OH MY JS

THE BEST JAVASCIRPT ARTICLES

</>

AZAT MARDANOV

Oh My JS

The Best JavaScript Articles

Azat Mardanov

This book is for sale at http://leanpub.com/ohmyjs

This version was published on 2014-02-01

Tweet This Book!

Please help Azat Mardanov by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Oh My JS CC @RPJSBook

The suggested hashtag for this book is #ohmyjs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#ohmyjs

Also By Azat Mardanov

Rapid Prototyping with JS

Express.js Guide

JavaScript and Node FUNdamentals

To God, My Teachers and Vladimir Nabokov

Contents

JavaScript Fundamentals

The World's Most Misunderstood Programming Language

Original Article

http://javascript.crockford.com/javascript.html

Douglas Crockford, crockford.com[a]

[a] http://crockford.com

JavaScript[1], aka Mocha, aka LiveScript, aka JScript, aka ECMAScript, is one of the world's most popular programming languages. Virtually every personal computer in the world has at least one JavaScript interpreter installed on it and in active use. JavaScript's popularity is due entirely to its role as the scripting language of the WWW.

Despite its popularity, few know that JavaScript is a very nice dynamic object-oriented general-purpose programming language. How can this be a secret? Why is this language so misunderstood?

The Name

The *Java-* prefix suggests that JavaScript is somehow related to Java, that it is a subset or less capable version of Java. It seems that the name was intentionally selected to create confusion, and from confusion comes misunderstanding. JavaScript is not interpreted Java. Java is interpreted Java. JavaScript is a different language.

JavaScript has a syntactic similarity to Java, much as Java has to C. But it is no more a subset of Java than Java is a subset of C. It is better than Java in the applications that Java (fka Oak) was originally intended for.

JavaScript was not developed at Sun Microsystems, the home of Java. JavaScript was developed at Netscape. It was originally called LiveScript, but that name wasn't confusing enough.

The *-Script* suffix suggests that it is not a real programming language, that a scripting language is less than a programming language. But it is really a matter of specialization. Compared to C, JavaScript trades performance for expressive power and dynamism.

[1] http://javascript.crockford.com/

Lisp in C's Clothing

JavaScript's C-like syntax, including curly braces and the clunky for statement, makes it appear to be an ordinary procedural language. This is misleading because JavaScript has more in common with functional languages like Lisp or Scheme[2] than with C or Java. It has arrays instead of lists and objects instead of property lists. Functions are first class. It has closures. You get lambdas without having to balance all those parens.

Typecasting

JavaScript was designed to run in Netscape Navigator. Its success there led to it becoming standard equipment in virtually all web browsers. This has resulted in typecasting. JavaScript is the George Reeves[3] of programming languages. JavaScript is well suited to a large class of non-Web-related applications

Moving Target

The first versions of JavaScript were quite weak. They lacked exception handling, inner functions, and inheritance. In its present form, it is now a complete object-oriented programming language. But many opinions of the language are based on its immature forms.

The ECMA committee that has stewardship over the language is developing extensions which, while well intentioned, will aggravate one of the language's biggest problems: There are already too many versions. This creates confusion.

Design Errors

No programming language is perfect. JavaScript has its share of design errors, such as the overloading of + to mean both addition and concatenation with type coercion, and the error-prone with statement should be avoided. The reserved word policies are much too strict. Semicolon insertion was a huge mistake, as was the notation for literal regular expressions. These mistakes have led to programming errors, and called the design of the language as a whole into question. Fortunately, many of these problems can be mitigated with a good lint[4] program.

The design of the language on the whole is quite sound. Surprisingly, the ECMAScript committee does not appear to be interested in correcting these problems. Perhaps they are more interested in making new ones.

---

[2] http://javascript.crockford.com/little.html

[3] http://www.amazon.com/exec/obidos/ASIN/B000KWZ7JC/wrrrldwideweb

[4] http://www.jslint.com/

Lousy Implementations

Some of the earlier implementations of JavaScript were quite buggy. This reflected badly on the language. Compounding that, those implementations were embedded in horribly buggy web browsers.

Bad Books

Nearly all of the books about JavaScript are quite awful. They contain errors, poor examples, and promote bad practices. Important features of the language are often explained poorly, or left out entirely. I have reviewed dozens of JavaScript books, and**I can only recommend one**: *JavaScript: The Definitive Guide (5th Edition)*[5] by David Flanagan. (Attention authors: If you have written a good one, please send me a review copy.)

Substandard Standard

The official specification for the language[6] is published by ECMA[7]. The specification is of extremely poor quality. It is difficult to read and very difficult to understand. This has been a contributor to the Bad Book problem because authors have been unable to use the standard document to improve their own understanding of the language. ECMA and the TC39 committee should be deeply embarrassed.

Amateurs

Most of the people writing in JavaScript are not programmers. They lack the training and discipline to write good programs. JavaScript has so much expressive power that they are able to do useful things in it, anyway. This has given JavaScript a reputation of being strictly for the amateurs, that it is not suitable for professional programming. This is simply not the case.

Object-Oriented

Is JavaScript object-oriented? It has objects which can contain data and methods that act upon that data. Objects can contain other objects. It does not have classes, but it does have constructors which do what classes do, including acting as containers for class variables and methods. It does not have class-oriented inheritance, but it does have prototype-oriented inheritance.

The two main ways of building up object systems are by inheritance (is-a) and by aggregation (has-a). JavaScript does both, but its dynamic nature allows it to excel at aggregation.

Some argue that JavaScript is not truly object oriented because it does not provide information hiding. That is, objects cannot have private variables and private methods: All members are public.

[5] http://www.amazon.com/exec/obidos/ASIN/0596101996/wrrrldwideweb

[6] http://www.ecma-international.org/publications/standards/Ecma-262.htm

[7] http://www.ecma-international.org/

But it turns out that JavaScript objects *can* have private variables and private methods. (Click here now to find out how.)[8] Of course, few understand this because JavaScript is the world's most misunderstood programming language.

Some argue that JavaScript is not truly object oriented because it does not provide inheritance. But it turns out that JavaScript supports not only classical inheritance, but other code reuse patterns as well.[9]

Understanding JavaScript Function Invocation and "this"

Original Article

http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this

Yehuda Katz, yehudakatz.com

Over the years, I've seen a lot of confusion about JavaScript function invocation. In particular, a lot of people have complained that the semantics of `this` in function invocations is confusing.

In my opinion, a lot of this confusion is cleared up by understanding the core function invocation primitive, and then looking at all other ways of invoking a function as sugar on top of that primitive. In fact, this is exactly how the ECMAScript spec thinks about it. In some areas, this post is a simplification of the spec, but the basic idea is the same.

The Core Primitive

First, let's look at the core function invocation primitive, a Function's `call` method[1]. The call method is relatively straight forward.

1. Make an argument list (`argList`) out of parameters 1 through the end
2. The first parameter is `thisValue`
3. Invoke the function with `this` set to `thisValue` and the `argList` as its argument list

For example:

[8] http://www.crockford.com/javascript/private.html
[9] http://javascript.crockford.com/inheritance.html
[10] mailto:douglas@crockford.com
[11] http://www.crockford.com/

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}

hello.call("Yehuda", "world") //=> Yehuda says hello world
```

As you can see, we invoked the `hello` method with `this` set to `"Yehuda"` and a single argument `"world"`. This is the core primitive of JavaScript function invocation. You can think of all other function calls as desugaring to this primitive. (to "desugar" is to take a convenient syntax and describe it in terms of a more basic core primitive).

[1] In the ES5 spec[12], the `call` method is described in terms of another, more low level primitive, but it's a very thin wrapper on top of that primitive, so I'm simplifying a bit here. See the end of this post for more information.

Simple Function Invocation

Obviously, invoking functions with `call` all the time would be pretty annoying. JavaScript allows us to invoke functions directly using the parens syntax (`hello("world")`. When we do that, the invocation desugars:

```
function hello(thing) {
  console.log("Hello " + thing);
}

// this:
hello("world")

// desugars to:
hello.call(window, "world");
```

This behavior has changed in ECMAScript 5 **only when using strict mode**[2]:

```
// this:
hello("world")

// desugars to:
hello.call(undefined, "world");
```

[12] http://es5.github.com/#x15.3.4.4

The short version is: **a function invocation like `fn(...args)` is the same as `fn.call(window [ES5-strict: undefined], ...args)`.**

Note that this is also true about functions declared inline: `(function() {})()` is the same as `(function() {}).call(window [ES5-strict: undefined)`.

[2] Actually, I lied a bit. The ECMAScript 5 spec says that `undefined` is (almost) always passed, but that the function being called should change its `thisValue` to the global object when not in strict mode. This allows strict mode callers to avoid breaking existing non-strict-mode libraries.

Member Functions

The next very common way to invoke a method is as a member of an object (`person.hello()`). In this case, the invocation desugars:

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this + " says hello " + thing);
  }
}

// this:
person.hello("world")

// desugars to this:
person.hello.call(person, "world");
```

Note that it doesn't matter how the `hello` method becomes attached to the object in this form. Remember that we previously defined `hello` as a standalone function. Let's see what happens if we attach is to the object dynamically:

```
function hello(thing) {
  console.log(this + " says hello " + thing);
}

person = { name: "Brendan Eich" }
person.hello = hello;

person.hello("world") // still desugars to person.hello.call(person, "world")

hello("world") // "[object DOMWindow]world"
```

Notice that the function doesn't have a persistent notion of its 'this'. It is always set at call time based upon the way it was invoked by its caller.

Using `Function.prototype.bind`

Because it can sometimes be convenient to have a reference to a function with a persistent `this` value, people have historically used a simple closure trick to convert a function into one with an unchanging `this`:

```
var person = {
  name: "Brendan Eich",
  hello: function(thing) {
    console.log(this.name + " says hello " + thing);
  }
}

var boundHello = function(thing) { return person.hello.call(person, thing); }

boundHello("world");
```

Even though our `boundHello` call still desugars to `boundHello.call(window, "world")`, we turn right around and use our primitive `call` method to change the `this` value back to what we want it to be.

We can make this trick general-purpose with a few tweaks:

```
var bind = function(func, thisValue) {
  return function() {
    return func.apply(thisValue, arguments);
  }
}

var boundHello = bind(person.hello, person);
boundHello("world") // "Brendan Eich says hello world"
```

In order to understand this, you just need two more pieces of information. First, `arguments` is an Array-like object that represents all of the arguments passed into a function. Second, the `apply` method works exactly like the `call` primitive, except that it takes an Array-like object instead of listing the arguments out one at a time.

Our `bind` method simply returns a new function. When it is invoked, our new function simply invokes the original function that was passed in, setting the original value as `this`. It also passes through the arguments.

Because this was a somewhat common idiom, ES5 introduced a new method `bind` on all `Function` objects that implements this behavior:

```
var boundHello = person.hello.bind(person);
boundHello("world") // "Brendan Eich says hello world"
```

This is most useful when you need a raw function to pass as a callback:

```
var person = {
  name: "Alex Russell",
  hello: function() { console.log(this.name + " says hello world"); }
}

$("#some-div").click(person.hello.bind(person));

// when the div is clicked, "Alex Russell says hello world" is printed
```

This is, of course, somewhat clunky, and TC39 (the committee that works on the next version(s) of ECMAScript) continues to work on a more elegant, still-backwards-compatible solution.

On jQuery

Because jQuery makes such heavy use of anonymous callback functions, it uses the `call` method internally to set the `this` value of those callbacks to a more useful value. For instance, instead of receiving `window` as `this` in all event handlers (as you would without special intervention), jQuery invokes `call` on the callback with the element that set up the event handler as its first parameter.

This is extremely useful, because the default value of `this` in anonymous callbacks is not particularly useful, but it can give beginners to JavaScript the impression that `this` is, **in general** a strange, often mutated concept that is hard to reason about.

If you understand the basic rules for converting a sugary function call into a desugared `func.call(thisValue, ...args)`, you should be able to navigate the not so treacherous waters of the JavaScript `this` value.

PS: I Cheated

In several places, I simplified the reality a bit from the exact wording of the specification. Probably the most important cheat is the way I called `func.call` a "primitive". In reality, the spec has a primitive (internally referred to as `[[Call]]`) that both `func.call` and `[obj.]func()` use.

However, take a look at the definition of `func.call`:

1. If IsCallable(func) is false, then throw a TypeError exception.
2. Let argList be an empty List.
3. If this method was called with more than one argument then in left to right order starting with arg1 append each argument as the last element of argList

4. Return the result of calling the [[Call]] internal method of func, providing thisArg as the this value and argList as the list of arguments.

As you can see, this definition is essentially a very simple JavaScript language binding to the primitive `[[Call]]` operation.

If you look at the definition of invoking a function, the first seven steps set up `thisValue` and `argList`, and the last step is: "Return the result of calling the [[Call]] internal method on func, providing thisValue as the this value and providing the list argList as the argument values."

It's essentially identical wording, once the `argList` and `thisValue` have been determined.

I cheated a bit in calling `call` a primitive, but the meaning is essentially the same as had I pulled out the spec at the beginning of this article and quoted chapter and verse.

Code Conventions for the JavaScript Programming Language

Original Article

http://javascript.crockford.com/code.html

Douglas Crockford, crockford.com

This is a set of coding conventions and rules for use in JavaScript programming. It is inspired by the Sun[13] document Code Conventions for the Java Programming Language[14]. It is heavily modified of course because JavaScript is not Java[15].

The long-term value of software to an organization is in direct proportion to the quality of the codebase. Over its lifetime, a program will be handled by many pairs of hands and eyes. If a program is able to clearly communicate its structure and characteristics, it is less likely that it will break when modified in the never-too-distant future.

Code conventions can help in reducing the brittleness of programs.

All of our JavaScript code is sent directly to the public. It should always be of publication quality.

Neatness counts.

[13] http://www.sun.com/

[14] http://java.sun.com/docs/codeconv/

[15] http://javascript.crockford.com/javascript.html

JavaScript Files

JavaScript programs should be stored in and delivered as `.js` files.

JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to pageweight with no opportunity for mitigation by caching and compression.

`<script src=filename.js>` tags should be placed as late in the body as possible. This reduces the effects of delays imposed by script loading on other page components. There is no need to use the `language` or `type` attributes. It is the server, not the script tag, that determines the MIME type.

Indentation

The unit of indentation is four spaces. Use of tabs should be avoided because (as of this writing in the 21st Century) there still is not a standard for the placement of tabstops. The use of spaces can produce a larger filesize, but the size is not significant over local networks, and the difference is eliminated by minification[16].

Line Length

Avoid lines longer than 80 characters. When a statement will not fit on a single line, it may be necessary to break it. Place the break after an operator, ideally after a comma. A break after an operator decreases the likelihood that a copy-paste error will be masked by semicolon insertion. The next line should be indented 8 spaces.

Comments

Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly yourself) who will need to understand what you have done. The comments should be well-written and clear, just like the code they are annotating. An occasional nugget of humor might be appreciated. Frustrations and resentments will not.

It is important that comments be kept up-to-date. Erroneous comments can make programs even harder to read and understand.

Make comments meaningful. Focus on what is not immediately visible. Don't waste the reader's time with stuff like

```
1       i = 0; // Set i to zero.
```

Generally use line comments. Save block comments for formal documentation and for commenting out.

[16] http://yuiblog.com/blog/2006/03/06/minification-v-obfuscation/

Variable Declarations

All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied globals[17]. Implied global variables should never be used.

The `var` statements should be the first statements in the function body.

It is preferred that each variable be given its own line and comment. They should be listed in alphabetical order.

```
    var currentEntry; // currently selected table entry
    var level;        // indentation level
    var size;         // size of table
```

JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages. Define all variables at the top of the function.

Use of global variables should be minimized. Implied global variables should never be used.

Function Declarations

All functions should be declared before they are used. Inner functions should follow the `var` statement. This helps make it clear what variables are included in its scope.

There should be no space between the name of a function and the `(` (left parenthesis) of its parameter list. There should be one space between the `)` (right parenthesis) and the `{` (left curly brace) that begins the statement body. The body itself is indented four spaces. The `}` (right curly brace) is aligned with the line containing the beginning of the declaration of the function.

```
    function outer(c, d) {
        var e = c * d;

        function inner(a, b) {
            return (e * a) + b;
        }

        return inner(0, 1);
    }
```

This convention works well with JavaScript because in JavaScript, functions and object literals can be placed anywhere that an expression is allowed. It provides the best readability with inline functions and complex structures.

[17] http://yuiblog.com/blog/2006/06/01/global-domination/

```
    function getElementsByClassName(className) {
        var results = [];
        walkTheDOM(document.body, function (node) {
            var a;                  // array of class names
            var c = node.className; // the node's classname
            var i;                  // loop counter

// If the node has a class name, then split it into a list of simple names.
// If any of them match the requested name, then append the node to the set of re\
sults.

            if (c) {
                a = c.split(' ');
                for (i = 0; i < a.length; i += 1) {
                    if (a[i] === className) {
                        results.push(node);
                        break;
                    }
                }
            }
        });
        return results;
    }
```

If a function literal is anonymous, there should be one space between the word `function` and the `(` (left parenthesis). If the space is omited, then it can appear that the function's name is `function`, which is an incorrect reading.

```
    div.onclick = function (e) {
        return false;
    };

    that = {
        method: function () {
            return this.datum;
        },
        datum: 0
    };
```

Use of global functions should be minimized.

When a function is to be invoked immediately, the entire invocation expression should be wrapped in parens so that it is clear that the value being produced is the result of the function and not the function itself.

```
var collection = (function () {
    var keys = [], values = [];

    return {
        get: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                return values[at];
            }
        },
        set: function (key, value) {
            var at = keys.indexOf(key);
            if (at < 0) {
                at = keys.length;
            }
            keys[at] = key;
            values[at] = value;
        },
        remove: function (key) {
            var at = keys.indexOf(key);
            if (at >= 0) {
                keys.splice(at, 1);
                values.splice(at, 1);
            }
        }
    };
}());
```

Names

Names should be formed from the 26 upper and lower case letters (`A` .. `Z`, `a` .. `z`), the 10 digits (`0` .. `9`), and _ (underbar). Avoid use of international characters because they may not read well or be understood everywhere. Do not use `$` (dollar sign) or `\` (backslash) in names.

Do not use _ (underbar) as the first character of a name. It is sometimes used to indicate privacy, but it does not actually provide privacy. If privacy is important, use the forms that provide private members[18]. Avoid conventions that demonstrate a lack of competence.

Most variables and functions should start with a lower case letter.

Constructor functions which must be used with the `[new](http://yuiblog.com/blog/2006/11/13/javascript-we` should start with a capital letter. JavaScript issues neither a compile-time warning nor a run-time

[18] http://javascript.crockford.com/private.html

[19] http://yuiblog.com/blog/2006/11/13/javascript-we-hardly-new-ya/

warning if a required `new` is omitted. Bad things can happen if `new` is not used, so the capitalization convention is the only defense we have.

Global variables should be in all caps. (JavaScript does not have macros or constants, so there isn't much point in using all caps to signify features that JavaScript doesn't have.)

Statements

Simple Statements

Each line should contain at most one statement. Put a `;` (semicolon) at the end of every simple statement. Note that an assignment statement which is assigning a function literal or object literal is still an assignment statement and must end with a semicolon.

JavaScript allows any expression to be used as a statement. This can mask some errors, particularly in the presence of semicolon insertion. The only expressions that should be used as statements are assignments and invocations.

Compound Statements

Compound statements are statements that contain lists of statements enclosed in `{ }` (curly braces).

- The enclosed statements should be indented four more spaces.
- The `{` (left curly brace) should be at the end of the line that begins the compound statement.
- The `}` (right curly brace) should begin a line and be indented to align with the beginning of the line containing the matching `{` (left curly brace).
- Braces should be used around all statements, even single statements, when they are part of a control structure, such as an `if` or `for` statement. This makes it easier to add statements without accidentally introducing bugs.

Labels

Statement labels are optional. Only these statements should be labeled: `while`, `do`, `for`, `switch`.

`return` Statement

A `return` statement with a value should not use `( )` (parentheses) around the value. The return value expression must start on the same line as the `return` keyword in order to avoid semicolon insertion.

`if` Statement

The `if` class of statements should have the following form:

```
if (condition) { statements }
if (condition) { statements } else { statements }
if (condition) { statements } else if (condition) { statements } else { statements }
```

`for` Statement

A `for` class of statements should have the following form:

```
for (initialization; condition; update) { statements }
for (variable in object) { if (filter) { statements } }
```

The first form should be used with arrays and with loops of a predeterminable number of iterations.

The second form should be used with objects. Be aware that members that are added to the prototype of the object ‘`will be included in the enumeration. It is wise to program defensively by using the` hasOwnProperty‘ method to distinguish the true members of the object:

```
for (variable in object) { if (object.hasOwnProperty(variable)) { statements } }
```

`while` Statement

A `while` statement should have the following form:

```
while (condition) { statements }
```

`do` Statement

A `do` statement should have the following form:

```
do { statements } while (condition);
```

Unlike the other compound statements, the `do` statement always ends with a `;` (semicolon).

`switch` Statement

A `switch` statement should have the following form:

```
switch (expression) { case expression: statements default: statements }
```

Each `case` is aligned with the `switch`. This avoids over-indentation.

Each group of statements (except the `default`) should end with `break`, `return`, or `throw`. Do not fall through.

`try` Statement

The `try` class of statements should have the following form:

`try {`
statements
`} catch (`variable`) {`
statements
}

`try {`
statements
`} catch (`variable`) {`
statements
‘ } finally { `statements` }‘

`continue` Statement

Avoid use of the `continue` statement. It tends to obscure the control flow of the function.

`with` Statement

The `with` statement should not be used[20].

Whitespace

Blank lines improve readability by setting off sections of code that are logically related.

Blank spaces should be used in the following circumstances:

- A keyword followed by `(` (left parenthesis) should be separated by a space.

```
1     while (true) {
```

- A blank space should not be used between a function value and its `(` (left parenthesis). This helps to distinguish between keywords and function invocations.
- All binary operators except `.` (period) and `(` (left parenthesis) and `[` (left bracket) should be separated from their operands by a space.
- No space should separate a unary operator and its operand except when the operator is a word such as `typeof`.
- Each ; (semicolon) in the control part of a `for` statement should be followed with a space.
- Whitespace should follow every , (comma).

[20] http://yuiblog.com/blog/2006/04/11/with-statement-considered-harmful/

Bonus Suggestions

`{}` and `[]`

Use `{}` instead of `new Object()`. Use `[]` instead of `new Array()`.

Use arrays when the member names would be sequential integers. Use objects when the member names are arbitrary strings or names.

`,` (comma) Operator

Avoid the use of the comma operator except for very disciplined use in the control part of `for` statements. (This does not apply to the comma separator, which is used in object literals, array literals, `var` statements, and parameter lists.)

Block Scope

In JavaScript blocks do not have scope. Only functions have scope. Do not use blocks except as required by the compound statements.

Assignment Expressions

Avoid doing assignments in the condition part of `if` and `while` statements.

Is

```
1     if (a = b) {
```

a correct statement? Or was

```
1     if (a == b) {
```

intended? Avoid constructs that cannot easily be determined to be correct.

`===` and `!==` Operators.

It is almost always better to use the `===` and `!==` operators. The `==` and `!=` operators do type coercion. In particular, do not use `==` to compare against falsy values.

Confusing Pluses and Minuses

Be careful to not follow a `+` with `+` or `++`. This pattern can be confusing. Insert parens between them to make your intention clear.

```
total = subtotal + +myInput.value;
```

is better written as

```
total = subtotal + (+myInput.value);
```

so that the `+ +` is not misread as `++`.

`eval` is Evil

The `eval` function is the most misused feature of JavaScript. Avoid it.

`eval` has aliases. Do not use the `Function` constructor. Do not pass strings to `setTimeout` or `setInterval`.

Semicolons in JavaScript are optional

Original Article

http://mislav.uniqpath.com/2010/05/semicolons/

Mislav Marohnić, mislav.uniqpath.com[a]

[a] http://mislav.uniqpath.com

JavaScript is a scripting language where semicolons as statement terminators are *optional*, as in:

> **op·tion·al** (*adjective*)
> Available to be chosen but not obligatory

However, there is a lot of FUD (fear, uncertainty, and doubt) around this feature and, as a result, most developers in our community will recommend always including semicolons[21], just to be safe.

Safe from *what?* I've been searching for reasons programmers have to force semicolons on themselves. Here's what I generally found:

[21] http://stackoverflow.com/questions/444080/do-you-recommend-using-semicolons-after-every-statement-in-javascript

Spec is cryptic and JavaScript implementations differ

Rules for automatic semicolon insertion[22] are right here and are, while somewhat difficult to comprehend by a casual reader, quite explicitly laid out. As for JavaScript implementations where interpretation of these rules *differs*, well, I've yet to find one. When I ask developers about this, or find archived discussions, typically they are yeah, there's this browser where this is utterly broken, I simply forgot which. Of course, they never remember.

I write semicolon-less code and, in my experience, there isn't a JavaScript interpreter that can't handle it.

You can't minify JavaScript code without semicolons

There are 3 levels of reducing size of JavaScript source files: *compression* (e.g. gzip), *minification* (i.e. removing unnecessary whitespace and comments) and *obfuscation* (changing code, shortening variable and function names).

Compression like gzip is the easiest; it only requires one-time server configuration, doesn't need extra effort by developers and *doesn't change* your code. There was a time when IE6 couldn't handle it, but if I remember correctly it was patched years ago and pushed as a Windows update, and today nobody really cares anymore.

Minification and obfuscation *change your code*. They are tools which you run on your source code saying "here are some JavaScript files, try to make them smaller, but **don't change functionality**". I'm reluctant to use these tools because many developers report that if I don't use specific coding styles, like writing semicolons, they will break my code. I'm OK with people (community) forcing certain coding styles on me, but not tools.

Suppose I have code that works in every JavaScript implementation that I target (major browsers and some server-side implementations). If I run it through your minification tool and that tools *breaks* my code, then I'm sad to report that your tool is *broken*. If this tool edits JavaScript code, it'd better understand it as a real interpreter would.

While on the topic of minification, let's do a reality check. I took the jQuery source and removed all semicolons[23], then ran it through Google Closure Compiler[24]. Resulting size was 76,673 bytes. The size of original "jquery.min.js" was 76,674 (1 byte more). So you see, there was almost no change; and of course, its test suite passed as before.

How is that possible? Well, consider this code:

[22] http://bclary.com/2004/11/07/#a-7.9

[23] http://github.com/mislav/jquery/commit/4a2faf8987fc3fcb8aefc99def5b5ed2b4de190c

[24] http://code.google.com/closure/compiler/

```
var a=1
var b=2
var c=3
```

That's 24 bytes right there. Stamp semicolons everywhere and run it through a minifier:

```
var a=1;var b=2;var c=3;
```

Still 24 bytes. So, adding semicolons and removing newlines saved us a whopping zero bytes right there. Radical. Most size reduction after minification isn't gained by removing newline characters — it's thanks to removing code comments and leading indentation.

Update: a lot of people have pointed out that their minifiers *rewrite* this expression as `var a=1,b=2,c=3`. I know that some tools do this, but the point of this article is just to explore how semicolons relate to whitespace. If a minifier is capable of rewriting expressions (e.g. Closure Compiler) it means that it can also insert semicolons automatically.

Also, some people recommend forcing yourself do use curly braces for blocks, even if they're only one line:

```
// before
if(condition) stuff()

// after
if(condition){
  stuff()
}

// after minification
if(condition){stuff()}
```

Enforced curly braces add at least a byte to our expression, even after minification. I'm not sure what the benefit is here—it's not size and it's not readability, either.

Here are some other whitespace-sensitive languages that you might have heard about:

- Ruby — messing with spaces in expressions with operators and method calls can break the code
- Python — duh.
- HTML — see notes about Kangax's HTML minifier[25]
- Haml templates[26]

[25] http://perfectionkills.com/experimenting-with-html-minifier/
[26] http://haml-lang.com/

Of course, there's no need for minification on the server-side. I made this list for the sake of the following argument: Whitespace can, and often is, part of the (markup) language. It's not necessarily a bad thing.

It's good coding style

Also heard as:

- It's good to have them for the sake of consistency
- JSLint[27] will complain
- Douglas Crockford says so.[28]

This is another way of expressing the "everybody else is doing it" notion and is used by people during online discussion in the (rather common) case of a lack of arguments.

My advice on JSLint: don't use it. Why would you use it? If you believed that it helps you have less bugs in your code, here's a newsflash; only people can detect and solve software bugs, not tools. So instead of tools, get more people to look at your code.

Douglas Crockford also says "four spaces", and yet most popular JavaScript libraries are set in either tabs or two spaces. Communities around different projects are *different*, and that's just how it should be. As I've said before: let *people* and yourself shape your coding style, not some single person or tool.

You might notice that in this article I'm not telling you *should* be semicolon-free. I'm just laying out concrete evidence that you *can* be. The choice should always be yours.

As for *coding styles*, they exist so code is more readable and easier to understand for a group of people in charge of working on it. Think deeply if semicolons actually improve the readability of your code. What improves it the most is whitespace—indentation, empty lines to separate blocks, spaces to pad out expressions—and good variable and function naming. Look at some obfuscated code[29]; there are semicolons in there. Does it help readability? No, but what would really help is a lot of whitespace and original variable names.

Semicolon insertion bites back in return statements

When I searched for "JavaScript semicolon insertion", here is the problem most blog posts described:

[27] http://www.jslint.com/

[28] http://javascript.crockford.com/code.html

[29] http://img.skitch.com/20100509-qf8t69ad7cpmudwdksbw5hu6te.png

```
function add() {
  var a = 1, b = 2
  return
    a + b
}
```

When you're done trying to wrap your brain around why would anyone in their right mind want to write a return statement on a new line, we can continue and see how this statement is interpreted:

```
return;
  a + b;
```

Alas, the function didn't return the sum we wanted! But you know what? This problem *isn't* solved by adding a semicolon to the end of our wanted `return` expression (that is, after `a + b`). It's solved by *removing* the newline after `return`:

```
return a + b
```

Still, in an incredible display of ignorance these people actually *advise* their readers to avoid such issues by adding semicolons everywhere. Uh, alright, only it doesn't help this particular case at all. We just needed to understand better how the language is parsed.

The only real pitfall when coding without semicolons

Here is the only thing you have to be aware if you choose to code semicolon-less:

```
// careful: will break
a = b + c
(d + e).print()
```

This is actually evaluated as:

```
a = b + c(d + e).print();
```

This example is taken from an article about JavaScript 2.0 future compatibility[30], but I've ran across this in my own programs several times while using the module pattern[31].

Easy solution: when a line starts with parenthesis, prepend a semicolon to it.

[30] http://www.mozilla.org/js/language/js20-2000-07/rationale/syntax.html

[31] http://www.yuiblog.com/blog/2007/06/12/module-pattern/

```
;(d + e).print()
```

This might not be elegant, but does the job. Michaeljohn Clement elaborates on this[32] even further:

> If you choose to omit semicolons where possible, my advice is to insert them immediately before the opening parenthesis or square bracket in any statement that begins with one of those tokens, or any which begins with one of the arithmetic operator tokens `/`, `+`, or `-` if you should happen to write such a statement.

Adopt this advice as a rule and you'll be fine.

[32] http://inimino.org/~inimino/blog/javascript_semicolons

Patterns and Code Organization

Common JavaScript "Gotchas"

Original Article

http://www.jblotus.com/2013/01/13/common-javascript-gotchas

James Fuller, jblotus[a]

[a] http://www.jblotus.com/author/jblotus/

PHP was my first programming language, and my initial exposure to JavaScript was through libraries like jQuery[33]. There were things about JavaScript that always seemed to trip me up in the beginning due to how they worked differently than PHP. Heck there are still some things today that are confusing. I want to share some of the things that I struggled when I started working with JavaScript. I am going to cover the global namespace, `this`, knowing the difference between **ECMAScript 3** and **ECMAScript 5**, asynchronous operations, prototypes, and simple JavaScript inheritance.

The Global Namespace

In PHP specifically, when you declare a variable function outside of a class (or namespace block[34]) you are essentially adding a function to the global namespace. In JavaScript, there is no such thing as a namespace per se, rather everything is attached to the global object. In the case of the web browser, that is the `window` object. The other key difference is that in JavaScript, functions and variables are attributes of the global object, which we typically refer to as properties.

This can be troublesome because you won't get a warning in JavaScript if you overwrite a global function or property and it can actually be quite dangerous.

[33] http://jquery.com/

[34] http://php.net/manual/en/language.namespaces.definitionmultiple.php

```
function globalFunction() {
  console.log('I am the original global function');
}

function overWriteTheGlobal() {
  globalFunction = function() {
    console.log('I am the new global function');
  }
}
globalFunction(); //outputs "I am the original global function"
overWriteTheGlobal(); //this will overwrite the original global function
globalFunction(); //outputs "I am the new global function"
```

One technique that is useful in JavaScript to ensure that your variables and functions are self contained is to use a **immediately-invoked function expression**, commonly known as a **self-executing anonymous function**. I typically expose things to the outside world by passing in a carrier object to the function. This is a variation of the **module pattern**.

```
var module = {};

(function(exports){

  exports.notGlobalFunction = function() {
    console.log('I am not global');
  };

}(module));

function notGlobalFunction() {
  console.log('I am global');
}

notGlobalFunction(); //outputs "I am global"
module.notGlobalFunction(); //outputs "I am not global"
```

Inside the **self-executing anonymous function**, all of the global scope is enclosed and we finish by attaching it the the `module` variable. Technically you could just append properties directly to the `module` variable, but the reason we are passing it in to the function is to make it explicitly clear what we are attaching our function to. It also allows us to alias the passed in object inside the function. The critical thing here is that we are declaring our dependencies upfront and not relying on global variables, other than the module variable.

You also might have noticed the `var` keyword. If you aren't sure of how it is used, a basic explanation is that by preceding a variable declaration with `var` creates a property on the nearest containing function. If you omit the `var` keyword than you are saying that you want to assign a new value to an existing variable higher up the scope chain, which may or may not be the global scope.

```
var imAGlobal = true;

function globalGrabber() {
  imAGlobal = false;
  return imAGlobal;
}

console.log(imAGlobal); //outputs "true"
console.log(globalGrabber()); //outputs "false"
console.log(imAGlobal); //outputs "false"
```

As you can see, it is quite dangerous to rely on globals in your functions, due to possible side effects and collisions that are bound to occur. Now what happens when we use the `var` keyword?

```
var imAGlobal = true;

function globalGrabber() {
  var imAGlobal = false;
  return imAGlobal;
}

console.log(imAGlobal); //outputs "true"
console.log(globalGrabber()); //outputs "false"
console.log(imAGlobal); //outputs "true"
```

JavaScript hoists the `var` declaration to the top of the function block, then initializes the variable. This is called **variable hoisting**.

To summarize: all variables are scoped to a function (which is itself an object), and where you declare those variables with `var` determines the function they are scoped to. Excluding var will imply global scope for a variable.

Let's look at how variable hoisting happens:

```
function variableHoist() {
  console.log(hoisty);
  hoisty = 1;
  console.log(hoisty);
  var hoisty = 2;
  console.log(hoisty);
}

variableHoist();
//outputs undefined (would get a ReferenceError if no var declaration existed in \
scope)
//outputs "1"
//outputs "2"

try {
  console.log(hoisty); //outputs ReferenceError (no global var "hoisty")
} catch (e) {
  console.log(e);
}
```

So as you can see, it doesn't actually matter where you put the `var` declaration in the function, because the property gets created before the function executes any code. Now in practice, generally you want to put your `var` declarations at the top of the function, since that is where they end up anyway. It is also totally acceptable to initialize you variables at the top of the function, just be aware of the order of events here.

Functions declared with the `function` keyword in JavaScript (not variable assignment) also get hoisted. Behind the scenes, the entire function gets hoisted up and is made available for execution.

```
myFunction(); //outputs "i exist"

function myFunction() {
  console.log('i exist');
}
```

**

** This wholesale function hoisting does not occur when you use the var form of function declaration:

```
try {
  myFunction();
} catch (e) {
  console.log(e); //throws "Uncaught TypeError: undefined is not a function"
}
var myFunction = function() {
  console.log('i exist');
}

myFunction();  //outputs "i exist"
```

Understanding "this"

Since JavaScript uses function scope, the meaning of `this` is quite different than what you get in **PHP**, and causes a lot of confusion. Consider the following:

```
console.log(this); // outputs window object

var myFunction = function() {
  console.log(this);
}

myFunction(); //outputs window object

var newObject = {
  myFunction: myFunction
}

newObject.myFunction(); //outputs newObject
```

`this` by default refers to the object a function is contained in. Since `myFunction()` is a property of the global object, `this` is a reference to the global object, which is `window`. Now when we mix `myFunction()` into a `newObject`, `this` now refers to `newObject`. In **PHP** and other similar languages, `this` always refers to the the instance of a class containing the method. You could argue that JavaScript is doing something stupid here, but truthfully much of the power of the JavaScript language comes from this feature. In fact, we can even replace the value of `this` when invoking our JavaScript functions by using the `call()` or `apply()` methods.

```
var myFunction = function(arg1, arg2) {
  console.log(this, arg1, arg2);
};

var newObject = {};

myFunction.call(newObject, 'foo', 'bar'); //outputs newObject "foo" "bar"
myFunction.apply(newObject, ['foo', 'bar']); //outputs newObject "foo" "bar"
```

But let's not get ahead of ourselves. All we are doing here is invoking the function `myFunction` by substituting an alternative value for `this` inside the function by placing the value of the object we want to use a substitute as the first argument. The fundamental difference between `call()` and `apply()` is the way you pass arguments to the function. `call()` will take an unlimited amount of arguments after the first argument and `apply()` expects and array of arguments as it's second argument.

Libraries like jQuery perform a lot of magic by invoking things this way. **Let's look at the `$.each()` method in jQuery:**

```
var $els = [$('div'), $('span')];
var handler = function() {
  console.log(this);
};

$.each($els, handler);

//iteration 1 outputs wrapped jquery dom element for "div" tag
//iteration 2 outputs wrapped jquery dom element for "span" tag

handler.apply({}); //outputs object
```

jQuery will often rewrite the value of `this`, so you should always try to be aware of what `this` means in the context of a jQuery event handler, or other such constructs.

Know the difference between ECMAScript 3 and ECMAScript 5

For many years, **ECMAScript 3** has been the standard in most browsers, but more recently **ECMAScript 5** has made it's way into most modern browsers (IE is still lagging behind). ECMAScript 5 introduced a lot of common sense features to JavaScript and some native methods that you previously relied upon a library for, such as `String.trim()` and `Array.forEach()`. The problem is you still can't rely on these methods being available in browser environments if you have users that are using Internet Explorer.

Take a look at what happens when we try to use `String.trim` in IE 8:

```
var fatString = "   my string   ";

//in modern browsers
console.log(fatString); //outputs "   my string   "
console.log(fatString.trim()); //outputs "my string"

//in IE 8
console.log(fatString.trim()); //error: Object doesn't support property or method\
 'trim'
```

So in the interim, we can use methods like `jQuery.trim` to do this for us, which I believe will fallback to `String.trim` if it is available in your browser for increased performance (native browser implementations are faster).

You might not care or even need to know about all of the differences between **ECMAScript 3** and **ECMAScript 5**, but it is generally a good idea to check out the Mozilla Developer Network (MDN)[35] for function reference to see what versions of the language a function is available in first. Generally speaking, you should be fine if you are using a library like **jQuery** or **underscore** to handle this for you.

If you are interested in using a polyfill of **ECMAScript 5** for older browser, please check out https://github.com/kriskowal/es5-shim[36]

Understanding Async

One of the things that tripped me up the most when beginning to work with JavaScript code, jQuery in particular is the fact that some operations are asynchronous. There were many times that I wrote code in a procedural manner expecting a result to be returned immediately without realizing it.

Take a look at this broken code:

```
var remoteValue = false;
$.ajax({
  url: 'http://google.com',
  success: function() {
    remoteValue = true;
  }
});

console.log(remoteValue); //outputs "false"
```

It took me a while to realize that you need to program around asynchronous calls using callbacks to deal with the outcome of my ajax calls.

[35] https://developer.mozilla.org/en-US/docs/JavaScript
[36] https://github.com/kriskowal/es5-shim

```
var remoteValue = false;

var doSomethingWithRemoteValue = function() {
  console.log(remoteValue); //outputs true on success
}

$.ajax({
  url: 'https://google.com',
  complete: function() {
    remoteValue = true;
    doSomethingWithRemoteValue();
  }
});
```

Another cool thing is deferred objects (sometimes called promises), which you can use to program in a more procedural style:

```
var remoteValue = false;

var doSomethingWithRemoteValue = function() {
  console.log(remoteValue);
}

var promise = $.ajax({
  url: 'https://google.com'
});

//outputs "true"
promise.always(function() {
  remoteValue = true;
  doSomethingWithRemoteValue();
});

//outputs "foobar"
promise.always(function() {
  remoteValue = 'foobar';
  doSomethingWithRemoteValue();
});
```

You can use promises to chain callbacks in a style that is in my opinion a bit easier to work with than nested callbacks in addition to a host of other benefits these objects offer.

Animations in the browser are also asyncronous, so this is also a common source of confusion. I'm not going to go into detail here, but you need to treat animations much like ajax requests in the way you handle them via callbacks. I'm not really an expert on the subject though so please take a look at the jQuery .animate() method[37].

Simple Inheritance in JavaScript

Grossly simplified, JavaScript clones objects to extend them, while PHP, Ruby, Python and Java use and extend classes. In JavaScript you have something called a `prototype`, and every object has one. In fact, all functions, strings, numbers and objects have a common ancestor, `Object`. There are two things about prototype to remember: blueprints and chains.

Each `prototype` is basically an object in itself that describes properties available when creating an instance of an object. The prototype chain is what allows prototypes to extend other prototypes. In fact, prototypes themselves can have prototypes. When a method or attribute does not exist on an object instance, then it is looked for in that object's `prototype`, and the prototypes's `prototype`, and so on until it finally reaches `undefined` if no such property exists.

Thankfully, beginners generally don't need to mess with this stuff at all, since it is easy enough to create an object literal and append properties to it at runtime.

```
var obj = {};

obj.newFunction = function() {
  console.log('I am a dynamic function');
};

obj.newFunction();
```

An easy way to extend objects that I use all the time is `jQuery.extend()`

```
var obj = {
  a: 'i am a lonely property'
};

var newObj = {
  b: function() {
    return 'i am a lonely function';
  }
};

```

[37] http://api.jquery.com/animate/

```
var finalObj = $.extend({}, obj, newObj);

console.log(finalObj.a); //outputs "i am a lonely property"
console.log(finalObj.b()); //outputs "i am a lonely function"
```

ECMAScript 5 offers us `Object.create()`, which you can use to extend from an existing object but you probably need to avoid using this if you need to support older browsers. It does offer distinct advantages to property creation and setting attributes of properties (yes, properties also have properties[38]).

```
var obj = {
  a: 'i am a lonely property'
};

var finalObj = Object.create(obj, {
  b: {
    get: function() {
      return "i am a lonely function";
    }
  }
});

console.log(finalObj.a); //outputs "i am a lonely property"
console.log(finalObj.b); //outputs "i am a lonely function"
```

You can get pretty deep into the subject of inheritance in JavaScript[39] but the beautiful thing here again is that you really don't have to due to the immense power and flexibility of the language.

Bonus Gotcha: Forgetting to use var in for loops

[38] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/defineProperty
[39] http://www.crockford.com/javascript/inheritance.html

```
var i = 0;

function iteratorHandler() {
  i = 10;
}

function iterate() {
 //this iteration will only run once
  for (i = 0; i < 10; i++) {
    console.log(i); //outputs 0
    iteratorHandler();
    console.log(i); //outputs 10
  }
}

iterate();
```

The example is contrived, but you can see the danger here. The solution is to declare you iterator variables with `var`.

```
var i = 0;

function iteratorHandler() {
  i = 10;
}

function iterate() {
  //this iteration will run 10 times
  for (var i = 0; i < 10; i++) {
    iteratorHandler();
    console.log(i);
  }
}

iterate();
```

This all goes back to our scope rules. Remember to use `var` properly.

Summary

JavaScript may be the only language people don't need to learn before using it, but eventually you are going to run in to some unexplained trouble. Other than avoiding your own bugs, learning JavaScript

makes a lot of sense these days considering it's rebirth and widespread availability. This blog by no means attempts to be a complete panacea, but hopefully it will help a few people understand some of the fundamentals before being forced into writing more awful JavaScript code, secretly hoping to get reassigned to a backend project buried in database queries in happy PHP land.

Asynchronous JS: Callbacks, Listeners, Control Flow Libs and Promises

Original Article

http://sporto.github.io/blog/2012/12/09/callbacks-listeners-promises

Sebastian Porto, sporto.github.io[a]

[a] http://sporto.github.io/

When it comes to dealing with **asynchronous** development in JavaScript there are many tool you can use. This post explains four of these tools and what their advantages are. These are Callbacks, Listeners, Control Flow Libraries and Promises.

Example Scenario

To illustrate the use of these four tools, let's create a simple example scenario.

Let's say that we want to find some records, then process them and finally return the processed results. Both operations (find and process) are asynchronous.

Callbacks

Let's start with callback pattern, this is the most basic and the best known pattern to deal with async programming.

A callback looks like this:

```
1 finder([1, 2], function(results) {
2   ..do something
3 });
```

In the callback pattern we call a function that will do the asynchronous operation. One of the parameters we pass is a function that will be called when the operation is done.

Setup

In order to illustrate how they work we need a couple of functions that will find and process the records. In the real world these functions will make an AJAX request and return the results, but for now let's just use timeouts.

```
function finder(records, cb) {
    setTimeout(function () {
        records.push(3, 4);
        cb(records);
    }, 1000);
}
function processor(records, cb) {
    setTimeout(function () {
        records.push(5, 6);
        cb(records);
    }, 1000);
}
```

Using the callbacks

The code that consumes these functions looks like this:

```
finder([1, 2], function (records) {
    processor(records, function(records) {
      console.log(records);
    });
});
```

We call the first function, passing a callback. Inside this callback we call the second function passing another callback.

These nested callbacks can be written more clearly by passing a reference to another function.

```
function onProcessorDone(records){
  console.log(records);
}

function onFinderDone(records) {
    processor(records, onProcessorDone);
}

finder([1, 2], onFinderDone);
```

In both case the console log above with log [1,2,3,4,5,6]

Working example here:

Pros

- They are a very well know pattern, so they are familiar thus easy to understand.
- Very easy to implement in your own libraries / functions.

Cons

- Nested callbacks will form the infamous pyramid of doom as shown above, which can get hard to read when you have multiple nested levels. But this is quite easy to fix by splitting the functions also as shown above.
- You can only pass one callback for a given event, this can be a big limitation in many cases.

Photo credit: Brandon Christopher Warren / Foter / CC BY-NC

Listeners

Listeners are also a well known pattern, mostly made popular by jQuery and other DOM libraries. A Listener might look like this:

```
finder.on('done', function (event, records) {
  ..do something
});
```

We call a function on an object that adds a listener. In that function we pass the name of the event we want to listen to and a callback function. ‘on’ is one of many common name for this function, other common names you will come across are ‘bind’, ‘listen’, ‘addEventListener’, ‘observe’.

Setup

Let’s do some setup for a listener demonstration. Unfortunately the setup needed is a bit more involving than in the callbacks example.

First we need a couple of objects that will do the work of finding and processing the records.

```
var finder = {
    run: function (records) {
        var self = this;
        setTimeout(function () {
            records.push(3, 4);
           self.trigger('done', [records]);
        }, 1000);
    }
}
var processor = {
    run: function (records) {
        var self = this;
        setTimeout(function () {
            records.push(5, 6);
            self.trigger('done', [records]);
        }, 1000);
    }
 }
```

Note that they are calling a method **trigger** when the work is done, I will add this method to these objects using a mix-in. Again 'trigger' is one of the names you will come across, others common names are 'fire' and 'publish'.

We need a mix-in object that has the listener behaviour, in this case I will just lean on jQuery for this:

```
var eventable = {
    on: function(event, cb) {
        $(this).on(event, cb);
    },
    trigger: function (event, args) {
        $(this).trigger(event, args);
    }
}
```

Then apply the behaviour to our finder and processor objects:

```
 $.extend(finder, eventable);
 $.extend(processor, eventable);
```

Excellent, now our objects can take listeners and trigger events.

Using the listeners

The code that consumes the listeners is simple:

```
finder.on('done', function (event, records) {
  processor.run(records);
});
processor.on('done', function (event, records) {
    console.log(records);
});
finder.run([1,2]);
```

Again the console run will output [1,2,3,4,5,6]

Working example here:

Pros

- This is another well understood pattern.
- The big advantage is that you are not limited to one listener per object, you can add as many listeners as you want. E.g.

  finder .on('done', function (event, records) { .. do something }) .on('done', function (event, records) { .. do something else });

Cons

- A bit more difficult to setup than callbacks in your own code, you will probably want to use a library e.g. jQuery, bean.js[40].

Photo credit: Nod Young / Foter / CC BY-NC-SA

A Flow Control Library

Flow control libraries are also a very nice way to deal with asynchronous code. One I particularly like is Async.js[41].

Code using Async.js looks like this:

```
async.series([
    function(){ ... },
    function(){ ... }
]);
```

Setup (Example 1)

Again we need a couple of functions that will do the work, as in the other examples these functions in the real world will probably make an AjAX request and return the results. For now let's just use timeouts.

[40] https://github.com/fat/bean

[41] https://github.com/caolan/async

```
function finder(records, cb) {
    setTimeout(function () {
        records.push(3, 4);
        cb(null, records);
    }, 1000);
}
function processor(records, cb) {
    setTimeout(function () {
        records.push(5, 6);
        cb(null, records);
    }, 1000);
}
```

The Node Continuation Passing Style Note the style used in the callbacks inside the functions above.

```
cb(null, records);
```

The first argument in the callback is null if no error occurs; or the error if one occurs. This is a common pattern in Node.js libraries and Async.js uses this pattern. By using this style the flow between Async.js and the callbacks becomes super simple.

Using Async

The code that will consume these functions looks like this:

```
async.waterfall([
    function(cb){
        finder([1, 2], cb);
    },
    processor,
    function(records, cb) {
        alert(records);
    }
]);
```

Async.js takes care of calling each function in order after the previous one has finished. Note how we can just pass the 'processor' function, this is because we are using the Node continuation style. As you can see this code is quite minimal and easy to understand.

Working example here:

Another setup (Example 2)

Now, when doing front-end development it is unlikely that you will have a library that follows the callback(null, results) signature. So a more realistic example will look like this:

```
function finder(records, cb) {
    setTimeout(function () {
        records.push(3, 4);
        cb(records);
    }, 500);
}
function processor(records, cb) {
    setTimeout(function () {
        records.push(5, 6);
        cb(records);
    }, 500);
}

// using the finder and the processor
async.waterfall([
    function(cb){
        finder([1, 2], function(records) {
            cb(null, records)
        });
    },
    function(records, cb){
        processor(records, function(records) {
            cb(null, records);
        });
    },
    function(records, cb) {
        alert(records);
    }
]);
```

It becomes a lot more convoluted but at least you can see the flow going from top to bottom.

Working example here:

Pros

- Usually code using a control flow library is easier to understand because it follows a natural order (from top to bottom). This is not true with callbacks and listeners.

Cons

- If the signatures of the functions don't match as in the second example then you can argue that the flow control library offers little in terms of readability.

Photo credit: Helmut Kaczmarek / Foter / CC BY-NC-SA

Promises

Finally we get to our final destination. Promises are a very powerful tool, but they are the least understood.

Code using promises may look like this:

```
finder([1,2])
    .then(function(records) {
      .. do something
    });
```

This will vary widely depending on the promises library you use, in this case I am using when.js[42].

Setup

Out finder and processor functions look like this:

```
function finder(records){
    var deferred = when.defer();
    setTimeout(function () {
        records.push(3, 4);
        deferred.resolve(records);
    }, 500);
    return deferred.promise;
}
function processor(records) {
     var deferred = when.defer();
    setTimeout(function () {
        records.push(5, 6);
        deferred.resolve(records);
    }, 500);
    return deferred.promise;
}
```

Each function creates a deferred object and returns a promise. Then it resolves the deferred when the results arrive.

[42] https://github.com/cujojs/when

Using the promises

The code that consumes these functions looks like this:

```
finder([1,2])
    .then(processor)
    .then(function(records) {
            alert(records);
    });
```

As you can see, it is quite minimal and easy to understand. When used like this, promises bring a lot of clarity to your code as they follow a natural flow. Note how in the first callback we can simply pass the 'processor' function. This is because this function returns a promise itself so everything will just flow nicely.

Working example here:

There is a lot to promises:

- they can be passed around as regular objects
- aggregated into bigger promises
- you can add handlers for failed promises

The big benefit of promises

Now if you think that this is all there is to promises you are missing what I consider the biggest advantage. Promises have a neat trick that neither callbacks, listeners or control flows can do. You can add a listener to promise even when it has already been resolved, in this case that listener will trigger immediately, meaning that you don't have to worry if the event has already happened when you add the listener. This works the same for aggregated promises. Let me show you an example of this:

This is a huge feature for dealing with user interaction in the browser. In complex applications you may not now the order of actions that the user will take, so you can use promises to track use interaction. See this other post[43] if interested.

Pros

- Really powerful, you can aggregate promises, pass them around, or add listeners when already resolved.

[43] http://sporto.github.com/blog/2012/09/22/embracing-async-with-deferreds/

Cons

- The least understood of all these tools.
- They can get difficult to track when you have lots of aggregated promises with added listeners along the way.

Conclusion

That's it! These are in my opinion the four main tools for dealing with asynchronous code. Hopefully I have help you to understand them better and gave you more options for you asynchronous needs.

The Design of Code: Organizing JavaScript

Original Article

http://alistapart.com/article/the-design-of-code-organizing-javascript

Anthony Colangelo, alistapart.com[a]

[a] http://alistapart.com

Great design is a product of care and attention applied to areas that matter, resulting in a useful, understandable, and hopefully beautiful user interface. But don't be fooled into thinking that design is left only for designers.

There is a lot of design in code, and I don't mean code that builds the user interface–I mean the design *of* code.

Well-designed code is much easier to maintain, optimize, and extend, making for more efficient developers. That means more focus and energy can be spent on building great things, which makes everyone happy–users, developers, and stakeholders.

There are three high-level, language-agnostic aspects to code design that are particularly important.

1. System architecture–The basic layout of the codebase. Rules that govern how various components, such as models, views, and controllers, interact with each other.
2. Maintainability–How well can the code be improved and extended?
3. Reusability–How reusable are the application's components? How easily can each implementation of a component be customized?

In looser languages, specifically JavaScript, it takes a bit of discipline to write well-designed code. The JavaScript environment is so forgiving that it's easy to throw bits and pieces everywhere and still have things work. Establishing system architecture early (and sticking to it!) provides constraints to your codebase, ensuring consistency throughout.

One approach I'm fond of consists of a tried-and-true software design pattern, the module pattern, whose extensible structure lends itself to a solid system architecture and a maintainable codebase. I like building modules within a jQuery plugin, which makes for beautiful reusability, provides robust options, and exposes a well-crafted API.

Below, I'll walk through how to craft your code into well-organized components that can be reused in projects to come.

The module pattern

There are a *lot* of design patterns out there, and equally as many resources on them. Addy Osmani[44] wrote an amazing (free!) book[45] on design patterns in JavaScript, which I highly recommend to developers of all levels.

The module pattern[46] is a simple structural foundation that can help keep your code clean and organized. A "module" is just a standard object literal containing methods and properties, and that simplicity is the best thing about this pattern: even someone unfamiliar with traditional software design patterns would be able to look at the code and instantly understand how it works.

In applications that use this pattern, each component gets its own distinct module. For example, to build autocomplete functionality, you'd create a module for the textfield and a module for the results list. These two modules would work together, but the textfield code wouldn't touch the results list code, and vice versa.

That decoupling of components is why the module pattern is great for building solid system architecture. Relationships within the application are well-defined; anything related to the textfield is managed by the textfield module, not strewn throughout the codebase–resulting in clear code.

Another benefit of module-based organization is that it is inherently maintainable. Modules can be improved and optimized independently without affecting any other part of the application.

I used the module pattern for the basic structure of jPanelMenu[47], the jQuery plugin I built for off-canvas menu systems. I'll use that as an example to illustrate the process of building a module.

Building a module

To begin, I define three methods and a property that are used to manage the interactions of the menu system.

[44] https://twitter.com/addyosmani

[45] http://addyosmani.com/resources/essentialjsdesignpatterns/book/

[46] http://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript

[47] http://jpanelmenu.com/

```
var jpm = {
    animated: true,
    openMenu: function( ) {
        ...
        this.setMenuStyle( );
    },
    closeMenu: function( ) {
        ...
        this.setMenuStyle( );
    },
    setMenuStyle: function( ) { ... }
};
```

The idea is to break down code into the smallest, most reusable bits possible. I could have written just one `toggleMenu( )` method, but creating distinct `openMenu( )` and `closeMenu( )` methods provides more control and reusability within the module.

Notice that calls to module methods and properties from *within* the module itself (such as the calls to `setMenuStyle( )`) are prefixed with the `this` keyword–that's how modules access their own members.

That's the basic structure of a module. You can continue to add methods and properties as needed, but it doesn't get any more complex than that. After the structural foundations are in place, the reusability layer–options and an exposed API–can be built on top.

jQuery plugins

The third aspect of well-designed code is probably the most crucial: reusability. This section comes with a caveat. While there are obviously ways to build and implement reusable components in raw JavaScript (we're about 90 percent of the way there with our module above), I prefer to build jQuery plugins for more complex things, for a few reasons.

Most importantly, it's a form of unobtrusive communication. If you used jQuery to build a component, you should make that obvious to those implementing it. Building the component as a jQuery plugin is a great way to say that jQuery is required.

In addition, the implementation code will be consistent with the rest of the jQuery-based project code. That's good for aesthetic reasons, but it also means (to an extent) that developers can predict how to interact with the plugin without too much research. Just one more way to build a better developer interface.

Before you begin building a jQuery plugin, ensure that the plugin does not conflict with other JavaScript libraries using the `$` notation. That's a lot simpler than it sounds–just wrap your plugin code like so:

```
(function($) {
    // jQuery plugin code here
})(jQuery);
```

Next, we set up our plugin and drop our previously built module code inside. A plugin is just a method defined on the jQuery `($)` object.

```
(function($) {
    $.jPanelMenu = function( ) {
        var jpm = {
            animated: true,
            openMenu: function( ) {
                ...
                this.setMenuStyle( );
            },
            closeMenu: function( ) {
                ...
                this.setMenuStyle( );
            },
            setMenuStyle: function( ) { ... }
        };
    };
})(jQuery);
```

All it takes to use the plugin is a call to the function you just created.

```
var jpm = $.jPanelMenu( );
```

Options

Options are essential to any truly reusable plugin because they allow for customizations to each implementation. Every project brings with it a slew of design styles, interaction types, and content structures. Customizable options help ensure that you can adapt the plugin to fit within those project constraints.

It's best practice to provide good default values for your options. The easiest way to do that is to use jQuery's `$.extend( )` method, which accepts (at least) two arguments.

As the first argument of `$.extend( )`, define an object with all available options and their default values. As the second argument, pass through the passed-in options. This will merge the two objects, overriding the defaults with any passed-in options.

```
(function($) {
    $.jPanelMenu = function(options) {
        var jpm = {
            options: $.extend({
                'animated': true,
                'duration': 500,
                'direction': 'left'
            }, options),
            openMenu: function( ) {
                ...
                this.setMenuStyle( );
            },
            closeMenu: function( ) {
                ...
                this.setMenuStyle( );
            },
            setMenuStyle: function( ) { ... }
        };
    };
})(jQuery);
```

Beyond providing good defaults, options become almost self-documenting–someone can look at the code and see all of the available options immediately.

Expose as many options as is feasible. The customization will help in future implementations, and flexibility never hurts.

API

Options are terrific ways to customize how a plugin works. An API, on the other hand, enables extensions to the plugin's functionality by exposing methods and properties for the implementation code to take advantage of.

While it's great to expose as much as possible through an API, the outside world shouldn't have access to all internal methods and properties. Ideally, you should expose only the elements that will be used.

In our example, the exposed API should include calls to open and close the menu, but nothing else. The internal `setMenuStyle( )` method runs when the menu opens and closes, but the public doesn't need access to it.

To expose an API, return an object with any desired methods and properties at the end of the plugin code. You can even map returned methods and properties to those within the module code–this is where the beautiful organization of the module pattern really shines.

```
(function($) {
    $.jPanelMenu = function(options) {
        var jpm = {
            options: $.extend({
                'animated': true,
                'duration': 500,
                'direction': 'left'
            }, options),
            openMenu: function( ) {
                ...
                this.setMenuStyle( );
            },
            closeMenu: function( ) {
                ...
                this.setMenuStyle( );
            },
            setMenuStyle: function( ) { ... }
        };

        return {
            open: jpm.openMenu,
            close: jpm.closeMenu,
            someComplexMethod: function( ) { ... }
        };
    };
})(jQuery);
```

API methods and properties will be available through the object returned from the plugin initialization.

```
var jpm = $.jPanelMenu({
    duration: 1000,
    ...
});
jpm.open( );
```

Polishing developer interfaces

With just a few simple constructs and guidelines, we've built ourselves a reusable, extensible plugin that will help make our lives easier. Like any part of what we do, experiment with this structure to see if it works for you, your team, and your workflow.

Whenever I find myself building something with a potential for reuse, I break it out into a module-based jQuery plugin. The best part about this approach is that it forces you to use–and test–the code you write. By using something as you build it, you'll quickly identify strengths, discover shortcomings, and plan changes.

This process leads to battle-tested code ready for open-source contributions, or to be sold and distributed. I've released my (mostly) polished plugins as open-source projects on GitHub[48].

Even if you aren't building something to be released in the wild, it's still important to think about the design of your code. Your future self will thank you.

Why AMD?

A javascript module loader

> ## Original Article
>
> http://requirejs.org/docs/why.html
>
> requirejs.org[a]
>
> [a] http://requirejs.org

This page talks about the design forces and use of the Asynchronous Module Definition (AMD) API[49] for JavaScript modules, the module API supported by RequireJS. There is a different page that talks about general approach to modules on the web[50].

Module Purposes § 1

What are JavaScript modules? What is their purpose?

- **Definition**: how to encapsulate a piece of code into a useful unit, and how to register its capability/export a value for the module.
- **Dependency References**: how to refer to other units of code.

The Web Today § 2

[48] https://github.com/acolangelo
[49] https://github.com/amdjs/amdjs-api/wiki/AMD
[50] http://requirejs.org/docs/why.html

```
(function () {
    var $ = this.jQuery;

    this.myExample = function () {};
}());
```

How are pieces of JavaScript code defined today?

- Defined via an immediately executed factory function.
- References to dependencies are done via global variable names that were loaded via an HTML script tag.
- The dependencies are very weakly stated: the developer needs to know the right dependency order. For instance, The file containing Backbone cannot come before the jQuery tag.
- It requires extra tooling to substitute a set of script tags into one tag for optimized deployment.

This can be difficult to manage on large projects, particularly as scripts start to have many dependencies in a way that may overlap and nest. Hand-writing script tags is not very scalable, and it leaves out the capability to load scripts on demand.

CommonJS § 3

```
var $ = require('jquery');
exports.myExample = function () {};
```

The original CommonJS (CJS) list[51] participants decided to work out a module format that worked with today's JavaScript language, but was not necessarily bound to the limitations of the browser JS environment. The hope was to use some stop-gap measures in the browser and hopefully influence the browser makers to build solutions that would enable their module format to work better natively. The stop-gap measures:

- Either use a server to translate CJS modules to something usable in the browser.
- Or use XMLHttpRequest (XHR) to load the text of modules and do text transforms/parsing in browser.

The CJS module format only allowed one module per file, so a "transport format" would be used for bundling more than one module in a file for optimization/bundling purposes.

With this approach, the CommonJS group was able to work out dependency references and how to deal with circular dependencies, and how to get some properties about the current module. However, they did not fully embrace some things in the browser environment that cannot change but still affect module design:

[51] http://groups.google.com/group/commonjs

- network loading
- inherent asynchronicity

It also meant they placed more of a burden on web developers to implement the format, and the stop-gap measures meant debugging was worse. eval-based debugging or debugging multiple files that are concatenated into one file have practical weaknesses. Those weaknesses may be addressed in browser tooling some day, but the end result: using CommonJS modules in the most common of JS environments, the browser, is non-optimal today.

AMD § 4

```
define(['jquery'] , function ($) {
    return function () {};
});
```

The AMD format comes from wanting a module format that was better than today's "write a bunch of script tags with implicit dependencies that you have to manually order" and something that was easy to use directly in the browser. Something with good debugging characteristics that did not require server-specific tooling to get started. It grew out of Dojo's real world experience with using XHR+eval and wanting to avoid its weaknesses for the future.

It is an improvement over the web's current "globals and script tags" because:

- Uses the CommonJS practice of string IDs for dependencies. Clear declaration of dependencies and avoids the use of globals.
- IDs can be mapped to different paths. This allows swapping out implementation. This is great for creating mocks for unit testing. For the above code sample, the code just expects something that implements the jQuery API and behavior. It does not have to be jQuery.
- Encapsulates the module definition. Gives you the tools to avoid polluting the global namespace.
- Clear path to defining the module value. Either use "return value;" or the CommonJS "exports" idiom, which can be useful for circular dependencies.

It is an improvement over CommonJS modules because:

- It works better in the browser, it has the least amount of gotchas. Other approaches have problems with debugging, cross-domain/CDN usage, file:// usage and the need for server-specific tooling.
- Defines a way to include multiple modules in one file. In CommonJS terms, the term for this is a "transport format", and that group has not agreed on a transport format.
- Allows setting a function as the return value. This is really useful for constructor functions. In CommonJS this is more awkward, always having to set a property on the exports object. Node supports module.exports = function () {}, but that is not part of a CommonJS spec.

Module Definition § 5

Using JavaScript functions for encapsulation has been documented as the module pattern[52]:

```
(function () {
   this.myGlobal = function () {};
}());
```

That type of module relies on attaching properties to the global object to export the module value, and it is difficult to declare dependencies with this model. The dependencies are assumed to be immediately available when this function executes. This limits the loading strategies for the dependencies.

AMD addresses these issues by:

- Register the factory function by calling define(), instead of immediately executing it.
- Pass dependencies as an array of string values, do not grab globals.
- Only execute the factory function once all the dependencies have been loaded and executed.
- Pass the dependent modules as arguments to the factory function.

  //Calling define with a dependency array and a factory function define([‘dep1’, ‘dep2’], function (dep1, dep2) {

```
//Define the module value by returning a value.
return function () {};
```

  });

Named Modules § 6

Notice that the above module does not declare a name for itself. This is what makes the module very portable. It allows a developer to place the module in a different path to give it a different ID/name. The AMD loader will give the module an ID based on how it is referenced by other scripts.

However, tools that combine multiple modules together for performance need a way to give names to each module in the optimized file. For that, AMD allows a string as the first argument to define():

[52] http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth

```
//Calling define with module ID, dependency array, and factory function
define('myModule', ['dep1', 'dep2'], function (dep1, dep2) {

    //Define the module value by returning a value.
    return function () {};
});
```

You should avoid naming modules yourself, and only place one module in a file while developing. However, for tooling and performance, a module solution needs a way to identify modules in built resources.

Sugar § 7

The above AMD example works in all browsers. However, there is a risk of mismatched dependency names with named function arguments, and it can start to look a bit strange if your module has many dependencies:

```
define([ "require", "jquery", "blade/object", "blade/fn", "rdapi",
         "oauth", "blade/jig", "blade/url", "dispatch", "accounts",
         "storage", "services", "widgets/AccountPanel", "widgets/TabButton",
         "widgets/AddAccount", "less", "osTheme", "jquery-ui-1.8.7.min",
         "jquery.textOverflow"],
function (require,   $,        object,         fn,         rdapi,
          oauth,   jig,         url,         dispatch,   accounts,
          storage,   services,   AccountPanel,           TabButton,
          AddAccount,           less,   osTheme) {

});
```

To make this easier, and to make it easy to do a simple wrapping around CommonJS modules, this form of define is supported, sometimes referred to as “simplified CommonJS wrapping”:

```
define(function (require) {
    var dependency1 = require('dependency1'),
        dependency2 = require('dependency2');

    return function () {};
});
```

The AMD loader will parse out the require(‘’) calls by using Function.prototype.toString(), then internally convert the above define call into this:

```
define(['require', 'dependency1', 'dependency2'], function (require) {
    var dependency1 = require('dependency1'),
        dependency2 = require('dependency2');

    return function () {};
});
```

This allows the loader to load dependency1 and dependency2 asynchronously, execute those dependencies, then execute this function.

Not all browsers give a usable Function.prototype.toString() results. As of October 2011, the PS 3 and older Opera Mobile browsers do not. Those browsers are more likely to need an optimized build of the modules for network/device limitations, so just do a build with an optimizer that knows how to convert these files to the normalized dependency array form, like the RequireJS optimizer[53].

Since the number of browsers that cannot support this toString() scanning is very small, it is safe to use this sugared forms for all your modules, particularly if you like to line up the dependency names with the variables that will hold their module values.

CommonJS Compatibility § 8

Even though this sugared form is referred to as the "simplified CommonJS wrapping", it is not 100% compatible with CommonJS modules. However, the cases that are not supported would likely break in the browser anyway, since they generally assume synchronous loading of dependencies.

Most CJS modules, around 95% based on my (thoroughly unscientific) personal experience, are perfectly compatible with the simplified CommonJS wrapping.

The modules that break are ones that do a dynamic calculation of a dependency, anything that does not use a string literal for the require() call, and anything that does not look like a declarative require() call. So things like this fail:

```
//BAD
var mod = require(someCondition ? 'a' : 'b');

//BAD
if (someCondition) {
    var a = require('a');
} else {
    var a = require('a1');
}
```

[53] http://requirejs.org/docs/optimization.html

These cases are handled by the callback-require[54], `require([moduleName], function (){})` normally present in AMD loaders.

The AMD execution model is better aligned with how ECMAScript Harmony modules are being specified. The CommonJS modules that would not work in an AMD wrapper will also not work as a Harmony module. AMD's code execution behavior is more future compatible.

Verbosity vs. Usefulness

One of the criticisms of AMD, at least compared to CJS modules, is that it requires a level of indent and a function wrapping.

But here is the plain truth: the perceived extra typing and a level of indent to use AMD does not matter. Here is where your time goes when coding:

- Thinking about the problem.
- Reading code.

Your time coding is mostly spent thinking, not typing. While fewer words are generally preferable, there is a limit to that approach paying off, and the extra typing in AMD is not that much more.

Most web developers use a function wrapper anyway, to avoid polluting the page with globals. Seeing a function wrapped around functionality is a very common sight and does not add to the reading cost of a module.

There are also hidden costs with the CommonJS format:

- the tooling dependency cost
- edge cases that break in browsers, like cross domain access
- worse debugging, a cost that continues to add up over time

AMD modules require less tooling, there are fewer edge case issues, and better debugging support.

What is important: being able to actually share code with others. AMD is the lowest energy pathway to that goal.

Having a working, easy to debug module system that works in today's browsers means getting real world experience in making the best module system for JavaScript in the future.

AMD and its related APIs, have helped show the following for any future JS module system:

- **Returning a function as the module value**, particularly a constructor function, leads to better API design. Node has module.exports to allow this, but being able to use "return function (){}" is much cleaner. It means not having to get a handle on "module" to do module.exports, and it is a clearer code expression.

[54] https://github.com/amdjs/amdjs-api/wiki/require

- **Dynamic code loading** (done in AMD systems via require([], function (){}))[55] is a basic requirement. CJS talked about it, had some proposals, but it was not fully embraced. Node does not have any support for this need, instead relying on the synchronous behavior of require(''), which is not portable to the web.
- **Loader plugins**[56] are incredibly useful. It helps avoid the nested brace indenting common in callback-based programming.
- **Selectively mapping one module** to load from another location makes it easy to provide mock objects for testing.
- There should only be at most **one IO action for each module**, and it should be straightforward. Web browsers are not tolerant of multiple IO lookups to find a module. This argues against the multiple path lookups that Node does now, and avoiding the use of a package.json "main" property. Just use module names that map easily to one location based on the project's location, using a reasonable default convention that does not require verbose configuration, but allow for simple configuration when needed.
- It is best if there is an **"opt-in" call** that can be done so that older JS code can partcipate in the new system.

If a JS module system cannot deliver on the above features, it is at a significant disadvantage when compared to AMD and its related APIs around callback-require[57], loader plugins[58], and paths-based module IDs.

AMD Used Today § 9

As of mid October 2011, AMD already has good adoption on the web:

- jQuery[59] 1.7
- Dojo[60] 1.7
- EmbedJS[61]
- Ender[62]-associated modules like bonzo[63], qwery[64], bean[65] and domready[66]
- Used by Firebug[67] 1.8+

[55] https://github.com/amdjs/amdjs-api/wiki/require
[56] http://requirejs.org/docs/plugins.html
[57] https://github.com/amdjs/amdjs-api/wiki/require
[58] https://github.com/amdjs/amdjs-api/wiki/Loader-Plugins
[59] http://jquery.com/
[60] http://dojotoolkit.org/
[61] http://uxebu.github.com/embedjs/
[62] http://ender.no.de/
[63] https://github.com/ded/bonzo
[64] https://github.com/ded/qwery
[65] https://github.com/fat/bean
[66] https://github.com/ded/domready
[67] http://getfirebug.com/

- The simplified CommonJS wrapper can be used in Jetpack/Add-on SDK[68] for Firefox
- Used for parts of sites on the BBC[69] (observed by looking at the source, not an official recommendation of AMD/RequireJS)

What You Can Do § 10

If you write applications:

- Give an AMD loader a try. You have some choices:
  - RequireJS[70]
  - curl[71]
  - lsjs[72]
  - Dojo[73] 1.7+
- If you want to use AMD but still use the **load one script at the bottom of the HTML page** approach:
  - Use the RequireJS optimizer[74] either in command line mode or as an HTTP service[75] with the almond AMD shim[76].

If you are a script/library author:

- Optionally call define()[77] if it is available. The nice thing is you can still code your library without relying on AMD, just participate if it is available. This allows consumers of your modules to:
  - avoid dumping global variables in the page
  - use more options for code loading, delayed loading
  - use existing AMD tooling to optimize their project
  - participate in a workable module system for JS in the browser today.

If you write code loaders/engines/environments for JavaScript:

[68] https://addons.mozilla.org/en-US/developers/docs/sdk/1.1/
[69] http://www.bbc.co.uk/
[70] http://requirejs.org/
[71] https://github.com/unscriptable/curl
[72] https://github.com/zazl/lsjs
[73] http://dojotoolkit.org/
[74] http://requirejs.org/docs/optimization.html
[75] https://github.com/jrburke/r.js/blob/master/build/tests/http/httpBuild.js
[76] https://github.com/jrburke/almond
[77] https://github.com/umdjs/umd

- Implement the AMD API[78]. There is a discussion list[79] and compatibility tests[80]. By implementing AMD, you will reduce multi-module system boilerplate and help prove out a workable JavaScript module system on the web. This can be fed back into the ECMAScript process to build better native module support.
- Also support callback-require[81] and loader plugins[82]. Loader plugins are a great way to reduce the nested callback syndrome that can be common in callback/async-style code.

JavaScript Dependency Injection

Original Article

http://merrickchristensen.com/articles/javascript-dependency-injection.html

Merrick Christensen, merrickchristensen.com[a]

[a] http://merrickchristensen.com/

Inversion of control[83] and more specifically dependency injection[84] have been growing in popularity in the JavaScript landscape thanks to projects like Require.js[85] and AngularJS[86]. This article is a brief introduction to dependency injection and how it fits into JavaScript. It will also demystify the elegant way AngularJS implements dependency injection.

Dependency Injection In JavaScript

Dependency injection facilitates better testing by allowing us to mock dependencies in testing environments so that we only test one thing at a time. It also enables us to write more maintainable code by decoupling our objects from their implementations.

With dependency injection, your dependencies are given to your object instead of your object creating or explicitly referencing them. This means the dependency injector can provide a different dependency based on the context of the situation. For example, in your tests it might pass a fake version of your services API that doesn't make requests but returns static objects instead, while in production it provides the actual services API.

[78] https://github.com/amdjs/amdjs-api/wiki/AMD
[79] https://groups.google.com/group/amd-implement
[80] https://github.com/amdjs/amdjs-tests
[81] https://github.com/amdjs/amdjs-api/wiki/require
[82] https://github.com/amdjs/amdjs-api/wiki/Loader-Plugins
[83] http://en.wikipedia.org/wiki/Inversion_of_control
[84] http://en.wikipedia.org/wiki/Dependency_injection
[85] http://requirejs.org/
[86] http://angularjs.org/

Another example could be to pass ZeptoJS[87] to your view objects when the device is running Webkit[88] instead of jQuery[89] to improve performance.

The main benefits experienced by adopting dependency injection are as follows:

1. Code tends to be more maintainable.
2. APIs are more elegant and abstract.
3. Code is easier to test.
4. Code is more modular and reuseable.
5. Cures cancer. (Not entirely true.)

Holding dependencies to an API based contract becomes a natural process. Coding to interfaces is nothing new, the server side world has been battle testing this idea for a long time to the extent that the languages themselves implement the concept of interfaces. In JavaScript we have to force ourselves to do this. Fortunately dependency injection and module systems are a welcome friend.

Now that you have some idea of what dependency injection is, lets take a look at how to build a simple implementation of a dependency injector using AngularJS style dependency injection[90] as a reference implementation. This implementation is purely for didactic purposes.

AngularJS Style Injection

AngularJS is one of the only front end JavaScript frameworks that fully adopts dependency injection right down to the core of the framework. To a lot of developers the way dependency injection is implemented in AngularJS looks completely magic.

When creating controllers in AngularJS, the arguments are dependency names that will be injected into your controller. The argument names are the key here, they are leveraged to map a dependency name to an actual dependency. Yeah, the word "key" was used on purpose, you will see why.

```
1                               /* Injected */
2  var WelcomeController = function (Greeter) {
3    /** I want a different Greeter injected dynamically. **/
4      Greeter.greet();
5  };
```

[87] http://zeptojs.com/
[88] http://www.webkit.org/
[89] http://jquery.com/
[90] http://docs.angularjs.org/guide/di

Basic Requirements

Lets explore some of the requirements to make this function work as expected.

1. The dependency container needs to know that this function wants to be processed. In the AngularJS world that is done through the Application object and the declarative HTML bindings. In our world we will explicitly ask our injector to process a function.
2. It needs to know what a Greeter before it can inject it.

Requirement 1: Making the injector aware.

To make our dependency injector aware of our WelcomeController we will simply tell our injector we want a function processed. Its important to know AngularJS ultimately does this same thing just using less obvious mechanisms whether that be the Application object or the HTML declarations.

```
var Injector = {
    process: function(target) {
        // Time to process
    }
};

Injector.process(WelcomeController);
```

Ok, now that the Injector has the opportunity to process the WelcomeController we can figure out what dependencies the function wants, and execute it with the proper dependencies. This process is called dependency resolution. Before we can do that we need a way to register dependencies with our Injector object...

Requirement 2: Registering dependencies

We need to be able to tell the dependency injector what a `Greeter` is before it can provide one. Any dependency injector worth it's bits will allow you to describe *how* it is provided. Whether that means being instantiated as a new object or returning a singleton. Most injection frameworks even have mechanisms to provide a constructor some configuration and register multiple dependencies by the same name. Since our dependency injector is just a simplified way to show how AngularJS does dependency mapping using parameter names, we won't worry about any of that.

Without further excuses, our simple register function:

```
Injector.dependencies = {};

Injector.register = function(name, dependency) {
    this.dependencies[name] = dependency;
};
```

All we do is store our dependency by name so the injector knows what to provide when certain dependencies are requested. Lets go ahead and register an implementation of Greeter.

```
var RobotGreeter = {
  greet: function() {
    return 'Domo Arigato';
  }
};

Injector.register('Greeter', RobotGreeter);
```

Now our injector knows what to provide when `Greeter` is specified as a dependency.

Moving Forward

The building blocks are in place it's time for the sweet part of this article. The reason I wanted to post this article in the first place, the nutrients, the punch line, the hook, the call toString() with some sweet reflection. This is where the magic is, in JavaScript we don't have to execute a function immediately. The trick is to call toString on your function which returns the function as a string, this gives a chance to preprocess our functions as strings and turn them back into functions using the Function constructor, or just execute them with the proper parameters after doing some reflection. The latter is exactly what we will do here.

toString Returns Winning

```
var WelcomeController = function (Greeter) {
    Greeter.greet();
};

// Returns the function as a string.
var processable = WelcomeController.toString();
```

You can try it in your console!

Now that we have the WelcomeController as a string we can do some reflection to figure out which dependencies to inject.

Dependency Checking

It's time to implement the process method of our Injector. First lets take a look at injector.js[91] from Angular. You'll notice the reflection starts on line 54[92] and leverages a few regular expressions to parse the function. Let's take a look at the regular expression, shall we?

```
var FN_ARGS = /^function\s*[^\(]*\(\s*([^\)]*)\)/m;
```

The `FN_ARGS` regular expression is used to select everything inside the parentheses of a function defintion. In other words the parameters of a function. In our case, the dependency list.

```
var args = WelcomeController.toString().match(FN_ARGS)[1];
console.log(args); // Returns Greeter
```

Pretty neat, right? We have now parsed out the WelcomeController's dependency list in our Injector *prior* to executing the WelcomeController function! Suppose the WelcomeController had multiple dependencies, this isn't terribly problematic since we can just split the arguments with a comma!

```
var MultipleDependenciesController = function(Greeter, OtherDependency) {
  // Implementation of MultipleDependenciesController
};

var args = MultipleDependenciesController
  .toString()
  .match(FN_ARGS)[1]
  .split(',');

console.log(args); // Returns ['Greeter', 'OtherDependency']
```

The rest is pretty straight forward, we just grab the requested dependency by name from our `dependencies` cache and call the target function passing the requested dependencies as arguments. Lets implement the function that maps our array of dependency names to their dependencies:

[91] https://github.com/angular/angular.js/blob/master/src/auto/injector.js
[92] https://github.com/angular/angular.js/blob/master/src/auto/injector.js#L54

```
Injector.getDependencies = function(arr) {
  var self = this;
  return arr.map(function(dependencyName) {
    return self.dependencies[dependencyName];
  });
};
```

The `getDependencies` method takes the array of dependency names and maps it to a corresponding array of actual dependencies. If this map function is foreign to you check out the Array.prototype.map documentation[93].

Now that we have implemented our dependency resolver we can head back over to our `process` method and execute the target function with it's proper dependencies.

```
target.apply(target, this.getDependencies(args));
```

Pretty awesome, right?

Injector.js

```
var Injector = {

    dependencies: {},

    process: function(target) {
        var FN_ARGS = /^function\s*[^\(]*\(\s*([^\)]*)\)/m;
        var text = target.toString();
        var args = text.match(FN_ARGS)[1].split(',');

        target.apply(target, this.getDependencies(args));
    },

    getDependencies: function(arr) {
        var self = this;
        return arr.map(function(value) {
            return self.dependencies[value];
        });
    },

    register: function(name, dependency) {
        this.dependencies[name] = dependency;
```

[93] https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array/map

```
22      }
23
24  };
```

Example & Excuses

You can see the functioning injector we created in this example[94] on jsFiddle.

This contrived example is not something you would use in an actual codebase it was simply created to demonstrate the rich functionality JavaScript provides and to explain how AngularJS provides dependency injection. If this interests you I highly recommend reviewing their code further. It's important to note this approach is not novel. Other projects use toString to preprocess code, for example Require.js[95] uses a similar approach to parse and transpile CommonJS style modules to AMD style modules.

I hope you found this article enlightening and continue to explore dependency injection and how it applies to the client side world.

I really think there is something special brewing here.

[94] http://jsfiddle.net/nMK6j/
[95] http://requirejs.org/

Tools

10 Must Have JavaScript Tools For Developers

Original Article

http://smashinghub.com/javascript-tools-for-developers.htm

Ali Qayyum, smashinghub.com[a]

[a] http://smashinghub.com/author/admin

JavaScript tools are still one of the most difficult issues when it comes to web-development. Many tools and applications could make your developing life pretty fast and simple. A few years back people only used JavaScript to add some effects like the blinking text etc but in the then people started taking it a lot more seriously, as it can enhance the web. So here are 10 great tools and script for all JavaScript developers.

FitText.js

FitText

This great tool does automatic resizing of a text considering the size of its parent element. Just view the website and resize your browser: The text will fit. A very good tool for modern sites and apps!

http://fittextjs.com/[96]

MicroJS

[96] http://fittextjs.com/

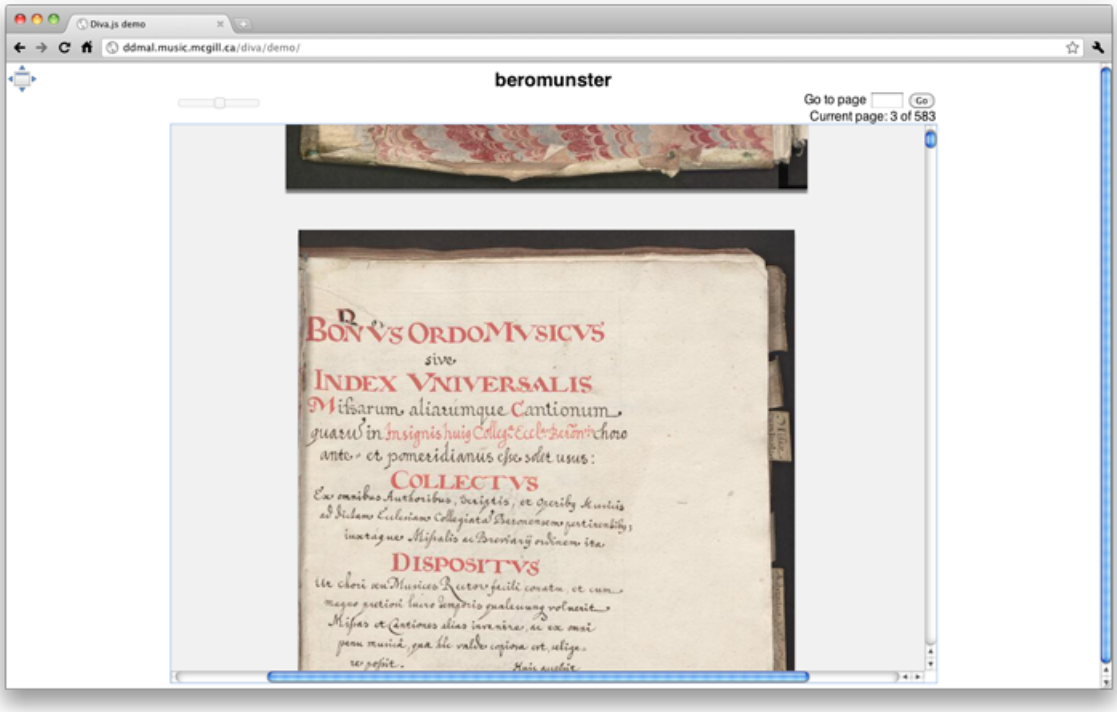
MicroJS

This is a new website that can provide you with a JavaScript code snippet quickly. It has JavaScript codes for almost all the common tasks: Ajax, Json, DOM and Object-Oriented JavaScript etc. Definitely a must have in your bookmarks!

http://microjs.com/[97]

Diva.js

[97] http://microjs.com/



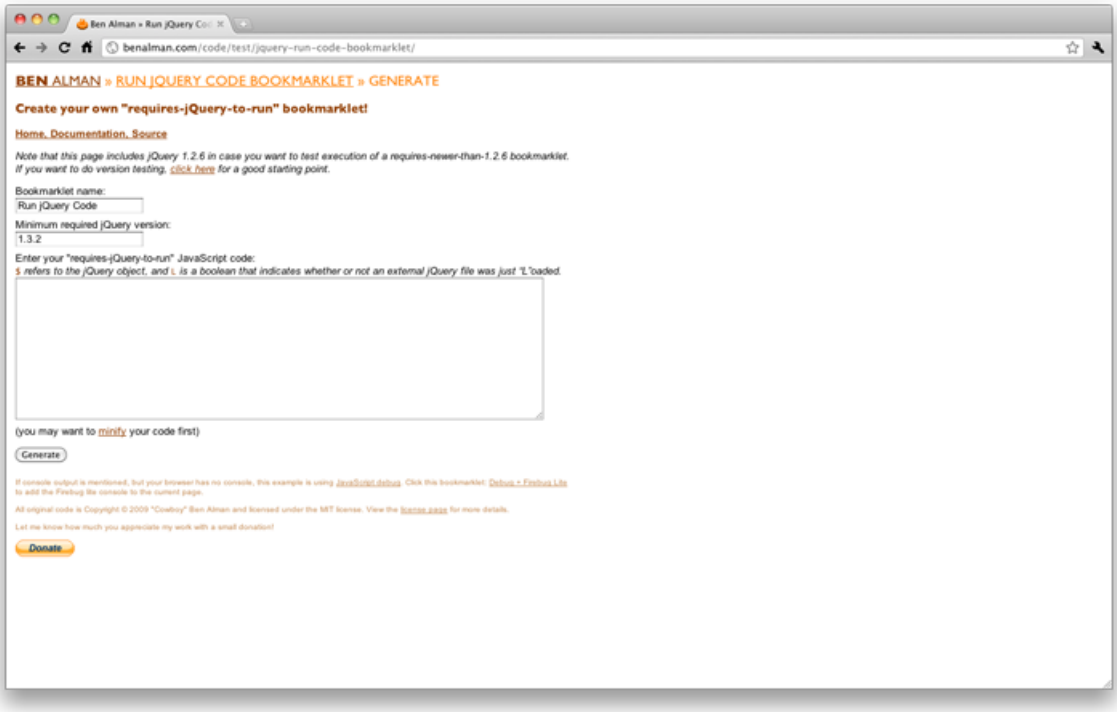
DivaJS

It is a Javascript frontend that lets you view documents. It is designed to work with digital libraries to display multi-page documents as a single, undivided item. It has been designed to work with a IIPImage server and will prove to be a great tool for those who work on library or bookstore websites. A demo is given here if you want to check it out.

http://ddmal.music.mcgill.ca/diva/[98]

Bookmarklet Generator

[98] http://ddmal.music.mcgill.ca/diva/



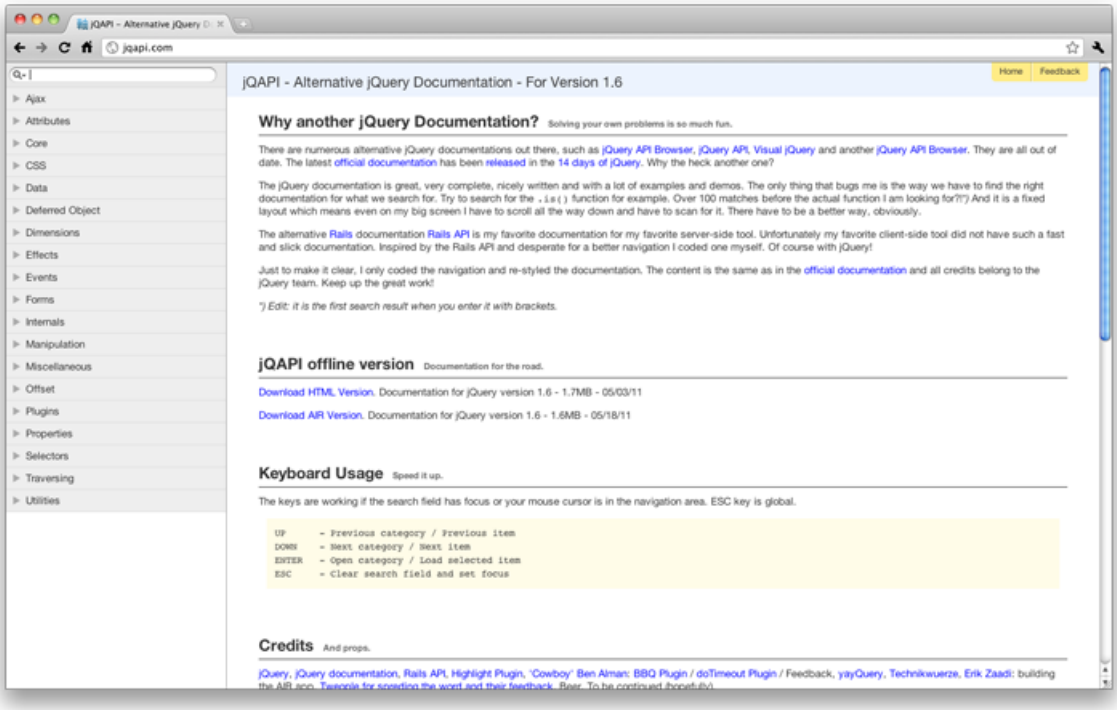
Bookmarklet

As the name suggests it is a bookmarklet generator: All you have to do is to paste your regular JavaScript code, press the button and you will get a bookmarklet ready to be installed on your browser bar.

http://benalman.com/code/test/jquery-run-code-bookmarklet/[99]

jQAPI

[99] http://benalman.com/code/test/jquery-run-code-bookmarklet/



jQAPI

It is an amazing site which provides the jQuery documentation in a very user-friendly way, so it can become your reference website when you need any jQuery help.

http://jqapi.com[100]

heatmap.js

[100] http://jqapi.com/

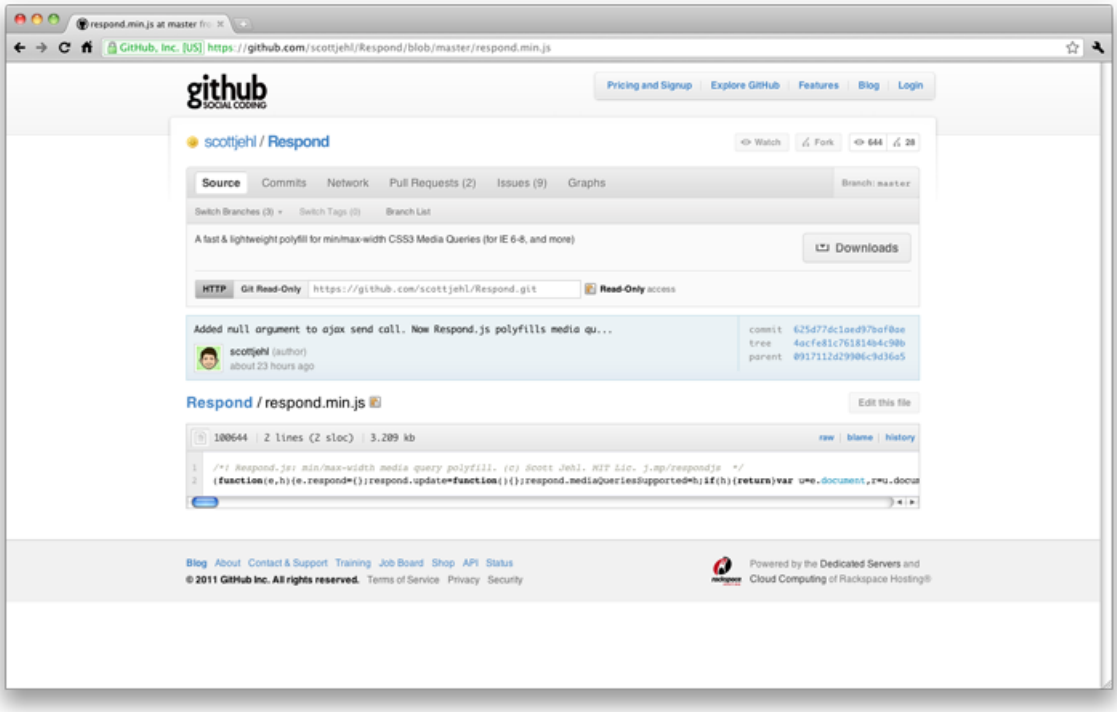
heatmap.js

JavaScript presents endless possibilities: For instance, heatmap.js lets you generate web heatmaps with the html5canvas element based on your data. Simple and sophisticated!

http://www.patrick-wied.at/static/heatmapjs/[101]

Respond.js

[101] http://www.patrick-wied.at/static/heatmapjs/



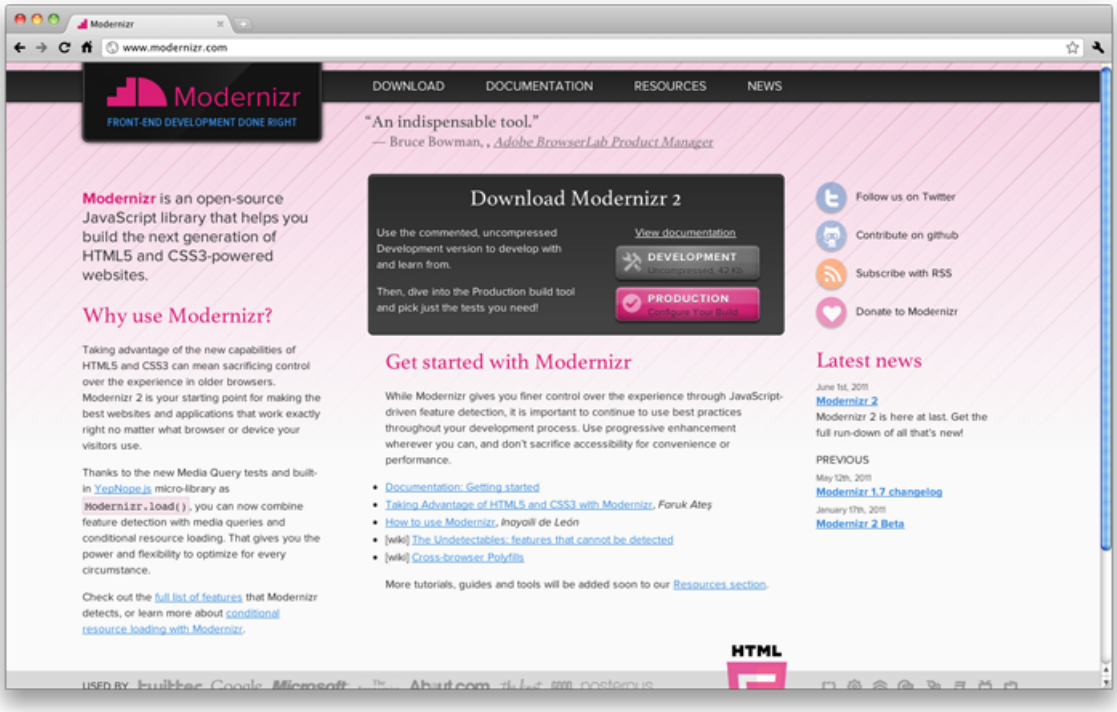
Respond.js

This one is a tiny script that lets you use CSS3 media queries on browsers that don't support it yet (like IE!)

https://github.com/scottjehl/Respond/blob/master/respond.min.js[102]

Modernizr

[102] https://github.com/scottjehl/Respond/blob/master/respond.min.js



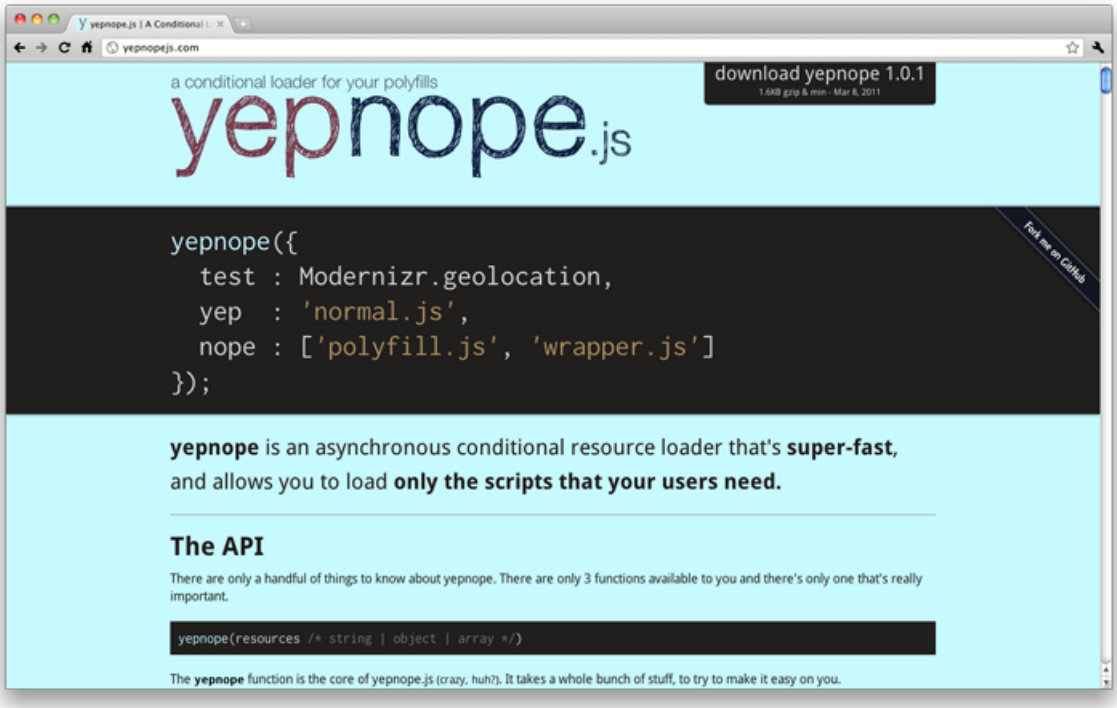
Modernizr

It is a script that lets older browsers to work almost as nicely as newest ones, so you can create modern apps that will work on IE6 and 7. Your clients are going to love it for sure.

http://www.modernizr.com/[103]

YepNope

[103] http://www.modernizr.com/



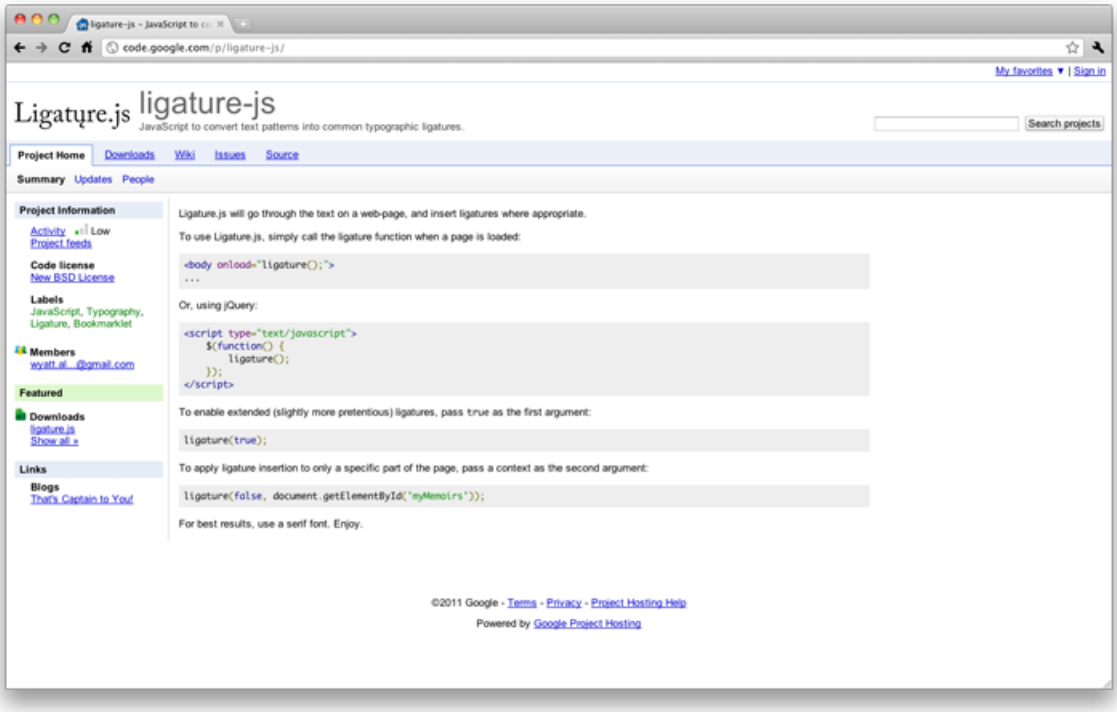
YepNope

As the name hints, it answers yep, or nope. For instance, ask YepNope is Modernizr is loaded. If yes, ask YepNope to do this, and if not, ask YepNope to do that. Simple and very useful!

http://yepnopejs.com/[104]

Ligature.js

[104] http://yepnopejs.com/



Ligature

The name says it all. Ligature.js is a script that adds amazing ligatures to any type of text. A must have if you are a typography lover.

http://code.google.com/p/ligature-js/[105]

Useful Node.js Tools, Tutorials And Resources

Original Article

http://coding.smashingmagazine.com/2011/09/16/useful-node-js-tools-tutorials-and-resources/

[Luca Degasperi, http://coding.smashingmagazine.com/[a]

[a] http://coding.smashingmagazine.com/

Created by Ryan Dahl in 2009, Node.js[106] is a relatively new technology which has gained a lot of popularity among Web developers recently. However, not everyone knows what it really is.

[105] http://code.google.com/p/ligature-js/

[106] http://nodejs.org/

Node.js is essentially a server-side JavaScript environment that uses an asynchronous event-driven model. What this means is simple: it's an environment which is intended for writing scalable, high performance network applications. It's like Ruby's Event Machine or Python's Twisted, but it takes the event model a bit further—it presents the event loop as a language construct instead of as a library.

And that's not all: what's really great about Node.js is the thousands of modules available for any purpose, as well as the vibrant community behind this young project. In this round-up, you will find the **most useful resources for Node.js**, from handy tools to detailed tutorials, not to mention in-depth articles and resources on this promising technology. Do you use Node.js already? Let us know in the comments to this post!

Useful Node.js Tools

Node Express Boilerplate[107]
Node Express Boilerplate gives the developer a clean slate, while bundling enough useful features to remove all of those redundant tasks that can derail a project before it even gets started.

Socket.IO[108]
Socket.IO is a cross-browser Web socket that aims to make real-time apps possible in every browser and mobile device, blurring the distinctions between the various transport mechanisms. It's care-free real time, in JavaScript.

Mastering Node[109]
With Mastering Node, you can write high-concurrency Web servers, using the CommonJS module system, Node.js's core libraries, third-party modules, high-level Web development and more.

Log.io[110]
Your infrastructure may have hundreds of log files spread across dozens of machines. To help you monitor deployments and troubleshoot, Log.io lets you instantly see composite streams of log messages in a single user interface.

Formaline[111]
Formaline is a low-level, full-featured (Node.js) module for handling form requests (HTTP POSTs and PUTs) and for parsing uploaded files quickly. It is also ready to use with, for example, middleware such as Connect.

LDAPjs[112]
LDAPjs is a pure-JavaScript, from-scratch framework for implementing LDAP clients and servers in Node.js. It is intended for developers who are used to interacting with HTTP services in Node.js and Express.

[107] https://github.com/mape/node-express-boilerplate
[108] http://socket.io/
[109] http://visionmedia.github.com/masteringnode/
[110] http://logio.org/
[111] https://github.com/rootslab/formaline
[112] http://ldapjs.org/

Node Supervisor[113]

This is a little supervisor script for Node.js. It runs your program and watches for code changes, so you can have hot-code reloading-ish behavior without worrying about memory leaks or having to clean up all of the inter-module references, and without a whole new `require` system.

Stylus: An Expressive CSS Markup Language for Node.js[114]

Stylus is supposed to be an expressive language that can be converted to CSS. The compiler is written in Node.js.

Jade – Template Engine[115]

Jade is a template engine for Node.js applications. It combines great power and flexibility with a nice and clean syntax.

Express[116]

This is a Sinatra-inspired Web development framework for Node.js: fast, flexible and sexy.

Hook.io[117]

hook.io creates a distributed node.js EventEmitter that works cross-process / cross-platform / cross-browser. Think of it like a real-time event bus that works anywhere JavaScript is supported.

Node Package Manager[118]

NPM is a package manager for node. You can use it to install and publish your node programs. It manages dependencies and does other cool stuff.

Node-QRcode[119]

Despite being quite young, Node.js already has a huge number of libraries for every possible application. This one is a QR code generator.

NWM[120]

NWM is a dynamic window manager for X that was written at NodeKO 2011. It uses libev to interface with X11, and it allows you to lay out windows in Node.js.

Bricks.js[121]

Bricks.js is an advanced modular Web framework built on Node.js. It is highly flexible. Bricks.js can be used as a standalone static Web server, a basic routing framework or a multi-level Apache-like routing system; and it is modular enough to have the capability to completely switch out its routing engine.

Node.js Modules[122]

[113] https://github.com/isaacs/node-supervisor
[114] http://learnboost.github.com/stylus/
[115] http://jade-lang.com/
[116] http://expressjs.com/
[117] https://github.com/hookio/hook.io
[118] http://npmjs.org/
[119] https://github.com/soldair/node-qrcode
[120] https://github.com/mixu/nwm
[121] http://bricksjs.com/index.html
[122] https://github.com/joyent/node/wiki/modules#templating

A list of almost all the Node.js most famous modules organized by category. This list definitively is worth a look.

90 open-source Node.js modules[123]
Recently, Browserling released over 90 Node.js modules to the open-source community. Some of them are small and strange modules, others might be pretty useful for your next Node.js project.

Calipso[124]
Calipso is a content management system (CMS) based on the NodeJS server.

PDFKit[125]
PDFKit is a PDF document-generation library for Node.js that makes it easy to create complex, multi-page, printable documents. It is written in pure CoffeeScript, but you can use the API in plain 'ol JavaScript if you like. The API embraces chain-ability, and it includes both low-level functions as well as abstractions for higher-level functionality.

Forever[126]
A simple CLI tool to ensure that a given script runs continuously (i.e. forever).

Introducing Node.js

Node.js Step by Step[127]
Node.js is an amazing new technology, but unless you're a JavaScript developer, the process of becoming acquainted with it can quickly become a bit overwhelming. If you want to learn how to use Node.js, this set of articles and screencasts might do the trick.

What Is Node.js?[128]
Another interesting discussion on StackOverflow about what Node.js is and is not. Recommended for those who are approaching Node.js for the first time.

Learning Server-Side JavaScript[129]
Node.js is all the buzz at the moment, and it makes creating high-performance, real-time Web applications easy. It allows JavaScript to be used end to end, on both the server and client. This tutorial walks you through from installing Node.js and writing your first "Hello World" program to building a scalable streaming Twitter server.

Node.js Is Important: An Introduction[130]
"Once in a while, you come across a technology and are blown away by it. You feel that something like this should have been around much earlier and that it will be a significant milestone, not just in your own life as a developer but in general.

[123] http://www.catonmat.net/blog/browserling-open-sources-90-node-modules/
[124] http://calip.so/
[125] http://pdfkit.org/
[126] https://github.com/indexzero/forever
[127] http://net.tutsplus.com/tutorials/javascript-ajax/this-time-youll-learn-node-js/
[128] http://stackoverflow.com/questions/1884724/what-is-node-js
[129] http://net.tutsplus.com/tutorials/javascript-ajax/learning-serverside-javascript-with-node-js/
[130] http://www.pavingways.com/nodejs-node-is-important-introduction_1618.html

The Secrets of Node's Success[131]
In the short time since its initial release in late 2009, Node.js has captured the interest of thousands of experienced developers, grown a package manager and a corpus of interesting modules and applications, and even spawned a number of start-ups. What is it about this technology that makes it interesting to developers? And why has it succeeded while other server-side JavaScript implementations linger in obscurity or fail altogether?

Asynchronous Code Design with Node.js[132]
The asynchronous event-driven I/O of Node.js is currently evaluated by many enterprises as a high-performance alternative to the traditional synchronous I/O of multi-threaded enterprise application server. The asynchronous nature means that enterprise developers have to learn new programming patterns, and unlearn old ones

A Giant Step Backwards?[133]
In this article, Fenn Bailey expresses his opinion of Node.js and why he sometimes thinks Node.js is a step backward compared to other solutions.

Node.js Is Backwards[134]
A hot topic in computing is parallel programming in languages such as Erlang. Will JavaScript join the party?

Videos And Screencasts On Node.js

Node.js Meetup: Distributed Web Architectures[135]
A series of videos from the Node.js Meetup at Joyent headquarters, discussing how to build distributed Web architectures with Node.js.

Introduction to Node.js with Ryan Dahl[136]
In this presentation Ryan Dahl, the man behind Node.js will introduce you to this event-driven I/O framework with a few examples showing Node.js in action.

SenchaCon 2010: Server-side JavaScript with Node, Connect and Express on Vimeo[137]
Node.js has unleashed a new wave of interest in server side Javascript. In this session, you'll learn how to get productive with node.js by leveraging Connect and Express node middleware.

Technical Articles And Tutorials On Node.js

Proxying HTTP and Web Sockets in Node[138]
This guide is geared to beginners and people who are unfamiliar with reverse HTTP proxying,

[131] http://radar.oreilly.com/2011/06/node-javascript-success.html
[132] http://shinetech.com/thoughts/thought-articles/139-asynchronous-code-design-with-nodejs
[133] http://fennb.com/nodejs-a-giant-step-backwards
[134] http://blog.ankurgoyal.com/post/6433642218/node-js-is-backwards
[135] http://joyeur.com/2011/08/11/node-js-meetup-distributed-web-architectures/
[136] http://www.youtube.com/watch?v=jo_B4LTHi3I
[137] http://vimeo.com/18077379
[138] http://blog.nodejitsu.com/http-proxy-intro

Web socket proxying, load balancing, virtual host configuration, request forwarding and other Web proxying concepts.

Bulletproof Node.js Coding[139]
"Right around the time that I started the third refactoring/rewrite of the code, I felt like I had gotten a feel for how to write bulletproof code, and I thought it would be worth sharing some of the style and conventions I came to adopt."

How to Write a Native Node.js Extension[140]
In this tutorial, you will learn how to write a native Node.js extension the right way, from the very basics to packaging the extension for NPM.

Let's Make a Web App: Nodepad[141]
This series will walk you through building a Web app with Node.js, covering all of the major areas you'll face when building your own applications.

HTML5 Canvas Drawing with Web Sockets, Node.JS and Socket.io[142]
Web sockets and canvas are two really cool features that are currently being implemented in browsers. This tutorial gives you a quick rundown of how they both work, and you'll create a real-time drawing canvas that is powered by Node.js and Web sockets.

Developing Multiplayer HTML5 Games with Node.js[143]
Inspired by the famous iOS game Osmos, developer Boris Smus has created an alternative version of the game using HTML5 canvas and Node.js. This article explains the main phases of the project.

Deploying Node.js on Amazon EC2[144]
Amazon's EC2 is a popular choice for cloud applications. This tutorial shows how Node.js can be deployed on an EC2 instance.

A Simple Node.js + CouchDB Calendar[145]
In this tutorial by Chris Storm, you will learn how to build a Web calendar with Node.js and CouchDB.

IIS7[146]
The IISnode project provides a native IIS 7.x module that enables hosting of Node.js applications on IIS. The project uses the Windows build of node.exe, which has recently seen major improvements.

Node.js + Phone to Control a Browser Game[147]
Someone wondered how easily a smart phone – specifically using its gyroscopes and accelerometers – could be used as a controller for a multi-player game on a larger screen. With a bit of Node.js and HTML5 magic, it turned out to be pretty simple.

[139] http://stella.laurenzo.org/2011/03/bulletproof-node-js-coding/
[140] http://syskall.com/how-to-write-your-own-native-nodejs-extension
[141] http://dailyjs.com/2010/11/01/node-tutorial/
[142] http://wesbos.com/html5-canvas-websockets-nodejs/
[143] http://smus.com/multiplayer-html5-games-with-node
[144] http://blog.carbonfive.com/2011/09/01/deploying-node-js-on-amazon-ec2/
[145] http://japhr.blogspot.com/2011/08/simple-nodejs-couchdb-calendar.html
[146] http://tomasz.janczuk.org/2011/08/hosting-nodejs-applications-in-iis-on.html
[147] http://cykod.com/blog/post/2011-08-using-nodejs-and-your-phone-to-control-a-browser-game

Is There a Template Engine for Node.js?[148]
An engaging discussion appeared on StackOverflow about the template engines that are available for Node.js. Really useful arguments came out of this discussion.

Blogs, Podcasts, Resources On Node.js

How to Node[149]
How to Node is a community-supported blog created by Tim Caswell. Its purpose is to teach how to do various tasks in Node.js and the fundamental concepts needed to write effective code.

Nodejitsu[150]
A really interesting blog about scaling Node.js apps in the cloud and about the Node.js events in general.

Node Up[151]
A podcast that reviews Node.js, explains its philosophy and goes over many of its popular libraries.

Node Tuts[152]
Free screencast tutorials.

Minute With Node.js[153]
Node.js is constantly changing and growing with each new version. New libraries and frameworks are coming out daily that allow you to write JavaScript for new and exciting projects that were previously impossible. This is a one-stop shop for news updates on the entire Node.js eco-system, with a heavy slant on hardcore nerdery.

Felix's Node.js Guide[154]
Over the past few months, Felix have given a lot of talks and done a lot of consulting on Node.js. He found himself repeating a lot of things over and over, so he used some of his recent vacation to start this opinionated and unofficial guide to help people getting started in Node.js.

Node.js Knockout[155]
Node.js Knockout is a 48-hour hackathon for Node.js. It's an online virtual competition, with contestants worldwide.

References And Books

Node.JS Help Sheet[156]
"Node.JS is an evented I/O framework for the V8 JavaScript engine. It's ideal for writing scalable

[148] http://stackoverflow.com/questions/1787716/is-there-a-template-engine-for-node-js
[149] http://howtonode.org/
[150] http://blog.nodejitsu.com/
[151] http://nodeup.com/
[152] http://nodetuts.com/
[153] http://node.minutewith.com/
[154] http://nodeguide.com/
[155] http://nodeknockout.com/
[156] http://www.gosquared.com/liquidicity/archives/1930

network programs, such as Web servers. We've been working on some exciting things with Node.js, and we felt it was only fair to share our knowledge in the form of an easy-to-read Help Sheet."

The Node Beginner Book[157]
The aim of this document is to get you started with developing applications for Node.js. It teaches you everything you need to know about advanced JavaScript along the way. It goes way beyond your typical "Hello World" tutorial.

Up and Running With Node.js[158]
"Many people use the JavaScript programming languages extensively for programming the interfaces of websites. Node.js allows this popular programming language to be applied in many more contexts, in particular on Web servers. There are several notable features about Node.js that make it worthy of interest."

Control the Complexity of Your JavaScript Functions with JSHint

Original Article

http://www.elijahmanor.com/2012/09/control-complexity-of-your-javascript.html

Elijah Manor, elijahmanor.com[a]

[a] http://www.elijahmanor.com/

New JSHint Features

Many of you are aware of the JSHint code quality tool that has been around for the past couple of years. As of recently, the following new options that have been added regarding the complexity of a function.

- `maxparams`
- `maxdepth`
- `maxstatements`
- `maxcomplexity`

[157] http://nodebeginner.org/index.html
[158] http://ofps.oreilly.com/titles/9781449398583/index.html

Parameters, Depth, and Statements

By reducing the number of parameters, the number of nesting, and the number of statements in your function you can dramatically increate the readability and modularity of your code.

The following piece of code shows using the `maxparams`, `maxdepth`, and `maxstatements` to warn us of possible issue with our functions.

```
/*jshint maxparams:3, maxdepth:2, maxstatements:5 */
/*global console:false */

(function( undefined ) {
    "use strict";

    function test1( arg1, arg2, arg3, arg4 ) {
        console.log( "too many parameters!" );
        if ( arg1 === 1 ) {
            console.log( arg1 );
            if ( arg2 === 2 ) {
                console.log( arg2 );
                if( arg3 === 3 ) {
                    console.log( "too much nesting!" );
                    console.log( arg3 );
                    console.log( arg4 );
                }
            }
        }
        console.log( "too many statements!" );
    }

    test1( 1, 2, 3, 4 );
}());
```

JSHint gives an error that too many parameters were used because we limited `maxparams` to 3 and the code accepted 4 parameters. An error occurred because the depth of logic is too deep because we limited it to a `maxdepth` of 2. We will also get an error about the number of lines in our function because we limited `maxstatements` to 5 and we have many more than that.

Cyclomatic Complexity

A less commonly known software metric used to evaluate functions is Cyclomatic Complexity. Like it sounds, it's purpose is to calculate the overall intricacy of a function and to give a score that reflects it's complexity.

> "The cyclomatic complexity of a section of source code is the count of the number of linearly independent paths through the source code." –http://en.wikipedia.org/wiki/Cyclomatic_complexity

In addition to the above parameters, depth, and statement metrics you can now track the overall complexity using the `maxcomplexity` option. "' /*jshint maxcomplexity:3 */ /global console:false */

(function(undefined) { "use strict";

```
function isPrime( number ) {
    var prime = true, i;

    if ( isNumber( number ) ) {
        for ( i = 2; i < number; i++ ) {
          if ( number % i === 0 ) {
              prime = false;
          }
        }
    } else {
        prime = false;
    }

    return prime;
}

function isNumber( number ) {
    return !isNaN( parseFloat( number ) ) && isFinite( number );
}

console.log( isPrime( 109 ) );
```

}()); "'

As you see above, the above function has more complexity that what we set in `maxcomplexity`.

You might be wondering what a reasonable `maxcomplexity` value is for your project. In the 2nd edition of Steve McConnell's Code Complete he recommends that a cyclomatic complexity from 0 to 5 is typically fine, but you should be aware if the complexity starts to get in the 6 to 10 range. He further explains that anything over a complexity of 10 you should strongly consider refactoring your code.

Global Options

Instead of adding these options to the top of each and every JavaScript file you an instead use a `.jshintrc` file in your project and JSHint should pick up those settings. This is handy if your project is large and you want some consistent settings across the board.

```
{
    "globals": {
        "console": false,
        "jQuery": false,
        "_": false
    },
    "maxparams": 5,
    "maxdepth": 5,
    "maxstatements": 25,
    "maxcomplexity": 10,
    "es5": true,
    "browser": true,
    "boss": false,
    "curly": false,
    "debug": false,
    "devel": false,
    "eqeqeq": true,
    "evil": true,
    "forin": false,
    "immed": true,
    "laxbreak": false,
    "newcap": true,
    "noarg": true,
    "noempty": false,
    "nonew": false,
    "nomen": false,
    "onevar": true,
    "plusplus": false,
    "regexp": false,
    "undef": true,
    "sub": true,
    "strict": false,
    "white": true,
    "unused": true
}
```

Testing

Leaner, Meaner, Faster Animations with requestAnimationFrame

Original Article

http://www.html5rocks.com/en/tutorials/speed/animations

Paul Lewis, html5rocks.com[a]

[a] http://www.html5rocks.com

It's a fair bet you've done some animation work in your time as a developer, whether that's smaller UI effects or large iteractive canvas pieces. Chances are you've also come across `requestAnimationFrame`, or rAF (we say it *raff* around these parts), and hopefully you've had a chance to use it in your projects. In case you don't know, `requestAnimationFrame` is the browser's native way of handling your animations. Because rAF is specifically designed to deal with animation and rendering, the browser can schedule it at the most appropriate time and, if we play our cards right, it will help us get a buttery smooth 60 frames per second.

What we want to do in this article is **outline some additional ways to ensure you're getting the maximum benefit from your animation code**. Even if you're using `requestAnimationFrame` there are other ways you can end up with bottlenecks in your animations. At 60 frames per second each frame that you draw has 16.67ms to get everything done. That's not a lot, so every optimisation counts!

> **TL;DR** Decouple your events from animations; avoid animations that result in reflow-repaint loops; update your rAF calls to expect a high resolution timestamp as the first parameter; only call rAF when you have visual updates to do.

Debouncing Scroll Events

Debouncing is the process of decoupling your animation from any inputs that affect it. Take, for example, an effect that is triggered when you scroll the page. The effect might check if some DOM elements are visible to the user and then, if they are, apply some CSS classes to those elements.

Or maybe you're coding a parallax scrolling effect where, as you scroll, background images change their position relative to page's scroll position. I'll go with the former of the two common uses, and the general gist of our code might be:

```
function onScroll() {
        update();
}

function update() {

        // assume domElements has been declared
        // by this point :)
        for(var i = 0; i < domElements.length; i++) {

                // read offset of DOM elements
                // to determine visibility - a reflow

                // then apply some CSS classes
                // to the visible items        - a repaint

        }
}

window.addEventListener('scroll', onScroll, false);
```

The main issue here is that we are triggering a reflow and repaint whenever we get a scroll event: we ask the browser to recalculate the **real positions** of DOM elements, an expensive reflow operation, and then we apply some CSS classes, which causes the browser to repaint. We end up ping-ponging between reflowing and repainting, and this is going to undermine your app's performance. We're picking on scroll events here, but the same applies to resize events. In fact, any event that you're going to make use of in this way can cause performance issues. Read Tony Gentilcore's Fastersite blog post for a breakdown of properties that cause a reflow in WebKit[159].

What we now need to do is decouple the scroll event from the `update` function, and this is exactly where `requestAnimationFrame` steps in to help. We need to change things around so that we are listening to our scroll events, but we will only store the most recent value:

[159] http://gent.ilcore.com/2011/03/how-not-to-trigger-layout-in-webkit.html

```
var latestKnownScrollY = 0;

function onScroll() {
        latestKnownScrollY = window.scrollY;
}
```

Now we're in a better place: `onScroll` runs whenever the browser chooses to execute it, but all we're doing is storing the window's scroll position. This code could run once, twenty or a hundred times *before* we try to use the value in our animation and it wouldn't matter. The point is that we're keeping a track on the value but it's not using it to trigger potentially unnecessary draw calls. If your draw call is expensive then you'll really benefit from avoiding those extra calls.

The other part of this change is to use `requestAnimationFrame` to handle the visual updates at the most convenient time for the browser:

```
function update() {
        requestAnimationFrame(update);

        var currentScrollY = latestKnownScrollY;

        // read offset of DOM elements
        // and compare to the currentScrollY value
        // then apply some CSS classes
        // to the visible items
}

// kick off
requestAnimationFrame(update);
```

Now we're just pulling the latest value from `lastKnownScrollY` when we need it and discarding everything else. If you need to capture all the event values since the last draw you could use an array and push all the values captured in `onScroll` onto it. When the time comes to do the drawing you could average the values or do whatever's most appropriate. In this case we're keeping it simple and only tracking the last value we captured.

What else can we do? Well for one thing we are constantly running `requestAnimationFrame` and that's not necessary if we haven't just scrolled since nothing will have changed. To fix that we have the onScroll initiate the `requestAnimationFrame`:

```
var latestKnownScrollY = 0,
        ticking = false;

function onScroll() {
        latestKnownScrollY = window.scrollY;
        requestTick();
}

function requestTick() {
        if(!ticking) {
                requestAnimationFrame(update);
        }
        ticking = true;
}
```

Now whenever we scroll we will try and call `requestAnimationFrame`, but if one is already requested we *don't initiate another*. This is an important optimization, since the browser will stack all the repeated rAF requests and we would be back to a situation with more calls to `update` than we need.

Thanks to this setup we no longer need to call `requestAnimationFrame` at the top of update because we know it will only be requested when one or more scroll events has taken place. We also no longer need the kick off call at the bottom, either, so let's update accordingly:

```
function update() {
        // reset the tick so we can
        // capture the next onScroll
        ticking = false;

        var currentScrollY = latestKnownScrollY;

        // read offset of DOM elements
        // and compare to the currentScrollY value
        // then apply some CSS classes
        // to the visible items
}

// kick off - no longer needed! Woo.
// update();
```

Hopefully you can see the benefits of debouncing the animations in your app from any scroll or resize events that influence it. If you're still in any doubt, John Resig wrote a great article about how Twitter was affected by scroll events[160] a while ago. Had rAF been around back then, the above technique would have probably been his recommendation.

[160] http://ejohn.org/blog/learning-from-twitter/

Debouncing Mouse Events

We've gone through one common use-case for using rAF to decouple animations from scroll and resize events, now let's talk about another one: using it to deal with interactions. In this instance we're going to have something stick to the current mouse position, but only when the mouse button is pressed. When it's released we'll stop the animation.

Let's jump straight into the code, then we'll pick it apart:

```
var mouseIsDown = false,
        lastMousePosition = { x: 0, y: 0 };

function onMouseDown() {
        mouseIsDown = true;
        requestAnimationFrame(update);
}

function onMouseUp() {
        mouseIsDown = false;
}

function onMouseMove(evt) {
        lastMousePosition.x = evt.clientX;
        lastMousePosition.y = evt.clientY;
}

function update() {
        if(mouseIsDown) {
                requestAnimationFrame(update);
        }

        // now draw object at lastMousePosition
}

document.addEventListener('mousedown', onMouseDown, false);
document.addEventListener('mouseup',   onMouseUp,   false);
document.addEventListener('mousemove', onMouseMove, false);
```

In this instance we are setting a boolean (`mouseIsDown`) depending on whether or not the mouse button is currently pressed. We can also piggy back on the `mousedown` event to initiate the first `requestAnimationFrame` call, which is handy. As we move the mouse we do a similar trick to the previous example where we simply store the last known position of the mouse, which we later use

in the `update` function. The last thing to notice is that `update` requests the next animation frame until we've called `onMouseUp` and `mouseIsDown` is set back to `false`.

Again our tactic here is to let the mouse events all proceed as often as the browser deems necessary, and we have the draw calls happen *independently* of those events. Not dissimilar to what we do with scroll events.

If things are a little more complex and you're animating something that carries on moving after `onMouseUp` has been called, you'll need to manage the calls to `requestAnimationFrame` differently. A suitable solution is to track the position of the animating object and when the change on two subsequent frames drops below a certain threshold you stop calling `requestAnimationFrame`. The changes to our code would look a little like this:

```
var mouseIsDown = false,
        lastMousePosition = { x: 0, y: 0 },
        rAFIndex = 0;

function onMouseDown() {
        mouseIsDown = true;

        // cancel the existing rAF
        cancelAnimationFrame(rAFIndex);

        rAFIndex = requestAnimationFrame(update);
}

// other event handlers as above

function update() {

        var objectHasMovedEnough = calculateObjectMovement();

        if(objectHasMovedEnough) {
                rAFIndex = requestAnimationFrame(update);
        }

        // now draw object at lastMousePosition
}

function calculateObjectMovement() {

        var hasMovedFarEnough = true;

        // here we would perhaps use velocities
```

```
32          // and so on to capture the object
33          // movement and set hasMovedFarEnough
34          return hasMovedFarEnough;
35  }
```

The main change in the above comes from the fact that if you release the mouse the rAF calls would continue until the object has come to a rest *but* you may start clicking and dragging again meaning you would get a second rAF call scheduled *as well as the original*. Not good. To combat this we make sure to cancel any scheduled `requestAnimationFrame` call (in `onMouseDown`) before we set about issuing a new one.

requestAnimationFrame and High Resolution Timestamps

While we're spending some time talking about `requestAnimationFrame` it's worth noting a recent change to how callbacks are handled in Canary. Going forward the parameter passed to your callback function will be a high resolution timestamp, accurate to a fraction of a millisecond. Two things about this:

1. It's awesome for your animations if they're time-based because now they can be really accurate
2. You'll need to update any code you have in place today that expects an object or element to be the first parameter

Get the full rundown of this at: requestAnimationFrame API: now with sub-millisecond precision[161]

An Example

OK, let's finish this article off with an example, just so you can see it all in action. It's slightly contrived, and we'll also throw in a bonus performance killer that we can fix as we go. Way too much fun!

We have a document with 800 DOM elements that we're going to move when you scroll the mouse. Because we're well-versed in modern web development we're going to use CSS transitions and `requestAnimationFrame` from the off. As we scroll down the page we'll determine which of our 800 DOM elements are now above the middle of the visible area of the screen and we'll move them over to the left hand side by adding a `left` class.

It's worth bearing in mind that we've chosen such a large number of elements because it will allow us to really see any performance issues more easily. And there are some.

Here's what our JavaScript looks like:

[161] http://updates.html5rocks.com/2012/05/requestAnimationFrame-API-now-with-sub-millisecond-precision

```
var movers = document.querySelectorAll('.mover');

/**
 * Set everthing up and position all the DOM elements
 * - normally done with the CSS but, hey, there's heaps
 * of them so we're doing it here!
 */
(function init() {

    for(var m = 0; m < movers.length; m++) {
        movers[m].style.top = (m * 10) + 'px';
    }

})();

/**
 * Our animation loop - called by rAF
 */
function update() {

        // grab the latest scroll position
    var scrollY             = window.scrollY,
        mover               = null,
        moverTop            = [],
        halfWindowHeight    = window.innerHeight * 0.5,
        offset              = 0;

        // now loop through each mover div
        // and change its class as we go
    for(var m = 0; m < movers.length; m++) {

        mover       = movers[m];
        moverTop[m] = mover.offsetTop;

        if(scrollY > moverTop[m] - halfWindowHeight) {
            mover.className = 'mover left';
        } else {
            mover.className = 'mover';
        }

    }

```

```
        // keep going
    requestAnimationFrame(update);
}

// schedule up the start
window.addEventListener('load', update, false);
```

Our demo page before performance and rAF optimizations

If you check out the pre-optimized page[162] you'll see it really struggle to keep up as you scroll, and there are a number of reasons why. Firstly we are brute force calling the `requestAnimationFrame`, whereas what we really should do is only calculate any changes when we get a scroll event. Secondly we are calling `offsetTop` which causes a reflow, but then we immediately apply the `className` change and that's going to cause a repaint. And then thirdly, for our bonus performance killer, we are using `className` rather than `classList`.

The reason using `className` is less performant than `classList` is that `className` will *always* affect the DOM element, even if the value of `className` hasn't changed. By just setting the value we trigger a repaint, which can be very expensive. Using `classList`, however, allows the browser to be much more intelligent about updates, and it will leave the element alone should the list already contain the class you're adding (which is `left` in our case).

If you want more information on using `classList` and the new-and-extremely-useful frame breakdown mode in Chrome's Dev Tools you should watch this video by Paul Irish:

So let's take a look at what a better version of this would look like:

```
var movers          = document.querySelectorAll('.mover'),
    lastScrollY     = 0,
    ticking         = false;

/**
 * Set everthing up and position all the DOM elements
 * - normally done with the CSS but, hey, there's heaps
 * of them so we're doing it here!
 */
(function init() {

    for(var m = 0; m < movers.length; m++) {
        movers[m].style.top = (m * 10) + 'px';
    }

})();
```

[162] http://www.html5rocks.com/en/tutorials/speed/animations/pre.html

```
17
18  /**
19   * Callback for our scroll event - just
20   * keeps track of the last scroll value
21   */
22  function onScroll() {
23      lastScrollY = window.scrollY;
24      requestTick();
25  }
26
27  /**
28   * Calls rAF if it's not already
29   * been done already
30   */
31  function requestTick() {
32      if(!ticking) {
33          requestAnimationFrame(update);
34          ticking = true;
35      }
36  }
37
38  /**
39   * Our animation callback
40   */
41  function update() {
42      var mover               = null,
43          moverTop            = [],
44          halfWindowHeight    = window.innerHeight * 0.5,
45          offset              = 0;
46
47          // first loop is going to do all
48          // the reflows (since we use offsetTop)
49      for(var m = 0; m < movers.length; m++) {
50
51          mover       = movers[m];
52          moverTop[m] = mover.offsetTop;
53      }
54
55          // second loop is going to go through
56          // the movers and add the left class
57          // to the elements' classlist
58      for(var m = 0; m < movers.length; m++) {
```

```
59
60          mover       = movers[m];
61
62          if(lastScrollY > moverTop[m] - halfWindowHeight) {
63              mover.classList.add('left');
64          } else {
65              mover.classList.remove('left');
66          }
67
68      }
69
70          // allow further rAFs to be called
71      ticking = false;
72  }
73
74  // only listen for scroll events
75  window.addEventListener('scroll', onScroll, false);
```

Our demo page after performance and rAF optimizations

If you look at our new optimized version of the demo[163] you will see much smoother animations as you scroll up and down the page. We've stopped calling `requestAnimationFrame` indiscriminantly, we now only do it when we scroll (and we ensure there is only one call scheduled). We've also moved the `offsetTop` property lookups into one loop and put the class changes into a second loop which means that we're avoiding the reflow-repaint problem. We've decoupled our events from the draw call so they can happen as often as they like and we won't be doing unnecessary drawing. Finally we've switched out `className` for `classList`, which is a massive performance saver.

Of course there are other things we can do to take this further, in particular not iterating through *all 800* DOM elements on each pass, but even just the changes we've made have given us great performance improvements.

Conclusion

It's important to not only use `requestAnimationFrame` for your animations, but also to use it in the right way. As you can hopefully see it's quite easy to inadvertently cause bottlenecks, but by understanding how the browser actually executes your code you can fix any problems quickly. Take some time with Chrome's Dev Tools, especially the frame mode, and see where your animations can be improved.

[163] http://www.html5rocks.com/en/tutorials/speed/animations/post.html

Node.js

Understanding event loops and writing great code for Node.js

Original Article

http://developer.yahoo.com/blogs/ydn/part-1-understanding-event-loops-writing-great-code-11401.html

Tom Hughes-Croucher, developer.yahoo.com/blogs[a]

[a] http://developer.yahoo.com/blogs

As usual JSConf.eu[164] was one of the best conferences I've been to this year. It's always a delight to hang out with some of the smartest JavaScript coders on the planet, and doing it in the very trendy city of Berlin was a nice bonus. There are a few great recaps of JSConf[165] so I'm not going to focus on those; instead I'm going to expand the talk I gave.

Programming inside an event loop is something most programmers only have a passing familiarity with. Node.js is quickly gaining traction as a very popular event loop-based server for JavaScript. It gives people the ability to write JavaScript on the server, which has access to things like the HTTP stack, TCP, file I/O, and databases. However the kind of programs that we write in this context tend to be very different from the kind of programs we write for browser-side applications or sites.

In the browser, the bulk of the program tends to be involved with setting up the user-interface, and then there are some small event callbacks that are enacted — most typically, based on user interaction with the browser. For example, when the user clicks Submit on a form, we intercept the click and validate their submission.

There is an important point here: in most browser programming, programmers are only dealing with events caused by the user, and the user can only do so many things at once. Moreover the callbacks for those events tend to be very discrete and don't cause other events. Some libraries, such as YUI, have custom event support. But it is often thought of an advanced feature.

On the server, however, there isn't a user to drive a variety of interactions. Instead we have a whole range of reactions to take on many different kinds of events. Node.js takes the approach that all I/O

[164] http://jsconf.eu/2010/

[165] http://palagpat-coding.blogspot.com/2010/09/in-case-you-missed-it-jsconfeu-2010-day.html

activities should be non-blocking (for reasons we'll explain more later). This means that all HTTP requests, database queries, file I/O, and so on, do not halt execution until they return. Instead they run independently and then emit an event when the data is available.

This means that programming in Node.js has lots of callbacks dealing with all kinds of I/O and then initiating other callbacks for other kinds of I/O. This is a very different from browser programming. There is still a certain amount of liner setup, but the bulk of the code involves dealing with callbacks.

Event drivenBecause of these different programming styles, we need to look for patterns to help us effectively program on the server. That starts with the event loop, so let's take a look at that in more depth.

I think that most people intuitively get event-driven programming because it's like everyday life. Imagine you are cooking. You are chopping a bell pepper and a pot starts to boil over. You finish the slice you are doing, and then turn down the stove. In daily life, we are used to having all sorts of internal callbacks for dealing with events, and yet, like JavaScript, we only do one thing at once.

Yes, yes, I can see you are rubbing your tummy and patting your head at the same time, well done. But, if you try to do any serious activities at the same time, it goes wrong pretty quick. This is like JavaScript. It's great at letting events drive the action, but it is "single-threaded" so only one thing happens at once.

Another way to think about this event loop is a post (or mail) man. To our event-loop postman, each letter is an event. He has a stack of events to deliver in order. For each letter (event) the postman gets, he walks to the route to deliver the letter. The route is the callback function assigned to that event (sometimes more than one). However, critically, since our mailman only has a single set of legs, he can only walk a single code path at once.

Sometimes, while the postman is walking a code route someone will give him another letter. This is because the callback function he is code walking has emitted an event. In this case the postman delivers the new message immediately (after all someone gave to him directly instead of going via the post office so it must be urgent). The postman will diverge from his current code path and walk to the code path to deliver the new event. He then carries on walking the original event that emitted the event he just walked.

Let's look at another example. Let's give the postman a letter to deliver that requires a gate to be open. He gets there and the gate is closed, so he simply waits and tries again, and again. He's trapped in an endless loop waiting for the gate to open. But if there is a letter on the stack that will ask someone to open the gate so the postman can get through, surely that will solve things, right?

Unfortunately it won't, because the postman will never get to deliver the letter because he's stuck waiting endlessly for the gate to open. This is because the event that opens the gate is external from the current event callback. If we emit the event from within a callback, we already know our postman will go and deliver that letter before carrying on. However, when events are emitted external to the currently executing piece of code, they will not be called until that piece of code has been fully evaluated to its conclusion.

We can use this to write a piece of code that creates a loop that Node.js (or a browser) will never break out of:

```
EE = require('events').EventEmitter;
ee = new EE();

die = false;

ee.on('die', function() {
  die = true;
});

setTimeout(function() {
  ee.emit('die');
}, 100);

while(!die) {
}

console.log('done');
```

In this example, console.log will never be called because the while loop stops Node from ever getting chance to callback the timeout and emit the `'die'` event.

This is a really important piece of information, because while it's unlikely we'd program a loop like this that relies on an external condition to exit, it clearly illustrates how Node.js can only do one thing at once. And getting a fly in the ointment can really screw up the whole server.

Let's look at the standard Node.js code for creating an HTTP server:

```
var http = require('http');
  http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/'
```

This code is the 101 example from the Node.js web site[166]. It creates an HTTP server using a factory method in the `http` library. The factory method creates a new HTTP server and attaches a callback to the `'request'` event. The callback is specified as the argument to the `createServer`.

What's interesting here is what happens when this code is run. The first thing Node.js does is run the preceding code from top to bottom. This can be considered the 'setup' phase of Node programming.

[166] http://nodejs.org/

Since we attached some event listeners, Node.js doesn't exit, but waits for an event to be fired. If we didn't attach any events, then Node.js would exit as soon as it had run the code.

So what happens when we get an HTTP request? When an HTTP request is sent to the server, Node.js emits the 'request' event that causes the callbacks attached to that event to be run in order. In this case there is only one callback, the anonymous function we passed as an argument to `createServer`. Let's assume it's the first request the server has had since it setup. Since there is no other code running, the `'request'` event is emitted and the callback is just run. It's a very simple callback and it runs pretty fast.

Let's assume that our site gets really popular and we get lots of requests. If, for the sake of argument, our callback takes 1 second, then if we received 2 requests at the same time, they can't both be run at once, and the second request isn't going to be acted on for another second or so. Obviously, a second is a really long time, but as we start to look at real-world applications, the problem of blocking the event loop becomes more dangerous as we can see the damage it could have on users.

The upshot of this is that we want to keep Node.js as event-driven and non-blocking as possible. In the same way that an I/O event that can be slow should use callbacks to indicate the presence of data Node.js can act on, the Node.js program itself shouldn't be written in such a way that any single callback ties up the event loop for extended pieces of time.

The operating system kernel actually handles the TCP connections to clients for the HTTP server, so there isn't a risk of not accepting new connections. But there is a real danger of not acting on them.

This means that we should employ two strategies for writing Node.js servers:

- Once set up has been completed, make all actions event driven
- If Node.js is required to process something that will take a long time, consider delegating with web workers[167]

Taking the event-driven approach works effectively with the event loop — I guess the name is hint it would — but it's also important to write event-driven code in a way that is easy to read and understand.

In the previous example, we used an anonymous function as the event callback. This makes things hard in a couple of ways. Firstly, we have no control over where the code lives; the anonymous function must live where it is attached to the event either via a factory method or the `on` method of an EventEmitter. The second issue is debugging: if everything is an anonymous event, it can sometimes be hard to distinguish similiar callbacks from each other when an exception occurs.

Callback Hell

A guide to writing asynchronous javascript programs

[167] http://developer.yahoo.com/blogs/ydn/posts/2010/07/multicore_http_server_with_nodejs/

Original Article

http://callbackhell.com

Max Ogden, github.com/maxogden[a]

[a] https://github.com/maxogden

What is "*callback hell*"?

Asynchronous javascript, or javascript that uses callbacks, is hard to get right intuitively. A lot of code ends up looking like this:

```
fs.readdir(source, function(err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function(filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function(err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function(width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(destination + 'w' + width + '_' + fi\
lename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

See all the instances of `function` and `})`? Eek! This is affectionately known as **callback hell**.

Writing better code isn't that hard! You only need to know about a few things:

Name your functions

Here is some (messy) browser javascript that uses browser-request[168] to make an AJAX request to a server:

```
var form = document.querySelector('form')
form.onsubmit = function(submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function(err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

This code has two anonymous functions. Let's give em names!

```
var form = document.querySelector('form')
form.onsubmit = function formSubmit(submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function postResponse(err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

As you can see naming functions is super easy and does some nice things to your code:

- makes code easier to read
- when exceptions happen you will get stacktraces that reference actual function names instead of "anonymous"
- allows you to keep your code shallow, or not nested deeply, which brings me to my next point:

[168] https://github.com/iriscouch/browser-request

Keep your code shallow

Building on the last example, let's go a bit further and get rid of the triple level nesting that is going on in the code:

```
function formSubmit(submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse(err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}

document.querySelector('form').onsubmit = formSubmit
```

Code like this is less scary to look at and is easier to edit, refactor and hack on later.

Modularize!

This is the most important part: **Anyone is capable of creating modules** (AKA libraries). To quote Isaac Schlueter[169] (of the node.js project): *"Write small modules that each do one thing, and assemble them into other modules that do a bigger thing. You can't get into callback hell if you don't go there."*

Let's take out the boilerplate code from above and turn it into a module by splitting it up into a couple of files. Since I write JavaScript in both the browser and on the server, I'll show a method that works in both but is still nice and simple.

Here is a new file called `formuploader.js` that contains our two functions from before:

[169] http://twitter.com/izs

```
function formSubmit(submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse(err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}

exports.submit = formSubmit
```

The `exports` bit at the end is an example of the CommonJS module system, which is used by Node.js for server side javascript programming. I quite like this style of modules because it is so simple – you only have to define what should be shared when the module gets required (that's what the `exports` thing is).

To use CommonJS modules in the browser you can use a command-line thing called browserify[170]. I won't go into the details on how to use it here but it lets you use `require` to load modules into your programs.

Now that we have `formuploader.js` (and it is loaded in the page as a script tag) we just need to require it and use it! Here is how our application specific code looks now:

```
var formUploader = require('formuploader')
document.querySelector('form').onsubmit = formUploader.submit
```

Now our application is only two lines of code and has the following benefits:

- easier for new developers to understand – they won't get bogged down by having to read through all of the `formuploader` functions
- `formuploader` can get used in other places without duplicating code and can easily be shared on github
- the code itself is nice and simple and easy to read

There are lots of module patterns for web browser[171] and on the server[172]. Some of them get very complicated. The ones shown here are what I consider to be the simplest to understand.

[170] https://github.com/substack/node-browserify

[171] http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth

[172] http://nodejs.org/api/modules.html

I still don't get it

Try reading my introduction to callbacks[173].

What about promises?

Promises[174] are a more abstract pattern of working with async code in JavaScript.

The scope of this document is to show how to write vanilla javascript. If you use a third party library that adds abstraction to your JS then make sure you're willing to force everyone that contributes to your library to also have the same views on JS as you.

In my own personal experience I use callbacks for 90% of the async code I write and when things get hairy I bring in something like the async[175] library.

That being said, everyone develops their own unique JavaScript style and you should do what you like. Just remember that there are no absolutes: some people like to use only callbacks, some people don't.

Additional reading

- A pattern for decoupling DOM events from @dkastner[176]

Please contribute new sections or fix existing ones by forking this project on github[177]!

Understanding Express.js

Original Article

http://evanhahn.com/understanding-express-js/

Evan Hahn, evanhahn.com[a]

[a] http://evanhahn.com/

[173] https://github.com/maxogden/art-of-node#callbacks
[174] http://domenic.me/2012/10/14/youre-missing-the-point-of-promises/
[175] https://github.com/caolan/async
[176] https://gist.github.com/3392235
[177] http://github.com/maxogden/callback-hell

This is aimed at people who have some familiarity with Node.js. They know how to run Node scripts and can install packages with npm. You don't have to be an expert, though – I promise. This guide was last updated for Express 3.2.5. It's an introduction and mostly deals with concepts.

Express.js[178] describes itself better than I can: "a minimal and flexible node.js web application framework". It helps you build web apps. If you've used Sinatra[179] in the Ruby world, a lot of this will be familiar.

Like any abstraction, Express hides difficult bits and says "don't worry, you don't need to understand this part". It does things for you so that you don't have to bother. In other words, it's magic.

It's good magic, too. Express catalogs some people using it[180], and there are some big names: MySpace, Klout, and even some stuff *I've* made. *Me*. I'm a *huge* deal. I've got a *blog*.

But all magic comes at a price[181]: you might not understand the inner workings of Express. This is like driving a car; I drive a car just fine without intimate knowledge of its workings, but I'd be better off with that knowledge. What if things break? What if you want to get all the performance you can out of the car? What if you have an *insatiable thirst for knowledge*?

So let's understand Express from the bottom, with Node.

Bottom layer: Node's HTTP server

Node has an HTTP module[182] which makes a pretty simple abstraction for making a webserver. Here's what that might look like:

```
1  // Require what we need
2  var http = require("http");
3
4  // Build the server
5  var app = http.createServer(function(request, response) {
6    response.writeHead(200, {
7      "Content-Type": "text/plain"
8    });
9    response.end("Hello world!\n");
10 });
11
12 // Start that server, baby
13 app.listen(1337, "localhost");
14 console.log("Server running at http://localhost:1337/");
```

[178] http://expressjs.com/
[179] http://www.sinatrarb.com/
[180] http://expressjs.com/applications.html
[181] http://shapeshed.com/all-magic-comes-with-a-price/
[182] http://nodejs.org/api/http.html

And if you run that app (if that file is called `app.js`, you'd run `node app.js`), you'll get a response of "Hello world!" if you visit `localhost:1337` in your browser. You'll get the same response no matter what, too. You can try visiting `localhost:1337/anime_currency` or `localhost:1337/?onlyfriend=anime`, and it's like talking to a brick wall: "Hello world!"

Let's break this down a bit.

The first line uses the `require` function to load a built-in Node module called `http`. It puts this lovely module inside of a variable called `http`. For more about the require function, check out Nodejitsu's docs[183].

Then we put a server in a variable called `app` by using `http.createServer`. This takes a function that listens for requests. We'll get back to this in a minute because they're Super Duper Important. Skip over it for the next two sentences.

The last thing we do is tell the server to listen for requests coming in on port 1337, and then we just log that out. And then we're in business.

Okay, back to the request handler function. That thing is *important*.

The request handler

Before I start this section, I should say that there's a bunch of cool HTTP stuff in here that I don't think is relevant to learning Express. If you're interested, you can look at the docs for the HTTP module[184] because they have a bunch of stuff.

Whenever we make a request to the server, that request handler function is called. If you don't believe me, try putting a `console.log` in there. You'll see that it logs out every time you load a page.

`request` is a request that comes from the client. In many apps, you'll see this shortened to `req`. Let's look at it. To do that, we'll modify the above request handler a bit:

```
var app = http.createServer(function(request, response) {

  // Build the answer
  var answer = "";
  answer += "Request URL: " + request.url + "\n";
  answer += "Request type: " + request.method + "\n";
  answer += "Request headers: " + JSON.stringify(request.headers) + "\n";

  // Send answer
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end(answer);

});
```

[183] http://docs.nodejitsu.com/articles/getting-started/what-is-require

[184] http://nodejs.org/api/http.html

Restart the server and reload `localhost:1337`. You'll see what URL you're requesting, that it's a GET request, and that you've sent a number of cool headers like the user-agent and more complicated HTTP stuff! If you visit `localhost:1337/what_is_anime`, you'll see the request URL change. If you visit it with a different browser, the user-agent will change. If you send it a POST request, you'll see the method change.

The `response` is the next part. Just like the prior argument is often shortened, this is often shortened to the three-letter `res`. With each response, you get the response all ready to send, and then you call `response.end`. Eventually, you *must* call this method; even the Node docs say so[185]. This method does the actual sending of data. You can try making a server where you don't call it, and it just hangs forever.

Before you send it out, you'll want to write some headers. In our example, we do this:

```
response.writeHead(200, { "Content-Type": "text/plain" });
```

This does two things. First, it sends HTTP status code[186] 200, which means "OK, everything is good". Then, it sets some response headers. In this case, it's saying that we're sending back the plaintext content-type. We could send other things like JSON or HTML.

I thirst for more

You want more? Okay. You asked nicely.

One could imagine taking these APIs and turning them into something cool. You could do something (sorta) like this:

```
var http = require("http");

http.createServer(function(req, res) {

  // Homepage
  if (req.url == "/") {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end("Welcome to the homepage!");
  }

  // About page
  else if (req.url == "/about") {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end("Welcome to the about page!");
```

[185] http://nodejs.org/api/http.html#http_response_end_data_encoding
[186] https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

```
  }

  // 404'd!
  else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("404 error! File not found.");
  }

}).listen(1337, "localhost");
```

You could clean this up and make it pretty, or you could be hardcore like the npm.org folks[187] and tough it out with vanilla Node. But you could also build a framework. That's what Sencha did. And they called it Connect.

Middle layer: Connect

Fitting that Connect[188] happens to be the middle layer of this JavaScript cake, because it calls itself "a middleware framework for node". Don't go searching "what is middleware" just yet – I'm about to explain it.

A little bit of Connect code

Let's say we wanted to write the "hello world" app that we had above, but with Connect this time. Don't forget to install Connect (`npm install`, baby). Once you've done that, the app is pretty similar.

```
// Require the stuff we need
var connect = require("connect");
var http = require("http");

// Build the app
var app = connect();

// Add some middleware
app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello world!\n");
});

// Start it up!
http.createServer(app).listen(1337);
```

[187] https://github.com/isaacs/npm-www#design-philosophy
[188] http://www.senchalabs.org/connect/

So let's step through this.

First, we require Connect. We then require Node's HTTP module just like we did before. We're ready.

Then we make a variable called `app` like we did before, but instead of creating the server, we call `connect()`. What's going on? What is this madness?

We then add some middleware – it's just a function. We pass this to `app.use`, and this function looks *an awful lot like* the request handlers from above. In fact, *I copy-pasted it.*

Then we create the server and start listening. `http.createServer` took a function before, so guess what – `app` is just a function. It's a Connect-made function that starts going through all the middleware until the end. But it's just a request handler like before.

(Worth noting that you might see people using `app.listen(1337)`, which just defers to `http.createServer`. This is true in both Connect and Express.)

Okay, now I'm going to explain middleware.

What is middleware?

I want to start by saying that Stephen Sugden's description of Connect middleware[189] is really good and does a better job than I can. If you don't like my explanation, read his.

Remember the request handlers from a few sections earlier? Each piece of middleware is a request handler. You start by looking at the first request handler, then you look at the next one, then the next, and so on.

Here's what middleware basically looks like:

```
function myFunMiddleware(request, response, next) {
   // Do stuff with the request and response.
   // When we're all done, call next() to defer to the next middleware.
   next();
}
```

When we start a server, we start at the topmost middleware and work our way to the bottom. So if we wanted to add simple logging to our app, we could do it!

[189] http://stephensugden.com/middleware_guide/

```
var connect = require("connect");
var http = require("http");
var app = connect();

// Logging middleware
app.use(function(request, response, next) {
  console.log("In comes a " + request.method + " to " + request.url);
  next();
});

// Send "hello world"
app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello world!\n");
});

http.createServer(app).listen(1337);
```

If you run this app and visit `localhost:1337`, you'll see that your server is logging some stuff and you'll see your page.

It's important to note that anything that works in the vanilla Node.js server also works in middleware. For example, if you want to inspect `req.method`, it's right there.

While you can totally write your own, Connect comes with a bunch of cool middleware and there's a bunch of third-party middleware[190] too. Let's remove our logger and use the one built into Connect[191]:

```
var connect = require("connect");
var http = require("http");
var app = connect();

app.use(connect.logger());
// Fun fact: connect.logger() returns a function.

app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello world!\n");
});

http.createServer(app).listen(1337);
```

[190] https://github.com/senchalabs/connect/wiki

[191] http://www.senchalabs.org/connect/logger.html

Visit `localhost:1337` and you'll see some logging!

I thirst for more

One could imagine stringing together some middleware to build an app. Maybe you'd do it like this:

```
var connect = require("connect");
var http = require("http");
var app = connect();

app.use(connect.logger());

// Homepage
app.use(function(request, response, next) {
  if (request.url == "/") {
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("Welcome to the homepage!\n");
    // The middleware stops here.
  } else {
    next();
  }
});

// About page
app.use(function(request, response, next) {
  if (request.url == "/about") {
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("Welcome to the about page!\n");
    // The middleware stops here.
  } else {
    next();
  }
});

// 404'd!
app.use(function(request, response) {
  response.writeHead(404, { "Content-Type": "text/plain" });
  response.end("404 error!\n");
});

http.createServer(app).listen(1337);
```

"This is ugly! I need to build a framework," you say. You *savage*. You're never satisfied, are you? *Will there ever be enough?*

Some people saw Connect and they said, "this code can be even easier". And so they built Expresss. (Actually, it seems like they saw Sinatra[192] and stole it.)

Top layer: Express

We've finally arrived at the third act of our nerdy quest. We're at the peak of our abstraction mountain. There is a beautiful sunset. Your long, golden locks wave in the cool breeze.

Just like Connect extends Node, Express extends Connect. The beginning looks very similar to Connect:

```
1 var express = require("express");
2 var http = require("http");
3 var app = express();
```

And so does the end:

```
1 http.createServer(app).listen(1337);
```

But the middle is what's different. Where Connect gives you one cool feature (middleware), I think that Express gives you three cool features: routing, better request handlers, and views. Let's start with routing.

Cool feature 1: routing

Routing is a way to map different requests to specific handlers. In many of the above examples, we had a homepage and an about page and a 404 page. We'd basically do this with a bunch of `if` statements in the examples.

But Express is smarter than that. Express gives us something called "routing" which I think is better explained with code than with English:

[192] http://www.sinatrarb.com/

```
var express = require("express");
var http = require("http");
var app = express();

app.all("*", function(request, response, next) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  next();
});

app.get("/", function(request, response) {
  response.end("Welcome to the homepage!");
});

app.get("/about", function(request, response) {
  response.end("Welcome to the about page!");
});

app.get("*", function(request, response) {
  response.end("404!");
});

http.createServer(app).listen(1337);
```

Ooh. That's hot.

After the basic requires, we say "every request goes through this function" with `app.all`. And that function looks an awful lot like middleware, don't it?

The three calls to `app.get` are Express's routing system. They could also be `app.post`, which respond to POST requests, or PUT, or any of the HTTP verbs. The first argument is a path, like `/about` or `/`. The second argument is a request handler similar to what we've seen before. To quote the Express documentation[193]:

> [These request handlers] behave just like middleware, with the one exception that these callbacks may invoke `next('route')` to bypass the remaining route callback(s). This mechanism can be used to perform pre-conditions on a route then pass control to subsequent routes when there is no reason to proceed with the route matched.

In short: they're basically middleware like we've seen before. They're just functions, just like before.

These routes can get smarter, with things like this:

[193] http://expressjs.com/api.html#app.VERB

```
app.get("/hello/:who", function(req, res) {
  res.end("Hello, " + req.params.who + ".");
});
```

Restart your server and visit `localhost:1337/hello/animelover69` for the following message:

> Hello, animelover69.

The docs[194] also show an example that uses regular expressions, and you can do lots of other stuff with this routing. For a conceptual understanding, I've said enough.

But it gets cooler.

Cool feature 2: request handling

Routing would be enough, but Express is absolutely ruthless.

Express augments the request and response objects that you're passed in every request handler. The old stuff is still there, but they add some new stuff too! The API docs[195] explain everything, but let's look at a couple of examples.

One nicety they give you is a `redirect` method. Here are some examples:

```
response.redirect("/hello/anime");
response.redirect("http://www.myanimelist.net");
response.redirect(301, "http://www.anime.org");  // HTTP status code 301
```

This isn't in vanilla Node and it's also absent from Connect, but Express adds this stuff. It adds things like `sendFile` which lets you just send a whole file:

```
response.sendFile("/path/to/anime.mp4");
```

The request gets a number of cool properties, like `request.ip` to get the IP address and `request.files` to get uploaded files.

Conceptually, there's not much to know, other than the fact that Express extends the request and response. For everything Express gives you, check out the API docs[196].

Cool feature 3: views

More features? *Oh, Express, I'm blushing.*

Express can handle views. It's not too bad. Here's what the setup looks like:

[194] http://expressjs.com/api.html#app.VERB
[195] http://expressjs.com/api.html
[196] http://expressjs.com/api.html

```
// Start Express
var express = require("express");
var app = express();

// Set the view directory to /views
app.set("views", __dirname + "/views");

// Let's use the Jade templating language
app.set("view engine", "jade");
```

The first block is the same as always. Then we say "our views are in a folder called 'views'". Then we say "use Jade". Jade[197] is a templating language. We'll see how it works in just a second!

Now, we've set up these views. How do we use them?

Let's start by making a file called `index.jade` and put it into a directory called `views`. It might look like this:

```
doctype 5
html
  body
    h1 Hello, world!
    p= message
```

This is basically HTML without all the brackets. It should be *fairly* straightforward if you know HTML. The only interesting part is the last line. `message` is a variable! Woah! Where did that come from? *I'll tell you.*

We need to render the view from within Express. Here's what that looks like:

```
app.get("/", function(request, response) {
  response.render("index", { message: "I love anime" });
});
```

Express adds a method to `response`, called `render`. It does a bunch of smart stuff, but it basically looks at the view engine and views directory (the stuff we defined earlier) and renders `index.jade`.

The last step (I suppose it could be the first step) is to install Jade, because it's not bundled with Express. Add it to your `package.json` or `npm install` it.

If you get all of this set up, you'll see this page[198]. Here's all the source code.[199]

[197] http://jade-lang.com/

[198] http://evanhahn.com/wp-content/uploads/2013/05/anime.html

[199] https://gist.github.com/EvanHahn/5673968

Bonus cool feature: everything from Connect and Node

I want to remind you that Express is built on top of Connect which is built on top of Node. This means that all Connect middleware works with Express. This is useful! For example:

```
var express = require("express");
var app = express();

app.use(express.logger());  // Inherited from Connect

app.get("/", function(req, res) {
  res.send("anime");
});

app.listen(1337);
```

If you learn one thing from this post, it should be this. If your brain has room for another fun fact: hippo milk is pink!

Actually building something

Most of the stuff in this post is conceptual, but let me push you in the right direction for building something you want to build. I don't want to delve into specifics.

You can install Express as an executable in your terminal. It spits out boilerplate code that's very helpful for starting your app. Install it globally with npm:

```
### You'll probably need `sudo` for this:
npm install -g express
```

If you need help, use `express --help`. It spits out some options. For example, let's say I want to use EJS templating and LESS for CSS. My app is called "myApp". Here's what I'd type to get started:

```
express --ejs --css less myApp
```

It'll generate a bunch of files and then tell you to go into that directory and `npm install`. If you do that, you'll have a basic app running with `node app`! I'd recommend looking through the generated files to see some boilerplate, and then messing with it a bunch. It's hardly a full app, but I found it very helpful to poke through these files and mess with them when getting started.

Also helpful are the many official examples on GitHub[200].

Some concluding miscellany

- If you love CoffeeScript like I do[201], you should know that all of this stuff works with

[200] https://github.com/visionmedia/express/tree/master/examples

[201] http://evanhahn.com/i-love-coffeescript/

CoffeeScript. You don't even need to compile it! Instead of starting your server with `node app.js`, start it with `coffee app.coffee`. This is what I do in my apps. *Me.* I'm a big deal. I've got a *blog.*

- I was confused when I saw `app.use(app.router)` – doesn't Express *always* use a router? The short answer is that `app.router` is Express's routing middleware, and it's implicitly included when you define a route. You can *explicitly* include it because you want the router middleware to come before other stuff, which is sometimes desirable. This StackOverflow answer explains it well.[202]
- This guide was written for Express 3, and the version 4 roadmap[203] shows the possibility of some dramatic changes. Most notably, it looks like Express might be split up into a bunch of small modules and eat some of Connect's features. It's up in the air, but the "plans" are worth a look.

I thirst for more

Is there no satisfying you? You *glutton.* You make me *sick.* Soon you're gonna be sitting in an opium den, eyes half-open, drooling out the last drop of your programming talent.

Just like Rails is the de-facto way to build web apps with Ruby and *demonic masochism* is the de-facto way to build web apps in PHP, I get the impression that Express is the de-facto way to build web apps in Node. But unlike Rails, Express is much lower-level. It seems like no high-level Node library has stuck out. I think this is going to change. Keep your eyes out.

I won't go into them here, but just as Express was built on Connect and Connect on Node, people have built things on top of Express. The Express wiki lists them[204] and many of them are Pretty Cool. You can use these frameworks if you'd prefer, or you can stick to the lower-level Express. Either way, go build cool stuff!

Previous: Concept: making asynchronous loading look synchronous[205]

[202] http://stackoverflow.com/a/12695813/804100
[203] https://github.com/visionmedia/express/wiki/4.x-roadmap
[204] https://github.com/visionmedia/express/wiki#frameworks-built-with-express
[205] http://evanhahn.com/javascript-method_missing/
[206] http://creativecommons.org/licenses/by/3.0/
[207] http://unlicense.org/
[208] http://www.luluspice.com/

About the Author

Azat Mardanov

Azat Mardanov has over 12 years of experience in web, mobile and software development. With a Bachelor's Degree in Informatics and a Master of Science in Information Systems Technology degree, Azat possesses deep academic knowledge as well as extensive practical experience.

Recently, Azat has worked as a CTO/co-founder at Gizmo[209] — an enterprise cloud platform for mobile marketing campaigns, and has undertaken the prestigious 500 Startups[210] business accelerator program. Previously, he was developing mission-critical applications for government

[209] http://www.crunchbase.com/company/gizmo

[210] http://500.co/

agencies in Washington, DC: National Institutes of Health, National Center for Biotechnology Information, Federal Deposit Insurance Corporation, and Lockheed Martin. Azat is a frequent attendee at Bay Area tech meet-ups and hackathons (AngelHack[211] hackathon '12 finalist with team FashionMetric.com[212]).

Currently, he works as an engineer at the curated social media news aggregator website, Storify.com[213]. He mentors entrepreneurs as a hacker in residence at startup accelerator and fund StartupMonthly[214], where he is teaching technical Rapid Prototyping with JavaScript and Node.js[215] training to much acclaim. In his spare time, Azat writes about technology on his blog: webAppLog.com[216].

Let's be Friends on the Internet

- Twitter: @RPJSbook[217] and @azat_co[218]
- Facebook: facebook.com/RapidPrototypingWithJS[219]
- Website: rapidprototypingwithjs.com[220]
- Blog: webapplog.com[221]
- GitHub: github.com/azat-co/rpjs[222]
- Storify: Rapid Prototyping with JS[223]

Other Ways to Reach Us

- Email: hi@rapidprototypingwithjs.com[224]
- Google Group: rpjs@googlegroups.com[225] and https://groups.google.com/forum/#!forum/rpjs

Share on Twitter

> "I've finished Rapid Prototyping with JS — book on agile development with JavaScript and Node.js by @azat_co #RPJS @RPJSbook" — http://clicktotweet.com/40dvj

[211] http://angelhack.com
[212] http://fashionmetric.com
[213] http://storify.com
[214] http://startupmonthly.org
[215] http://www.startupmonthly.org/rapid-prototyping-with-javascript-and-nodejs.html
[216] http://webapplog.com
[217] https://twitter.com/rpjsbook
[218] https://twitter.com/azat_co
[219] https://www.facebook.com/RapidPrototypingWithJS
[220] http://rapidprototypingwithjs.com/
[221] http://webapplog.com
[222] https://github.com/azat-co/rpjs
[223] https://storify.com/azat_co/rapid-prototyping-with-js
[224] mailto:hi@rapidprototypingwithjs.com
[225] mailto:rpjs@googlegroups.com