What & Why of Underscore JS-

*Underscore is a utility-belt library for JavaScript that provides a lot of the functional programming support that you would expect in Prototype.js (or Ruby), but without extending any of the built-in JavaScript objects.*

As you read above, Underscore.js is a nifty little JavaScript library that brings in a ridiculous amount of functionality at a mere 4kb.

If your use of JavaScript is limited to merely playing around with the DOM, then the answer is mostly no since jQuery does most of what you'd want to do.

On the other hand, if you're dealing with non-DOM code or even complex, think MVC, front end code, Underscore is an absolute boon.

Underscore provides you with a nice set of abstractions that work everywhere.

And if you're a performance oriented person, as you should be, Underscore falls back to native implementations, if available, to make sure performance is as optimal as possible.

**JavaScript is missing stuff (and it's weird)**

Standard JavaScript defines a very minimal set of basic library routines or functions.  The current version of JavaScript, sadly, is fairly barebones when it comes to low level utilities. The glaringly obvious omission of being able to iterate with a forEach led that feature to be reinvented countless times...with different syntax. Other functions that *should* be simple--like testing to see if a variable is a string or not--wind up looking strange if done "right":

if (typeof myVar == 'string' || myVar instanceof String)

// it's a string

else

// it's something else

You can look over a StackOverflow discussion about [why this top-voted test is so ugly](http://stackoverflow.com/questions/4059147/check-if-a-variable-is-a-string). There's controversy on whether one needs to be that careful or not, but it's kind of unreasonable to expect new programmers to absorb the ramifications. So this creates a problem that needs to be solved with abstractions that are what you meant: forEach and isString *(or at least something that can reliably tell you type to test against 'string')*.

In the days when JavaScript was only in the browser, library developers would keep reinventing the basic routines like this. Since there wasn't a standard, no one wanted to risk a dependency that their project didn't need. Yet jQuery had its own way of covering these bases, and wound up the de-facto foundation of browser development. So its set of built-ins like $.each() and $.type() spread wide and far into code samples on the web.

**Node (in itself) doesn't address any of this**

Perhaps you heard that Node comes with some really great functionality out of the box. But when people say that, they mean stuff like libraries for streaming http and tcp. They are not talking about basic enhancements to the datatypes.

No...under the hood, Node.JS is just running the standard V8 JavaScript engine that's in Chrome. So you're getting a pretty fast and modern JavaScript, to the extent we can use those words to describe JavaScript. :-P That means all Node installs use ECMAScript 5 *(at least)*, which actually [did add a forEach](http://stackoverflow.com/a/9329476/211160)!

Don't get too excited. Because forEach is really about the only thing they added like that; and they didn't add it because they were all that concerned that people were having to wrap other things. After all--JavaScript is successful enough that people have already shown they'd deal with it. Language purists weren't going to switch from Haskell because testing for strings got cleaner. :-) Plus any "cleanups" would run the risk of breaking backward compatibility.

Note What you actually get in ECMAScript5 are things like ["strict mode"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) *(which you should be using!)* and more functionality for [Objects and Properties](http://ejohn.org/blog/ecmascript-5-objects-and-properties/) *(which you should use if you need it, I dunno)*.

This might make you curious why they would pick out forEach...which is partially because of its importance, but mostly for performance reasons. Wrappers like jQuery's $.each() were having to do too much work under the hood and were limited in their efficiency. Now that the libraries can check for the existence of the faster ECMAScript5 primitive, they can "shim" it invisibly in their implementations.

Note I hadn't heard the word "shim" used in this particular sense before. But JavaScript developers apply it to an abstraction that probes for the native presence of a piece of functionality, and prefers the native but can fulfill the request itself if the need arises.

So given that the standards committee didn't particularly add forEach for you anyway, there's not much point in rewriting that jQuery code you copy-pasted off StackOverflow to use it. Except...

**You almost certainly shouldn't use jQuery in Node**

Using jQuery in Node has been technically possible for a while, and [as of late 2013](http://stackoverflow.com/a/4129032/211160) the official jQuery team took over management of the Node.JS package jquery in NPM. But it is a little bit *out there* as most server-side code does not want--or need--to be doing manipulations on a browser DOM. Thus jQuery's lack of appeal on server-side Node gave an early usage boost to a leaner set of foundations: and that would be Underscore.JS.

Underscore doesn't have anything to do with web browser abilities like setting up click methods or styling things with CSS. It's just the basics like those missing functions for iterating and testing datatypes in a safe way. It also helps fill in other generally useful methods--like getting all the keys from an object and putting them in an array:

[Underscore.JS one-page function reference](http://documentcloud.github.io/underscore/)

While jQuery and Underscore have *slight* overlap, it actually isn't much. And if you're programming on the browser side in jQuery and are worried about yet-another-dependency... the dependency's *size* is actually not all that much of an issue. In fact, you can read the nicely presented [Annotated Underscore.JS Source](http://underscorejs.org/docs/underscore.html) in a novel two column format, showing it to be short and clear.

### Conclusion

So to sum up the points:

* Don't use jQuery on Node.JS server-side (unless you have a really weird situation you're sure needs it)
* Underscore is used by many-if-not-most Node projects, and you should probably use it too. For its size, it provides some useful general stuff.
* Given its small size, Underscore is a fairly inconsequential dependency for your browser client-side projects. Plus, if you're using jQuery the last thing you should be claiming to worry about is the size. :-P

[Underscore.js](http://underscorejs.org/) is a popular JavaScript library that contains 112 commonly needed helper functions.

Compare these two bits of code, the first using just JavaScript and the second using Underscore:

list[list.length - 1]

vs.

\_.last(list)

While Underscore doesn't save much code here, it makes the code easier to read than its plain JavaScript equivalent.

Another example. A common hack in JavaScript is to wait for all other processing to finish before running a particular bit of code.

This is done by setting a timeout with a wait time of 0 milliseconds:

setTimeout(newTask, 0)

By contrast, Underscore has a [defer](http://underscorejs.org/#defer) method - all it does is give a more meaningful name to "wait for 0 milliseconds":

\_.defer(newTask)

Underscore is full of handy methods like these. Here are three more of them:

* \_.toArray(arguments) instead of Array.prototype.slice.call(arguments).
* \_.contains(list, item) instead of list.indexOf(item) !== -1.
* \_.isUndefined(value) instead of value === void(0)

 Underscore helps you write code that clearly expresses your intentions

Instead of having to cater to the expectations of the JavaScript language, Underscore helps you write code that clearly expresses your intentions.

