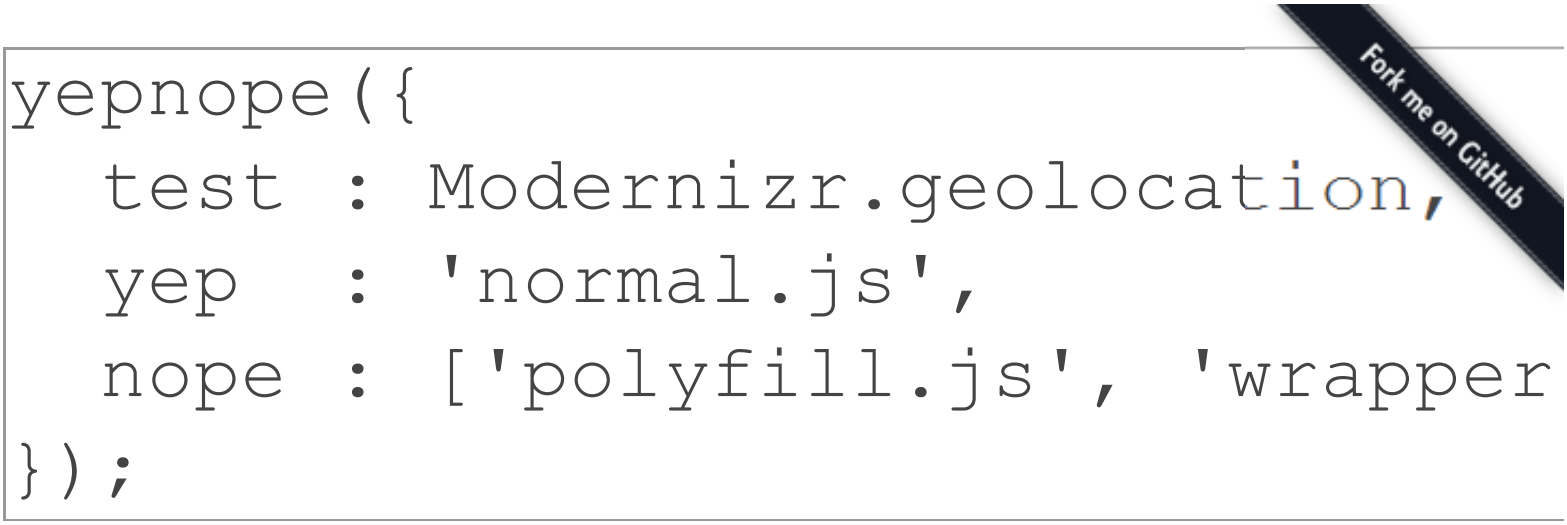
(https://g



(http://github.com/SlexAxton/yepnope.js)

yepnope is an asynchronous conditional resource loader tha

and allows you to load **only the scripts that your users ne**

The API

There are only a handful of things to know about yepnope. There are only 3 functions available to you and th important.

```
yepnope(resources /* string | object | array */)
```

The **`yepnope`** function is the core of yepnope.js (crazy, huh?). It takes a whole bunch of stuff, to try to make i

The recommended type for **`resources`** is an array of test objects (#testObject). The consistency helps ma stay maintainable.

However, if you don't need more than one test group, then you can avoid putting it in an array, and just send

If you're *really* feeling lazy, you can just pass a string in, as well.

On top of all that, inside of your array, each item can be a test object, another array, or a string literal. Just s
there's a good chance yepnope will take it (as long as it's a string, array, or test object)

What's in a test object?!

Great question! Here's a list of the things that yepnope will care about if you put it in a test object. *All of thes*

```
yepnope([{
  test : /* boolean(ish) - Something truthy that you want to test
  yep  : /* array (of strings) | string - The things to load if te
  nope : /* array (of strings) | string - The things to load if te
  both : /* array (of strings) | string - Load everytime (sugar)
  load : /* array (of strings) | string - Load everytime (sugar)
  callback : /* function ( testResult, key ) | object { key : fn }
  complete : /* function
}, ... ]);
```

Note that yepnope can load CSS or JS files. All resources will be considered JS files unless they end in '.c
other extensions (cache-busted scripts, for example) to be interpreted as CSS files if you use the `css!` pre

---

Another Script Loader?#?@$!@# - Why Use Yep

You're right. The last few months we've been **inundated** by new script loaders. I'd like to first point out that t
that there is a real benefit to using a script/resource loader. In fact, the yepnope prototype launched at JSC
2010. Since then, we've used some of the best techniques from other script loaders and packaged up the r
API.

I'd like it to be very clear why/when you'd want to use yepnope instead of one of the many other options, as
appropriate.

Good Things

- yepnope.js is only 1.6kb - smaller than most and certainly a good size for its functionality set.
- yepnope.js is called a "resource loader" because it can work with both JavaScript and CSS.
- yepnope.js has a full test suite in QUnit that you can run in your set of supported browsers to make sure TestSwarm in every browser we can get our hands on)

- yepnope.js fully decouples preloading from execution. This means that you have ultimate control of wher you can change that order on the fly.
- The yepnope.js api is friendly and encourages logical grouping of resources.
- yepnope.js is modular. It has a whole system for adding your own functionality and a couple examples o and filters).
- The yepnope.js api encourages you to only load the resources that you need. This means that even whe loader, it still can come out on top, because you could avoid an entire resource.
- yepnope.js is integrated into Modernizr (http://modernizr.github.com/Modernizr/2.0-beta/).
- yepnope.js always executes things in the order they are listed. This is a pro for some, and a con for othe
- yepnope.js has the capability to do resource fallbacks and still download dependent scripts in parallel w

```
yepnope([{
  load: 'http://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jquer
  complete: function () {
    if (!window.jQuery) {
      yepnope('local/jquery.min.js');
    }
  }
}, {
  load: 'jquery.plugin.js',
  complete: function () {
    jQuery(function () {
      jQuery('div').plugin();
    });
  }
}]);
```

In most loaders, in order to test for and potentially use the fallback version of jQuery, said loader must do "chain" of scripts in the callback for jQuery:

```
someLoader('http://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jq
ion () {
  if (!window.jQuery) {
    someLoader('local/jquery.min.js', 'jquery.plugin.js');
  } else {
```

```
    someLoader('jquery.plugin.js');
  }
});
```

Not only is this not very pretty, and not very DRY, it's slow. It essentially means that jQuery will load in seri anything that depends on or comes after jQuery wont start downloading until after jQuery has executed.

yepnope, on the other hand, can start downloading `jquery.plugin.js` right away (at the same time as the the normal case, these will download at the same time and they will execute in order. In the case where have to wait the additional time for the second jQuery to load in order to continue on. This is made possi preloading and executing.

Things to keep in mind

- It is not always the fastest. There are a number of other script loaders (such as labjs) that optimize differ should run tests for your use-case if this is a priority to you.
- We require that you have proper cache headers. Other loaders get around this on local resources.

```
Expires: -future date-
// e.g.
Expires: Thu, 31 Dec 2020 20:00:00 GMT
```

- Other, more feature-rich libraries like RequireJS have build tools and awesome apis to help you structur application. yepnope only scratches the surface compared to RequireJS and similar libraries.
- yepnope.js always executes things in the order they are listed. This is a pro for some, and a con for othe

Conclusion

In short, we want you to make the right decision for your application. We can certainly think of times when ye correct choice for a resource loader, and we hope we've made those situations abundantly clear in our doc think that yepnope meets a lot of people where they are, when it comes to a balance of performance, usabi

If you have questions about if yepnope is right for your application. Send us a tweet or an email and we'll he script-loader author scene are total BFFs and we aren't about competition, we're about pushing the edge. I LABjs and RequireJS in a bunch of our projects, because they're badass (and the other ones are probably

---

Yepnope: The Guide

Why should you use yepnope?

Yepnope has a simple API to help you order load and execute your scripts in the fastest possible way. Yepr and CSS resources, and loads them asynchronously and in parallel, but always executes them in the order customizable and extendable so you can add your own customizations with ease! Yepnope is small, weighi and gzipped.

Why not just concatenate all my scripts and put them at the bottom?

While this is not a bad practice, it is advice from the early days of performance driven script loading. Since cases, you can get better performance. For instance, if you concatenate all of your JavaScript into a single chunks of JavaScript that don't actually run on the page that you're on. These are wasted bytes, and cost yo street cred. Also, it's been shown (http://blog.getify.com/2009/11/labjs-why-not-just-concat/) that loading a s is actually faster than loading them all concatenated together, since they can all download at the same time http requests).

More specifically, a lot of the time, code is specific to a certain situation, yepnope capitalizes on this fact, a based on the situation you find yourself in. Many times, this manifests itself as Feature Tests. Feature Tests feature in the executing browser environment. Usually when a feature is not supported, a "polyfill" is loaded compatibility in non-supporting browsers. With yepnope, the only people who pay the cost of the polyfill file a

If yepnope saves you from loading a single JavaScript file, it probably paid for itself. Everything after that is

---

Can't I do this with other script loaders?

Yep. We just think this approach is specifically well-suited for this use-case, and other loaders are geared t them are super-awesome too!

What does this have to do with Modernizr? (or What is Modernizr.load ?

Currently yepnope is being included in special builds of Modernizr, because it's such a good companion to output of Modernizr is a fantastic input to yepnope. Yepnope is currently included unmodified from its origin Modernizr.load - you can use the two interchangeably, though we suggest that you stick to one or the other.

Using Yepnope

```
yepnope();
```

Takes: string, object, array of either

The yepnope function is where all the magic happens. It's built to take what you give it, so pay close attentio

---

A String

The most simple thing you can pass the yepnope function is a string.

```
yepnope('/url/to/your/script.js');
```

This will load your js file at sometime in the future (but as soon as possible into the future). That's because y
doesn't wait for the script to be finished loading before the next thing is allowed to happen. This is why yepn

"I'm still confused, why can't I use the script that I just loaded?"

You can! Just not like that. If you need to do something after your file has been loaded, you'll want to use a c

"You mean I can't do this?":

```
yepnope('js/jquery.js');
$.ajax(...); // NOPE
```

Sorry, but that's not the way it works! If you're not used to asynchronous code, this can be a chore, but we'll g
work like you want!

An Object

```
yepnope({
  load: '/url/to/your/script.js'
});
```

This is exactly equivalent to the first example with a string, except we can add on to this one. For instance, v
run whenever the file is done loading. This is done with the 'callback' property (go figure!). From now on, we
Object (even though sometimes it doesn't have a test in it ;).

```
yepnope({
  load: '/url/to/your/script.js',
  callback: function (url, result, key) {
    // whenever this runs, your script has just executed.
    alert('script.js has loaded!');
  }
});
```

If you're used to the asynchronous pattern, this should be a breeze, and if not, it'll seem natural in no time.

One of the most important parts of yepnope is your ability to pass in feature test results to determine whethe
script. Here's yepnope's API for doing that:

```
yepnope({
  test: window.JSON,
  nope: 'json2.js',
  complete: function () {
    var data = window.JSON.parse('{ "json" : "string" }');
  }
});
```

The power here, is that you were able to parse a JSON string, but if your browser already had built in suppo
require you to load anything extra!

Notice that we don't have a 'yep' script property set. That's because it isn't mandatory. If you don't have anyt
passes, you can leave out the 'yep', the same goes for 'nope'.

This is a simple example, and one based on a very easily replaceable browser feature. Many other things th
require some set of scripts and styles that aren't always able to be exact fill-ins for their native counterparts.
a way to help you structure it nicely.

Now if you want to load more than one script, you have a few options (based on your app, one might feel be
developers suggest that you always do what feels better).

You can set the load property in the test object to an array of urls.

```
yepnope({
```

```
  load: ['script1.js', 'style1.css'],
  callback: function (url, result, key) {
    // Wait! What happens in here now?! Does this get called once
  }
});
```

The callback becomes a little bit confusing when we introduce a multiple file load. In fact, the callback functi you load in the test object (keep that in mind when you learn new ways to load files).

It's about time you learned what the parameters that get passed into your function are for.

```
...
callback: function (url, result, key) {
  console.log(url, result, key);
}...
```

- **url** - This is the url (without any prefixes attached) that was loaded
- **result** - This is the result of your test, treat it as a truthy value
- **key** - If you provided a key mapping to your file, this is that key. Also if you use an array, it's the index of confusing if you used multiple arrays). In cases where you just use a string to load a resource, it's 0.

These are here to help you make decisions in your callbacks.

As shown in the previous example, often times we end up loading more than just one thing at a time. Yepno to act on either outcome of a test, so your callback often needs to know how that test turned out (without you again).

In the following example, assume your browser does not support geolocation (and the 'nope' array is what g

```
yepnope({
  test: Modernizr.geolocation,
  yep: 'regular-styles.css',
  nope: ['modified-styles.css', 'geolocation-polyfill.js'],
  callback: function (url, result, key) {
    alert(url);
  }
});
```

This will cause *TWO* separate alerts to occur. They will each contain the exact string (minus any prefixes, if url that was loaded from the `nope` array.

So in a simple case you could check to see which url is being loaded in order to know to do a certain thing:

```
yepnope({
  test: Modernizr.geolocation,
  yep: 'regular-styles.css',
  nope: ['modified-styles.css', 'geolocation-polyfill.js'],
  callback: function (url, result, key) {
    if (url === 'modified-styles.css') {
      alert('The Styles loaded!');
    }
  }
});
```

In this example, unlike the previous one, there will only be one alert. After the css has loaded, the url will mat
alert will occur. Since the url of the 'geolocation-polyfill.js' file does not match, there will be no alert.

Another thing you might want to know in this situation is the result of your test. Luckily, that's the second para
to mention which styles were loaded, then you might try something like this:

```
yepnope({
  test: Modernizr.geolocation,
  yep: 'regular-styles.css',
  nope: ['modified-styles.css', 'geolocation-polyfill.js'],
  callback: function (url, result, key) {
    if (url === 'modified-styles.css') {
      alert('The Styles loaded!');
      if (result) {
        alert('The Test Passed!');
      } else {
        alert('The test Failed!');
      }
    }
  }
});
```

Obviously, this example is a little bit unrealistic, but hopefully it helps you picture how you might use these in

You may be thinking that it's a bit cumbersome to always have to repeat the urls of the files that you load in c
for them. We agree!

There are a few options if you need very unique callbacks for your scripts. The first is the most simple, but a

yepnope supports an array of test objects, you can just separate the file declarations!

An Array

```
yepnope([{
  load: 'file1.js',
  callback: function (url, result, key) {
    alert('This is definitely file1!');
  }
}, {
  load: 'file2.js',
  callback: function (url, result, key) {
    alert('This is definitely file2!');
  }
}]);
```

This can get a little bit long and when the files rely on the same test, you don't want to have to rerun your feat
load based on its result. So you can use a key/value mapping in an object literal to name your resources. Th
know which of the files the callback is related to.

```
yepnope({
  test: Modernizr.geolocation,
  yep: {
    'rstyles': 'regular-styles.css'
  },
  nope: {
    'mstyles': 'modified-styles.css',
    'geopoly': 'geolocation-polyfill.js'
  },
  callback: function (url, result, key) {
    if (key === 'geopoly') {
      alert('This is the geolocation polyfill!');
    }
  }
});
```

Now we have a much more succinct way of knowing which one of our files our callback is related to. But as
you would have to write in order to find every single key would get ugly fast. So we also allow you to give a k

```
yepnope({
  test: Modernizr.geolocation,
  yep: {
    'rstyles': 'regular-styles.css'
  },
  nope: {
    'mstyles': 'modified-styles.css',
    'geopoly': 'geolocation-polyfill.js'
  },
  callback: {
    'rstyles': function (url, result, key) {
      alert('This is the regular styles!');
    },
    'mstyles': function (url, result, key) {
      alert('This is the modified styles!');
    },
    'geopoly': function (url, result, key) {
      alert('This is the geolocation polyfill!');
    }
  }
});
```

Only the keys that correspond to a file that actually got loaded will be called. So in this case, if geolocation w
callback would never fire.

We don't want this tutorial to sound too much like it's reading your mind, but we know what you're asking you
*all my callbacks, howdo I just run some code at the very end, after **everything** is done loading?"*

That's a **great** question! You are absolutely right. With this current situation, you'd have to duplicate code in
ran at the very end (depending on if the test passed or failed). That's why there's also the '**`complete`**' calll

```
yepnope({
  test: Modernizr.geolocation,
  yep: {
    'rstyles': 'regular-styles.css'
  },
  nope: {
    'mstyles': 'modified-styles.css',
    'geopoly': 'geolocation-polyfill.js'
  },
  callback: {
    'rstyles': function (url, result, key) {
```

```
      alert('This is the regular styles!');
    },
    'mstyles': function (url, result, key) {
      alert('This is the modified styles!');
    },
    'geopoly': function (url, result, key) {
      alert('This is the geolocation polyfill!');
    }
  },
  complete: function () {
    alert('Everything has loaded in this test object!');
  }
});
```

Now you can act on individual file loads and entire test object groups with clarity.

Here are some more questions and answers now that you're a little more familiar with how yepnope works:

How do I know what order my files are going to execute in? Can I force a

In short, whatever order you put them in, that's the order that we execute them in. The '**`load`**' and '**`both`**' set '**`yep`**' or '**`nope`**' sets, but the order that you specificy within those sets is also preserved. This doesn't mean order, but we guarantee that they **execute** in this order. Since this is an asynchronous loader, we load ever we just delay running it (or injecting it) until the time is just right.

What happens if I use yepnope inside of a yepnope callback?

This is actually an advanced feature of yepnope. We call it 'recursive yepnope'. It actually places the files of correct place in the outer yepnope call. So the next file in the outer yepnope call will wait until the inner yepn before it executes. This is a great feature to use as a fallback enabler. We encourage you to use it with cau script loading since we can't start preloading the files until the callback occurs (serial loading instead of par option, it's nice. Here is an example of how it could be used as a fallback technique.

```
yepnope([{
  // Load jquery from a 3rd party CDN
  load: 'http://code.jquery.com/jquery-1.5.0.js',
  callback: function (url, result, key) {
```

```
    // The boss doesn't trust the jQuery CDN, so you have to have
    // So here you can check if your file really loaded (since cal
    // fire after an error or a timeout)
    if (!window.jQuery) {
      // Load jQuery from our local server
      // Inject it into the middle of our order of scripts to exec
      // even if other scripts are listed after this one, and are
      // done loading.
      yepnope('local/js/jquery15.js');
    }
  }
}, {
  // This file will start downloading as soon as this line is exec
  // but if the jQuery CDN was down it won't be executed until aft
  // local version was loaded.
  load: 'jQuery.plugins.js',
  callback: function (url, result, key) {
    jQuery('.fun').plugin();
  }
}]);
```

Depending on your browser, when there's an error, your callbacks may fire back immediately, but in other b
between a successful script load and an error state, so we opt not to expose potentially wrong information.
unavailable, there is a maximum timeout that you can set for how long it will wait before just calling back any
`yepnope.errorTimeout`.

I'm seeing two requests in my dev tools, why is it loading everything twi

Depending on your browser and your server this could mean a couple different things. Due to the nature of l
two requests made for every file. The first request is to load the resource into the cache and the second req
in the cache, it should execute immediately). Seeing two requests is pretty normal as long as the second re
the second request isn't cached (and your script load times are doubling), then make sure you are sending
allow the caching of your scripts. This is **vital** to yepnope. It will not work without proper caching enabled. W
things aren't loaded twice in our test suite, so if you think we may have a bug in your browser regarding dou
run the test suite to see if the double loading test passes.

Yepnope Plugins!

Yepnope was designed to be customizable. There are a few simple hooks throughout the loading process t change settings around on the fly. There are two main ways to do this: Prefixes and Filters.

Prefixes and Filters do the same thing, except that a prefix is only applied to scripts that explicitly have the r global modication of every script that goes through yepnope. Prefixes are added to files individually by prep name and separating them with a ! character. Examples to follow.

Yepnope officially supports a few prefixes and filters.

In no specific order, they are:

css! Prefix

The css! prefix is for those people who are dynamically generating css, or have the need to use query string on their CSS files. By default, yepnope assumes that any files that end with a '.css' are CSS files, but it also everything else is a JavaScript file. With the css! prefix, you can prepend it to any file name and yepnope wi

```
yepnope('css!mystyles.php?version=1532'); // loads as a style!
```

preload! Prefix

The preload! prefix does pretty much what it says it does. When used on any file, it loads the file without exe when the file has been completely loaded. If proper cache headers have been set up, these files will be cac be potentially beneficial as a technique to load future resources and prime the caches of your users at your

```
yepnope({
  load: 'preload!jquery.1.5.0.js',
  callback: function (url, result, key) {
    window.jQuery; // undefined (but it's cached!);
  }
});
```

Please note that the only types of things that are tested for the preload plugin are script and javascript files, a few other types.

ie! Prefix(es)

While everyone knows that sniffing for browsers is bad practice, sometimes it's necessary. This plugin allov related prefixes to only load content for the dreaded IE browser of your choosing.

Here is a list of supported prefixes in the IE package:

`ie`, `ie5`, `ie6`, `ie7`, `ie8`, `ie9`, `iegt5`, `iegt6`, `iegt7`, `iegt8`, `ielt7`, `ielt8`, and `ielt9`

The function of these should be pretty self-explanatory (hopefully), but if it's not clear: ie followed by a single script if the IE version matches that number. `iegt` is for 'versions greater than' x, and `ielt` is for 'versions allow for any Internet Explorer version.

Since you can chain multiple prefixes, these prefixes work together to perform a logical OR. So if any of you resource will load (assuming there isn't something else that's stopping it).

```
yepnope({
  load: ['normal.js', 'ie6!ie7!ie-patch.js'] // ie6 or ie7 only (
});
```

Adding Your Own Prefixes and Filters

Yepnope allows you to add your own custom prefixes and filters.

There are two functions on the yepnope object that allow you to add these: `addFilter` and `addPrefix`. them, is that '`addPrefix`' takes a name (string) as the first value, and the filter has no name, so only takes

The function that gets run is passed a resource object that describes things about each individual file that g this function you should modify it to your choosing, and then return it. Consider it middleware if you believe i

The resource object has the following properties that you are allowed to modify:

- `url` - The url that is injected
- `prefixes` - the array of applied prefixes
- `autoCallback` - a callback function that runs after each script load separately from the others
- `noexec` - boolean to force a preload, but no execution
- `instead` - a function that takes all the same parameters as our internal loader (advanced)
- `forceJS` - boolean force the file to be treated as a JS file
- `forceCSS` - boolean for force the file to be treated as a CSS file
- `bypass` - boolean for do or don't load the file

Here's a somewhat silly example:

```
yepnope.addPrefix('bypass', function (resourceObj) {
  resourceObj.bypass = true;

  return resourceObj;
});
```

This will cause any file with the **`bypass!`** prefix to not load or execute.

A bypass Filter would look almost exactly the same, except it would apply to all files, and you don't need to p

```
yepnope.addFilter(function (resourceObj) {
  resourceObj.bypass = true;

  return resourceObj;
});
```

Each function passed in is required to send back the request object (whether modified or not).

Errors and Timeouts

Yepnope (or any other script loader that we know of) cannot accurately report script loading errors in a cro reason, we have a good amount of code in place to ensure that errors that we can detect fail quickly (so you and that errors we cannot detect are automatically called back after the amount of time that's specified in th property. The default for yepnope is 10,000 milliseconds (or 10 seconds for all you humans). At any point i and any script loaded after it's change will obey it's timeout length. Different organizations tend to require di your liking before making any requests.

example:

```
yepnope.errorTimeout = 4000; // set to 4 second error timeout
```

Check out the section on fallback loading if you have any questions on how to set it up.

COMMON GOTCHAS

- You cannot use **`document.write()`** (which means no google maps or ads) in the scripts that you loa

every asynchronous script loader. We suggest that you avoid document.writes **all the time** though.

- IEs less than 9 don't guarantee that the callbacks for scripts run **immediately** after the related script exe but they are not universally applicable to all scripts. Please see Julian Aubourg's blog post on this subje (http://jaubourg.net/archive/7/2010) if you you'd like to learn more information.
- Just because your script is done, doesn't mean the document is ready. Don't forget that you can use doc your yepnope callbacks. If you're toying with the DOM, we'd heavily encourage you to do so, because yo differently than your production server the speeds are dramatically different.
- Loading too many scripts can be worse than not using yepnope. Yepnope is not to be used as an excus styles into single files. This is still a great practice. We just don't want you to group it into ***one***.
- Old IEs can only load 2 things at a time from the same domain, others only 6. This means that if you are the same domain, you might consider getting some subdomains and serving your files from those in ord

---

Brought to you by

Alex Sexton

yayQuery (//yayquery.com) / Modernizr (//modernizr.com) / Bazaarvoice (//bazaarvoice.com)
@slexaxton (//twitter.com/slexaxton)

Ralph Holzmann

jQuery enthusiast (http://docs.jquery.com/Plu
Groupcard (//groupcard.co
@ralphholzmann (//twitter.com/ralphholz
