year of moo

View Archive View Tags
Search for Programming, AngularJS, Rails, Testing ...

AngularJS and SEO

Turns out it is possible to have your AngularJS application indexed

AngularJS and just about all JavaScript MVC frameworks modify the contents of your HTML structure which make the pre-rendered HTML invalid for search engines. Luckily there is a way to get around this and to have full SEO support for your AngularJS application by using some special URL routing and a headless browser to retrieve the HTML for you.

Continue reading this article to figure out how to make this amazing framework work well with your favourite search engines.

Last Updated

This page was first published on November 7th 2012 and was last updated on January 21st 2013.

Download the Repo

To best follow along with this article, be sure to clone the repo, yearofmoo/AngularJS-SEO-Article and look through the code and compare it to what's on the blog to get a better idea of how everything works. Here's the command to do that.

```
git clone git://github.com/yearofmoo-articles/AngularJS-SEO-Article.git
cd ./AngularJS-SEO-Article
```

If you're using apache, then modify the .htaccess file to the path of where it is on your machine and then you can access the website using http://localhost/path/to/this/repo (or a vhost if you wish). For me, the vhost that I prepared looked like this:

```
<VirtualHost *:80>
  DocumentRoot "/path/to/this/article"
  ServerName yom-angularjs-seo-article
</VirtualHost>
```

You will also need to set the matching ServerName value in your /etc/hosts file (or your HOSTS file in Windows).

If you do not use a vhost then be sure to set the RewriteBase value in the .htaccess file (which is found in the repository) to that of where you have placed this demo repository.

How Google and Bing index AJAX Applications

Google and Bing both support hashbang URLs which are used to inform the search engine that the current webpage being accessed at the given URL contains AJAX content. Below is an example of a website that contains a hashbang within a URL:

```
http://yourwebsite.com/#!/some/page/with/ajax/content
```

Google will now convert this URL to a special URL which is designed to be accessible by the server. Google will then visit the URL and expects the content retrieved from that URL to be the final, fully generated content for that page. In other words Google expects to get the same HTML that is fetched from that URL to that of what your browser would contain in it's DOM for your AJAX application once it's run on that page. Here is what the final URL that Google/Bing queries to get that content:

```
http://yourwebsite.com/?_escaped_fragment_=/some/page/with/ajax/content
```

So now it's up to you to route this request to the pre-generated HTML version of your webpage.

Now that you understand how to make any AJAX application indexable then how do you do this with AngularJS? Well AngularJS is designed for the client side so you will need to configure your web server to summon up a headless HTML browser to access your webpage and spit out the HTML for that given hashbang URL which is delievered within the special Google URL.

Click here to read more about AJAX Indexing

Detecting the Search Engine

As stated above, whenever a page is addressed with a hashbang (or a meta fragment tag) then Google and Bing will access the website with a special querystring prefix (the one shown above). So to get your application to detect this then you will need to setup a special URL rewrite/route to take advantage of it.

Here is an example of this using mod_rewrite with Apache:

```
RewriteEngine On
RewriteCond %{REQUEST_URI}  ^/$
RewriteCond %{QUERY_STRING} ^_escaped_fragment_=/?(.*)$
RewriteRule ^(.*)$ /snapshots/%1? [NC,L]
```

And here's an example of this using NGINX:

```
server {
  # ... your config for your server daemon
  # listen 80;
  # server_name  localhost;
  # root   /path/to/root/
  # index  index.html;
  if ($args ~ "_escaped_fragment_=/?(.+)") {
    set $path $1;
    rewrite ^ /snapshots/$path;
  }
}
```

This will redirect any URLs that contain the querysting ?_escaped_fragment_= directly to it's matching file found within the snapshots directory. It is up to you to make sure that the file exists within the directory. Details on how to do this will be

covered shortly later on in the article.

Hashbangs or HTML5 History URLs?

If you use hashbang URLs then all you need to do is instruct AngularJS to use them instead of regular hash values:

```
angular.module('HashBangURLs', []).config(['$locationProvider', function($location) {
  $location.hashPrefix('!');
}]);
//then add this module as a dependency within your primary module(s).
```

Google will now recognize the URL and then do what it has to do to fetch the HTML version.

If you wish not to use hashbang URLs but still inform Google that your website contains AJAX content then include this meta tag within the head tag of the file being accessed at the given URL.

```
<meta name="fragment" content="!" />
```

Then configure AngularJS to use HTML5 URLs when handling URLs and routes:

```
//get the module from creating an angular module
angular.module('HTML5ModeURLs', []).config(['$routeProvider', function($route) {
  $route.html5Mode(true);
}]);
```

Then whichever method you choose can be installed via module:

```
//get the module from creating an angular module
var App = angular.module('App', ['HashBangURLs']);
//or
var App = angular.module('App', ['HTML5ModeURLs']);
```

My suggestion is to stick to using hashbang URLs since you can mix both non-AJAX pages (like basic forms and so on) to work with non-hashbang HTTP URLs (like registration pages, forms, etc...).

Spitting out the HTML Pages

You will need to have a headless browser access the regular URL of the page and grab it's content when it's ready. A good option is to use PhantomJS to download the content of that page, run the JavaScript, and then spit out the HTML into a temporary file once it's ready.

But how do you know if your webpage is ready and the HTML returned from the headless browser is the final HTML for that given page? Well you need to set a flag in the DOM of your webpage that lets the headless browser know that it's ready. You can also wait for all JavaScript to be executed, but the problem here is that you do not know when any remote XHR requests have completed their job, nor do you know if there are any important asynchronous scripts going on in the background.

```
<!-- set your body tag to have this attribute -->
<body data-status="{{ status }}">
```

And then add this to the end of each of your controllers:

```
//...something happens in the controller and then...
$scope.status = 'ready';
```

Now have your headless browser poll for the status attribute within the page to see if it changes to ready and then spit out the data to the file. In this case PhantomJS doesn't look like it can write to files (I couldn't find it anywhere) so there is a bash file which does the fetching and writing.

```
# run this from the command line at the root of the repository
./make-snapshot.sh http://your-website/#!/home
# this will create the file inside of the snapshots/ directory
```

The .phantomjs-runner.js (which is provided within the github repo) takes in any URL as input, downloads and parses the HTML into it's DOM and then polls the body tag to see if the data-status attribute is set to "ready". The HTML content is then redirected (saved) to the filename maching it's path within the snapshots folder. Keep in mind that this is all subjective to the design of the example github repo, so changing the directory and/or filenames is up to you.

Test your Post JavaScript HTML Pages Properly

The major challenge with this approach is that your JavaScript code has be fully functional for all pages of your application that are AJAX-indexed pages. This means that if you have any plugin or snippet of code that fails to run then it may cause issues for your website SEO. The best option is to have a unit test for each page of your application to see if it loads properly. This can be done very easily by polling the data-status HTML attribute, which was mentioned earlier, to check see if it fires for each page. If it doesn't pass or execute, or an error occurs, then it's likely that that given page contains some sort or JavaScript error or the route itself is invalid or has been changed or there are pending XHR calls going on in your controller(s).

Here is a example of a JavaScript unit testing framework called Mocha which can be used to test each page directly at it's route path.

```
//
// All the code (including the helper methods) can be found at
// https://github.com/yearofmoo-articles/AngularJS-SEO-Article/blob/master/test/spec/pages_spec.js
//
describe('Testing all pages to see if they load properly', function() {

  beforeEach(function() {
    _resetVars();
  });

  it('GET /home.html', function(done) {
    expect(_isScopeReset()).to.equal(true);
    _checkIfLoaded('/home.html', done);
  });

  it('GET /videos.html', function(done) {
    expect(_isScopeReset()).to.equal(true);
    _checkIfLoaded('/videos.html', done);
```

```
  });

});
```

[Click here to view the full code for this test spec]

Now you can run the code by accessing the page by visiting the /test/ path. I will explain how to set this up with a command-line testing tool called Testacular in a very near future article.

Beware of DOM-altering Plugins

It is best to avoid or disable the use of any interactive plugins that change the DOM of the webpage when extracting out the HTML from the DOM for your snapshot files. The reason why is that some of these plugins may add/remove important markup or even distort existing markup and text for their own purpose. Or when your HTML is extracted, and the plugin is still doing it's work, then you may end up with a partial snapshot of the HTML within that given area. An example of a useful plugin that does change the DOM around is Cufon.

Here is an example of text that includes a heading:

```
<h2>Hello World</h2>
```

And after cufon does it's magic on the text then the HTML of that H2 element looks like so:

```
<h2>
  <cufon class="cufon cufon-canvas" alt="Hello " style="width: 92px; height: 32px; ">
    <canvas width="117" height="37" style="width: 117px; height: 37px; top: -4px; left: -2px; "></canvas>
    <cufontext>Hello </cufontext>
  </cufon>
  <cufon class="cufon cufon-canvas" alt="World" style="width: 93px; height: 32px; ">
    <canvas width="111" height="37" style="width: 111px; height: 37px; top: -4px; left: -2px; "></canvas>
    <cufontext>World</cufontext>
  </cufon>
</h2>
```

As you can see, the HTML for that simple heading becomes a lot more complex with cufon and a search engine would not make use of the HTML. So it's best to turn it off when fetching DOM HTML.

Keeping everything in Sync

Now that you understand how SEO works with AJAX applications, the next step is to keep everything in sync. You wouldn't want your static HTML to be an older version of the data served from your website. Since this is a static website (the one in the github repo) there isn't any automated synchronization going on. Therefore, If you're using a server-side programming language and/or framework then the logical solution to this is to have your application try to check if the snapshot is available and, if not, then recreate the HTML file directly and save it's contents to the snapshot folder with the path being the same as the path to access the file normally. Then when any content changes on your webserver (due to new uploads, database changes and/or new content) then issue a cache sweep for any of the files that are related to the cache that are present within the snapshots directory. This way, if Google, Bing and/or any other Search Engines visit these files then they will be fast and up to date. Having static

files also prevents any bottlenecks as well as various security holes in your application since the server won't hang each time a snapshot path is being queried.

Make a Sitemap as well

Another great idea is to create a sitemap for the website. Within your website, use the hashbang or HTML5 URLS to point to the content--Google and Bing are smart enough to figure out that they're AJAX-generated pages.

Here is an example of a sitemap.xml entry for a AJAX webpage.

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  ...
  <url>
    <loc>http://www.yourwebsite.com/#!/home</loc>
    <changefreq>daily</changefreq>
    <priority>1.0</priority>
  </url>
  ...
</urlset>
```

The example Github repo also includes a sitemap.xml. You will, however, more than likely have a server-side script spit out a sitemap.xml file, but this is here for you to get an idea of how to further improve the success your website being indexed effectively.

This is Great!

Understanding how to make an AngularJS application indexable by Google and Bing just about eliminates all the reason not to use AngularJS for just about any web application out there. Before the major concern was HTML, but now with the solution of AJAX indexed content, you can do just about anything :)

Truly amazing! Thank you Google/Bing/AngularJS :)