users.length} users.  
{/generalCondition}
```



---

## Asynchronous generation

---

You can have promises in your data.

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

try {
  doc.compile(); // You need to compile your document first.
}
catch (error) {
  // The error thrown here contains additional information when logged with JSON.
  ↪stringify (it contains a properties object).
  var e = {
    message: error.message,
    name: error.name,
    stack: error.stack,
    properties: error.properties,
  }
  console.log(JSON.stringify({error: e}));
  if (error.properties && error.properties.errors instanceof Array) {
    const errorMessages = error.properties.errors.map(function (error) {
      return error.properties.explanation;
    }).join("\n");
    console.log('errorMessages', errorMessages);
    // errorMessages is a humanly readable message looking like this :
    // 'The tag beginning with "foobar" is unopened'
  }
  throw error;
}

doc.resolveData({user: new Promise(resolve) { setTimeout(()=> resolve('John'), 1000)}}
↪)
  .then(function() {
    doc.render();
```

(continues on next page)

```
var buf = doc.getZip()
    .generate({type: 'nodebuffer'});
fs.writeFileSync(path.resolve(__dirname, 'output.docx'), buf);
}).catch(function(error) {
    // The error thrown here contains additional information when logged with
    ↪JSON.stringify (it contains a properties object).
    var e = {
        message: error.message,
        name: error.name,
        stack: error.stack,
        properties: error.properties,
    }
    console.log(JSON.stringify({error: e}));
    if (error.properties && error.properties.errors instanceof Array) {
        const errorMessages = error.properties.errors.map(function (error) {
            return error.properties.explanation;
        }).join("\n");
        console.log('errorMessages', errorMessages);
        // errorMessages is a humanly readable message looking like this :
        // 'The tag beginning with "foobar" is unopened'
    }
    throw error;
});
```

---

## Platform Support

---

docxtemplater works on most modern platforms, and also some older ones. Here is a list of what is tested regularly :

- Node.js versions 6, 7, 8, 9, 10, 11, 12 and all future versions (older versions will also work, but support has ended)
- Chrome version 58,71,73
- Firefox 55,60,66
- Safari 11,12
- IE10, IE11, Edge 16-18
- Android 4.2+
- iPads and iPhones v8.1, 10.3

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

### 8.1 Dependencies

1. [PizZip](#) to zip and unzip the docx files
2. [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) {
  // The error thrown here contains additional information when logged with JSON.
  ↪stringify (it contains a properties object).
  var e = {
    message: error.message,
    name: error.name,
    stack: error.stack,
    properties: error.properties,
  }
  console.log(JSON.stringify({error: e}));
  if (error.properties && error.properties.errors instanceof Array) {
    const errorMessages = error.properties.errors.map(function (error) {
      return error.properties.explanation;
    }).join("\n");
    console.log('errorMessages', errorMessages);
    // errorMessages is a humanly readable message looking like this :
    // 'The tag beginning with "foobar" is unopened'
  }
  throw error;
}
```

## 9.1 Error Schema

All errors thrown by docxtemplater have the following schema:

```

{
  name: One of [GenericError, TemplateError, ScopeParserError, InternalError,
↳MultiError],
  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.
  }
}

```

## 9.2 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 "
}

```

## 9.3 List of all Error Identifiers

All errors can be identified with their id (`e.properties.id`).

The ids are :

**multi\_error**: This error means that multiple errors were found in the template (1 or more). See below for handling these errors.

**unopened\_tag**: This error happens if a tag is closed but not opened. For example with the following template :

```
Hello name} !
```

**unclosed\_tag**: This error happens if a tag is opened but not closed. For example with the following template :

```
Hello {name !
```

**no\_xml\_tag\_found\_at\_left** and **no\_xml\_tag\_found\_at\_right**: This error happens if a rawXMLTag doesn't find a `<w:p>` element

```

<w:p><w:t>{@raw}</w:t>
// Note that the `</w:p>` tag is missing.

```

**utf8\_decode** is an internal error, please report it if you see it

**xmltemplater\_content\_must\_be\_string** is an internal error that happens if you try to template something that is not a string (a number for example)

**raw\_xml\_tag\_should\_be\_only\_text\_in\_paragraph** happens when a rawXMLTag { @raw } is not the only text in the paragraph. For example, writing ' { @raw } ' (Note the spaces) is not acceptable because the { @raw } tag replaces the full paragraph. We prefer to throw an Error now rather than have “strange behavior” because the spaces “disappeared”.

To correct this error, you have to add manually the text that you want in your raw tag. (Or you can use the <https://docxtemplater.com/modules/word-run/> which adds a tag that can replace rawXML inside a tag).

Writing

```
` { @my_first_tag } { my_second_tag } `
```

Or even

```
` Hello { @my_first_tag } `
```

Is misusing docxtemplater.

The @ at the beginning means “replace the xml of **the current paragraph** with scope.my\_first\_tag” so that means that everything else in that Paragraph will be removed.

A workaround is to put the text of the second tag in the first tag. (The tag must of course be valid xml)

```
Hello { @raw } !
```

**unclosed\_loop** and **unopened\_loop** happen when a loop is closed but never opened : for example

```
{ #users } { name }
```

or

```
{ name } { /users }
```

**closing\_tag\_does\_not\_match\_opening\_tag** happens when a loop is closed but doesn't match the opening tag

```
{ #users } { name } { /people }
```

**scopeparser\_compilation\_failed** happens when your parser throws an error during compilation. The parser is defined in doc.setOptions({parser: function parser(tag) {}})

For example, if your template is :

```
{ name++ }
```

and you use the angularParser, you will have this error. The error happens when you call parser('name++'); The underlying error can be read in *e.properties.rootError*

**unimplemented\_tag\_type** happens when a tag type is not implemented. It should normally not happen, unless you changed docxtemplater code.

**malformed\_xml** happens when a xml file of the document cannot be parsed correctly.

**loop\_position\_invalid** happens when a loop would produce invalid XML.

For example, if you write :

```
=====
| header1 | header2 |
-----
| { #users } | content |
```

(continues on next page)

(continued from previous page)

```
=====
{/users}
```

this is not allowed since a loop that starts in a table should also end in that table.

## 9.4 Handling multiple errors

docxtemplater now has the ability to detect multiple errors in your template. If it detects multiple errors, it will throw an error that has the id **multi\_error**

You can then have the following to view all errors :

```
e.properties.errors.forEach(function(err) {
  console.log(err);
});
```

## CHAPTER 10

---

### Command Line Interface (CLI)

---

This section is about the commandline interface of docxtemplater.

To install the cli, please use this command :

```
npm install -g docxtemplater-cli
```

<https://github.com/open-xml-templating/docxtemplater-cli>

The syntax is the following:

```
docxtemplater input.docx data.json output.docx
```



### 11.1 Constructor

```
new Docxtemplater()
```

This function returns a new Docxtemplater Object

### 11.2 Methods

```
loadZip(zip)
```

You have to pass a zip instance to that method, coming from pizzip or 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://github.com/open-xml-templating/pizzip/blob/master/documentation/api\\_pizzip/generate.md](https://github.com/open-xml-templating/pizzip/blob/master/documentation/api_pizzip/generate.md))

(continues on next page)

(continued from previous page)

`attachModule(module)`

This will attach a module to the docxtemplater instance, which is usually used to  
→add new generation features (possibility to include images, HTML, ...). Pro modules  
→can be bought on <https://docxtemplater.com/>

This method can be called multiple times, for example : ``doc.loadZip(zip).  
→attachModule(imageModule).attachModule(htmlModule)``



---

## Frequently asked questions

---

### 12.1 Inserting new lines

```
doc.setOptions({linebreaks: true})
```

then in your data, if a string contains a newline, it will be translated to a linebreak in the document.

### 12.2 Insert HTML formatted text

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

### 12.3 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"
});
```

### 12.4 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](#).

## 12.5 Conditional Formatting

With the [PRO styling module](#) it is possible to have a table cell be styled depending on a given condition (for example).

## 12.6 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

## 12.7 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.

There is also an interesting blog article <https://javascript-ninja.fr/optimizing-speed-in-node-js/> that explains how I optimized loops in docxtemplater.

## 12.8 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 recommended code to load the zip from the browser:

[https://github.com/open-xml-templating/pizzip/blob/master/documentation/howto/read\\_zip.md](https://github.com/open-xml-templating/pizzip/blob/master/documentation/howto/read_zip.md)

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

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

## 12.9 Get list of placeholders

To be able to construct a form dynamically or to validate the document beforehand, it can be useful to get access to all placeholders defined in a given template. Before rendering a document, docxtemplater parses the Word document

into a compiled form. In this compiled form, the document is stored in an AST which contains all the necessary information to get the list of the variables and list them in a JSON object.

With the simple inspection module, it is possible to get this compiled form and show the list of tags. suite :

```
var InspectModule = require("docxtemplater/js/inspect-module");
var iModule = InspectModule();
doc.attachModule(iModule);
doc.render(); // doc.compile can also be used to avoid having runtime errors
var tags = iModule.getAllTags();
console.log(tags);
```

With the following template :

```
{company}

{#users}
{name}
{age}
{/users}
```

It will log this object :

```
{
  "company": {},
  "users": {
    "name": {},
    "age": {}
  },
}
```

You can also get a more detailed tree by using :

```
console.log(iModule.fullInspected["word/document.xml"]);
```

The code of the inspect-module is very simple, and can be found here : <https://github.com/open-xml-templating/docxtemplater/blob/master/es6/inspect-module.js>

## 12.10 Convert to PDF

It is not possible to convert docx to PDF with docxtemplater, because docxtemplater is a templating engine and doesn't know how to render a given document. There are many tools to do this conversion.

The first one is to use *libreoffice headless*, which permits you to generate a PDF from a docx document :

You just have to run :

```
libreoffice --headless --convert-to pdf --outdir . input.docx
```

This will convert the input.docx file into input.pdf file.

The rendering is not 100% perfect, since it uses libreoffice and not microsoft word. If you just want to render some preview of a docx, I think this is a possible choice. You can do it from within your application by executing a process, it is not the most beautiful solution but it works.

If you want something that does the rendering better, I think you should use some specialized software. PDFtron is one of them, I haven't used it myself, but I know that some of the users of docxtemplater use it. (I'm not affiliated to PDFtron in any way).

## 12.11 Pptx support

Docxtemplater handles pptx files without any special configuration (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.

## 12.12 My document is corrupted, what should I do ?

If you are inserting multiple images inside a loop, it is possible that word cannot handle the docPr attributes correctly. You can try to add the following code just after doing `const doc = new Docxtemplater()`:

```
const doc = new Docxtemplater();
doc.attachModule({
  set(options) {
    if (options.Lexer) {
      this.Lexer = options.Lexer;
    }
    if (options.zip) {
      this.zip = options.zip;
    }
  },
  on(event) {
    if (event !== "syncing-zip") {
      return;
    }
    const zip = this.zip;
    const Lexer = this.Lexer;
    let prId = 1;
    function setSingleAttribute(partValue, attr, attrValue) {
      const regex = new RegExp(`<.* ${attr}="` + ([^"]+) + `".*`>`);
      if (regex.test(partValue)) {
        return partValue.replace(regex, `${prId}${attrValue}$3`);
      }
      let end = partValue.lastIndexOf(">");
      if (end === -1) {
        end = partValue.lastIndexOf(">");
      }
      return (
        partValue.substr(0, end) +
        ` ${attr}="${attrValue}"` +
        partValue.substr(end)
      );
    }
    zip.file(/\.xml$/).forEach(function(f) {
      let text = f.asText();
      const xmllixed = Lexer.xmlparse(text, {
        text: [],
        other: ["wp:docPr"],
      });
      if (xmllixed.length > 1) {
        text = xmllixed.reduce(function(fullText, part) {
          if (part.tag === "wp:docPr") {
            return fullText + setSingleAttribute(part.value, "id",
↵prId++);
```

(continues on next page)

(continued from previous page)

```

        }
        return fullText + part.value;
    }, "");
    }
    zip.file(f.name, text);
  });
}
});

```

## 12.13 Attaching modules for extra functionality

If you have created or have access to docxtemplater PRO modules, you can attach them with the following code :

```

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

// You can call attachModule for each modules you wish to include
doc.attachModule(imageModule)
doc.attachModule(htmlModule)

//set the templateVariables
doc.setData(data);

```

## 12.14 Ternaries are not working well with angular-parser

There is a common issue which is to use ternary on scopes that are not the current scope, which makes the ternary appear as if it always showed the second option.

For example, with following data :

```

doc.setData({
  user: {
    gender: 'F',
    name: "Mary",
    hobbies: [{
      name: 'play football',
    }, {
      name: 'read books',
    }]
  }
})

```

And by using the following template :

```

{#user}
{name} is a kind person.

{#hobbies}
- {gender == 'F' : 'She' : 'He'} likes to {name}
{/hobbies}
{/}

```

This will print :

```
Mary is a kind person.  
  
- He likes to play football  
- He likes to read books
```

Note that the pronoun “He” is used instead of “She”.

The reason for this behavior is that the `{gender == 'F' : "She" : "He"}` expression is evaluating in the scope of hobby, where `gender` does not even exist. Since the condition `gender == 'F'` is false (since `gender` is undefined), the return value is “He”. However, in the scope of the hobby, we do not know the gender so the return value should be null.

We can instead write a custom filter that will return “She” if the input is “F”, “He” if the input is “M”, and null if the input is anything else.

The code would look like this :

```
expressions.filters.pronoun = function(input) {  
  if(input === "F") {  
    return "She";  
  }  
  if(input === "M") {  
    return "He";  
  }  
  return null;  
}
```

And use the following in your template :

```
{#user}  
{name} is a kind person.  
  
{#hobbies}  
- {gender | pronoun} likes to {name}  
{/hobbies}  
{/}
```

## 12.15 Multi scope expressions do not work with the angularParser

If you would like to have multi-scope expressions with the angularparser, for example :

You would like to do : `{#users}{ date - age }{/users}`, where `date` is in the “global scope”, and `age` in the subscope `users`, as in the following data :

```
{  
  "date": 2019,  
  "users": [  
    {  
      "name": "John",  
      "age": 44  
    },  
    {  
      "name": "Mary",  
      "age": 22  
    }  
  ]  
}
```

(continues on next page)

(continued from previous page)

```

]
}

```

You can make use of a feature of the `angularParser` and the fact that `docxtemplater` gives you access to the whole `scopeList`.

```

var expressions = require("angular-expressions");
var merge = require("lodash/merge");
function angularParser(tag) {
  if (tag === ".") {
    return {
      get(s) {
        return s;
      },
    };
  }
}
const expr = expressions.compile(tag.replace(/('|"|`|`)/g, ""));
return {
  get(scope, context) {
    let obj = {};
    const scopeList = context.scopeList;
    const num = context.num;
    for (let i = 0, len = num + 1; i < len; i++) {
      obj = merge(obj, scopeList[i]);
    }
    return expr(scope, obj);
  },
};
}
doc.setOptions({parser: angularParser})

```

## 12.16 Access to XMLHttpRequest at file.docx from origin 'null' has been blocked by CORS policy

This happens if you use the HTML sample script but are not using a webserver.

If your browser window shows a URL starting with `file://`, then you are not using a webserver, but the filesystem itself.

For security reasons, the browsers don't let you load files from the local file system.

To do this, you have to setup a small web server.

The simplest way of starting a webserver is to run following command :

```

npx http-server
# if you don't have npx, you can also do :
# npm install -g http-server && http-server .

```

On your production server, you should probably use a more robust webserver such as `nginx`, or any webserver that you are currently using for static files.





This page documents how docxtemplater is tested.

First, there are multiple types of tests done in docxtemplater

- **Integration tests**, that are tests where we take a real .docx document, some JSON data, render the document and then verify that it the same as the expected document (this can be seen as snapshot testing)
- **Regression tests**, that are tests where we take real or fake docx to ensure that bugfixes that have been found can't occur in the future
- **Unit tests**, that help understand the internals of docxtemplater, and allows to verify that the internal data structures of the parsed template are correct
- **Speed tests**, that help to optimize the speed of docxtemplater

## 13.1 Integration

The integration tests are in `es6/tests/integration.js`

```
it("should work with table pptx", function () {
  const doc = createDoc("table-example.pptx");
  doc.setData({users: [{msg: "hello", name: "mary"}, {msg: "hello", name: "john"}]}).render();
  ↪ shouldBeSame({doc, expectedName: "expected-table-example.pptx"});
});
```

All of the test documents are in the folder *examples/*

- We first load a document from `table-example.pptx`
- We then set data and render the document.
- We then verify that the document is the same as “`expected-table-example.pptx`”

`shouldBeSame` will, for each XML file that is inside the zip document, pretty print it, and then compare them. That way, we have a more beautiful diff and spacing differences do not matter in the output document.

## 13.2 Regression tests

There are many regression tests, e.g. tests that are there to ensure that bugs that occurred once will not appear again in the future.

A good example of such a test is

Docxtemplater <https://github.com/open-xml-templating/docxtemplater/issues/14>

Docxtemplater was not able to render text that was written in Russian (because of an issue with encoding).

```
it("should insert russian characters", function () {
  const russian = ""
  const doc = createDoc("tag-example.docx");
  const zip = new PizZip(doc.loadedContent);
  const d = new Docxtemplater().loadZip(zip);
  d.setData({last_name: russian});
  d.render();
  const outputText = d.getFullText();
  expect(outputText.substr(0, 7)).to.be.equal(russian);
});
```

This test ensures that the output of the document is correct.

Every time we correct a bug, we should also add a regression test to make sure that bug cannot appear in the future.

## 13.3 Unit tests

The input/output for the unit tests can be found in `es6/tests/fixtures.js` :

For example

```
simple: {
  it: "should handle {user} with tag",
  content: "<w:t>Hi {user}</w:t>",
  scope: {
    user: "Foo",
  },
  result: '<w:t xml:space="preserve">Hi Foo</w:t>',
  lexed: [
    {type: "tag", position: "start", value: "<w:t>", text: true},
    {type: "content", value: "Hi ", position: "insidetag"},
    {type: "delimiter", position: "start"},
    {type: "content", value: "user", position: "insidetag"},
    {type: "delimiter", position: "end"},
    {type: "tag", value: "</w:t>", text: true, position: "end"},
  ],
  parsed: [
    {type: "tag", position: "start", value: "<w:t>", text: true},
    {type: "content", value: "Hi ", position: "insidetag"},
    {type: "placeholder", value: "user"},
    {type: "tag", value: "</w:t>", text: true, position: "end"},
  ],
  postparsed: [
    {type: "tag", position: "start", value: '<w:t xml:space="preserve">', text: true},
    {type: "content", value: "Hi ", position: "insidetag"},
```

(continues on next page)

(continued from previous page)

```
    {type: "placeholder", value: "user"},  
    {type: "tag", value: "</w:t>", text: true, position: "end"},  
  ],  
},
```

There you can see what the different steps of docxtemplater are, lex, parse, postparse.

## 13.4 Speed tests

To ensure that there is no regression on the speed of docxtemplater, we test the performance by generating multiple documents and we expect that the time to generate these documents should be less than for example 100ms.

These tests can be found in `es6/tests/speed.js`

For example for this test:

```
it("should be fast for loop tags", function () {  
  const content = "<w:t>{#users}{name}{/users}</w:t>";  
  const users = [];  
  for (let i = 1; i <= 1000; i++) {  
    users.push({name: "foo"});  
  }  
  const time = new Date();  
  createXmlTemplaterDocx(content, {tags: {users}}).render();  
  const duration = new Date() - time;  
  expect(duration).to.be.below(60);  
});
```

Here we verify that rendering a loop of 1000 items takes less than 60ms. This happens to also be a regression test, because there was a problem when generating documents with loops (the loops became very slow for more than 500 items), and we now ensure that such a regression cannot occur in the future.



# CHAPTER 14

---

Online Demos

---

Including:

- Replace Variables
- Conditions
- Loops
- Loops and tables
- Lists
- Raw Xml Insertion
- HTML
- Image
- Slides
- Subtemplate



## A

Angular parser, 20  
API, 33  
Async, 23

## C

Command Line Interface (*CLI*), 32  
Configuration, 13

## E

Errors, 27

## F

FAQ, 36

## G

Generate a Document, 4  
Goals, 1

## I

Installation, 1

## P

platform\_support, 26

## T

Testing, 43  
Types of tags, 8