SEO for Single Page Applications

Matthias Kunnen
Feb 13, 2018 · 4 min read

Single Page Applications, commonly referred to as SPAs, are a way to reduce page load and increase user experience. This is achieved by not reloading the page. Absence of page reload ensures that, after the initial load, only data is send over the wire. The result is a drastic decrease in time to navigate and display data.

So, why isn't everyone using this? A major disadvantage of SPAs is poor SEO. SPAs depend on JavaScript to render content and navigate. However, the programs that index websites for search engines, a.k.a. spiders or crawlers, are not really keen on JavaScript. Some spiders, such as Googlebot, render some JavaScript. Out of experience, however, our team has learned that support is limited.

Our website, Forcit.be, is a Single Page Application written in Angular 4. After some analysis, our team found out that Googlebot was poorly crawling the website. I would like to share some methods for SEO on SPAs with you. We will take a look at Angular 4 in particular followed by a general method of providing indexable pages.

SEO in Angular 4

Server-side rendering

Angular has its own way of dealing with SEO called Angular Universal. This technique pre-renders HTML on the server and transmits it to the client. Server-side rendering ensures that the spider crawling your page receives old-fashioned HTML, which they are build to handle.

There are some disadvantages to this method:

- The server hosting your website should have Node enabled. Most hosting providers don't offer Node yet.

- Checks on the environment should be made before accessing the window, document and navigator variables as these variables only exist in the browser.

There were some problems implementing this into our website. Therefore, and for the sake of experimentation, I started my search for a method to provide indexable pages.

A general method for indexable pages

The requirements for my solution were:

- No changes to the current codebase
- Low load & response time for serving the spiders
- And last but not least, serve an easily indexable page

Earlier in this post I have noted that spiders are build to process HTML. Therefore, the decision to serve static HTML was easily made. I had to serve static HTML to the spiders and present the SPA to normal users.

Identifying spiders

Making a distinction between spiders and normal users can be done via an HTTP header that is sent along with requests called the *user agent*. This string is used by the spider to ID itself. Examples of spider's *user agent* are:

- Google: googlebot
- Bing: bingbot
- DuckDuckGo: duckduckbot
- Facebook: facebookexternalhit
- …

Generating static HTML

Generating the static HTML was without a doubt the most difficult part of this approach. I explored multiple libraries such as npm-snapshot and html-snapshot, … All, to no avail. The render libraries didn't render the page correctly and the snapshot libraries didn't wait for page completion or outright errored on executing the JavaScript.

I was feeling a little bit worried about this endeavor until I decided to look into headless browsers. A headless browser behaves like a normal browser, but does not have a graphical user interface. While this makes it pointless for normal users, this is ideal for testing as the processing overhead for rendering the view is absent.

With renewed courage I found Puppeteer, a Node library to control headless Chrome. This library was exactly what I needed. It allows surfing to a page and extracting its HTML after the SPA finished rendering. All you need is the routes to all your pages.

The code beneath visits all supplied routes and stores them into the following tree.

```
Routes
[
    "",
    "approaches",
    "contact",
    "projects",
    "approach/ideation",
    "approach/prototyping",
    "approach/mvp",
    "approach/full-product",
    "project/x",
    "project/y",
]

Generated tree
├── approach
│   ├── full-product.html
│   ├── ideation.html
│   ├── mvp.html
│   └── prototyping.html
├── approaches.html
├── contact.html
├── index.html
├── project
│   ├── x.html
│   └── y.html
└── projects.html
```

```
import * as cheerio from 'cheerio';
import * as fs from 'fs';
import * as mkdirp from 'mkdirp';
import * as path from 'path';
import * as puppeteer from 'puppeteer';

import { routes } from './routes'; // Array of strings representing routes

```

```
const host = 'https://forcit.be/';

function writeFile(filePath: string, contents: any, cb: (err) => void) {
    mkdirp(path.dirname(filePath), err => {
        if (err) {
            return cb(err);
        }

        fs.writeFile(filePath, contents, cb);
    });
}

(async () => {
    const browser = await puppeteer.launch();

    for (const route of routes) {
        const fullRoute = host + route;
        console.log(`Statisfying ${fullRoute}`);

        let html: any; // Type coercion problem otherwise
        let success = false;
        let tryCounter = 0;

        while (!success && tryCounter <= 3) {
            try {
                const page = await browser.newPage();
                await page.goto(fullRoute);
                html = await page.evaluate(() => document.documentElement.outerHTML);
                await page.close();
                success = true;
            } catch (e) {
                console.warn(`Could not evaluate ${fullRoute} in try ${tryCounter++}.`)
                console.warn(`Error: ${e}`);
            }
        }

        if (!success) {
            console.error(`Could not evaluate ${fullRoute} in ${tryCounter} tries.`);
            continue;
        }

        writeFile(`static/static/${route ? route : 'index'}.html`, html, err => {
            if (err) {
                console.log(err);
                process.exit(1);
            }
        });
    }

    await browser.close();
})();
```

build-static.ts hosted with ❤ by GitHub view raw

Getting a HTML snapshot via puppeteer

We now have an HTML file for each route of our application. However, we are not done. The HTML we extracted still contains the SPA's bootstrap code. This code needs to be removed for the page to be static as it can try to bootstrap but fail, leaving us a blank page.

Removing bootstrap code

The following approach works for Angular 4, your mileage may vary.

To remove bootstrap code, we need to remove the script tags containing the code. This can be done via cheerio, an implementation of jQuery for server side usage.

```
const dom = cheerio.load(html);
dom('script').each((index, item) => {
    if ('src' in item.attribs && !item.attribs.src.startsWith('http')) {
        dom(item).remove();
    }
});

const editedDom = dom.html();
```

remove-non-local-sources.ts hosted with ❤ by GitHub view raw

Removing non-local CSS and JavaScript from HTML

Serving the static HTML

All that is left now is to serve the static HTML to the spiders. This can be done using a `.htaccess` file, a configuration file that is used by Apache which is the most common web server provided by hosting services.

Assuming the static files are located in `/static` you can use the following configuration.

```
RewriteEngine On

# Remove trailing /
RewriteRule ^(.*)/$ /$1 [L,R=301]
```

```

# Rewrite spiders to static html
RewriteCond %{HTTP_USER_AGENT} (googlebot|bingbot|msnbot|yahoo|Baidu|aolbuild|facebooke
RewriteCond %{DOCUMENT_ROOT}/static%{REQUEST_URI}.html -f
RewriteRule ^(.*)$ /static/$1.html [L]

# Rewrite spiders to static index.html
RewriteCond %{HTTP_USER_AGENT} (googlebot|bingbot|msnbot|yahoo|Baidu|aolbuild|facebooke
RewriteCond %{REQUEST_URI} "^/$"
RewriteRule ^ /static/index.html [L]
```

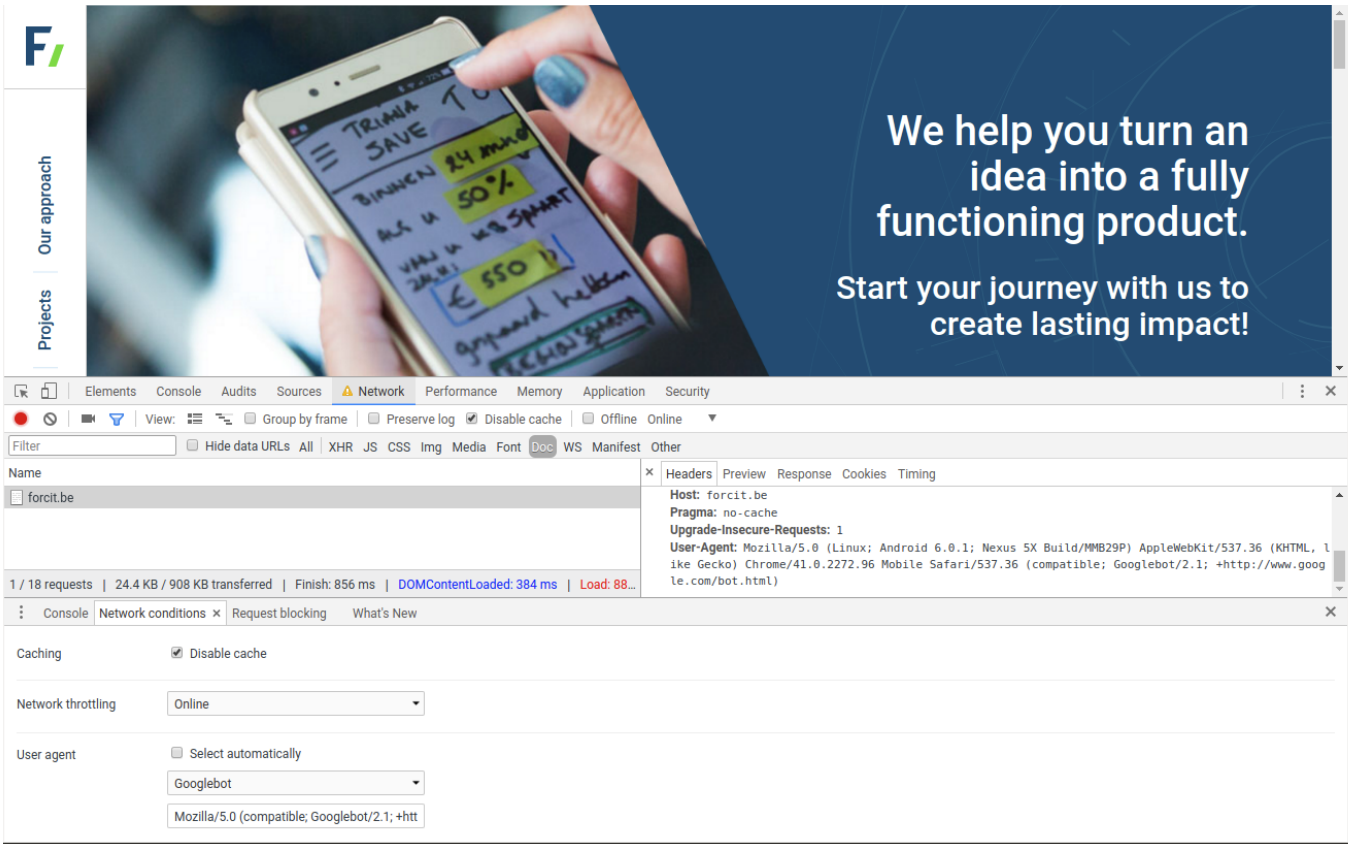
rewrite-spiders.htaccess hosted with ❤ by GitHub view raw

Rewrite spiders to static HTML files

The example does not contain all the spiders out there, but the most common ones are present.

Testing the solution

There are multiple ways to change your browser's *user agent*. Chrome has a built-in setting to change it using the Chrome Developer Tools. This setting is located in `settings > more tools > network conditions`.



Our approach
Projects
We help you turn an idea into a fully functioning product.
Start your journey with us to create lasting impact!
Elements Console Audits Sources Network Performance Memory Application Security
View: Group by frame Preserve log Disable cache Offline Online
Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other
Headers Preview Response Cookies Timing
Console Network conditions Request blocking What's New

Name
forcit.be

1 / 18 requests | 24.4 KB / 908 KB transferred | Finish: 856 ms | DOMContentLoaded: 384 ms | Load: 88...

```
Host: forcit.be
Pragma: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Linux; Android 6.0.1; Nexus 5X Build/MMB29P) AppleWebKit/537.36 (KHTML, l
ike Gecko) Chrome/41.0.2272.96 Mobile Safari/537.36 (compatible; Googlebot/2.1; +http://www.goog
le.com/bot.html)
```

Caching	✓ Disable cache
Network throttling	Online
User agent	☐ Select automatically
	Googlebot
	Mozilla/5.0 (compatible; Googlebot/2.1; +htt

Changing your browser's user agent using Chrome DevTools

The scraping of the routes and rendering of the static HTML is integrated in our *Continuous Integration* pipeline. A change to the website's content will queue the static HTML build process. That being said, CI is a topic for another post.

This solution is available as npm package named angular-statisfy. View the code at GitHub.

Learned something? Click the 👏 to say "thanks!" and feel free to share stories with your network and be part of our journey to inspire people, adventuring the future together and have a lasting impact.

Dev Angular 4 SEO Tech Developer