docxtemplater Documentation

Release 0.5.4

Edgar Hipp

October 10, 2014

Contents

1 Goals **3**
1.1 Why you should use a library for this . . . 3

2 Platform Support **5**

3 Dependencies **7**

4 Installation **9**
4.1 Node . . . 9
4.2 Browser . . . 9

5 Syntax **11**
5.1 Synopsis . . . 11
5.2 Tag types . . . 11
5.3 Loop syntax . . . 11
5.4 Dash syntax . . . 12
5.5 Inverted Selections . . . 12

6 Generate a document **15**
6.1 Loading a document: . . . 15
6.2 Setting the tags: . . . 15
6.3 Applying the tags: . . . 16
6.4 Outputing the document: . . . 16

7 Configuration **19**
7.1 Image Replacing . . . 19
7.2 Angular Parser . . . 20
7.3 Intelligent LoopTagging . . . 21

8 Command Line Interface (CLI) **23**
8.1 Config.json Syntax . . . 23

9 Full Documentation per method **25**
9.1 Creating a new Docxgen Object . . . 25
9.2 Docxgen methods . . . 26

10 Copyright **29**

11 Demos **31**

12 Indices and tables 33

Contents:

CHAPTER 1

Goals

Docxtemplater was born out of the idea that you should be able to generate Docx as easily as you generate Html with something like Mustache.

They are a lots of solutions like docx.js, docx4j, ... that generate docx, but you will have to write specific code to create a title, an image, ...

I think this is a waste when you can just write your template with plain old Microsoft Word.

Docxtemplater is just there for that

1.1 Why you should use a library for this

Docx is a zipped format that contains some xml. If you want to build a simple replace {tag} by value system, it can already become complicated, because the {tag} is internally separated into *<w:t>{</w:t><w:t>tag</w:t><w:t>}</w:t>*. If you want to embed loops to iterate over an array, it becomes a real hassle.

CHAPTER 2

Platform Support

docxtemplater works with

- Node.js with 0.10 and 0.11
- Chrome **tested** on version 26
- Firefox 3+ (**tested** on version 21, but should work with 3+)
- Safari **not tested**

Internet explorer is not supported -even IE10- (basically because xhr Requests can't be made on binary files)

You can test if everything works fine on your browser by using the test runner: http://javascript-ninja.fr/docxgenjs/test/SpecRunner.html

Firefox has an other implementation of the xml parser, that's why all tests don't pass now. However, all of the functionality works on Firefox too. The output files are not exactly the same byte wise but the generated XML is correct.

CHAPTER 3

Dependencies

1. **docxgen.js** uses [jszip.js](http://stuk.github.io/jszip/) to zip and unzip the docx files
2. Optionally, if you want to be able to name the output files, you can use **Downloadify.js**, which is required to use method download. Be informed that it uses flash, this is why the method is not recommended. This method is howewer useful because a lot of browsers are limited for the download size with the Data-URI method. **Update**: I will probably implement in the future a way to use the FileSaver API, with [FileSaverJS](http://eligrey.com/demos/FileSaver.js/)
3. Optionnaly, if you want to replace images by images situated at a particular URL, you can use QR codes. For example If you store an image at http://website.com/image.png , you should encode the URL in QR-Code format. ![Qr Code Sample](http://qrfree.kaywa.com/?l=1&s=8&d=http%3A%2F%2Fwebsite.com%2Fimage.png "Qrcode Sample to http://website.com/image.png"). You can even use bracket tags in images. http://website.com/image.png?color={color} will take the *Tags[color]* variable to make a dynamic URL. For this too work, you will need [jsqrcode](http://github.com/edi9999/jsqrcode "jsqrcode repositoty forked") and include the following files, in this order (only for browser support, node support already comes out of the box):

```
<script type="text/javascript" src="grid.js"></script>
<script type="text/javascript" src="version.js"></script>
<script type="text/javascript" src="detector.js"></script>
<script type="text/javascript" src="formatinf.js"></script>
<script type="text/javascript" src="errorlevel.js"></script>
<script type="text/javascript" src="bitmat.js"></script>
<script type="text/javascript" src="datablock.js"></script>
<script type="text/javascript" src="bmparser.js"></script>
<script type="text/javascript" src="datamask.js"></script>
<script type="text/javascript" src="rsdecoder.js"></script>
<script type="text/javascript" src="gf256poly.js"></script>
<script type="text/javascript" src="gf256.js"></script>
<script type="text/javascript" src="decoder.js"></script>
<script type="text/javascript" src="qrcode.js"></script>
<script type="text/javascript" src="findpat.js"></script>
<script type="text/javascript" src="alignpat.js"></script>
<script type="text/javascript" src="databr.js"></script>
```

CHAPTER 4

Installation

4.1 Node

To install docxtemplater, we recommend you to use npm.

```
npm install docxtemplater
```

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

```
npm install docxtemplater -g
```

4.2 Browser

You will just have to include the docxgen.js or docxgen.min.js file.

CHAPTER 5

Syntax

The syntax is highly inspired by Mustache. The template is created in Microsoft Word or any equivalent that saves to docx.

5.1 Synopsis

A typical docxtemplater template:

```
Hello {name} !
```

Given the following hash:

```
{
    name:'Edgar'
}
```

Will produce:

```
Hello Edgar !
```

5.2 Tag types

Like Mustache, it has the loopopening {#} and loopclosing {/} brackets

5.3 Loop syntax

The following template:

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

Given the following hash:

```
{
    "products":
        [
         {name:"Windows",price:100},
```

```
        {name:"Mac OSX",price:200},
        {name:"Ubuntu",price:0}
      ]
}
```

will result in :

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

The loop behaves in the following way:

- If the value is an array, it will loop over all the elements of that array.
- If the value is a boolean, it will loop once if the value is true, keeping the same scope, and not loop at all if the value is false

Note: Because the loops work also with boolean values, you can also use them for conditions.

5.4 Dash syntax

It is quite difficult to know on which element you are going to loop. By default, when using the for loop, docxgen will find that by himself:

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

- they is the Xml Tag <w:tc> -> you are in a table, and it will loop over <w:tr>
- else -> it will loop over <w:t>, which is the default Text Tag

With the Dash syntax you pass as a first argument the tag you want to loop on:

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

In this case this will loop over the first parent <w:p> tag

5.5 Inverted Selections

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.

Template:

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

Hash:

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

Output:

```
No repos :(
```

CHAPTER 6

Generate a document

Here's a sample code to generate a document:

```
DocxGen=require('docxtemplater'); //Only for Node Usage
new DocxGen().loadFromFile("tagExample.docx",{async:true}).success(function(doc){
        doc.setTags({
        "first_name":"Hipp",
                "last_name":"Edgar",
                "phone":"0652455478",
                "description":"New Website"
            }) //set the templateVariables
        doc.applyTags() //apply them (replace all occurences of {first_name} by Hipp, ...)
        doc.output() //Output the document using Data-URI
});
```

The different steps are the following:

6.1 Loading a document:

A docx can be loaded by it's base64 representation, like this:

```
doc=new DocxGen(base64Data,options);
```

However, loading the base64Data is not that easy, so I created a wrapper function to load a docx from a file via Ajax.

Note: The options parameter are explained in *Configuration*

```
doc=new DocxGen().loadFromFile(documentPath,options);
```

The documentPath is the path to the document you want to load. The options are the same as in the constructor function, and I've added the async parameter.

- If async is true, the document is loaded asynchronously and returns a promise (eg you can write .success(callback) to get when the document is loaded)
- If async is false, the document is loaded synchronously and returns the document.

6.2 Setting the tags:

The tags are the variables that are going to be replaced by their values. To set them, just use:

```
doc.setTags(tags);
```

6.3 Applying the tags:

Applying the tags means opening all files that contain text (eg: footer, header, main document), and replace the variables by their values.

```
doc.applyTags(tags)
```

Note: If you specify an argument for the applyTags method, the function setTags(tags) will be called before applying the tags.

6.4 Outputing the document:

They are several ways to output the document. The most basic usage is to download the document.

```
doc.output(options)
```

Depending on your environment, if you don't set any options, this will:

- In the browser: Download the document using DataURI
- In Node: Save the document with the given fileName (output.docx by default)

Here's the different options parameters:

```
name:
    Type:string["output.docx"]
    The name of the file that will be outputed (doesnt work in the browser because of dataUri downloa

callback:
   Type:function
   Function that is called without arguments when the output is done. Is used only in Node (because i

download:
    Type:boolean[true]
    If download is true, file will be downloaded automatically with data URI.
    returns the output file.

type:
    Type:string["base64"]
    The type of zip to return. The possible values are : (same as in http://stuk.github.io/jszip/ @ge
    base64 (default) : the result will be a string, the binary in a base64 form.
    string : the result will be a string in "binary" form, 1 byte per char.
    uint8array : the result will be a Uint8Array containing the zip. This requires a compatible brows
    arraybuffer : the result will be a ArrayBuffer containing the zip. This requires a compatible bro
    blob : the result will be a Blob containing the zip. This requires a compatible browser.
    nodebuffer : the result will be a nodejs Buffer containing the zip. This requires nodejs.
```

This function creates the docx file and downloads it on the user's computer. The name of the file is download.docx for Chrome, and some akward file names for Firefox: VEeTHCfS.docx.part.docx, and can't be changed because it is handled by the browser. For more informations about how to solve this problem, see the **Filename Problems** section on http://stuk.github.io/jszip/

Note: Note: All browsers don't support the download of big files with Data URI, so you **should** use the *download* method for files bigger than 100kB

In Node, to send the document to the client, you can write the following code snippet:

```
out=doc.output({download:false,type:"string"})
res.send(new Buffer(out,"binary"));
```

CHAPTER 7

Configuration

Here are documented the special options that you can set when creating a new DocxGen to get some more superpower :

It documents the options parameter when you do:

```
new DocxGen(data,options);
```

7.1 Image Replacing

The name of this option is *qrCode* (function) (This was a boolean in 0.6.3 and before).

To stay a templating engine, I wanted that DocxTemplater doesn't add an image from scratch, but rather uses an existing image that can be detected, and DocxTemplater will just change the contents of that image, without changing it's style (border, shades, ...). The size of the replaced images will stay the same, ...

So I decided to use the qrCode format, which is a format that lets you identify images by their content.

The option for this is *qrCode* (false for off, a function for on, default off)

The function takes two parameter: The first one is the string, the second the callback.

For example your configuration could be:

```
new DocxGen(data,{qrCode:function(result,callback){
            urloptions=(result.parse(path))
            options =
                    hostname:urloptions.hostname
                    path:urloptions.path
                    method: 'GET'
                    rejectUnauthorized:false
            errorCallback= (e) ->
                    throw new Error("Error on HTTPS Call")
            reqCallback= (res)->
                    res.setEncoding('binary')
                    data = ""
                    res.on('data', (chunk)->
                            data += chunk
                    )
                    res.on('end', ()->
                            callback(null,data))
            req = http.request(options, reqCallback).on('error',errorCallback)
}})
```

Note: If you don't use that functionality, you should disable it, because it is quite slow (the image decoding)

Warning: The qrCode functionality only works for PNG ! They is no support for other file formats yet.

Warning: They is a security warning if you use true as the value for qrCode, because this will use the older qrcode loading function. This function can load any file on the filesystem, with a possible leak in api-keys or whatever you store on docxtemplater's server.

To generate qrcodes with nodejs, you can use for example this script brought by @ssured in issue #69 https://github.com/edi9999/docxtemplater/issues/69

npm install qr-image canvas

To install the dependencies of canvas, look here (platform specific) https://github.com/Automattic/node-canvas/wiki

```
var Canvas, Image, canvas, column, ctx, fs, matrix, qr, qrString, size, textSize, value, x, y, _i, _
qr = require('qr-image');
fs = require('fs');
Canvas = require('canvas');
Image = Canvas.Image;
qrString = 'gen:{image}';
size = 10;
matrix = qr.matrix(qrString);
canvas = new Canvas((2 + 5 + matrix.length) * size, (2 + 5 + matrix.length) * size);
ctx = canvas.getContext('2d');
ctx.fillStyle = '#000';
ctx.fillRect(0, 0, canvas.width, canvas.height);
ctx.fillStyle = '#fff';
ctx.fillRect(1, 1, canvas.width - 2, canvas.height - 2);
ctx.fillStyle = '#000';
for (y = _i = 0, _len = matrix.length; _i < _len; y = ++_i) {
  column = matrix[y];
  for (x = _j = 0, _len1 = column.length; _j < _len1; x = ++_j) {
    value = column[x];
    if (value === 1) {
      ctx.fillRect((x + 1 + 2.5) * size, (y + 1) * size, size, size);
    }
  }
}
ctx.font = 4 * size + 'px Helvetica';
ctx.fillStyle = '#000';
textSize = ctx.measureText(qrString);
ctx.fillText(qrString, (canvas.width - textSize.width) / 2, canvas.height - size - textSize.actualBou
canvas.pngStream().pipe(fs.createWriteStream('qr.png'));
```

7.2 Angular Parser

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

You can set the angular parser with the following code:

With a custom parser you can parse the tags to for example add operators like '+', '-', or whatever the way you want to parse expressions. See for a complete reference of all possibilities http://teropa.info/blog/2014/03/23/angularjs-expressions-cheatsheet.html

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

docxtemplater comes shipped with this parser:

```
parser=function(expression)
{
    return {
        get:function(scope) {
            return scope[expression]
        }
    };
}
```

To use the angular-parser, do the following:

```
expressions= require('angular-expressions');
// define your filter functions here, eg:
// expressions.filters.split = function(input, str) { return input.split(str); }
angularParser= function(tag) {
    return {
        get: tag == '.' ? function(s){ return s;} : expressions.compile(tag)
    };
}
new DocxGen(data,{parser:angularParser})
```

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

7.3 Intelligent LoopTagging

The name of this option *intelligentTagging* (boolean).

When looping over an element, docxtemplater needs to know over which element you want to loop. By default, it tries to do that intelligently (by looking what XML Tags are between the {tags}). However, if you want to always use the <w:t> tag by default, set this option to false.

You can always specify over which element you want to loop with the dash loop syntax

CHAPTER 8

Command Line Interface (CLI)

This section is about the commandline interface of docxtemplater.

The syntax is the following:

    docxtemplater config.json

The full config.json should be something like that:

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

8.1 Config.json Syntax

8.1.1 Config properties:

These are the properties to configure docxtemplater:

```
{
    "config.docxFile":"input.docx", //The input file path
    "config.outputFile":"output.docx", //The output file path
    "config.qrcode":true, //whether the images should be scanned to replace them by qrcodes (slows d
    "config.debug":true //whether to show debug output or not
}
```

8.1.2 Data properties:

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

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

CHAPTER 9

Full Documentation per method

9.1 Creating a new Docxgen Object

```
new DocxGen()

    This function returns a new DocxGen Object

new DocxGen(content[,Tags,options])

    content:
        Type: string
        The docx template you want to load as plain text

    Tags:
        Type: Object {tag_name:tag_replacement} [{}]
        Object containing for each tag_name, the replacement for this tag. For example, if you want t

    options: object

        parser:
            Type: function
            A custom parser to use. See angular.js like parsing

        intelligentTagging:
            Type: boolean [false]
            If intelligent Tagging is not set to true, when using recursive tags ({#tag} and {/tag}),
            If intelligent Tagging is set to true, and when using recursive tags inside tables, the w

        qrCode:
            Type: boolean [false]
            If qrCode is set to true, DocxGen will look at all the images to find a Qr-Code. If the Q
            **Important**: the qrCode functionality only works for PNG, I don't think I will enable t

    localImageCreator
        Type: function(arg,callback) [function that returns an arrow]
        The function has to be customized only if you want to use the qrCode options (**qrCode=true**

        The default localImageCreator returns a red arrow, no matter what the arguments are:

        @localImageCreator= (arg,callback) ->
        result=JSZipBase64.decode("iVBORw0KGgoAAAANSUhEUgAAABcAAAAXCAIAAABvSEP3AAAAAXNSR0IArs4c6QAAAA
```

```
        [Default Image](

    qrFinishedCallBack:
        Type: function () [function that console.log(ready)]
            This function is called if you specify qrCode argument to true in this constructor, and w
```

9.2 Docxgen methods

```
load(content)

    content:
        Type: string
        The docx template you want to load as plain text

loadFromFile(path)
    path
        Type: string
        Loads a docx from a file path
setTags(Tags)

    Tags:
        Type: Object {tag_name:tag_replacement}
        Object containing for each tag_name, the replacement for this tag. For example, if you want t


applyTags([Tags])
    Tags:
        Type: Object {tag_name:tag_replacement}
        same as setTags
        Default:this.Tags

    This function replaces all template variables by their values

output([options])

    options: object[{}]

        name:
            Type:string["output.docx"]
            The name of the file that will be outputed (doesnt work in the browser because of dataUri

        callback:
           Type:function
           Function that is called without arguments when the output is done. Is used only in Node (h

        download:
            Type:boolean[true]
            If download is true, file will be downloaded automatically with data URI.
            returns the output file.

        type:
            Type:string["base64"]
            The type of zip to return. The possible values are : (same as in http://stuk.github.io/js
            base64 (default) : the result will be a string, the binary in a base64 form.
            string : the result will be a string in "binary" form, 1 byte per char.
            uint8array : the result will be a Uint8Array containing the zip. This requires a compatib
```

```
            arraybuffer : the result will be a ArrayBuffer containing the zip. This requires a compat
            blob : the result will be a Blob containing the zip. This requires a compatible browser.
            nodebuffer : the result will be a nodejs Buffer containing the zip. This requires nodejs

    This function creates the docx file and downloads it on the user's computer. The name of the file
    For more informations about how to solve this problem, see the **Filename Problems** section on

    Note: All browsers don't support the download of big files with Data URI, so you **should** use t

download(swfpath,imgpath[,fileName])

    swfpath
        Type:string
        Path to the swfobject.js in downloadify

    imgpath
        Type:string
        Path to the image of the download button

    [fileName]
        Type:string
        Default:"default.docx"
        Name of the file you would like the user to download.

    This requires to include Downloadify.js, that needs flash version 10. Have a look at the *output*

getImageList()

    this gets all images that have one of the following extension: 'gif','jpeg','jpg','emf','png'
    Return format: Array of Object:
    [{path:string,files:ZipFile Object}]

    You should'nt call this method before calling **applyTags()**, because applyTags can modify the i

setImage(path,imgData)

    path
        Type:"String"
        Path of the image, given by getImageList()
    imgData
        Type:"String"
        imgData in txt/plain

    This sets the image given by a path and an imgData in txt/plain.
    You should'nt call this method before calling **applyTags()**, because applyTags can modify the i

getFullText:([path])

    path
        Type:"String"
        Default:"word/document.xml"
        This argument determines from which document you want to get the text. The main document is c

    @returns
        Type:"String"
        The string containing all the text from the document
```

```
    This method gets only the text of a given document (not the formatting)

getTags()

    This function returns the template variables contained in the opened document. For example if the

        {name}
        {first_name}
        {phone}

    The function will return:
        [{
            filename:"document.xml",
            vars:
            {
                name:true,
                first_name:true,
                phone:true
            }
        }]

    If the content contains tagLoops:

        {title}
        {#customer}
        {name}
        {phone}
        {/customer}


    The function will return:

        [{
            filename:"document.xml",
            vars:
            {
                title:true,
                customer:
                {
                    name:true,
                    phone:true
                }
            }
        }]
```

CHAPTER 10

Copyright

License

(The MIT license)

CHAPTER 11

Demos

Including:

- Replace Variables
- Formating
- Angular Parsing
- Loops
- Loops and tables
- Lists
- Replacing images
- Naming the output
- Using QrCodes
- Replacing many images with QrCode
- Raw Xml Insertion

CHAPTER 12

Indices and tables

- *genindex*
- *modindex*
- *search*

C

Command Line Interface (CLI), 21
Configuration, 17
Copyright, 28

F

Full_doc, 23

G

Generate a Document, 13
Goals, 1

I

Installation, 7

P

platform_support, 3

S

Syntax, 9