LEARNING jade

#jade

Table of Contents

About

Chapter 1: Getting started with jade

Remarks

Pug (formerly known as jade) is a high performance template engine heavily influenced by Haml and implemented with JavaScript for Node.js and browsers.

1. Produces HTML
2. Supports dynamic code
3. Supports reusability (DRY)

Home page

Repository on GitHub

Versions

version	release date
0.0.2	2010-07-03
1.0.0	2013-12-22
1.11.0	2015-06-12

Examples

Installation or Setup

Before to launch you to code with Pug, you need to have some prerequisits.

You will need to install:

- NodeJS with NPM
- ExpressJS (optional)

After installing NodeJS, you can check in your terminal the correct installation doing:

```
$ node -v
```

If successful, it will print the number of Node's version.

To install Pug into your project, the preferred and easy way is through NPM (Node Package Manager). If you are familiar with that, simply execute this line of code in your Terminal:

```
$ npm install pug
```

If you want to install globally, you can type:

```
$ npm install pug-cli -g
```

and run with

```
$ pug --help
```

Syntax

Pug (old name is Jade) is a clean, whitespace sensitive syntax for writing HTML. Here is a simple example:

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) bar(1 + 5)
  body
    h1 Pug - node template engine
    #container.col
      if youAreUsingPug
        p You are amazing
      else
        p Get on it!
      p.
        Pug is a terse and simple templating language with a
        strong focus on performance and powerful features.
```

Produces following output as HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Pug</title>
    <script type="text/javascript">
      if (foo) bar(1 + 5)
    </script>
  </head>
  <body>
    <h1>Pug - node template engine</h1>
    <div id="container" class="col">
      <p>You are amazing</p>
      <p>Pug is a terse and simple templating language with a strong focus on performance and
powerful features.</p>
    </div>
  </body>
</html>
```

Here are the rules to render Pug to HTML code:

1. By indenting the text, the HTML tree will be build. indenting could be used with spaces or tabs. This could not be mixed!
2. HTML tags are written without `<` and `>`. Attributes are places between round brackets.
3. Comment could be made with `//` or `<!-- -->`. Comments with `//-` are not visible in the rendered HTML.
4. With `#{ }` will an offered model generated: `#{header} #{user.username}`.
5. The `#` (hashtag) without braces will a `div` element created with the text as ID. Example `#myID` will be rendered as `<div id="myID"></div>`.
6. With a `.` *(point)* will a `div` generated with a class attribute. Example: `.myClass` will be rendered as `<div class="myClass"></div>`
7. With `=` (equality sign followed by a space), a variable will be retrieved. Exaple: `h1= title`
8. A `!=` (not equal to) retrieved a variable without escaping.
9. A `-` (hyphen) allows you to write JavaScript. Example: `- console.log("foo");`
10. Linking to an external file can as follow: `script(src="/js/chat.js")`
11. Inline script could by using this `script.`.
12. A directive for adding the basic layout: `extends ../layout`.
13. At `layout.pug` happens the inserting by using `block content`
14. Use of partials could on two ways:
    1. by partial: `!= partial(template file name/options)`.
    2. By include: `include ../includes/footer`
15. The inverse of include is `extend`. This allows from a page "html `block` parts" to send to a layout page for example: `extend layout`
16. Concatenating happens with the `+` (plus) or `#` (hashtag) char. The plus is used at JavaScript code. The hashtag in HTML and renders the content: `p The name is: #{myName}

Using pug with Node.js

```
var pug = require('pug');

// compile
var fn = pug.compile('string of pug', options);
var html = fn(locals);

// render
var html = pug.render('string of pug', merge(options, locals));

// renderFile
var html = pug.renderFile('filename.pug', merge(options, locals));
```

Options

- *filename* Used in exceptions, and required when using includes
- *compileDebug* When false no debug instrumentation is compiled
- *pretty* Add pretty-indentation whitespace to output (false by default)

Read Getting started with jade online: https://riptutorial.com/jade/topic/1709/getting-started-with-jade

Chapter 2: Attributes

Examples

Normal Attributes

HTML Tag:

Tag attributes look similar to html, however their values are just regular JavaScript.

```
a(href='google.com') Google
a(class='button', href='google.com') Google
```

Result:

```
<a href="google.com">Google</a><a href="google.com" class="button">Google</a>
```

Variable:

All the normal JavaScript expressions work fine too:

```
- var authenticated = true
body(class=authenticated ? 'authed' : 'anon')
```

Result:

```
<body class="authed"></body>
```

Many Attributes

If you have many attributes, you can also spread them across many lines:

```
input(
  type='checkbox'
  name='agreement'
  checked
)
```

Result:

```
<input type="checkbox" name="agreement" checked="checked"/>
```

Unescaped Attributes

By default, all attributes are escaped (replacing special characters with escape sequences) to prevent attacks such as cross site scripting. If you need to use special characters you can use `!=` instead of `=`.

Code:

```
div(escaped="<code>")
div(unescaped!="<code>")
```

Result:

```
<div escaped="&lt;code&gt;"></div>
<div unescaped="<code>"></div>
```

Boolean Attributes

Boolean attributes are mirrored by Jade, and accept bools, aka `true` or `false`. When no value is specified true is assumed.

Code:

```
input(type='checkbox', checked)
input(type='checkbox', checked=true)
input(type='checkbox', checked=false)
input(type='checkbox', checked=true.toString())
```

Result:

```
<input type="checkbox" checked="checked"/>
<input type="checkbox" checked="checked"/>
<input type="checkbox"/>
<input type="checkbox" checked="true"/>
```

If the doctype is `html` jade knows not to mirror the attribute and uses the terse style (understood by all browsers).

Code:

```
doctype html
input(type='checkbox', checked)
input(type='checkbox', checked=true)
input(type='checkbox', checked=false)
input(type='checkbox', checked=true && 'checked')
```

Result:

```
<!DOCTYPE html>
<input type="checkbox" checked>
<input type="checkbox" checked>
<input type="checkbox">
<input type="checkbox" checked="checked">
```

Style Attributes

The `style` attribute can be a string (like any normal attribute) but it can also be an object, which is handy when parts of the style are generated by JavaScript.

Code:

```
a(style={color: 'red', background: 'green'})
```

Result:

```
<a style="color:red;background:green"></a>
```

Class Attributes

The `class` attribute can be a string (like any normal attribute) but it can also be an array of class names, which is handy when generated from JavaScript.

Code:

```
- var classes = ['foo', 'bar', 'baz']
a(class=classes)
//- the class attribute may also be repeated to merge arrays
a.bing(class=classes class=['bing'])
```

Result:

```
<a class="foo bar baz"></a><a class="bing foo bar baz bing"></a>
```

It can also be an object mapping class names to true or false values, which is useful for applying conditional classes

Code:

```
- var currentUrl = '/about'
a(class={active: currentUrl === '/'} href='/') Home
a(class={active: currentUrl === '/about'} href='/about'
```

Result:

```
<a href="/">Home</a><a href="/about" class="active">About</a>
```

Class Literal

Classes may be defined using a `.classname` syntax:

Code:

```
a.button
```

Result:

```
<a class="button"></a>
```

Since div's are such a common choice of tag, it is the default if you omit the tag name:

Code:

```
.content
```

Result:

```
<div class="content"></div>
```

ID Literal

IDs may be defined using a `#idname` syntax:

Code:

```
a#main-link
```

Result:

```
<a id="main-link"></a>
```

Since div's are such a common choice of tag, it is the default if you omit the tag name:

Code:

```
#content
```

Result:

```
<div id="content"></div>
```

&attributes

Pronounced "and attributes", the `&attributes` syntax can be used to explode an object into attributes of an element.

Code:

```
div#foo(data-bar="foo")&attributes({'data-foo': 'bar'})
```

Result:

```
<div id="foo" data-bar="foo" data-foo="bar"></div>
```

The object does not have to be an object literal. It can also just be a variable that has an object as its value (see also Mixin Attributes)

Code:

```
- var attributes = {'data-foo': 'bar'};
div#foo(data-bar="foo")&attributes(attributes)
```

Result:

```
<div id="foo" data-bar="foo" data-foo="bar"></div>
```

Read Attributes online: https://riptutorial.com/jade/topic/6641/attributes

Chapter 3: Case

Examples

Case

```
- var friends = 10
case friends
  when 0
    p you have no friends
  when 1
    p you have a friend
  default
    p you have #{friends} friends
```

Result is:

```
<p>you have 10 friends</p>
```

Read Case online: https://riptutorial.com/jade/topic/4012/case

Chapter 4: harp js

Introduction

Harp is a static web server with built-in preprocessing. Harp can compile your project down to static assets, HTML, CSS and JavaScript, with no configuration necessary. You may also use Harp as a Node library for compiling or running as a server.

Harp includes the common, useful preprocessors by default. It serves Jade (Pug), Markdown, EJS, CoffeeScript, LESS, Sass and Stylus.

Examples

How to set up Harp

Harp doesn’t require any configuration to get started. Install Harp in your terminal using the command: `npm install -g harp`.

```
$ sudo npm install -g harp
$ harp init myproject
$ harp server myproject
```

Read harp js online: https://riptutorial.com/jade/topic/9863/harp-js

Credits

S. No	Chapters	Contributors
1	Getting started with jade	Community, David Dias, H. Pauwelyn, Huy Nguyen, Naeem Shaikh
2	Attributes	Huy Nguyen
3	Case	Huy Nguyen
4	harp js	Alice Tribuleva, Mohit Bhardwaj