

docco.js

J. Ashkenas

05-04-2013

Contents

1	Docco	2
	Partners in Crime:	3
	Main Documentation Generation Functions	3
	Configuration	6
	Helpers & Initial Setup	6
	Command Line Interface	7
	Public API	8

Chapter 1

Docco

Docco is a quick-and-dirty documentation generator, written in Literate CoffeeScript¹. It produces an HTML document that displays your comments intermingled with your code. All prose is passed through Markdown², and code is passed through Highlight.js³ syntax highlighting. This page is the result of running Docco against its own source file⁴.

1. Install Docco with **npm**: `sudo npm install -g docco`
2. Run it against your code: `docco src/*.coffee`

There is no “Step 3”. This will generate an HTML page for each of the named source files, with a menu linking to the other pages, saving the whole mess into a `docs` folder (configurable).

The Docco source⁵ is available on GitHub, and is released under the MIT license⁶.

Docco can be used to process code written in any programming language. If it doesn’t handle your favorite yet, feel free to add it to the list⁷. Finally, the “literate” style⁸ of *any* language is also supported — just tack an `.md` extension on the end: `.coffee.md`, `.py.md`, and so on.

¹<http://coffeescript.org/#literate>

²<http://daringfireball.net/projects/markdown/syntax>

³<http://highlightjs.org/>

⁴<https://github.com/jashkenas/docco/blob/master/docco.litcoffee>

⁵<http://github.com/jashkenas/docco>

⁶<http://opensource.org/licenses/MIT>

⁷<https://github.com/jashkenas/docco/blob/master/resources/languages.json>

⁸<http://coffeescript.org/#literate>

Partners in Crime:

- If **Node.js** doesn't run on your platform, or you'd prefer a more convenient package, get Ryan Tomayko⁹'s Rocco¹⁰, the Ruby port that's available as a gem.
- If you're writing shell scripts, try Shocco¹¹, a port for the **POSIX shell**, also by Mr. Tomayko.
- If **Python** is more your speed, take a look at Nick Fitzgerald¹²'s Pycco¹³.
- For **Clojure** fans, Fogus¹⁴'s Marginalia¹⁵ is a bit of a departure from "quick-and-dirty", but it'll get the job done.
- There's a **Go** port called Gocco¹⁶, written by Nikhil Marathe¹⁷.
- Your all you **PHP** buffs out there, Fredi Bach's sourceMakeup¹⁸ (we'll let the faux pas with respect to our naming scheme slide), should do the trick nicely.
- **Lua** enthusiasts can get their fix with Robert Gieseke¹⁹'s Locco²⁰.
- And if you happen to be a **.NET** aficionado, check out Don Wilson²¹'s Nocco²².
- Going further afield from the quick-and-dirty, Groc²³ is a **CoffeeScript** fork of Docco that adds a searchable table of contents, and aims to gracefully handle large projects with complex hierarchies of code.

Note that not all ports will support all Docco features ... yet.

Main Documentation Generation Functions

Generate the documentation for our configured source file by copying over static assets, reading all the source files in, splitting them up into prose+code sections,

⁹<http://github.com/rtomayko>

¹⁰<http://rtomayko.github.com/rocco/rocco.html>

¹¹<http://rtomayko.github.com/shocco/>

¹²<http://github.com/fitzgen>

¹³<http://fitzgen.github.com/pycco/>

¹⁴<http://blog.fogus.me/>

¹⁵<http://fogus.me/fun/marginalia/>

¹⁶<http://nikhilm.github.com/gocco/>

¹⁷<https://github.com/nikhilm>

¹⁸<http://jquery-jkit.com/sourcemakeup/>

¹⁹<https://github.com/rgieseke>

²⁰<http://rgieseke.github.com/locco/>

²¹<https://github.com/dontangg>

²²<http://dontangg.github.com/nocco/>

²³<http://nevir.github.com/groc/>

highlighting each file in the appropriate language, and printing them out in an HTML template.

```
document = (options = {}, callback) ->
  configure options

  exec "mkdir -p #{config.output}", ->

    callback or= (error) -> throw error if error
    complete = ->
      exec [
        "cp -f #{config.css} #{config.output}"
        "cp -fR #{config.public} #{config.output}" if
        { fs.existsSync config.public
      ].join(' && '), callback

    files = config.sources.slice()

    nextFile = ->
      source = files.shift()
      fs.readFile source, (error, buffer) ->
        return callback error if error

        code = buffer.toString()
        sections = parse source, code
        format source, sections
        write source, sections
        if files.length then nextFile() else complete()

    nextFile()
```

Given a string of source code, **parse** out each block of prose and the code that follows it — by detecting which is which, line by line — and then create an individual **section** for it. Each section is an object with `docsText` and `codeText` properties, and eventually `docsHtml` and `codeHtml` as well.

```
parse = (source, code) ->
  lines = code.split '\n'
  sections = []
  lang = getLanguage source
  hasCode = docsText = codeText = ''

  save = ->
    sections.push {docsText, codeText}
    hasCode = docsText = codeText = ''
```

Our quick-and-dirty implementation of the literate programming style. Simply invert the prose and code relationship on a per-line basis, and then continue as normal below.

```
if lang.literate
  for line, i in lines
    lines[i] = if /\s*$/ .test line
      ''
    else if match = (/^([\ ]{4}|\t)/).exec line
      line[match[0].length..]
```

```

        else
            lang.symbol + ' ' + line

for line in lines
    if (not line and prev is 'text') or
        (line.match(lang.commentMatcher) and not
            { line.match(lang.commentFilter)})
        save() if hasCode
        docsText += (line = line.replace(lang.commentMatcher, ''))
        { + '\n'
        save() if /^(---+|===+)$/.test line
        prev = 'text'
    else
        hasCode = yes
        codeText += line + '\n'
        prev = 'code'
    save()

sections

```

To **format** and highlight the now-parsed sections of code, we use **Highlight.js** over **std.io**, and run the text of their corresponding comments through **Mark-down**, using **Marked**²⁴.

```

format = (source, sections) ->
    language = getLanguage source
    for section, i in sections
        code = highlight(language.name, section.codeText).value
        code = code.replace(/\s+$/, '')
        section.codeHtml = "<div
            { class='highlight'><pre>#{code}</pre></div>"
        section.docsHtml = marked(section.docsText)

```

Once all of the code has finished highlighting, we can **write** the resulting documentation file by passing the completed HTML sections into the template, and rendering it to the specified output path.

```

write = (source, sections) ->

    destination = (file) ->
        path.join(config.output, path.basename(file,
            { path.extname(file)}) + '.html')

```

The **title** of the file is either the first heading in the prose, or the name of the source file.

```

first = marked.lexer(sections[0].docsText)[0]
hasTitle = first and first.type is 'heading' and first.depth
    { is 1
title = if hasTitle then first.text else path.basename source

html = config.template {sources: config.sources, css:
    { path.basename(config.css),
        title, hasTitle, sections, path, destination,}

```

²⁴<https://github.com/chjj/marked>

```

console.log "docco: #{source} -> #{destination source}"
fs.writeFileSync destination(source), html

```

Configuration

Default configuration **options**. All of these may be overridden by command-line options.

```

config =
  layout:      'parallel'
  output:      'docs/'
  template:    null
  css:         null
  extension:   null

```

Configure this particular run of Docco. We might use a passed-in external template, or one of the built-in **layouts**. We only attempt to process source files for languages for which we have definitions.

```

configure = (options) ->
  _.extend config, _.pick(options, _.keys(config)...)

  if options.template
    config.layout = null
  else
    dir = config.layout = "#{__dirname}/resources/#{config.layout}"
    config.public      = "#{dir}/public" if fs.existsSync `
    & "#{dir}/public"
    config.template    = "#{dir}/docco.jst"
    config.css          = options.css or "#{dir}/docco.css"
  config.template = _.template `
    & fs.readFileSync(config.template).toString()

  config.sources = options.args.filter((source) ->
    lang = getLanguage source, config
    console.warn "docco: skipped unknown type (#{m})" unless lang
    lang
  ).sort()

```

Helpers & Initial Setup

Require our external dependencies.

```

_           = require 'underscore'
fs          = require 'fs'
path       = require 'path'
marked     = require 'marked'
commander  = require 'commander'
{highlight} = require 'highlight.js'
{spawn, exec} = require 'child_process'

```

Languages are stored in JSON in the file `resources/languages.json`. Each item maps the file extension to the name of the language and the `symbol` that indicates a line comment. To add support for a new programming language to Docco, just add it to the file.

```
languages = JSON.parse `
  fs.readFileSync("#{__dirname}/resources/languages.json")
```

Build out the appropriate matchers and delimiters for each language.

```
for ext, l of languages
```

Does the line begin with a comment?

```
  l.commentMatcher = ///^\s*#{l.symbol}\s?///
```

Ignore hashbangs²⁵ and interpolations...

```
  l.commentFilter = /(^\#![/]|^\s*#\{)/
```

A function to get the current language we’re documenting, based on the file extension. Detect and tag “`iterate`” `.ext.md` variants.

```
getLanguage = (source) ->
  ext = config.extension or path.extname(source) or `
    path.basename(source)
  lang = languages[ext]
  if lang and lang.name is 'markdown'
    codeExt = path.extname(path.basename(source, ext))
    if codeExt and codeLang = languages[codeExt]
      lang = _.extend {}, codeLang, {iterate: yes}
  lang
```

Keep it DRY. Extract the docco **version** from `package.json`

```
version = `
  JSON.parse(fs.readFileSync("#{__dirname}/package.json")).version
```

Command Line Interface

Finally, let’s define the interface to run Docco from the command line. Parse options using Commander²⁶.

```
run = (args = process.argv) ->
  c = config
  commander.version(version)
  .usage('[options] files')
  .option('-l, --layout [name]', 'choose a layout `
    (parallel, linear or classic)', c.layout)
```

²⁵[http://en.wikipedia.org/wiki/Shebang_\(Unix\)](http://en.wikipedia.org/wiki/Shebang_(Unix))

²⁶<https://github.com/visionmedia/commander.js>


```

.option('-o, --output [path]',    'output to a given 2
  ↳ folder', c.output)
.option('-c, --css [file]',      'use a custom css file', 2
  ↳ c.css)
.option('-t, --template [file]', 'use a custom .jst 2
  ↳ template', c.template)
.option('-e, --extension [ext]', 'assume a file extension 2
  ↳ for all inputs', c.extension)
.parse(args)
.name = "docco"
if commander.args.length
  document commander
else
  console.log commander.helpInformation()

```

Public API

```

Docco = module.exports = {run, document, parse, version}

```