Globant

we are ready

Front End Engineer

Developer Tools and Evaluation

TOPICS

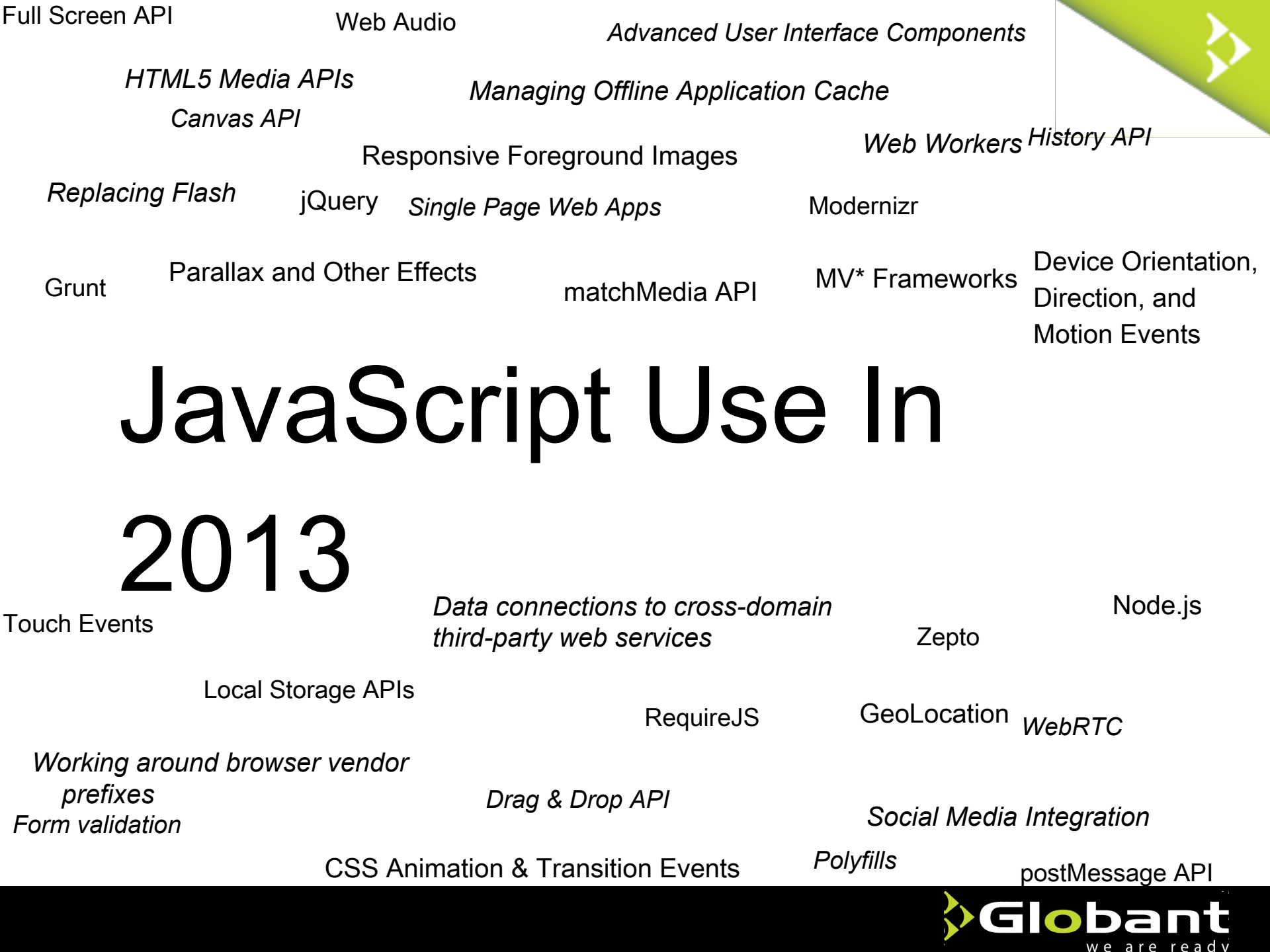
Writing valuable documentation
Debugging
Introduction to TDD in JavaScript.

Basic page manipulation

Form validation

JavaScript Use In 2003

Custom cross-browser code to work around differences in DOM


Full Screen API
Web Audio
Advanced User Interface Components
HTML5 Media APIs
Managing Offline Application Cache
Canvas API
Responsive Foreground Images
Web Workers
History API
Replacing Flash
jQuery
Single Page Web Apps
Modernizr
Grunt
Parallax and Other Effects
matchMedia API
MV* Frameworks
Device Orientation, Direction, and Motion Events
JavaScript Use In 2013
Touch Events
Data connections to cross-domain third-party web services
Zepto
Node.js
Local Storage APIs
RequireJS
GeoLocation
WebRTC
Working around browser vendor prefixes
Drag & Drop API
Social Media Integration
Form validation
CSS Animation & Transition Events
Polyfills
postMessage API

Maintainability

Why do we care?

Most of your time is spent maintaining code

Now, you can see:

01 Introduction to debugging
02 Tracing through a section of code
03 Understanding error messages

http://youtu.be/otubD6inrO4

http://youtu.be/OCT39W704Es

http://youtu.be/6JhIJQQ9zKk

JavaScript code is getting large and complex.

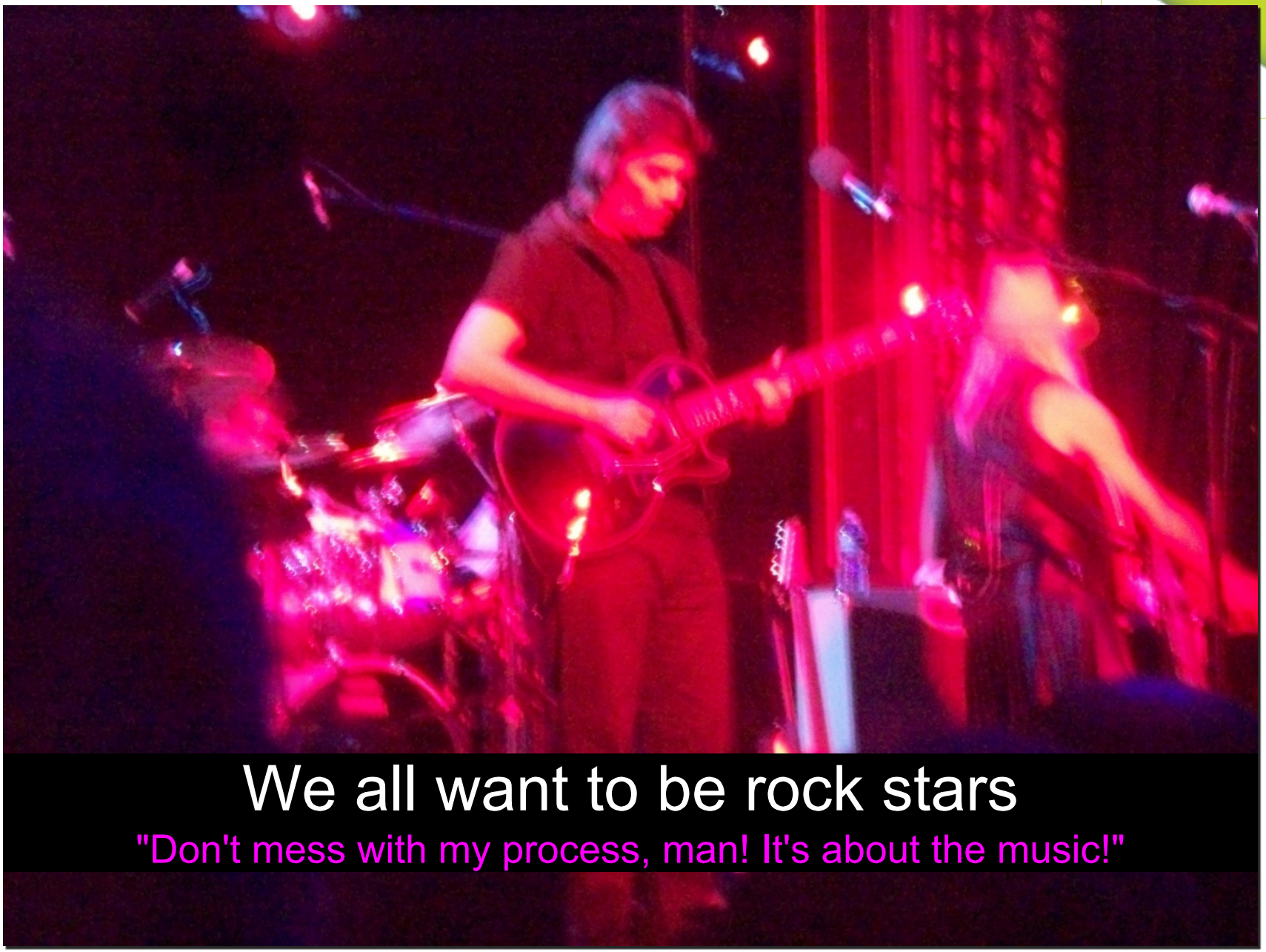
How Can We Be Certain We Have High-Quality JavaScript Code?

We can't afford to have errors occur in such complex systems.
A single error in the front-end can stop a user interface from responding entirely.

That means code we can have confidence in - error free, bug free, efficient and performant, with no memory leaks

Maintainability

Who cares?


We all want to be rock stars
"Don't mess with my process, man! It's about the music!"

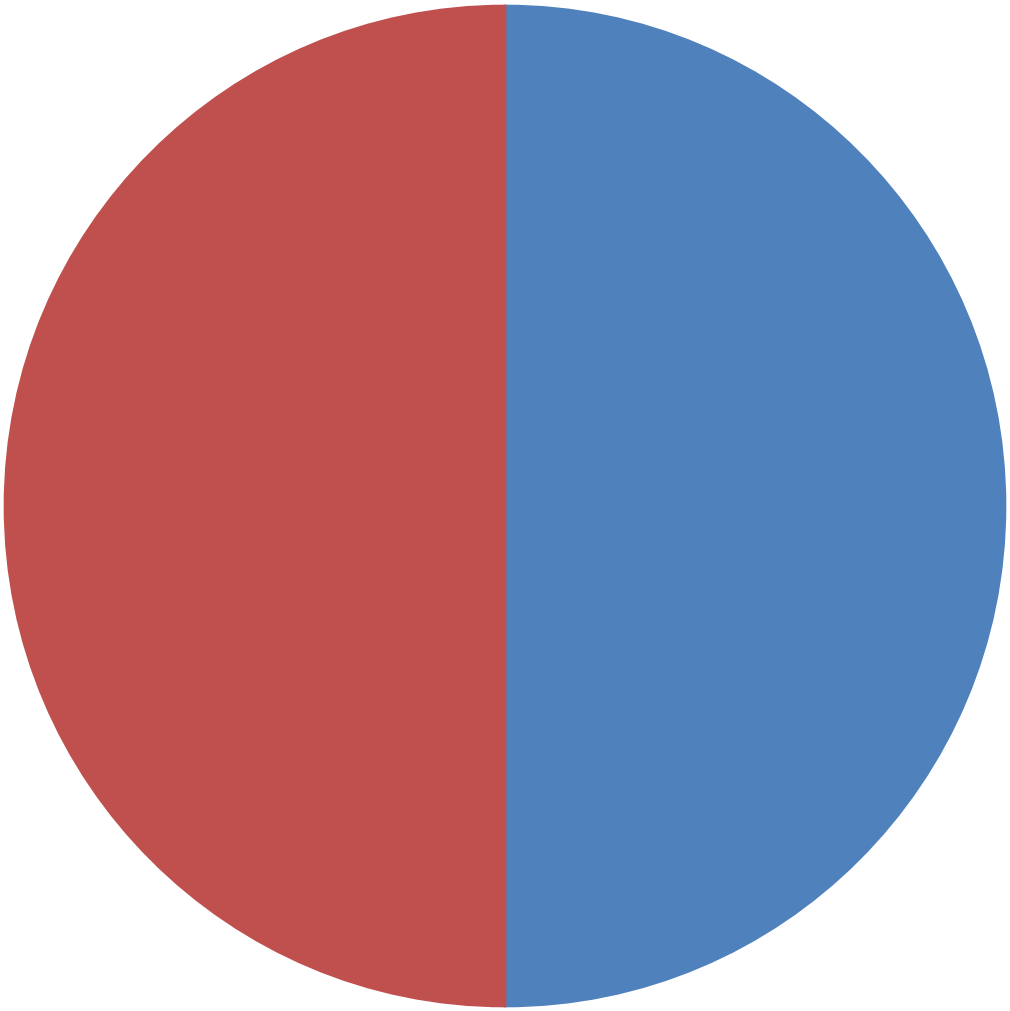
Maintainability

What is maintainable code?


Maintainable code works for five years without major changes
5
YEARS
• Intuitive
• Understandable
• Adaptable
• Extendable
• Debuggable
• Testable

Be kind to your future self.

Chris Eppstein,
Creator of Compass

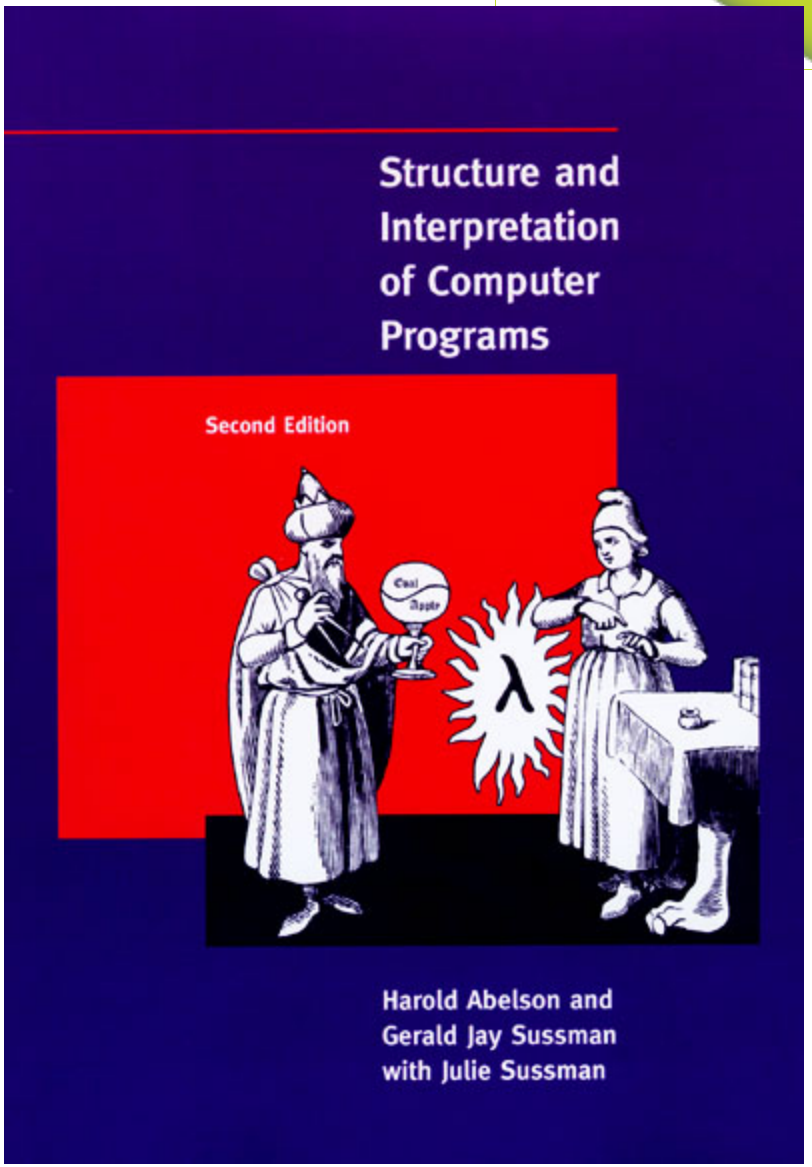


Code Style Guide

Communicating with each other through code

Programs are meant to be read by humans and only incidentally for computers to execute.

H. Abelson and G. Sussman, *The Structure and Interpretation of Computer Programs*



Code Conventions for the JavaScript Programming Language

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

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.

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.

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

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

Google JavaScript Style Guide

Revision 2.28

Aaron Whyte
Bob Jervis
Dan Pupius
Eric Arvidsson
Fritz Schneider
Robby Walker

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ▶ . You may toggle all summaries with the big arrow button:

▶ Toggle all summaries

Table of Contents

Important Note

Displaying Hidden Details in this Guide

▶ This style guide contains many details that are initially hidden from view. They are marked by the triangle icon, which you see here on your left. Click it now. You should see "Hooray" appear below.

Background

JavaScript is the main client-side scripting language used by many of Google's open-source projects. This style guide is a list of *do*s and *don't*s for JavaScript programs.

JavaScript Language Rules

http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml

Google JavaScript Style Guide

Revision 2.28

Aaron Whyte
Bob Jervis
Dan Pupius
Eric Arvidsson
Fritz Schneider
Robby Walker

Each style point has a summary for which additional information is available by toggling the accompanying arrow button that looks this way: ▶ . You may toggle all summaries with the big arrow button:

▶ Toggle all summaries

Table of Contents

Important Note

Displaying Hidden Details in this Guide

▶ This style guide contains many details that are initially hidden from view. They are marked by the triangle icon, which you see here on your left. Click it now. You should see "Hooray" appear below.

Background

JavaScript is the main client-side scripting language used by many of Google's open-source projects. This style guide is a list of *do*s and *don't*s for JavaScript programs.

JavaScript Language Rules

http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml


Tabs for indentation
4 spaces for indentation

```
if (wl && wl.length) {
            for (i = 0, l = wl.length; i < l; ++i) {
          p = wl[i];
          type = Y.Lang.type(r[p]);
          if (s.hasOwnProperty(p)) { if (merge && type
     == 'object') {

    Y.mix(r[p], s[p]);
} else if (ov || !(p in r)) {
                    r[p] = s[p];
                }
            }
        }
    }
```

```
if (wl && wl.length) {
    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);
        if (s.hasOwnProperty(p)) {
            if (merge && type == 'object') {
                Y.mix(r[p], s[p]);
            } else if (ov || !(p in r)) {
                r[p] = s[p];
            }
        }
    }
}
```

Nicholas C. Zakas
@slicknet

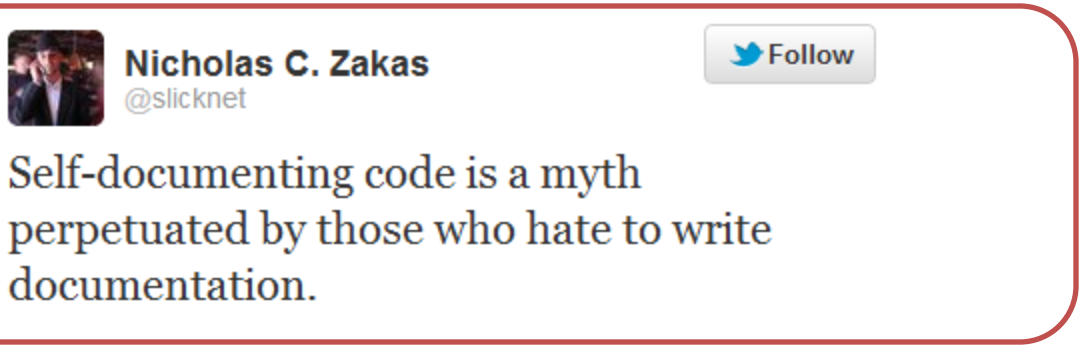
The less code on one line, the less likely you'll encounter a merge conflict.

twitter.com/slicknet/status/169903570047614977

```
if (found) { doSomething(); doSomethingElse(); }
else { doAThirdThing(); doAFourthThing(); }
```

```
if (found) {
    doSomething();
    doSomethingElse();
} else {
    doAThirdThing();
    doAFourthThing();
}
```


Comments



https://twitter.com/slicknet/statuses/3559283285

```
/**
 * Returns a new object containing all of the properties of
 * all the supplied objects.  The properties from later objects
 * will overwrite those in earlier objects.  Passing in a
 * single object will create a shallow copy of it.  For a deep
 * copy, use clone.
 * @method merge
 * @for YUI
 * @param arguments {Object*} the objects to merge.
 * @return {object} the new merged object.
 */
Y.merge = function() {
    var a = arguments, o = {}, i, l = a.length;
    for (i = 0; i < l; i = i + 1) {
        Y.mix(o, a[i], true);
    }
    return o;
};
```

Every method

```
if (mode) {
    switch (mode) {
        case 1: // proto to proto
            return Y.mix(r.prototype, s.prototype, ov, wl, 0,
                         merge);
        case 2: // object to object and proto to proto
            Y.mix(r.prototype, s.prototype, ov, wl, 0, merge);
            break; // pass through
        case 3: // proto to static
            return Y.mix(r, s.prototype, ov, wl, 0, merge);
        case 4: // static to proto
            return Y.mix(r.prototype, s, ov, wl, 0, merge);
        default:  // object to object is what happens below
    }
}
```

Difficult-to-understand code

```
while (element &&(element = element[axis])){ //NOTE: assignment
    if ( (all || element[TAG_NAME]) &&
       (!fn || fn(element)) ) {
            return element;
    }
}
```

Code that might seem to be wrong

Naming

- Use logical names for variables and functions
  - Don't worry about length
- Variable names should be nouns
- Function names should begin with a verb (i.e. getName())
  - Functions return booleans should begin with "is" or "has", such as isValid() or hasItem()
- Avoid useless names such as foo and temp

```
if (wl && wl.length) {
    for (i = 0, l = wl.length; i < l; ++i) {
        p = wl[i];
        type = Y.Lang.type(r[p]);
        if (s.hasOwnProperty(p)) {
            if (merge && type == 'object') {
                Y.mix(r[p], s[p]);
            } else if (ov || !(p in r)) {
                r[p] = s[p];
            }
        }
    }
}
```

```
// Variables, functions, properties methods
var myName = "Nicholas";

function sayName() {
    alert(myName);
}

var person = {
    name: "Nicholas",
    sayName: function() {
        alert(this.name);
    }
};
```

Camel Casing

What about acronyms?

```
var element = document.getElementById("my-div");

element.innerHTML = "Hello world!";
```

NO!

```
var xhr = new XMLHttpRequest();
```

NO

Camel Casing

```
// Constant-like variables
var HOVER_CLASS = "mouse-over";

// Constructors
function Person(name) {
    this.name = name;
}

var me = new Person("Nicholas");
```

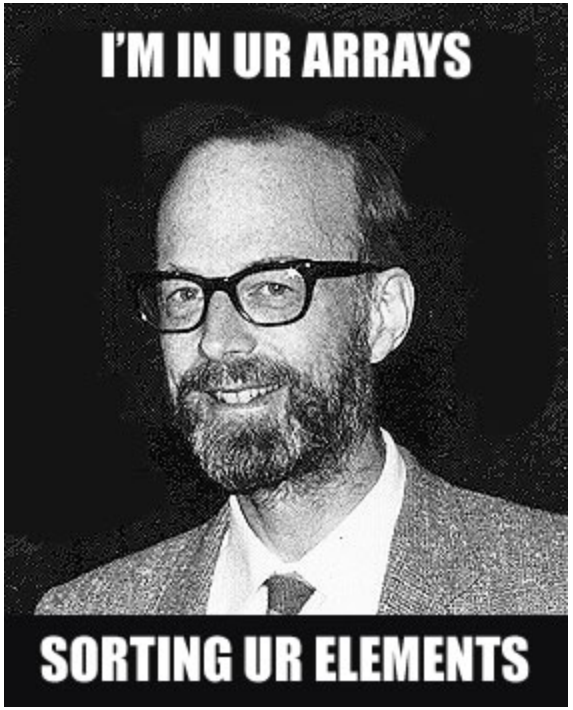
Camel Casing- Exceptions

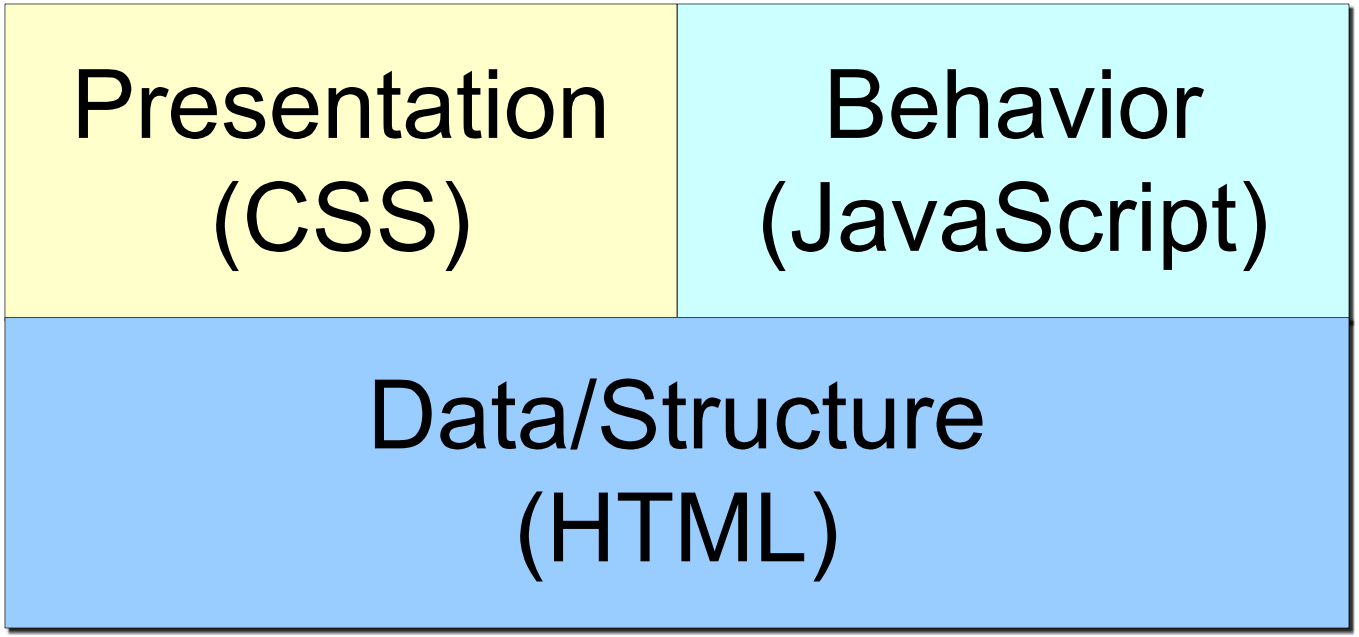
Programming Practices

Small patterns for common problems

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

C.A.R. Hoare,
Quicksort Developer




Presentation
(CSS)
Behavior
(JavaScript)
Data/Structure
(HTML)


Don’t cross
the streams

```
<button onclick="doSomething()">Click Me</button>
```

Keep JavaScript out of HTML

```
var element = document.getElementById("container");
element.innerHTML = "<div class=\"popup\"></div>";
```

Keep HTML out of JavaScript

```
.foo {
    width: expression(document.offsetWidth + "px");
}
```

Keep JavaScript out of CSS

```
var element = document.getElementById("container");
element.style.color = "red";
element.style.cssText = "background:blue;border:1px solid red";
```

Keep CSS out of JavaScript

```
//the wrong way!!!
function handleClick(event){

    var popup = document.getElementById("popup");
    popup.style.left = event.clientX + "px";
    popup.style.top = event.clientY + "px";
    popup.className = "reveal";

}
```

Event handlers should only handle events

```
//better, but still wrong
function handleClick(event){
    showPopup(event);
}

function showPopup(event){
    var popup = document.getElementById("popup");
    popup.style.left = event.clientX + "px";
    popup.style.top = event.clientY + "px";
    popup.className = "reveal";
}
```

Don't pass the event object around

```
//win!!
function handleClick(event){
    showPopup(event.clientX, event.clientY);
}

function showPopup(x, y){
    var popup = document.getElementById("popup");
    popup.style.left = x + "px";
    popup.style.top = y + "px";
    popup.className = "reveal";
}
```

Properly separated event handling

```
//don't add new methods
Array.prototype.awYeah = function(){
    alert("Aw yeah!");
};

//don't override methods
YUI.use = function(){
    alert("Aw yeah!");
};
```

Don't modify objects you don't own

If you didn't define the object yourself, you don't own it

NCZOnline

HOME BLOG WRITING SPEAKING DOWNLOADS ABOUT CONTACT

Maintainable JavaScript: Don't modify objects you don't own

Posted at March 2, 2010 09:00 am by Nicholas C. Zakas
Tags: JavaScript, Maintainable

The first talk I gave after arriving at Yahoo! was entitled Maintainable JavaScript (video). As with most topics I write or speak about, I didn't think it would be terribly controversial. The basis of the talk is that hacking around on your own and writing code in an enterprise environment are two different things. Web developers are truly unique in that none of us learned what we know in school; we all began as hobbyists one way or another and taught ourselves most (if not all) of what we know.

Professionalization

The professionalization of web development has been a difficult journey because of our disparate beginnings. Even those who end up at large companies such as Yahoo! inevitably began on their own, hacking around. Perhaps you were even "the web guy" at a small company and could do pretty much whatever you wanted. When the large companies started tapping this previously undiscovered resource, it brought a lot of hackers into a corporate environment where they were met with constraints. No longer a lone soldier in a small battle, all of these self-taught, self-directed individuals had to figure out how to work together as a team.

At the time that I gave the talk (2007), web development was evolving into front-end engineering and people were having trouble with the transition. Smart folks like Nate Koechley talked about the professionalization of front-end engineering (video) and how our discipline was evolving. My talk was aimed at the same goal: helping front-end engineers adapt to JavaScript development in a team environment by making sure that their code was as maintainable as possible.

Why can't I modify objects I don't own?

I still gets email and comments about Maintainable JavaScript, and the most popular question is, "why can't I modify objects I don't own?" JavaScript is, of course, a dynamic language that allows you to add and remove objects and their members at any point in time. For many, this is precisely why they enjoy the language: there are very few constraints imposed by the language. And I was telling them not to do this. Why?

Dependability

The simple explanation is that an enterprise software product needs a consistent and dependable execution environment to be maintainable. In other languages, you consider already-existing objects as libraries for you to use in order to complete your task. In JavaScript, people saw already-existing objects as a playground in which you could do anything you wanted. My point was that you should treat the already-existing JavaScript objects as you would a library of utilities. Don't override methods, don't add new methods, don't remove existing methods.

When you're the only one working on a project, it's easy to get away with these types of modifications because you know them and expect them. When working with a team on a large project, making changes

was an incredible mess and I'd be very happy if no engineers ever had to go through a similar exercise.

More of Me

Twitter: @slicknet
LinkedIn: Resume
GitHub: Projects
Slideshare: Presentations

My Books

http://nczonline.net/blog/2010/03/02/maintainable-javascript-dont-modify-objects-you-down-own/

```
function handleClick(event){
    showPopup(event.clientX, event.clientY);
}

function showPopup(x, y){
    var popup = document.getElementById("popup");
    popup.style.left = x + "px";
    popup.style.top = y + "px";
    popup.className = "reveal";
}
```

Avoid global functions and variables

```
var Controller = {
    handleClick: function(event){
        this.showPopup(event.clientX, event.clientY);
    },

    showPopup: function (x, y){
        var popup = document.getElementById("popup");
        popup.style.left = x + "px";
        popup.style.top = y + "px";
        popup.className = "reveal";
    }
};
```

Avoid global functions and variables

Create a single global (if necessary) and attach everything to it

```
var Controller = {
    addClass: function(element, className){
        element.className += " " + className;
    }
};
```

Throw your own errors

When you know a function will fail

```
var Controller = {
    addClass: function(element, className){
        if (!element) {
            throw new Error("addClass: 1st argument missing.");
        }
        element.className += " " + className;
    }
};
```

Throw your own errors

When you know a function will fail

```
var Controller = {
    process: function(items){
        if (items != null){
            items.sort();
            items.forEach(function(item){
                //do something
            });
        }
    }
};
```

Avoid null comparisons

```
var Controller = {
    process: function(items){
        if (items instanceof Array){
            items.sort();
            items.forEach(function(item){
                //do something
            });
        }
    }
};
```

Avoid null comparisons

Test for precisely what you want to know if it matters

- Use **instanceof** to test for specific object types
  - **object instanceof MyType**
- Use **typeof** to test for primitive types
  - **typeof value == "string"**
  - BEWARE: **typeof null == "object"**

```
function validate(value) {
    if (!value) {
        alert("Invalid value");
        location.href = "/errors/invalid.php";
    }
}
```

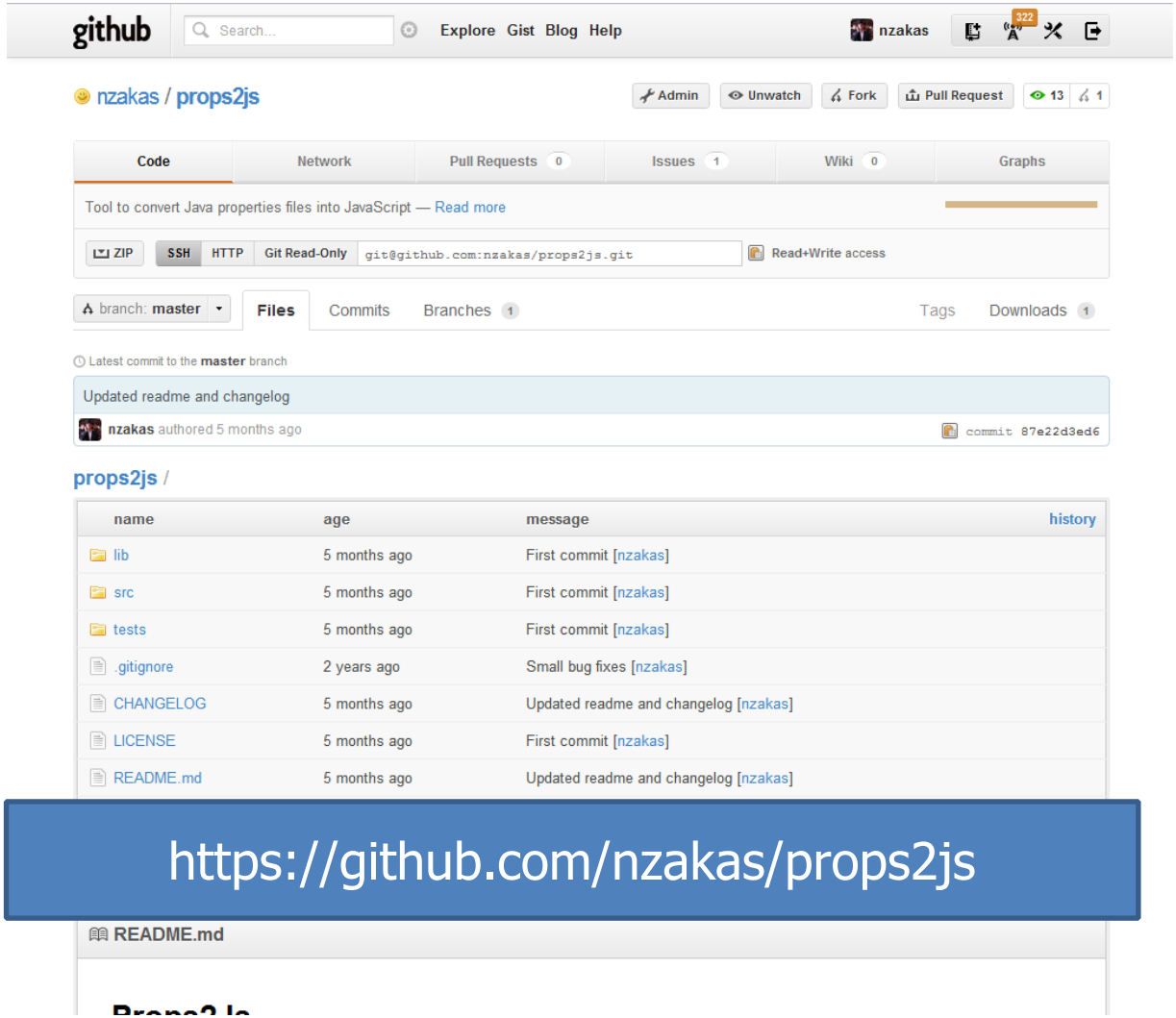
Separate config data

```
var config = {
    urls: {
        invalid: "/errors/invalid.php"
    },
    strs: {
        invalidmsg: "Invalid value"
    }
};

function validate(value) {
    if (!value) {
        alert(config.strs.invalidmsg);
        location.href = config.urls.invalid;
    }
}
```

Separate config data

- All URLs needed by the JavaScript
- Any strings that are displayed to the user
- Any HTML that needs to be created from JavaScript
- Settings (i.e., items per page)
- Repeated unique values
- Any value that may change in the future



https://github.com/nzakas/props2js

Now, you can see:

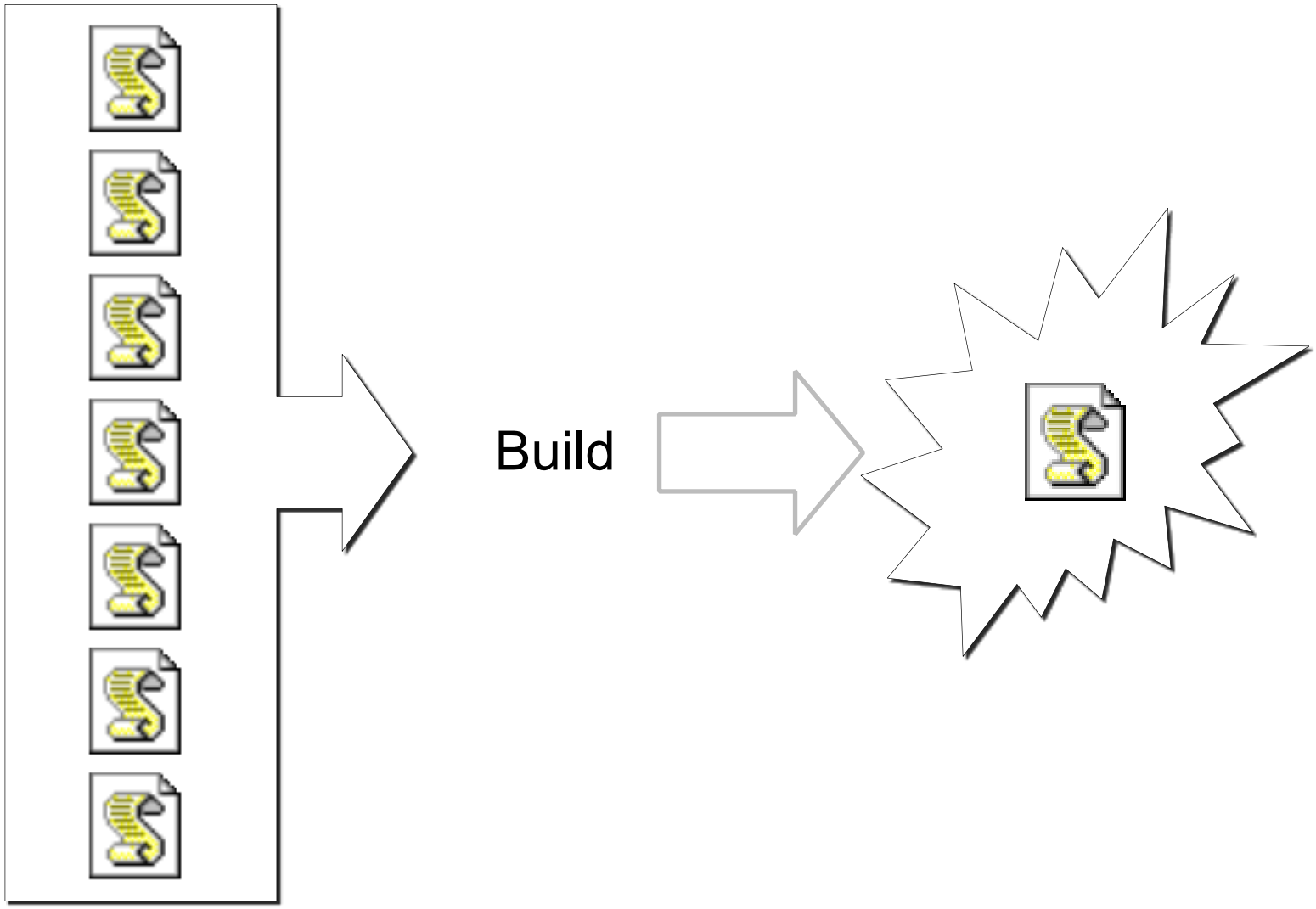
04 Using debuggers

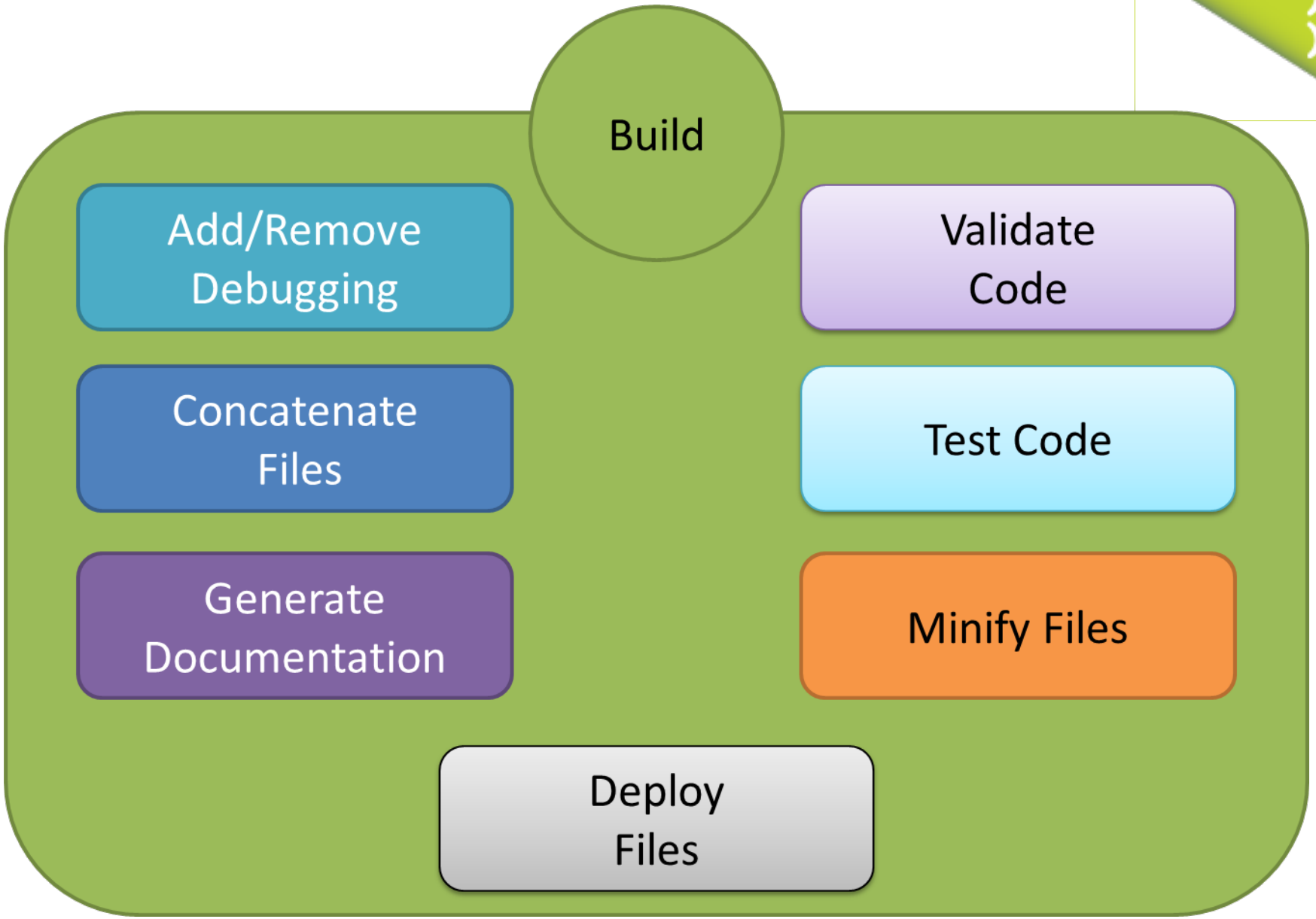
http://youtu.be/SqBEu-W7AXg

Automation

Make everyone’s life easier

Build Process




Build
Add/Remove Debugging
Concatenate Files
Generate Documentation
Validate Code
Test Code
Minify Files
Deploy Files

js-build-tools

Collection of Ant tasks for a JavaScript build process.

AntTasks

Ant tasks currently available

Updated Aug 13, 2009 by spo...@gmail.com

This page shows a list of ant task currently available in the JS Build Tools package.

preprocess

The JS Build Tools package comes with a Javascript preprocessor engine. This Ant task processes statements like #ifdef, #ifndef or #endif.

Parameters

- infile - Input file to process.
- outfile - Output file to write the processed contents to.
- defines - Comma separated list of defined values to check for.

Here is an example on how to use the preprocessor in a JS file:

```
// #ifdef somevalue1
someFunction();
// #endif
// #ifndef somevalue1
someOtherFunction();
```

https://github.com/moxiecode/js-build-tools

The preprocessor enables you to include/exclude code depending in defined values. This is very useful if you want to for example remove all debug code when making a distribution package.

yuicompress

This task compresses the specified script using the YUICompressor package. The YUICompressor will remove all unneeded white space and obfuscate long variable names.

Parameters

Add/Remove Debugging

@use JSDoc

Index

DRAFT

Getting Started

An introduction to JSDoc 3
A quick-start to documenting JavaScript with JSDoc.

JSDoc 3 Tag Dictionary

@augments or @extends
This object adds onto a parent object.

@borrows
This object uses something from another object.

@constructor or @class
This function is intended to be called with the "new" keyword.

@classdesc
Use the following text to describe the entire class.

@constant or @const
Document an object as a constant.

@copyright
[todo] Document some copyright information.

@constructs
[todo] This function member will be the constructor for the previous class.

@default
Document the default value.

@deprecated
[todo] This is no longer the preferred way.

@event
[todo] Document an event.

@example
[todo] Provide an example.

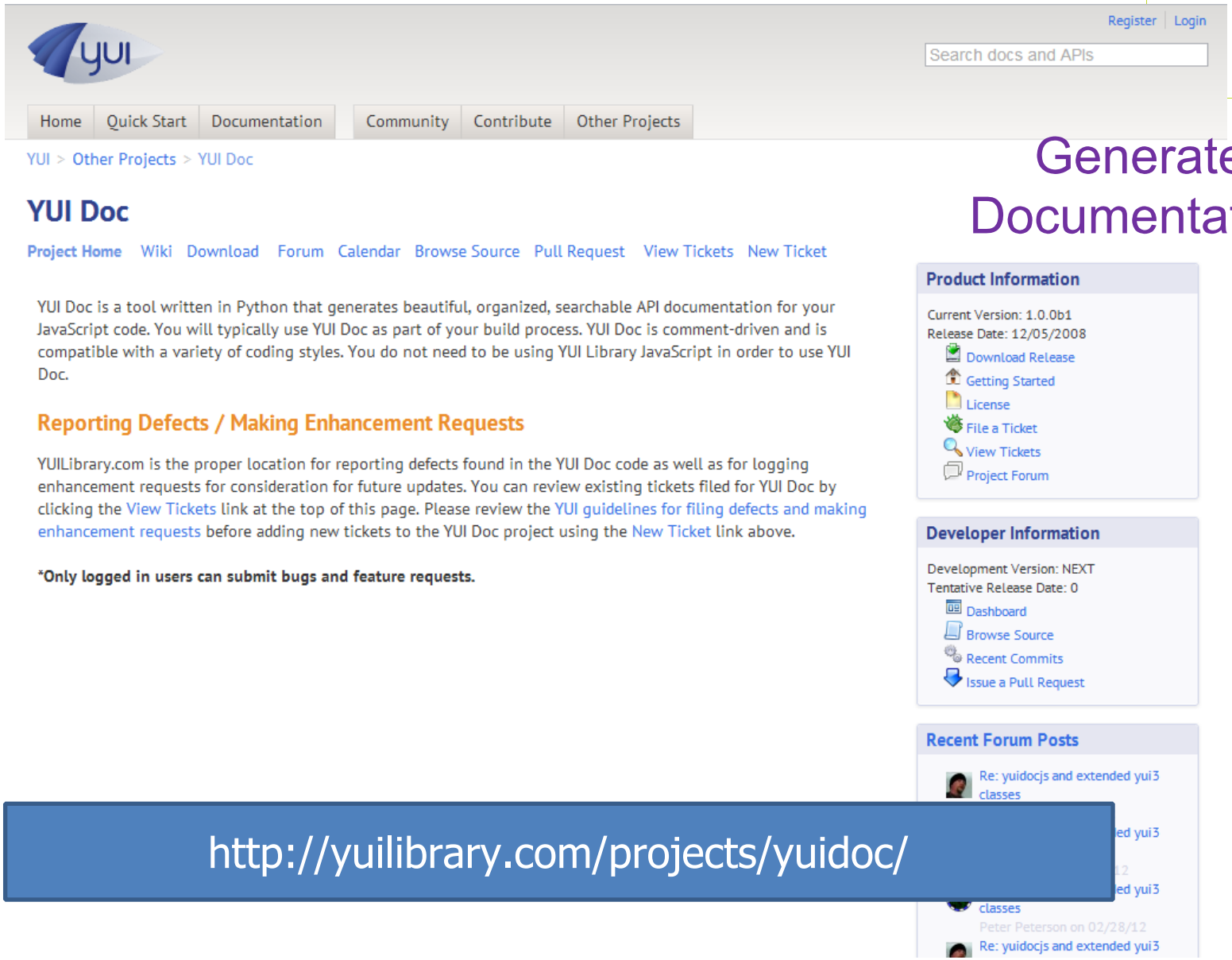
@throws or @exception
[todo] Describe what errors could be thrown.

@exports
[todo] Document the name of a JavaScript module.

Generate Documentation

http://usejsdoc.org

Generate Documentation



http://yuilibrary.com/projects/yuidoc/

docco.coffee

¶ **Docco** is a quick-and-dirty, hundred-line-long, literate-programming-style documentation generator. It produces HTML that displays your comments alongside your code. Comments are passed through Markdown, and code is passed through Pygments syntax highlighting. This page is the result of running Docco against its own source file.

If you install Docco, you can run it from the command-line:

```
docco src/*.coffee
```

...will generate linked HTML documentation for the named source files, saving it into a `docs` folder.

The source for Docco is available on GitHub, and released under the MIT license.

To install Docco, first make sure you have Node.js, Pygments (install the latest dev version of Pygments from its Mercurial repo), and CoffeeScript. Then, with NPM:

```
sudo npm install docco
```

Partners in Crime:

- If **Node.js** doesn't run on your platform, or you'd prefer a more convenient package, get Ryan Tomayko's Rocco, the

- If Python's more your speed, take a look at Nick Fitzgerald's Pycco.

- For **Clojure** fans, Fogus's Marginalia is a bit of a departure from "quick-and-dirty", but it'll get the job done.

- **Lua** enthusiasts can get their fix with Robert Gieseke's Locco.

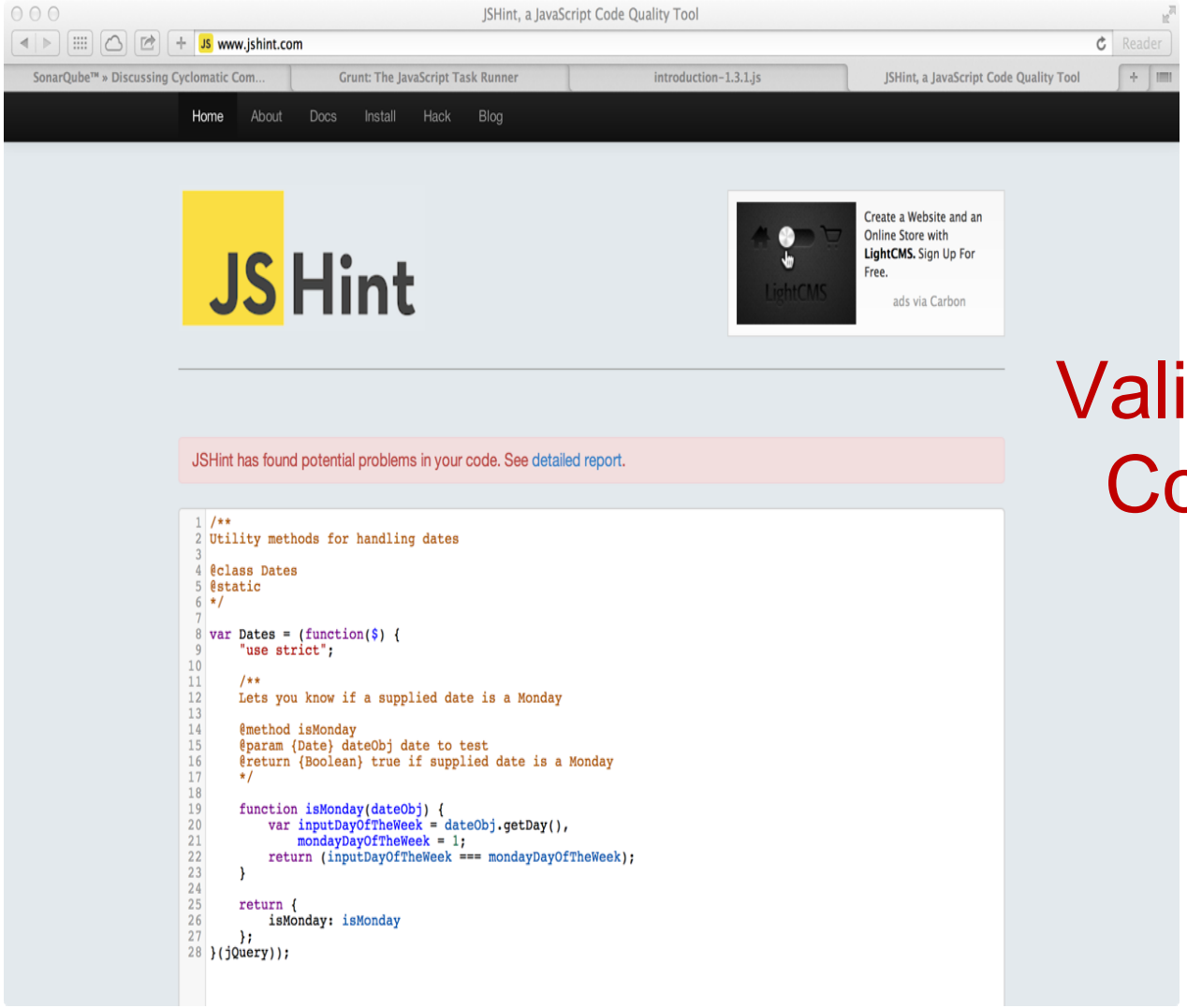
Generate Documentation

http://jashkenas.github.com/docco/

JSHint (and JSLint) allow you to perform an analysis on your code, without it actually running.

It spots what look like errors in your code so you can fix them early before they cause any issues - things like undeclared variables, or variables declared but not used (good for spotting spelling mistakes!).

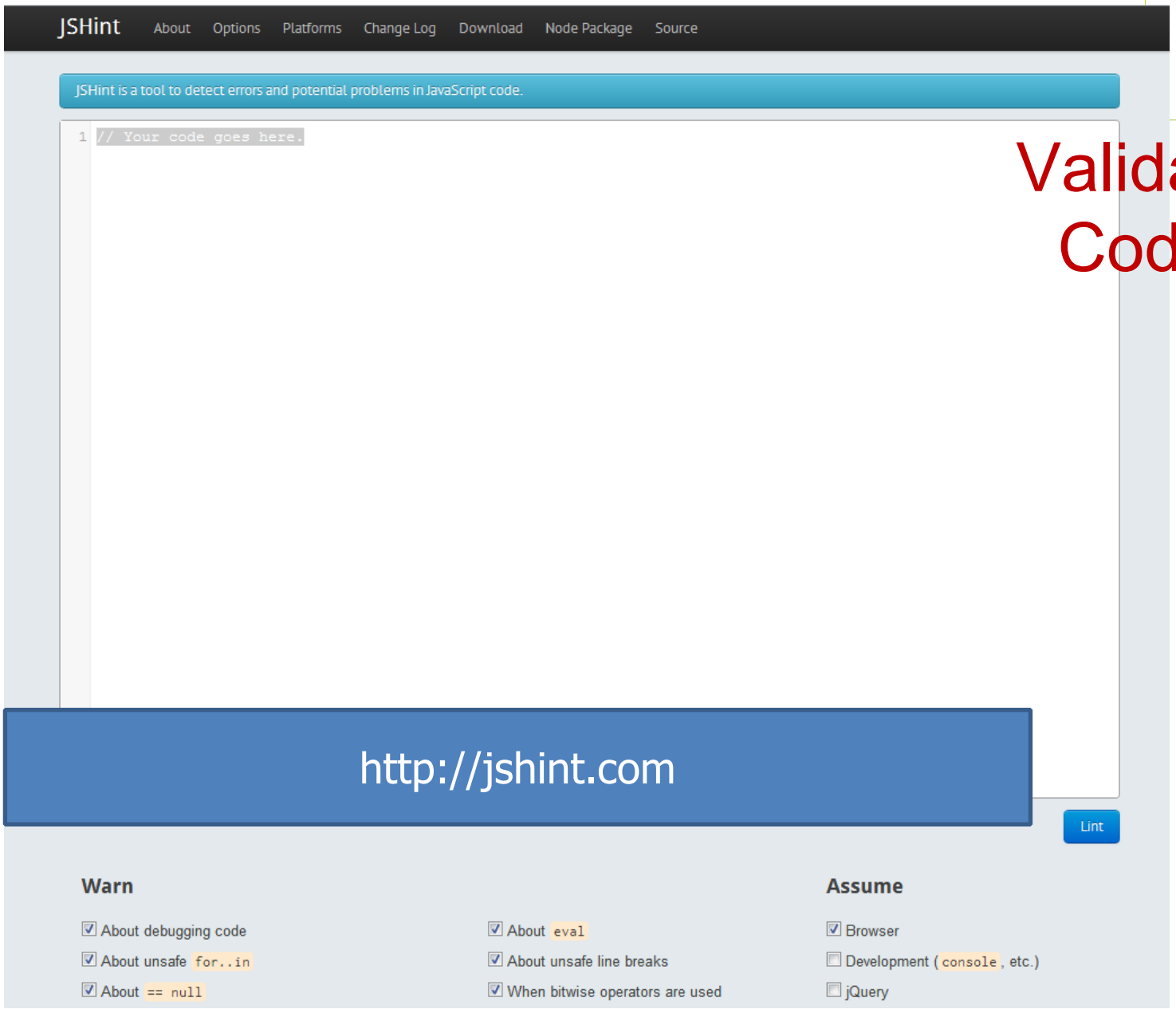
Static Code Analysis



Validate Code

Validate Code

http://jslint.com



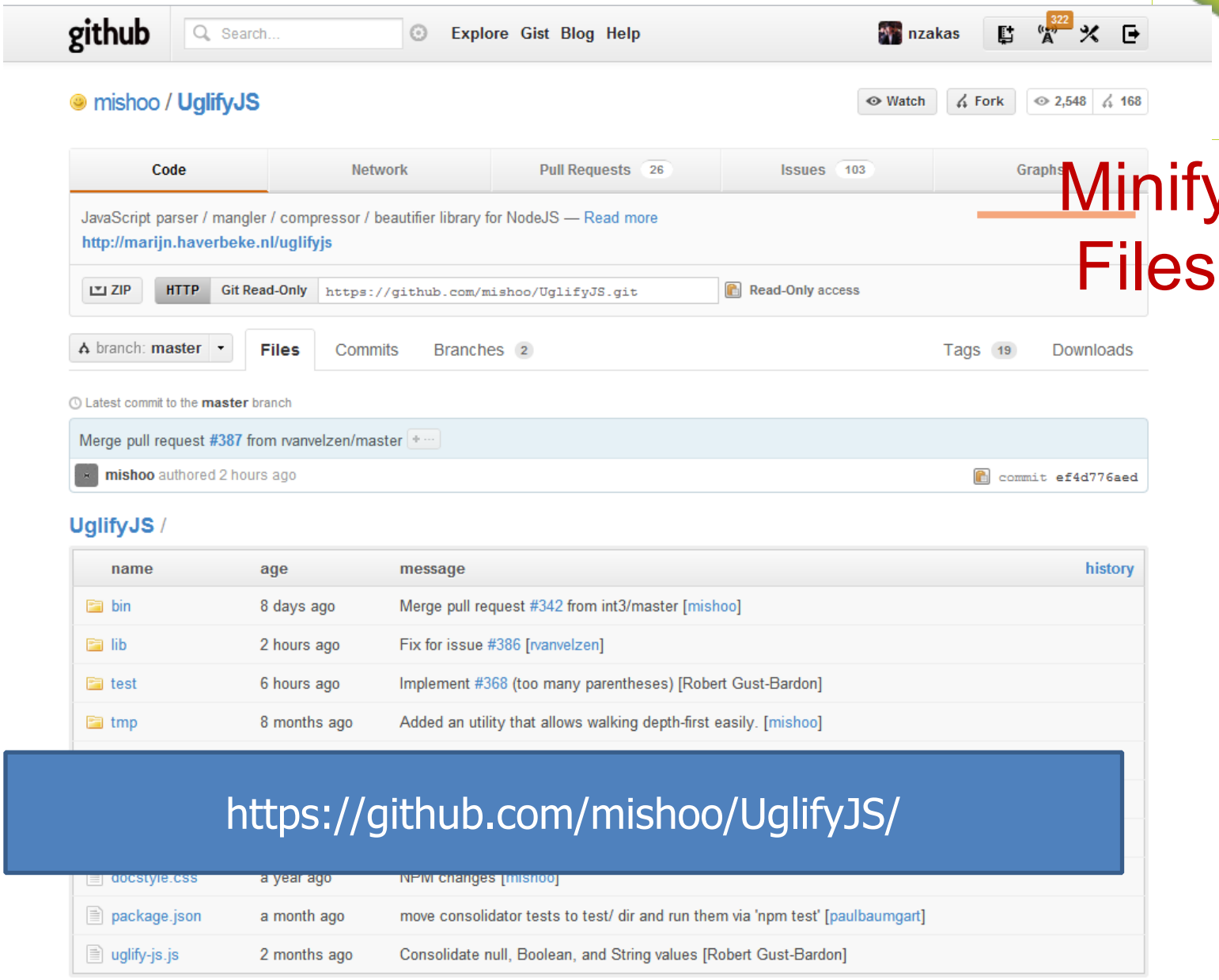
Validate Code

http://jshint.com

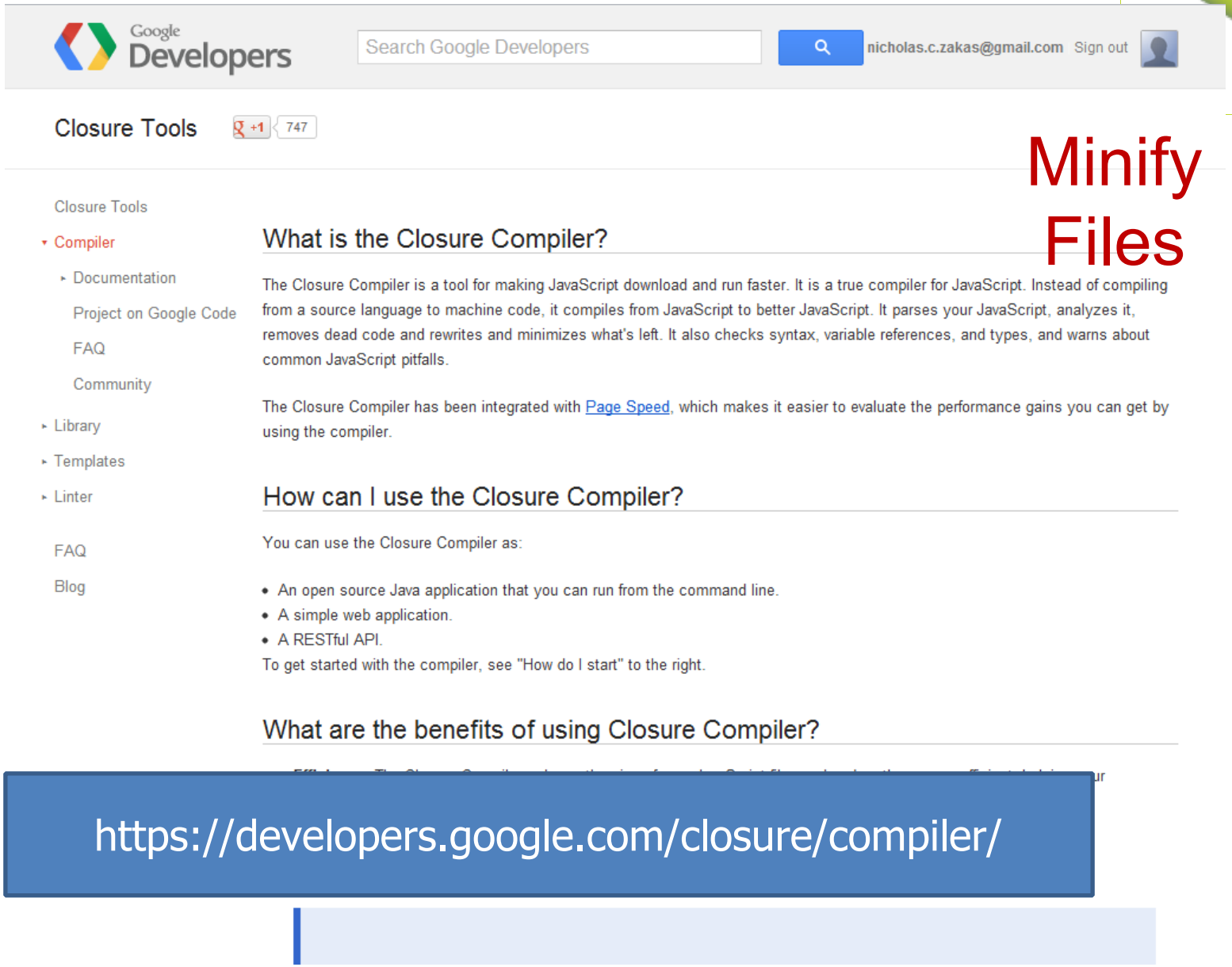
Minify Files

http://yuilibrary.com/projects/yuicompressor/

Minify Files



https://github.com/mishoo/UglifyJS/



Minify Files

https://developers.google.com/closure/compiler/

Unit tests are a fairly new concept to JS developers, but familiar to those of many other languages.

A unit test is a small function that calls one of the functions in your code with a set of known inputs and checks that the output of each call is what was expected.

We use Jasmine, but there are many other unit test frameworks out there.

You create a HTML file, include the Jasmine library, the JS file you wrote that you wish to test, and the JS file you wrote your unit tests in.

It then runs the tests automatically and shows you the results.

Write Unit Test

Now, you can see:

05 JavaScript Testing With Jasmine Introduction

http://youtu.be/2CiKlcABwW0

Now, you can see:

06 JavaScript Testing With Jasmine The First Application

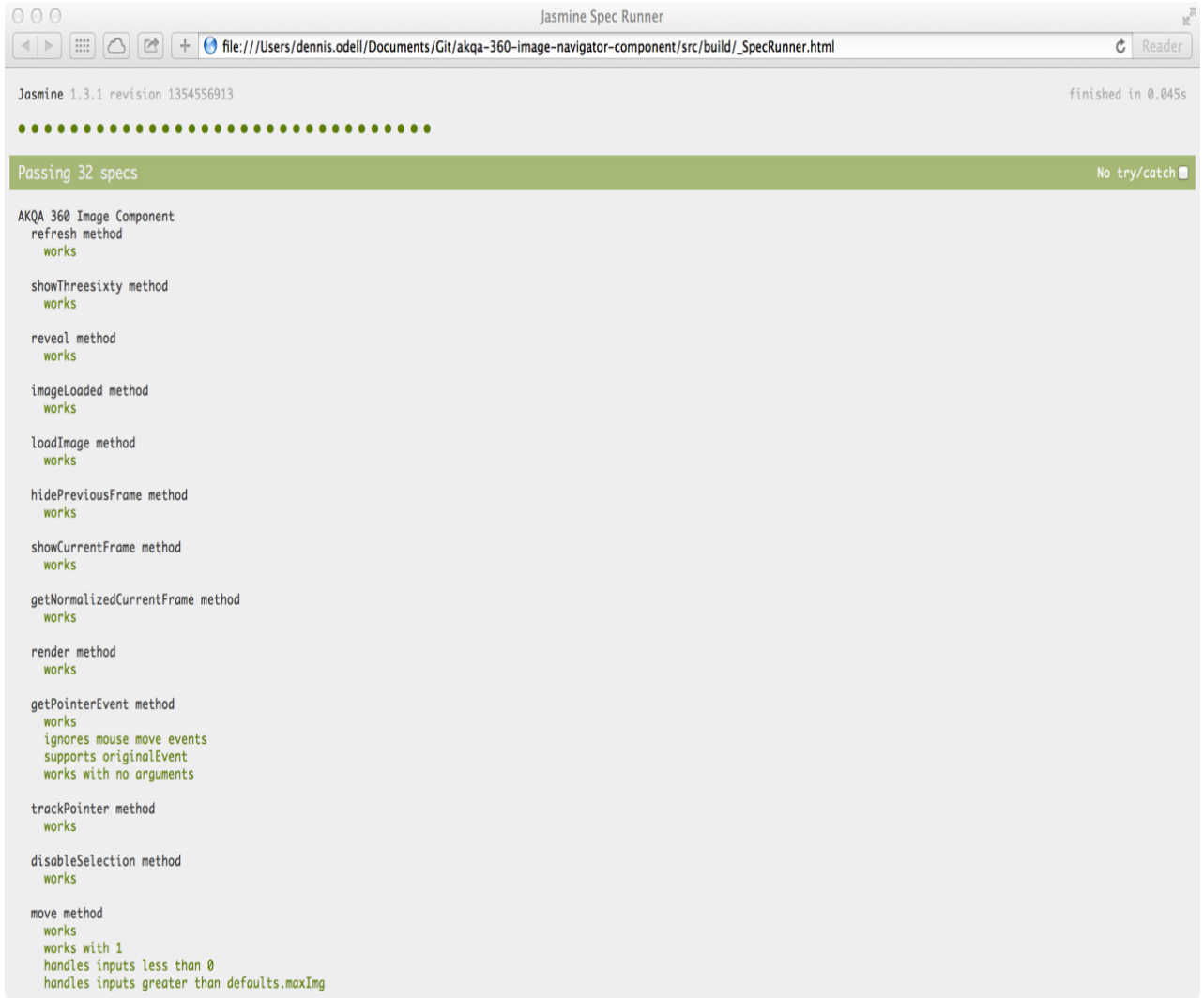
http://youtu.be/_e53fKh1XuQ

Now, you can see:

07 JavaScript Testing With Jasmine Bundles for textmate

http://youtu.be/sHn2VcYKlFs

Assuming the tests pass, you will see the green bar at the top of the screen, together with a list of the individual tests that ran and their results.



Here's an example unit test for our isMonday method.

Groups of tests in Jasmine are wrapped in a call to its 'describe' method, and individual tests are wrapped in a call to its 'it' method. You can then describe each test with a string before containing the test itself within a function.

The unit tests execute the isMonday method with known inputs, and check the output is as expected with the 'expect' and 'toBe' methods of Jasmine.

Unit tests prove that your code works as expected, which gives you and your team confidence in it.

Example

```
describe('Calculator', function () {
    var calc;
    beforeEach(function () {
        calc = new Calculator();
    });
    it('should multiply', function () {
        expect(calc.multiply(6, 7)).toBe(42);
    });
});
```

Now, you can see:

08 JavaScript Testing With Jasmine Matchers

http://youtu.be/TpDXZ5Bpf9k

We want to test code in isolation
here the code is the 'keypress' event handler
and isolation means not invoking the getMatch() method

```
'keypress': function (element, event) {
    var pattern = this.element.val();
    pattern += String.fromCharCode(event.charCode);
    var match = this.getMatch(pattern);
    if (match) {
        event.preventDefault();
        this.element.val(match);
    }
}
```

We can mock the getMatch() method
decide how the mock should behave
verify that the mocked method was called correctly

```
spyOn(autoComplete, 'getMatch').andReturn('monique');

$('#name').trigger($.Event('keypress', {charCode: 109}));

expect(autoComplete.getMatch).toHaveBeenCalledWith('m');

expect($('#name')).toHaveValue('monique');
```

Global functions are properties of the window object

```
openPopup: function (url) {
    var popup = window.open(url, '_blank', 'resizable');
    popup.focus();
}
```

```
var popup;
spyOn(window, 'open').andCallFake(function () {
    popup = {
        focus: jasmine.createSpy()
    };
    return popup;
});

autoComplete.openPopup('zealake.com');

expect(window.open).toHaveBeenCalledWith('zealake.com', '_blank', 'resiza

expect(popup.focus).toHaveBeenCalledWith();
```

Constructors are functions
with this being the object to construct

```
this.input = new window.AutoComplete(inputElement, {
    listUrl: this.options.listUrl
});
this.input.focus();
```

```
spyOn(window, 'AutoComplete').andCallFake(function () {
    this.focus = jasmine.createSpy();
});

expect(window.AutoComplete.callCount).toBe(1);
var args = window.AutoComplete.mostRecentCall.args;
expect(args[0]).toBe('#name');
expect(args[1]).toEqual({listUrl: '/someUrl'});

var object = window.AutoComplete.mostRecentCall.object;
expect(object.focus).toHaveBeenCalledWith();
```

We want the tests to be fast

So don't use Jasmine waitsFor()

But we often need to wait

- For animations to complete
- For AJAX responses to return

```
delayHide: function () {
    var self = this;
    setTimeout(function () {
        self.element.hide();
    }, this.options.hideDelay);
}
```

Use Jasmine's mock clock

Control the clock explicitly

Now the test completes in milliseconds without waiting

```
jasmine.Clock.useMock();

autoComplete.delayHide();

expect($('#name')).toBeVisible();

jasmine.Clock.tick(500);

expect($('#name')).not.toBeVisible();
```

new Date() tends to return different values over time
Actually, that's the whole point :)
But how do we test code that does that?
We cannot expect on a value that changes on every run
We can mock the Date() constructor!

```
var then = new Date();

jasmine.Clock.tick(42000);
var now = new Date();

expect(now.getTime() - then.getTime()).toBe(42000);
```

Keep Date() and setTimeout() in sync

```
jasmine.GlobalDate = window.Date;

var MockDate = function () {
    var now = jasmine.Clock.defaultFakeTimer.nowMillis;
    return new jasmine.GlobalDate(now);
};
MockDate.prototype = jasmine.GlobalDate.prototype;
window.Date = MockDate;
jasmine.getEnv().currentSpec.after(function () {
    window.Date = jasmine.GlobalDate;
});
```

To test in isolation

To vastly speed up the tests

Many options

- can.fixture
- Mockjax
- Sinon

```
can.fixture('/getNames', function (original, respondWith) {
    respondWith({list: ['rachel', 'lakshmi']});
});
autoComplete = new AutoComplete('#name', {
    listUrl: '/getNames'
});
jasmine.Clock.tick(can.fixture.delay);
```

```
respondWith(500); // Internal server error
```

Supply the markup required by the code

Automatically cleanup markup after every test

Various solutions

- Built into QUnit as #qunit-fixture
- Use jasmine-jquery

```
var fixtures = jasmine.getFixtures();
fixtures.set(fixtures.sandbox());

$('<input id="name">').appendTo('#sandbox');

autoComplete = new AutoComplete('#name');
```

How do we test that an event was triggered?

Or prevented from bubbling?

Use jasmine-jquery!

```
'keypress': function (element, event) {
    var pattern = this.element.val() +
                              String.fromCharCode(event.charCode);
    var match = this.getMatch(pattern);
    if(match) {
        event.preventDefault();
        this.element.val(match);
    }
}
```

```
keypressEvent = spyOnEvent('#name', 'keypress');

$('#name').trigger($.Event('keypress', {charCode: 109}));

expect(keypressEvent).toHaveBeenPrevented();
```

Now, you can see:

09 JavaScript Testing With Jasmine JavaScript Koans

http://youtu.be/it4BXV49B9g

Now, you can see:

10 JavaScript Automated Testing Ruby Gem

http://youtu.be/PWUJHN_0L4g

Now, you can see:

11 JavaScript Automated Testing Using LiveReload

http://youtu.be/PWUJHN_0L4g

Now, you can see:

12 JavaScript Automated Testing jasmine headless webkit

http://youtu.be/DohifjP2ThQ

We use three main tools for testing. The first is Grunt, a JS task runner built on Node.js.

It can be configured to run JSHint and Jasmine unit tests automatically via plugin ‘tasks’.

This gets run on the developer’s local machine before their code is committed.


Grunt: The JavaScript Task Runner
gruntjs.com
Reader
SonarQube™ » Discussing Cyclomatic Complexity
Grunt: The JavaScript Task Runner
Getting Started
Plugins
Documentation
API
GRUNT
The JavaScript Task Runner
Latest Version
Stable: v0.4.1
Development: N/A
BACKBONECONF 2013
The conference for common sense JavaScript applications, July 31-August 1 in Boston.
Ads by Bocoup.
Latest News
Grunt 0.4.1 released
March 13, 2013
Grunt 0.4.0 released
February 18, 2013
Why use a task runner?
In one word: automation. The less work you have to do when performing repetitive tasks like minification, compilation, unit testing, linting, etc, the easier your job becomes. After you've configured it, a task runner can do most of that mundane work for you—and your team—with basically zero effort.
Why use Grunt?
The Grunt ecosystem is huge and it's growing every day. With literally hundreds of plugins to choose from, you can use Grunt to automate just about anything with a minimum of effort. If someone hasn't already built what you need, authoring and publishing your own Grunt plugin to npm is a breeze.
Available Grunt plugins
Many of the tasks you need are already available as Grunt Plugins, and new plugins are published every day. While the plugin listing is more complete, here's a few you may have heard of.

Build

The second tool we use for testing is BrowserStack.

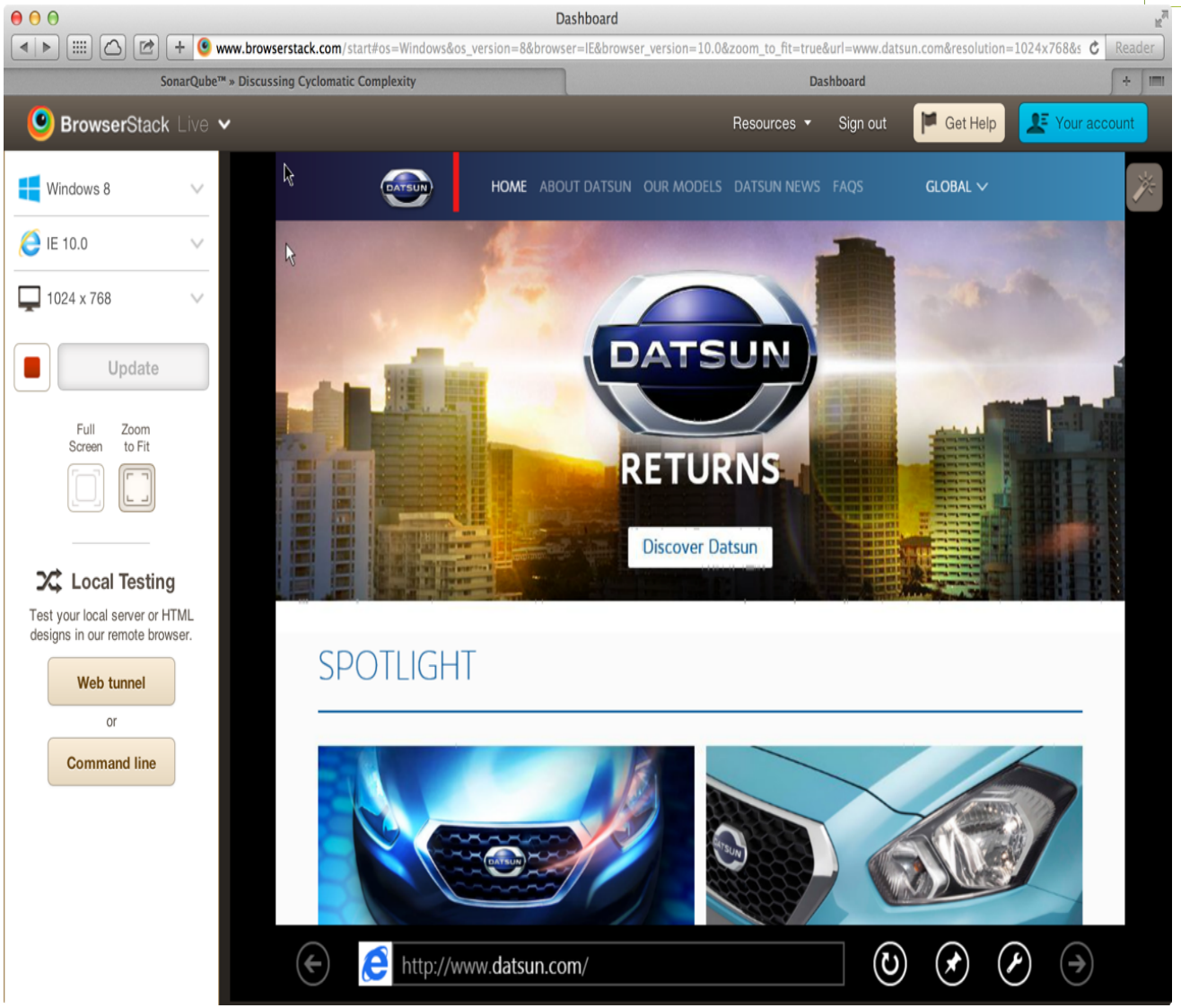
This is a site that spins up virtual machines on the fly which contain only browsers and dev tools.

You can select an OS and a browser, give it a URL, and it'll spin up a VM which you can then interact with as if you were using the real thing.

Desktop & mobile OSes are supported, and it allows the creation of a ‘tunnel’ through which you can run local sites from your machine within their VMs.

Very handy for mobile and old IE testing!

Automated & Manual Testing

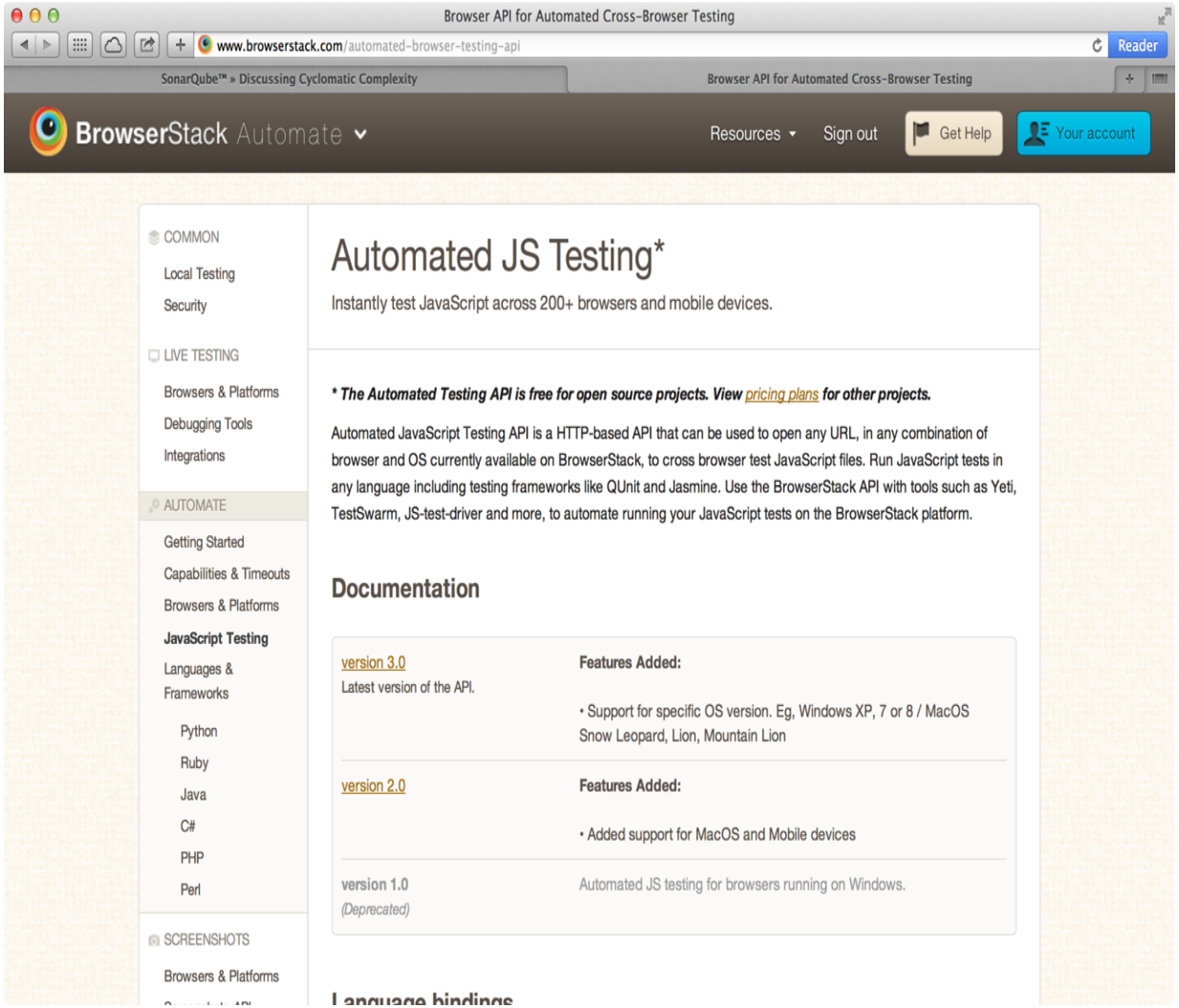


BrowserStack also have an automated service API for testing JavaScript code with.

It allows you to script up a number of VMs that connect to your unit tests for running automatically across a wide selection of browsers. They then pass you back the results for you to parse and check for errors.

This is good for running on a build server.

Automated & Manual Testing



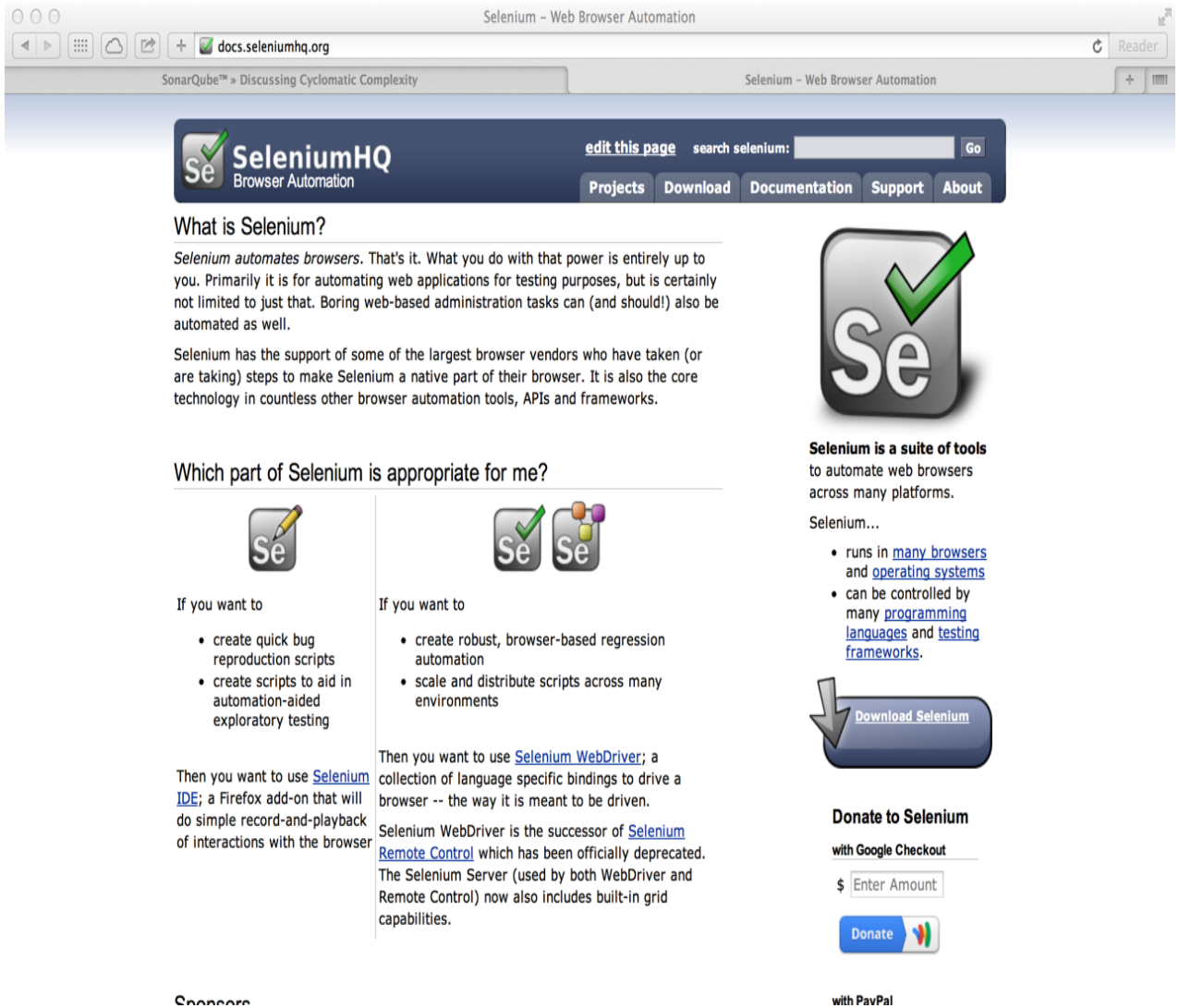
Build

Finally, the third tool we use for testing is Selenium - this allows you to script interaction testing in a browser, allowing you to test behaviour at a high level.

E.g. if I click in one place, does a modal window popup somewhere else?

This is best run on a build and/or staging server.

Automated & Manual Testing



Build

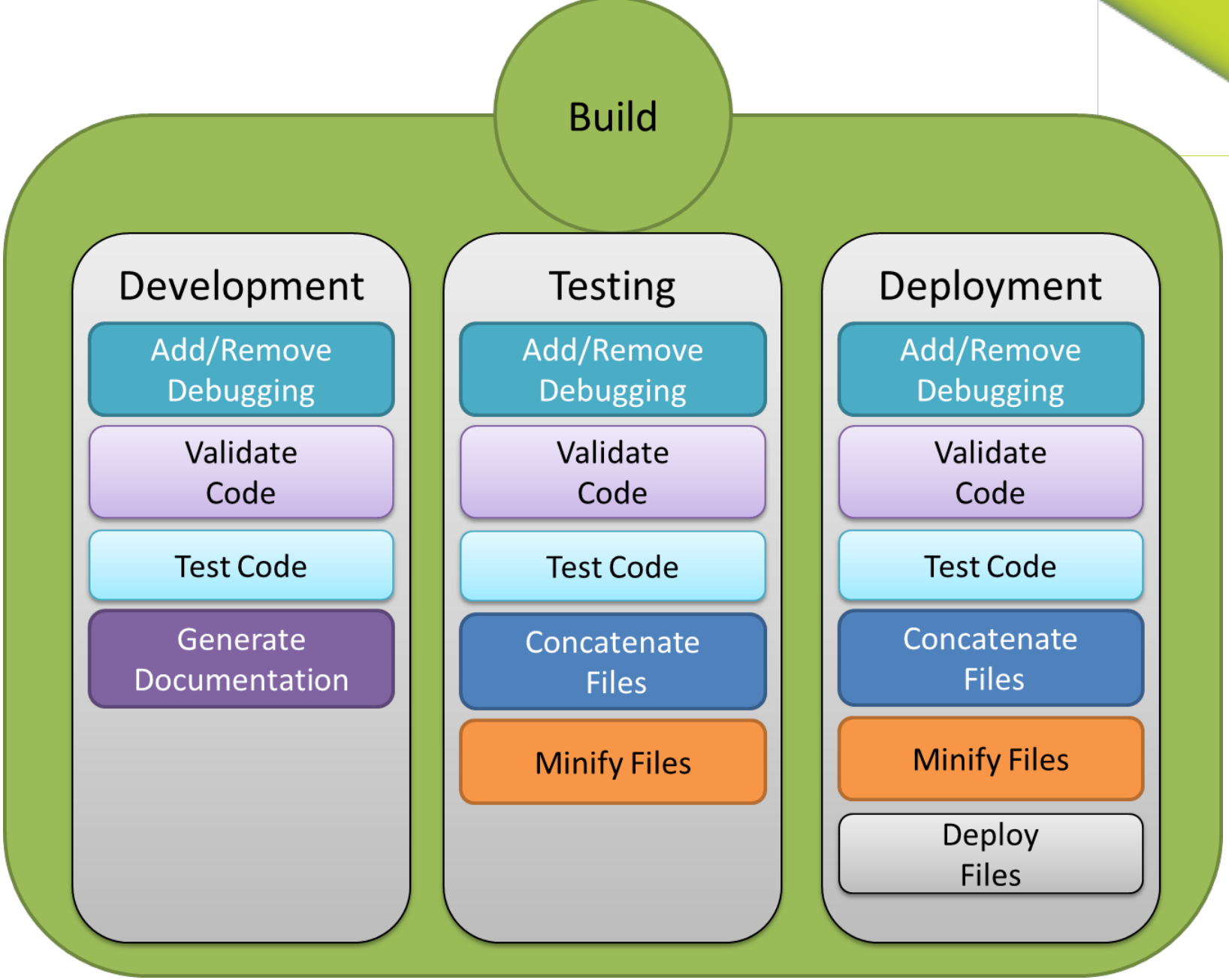
So we use Grunt, BrowserStack and Selenium in addition to manual, cross-browser checks to create a quadruple-lock on our testing process, right from the local development stage all the way up to directly on the pre-release server.

Configure Grunt To Run Static Code Analysis and Unit Tests

Run Unit Tests Cross-Browser Via BrowserStack API

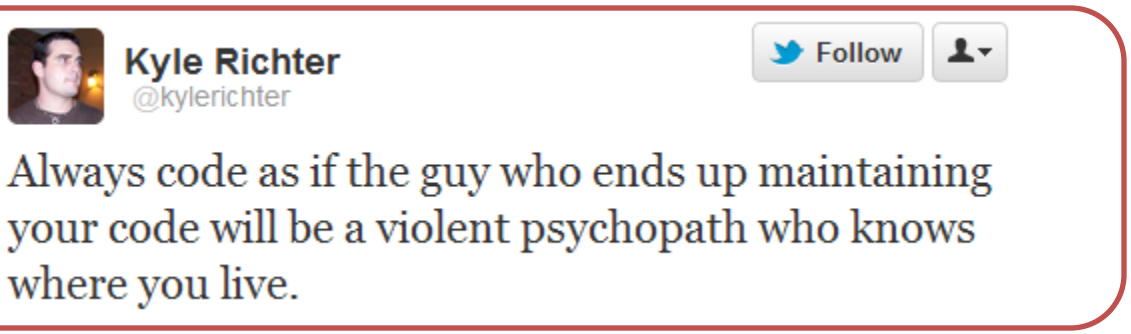
Use Selenium For Automated Integration Testing

Perform Manual, Cross-Browser Testing


Build
Development
Add/Remove Debugging
Validate Code
Test Code
Generate Documentation
Testing
Add/Remove Debugging
Validate Code
Test Code
Concatenate Files
Minify Files
Deployment
Add/Remove Debugging
Validate Code
Test Code
Concatenate Files
Minify Files
Deploy Files

Recap

- Code style guidelines ensure everyone's speaking the same language
- Loose coupling of layers make changes and debugging easier
- Good programming practices allow for easier debugging
- Code organization and automation help to bring sanity to an otherwise crazy process



https://twitter.com/kylerichter/status/1510115129269452 9

Thank you!

Globant
we are ready