A Beginner's Introduction to CoffeeKup

By Mark Hahn

CoffeeKup uses a simple scheme to provide a concise, expressive, easy-to-read, and time-saving HTML templating solution. It is based on the CoffeeScript language, with which you will need to be familiar. If you aren't already hooked on CoffeeScript then visit http://coffeescript.org first to find out what you are missing. Then come back here to also get hooked on CoffeeKup.

This introduction is for CoffeeKup beginners like myself (I'm learning it as I write this). Let's go through this together step by step. Once you complete this I suggest you go to CoffeeKup's github page to learn more. Currently the only discussion of CoffeeKup is on CoffeeKup's issues page.

Unlike most tutorials I will not need to help you install CoffeeKup to follow along with the examples. I will give the results of the template with each example. You might also want to bring up http://coffeekup.org in another window and paste these examples into the left pane. This will allow you to play around with the template and see the results immediately. (This also makes a great tool to use while you are writing your own CoffeeKup code).

Let's Get Started - Hello World

First our mandatory friend, Hello World. In each example the CoffeeKup template code appears first followed by the rendered HTML.

```
head ->
    title 'Hello World'
body ->
```

```
<head>
    <title>Hello World</title>
</head>
<body>
</body>
```

First of all, note that the template code is real CoffeeScript code. CoffeeKup is CoffeeScript. Except for some important CoffeeScript code added invisibly to the top and bottom of the template, the CoffeeScript you write in the template is executed directly to render the output. This is very different from most template engines and is the reason CoffeeKup offers all the great features mentioned at the beginning.

So how did `head ->` become <head>? And where does the output come from? There is nothing like a `write` function in the template to send out the results. And how did </head> appear out of thin air? The secret ingredient in the coffee recipe is the extra code that was mentioned above.

The top part of the added code defines a lot of things, the most important of which are the functions who share their names with all the possible HTML tags. These functions, when executed, generate the associated HTML code and append it to a buffer, also defined in the invisible code, that accumulates all of the output HTML. When all of your template code has been executed, the buffer containing

the complete HTML is then returned as output by an invisible `return buffer` statement added to the bottom of the template.

Let's walk through the execution of the Hello World template code. First the `head` function is called with one argument, a one-line function that calls `title`. The `head` function adds the <head> text to the buffer, then calls the `title` function which adds it's own HTML to the buffer, and finally adds the closing </head>.

The `title` function was called with `"Hello World"` as its only argument. In a case like this, the function only had to wrap <title> and </title> around the string it was passed and add the whole thing to the buffer. The `body` function did the same thing as the `head` function except that the function passed to it added nothing to the output buffer.

So the "tag" functions create all the resulting HTML by adding their arguments to the output buffer while executing the function arguments to create the nesting. Quite elegant, yes?

Adding Attributes

We know how to insert anything we want for the inner HTML of a tag. We need only include an arbitrary string as an argument to the tag function. But how do you put attributes inside the tag itself? Luckily that is very easy. Check this out ⋯

```
div id:"ugly-box", style:"width:90px, height:90px,
        background-color: purple, border: 5px green"

<div id="ugly-box"
     style="width:100px, height:100px, background-color:
        purple, border: 5px green">
</div>
```

Any `object` (aka `hash`) used as an argument to a tag function is interpreted as a set of attributes. The hash keys are the attribute names and the hash values are the attribute values. In CoffeeScript hashes are created easily and they are perfect for CoffeeKup's attributes.

Let's look at this more complicated example which ties everything we know together ⋯

```
div id:"another-ugly-box", style:"background-color:
     purple, border: 5px yellow", ->
    span color:"green", "And I'm ugly text"

<div id="another-ugly-box" style="background-color:
     purple, border: 5px yellow">
  <span color="green">And I'm ugly text</span>
</div>
```

Now it is starting to look like real HTML you'd find on an ugly web page.

Lonely Text

At this point in my use of CoffeeKup I was starting to think I knew how to generate any HTML, but then I ran into a stumbling block. I needed to put some text between tags and not inside a tag. This is a hole in the CoffeeKup logic described so far, but the hole has been filled with a fake tag named `text` ···

```
span color:"red", "I'm bright red!"
text "I'm boring black"
span color:"blue", "I'm feeling blue"

<span color="red">I'm bright red!</span>
I'm boring black
<span color="blue">I'm feeling blue</span>
```

The `text` tag (function) just adds whatever text is in its string argument to the output buffer. If we removed `text` from the beginning of the middle line, that line with only the string would be legal CoffeeScript, and the template would execute without error, but the text would be lost because there would be no function to add it to the output buffer.

Before we leave the discussion of general text I'd like to point something out. Whether it is a string argument to a real tag like `div`, or a string argument to the fake `text` tag, a string can contain any text, even HTML. We will learn how to use this to our advantage in the section Homemade Html.

Variables, Conditionals, and Loops

If this was all there was to CoffeeKup then it would already be quite useful as a way to write all your HTML in a concise way. No more adding all those nasty closing tags. But wait, there's more ···

As you might have guessed, because CoffeeKup is executing arbitrary CoffeeScript code, there are a lot of fancy things we can do other than just generate static HTML. Let's look at another example ···

```
if true
    for i in [2..4]
        p ->
            text "I want #{i} hamburgers"

<p>I want 2 hamburgers</p>
<p>I want 3 hamburgers</p>
<p>I want 4 hamburgers</p>
```

First note that the entire snippet is conditional on the `if` statement evaluating to `true`. If you changed `true` to `false` then this example would not output anything. This is easy to understand since code must execute to add things to the output buffer. This example is good at showing how CoffeeKup is just CoffeeScript code executing with no magic happening behind the scenes, except for the magical output buffer.

The `for` loop simply executes its block of code, which happens to output a paragraph of text. It executes it three times so that block of code added its HTML to the buffer three times.

Note also the use of the variable `i` in the text string. It is evaluated and added to the string, which is called interpolation. The syntax `#{i}` that mixes it in is straight CoffeeScript. Once again CoffeeKup got a cool feature for free from CoffeeScript. It looks almost like a more traditional template syntax such as mustache, which would have {{i}}. Remember that this CoffeeScript interpolation only works inside double quotes `"`, not single.

Variables can be defined and used freely in your CoffeeKup template. In a later section, Keeping Things In Context, we will see that variables can be used that are defined outside of the template.

Cool Formatting

If you are fluent in CoffeeScript, then this will be obvious, but there are cool ways to clean up the last example. There are two or three CoffeeScript features that can be used to turn the four-line example above into this one-liner ⋯

```
p "I want #{i} hamburgers" for i in [2..4] unless false

<p>I want 2 hamburgers</p>
<p>I want 3 hamburgers</p>
<p>I want 4 hamburgers</p>
```

If you don't see how this works, then go do the next lesson in your CoffeeScript class. We'll be waiting here until you get back.

Tag Function Conjunction

Let's step back and look at how tag functions work with different types of arguments.

There are three types of arguments that can be passed to a tag function. They are an object (hash), a function, and simple types like strings, numbers, true, false, etc. You should know by now what each one of these does ⋯

- **object**: Any object that is an argument of a tag function specifies the attributes for that tag.
- **function**: Executes code that adds HTML to the output buffer. The tag function adds its text like <script> to the output buffer, then runs the function, and then adds its closing text like </script> afterwards. The HTML that function argument adds to the output buffer is nested inside the begin/end tags, as inner HTML. So the nesting of tag functions creates the resulting HTML nesting.
- **string and friends**: These are all converted to strings and directly added to the output buffer. You should know that, by default, HTML entity characters are not escaped. See the Home-made Html section.

You might be wondering what the remaining type of javascript variable, the `array`, does. It is treated exactly like an `object`, which happens to create useless attributes ⋯

```
div ['a','b']

<div 0="a" 1="b"></div>
```

Maybe some smart person will figure out a cool use for arrays in CoffeeKup.

Cool Running

So great, we have this CoffeeScript that executes and produces our html. But how do we actually get this to happen in our app? It's not going to happen by itself.

First we need to get the CoffeeKup module loaded. CoffeeKup (actually CoffeeScript) compiles to vanilla JavaScript so it can run anywhere JavaScript is available. I've only run it in Node and the Browser so let's consider those environments. Note that the same CoffeeKup JavaScript file runs without change in either environment thanks to some fancy footwork.

I'll assume you know how to install CoffeeKup. You should know how to install modules in node using `npm` and/or in the browser using the `<script>` tag. Then include it in your app ⋯

```
# in node
coffeekup = require 'coffeekup'

# in the browser
coffeekup = window.CoffeeKup
```

The `coffeekup` namespace object you just created has the functions you need to run and a lot of other usefull stuff as properties. Of course you can use any name for the namespace object, but we'll stick with `coffeekup` here.

Next, we need to include our CoffeeKup template. This is just a function assigned to a var. Let's use a new simpler hello world example ⋯

```
helloTemplate = ->
    div style:'font-size:96px', 'Hello World'
```

Even people my age are going to be able to read that.

Now we need to execute our template function using a special function, `coffeekup.render`. This renders the desired html.

```
helloHtml = coffeekup.render helloTemplate
```

That was easy. I'm sure you can figure out how to use the html in `helloHtml`. In node you do something like `result.write helloHtml` and in the browser you can stick it in the dom (where the sun never shines), `$('body').append helloHtml`. This assumes jQuery is present. If you don't have jQuery then don't look to me for help. I learned jQuery at the same time I learned JavaScript so I'm useless without it.

So putting it all together (in the browser) ⋯

```
coffeekup = window.CoffeeKup
helloTemplate = ->
    div style:'font-size:96px', 'Hello World'
helloHtml = coffeekup.render helloTemplate
$('body').append helloHtml
```

Or if you are maniac who likes unreadable source files ⋯

```
coffeekup = window.CoffeeKup
$('body').append coffeekup.render ->
    div style:'font-size:96px', 'Hello World'
```

This tiny code will display those giant words. You might wonder though, how can we use the `div` function when it was never defined? Surely it isn't defined as a global by CoffeeKup? That would be uncool, and to be proper coding style it would need to be `coffeekup.div`, which kind of defeats the purpose of CoffeeKup. It also can't be a local which would have required an `eval` somewhere.

The answer is that `div` is inside a function that is only defined and not executed before it is passed as an argument to `coffeekup.render`. From there, `render` "compiles" the function. It does this by using the wonderful `toString()` function to get the original source code. Then it adds the magic code to the beginning and end of the source, as described in the first section above. And finally it turns it back into a function using `new Function srcCode`. Now we have a function that includes the definition for `div` so the problem is solved. Note that this "compiling" turns a function into another function, which is why I put the quotes around the word "compile". I'll leave them out to save typing from here on.

`coffeekup.render helloTemplate` calls `coffeekup.compile helloTemplate` to produce this tricked-out function. Then simply executing the new compiled function renders the html.

You can do `compiledFunc = coffeekup.compile helloTemplate` yourself and keep the compiled template function around for speedy rendering. Later, just execute `compiledFunc()` to get the coveted html. You can also just let `coffeekup.render` do this for you. By default, `render` keeps a copy of each compiled template in a cache and uses the compiled version when available. I've always used this option.

Keeping Things In Context

When I first used CoffeeKup I tried code like this ⋯

```
fontSize = 96
helloTemplate = ->
    div style:"font-size:#{fontSize}px", 'Hello World'
```

Feels quite natural, right? Well all I got for my trouble was an exception saying `fontSize` was undefined.

This is another dirty little secret of CoffeeKup. If you are an advanced JavaScript programmer then you might have noticed that the compile operation covered in the last section destroys all closures for the helloTemplate function. Changing it to source and back to a function tends to do that. So later when the compiled template function ran, the var `fontSize` was not in scope. What to do, what to do.

The complete signature for `coffeekup.render` is `coffeekup.render template, options`. The options argument can contain a lot of properties, but the one we need here is a hash `options.locals`. Every key, value pair in the `options.locals` object is turned into a local var in the compile process.

The source code `key = value` is added with the magic code before your template code. This creates locals to the compiled template function, hence the name. Now I can do ⋯

```
fontSize = 96
helloTemplate = ->
    div style:"font-size:#{fontSize}px", 'Hello World'

coffeekup.render helloTemplate, locals: {fontSize}
```

This passes fontSize into the compile operation making it avalable as a local and now my code works. I tend to use the CoffeeScript shortcut `{fontSize, a, b, c}` a lot. This creates `{fontSize:fontSize, a:a, b:b, c:c}`. So now `fontSize`, `a`, `b`, and `c` are "passed in" to the template to be available as a local.

Time for another wrench in the works. What happens if you change the value of fontSize between the time you compile the template and the time you render it? The local value in the template doesn't change. The original value was "baked" in to the source code of the compiled template. So you have to think of the `options.locals` values as constants. This is especially a problem if you compile once and render multiple times, which the `render` function will do by default if you call `render` more than once with the same source template function.

Another option, `dynamic_locals`, comes to the rescue. Setting `options.dynamic_locals` to true causes all the locals passed in through `options.locals` to be able to change between uses of the compiled template. I have personally not used this feature for two reasons. One is that I've never used a compiled template more than once (duh). But a more serious reason is that `dynamic_locals` works its magic by enclosing all the template code in a JavaScript `with` statement. I'm sure you've heard the experts whine about how evil the `with` statement is. Well, even if you disagree with them (as I do) then you should still consider that the upcoming `strict` context will not allow any `with` statement at all. It might be nice to use strict in the future.

I should also mention that you can define another option object, called `options.context`, that makes locals available to the template like `options.locals` does. This is passed in as the context to the compiled function so the values are available on the `this` object, or `@` as we CoffeeScript nuts know it. So `@fontSize` could be used. Some might consider this safer from a namespace standpoint and/or more readable as it makes the passed-in locals stand out.

The Option To Use Options

Let's cover all options here in this one convenient section. Remember that the signature for `coffeekup.render` is `coffeekup.render template, options`. Here is the list of all available options as of this writing ⋯

- `options.locals`: An object containing key/value pairs to be passed in to the template as constants. See the last section.
- `options.dynamic_locals`: If `true` then `options.locals` are made dynamic by using the JavaScript `with` statement. See the last section.
- `options.context`: Passed in to the template as the context object, aka `this` or `@`. See the last section.

- `options.cache`: If `true`, then `render` keeps all the compiled templates around and skips the compile step when that same template needs rendering. The default is `true`.
- `options.format`: If `true`, then returns and indentation are added to the compiled source to make it "pretty". The default is `false`.
- `options.autoescape`: If `true` then any html entities are escaped in the rendered template. i.e. & is changed to `&`, < to `<`, etc. The default is `false`.

Homemade Html

I mentioned earlier in the section Lonely Text that text added with CoffeeCup can be plain html that is passed through without being treated as CoffeeKup code. It is as easy as saying …

```
div "<div>I'm a homemade div in a div</div>"
text "<div>I'm an orphan div with no parent div</div>"
```

In order to do this, you must make sure the `autoescape` option is `off` (`false`). Otherwise you will be surprised, as I was, to see the html code in your web page.

What the heck are Helpers, Express, Zappa, and Meryl?

No, seriously, I don't know what these are. (I do know who Zappa was. He was one of my favorites in the good old days). Someone needs to explain them to me as I am too lazy to Google them. Better yet, please issue a Pull Request to add explanations of them to this document.

Where To Go From Here

I'm sure I've missed some topics other than just helpers. Leave an issue on this github if you think of any. Meanwhile, to keep up on the latest CoffeeKup developments, check in on CoffeeKup's github page. As I said before, currently the only discussion of CoffeeKup is on CoffeeKup's issues page. Someone should create a Google Group for CoffeeKup.

Credit Where Credit Is Due

Of course the most credit goes to Maurice Machado, aka mauricemach who wrote CoffeeKup. Maurice (and we) are indebted to Tim Fletcher and Why The Lucky Stiff who wrote Markaby ("Markup as Ruby"), the predecessor and inspiration for CoffeeKup. Thanks to Loren Sands-Ramshaw for a clean-up pass. You can add yourself to this list by helping with this document. All it takes is a spelling correction. My biggest spelling correction came from Maurice, who pointed out that I spelled CoffeeKup as KoffeeKup everywhere. It figures that I'd mispell the thing I'm writing about.