UWMCMS Test Automation

Overview

What we have come up with is a relatively small, but still somewhat complex tech stack. Most of everything, including the tests you write, are going to be basically straight Node.js (JavaScript). These will be run with `npm`, with Nightwatch set to be `npm`'s test runner. Using either JSON config files, or .js files that return a config in JSON format, these tests will then either run locally, on your machine, using a *webdriver* for the particular browser you are testing, or they will run remotely on Browserstack. Remote runs will be managed and run by Browserstack, with the config file setting parameters such as which browser, OS, and tests to run. Reporting is visible on Browserstack's Dashboard, and a report is displayed during the run on the command line.

How all the pieces fit

Browserstack is the remote service we have chosen to run our cloud-based, cross-platform tests. They provide virtual machines of Windows (*Windows XP, Win 7, 8, 8.1, and Windows 10*), and OS X devices running from current Catalina, all the way back to Snow Leopard. For Mobile, they have a huge selection of devices: Android has hundreds of phones and tablets, from nearly all device manufacturers and Android OS versions, while iOS has iPhones ranging from the most recent iPhone 11 Pro running iOS 13, down to iPhone 5S running iOS7 and iPads from iPad Pro 12.9 2018, down to iPad Mini 2, as well as several simulators for older devices, if needed. Finally, they have a small selection of Windows Phone if you want to be super complete.

Unless otherwise mentioned, all of these mobile devices are real-world devices attached to the cloud, not simulators. For the Desktops, they are all virtual machines, but they are running physical versions of the browsers.

As for browsers, it varies from device to device, but nearly everything will have a choice of Microsoft Edge (latest), Internet Explorer 11, Firefox (versions up to 75 beta), Google Chrome (up to version 80, also including the latest Beta and Dev builds), Opera (version 67 latest, 98 dev), Safari (5.1), and if you really want to push it, they have the latest build for Yandex.
All of these are available either through automation, or to run manually (see https://live.browserstack.com/, login, and have at it.)

Node.js (https://en.wikipedia.org/wiki/Node.js) is the runtime environment we are using, and you should have access to most, if not all, Node libraries from within your test cases. Node also has a package manager named **`npm`** (Node Package Manager), which will be how you install and manage these various pieces of software. More on this later.

Nightwatch (https://nightwatchjs.org/) is the testing solution that connects all of these parts together. It is fairly intuitive, but it is also fairly new. You can use a mixture of the Nightwatch-provided *Assert/Expect* libraries, along with other *assert/expect* and testing libraries provided by

and for Node.js to make up any given test. All of these tests and the Nightwatch API are going to be using ***Selenium*** commands under the hood. ***Selenium/Webdriver*** has long been the go-to for automating websites, with ***Appium*** doing the same for mobile devices. You will likely find more documentation and answered questions out there for Selenium than you will for Nightwatch in particular but the majority of the time, the answer will apply to Nightwatch as well.

Overview of file structure

The file structure we've went with is a rough mirror of the site itself:

```
.
├── conf
├── node_modules
├── tests
│   ├── custom_assertions
│   ├── custom_commands
│   ├── page_objects
│   │   ├── footer
│   │   ├── header
│   │   └── provider_cards
│   └── site
│       ├── ecare
│       │   └── prod01
│       ├── education
│       ├── healthlibrary
│       ├── providerresource
│       ├── rightasrain
│       ├── root
│       │   └── patient-resources
│       ├── www
│       │   ├── about
│       │   │   ├── awards
│       │   │   ├── career-opportunities
│       │   │   ├── compliance
│       │   │   ├── contact-us
│       │   │   ├── policies-and-notices
│       │   │   ├── the-uwmedicine-family
│       │   │   └── uw-medicine-media-style-guide
│       │   ├── bios
│       │   ├── forms
│       │   ├── general
│       │   ├── locations
│       │   ├── online
│       │   ├── patient-care
│       │   ├── patient-resources
```

```
│       │   ├── provider-resource
│       │   ├── referrals
│       │   ├── research
│       │   ├── school-of-medicine
│       │   │   ├── about
│       │   │   ├── admissions
│       │   │   ├── gme
│       │   │   ├── md-program
│       │   │   └── resources
│       │   ├── search
│       │   ├── services
│       │   └── specialties
│       └── www2
│           └── referrals
│               └── refer-a-patient
└── tests_output
```

This shows 4 top-level directories:

conf: Intended to house the configuration files. For ease of getting this up quickly, we've just been dropping the main conf file we need in the project root directory. It would probably be a good thing to move it inside of here.

node_modules: All the different modules that have been installed and can be used by Nightwatch, Node, or anything else in this project.

tests: This is where all of the written tests are stored, see below.

tests_output: This is where various output from the tests are stored locally. Screenshots, logs, etc.

The main directory you will be spending time within is **tests/**. Within this are 4 directories that contain the real meat of the tests and project:

custom_assertions: Stores any assert-style functions you have created. If you find yourself using the same set of *N* assertions over and over, or if you find something that requires an intricate function to verify, AND that function will be used elsewhere to verify something, consider writing a custom assertion for it. There is a setting in the config file pointing to this folder, and everything found within it can be considered a global function.

custom_commands: Similar to custom_assertions, this is where you store commands that you have written, that are *not bound tightly with a particular page* (those would be store in *page_objects*, discussed later). These are globally available helper functions that might be used by any given test. A good example is ***acceptCookiesDialog.js***, a custom_command. Any page under test might need to handle the EU Cookie Acceptance pop up, so it makes sense for it to live here, rather than each *page_object* or test repeating the same code.

page_objects: Page Objects are a way to abstract the behavior and elements of a given page. Most of the tests I wrote utilize these heavily (see discussion on the 2 "styles" of writing test cases in Nightwatch, later for a more thorough discussion). A Page Object is made of basically 2 parts. First, an ***elements*** section which breaks down any (or all) of the DOM elements you want to work with, giving them a "friendly name" or alias, connected to a CSS Selector or XPath Selector to find them. This makes it so that you can simply refer the alias (`'@pageTitle'`) instead of needing to use the fully involved selector (`'body.somePage.class > div.main > div.title h1.main_page_title.mobile'`), whenever you need to refer to an element (which is *ALL THE TIME*). If for some reason the underlying CSS of the page changes, this also allows you to change that CSS in one simple, easy to find place, and have that change reflect across any test that tries to access that particular element.
Second is a ***commands*** section. This section is composed of functions that are callable from an instance of the page object. So, set up all the elements, set up a function for each discreet thing you want to verify on the page, then an automated test will just need to make an instance of page object, then call those tests (or a subset of them). There will be examples later.

site: This is where the actual test case/suites live. It is subdivided into directories that mimic the page layout of the site itself. Currently, the config is set to point to this directory and will look for runnable tests anywhere within `site/` or any of its subdirectories, recursively. Also currently, there are branches without any tests, simply to make a framework on which you can hang new tests, without having to create the underlying folder structure.

Test Setup and Design

There are 2 basic "styles" of how you can write a test. The style that a lot of the Nightwatch documentation and answered questions is what I will call Style One. This style does not rely on Page Objects and does almost all the work within the test case file itself. It tends to use built-in *before_all-*, *before-*, *after-*, and *after_all-* functions for test set up and tear down (note that Style Two can also use these), then have a separate function or set of functions for individual test cases. Each test case will run as an independent case, with the helper functions running before or after them: if you have a Style One case with *test1*, *test2*, and using all the helper functions, it would run in this order:

before_all: this will run before anything else, one time.
before: this will run before each test
test1: First test in the file
after: runs after test1
before: runs before test2
test2: runs after test1
after: runs after test2
after-all: runs last, used to finish any tear down that you may need

The output you get will show 2 different test runs, one for *test1* and one for *test2*.

With Style Two, you utilize page objects to abstract out the page that is under test and use your test cases to call the page object commands. This is a little more complicated than Style One, but it has several things that make it very useful:

- The entire page under test is abstracted out in one big file. If any CSS Selectors change, you only need to edit the elements section of the Page Object in order to fix it.
- It makes the tests themselves much, much shorter, and much, much easier to read, especially if you take care to name your Page Object commands in such a way that they clearly state what they are doing, for instance, ***verifyCTALinksAppear()*** or ***verifyClickingSocialMediaLinks()***
- Instead of having one huge monolithic test, you can now break it up into sections, allowing you to mix or match what exact things you want to test at any given time. For example, suppose you have a page object that has the following commands in it:

  verifyHeaderLinks()
  verifyHeaderImages()
  verifyAllCTAButtons()
  verifyAllPageImageSources()
  verifySearchByName()
  verifySearchByZIP()
  verifySearchBySpecialty()
  verifySearchBySpecialtyDropDown()
  verifyFooterLinks()

  Now, you could write a series of test files:

  - **`homepage.js`**: Fully tests everything on the homepage. Calls all of the ***verify..()*** functions above.
  - **`headerFooter.js`**: just calls the ***verifyHeaderLinks()***, ***verifyHeaderImages()*** and ***verifyFooterLinks()*** commands.
  - **`verifySearchAll.js`**: Calls all the ***verifySearch...()*** functions

  If you had a major homepage change, then your test case could just call `homepage.js`, if you had a change that affected the header or footer, call `headerFooter.js`, and so on.
- This also allows you to have one person who is not as technical, able to look at your "library" of Page Object commands for a page, then easily write a custom test case using those commands, without having to dive deeply into writing JavaScript, while your devs/more technical person can focus on abstracting pages into Page Objects and writing commands for them.

Both of these style can utilize parts of the other, and both can be used in the same suite, it's merely different accents of the same language.

Disabling a single test case

In all of the tests I've written, near the top of the file is a line:

```
'disabled': false,
```

Setting this to true will prevent the test from running. So, if you have some failures, and still want to run a set of tests, but know that one of them is broken and being fixed, just set disabled to true, and it will eliminate it from the run.

The configuration file

A big part of running any of these test cases is having a configuration file set up correctly. Currently, our conf file is located at

```
../uwmcms/tests/nightwatch/nightwatch.conf.js
```

Let's step through this file section by section.

```
/**
 * @file
 * Default configurations for NightwatchJs.
 */
```

Standard file boilerplate.

```
require('dotenv').config();
const args = require('minimist')(process.argv);
```

Makes sure we have some helper libraries, in this case to help with storing our Browserstack user/pass cleanly and securely, and to make it so we can have one monolithic `nightwatch.conf.js` file, which makes it easier when running (you don't need to remember which specific conf file you want.)

```
custom_assertions_path: 'tests/custom_assertions',
custom_commands_path: [
    'tests/custom_commands',
],
globals_path: 'tests/globals.js',
page_objects_path: [
    'tests/page_objects',
    'tests/page_objects/header',
    'tests/page_objects/footer',
    'tests/page_objects/provider_cards',
],
```

This block lets Nightwatch know what paths to look for things. Add more paths as needed. In this case, we have given it the path for Custom Assertions, Custom Commands, where we would store any globals, and which paths to look for Page Objects. Also part of this section is:

```
src_folders: ['tests/site'],
```

Which lets Nightwatch know where to look for tests. It will look in this folder(s), and any subfolders of it, recursively.

```
get webdriver() {
...
},
```

This sets up the webdriver, and various settings related to it, based on if it is a local run or a Browserstack run.

```
// 'test_settings': {}
get test_settings() {
```

This returns a JSON object with settings for how to run the tests. It is broken up into 2 sections, the top for Browserstack runs and the bottom for local runs. Let's look at the bottom, for local runs, first:

```
   if (args.browserstack == true) {
    // Setting for local runs
    ...
   else {
      return {
        'default': {
          'screenshots': {
            'enabled': true,
            'on_failure': true,
            'on_error': true,
            'path': 'tests_output/screenshots'
          },
          'desiredCapabilities': {
            'browserName': 'chrome',
            'chromeOptions': {
              'args': [
                'headless',
                'window-size=1120,800'
              ]
            },
          }
        },
      }
    }
```

```
}
```

This is the basic setup for each run, handling all the things Nightwatch (and Nightwatch and Browserstack for cloud runs) needs to know to run.

It sets values for the runner itself:

- Are screenshots enabled?
- Do I take a screenshot on a test failure?
- How about on an error (network, etc)?
- If so, where do I store them?

Then it sets the capabilities of browser this particular run is using:

- Which browser?
- Any special options?
- Do you want to fire up a full browser with UI, or just a "headless" one with no UI?
- How big is the window?

Currently our conf is set up to run local runs on Chrome. If you wanted the option to run it on Chrome or IE, you would add another nearly identical section, with the new information. You can see this in action if you look at the top part, ie the config for Browserstack, which is similar. Here, I've trimmed this down for size, but each section will be very similar, with the same keys, but slightly different values, based on browser/os/etc:

```
  if (args.browserstack == true) {
    return {
      'default': {
        'screenshots': {
          'enabled': true,
          'on_failure': true,
          'on_error': true,
          'path': 'tests_output/screenshots'
        },
        'desiredCapabilities': {
          'browserstack.user': process.env.BROWSERSTACK_USER,
          'browserstack.key': process.env.BROWSERSTACK_KEY,
          ['browserstack.local']: false,
        }
      },
```

This sets up the basic, default value for running a test. These settings act as a base that the next entries can "extend":

```
      'browserstack': {
        'extends': 'default',
        'desiredCapabilities': {
          'browserstack.user': process.env.BROWSERSTACK_USER,
          'browserstack.key': process.env.BROWSERSTACK_KEY,
```

```
      'project': 'UWM Web',
      'build': 'Browserstack',
      ['browserstack.local']: false
    },
  },
```

Ok, now we have a base for tests on Browsestack, these values won't change (probably) from test to test, and they themselves can be extended to a more sprecific setup:

```
  'browserstack.win10ie': {
    'extends': 'browserstack',
    'desiredCapabilities': {
      'os': 'Windows',
      'os_version': '10',
      'browserName': 'IE',
      'browser_version': '11.0',
      'resolution': '2048x1536',
      'name': 'Win10-IE11-2048',
      ['browserstack.local']: false,
    }
  },
```

Specific settings for running a test on Browserstack, using Windows 10, Internet Explorer Version 11, at 2048x1536 resolution. It also adds a "Name", which is an optional Browserstack string that will display when you view the results on Browserstack's Dashboard. I've been setting them to a short-form of OS-Browser|Version-screenwidth, but it can be whatever you choose it to be. The first line, `'browserstack.win10ie'` provides the name you can use for this particular combination when running tests from the command line. The config file goes on, with each combination getting its own block:

```
  'browserstack.win10chrome': {
    'extends': 'browserstack',
    'desiredCapabilities': {
      'os': 'Windows',
      'os_version': '10',
      'browserName': 'Chrome',
      'browser_version': '80.0',
      'resolution': '2048x1536',
      'project': 'UWM Web',
      'build': 'Browserstack',
      'name': 'Win10-Chrome80-2048',
      ['browserstack.local']: false,
    }
  },
  'browserstack.macsafari': {
    'extends': 'browserstack',
    'desiredCapabilities': {
```

```
        'os': 'OS X',
        'os_version': 'Mojave',
        'browserName': 'Safari',
        'browser_version': '12.0',
        'resolution': '1920x1080',
        'project': 'UWM Web',
        'build': 'Browserstack',
        'name': 'MAC-Catalina-Safari13-1920',
        ['browserstack.local']: false,
      }
    },
```

This sets up 2 more combinations

- Win10 with Chrome80
- Mac OS Mojave running Safari

Note that it also adds 2 new optional string keys: `project` and `build`. Again, these are user-customizable. They will appear on the results page, allowing you to look only at certain Projects or Builds or Names. This will be one thing that you will need to discuss to decide on the best naming conventions here.

I've trimmed the rest of the file for space, but it's all basically this. Have a base set of config options, then moving from more general to more specific, use "extend" to build out a set of combinations with differing OS/OS Version/Browser/Browser Version and Resolutions, then use the name (ie browserstack.macsafari) to specify which combo to use when initiating a run via the command line.

How to Run Tests

Let's start with the most simple first; we have a test file with 1 test case, and we want to run it locally.

First, check `nightwatch.conf.js`, and in the local section, check to see if the 'headless' option is set. If it is, you will not see a browser launch, it will run in 'headless' mode. This is fine, but if you want to see it launch the browser and run the test on it, just comment out 'headless'.
I'm also assuming you are in the Nightwatch 'root' directory, in my case:

`~/gray/git/uwmcms/tests/nightwatch/`

(aka, the same place the nightwatch.conf.js file is located).

We have configured `npm` to have Nightwatch as it's test-runner, so we can run this particular test by typing the following on the command line:

```
npm run tests -- --test tests/site/www/bios/bioPage.js
```

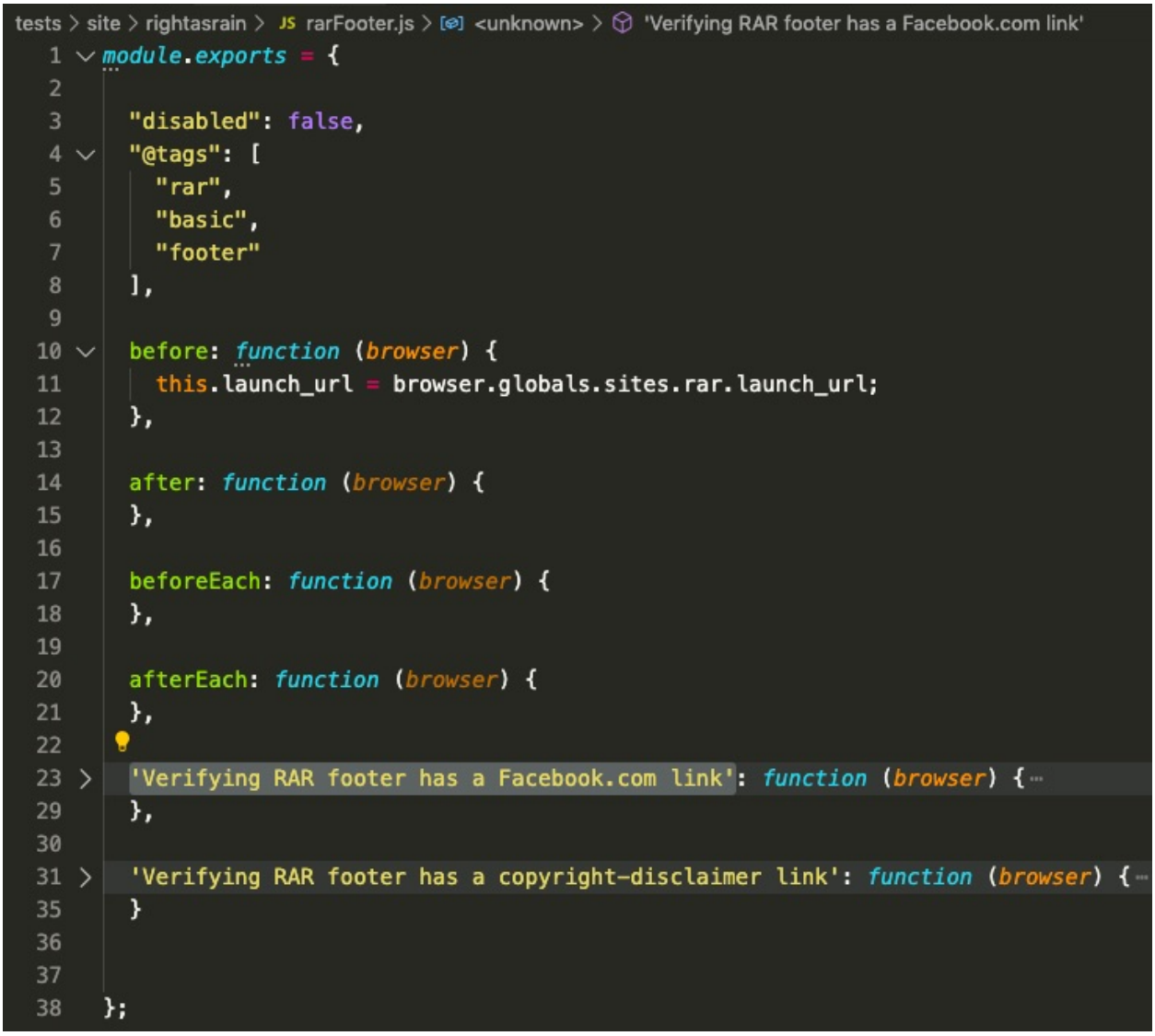
This will run the bioPage.js test file, and give the following output:

```
>> npm run tests -- --test tests/site/www/bios/bioPage.js

> uwmcms@1.0.0 tests /Users/gray/git/uwmcms/tests/nightwatch
> nightwatch -c nightwatch.conf.js "--test" "tests/site/www/bios/bioPage.js"


[Www/Bios/Bio Page] Test Suite
==============================
i Connected to localhost on port 9515 (1209ms).
  Using: chrome (80.0.3987.149) on Mac OS X platform.

https://stevie.cmsstage.uwmedicine.org/bios/rebecca-abay
Running:  Provder name will appear in content and page title

✔ Passed [ok]: The accept cookies dialog is visible; clicking accept.
✔ Testing if the page title contains 'UW Medicine' (9ms)
✔ Found Proivider Card.
✔ Found Provider Card title.
✔ Passed [ok]: Provider <h1> title is longer than 20 characters.
✔ The <head> title includes the provider's <h1> title (5ms)

OK. 6 assertions passed. (4.19s)
```

This file only has one test case in it, so it only runs that one test case. Had I split this up, and had 2 test cases in it, it would run each of them.

Let's go over the output here. It will be very similar no matter what test cases you are running.
The first line breaks down the location of the test file it is running. In this case, [Www/Bios/Bio Page] Test Suite means it is running the `/tests/site/www/bios//bioPage.js` file.
It is connecting to localhost (it is running locally, it would show the Browsrstack host url and port otherwise).
The rebecca-abay URL is a bit of logging from the test case.
Running: shows which particular case in the file it is currently running. In this case, there is only one, so it shows the title of that test case.
Next, we get a series of green checkmarks. Each of these is for a particular *assert* within the test, think of this as test steps.
Finally, we get an OK, indicating all the assertions in the test passed (ie, the test passed), and a count of how many assertions (steps) it went through, as well as the time from start to finish, in this case, a hair over 4 seconds.

If you have a test case file with multiple cases in it (referred to in Nightwatch docs as a 'suite'), you can either run as above and have all cases within it processed, or you can specify a particular case, and run only it. For instance, one of Nick's RAR test cases, `rarFooter.js`, is shown below. Note that it has 2 cases within it, *'Verifying RAR footer has a copyright-disclaimer link'* and *'Verifying RAR footer has a Facebook.com link':*



Let's just run the Facebook link test, using the 'testcase' flag:

```
npm run tests -- --test tests/site/rightasrain/rarFooter.js --testcase
'Verifying RAR footer has a Facebook.com link'
```

```
>> npm run tests -- --test tests/site/rightasrain/rarFooter.js --testcase 'Verifying
RAR footer has a Facebook.com link'

> uwmcms@1.0.0 tests /Users/gray/git/uwmcms/tests/nightwatch
> nightwatch -c nightwatch.conf.js "--test" "tests/site/rightasrain/rarFooter.js" "--
testcase" "Verifying RAR footer has a Facebook.com link"


[Rightasrain/Rar Footer] Test Suite
===================================
i Connected to localhost on port 9515 (1179ms).
  Using: chrome (80.0.3987.149) on Mac OS X platform.

Running:  Verifying RAR footer has a Facebook.com link

✔ Expected element <.footer a[href*="facebook.com"]> to be present (20ms)

OK. 1 assertions passed. (1.345s)
```

Okay, now let's move on to running our tests on Browserstack. Below is the command line and output for doing the above two examples, but this time on Browserstack, by simply adding the `--browserstack` flag.

```
>> npm run tests -- --test tests/site/www/bios/bioPage.js --browserstack

> uwmcms@1.0.0 tests /Users/gray/git/uwmcms/tests/nightwatch
> nightwatch -c nightwatch.conf.js "--test" "tests/site/www/bios/bioPage.js" "--brows
erstack"


[Www/Bios/Bio Page] Test Suite
==============================
i Connected to hub-cloud.browserstack.com on port 443 (10864ms).
  Using: internet explorer (11) on WINDOWS platform.

https://stevie.cmsstage.uwmedicine.org/bios/rebecca-abay
Running:  Provder name will appear in content and page title

✔ Passed [ok]: The accept cookies dialog is visible; clicking accept.
✔ Testing if the page title contains 'UW Medicine' (385ms)
✔ Found Proivider Card.
✔ Found Provider Card title.
✔ Passed [ok]: Provider <h1> title is longer than 20 characters.
✔ The <head> title includes the provider's <h1> title (304ms)

OK. 6 assertions passed. (11.741s)

  See more info, video, & screenshots on Browserstack:
  https://automate.browserstack.com/builds/f5f9add442e4a8603945aff9be04a0cc568a5eba/s
essions/84ebe1cfcaaa07dab07c36153f8a1188f6b24d75

/Users/gray/git/uwmcms/tests/nightwatch [git::tests/tests-structure  ✗]  [Wed 1, 11:4
2AM]
>> npm run tests -- --test tests/site/rightasrain/rarFooter.js --testcase 'Verifying
RAR footer has a Facebook.com link' --browserstack

> uwmcms@1.0.0 tests /Users/gray/git/uwmcms/tests/nightwatch
> nightwatch -c nightwatch.conf.js "--test" "tests/site/rightasrain/rarFooter.js" "--
testcase" "Verifying RAR footer has a Facebook.com link" "--browserstack"


[Rightasrain/Rar Footer] Test Suite
===================================
i Connected to hub-cloud.browserstack.com on port 443 (4702ms).
  Using: internet explorer (11) on WINDOWS platform.

Running:  Verifying RAR footer has a Facebook.com link

✔ Expected element <.footer a[href*="facebook.com"]> to be present (654ms)

OK. 1 assertions passed. (3.065s)

  See more info, video, & screenshots on Browserstack:
  https://automate.browserstack.com/builds/f5f9add442e4a8603945aff9be04a0cc568a5eba/s
essions/e20ab77c3e27b7274633770c3a320f852cd89361
```

Notice that we did not declare what OS or Browser to use. If you look at the nightwatch.conf.js

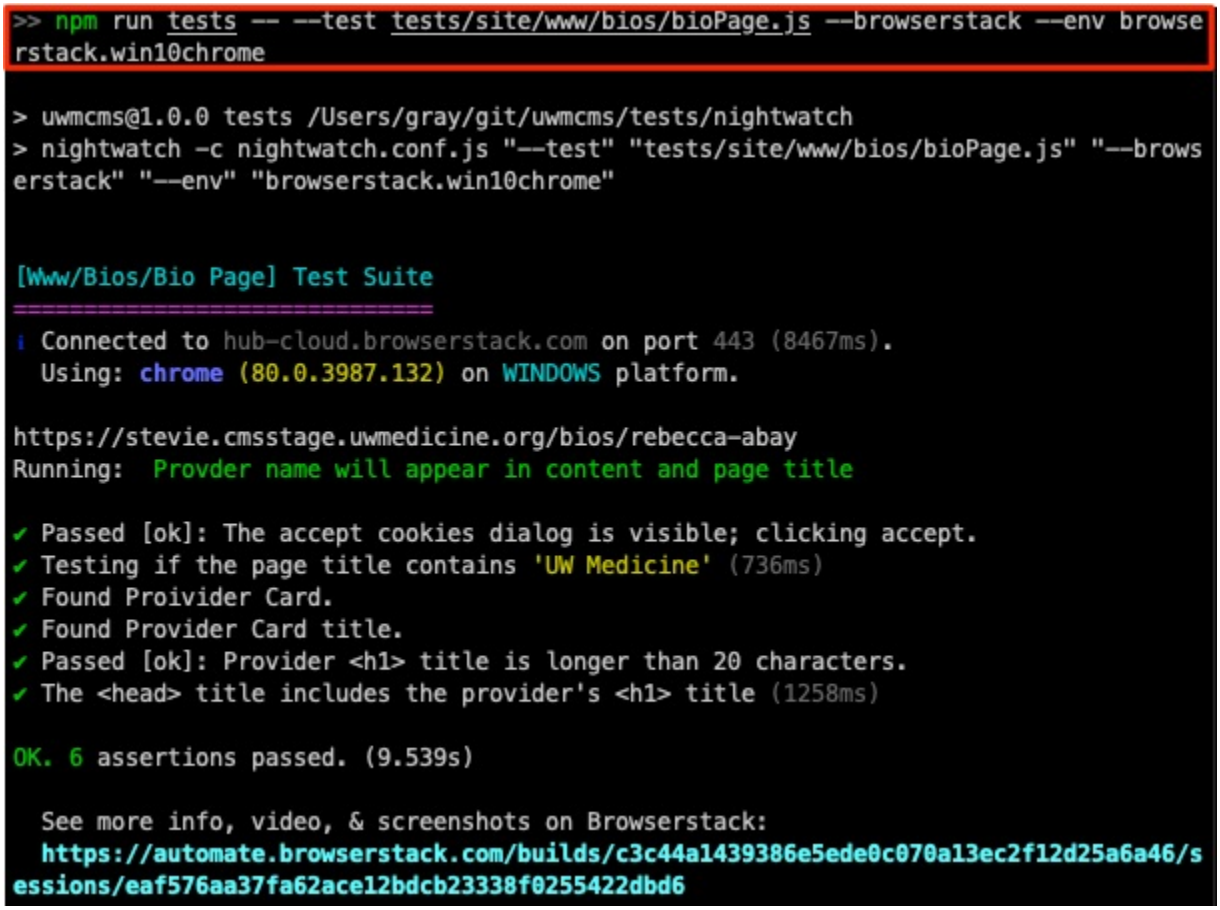
file:

```
'default': {
  'screenshots': {
    'enabled': true,
    'on_failure': true,
    'on_error': true,
    'path': 'tests_output/screenshots'
  },
  'desiredCapabilities': {
    'browserstack.user': process.env.BROWSERSTACK_USER,
    'browserstack.key': process.env.BROWSERSTACK_KEY,
    'os': 'Windows',
    'os_version': '10',
    'browser': 'IE',
    'browser_version': '11.0',
    'resolution': '2048x1536',
    ['browserstack.local']: false,
  }
},
```

You will notice that the 'default' we have sets OS to Windows 10, and browser to IE 11. So, this is picked by default [Issue 3]. If you want to specify a different config, you will use the `--env` flag. For instance, we have a test settings group in `nightwatch.conf.js` named '*browserstack.win10chrome*':

```
100      'browserstack.win10chrome': {
101        'extends': 'browserstack',
102        'desiredCapabilities': {
103          'os': 'Windows',
104          'os_version': '10',
105          'browserName': 'Chrome',
106          'browser_version': '80.0',
107          'resolution': '2048x1536',
108          'project': 'UWM Web',
109          'build': 'Browserstack',
110          'name': 'Win10-Chrome80-2048',
111          ['browserstack.local']: false,
112        }
```

So, we can easily run a test on it, by setting the `--env` flag to the combination from the conf file we want:



You can also use the --env flag to run on multiple combinations. Let's say we want to run this test case on Windows 10/Chrome and OSX Catalina/Chrome. Just add both combo names, separated by commas, no spaces, to the env flag:

Note that it automatically color-differentiates the different browsers in the report/output to make it easier to read.

Now, what about running a bunch of different tests, all at the same time? There are several ways to do this:

- use the `--test` flag, and give it a list of filenames, comma-separated, no spaces
- use the `--test` flag, giving it wildcard or regex for the test file names. Note that I've not played with this at all, so check the Nightwatch documentation to find examples.
- most powerfully, use the `--tag` flag to run any cases that have the specific tag or tags you give it. This is the way I've nearly always used. Based on how you have tags in the files, you can really fine-tune which specific tests you want to run at any time. For instance, you can do something like:

  ```
  npm run tests -- --tag stevie --browserstack -env
  browserstack.macchrome
  ```

  to run all tests tagged as 'stevie', on Browserstack, using OSX Catasline/Chrome.

More:

- You can run multiple tags. ***Unlike most other syntax***, instead of a comma-separated list, you use a separate `--tag` flag for each tag:

  ```
  npm run tests -- --tag grid1 --tag grid2 --browserstack --
  env browserstack.win10chrome
  ```

- You can use tags to SKIP certain files. For instance, run all the 'stevie' tagged cases, EXCEPT the ones with a tag of 'grid1':

  ```
  npm run tests -- --tag stevie --skiptags grid1 --
  browserstack --env browserstack.win10chrome
  ```

- You can use the `--config` flag to point to a specific conf file. For instance, you could make separate config files for Dev, Prod, Smoketest, and FullPass, then call each by passing it the path to the particular conf file you want to use. This is one way of helping the `nightwatch.conf.js` from getting huge. Note that if you don't specify the `--conf` flag, it will look for `nightwatch.conf.js` in the 'root' Nightwatch' directory (as shown in all of the examples above).

For many more option and much more detail see https://github.com/nightwatchjs/nightwatch-docs/blob/master/guide/running-tests/runner-options

This is (at this time) a complete list of all tags on all the cases that have been written:

404, allergy, appointments, basic, bios, feedback, footer, footer, form, general, header, home, jnm4, locations, patient-resources, primarycare, provider-book-online, providers, rar, search, services, specialties, stevie, urgent-care, virtualclinic
The test cases I wrote and verified for multiple OS/Browser/Device, each has a tag, *grid1* to *grid14*, that allowed me to easily keep track of what I was running and testing. These tags can go away whenever.

Useful Links

<< insert links list >>

Outstanding issues

Issue 1: `page_objects/uwmHome.js`
lines 25-39
When running this with Mac Chrome, clicking the video is not working as expected. It says that the video iframe is visible... but if you look at the Browserstack video, it never actually launches. So then, when it looks for the close element, it fails.

Issue 2: `page_objects/uwmHome.js`
When running header and footer, this is way to long, taking >5 minutes to run, which just invites flakiness. At bare minimum I suggest breaking it into a homepage test (no header footer, like the rest of the page tests), a header test, and a footer test. It might actually be better to break it up into several small tests.

Issue 3: `nightwatch.conf.js`
lines 70-74
If these are commented out, running all the *browserstack.nnnn* combos work, but running default (not specifying a browser combo) will fail. If they are commented out, the default will fail (it looks to me like it is trying to build both the default and the combo you call). Right now, I've just left it commented out because, for the most part, I always want to declare exactly what combo(s) I want to run. Obviously I have some small syntax error in how I have the default set up and/or how the *browserstack.nnn* entries extend that default.

Other

- Nightwatch is very configurable. If you do not like the included report-system/output, you can, with a simple config change, use another third-party reporter, or one custom build, in-house.
- The service used to run cloud-based tests is also configurable. For instance, you could just as easily use SauceLabs instead of Browserstack.
- Also, note that the developers are Nightwatch are also apparently working on a similar service as Browserstack and SauceLabs; this might be worth keeping an eye on since it will be built from the ground up with Nightwartch in mind.