docxtemplater Documentation

Edgar Hipp

Sep 29, 2021

Contents

CHAPTER 1

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 straighforward: 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/

CHAPTER 2

Installation

2.1 Node

npm is the easiest way to install docxtemplater

```
npm install docxtemplater pizzip
```

2.2 Browser (Angular, React, Next.JS, Vue)

For React, Angular, and Vue, you can use the npm packages and use these code samples from the FAQ:

- React
- Angular
- Vue
- Next.js

2.3 Browser (JS files)

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

You will also need Pizzip, which you can download here

2.4 Build the JS Files 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.

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

2.5 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"),
        },
    },
    // ...
    // ...
}
```

2.6 Bower

You can use bower to install docxtemplater

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

When using bower, you can include the following script tag in your HTML:

```
<script src="bower_components/docxtemplater/build/docxtemplater-latest.min.js"></
↪script>
```

This tag will expose docxtemplater in *window.docxtemplater*.

CHAPTER 3

Generate a document

3.1 Node

```
var PizZip = require('pizzip');
var Docxtemplater = require('docxtemplater');

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

// The error object contains additional information when logged with JSON.stringify␣
↪(it contains a properties object containing all suberrors).
function replaceErrors(key, value) {
    if (value instanceof Error) {
        return Object.getOwnPropertyNames(value).reduce(function(error, key) {
            error[key] = value[key];
            return error;
        }, {});
    }
    return value;
}

function errorHandler(error) {
    console.log(JSON.stringify({error: error}, replaceErrors));

    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;
}
```

(continues on next page)

(continued from previous page)

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

var zip = new PizZip(content);
var doc;
try {
    doc = new Docxtemplater(zip, { paragraphLoop: true, linebreaks: true });
} catch(error) {
    // Catch compilation errors (errors caused by the compilation of the template:␣
↪misplaced tags)
    errorHandler(error);
}

try {
    // render the document (replace all occurences of {first_name} by John, {last_
↪name} by Doe, ...)
    doc.render({
        first_name: 'John',
        last_name: 'Doe',
        phone: '0652455478',
        description: 'New Website'
    })
}
catch (error) {
    // Catch rendering errors (errors relating to the rendering of the template:␣
↪angularParser throws an error)
    errorHandler(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/3.25.4/
↪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.
```

(continues on next page)

(continued from previous page)

```
    -->
    <!--[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 };

            // The error object contains additional information when logged with JSON.
↪stringify (it contains a properties object containing all suberrors).
            function replaceErrors(key, value) {
                if (value instanceof Error) {
                    return Object.getOwnPropertyNames(value).reduce(function(error,␣
↪key) {
                        error[key] = value[key];
                        return error;
                    }, {});
                }
                return value;
            }

            function errorHandler(error) {
                console.log(JSON.stringify({error: error}, replaceErrors));

                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 zip = new PizZip(content);
            var doc;
            try {
                doc = new window.docxtemplater(zip, { paragraphLoop: true,␣
↪linebreaks: true });
            } catch(error) {
                // Catch compilation errors (errors caused by the compilation of the␣
↪template: misplaced tags)
                errorHandler(error);
            }

            try {
                // render the document (replace all occurences of {first_name} by␣
↪John, {last_name} by Doe, ...)
                doc.render({
                    first_name: 'John',
```

(continues on next page)

(continued from previous page)

```
                    last_name: 'Doe',
                    phone: '0652455478',
                    description: 'New Website'
                });
            }
            catch (error) {
                // Catch rendering errors (errors relating to the rendering of the␣
↪template: angularParser throws an error)
                errorHandler(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")
    }
    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.
    }
}
```

3.3 React, Angular, Vue, Next.JS

There are examples of usage to generate a document in the FAQ for the following libraries:

- React
- Angular

- Vue
- Next.js

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 detailled 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**.

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}
```

(continues on next page)

(continued from previous page)

```
  No repos :(
{/repo}
```

Data:

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

Output:

```
No repos :(
```

4.6 Sections and newlines

New lines are kept inside sections, so the template:

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

Data:

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

Will actually render

```
NL
  John
NL
NL
  Jane
NL
```

(where NL represents an emptyline)

The easiest to make this work is to enable the paragraphLoop option, like this :

```
// Now, all sections in the form of :
// {#section}
// something
// {/section}
// will keep just the inner paragraphs, and drop the newlines of the outer section
const doc = new Docxtemplater(zip, {paragraphLoop: true});
```

An other less recommended way if you don't want to set this option, is to remove the new lines after the start of the section and before the end of the section.

For our example , that would be:

```
{#repo} {name}
{/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:

```
doc.render({
    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 Lambdas

```
const doc = new Docxtemplater(zip);
doc.render({
  userGreeting: (scope) => {
```

(continues on next page)

(continued from previous page)

```
    return "How is it going, " + scope.user + " ? ";
  },
  users: [
    {
      name: "John",
    },
    {
      name: "Mary",
    },
  ],
});
```

With the following template :

```
`txt {#users} {userGreeting} {/} `
```

It will call the function userGreeting twice, with the current scope as first argument, and the scopeManager as the second argument.

4.9 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 by using docxtemplater options object.

4.10 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}
```

CHAPTER 5

Configuration

You can configure docxtemplater with an options object by using the *v4 constructor* with two arguments.

```
var doc = new Docxtemplater(zip, 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 those features, you need to specify a custom parser.

5.1.1 Introduction

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

Whenever docxtemplater needs to render any tag, for example *{name}*, docxtemplater will use a scopemanager to retrieve the value for a given tag.

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 additional meta data about the tag of the document (and you can for example test 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 you call *doc.render({user: "John"})*

Then you will have the following:

```
const options = {
    // This is how docxtemplater is configured by default
    parser: function(tag) {
      // tag is the string "user", or whatever you have put inside the
      // brackets, eg if your tag was {a==b}, then the value of tag would be
      // "a==b"
      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];
          }
        }
      };
    },
};
const doc = new Docxtemplater(zip, options);
```

5.1.2 Angular Parser

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

See angular parser for comprehensive documentation

5.1.3 Deep Dive on the parser

The parser get function is given two arguments,

For the template

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

Using following data:

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

And with this parser

```
const options = {
    // This is how docxtemplater is configured by default
    parser: function(tag) {
      return {
         get: function parser(scope, context) [
            console.log(scope);
            console.log(context);
            return scope[tag];
         }
     }
};
const doc = new Docxtemplater(zip, options);
```

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": [
    {
      "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)
}
```

5.1.4 Simple Parser example for [lower] and [upper]

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

```
options = {
    parser: function(tag) {
      // tag can be "user[lower]", "user", or "user[upper]"
      const lowerRegex = /\[lower\]$/;
```

(continues on next page)

(continued from previous page)

```
      const upperRegex = /\[upper\]$/;
      let changeCase = "";
      if(lowerRegex.test(tag)) {
        changeCase = "lower";
        // transform tag from "user[lower]" to "user"
        tag = tag.replace(lowerRegex, "")
      }
      if(upperRegex.test(tag)) {
        changeCase = "upper";
        // transform tag from "user[upper]" to "user"
        tag = tag.replace(upperRegex, "")
      }
      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();
            }
          }
          return result;
        }
      };
    },
    paragraphLoop: true,
    linebreaks: true,
};
new Docxtemplater(zip, options);
```

5.1.5 Simple Parser example for {$index} and {$isLast} inside loops

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];
```

(continues on next page)

(continued from previous page)

```
                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];
        },
    };
}
```

5.1.6 Parser example to avoid using the parent scope if a value is null on the main scope

When using following template:

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

With following data:

```
doc.render({
    name: 'Santa Katerina',
    products: [
      {
        price: '$3.99'
      }
    ]
});
```

The {name} tag will use the "root scope", since it is not present in the products array.

If you explicitly don't want this behavior because you want the nullGetter to handle the tag in this case, you could use the following parser:

```
function parser(tag) {
    return {
        get(scope, context) {
            if (context.num < context.scopePath.length) {
                return null;
            }
            // You can customize your parser here instead of scope[tag] of course
            return scope[tag];
        },
    };
},
```

The context.num value contains the scope level for this particular evaluation.

When evaluating the {name} tag in the example above, there are two evaluations:

```
// For the first evaluation, when evaluating in the {#users} scope
context.num = 1;
context.scopePath = ["users"];
// This evaluation returns null because the
// first product doesn't have a name property

// For the second evaluation, when evaluating in the root scope
context.num = 0;
context.scopePath = ["users"];
// This evaluation returns null because of the extra added condition
```

Note that you could even make this behavior dependent on a given prefix, for example, if you want to by default, use the mechanism of scope traversal, but for some tags, allow only to evaluate on the deepest scope, you could add the following condition:

```
function parser(tag) {
    return {
        get(scope, context) {
            const onlyDeepestScope = tag[0] === '!';
            if (onlyDeepestScope) {
                if (context.num < context.scopePath.length) {
                    return null;
                }
                else {
                    // Remove the leading "!", ie: "!name" => "name"
                    tag = tag.substr(1);
                }
            }
            // You can customize the rest of your parser here instead of
            // scope[tag], by using the angular-parser for example.
            return scope[tag];
        },
    };
},
```

5.1.7 Parser example to always use the root scope

Let's say that at the root of your data, you have some property called "company".

You need to access it within a loop, but the company is also part of the element that is looped upon.

With following data:

```
doc.render({
    company: 'ACME Company',
    contractors: [
        { company: "The other Company" },
        { company: "Foobar Company" },
    ]
});
```

If you want to access the company at the root level, it is not possible with the default parser.

You could implement it this way, when writing *{$company}*:

```
const options = {
    parser: function(tag) {
        return {
            get(scope, context) {
                const onlyRootScope = tag[0] === '$';
                if (onlyRootScope) {
                    if (context.num !== 0) {
                        return null;
                    }
                    else {
                        // Remove the leading "$", ie: "$company" => "company"
                        tag = tag.substr(1);
                    }
                }
                // You can customize the rest of your parser here instead of
                // scope[tag], by using the angular-parser for example.
                return scope[tag];
            },
        };
    },
};
const doc = new Docxtemplater(zip, options);
```

5.2 Custom delimiters

You can set up your custom delimiters:

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

5.3 paragraphLoop

The paragraphLoop option has been added in version 3.2.0. Since it breaks backwards-compatibility, it is turned off by default.

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

```
new Docxtemplater(zip, {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 (and there is no other content on those 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 open and loop close 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, if your data contains newlines, those will be shown as linebreaks in the document

```
const doc = new Docxtemplater(zip, {linebreaks:true});
doc.render({text: "My text,\nmultiline"});
```

CHAPTER 6

Angular parser

6.1 Introduction

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.

6.2 Setup

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

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

(continues on next page)

(continued from previous page)

```
function angularParser(tag) {
    tag = tag.replace(/^\.$/, "this").replace(/(’|‘)/g, "'").replace(/(“|”)/g, '"')
    const expr = expressions.compile(tag);
    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 = assign(obj, scopeList[i]);
            }
            return expr(scope, obj);
        }
    };
}
new Docxtemplater(zip, {parser:angularParser});
```

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

6.3 Conditions

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

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

{#userName == "John"}
Hello John, welcome back
{/}
```

The first conditional will render the section only if there are 2 users or more.

The second conditional will render the section only if the userName is the string “John”.

It also handles the boolean operators AND `&&`, OR `||`, +, −, the ternary operator `a ? b : c`, operator precendence 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
{/}
{#cond4 ? users : usersWithAdminRights}
Paragraph 3
{/}
```

(continues on next page)

(continued from previous page)

```
There are {users.length} users.
{/generalCondition}
```

6.4 Filters

With filters, it is possible to write the following template to have the resulting string be uppercased:

```
{user.name | upper}
```

```
var expressions = require('angular-expressions');
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();
}
```

More complex filters are possible, for example, if you would like to list the names of all active users. If your data is the following:

```
{
    users: [
        {
            name: "John",
            age: 15,
        },
        {
            name: "Mary",
            age: 26,
        }
    ],
}
```

You could show the list of users that are older than 18, by writing the following code:

```
var expressions = require('angular-expressions');
expressions.filters.olderThan = function(users, minAge) {
    // 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(!users) return users;
    return users.filter(function(user) {
        return user.age >= minAge;
    });
}
```

And in your template,

```
The allowed users are:

{#users | olderThan:15}
{name} - {age} years old
{/}
```

There are some interesting use cases for filters

6.4.1 Data filtering

You can write some generic data filters using angular expressions inside the filter itself.

```
{
    users: [
        {
            name: "John",
            age: 10,
        },
        {
            name: "Mary",
            age: 20,
        }
    ]
}
```

```
{#users | where:'age > 15'}
Hello {name}
{/}
```

The argument inside the where filter can be any other angular expression, with ||, &&, etc

The code for this filter is extremely terse, and gives a lot of possibilities:

```
const expressions = require("angular-expressions");
expressions.filters.where = function (input, query) {
    return input.filter(function (item) {
        return expressions.compile(query)(item);
    })
}
```

6.4.2 Data sorting

If your data is the following:

```
{
    "items": [
        {
            "name": "Acme Computer",
            "price": 1000,
        },
        {
            "name": "USB storage",
            "price": 15,
        },
        {
            "name": "Mouse & Keyboard",
            "price": 150,
        }
    ],
}
```

You might want to sort the items by price (ascending).

You could do that again with a filter, like this:

```
{#items | sortBy:'price'}
{name} for a price of {price} €
{/}
```

The code for this filter is:

```
const { sortBy } = require("lodash");
expressions.filters.sortBy = function(input, ...fields) {
    // In our example field is the string "price"
    // 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 sortBy(input, fields);
}
```

6.4.3 Data aggregation

If your data is the following:

```
{
    "items": [
        {
            "name": "Acme Computer",
            "price": 1000,
        },
        {
            "name": "Mouse & Keyboard",
            "price": 150,
        }
    ],
}
```

And you would like to show the total price, you can write in your template:

```
{#items}
{name} for a price of {price} €
{/}
Total Price of your purchase: {items | sumby:'price'}€
```

The *sumby* is a filter that you can write like this:

```
expressions.filters.sumby = function(input, field) {
    // In our example field is the string "price"
    // 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.reduce(function(sum, object) {
        return sum + object[field];
    }, 0);
}
```

6.4.4 Data formatting

This example is to format numbers in the format: "150.00" (2 digits of precision) If your data is the following:

```
{
    "items": [
        {
            "name": "Acme Computer",
            "price": 1000,
        },
        {
            "name": "Mouse & Keyboard",
            "price": 150,
        }
    ],
}
```

And you would like to show the price with two digits of precision, you can write in your template:

```
{#items}
{name} for a price of {price | toFixed:2} €
{/}
```

The *toFixed* is an angular filter that you can write like this:

```
expressions.filters.toFixed = function(input, precision) {
    // In our example precision is the integer 2
    // 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.toFixed(precision);
}
```

6.5 Assignments

With the angular expression option, it is possible to assign a value to a variable directly from your template.

For example, in your template, write:

```
{full_name = first_name + last_name}
```

The problem with this expression is that it will return the value of full_name. There are two ways to fix this issue, either, if you still would like to keep this as the default behavior, add *; ''* after your expression, for example

```
{full_name = first_name + last_name; ''}
```

This will first execute the expression, and then execute the second statement which is an empty string, and return it.

An other approach is to automatically silence the return values of expression containing variable assignments.

You can do so by using the following parser option:

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

(continues on next page)

(continued from previous page)

```
function angularParser(tag) {
    tag = tag.replace(/^\.$/, "this").replace(/(’|‘)/g, "'").replace(/(“|”)/g, '"')
    const expr = expressions.compile(tag);
    // isAngularAssignment will be true if your tag contains a `=`, for example
    // when you write the following in your template:
    // {full_name = first_name + last_name}
    // In that case, it makes sense to return an empty string so
    // that the tag does not write something to the generated document.
    const isAngularAssignment =
        expr.ast.body[0] &&
        expr.ast.body[0].expression.type === "AssignmentExpression";

    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 = assign(obj, scopeList[i]);
            }
            const result = expr(scope, obj);
            if (isAngularAssignment) {
                return "";
            }
            return result;
        },
    };
}
new Docxtemplater(zip, {parser:angularParser});
```

Note that if you use a standard tag, like *{full_name = first_name + last_name}* and if you put no other content on that paragraph, the line will still be there but it will be an empty line. If you wish to remove the line, you could use a rawXML tag which will remove the paragraph, like this:

```
{@full_name = first_name + last_name}
{@vat = price * 0.2}
{@total_price = price + vat}
```

This way, all these assignment lines will be dropped.

6.6 Retrieving $index as part of an expression

One might need to have a condition on the $index when inside a loop.

For example, if you have two arrays of the same length and you want to loop over both of them at the same time:

```
{
    "names": [ "John", "Mary" ],
    "ages": [ 15, 26 ],
}
```

```
{#names}
{#$index == 0}First item !{/}
{names[$index]}
```

(continues on next page)

(continued from previous page)

```
{ages[$index]}
{/names}
```

To do this, you can use the following parser:

```
var expressions = require('angular-expressions');
var assign = require("lodash/assign");
var last = require("lodash/last");
function angularParser(tag) {
    tag = tag.replace(/^\.$/, "this").replace(/(’|‘)/g, "'").replace(/(“|”)/g, '"')
    const expr = expressions.compile(tag);
    return {
        get: function(scope, context) {
            let obj = {};
            const index = last(context.scopePathItem);
            const scopeList = context.scopeList;
            const num = context.num;
            for (let i = 0, len = num + 1; i < len; i++) {
                obj = assign(obj, scopeList[i]);
            }
            obj = assign(obj, {"$index": index});
            return expr(scope, obj);
        }
    };
}
new Docxtemplater(zip, {parser:angularParser});
```

CHAPTER 7

Asynchronous Data Resolving

You can have promises in your data. Note that the only step running asynchronously is the resolving of your data. The compilation (parsing of your template to parse position of each tags), and the rendering (using the compiled version + the resolved data) will still be fully synchronous

```
// The error object contains additional information when logged with JSON.stringify␣
↪(it contains a properties object containing all suberrors).
function replaceErrors(key, value) {
    if (value instanceof Error) {
        return Object.getOwnPropertyNames(value).reduce(function(error, key) {
            error[key] = value[key];
            return error;
        }, {});
    }
    return value;
}

function errorHandler(error) {
    console.log(JSON.stringify({error: error}, replaceErrors));

    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 doc;
try {
    // Compile your document
    doc = new Docxtemplater(zip, options);
```

(continues on next page)

(continued from previous page)

```
}
catch (error) {
    // Catch compilation errors (errors caused by the compilation of the template:␣
↪misplaced tags)
    errorHandler(error);
}

doc.renderAsync({user: new Promise(resolve) { setTimeout(()=> resolve('John'), 1000)}}
↪)
   .catch((err) => errorHandler(err));
   .then(function() {
       var buf = doc.getZip()
           .generate({type: 'nodebuffer'});
       fs.writeFileSync(path.resolve(__dirname, 'output.docx'), buf);
   });
```

CHAPTER 8

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

8.1 Dependencies

1. PizZip to zip and unzip the docx files
2. xmldom to parse the files as xml

CHAPTER 9

Error handling

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 occurences 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 containing all suberrors).
    function replaceErrors(key, value) {
        if (value instanceof Error) {
            return Object.getOwnPropertyNames(value).reduce(function(error, key) {
                error[key] = value[key];
                return error;
            }, {});
        }
        return value;
    }
    console.log(JSON.stringify({error: error}, replaceErrors));

    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 follow this 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:

9.3.1 multi_error

This error means that multiple errors where found in the template (1 or more). See below for handling these errors.

9.3.2 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 !
```

9.3.3 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.
```

9.3.4 utf8_decode

This is an internal error, please report it if you see it

9.3.5 xmltemplater_content_must_be_string

This is an internal error that happens if you try to template something that is not a string (a number for example)

9.3.6 raw_xml_tag_should_be_only_text_in_paragraph

This 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.

9.3.7 unclosed_loop (and unopened_loop)

This happens when a loop is closed but never opened: for example

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

or

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

9.3.8 closing_tag_does_not_match_opening_tag

This happens when a loop is closed but doesn't match the opening tag, for example:

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

9.3.9 scopeparser_compilation_failed

This happens when your parser throws an error during compilation. The parser is the second argument of the constructor `new Docxtemplater(zip, {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*

9.3.10 unimplemented_tag_type

This happens when a tag type is not implemented. It should normally not happen, unless you changed docxtemplater code.

9.3.11 malformed_xml

This happens when an xml file of the document cannot be parsed correctly.

9.3.12 loop_position_invalid

This happens when a loop would produce invalid XML.

For example, if you write:

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

{/users}
```

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

9.4 Cannot attach a module that was already attached

You might get this error:

Cannot attach a module that was already attached: "ImageModule". Maybe you are instantiating the module at the root level, and using it for multiple instances of Docxtemplater

In previous versions the error was *Cannot attach a module that was already attached*

This happens if you are reusing the same module instance twice.

It usually means that you are calling *new ImageModule()* just once, but you should call it for each instance of docxtemplater.

The following code will throw the error when calling "generate" twice:

```
var Docxtemplater = require("docxtemplater");
var ImageModule = require("docxtemplater-image-module");
var imageModule = new ImageModule(opts);

function generate(content) {
    var zip = new PizZip(content);
    var doc = new Docxtemplater(zip, {modules: [imageModule]});
    doc.render(data)
}
```

You should always reconstruct an imageModule for each Docxtemplater instance.

The following code will no more throw the error:

```
var Docxtemplater = require("docxtemplater");
var ImageModule = require("docxtemplater-image-module");

function generate(content) {
    var zip = new PizZip(content);
    var imageModule = new ImageModule(opts);
    var doc = new Docxtemplater(zip, { paragraphLoop: true, linebreaks: true,␣
↪modules: [imageModule] });
    doc.render(data)
}
```

9.5 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
```

CHAPTER 11

API

11.1 Constructor

```
new Docxtemplater()

    This function returns a new Docxtemplater instance
```

```
new Docxtemplater(zip[, options])

    This constructor is preferred over the constructor without any arguments.
    The constructor without arguments will be removed in docxtemplater version 4.
    When calling the constructor with a zip file as the first argument,
    the document will be compiled during instantiation, meaning that
    this will throw an error if some tag is misplaced in your document.

    The options parameter allows you to attach some modules, and they will be␣
↪attached before compilation.

        zip:
            a zip instance, coming from pizzip or jszip version 2.

        options: (default {modules:[]})
            You can use this object to configure docxtemplater.
            It is possible to configure in the following ways:

                * You can pass options to change custom parser, custom delimiters,␣
↪etc.
                * You can pass the list of modules that you would like to attach.

            For example:
            const options = {
                modules: [ new ImageModule(imageOpts) ],
                delimiters: {
```

(continues on next page)

(continued from previous page)

```
                    start: "<",
                    end: ">",
                },
            }
            const doc = new Docxtemplater(zip, options);

    This function returns a new Docxtemplater instance
```

11.2 Methods

```
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 should 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)

compile()

    This function is **deprecated** and you should instead use
    the new constructor with two arguments.

    This function parses the template to prepare for the rendering.
    If your template has some issues in the syntax
    (for example if your tag is never closed like in : `Hello {user`),
    this function will throw an error with extra properties describing the error.
    This function is called for you in render() if you didn't call it yourself.
    This function should be called before doing resolveData() if you have some async␣
↪data.

loadZip(zip)

    This function is **deprecated** and you should instead use
    the new constructor with two arguments.

    You have to pass a zip instance to that method, coming from pizzip or jszip␣
↪version 2

setOptions()
```

(continues on next page)

(continued from previous page)

```
    This function is **deprecated** and you should instead use
    the new constructor with two arguments.

    This function is used to configure the docxtemplater instance by
    changing the parser, delimiters, etc.
    (See https://docxtemplater.readthedocs.io/en/latest/configuration.html).

attachModule(module)

    This function is **deprecated** and you should instead use
    the new constructor with two arguments.

    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)
```

CHAPTER 12

Frequently asked questions

12.1 Inserting new lines

```
const doc = new Docxtemplater(zip, { 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:

```
doc.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)

```
const zip = doc.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.

12.5 Using boolean operators (AND, OR) and comparison operators (<, >)

You can also have conditions with comparison operators (< and >), or boolean operators (&& and ||) using angular parser conditions.

12.6 Conditional Formatting

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

12.7 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

12.8 Keep placeholders that don't have data

It is possible to define which value to show when a tag resolves to undefined or null (for example when no data is present for that value).

For example, with the following template

```
Hello {name}, your hobby is {hobby}.
```

```
{
    "hobby": "football",
}
```

The default behavior is to return "undefined" for empty values.

```
Hello undefined, your hobby is football.
```

You can customize this to either return another string, or return the name of the tag itself, so that it will show:

```
Hello {name}, your hobby is football.
```

It is possible to customize the value that will be shown for {name} by using the nullGetter option. In the following case, it will return "{name}", hence it will keep the placeholder {name} if the value does not exist.

```
function nullGetter(part, scopeManager) {
    /*
        If the template is {#users}{name}{/} and a value is undefined on the
        name property:
```

(continues on next page)

(continued from previous page)

```
        - part.value will be the string "name"
        - scopeManager.scopePath will be ["users"] (for nested loops, you would have␣
↪multiple values in this array, for example one could have ["companies", "users"])
        - scopeManager.scopePathItem will be equal to the array [2] if
          this happens for the third user in the array.
        - part.module would be empty in this case, but it could be "loop",
          "rawxml", or or any other module name that you use.
    */

    if (!part.module) {
        // part.value contains the content of the tag, eg "name" in our example
        // By returning '{' and part.value and '}', it will actually do no␣
↪replacement in reality. You could also return the empty string if you prefered.
        return '{' + part.value + '}';
    }
    if (part.module === "rawxml") {
        return "";
    }
    return "";
}
const doc = new Docxtemplater(zip, {nullGetter: nullGetter});
```

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

12.10 Support for IE9 and lower

docxtemplater should work on almost all browsers: 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

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
  }

  const zip = new PizZip(data);
});
```

12.11 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:

```
const InspectModule = require("docxtemplater/js/inspect-module");
const iModule = InspectModule();
const doc = new Docxtemplater(zip, { modules: [iModule] });
const tags = iModule.getAllTags();
console.log(tags);
// After getting the tags, you can render the document like this:
// doc.render(data);
```

With the following template:

```
{company}

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

It will log this object:

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

You can also get a more detailled 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

12.12 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).

12.13 Pptx support

Docxtemplater handles pptx files without any special configuration (since version 3.0.4).

It does so by looking at the content of the "[Content_Types].xml" file and by looking at some docx/pptx specific content types.

12.14 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 before instantiating the Docxtemplater instance.

```
const fixDocPrCorruptionModule = {
    set(options) {
        if (options.Lexer) {
            this.Lexer = options.Lexer;
        }
        if (options.zip) {
            this.zip = options.zip;
        }
    },
    on(event) {
        if (eventName === "attached") {
            this.attached = false;
        }
        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, `$1${attrValue}$3`);
            }
            let end = partValue.lastIndexOf("/>");
            if (end === -1) {
                end = partValue.lastIndexOf(">");
            }
            return (
                partValue.substr(0, end) +
                ` ${attr}="${attrValue}"` +
                partValue.substr(end)
            );
        }
```

(continues on next page)

(continued from previous page)

```
        zip.file(/\.xml$/).forEach(function (f) {
            let text = f.asText();
            const xmllexed = Lexer.xmlparse(text, {
                text: [],
                other: ["wp:docPr"],
            });
            if (xmllexed.length > 1) {
                text = xmllexed.reduce(function (fullText, part) {
                    if (
                        part.tag === "wp:docPr" &&
                        ["start", "selfclosing"].indexOf(part.position) !== -1
                    ) {
                        return (
                            fullText +
                            setSingleAttribute(part.value, "id", prId++)
                        );
                    }
                    return fullText + part.value;
                }, "");
            }
            zip.file(f.name, text);
        });
    },
};
const doc = new Docxtemplater(zip, { modules: [fixDocPrCorruptionModule] });
```

12.15 Attaching modules for extra functionality

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

```
const doc = new Docxtemplater(zip, { paragraphLoop: true, linebreaks: true, modules:␣
↪[...] });
doc.setData(data);
```

12.16 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.render({
   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 condtion *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}
{/}
```

12.17 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
    }
  ]
}
```

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

```
// Please make sure to use angular-expressions 1.1.2 or later
// More detail at https://github.com/open-xml-templating/docxtemplater/issues/589
const expressions = require("angular-expressions");
const assign = require("lodash/assign");
function angularParser(tag) {
   tag = tag.replace(/^\.$/, "this").replace(/(’|‘)/g, "'").replace(/(“|”)/g, '"')
   const expr = expressions.compile(tag);
   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 = assign(obj, scopeList[i]);
         }
         return expr(scope, obj);
      },
   };
}

const doc = new Docxtemplater(zip, {parser: angularParser});
```

12.18 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.

12.19 Docxtemplater in a React project

There is an online react demo available on stackblitz.

You can use the following code:

```
import React, { Component } from 'react';
import React from 'react';
import Docxtemplater from 'docxtemplater';
import PizZip from 'pizzip';
import PizZipUtils from 'pizzip/utils/index.js';
import { saveAs } from 'file-saver';

function loadFile(url, callback) {
  PizZipUtils.getBinaryContent(url, callback);
}

export const App = class App extends React.Component {
  render() {
    const generateDocument = () => {
      loadFile('https://docxtemplater.com/tag-example.docx', function(
        error,
        content
      ) {
        if (error) {
          throw error;
        }
        const zip = new PizZip(content);
        const doc = new Docxtemplater(zip, {
          paragraphLoop: true,
          linebreaks: true
        });
        try {
          // render the document (replace all occurences of {first_name} by John,
↪{last_name} by Doe, ...)
          doc.render({
            first_name: 'John',
            last_name: 'Doe',
            phone: '0652455478',
            description: 'New Website'
          });
        } catch (error) {
          // The error thrown here contains additional information when logged with␣
↪JSON.stringify (it contains a properties object containing all suberrors).
          function replaceErrors(key, value) {
            if (value instanceof Error) {
              return Object.getOwnPropertyNames(value).reduce(function(
                error,
                key
              ) {
                error[key] = value[key];
                return error;
              },
```

(continues on next page)

(continued from previous page)

```
            {});
          }
          return value;
        }
        console.log(JSON.stringify({ error: error }, replaceErrors));

        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;
      }
      const out = doc.getZip().generate({
        type: 'blob',
        mimeType:
          'application/vnd.openxmlformats-officedocument.wordprocessingml.document'
      }); //Output the document using Data-URI
      saveAs(out, 'output.docx');
    });
  };

  return (
    <div className="p-2">
      <button onClick={generateDocument}>Generate document</button>
    </div>
  );
 }
};
```

12.20 Docxtemplater in an angular project

There is an online angular demo available on stackblitz.

If you are using an angular version that supports the *import* keyword, you can use the following code:

```
import { Component } from "@angular/core";
import Docxtemplater from "docxtemplater";
import PizZip from "pizzip";
import PizZipUtils from "pizzip/utils/index.js";
import { saveAs } from "file-saver";

function loadFile(url, callback) {
  PizZipUtils.getBinaryContent(url, callback);
}

@Component({
  selector: "app-product-list",
  templateUrl: "./product-list.component.html",
```

(continues on next page)

(continued from previous page)

```
  styleUrls: ["./product-list.component.css"]
})
export class ProductListComponent {
  generate() {
    loadFile("https://docxtemplater.com/tag-example.docx", function(
      error,
      content
    ) {
      if (error) {
        throw error;
      }
      const zip = new PizZip(content);
      const doc = new Docxtemplater(zip, { paragraphLoop: true, linebreaks: true });
      try {
        // render the document (replace all occurences of {first_name} by John, {last_
↪name} by Doe, ...)
        doc.render({
          first_name: "John",
          last_name: "Doe",
          phone: "0652455478",
          description: "New Website"
        });
      } catch (error) {
        // The error thrown here contains additional information when logged with␣
↪JSON.stringify (it contains a properties object containing all suberrors).
        function replaceErrors(key, value) {
          if (value instanceof Error) {
            return Object.getOwnPropertyNames(value).reduce(function(
              error,
              key
            ) {
              error[key] = value[key];
              return error;
            },
            {});
          }
          return value;
        }
        console.log(JSON.stringify({ error: error }, replaceErrors));

        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;
      }
      const out = doc.getZip().generate({
        type: "blob",
        mimeType:
          "application/vnd.openxmlformats-officedocument.wordprocessingml.document"
      });
```

(continues on next page)

(continued from previous page)

```
      // Output the document using Data-URI
      saveAs(out, "output.docx");
    });
  }
}
```

12.21 Docxtemplater in a Vuejs project

There is an online vuejs demo available on stackblitz.

If you are using vuejs 2.0 version that supports the *import* keyword, you can use the following code:

```
import Docxtemplater from "docxtemplater";
import PizZip from "pizzip";
import PizZipUtils from "pizzip/utils/index.js";
import { saveAs } from "file-saver";

function loadFile(url, callback) {
  PizZipUtils.getBinaryContent(url, callback);
}

export default {
  methods: {
    renderDoc() {
      loadFile("https://docxtemplater.com/tag-example.docx", function(
        error,
        content
      ) {
        if (error) {
          throw error;
        }
        const zip = new PizZip(content);
        const doc = new Docxtemplater(zip, { paragraphLoop: true, linebreaks: true });
        try {
          // render the document (replace all occurences of {first_name} by John,
↪{last_name} by Doe, ...)
          doc.render({
            first_name: "John",
            last_name: "Doe",
            phone: "0652455478",
            description: "New Website"
          });
        } catch (error) {
          // The error thrown here contains additional information when logged with␣
↪JSON.stringify (it contains a properties object containing all suberrors).
          function replaceErrors(key, value) {
            if (value instanceof Error) {
              return Object.getOwnPropertyNames(value).reduce(function(
                error,
                key
              ) {
                error[key] = value[key];
                return error;
              },
```

(continues on next page)

(continued from previous page)

```
            {});
          }
          return value;
        }
        console.log(JSON.stringify({ error: error }, replaceErrors));

        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;
      }
      const out = doc.getZip().generate({
        type: "blob",
        mimeType:
          "application/vnd.openxmlformats-officedocument.wordprocessingml.document"
      });
      // Output the document using Data-URI
      saveAs(out, "output.docx");
    });
  }
},

template: `
  <button @click="renderDoc">
     Render docx template
  </button>
`
};
```

12.22 Docxtemplater in a Next.js project

There is an online nextjs demo available on codesandbox.

You can use the following code:

```
import SiteLayout from "../components/SiteLayout";
import React from "react";
import Docxtemplater from "docxtemplater";
import PizZip from "pizzip";
import { saveAs } from "file-saver";
let PizZipUtils = null;
if (typeof window !== "undefined") {
  import("pizzip/utils/index.js").then(function (r) {
    PizZipUtils = r;
  });
}
```

(continues on next page)

(continued from previous page)

```
function loadFile(url, callback) {
  PizZipUtils.getBinaryContent(url, callback);
}

const generateDocument = () => {
  loadFile("https://docxtemplater.com/tag-example.docx", function (
    error,
    content
  ) {
    if (error) {
      throw error;
    }
    const zip = new PizZip(content);
    const doc = new Docxtemplater().loadZip(zip);
    try {
      // render the document (replace all occurences of {first_name} by John, {last_
↪name} by Doe, ...)
      doc.render({
        first_name: "John",
        last_name: "Doe",
        phone: "0652455478",
        description: "New Website"
      });
    } catch (error) {
      // The error thrown here contains additional information when logged with JSON.
↪stringify (it contains a properties object containing all suberrors).
      function replaceErrors(key, value) {
        if (value instanceof Error) {
          return Object.getOwnPropertyNames(value).reduce(function (
            error,
            key
          ) {
            error[key] = value[key];
            return error;
          },
          {});
        }
        return value;
      }
      console.log(JSON.stringify({ error: error }, replaceErrors));

      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;
    }
    const out = doc.getZip().generate({
      type: "blob",
      mimeType:
        "application/vnd.openxmlformats-officedocument.wordprocessingml.document"
```

(continues on next page)

(continued from previous page)

```
    });
    // Output the document using Data-URI
    saveAs(out, "output.docx");
  });
};

const Index = () => (
  <SiteLayout>
    <div className="mt-8 max-w-xl mx-auto px-8">
      <h1 className="text-center">
        <span className="block text-xl text-gray-600 leading-tight">
          Welcome to this
        </span>
        <span className="block text-5xl font-bold leading-none">
          Awesome Website
        </span>
      </h1>
      <div className="mt-12 text-center">
        <button
          onClick={generateDocument}
          className="inline-block bg-gray-900 hover:bg-gray-800 text-white font-
↪medium rounded-lg px-6 py-4 leading-tight"
        >
          Generate document
        </button>
      </div>
    </div>
  </SiteLayout>
);

export default Index;
```

12.23 Getting access to page number / total number of pages or regenerate Table of Contents

Sometimes, you would like to know what are the total number of pages in the document, or what is the page number at the current tag position.

This is something that will never be achievable with docxtemplater, because docxtemplater is only a templating engine: it does know how to parse the docx format. However, it has no idea on how the docx is rendered at the end: the width, height of each paragraph determines the number of pages in a document.

Since docxtemplater does not know how to render a docx document, (which determines the page numbers), this is why it is impossible to regenerate the page numbers within docxtemplater.

Also, even across different "official" rendering engines, the page numbers may vary. Depending on whether you open a document with Office Online, Word 2013 or Word 2016 or the Mac versions of Word, you can have some slight differences that will, at the end, influence the number of pages or the position of some elements within a page.

The amount of work to write a good rendering engine would be very huge (a few years at least for a team of 5-10 people).

12.24 Special character keys with angular parser throws error

The error that you could see is this, when using the tag *{être}*.

```
Error: [$parse:lexerr] Lexer Error: Unexpected next character  at columns 0-0 [ê] in␣
↪expression [être].
```

This is because angular-expressions does not allow non-ascii characters. You will need angular-expressions version 1.1.0 which adds the *isIdentifierStart* and *isIdentifierContinue* properties.

You can fix this issue by adding the characters that you would like to support, for example:

```
function validChars(ch) {
    return (
        (ch >= "a" && ch <= "z") ||
        (ch >= "A" && ch <= "Z") ||
        ch === "_" ||
        ch === "$" ||
        "ÀÈÌÒÙàèìòùÁÉÍÓÚáéíóúÂÊÎÔÛâêîôûÃÑÕãñõÄËÏÖÜŸäëïöüÿß".indexOf(ch) !== -1
    );
}
function angularParser(tag) {
    tag = tag.replace(/^\.$/, "this").replace(/(’|‘)/g, "'").replace(/(“|”)/g, '"')
    const expr = expressions.compile(tag,
        {
            isIdentifierStart: validChars,
            isIdentifierContinue: validChars
        }
    );
    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 = assign(obj, scopeList[i]);
            }
            return expr(scope, obj);
        }
    };
}
new Docxtemplater(zip, {parser:angularParser});
```

12.25 Remove proofstate tag

The proofstate tag in a document marks the document as spell-checked when last saved. After rendering a document with docxtemplater, some spelling errors might have been introduced by the addition of text. The proofstate tag is by default, not removed.

To remove it, one could do the following, starting with docxtemplater 3.17.2

```
const proofstateModule = require("docxtemplater/js/proof-state-module.js");
doc = new Docxtemplater(zip, {modules: [proofstateModule] });
```

12.26 Adding page break except for last item in loop

It is possible, in a condition, to have some specific behavior for the last item in the loop using a custom parser. You can read more about how custom parsers work here.

It will allow you to add a page break at the end of each loop, except for the last item in the loop.

The template will look like this:

```
{#users}
The user {name} is aged {age}
{description}
Some other content
{@$pageBreakExceptLast}
{/}
```

And each user block will be followed by a pagebreak, except the last user.

```
function angularParser(tag) {
    tag = tag.replace(/^\.$/, "this").replace(/(’|‘)/g, "'").replace(/(“|”)/g, '"')
    const expr = expressions.compile(tag);
    return {
        get: function(scope, context) {
            let obj = {};
            const scopeList = context.scopeList;
            const num = context.num;
            for (let i = 0, len = num + 1; i < len; i++) {
                obj = assign(obj, scopeList[i]);
            }
            return expr(scope, obj);
        }
    };
}
function parser(tag) {
    // We write an exception to handle the tag "$pageBreakExceptLast"
    if (tag === "$pageBreakExceptLast") {
        return {
            get(scope, context) {
                const totalLength = context.scopePathLength[context.scopePathLength.
↪length - 1];
                const index = context.scopePathItem[context.scopePathItem.length - 1];
                const isLast = index === totalLength - 1;
                if (!isLast) {
                    return '<w:p><w:r><w:br w:type="page"/></w:r></w:p>';
                }
                else {
                    return '';
                }
            }
        }
    }
    // We use the angularParser as the default fallback
    // If you don't wish to use the angularParser,
    // you can use the default parser as documented here:
    // https://docxtemplater.readthedocs.io/en/latest/configuration.html#default-
↪parser
    return angularParser(tag);
}
```

(continues on next page)

(continued from previous page)

```
const doc = new Docxtemplater(zip, {parser: parser});
doc.render();
```

12.27 Encrypting files

Docxtemplater itself does not handle the Encryption of the docx files.

There seem to be two solutions for this:

- Use a Python tool that does exactly this, it is available here: https://github.com/nolze/msoffcrypto-tool
- The xlsx-populate library also implements the Encryption/Decryption (algorithms are inspired by msoffcrypto-tool), however, the code probably needs to be a bit changed to work with docxtemplater: https://github.com/dtjohnson/xlsx-populate/blob/7480a02575c9140c0e7995623ea192c88c1886d3/lib/Encryptor.js#L236

12.28 Assignment expression in template

By using the angular expressions options, it is possible to add assignment expressions (for example *{full_name = first_name + last_name}* in your template. See following part of the doc.

12.29 Changing the end-user syntax

If you find that the loop syntax is a bit too complex, you can change it to something more human friendly (but more verbose). This could be used to have a syntax more similar to what the software "HotDocs" provides.

For example, you could be willing to write loops like this :

```
{FOR users}
Hello {name}
{ENDFOR}
```

Instead of

```
{#users}
Hello {name}
{/}
```

This can be done by changing the prefix of the loop module, which is a builtin module.

```
const doc = new Docxtemplater(zip);
doc.modules.forEach(function (module) {
    if (module.name === "LoopModule") {
        module.prefix.start = "FOR "
        module.prefix.start = "ENDFOR "
    }
});
```

Note that if you don't like the default delimiters which are *{* and *}*, you can also change them, for example :

If you prefer to write :

```
[[FOR users]]
Hello [[name]]
[[ENDFOR]]
```

You could write your code like this :

```
const doc = new Docxtemplater(zip, { delimiters: { start: "[[", end: "]]" } });
doc.modules.forEach(function (module) {
    if (module.name === "LoopModule") {
        module.prefix.start = "FOR "
        module.prefix.start = "ENDFOR "
    }
});
```

Note that it is however not possible to use no delimiters at all, docxtemplater forces you to have some delimiters.

Similarly, for each paid module (image module, ...), you can set your own prefixes as well.

For example, for the image module, if you would like to write {IMG mydata} instead of {%mydata} and {CENTERIMG mydata} instead of {%%mydata}, you can write your code like this :

```
const ImageModule = require("docxtemplater-image-module");

const opts = {};
opts.centered = false;
opts.getImage = function (tagValue, tagName) {
  return fs.readFileSync(tagValue);
};

opts.getSize = function (img, tagValue, tagName) {
  return [150, 150];
};

const imageModule = new ImageModule(opts);
imageModule.prefix.normal = "IMG "
imageModule.prefix.centered = "CENTERIMG "
const doc = new Docxtemplater(zip, { modules: [imageModule], delimiters: { start: "[[
↪", end: "]]" } });
doc.modules.forEach(function (module) {
    if (module.name === "LoopModule") {
        module.prefix.start = "FOR "
        module.prefix.start = "ENDFOR "
    }
});
```

CHAPTER 13

Testing

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.render({users: [{msg: "hello", name: "mary"}, {msg: "hello", name: "john"}
↪]});
        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, ie tests that are there to ensure that bugs that occured 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(zip);
    d.render({last_name: russian});
    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"},
        {type: "placeholder", value: "user"},
```

(continues on next page)

(continued from previous page)

```
        {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 regressions will not appear again.

CHAPTER 14

Online Demos

Including:

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

Index