net.tutsplus.com

An Introduction to YepNope.js

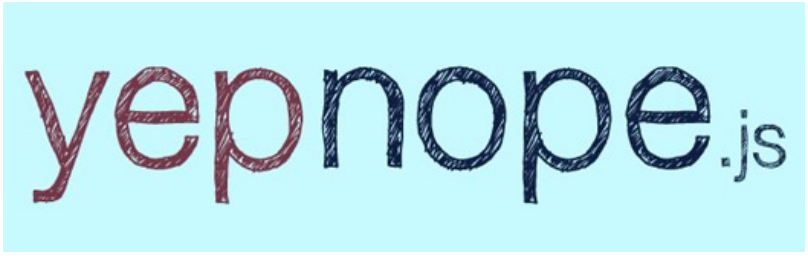
Officially released by Alex Sexton and Ralph Holzmann in late February of 2011, the yepnope.js resource loader features asynchronous, conditional loading and preloading of both JavaScript and CSS resources. Although yepnope downloads in parallel, it maintains the order in which scripts are executed and provides a callback for each script's load event as well, as a global callback which fires once everything is loaded. This makes managing dependant, conditional code a breeze.

This nifty resource loader, which is only 1.6KB minified and gzipped, is now bundled with Modernizer and is great for loading polyfills, preloading or "priming" the users cache, or as a simple asynchronous resources loader / filter!

> For those of you unfamiliar with polyfills, they are essentially plugins, or shims, that enable the use of new or future technologies in older browsers, e.g. web sql databases, CSS3 transformations etc.

Yepnope now also supports a number of prefixes and filters, which, when prepended to the resource url, add another layer of fine tuning or customization to its core functionality. As if this weren't already great, yepnope also provides you with a mechanism to define your own prefixes and filters. Let's have a look at what yepnope.js can do!

Background – Asynchronous Script Loading



Before we delve into yepnope and its features, it's important to understand a bit about how asynchronous script loading works, why it's useful and how it's different from vanilla script loading.

> Asynchronous loaders remove the inherent blocking nature of a script.

Typically, JavaScript files loaded with the `<script>` tag, block the download of resources as well as the rendering of elements within the web page. So, even though most modern browsers tend to support the parallel download of JavaScript files, image downloads and page rendering still have to wait for the scripts to finish loading. In turn, the amount of time a user has to wait for the page to display increases.

This is where asynchronous loaders come in to play. Using one of several different load techniques, they remove the inherient blocking nature of a script, which allows for parallel downloading of both the JavaScripts and resources while not interfering with page rendering. In many cases, this can reduce – sometimes drastically – page load times.

> Most loaders preserve the order in which scripts are executed while providing a callback for when the script is loaded and ready.

Asynchronous loading doesn't come without its caveats, though. When scripts are loaded the traditional way, inline code is not parsed or executed until the external scripts are fully loaded, sequentially. This is not the case with asynchronous loading. In fact, inline scripts will usually parse / execute **while** the scripts are still being downloaded. In like manner, the browser is also downloading resources and rendering the page as the scripts are being loaded. Thus, we can arrive at situations where inline code, which is perhaps dependant on a script / library being loaded, is executed before its dependency is ready or before / after the DOM itself is ready. As such, most loaders preserve the order in which scripts are executed while providing a callback for when the script is loaded and ready. This allows us to run any dependant inline code as a callback, perhaps, within a DOM ready wrapper, where applicable.

Also, when dealing with a small or well optimized page, the DOM can actually be ready or even loaded before the scripts themselves have finished loading! So, if the page in question isn't progressively enhanced, in that it relies heavily on JavaScript for styling, there may be a FOUC or flash of unstyled content. Similarly, users may even experience a brief FUBC or flash of unbehaviored content. It's important to keep these things in mind whenever you use a script / resource loader.

Step 1 – The `yepnope` Test Object

The `yepnope` test object has seven basic properties, any of which are optional. This object includes the actual test, resources which will be loaded as a result of the test, resources which will be loaded regardless of the test as well as callbacks. Here's a look at the yepnope test object's props:

- **test:**

  A boolean representing the condition we want to test.

- **yep:**

  A string or an array / object of strings representing the url's of resources to load if the test is *truthy*.

- **nope:**

  A string or an array / object of strings representing the url's of resources to load if the test is *falsey*.

- **load:**

  A string or an array / object of strings representing the url's of resources to load regardless of the test result.

- **both:**

  A string or an array / object of strings representing the url's of resources to load regardless of the test result. This is, basically, syntactic sugar as its function is generally the same as the `load` function.

- **callback:**

A function which will be called for **each** resource as it is loaded sequentially.

- **complete:**

  A function which will be called **once** when all of the resources have been loaded.

Now, to get an idea of the syntax, let's take a look at the simplest possible use of yepnope: loading a single resource.

```
yepnope('resources/someScript.js');
```

… or perhaps loading an array of resources.

```
yepnope([
        'resources/someScript.js',
        'resources/someStyleSheet.css'
]);
```

How about an object literal so that we can use named callbacks later?

```
yepnope({
        'someScript'     : 'resources/someScript.js',
        'someStyleSheet' : 'resources/someStyleSheet.css'
});
```

Remember, these resources will be loaded asynchronously as the page is downloading and rendering.

Step 2 – Conditions – Testing for the Features of the Future!

So, we can load resources asynchronously! That's great, but, what if some pages don't require a certain resource? Or, what if a resource is only needed in a particular browser which doesn't support a cutting edge new technology?

No problem! This is where yepnope's underlying purpose comes into focus. Using the test property, we can conditionally load resources based on need. For example, let's assume that the Modernizer library is loaded.

> For those of you unfamiliar with Modernizer, it's a nifty test suite used for detecting HTML5 and CSS3 feature support in browsers.

Modernizer adds appropriate classnames to the pages `html` element, representing the features supported and not supported, e.g. "`js flexbox no-canvas`" etc. Additionally, you can access each of Modernizer tests, which return boolean values, individually, within your code.

So, using Modernizer, let's test for `hashchange` event support as well as session history support!

Here's a look at our test:

```
yepnope({
        test : Modernizr.hashchange && Modernizr.history
});
```

This test will, of course, return `true` only if the browser supports both of these features.

Step 3 – Loading Resources Conditionally

With our test condition set, we'll now define which resources to load based on the result of this test. In other words, if you only need to load a specific resource when the browser lacks a feature, or the test fails, you can simply define that resource in the `nope` clause. Conversely, you can load resources when the test passes, within the `yep` clause.

So, assuming the browser doesn't support one of these two features, we'll load up Ben Alman's jQuery hashchange plugin, which enables `hashchange` and history support in older browsers which don't support either of these features.

Let's load up the hashchange plugin:

```
yepnope({
        test : Modernizr.hashchange && Modernizr.history,
        nope : 'resources/jquery.ba-hashchange.js'
});
```

In the above example, we won't use the `yep` property as we're only providing a shim should it be needed.

To illustrate the `yep` clause, though, let's test for CSS3 transformation support and then load a stylesheet for browsers which support transformations and a vanilla stylesheet for browsers that don't. Additionally, we'll load a jQuery plugin which mimics CSS3 transformations as well.

Using both yep and nope:

```
yepnope({
        test : Modernizr.csstransforms,
        yep      : 'resources/cssTransform.css'
        nope : ['resources/noTransform.css', 'jQuery.pseudoTransforms.js']
});
```

Note that both of these examples will load all resources asynchronously as the rest of the page downloads and renders!

Step 4 – Loading Resources Regardless of the Test Condition

Yepnope also provides a way to load resources independently of the test results by way of the the `load` property. The `load` function will always load any resource it's fed, regardless of the `test` result. Similarly, the `both` prop, which is, again, essentially just syntactic sugar, also loads resources regardless of the test result, or more accurately, on either result.

Loading by default:

```
yepnope({
        test : Modernizr.hashchange && Modernizr.history,
        nope : 'resources/jquery.ba-hashchange.js',
        load : 'resources/somethingWhichIsAlwaysLoaded.css',
});
```

Loading on both conditions, syntactic sugar :

```
yepnope({
        test : Modernizr.hashchange && Modernizr.history,
        nope : 'resources/jquery.ba-hashchange.js',
        both : 'resources/somethingWhichIsAlwaysLoaded.css',
});
```

In both of the above examples, resources will be loaded, asynchronously, no matter what.

Step 5 – Callbacks – Dependent Code After the Load

As mentioned earlier, we can't write in-line code in the usual manner if that code is dependant on one of the scripts being loaded. Thus, we'll use yepnope's callback function which fires once for each resource *after* it has finished loading. The callback function accepts three parameters which are assigned the following:

- **url**

  This string represent the url of the resource which was loaded

- **result**

  A boolean representing the status of the load.

- **key**

  If using an array or object of resources, this will represent the index or the property name of the file which was loaded

Let's take a look at a simple callback with the hashchange plugin example from earlier. We'll use jQuery's bind method to bind a handler to the hashchange event of the `window`:

A simple callback:

```
yepnope({
        test : Modernizr.hashchange && Modernizr.history,
        nope : 'resources/jquery.ba-hashchange.js',
        callback : function(url, result, key){

                $(function(){
                        $(window).bind('hashchange', function(){
                                console.info(location.hash);
                        });
                });

        },
});
```

Regardless of what state the DOM is in, this callback, which in this particular case is within a document ready wrapper, will fire as soon as the resource is loaded.

Let's say, however, that we are loading more than one script and that we need to fire off a callback for each script as it loads. Specifying the code we need to run in the above manner would create a redundancy as the callback is fired each time a resource is loaded. Yepnope, however, provides a great way to handle callbacks for each resource, independently of any other callbacks.

> By using an object literal to define the resources we are loading, we can reference each resource key, individually, within the callback.

Let's take a look at an example where we load jQuery as well as the jQuery hashchange plugin, which is dependant on jQuery being loaded first. This time, however, we'll use object literals!

```
yepnope({
        test : Modernizr.hashchange && Modernizr.history,
        nope : {
                'jquery' : 'resources/jquery-1.5.1.min.js',
                'hashch' : 'resources/jquery.ba-hashchange.js'
        },
        callback : {
                'jquery' : function(url, result, key){
                        console.info('I will fire only when the jquery script is loade
                },
                'hashch' : function(url, result, key){
                        console.info('I will fire only when the hashchange script is l

                        // This code will be added to jQuerys DOM ready call stack
                        $(function(){
                                $(window).bind('hashchange', function(){
                                        console.info(location.hash);
                                });
                        });
                }
        }
});
```

Using the above example as a reference, you can implement your own callbacks for each resource load in an orderly manner.

Step 6 – Complete – When all is Said and Done!

Lastly, we have the `complete` callback which is only called once, after all the resources have finished loading. So, for example, if you're "bootstrapping" a web application and the code you need to run is dependent on all the files you're loading, rather than specifying a `callback` for each resource, you would write your code within the `complete` callback so that it's only fired once, after all its dependencies have loaded. Unlike the `callback` function, `complete` doesn't take any parameters or have access to the `url`, `result` or `key` props.

The `complete` callback:

```
yepnope({
        test : Modernizr.hashchange && Modernizr.history,
        nope : [
                'resources/jquery-1.5.1.min.js',
                'resources/jquery.ba-hashchange.js'
        ],
        complete : function(){

                console.info('I will fire only once when both jquery and the hashchang

                // This code will be added to jQuerys DOM ready call stack
                $(function(){
                        $(window).bind('hashchange', function(){
                                console.info(location.hash);
                        });
                });

        }
});
```

So, essentially, the `complete` callback is useful for anything which needs to be done once all the resources are loaded.

Step 7 – Yepnope Plugins, Prefixes and More!

Yepnope also provides us with another nifty little feature: prefixes and filters! The default prefixes provided by yepnope, which are always prepended to the beginning of a resource url, are used for defining a file as CSS, preloading a resource or targeting Internet Explorer or one of its versions, respectively. Let's have a look:

- **css!**

  This prefix is used for forcing yepnope to treat a resource as a stylesheet. By default, yepnope treats .css files as stylesheets and everything else as a JavaScript file. So, if you're serving up CSS dynamically, this prefix would force yepnope to treat that resource as a stylesheet.

  ```
  yepnope('css!styles.php?colorscheme=blue');
  ```

- **preload!**

  This prefix allows you to load / cache a resource without executing it.

  ```
  yepnope('preload!userInterface.js');
  ```

- **ie!**

There may be circumstances where you need to load specific resources only if you're working with Internet Explorer or a particular version of Internet Explorer. Thus, the `ie` prefixes help you target resource loading to `ie` or specific versions of it. Here's a list of the supported `ie` prefixes where `gt` stands for "versions greater than" and `lt` stands for "versions less than".

- **Internet Explorer:**
  `ie!`
- **Internet Explorer by version number:**
  `ie5!`, `ie6!`, `ie7!`, `ie8!`, `ie9!`
- **Internet Explorer versions greater than:**
  `iegt5!`, `iegt6!`, `iegt7!`, `iegt8!`
- **Internet Explorer versions less than:**
  `ielt7!`, `ielt8!`, `ielt9!`

All of these filters are chainable and serve as a sort of `OR` operator in that if one of them evaluates to `true` the resource will be loaded. So, should we need to target ie7 and ie8, we would simply prepend the appropriate filters to the url of the resource as follows:

```
yepnope('ie7!ie8!userInterface.js');
```

Creating your own filters!

Should you ever need to, `yepnope` also provides the means with which to create your own filters and prefixes by way of the `addFilter` and `addPrefix` methods. Any filter or prefix you create is passed a `resourceObject` containing a number of useful props. Remember, though, to return the `resourceObject` as yepnope requires that you do so. Here's a look at the `resourceObject`:

- **url:**

  The url of the resource being loaded.

- **prefixes**

  The array of applied prefixes.

- **autoCallback**

  A callback which runs after each script loads, separate from the others.

- **noexec**

  A boolean value which forces preload without execution.

- **instead**

  An advanced function which takes the same parameters as the loader.

- **forceJS**

A boolean which forces the resource to be treated as javascript.

- **forceCSS**

  A boolean which forces the resource to be treated as a stylesheet.

- **bypass**

  A boolean which determines whether or not load the current resource

Let's say, for example, you want the ability to toggle resource loading between your CDN and web server, on the fly. Can we do that, though!? Yep! Let's create two prefixes, one for loading from the CDN and the other for loading from your web server.

```
yepnope.addPrefix('local', function(resourceObj) {

        resourceObj.url = 'http://mySite/resources/' + resourceObj.url;
        return resourceObj;

});

yepnope.addPrefix('amazon', function(resourceObj) {

        resourceObj.url = 'http://pseudoRepository.s3.amazonaws.com/' + resourceObj.ur
        return resourceObj;

});
```

Using these prefixes, we can now easily switch between our CDN and web server!

```
yepnope([
        'local!css/typography.css',
        'amazon!defaultStyle.css'
]);
```

Step 8 – A few Caveats

So, while maintaining a very small footprint, the yepnope conditional loader is power-packed with a number of useful features! There are, however, a few things you should be aware of before using it.

- **No `document.write`**

  As with any asynchronous loader, you can't use `document.write`.

- **Internet Explorer less than 9 and callback execution**

  Internet Explorer versions less than nine don't guarantee that callbacks run *immediately* after the related script fires.

- **Be careful with the DOM**

  Your script may be loaded and executed before the DOM is ready. So, if you're manipulating the DOM, it's advisable to use a DOM ready wrapper.

- **You should still combine where you can**

  Just because you're using an asynchronous loader doesn't mean that you shouldn't combine your resources where you can.

- **Internet Explorer asynchronous load limits**

  Older versions of Internet Explorer can only load two resources from the same domain at the same time, where as other versions can load up to six. So, if you're loading multiple files, consider using a sub-domain or CDN.

Conclusion – Thoughts on yepnope.js

All in all, I found yepnope to be a great utility! Not only does it support the asynchronous loading of both scripts and stylesheets, but it provides you with a nice, clean way to load HTML5 and CSS3 polyfills conditionally. The callback mechanism is well thought out and the ability to add your own prefixes and filters is just great! Performance wise, I found yepnope to be somewhat on par with other loaders, such Getify Solutions' LABjs and James Burke's require.js. Obviously, each loader is different and suits a different need but if you haven't yet, I encourage you to give yepnope.js a go!