Nathan Cochran
Web Development – How To Guide
8/12/2013

Intro:

Do you feel like your HTML files are cluttered and hard to read, especially once you add all that JavaScript? Do you find yourself muddling through JavaScript's syntax? Does forgetting that curly brace cause you hours of trouble, and significantly increase your rate of hair loss? Do you want the power of JavaScript, but not all the syntactic baggage? Well look no farther! I hereby introduce: CoffeeScript.

Perhaps you've heard of CoffeeScript. After all, Dropbox rewrote their client-side code in CoffeeScript just last September. Or maybe when you were browsing the github style guide for JavaScript (because how are you supposed to format all that messy JavaScript, anyways?), you noticed that the very first line says: “write new JS in CoffeeScript.” Or perhaps you noticed that CoffeeScript is now the tenth most popular language on github. But what exactly *is* CoffeeScript?

The CoffeeScript website declares that the golden rule of CoffeeScript is: “*It's just JavaScript.*” But let's stop going in circles and get to the point. CoffeeScript is actually a language (or, more accurately, a layer of syntactic sugar for JavaScript) that compiles, predictably and readably, into JavaScript. It's an “attempt to expose the good parts of JavaScript in a simple way.” You might be thinking: I just learned JavaScript – why should I learn another language that's really just JavaScript all over again? Well, maybe because CoffeeScript programs can typically be written in 2/3 the lines compared to their equivalent JavaScript counterparts (and are easier to read and understand, to boot). Or maybe because CoffeeScript can help prevent errors that non-expert JavaScript programmers might easily make (like accidental leaked globals). Or maybe because learning CoffeeScript doesn't really mean learning another language at all, since CoffeeScript *is just JavaScript* anyways!

So now I bet you're ready for an example, a little taste of what you can look forward to. Remember that “function()” keyword that kept popping up all over the place in our JavaScript code? For example, in the code that specifies the function that should execute when the DOM is loaded:

```
3 $(document).ready(function() {
4
5 });
```

Well, check this out:

```
1 ($document).ready ->
```

What? Where's the “function” keyword? Or that annoying little “ }); ” thing at the end, one of whose three characters you inevitably forget? Well, um, they're gone! That's right – *gone*. Welcome to CoffeeScript.

Note About This Guide:

While the little differences in syntax between JavaScript and CoffeeScript are certainly interesting and exciting, this guide will not be primarily focused on explaining and demonstrating them. It will inevitably cover some of the differences, but it will hardly scratch the surface. The reasons for this are threefold: first, the differences in syntax are already well documented on the CoffeeScript website; second, it would be too much to cover in a guide of this size; and third, CoffeeScript is so closely related to JavaScript that it really doesn't take much effort to start to get the hang of it. Instead, this guide will be focused primarily on installing, using, and understanding the CoffeeScript compiler, setting up your files to efficiently and effectively make us of CoffeeScript's functionality, and exposing some common problems and unexpected consequences of using CoffeeScript.

Installation:

So how do we get this thing up and running? The command line version of CoffeeScript, which is what we want, is a Node.js utility (you can do some fancy things if you want to use the CoffeeScript compiler without installing Node.js, but it's a bit more difficult – see here). This means that it can be installed with npm – the Node Package Manager – which comes with all versions of Node.js since version 0.6.3. But what is Node.js, and why do we need it?

Node.js is a software platform that makes it possible to build an entire web application – both server-side and client-side – in JavaScript. In other words, it makes it possible to not only use JavaScript as a client-side language, as it is commonly used, but also as a server-side language (I know – crazy, right?) This is not directly relevant to our purposes here, but if you want to learn more, you should check out the wikipedia page or the Node.js website. The important point, for us, is that to be able to use JavaScript as a server-side language, Node.js needs to be able to run JavaScript outside of a browser. This is made possible by the inclusion of Google's open source V8 JavaScript Engine (the same JavaScript engine used in Google Chrome) in the package compilation that is Node.js.

So what's the point? Well, the CoffeeScript compiler is itself written in CoffeeScript, which compiles to JavaScript. So if we want to compile our CoffeeScript code outside of a web browser, we

need a way to run JavaScript outside of a web browser. And, as I've just explained, that's exactly what Node.js does (the compiler does, indeed, run in any JavaScript environment, including a web browser – see the "Try CoffeeScript" option at the top of the CoffeeScript website).

We can download Node.js from the Node.js download page. There are several different options, so if you want to, you can go ahead and download the installer or binaries for your system. If you choose one of those options, feel free to skip ahead to the "Getting Started" section. Otherwise, I'll walk you through compiling Node.js from the source code. I should note that I'm running Linux Mint 14, a derivative of Ubuntu, and that these instructions may need some tweaking for other, non-Linux operating systems. I'll try to point out where those differences might arise, but you might need to do some exploring on your own.

First, save the node-v0.10.15.tar.gz file into a directory of your choosing. It doesn't really matter where, as it will install to the proper location in any case. I chose to save mine in my ~/Downloads folder, for obvious reasons. Now, navigate to the folder in which you saved the .tar.gz file (using "cd") and extract it with the following command:

```
nathan@nathan ~/Downloads $ tar xvzf node-v0.10.15.tar.gz
```

You should see a bunch of files being created in a folder named: node-v0.10.15/ (or something like this, depending on which version you're installing). When it's finished extracting, you should see the newly created folder in your current directory – the directory in which you extracted the .tar.gz file.

Navigate into the new folder and look around (using the "ls" command). Two files should catch your eye: a README.md file and a Makefile. The README contains instructions for how to install the source code on different operating systems, as well as a list of resources for newcomers. From this point, installation is only takes three commands for Unix/Mac, and a single command for windows. Since I'm running Linux, I'll run the following commands (Max users do the same):

```
nathan@nathan ~/Downloads/node-v0.10.15 $ ./configure
nathan@nathan ~/Downloads/node-v0.10.15 $ make
nathan@nathan ~/Downloads/node-v0.10.15 $ sudo make install
```

Don't forget the "sudo" or "su" command – this is essential, since you'll need root permissions to write to your computer's usr/local/ directory, where the files are installed. Furthermore, in order to ensure that all the unnecessary temporary files created during the install get removed, you probably also want to enter:

```
nathan@nathan ~/Downloads/node-v0.10.15 $ make clean
```

For Windows users, the README says to run (Note: I haven't tested this one out):

vcbuild.bat

And that's it! Assuming nothing went horribly wrong, Node.js is now installed, which means we have access to npm – the Node Package Manager. Only one more step left, and we'll have CoffeeScript installed and ready to use. Just enter the command:

```
nathan@nathan ~/Downloads/node-v0.10.15 $ sudo npm -g coffee-script
```

To make sure it worked, try the following command (you should see output like mine):

```
nathan@nathan ~/Downloads/node-v0.10.15 $ which coffee
/usr/local/bin/coffee
```

Alternatively, check which CoffeeScript version you have installed:

```
nathan@nathan ~/Downloads/node-v0.10.15 $ coffee -v
CoffeeScript version 1.6.3
```

Getting Started

You now have the "coffee" command-line tool available to you. Use the help (-h) option to see what it can do:

```
nathan@nathan ~/Downloads/node-v0.10.15 $ coffee -h

Usage: coffee [options] path/to/script.coffee -- [args]

If called without options, `coffee` will run your script.

  -b, --bare         compile without a top-level function wrapper
  -c, --compile      compile to JavaScript and save as .js files
  -e, --eval         pass a string from the command line as input
  -h, --help         display this help message
  -i, --interactive  run an interactive CoffeeScript REPL
  -j, --join         concatenate the source CoffeeScript before compiling
  -m, --map          generate source map and save as .map files
  -n, --nodes        print out the parse tree that the parser produces
      --nodejs       pass options directly to the "node" binary
  -o, --output       set the output directory for compiled JavaScript
  -p, --print        print out the compiled JavaScript
  -s, --stdio        listen for and compile scripts over stdio
  -l, --literate     treat stdio as literate style coffee-script
  -t, --tokens       print out the tokens that the lexer/rewriter produce
  -v, --version      display the version number
  -w, --watch        watch scripts for changes and rerun commands
```

There are a few things worth paying attention to, right off the bat. First, if you pass a .coffee file to the coffee command with the -c option, it will compile the .coffee file into a .js file with the same name. Let's go ahead and create *very* simple hello world program and compile it with the -c option. First, create a file with the .coffee extension using the text editor of your choice:

```
nathan@nathan ~ $ vim example.coffee
```

Enter the following line of code:

```
1
2 alert "Hello World!"
3
```

Now, let's return to the command line and try it out:

```
nathan@nathan ~ $ coffee -c example.coffee
```

We should now have the file example.js in our current working directory. Let's open it up and take a look:

```
1 // Generated by CoffeeScript 1.6.3
2 (function() {
3   alert("Hello World!");
4
5 }).call(this);
```

One thing you'll notice is that the compilation process wrapped our code in an anonymous function call. This is to prevent global variables from being unintentionally created. This is almost always a good thing. However, if you have a very good reason for not wanting your code wrapped in this way, you can compile with the --bare (-b) option, like so:

```
nathan@nathan ~ $ coffee -c -b example.coffee
```

Our output file now looks more like we might have expected it to:

```
1 // Generated by CoffeeScript 1.6.3
2 alert("Hello World!");
3
```

Getting Into Some Examples:

Using the bare (-b) option like I've just shown you is strongly discouraged in most circumstances. It is undoubtedly tempting, though – especially for programmers new to CoffeeScript. Let's consider an example circumstance in which one might be tempted to use the bare (-b) option, and then I'll demonstrate the more acceptable way to structure the same code. We'll run into a few snags along the way, and, as a result, learn a little more about what CoffeeScript can and cannot do.

For this example, we're going to create an actual HTML file with a functional click handler. Let's call it example2.html:

```
 1 <!DOCTYPE html>
 2 
 3 <html xmlns="http://www.w3.org/1999/xhtml">
 4 
 5 <head>
 6     <meta charset="UTF-8">
 7     <title>Example</title>
 8 </head>
 9 <body>
10 
11     <div onclick='click_handler()'>Click Here! </div>
12 
13     <script type="text/javascript">
14         click_handler = () -> alert "You clicked it!"
15     </script>
16 
17 </body>
18 </html>
```

Pretty simple, right? The JavaScript function I normally would have used as a click handler:

```
click_handler = function() {
  alert("You clicked it!");
};
```

has been replaced by the sleeker CoffeeScript function:

```
click_handler = () -> alert "You clicked it!"
```

Now let's try to compile it:

```
nathan@nathan ~ $ coffee -c example2.html
example2.html:1:1: error: unexpected COMPARE
<!DOCTYPE html>
^
```

What happened!?! Unexpected COMPARE? But it's not a less-than sign, it's an HTML tag!

Unfortunately, we've hit our first snag. The fact of the matter is: the CoffeeScript compiler can't compile CoffeeScript that's embedded in HTML. That's right, folks. If you've gotten comfortable writing big intermingled messes of HTML and JavaScript, and you want to use the CoffeeScript compiler, you're going to need to get in the habit of separating out your CoffeeScript from your HTML (there is a caveat, but it's highly discouraged by virtually everyone. If you're interested anyways, see here). If it's any consolation, separating your HTML from your CoffeeScript (or your JavaScript, for that matter) is a very good habit to get into. So let's do just that.

Our new HTML file (example3.html) looks like this:

```
1 <!DOCTYPE html>
2
3 <html xmlns="http://www.w3.org/1999/xhtml">
4
5 <head>
6     <meta charset="UTF-8">
7     <title>Example</title>
8     <script src="example3.js" type="text/javascript"></script>
9 </head>
10 <body>
11
12     <div onclick='click_handler()'>Click Here! </div>
13
14 </body>
15 </html>
```

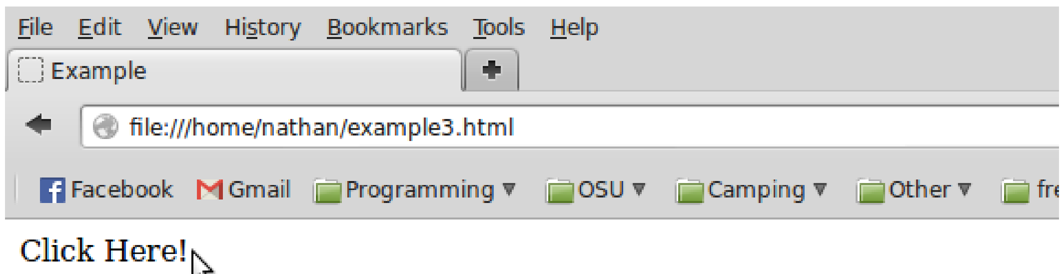
Notice that I've stripped the CoffeeScript click handler from the file, but included a <script> tag in the header that links to example3.js. This is the file that our CoffeeScript file will compile to. Speaking of which, our new CoffeeScript file (example3.coffee) looks like this:

```
2 click_handler = () -> alert "You clicked it!"
```

So let's go ahead and compile it:

```
nathan@nathan ~ $ coffee -c example3.coffee
nathan@nathan ~ $ █
```

Hey! It looks like it actually worked this time! Let's open up our example3.html file and try it out:



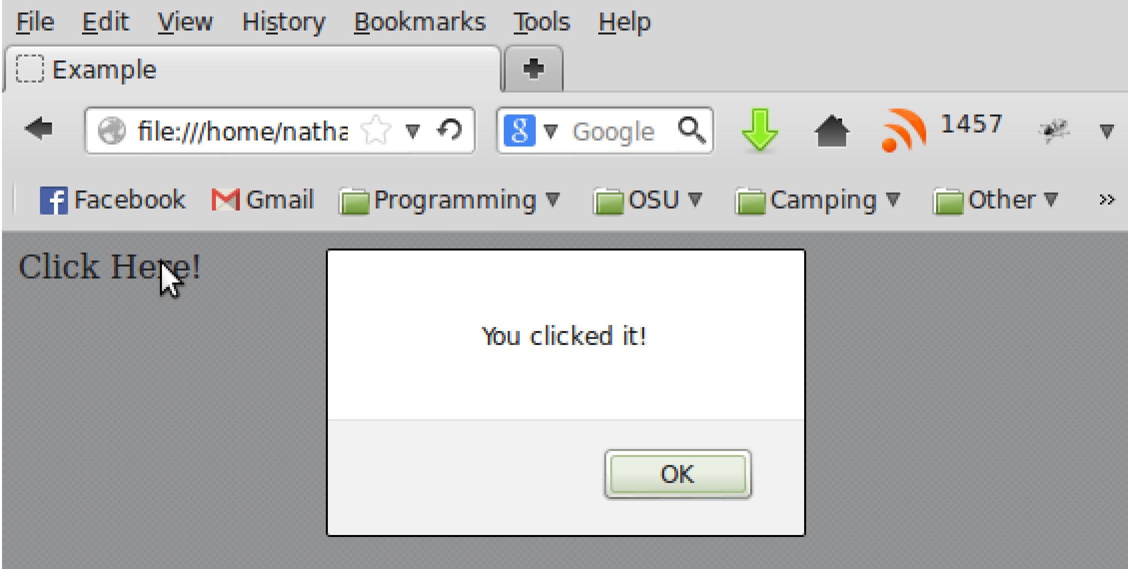
But wait a second... What's going on!? I'm clicking it, but I'm not getting any alerts! Let's go back and examine the example3.js file that the compiler created:

```
1 // Generated by CoffeeScript 1.6.3
2 (function() {
3   var click_handler;
4 
5   click_handler = function() {
6     return alert("You clicked it!");
7   };
8 
9 }).call(this);
```

As you can see, our code is in the anonymous function wrapper mentioned earlier, which is limiting the scope in which our click handler is visible. So that's why it isn't working. And maybe now you're finally starting to get tempted by the bare (-b) option I mentioned earlier. So what the heck, let's try it out:

```
nathan@nathan ~ $ coffee -c -b example3.coffee
```

And now when we test out our example3.html page:



It works! But we've violated the rule and used the bare (-b) option without a really really good reason. As a result, our click_handler() function is an exposed global variable, and that's not really a good thing. So let's see if we can do better. At this point, we'll throw in a little jQuery too, just to make

things a little easier (and to show off the awesome power that is CoffeeScript + jQuery).

Our last and final HTML file, example4.html, looks like this:

```
1 <!DOCTYPE html>
2
3 <html xmlns="http://www.w3.org/1999/xhtml">
4
5 <head>
6     <meta charset="UTF-8">
7     <title>Example</title>
8     <script src="jquery-1.10.2.min.js" type="text/javascript"></script>
9     <script src="example4.js" type="text/javascript"></script>
10 </head>
11 <body>
12     <div class="click_me">Click Here! </div>
13
14 </body>
15 </html>
```

As you can see, we're removed the old-school click_handler (onclick = 'click_handler()') and replaced it with the div class “click_me”. Furthermore, we've added a script tag in the header linking to the jQuery source file, which I'v stored in the same directory in which we're working.

So what's our CoffeeScript file look like now? Well, remember at the beginning, when I told you that you could replace:

```
3 $(document).ready(function() {
4
5 });
```

with:

```
1 ($document).ready ->
```

Well, that wasn't the whole truth. You can actually go so far as to replace the entire thing with:

```
$ ->
```

No, I didn't just make a mistake. That's really the page ready function (I didn't want to *completely* blow your mind at the start of the tutorial. And besides, I had to leave something good for those of you willing to stick it out through this entire tutorial).

So anyways, our final CoffeeScript file, example4.coffee, looks like this:

```
$ ->
    click_handler = () -> alert "You clicked it!"
    $(".click_me").click(click_handler)
```

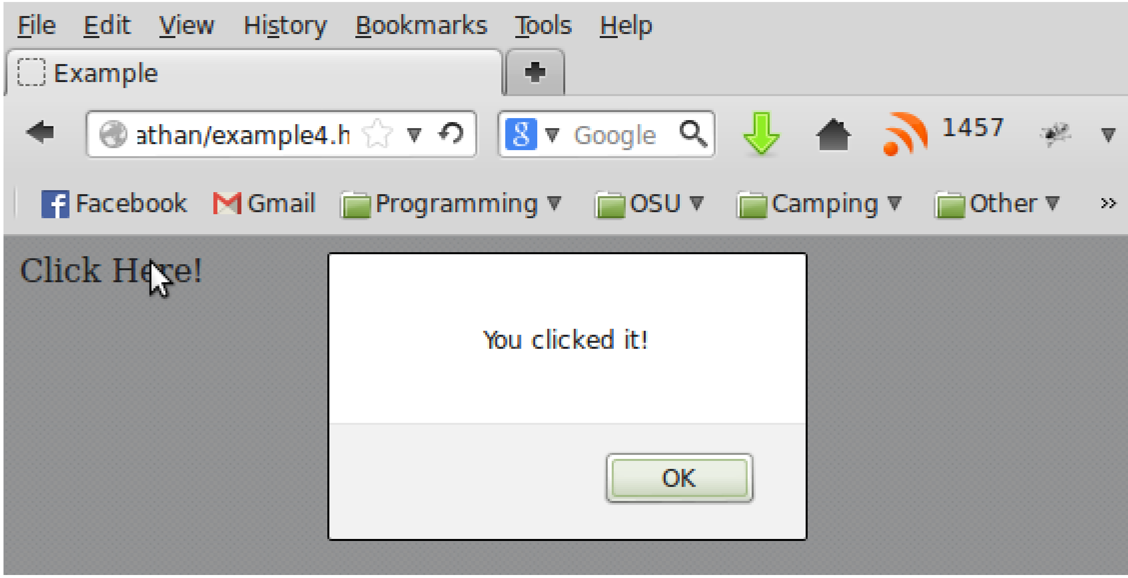
What exactly does this do? Well, when the page loads, it defines the click_handler function, then uses jQuery to attach that function to the “click_me” class of div tags (of which there is only one) as the function to be called on a click event. Actually, if you wanted to get really sleek and slim, you could do the same thing as a one-liner:

```
$ ->
  $(".click_me").click(-> alert "You clicked it!")
```

Now let's compile it:

```
nathan@nathan ~ $ coffee -c example4.coffee
```

Note that we *didn't use the bare (-b) option this time!* And when we test it out:



It works! Believe it or not. And finally, as a kind of parting message, let's see what that last CoffeeScript file (the one-liner) compiled to, just for curiosity's sake. I think the difference between the two will speak for itself:

```
1 // Generated by CoffeeScript 1.6.3
2 (function() {
3   $(function() {
4     return $(".click_me").click(function() {
5       return alert("You clicked it!");
6     });
7   });
8
9 }).call(this);
```