highlight.js Documentation

Release 9.15.9

Ivan Sagalaev

Aug 19, 2019

Contents

Contents:

CHAPTER 1

Library API

Highlight.js exports a few functions as methods of the `hljs` object.

1.1 highlight(name, value, ignore_illegals, continuation)

Core highlighting function. Accepts a language name, or an alias, and a string with the code to highlight. The `ignore_illegals` parameter, when present and evaluates to a true value, forces highlighting to finish even in case of detecting illegal syntax for the language instead of throwing an exception. The `continuation` is an optional mode stack representing unfinished parsing. When present, the function will restart parsing from this state instead of initializing a new one. Returns an object with the following properties:

- `language`: language name, same as the one passed into a function, returned for consistency with `highlightAuto`
- `relevance`: integer value
- `value`: HTML string with highlighting markup
- `top`: top of the current mode stack

1.2 highlightAuto(value, languageSubset)

Highlighting with language detection. Accepts a string with the code to highlight and an optional array of language names and aliases restricting detection to only those languages. The subset can also be set with `configure`, but the local parameter overrides the option if set. Returns an object with the following properties:

- `language`: detected language
- `relevance`: integer value
- `value`: HTML string with highlighting markup

- `second_best`: object with the same structure for second-best heuristically detected language, may be absent

1.3 fixMarkup(value)

Post-processing of the highlighted markup. Currently consists of replacing indentation TAB characters and using `<br>` tags instead of new-line characters. Options are set globally with `configure`.

Accepts a string with the highlighted markup.

1.4 highlightBlock(block)

Applies highlighting to a DOM node containing code.

This function is the one to use to apply highlighting dynamically after page load or within initialization code of third-party Javascript frameworks.

The function uses language detection by default but you can specify the language in the `class` attribute of the DOM node. See the *class reference* for all available language names and aliases.

1.5 configure(options)

Configures global options:

- `tabReplace`: a string used to replace TAB characters in indentation.
- `useBR`: a flag to generate `<br>` tags instead of new-line characters in the output, useful when code is marked up using a non-`<pre>` container.
- `classPrefix`: a string prefix added before class names in the generated markup, used for backwards compatibility with stylesheets.
- `languages`: an array of language names and aliases restricting auto detection to only these languages.

Accepts an object representing options with the values to updated. Other options don't change

```
hljs.configure({
  tabReplace: '    ', // 4 spaces
  classPrefix: ''     // don't append class prefix
                      // ... other options aren't changed
})
hljs.initHighlighting();
```

1.6 initHighlighting()

Applies highlighting to all `<pre><code>..</code></pre>` blocks on a page.

1.7 initHighlightingOnLoad()

Attaches highlighting to the page load event.

1.8 registerLanguage(name, language)

Adds new language to the library under the specified name. Used mostly internally.

- `name`: a string with the name of the language being registered
- `language`: a function that returns an object which represents the language definition. The function is passed the `hljs` object to be able to use common regular expressions defined within it.

1.9 listLanguages()

Returns the languages names list.

1.10 getLanguage(name)

Looks up a language by name or alias.

Returns the language object if found, `undefined` otherwise.

CHAPTER 2

Language definition guide

2.1 Highlighting overview

Programming language code consists of parts with different rules of parsing: keywords like `for` or `if` don't make sense inside strings, strings may contain backslash-escaped symbols like `\"` and comments usually don't contain anything interesting except the end of the comment.

In highlight.js such parts are called "modes".

Each mode consists of:

- starting condition
- ending condition
- list of contained sub-modes
- lexing rules and keywords
- ...exotic stuff like another language inside a language

The parser's work is to look for modes and their keywords. Upon finding, it wraps them into the markup `<span class="...">...</span>` and puts the name of the mode ("string", "comment", "number") or a keyword group name ("keyword", "literal", "built-in") as the span's class name.

2.2 General syntax

A language definition is a JavaScript object describing the default parsing mode for the language. This default mode contains sub-modes which in turn contain other sub-modes, effectively making the language definition a tree of modes.

Here's an example:

```
{
  case_insensitive: true, // language is case-insensitive
  keywords: 'for if while',
```

(continues on next page)

(continued from previous page)

```
  contains: [
    {
      className: 'string',
      begin: '"', end: '"'
    },
    hljs.COMMENT(
      '/\\*', // begin
      '\\*/', // end
      {
        contains: [
          {
            className: 'doc', begin: '@\\w+'
          }
        ]
      }
    )
  ]
}
```

Usually the default mode accounts for the majority of the code and describes all language keywords. A notable exception here is XML in which a default mode is just a user text that doesn't contain any keywords, and most interesting parsing happens inside tags.

2.3 Keywords

In the simple case language keywords are defined in a string, separated by space:

```
{
  keywords: 'else for if while'
}
```

Some languages have different kinds of "keywords" that might not be called as such by the language spec but are very close to them from the point of view of a syntax highlighter. These are all sorts of "literals", "built-ins", "symbols" and such. To define such keyword groups the attribute `keywords` becomes an object each property of which defines its own group of keywords:

```
{
  keywords: {
    keyword: 'else for if while',
    literal: 'false true null'
  }
}
```

The group name becomes then a class name in a generated markup enabling different styling for different kinds of keywords.

To detect keywords highlight.js breaks the processed chunk of code into separate words — a process called lexing. The "word" here is defined by the regexp `[a-zA-Z][a-zA-Z0-9_]*` that works for keywords in most languages. Different lexing rules can be defined by the `lexemes` attribute:

```
{
  lexemes: '-[a-z]+',
  keywords: '-import -export'
}
```

2.4 Sub-modes

Sub-modes are listed in the `contains` attribute:

```
{
  keywords: '...',
  contains: [
    hljs.QUOTE_STRING_MODE,
    hljs.C_LINE_COMMENT,
    { ... custom mode definition ... }
  ]
}
```

A mode can reference itself in the `contains` array by using a special keyword `'self'`. This is commonly used to define nested modes:

```
{
  className: 'object',
  begin: '{', end: '}',
  contains: [hljs.QUOTE_STRING_MODE, 'self']
}
```

2.5 Comments

To define custom comments it is recommended to use a built-in helper function `hljs.COMMENT` instead of describing the mode directly, as it also defines a few default sub-modes that improve language detection and do other nice things.

Parameters for the function are:

```
hljs.COMMENT(
  begin,      // begin regex
  end,        // end regex
  extra       // optional object with extra attributes to override defaults
              // (for example {relevance: 0})
)
```

2.6 Markup generation

Modes usually generate actual highlighting markup — `<span>` elements with specific class names that are defined by the `className` attribute:

```
{
  contains: [
    {
      className: 'string',
      // ... other attributes
    },
    {
      className: 'number',
      // ...
    }
  ]
}
```

Names are not required to be unique, it's quite common to have several definitions with the same name. For example, many languages have various syntaxes for strings, comments, etc. . .

Sometimes modes are defined only to support specific parsing rules and aren't needed in the final markup. A classic example is an escaping sequence inside strings allowing them to contain an ending quote.

```
{
  className: 'string',
  begin: '"', end: '"',
  contains: [{begin: '\\\\.'}],
}
```

For such modes `className` attribute should be omitted so they won't generate excessive markup.

2.7 Mode attributes

Other useful attributes are defined in the *mode reference*.

2.8 Relevance

Highlight.js tries to automatically detect the language of a code fragment. The heuristics is essentially simple: it tries to highlight a fragment with all the language definitions and the one that yields most specific modes and keywords wins. The job of a language definition is to help this heuristics by hinting relative relevance (or irrelevance) of modes.

This is best illustrated by example. Python has special kinds of strings defined by prefix letters before the quotes: `r"..."`, `u"..."`. If a code fragment contains such strings there is a good chance that it's in Python. So these string modes are given high relevance:

```
{
  className: 'string',
  begin: 'r"', end: '"',
  relevance: 10
}
```

On the other hand, conventional strings in plain single or double quotes aren't specific to any language and it makes sense to bring their relevance to zero to lessen statistical noise:

```
{
  className: 'string',
  begin: '"', end: '"',
  relevance: 0
}
```

The default value for relevance is 1. When setting an explicit value it's recommended to use either 10 or 0.

Keywords also influence relevance. Each of them usually has a relevance of 1, but there are some unique names that aren't likely to be found outside of their languages, even in the form of variable names. For example just having `reinterpret_cast` somewhere in the code is a good indicator that we're looking at C++. It's worth to set relevance of such keywords a bit higher. This is done with a pipe:

```
{
  keywords: 'for if reinterpret_cast|10'
}
```

2.9 Illegal symbols

Another way to improve language detection is to define illegal symbols for a mode. For example in Python first line of class definition (`class MyClass(object):`) cannot contain symbol "{" or a newline. Presence of these symbols clearly shows that the language is not Python and the parser can drop this attempt early.

Illegal symbols are defined as a a single regular expression:

```
{
  className: 'class',
  illegal: '[${]'
}
```

2.10 Pre-defined modes and regular expressions

Many languages share common modes and regular expressions. Such expressions are defined in core highlight.js code at the end under "Common regexps" and "Common modes" titles. Use them when possible.

2.11 Contributing

Follow the *contributor checklist*.

CHAPTER 3

Mode reference

3.1 Types

Types of attributes values in this reference:

identifier	String suitable to be used as a Javascript variable and CSS class name (i.e. mostly `/[A-Za-z0-9_]+/`)
regexp	String representing a Javascript regexp. Note that since it's not a literal regexp all back-slashes should be repeated twice
boolean	Javascript boolean: `true` or `false`
number	Javascript number
object	Javascript object: `{ ... }`
array	Javascript array: `[ ... ]`

3.2 Attributes

3.2.1 case_insensitive

type: boolean

Case insensitivity of language keywords and regexps. Used only on the top-level mode.

3.2.2 aliases

type: array

A list of additional names (besides the canonical one given by the filename) that can be used to identify a language in HTML classes and in a call to *getLanguage*.

3.2.3 className

type: identifier

The name of the mode. It is used as a class name in HTML markup.

Multiple modes can have the same name. This is useful when a language has multiple variants of syntax for one thing like string in single or double quotes.

3.2.4 begin

type: regexp

Regular expression starting a mode. For example a single quote for strings or two forward slashes for C-style comments. If absent, `begin` defaults to a regexp that matches anything, so the mode starts immediately.

3.2.5 end

type: regexp

Regular expression ending a mode. For example a single quote for strings or "$" (end of line) for one-line comments.

It's often the case that a beginning regular expression defines the entire mode and doesn't need any special ending. For example a number can be defined with `begin: "\\b\\d+"` which spans all the digits.

If absent, `end` defaults to a regexp that matches anything, so the mode ends immediately.

Sometimes a mode can end not by itself but implicitly with its containing (parent) mode. This is achieved with *endsWithParent* attribute.

3.2.6 beginKeywords

type: string

Used instead of `begin` for modes starting with keywords to avoid needless repetition:

```
{
  begin: '\\b(extends|implements) ',
  keywords: 'extends implements'
}
```

... becomes:

```
{
  beginKeywords: 'extends implements'
}
```

Unlike the *keywords* attribute, this one allows only a simple list of space separated keywords. If you do need additional features of `keywords` or you just need more keywords for this mode you may include `keywords` along with `beginKeywords`.

3.2.7 endsWithParent

type: boolean

A flag showing that a mode ends when its parent ends.

This is best demonstrated by example. In CSS syntax a selector has a set of rules contained within symbols "{" and "}". Individual rules separated by ";" but the last one in a set can omit the terminating semicolon:

```
p {
  width: 100%; color: red
}
```

This is when `endsWithParent` comes into play:

```
{
  className: 'rules', begin: '{', end: '}',
  contains: [
    {className: 'rule', /* ... */ end: ';', endsWithParent: true}
  ]
}
```

3.2.8 endsParent

type: boolean

Forces closing of the parent mode right after the current mode is closed.

This is used for modes that don't have an easily expressible ending lexeme but instead could be closed after the last interesting sub-mode is found.

Here's an example with two ways of defining functions in Elixir, one using a keyword `do` and another using a comma:

```
def foo :clear, list do
  :ok
end

def foo, do: IO.puts "hello world"
```

Note that in the first case the parameter list after the function title may also include a comma. And if we're only interested in highlighting a title we can tell it to end the function definition after itself:

```
{
  className: 'function',
  beginKeywords: 'def', end: /\B\b/,
  contains: [
    {
      className: 'title',
      begin: hljs.IDENT_RE, endsParent: true
    }
  ]
}
```

(The `end: /\B\b/` regex tells function to never end by itself.)

3.2.9 endSameAsBegin

type: boolean

Acts as `end` matching exactly the same string that was found by the corresponding `begin` regexp.

For example, in PostgreSQL string constants can uee "dollar quotes", consisting of a dollar sign, an optional tag of zero or more characters, and another dollar sign. String constant must be ended with the same construct using the same tag. It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level:

```
$foo$
  ...
  $bar$ nested $bar$
  ...
$foo$
```

In this case you can't simply specify the same regexp for `begin` and `end` (say, `"\\$[a-z]\\$"`), but you can use `begin: "\\$[a-z]\\$"` and `endSameAsBegin: true`.

3.2.10 lexemes

type: regexp

A regular expression that extracts individual lexemes from language text to find *keywords* among them. Default value is `hljs.IDENT_RE` which works for most languages.

3.2.11 keywords

type: object

Keyword definition comes in two forms:

- `'for while if else weird_voodoo|10 ... '` – a string of space-separated keywords with an optional relevance over a pipe
- `{'keyword': ' ... ', 'literal': ' ... '}` – an object whose keys are names of different kinds of keywords and values are keyword definition strings in the first form

For detailed explanation see *Language definition guide*.

3.2.12 illegal

type: regexp

A regular expression that defines symbols illegal for the mode. When the parser finds a match for illegal expression it immediately drops parsing the whole language altogether.

3.2.13 excludeBegin, excludeEnd

type: boolean

Exclude beginning or ending lexemes out of mode's generated markup. For example in CSS syntax a rule ends with a semicolon. However visually it's better not to color it as the rule contents. Having `excludeEnd: true` forces a `<span>` element for the rule to close before the semicolon.

3.2.14 returnBegin

type: boolean

Returns just found beginning lexeme back into parser. This is used when beginning of a sub-mode is a complex expression that should not only be found within a parent mode but also parsed according to the rules of a sub-mode.

Since the parser is effectively goes back it's quite possible to create a infinite loop here so use with caution!

3.2.15 returnEnd

type: boolean

Returns just found ending lexeme back into parser. This is used for example to parse Javascript embedded into HTML. A Javascript block ends with the HTML closing tag `</script>` that cannot be parsed with Javascript rules. So it is returned back into its parent HTML mode that knows what to do with it.

Since the parser is effectively goes back it's quite possible to create a infinite loop here so use with caution!

3.2.16 contains

type: array

The list of sub-modes that can be found inside the mode. For detailed explanation see *Language definition guide*.

3.2.17 starts

type: identifier

The name of the mode that will start right after the current mode ends. The new mode won't be contained within the current one.

Currently this attribute is used to highlight Javascript and CSS contained within HTML. Tags `<script>` and `<style>` start sub-modes that use another language definition to parse their contents (see *subLanguage*).

3.2.18 variants

type: array

Modification to the main definitions of the mode, effectively expanding it into several similar modes each having all the attributes from the main definition augmented or overridden by the variants:

```
{
  className: 'string',
  contains: [hljs.BACKSLASH_ESCAPE],
  relevance: 0,
  variants: [
    {begin: /"/, end: /"/},
    {begin: /'/, end: /'/, relevance: 1}
  ]
}
```

3.2.19 subLanguage

type: string or array

Highlights the entire contents of the mode with another language.

When using this attribute there's no point to define internal parsing rules like *lexemes* or *keywords*. Also it is recommended to skip `className` attribute since the sublanguage will wrap the text in its own `<span class="language-name">`.

The value of the attribute controls which language or languages will be used for highlighting:

- language name: explicit highlighting with the specified language
- empty array: auto detection with all the languages available
- array of language names: auto detection constrained to the specified set

3.2.20 skip

type: boolean

Skips any markup processing for the mode ensuring that it remains a part of its parent buffer along with the starting and the ending lexemes. This works in conjunction with the parent's *subLanguage* when it requires complex parsing.

Consider parsing PHP inside HTML:

```
<p><? echo 'PHP'; /* ?> */ ?></p>
```

The `?>` inside the comment should **not** end the PHP part, so we have to handle pairs of `/* .. */` to correctly find the ending `?>`:

```
{
  begin: /<\?/, end: /\?>/,
  subLanguage: 'php',
  contains: [{begin: '/\\*', end: '\\*/', skip: true}]
}
```

Without `skip: true` every comment would cause the parser to drop out back into the HTML mode.

3.2.21 disableAutodetect

type: boolean

Disables autodetection for this language.

CHAPTER 4

CSS classes reference

4.1 Stylable classes

General-purpose	
keyword	keyword in a regular Algol-style language
built_in	built-in or library object (constant, class, function)
type	user-defined type in a language with first-class syntactically significant type
literal	special identifier for a built-in value (“true”, “false”, “null”)
number	number, including units and modifiers, if any.
regexp	literal regular expression
string	literal string, character
subst	parsed section inside a literal string
symbol	symbolic constant, interned string, goto label
class	class or class-level declaration (interfaces, traits, modules, etc)
function	function or method declaration
title	name of a class or a function at the place of declaration
params	block of function arguments (parameters) at the place of declaration
Meta	
comment	comment
doctag	documentation markup within comments
meta	flags, modifiers, annotations, processing instructions, preprocessor directive
meta-keyword	keyword or built-in within meta construct
meta-string	string within meta construct
Tags, attributes, configs	
section	heading of a section in a config file, heading in text markup
tag	XML/HTML tag
name	name of an XML tag, the first word in an s-expression
builtin-name	s-expression name from the language standard library
attr	name of an attribute with no language defined semantics (keys in JSON, set
attribute	name of an attribute followed by a structured value part, like CSS propertie

Table 1 – continued from previous page

variable	variable in a config or a template file, environment var expansion in a script
Markup	
bullet	list item bullet in text markup
code	code block in text markup
emphasis	emphasis in text markup
strong	strong emphasis in text markup
formula	mathematical formula in text markup
link	hyperlink in text markup
quote	quotation in text markup
CSS	
selector-tag	tag selector in CSS
selector-id	#id selector in CSS
selector-class	.class selector in CSS
selector-attr	[attr] selector in CSS
selector-pseudo	:pseudo selector in CSS
Templates	
template-tag	tag of a template language
template-variable	variable in a template language
diff	
addition	added or changed line in a diff
deletion	deleted line in a diff
ReasonML	
operator	reasonml operator such as pipe
pattern-match	reasonml pattern matching matchers
typing	type signatures on function parameters
constructor	type constructors
module-access	scope access into a ReasonML module
module	ReasonML module reference within scope access

CHAPTER 5

Style guide

5.1 Key principle

Highlight.js themes are language agnostic.

Instead of trying to make a *rich* set of highlightable classes look good in a handful of languages we have a *limited* set of classes that work for all languages.

Hence, there are two important implications:

- Highlight.js styles tend to be minimalistic.
- It's not possible to exactly emulate themes from other highlighting engines.

5.2 Defining a theme

A theme is a single CSS defining styles for class names listed in the *class reference*. The general guideline is to style all available classes, however an author may deliberately choose to exclude some (for example, `.attr` is usually left unstyled).

You are not required to invent a separate styling for every group of class names, it's perfectly okay to group them:

```
.hljs-string,
.hljs-section,
.hljs-selector-class,
.hljs-template-variable,
.hljs-deletion {
  color: #800;
}
```

Use as few or as many unique style combinations as you want.

5.3 Typography and layout dos and don'ts

Don't use:

- non-standard borders/margin/paddings for the root container `.hljs`
- specific font faces
- font size, line height and anything that affects position and size of characters within the container

Okay to use:

- colors (obviously!)
- italic, bold, underlining, etc.
- image backgrounds

These may seem arbitrary at first but it's what has shown to make sense in practice.

There's also a common set of rules that *has* to be defined for the root container verbatim:

```
.hljs {
  display: block;
  overflow-x: auto;
  padding: 0.5em;
}
```

5.4 `.subst`

One important caveat: don't forget to style `.subst`. It's used for parsed sections within strings and almost always should be reset to the default color:

```
.hljs,
.hljs-subst {
  color: black;
}
```

5.5 Contributing

You should include a comment at the top of the CSS file with attribution and other meta data if necessary. The format is free:

```
/*

Fancy style (c) John Smith <email@domain.com>

*/
```

If you're a new contributor add yourself to the authors list in AUTHORS.en.txt Also update CHANGES.md with your contribution.

Send your contribution as a pull request on GitHub.

CHAPTER 6

Language contributor checklist

6.1 1. Put language definition into a .js file

The file defines a function accepting a reference to the library and returning a language object. The library parameter is useful to access common modes and regexps. You should not immediately call this function, this is done during the build process and details differ for different build targets.

```
function(hljs) {
  return {
    keywords: 'foo bar',
    contains: [ ..., hljs.NUMBER_MODE, ... ]
  }
}
```

The name of the file is used as a short language identifier and should be usable as a class name in HTML and CSS.

6.2 2. Provide meta data

At the top of the file there is a specially formatted comment with meta data processed by a build system. Meta data format is simply key-value pairs each occupying its own line:

```
/*
Language: Superlanguage
Requires: java.js, sql.js
Author: John Smith <email@domain.com>
Contributors: Mike Johnson <...@...>, Matt Wilson <...@...>
Description: Some cool language definition
*/
```

`Language` — the only required header giving a human-readable language name.

`Requires` — a list of other language files required for this language to work. This make it possible to describe languages that extend definitions of other ones. Required files aren't processed in any special way. The build system just makes sure that they will be in the final package in `LANGUAGES` object.

The meaning of the other headers is pretty obvious.

6.3 3. Create a code example

The code example is used both to test language detection and for the demo page on https://highlightjs.org/. Put it in `test/detect/<language>/default.txt`.

Take inspiration from other languages in `test/detect/` and read *testing instructions* for more details.

6.4 4. Write class reference

Class reference lives in the *CSS classes reference*.. Describe shortly names of all meaningful modes used in your language definition.

6.5 5. Add yourself to AUTHORS.*.txt and CHANGES.md

If you're a new contributor add yourself to the authors list. Also it will be good to update CHANGES.md.

6.6 6. Create a pull request

Send your contribution as a pull request on GitHub.

CHAPTER 7

Building and testing

To actually run highlight.js it is necessary to build it for the environment where you're going to run it: a browser, the node.js server, etc.

7.1 Building

The build tool is written in JavaScript using node.js. Before running the script, make sure to have node installed and run `npm install` to get the dependencies.

The tool is located in `tools/build.js`. A few useful examples:

- Build for a browser using only common languages:

```
node tools/build.js :common
```

- Build for node.js including all available languages:

```
node tools/build.js -t node
```

- Build two specific languages for debugging, skipping compression in this case:

```
node tools/build.js -n python ruby
```

On some systems the node binary is named `nodejs`; simply replace `node` with `nodejs` in the examples above if that is the case.

The full option reference is available with the usual `--help` option.

The build result will be in the `build/` directory.

7.2 Basic testing

The usual approach to debugging and testing a language is first doing it visually. You need to build highlight.js with only the language you're working on (without compression, to have readable code in browser error messages) and then use the Developer tool in `tools/developer.html` to see how it highlights a test snippet in that language.

A test snippet should be short and give the idea of the overall look of the language. It shouldn't include every possible syntactic element and shouldn't even make practical sense.

After you satisfied with the result you need to make sure that language detection still works with your language definition included in the whole suite.

Testing is done using Mocha and the files are found in the `test/` directory. You can use the node build to run the tests in the command line with `npm test` after installing the dependencies with `npm install`.

Note: for Debian-based machine, like Ubuntu, you might need to create an alias or symbolic link for nodejs to node. The reason for this is the dependencies that are requires to test highlight.js has a reference to "node".

Place the snippet you used inside the browser in `test/detect/<language>/default.txt`, build the package with all the languages for node and run the test suite. If your language breaks auto-detection, it should be fixed by *improving relevance*, which is a black art in and of itself. When in doubt, please refer to the discussion group!

7.3 Testing markup

You can also provide additional markup tests for the language to test isolated cases of various syntactic construct. If your language has 19 different string literals or complicated heuristics for telling division (`/`) apart from regexes (`/ .. /`) – this is the place.

A test case consists of two files:

- `test/markup/<language>/<test_name>.txt`: test code
- `test/markup/<language>/<test_name>.expect.txt`: reference rendering

To generate reference rendering use the Developer tool located at `tools/developer.html`. Make sure to explicitly select your language in the drop-down menu, as automatic detection is unlikely to work in this case.

CHAPTER 8

Maintainer's guide

8.1 Commit policy

- Pull requests from outside contributors require a review from a maintainer.
- Maintainers should avoid working on a master branch directly and create branches for everything. A code review from another maintainer is recommended but not required, use your best judgment.

8.2 Release process

Releases happen on a 6-week schedule. Currently due to a long break the date of the next release is not set.

- Update CHANGES.md with everything interesting since the last update.
- Update version numbers using the three-part x.y.z notation everywhere:
  - The header in CHANGES.md (this is where the site looks for the latest version number)
  - `"version"` attribute in package.json
  - `"version"` attribute in package-lock.json (run *npm install*)
  - Two places in docs/conf.py (`version` and `release`)
- Commit the version changes and tag the commit with the plain version number (no "v." or anything like that)
- Push the commit and the tags to master (`git push && git push --tags`)

Pushing the tag triggers the update process which can be monitored at http://highlightjs.org/api/release/

When something didn't work *and* it's fixable in code (version numbers mismatch, last minute patches, etc), simply make another release incrementing the third (revision) part of the version number.

Miscellaneous:

CHAPTER 9

Line numbers

Highlight.js' notable lack of line numbers support is not an oversight but a feature. Following is the explanation of this policy from the current project maintainer (hey guys!):

> One of the defining design principles for highlight.js from the start was simplicity. Not the simplicity of code (in fact, it's quite complex) but the simplicity of usage and of the actual look of highlighted snippets on HTML pages. Many highlighters, in my opinion, are overdoing it with such things as separate colors for every single type of lexemes, striped backgrounds, fancy buttons around code blocks and — yes — line numbers. The more fancy stuff resides around the code the more it distracts a reader from understanding it.
>
> This is why it's not a straightforward decision: this new feature will not just make highlight.js better, it might actually make it worse simply by making it look more bloated in blog posts around the Internet. This is why I'm asking people to show that it's worth it.
>
> The only real use-case that ever was brought up in support of line numbers is referencing code from the descriptive text around it. On my own blog I was always solving this either with comments within the code itself or by breaking the larger snippets into smaller ones and describing each small part separately. I'm not saying that my solution is better. But I don't see how line numbers are better either. And the only way to show that they are better is to set up some usability research on the subject. I doubt anyone would bother to do it.
>
> Then there's maintenance. So far the core code of highlight.js is maintained by only one person — yours truly. Inclusion of any new code in highlight.js means that from that moment I will have to fix bugs in it, improve it further, make it work together with the rest of the code, defend its design. And I don't want to do all this for the feature that I consider "evil" and probably will never use myself.

This position is subject to discuss. Also it doesn't stop anyone from forking the code and maintaining line-numbers implementation separately.

CHAPTER 10

On requesting new languages

This is a general answer to requests for adding new languages that appear from time to time in the highlight.js issue tracker and discussion group.

> Highlight.js doesn't have a fundamental plan for implementing languages, instead the project works by accepting language definitions from interested contributors. There are also no rules at the moment forbidding any languages from being added to the library, no matter how obscure or weird.
>
> This means that there's no point in requesting a new language without providing an implementation for it. If you want to see a particular language included in highlight.js but cannot implement it, the best way to make it happen is to get another developer interested in doing so. Here's our *Language definition guide*.

Links:

- Code: https://github.com/highlightjs/highlight.js
- Discussion: http://groups.google.com/group/highlightjs
- Bug tracking: https://github.com/highlightjs/highlight.js/issues

CHAPTER 11

Indices and tables

- genindex
- modindex
- search