Use Nightmare.js to Generate a Confluence Sitemap

Created by Brian Immel on May 2019.

Introduction

In a recent project, I had a task to generate a list of documents in a particular Confluence space for the purpose of exporting them later. I chose to explore my options using Nightmare.js (http://www.nightmarejs.org/). Using this Node.js (https://nodejs.org/en/) (version 10.11.0) module, it allowed me to programmatically enter my credentials into Confluence, navigate to a specific document, and gather a list of documents (thanks to the target document using the Children Display macro that listed all the documents of the parent page of the target space). I also wanted this script to take arguments (flags) such as the username, password, spacekey, output file, and a delay value so that the process can be automated for a variety of reasons.

Note: this document is part of a series of Confluence notes and tutorials that you can find in my personal blog called CRUD The Docs (https://crudthedocs.blogspot.com/search/label/confluence)

Required skills and npm packages

This tutorial requires a number of skills and/or npm modules to complete everything mentioned herein:

- Confluence (5.x): You should be comfortable with creating pages that utilize the Children Display macro
- Nightmare (3.0.1): have some familiarity with the basics of this module
- Commander (2.19.0): have some familiarity with the basics of this module
- Cheerio (1.0.0-rc.2): have some familiarity with the basics of this module
- CSS: basic knowledge of how to select elements
- JavaScript: fair knowledge of how to use JavaScript

Setting up requirements

First, we set off with requiring a number of modules in our base app.js file:

```
const Nightmare = require("nightmare");
const cheerio = require('cheerio');
const program = require('commander');
const fs = require('fs');

....
```

Set up nightmare and flag options

The next two lines set up nightmare to display its process as it's going through the steps we'll program it to navigate and a selector to find the content we're looking for in our target document. The `confluenceSelector` is the CSS selector that will be used to find the desired content in the main body of the Confluence document.

```
....
const nightmare = Nightmare({
     show: true
});
const confluenceSelector = '#main-content';

....
```

Note: you don't want to see an Electron window pop up and nightmare to do it's stuff, set show to `false`.

Next, we set up the flags and their usage using commander module features:

```
...
program
     .version('0.0.1')
     .usage('-u -p -s -f -d ')
     .option('-u, --user', '*required* Username id')
     .option('-p, --password', '*required* User\'s password')
     .option('-s, --spacekey', '*required* Spacekey for the Confluence space')
     .option('-f --file', 'Text file to be used for tracking Confluence document names. Can be set to either true (defaults to the
spacekey naming scheme) or a file name.')
     .option('-d, --delay', 'Delay (in milliseconds) to wait for server response')
     .parse(process.argv);
...
```

With the flags set, we now need to parse them into an object that we'll use throughout the rest of the script. We loop through the program.rawArgs value provided by the commander module. In this loop, we are looking for specific flags so we can associate the flag with the value associated with it.

```
...
var argument = {};

for (var i = 0; i < program.rawArgs.length; i++) {
     if (program.rawArgs[i] == '--user' || program.rawArgs[i] == '-u') {
          arguments.user = program.rawArgs[i + 1];
     }
     if (program.rawArgs[i] == '--password' || program.rawArgs[i] == '-p') {
          arguments.pass = program.rawArgs[i + 1];
     }
     if (program.rawArgs[i] == '--spacekey' || program.rawArgs[i] == '-s') {
          arguments.spacekey = program.rawArgs[i + 1];
     }
     if (program.rawArgs[i] == '--delay' || program.rawArgs[i] == '-d') {
          arguments.delay = parseInt(program.rawArgs[i + 1]);
     }
     if (program.rawArgs[i] == '--file' || program.rawArgs[i] == '-f') {
          arguments.file = program.rawArgs[i + 1];
     }
}
...
```

Since the delay flag is optional, we should set up a fallback if the user doesn't supply one. In this case, we're setting the delay to 10 seconds though you can adjust this delay value to a number you're comfortable with your Confluence server responding to a login page request.

```
...
if (!arguments.delay) {
     arguments.delay = 10000;
     console.log('Server response delay not set. Assuming ' + arguments.delay + ' millisecond delay.');
}

...
```

Now we should set up the file path where we keep the site map information. If the user doesn't supply a file to output our data to, the script will use a fallback based on the submitted spacekey name.

```
...
if (arguments.file) {
     if (arguments.file.length > 5) {
          var confluenceSiteMap = arguments.file;
     } else {
          var confluenceSiteMap = arguments.spacekey + '-site_map.txt';
     }
} else {
     var confluenceSiteMap = confluenceSiteMap.txt;
}
...
```

The next thing our script will need is the Confluence URL to the site map document. Using the Children Display macro in your target Confluence space, we can gather all the document links in a single space by scraping this one document. Note: you should set up this Confluence document accordingly before executing this script and ensure it's named Site Map. Otherwise, you'll need to change the values in `arguments.confluence`.

```
...
if (arguments.spacekey) {
     arguments.confluence = + '/display/' + arguments.spacekey + '/Site+Map';
}
...
```

With the arguments parsed, we should check that the user supplied the required flags. If any of these flags weren't submitted, then the script should gracefully exit.

```
...
if (!arguments.user || !arguments.pass || !arguments.spacekey) {
     if (!arguments.user) { // user id is required
          console.log('Username is required.');
     }
     if (!arguments.pass) { // password is required
          console.log('Password is required.')
     }
     if (!arguments.spacekey) {
          console.log('Spacekey is required.')
     }

     process.exit(1);
...
```

Pull content with nightmare

With the required flags set for the CLI side of the script, we can now request a document from Confluence using your credentials. This chunk of code starts the nightmare.js process by navigating the Electron browser to the site map page in Confluence. The process below assumes that a login is required when the target page is loaded, enters user supplied username and password in the appropriate fields (denoted by their element ids), click the login button (denoted by it's element id), wait for a period of time (hopefully long enough for the server to respond), grab the content from the predetermined CSS selector via the evaluate method, return the data for parsing later, and close the Electron browser.

```
...
     } else {
          console.log('Getting document link list from ' + arguments.confluence);

          nightmare
               .goto(arguments.confluence)
               .type('#os_username', arguments.user)
               .type('#os_password', arguments.pass)
               .click('#loginButton')
               .wait(arguments.delay)
               .evaluate(confluenceSelector => {
                    return {
                         html: document.querySelector(confluenceSelector).innerHTML
                    }
               }, confluenceSelector)
               .end()
...
```

Parse content with Cheerio

Now that nightmare.js has retrieved the document in question, we use the then method to load the HTML content into cheerio.js to generate a list of links. Generally speaking, the links listed in a Confluence document usually follow the li span a selector pattern inside the body of the document. Here, we use the output variable to hold the list of links found in the retrieved data.

```
...
.then(obj => {
     $ = cheerio.load(obj.html.toString());

     var output = '';

     $('li span a').each(function() {
          output += $(this).html() + '\n';
     });
...
```

Then, we write out the list of links we found in the Confluence document to our predetermined text file.

```
...
     fs.writeFileSync(confluenceSiteMap, output, 'utf8');
})
...
```

Finally, we use the catch method to report back any errors.

```
...
     .catch(error => {
          console.error(error);
     });
}
```

Wrapping up

With the script complete, we should save it something like `confluenceSitemap.js` (or use the default name of `app.js`). From there, we can execute this command to generate our list of links text file: `node confluenceSitemap.js -u {username} -p {password} -s {spacekey} -f {links.txt}`

In a future version of this script, the error handling routine would include a process to handle error messaging instead of just dropping it in the console log of your browser.