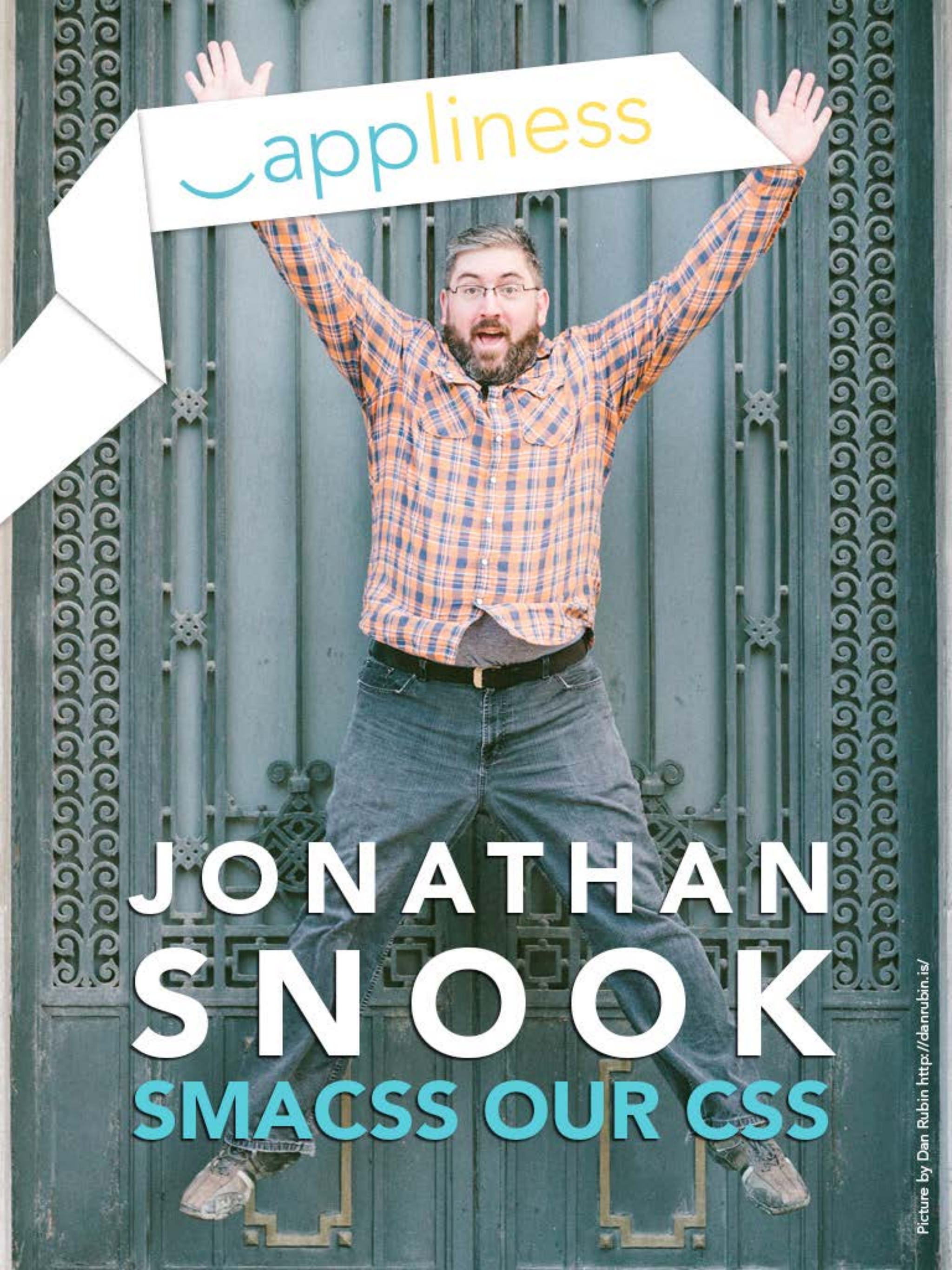

appliness
JONATHAN SNOOK
SMACSS OUR CSS
Picture by Dan Rubin http://danrubin.is/

Combining CSS Shapes with CSS Regions

by Zoltan Horvath

by Zoltan Horvath

"I'm incredibly excited to talk about these features and how you can combine them together."

HOLY CRAP - ANOTHER JAVASCRIPT SITE!

I have been working on rendering for almost a year now. Since I landed the initial implementation of Shapes on Regions in both Blink and WebKit, I'm incredibly excited to talk a little bit about these features and how you can combine them together.

Don't know what CSS Regions and Shapes are? Start here!

The first ingredient in my HTML alchemy kitchen is CSS Regions. With CSS Regions, you can flow content into multiple styled containers, which gives you enormous creative power to make magazine style layouts. The second ingredient is CSS Shapes, which gives you the ability to wrap content inside or outside any shape. In this post I'll talk about the "shape-inside" CSS property, which allows us to wrap content inside an arbitrary shape.

Let's grab a bowl and mix these two features together, CSS Regions and CSS Shapes to produce some really interesting layouts!

In the latest Chrome Canary and Safari WebKit Nightly, after enabling the required experimental features, you can flow content continuously through multiple kinds of shapes. This rocks! You can step out from the rectangular text flow world and break up text into multiple, non-rectangular shapes.

DEMO

If you already have the latest Chrome Canary/Safari WebKit Nightly, you can just go ahead and try a simple example on codepen.io. If you are too lazy, or if you want to extend your mouse button life by saving a few button clicks, you can continue reading.

In the picture above we see that the "Lorem ipsum" story flows through 4 different, colorful regions. There is a circle shape on each of the first two fixed size regions. Check out the code below to see how we apply the shape to the region. It's pretty straightforward, right?

```
#region1, #region2 {
    -webkit-flow-from: flow;
    background-color: yellow;
    width: 200px;
    height: 200px;
    -webkit-shape-inside: circle(50%, 50%, 50%);
}
```

The content flows into the third (percentage sized) region, which represents a heart (drawn by me, all rights reserved). I defined the heart's coordinates in percentages, so the heart will stretch as you resize the window.

```
#region3 {
    -webkit-flow-from: flow;
    width: 50%;
    height: 400px;
    background-color: #EE99bb;
    -webkit-shape-inside: polygon(11.17% 10.25%,2.50% 30.56%,3.92%
55.34%,12.33% 68.87%,26.67% 82.62%,49.33% 101.25%,73.50% 76.82%,85.17%
65.63%,91.63% 55.51%,97.10% 31.32%,85.79% 10.21%,72.47% 5.35%,55.53%
14.12%,48.58% 27.88%,41.79% 13.72%,27.50% 5.57%);
}
```

The content that doesn't fit in the first three regions flows into the fourth region. The fourth region (see the retro-blue background color) has its CSS width and height set to auto, so it grows to fit the remaining content.

REAL WORLD EXAMPLES

- Robert Sedovšek: CSS EXCLUSIONS
- Adobe Explores the Future of Responsive Digital Layout with National Geographic Content

After trying the demo and checking out the links above, I'm sure you'll see the opportunities for using shape-inside with regions in your next design. If you have some thoughts on this topic, don't hesitate to comment. Please keep in mind that these features are under development, and you might run into bugs. If you do, you should report them on WebKit's Bugzilla for Safari or Chromium's issue tracker for Chrome. Thanks for reading!

This entry was posted in Regions, Shapes, Web Platform Features

wpt

Zoltan Horvath

Adobe Web Platform

A Taste of FruitJS

by Andrew Husbeck

by Andrew Hushbeck

"FruitJS (pronounced Fruit Juice) is a new Node.js utility."

This article was originally published on Flippin' Awesome on September 16, 2013.

FruitJS (pronounced Fruit Juice) is a new Node.js utility for writing your technical documentation in Markdown and compiling to an easy to use HTML site. You can install and use FruitJS through npm and the command line utility, or by grabbing the source yourself at https://github.com/ktsashes/fruitjs. The project was built with the goal of making it simple and quick for both technical and non-technical people to get beautiful and useful documentation up quickly. In addition to this, it allows documentation to be on Github (and rendered nicely there) as well as on its own site, which will ensure it is easy to find.

This article will walk you through how to use FruitJS and a little about how it was built.

GETTING STARTED

Compiling your documentation is meant to be as dead simple as possible. Once you've written your documentation, simply do the following:

1. If FruitJS is not installed, you'll want to run `npm install -g fruitjs` via the command line.
2. You'll want to make a small JSON file describing what pages you'd like to include, and in what order. Save that in the directory alongside your Markdown. It will look something like this:

```
{
    "pages": ["page1.md", "page2.md"]
}
```

3. Then run the script via the command line using `fruitjs manifest.json`

Your site should be rendered into a subdirectory within the current directory named "output." There are flags and options available in the command line to

change where FruitJS outputs or to render everything on a single page. You can also specify images, LESS, CSS and JavaScript files to include on and customize your pages. You can check out the full documentation (rendered with the script of course) on http://ktsashes.github.io/FruitJS/.

MAKING FRUITJS

When building a Node app like this, basic building blocks are the most important things. The first thing to do was research what modules existed that I could take advantage of. First, and most obviously, was finding a good module for converting Markdown to HTML. I could have ported the Markdown compiler myself, but standing on others shoulders is much easier.

Depending on 3rd Party Libaries

The code was originally written using the Node module "markdown," but because of a poor middle representation, I switched to using marked. I needed a parser that was flexible enough to handle Github flavored Markdown, but I also needed an intermediate representation that I could run through in JavaScript. Marked offered both of these, and was pretty simple to use.

One interesting thing I found was that the library had some pull requests for features I needed to work properly. It's tough when your module isn't quite what you need, and I found I really had two choices. The first was to try and find a different module to use that had all the bells and whistles (or, failing that, write one) or to make the modifications myself. I actually opted for making the modifications myself and including the source in my package. The npm package format has an option to specify you are controlling the versioning of a particular dependency manually, and I ended up making another change or two to help myself out in the long run. A little bit more overhead, but a customized tool works a lot better than none at all.

After the major module, most of the other things I needed were pretty straightforward. Because file actions are asynchronous, and callbacks and streams aren't much fun, I opted to use RSVP to add promises to the system. I used Underscore for templating, and LESS for CSS support. Optimist is good for command line arguments, and mkdirp is a nice utility for writing files, so those got thrown in as well.

Adding Extensibility

When writing the rest of the module, I had two major goals. The first was to make it as easy as possible to go from Markdown to HTML. One simple command should be enough. My second major goal was to make it easy to make your docs your own. So, right from the start, I've been working on integrating a template system with extensibility so that if you don't like the style, you can pick another, or make your own. With all that in mind, I came up with a one line command, a manifest file, and the ability to add images, CSS, and JavaScript to your HTML.

MORE FRUITJS

If you take a look at the docs, there are a few things you might notice. Things are a little sparse at the moment. There are no templates other than the default, your manifest file gets long quickly, and there isn't really support if you want to have a custom menu. All of the support for this is planned, but not quite finished yet.

My goal is to allow you to get the documentation you want by simply going to the command line and saying `fruitjs`. Templates, assets, custom menus, and making the manifest optional are all planned on the way for the 1.0 release (currently we're at 0.6.1). So look forward to more coming soon, and if you have ideas you'd like to see, feel free to add them to the Github, or hack away yourself.

Andrew Hushbeck

Web Developer

HIS BLOG | SHARE | TWITTER | GITHUB

The Angular Way

by Nicolas Bevacqua

by Nicolas Bevacqua

"I can hardly imagine doing front-end web app development without some kind of data binding framework."

This article was originally published on Flippin' Awesome on September 3, 2013. You can read it here.

flippin' awesome!

For the past few months I've been sailing around the world of Angular. Today I can hardly imagine myself doing day to day coding on a large front-end web application without some kind of data binding framework, such as Angular.js, Backbone.js, and friends, and I can't believe I've done so in the past.

I might however be a bit biased, considering the application I'm working on is a PhotoShop-esque editor in the browser, which often presents the same data in radically different ways. For example:

- Layers are presented graphically, taking up large portions of the screen. They are also listed in a panel where you can delete them.
- When you select a layer it gets the typical dashed line around its edges, and it also gets highlighted in the list view.
- Similarly, properties like the dimensions of a layer show up both in a panel and define their size upon the canvas.
- The panels I've mentioned can be dragged around, collapsed, and closed.

This kind of interaction, data binding, and view synchronization would be easily be a maintenance nightmare if it wasn't for a framework such as Angular. Being able to update a model in one place, and have Angular update all relevant views almost feels like cheating. Adding, removing, or moving a layer is just a matter of changing an object.layer.x += 10 and we're done. There's no need to invalidate the view by hand, or to manually update each instance of the layer in the DOM, or to even interact with the DOM, for that matter.

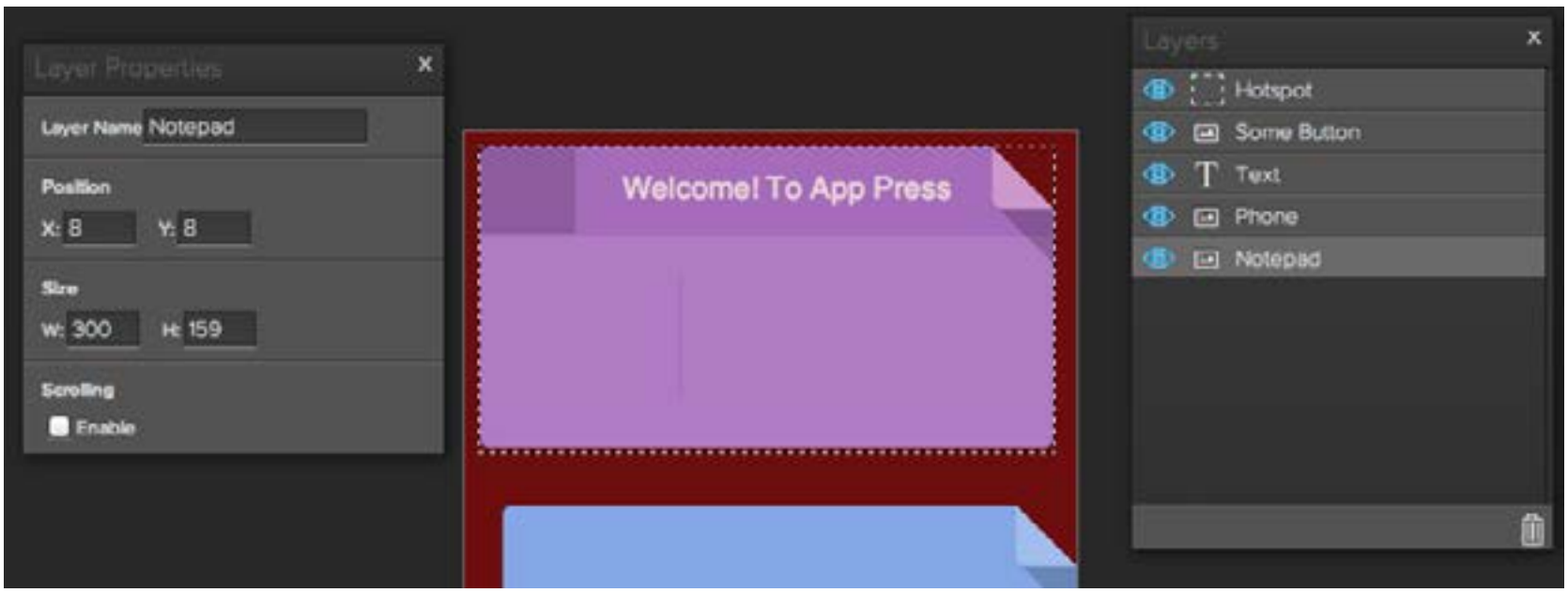
Angular enabled us to go places we wouldn't ever have dreamt of, such as setting up a bunch of keyboard shortcuts that are enabled based on the current context of the application. For example, text editing shortcuts, such as ctrl/cmd + B to toggle bold text, are just enabled when we're editing a text layer.

Similarly, we tacked a description onto these shortcuts (which are registered through a service we created), and we are then able to show a list of shortcuts, along with their descriptions, in a handy cheat sheet. Furthermore, we wrote a directive which enables us to bind individual DOM elements with their keyboard shortcut counterparts, showing a tooltip when you hover over them for a little while, letting you know a keyboard shortcut is available, too.

BECOMING MORE ANGULAR

Honestly, it's as if we weren't writing a web application anymore. The web is just the medium. As we improve our understanding of Angular, the code gets more modular, more self-contained, and yet more inter-connected. It is simply becoming more Angular.

And by Angular I mean the highly interactive rich application development philosophy behind Angular.js, the same one that's enabled us to develop a piece of software that I wouldn't have thought possible a while back.



We were even able to develop a full-fledged history panel that updates the DOM to the currently selected point in history, and it performs well, too! Seeing the data binding capabilities of Angular update every small detail in your view work flawlessly as you go back and forth in the history panel is inspiring, to say the least.

IT WASN'T ALWAYS THIS EASY

…the code-base used to be an uncontrollable mess.

Indeed, in the last few weeks we've been updating and re-writing the overall architecture of our front-end. Before we took up this re-write, looking to update Angular to edge, all the way from 0.10.6. That's a pretty long way to go, if you look at the change log.

Going through this refactoring, we went from doing Angular the wrong way, to doing Angular the Angular way.

The wrong way, in our case, encompassed quite a few issues we had to work through before getting to the lovable state our code-base is in at the moment. In this article, I'll discuss some of those issues and how we overcame them.

CONTROLLERS DECLARED ON THE GLOBAL SCOPE

This one is, sadly, pretty common amongst folks who've been using Angular since the early days. If you're familiar with Angular, you might be familiar with this pattern, too.

```
// winds up on window.LoginCtrl ...
var LoginCtrl = function ($scope, dep1, dep2) {
    // scope defaults
};

LoginCtrl.prototype.resetPassword = function () {
    // reset password button click handler
};

// more on this one later
LoginCtrl.$inject = ['$scope', dep1', 'dep2'];
```

That kind of file isn't wrapped in a closure, either, meaning anything declared on the root scope goes to the global `window` object – yuck. The Angular way to do that is to use the module API they provide. But, as you can see, even in the documentation the recommended setup is still outdated and suggests you use the global scope mercilessly:

"Do this, and wonderful things will happen to you!"

```
// A Controller for your app
var XmplController = function($scope, greeter, user) {
  $scope.greeting = greeter.greet(user.name);
}
```

– the Angular.js documentation

Modules allow us to rewrite controllers in the following way:

```
angular.module('myApp').controller('loginCtrl', [
    '$scope', 'dep1', 'dep2',
    function ($scope, dep1, dep2) {
        'use strict';

        // scope defaults

        $scope.resetPassword = function () {
            // reset password button click handler
        };
    }
]);
```

The beauty I find in the way Angular approaches controllers is that you you need the controller `function` anyways, because that's used to inject the dependencies required by the controller, and it provides a new scope, saving us from the need to wrap all of our script files in self-invoking function expressions like `(function() {})()`.

DEPENDENCY $INJECTION

You might've noticed that in the earliest example, dependencies are injected using `$inject`. Most of the module API, on the other hand, allows you to pass either a `function` or an `Array` containing the list of dependencies, followed by the `function` that depends on those. This is the one thing that I don't like in Angular, and it's probably the documentation's fault. Most of the examples in the documentation are treated as if you don't really need the `Array` form, but the thing is, you do. If you minify your code using a minifier without running ngmin first, you're going to have a bad time.

Since you didn't explicitly declare your dependencies using the `Array` form `['$scope',...]`, your clean-looking function arguments are going to get minified to something like `b,c,d,e,` effectively killing Angular's dependency injection capabilities. I consider this to be a gross mistake in the way they built the framework, for a similar reason to why I strongly dislike Require.js and their troubling AMD modules.

If it's not going to work in production, what good is it for?
My fundamental problem with this kind of behavior is that they have code in their framework that is dead as soon as you go in production. That's fine for utilities like `console` and error reporting, which are useful during development, but it doesn't make any sense in syntactic sugar that just works during development.

These things infuriate me, but enough ranting.

CUTTING DOWN ON THE JQUERY PROLIFERATION

Going in, the application was "kind of Angular", in that it was just wrapped in Angular, but most of the DOM interaction happened through jQuery, rendering Angular pretty much moot. However, if I were to write an Angular.js application from scratch today, I wouldn't include jQuery right away, instead forcing myself to use `angular.element` instead.

The `angular.element` API wraps jQuery if it's present, and it alternatively provides the Angular team's implementation of jQuery's API, called jqLite. It's not that jQuery is evil, or that we need yet another implementation that somewhat reflects their API. It's just that using jQuery isn't very Angular.

Lets look at a concrete, and dumb, example. This uses jQuery to do class manipulation on the element where the controller has been declared.

```
div.foo(ng-controller='fooCtrl')
```

```
angular.module('foo').controller('fooCtrl', function ($scope) {
    $('.foo').addClass('foo-init');

    $scope.$watch('something', function () {
        $('.foo').toggleClass('foo-something-else');
    });
});
```

However, we could be using Angular the way we're supposed to, instead.

```
angular.module('foo').controller('fooCtrl', function ($scope, $element) {
    $element.addClass('foo-init');

    $scope.$watch('something', function () {
        $element.toggleClass('foo-something-else');
    });
});
```

The bottom line is: you shouldn't be manipulating the DOM (changing attributes, adding event listeners) directly or through jQuery. You should be using directives instead (p.s. that's an excellent article, go read it).

If you're still jQuery-lized, there's lots of articles you could read, such as this migration guide and my article on critically thinking about whether to use jQuery.

I won't sit here and claim we've managed to remove jQuery altogether. We have other, more important goals in place, such as releasing the product. It was still worthwhile to remove as much jQuery spam as possible. Doing so simplified every controller we went through. We created directives that manipulate the DOM, and use `angular.element`, even if it just maps to jQuery today.

We have a dependency on jQuery UI, which I'm not pleased about. We're clearly not using it just for the sake of dialogs, we have directives for that. But dragging, drag and drop, and, in particular, being able to drag something and drop it in a sorted list, is just something that involves a lot of work if you're not using jQuery UI; there is no real alternative. The drag and drop problem has been solved, we could (and probably should) be using angular-dragon-drop, which is a really simple drag and drop implementation. Sortable, on the other hand, just depends on jQuery UI.

ORGANIZING A CODE BASE

Another illness we had to deal with during our migration, was that the entire code-base was crammed together in a single large file. This file contained all controllers, services, directives, and code specific to each controller. I made it a point to break it down so that we had exactly one file per component. Right now, we have very few files with more than one component, and most of those happened because a directive used a service to share its data with the outside world.

Although unrelated to Angular, we also modularized our stylesheets. We added a two letter prefix to every class name we use in our code. This prefix represents the component the class belonged to. `pn-`, for example, represents classes that style the panels; `ly-` for layers, and so on. The immediate benefit this provides is that you don't have to think about class names anymore. Because you're namespacing them, it becomes much harder for you to accidentally re-use a class name. Another benefit is reduced nesting, we used to have selectors such as `#layoutEditor div.layer .handle div`, which now might be `.ly-handle-content`. The deepest "nesting" we have now only occurs on overloaded selectors such as `.fo-bar[disabled]:hover` or, at worst, something like `.fo-bar .br-baz`.

A few rules we laid out for this CSS class naming style were:

- Two characters to describe the component name. l`y-`, `dd-`, `dg-`, etc.
- Instead of nesting classes such as `.ly-foo .bar`, we gave `.bar` a more appropriate `.ly-foo-bar` name
- Avoid styling tags directly, use classes for everything. This reduces confusion and improves your ability to use semantic markup.
- Never use an ID in CSS.

After implementing this component-oriented CSS declaration approach, I have a hard time thinking of going back to doing it "the class soup way".

Angular forces you to write good code, but on a deeper level than that, it forces you to think. A while later, it will either feel like a server-side implementation, or it'll become an unbearable hack-fest that you won't be able to stand on. The choice is up to you.

A PIECE OF HEAVEN

Let's decompose one of the pieces of our application, the layers.

```
div.cv-layer(
    ng-repeat="layer in page.layers | reverse",
    ap-layer,
    ng-mousedown="selectLayer(layer.id)",
    ng-mouseup="selectLayer(layer.id)",
    ng-dblclick="doubleClickLayer(layer)",
    ng-hide="layer.invisible"
)
```

Here, we're using the `cv-layer` class, given that the element is part of the canvas component (the canvas is where our layers are drawn to, not to be confused with an HTML5 canvas). We're then using the `ngRepeat` directive to create one of these elements per layer, in a `for-each` kind of fashion. It is passed through a `reverse` filter we wrote, so that the last layer is visually on top. The `apLayer` directive is tasked with actually rendering the layer, whether it's an image, some text, HTML, or something else. The event directives (`ng-mousedown`, `ng-mouseup`, `ng-dblclick`) simply delegate the event to our layer selection service, which handles it from there. Lastly, `ngHide` probably doesn't really need a lot of explaining.

That's a huge amount of functionality and Angular manages to make it look simple with readable HTML that sort of tells you what's going on. Furthermore, it allows you to separate the different concerns so that you can write concise pieces of code that don't try to do everything at once. In summary, it reduces complexity – making the complex, simple. And the "hard to even fathom", possible.

This article was originally published at http://blog.ponyfoo.com/2013/08/27/the-angular-way

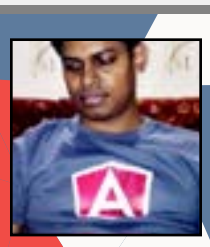
Nicolas Bevacqua

Web Developer

HIS BLOG SHARE TWITTER GITHUB

Migration guide for jQuery developers

by Amit Gharat



by Amit Gharat

"This is dedicated to the jQuery developers who wish to know AngularJS before the dive in."

There are couple of links explaining AngularJS to jQuery developers and my effort also goes to do the same. This is dedicated to all the jQuery developers who wish to know AngularJS before they dive in. I myself worked on jQuery since a very long time and literally its hard to switch to something which is radically different and way cooler than jQuery (sorry, darling!).

DOCUMENT IS READY?

We all know about `$(document).ready()` or `$(window).load()` methods to setup a safe wrapper around the code to make sure that everything runs after the DOM is loaded.

Also, you might have used literal styled javascript with jQuery:

```
var App = {
  run: function() {
    var bootstrapper = 'You have successfully bootstrapped jQuery';
    $('div.alert').html(bootstrapper);
  }
};

// Finally I'll call the main method
$(document).ready(function() {
  App.run();
});
```

Do not panic! Angular itself provides the manual bootstrapping the same way. Angular advocates modularity so that every piece of an application should have its own module and there is a main module that holds rest of the modules together. The subsequent code defines the main module (similar to jQuery example above) and run() block is the built-in (sort of) main method in Angular which runs once the application is bootstrapped.

```
// Define a module, App
var App = angular.module('App', []);

// Built-in run method
App.run(function($rootScope) {
  $rootScope.bootstrapper = 'You have successfully bootstrapped AngularJS';
});

// Finally bootstrap angular manually
angular.element(document).ready(function() {
  angular.bootstrap(document, ['App']);

  // Target an element instead
  // angular.bootstrap(jQuery('body'), ['App']);
});
```

The one thing to notice here is `$rootScope` – which is a global variable you can bind properties to that can be injected in other modules for further use.

Finally we'll use angular's templating syntax `{{ }}` to declaratively place data into the markup instead of grabbing the element explicitly like jQuery. Can you see?

```
<div class="alert alert-success">{{bootstrapper}}</div>
```

In case you are not a fan (like me) of manual bootstrapping, you can make it automatic as well by setting ng-app directive on any element such as html, body etc. Angular looks for an element having `ng-app` on it and uses it as a target during bootstrapping. Now you do not need `angular.element(document).ready()` block as defined above.

```
<html ng-app="App">
```

XHR

AJAX is the second most important thing people use jQuery for. jQuery provides different methods to fire up an XHR call.

$.get
$.post
$.getJSON
$.load
$.ajax

In angular, there is `$http` service which is a wrapper around the browser's XMLHttpRequest object that returns promises `.then()`, `.success()`, and `.error()` based on the request was successful or failed.

$http.get
$http.head
$http.post
$http.put
$http.delete
$http.jsonp

The only difference is that you have to manually inject it into the module definition. Suppose I want it inside the run() block we'd seen before:

```
App.run(function($rootScope, $http) {
  $http.get('foobar.php').success(function(data, status) {
    // I succedded
  }).error(function(data, status) {
    // I failed
  });
});
```

DO NOT TOUCH THE DOM

jQuery was made to do so, Angular is not!

In AngularJS, try to avoid touching DOM implicitly as much as possible so that you can fully leverage the 2-way data binding between your data and the DOM. We are going on journey to build a process order page using following directives.

ng-model

In jQuery, `.val()` is a setter/getter method to interact with form controls which was quite amazing but it gets worst when you want to show the value elsewhere instantly, means we had to bind keypress events on the input control that will update the DOM as expected. But its too much effort as well as code for such a small task.

Angular provides a neat directive to rule them all. Always use `ng-model` in order to read values out of form controls. You do not have to manually select an element to set/get its value – Just play with plain old javascript object/array.

```
<input type="text" ng-model="name">

App.run(function($rootScope) {
  // setting the default value for the input
  $rootScope.name = 'AngularJS';

  // this will return the existing value of the input
  // console.log($rootScope.name);
});
```

ng-options

Use to lay out options based on model for select element. Know more about it.

```
App.run(function($rootScope) {
  $rootScope.cities = [
    {id: 'NM', name: 'Navi Mumbai'},
    {id: 'PN', name: 'Pune'}
  ];
});

<select ng-model="city" ng-options="c.name for c in cities"></select>
```

ng-class

Toggle CSS classes based on expression. This is a very common thing in jQuery:

```
if (same) {
  $('small').removeClass('strike');
} else {
  $('small').addClass('strike');
}
```

In Angular, you can use ng-class directive to apply CSS classes conditionally. If `same` is boolean true then apply `strike` class. Otherwise remove it.

```
<small ng-class="{'strike': !same}">same as billing</small>
```

ng-disabled

Similar to ng-class, you can disable/enable form controls using ng-disabled directive. In jQuery,

```
$('[ng-model="shipping_name"]').prop('disabled', same);
In angular,

<input ng-disabled="same"
       type="text"
       ng-model="shipping_name"
       placeholder="Full Name">
```

ng-change

You may want to reflect your billing address as shipping address instantly.
In jQuery,

```
<input onkeypress="$('[ng-model="shipping_name"]').val($('[ng-model="billing_
name"]').val());"
       type="text"
       ng-model="billing_name">
```

In Angular,

```
<input ng-change="reflect()" type="text" ng-model="billing_name">

$rootScope.reflect = function() {
  $rootScope.shipping_name = $rootScope.billing_name;
  $rootScope.shipping_address = $rootScope.billing_address;
  $rootScope.shipping_city = $rootScope.billing_city;
};
```

ng-click

In jQuery,

```
$('div.btn-primary').click(function() {
  processed = true;
});
```

In Angular,

```
<button class='btn btn-primary' ng-click="processed = true">Process Order</
button>
```

ng-show/ng-hide

This is again very common thing in jQuery to show/hide elements based on conditions.

In jQuery,

```
if (processed) {
  $('div.alert').show();
} else {
  $('div.alert').hide();
}
```

In Angular,

```
<div class="alert alert-success" ng-hide='!processed'>
  <b>Well done!</b> We've successfully processed the order.
</div>
```

Here is the Final Demo of our Process Order form using all above directives.

LOADING PARTIALS

Many of us often create a separate header and footer partials to be injected into many pages instead of repeating all over again.
In jQuery,

```
<body>
    <div id='header'></div>
    <script type="text/javascript">
      $('#header').load('header.html');
    </script>

    <!-- Body -->

    <div id='footer'></div>
    <script type="text/javascript">
      $('#footer').load('footer.html');
    </script>
</body>
```

In Angular, ng-include works similar to jquery.load but also allows to compile and include an external HTML fragment. Do not forget to wrap your html fragment in single quotes as I've already wasted an hour to figure that out :-(

```
<body>
    <div ng-include="'header.html'"></div>

    <!-- Body -->

    <div ng-include="'footer.html'"></div>
</body>
```

HISTORY - PAGE NAVIGATION

Take an example of a simple addressbook application that contains 3 links shown below and `div.abPanel` is the place where you would wish to load the appropriate template.

```
<div class='cell abLinks'>
    <a href='javascript:void(0);' id='addNewContact'>Add New Contact</a><br />
    <a href='javascript:void(0);' id='listContacts'>List all Contact</a><br />
    <a href='javascript:void(0);' id='searchContacts'>Search any Contact</a><br
/>
</div>
<div class='cell abPanel'>Loading... Please Wait.</div>
```

In jQuery, you would probably do this.

```
$('#addNewContact').click(function () {
  $('div.abPanel').load('add_new_contact.html', function () {
    // do all DOM manipulations or event bindings here
  });
});
```

In Angular, we'll use awesome `$routeProvider` service without writing much boilerplate code. This simply loads a template based on hash.

```
App.config(function ($routeProvider) {
  $routeProvider
    .when('/add', { templateUrl: 'partials/add.html' })
    .when('/list', { templateUrl: 'partials/list.html' })
    .when('/search', { templateUrl: 'partials/search.html' })
    .otherwise({
      redirectTo: '/add'
    });
});
```

And finally change above links a little bit to:

```
<div class='cell abLinks'>
    <a href='#/add'>Add New Contact</a><br />
    <a href='#/list'>List all Contact</a><br />
    <a href='#/search'>Search any Contact</a><br />
</div>
<div ng-view>Loading... Please Wait.</div>
```

The one thing to notice here is ng-view directive that lets your render the template of the current route automagically.

ANIMATION

The .animate() method allows us to create animation effects in jQuery. In order to fade out the process order page while hiding, we can write something like this:

```
$('div.row').animate({opacity: 0}, 1000);
```

In angular(version 1.1.4+), ng-animate allows us to control the animation either using CSS or JS. This directive can gel up with other directives like ng-show/ng-hide, ng-view, ng-include, ng-switch, etc.

We'll simply add it to our `div.alert` which takes `processed` CSS class as an option.

```
<div class="alert alert-success" ng-show='processed' ng-animate="'processed'">
  <b>Well done!</b> We've successfully processed the order.
</div>
```

Then we'll define necessary CSS classes. What happens here is when `div.alert` is about to be shown (means processed becomes boolean true), both processed-show and processed-show-active classes will be applied which will help fade in the element for 0.5 seconds and later be removed. And display property sets to block.

```
.processed-show {
  -webkit-transition:all linear 0.5s;
  -moz-transition:all linear 0.5s;
  -ms-transition:all linear 0.5s;
  -o-transition:all linear 0.5s;
  transition:all linear 0.5s;
  opacity:0;
}
.processed-show-active {
  opacity:1;
}
```

We can use similar css while hiding the element but opacity will change from 100% to 0%.

```
.processed-hide {
  -webkit-transition:all linear 0.5s;
  -moz-transition:all linear 0.5s;
  -ms-transition:all linear 0.5s;
  -o-transition:all linear 0.5s;
  transition:all linear 0.5s;
  opacity:1;
}
.processed-hide-active {
  opacity:0;
}
```

Check out the demo. There are more in-depth articles on yearofmoo.com and nganimate.org.

JQUERY PLUGINS

We can not live without it. A few weeks ago, I'd written an article on how to use jQuery plugin the angular way.

CUSTOM DIRECTIVES

AngularJS really shines where it lets you write custom directives also. There are hell of a lot of articles to get you started.

WRAP UPS

I'd seen many of my colleagues struggled to get going with AngularJS and hence I'd decided to write this post. Time has come to expand the horizon. All the best.

Amit Gharat

Web Developer

HIS BLOG SHARE TWITTER GITHUB

INTERVIEW

Jonathan Snook

by Maile Valentine
photos Dan Rubin

APPLINESS: Hello Jonathan, thank you for joining us! Could you introduce yourselves to our readers?

Hi! I'm Jonathan Snook. Most people call me Snook. A few people call me Snookums. Those that call me Snooki get a stern look. I call myself a web developer but these days, I'm a product manager at Shopify.

Can you tell us about your role with Shopify?

As the Shopify Core Product Manager, I define the feature roadmap of the store administration tool that store owners use to manage their stores. I also ensure that other projects integrate well with what we're doing and that we're serving the mission of Shopify to *Make Commerce Better*.

What inspired you to move freelance work to working as a Product Manager for a large company?

I really enjoyed the freelance life. I had the freedom of working from home, working with great clients around the world, and having the flexibility to work around my kids' schedules. Unfortunately, the financial instability of freelance coinciding with some personal issues led me to seek full-time employment.

That actually led me to Yahoo! first where I worked with the Communication Design team to undertake the large task of redesigning Yahoo! Mail, Messenger, Calendar and more. I spent two great years there working with great people who were focused on building a great product. It was a wonderful learning experience and was the first time that I was able to stick with a product over a longer period of time. It also led me to write a book called Scalable and Modular Architecture for CSS (or SMACSS, for short). After two years, I found myself no longer excited about the work and decided to set my focus elsewhere.

I ran into Daniel Weinand, the Chief Design Officer at Shopify, at a conference. Shopify is based in Ottawa, where I also happen to live, and I've been following their growth for a number of years. After some discussion, the timing was right for me to join the team. I started on the design team and then, when an official product team sprang up, I switched roles to become a Product Manager.

While I'm doing less hands-on design and development these days, I'm really enjoying the role of being a product manager. It has been a great learning experience and has allowed me to stretch myself in new areas. It has been nice to have that personal growth again.

JONATHAN SNOOK

EXPERIENCE

You are fluent in a variety of technologies from client-side, server-side and you have a knack for the design side. How have you managed to acquire this blend of skills?

They say it takes 10,000 hours to master something. I can only imagine how many hours I've spent building web sites, although I'm still reluctant to consider myself a master.

One of the best ways to build a breadth of knowledge in web development is to work for an agency. I found myself thrown into the deep end on every project having to figure out the client environment and how to develop something that could work. It seemed like every client had something different. One project was ASP, the next was ColdFusion, the next was

PHP. It was fun to play with whatever technology I could get my hands on.

At the end of the day, users needed to use the products I was building. Therefore, I wanted to make sure that the apps I built were functional and usable. I took whatever design knowledge and content strategy knowledge I learned from co-workers and applied it what I was building. Over the years, I've had the privilege of working with some very talented people that have influenced my work considerably.

Switching to freelancing allowed me to flex my design muscle more than I could working in an agency. This melding of skills has proven very beneficial over the years.

What sort of opportunities, or freedom, has this unique set of skills offered you throughout your career?

Just doing good work is one thing but opportunities are more likely to present themselves when you stick your neck out. I made an effort to experiment with my blog, to write about what I learned, to write for other magazines, and then to eventually turn that into speaking at conferences and meetups.

I attribute my success to being able to publicize my experience. It connected me with people that I knew (and who knew of me) and they were willing to give me a chance even when the situation was a little risky—namely, working remotely—because I had already demonstrated the knowledge and experience.

Now that I've shifted into Product Management, the breadth of knowledge has been extremely helpful. I have a really good sense of what is required from both design and development.

JONATHAN SNOOK

YOUR PROJECTS

Can you tell us a bit about SMACSS?

SMACSS stands for Scalable and Modular Architecture for CSS. It's an approach for writing CSS. It's not a library or a framework. It's a collection of concepts that I have found to work well when working on long-term projects with larger teams.

How does SMACSS help smaller sites? Bigger sites?

The longer you work on a project, the larger the project, the larger the team, the more benefit you'll get from using an approach like SMACSS. That's not to say that it's not appropriate for smaller sites. I could've used the concepts on my own blog—a simple single layout design—and they would've worked just fine. But the design of my blog has gone mostly unchanged in 4 years. That code could be absolutely atrocious and it really doesn't matter because I never have to go back and change it ever again.

When you have ever-evolving projects with dozens upon dozens of layouts,

and widgets, and components that need to be added to, removed, shifted, or modified, then you need a system that can accommodate that.

That's where SMACSS comes in. It advocates three key things: categorizing your CSS, modularizing your CSS, and use a naming convention. Everything else stems from those 3 things.

Are there any big changes coming to SMACSS?

No big plans in the works. I think the concepts are just as applicable now as they were 2 years ago when I first released the book.

If I were to do another edition of the book, a few things would be tweaked. Some thoughts on how or what to modularize and how to name things effectively have solidified over the last couple years that I think would be a great addition to the book.

Do you have any other side projects in the works these days?

Not right now. Every now and then I work on a small project here and there. Phmr.al was the most recent one from a few months ago. It was a fun little weekend project. I have a half dozen project ideas that I'd love to work on if I had the time.

Working at Shopify has filled up most of my time these days making it more difficult to work on side projects for longer than a few hours here or there.

You have written and co-authored some popular books - Scalable and Modular Architecture for CSS, The Art and Science of CSS and Accelerated DOM Scripting with Ajax, APIs, and Libraries (Expert's Voice). What inspired you to write these books? How was the overall experience? Do you think you'll write any more?

For The Art and Science of CSS, it was a single chapter contribution and my first foray into print. It's a great feeling to be able to walk into a bookstore and see your face on the shelf. For Accelerated DOM Scripting, the effort was much larger. Too large, in fact. I had to get some co-authors to help out and very happy that Stuart Langridge, Dan Webb, and Aaron Gustafson agreed to each write a chapter.

I was inspired to write the book as a response to many of the JavaScript libraries that were gaining (and still gaining) in popularity. I wanted to provide some lower level understanding. I still believe that many people are too quick to pull a library into their project instead of relying on plain ol' JavaScript. That's not to say that I'm against libraries. I just think their inclusion on a project should be well considered.

After Accelerated DOM Scripting, and a couple failed book writing efforts afterwards, I vowed to never write another book. Writing is a painful

process, seemingly for everyone involved. However, after formulating my thoughts on CSS architecture, I really felt the need to write it down. Writing allows a concept to be explored and critiqued and picked apart to see if it really stands up to scrutiny. Writing SMACSS was a much different beast than writing a book on JavaScript. It still took a long time (and I still became frustrated along the way) but the effort didn't feel quite as painful as before. I'm happy with the way things turned out.

Will I write another book? That remains to be seen. Right now, nothing has inspired me enough to go through the months of pain that inevitably come from writing a book. (That also means that even if inspiration struck me tomorrow, it'd still be at least half a year before anything saw the light of day. Ah, the joys of writing!)

JONATHAN SNOOK

DEVELOPMENT

What are some of your favorite development tools in your arsenal? Design tools?

On the dev side, I keep things pretty simple. I use Vim (and MacVim) as my text editor of choice. I use Git now as my source control, although I still have a couple personal projects on Subversion. For SQL development, which I don't do too often anymore, I still use Navicat. I've used it for years and haven't had much incentive to move off it. I use VMWare for my cross-platform testing. On the design side, I love Adobe Fireworks. Sadly, I'll need to find something new someday.

I feel like my toolset isn't very exciting these days. I've moved increasingly to command line tools that have more portability. It's handy being able to open a shell and being able to do what you need regardless of whether it's local or on a server somewhere.

Do you prefer working on small or large teams? What are the pros and cons of each?

When any company grows, I think the secret isn't to grow the team, but to have multiple small teams work on very distinct problems. I saw this at Yahoo! and I see this at Shopify. Sure, the overall team might be 5, 10, 20, or 30 people in size but everybody is still working in a small team that is manageable. Divide and conquer!

What I have learned is that the more teams you have, the better you need to be at coordinating efforts across multiple teams to ensure everything comes together properly. There's a lot of communication that needs to happen. And as someone who doesn't like meetings, I'm always looking for ways to share better.

Do you have any design patterns, frameworks, etc. you like to use on a regular basis? How do you choose the right tool for the job?

Depends on what type of project I'm working on. At the PHP level, I've settled into CakePHP and sometimes Zend. There are other frameworks out there but no longer have the time to really expand out from what I know. That's a little disheartening to say out loud. I'll have to explore other options at some point.

Choosing the right tool for the job comes from an understanding of the tools. If you have an in-depth knowledge of only one tool (say, a hammer) then, as the saying goes, everything looks like a nail. Get to know the variety of tools that are available, use them, and learn where they shine and don't shine. Through experience, you learn what tools to use when.

Of course, in looking for good tools, there are things to look out for. Is it extensible? Is it well supported? Does it have good documentation? Does it have a good community? Is it worth jumping on something that hasn't even hit 1.0 yet? These are just a few of the questions I think people need to ask themselves before using other people's code.

What is your workflow for developing CSS for a brand new site or app?

SMACSS reflects a bit of that workflow. Everything starts from a design. I can't think of a project that I've started without some level of design having been done elsewhere like Photoshop, Fireworks, or even just sketches.

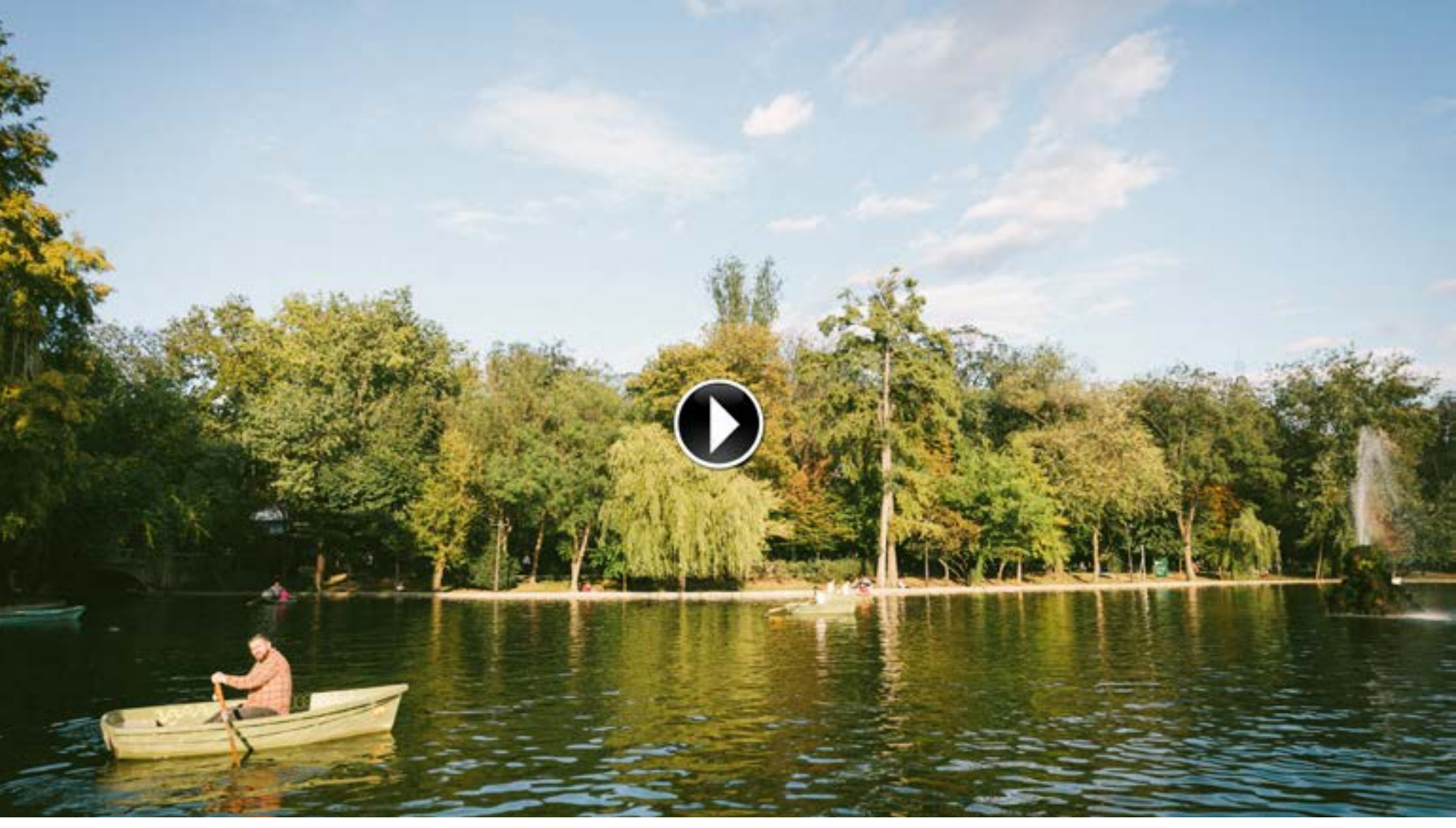
Once I have a sense of what I'm going to code, I start with my base styles. What do I want my elements to look like by default? From there, I start to build out the layout. What are the major content areas? Do I need a grid system? After that, I start looking at what the design patterns are. What is the visual language that needs to be codified? These are the modules that I begin to build out.

How do you manage the various aspects of developing for large sites while working in a team? For example, how can a large team prevent duplication of efforts, develop clean code, provide ongoing maintenance, etc.?

I think the SMACSS approach lends itself really well to developing in large teams. The modularization helps codify reusable design components and makes it more evident when there's duplication. That type of evidence encourages cleaner code. The modularization also helps to have different people working on different components at the same time.

What are the top mistakes you see when looking at other developer's CSS?

Overly qualified selectors (often exacerbated by the use of preprocessors), the use of !important, and no consideration for naming convention. CSS is, in a way, like a language with only global variables. Some people combine selectors as a way of namespacing. That leads to clashes where a class name ends up doing two (or more) different things and the only way to tell what is going on is to go back to the CSS—which might be spread across multiple files.



JONATHAN SNOOK

& FUN

You showed a game called CSS Panic at Smashing Conf 2012 that was very impressive in showing what can be done with just CSS. Have you seen any other examples since then that have impressed you?

None nearly as impressive as that one. CSS Panic is impressively done for so many aspects that come together in one piece. I love showing that off.

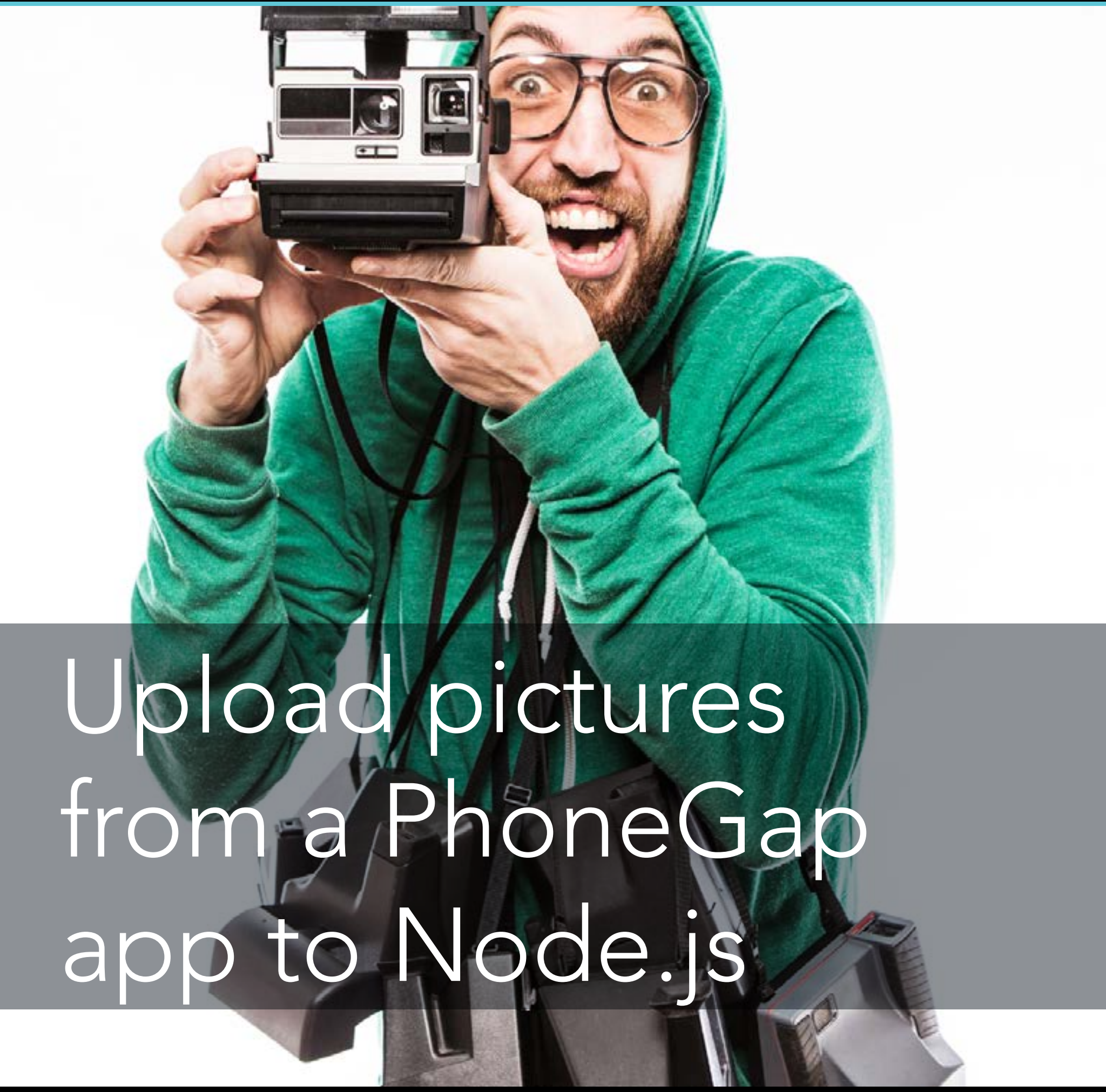
Your archive page of blog posts goes back to 2001 and reads like an historical timeline of the evolution of the web. Do you ever marvel at how much things have change in the time you've been working with the web?

I'm such a cynic in some sense. If I go back to the early posts, I'm talking about client-side and server-side development. It's not much different than now. The technologies have changed a bit. The techniques have evolved. And yet, at the end of the day, we're mostly just building text files. I use Vim—an editor that has been around for decades—to edit a bunch

of HTML, CSS, and JavaScript. It feels like things really haven't changed in the 13 years I've had a blog.

Outside of your work life, what would you like to have more time for in 2013 and beyond?

There's life outside work? I'm a lucky guy. I get to do something that I enjoy day in and day out. I just hope I'm lucky enough to enjoy what I'm doing for years to come. How many people get to say that?



Upload pictures from a PhoneGap app to Node.js

by Christophe Coenraets

by Christophe Coenraets

"A number of people have asked for an end-to-end example showing how to take pictures and upload to a server."

You have seen the demos showing you how to access your camera and take pictures from a PhoneGap application. But these demos often end there, and a number of people have asked me for an end-to-end example showing how to take pictures and upload them to a server.

I created a simple app that I called Picture Feed. It shows you the last pictures uploaded by users, and lets you take pictures that are automatically uploaded to a server for other people to see.

In this sample app, I upload the pictures to a Node.js server, and I keep track of the list of pictures and associated information (if any) in a MongoDB collection. But the same approach would work with other server stacks.

Here is a short video:

SOURCE CODE

The client-side and server-side (Node.js) code is available in this GitHub repository.

In my next post I'll modify this example to demonstrate how to upload pictures to Amazon S3.

This application was created with PhoneGap 3. Don't forget to add the plugins required by the application:

```
phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-device.git
phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-console.git
phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-file.git
phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-file-transfer.git
phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-camera.git
```

To test the application on iOS, make sure you modify the "widget id" value in config.xml and specify the namespace that matches your application provisioning profile.

Christophe Coenraets

Developer Evangelist

PhoneGap 3.0 Stuff you should know

by Holly Schinsky

by Holly Schinsky

"PhoneGap 3.0 was announced and I am summarizing some important things to note."

PhoneGap 3.0 was announced recently and there are some important things to note that I have attempted to summarize below.

PHONEGAP CLI

If you hadn't heard yet, the fastest and easiest way to use PhoneGap 3.0 is with the new Command Line Interface. To install the CLI, follow the specific instructions here.

> *This is a completely different approach than previous versions of PhoneGap where you would download a zip file of the latest version from here for instance (as listed under Archives). With this new CLI approach, once you run* `sudo npm install -g phonegap`*, you will have everything installed on your hard drive to start creating PhoneGap 3.0 projects.*

SO WHAT DOES THE CLI CREATE?

A lot happens behind the scenes when running the commands from the PhoneGap 3.0 CLI and understanding it will save time or possible frustration with project configuration. It's actually really cool and easy and makes the whole development process much faster than prior versions of PhoneGap.

```
$ phonegap create ...
```

Outputs a sample application (www folder) with the following key components:

1. **config.xml** sample for specifying application attributes for PhoneGap Build
2. **index.html** (with included tags for phonegap.js script and CSS etc)
3. **index.css** with basic CSS styles (css folder)
4. **index.js** file with ***deviceReady*** handler (js folder)

TIP: When you create a new project, you should specify the name and project id using the long version of the create command, otherwise you will get a project named HelloWorld with an id of com.phonegap.hello-world. Fine for testing, but not what you'll want for your real apps so I recommend getting used to the longer syntax off the bat. I personally like to explicity state the optional –id and –name arguments so I remember exactly what each parameter is doing. See the docs for more options.

```
$ phonegap create ~/Documents/PhoneGapProjects/PG30Testing --id “org.devgirl.pg30testing” --name “PG30Testing”
```

If you're already familiar with the Cordova CLI, the PhoneGap CLI is actually built on top of it and provides the same functionality plus integration with PhoneGap Build. So with the PhoneGap CLI you can also build your application remotely from the command line without having to have all the native SDKs for each platform installed. See the paragraph titled Build Applications Remotely for more details.

CORE PLUGINS MUST NOW BE ADDED...BUT IT'S EASY

WARNING: The newly created project will **NOT** include access to all of the API's (aka features and plugins) documented in the PhoneGap API docs (geolocation, contacts etc) initially as it did in prior versions. Instead you install only those you intend to use via the CLI so your application's performance is optimal and not bulky with code not being used. For instance, to install the geolocation plugin you would use the following command:

```
$ phonegap local plugin add https://git-wip-us.apache.org/repos/asf/cordova-plugin-geolocation.git
```

Where you're specifying the `phonegap local plugin` add command, followed by the location of the project in git. The list of them all is here.

See the docs here as well for other core plugin locations.

> *Note that the* **plugin.xml** *does all the work for you in the new plugin architecture so you do not have to include the script tag in your* **index.html** *for that specific plugins' .js file anymore, you simply add the plugin with the CLI and it's ready to go. You also do not access it off the window object as before. So if you've added the* **Media** *plugin, you simply say* `var media = new Media(src)` *for instance. You can see the entry in the plugin.xml that refers to the Javascript source etc such as the following:*

```
...
<js-module src="www/Media.js" name="Media">
        <clobbers target="window.Media" />
</js-module>
...
```

I recommend checking out the whole plugin.xml specification for more information. Also, my colleague Ray Camden did a post about using other plugins in 3.0 that can be found here.

WHY ALL THE WWW FOLDERS?

When you add a platform via the **phonegap build**, **install or run** commands, you will see another **www** folder created within each of the platform folders (ie: *yourprojectroot/platforms/ios/www* for iOS and *yourprojectroot/platforms/android/assets/www for android*). DO NOT use this folder directly as it is going to be overwritten every time those commands are run. You should always work from the root **www** folder and use the **merges** folder for changes specific to a platform. See Customize Each Platform in the docs for details on merges.

> *You may wonder why there isn't a* **phonegap.js** *file in your root* **www** *folder. A different* **phonegap.js** *file is actually used per platform and copied into that platform-generated www folder at build and run time, but the included script tag was already put into your root index.html file for you.*

CONFIG FILES GALORE!

It might seem confusing to see the different **config.xml** files if you're perusing through what's created by the CLI commands. There is a sample **config.xml** in your root **www** folder that is used when a project is built remotely by PhoneGap Build (via the phonegap build remote command). However, when you are building locally for a platform you will make platform-specific changes to a config.xml file for each platform in a separate location. For instance, for **android** you will find it under a path like **yourprojectroot/platforms/android/res/xml/config.xml**. For **ios** you will find it under **yourprojectroot/platforms/ios/yourprojectname/config.xml**. This might lead to confusion if you're not paying attention since you will also see the sample one in the platforms **www** folder which was copied down from the root project. The one in the platform-specific directory is the one to update and modify.

> *Note that you no longer have to add* **feature** *tag entries to your* **config.xml** *file for each plugin you add if they are installed with the above CLI approach and are 3.0-compliant. They will automatically be added into the platform-specific* **config.xml** *file at those platform-specific locations noted above.*

Below are samples of the three **config.xml** files noted above to compare. I had added one Media plugin to my project before sharing this sample so you can see that one plugin specified as a feature…

PHONEGAP BUILD SAMPLE CONFIG.XML (IN WWW FOLDERS)

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="org.devgirl.pg30testing2" version="1.0.0" xmlns="http://www.w3.org/
ns/widgets" xmlns:gap="http://phonegap.com/ns/1.0">
    <name>PG30Testing2</name>
    <description>
        Hello World sample application that responds to the deviceready event.
    </description>
    <author email="support@phonegap.com" href="http://phonegap.com">
        PhoneGap Team
    </author>
    <feature name="http://api.phonegap.com/1.0/device" />
    <preference name="permissions" value="none" />
    <preference name="orientation" value="default" />
    <preference name="target-device" value="universal" />
    <preference name="fullscreen" value="true" />
```

```
    <preference name="webviewbounce" value="true" />
    <preference name="prerendered-icon" value="true" />
    <preference name="stay-in-webview" value="false" />
    <preference name="ios-statusbarstyle" value="black-opaque" />
    <preference name="detect-data-types" value="true" />
    <preference name="exit-on-suspend" value="false" />
    <preference name="show-splash-screen-spinner" value="true" />
    <preference name="auto-hide-splash-screen" value="true" />
    <preference name="disable-cursor" value="false" />
    <preference name="android-minSdkVersion" value="7" />
    <preference name="android-installLocation" value="auto" />
    <icon src="icon.png" />
    <icon gap:density="ldpi" gap:platform="android" src="res/icon/android/icon-
36-ldpi.png" />
    <icon gap:density="mdpi" gap:platform="android" src="res/icon/android/icon-
48-mdpi.png" />
    <icon gap:density="hdpi" gap:platform="android" src="res/icon/android/icon-
72-hdpi.png" />
    <icon gap:density="xhdpi" gap:platform="android" src="res/icon/android/
icon-96-xhdpi.png" />
    <icon gap:platform="blackberry" src="res/icon/blackberry/icon-80.png" />
    <icon gap:platform="blackberry" gap:state="hover" src="res/icon/blackberry/
icon-80.png" />
    <icon gap:platform="ios" height="57" src="res/icon/ios/icon-57.png"
width="57" />
    <icon gap:platform="ios" height="72" src="res/icon/ios/icon-72.png"
width="72" />
    <icon gap:platform="ios" height="114" src="res/icon/ios/icon-57-2x.png"
width="114" />
    <icon gap:platform="ios" height="144" src="res/icon/ios/icon-72-2x.png"
width="144" />
    <icon gap:platform="webos" src="res/icon/webos/icon-64.png" />
    <icon gap:platform="winphone" src="res/icon/windows-phone/icon-48.png" />
    <icon gap:platform="winphone" gap:role="background" src="res/icon/windows-
phone/icon-173.png" />
    <gap:splash gap:density="ldpi" gap:platform="android" src="res/screen/
android/screen-ldpi-portrait.png" />
    <gap:splash gap:density="mdpi" gap:platform="android" src="res/screen/
android/screen-mdpi-portrait.png" />
    <gap:splash gap:density="hdpi" gap:platform="android" src="res/screen/
android/screen-hdpi-portrait.png" />
    <gap:splash gap:density="xhdpi" gap:platform="android" src="res/screen/
android/screen-xhdpi-portrait.png" />
    <gap:splash gap:platform="blackberry" src="res/screen/blackberry/
screen-225.png" />
    <gap:splash gap:platform="ios" height="480" src="res/screen/ios/screen-
iphone-portrait.png" width="320" />
    <gap:splash gap:platform="ios" height="960" src="res/screen/ios/screen-
```

```
iphone-portrait-2x.png" width="640" />
    <gap:splash gap:platform="ios" height="1024" src="res/screen/ios/screen-
ipad-portrait.png" width="768" />
    <gap:splash gap:platform="ios" height="768" src="res/screen/ios/screen-
ipad-landscape.png" width="1024" />
    <gap:splash gap:platform="winphone" src="res/screen/windows-phone/screen-
portrait.jpg" />
    <access origin="http://127.0.0.1*" />
</widget>
```

IOS-SPECIFIC CONFIG.XML (UNDER PLATFORMS/IOS/YOURPROJECTNAME/)

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="io.cordova.helloCordova" version="2.0.0" xmlns="http://www.w3.org/
ns/widgets">
    <name>Hello Cordova</name>
    <description>
        A sample Apache Cordova application that responds to the deviceready
event.
    </description>
    <author email="dev@cordova.apache.org" href="http://cordova.io">
        Apache Cordova Team
</author>
<content src="index.html" />
<feature name="LocalStorage">
    <param name="ios-package" value="CDVLocalStorage" />
</feature>
<access origin="http://127.0.0.1*" />
<preference name="KeyboardDisplayRequiresUserAction" value="true" />
<preference name="SuppressesIncrementalRendering" value="false" />
<preference name="UIWebViewBounce" value="true" />
<preference name="TopActivityIndicator" value="gray" />
<preference name="EnableLocation" value="false" />
<preference name="EnableViewportScale" value="false" />
<preference name="AutoHideSplashScreen" value="true" />
<preference name="ShowSplashScreenSpinner" value="true" />
<preference name="MediaPlaybackRequiresUserAction" value="false" />
<preference name="AllowInlineMediaPlayback" value="false" />
<preference name="OpenAllWhitelistURLsInWebView" value="false" />
<preference name="BackupWebStorage" value="cloud" />
<preference name="permissions" value="none" />
<preference name="orientation" value="default" />
<preference name="target-device" value="universal" />
<preference name="fullscreen" value="true" />
<preference name="webviewbounce" value="true" />
```

```
    <preference name="prerendered-icon" value="true" />
    <preference name="stay-in-webview" value="false" />
    <preference name="ios-statusbarstyle" value="black-opaque" />
    <preference name="detect-data-types" value="true" />
    <preference name="exit-on-suspend" value="false" />
    <preference name="show-splash-screen-spinner" value="true" />
    <preference name="auto-hide-splash-screen" value="true" />
    <preference name="disable-cursor" value="false" />
    <preference name="android-minSdkVersion" value="7" />
    <preference name="android-installLocation" value="auto" />
    <feature name="Media">
        <param name="ios-package" value="CDVSound" />
    </feature>
</widget>
```

ANDROID-SPECIFIC CONFIG.XML (UNDER PLATFORMS/ANDROID/RES/)

```
<?xml version='1.0' encoding='utf-8'?>
<widget id="io.cordova.helloCordova" version="2.0.0" xmlns="http://www.w3.org/
ns/widgets">
    <name>Hello Cordova</name>
    <description>
        A sample Apache Cordova application that responds to the deviceready
event.
    </description>
    <author email="dev@cordova.apache.org" href="http://cordova.io">
        Apache Cordova Team
    </author>
    <content src="index.html" />
    <feature name="App">
        <param name="android-package" value="org.apache.cordova.App" />
    </feature>
    <access origin="http://127.0.0.1*" />
    <preference name="useBrowserHistory" value="true" />
    <preference name="exit-on-suspend" value="false" />
    <preference name="permissions" value="none" />
    <preference name="orientation" value="default" />
    <preference name="target-device" value="universal" />
    <preference name="fullscreen" value="true" />
    <preference name="webviewbounce" value="true" />
    <preference name="prerendered-icon" value="true" />
    <preference name="stay-in-webview" value="false" />
    <preference name="ios-statusbarstyle" value="black-opaque" />
    <preference name="detect-data-types" value="true" />
    <preference name="show-splash-screen-spinner" value="true" />
```

```
    <preference name="auto-hide-splash-screen" value="true" />
    <preference name="disable-cursor" value="false" />
    <preference name="android-minSdkVersion" value="7" />
    <preference name="android-installLocation" value="auto" />
    <feature name="Media">
        <param name="android-package" value="org.apache.cordova.media.
AudioHandler" />
    </feature>
</widget>
```

ONE MORE THING... JAVA API CLASSES-PACKAGE NAME CHANGE

If you're using any 3rd party plugins for Android and they haven't yet been updated to the PhoneGap 3.0 spec then you will need to change the path to the following two classes within the Java plugin code for android:

```
import org.apache.cordova.api.CallbackContext;
import org.apache.cordova.api.CordovaPlugin;
```

-to-

```
import org.apache.cordova.CallbackContext;
import org.apache.cordova.CordovaPlugin;
```

MORE INFO...

- PhoneGap 3.0 FAQ
- Ray Camden's PhoneGap 3.0 Released – Things You Should Know
- PhoneGap 3.0 Plugin Spec
- Ray Camden on Working with Plugins in PhoneGap 3.0

Holly Schinsky

Developer Evangelist

HER BLOG SHARE TWITTER GITHUB

Repeat Properties in Webkit

by Andrei Parvu

by Andrei Parvu

"Exploring the round and space values of CSS Masking, Backgrounds and Borders."

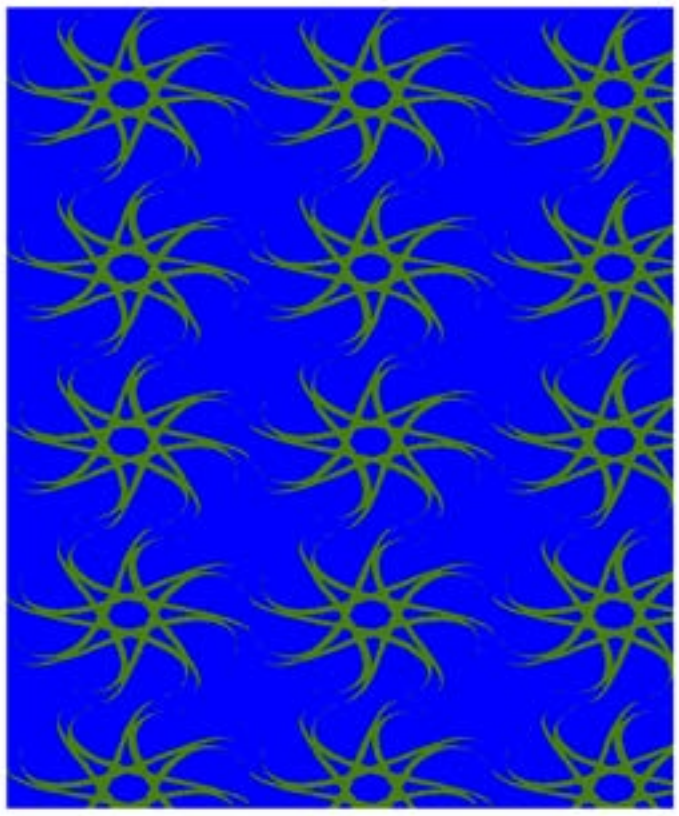
As a part of the CSS Masking and CSS Backgrounds and Borders specifications, the `-webkit-mask-repeat` and `background-repeat` properties can have the values of `round` and `space`.

For example, let's say we have a green div with the following style, which is located on a blue background:

```
div {
width: 250px;
height: 300px;
background-color: green;
-webkit-mask-image: url(star.png);
-webkit-mask-size: 91px 65px;
-webkit-mask-repeat: repeat;
}
```

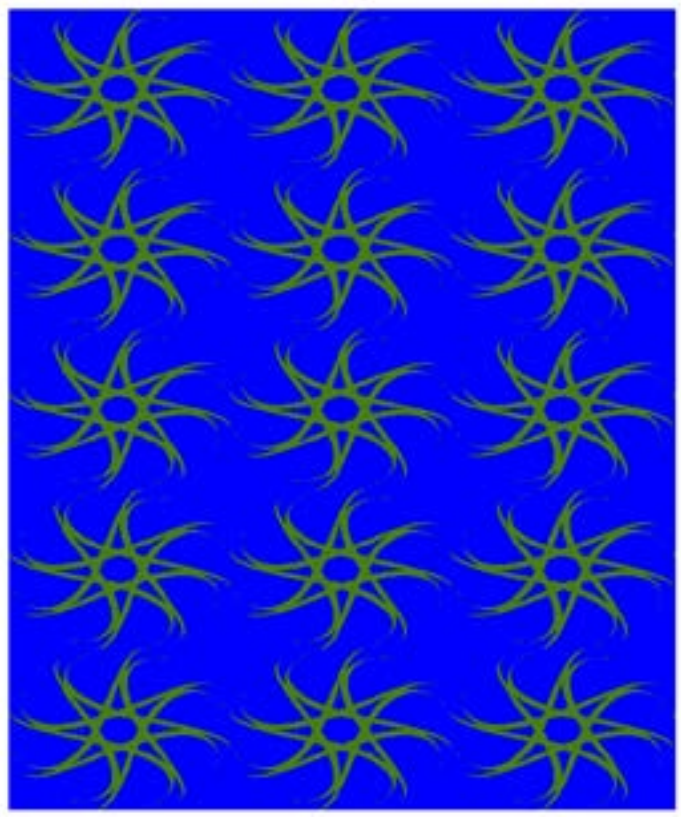
where **star.png** is the following image:

As you can see from figure 1, because the mask image does not fit a whole number of times in the background positioning area (which is determined by the `background-origin` or `-webkit-mask-origin` properties), it gets clipped. To avoid this, we can specify a mask-repeat value of either `round` or `space`.



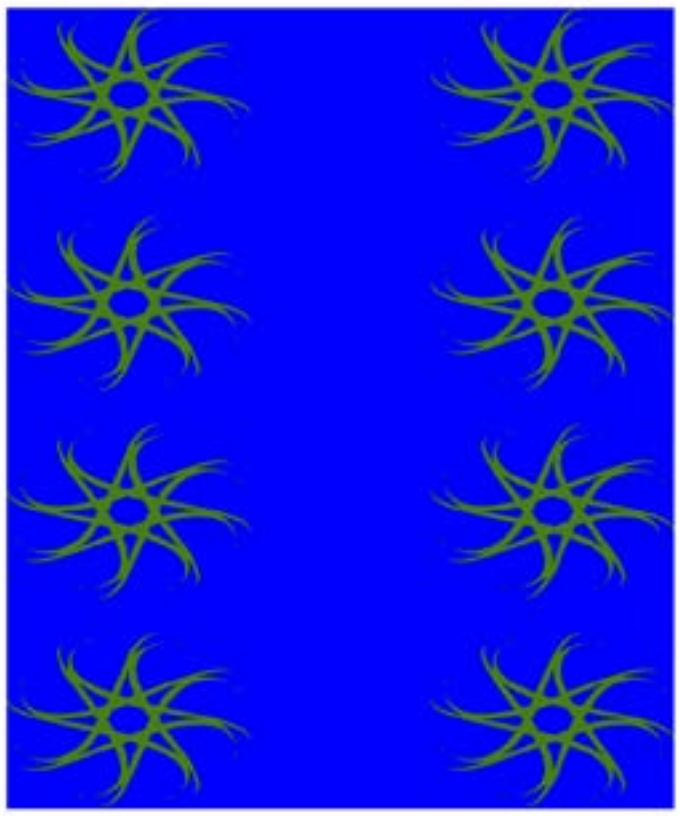
ROUND

When using a value of round, the mask image gets scaled so that it will fit a whole number of times, as shown in figure 2. From the original size of 91px x 65px the image is shrunk to 83px x 60px, thus appearing exactly 3 times on the x-axis and 5 times on the y-axis. Quoting the spec: “In the case of the width (height is analogous): if X ≠ 0 is the width of the image and W is the width of the background positioning area, then the rounded width will be X′ = W / round(W / X) where round() is a function that returns the nearest natural number (integer greater than zero)”.



SPACE

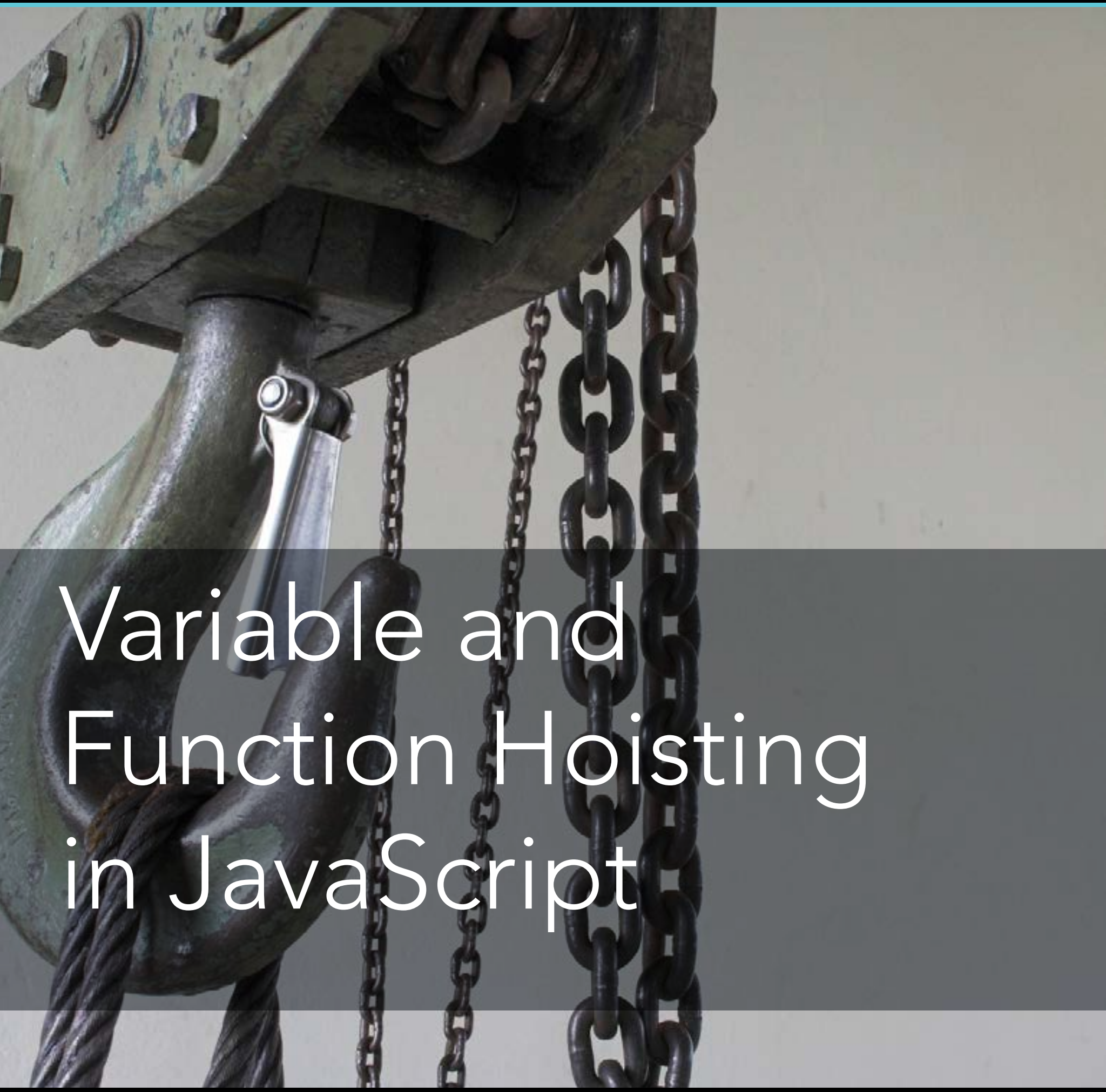
When using a value of space, the mask image is repeated as often as it fits in the background positioning area without being clipped. Then, the images are spaced out to fill the area, in such a way that the first and last images touch the edges of the area. Figure 3 shows how the masks are spaced so that 2 masks will appear on the x-axis and 4 masks on the y-axis.



The repeat and space values also are implemented for the background-repeat property. While writing this blogpost we discovered that when resizing a window containing an element with a background-repeat: space value, the portion between the spaced images is not drawn. We will fix this here: https://bugs.webkit.org/show_bug.cgi?id=120607.

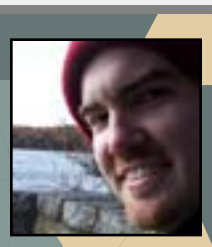
This entry was posted in Web Platform Features.

Andrei Parvu
Adobe Web Platform Team



Variable and Function Hoisting in JavaScript

by Joshua Clanton



by Joshua Clanton

"One of the trickier aspects of JavaScript is that variables and functions are hoisted."

Originally published in A Drip of JavaScript.

One of the trickier aspects of JavaScript for new JavaScript developers is the fact that variables and functions are "hoisted." Rather than being available after their declaration, they might actually be available beforehand. How does that work? Let's take a look at variable hoisting first.

```
// ReferenceError: noSuchVariable is not defined
console.log(noSuchVariable);
```

This is more or less what one would expect. An error is thrown when you try to access the value of a variable that doesn't exist. But what about this case?

```
// Outputs: undefined
console.log(declaredLater);

var declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```

What is going on here? It turns out that JavaScript treats variables which will be declared later on in a function differently than variables that are not declared at all. Basically, the JavaScript interpreter "looks ahead" to find all the variable declarations and "hoists" them to the top of the function. Which means that the example above is equivalent to this:

```
var declaredLater;

// Outputs: undefined
console.log(declaredLater);

declaredLater = "Now it's defined!";

// Outputs: "Now it's defined!"
console.log(declaredLater);
```

One case where this is particularly likely to bite new JavaScript developers is when reusing variable names between an inner and outer scope. For example:

```
var name = “Baggins”;

(function () {
    // Outputs: “Original name was undefined”
    console.log(“Original name was “ + name);

    var name = “Underhill”;

    // Outputs: “New name is Underhill”
    console.log(“New name is “ + name);
})();
```

In cases like this, the developer probably expected name to retain its value from the outer scope until the point that name was declared in the inner scope. But due to hoisting, name is undefined instead.

Because of this behavior JavaScript linters and style guides often recommend putting all variable declarations at the top of the function so that you won’t be caught by surprise.

So that covers variable hoisting, but what about function hoisting? Despite both being called “hoisting,” the behavior is actually quite different. Unlike variables, a function declaration doesn’t just hoist the function’s name. It also hoists the actual function definition.

```
// Outputs: “Yes!”
isItHoisted();

function isItHoisted() {
    console.log(“Yes!”);
}
```

As you can see, the JavaScript interpreter allows you to use the function before the point at which it was declared in the source code. This is useful because it allows you to express your high-level logic at the beginning of your source code rather than the end, communicating your intentions more clearly.

```
travelToMountDoom();
destroyTheRing();

function travelToMountDoom() { /* Traveling */ }
function destroyTheRing() { /* Destruction */ }
```

However, function definition hoisting only occurs for function declarations, not function expressions. For example:

```
// Outputs: “Definition hoisted!”
definitionHoisted();

// TypeError: undefined is not a function
definitionNotHoisted();

function definitionHoisted() {
    console.log(“Definition hoisted!”);
}

var definitionNotHoisted = function () {
    console.log(“Definition not hoisted!”);
};
```

Here we see the interaction of two different types of hoisting. Our variable definitionNotHoisted has its declaration hoisted (thus it is undefined), but not its function definition (thus the TypeError.)

You might be wondering what happens if you use a named function expression.

```
// ReferenceError: funcName is not defined
funcName();

// TypeError: undefined is not a function
varName();

var varName = function funcName() {
    console.log(“Definition not hoisted!”);
};
```

As you can see, the function's name doesn't get hoisted if it is part of a function expression.

And that is how variable and function hoisting works in JavaScript.

Thanks for reading!

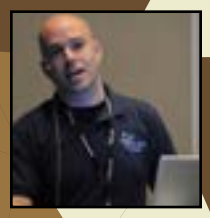
Joshua Clanton

Web Developer

HIS BLOG SHARE TWITTER GITHUB

Responsive Design and Bootstrap 3

by Burke Holland



by Burke Holland

"I don't just want to code, I want to be an artist."

This article was originally published on Flippin' Awesome on September 16, 2013. You can read it here.

flippin' awesome!

This is me and design. I want to be a designer. I want to build beautiful sites. I don't just want to code, I want to be an artist. A creative. I work for a UI company. Everything that I live and breathe is UI.

The sad truth of the matter is that I am not a designer, and no amount of trying or wanting is going to make me one. Just like Kip though, I refuse to give up on my dream. It does mean though, that I need help. I need all the help I can get. Also, just like Kip, even with all the help in the world, I still will never be able to build truly beautiful and original sites. I will never be a cage fighter.

And that's OK. If high school drilled one thing into us, it's that you have to know who you are and fully support that person. That and how to make a beer bong. However, this means that I need help when it comes to CSS, and a lot of it. This is why I love Twitter Bootstrap.

RESPONSIVE DESIGN BASICS

Responsive design is the concept that instead of your web page looking great on a desktop and really tiny on a mobile phone, content is rearranged and kept large so that instead it looks like a website that was written specifically for a mobile device.

Responsive design is generally broken up into three different conceptual areas…

1. Fluid Grids
2. Fluid Media
3. Media Queries

Bootstrap 3 handles all three of these areas for you on some level, providing you with a solid base on which to build your responsive application.

BOOTSTRAP BASICS

Before we get into the guts of Bootstrap 3 and how it wrangles responsive design for you, lets first take a look at the groundwork that Bootstrap has laid for you with it's typography and base CSS resets.

TYPOGRAPHY

In this example, you see the various headings (h1 – h6), and a paragraph tag. Notice that there is no left or right padding or margin on the text at all.

View Demo In New Window

If you toggle bootstrap off, you see how things change. The text is ugly for one, using the default browser style. WebKit based browsers will also add a `-webkit-margin-before` and `-webkit-margin-after` CSS class that adds some padding to the text. Also, the size of the text is changed. If you change the font size on the body to say, 20px, you can see that the browser resizes all the text.

```
body {
  font-size: 20px;
}
```

At its core, Bootstrap includes normalize.css. This is a library that does a "reset" on your CSS. In other words, it provides a baseline for font, margin and padding that assures your simple HTML will look the same in all browsers, as well as fix some basic browser bugs. Note that in most modern browsers, normalize is almost unnecessary. That's encouraging.

MEDIA QUERIES

Let's start by doing a brief examination of media queries. Media queries are one of the 3 core pieces of the responsive design puzzle, but you can't really understand the other two if you don't get this one.

A media query is defined by MDN in this rather verbose but precise description.

> *A media query consists of a media type and at least one expression that limits the style sheets' scope by using media features, such as width, height, and color. Media queries, added in CSS3, let the presentation of content be tailored to a specific range of output devices without having to change the content itself.*

I overly simplify it to just this.

> *Conditionally applying CSS based on the size of the browser window.*

Media queries can be used to load in entire stylesheets…

```
<!-- CSS media query on a link element -->
<link rel="stylesheet" media="(max-width: 800px)" href="example.css" />
```

But they are more commonly found inline in stylesheets loaded by the browser…

```
<!-- CSS media query within a style sheet -->
<style>
  @media tv and (min-width: 700px) and (orientation: landscape) {
    .facet_sidebar {
      display: none;
    }
  }
</style>
```

As you can see from this fancy pants media query for a tv, media queries consist of two parts – a media type and a size rule that can have multiple conditions. Media queries can get unwieldy. Most often, you will not see the media type. Usually responsive design is done based on the size of the screen, not the media type that the device reports itself to be. A device may lie about its identity, but it cannot lie about it's screensize.

Let's take a look at a simple media query in action.

View Demo In New Window

This media query says that when the width of the screen gets below 700 pixels, change the color of the text to "gentleman's pink" – otherwise known as Salmon. This is done by setting a `max-width`, which effectively says "This is the maximum width at which the following CSS rule applies". You can apply any CSS like this, including transitions – where supported; like fading the text out using a transition. Try changing the CSS above to the following and then resize the browser window. The gray box displays the current width of the window. Notice that the CSS is effectively removed when the media query's conditions are not met.

```
h1 {
  -webkit-transition: opacity 1s;
  -o-transition: opacity 1s;
  -mos-transition: opacity 1s;
  transition: opacity 1s;
}

@media (max-width: 400px) {
  h1 {
    opacity: 0;
  }
}
```

We're almost ready to get into Bootstrap itself and talk about how you use it to build gorgeous responsive sites. Before we do though, a word from the legal department of the web.

DISCLAIMER

Bootstrap is not magic. It does not cure cancer. Only Chuck Norris' tears do that. You are not going to drop Bootstrap onto your site and magically have a cross platform application that works on every device and is future proof. In fact, you are going to have to do a fair bit of extra work on top of what Bootstrap gives you out of the box to be fully responsive.

Also, keep in mind that Responsive Design is really just the beginning when it comes to building web experiences on mobile devices. Today's phones and tablets are capable of so much more than just laying out content with a font big enough for you to read it. We are still trying to figure out how to simply display the content for these devices, but they are poised to do so much more if we can leverage all of the power they afford to users above and beyond what a desktop was ever capable of.

Appropriate disclaimers in place, lets dig in.

BOOTSTRAP 3

Before you do ANY responsive design, you need to add a meta tag to your page. All of the bootstrap in the world won't do you a lick of good if you leave this tag out.

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

What does this tag do?

When Apple first came up with a phone that could actually browse the internet, they had a huge problem to solve. Namely, how on earth do you browse the internet on such a tiny screen? The solution? Pan and zoom. Because the iPhone did touch, did multi-touch and did all of it so well, Apple decided to have the browser treat pages as if it were a desktop browser. By default, the device will zoom all the way out on any page so that you can see the entire site. It's then up to you to double tab to zoom in. This was an enormously effective strategy, and as it turns out, people don't mind tapping and zooming. But we can do better. Using a viewport meta tag means that your page will be automatically zoomed rather than using the default width, which is usually 980px wide.

MOBILE FIRST

Bootstrap has always had an emphasis on mobile. However, up until version 3, mobile was a sort of "bolt-on" addition. In fact, the responsive styles were in a separate stylesheet that you could choose not to include.

As of Bootstrap 3, this has completely changed.

Bootstrap is now mobile first. This means that you are expected to be building a mobile app first, and special considerations are made to accommodative larger screens, not the other way around as it has been for quite some time.

This means that there has been a fundamental change in the grid system. If you knew the grid system in Bootstrap 2, it's going to require you to reacquaint yourself with the new "mobile first" Bootstrap 3 grid.

BOOTSTRAP 3 FLUID GRIDS

Bootstrap is still comprised of a container that – well – "contains" rows. Each row contains 1 – 12 columns that behave differently depending on the CSS class you choose to apply. You used to be able to choose between a fixed and fluid container. Now there is only the fluid container. The container centers the content on the page, providing equal spacing on the left and the right. Containers have a max width set at various breakpoints inside of Bootstrap.

View Demo In New Window

You can resize that example to see the container change size at different points. Notice that the grid has a max width of `1140px`. Once the width is reduced below `1200px`, the grid width is reduced to `940px`. Once we break `992px` on the window, the container drops to a max width of `720px`. The padding is always at least `15px`, and then set to auto on anything greater than `768px` – perfectly centering the container on the page.

The breakpoints are important to know, because they are going to tell you which point you grid is going to change it's layout, based on which column types you choose. That's a bit confusing, so let's just look at the media queries that Bootstrap has first.

BOOTSTRAP 3 MEDIA QUERIES

Bootstrap divides up the layout into 4 categories, phones – or anything 400 pixels or larger. Tablets, which are 768 pixels or larger, medium sized devices – enormous tablets or desktops, and large devices, like giant cinema displays.

```
/* Extra small devices (phones, up to 480px) */
/* No media query since this is the default in Bootstrap */

/* Small devices (tablets, 768px and up) */
@media (min-width: @screen-sm) { ... }

/* Medium devices (desktops, 992px and up) */
@media (min-width: @screen-md) { ... }

/* Large devices (large desktops, 1200px and up) */
@media (min-width: @screen-lg) { ... }

*Source: [Bootsrap 3 Docs / CSS / Grid system / Media queries](http://
getbootstrap.com/css/#grid-media-queries)*
```

Based on these media queries, Bootstrap 3 provides you with 4 different types of columns. Each of these columns resides in a `row`, and the entire layout resides in a singular `container` classed element. You may have a maximum of 12 columns in any row.

Lets look at the four different column types.

`col-xs`

The first type is based on the first media query and is targeted at phones. These columns never stack or break. They are always horizontal and will continue to collapse as much as the viewport does. They also have `15px` of left and right padding as you can see from the example below.

View Demo In New Window

`col-sm`

The second type is the the small column, abbreviated `col-sm`. It stays horizontal at anything above `768px` and breaks below that to a stacked layout.

View Demo In New Window

`col-md`

The medium column is stacked at anything below `992px`, and is intended for a smaller desktop layout.

View Demo In New Window

`col-lg`

The last type is the `col-lg` column, which is stacked below `1200px`. On really big screens it's horizontal, but stacked everywhere else.

View Demo In New Window

WHY FOUR COLUMN TYPES

So why do we need for column types? Didn't the previous 1 responsive column type cut it? On their own, each of these columns types is relatively identical in it's behavior (except the xs columns). You get below a certain resolution and it stacks. The answer is really choice.

Having four column types gives you control of exactly how your layout behaves at the 4 major decision points as defined by Bootstrap. If all of your columns immediately stack at anything below desktop size, you are lumping all mobile devices in together, and that's not fair. Many tablets have a resolution that's not quite big enough to handle a full desktop site, but a mobile experience feels like a "Fisher Price" UI. Having four columns enables you to build that UI that is "right for the resolution", without having leave anyone out.

OFFSETS

Bootstrap offers some additional classes to help you with precise positioning. You can pad a column left by a specified number of columns using an offset. Note that this only works for `col-sm-*` and larger column types. Offsets are not respected on the `col-xs-*` layout.

View Demo In New Window

PUSH ME, PULL YOU

Grids also allow you to push and pull columns. This lets you to re-order the columns based on their class, not their position in the DOM. On first glance, this might seem like a bad idea, but again, this all about choice. It allows you to reorder your content based on the viewport size. You may want to do this in the case that you have certain content that is important and gets pushed below the fold on smaller screens. For example, you have have a 2 column layout on desktop, you may want a very important image on the right and the text on the left. However, on mobile, you want it to stack on top of the text because it's more important to you to have the user see the image if nothing else. You shouldn't have to be forced to put it on the left for desktop just to achieve a higher stack position on mobile.

Pushing a column moves it right, and pulling it moves it left.

View Demo In New Window

Again, these classes are not available for the smallest `col-xs-*` sizes. This is because it is mobile first. Your base design is the smallest, and you only rearrange content as the screen gets bigger. This is hard to wrap your head around when you first start with Bootstrap 3, but once it sinks in, things really start to fall into place.

FLUID MEDIA

The last thing for us to discuss in terms of the three pieces of responsive design is responsive media. As far as what Bootstrap covers, this refers to images and not video. Responsive video is a scenario not covered out-of-the-box by Bootstrap since it generally requires some JavaScript trickery.

IMAGES

Lets look at the way an image behaves on the web. Usually you know the size of your images before they are loaded into the page. As a general rule of thumb, you want to specify their width and height when possible. Otherwise the browser has no way of knowing what the image dimensions are until the image loads, and therefore no spot on the page is reserved for it. This will cause your layout to jump.

Have a look at a kitty image which has a fixed size of 400 by 400 pixels. In this example, I have added a fixed height and width inline on the element. Typically, your images should have a specified width and height if you know it ahead of time.

Notice though that the image is too large for the viewport, and while the text wraps in it's container, the images stays at it's fixed size.

View Demo In New Window

What we want is for the image to resize itself as it's container shrinks. As it turns out, making images fluid is pretty easy. All you have to do is add a `width` of 100% and set the `height` to `auto`. Done. Bootstrap does this for you by use of the `img-responsive` tag. This is another change from Bootstrap 2. All images used to be responsive when the responsive CSS was added. Now you get to toggle that on and off with just a class.

Some of the other Bootstrap 3 utility classes for images will make your image responsive by default. For instance, applying the `img-thumbnail` class for an image thumbnail makes your image responsive as well. Using rounded corners or the `img-circle` class still requires use of the `img-responsive` class to have the image effect and a still get a responsive image.

RESPONSIVE SUGAR

As of right now, we have discussed how Bootstrap 3 addresses fluid grids, fluid media and media queries. If it left you there and provided nothing else, this would be a great framework for navigation and conditionally showing and hiding content. The Bootstrap 3 creators know that, and have included a bit of responsive design 'sugar' to really jump start your app.

COLLAPSIBLE NAVBARS

The responsive navbar is the defining feature of Bootstrap. I'm not going to lie. I love it. Figuring out how to adapt your main navigation to smaller screens is a giant obstacle to your success, and Bootstrap effectively negates this with the use of the collapsible navbar.

That navbar that we all love so much stays much the same in this release. Some of the classes have changed, but it's largely the same concept. I think we all know what a Bootstrap 3 navbar is, and I highly doubt you need to see it again, but here it is in all it's glory.

View Demo In New Window

A couple of notes on the navbar:

- You must have the Bootstrap JavaScript collapse plugin in order for the responsive features to work
- You don't have to have the navbar at the top! You can fix it to the bottom too. True story.
- You should add a `data-role="navigation"` to every navbar if you want keyboard navigation (accessibility).
- You can tweak a navbar so it collapses at a different breakpoint than what Bootstrap specifies

RESPONSIVE UTILITY CLASSES

Bootstrap still comes with a set of handy classes for toggling the visibility of elements at its four specified breakpoints. The following example has four icons: a phone; a tablet; a laptop; and a desktop. Each of these icons is assigned Bootstrap 3 utility classes so that it is only visible at it's corresponding size. For instance, the phone icon in the demo looks like this:

```
<i class="icon-mobile-phone" .visible-xs></i>
```

Note that if you specify `visible-xs` on an element, Bootstrap will automatically hide it for you at any other breakpoints. If you want it visible again at a breakpoint, you need to add in a visibility class for that size.

View Demo In New Window

BOOTSTRAP 3 IS A BETTER BOOTSTRAP

After working with Bootstrap 3 and getting over the changes, I really like what they have done. Bootstrap 2 was incredibly comprehensive. Bootstrap 3 takes that all encompassing framework and adds in very granular control for laying out elements at different resolutions.

My favorite feature of all though, is the way that Bootstrap 3 is nudging us towards building for mobile devices before we build for desktops. It's subtle, but it will change the way you think without you even realizing it.

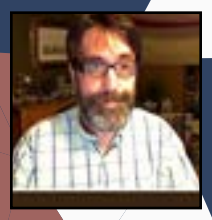
Burke Holland

Web Developer

HIS BLOG | SHARE | TWITTER | GITHUB

Adding a file display list to a multiple-file HTML upload

by Ray Camden



by Raymond Camden

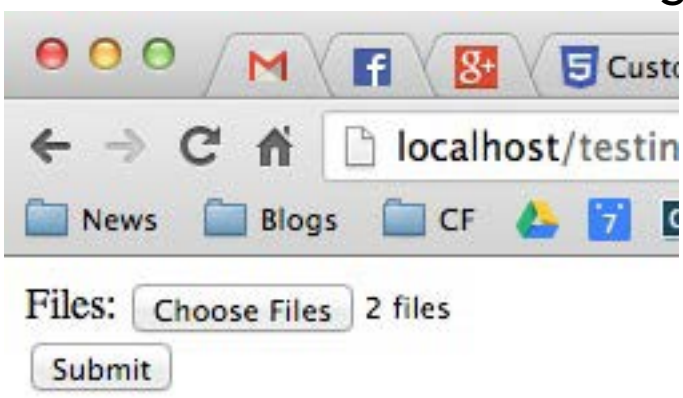
"I thought I'd share a quick tip about working with multi-file upload controls."

I'm working on something a bit interesting with a multi-file upload control, but while that is in development, I thought I'd share a quick tip about working with multi-file upload controls in general.

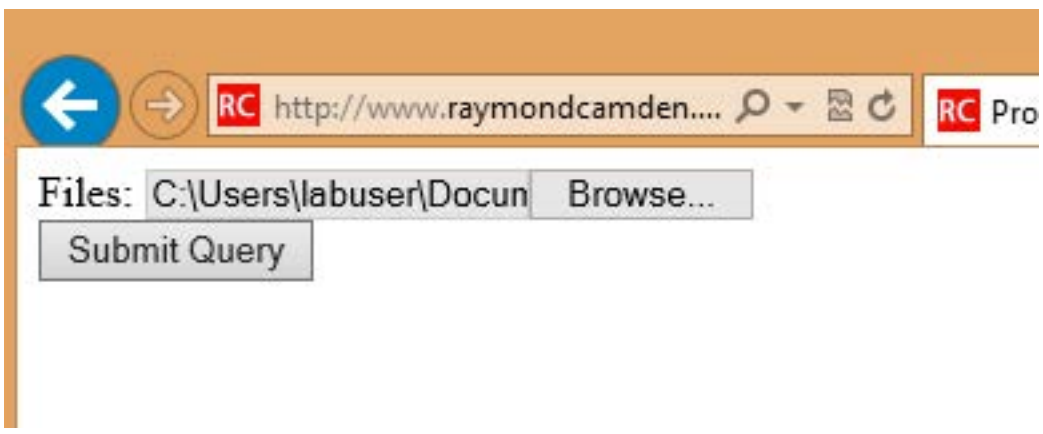
If you are not clear about what I'm talking about, I simply mean adding the multiple attribute to the input tag for file uploads. Like so:

```
<input type="file" name="foo" id="foo" multiple>
```

In browsers that support it, the user will be able to select multiple files. In browsers that don't support it, it still works fine as a file control, but they are limited to one file. In theory, this is pretty trivial to use, but there's a UX issue that kind of bugs me. Here is a screen shot of a form using this control. I've selected two files:



Notice something? The user isn't told what files they selected. Now obviously in a form this small it isn't that big of a deal, but in a larger form the user may forget or simply want to double check before they submit the form. Unfortunately there is no way to do that. Clicking the Browse button simply opens the file picker again. Surprisingly, IE handles this the best. It provides a read-only list of what you selected:



One could use a bit of CSS to make that field a bit larger for sure and easier to read, but you get the idea. So how can we provide some feedback to the user about what files they have selected?

First, let's add a simple change handler to our input field:

```
document.addEventListener("DOMContentLoaded", init, false);

function init() {
  document.querySelector('#files').addEventListener('change', handleFileSelect,
false);
}
```

Next, let's write an event handler and see if we can get access to the files property of the event. Not all browsers support this, but in the ones that do, we can enumerate over them.

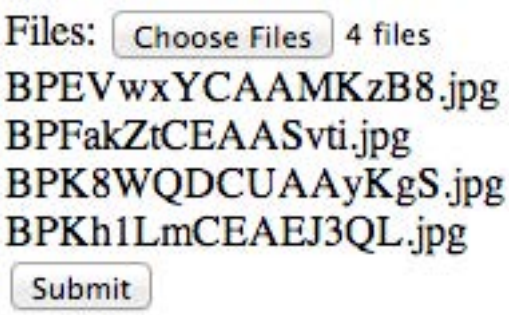
```
function handleFileSelect(e) {

  if(!e.target.files) return;

  var files = e.target.files;
  for(var i=0; i < files.length; i++) {
      var f = files[i];
  }

}
```

The file object gives us a few properties, but the one we care about is the name. So let's create a full demo of this. I'm going to add a little div below my input field and use it as place to list my files.

```
<!doctype html>
<html>
<head>
<title>Proper Title</title>
</head>

<body>

  <form id="myForm" method="post" enctype="multipart/form-data">

      Files: <input type="file" id="files" name="files" multiple><br/>
```

```
        <div id="selectedFiles"></div>

        <input type="submit">
    </form>

    <script>
    var selDiv = "";

    document.addEventListener("DOMContentLoaded", init, false);

    function init() {
        document.querySelector('#files').addEventListener('change',
handleFileSelect, false);
        selDiv = document.querySelector("#selectedFiles");
    }

    function handleFileSelect(e) {

        if(!e.target.files) return;

        selDiv.innerHTML = "";

        var files = e.target.files;
        for(var i=0; i<files.length; i++) {
            var f = files[i];

            selDiv.innerHTML += f.name + "<br/>";

        }
    }
    </script>
</body>
</html>
```

Pretty simple, right? You can view an example of this here: http://www.raymondcamden.com/demos/2013/sep/10/test0A.html. And here is a quick screen shot in case you are viewing this in a non-compliant browser.



Pretty simple, right? Let's kick it up a notch. Some browsers support FileReader (MDN Reference), a basic way of reading files on the user system. We could check for FileReader support and use it to provide image previews. I'll share the code first and then explain how it works.

A big thank you to Sime Vidas for pointing out a stupid little bug in my code I missed on first pass around. I made a classic array/callback bug and didn't notice it. I fixed the code and the screen shot, but if you want to see the broken code, view source on http://www.raymondcamden.com/demos/2013/sep/10/test0orig.html.

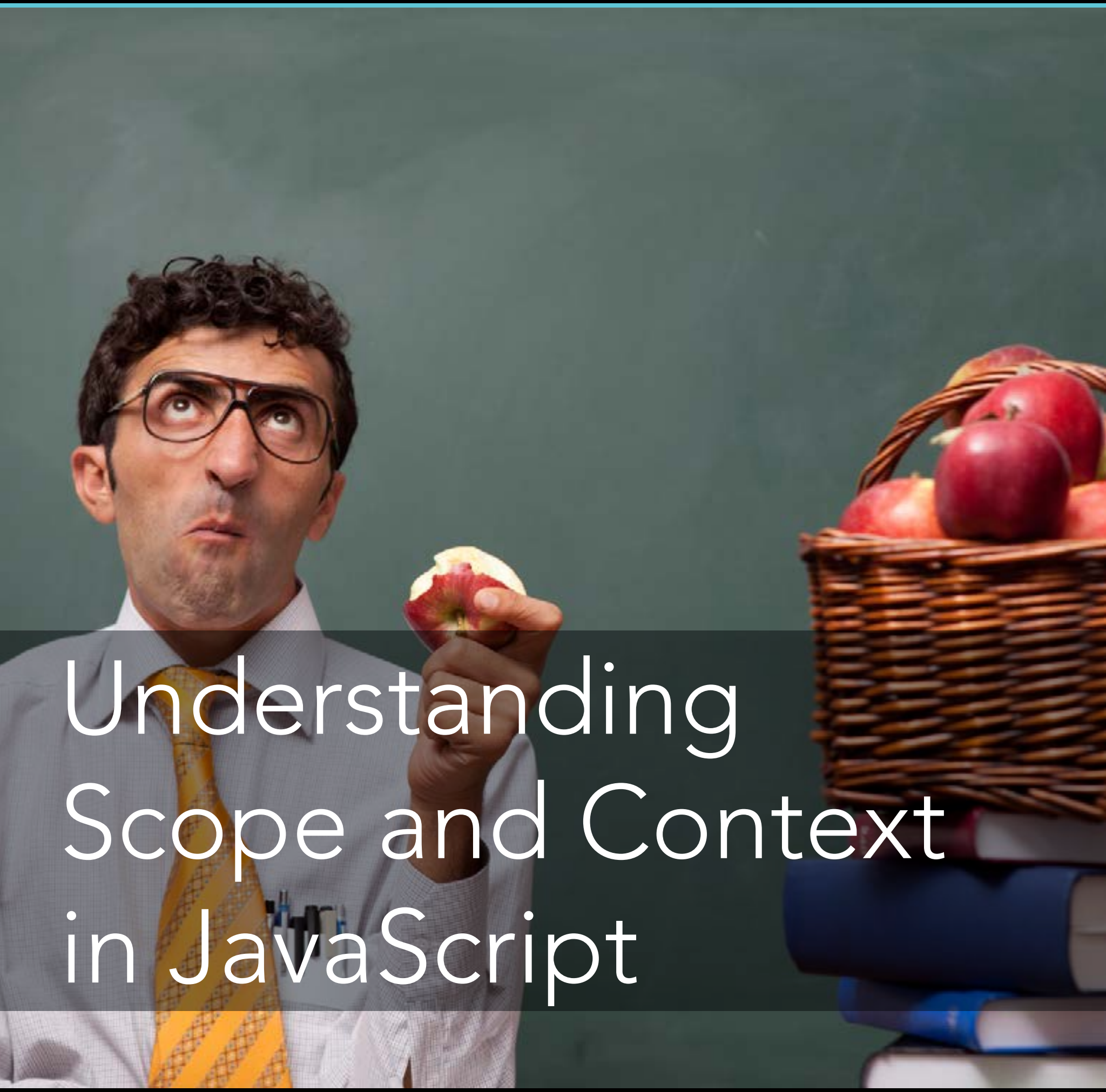
```
<!doctype html>
<html>
<head>
<title>Proper Title</title>
<style>
   #selectedFiles img {
        max-width: 125px;
        max-height: 125px;
        float: left;
        margin-bottom:10px;
   }
</style>
</head>

<body>

   <form id="myForm" method="post" enctype="multipart/form-data">

        Files: <input type="file" id="files" name="files" multiple
accept="image/*"><br/>

        <div id="selectedFiles"></div>

        <input type="submit">
   </form>

   <script>
   var selDiv = "";

   document.addEventListener("DOMContentLoaded", init, false);

   function init() {
       document.querySelector('#files').addEventListener('change',
handleFileSelect, false);
       selDiv = document.querySelector("#selectedFiles");
```

```
    }

    function handleFileSelect(e) {

        if(!e.target.files || !window.FileReader) return;

        selDiv.innerHTML = “”;

        var files = e.target.files;
        var filesArr = Array.prototype.slice.call(files);
        filesArr.forEach(function(f) {
            var f = files[i];
            if(!f.type.match(“image.*”)) {
                return;
            }

            var reader = new FileReader();
            reader.onload = function (e) {
                var html = “<img src=\”” + e.target.result + “\”>” + f.name +
“<br clear=\”left\”/>”;
                selDiv.innerHTML += html;
            }
            reader.readAsDataURL(f);
        });

    }
    </script>

</body>
</html>
```

I’ve modified the handleFileSelect code to check for both the files array as well as FileReader. (Note - I should do this before I even attach the event handler. I just thought of that.) I’ve updated my input field to say it accepts only images and added a second check within the event handler. Once we are sure we have an image, I use the FileReader API to create a DataURL (string) version of the image. With that I can actually draw the image as a preview.

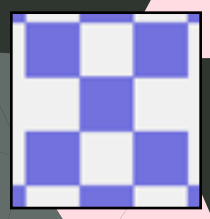
You can view a demo of this here: http://www.raymondcamden.com/demos/2013/sep/10/test0.html. And again, a screen shot:

Check it out and let me know what you think. As I said, it should be fully backwards compatible (in that it won't break) and works well in Chrome, Firefox, IE10, and Safari.



Understanding Scope and Context in JavaScript

by Ryan Morr



by Ryan Morr

"Javascript's implementation of scope and context is a unique feature."

This article was originally published on Flippin' Awesome on August 26, 2013. You can read it here.

JavaScript's implementation of scope and context is a unique feature of the language, in part because it is so flexible. Functions can be adopted for various contexts and scope can be encapsulated and preserved. These concepts are behind some of the most powerful design patterns JavaScript has to offer. However, this is also a tremendous source of confusion amongst developers, and for good reason. The following is a comprehensive explanation of scope and context in JavaScript, the difference between them, and how various design patterns make use of them.

CONTEXT VS. SCOPE

The first important thing to clear up is that context and scope are not the same. I have noticed many developers over the years often confuse the two terms, incorrectly describing one for the other. To be fair, the terminology has become quite muddled over the years.

Every function invocation has both a scope and a context associated with it. Fundamentally, scope is function-based while context is object-based. In other words, scope pertains to the variable access of a function when it is invoked and is unique to each invocation. Context is always the value of the `this` keyword, which is a reference to the object that "owns" the currently executing code.

VARIABLE SCOPE

A variable can be defined in either local or global scope, which establishes the variables' accessibility from different scopes during runtime. Any defined global variable, meaning any variable declared outside of a function body, will live throughout runtime and can be accessed and altered in any scope. Local variables exist only within the function body of which they are defined and will have a different scope for every call of that function. There it is subject for value assignment, retrieval, and manipulation only within that call and is not accessible outside of that scope.

JavaScript presently does not support block scope which is the ability to define a variable to the scope of an if statement, switch statement, for loop, or while loop. This means the variable will not be accessible outside the opening and closing curly braces of the block. Currently any variables defined inside a block are accessible outside the block. However, this is soon to change, the `let` keyword has officially been added to the ES6 specification. It can be used as an alternative to the `var` keyword in order to support the declaration of block scope local variables.

WHAT IS "THIS" CONTEXT

Context is most often determined by how a function is invoked. When a function is called as a method of an object, `this` is set to the object the method is called on:

```
var object = {
    foo: function(){
        alert(this === object);
    }
};

object.foo(); // true
```

The same principle applies when invoking a function with the `new` operator to create an instance of an object. When invoked in this manner, the value of `this` within the scope of the function will be set to the newly created instance:

```
function foo(){
    alert(this);
}

foo() // window
new foo() // foo
```

When called as an unbound function, `this` will default to the global context or window object in the browser. However, if the function is executed in strict mode, the context will default to undefined.

EXECUTION CONTEXT AND SCOPE CHAIN

JavaScript is a single threaded language, meaning only one thing can be done at a time in the browser. When the JavaScript interpreter initially executes code, it first enters into a global execution context by default. Each invocation of a function from this point on will result in the creation of a new execution context.

This is where confusion often sets in, the term "execution context" is actually for all intents and purposes referring to scope and not context as previously discussed. It is an unfortunate naming convention, however it is the terminology as defined by the ECMAScript specification, so were kinda stuck with it.

Each time a new execution context is created, it is appended to the top of what is called a scope chain, sometimes referred to as an execution or call stack. The browser will always execute the current execution context that is atop the scope chain. Once completed, it will be removed from the top of the stack and control will return to the execution context below. For example:

```
function first(){
    second();
    function second(){
        third();
        function third(){
            fourth();
            function fourth(){
                // do something
            }
        }
    }
}
first();
```

Running the preceeding code will result in the nested functions being executed all the way down to the `fourth` function. At this point the scope chain would be, from top to bottom: fourth, third, second, first, global. The `fourth` function would have access to global variables and any variables defined within the `first`, `second` and `third` functions as well as the functions themselves. Once the `fourth` function has

completed execution, it will be removed from the scope chain and execution will return to the `third` function. This process continues until all code has completed executing.

Name conflicts amongst variables between different execution contexts are resolved by climbing up the scope chain, moving locally to globally. This means that local variables with the same name as variables higher up the scope chain take precedence.

An execution context can be divided into a creation and an execution phase. In the creation phase, the interpreter will first create a variable object (also called an activation object) that is composed of all the variables, function declarations and arguments defined inside the execution context. From there the scope chain is initialized next and the value of `this` is determined last. Then in the execution phase, code is interpreted and executed.

To put it simply, each time you attempt to access a variable within a function's execution context, the look-up process will always begin with its own variable object. If the variable is not found in the variable object, the search continues into the scope chain. It will climb up the scope chain examining the variable object of every execution context looking for a match to the variable name.

CLOSURES

A closure is formed when a nested function is made accessible outside of the function in which it was defined, so that it may be executed after the outer function has returned. It maintains access to the local variables, arguments, and inner function declarations of its outer function. Encapsulation allows us to hide and preserve the execution context from outside scopes while exposing a public interface and thus is subject to further manipulation. A simple example of this looks like the following:

```
function foo(){
    var local = 'private variable';
    return function bar(){
        return local;
    }
}

var getLocalVariable = foo();
getLocalVariable() // private variable
```

One of the most popular types of closures is what is widely known as the module pattern. It allows you to emulate public, private and privileged members:

```
var Module = (function(){
    var privateProperty = ‘foo’;

    function privateMethod(args){
        //do something
    }

    return {

        publicProperty: “”,

        publicMethod: function(args){
            //do something
        },

        privilegedMethod: function(args){
            privateMethod(args);
        }
    }
})();
```

The module acts as if it were a singleton, executed as soon as the compiler interprets it, hence the opening and closing parenthesis at the end of the function. The only available members outside of the execution context of the closure are your public methods and properties located in the return object (`Module.publicMethod` for example). However, all private properties and methods will live throughout the life of the application as the execution context is preserved, meaning variables are subject to further interaction via the public methods.

Another type of closure is what is called an immediately-invoked function expression (IIFE) which is nothing more than a self-invoked anonymous function executed in the context of the window:

```
function(window){

    var a = ‘foo’, b = ‘bar’;

    function private(){
        // do something
    }

    window.Module = {

        public: function(){
            // do something
        }
    };

})(this);
```

This expression is most useful when attempting to preserve the global namespace as any variables declared within the function body will be local to the closure but will still live throughout runtime. This is a popular means of encapsulating source code for applications and frameworks, typically exposing a single global interface to interact with.

CALL AND APPLY

These two simple methods, inherent to all functions, allow you to execute any function in any desired context. The `call` function requires the arguments to be listed explicitly while the `apply` function allows you to provide the arguments as an array:

```
function user(first, last, age){
    // do something
}
user.call(window, ‘John’, ‘Doe’, 30);
user.apply(window, [‘John’, ‘Doe’, 30]);
```

The result of both calls is exactly the same, the user function is invoked in the context of the window and provided the same three arguments.

ECMAScript 5 (ES5) introduced the `Function.prototype.bind` method that is used for manipulating context. It returns a new function that is permanently bound to the first argument of `bind` regardless of how the function is being used. It works by using a closure that is responsible for redirecting the call in the appropriate context. See the following polyfill for unsupported browsers:

```
if(!('bind' in Function.prototype)){
    Function.prototype.bind = function(){
        var fn = this, context = arguments[0], args = Array.prototype.slice.
call(arguments, 1);
        return function(){
            return fn.apply(context, args);
        }
    }
}
```

It is commonly used where context is frequently lost: object-orientation and event handling. This is necessary because the `addEventListener` method of a node will always execute the callback in the context of the node the event handler is bound to, which is the way it should be. However if your employing advanced object-oriented techniques and require your callback to be a method of an instance, you will be required to manually adjust the context. This is where bind comes in handy:

```
function MyClass(){
    this.element = document.createElement('div');
    this.element.addEventListener('click', this.onClick.bind(this), false);
}

MyClass.prototype.onClick = function(e){
    // do something
};
```

While reviewing the source of the `bind` function, you may have also noticed what appears to be a relatively simple line of code involving a method of an Array:

```
Array.prototype.slice.call(arguments, 1);
```

What is interesting to note here is that the `arguments` object is not actually an array at all, however it is often described as an array-like object much like a nodelist (anything returned by `document.getElementsByTagName()`). They contain a length property and indexed values but they are still not arrays, and subsequently don't support any of the native methods of arrays such as slice and push. However, because of their similar behavior, the methods of Array can be adopted or hijacked, if you will, and executed in the context of an array-like object, as in the case above.

This technique of adopting another object's methods also applies to object-orientation when emulating classical inheritance in JavaScript:

```
MyClass.prototype.init = function(){
    // call the superclass init method in the context of the “MyClass” instance
    MySuperClass.prototype.init.apply(this, arguments);
}
```

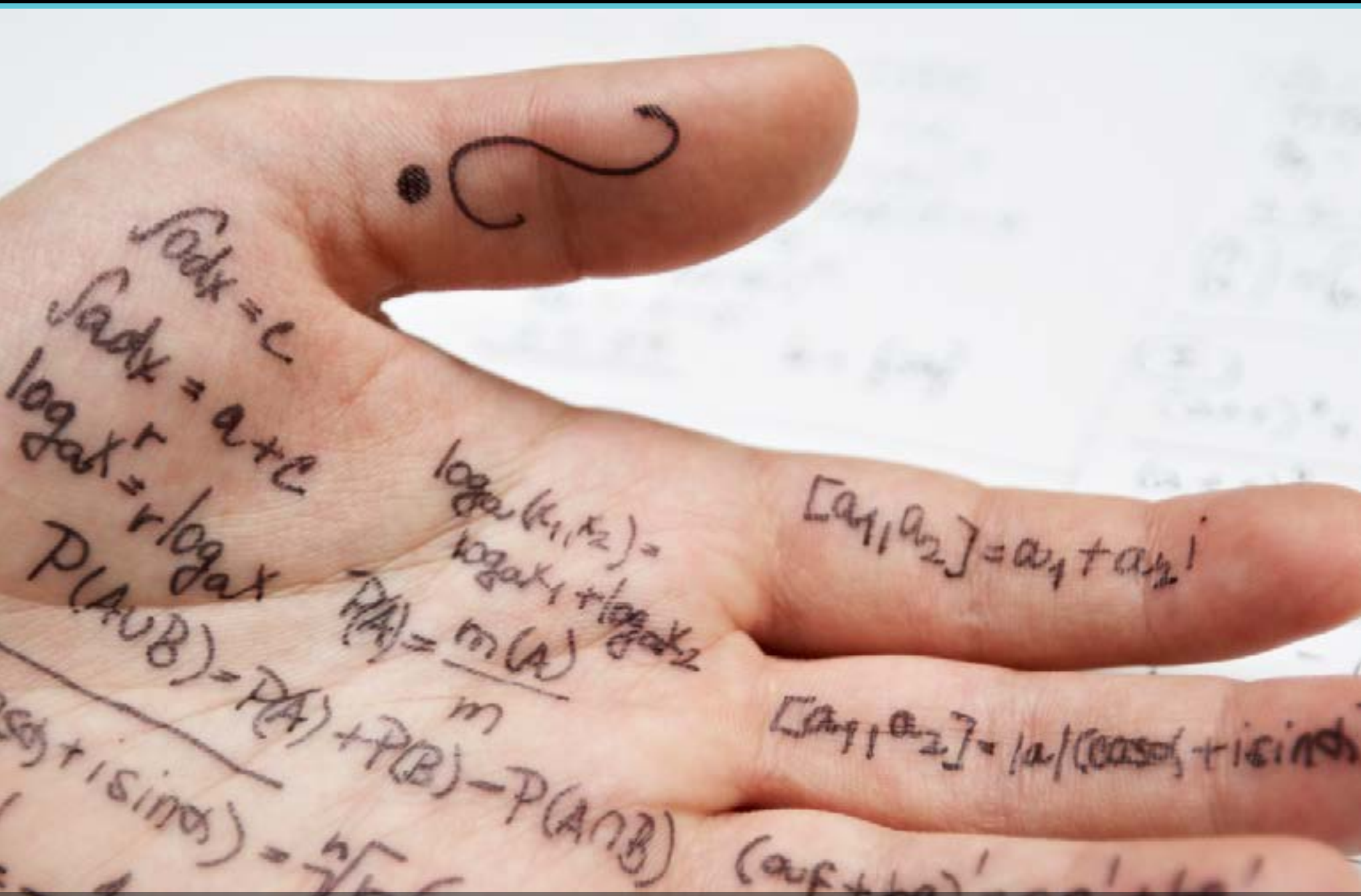
By invoking the method of the superclass (`MySuperClass`) in the context of an instance of a subclass (`MyClass`), we can mimic this powerful design pattern.

CONCLUSION

It is important to understand these concepts before you begin to approach advanced design patterns, as scope and context play a significant and fundamental role in modern JavaScript. Whether were talking about closures, object-orientation and inheritance, or various native implementations, context and scope play an important role in all of them. If your goal is to master the JavaScript language and better understand all it encompasses, then scope and context should be one of your starting points.

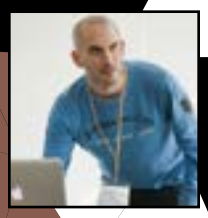
This article was originally published at http://ryanmorr.com/understanding-scope-and-context-in-javascript/

Ryan Morr
Web Developer



Data in prototype properties

by Dr. Axel Rauschmayer



by Dr. Axel Rauschmayer

"This explains why you should and should not put data in prototype properties."

This article explains when you should and should not put data in prototype properties.

AVOID: PROTOTYPE PROPERTIES WITH INITIAL VALUES FOR INSTANCE PROPERTIES

Prototypes contain properties that are shared by several objects. As such, they work well for methods. Additionally, with the technique shown below, you can also use them to provide initial values for instance properties. I'll later explain why that is not recommended.

A constructor usually sets instance properties to initial values. If one such value is a default then you don't need to create an instance property. You only need a prototype property with the same name whose value is the default. For example:

```
/**
 * Anti-pattern: don't do this
 *
 * @param data an array with names
 */
function Names(data) {
    if (data) {
        // There is a parameter
        // => create instance property
        this.data = data;
    }
}
Names.prototype.data = [];
```

The parameter `data` is optional. If it is missing, the instance does not get a property data, but inherits `Names.prototype.data`, instead.

This approach mostly works: You can create an instance `n` of `Names`. Getting `n.data` reads `Names.prototype.data`. Setting `n.data` creates a new own property in `n` and preserves the shared default value in the prototype. We only have a problem if we change the default value (instead of replacing it with a new value):

```
> var n1 = new Names();
> var n2 = new Names();

> n1.data.push('jane'); // change default value
> n1.data
[ 'jane' ]

> n2.data
[ 'jane' ]
```

Explanation: `push()` changed the array in `Names.prototype.data`. Since that array is shared by all instances without an own property `data`, `n2.data`'s initial value has changed, too.

BEST PRACTICE: DON'T SHARE DEFAULT VALUES

Therefore, it is better to not share default values and to always create new ones:

```
function Names(data) {
        this.data = data || [];
    }
```

Obviously, the problem of modifying a shared default value does not arise if that value is immutable (as all primitives [1] are). But for consistency's sake, it's best to stick to a single way of setting up properties. I also prefer to maintain the usual separation of concerns [2]: the constructor sets up the instance properties, the prototype contains the methods.

ECMAScript 6 will make this even more of a best practice, because constructor parameters can have default values and you can define prototype methods in class bodies, but not prototype properties with data.

CREATING INSTANCE PROPERTIES ON DEMAND

Occasionally, creating a property value is an expensive operation (computationally or storage-wise). Then you can create an instance poperty on demand:

```
function Names(data) {
    if (data) this.data = data;
}
Names.prototype = {
    constructor: Names,
    get data() {
        // Define, don't assign [3]
        // => ensures an own property is created
        Object.defineProperty(this, 'data', {
            value: [],
            enumerable: true
            // Default - configurable: false, writable: false
            // Set to true if property's value must be changeable
        });
        return this.data;
    }
};
```

(As an aside, we have replaced the original object in `Names.prototype`, which is why we need to set up the property `constructor` [4].)

Obviously, that is quite a bit of work, so you have to be sure it is worth it.

AVOID: NON-POLYMORPHIC PROTOTYPE PROPERTIES

If the same property (same name, same semantics) exists in several prototypes, it is called polymorphic. Then the result of reading the property via an instance is dynamically determined via that instance's prototype. Prototype properties that are not used polymorphically can be replaced by variables (which better reflects their non-polymorphic use).

Example: You can store a constant in a prototype property and access it via `this`.

```
function Foo() {}
Foo.prototype.FACTOR = 42;  // primitive value, immutable
Foo.prototype.compute = function (x) {
    return x * this.FACTOR;
};
```

This constant is not polymorphic. Therefore, you can just as well access it via a variable:

```
// This code should be inside an IIFE [5] or a module
function Foo() {}
var FACTOR = 42;  // primitive value, immutable
Foo.prototype.compute = function (x) {
    return x * FACTOR;
};
```

The same holds for storing mutable data in non-polymorphic prototype properties. Mutable prototype properties are difficult to manage. If they are non-polymorphic then you can at least replace them with variables.

POLYMORPHIC PROTOTYPE PROPERTIES

An example of polymorphic prototype properties with immutable data: Tagging instances of a constructor via prototype properties enables you to tell them apart from instances of a different constructor.

```
function ConstrA() { }
ConstrA.prototype.TYPE_NAME = 'ConstrA';

function ConstrB() { }
ConstrB.prototype.TYPE_NAME = 'ConstrB';
```

Thanks to the polymorphic "tag" `TYPE_NAME`, you can distinguish the instances of `ConstrA` and `ConstrB` even when they cross frames (then `instanceof` does not work [6]).

REFERENCES

[1] Categorizing values in JavaScript
[2] JavaScript inheritance by example
[3] Properties in JavaScript: definition versus assignment
[4] What's up with the "constructor" property in JavaScript?
[5] JavaScript variable scoping and its pitfalls
[6] Categorizing values in JavaScript [Sect. 2.4 explains that `instanceof` doesn't work if objects cross frames]

Dr. Axel Rauschmayer

JavaScript Consultant

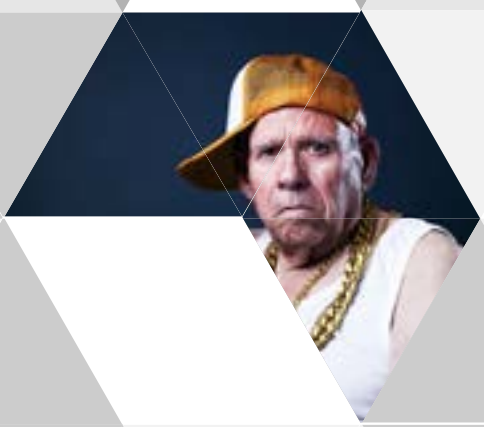
HIS BLOG | SHARE | TWITTER | GITHUB

Creative Loading Effects
by Mary Lou

Image Com-pression for web devs
by Colt McAnlis

Control CSS Animations with JavaScript
by
Zach Saucier

Videos as Backgrouns
Johnny Simpson



Playing with the Details/ Summary Tag
by Raymond Camden

Node. js Jumpstart
by Jeremy Osborne

ECMAScript Internationalization
by Dr. Rauschmayer

PhoneGap for Android
by Holly Schinsky

Three. js Texture updating w/ Photoshop CC
by Renaun Erickson

Upload Pix from PhoneGap to Amazon
Christophe Coenraets