Functional JavaScript with CoffeeScript and Node

Functional scripting masters web application complexity

Andrew Glover (ajglover@gmail.com)
CTO
App47

21 February 2012

CoffeeScript is famous for smoothing out JavaScript's rough edges, but it has other advantages worth exploring. In this article, Andrew Glover shows you how CoffeeScript's cleaner syntax makes it easer to leverage functional constructs in JavaScript libraries, especially for server-side programming in Node.js. He concludes with a series of short demonstrations in using Underscore.js, a utility library for JavaScript, to handle collections in CoffeeScript and Node.

CoffeeScript is a relatively new language that presents an appealing alternative for developers weary of JavaScript's deficiencies. With CoffeeScript, developers can do their coding in a lightweight, intuitive language that feels like a hybrid of Ruby and Python. CoffeeScript then compiles to JavaScript for browser-compatible Web applications, and it also works seamlessly with Node.js for server-side applications. Central to this article is a third benefit of using CoffeeScript, which is its handling of the *functional* side of JavaScript. CoffeeScript's cleaned up, modernized syntax opens up the world of functional programming latent in JavaScript libraries.

Functional programming in the mainstream

While none of the mainstream programming languages (such as the Java™ language, C++, or C#) is explicitly a functional programming language, add-on libraries and frameworks do enable varying levels of functional programming in any of these languages. More important, languages like Clojure, F#, and Erlang are becoming mainstream precisely because functional programming promises to produce fewer bugs and increased productivity in complex applications.

Similar to JavaScript, functional programming is both exceptionally useful and historically unpopular. Whereas JavaScript was deemed a toy language, functional programming developed a reputation for being overly complex. But as the demand for highly concurrent applications has increased, so has the need for an alternative to state-changing imperative programming styles. Functional programming, as it turns out, isn't so much needlessly complex as it is an elegant tool for codifying the complexity inherent in certain types of applications.

In this article, we'll explore functional scripting in CoffeeScript and Node, with the help of a JavaScript library called Underscore. When combined, these three technologies make a powerful

stack for using JavaScript to develop server-side and browser-based apps that leverage functional programming.

Note that this article builds on my previous introductions to JavaScript for Java developers and Node.js for Java developers. I assume that your development environment includes Node.js and that you are familiar with basic programming in Node.

Set up CoffeeScript and Node

If Node.js is already part of your development environment, then you can use its package manager (NPM) to install CoffeeScript. The following command instructs NPM to install the package globally:

```
$> npm install -g coffee-script
```

Most of your time in CoffeeScript will be spent writing programs, saving them to a .coffee file, then compiling the results to JavaScript. CoffeeScript's syntax is close enough to JavaScript's that it should be familiar to most developers; for instance the CoffeeScript in Listing 1 looks much like JavaScript, but without the usual clutter of parentheses and semicolons:

Listing 1. Typical CoffeeScript

```
$> coffee -bpe "console.log 'hello coffee'"
console.log('hello coffee');
```

The `coffee` command is a shortcut for some management tasks. It compiles CoffeeScript files into JavaScript, runs CoffeeScript files, and even acts as an interactive environment or REPL (similar to Ruby's `irb`).

Here, I put my script into a file:

```
console.log "hello coffee"
```

Then I compile (or convert) that file to JavaScript:

```
$> coffee -c hello.coffee
```

The result is a file dubbed hello.js. Because the resulting JavaScript is valid for Node, I can quickly run it in my Node environment:

Listing 2. JavaScript in Node

```
$> node hello.js
hello coffee!
```

Alternately, I could use the `coffee` command to run the original .coffee file, as in Listing 3:

Listing 3. CoffeeScript in Node

```
$> coffee hello.coffee
hello coffee!
```

Mind the watchr

The open source community has produced a number of handy file watcher utilities that do things like run tests, compile code, and so on. These generally work via the command-line and are very lightweight. We'll configure a watcher tool to watch all the .coffee files in our development environment and compile them to .js upon being saved.

The utility I like for this purpose is `watchr`, which is a Ruby library. In order to use `watchr`, you'll need to have Ruby and RubyGems installed in your development environment. Once you've got that set up, you can run the following command to install `watchr` as a global Ruby library (including the corresponding utility):

```
$> gem install watchr
```

In `watchr`, you use regular expressions to define the files to be watched and what should happen to them. The following command configures `watchr` to compile all .coffee files found in a `src` directory:

```
watch('src\/.*\.coffee') {|match| system "coffee --compile --output js/ src/"}
```

Note that the `coffee` command in this case will put the resulting .js files into a `js` directory.

I can fire this thing up in a terminal window, like so:

```
$> watchr project.watchr
```

Now, whenever I make a change to any .coffee file in my `src` directory, `watchr` will ensure that a new `.js` file is created and placed in my `js` directory.

CoffeeScript at 10,000 feet

CoffeeScript introduces many valuable features that make it much easier to work with than JavaScript. At a high level, CoffeeScript obviates the need for curly braces, semi-colons, the `var` keyword, and the `function` keyword. In fact, one of my favorite CoffeeScript features is its *function* definition, shown in Listing 4:

Listing 4. CoffeeScript functions are easy!

```
capitalize = (word) ->
  word.charAt(0).toUpperCase() + word.slice 1

console.log capitalize "andy" //prints Andy
```

Here I've declared a simple function to capitalize the first letter of a word in CoffeeScript. In CoffeeScript, the syntax of a function definition follows an arrow. The body is also delimited by spacing; that's why there are no curly braces in CoffeeScript. Also notice the absence of parentheses. CoffeeScript's `word.slice 1` simply compiles to JavaScript's `word.slice(1)`. Similarly, note that the function's body is delimited by spacing: it's all the code indented under the function-definition line. The un-indented `console.log` beneath that signifies that the

method's definition is complete. (These two features come to CoffeeScript via Ruby and Python, respectively.)

If you're wondering, the corresponding JavaScript function would look something like Listing 5:

Listing 5. Even a one-liner in JavaScript is noisy

```
var capitalize = function(word) {
  return word.charAt(0).toUpperCase() + word.slice(1);
};

console.log(capitalize("andy"));
```

Variables

CoffeeScript automatically places a JavaScript-*ian* `var` in front of any variable you define. As a result, you never need to remember the `var` when coding in CoffeeScript. (The `var` keyword is optional in JavaScript. Without it, your variable becomes global, which is almost always a bad thing.)

CoffeeScript also lets you define default values for parameters, shown in Listing 6:

Listing 6. Default parameter values!

```
greeting = (recipient = "world") ->
  "Hello #{recipient}"

console.log greeting "Andy" //prints Hello Andy
console.log greeting()      //prints Hello world
```

Listing 7 shows how this default parameter value is handled in the corresponding JavaScript:

Listing 7. Noisy JavaScript

```
var greeting;

greeting = function(recipient) {
 if (recipient == null) recipient = "world";
 return "Hello " + recipient;
};
```

Conditionals

CoffeeScript handles conditionals by introducing keywords like `and`, `or`, and `not`, as shown in Listing 8:

Listing 8. CoffeeScript conditionals

```
capitalize = (word) ->
 if word? and typeof(word) is 'string'
  word.charAt(0).toUpperCase() + word.slice 1
 else
  word

console.log capitalize "andy"   //prints Andy
console.log capitalize null     //prints null
console.log capitalize 2        //prints 2
console.log capitalize "betty"  //prints Betty
```

In Listing 8, I used the `?` operator to test for existence. Before attempting to capitalize a word, this script would ensure that the parameter `word` wasn't `null`, and that it was a `string` type. It's pretty nifty that CoffeeScript allows you to use `is` in place of `==`.

Class definition for functional programming

JavaScript doesn't support classes directly; rather, it is a prototype-oriented language. For those of us steeped in object-oriented programming, this can be confusing — we want our classes! To keep us happy, CoffeeScript provides a `class` syntax, which yields a series of functions defined within functions when compiled into standard JavaScript.

In Listing 9, I use the `class` keyword to define a class called `Message`:

Listing 9. Yeah, CoffeeScript is classy

```
class Message
 constructor: (@to, @from, @message) ->

 asJSON:  ->
  JSON.stringify({to: @to, from: @from, message: @message})

mess = new Message "Andy", "Joe", "Go to the party!"
console.log mess.asJSON()
```

In Listing 9, I used the `constructor` keyword to define a constructor. I then defined a method (`asJSON`) by typing a name followed by a function.

CoffeeScript and Node

Because it compiles to JavaScript, CoffeeScript is a natural fit for programming in Node and can be especially helpful for making Node's already clean code even neater. CoffeeScript is particularly adept at smoothing out Node's many callbacks, as you can see with a simple code comparison. In Listing 10, I define a simple Node web app using pure JavaScript:

Listing 10. A Node.js web app in JavaScript

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

var app = express.createServer(express.logger());

app.put('/', function(req, res) {
  res.send(JSON.stringify({ status: "success" }));
});

var port = process.env.PORT || 3000;

app.listen(port, function() {
  console.log("Listening on " + port);
});
```

Rewriting the same web app in CoffeeScript wipes out the syntactic noise of Node callbacks, as Listing 11 shows:

Listing 11. CoffeeScript simplifies Node.js

```
express = require 'express'

app = express.createServer express.logger()

app.put '/', (req, res) ->
  res.send JSON.stringify { status: "success" }

port = process.env.PORT or 3000

app.listen port, ->
  console.log "Listening on " + port
```

In Listing 11, I added an `or` operator in place of the JavaScript `||`. I also found that using an arrow to indicate the anonymous function in `app.listen` was easier to type than `function()`.

CoffeeScript talks to me

You've probably realized by now that CoffeeScript prefers everyday English to abstract symbols. Rather than type `!==`, CoffeeScript has us use the more intuitive `isnt`; likewise, `===` becomes `is`.

If you issue a `coffee -c` on this file, you'll see that CoffeeScript produces almost the exact JavaScript defined in Listing 10. CoffeeScript's 100% valid JavaScript works with any JavaScript library.

Functional collections with Underscore

Billed as a functional utility belt for JavaScript programming, Underscore.js is a library of functions for making JavaScript development easier. Among other things, Underscore offers a bevy of collection-oriented functions, each perfectly suited to its particular task.

For instance, suppose you needed to find all the odd numbers in a collection of numbers; say 0 through 10, exclusive. While you could hack it out, leveraging CoffeeScript and Underscore together would save you some typing, and probably some bugs. In Listing 12, I provide the basic algorithm, while Underscore provides the aggregation function, in this case `filter`:

Listing 12. Underscore's filter function

```
_ = require 'underscore'

numbers = _.range(10)

odds = _(numbers).filter (x) ->
 x % 2 isnt 0

console.log odds
```

First off, because `_` (that is, *underscore*) is a valid variable name, I set that to reference the Underscore library. Next, I attached an anonymous function to the `filter` function that tests for odd numbers. Note that I used the CoffeeScript `isnt` keyword rather than JavaScript's `!==`. I then used the `range` function to specify that I wanted to sort the numbers 0 through 9. Alternately, I could have provided a step count for my range (that is, counting by two) and started from any number.

The `filter` function returns an array that is a filtered version of whatever was passed into it, which in this case was `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. So running the code in Listing 12 would yield `[ 1, 3, 5, 7, 9 ]`.

The `map` function is another one of my favorites to apply to collections in JavaScript, as shown in Listing 13:

Listing 13. Underscore's map function

```
oneUp = _(numbers).map (x) ->
 x + 1

console.log oneUp
```

Here, the output would be `[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]`. Basically, Underscore increments each value in my `numbers` range by one, so I don't need to iterate over reach integer manually.

If you need to test aspects of a collection, Underscore makes it so simple! Just create a function like the one in Listing 14, which tests for evenness:

Listing 14. Underscore's even function

```
even = (x) ->
 x % 2 is 0

console.log _(numbers).all(even)
console.log _(numbers).any(even)
```

Once I've defined my `even` function, I can easily attach it to Underscore functions like `all` and `any`. In this case, `all` applies my `even` function to each value in the `numbers` range. It then returns a boolean indicating whether *all* values were even (false). Similarly, the `any` function would return a boolean if *any* were even (true).

Do more with Underscore

Here I can only scratch the surface of what Underscore has to offer. Underscore's other tricks include function binding, JavaScript templating, and deep equality testing. (See Resources.)

What if you don't need to apply any of these functions to a collection of values, but need to do something else? No problem! Leverage Underscore's `each` function. `each` acts as an easy iterator (meaning that it handles the looping logic behind the scenes, passing in the specified function upon each iteration). This function should be familiar if you've worked with Ruby or Groovy.

Listing 15. Underscore's each function

```
_.each numbers, (x) ->
 console.log(x)
```

In Listing 15, the `each` function takes a collection (my `numbers` range) and a function to be applied to each value in the iterated array. In this case, I used `each` to print the value of the current iteration

to the console. I could just as easily have done something like persist data to a database, return results to a user, and so on.

In conclusion

CoffeeScript infuses a freshness and ease into JavaScript programming that should make it feel like second nature, especially for anyone familiar with Ruby or Python. In this article, I've showed you how CoffeeScript borrows from each of these languages to make JavaScript-style code easier to read and a lot faster to produce. Combining CoffeeScript, Node, and Underscore, as I've demonstrated, results in an incredibly lightweight and fun development stack for basic functional programming scenarios. With time and practice, you could easily extend what you've learned here for more complex business applications that rely upon dynamic web and mobile interactions.

Resources

Learn

- Read earlier articles in this developerWorks series of three: "Java development 2.0: JavaScript for Java developers" (Andrew Glover, April 2011), and "Node.js for Java developers" (Andrew Glover, November 2011).
- Neal Ford's *Functional thinking* series gets beyond the syntax of functional programming and into the underlying concepts, with examples written in Groovy, Scala, and the Java language.
- If you want more CoffeeScript, see:
  - "Your first cup of CoffeeScript, Part 1: Getting started" (Michael Galpin, developerWorks, December 2011).
  - "The Java technology zone technical podcast series: Ryan McGeary" (Andrew Glover, developerWorks, February 2011).
- For answers to your questions about Node, start with "Just what is Node.js?" (Michael Abernethy, developerWorks, May 2011).
- *Java development 2.0* is Andrew Glover's developerWorks series exploring technologies that are redefining the Java development landscape.
- Browse the Java technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- The application development stack demonstrated in this article includes the following:
  - CoffeeScript
  - Node.js
  - Underscore.js
  - Watchr
  - Ruby
  - Ruby Gems

Discuss

- Get involved in the developerWorks community. Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.

About the author

Andrew Glover

Andrew Glover is a developer, author, speaker, and entrepreneur with a passion for behavior-driven development, Continuous Integration, and Agile software development. He is the founder of the easyb Behavior-Driven Development (BDD) framework and is the co-author of three books: Continuous Integration, Groovy in Action, and Java Testing Patterns. You can keep up with him at his blog and by following him on Twitter.