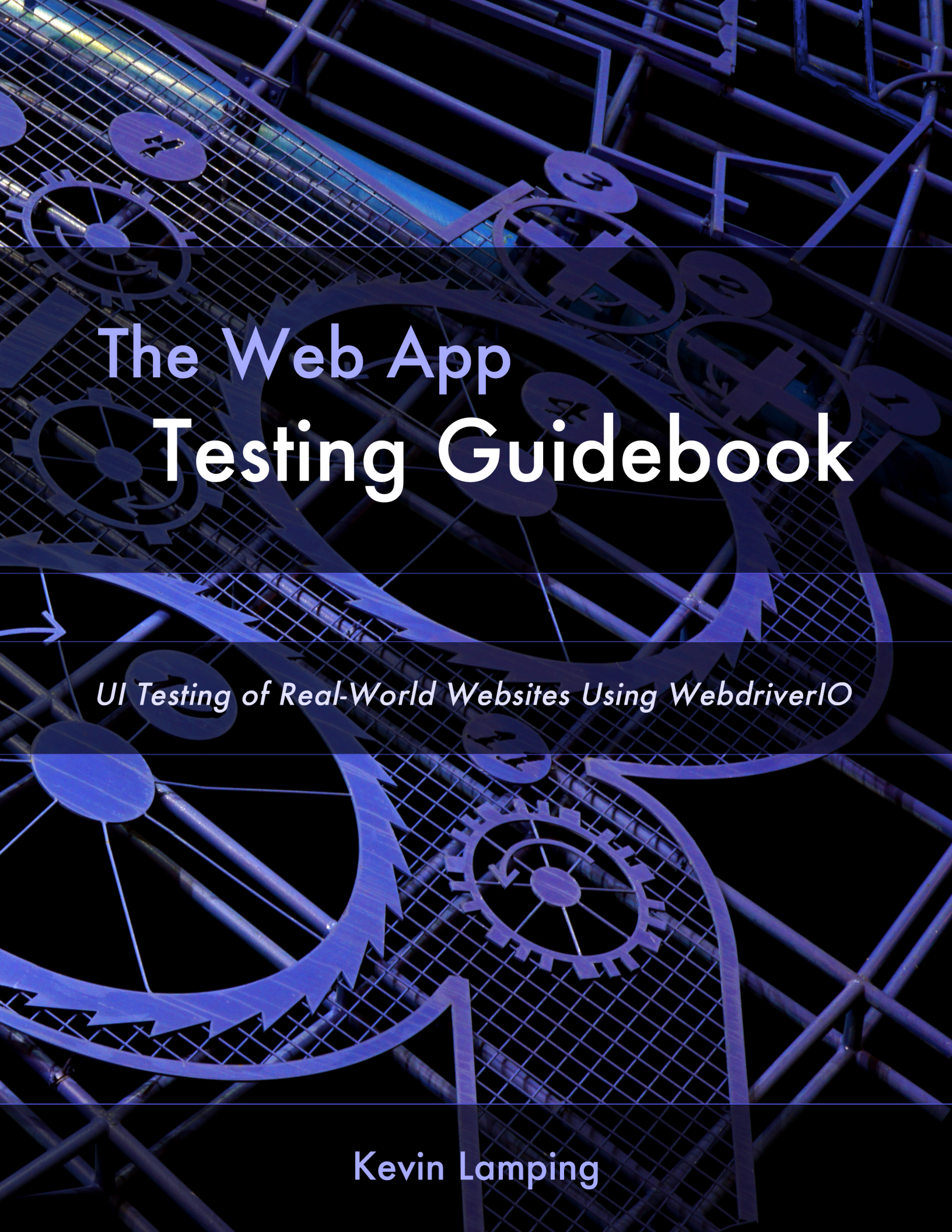


The Web App Testing Guidebook

UI Testing of Real-World Websites Using WebdriverIO

Kevin Lamping

The Web App Testing Guidebook

UI Testing of Real World Websites Using WebdriverIO

Kevin Lamping

This book is for sale at http://leanpub.com/webapp-testing-guidebook

This version was published on 2020-10-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

1.1 Introduction

1.1.1 Why Read This Book?

This may sound a little odd…

While WebdriverIO is in this book's title, this isn't a book *about* WebdriverIO.

Yes, it does cover the popular test automation framework in-depth. More importantly though, this book is about teaching you how to effectively use UI Test Automation to validate Web App functionality.

The lessons here focus less on how specific WebdriverIO commands work and more on how specific approaches to testing are beneficial or harmful.

And yet, while any test framework would work for this, I did choose WebdriverIO for a specific reason: It's a mature framework that allows us to spend less time on code and more time on important testing concepts.

Why do I start off the book saying all of this?

Well, over the past decade, website complexity has grown substantially, requiring much more effort when it comes to testing. While test automation is certainly not a recent development, it has grown in popularity lately due to the increased complexity of the sites we build.

In fact, many organizations have full teams dedicated just to testing their site. Quality Assurance (QA) is an important role for any company that wants to take its technology seriously.

However, having humans manually run through test scripts is a time-consuming task.

By automating our tests, we hope to shift the workload from manual labor to speedy CPUs. Without humans and their need to sleep and eat, we can theoretically test our sites on-demand, around the clock.

Wouldn't it be ideal to have a test suite so effective that your QA team focused solely on keeping it up-to-date?

Unfortunately, that's not the reality.

I've seen, heard, and have been part of many teams that set out after this ideal, only to realize months later that all the effort has provided them little benefit. Yes, they have test automation in place, but it's constantly breaking and causing endless headaches. It seems they either have tests that run well but don't validate much or have tests that check for everything but report false errors.

In 2017, I spent the year recording screencasts covering WebdriverIO in-depth. While I covered the details of the framework quite well (or so I was told), I was left with nagging questions of "is knowing the tool valuable enough?"

Is knowing how to mix colors and hold a paintbrush enough to be able to paint beautiful art?

Does knowing what the 'addValue' command does in WebdriverIO teach you enough to write tests that are effective?

This time around, I'm focusing on that second part. Yes, it's important to cover the details of commands and code. More importantly though, you need to see how you can combine all that technology to create a test suite that provides actual value.

In this book, I cover not just what WebdriverIO can do, but specifically how you'll be using it day-to-day. I've built the examples around real-world scenarios that demonstrate how you would actually set things up. I'm not here to only teach you what WebdriverIO does. I'm here to teach you how to approach problems from various angles and come to the right solution.

It takes a little more work on my part, and extra effort on your's to get started, but the payoff is there, I promise.

With that, let's get started.

What is User Interface (UI) Test Automation?

I mentioned that test automation isn't anything new, but I didn't explain what it is.

Truthfully, test automation is a lot of things. There are a multitude of programs and tools surrounding it, not to mention the various ideologies about automated testing in general (e.g., Test-driven Development vs. Behavior-driven Development).

Aside from that, there are also different types of tests. In this book, we're focusing solely on UI automation, but there's also unit, integration, performance, accessibility, usability, you-name-it testing.

All of these are important to know about, and can provide as much or more value than UI testing. So why focus on UI testing?

Well, you don't really have an option. You can't skip accessibility testing if you want to have a site that's accessible. Equally, you can't skip UI testing if you want to have a site that's not broken.

Truthfully, we're doing tons of UI testing. Every time someone loads up a website, they're testing the UI. When they click that button, does it make that thing happen? When I use my mobile device, does the site fit on my small screen?

If you have a site in production, every user is testing your UI. Hopefully for them (and you), they're not the first ones to try it out.

UI testing is constantly being done, just in a very labor-intensive way. Front-end developers working on the code will spend half of their day in the browser manually testing if their changes worked. If they're testing in an older browser, it will be 3/4ths of their day.

UI testing is the simple (hah) task of validating that the code written for the browser, actually works in that browser, along with all the other components that went into that webpage.

UI test automation is a way to convert those manual mouse clicks and keystrokes into coded scripts that we can run on a regular basis.

Let's Talk Benefits

I've alluded to this before, but it's worth repeating. Here are some of the many real-world benefits of automated UI testing:

- Jamie, your ace front-end developer, just finished their latest task and is itching to release it. While they were careful to validate that the new functionality works, unfortunately they forgot to test whether that new code broke the zip code widget on the contact page. Lucky for you, the UI automation test caught the error, and a fix was in place before the end of the day.
- Taylor is an awesome full-stack developer. They've got everything about their coding environment fine-tuned so that they can focus solely on pumping out fixes and new features… except for one thing: Taylor always forgets to check their code on slower, outdated computers. That's okay though, as our UI tests are configured to run on a variety of setups, and it just so happens that it caught that issue when running inside a Windows 8 environment.
- Casey is a great product manager, but sometimes forgets to clarify the small details. This means developers can make the wrong assumption during development, which is only discovered by Casey during product demos. By adding automated tests to the mix, developers and product managers are pushed to have regular conversations on the desired behavior of certain functionality, resulting in fewer suprises later on.
- Avery is a superb QA tester. With great attention to detail, Avery always thinks of unique ways in which the website would be seen. Unfortunately, manually testing these edge cases takes a fair amount of time to get in place. By adding automated scripts, Avery is able to programmatically generate the needed data for their tests, and allow them to run these scripts on a regular basis.

These are just a few ways test automation can help. I skipped over many other benefits for the sake of time, but there's one I omitted on purpose. Many folks claim that UI tests allow you to have fewer developers/testers. While that could be an outcome of automation, I don't think it's a valuable argument.

In all the scenarios above, having an extra human around wouldn't have necessarily helped. That's because human's have blind spots, and many of them are the same. Teams have collective blindspots and that's difficult to get around. We naturally tend to focus on what's in front of us, forgetting about realities outside our own influence.

UI testing isn't about replacing humans, but rather augmenting their abilities. We use automation to shore up our limitations, allowing us to focus on our strengths.

Developers waste their time testing on hundreds of different devices, when they could instead let a computer do that part of the work. Your QA team wastes its time running through the same test scenario week after week, when instead they could be thinking of new and undiscovered test cases.

UI automation isn't about replacing humans with machines, but rather giving us more freedom to work better than machines.

And let's not forget, UI testing requires a fair amount of work. It's not magic (although it's okay to convince management it is). This brings me to an important consideration…

There Are Always Drawbacks

How long do you think it takes to get a set of automated tests up and running? A week… maybe two?

Sure, if all you want to do is test that a page loads properly.

But websites are incredibly complex — the number of features we jam on a page grows each day.

Consider a "simple" homepage. Here are some things you'll want to test on it:

- Do all the parts look right on a laptop and desktop computer?
- Do all the parts look right on a tablet?
- Do all the parts look right on a phone?
- Does the site navigation work?
- Does the "need help" chatbox pop up after five seconds?
- Does the autocomplete in the site searchbar work?
- Does the hidden menu show after you click the menu icon?
- Does the carousel on the homepage rotate correctly?

Okay, I'll stop there. Hopefully you get my point that there's a lot to test even on a single page. A basic script that loads a page doesn't provide much reassurance.

So if you want to test all your functionality on all your pages, you're going to have to write a lot of code.

Covering all of that ground takes time. Time that could be spent on other, possibly more important, tasks.

And as you're writing test after test, the website your testing is going to keep changing. New features and fixes will be continuously introduced. The same feature will be tweaked again and again, causing your once-solid test suite to mysteriously start failing.

When the glorious day comes and you're "done" writing tests, you still have to maintain them. Now, there are ways to write tests to make them more maintainable, and we'll cover that throughout the examples, but there's no such thing as a future-proof test. You're always going to need to update it.

There's one more important point to consider.

While you'll breeze through some parts of test writing, expect to sink 80% of your time figuring out how to test that special 20% of your site's functionality. There are many areas of writing tests that

aren't trivial. It's not as simple as calling the command to fill out a textbox and click the submit button.

You'll need to work with databases, integrate with third-party services and see if you can get commands to work in browsers with poor automation support. You're also going to run into instances where the way the website was coded just doesn't jibe with test automation.

Animations are a good example of that. How do you test that an animation works? You can fairly easily check the properties before and after an animation, but what about everything in between those two states?

Honestly, you'd need to do a screen recording of every frame of the animation, and compare that screen recording to a previous run to see if they match. I don't know about you, but that doesn't sound easy to me.

Simpler Sites for UI Testing

If you find yourself facing a complex site that would require major work to test properly, there are other options to gain some value without too much effort.

Instead of testing a fully working site, you may want to create a variation of your site that represents portions of the real thing. For example, pattern libraries are a popular option out there, especially for larger websites.

In case you're not familiar with them, pattern libraries are essentially living demos of the components that make up a website's interface. Mailchimp has a public pattern library if you'd like to see one in action (https://ux.mailchimp.com/patterns[1]).

For instance, a pattern library may contain standalone versions of the following:

- Site layout components like the main navigation and site footer
- Components used across multiple pages, like buttons, form inputs, and tabs
- Style guidelines for simple page elements like links, headings, and lists

Pattern libraries are very helpful for teams, to document how the website should look and act. If you're tasked with adding a sortable table to your companies site, having a pattern library with a living example of one makes the job simple.

They're also useful for testers, as it gives us testable examples of individual components isolated from the complexity of the full website.

[1] https://ux.mailchimp.com/patterns

Skipping Automation is Sometimes the Best Option

Hopefully I haven't scared you away from UI test automation entirely. I only wanted to get you thinking about the whole picture.

And that picture should include saying, "Maybe we shouldn't write test automation for that… at least not yet."

Let's consider a few things…

New Features Aren't User Tested

Business is booming and your team is tasked with a brand-new feature idea that management thinks the customer will love. While it's tempting to say, "Yes, and let's write tests to go side-by-side with this new feature," it might not be the right time.

This brand-new feature has never seen the light of day, and the second a customer sees it there's going to be something they don't like about it. If a decision is made to rework the concept based on customer feedback, all those tests you've written become useless.

There's no point in writing a test if you haven't user-tested your site. So before you spend time writing assertions, ensure the assumptions about the user interactions are initially put to the test.

Time Writing Tests Takes Away from Writing Features

You're not paid to write tests; tests only serve the application they're testing. If an app is useless, tests won't help.

If you're working on a side project for a tool that no one uses, spending time writing tests takes away from time spent on more important tasks, like getting people to use your work.

Users don't care whether you have good unit tests. There's no difference between an unused tool and an unused unit tested tool.

Let yourself have untested code. Worry about that problem when it actually becomes one.

Tests Are Only Valuable When You Use Them

Don't write more tests when you're not using the ones you already have.

If you have 500 UI tests, but never put in the time to integrate them in your build and deployment process, you have 500 useless tests. Writing 500 more won't help.

Your tests should run on every code push. They should run before every deploy. Every developer on the team should see that the tests passed or failed.

If that's not true, you shouldn't be writing more tests, you should be taking advantage of the tests you already have.

Parts of the Site Might be Better Tested by People

Remember when I said tests shouldn't replace people, but rather augment their abilities. Well, in the scenario of testing an animation, we were stuck with a really complex solution. What if we just went with manual visual validation of the effect?

It's okay to have some parts of your site that are too complex for automation. Grab that low-hanging fruit and leave the stuff higher in the tree for a later time when you have a ladder.

Are Tests Worth It Then?

I've outlined the benefits and drawbacks of test automation, including reasons to entirely skip some of it.

So how do you ensure you're getting more benefits than drawbacks? Focus on these two goals:

- Ensure you're gaining value out of every test you write
- Ensure you're selling that value to those in charge

It's really easy to get caught up in automation, trying to cover every nook and cranny of the site. But if a feature of your site doesn't provide much value, how much less value would a test for that feature be?

The site login component breaking… that's bad. The site's About page having a typo… not such a big deal. Sure, we'd want to fix it, but it (hopefully) won't cost the company a lot of money.

By focusing on gaining and selling that value, you can keep yourself honest in your test writing, and focus on what's important: having a site that's running smoothly and providing value to the user.

1.1.2 Why Use WebdriverIO?

Back in the late 2000s, I learned of a tool called Selenium that the tester's on my team were fairly interested in using.

I thought it was a neat idea, but there was one big red flag to me. It required writing the tests in the Java programming language.

I had taken a couple semesters of Java programming in college and actually enjoyed the object-oriented nature of the language. I had to wrap my head around some of the complexities of the language, but overall I found it a useful language to understand.

But thinking about writing automated tests in Java gave me pause. Java is a very verbose language requiring a fair amount of setup and some tedious coding. I just didn't think test automation was a good fit for it. So I stopped researching the idea in favor of other pending tasks.

Years later, a new tool came out called PhantomJS. It was based in Node.js, and promised the ability to automate browser usage. That definitely perked my interest, and I'll explain in a minute, but first…

What's a Node.js?

You may not be familiar with Node.js, so I'll explain it a little here. If you are familiar, feel free to skip this part.

JavaScript is the coding language of the web. Starting in the mid 1990's, an early version of JavaScript (originally called LiveScript) was included in the Netscape Navigator browser. Microsoft, eager to match and beat the features of Netscape, saw this new language and decided to add their own version (calling it JScript) to Internet Explorer (IE).

As the browser battle continued, JavaScript and JScript continued to grow in popularity among website authors. I recall my first use of JavaScript was to make a "mouse trail" on my very first website1[2]. My second use was to float an animation of Ralph Wiggum eating glue across the screen of my "from the local police blotter" page.

It was dumb, but boy was it fun to play around with.

Most JavaScript usage for the next decade revolved around either cheesy browser effects, or useful add-ons like drop-down menus and browser-based form field validation. As a front-end developer, learning JavaScript was an important part of your job, although not a critical part like it is today.

In 2009, Ryan Dahl combined Chrome's JavaScript engine (called V8) with a few new tricks, and have it run entirely outside the browser 2[3]. While the initial idea was to use JavaScript and Node.js to create servers that could better handle high-traffic sites, developers across the globe saw even more power in the tool.

From 2009 to 2019, Node.js has grown tremendously. While development of the tool did stagnate around 2014, a fork of Node.js called io.js kicked those in charge back into gear, and eventually the two tools were combined to create a better future.

And a better future it has become. Node.js has become one of the most popular programming environments out there, and it's used for everything from servers to development tools, and even desktop applications.

Back to PhantomJS

PhantomJS's popularity grew as developers realized they now knew how to write code that would automate a browser. Automated testing immediately came to mind, and a sister-tool called CasperJS was created to compliment PhantomJS.

[2] https://codepen.io/falldowngoboone/pen/PwzPYv
[3] https://en.wikipedia.org/wiki/Node.js#History

There was only one problem: PhantomJS was a paired-down version of Google Chrome. Sure, it was fast and had a lot of features of a normal browser, but it wasn't the browser that site visitors would be using.

You could write all the test automation you wanted, but it still wouldn't catch bugs that only occur in Internet Explorer. Remember, at that time, many websites still needed to support the bug-ridden versions of that browser (7, 8 and 9).

So PhantomJS's popularity as a testing tool was always limited. The benefit of automated testing in that single non-traditional browser just never seemed worth the cost.

Enter WebdriverIO

In 2015 I found out about a tool called WebdriverCSS. It was a Visual Regression Testing tool used to compare two screenshots of a page and see if they're visually different from each other. I had tried many tools like this in the past, but this one was unique.

WebdriverCSS was actually a plugin for a library called WebdriverIO. WebdriverIO came with all the great features of PhantomJS (being able to automate a browser through a Node.js script), but had the added benefit of supporting Selenium.

Remember Selenium? That tool from the mid-2000s that I glossed over because I was a little fearful of Java?

WebdriverIO made Selenium approachable to me, and that was literally life-changing (yes, literally). Since investing myself in WebdriverIO, my actual job duties had shifted from primarily front-end development (writing HTML and CSS) to a focus on front-end testing (writing WebdriverIO test scripts).

There were four selling points that convinced me to use WebdriverIO. But before I list my reasons, how about we hear from some other folks[4]:

"Webdriverio is a comprehensive, well documented project with great coverage of the Selenium/Webdriver/Appium specs, as well as loads of very useful helper abstractions. @christian-bromann has been amazing to work with, providing fantastic support and encouraging a helpful community in general." - Goerge Crawford, who is now a core contributor on the project[5]

"The framework of choice at Oxford University Press, I have been using webdriverio since 2016 and it has made life so much easier, especially now with v5, hats off to @christian-bromann and his crew who maintain and continualy support it, keep up the good work guys." - Larry G. - Automation Architect[6]

"Give a QA Engineer a web-automation framework and he might automate some tests. Give him WebdriverIO and he will build a full-fledged, robust automation harness in a matter of days. All

[4] https://github.com/webdriverio/webdriverio/issues/1000
[5] https://github.com/webdriverio/webdriverio/issues/1000#issuecomment-171286700
[6] https://github.com/webdriverio/webdriverio/issues/1000#issuecomment-485683249

jokes aside, WebdriverIO was a blessing in disguise for us @Avira. I'll never look back to other JS-based automation frameworks!" Dan Chivescu, QA Lead[7]

And what about my reasons for choosing WebdriverIO?

WebdriverIO is "Front-end Friendly"

Unlike most other Selenium tools out there, WebdriverIO is written entirely in JavaScript. It's also not restricted to just Selenium, as support for the Chrome Devtools protocol (which we'll talk about later) was added in Version 5.13 of WebdriverIO[8]. This means you can use WebdriverIO without installing Java or running Selenium.

Like I said, I always thought browser automation meant figuring out how to get some complex Java app running. There was also the Selenium IDE, but writing tests through page recordings reminded me too much of WYSIWYG web editors like Microsoft FrontPage (you'll need to look that up if you weren't doing web development in the early 2000's).

Instead, WebdriverIO lets me write in a language I'm familiar with, and integrates with the same testing tools that I use for unit tests (e.g., Mocha).

As a developer, the mental switch from writing the functionality to writing the test code requires minimal effort (since it's all just JavaScript), and I love that.

The other great thing, and this is more to credit WebDriver than WebdriverIO (there is a difference and we'll talk about it), is that I can use advanced CSS selectors to find elements.

xPath scares me for no good reason. Something about slashes instead of spaces just chills my bones. But I don't have to learn xPath.

Using WebdriverIO, I simply pass in my familiar CSS selector and it knows exactly what I'm talking about.

I believe front-end developers should write tests for their own code (both unit and UI), and WebdriverIO makes it incredibly easy.

It Has the Power of Selenium

I always felt held back when writing tests in PhantomJS, knowing that it could never validate functionality in popular, but buggy, browsers like IE.

But because WebdriverIO has built-in support for Selenium, I'm able to run my tests in all sorts of browsers.

Selenium is an incredibly robust platform and an industry leader for running browser automation. WebdriverIO stands on the shoulders of giants by piggy-backing on top of Selenium. All the great things about Selenium are available, without the overhead of writing Java-based tests.

[7] https://github.com/webdriverio/webdriverio/issues/1000#issuecomment-355206891
[8] https://webdriver.io/blog/2019/09/16/devtools.html

It Strives for Simplicity

The commands you use in your WebdriverIO tests are concise and common sense.

What I mean is that WebdriverIO doesn't make you write code to connect two parts together that are obviously meant for each other.

For example, if I want to click a button via a normal Selenium script, I have to use two commands. One to get the element and another to click it.

Why? It's obvious that if I want to click something, I'm going to need to identify it.

WebdriverIO simplifies the 'click' command by accepting the element selector right in to the command, then converts that in to the two Selenium actions needed. That means instead of writing this:

```
driver.findElement(By.id('submit')).click();
```

I can just write this:

```
$('#submit').click();
```

It's so much less mind-numbing repetition when writing tests...

Speaking of simple, I love how WebdriverIO integrates in to Selenium. Instead of creating its own Selenium implementation, it uses the common REST API that Selenium 2.0 provides.

If you haven't worked with API endpoints before, this may not make sense. Don't worry, it's not necessary to understand. But if you're interested, here's how it goes.

WebdriverIO sees that you want to run a command (say "getUrl"). It takes that command and converts it into a request to the Selenium server (it would look like "/session/someSessionIdHere/url"). The Selenium server processes the request and returns the result to WebdriverIO, which then returns the found URL to your code.

Most of WebdriverIO is made up of these small commands living in their own separate small file. This means that updates are easier, and integration into cloud Selenium services like Sauce Labs or BrowserStack are incredibly simple.

Too many tools out there try to reinvent the wheel. I'm glad WebdriverIO keeps it simple and uses what is already out there. This, in turn, helps me easily understand what's going on behind the scenes.

It's Easily Extendable/Scalable

As someone who has spent a considerable portion of their career working for large organizations, it's important to me that the tools I'm using are easily extendable.

I'll have custom needs and will want to write my own functionality. WebdriverIO does a great job at this in two ways:

Custom Commands

There are many commands available by default via WebdriverIO, but there are times when you want to write a custom command just for your application.

WebdriverIO makes this really easy. Just call the "addCommand" function, and pass in your custom steps.

Here's an example from their docs:

```
browser.addCommand('getUrlAndTitle', function () {
    // `this` refers to the `browser` scope
    return {
        url: this.getUrl(),
        title: this.getTitle()
    };
});
```

Now, any time I want both the URL and title in my test, I've got a single command available to get that data.

```
browser.url('http://www.github.com');
const result = browser.getUrlAndTitle();
```

Page Objects

With the 4.x release of WebdriverIO, they introduced a new pattern for writing Page Objects. For those unfamiliar with the term, Page Objects are a way of representing interactions with a page or component.

Rather than repeating the same selector across your entire test suite for a common page element, you can write a Page Object to reference that component.

Then, in your tests, request what you need from the Page Object and it handles it for you. This helps your tests be more maintainable and easier to read.

They're more maintainable because updating selectors and actions occur in a single file.

When a simple HTML change to the login page breaks half your tests, you don't have to find every reference to `input[id="username"]` in your code. You only have to update the Login Page Object and you're ready to go again.

They're easier to read because tests become less about the specific implementation of a page and more about what the page does.

For example, say we need to log in to our website for most of our tests. Without Page Objects, all the tests would begin with:

```
browser.url('login-page');
browser.setValue('#username', 'testuser');
browser.setValue('#password', 'hunter2');
browser.click('#login-btn');
```

With Page Objects, that can become as simple as:

```
LoginPage.open();
LoginPage.login('testuser', 'hunter2');
```

No reference to specific selectors. No knowledge of URLs. Just self-documenting steps that read out more like instructions than code.

Now, Page Objects aren't a new idea that WebdriverIO introduced. But they way they've set it up to use plain JavaScript objects is brilliant. There is no external library or custom domain language to understand. It's just JavaScript and a little bit of prototypical inheritance. (We'll definitely cover Page Objects in more detail later in this book.)

Summing It Up

I wouldn't call myself a real software tester. I'm far too clumsy to be put in charge of ensuring a bug-free launch.

Yet, I can't help but love what WebdriverIO provides me, and I'm a fan of what's going on with the project and its future. Hopefully this book helps you feel the same way.

1.1.3 Technical Details

Versions

Because technology changes fast, it's good to cover what versions of software I used when creating these exercises.

- Node.js: v12.16.1
- WebdriverIO: 6.3.5
- Java 8 (optional)

Git Repository

To download code samples for the main part of the book, visit the official git repo[9].

[9] https://github.com/klamping/wdio-book-examples

Where to check for updates/corrections

I've written the book using the most recent version of these technologies as I could. However, with an ever changing landscape, updates will need to be made.

You can see a list of changes by visiting the book's changelog[10].

Where to find help

I've done my best to make the material clear and understandable, but I'm sure to have fallen short in some areas. To get extra help, try one of these three options:

- This Book's GitHub Issues Page[11]
- **Gitter** - WebdriverIO runs an official chat room for folks seeking help with the tool itself.
- **My personal email** - Although I'm usually slow to respond, you can reach out to me at kevin at learnwebdriverio dot com. I'll do my best to get back to you in a timely manner.

Errata

I've worked to ensure that the content of this book is accurate and that the examples actally run. However, I definitely can make mistakes (that's why I'm a fan of testing after all). Also, technology changes, and what worked when I wrote this doesn't necessarily work anymore.

If you find errata, please submit an issue on the GitHub repository[12] and I'll work to get the error resolved.

Technical Knowledge Requirements

What sort of skill level do you need in order to understand the material covered in this book?

While I've worked to explain the many concepts introduced through UI testing, I do make the assumption that readers are familiar with the following technologies:

- HTML (basic understanding of how HTML structure is composed)
- CSS (basic understanding of how CSS selectors work)
- JavaScript
- NodeJS
- Terminal/Shell commands

[10] https://github.com/klamping/ui-testing-book/blob/master/CHANGELOG.md
[11] https://github.com/klamping/ui-testing-book/issues
[12] https://github.com/klamping/ui-testing-book/issues

In regards to JavaScript and NodeJS, here are some important concepts to understand:

- Data types: strings, objects, arrays…
- Functions: how to call and create them
- Conditionals: if/else, ternary
- ES6 updates:
  - const & let
  - Array functions: map, forEach
  - Classes (we will cover this in more detail in the page objects section)

That said, you don't need to know how to create a Node.js server, build a website or use the latest JavaScript framework.

Where can I freshen up?

While I won't be covering it in this book, here's a few free resources you might find helpful if you'd like to refresh your knowledge:

- HTML Crash Course For Absolute Beginners[13]
- CSS Crash Course For Absolute Beginners[14]
- JavaScript Basics Course (YouTube Playlist)[15]
- JavaScript for Testers (YouTube Playlist)[16]
- JavaScript Course (Codecademy)[17]
- Must-know JavaScript Features (YouTube Playlist)[18]
- Start Using ES6 Today (Presenation)[19]
- Linux Terminal (YouTube)[20]
- Windows Terminal (YouTube Playlist)[21]
- NPM Crash Course[22]
- Beginner JavaScript Course (Paid)[23]

[13] https://www.youtube.com/watch?v=UB1O30fR-EE
[14] https://www.youtube.com/watch?v=yfoY53QXEnI
[15] https://www.youtube.com/playlist?list=PLWKjhJtqVAbk2qRZtWSzCIN38JC_NdhW5
[16] https://www.youtube.com/playlist?list=PLzDWIPKHyNmLxpL8iQWZXwl_ln0BgckLt
[17] https://www.codecademy.com/learn/learn-javascript
[18] https://www.youtube.com/playlist?list=PL0zVEGEvSaeHJppaRLrqjeTPnCH6vw-sm
[19] https://www.youtube.com/watch?v=493p5FSFHz8
[20] https://www.youtube.com/watch?v=oxuRxtrO2Ag
[21] https://www.youtube.com/watch?v=MBBWVgE0ewk&list=PL6gx4Cwl9DGDV6SnbINlVUd0o2xT4JbMu
[22] https://www.youtube.com/watch?v=jHDhaSSKmB0
[23] https://beginnerjavascript.com/

1.2 Installation and Configuration

1.2.1 Software Requirements

While WebdriverIO is a Node.js based system, there are a few other tools needed to run the tests. You'll want:

- A recent version of Node.js (8+)
- A text-editor (I use Sublime Text 3[24], but Atom[25], Webstorm[26] and VSCode[27] are other great options)
- A terminal/command line tool (I use iTerm[28] with Oh My Zsh[29] thrown on top)
- A Webdriver-compliant browser for testing (Chrome is what we'll be using)

Optionally, you may also want to install Java 8[30]. This will allow you to run selenium-standalone[31], which gives you the ability to test on different browsers in the same test run (e.g., both Firefox AND Chrome).

Installing Node.js

There are many great tutorials for how to install Node.js on a variety of systems. A quick search should bring up many results should you need additional help with this installation.

Overall though, there are two common ways to install Node.js.

Install via official site:

Go to nodejs.org[32] and download the release labelled "Recommended For Most Users". This will start with an even number (e.g., 10.19.0). Be aware that releases starting with odd numbers (e.g., 11.10.0) are not supported long term, so while they may have the latest features, they will stop recieving support and updates after 6 months. For more information on this, have a read through the Node.js release plan[33].

[24] https://www.sublimetext.com/3
[25] https://atom.io/
[26] https://www.jetbrains.com/webstorm
[27] https://code.visualstudio.com/
[28] https://iterm2.com/
[29] https://ohmyz.sh/
[30] https://java.com/en/download/
[31] https://github.com/vvo/selenium-standalone
[32] https://nodejs.org/
[33] https://github.com/nodejs/Release#release-plan

Install via a 'version manager'

The main reason for using a version manager is "the future". In "the future", you're probably going to want to update your version of Node.js to a more recent release. While it's possible to manually uninstall the old version, then install the new one using the official site, it can be a little tedious to do so on a regular basis.

With a version manager, it takes care of this for you. You simply ask for the Node.js version you want, and it does all the grunt work.

Two popular version managers are:

- NVM[34] (this is what I use)
- N[35]

Installation instructions are on both of those sites, so I won't copy them over here (plus any copied instructions are likely to be out-of-date by the time you read this.)

Getting Your Terminal Ready

As I mentioned, you'll need to know the basics of how to use a terminal/command prompt in order to take advantage of all the WebdriverIO has to offer.

All major Operating Systems provide a pre-installed terminal for you to use. These are:

- Windows 10: cmd.exe or Powershell
- Mac OSX: Terminal
- Linux: konsole, gnome-terminal, terminal or xterm

In the terminal of your choice, ensure you have Node.js installed correctly by running `node -v` in it. This should output the version number of Node.js that you have installed. If you see a message like `command not found: node`, then something went wrong with your installation and you'll need to debug it.

A Note for Windows Users

The commands you use in the default Windows terminal (cmd.exe) are different from what I'll be showing in my code samples.

Some examples:

- Instead of using `ls` to print the contents of a directory, you need to use `dir`

[34] https://github.com/creationix/nvm
[35] https://github.com/tj/n

- Windows uses a back slash `\` instead of a forward slash `/` for path commands (e.g., `node_modules\.bin\` versus `node_modules/.bin/`)
- Windows users are also required to enter `.\` before every function call that involes a path (e.g., `dir .\node_modules\.bin\` instead of `ls node_modules/.bin/`)
- The way you define environment variables is different (we'll go into detail on this later)

For a more comprehensive list of differences, RedHat has put together a comparison chart[36].

I'll try to provide the Windows equivalent the first time I introduce a command. However, if you'd like to stick with the commands that I use throughout the book, consider installing an alternative console. Here are some suggestions:

- cmder[37]
- Microsoft Terminal[38]
- clink[39]
- ConEmu[40]
- Cygwin[41]

These terminals use bash-style commands, which is what I use in my examples.

1.2.2 Browsers and "Driving" Them

We normally use browsers by clicking with our mouse and typing with our keyboard. That works well for humans, but doesn't make sense when trying to write automated tests.

Instead of building some sort of physical robot that can control a mouse and type on a keyboard, we invented software that mimics these actions. Selenium RC was one of the original tools to do this. WebDriver, which was also developed around the same time as Selenium RC, became a popular alternative. In 2009, the two teams combined forces to create Selenium WebDriver.

Over the years, standardization on the Selenium WebDriver commands occurred, and now there is an official W3C spec for WebDriver[42]. The teams behind the browsers we use have also started to implement that spec (e.g., ChromeDriver), allowing the use of WebDriver commands outside of Selenium.

Recently, Chrome has released support for their own protocol called "Chrome DevTools". WebdriverIO has added support for this protocol through the devtools package[43]. The industry has evolved its

[36] https://web.archive.org/web/20170715114407/https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Step_by_Step_Guide/ap-doslinux.html
[37] https://cmder.net/
[38] https://github.com/microsoft/terminal
[39] http://mridgers.github.io/clink/
[40] https://conemu.github.io/
[41] https://www.cygwin.com/
[42] https://w3c.github.io/webdriver/
[43] https://www.npmjs.com/package/devtools

tooling over the years and WebdriverIO has kept up giving you the flexibility to pick what works best.

This is why WebdriverIO has the tagline "Next-gen browser and mobile automation test framework for Node.js," excluding any specific protocol. While you can use Selenium in your WebdriverIO tests, it's really just about running commands through any protocol with support. WebdriverIO doesn't want to box you in to a specific solution, and we appreciate that :)

Now, it's important not to confuse terms, so to be clear, the following list contains many *different* things:

- **WebDriver**: A technical specification defining how tools should work.
- **The Selenium Project**: An organization providing tools used for automated testing.
- **Selenium/Selenium WebDriver**: Language-specific bindings for the WebDriver spec that are officially supported by the Selenium project, like the NPM package selenium-webdriver[44].
- **Browser Driver**: Browser specific implementations of the WebDriver spec (e.g., ChromeDriver, GeckoDriver, etc).
- **Selenium Server**: A proxy server used to assist a variety of browser drivers.
- **Chrome Devtools Protocol**: A protocol that allows for tools to instrument, inspect, debug and profile Chromium, Chrome and other Blink-based browsers. (Project Homepage[45])
- **Puppeteer** A Node.js library which provides a high-level API to control Chrome or Chromium over the DevTools Protocol.
- **WebdriverIO**: A test framework written in Node.js that provides bindings for tools like Selenium Server, Chrome DevTools (via Puppeteer) and WebDriver-based browser drivers (e.g., ChromeDriver).

That's a fair number of terms to keep in mind. I don't have a great suggestion for how to memorize everything, but maybe just reference this section when you need a good reminder.

What Do We Use?

Right now there are essentially two different approaches to how you can automate a browser. One uses the official W3C web standard (i.e., WebDriver) and the other uses native browser interfaces that some of the browsers expose (e.g., Chrome DevTools).

The WebDriver protocol is the de-facto standard automation technique. It allows you to not only automate all desktop browsers, but also run automation on mobile devices, desktop applications or even Smart TVs. This gives us a tremendous amount of power in being able to run our tests across a variety of systems.

On the other side of things, that are many native browser interfaces to run automation on. In the past, every browser had its own (often not documented) protocol. But these days a lot of browsers,

[44] https://www.npmjs.com/package/selenium-webdriver
[45] https://chromedevtools.github.io/devtools-protocol/

including Chrome, Edge and soon Firefox, come with a somewhat unified interface revolving around the Chrome DevTools Protocol.

While WebDriver provides true cross browser support and allows you to run tests on a large scale in the cloud using vendors like Sauce Labs, native browser interfaces often allow many more automation capabilities like listening and interacting with network or DOM events while often being limited to a single browser only. These native interfaces also run much faster than their WebDriver counterparts, as they're a bit "closer to the metal".

We're going to take a minute to look at how to get set up with a few of these solutions. Throughout the book though, our examples with use the WebDriver protocol, since it's the most popular standard in use as of this writing. Thankfully though, it's very easy to switch between protocols in WebdriverIO, so we're not boxing ourselves in by picking one or the other.

Using the Chrome DevTools Protocol

Starting with Version 6, WebdriverIO now provides support for the Chrome DevTools protocol by default. This means that to run a local test script, you don't need to download a driver or Selenium. When running your test, WebdriverIO will first check if a browser driver is running and available. If not, it falls back to using Puppeteer (assuming you have a Chromium, Chrome or other Blink-based browser installed). Seeing that Chrome is the most popular browser in use as of the writing of this book, chances are you already have it installed.

To use the Chrome DevTools protocol for your tests, simply ensure you have Chrome (or an equivalent) installed. Everything else is handled by default.

How To Use a 'Driver'?

If you are interested running your tests for a browser that isn't based on the 'Blink' engine (or just prefer to stick with the WebDriver standard), you'll want to use some sort of WebDriver-based browser driver. There are several WebDriver clients available, Selenium Server being the most popular. Let's walk through setting up one of these clients so that you can start writing tests.

All major browsers have 'drivers' that mostly follow the WebDriver spec (unfortunately there are still differences between them).

Here are the drivers for each major browser:

- GeckoDriver[46] for Firefox (v48 and above)
- ChromeDriver[47] for Chromium
- EdgeDriver[48] for Microsoft Edge

[46] https://github.com/mozilla/geckodriver
[47] https://chromedriver.chromium.org/downloads
[48] https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/

- SafariDriver[49] for Safari (implemented as a Safari browser extension)
- IEDriver[50] for Internet Explorer

To see how your favorite browser driver stacks up in regards to WebDriver support, check out the Web Platform Tests page[51]. This site runs regular tests against clients implementing the WebDriver spec, and provides the results showing how well they support it.

There are drivers available for mobile testing (e.g., Appium), but they won't be covered in this book.

Installing and Running ChromeDriver

Installation instructions for these clients can be found on their respective websites, but in many cases, you can search on npmjs.org for Node.js-based installation tools.

For example, you can download and install ChromeDriver using the NPM ChromeDriver package[52].

To install the NPM package, run

```
npm install -g chromedriver
```

Terminal Output from installing ChromeDriver Globally

You can then start a ChromeDriver instance by running:

```
chromedriver
```

[49] https://developer.apple.com/documentation/webkit/about_webdriver_for_safari
[50] https://github.com/SeleniumHQ/selenium/wiki/InternetExplorerDriver
[51] https://wpt.fyi/results/webdriver/tests
[52] https://www.npmjs.com/package/chromedriver

Terminal Output from manually running ChromeDriver

This instance will continue to run until you stop it. To do that, issue an 'exit' command by pressing the `ctrl+c` key combo.

Installing and Running the Selenium Standalone Server

First off, if you're going to be using this method, you need to ensure you have a recent version of Java installed on your computer. Be sure to take care of that before trying the following. None of the content of this book requires a Selenium instance, so feel free to skip this section.

If you're looking to run tests on a variety of browsers, you'll probably want to check out what the Selenium Server project does. It offers a 'hub' that allows you to start multiple browser instances and control them all through one single location.

While it is possible to manually download and start a selenium server[53], there is an NPM tool called "selenium-standalone[54]" that makes this much easier.

To install and use it, run the following command in your terminal:

```
npm i -g selenium-standalone
```

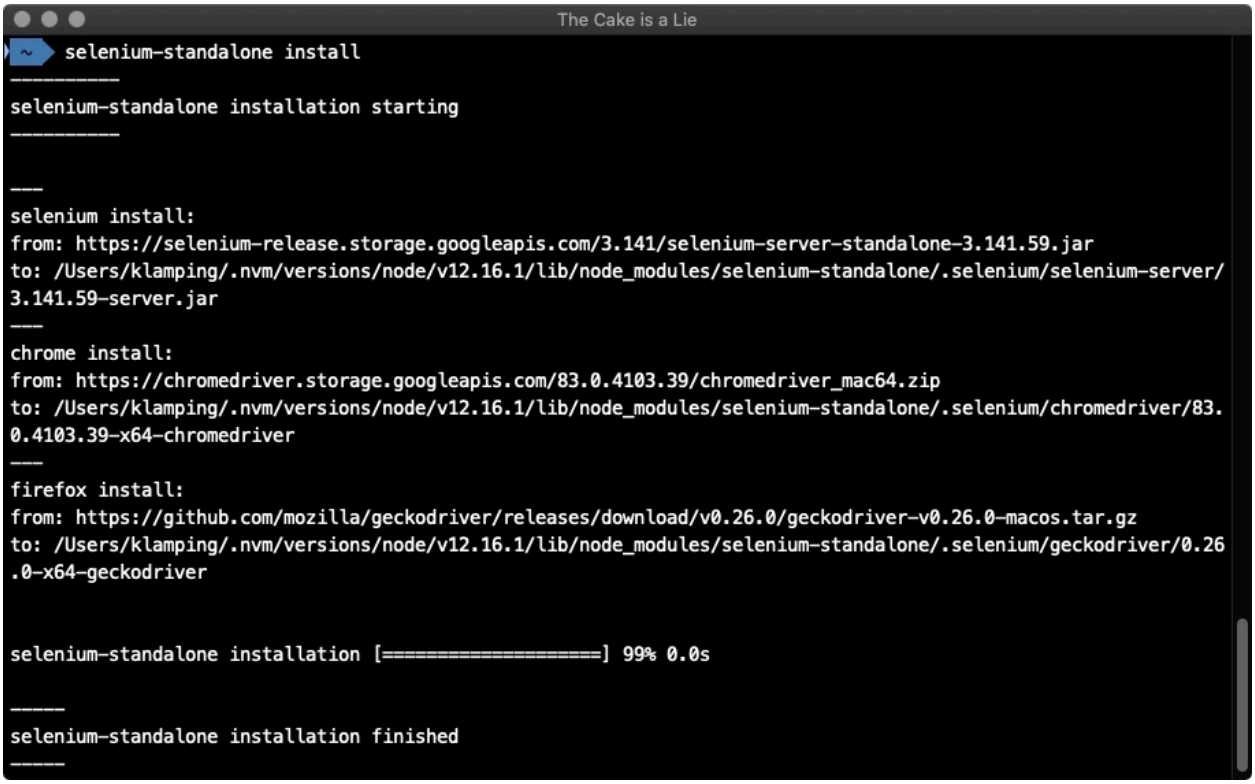
This will make a global command available called `selenium-standalone`. With this command, we can do the following:

1. Install the four supported WebDriver clients (ChromeDriver, FirefoxDriver, IEDriver, Microsoft Edge Driver)
2. Start a Selenium Server that acts as a proxy to these clients

To run the install, issue this command:

```
selenium-standalone install
```

[53] https://www.seleniumhq.org/download/
[54] https://github.com/vvo/selenium-standalone



Terminal Output from running selenium-standalone 'install' command

You should only need to do this once (although you may need to run it again after driver updates occur).

Then, to start your server, run:

```
selenium-standalone start
```

Terminal Output from running selenium-standalone 'start' command

This server will run until it receives an `exit` command (similar to how ChromeDriver works). You can issue that command with the `ctrl+c` key combo.

We'll talk more about using Chrome DevTools, the Selenium Standalone Server and ChromeDriver (including services to integrate them with WebdriverIO) in a little bit.

1.2.3 Installing WebdriverIO and Basic Usage

The time has finally come! We've laid all the groundwork to understand the nuts and bolts behind UI testing. Now it's time to do some!

To start off, we're going to create a new folder for our first example. In a directory of your choice, make a new folder called `wdio-standalone`:

```
mkdir wdio-standalone
```

Why `wdio-standalone`?

Well, WebdriverIO allows you to use it through two modes. The first, which we're going through here, is called "standalone" mode. It's meant as a simple way to use WebdriverIO, and allows you to build wrappers around the tool.

"Testrunner" mode, which we'll cover in the next section, is a bit more complicated. It provides an entire set of tools and hooks for full-fledged integration testing. I mentioned that standalone mode allows you to build wrappers around it. Well, the testrunner is essentially that.

Right now, just to introduce you to WebdriverIO, we're going to use the standalone runner. This is only for this exercise though, and we'll be upgrading to the testrunner soon.

With all that said, let's 'move' our terminal into this `wdio-standalone` folder:

```
cd wdio-standalone
```

(For Windows, it's the same command for both actions)

Inside our new folder, we're going to initialize it as an NPM project. This will allow us to save the project dependencies that we'll be installing through NPM.

To do that, run:

```
npm init -y
```

The `-y` will answer 'yes' to all the prompts, giving us a standard NPM project. Feel free to omit the `-y` if you'd like to specify your project details.

With that out of the way, let's install WebdriverIO:

```
npm install webdriverio
```

Now is a good time to mention that WebdriverIO is split into multiple NPM packages. We'll be looking at those packages in detail later on, but note that installing `webdriverio` via the command above does not give you everything.

What it does give us is a Node.js module that we can use inside of a Node.js file. Let's use that.

First, we'll create a new file called 'test.js':

```
touch test.js
```

On Windows, that command is:

```
type nul > test.js
```

Now we have a file to add our first test to. Go ahead and open that file up in the text editor of your choice.

Next, we'll copy the example given on the official WebdriverIO website. Throw the following code into your `test.js` file and save it:

test.js

```
const { remote } = require('webdriverio');

(async () => {
    const browser = await remote({
        capabilities: {
            browserName: 'chrome'
        }
    });

    await browser.url('https://webdriver.io');

    const title = await browser.getTitle();
    console.log('Title was: ' + title);

    await browser.deleteSession();
})().catch((e) => console.error(e));
```

Here's a quick overview of the file:

1. We load the `remote` object from the WebdriverIO package.
2. We wrap our code in an `async` function so we can use `await` statements.
3. We create a new session using `remote`, saving the reference to a `browser` object which we use to send commands.
4. We send a `url` command, requesting the browser go to the WebdriverIO website.
5. We then get the title of the page, storing it as a local variable.
6. The title of the page is logged to the terminal.
7. The session is ended, since we're done with our test.
8. A simple catch statement is added in case anything goes wrong.

Okay, that's what it does; let's run it to see it in action.

To do that, we need to have a browser available to running. This can be through the built-in support from a Chrome install, through a specific browser driver, or through Selenium server.

In "Browsers and 'Driving' Them", I detailed Chrome DevTools, along with how to install and run ChromeDriver and the selenium-standalone NPM package. Now let's put that knowledge to use.

Running Through Chrome DevTools

So long as you have Chrome (or a Blink-based browser installed), there's really nothing you need to do here for installation/start-up. All you need to do is run your test file through the node CLI. We do that by telling Node.js to execute our test file. That command looks like:

```
node test.js
```

After a second, you should see a Chrome browser pop-up for a moment, and some similar output in your terminal:

```
2020-07-24T21:03:30.968Z INFO webdriverio: Initiate new session using the
devtools protocol 2020-07-24T21:03:30.974Z INFO devtools: Launch Google Chrome
with flags: --disable-extensions --disable-background-networking
--disable-background-timer-throttling --disable-backgrounding-occluded-windows
--disable-sync --metrics-recording-only --disable-default-apps --mute-audio
--no-first-run --disable-hang-monitor --disable-prompt-on-repost
--disable-client-side-phishing-detection --password-store=basic
--use-mock-keychain --disable-component-extensions-with-background-pages
--disable-breakpad --disable-dev-shm-usage --disable-ipc-flooding-protection
--disable-renderer-backgrounding
--enable-features=NetworkService,NetworkServiceInProcess
--disable-features=site-per-process,TranslateUI,BlinkGenPropertyTrees
--window-position=0,0 --window-size=1200,900 2020-07-24T21:03:31.535Z INFO
devtools: Connect Puppeteer with browser on port 50210 2020-07-24T21:03:32.772Z
INFO devtools: COMMAND navigateTo("https://webdriver.io/")
2020-07-24T21:03:35.366Z INFO devtools: RESULT null 2020-07-24T21:03:35.373Z
INFO devtools: COMMAND getTitle() 2020-07-24T21:03:35.377Z INFO devtools: RESULT
WebdriverIO · Next-gen browser and mobile automation test framework for Node.js
Title was: WebdriverIO · Next-gen browser and mobile automation test framework
for Node.js 2020-07-24T21:03:35.380Z INFO devtools: COMMAND deleteSession()
2020-07-24T21:03:35.382Z INFO devtools: RESULT null
```

Congrats, you've just run your first WebdriverIO test!

Notice that the first line says "Initiate new session using the devtools protocol". That will change depending on which protocol you use.

If you choose to go with the DevTools protocol, support for the various commands does differ from WebDriver. While in general everything is supported the same, there are still differences which can cause hiccups along the way. The book is written with support for the WebDriver protocol, so if you choose to stick with the DevTools protocol, expect some differences.

Now let's look at running via ChromeDriver.

Running in ChromeDriver

The basic idea is the same, although we do need to tweak our settings just a little bit.

This is a little technical, but by default, a ChromeDriver server uses port `9515` to listens for commands (e.g., http://localhost:9515)

But by default, WebdriverIO expects the WebDriver server to be running on port `4444`.

So, we can either override the WebdriverIO defaults, or tell our ChromeDriver server to use port `4444`.

It's most useful to see how to overwrite the WebdriverIO defaults, so let's do that next. If you are interested, you can do the latter by running `chromedriver --port=4444` when starting the ChromeDriver server.

Back in your `test.js` file, take a look at lines 4-8:

```
const browser = await remote({
    capabilities: {
        browserName: 'chrome'
    }
});
```

What we're doing here is creating a new `remote` WebDriver session and telling it that we want to open up the 'chrome' browser. We'll get into capabilities at a later point, so don't worry too much about it right now.

What we will worry about is how to customize that 'remote' session to use post 9515 instead of the default `4444`.

Along with customizing the `capabilities`, there are a number of other options available to us. The official documentation[55] gives the entire list, but there are three options that we're going to change:

hostname Host of your driver server. Type: String Default: localhost

port Port your driver server is on. Type: Number Default: `4444`

[55] https://webdriver.io/docs/options.html#webdriver-options

path Path to driver server endpoint. Type: String Default: /wd/hub

We're only going to be looking at the `port` option right now, but I wanted to mention all three as they're related and important to know about (which we'll see when we get to the Selenium Standalone instructions.)

So to use a custom port, we pass it in as an option to the `remote` function:

```
const browser = await remote({
    port: 9515,
    capabilities: {
        browserName: 'chrome'
    }
});
```

Note that it's a number, not a string (i.e., `9515` versus `'9515'`). If you try using a string, you will get an error of `Error: Expected option "port" to be type of number but was string`.

If you still have your ChromeDriver instance running from before, leave it up and running (you can check http://localhost:9515/ to see if it gives you a response). If not, start an instance in a separate terminal window.

With our WebdriverIO settings updated and ChromeDriver ready to go, we can call our test script again.

Run `node test.js` one more time and validate that it all works as expected. The output should be similar to before:

```
2020-07-18T15:29:43.175Z INFO webdriverio: Initiate new session using the
webdriver protocol 2020-07-18T15:29:43.183Z INFO webdriver: [POST]
http://localhost:9515/session 2020-07-18T15:29:43.183Z INFO webdriver: DATA {
capabilities: { alwaysMatch: { browserName: 'chrome' }, firstMatch: [ {} ] },
desiredCapabilities: { browserName: 'chrome' } } 2020-07-18T15:29:46.174Z INFO
webdriver: COMMAND navigateTo("https://webdriver.io/") 2020-07-18T15:29:46.175Z
INFO webdriver: [POST]
http://localhost:9515/session/8c1bfcbb0617b87676343fe9c658fc93/url
2020-07-18T15:29:46.175Z INFO webdriver: DATA { url: 'https://webdriver.io/' }
2020-07-18T15:29:48.210Z INFO webdriver: COMMAND getTitle()
2020-07-18T15:29:48.210Z INFO webdriver: [GET]
http://localhost:9515/session/8c1bfcbb0617b87676343fe9c658fc93/title
2020-07-18T15:29:48.524Z INFO webdriver: RESULT WebdriverIO · Next-gen browser
and mobile automation test framework for Node.js Title was: WebdriverIO ·
Next-gen browser and mobile automation test framework for Node.js
2020-07-18T15:29:48.525Z INFO webdriver: COMMAND deleteSession()
2020-07-18T15:29:48.525Z INFO webdriver: [DELETE]
http://localhost:9515/session/8c1bfcbb0617b87676343fe9c658fc93
```

While there are a few differences from before, the important one is the first line. See how it says it's using the WebDriver protocol? That's how we can know things are working as we want.

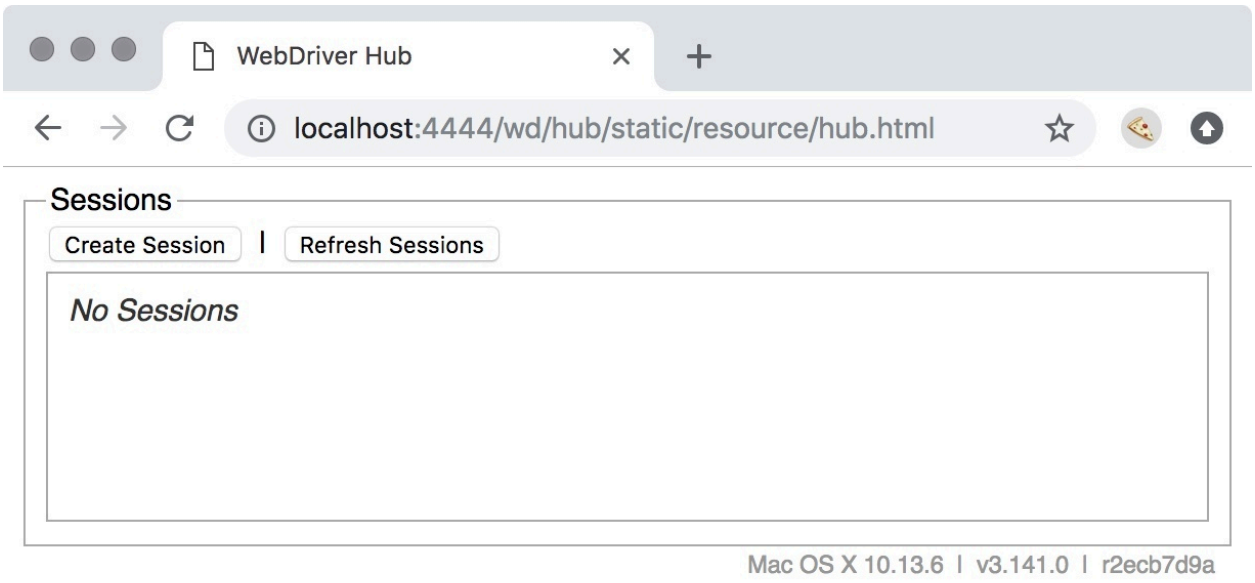
Running Through Selenium Standalone

*Note: Selenium Standalone is **not** the same thing as WebdriverIO Standalone mode. They simply share the same name to describe their "independent" nature.*

If you already have your Selenium server running from before, great! If not, open up a new terminal window and run `selenium-standalone start`.

Aside from seeing the server running in your terminal, you can check that you have a Selenium instance up and running by visiting the following URL in your browser: http://localhost:4444/wd/hub[56]

You should see a website looking a lot like this:



Preview of Selenium Standalone 'hub' page

Note: If you get a `404` error, something went wrong while starting your server, and you'll need to resolve it before proceeding.

The next thing we need to do is configure WebdriverIO to use the Selenium server.

Unlike ChromeDriver, when you start Selenium, it runs on port `4444` by default. That means we can comment out or remove the `port` option we had for our ChromeDriver usage.

That said, Selenium waits for requests to come through the `/wd/hub` URL endpoint/path (hence `http://localhost:4444/wd/hub` being mentioned before). But if you recall from our options, WebdriverIO doesn't have that at it's default `path` option (which is just `/`).

[56] http://localhost:4444/wd/hub

To use Selenium, we'll need to update that path setting to match where Selenium defaults to:

```
const browser = await remote({
    path: '/wd/hub',
    capabilities: {
        browserName: 'chrome'
    }
});
```

Running our test once more with `node test.js`, you should see similar output:

```
2020-07-18T15:33:51.348Z INFO webdriverio: Initiate new session using the webdri
ver protocol
2020-07-18T15:33:51.356Z INFO webdriver: [POST] http://localhost:4444/wd/hub/ses
sion
2020-07-18T15:33:51.356Z INFO webdriver: DATA { capabilities:
   { alwaysMatch: { browserName: 'chrome' }, firstMatch: [ {} ] },
  desiredCapabilities: { browserName: 'chrome' } }
2020-07-18T15:33:54.807Z INFO webdriver: COMMAND navigateTo("https://webdriver.i
o/")
2020-07-18T15:33:54.807Z INFO webdriver: COMMAND navigateTo("https://webdriver.i
o/")
2020-07-18T15:33:54.808Z INFO webdriver: [POST] http://localhost:4444/wd/hub/ses
sion/8e71129f6b0b4cb3cd09bd17b06bd6ca/url
2020-07-18T15:33:54.808Z INFO webdriver: DATA { url: 'https://webdriver.io/' }
2020-07-18T15:33:57.275Z INFO webdriver: COMMAND getTitle()
2020-07-18T15:33:57.275Z INFO webdriver: [GET] http://localhost:4444/wd/hub/sess
ion/8e71129f6b0b4cb3cd09bd17b06bd6ca/title
2020-07-18T15:33:57.288Z INFO webdriver: RESULT WebdriverIO · Next-gen browser a
nd mobile automation test framework for Node.js
Title was: WebdriverIO · Next-gen browser and mobile automation test framework f
or Node.js
2020-07-18T15:33:57.289Z INFO webdriver: COMMAND deleteSession()
2020-07-18T15:33:57.289Z INFO webdriver: [DELETE]
 http://localhost:4444/wd/hub/session/8e71129f6b0b4cb3cd09bd17b06bd6ca
```

Again, line one shows that we're using the WebDriver protocol. And notice on line two that it posts to `http://localhost:4444/wd/hub/session`, using the path we provided.

If you instead of all that output you see an error that includes `RequestError: connect ECONNREFUSED 127.0.0.1:4444`, this means your Selenium server wasn't running. Start it back up and try again.

Leaving It at That

This will be the end of our little test file. We're not going to be updating it anymore, and will in fact be leaving this whole `wdio-standalone` folder behind.

Why? Because we're moving on to a much better way of using WebdriverIO through its test runner. That's coming up next.

1.2.4 Upgrading to the WDIO Test Runner

There are a plethora of test automation tutorials out there. Many of them cover the basics of installing and running a specific tool, but lack the depth to teach you how to test on a larger scale.

Here are several things that most tutorials omit (mostly for the sake of time):

- How to manage settings across files
- How to integrate your tests with other tools
- How to extend these test tools with new features
- How to debug tests when they inevitably fail

You probably wouldn't think about these questions when first starting out (I know I didn't). It isn't until you've invested a fair amount of time that you're going to start feeling some pain from managing all your tests.

Unfortunately, that pain can stop you from growing your test suite. Instead of writing new tests, you're stuck writing functionality getting everything working together smoothly.

This is where WebdriverIO goes above and beyond many other test tools. Instead of just being a Node.js tool for running test commands, it provides over 25 different packages that fit specific needs.

Looking for a way to organize your code in a test framework like Mocha or Jasmine? That's already set up for you.

Want to integrate your tests with a third-party cloud testing platform like Sauce Labs or BrowserStack? WebdriverIO has services for that.

Need to output the results of your tests through a reporter like Allure or Sumo Logic? There are packages for that as well.

WebdriverIO does all of this through a tool it calls the 'test runner.'

Instead of using WebdriverIO in 'standalone' mode, we run everything through the test runner, which integrates all the various moving parts together.

Standalone mode (how we wrote our first test) isn't bad; in fact, the test runner uses the standalone package to run everything. It's just that the test runner adds a great deal of functionality on top of it.

It also simplifies a lot of the complexity introduced when trying to solve the problems I listed above.

Here are some other benefits of the test runner:

- **Implicit async/await usage**: WebDriver commands are 'asynchronous' by nature. Your code needs a way to handle that. In JavaScript, you do that with the `async` & `await` keywords. Unfortunately, that concept can be a bit complex to understand. But if you aren't familiar with the terms `async` & `await`, that's perfectly fine, the test runner allows you to not worry about it. This lowers the barrier to entry in writing your first real test (it's one less thing to learn). And we will cover what async/await means at a later point in time.
- **Extra hooks**: Hooks allow you to run code during specific parts of your test flow. Fourteen separate hooks let you integrate code into different parts of your test run, giving you full control on how things execute.
- **Parallel test runs**: UI tests can certainly be slow. By running your tests in parallel (i.e., multiple tests at one time), you're able to reduce the time needed to run your test suite. The test runner takes care of the details.
- **Suites**: Chances are, on a site large enough, you're going to have multiple groups of functionality to test. With suites, you're able to divide sections of your test suite and run them individually. This is especially useful when debugging or developing new tests.

For all those reasons, and many more, the remaining portion of our examples will be carried out through the test runner. It makes the most sense for the majority of users.

Let's Get Going

So, how do we go about using the test runner?

Well, let's set up a new folder for writing our tests in. Outside the `wdio-standlone` folder you created, make a new one called `wdio-testrunner`:

```
mkdir wdio-testrunner && cd wdio-testrunner
```

From inside that folder, initialize a new NPM project:

```
npm init -y
```

That will create a `package.json` file that we'll use for saving dependencies (among other things).

The test runner is part of the WDIO CLI package. CLI stands for "Command Line Interface," which is a way of using a program controlled by text input in the terminal.

We install this interface using NPM:

```
npm install @wdio/cli
```

All the official WebdriverIO packages are namespaced under the `@wdio` scope. This makes it easier to distinguish between officially supported packages and third-party work contributed by the WebdriverIO community (the `webdriver` and `webdriverio` packages are officially supported, even though they don't use the `@wdio` namespace).

One thing we won't do is install the `webdriverio` package as we did before. That's because this package is included when installing the CLI package. Unless you're doing very custom work, you likely won't need to install the main `webdriverio` package. The CLI tool is a wrapper around the main `webdriverio` functionality, so it's your new go-to tool for using the overall WebdriverIO tool suite.

Okay, enough about that. Let's get on with using this fancy new CLI tool.

Having installed the tool, you now have access to the `wdio` command in your terminal. This command lets you do a few things:

- Run a setup utility to create a common configuration file
- Run test suites using said configuration files
- Run commands through a REPL, which is an interactive terminal utility for running one-off WebdriverIO commands

We'll take a look at the third option later, but for now, let's check out the first two.

Setting Up Our Configuration

In our original example, we defined a few options when initializing our WebdriverIO session. As a reminder, the code looked like:

```
const browser = await remote({
    // port: 9515, // used for chromedriver
    // path: '/wd/hub', // used for selenium standalone
    capabilities: {
        browserName: 'chrome'
    }
});
```

All of the information inside the `remote` function call has to do with configurations on how we want our WebDriver session to be set up. If we were to stick with using standalone mode, we'd need to either duplicate this configuration across all of our test files, or figure out a way to share it through a common file.

Well, the test runner figures you'd probably want to go the route of sharing the configuration, and takes the step to do that work for you. By convention (meaning you can override this), your configurations will be stored inside a file called `wdio.conf.js`. Using the configurations we set above, that file may look like this:

```
exports.config = {
    // port: 9515, // used for chromedriver
    // path: '/wd/hub', // used for selenium standalone
    capabilities: {
        browserName: 'chrome'
    }
};
```

The main difference from before is the `exports.config` part. That gives us the ability to share our configuration for all our tests.

One thing missing from the previous code sample is that there are a lot more options in the `wdio.conf.js` file. The test runner provides support for many different utilities (e.g., custom reporters and services), so the file is usually over 200 lines long. That's a lot of options to set!

To help you with this, a configuration utility was provided through the CLI interface. This utility is run either when you specifically ask for it using `wdio config`, or if you try to run the `wdio` command and it can't find an existing configuration file.

That's enough talk, let's give it a shot.

Since `@wdio/cli` is a command-line utility, it's set up to install into a special `.bin` folder inside our `node_modules` folder. This gives us two ways to run the utility:

```
./node_modules/.bin/wdio
```

or

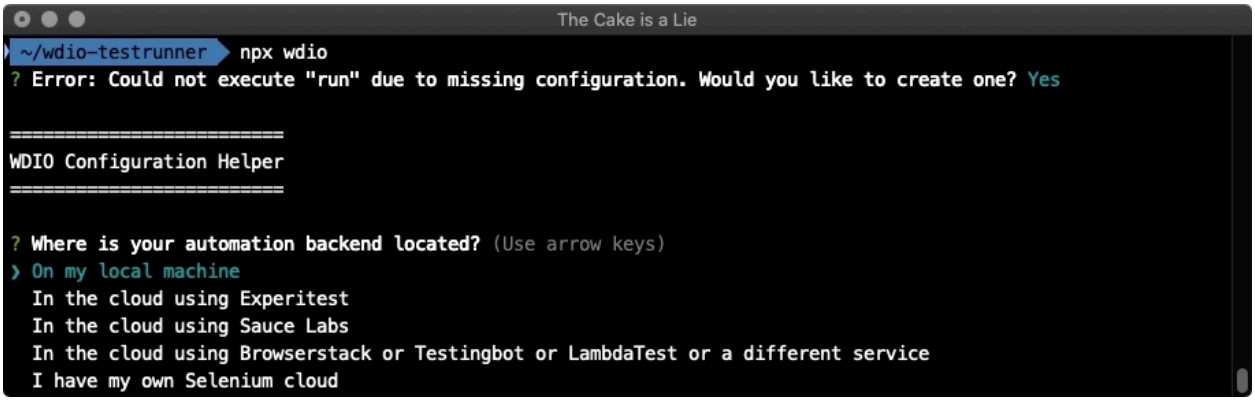
```
npx wdio
```

Either way works, and they both do basically the same thing. Personally, I prefer using the `npx` method, as it's a bit easier to type. I'll be giving my examples using the second way.

If you haven't done so already, run either of those two commands we just looked at.

Even though we're not passing in the `config` flag to that command (although you certain can), it will notice that we don't have a `wdio.conf.js` file available and ask to start the config utility for us (`? Error: Could not execute "run" due to missing configuration. Would you like to create one? (y/N)`).

Enter `y` and press enter, and you should see a screen like this:



Terminal Output from running `npx wdio` command with no config file available

This configuration utility will ask you a series of questions in regards to how you want to run your tests.

Stepping Through the Configuration Utility

Important: This section was written using the latest version of the configuration utility. However, this utility is regularly updated and the questions asked are frequently changed/improved based on feedback. This means that what's written next may not match exactly what you see. I'll try and keep this section up-to-date, but expect to see some difference.

The first question the utility asks us is `Where is your automation backend located?`. 'Automation Backend' refers to the computer that hosts either your Chrome DevTools-capable browser, or your WebDriver server.

There are many instances where you'd want to customize this, but for us, we're going to be running it locally using Chrome DevTools, so we'll stick with the first choice and hit 'enter' on `On my local machine`.

Next up, it asks `Which framework do you want to use?`. We have three options: Mocha, Jasmine and Cucumber. All are popular JavaScript test frameworks that have been around for years, but why do we need a test framework? Isn't that the role of WebdriverIO?

No, not really. One of the great things about WebdriverIO is that it relies on existing solutions instead of trying to invent its own. This gives you a ton of features with proven code that the community has already tested through years of use.

In this case, the features we're getting are improved organization and better error reporting, along with additional functionality such as "pre" and "post" test hooks. We'll look at all that later, but back to our options.

While the Jasmine framework is more than capable of meeting our needs, all the examples in this book will be using Mocha. This is for two reasons:

- Mocha is a very popular framework, meaning you'll run into it more often.

- Almost all my professional experience has been on projects using Mocha, so I'm much more familiar with how it works and how you can take advantage of it.

With that said, hit that `enter` button again to choose Mocha.

You should now be prompted with the question `Do you want to run WebdriverIO commands synchronous or asynchronous?` with the choices `sync` and `async`.

If you recall in our first test script, there were `async` and `await` keywords scattered throughout the file:

```
(async () => {
    const browser = await remote(...);

    await browser.url('https://webdriver.io');

    const title = await browser.getTitle();

    await browser.deleteSession();
})().catch((e) => console.error(e));
```

I also mentioned `async/await` earlier in this chapter. Why is this all needed?

When commands are issued to an automation server, they don't happen immediately. There is a variable amount of time between sending the command and the browser actually running it. Usually it only takes a few milliseconds, but in some cases it can take a second or two.

What this means for our script is that we need to wait for that command to finish before moving on to the next step. That's done through these `await` keywords (`async` is needed to tell Node.js that we'll be using these keywords.)

While helpful, they clutter up the code of our test. Wouldn't it be nicer if we didn't have to type all that out, so that it looked more like this:

```
const browser = remote(...);

browser.url('https://webdriver.io');

const title = browser.getTitle();

browser.deleteSession();
```

That's a lot cleaner and easier to scan. Well, that's a feature the test runner gives you. It modifies the code you're running to automatically include those `await` statements, without having to do the work yourself.

Going back to our options, if you wanted to, you could stick with the `async` style of writing all your `await` keywords yourself. But I'm lazy, so I'll take any chance I can to do less work.

I'm going to assume you're like me and want to go with WebdriverIO handling the `async/await` nature of our tests (although you probably have better reasons than "I'm lazy"). To do that, choose `sync` as your option and hit `enter`. We'll still have a way to run custom `async` commands, but for now, we've enabled a much simpler way to write our tests.

Next, it asks `Where are your test specs located? (./test/specs/**/*.js)`.

By default, they mark the location as the `test/specs` folder. However, you can change this, if you prefer. We haven't created the folders yet, but will in a second, so go with the default by pressing `enter`. Again, you can customize this if you prefer, but the examples in the book assume you used this default path.

Just so you know, the ** and * in the path is called a Glob pattern. That's a convention that defines where multiple files are located.

The ** section says to look in all the subfolders for files, so if we later organize our tests by feature (say we add a `login` or `checkout` folder), WebdriverIO will know to look in those subfolders as well as the main `test/specs` folder.

The * portion in `*.js` matches any file that ends with a `.js` extension. So it will match `test.js` and `login.js` but not `test.txt` or just `login`.

Okay, back to the questions. After that test file location, it asks `Do you want WebdriverIO to autogenerate some test files? (Y/n)`. We're going to say no here, as I'll be walking you through everything from scratch.

After entering `n` for no and hitting enter, it asks "Are you using a compiler?". Compilers are useful utilities for improved code support, but are more complicated than I want to get into. We're going to stick with the default emphatic answer of `No!`.

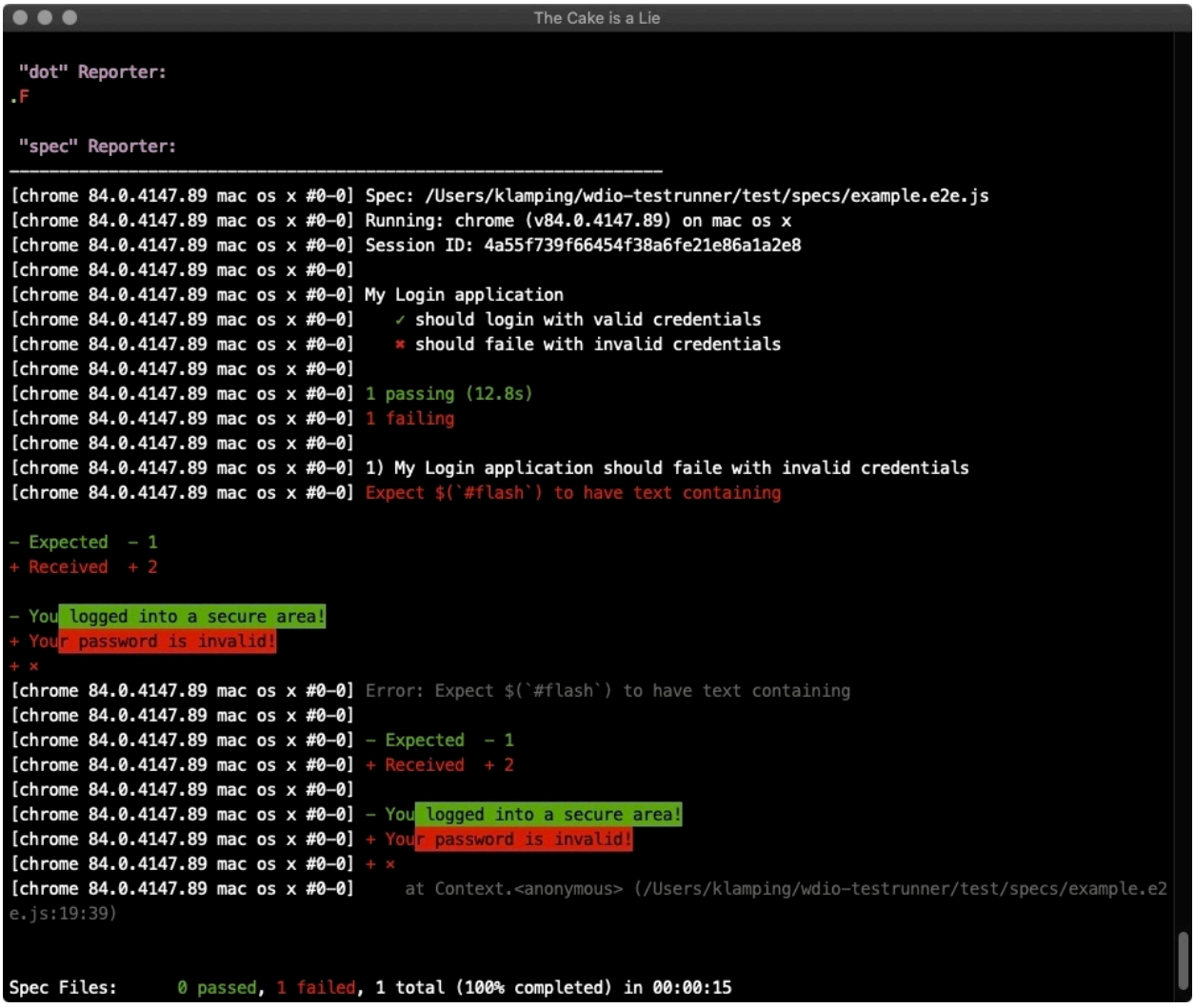
Next it asks "Which reporter do you want to use?". Here we can make multiple choices.

Reporters are utilities that display the results of our tests in a more consumable format. Some print directly to the terminal and others display as websites you visit. Most of the reporters in this list are intended for advanced usage. The two you'd be interested in right now are `dot` and `spec`.

`dot` is the simpler of the two, printing out either red or green dots depending on whether the test passed for failed (I'll let you guess which color means what).

`spec` will print out the name of the test in a hierarchical fashion, giving you a lot more information on how the test ran.

Here are the two reporters showing the results of the same test run:



Terminal Output showing pass/failure reporting in 'dot' and 'spec' reporters

On top is `dot`, which is quite succinct. `spec` is much more verbose, giving us more detail into which specific test failed.

I much prefer `spec` over `dot`, as I need to know these details. You'll probably want it as well, so leave it selected as the default choice.

We're now asked if we want any services set up for us. Services are useful extensions to the main WebdriverIO functionality. There are many official services, a plethora of third-party ones, and you can even write your own.

There are two services we'd really be interested in installing right now. They are the `chromedriver` and `selenium-standalone` service.

We've talked about both of these before, so what's going on with these services? Well, if we want to run ChromeDriver or Selenium Standalone, we have to manually start and stop the servers ourselves. What these two services do is hook into the WebdriverIO startup/shutdown sequence

and automatically start/stop the server. This is really helpful long-term, and if you are going to use either ChromeDriver or Selenium Standlone, I highly recommend using the related service.

I'm going to choose ChromeDriver, since it's a little bit faster than using Selenium Standalone and doesn't require Java to be installed.

An important note here. You shouldn't use both services at the same time. While you technically can, since they both provide the same functionality (running a WebDriver server), it doesn't make sense to have them both running.

Another important note: You can shut down a ChromeDriver or Selenium Standalone instance running in your terminal from the previous chapter. Now that we have our service set up, it will take care of starting/stopping that for us.

To finish off, the last question asks what `baseUrl` we want. By setting this value, we can shorten our `browser.url` calls. We'll talk about it soon, but for now just type `https://webdriver.io` and hit `enter`.

That completes the list of questions. Now WebdriverIO takes over and runs the NPM installs for the packages we requested, along with building out our configuration file.

If all went well, you should see this message:

```
Packages installed successfully, creating configuration file...

Configuration file was created successfully!
To run your tests, execute:
$ npx wdio run wdio.conf.js
```

If you made a mistake and need to run the config generator again, you can either delete your `wdio.conf.js` file and run the command again, or manually request the config builder with `npx wdio config`. This second option will overwrite any existing `wdio.conf.js` file.

As many options as we covered just now, the config file actually sets quite a few more. We'll take a look through this newly generated file next.

1.2.5 Reviewing the Standard WebdriverIO Configuration File

We just used the WebdriverIO test runner to help us create a configuration file to hold all of our settings. Let's take a look at that file.

Open the newly created `wdio.conf.js` file in your text editor of choice. The file is saved in JavaScript format, so using a text editor with JavaScript syntax highlighting is a good idea.

In our file, the first thing we'll check out is the `exports.config` line. If you're familiar with Node.js, you'll recognize the use of the `exports` global variable.

Because this is just a plain JavaScript file, it gives us the ability to run normal Node.js commands. For example, at the very top of the file, add a `console.log` command to show that we're in fact using this file.

```
console.log('I am inside your configuration file, running your tests!')

exports.config = { // rest of the file would be below
```

There are some added benefits to being able to run Node.js code. One common usage is to customize your WebdriverIO configuration based on environmental variables. Alternatively, you can handle that through custom command line arguments. This is a pretty in-depth topic though, so we'll leave that alone for now.

Let's check out all the settings defined in our file. There are actually several more options here than just what we answered during our config step and it's good to know what those are.

Runner

```
// ====================
// Runner Configuration
// ====================
//
// WebdriverIO allows it to run your tests in arbitrary locations (e.g., locally
// or on a remote machine).
runner: 'local',
```

The first setting is `runner`, which is based on our answer to the first question in the configuration. We're going to skip this option as it's not much use to us right now.

Specs and Exclude

```
// ==================
// Specify Test Files
// ==================
// Define which test specs should run. The pattern is relative to the directory
// from which `wdio` was called. Notice that, if you are calling `wdio` from an
// NPM script (see https://docs.npmjs.com/cli/run-script) then the current working
// directory is where your package.json resides, so `wdio` will be called from there.
//
specs: [
    './test/specs/**/*.js'
```

```
],
// Patterns to exclude.
exclude: [
    // 'path/to/excluded/files'
],
```

Next is 'specs', which defines the folder path to our tests. We talked about this during our config setup, but notice that it's stored as an array. The reason for that is so we can add multiple patterns to search for, or specific files we'd like to run.

Likewise, there's an `exclude` option, which allows us to exclude files based on a pattern or specific path.

Both patterns are relative to the directory from which `wdio` was called.

Let's move on to the capabilities section, which groups the information needed by WebdriverIO to initiate our desired WebDriver session. There are two parts to it:

Max Instances

```
// First, you can define how many instances should be started at the same time. Let's
// say you have 3 different capabilities (Chrome, Firefox, and Safari) and you have
// set maxInstances to 1; wdio will spawn 3 processes. Therefore, if you have 10 spec
// files and you set maxInstances to 10, all spec files will get tested at the same
// time and 30 processes will get spawned. The property handles how many capabilities
// from the same test should run tests.
//
maxInstances: 10,
```

WebdriverIO can run multiple capabilities at the same time, saving time in your test runs.

Say you have five test files, and each test file takes exactly one minute to run.

Without this ability, the total time it would take your tests to run is five minutes. As each test file finishes, the next file is started.

If you set `maxInstances` to 3, WebdriverIO will start three separate sessions to run your separate files. The first three tests would all run at the same time, taking one minute to complete overall. After the first two tests finish, WebdriverIO will automatically start running the remaining two tests. In total, the test suite would take two minutes to run (one minute for the first three tests and an additional minute for the final two).

If you set `maxInstances` to 5, all five files would run at the same time, taking a total of one minute for all your tests to run.

A word of warning: While running tests in parallel can certainly save time, it also requires you to have a powerful computer to process all the commands. If you find your CPU is maxing out when

running all your tests, consider lowering this number to reduce the strain on your processor. Maxing out your CPU can definitely lead to failures in your test runner as the computer isn't able to keep up with all the requests.

Capabilities

```
// If you have trouble getting all important capabilities together, check out the
// Sauce Labs platform configurator - a great tool to configure your capabilities:
// https://docs.saucelabs.com/reference/platforms-configurator
//
capabilities: [{

    // maxInstances can get overwritten per capability. So if you have an
    // in-house Selenium grid with only 5 firefox instances available you can
    // make sure that not more than 5 instances get started at a time.
    maxInstances: 5,
    //
    browserName: 'chrome',
    acceptInsecureCerts: true
    // If outputDir is provided WebdriverIO can capture driver session logs
    // it is possible to configure which logTypes to include/exclude.
    // excludeDriverLogs: ['*'], // pass '*' to exclude all driver session logs
    // excludeDriverLogs: ['bugreport', 'server'],
}],
```

Up next is our `capabilities` setting. Again, the value is an array, which means that we can run our tests in multiple browsers each time we use the test runner.

The configuration here looks similar to what we provided in the `desiredCapabilities` of our first `test.js` file. Your capabilities setting can get pretty complex, but for now we can leave it as is.

Note that within your capabilities you can overwrite the `spec`, `exclude`, and `maxInstances` options in order to group specific specs to a specific capability.

Log Level

```
// ====================
// Test Configurations
// ====================
// Define all options that are relevant for the WebdriverIO instance here
//
// Level of logging verbosity: trace | debug | info | warn | error | silent
logLevel: 'info',
```

Next in our settings file is the test configurations section. These levels correspond to common log levels in code output, and the higher up the list you choose (trace being highest), the more log output you'll get. This can be helpful for debugging later on. `silent`, on the other hand, doesn't log anything when running tests, and can be useful for avoiding a lot of noise in your test run.

It defaults to `info`, which is a good mix between the two ends of the spectrum. Feel free to change this as you see fit.

Bail

```
// If you only want to run your tests until a specific amount of tests have failed
// use bail (default is 0 - don't bail, run all tests).
bail: 0,
```

The `bail` setting can be useful if you only want to run your tests until a specific amount of tests have failed use. This can help save time when debugging tests.

Base URL

```
// Set a base URL in order to shorten url command calls. If your `url` parameter
// starts with `/`, the base url gets prepended, not including the path portion
// of your baseUrl. If your `url` parameter starts without a scheme or `/`
// (like `some/path`), the base url gets prepended directly.
baseUrl: 'http://localhost',
```

We've already talked about `baseUrl`, so we'll jump past that setting as well.

"waitFor" Timeout

```
// Default timeout for all waitFor* commands.
waitforTimeout: 10000,
```

`waitforTimeout` defines how long `waitFor` commands should wait until erroring out. We haven't covered these commands yet, so let's leave this at the default value.

Connection Retry Options

```
// Default timeout in milliseconds for request
// if browser driver or grid doesn't send response
connectionRetryTimeout: 120000,
//
// Default request retries count
connectionRetryCount: 3,
```

The `connectionRetryTimeout` and `connectionRetryCount` options are useful to adjust if you're having trouble connecting to your Selenium Grid. You should be good to leave these alone though.

Services

```
// Test runner services
// Services take over a specific job you don't want to take care of. They enhance
// your test setup with almost no effort. Unlike plugins, they don't add new
// commands. Instead, they hook themselves up into the test process.
services: ['chromedriver'],
```

Here's an array of services we have chosen to run when using the test runner. These services provide added features. In this instance, we've got the 'chromedriver' service enabled (assuming you chose to install it). The overall option is an array, so yes, you can have multiple services running at once.

Do note that not all services work together. For example, you wouldn't want to have the following services array:

```
services: ['selenium-standalone', 'chromedriver', 'sauce'],
```

All three of those services do the same thing: host a WebDriver server for you to run your tests against. By having all three services set, they'd have conflicts when starting up. Instead, you'd just choose one and go with it.

Framework

```
// Framework you want to run your specs with.
// The following are supported: Mocha, Jasmine, and Cucumber
// see also: https://webdriver.io/docs/frameworks.html
//
// Make sure you have the wdio adapter package for the specific framework installed
// before running any tests.
framework: 'mocha',
```

Next is the `framework` setting set to `mocha`, which is what we provided.

Reporters

```
// Test reporter for stdout.
// The only one supported by default is 'dot'
// see also: https://webdriver.io/docs/dot-reporter.html
reporters: ['spec'],
```

You'll also see the `reporter` option set to 'spec'. Since this is an array, you can pass in multiple reporters if you'd like.

Mocha Options

```
// Options to be passed to Mocha.
// See the full list at http://mochajs.org/
mochaOpts: {
    ui: 'bdd',
    timeout: 60000
},
```

The `mochaOpts` option is useful for passing configurations for Mocha to use. Here it defines using the `bdd` ui type, which says we want to write our tests in the Behavior-driven Development style[57].

A `timeout` setting of 60000 is provided, which says that after 60000 milliseconds of running (or 60 seconds), our test will time out. Some tests can take up to three minutes to run, so updating this value to 180000 can be useful if your tests take a little longer to complete.

There are more options you can define here for Mocha, and you can get a full list of options to set via the Mocha website[58].

Hooks

Finally, there are several hooks we can use to add functionality in the middle of the test process. This opens up a fair amount of potential to really add on to the default WebdriverIO test runner functionality, and we'll look at an example of that later on.

The End

That's the end of our settings file. We'll be jumping back in this file often to adjust settings and add new functionality.

For now though, let's get our test file setup so we can try out all these new features.

[57] https://mochajs.org/#interfaces
[58] https://mochajs.org/#command-line-usage

1.2.6 Running the Example Test Runner Test

Now that we've created and reviewed our configuration file, it's time to write a test to make use of it.

Since this is the first time we're creating a test, we need to create our folders for the test files to go into.

During setup, we defined out specs path as `./test/specs/**/*.js`. But we don't have a test folder, nor a specs folder inside it.

Let's create those folders by running the following command from our `wdio-testrunner` project path:

```
mkdir -p test/specs
```

On Windows that's:

```
mkdir test\specs
```

That will create both folders for us. Now we can create our first test file:

```
touch ./test/specs/example.js
```

On Windows, `touch` will not work, so you can instead go to the `test/specs/` folder and create a JavaScript file named 'example.js'

Open that file in a text editor and paste in the example from the official WebdriverIO website:

test/specs/example.js

```
describe('webdriver.io page', () => {
    it('should have the right title', () => {
        browser.url('https://webdriver.io');
        expect(browser).toHaveTitle(
            'WebdriverIO · Next-gen browser and mobile automation test \
framework for Node.js'
        );
    });
});
```

There is one thing we want to change. Where it sets the `url`, we instead want to use the `baseUrl` we defined in our configuration file.

So change `browser.url('https://webdriver.io');` to be `browser.url('./');`. When WebdriverIO runs that command, it will see that it isn't a full website URL, and prepend our `baseUrl` from the config file.

Okay, let's save the file and try running the test.

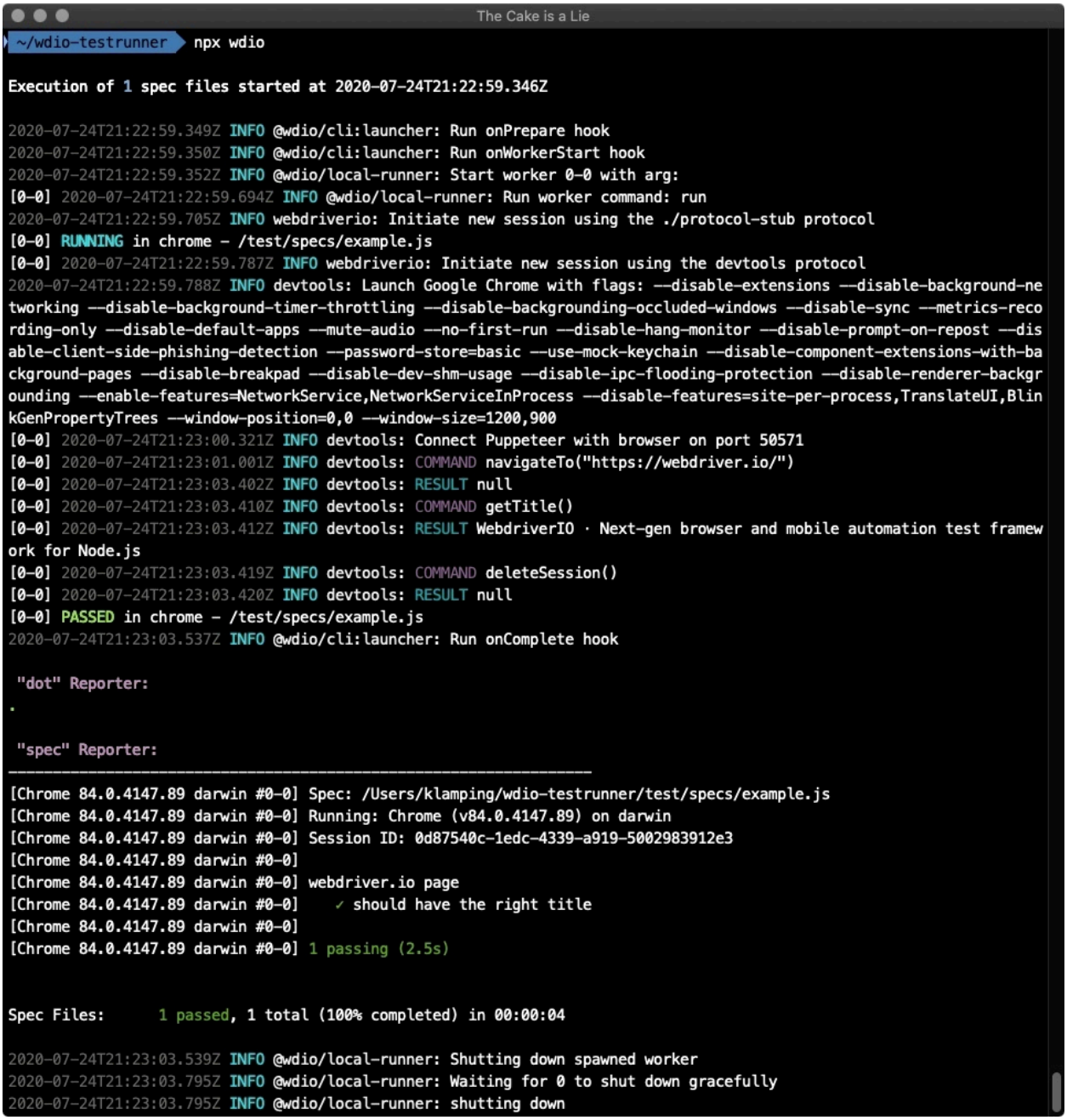
Running Through Chrome DevTools

Since the Chrome Devtools protocol only requires a supported browser to be installed, if you have a browser like that already installed, you don't need to do anything else. Do check that if you selected `chromedriver` from the `services` list during the `wdio config` section, you go to your `wdio.conf.js` file and comment out the `services` line (this was chromedriver won't start up when running your test).

Now, to run the tests, we call the testrunner in our terminal:

```
npx wdio
```

The test will run, you should see a browser pop-up, and successful test output that looks something like:



Terminal output running our test via Chrome DevTools protocol

Again, notice the line `INFO webdriverio: Initiate new session using the devtools protocol`. That let's us know we really are using the DevTools protocol.

Running via Selenium Standalone

Assuming you selected the Selenium Standalone service during our config run-through (and have it enabled in your `services` section in your `wdio.conf.js` file, `services: ['selenium-standalone']`), we don't have to do anything to run our tests through Selenium Standalone. This is because the service takes care of starting and stopping our Selenium for us.

The one thing we should check on is making sure the Selenium server we started up earlier isn't still running. Check your terminal windows to make sure this isn't the case (remember you can stop the server by sending the `ctrl-c` key command).

You can also go to `http://localhost:4444/wd/hub` and make sure it returns a "site can't be reached" error.

Okay, with that out of the way, let's run our tests by calling the WDIO testrunner inside our terminal:

```
npx wdio
```

Like before, after a few moments, test output will start appearing. You'll probably see a browser window pop up. Just let it sit there and do it's thing (this can make for a neat magic trick for someone unfamiliar with automation).

The test will take a few seconds to run, then shut everything down. Once finished, it'll print out the final results, something like:

Terminal output running our test via Selenium Standalone

And remember, our `selenium-standalone` service took care of shutting down the server it started, so we don't have to do anything.

If you want to try this out, but didn't select the `selenium-standalone` service during the config steps, you can manually install the package by running:

```
npm install @wdio/selenium-standalone-service
```

Then add it to your `services` array and give it a shot.

Running with ChromeDriver

ChromeDriver also comes with its own WebdriverIO service, allowing us to automatically start and stop a `chromedriver` server during our test run.

If you didn't select `chromedriver` from the `services` list during the config, installation and usage is much like `selenium-standalone`, beginning with our NPM install step.

You'll need to install two packages:

```
npm install wdio-chromedriver-service chromedriver
```

Even though, in an earlier example, we installed the `chromedriver` package globally, we're installing it locally so that it's added to our package.json file.

Information on the packages we install locally automatically get saved to a `dependency` section of our package.json file by NPM. This gives us the ability to re-install any missing dependencies later on by running `npm install`. This is helpful if you want to load your project from scratch.

With the installation done, we need to do two more things. Jump back into your `wdio.conf.js` file and skip down to the services section. Ensure that the only service defined is `chromedriver` (e.g., `services: ['chromedriver']`)

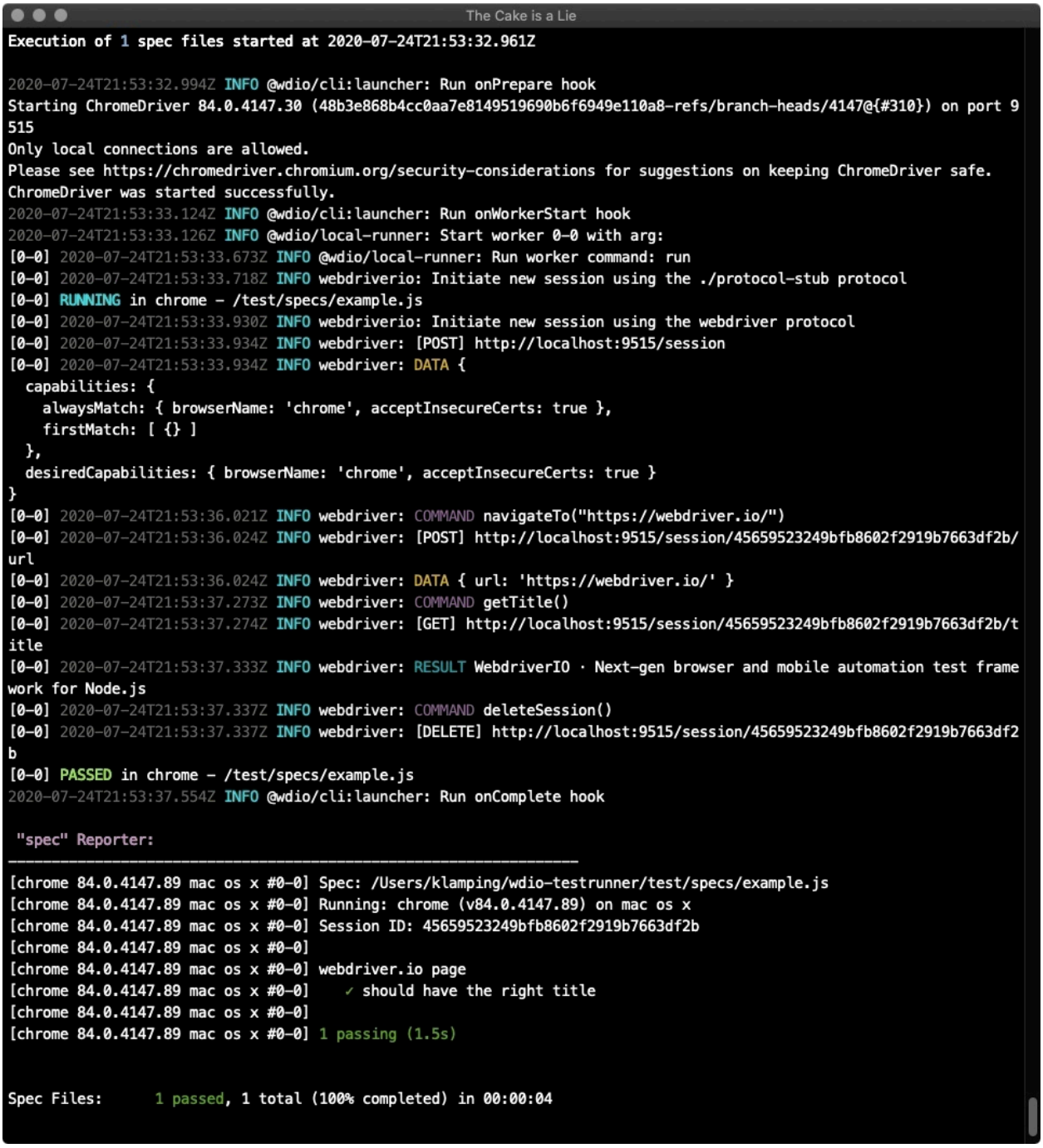
As I've mentioned before, you don't want to have both `selenium-standalone` and `chromedriver` in your services at the same time as they would conflict with each other.

With that done, let's talk about one more thing. You may recall during our first time using ChromeDriver that we had to update the `port` and `path` settings in our WebDriver session initialization. Well, to help us out, the ChromeDriver service configures this for us. Before you'd need to set this manually in your config, but now it's all handled in the service.

With that all set up, let's run our tests again with our `wdio` command:

```
npx wdio
```

The output should look almost identical to before:



Terminal output running our test via ChromeDriver

Notice the line `INFO webdriver: [POST] http://localhost:9515/session`. See how the port is set to `9515`? That's the ChromeDriver service managing it all for us!

Whether you use ChromeDriver or Selenium Standalone is entirely up to you. The benefit of ChromeDriver is that you don't need to have Java installed to run, but the drawback is that you can only run your tests in Chrome.

On the other hand, Selenium Standalone installs and manages all the browser drivers for you, so you're free to run most any browser installed on your computer.

Personally I use either of them depending on the situation. I usually lean on ChromeDriver when starting out, as it's usually quicker to run, then switch to Selenium Standalone when I need to expand my browser coverage.

For the rest of the book, we'll be using ChromeDriver going forward.

Now that we've got our test running, let's take some time to review the actual code.

Reviewing the Example Test Code

Just for a quick comparison, below is the sample code for standalone mode versus running through the test runner:

Standalone:

test.js

```
const { remote } = require('webdriverio');

(async () => {
    const browser = await remote({
        capabilities: {
            browserName: 'chrome'
        }
    });

    await browser.url('https://webdriver.io');

    const title = await browser.getTitle();
    console.log('Title was: ' + title);

    await browser.deleteSession();
})().catch((e) => console.error(e));
```

Test Runner:

test/specs/example.js

```
describe('webdriver.io page', () => {
    it('should have the right title', () => {
        browser.url('https://webdriver.io');
        expect(browser).toHaveTitle(
            'WebdriverIO · Next-gen browser and mobile automation test \
framework for Node.js'
        );
    });
});
```

Overall our test runner code is much more succinct. This is due to quite a few things:

- **No need for `remote` and setting up our `browser` object**: This is all done via the configuration file and test runner
- **No need for `async` and `await` keywords**: By running our tests in `sync` mode, we no longer need to add all that extra code.
- **No need to delete the session**: The test runner manages both creating and ending the sessions, so we don't manually have to do this.

Those are all the things that have been removed, but there are a couple new bits added as well.

There are these `describe` and `it` function calls. They are part of the Mocha framework, and help organize our tests into individual test cases. This is helpful for overall readability, plus we can use them to run or exclude certain cases from our test runs.

- `describe` is used to group sets of tests by the feature they are testing.
- `it` defines a specific test to run.

There are usually multiple `it` functions nested inside each `describe`, and sometimes `describe` functions are nested inside each other for a better defined test hierarchy.

As you can see in our file, `describe` is called as a function, passing in two parameters.

```
describe('name of test section', function () { });
```

The first parameter is a name for the feature we are testing, in this case `'name of test section'`. The second half of our `describe` call is a function which contains all of the code we want to associate with this feature.

Inside this function we'll be adding the `it` call. Similar to `describe`, `it` is a function call that takes two parameters:

```
it('name of individual test', function () { });
```

- The first parameter is the name we give our test, in this case `'name of individual test'`. I like to begin my test case names with `should`, so that it reads `it('should do whatever'...`
- The second parameter is a function which contains our browser commands and assertions.

To bring it all together, here's an example test file with just the structure in place:

```
describe('Login Page', function () {
    it('should allow you to log in using valid credentials', function () {
        // valid login code here
    });
    it('should not allow you to log in using invalid credentials', function () {
        // invalid login code here
    });
});
```

1.2.7 Command Line Options (and Logging)

Now that we've gotten comfortable with the Test Runner, let's dig into some advanced usage of the tool.

From the start, there are several command line overrides you can use to customize your tests runs on a case-by-case basis.

To get a full listing of all options, pass in the `help` option when running `npx wdio`:

```
npx wdio --help
```

It will output the various settings you can update from the command line.

Terminal Output from `npx wdio --help` command

There are plenty of settings to configure, but most you won't use. There are a few common settings you'll tinker with on a regular basis though. Let's take a look at them.

Config File

If you look at the first example near the bottom, it shows:

```
wdio run wdio.conf.js --suite foobar      Run suite on testsuite "foobar"
```

There are three parts: First, the `wdio run` command (note `npx wdio` and `npx wdio run` do the same thing, as `wdio` calls `run` by default).

Then, a path to a configuration file (i.e., `wdio.conf.js`, we'll talk about this in just a second).

Finally, an option to set the `suite` to `foobar`. We can provide many options here, and we'll get to that soon.

Back to the configuration file. By default, WebdriverIO will use the `wdio.conf.js` file (assuming you have one). If you want to have a different configuration file with different settings, this is how you can get WebdriverIO to use it.

Say I create a second configuration file called `wdio.alternative.conf.js`. I can use that file by running:

```
npx wdio wdio.alternative.conf.js
```

Now WebdriverIO will use that instead of `wdio.conf.js`.

As you get into more advanced usage of the framework, you'll probably want to have custom configuration files. We won't be covering that in this book though, as it's a bit too advanced for our needs. So, let's move on and explore those options.

Run Options

In the usage examples, you see two main options being passed: `suite` and `spec`. We can get the full list of options however by running `npx wdio run --help`:

```
Options:
  --version             Show version number                            [boolean]
  --watch               Run WebdriverIO in watch mode                  [boolean]
  --hostname, -h        automation driver host address                  [string]
  --port, -p            automation driver port                          [number]
  --path                path to WebDriver endpoints (default "/")       [string]
  --user, -u            username if using a cloud service as automation backend
                                                                        [string]
  --key, -k             corresponding access key to the user            [string]
  --logLevel, -l        level of logging verbosity
```

```
                [choices: "trace", "debug", "info", "warn", "error", "silent"]
  --bail                stop test runner after specific amount of tests have
                        failed                                          [number]
  --baseUrl             shorten url command calls by setting a base url [string]
  --waitforTimeout, -w  timeout for all waitForXXX commands             [number]
  --framework, -f       defines the framework (Mocha, Jasmine or Cucumber) to
                        run the specs                                   [string]
  --reporters, -r       reporters to print out the results on stdout     [array]
  --suite               overwrites the specs attribute and runs the defined
                        suite                                           [array]
  --spec                run only a certain spec file - overrides specs piped
                        from stdin                                      [array]
  --exclude             exclude certain spec file from the test run - overrides
                        exclude piped from stdin                        [array]
  --mochaOpts           Mocha options
  --jasmineNodeOpts     Jasmine options
  --cucumberOpts        Cucumber options
  --help                Show help                                    [boolean]
```

That's a lot of choices! Let's look at the most useful ones...

Spec

The first option you'll likely be using often is passing in a specific `spec` path.

This is quite helpful when you need to test out a specific file. Instead of having to test all the files in your test folder, you can single one out from the rest.

How about a couple of examples? Say you have the following test files:

```
/test/homepage.js
/test/search.js
/test/auth/login.js
/test/auth/logout.js
/test/auth/register.js
```

Five files total, with three of them being inside the `test/auth` folder.

Assuming the `specs` setting in your configuration file is set to `./test/**/*.js`, normally you'd run all five tests on each run.

If you want to run just the homepage test, you could do any one of the following:

```
npx wdio --spec=./test/homepage.js
npx wdio --spec=homepage.js
npx wdio --spec=home
```

The `spec` option takes either the exact path to the file you want to test, or a filter by filename. So all three variations above work because `./test/homepage.js` matches against all three.

Here's a different example: I want to run just the login/logout tests. I could do any of the following:

```
npx wdio --spec=./test/auth/login.js --spec=./test/auth/logout.js
npx wdio --spec=login --spec=logout
npx wdio --spec=./test/auth/log
npx wdio --spec=log
```

Notice in the first two options, we define the `spec` option twice, allowing us to choose two different files specifically.

As mentioned before, the `spec` option is very helpful when debugging individual tests. We'll be using it a lot going forward.

Bail

One option we briefly talked about while going through the configuration file is `bail`. This will tell WebdriverIO to stop running tests after a certain number of test failures.

In our configuration, we have it set to `0`, which means that it will run all the tests no matter the number of failures. This is useful when we want to see how all of our tests run.

But when debugging a set of tests, it can be useful to stop running them if there's a failure. Maybe you're uncertain if all your tests will pass, and you just want to see if there are any failures at all.

By passing in `--bail=1` as a command-line option, we can achieve that.

```
npx wdio --bail=1
```

Of course, we could set this to any number, but `1` is what will be used most, as you'll want to get back to fixing your broken tests right away.

Base URL

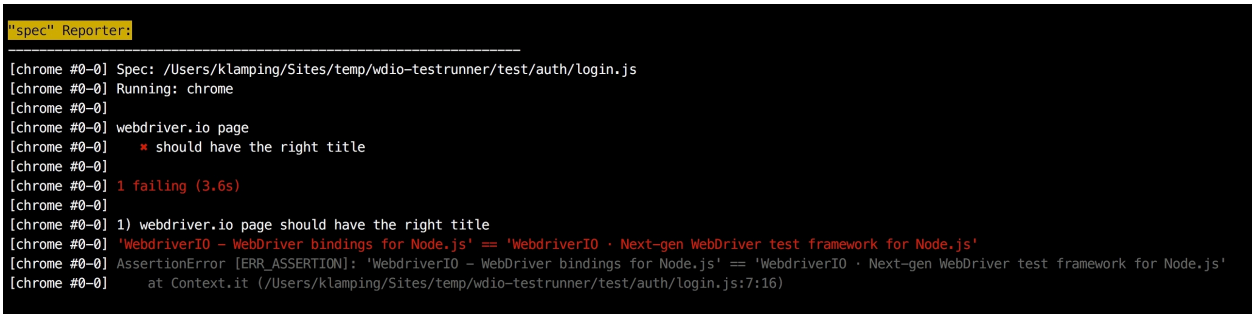
Overriding the `baseUrl` can be helpful for times when you need to test the same site on a different server. Most often this occurs when you're testing a server on your local computer versus the test server. It can also be useful for one-time tests of special servers spun up to run specific code.

In our settings, we defined our `baseUrl` as `https://webdriver.io`. And in our test, we used that URL by changing our `browser.url` call.

Let's say we want to test the old version of the WebdriverIO website. That URL is `http://v4.webdriver.io`. So to pass it in via the command line, we'd run:

```
npx wdio --baseUrl=http://v4.webdriver.io
```

When we run that, we're going to get an assertion error. That's because it went to the old site, which had a different page title from the new one. At least we know our test correctly catches errors!



Terminal output showing error message if `baseUrl` is incorrect

I use the `baseUrl` override only every so often, but it's certainly handy to have around.

Log Level

Let's take a look at another option. We've run across the `logLevel` a couple times, which defines how much console output to show when running our tests.

When you're debugging your tests, it can be helpful to see all the logs that WebdriverIO outputs. Other times you may not want any output at all. You can tweak this on a run-by-run basis by setting the `logLevel` option.

So if I want to get as much output as possible, I'd do:

```
npx wdio --logLevel=trace
```

If I didn't want any logs, I'd do:

```
npx wdio --logLevel=silent
```

Let's try out that first option. As the tests run, you should see logs appearing, which are describing what's currently happening. For example, the start of our logs would show:

```
2020-07-24T22:50:20.066Z INFO @wdio/cli:launcher: Run onWorkerStart hook
2020-07-24T22:50:20.067Z INFO @wdio/local-runner: Start worker 0-0 with arg:
--logLevel=trace
[0-0] 2020-07-24T22:50:20.425Z INFO @wdio/local-runner: Run
worker command: run
[0-0] 2020-07-24T22:50:20.431Z DEBUG
@wdio/local-runner:utils: init remote session
```

As the test continues, old messages will "scroll" off the top of the log, while new messages appear at the bottom.

Notice that each log message has a 'type.. The first message in the example output above is INFO and the last is DEBUG. These types are defined inside the core WebdriverIO code, and generally denote different types of messages.

The log level is a hierarchical setup, having five levels:

- Trace
- Debug
- Info
- Warn
- Error

When setting the log level to `trace`, you'll get all the `trace` messages, plus all the types below it (so `debug`, `info`, `warn`, and `error`).

If you were to choose `warn`, you'd only get those messages, plus any `error` messages.

After our tests have completed, WebdriverIO prints out the entirety of messages in a single section. Here's what that looks like:

```
2020-07-24T22:50:19.909Z DEBUG @wdio/utils:initialiseServices: initialise
service "chromedriver" as NPM package
2020-07-24T22:50:19.933Z INFO @wdio/cli:launcher: Run onPrepare hook
Starting ChromeDriver 84.0.4147.30 (48b3e868b4cc0aa7e8149519690b6f6949e110a8-ref
s/branch-heads/4147@{#310}) on port 9515
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggest
ions on keeping ChromeDriver safe.
ChromeDriver was started successfully.
2020-07-24T22:50:20.066Z INFO @wdio/cli:launcher: Run onWorkerStart hook
2020-07-24T22:50:20.067Z INFO @wdio/local-runner: Start worker 0-0 with arg: --l
ogLevel=trace
[0-0] 2020-07-24T22:50:20.425Z INFO @wdio/local-runner: Run worker command: run
```

```
[0-0] 2020-07-24T22:50:20.431Z DEBUG @wdio/local-runner:utils: init remote sessi
on
2020-07-24T22:50:20.433Z INFO webdriverio: Initiate new session using the ./prot
ocol-stub protocol
[0-0] RUNNING in chrome - /test/specs/example.js
[0-0] 2020-07-24T22:50:20.576Z DEBUG @wdio/utils:initialiseServices: initialise
service "chromedriver" as NPM package
[0-0] 2020-07-24T22:50:20.592Z DEBUG @wdio/local-runner:utils: init remote sessi
on
[0-0] 2020-07-24T22:50:20.592Z INFO webdriverio: Initiate new session using the
webdriver protocol
[0-0] 2020-07-24T22:50:20.599Z INFO webdriver: [POST] http://localhost:9515/sess
ion
[0-0] 2020-07-24T22:50:20.599Z INFO webdriver: DATA {
  capabilities: {
    alwaysMatch: { browserName: 'chrome', acceptInsecureCerts: true },
    firstMatch: [ {} ]
  },
  desiredCapabilities: { browserName: 'chrome', acceptInsecureCerts: true }
}
[0-0] 2020-07-24T22:50:22.453Z INFO webdriver: COMMAND navigateTo("https://webdr
iver.io/")
[0-0] 2020-07-24T22:50:22.454Z INFO webdriver: [POST] http://localhost:9515/sess
ion/32eec5f507ed6f86473e3c4b685fd96b/url
[0-0] 2020-07-24T22:50:22.454Z INFO webdriver: DATA { url: 'https://webdriver.io
/' }
[0-0] 2020-07-24T22:50:23.711Z INFO webdriver: COMMAND getTitle()
[0-0] 2020-07-24T22:50:23.711Z INFO webdriver: [GET] http://localhost:9515/sessi
on/32eec5f507ed6f86473e3c4b685fd96b/title
[0-0] 2020-07-24T22:50:24.119Z INFO webdriver: RESULT WebdriverIO · Next-gen bro
wser and mobile automation test framework for Node.js
[0-0] 2020-07-24T22:50:24.124Z INFO webdriver: COMMAND deleteSession()
2020-07-24T22:50:24.124Z INFO webdriver: [DELETE] http://localhost:9515/session/
32eec5f507ed6f86473e3c4b685fd96b
2020-07-24T22:50:24.292Z DEBUG @wdio/local-runner: Runner 0-0 finished with exit
 code 0
[0-0] PASSED in chrome - /test/specs/example.js
2020-07-24T22:50:24.297Z INFO @wdio/cli:launcher: Run onComplete hook
```

Let's take a brief detour to walk through this activity. The first thing WebdriverIO does is initialize any services we requested. In this instance, we're using the `chromedriver` service, so we see log output for that.

Next, it runs any `onPrepare` hooks we have defined. We don't have any defined, but the ChromeDriver

service does. That's why you see output stating that ChromeDriver is starting up. This is how services "hook" into the WebdriverIO flow to add functionality.

Following that, WebdriverIO starts "workers" for our tests. These are sub-processes spun up that our test will run in. The point of doing this is to allow for multiple tests to run at the same time (they'd all be different "workers").

It then sends the `run` command to the worker (i.e., `INFO @wdio/local-runner: Run worker command: run`), letting it know it should get to work. The worker itself then initializes the `chromedriver` service, in case it needed to run anything inside of a specific worker (which it doesn't).

The next step is to get a browser running for use. These are called "sessions". To get one, WebdriverIO sends a POST request (`INFO webdriver: [POST] http://localhost:9515/session`) with the capability data. This data is sent to chromedriver, which receives this request and initializes the session with the provided data. Combined, the two logs are:

```
[0-0] 2020-07-24T22:50:20.599Z INFO webdriver: [POST] http://localhost:9515/session
[0-0] 2020-07-24T22:50:20.599Z INFO webdriver: DATA {
  capabilities: {
    alwaysMatch: { browserName: 'chrome', acceptInsecureCerts: true },
    firstMatch: [ {} ]
  },
  desiredCapabilities: { browserName: 'chrome', acceptInsecureCerts: true }
}
```

You'll notice this matches the capabilities settings in our config file for the most part, but has a couple of different options specified. These are WebdriverIO defaults used to start a normal browser session. We could override them via the capabilities object in our config file if we so desired, but we don't right now, so we'll omit them.

Understanding the relationship between WebdriverIO and WebDriver/Selenium is helpful, so I want to take a little bit of extra time to review it. WebdriverIO doesn't actually run the browser automation, WebDriver (or whatever WebDriver endpoint we're using) takes care of all of that.

What WebdriverIO does provide is a JavaScript interface for sending commands to run. It does this through REST API calls, which means that it sends an HTTP request to specific endpoints on the WebDriver server (e.g., ChromeDriver). Basically, WebdriverIO and WebDriver have a common language they share (defined in the official WebDriver spec[59]) to send data back and forth. WebdriverIO sends commands to the WebDriver server for it to run, then the server sends the results of those commands back.

Take the `getUrl` command in the next few lines. WebdriverIO sends a request to the Chromdriver instance. In the request, it passes along information about the URL for the browser to go to. After ChromeDriver receives and processes this request, it returns the results of the command execution (which is empty). WebdriverIO doesn't output this information, as there isn't anything to show.

[59] https://w3c.github.io/webdriver/

Overall, the logs for this look like:

```
[0-0] 2020-07-24T22:50:20.599Z INFO webdriver: [POST] http://localhost:9515/session
[0-0] 2020-07-24T22:50:20.599Z INFO webdriver: DATA {
  capabilities: {
    alwaysMatch: { browserName: 'chrome', acceptInsecureCerts: true },
    firstMatch: [ {} ]
  },
  desiredCapabilities: { browserName: 'chrome', acceptInsecureCerts: true }
}
```

In the next set of logs, WebdriverIO requests the page title from the browser. No data is sent to ChromeDriver, as there isn't any information to send. Instead, ChromeDriver returns data back to us, namely, the title of the page. You see this in the "result" log output and sequentially, the console output we sent in our test:

```
[0-0] 2020-07-24T22:50:23.711Z INFO webdriver: COMMAND getTitle()
[0-0] 2020-07-24T22:50:23.711Z INFO webdriver: [GET] http://localhost:9515/session/3
2eec5f507ed6f86473e3c4b685fd96b/title
[0-0] 2020-07-24T22:50:24.119Z INFO webdriver: RESULT WebdriverIO · Next-gen browser
 and mobile automation test framework for Node.js
```

Finally, WebdriverIO closes our browser by sending a `DELETE` request for our session endpoint. Once it gets the successful close message, it closes down the worker and runs the `onComplete` hook. With that the test is complete.

That was a lot of logs. What if we want to ignore all this output and only show the basics (plus any `console.log` messages we may have added). Let's see what the log output looks like when running it with `logLevel` set to `silent`:

```
Starting ChromeDriver 84.0.4147.30 (48b3e868b4cc0aa7e8149519690b6f6949e110a8-refs/br
anch-heads/4147@{#310}) on port 9515
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions
 on keeping ChromeDriver safe.
ChromeDriver was started successfully.
[0-0] RUNNING in chrome - /test/specs/example.js
[0-0] PASSED in chrome - /test/specs/example.js
```

There's still some output, but not much. This is helpful if you have console.log messages that you want to be able to see without extra noise.

Other Options

That sums it up for the most important options. There are other settings you can configure, but that would be too in-depth and personalized for the good of this book. Feel free to experiment, and remember, you can get all the options by running `npx wdio run --help`.

2.0 A 'Real World' App

2.0.1 Why 'Real World'?

Over the past few chapters, we've covered the basics of getting started with WebdriverIO. Now we're ready to fully jump into writing tests.

In my experience, the best way to learn a subject is to use real-world examples. Knowing why and when you would use specific code is just as important as knowing how to use that code. To be 'real-world', we need a good website to test. It needs to be something more than a barebones example, which always seems "too clean." We need a website we can get our hands dirty with.

After much searching and contemplation, I found an option that fits the bill. The folks at Thinkster[60] have created 'The mother of all demo apps', which they've named "The RealWorld Project[61]."

At its core, the RealWorld project is just an architecture for trying out different technologies (e.g., React and/or Express). Developers from all over the world have built out various implementations of the same structure, and made them all available for us to try out.

On their GitHub homepage[62], you can see the many clones available from the community. They split the clones between the front-end and back-end implementations. There are over 15 front-end variants to choose from, and over 30 back-end choices.

Here are some benefits of this:

1. All the variants of implementations allow you to fit the project to your needs. There are many choices available to developers in the web industry today, so the framework that you're working with is going to be different from other companies out there. With this app, you're able to choose the flavor that best fits your needs.
2. Wherever you fall in the 'Battle of the JavaScript Frameworks', you have to admit that most websites are built with one of them. Having an example built that utilizes these technologies provides a more real-life type website to test.
3. It has a myriad of features that can throw you for a loop when testing. This makes it good for teaching how not to get thrown for a loop.
4. The functionality is complex, without being overbearing. A more "pure" example wouldn't have this, which means you're more likely to be thrown for a loop when testing. And you know how I feel about that.

That's enough reasons I think.

[60] https://thinkster.io/
[61] https://realworld.io/
[62] https://github.com/gothinkster/realworld

So how do we use this app?

We've got a few options, with benefits and drawbacks of each. Let's go over those.

2.0.2 Use the Shared Demo App

I've put together an instance of the project using the VueJS front-end[63] and the Node/Express back-end[64], with a few modifications. These modifications are necessary to complete the examples in the later chapters, so make sure you don't use other RealWorld code repos (unless they've also been modified to fit our needs).

To make things easier, I have a demo site of this code at: https://demo.learnwebdriverio.com/[65]

Feel free to look around there to see what the site is all about. You're also welcome to create an account and add some posts; it's a sandbox for anyone's use. And because of that, it's not ideal for writing tests with.

I'll cover this in later chapters, but there is specific data needed in order to successfully run certain tests. Because this is a public app, anyone can change the data on this demo site, which would cause those tests to fail.

My recommendation is: If you don't want to worry about setting up this app on your own right now, you should be able to complete the next few chapters without issue. However, as you get further into the exercises, you may find your tests failing because the data has been modified by someone else. That brings up the next option.

2.0.3 Run Your Own Server Locally

While we won't be modifying the front-end/back-end system code itself, there a good reason you'd want to install a local server. As I mentioned, the data on the demo app is all shared, so someone could change it in a way that breaks the test examples. If you get things set up locally, this won't be an issue.

A word of caution. With either option provided below, **ensure your local server is running when you execute your tests**. Much pain has been caused by testers running their tests, seeing a lot of failures, and not realizing they didn't have their local server running at the time. Because there are a lot of examples to go through and it will likely take you days/weeks to digest the information, there is a good chance your server will be shutdown at some point and you'll need to restart it. Keep that in mind if you run across unexpected failures.

With that said, while I can't walk you through your specific system, here are my general installation instructions.

[63] https://github.com/klamping/vue-realworld-example-app
[64] https://github.com/klamping/node-express-realworld-example-app
[65] https://demo.learnwebdriverio.com/

2.0.4 Using a Docker Image (Recommended)

Docker is an "Open-source software for deploying containerized applications." Chances are, if you know what that means, you're already familiar with Docker.

If not, maybe I can simplify that description. Docker is an application that you can install on your local computer. It allows you to download "images" of applications that can be quickly set up and started locally without configuration or installation of other dependencies. The goal is to make it simple to run applications on your personal computer without having to process through endless installation steps.

There are many well-written guides to Docker already available, so I won't cover in detail how to get it installed and set up. I'm going to assume you're familiar enough with Docker and have it installed for the next set of tasks. If you don't want to use Docker, feel free to skip to the next section.

I've done some work "containerizing" the two apps we need to run in order to have a local server available for our testing. I've also created a `docker-compose.yml` file that you can use to run your own server setup with only a few commands.

First, download the `docker-compose.yml` file[66] (or copy it from below) to somewhere safe for it to live. It can be in its own folder, or a shared folder. Just don't forget where you put it as you'll want to access it later.

Next, open a terminal, then go to the location where you stored your `docker-compose.yml` file. Finally, run `docker-compose up` (alternatively use `docker-compose up -d` to have it run in the background).

Give it a few moments to load the server (you should see the logs stop loading and a "App running at" message), then visit http://localhost:8080[67] to see your local site!

Here's the full docker-compose file for reference:

```
version: '3'
services:
  web:
    container_name: realworld-web
    restart: always
    image: klamping/realworld-web
    environment:
      - APIURL=local
    ports:
      - "8080:8080"
    depends_on:
```

[66] https://gist.githubusercontent.com/klamping/669d63c7f7f712b27bcfc611b22ec41c/raw/936e2b9cb3d7b668d66e6531e1674736c805da46/docker-compose.yml

[67] http://localhost:8080

```
    - api
api:
  container_name: realworld-api
  restart: always
  image: klamping/realworld-api
  environment:
    - NODE_ENV=production
    - SECRET=hunter2
    - MONGODB_URI=mongodb://mongo:27017/conduit
  ports:
    - "3000:3000"
  depends_on:
    - mongo-seed
mongo-seed:
  restart: on-failure
  network_mode: host
  image: klamping/realworld-mongo-seed
  depends_on:
    - mongo
mongo:
  container_name: realworld-mongo
  image: mongo
  ports:
    - "27017:27017"
```

If you want to check on the status of your Docker server, run `docker ps` to get a quick status on all three containers.

2.0.5 Self-install and Run

If you prefer not to go the Docker route, you can try downloading the server source code and run it on your own. There are two code repositories you need to clone:

API Server

Open a terminal and do the following:

1. Run `git clone git@github.com:klamping/node-express-realworld-example-app.git`
2. Change to the created directory and run `npm install`
3. Install MongoDB Community Edition (instructions[68]) and run it by executing `mongod`

[68] https://docs.mongodb.com/manual/installation/#tutorials

4. Import the test data by running: `sh mongoImport.sh`
5. Start the API server with `npm start`

Front-end

Open a terminal and do the following:

1. Run `git clone git@github.com:klamping/vue-realworld-example-app.git`
2. Change to the created directory and run `npm install`
3. Start the Front-end server with `npm run serve`

Give it a few moments to load the server (you should see the logs stop loading and a "App running at" message), then visit http://localhost:8080[69] to see your local site!

2.0.6 A Final Reminder

I mentioned this earlier, but it's worth saying again. As you continue through the examples, you may be coming back to the material after a break. If you run your tests and they're not working, double-check that your server started. You can always do that by opening up http://localhost:8080[70] in a browser. With that said, let's jump into testing!

[69] http://localhost:8080
[70] http://localhost:8080

2.1 Site Loading and Navigation

2.1.1 Avoiding Troubles

I want to jump right in to setting up our WebdriverIO configuration, but first, a warning.

One of the biggest traps I see newcomers face is their ambition. They start off over-optimistic, and use as many tools as they can get their hands on. It's easy (and kind of fun) to come up with a list of features you want your test framework to have. These tools may look easy-to-use on the outside, but getting them all to work together is a very different story.

It's not that they can't work together; they get along just fine most of the time. But when starting out, any mistake made along the way (or difference in setup), can cause errors that are difficult to understand.

Take the following error I've run into before:

```
TypeError: Cannot read property 'trim' of undefined
```

That's not very helpful. Here, have a hint:

```
(node:2155) UnhandledPromiseRejectionWarning: Unhandled promise rejection (rejection
 id: 1): channel closed
(node:2155) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecate
d. In the future, promise rejections that are not handled will terminate the Node.js
 process with a non-zero exit code.
```

So, is that caused by WebdriverIO, the Selenium-Standalone server, CucumberJS or the Allure reporter?

Maybe some more error output will help?

```
/webdriverio/build/lib/utils/BaseReporter.js:336
    throw _iteratorError;
    ^

TypeError: Cannot read property 'trim' of undefined
    at getTestStatus (/node_modules/wdio-allure-reporter/build/reporter.js:68:41)
```

Ah, it must be the Allure reporter, right?

Well, despite Allure throwing the error, it has nothing to do with the reporter. Instead, it's caused by having a CucumberJS step definition with an incorrect number of parameters in the callback function.

Now, if you're unfamiliar with Allure or Cucumber, that may not make sense, but worry not (I'm certain you were concerned). My point isn't to narrow down on a specific error, but to show an example of how introducing extra features too early can really slow you down. I have a fair amount of experience with these two tools and this bug still took me a while to figure out.

I'll state this one last time just for dramatic emphasis: The biggest roadblock newcomers face is not in the fundamentals, but trying to do too much at once.

So we're going to start out simple and introduce new tools and concepts one-by-one. We'll build up until we have a full-featured framework ready to tackle any situation. As errors arise, we'll have fewer dark alleys to look down, making it easier to identify what's going wrong and get back to doing things right.

I'll also be introducing some concepts that won't be sticking around. They'll just be stepping stones to help you get across the scary swamp of some terrible thing that starts with an S (I don't know, it's just for alliteration purposes).

For example, in our first exercise, we'll be talking about assertions. In it, I'll show three different ways to achieve the same effect. We'll only stick with one of them though. I'll explain throughout the process the downfalls of the other options and why I chose my recommended one.

I believe it's really important to know what options you have, how some of them build upon others, and why you'd choose one over the other. Expect to see a fair amount of this throughout the book!

2.1.2 Let's Start

We've already covered installation and basic test writing, but we're going to go over it again. Hopefully a bit quicker this time.

The first step we'll take is to create a new folder to store our up-and-coming tests. **This should not be inside the same folder you used for the 'testrunner' exercise**. So, create a new folder outside of that. You can call it `wdio-realworld` just for fun.

In there, do the usual NPM setup:

```
npm init -y
npm install @wdio/cli
```

Next, let's get that config setup through the WDIO CLI:

```
npx wdio
```

Our answers will be:

```
=========================
WDIO Configuration Helper
=========================

? Where is your automation backend located? On my local machine
? Which framework do you want to use? mocha
? Do you want to run WebdriverIO commands synchronous or asynchronous? sync
? Where are your test specs located? ./test/specs/**/*.js
? Do you want WebdriverIO to autogenerate some test files? No
? Are you using a compiler? No!
? Which reporter do you want to use? spec
? Do you want to add a service to your test setup? chromedriver
? What is the base url? http://localhost:8080/
```

Almost all of the above answers are defaults, except for a few:

- We'll use the `chromedriver` service. Feel free to use `selenium-standalone` if you prefer (or neither).
- For the base url, we use the local server you set up in the previous chapter `http://localhost:8080/`. You're welcome to use the book's demo site here if you'd prefer (https://demo.learnwebdriverio.com[71]), but do note it may cause problems in later chapters.

Okay, now that we have our config ready, it's time to create our test directory. Since we went with the default path, we need to create two directories, `test` and a subdirectory called `specs`. You can do that how you like, but here's how I do it:

In a bash shell (Linux/Mac): `mkdir -p test/specs` In a Windows shell: `mkdir test\specs`

Finally, create a new file inside the specs directory (you can call it `navigation.js`) and open it in your preferred text editor.

- In a bash shell (Linux/Mac): `touch test/specs/navigation.js`
- In a Windows shell: `echo.>test\specs\navigation.js`

2.1.3 Writing our First Real Test

In our file, we'll start things off by writing the `describe` block needed for Mocha to understand we're defining a new test:

[71] https://demo.learnwebdriverio.com

```
describe('Homepage', function () { });
```

We call `describe` like a function, passing in two parameters. The first is a name for the feature we are testing. I put that we're testing the 'Homepage'. This name will be used in our test reporting, which you'll see in a moment.

The second half of our `describe` call is a function which contains all of the code we want to associate with this feature.

Inside this function we'll be adding the `it` call. Similar to `describe`, `it` is a function call that takes two parameters, the first is the name of the specific test (we'll say it 'should take you to the login page'), and the second is a function which contains our actual test code. The full `it` call looks like this:

```
it('should load properly', function () { });
```

Just to recap and refresh on the Mocha syntax:

- `describe` is used to group sets of tests by the feature they are testing.
- `it` defines a specific test to run.

And remember, `it` always goes inside `describe`. Here's what the whole thing looks like:

```
describe('Homepage', function () {
  it('should load properly', function () {
    // test code will go here
  });
});
```

Okay, now that we have our test structure in place, it's time to add our commands. Let's start by loading the main page of the site. You can find a full list of commands on the WebdriverIO site in their API docs[72].

The command we'll use is `browser.url`[73]. This command tells the browser to open a URL (bet you didn't guess that). It does have a trick up its sleeve though.

Remember that `baseUrl` setting we provided? I mentioned this before, but in case you didn't catch it, `browser.url` will prepend that `baseUrl` value if you define a relative path.

For example, if I say `browser.url('./tag/wdio')`, it will load the full URL of `http://localhost:8080/tag/wdio`.

Here are some more complex examples, assuming the `baseUrl` is set to `http://example.com/one/`:

[72] https://webdriver.io/docs/api.html
[73] https://webdriver.io/docs/api/browser/url.html

```
browser.url('./two');
// goes to 'http://example.com/one/two'

browser.url('/two');
// goes to 'http://example.com/two'

browser.url('http://anotherexample.com/one');
// goes to 'http://anotherexample.com/one'
```

Essentially, the `./` at the beginning of the value is used to say "keep the entire `baseUrl`." Alternatively, using just `/` at the start means that it will use only the domain of the URL, stripping off any subfolders.

Finally, you can have it ignore the `baseUrl` by passing your own domain.

All right, let's use a simple `browser.url('./')` in our test to load up the homepage of the site:

```
describe('Homepage', function () {
    it('should load properly', function () {
        // load the page
        browser.url('./');
    });
});
```

To ensure the page has loaded, we're going to log out the title of the page we're on using the `browser.getTitle()` command[74]. Can you guess what that does?

Yes, it gets the title of the page.

In our case, the title is `Conduit`, so we should see that logged out when running our test:

```
describe('Main Navigation', function () {
    it('should take you to the login page', function () {
        // Load the page
        browser.url('./');

        // Get the title of the homepage, should be 'Conduit'
        console.log(browser.getTitle());
    });
});
```

Pretty exciting, I know.

[74] https://webdriver.io/docs/api/webdriver.html#gettitle

2.1.4 Elements and Actions

Now let's try clicking something, like the `Sign In` link.

To do that, we're going to use the `click` command[75], but it won't be `browser.click()` which would throw an error in your face.

No, in order to click an element, you need to let WebdriverIO know which element to click.

If you've used jQuery, which was a very popular JavaScript library from 2006 to 2016-or-so, you'd be familiar with the `$` function. If not, what it does is pick an element out of the HTML based on a "Selector." Selectors are like getting someone's attention by calling their name.

Some selectors can be very specific, like getting the attention of someone named "Averyuniquenamethatnooneelsehas". Some are more general and might get multiple results, like yelling "John" at the store (apologies to the Johns out there and the store manager for yelling in her store). We'll get into selectors more in a little bit, but know that they come in all shapes and sizes.

So WebdriverIO has mimicked the functionality of jQuery and provides its own `$` function[76]. We're going to pretend you know what I mean and use a partial text based selector (we'll talk more about selectors soon).

What element are we getting? Well, we want to click the "Sign In" link at the top of the page. Here's what that HTML looks like:

```
<ul data-qa-id="site-nav" class="nav navbar-nav pull-xs-right">
    <li class="nav-item">
        <a href="/" class="nav-link">
            Home
        </a>
    </li>
    <li class="nav-item">
        <a href="/login" class="nav-link">
            <i class="ion-compose"></i>
            Sign in
        </a>
    </li>
    <li class="nav-item">
        <a href="/register" class="nav-link">
            <i class="ion-compose"></i>
            Sign up
        </a>
    </li>
</ul>
```

[75] https://webdriver.io/docs/api/element/click.html
[76] https://webdriver.io/docs/api/element/\protect\char"0024\relax.html

Note: If you're unfamiliar with HTML syntax structure, I recommend having a watch through this video tutorial teaching HTML basics[77] before moving on.

There are a couple of ways we can specify the 'Sign In' link, but let's go with the simplest. We're going to request an `a` element on the page that has text containing `Sign In`. We could add more to this selector, like the link needs a class name of `nav-link`, but this is good enough for now. As I mentioned, we'll cover this more soon.

To say we want a link that has specific text in it, we write our selector like this: `$('=the text to search by')`. For our needs, we're going to do `$('=Sign In')`.

By using that `$` function with our selector, we now have an element that we can call the `click` command on:

```
// Click the 'Sign In' navigation link
$('=Sign In').click();
```

This triggers a mouse click on that element, which the website will take action on and route us to the 'Sign In' page. How can we know though?

2.1.5 Finishing up

Well, we could log out the title of the page again, but there's a problem: The developers of the website didn't code in a title change. As with many things in testing, we need an alternate form of validation.

Let's use the `getUrl` command[78] instead. I'll let you guess what this command does.

```
// Get the URL of the sign in page. It should include 'login'
console.log(browser.getUrl());
```

That's the end of the test. Here's what the whole file looks like:

test/specs/navigation.js

```
describe('Homepage', function () {
    it('should load properly', function () {
        // load the page
        browser.url('./');

        // Get the title of the homepage, should be 'Conduit'
        console.log(browser.getTitle());
```

[77] https://www.youtube.com/watch?v=UB1O30fR-EE
[78] https://webdriver.io/docs/api/webdriver.html#geturl

```
        // Click the 'Sign in' navigation link
        $('=Sign in').click();

        // Get the URL of the sign in page. It should include 'login'
        console.log(browser.getUrl());
    });
});
```

Make sure it's saved because it's now time to run it. Back on the command line, start your test by entering `npx wdio`. This will kick off the test runner, opening up a Chrome browser.

The test should take about half a minute to complete. Once finished, you'll see log output that looks very similar to this:

```
/webdriverio/build/lib/utils/BaseReporter.js:336
    throw _iteratorError;
    ^

TypeError: Cannot read property 'trim' of undefined
    at getTestStatus
    (/node_modules/wdio-allure-reporter/build/reporter.js:68:41)aExecution of 1 spec
 files started at 2020-07-24T23:11:23.203Z

2020-07-24T23:11:23.231Z INFO @wdio/cli:launcher: Run onPrepare hook
Starting ChromeDriver 84.0.4147.30 (48b3e868b4cc0aa7e8149519690b6f6949e110a8-refs/br
anch-heads/4147@{#310}) on port 9515
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions
 on keeping ChromeDriver safe.
ChromeDriver was started successfully.
2020-07-24T23:11:23.356Z INFO @wdio/cli:launcher: Run onWorkerStart hook
2020-07-24T23:11:23.357Z INFO @wdio/local-runner: Start worker 0-0 with arg:
[0-0] 2020-07-24T23:11:23.755Z INFO @wdio/local-runner: Run worker command: run
2020-07-24T23:11:23.765Z INFO webdriverio: Initiate new session using the ./protocol
-stub protocol
[0-0] RUNNING in chrome - /test/specs/navigation.js
[0-0] 2020-07-24T23:11:23.899Z INFO webdriverio: Initiate new session using the webd
river protocol
[0-0] 2020-07-24T23:11:23.902Z INFO webdriver: [POST] http://localhost:9515/session
[0-0] 2020-07-24T23:11:23.902Z INFO webdriver: DATA {
  capabilities: {
    alwaysMatch: { browserName: 'chrome', acceptInsecureCerts: true },
    firstMatch: [ {} ]
```

```
  },
  desiredCapabilities: { browserName: 'chrome', acceptInsecureCerts: true }
}
[0-0] 2020-07-24T23:11:26.215Z INFO webdriver: COMMAND navigateTo("http://localhost:
8080/")
[0-0] 2020-07-24T23:11:26.216Z INFO webdriver: [POST] http://localhost:9515/session/
92360947699c9250dbef80ada5d299fd/url
[0-0] 2020-07-24T23:11:26.217Z INFO webdriver: DATA { url: 'http://localhost:8080/'
}
[0-0] 2020-07-24T23:11:27.928Z INFO webdriver: COMMAND getTitle()
2020-07-24T23:11:27.928Z INFO webdriver: [GET] http://localhost:9515/session/9236094
7699c9250dbef80ada5d299fd/title
[0-0] 2020-07-24T23:11:27.940Z INFO webdriver: RESULT Conduit
[0-0] Conduit
[0-0] 2020-07-24T23:11:27.941Z INFO webdriver: COMMAND findElement("link text", "Sig
n in")
[0-0] 2020-07-24T23:11:27.941Z INFO webdriver: [POST] http://localhost:9515/session/
92360947699c9250dbef80ada5d299fd/element
[0-0] 2020-07-24T23:11:27.941Z INFO webdriver: DATA { using: 'link text', value: 'Si
gn in' }
[0-0] 2020-07-24T23:11:27.969Z INFO webdriver: RESULT {
  'element-6066-11e4-a52e-4f735466cecf': '34d561d8-4300-48b5-a085-b30dce080609'
}
[0-0] 2020-07-24T23:11:27.975Z INFO webdriver: COMMAND elementClick("34d561d8-4300-4
8b5-a085-b30dce080609")
2020-07-24T23:11:27.975Z INFO webdriver: [POST] http://localhost:9515/session/923609
47699c9250dbef80ada5d299fd/element/34d561d8-4300-48b5-a085-b30dce080609/click
[0-0] 2020-07-24T23:11:28.052Z INFO webdriver: COMMAND getUrl()
[0-0] 2020-07-24T23:11:28.053Z INFO webdriver: [GET] http://localhost:9515/session/9
2360947699c9250dbef80ada5d299fd/url
[0-0] 2020-07-24T23:11:28.102Z INFO webdriver: RESULT http://localhost:8080/login
[0-0] http://localhost:8080/login
[0-0] 2020-07-24T23:11:28.109Z INFO webdriver: COMMAND deleteSession()
[0-0] 2020-07-24T23:11:28.110Z INFO webdriver: [DELETE] http://localhost:9515/sessio
n/92360947699c9250dbef80ada5d299fd
[0-0] PASSED in chrome - /test/specs/navigation.js
2020-07-24T23:11:28.334Z INFO @wdio/cli:launcher: Run onComplete hook

 "spec" Reporter:
------------------------------------------------------------------
[chrome 84.0.4147.89 mac os x #0-0] Spec: /Users/klamping/wdio-testrunner/test/specs
/navigation.js
[chrome 84.0.4147.89 mac os x #0-0] Running: chrome (v84.0.4147.89) on mac os x
```

```
[chrome 84.0.4147.89 mac os x #0-0] Session ID: 92360947699c9250dbef80ada5d299fd
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] Homepage
[chrome 84.0.4147.89 mac os x #0-0]    ✓ should load properly
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1 passing (2.2s)


Spec Files:  1 passed, 1 total (100% completed) in 00:00:05

2020-07-24T23:11:28.336Z INFO @wdio/local-runner: Shutting down spawned worker
2020-07-24T23:11:28.589Z INFO @wdio/local-runner: Waiting for 0 to shut down gracefu
lly
2020-07-24T23:11:28.589Z INFO @wdio/local-runner: shutting down
```

In the middle of all that output are two lines with the content we need. Let's hide all of those extra messages so we get just those two `console.log`'s we used by passing in the `logLevel` setting: `npx wdio --logLevel=silent`. Now our log output will be a lot shorter:

```
Execution of 1 spec files started at 2020-07-24T23:12:13.610Z

Starting ChromeDriver 84.0.4147.30 (48b3e868b4cc0aa7e8149519690b6f6949e110a8-refs/br
anch-heads/4147@{#310}) on port 9515
Only local connections are allowed.
Please see https://chromedriver.chromium.org/security-considerations for suggestions
 on keeping ChromeDriver safe.
ChromeDriver was started successfully.
[0-0] RUNNING in chrome - /test/specs/navigation.js
[0-0] Conduit
[0-0] http://localhost:8080/login
[0-0] PASSED in chrome - /test/specs/navigation.js

 "spec" Reporter:
------------------------------------------------------------------
[chrome 84.0.4147.89 mac os x #0-0] Spec: /Users/klamping/wdio-testrunner/test/specs
/navigation.js
[chrome 84.0.4147.89 mac os x #0-0] Running: chrome (v84.0.4147.89) on mac os x
[chrome 84.0.4147.89 mac os x #0-0] Session ID: dd4009892dde40a2ffef0088bdfd27a5
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] Homepage
[chrome 84.0.4147.89 mac os x #0-0]    ✓ should load properly
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1 passing (2.3s)
```

```
Spec Files:  1 passed, 1 total (100% completed) in 00:00:05
```

As you see, the page title and URL are logged out. Looks like the site functions as hoped and our test passes!

2.1.6 Adding Assertions

So far, we validated that our test successfully ran by checking the log output after execution. This is simple enough to do while we only have a test or two, but it quickly becomes tedious as more and more tests are added.

Instead of us manually checking the values, what if we could tell the computer what to expect and have it check for us? This is what assertions do. They check a value and "assert" that it's a certain way (e.g., it equals another value, it doesn't equal another value, it matches part of another value). If the value isn't what's expected, an error is thrown and the test is marked as failing.

Mocha watches for these assertion errors and catches them during the tests, marking the test as failed if one occurs.

We can update our test to "throw" an error if the values we're looking for aren't correct.

Let's take that first log message. Replacing it, we want to assert that the title is `Conduit`. A simple `if` statement will do the trick:

```
if (browser.getTitle() !== 'Conduit') {
    // throw an error explaining what went wrong
    throw new Error('Title of the homepage should be "Conduit"');
}
```

Now if the title isn't what we're looking for, it'll jump into the if statement and we can do our dirty work throwing an error. In JavaScript, we do that by saying `throw new Error('My error message')`. You can see the custom error message I include in the previous example.

Let's make the update for our `getUrl` call as well:

```
if (browser.getUrl() !== 'http://localhost:8080/login') {
    throw new Error('URL of "login" page should be correct');
}
```

Here's the entirety of our updated test:

```
describe('Homepage', function () {
    it('should load properly', function () {
        // load the page
        browser.url('./');

        // Get the title of the homepage, should be 'Conduit'
        if (browser.getTitle() !== 'Conduit') {
            // throw an error explaining what went wrong
            throw new Error('Title of the homepage should be "Conduit"');
        }

        // Click the 'Sign in' navigation link
        $('=Sign in').click();

        // Get the URL of the sign in page. It should include 'login'
        if (browser.getUrl() !== 'http://localhost:8080/login') {
            throw new Error('URL of "login" page should be correct');
        }
    });
});
```

If we were to run our tests again, the only difference we'd see is that it no longer logs out those two messages (assuming the site didn't break in the time between our test runs). So how do we know our `if` conditions work? Well, say we changed one of the `if` conditions to no longer match the site values (e.g., `if (browser.getTitle() !== 'The wrong title') {` and ran the test again, you'd see the following message:

```
[0-0] RUNNING in chrome - /test/specs/navigation.js
[0-0] Error in "Homepage should load properly"
Title of the homepage should be "Conduit"
[0-0] FAILED in chrome - /test/specs/navigation.js

 "spec" Reporter:
------------------------------------------------------------------
[chrome 84.0.4147.89 mac os x #0-0] Spec: /Users/klamping/wdio-testrunner/test/specs
/navigation.js
[chrome 84.0.4147.89 mac os x #0-0] Running: chrome (v84.0.4147.89) on mac os x
[chrome 84.0.4147.89 mac os x #0-0] Session ID: f237a09f87b12748ed7ef7e3ac862554
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] Homepage
[chrome 84.0.4147.89 mac os x #0-0]    x should load properly
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1 failing (2.7s)
```

```
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1) Homepage should load properly
[chrome 84.0.4147.89 mac os x #0-0] Title of the homepage should be "Conduit"
[chrome 84.0.4147.89 mac os x #0-0] Error: Title of the homepage should be "Conduit"

[chrome 84.0.4147.89 mac os x #0-0]     at Context.<anonymous> (/Users/klamping/wdio
-testrunner/test/specs/navigation.js:9:19)
```

Now our test fails because it didn't get what it was expecting. We no longer need to check the log messages, having added a little more automation in our tests.

2.1.7 Expanding Assertions

In the real world, you'll rarely see errors thrown like this. Instead, developers use "Assertion Libraries". These libraries make writing assertions and throwing errors a lot simpler. Let's spend the rest of the chapter looking at a few options.

The first Assertion Library we'll look at comes built in to the Node.js ecosystem. The NodeJS Assert API[79] is a standard library in Node.js, and we can use it without any extra installations.

At the top of our test file, add the following line to load this API:

```
const assert = require('assert');
```

Pay attention that this statement is outside the `describe` block. If we were to include it inside the `describe` block, it would only be available in that section. Any other `describe` blocks would also have to include the statement. That's because variables declared inside the function block are scoped to that function. This is normal JavaScript behavior and is something to watch out for when writing tests.

Next, we need to replace our `if` statement and thrown error with a call to the assert library. There are many ways to assert a value:

```
const myValue = 'a';

assert(myValue); // You can check that it's a 'truthy' value
assert.strictEqual(myValue, 'a'); // You can check that it's equal to something
assert.notStrictEqual(myValue, 'b'); // You can check that it doesn't equal something
```

We want to validate that the page title is a specific string of text. To do this, we use `assert.strictEqual` and pass in the actual value and then the expected value. The actual value is the result of the `getTitle` call and the expected value is a simple string of text containing our page title.

[79] https://nodejs.org/api/assert.html#assert_assert

```
assert.strictEqual(browser.getTitle(), 'Conduit');
```

Now if our test fails, it will look like:

```
[0-0] RUNNING in chrome - /test/specs/navigation.js
[0-0] AssertionError in "Homepage should load properly"
Expected values to be strictly equal:
+ actual - expected

+ 'Conduit'
- 'The Wrong Title'
[0-0] FAILED in chrome - /test/specs/navigation.js

 "spec" Reporter:
------------------------------------------------------------------
[chrome 84.0.4147.89 mac os x #0-0] Spec: /Users/klamping/wdio-testrunner/test/specs
/navigation.js
[chrome 84.0.4147.89 mac os x #0-0] Running: chrome (v84.0.4147.89) on mac os x
[chrome 84.0.4147.89 mac os x #0-0] Session ID: 1adcc0cea040b954eb9722daa2b8f49b
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] Homepage
[chrome 84.0.4147.89 mac os x #0-0]    x should load properly
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1 failing (2.2s)
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1) Homepage should load properly
[chrome 84.0.4147.89 mac os x #0-0] Expected values to be strictly equal:
+ actual - expected

+ 'Conduit'
- 'The Wrong Title'
[chrome 84.0.4147.89 mac os x #0-0] AssertionError [ERR_ASSERTION]: Expected values
to be strictly equal:
[chrome 84.0.4147.89 mac os x #0-0] + actual - expected
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] + 'Conduit'
[chrome 84.0.4147.89 mac os x #0-0] - 'The Wrong Title'
[chrome 84.0.4147.89 mac os x #0-0]     at Context.<anonymous> (/Users/klamping/wdio
-testrunner/test/specs/navigation.js:15:16)
```

Not only is the assertion a little less code to write, it's actually a more helpful message in that it provides the expected and actual values. Assuming we've updated our second check as well, here's what our test now looks like:

```
const assert = require('assert');

describe('Homepage', function () {
    it('should load properly', function () {
        // load the page
        browser.url('./');

        // Get the title of the homepage, should be 'Conduit'
        assert.strictEqual(browser.getTitle(), 'Conduit');

        // Click the 'Sign in' navigation link
        $('=Sign in').click();

        // Get the URL of the sign in page. It should include 'login'
        assert.strictEqual(browser.getUrl(), 'http://localhost:8080/login');
    });
});
```

2.1.8 The WebdriverIO Expect Library

One of the valuable additions of the Version 6 upgrade includes the bundling of "WebdriverIO Expect" assertions[80]. This library makes writing assertions cleaner by adding support for WebdriverIO syntax.

It's similar to how Node's `assert` library works, but the syntax is a bit different. For example, take the assertions we just looked at:

```
assert.strictEqual(browser.getTitle(), 'Conduit');
assert.strictEqual(browser.getUrl(), 'http://localhost:8080/login');
```

With the `expect-webdriverio` library, they can be changed to:

```
expect(browser).toHaveTitle('Conduit');
expect(browser).toHaveUrl('http://localhost:8080/login');
```

Can you tell why I like this format? To me, it's more sentence-like compared to the assert code: We expect the `browser` to have a title of 'Conduit'. I like when my tests read more like specifications and less like code.

We also don't need to load any libaries, as it's included in the base WebdriverIO installation. Here's what the updated test file looks like:

[80] https://webdriver.io/docs/assertion.html

```
describe('Homepage', function () {
    it('should load properly', function () {
        // load the page
        browser.url('./');

        // Get the title of the homepage, should be 'Conduit'
        expect(browser).toHaveTitle('Conduit');

        // Click the 'Sign in' navigation link
        $('=Sign in').click();

        // Get the URL of the sign in page. It should include 'login'
        expect(browser).toHaveUrl('http://localhost:8080/login');
    });
});
```

And while functionally it's much the same, there are a few features that are worth noting:

- It's easier to write (in my opinion, you're welcome to disagree)
- You get better error messages
- It automatically retries failed assertions

By "better error messages", let's compare the "failure" output if our title was incorrect:

Using `assert.strictEqual(browser.getTitle(), 'Conduit')`:

```
[chrome 83.0.4103.106 mac os x #0-0] Input A expected to strictly equal input B:
+ expected - actual

- 'The Wrong Title'
+ 'Conduit'
```

It's helpful, but let's see what it looks like with `expect(browser).toHaveTitle('Conduit')`:

```
[chrome 83.0.4103.106 mac os x #0-0] Expect window to have title
Expected: "Conduit"
Received: "Incorrect Page Title"
[chrome 83.0.4103.106 mac os x #0-0] Error: Expect window to have title
[chrome 83.0.4103.106 mac os x #0-0]
[chrome 83.0.4103.106 mac os x #0-0] Expected: "Conduit"
[chrome 83.0.4103.106 mac os x #0-0] Received: "The Wrong Title"
[chrome 83.0.4103.106 mac os x #0-0]     at Context.<anonymous>
(/Users/klamping/Sites/wdio/test/specs/home.js:5:25)
```

Now our message provides not just what the two conflicting values are, but where they came from (the window title). Small improvements like this can really help speed along debugging efforts, as you're able to more quickly understand what is failing. You don't have to look through the code to understand that it's the page title that's wrong.

I also mentioned that the 'expect' functionality adds "automatic retries". It will automatically retry the assertion multiple times (assuming it's failing), which helps if the page isn't quite ready for the test yet. While this likely won't be helpful for checking the page title, it can be very valuable for asserting that page elements exist (or don't).

Having elements appear and disappear is an essential part of today's Web Apps, but that can definitely make testing more difficult, as you need to account for this. We'll get into this more later on, but here's a quick example. With an assertion like `expect($('#someElem')).toExist()`, WebdriverIO will try multiple times checking for that element to exist, (in case your assertion ran a second or two too early). This can really help simplify the code you need to write, as the waiting is now implicit (although it can be configured to not retry if you'd like).

2.1.9 Exceeding Expectations

Our new assertions are pretty nice, especially since we didn't have to install anything, but we have an issue with them.

Looking at the second assertion, it's limited in its flexibility. In it, we validated the entire URL. But what if we want to test a different instance of the same site? Our assertion would fail even though the navigation to the Login page still worked (because the top-level domain is different, even though the `/login` part of the URL was correct).

We want the test to pass no matter the domain name (since we could be testing this same site on any number of different domains). Right now, with our `toHaveUrl` assertion, it will only pass if we're testing the `http://localhost:8080` site.

Thankfully, there are options. Literally. The `expect-webdriverio` additions comes with many options[81] to customize our assertions.

[81] https://webdriver.io/docs/api/expect.html#matcher-options

For our needs, we can pass a `containing` boolean to tell it to "expect actual value to contain expected value". All options are passed in as an object after our expected value. For our URL check, that will be: `expect(browser).toHaveUrl('/login', { containing: true });`. Notice that we don't have to define every option available, only those we want to customize. There are several other options avialable that you'll want to be aware of. Again, check out the documention[82] for more information.

Here's the fully updated file:

```
describe('Homepage', function () {
    it('should load properly', function () {
        // load the page
        browser.url('./');

        // Get the title of the homepage, should be 'Conduit'
        expect(browser).toHaveTitle('Conduit');

        // Click the 'Sign in' navigation link
        $('=Sign in').click();

        // Get the URL of the sign in page. It should include 'login'
        expect(browser).toHaveUrl('/login', { containing: true });
    });
});
```

With our test updated, save the file and give it another run. Assuming no errors, the output should look the same. If I fail one of the assertions, however, you may see something like:

```
[0-0] RUNNING in chrome - /test/specs/navigation.js
[0-0] Error in "Homepage should load properly"
Expect window to have url containing

Expected: "/other-login"
Received: "http://localhost:8080/login"
[0-0] FAILED in chrome - /test/specs/navigation.js

 "spec" Reporter:
------------------------------------------------------------------
[chrome 84.0.4147.89 mac os x #0-0] Spec: /Users/klamping/wdio-testrunner/test/specs
/navigation.js
[chrome 84.0.4147.89 mac os x #0-0] Running: chrome (v84.0.4147.89) on mac os x
[chrome 84.0.4147.89 mac os x #0-0] Session ID: 401a1cda9409618a995325421823eae4
[chrome 84.0.4147.89 mac os x #0-0]
```

[82] https://webdriver.io/docs/api/expect.html#matcher-options

```
[chrome 84.0.4147.89 mac os x #0-0] Homepage
[chrome 84.0.4147.89 mac os x #0-0]    x should load properly
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1 failing (11.9s)
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] 1) Homepage should load properly
[chrome 84.0.4147.89 mac os x #0-0] Expect window to have url containing

Expected: "/other-login"
Received: "http://localhost:8080/login"
[chrome 84.0.4147.89 mac os x #0-0] Error: Expect window to have url containing
[chrome 84.0.4147.89 mac os x #0-0]
[chrome 84.0.4147.89 mac os x #0-0] Expected: "/other-login"
[chrome 84.0.4147.89 mac os x #0-0] Received: "http://localhost:8080/login"
[chrome 84.0.4147.89 mac os x #0-0]     at Context.<anonymous>
(/Users/klamping/wdio-testrunner/test/specs/navigation.js:13:25)

Spec Files:  0 passed, 1 failed, 1 total (100% completed) in 00:00:14
```

One final note for now on this. Aside from the WebdriverIO specific assertions `expect-webdriverio` adds, you also gain access to the built-in ExpectJS assertions[83]. Some example assertions are:

```
expect(browser.getUrl()).toEqual('https://jestjs.io/docs/en/expect');
// compared to
expect(browser).toHaveUrl('https://jestjs.io/docs/en/expect');

// Check that there are exactly five links on the page
const links = $$('a');
expect(links).toHaveLength(5);
```

We also can use the `.not` switch to flip our assertion:

```
expect(browser).not.toHaveUrl('https://jestjs.io/docs/en/expect');

// Check that there are not exactly five links on the page
const links = $$('a');
expect(links).not.toHaveLength(5);
```

We'll take a look at using more of these as we move forward.

But that finishes off our first test, providing us with the foundation to move towards more complex checks down the road. Assertions are going to be essential in our automation, so taking the time to understand how they work is well worth it.

[83] https://jestjs.io/docs/en/expect

Next up, we'll look in-depth at a plethora of element selectors, along with how we can wait for content to load and do other funny things.

2.1.10 Chapter Challenge

Over the following chapters, I'll include a "Chapter Challenge" at the end of each of them. This is a challenge for you to tackle on your own if you're interested. Brief explanations and my recommended answer (but not necessarily the only answer) are in Chapter 3.0 near the end of this book.

For your first challenge, in your existing test, add a check that validates that when you click on the "Conduit" logo in the top left of the site, it returns you to the homepage.

2.2 Selectors of Every Shape and Size

2.2.1 Selector Mania

In the previous chapter, amid the talk of actions and assertions, we touched on selectors a little bit. We used a link text selector to pick out a link with the text "Sign In" in it. Now it's time to take a full look at all the options we have when it comes to picking elements.

Let's get started with the official W3C list of supported selector types:

- CSS selector
- Link text selector
- Partial link text selector
- Tag name
- XPath selector

Also, I should be calling all of these "Locator Strategies[84]", which is how they're referenced in the W3C specification.

There are actually a few other "unofficial" strategies[85], but I don't think they're worth mentioning because they're pretty much obsolete these days.

Normally, when using one of these strategies, you need to specify which one you want to use. In many test libraries, you'll see something akin to `element(by.partialButtonText('About Us'))`. WebdriverIO helps us out here by inferring what type of strategy you want to use[86], so you don't have to do as much typing.

Selectors by Example

To showcase the different selectors strategies, let's use the following HTML:

[84] https://w3c.github.io/webdriver/#locator-strategies
[85] https://stackoverflow.com/a/48376890/150552
[86] https://github.com/webdriverio/webdriverio/blob/master/packages/webdriverio/src/utils/findStrategy.js

```
<ul class="items" dropdown-menu id="main-menu">
    <li class="item">
        <a ng-click="showProjects()">
            Projects
            <span class="icon"></span>
        </a>
    </li>
    <li class="item">
        <a ng-click="showArchive()">
            Archive
            <span class="icon"></span>
        </a>
    </li>
</ul>
```

The simplest type of selector in my mind would be "tag name". It selects any matching element with the tag name specified. So if we wanted to choose the `ul` element, in WebdriverIO we'd do `$('ul')`. If we wanted to get the first `li`, we'd do `$('li')`. But what if we wanted the second one?

There are actually several ways to achieve this, but the one we'll look at first is to select both `li` elements, then pick out the second one.

Similar to the `$` function, WebdriverIO also provides a `$$` function, which returns an array of all of the matching elements. Where `$` returns the first matching element, `$$` returns all of them.

If we use `$$('li')`, we'll get an array with two elements in it. We can use JavaScript array syntax to select the second one: `$$('li')[1]`. Remember, arrays in JavaScript start at `0` (zero-indexed), so to choose the second item we use `1`.

Link text selectors and partial link text selectors are also pretty simple. To get the first link, we could do `$('=Projects')`. To get the second `li` using a partial link text selector, we can do `$('*=chive')`. By adding an asterisk (i.e., `*`), WebdriverIO will look for a partial text match, not a full one.

Also worth noting... while WebDriver officially only supports link based text selectors, WebdriverIO takes it one step further and provides support for any type of element to be checked. It does this by converting what looks like a text selector into an XPath one (more on XPath in a second). So we could also do `$('li*=chice')` as well and it would still work (since there is an `<li>` element which contains the text `chive`). That's pretty neat!

2.2.2 CSS Selectors

If you've done any front-end web development, you've likely worked with CSS selectors. They're the backbone not just for styles, but also for JavaScript as well.

CSS selectors have quite a legacy and feature set, and an entire course could be put together covering the various forms they can take. To use a CSS selector in WebdriverIO, just pass it into a `$` or `$$` function. Here's one that grabs the `ul` by class name: `$('ul.items')`, and one that grabs the two `li` elements by class as well: `$$('li.item')`.

If you're new to CSS, I highly recommend running through a few tutorials on selectors. Here are a few recommendations:

- TutsPlus "The 30 CSS Selectors You Must Memorize"[87]
- Sauce Labs "CSS Selector Tips"[88]
- CSS Tricks "CSS Almanac / Selectors"[89]
- Ghost Selector "CSS Selector Strategies for Automated Browser Testing"[90]

Getting to know all your options with CSS selectors is quite valuable. The majority of styles out there rely on class or ID-based selectors, but it definitely can be an advantage to know about other types as well. Here are couple more to know about:

Attribute Selectors

Attributes are the properties of HTML elements (e.g., the "class" part of `<li class="item">`). These attributes can be helpful for singling out specific elements.

Using CSS Attribute Selectors[91] (shorter version[92]), we can target the `ng-click` attribute of the link, like so:

```
$('[ng-click="showProjects()"]');
```

Notice how the selector includes both the attribute and its value. You don't have to include both, but for our needs we needed to target the element with that specific `ng-click` value.

We can also combine attribute selectors to be more specific with our link:

```
$('[dropdown-menu] a[ng-click="showProjects()"]');
```

That selector asks for a link with an attribute of `ng-click` and an attribute value of `showProjects()` that is the child of an element with an attribute of `dropdown-menu` (which we didn't include a value for since there is none).

[87] https://code.tutsplus.com/tutorials/the-30-css-selectors-you-must-memorize--net-16048
[88] https://saucelabs.com/resources/articles/selenium-tips-css-selectors
[89] https://css-tricks.com/almanac/selectors/
[90] https://ghostinspector.com/blog/css-selector-strategies-automated-browser-testing/
[91] https://css-tricks.com/attribute-selectors/
[92] https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors

nth-child Selectors

Another method to find your element is to base it on the position in the HTML. The `first-child` selector[93] allows us to do this:

```
$('li:first-child');
```

That will select any list item (`<li>`) that is the first child of its parent. That's a pretty broad selector though, so we should narrow this down to our dropdown using the attribute selector from before:

```
$('[dropdown-menu] li:first-child');
```

What if we wanted to get the second child? We can use The `nth-child` selector[94]:

```
$('[dropdown-menu] li:nth-child(2)');
```

`nth-child` takes a numeric value, which corresponds to the position in the HTML (it is *not* zero-indexed, so counting starts at 1).

`nth-child` can get pretty complicated and is a very powerful selector. There are also related selectors like `nth-of-type` or `nth-last-child`. CSS-Tricks[95] and nthmaster.com[96] go into more detail on it.

Why use `nth-child` if you could just get all elements using `$$` and grab the one you want by index? Well, selecting multiple elements is consistently slower than grabbing a single one, so using `nth-child` can help keep our tests speedy.

To restate that, using `$('li:nth-child(2)')` will be faster than using `$$('li')[1]`. Should you always use `:nth-child` then? Probably not. There are definitely times when you want to get all elements, and `:nth-child` doesn't work well with nested elements, but it's a good trick to keep in mind.

2.2.3 XPath

The final type of selector I'll mention is XPath. Traditionally XPath has been the predominant selector strategy for Selenium/WebDriver, so many folks in the testing industry are most familiar with it.

To use in WebdriverIO, you pass it in just like you would a CSS selector. So to grab that `ul` by ID: `$('//ul[@id="main-menu"]')`. And to get both `li` elements by class: `$$('//li[@class="item"]')`.

As I mentioned with CSS, if you're new to XPath, I recommend running through a few tutorials on its usage. Here are a couple recommendations:

[93] https://developer.mozilla.org/en-US/docs/Web/CSS/:first-child
[94] https://css-tricks.com/how-nth-child-works/
[95] https://css-tricks.com/examples/nth-child-tester/
[96] http://nthmaster.com/

- MDN XPath documentation[97]
- Scrapy XPath Tutorial[98]

So which one should you use, XPath or CSS selectors? Truthfully, either one works. I end up using both depending on the circumstance. CSS selectors are normally more brief than XPath, but sometimes they're not as powerful. Which one you use is really a matter of what you're more comfortable with. Front-end developers are likely more familiar with CSS selectors, so I lean that way, but there are some things CSS can't do, which is when I'll choose XPath.

If you're looking to compare the two, this cheatsheet put together by devhints.io is invaluable[99].

There are two big tricks that XPath can do that CSS currently can't. The first is to select an element by its text content, which looks something like: `//a[contains(text(),"chive")]`. The other is the ability to move "up" from an element. So you can use XPath to select a parent of an element, which isn't possible yet in CSS.

We can combine these two techniques to find the parent container of our "Archive" link:

```
//a[contains(text(),"chive")]/ancestor::ul
```

This technique can be very helpful when you're limited in customizing the HTML of a page and need to rely on what the text says.

Regardless of CSS vs. XPath, you can use the Chrome Developer tools to help evaluate and validate XPath/CSS selectors[100] you're working on. This is a useful tool for building and testing selectors.

2.2.4 Chaining Selectors

Aside from the strategies we've touched on so far, there's one neat trick that I love to take advantage of (and no, doctors won't hate that you know this[101]).

Many times you'll have a container element that you want to use as reference for future searches. With WebdriverIO, you can select your parent container, then search from within that element.

From our example, we can select the link we want using a text selector, then select the `span` inside that link using a plain old tag selector: `$('=Projects').$('span')`. It wouldn't work to do `$('=Projects span')` since WebdriverIO would think you're looking for a link with the text `projects span`.

You can also search for multiple elements from a single parent. To get both `li` elementss from the parent `ul`, do `$('ul').$$('li')`. Of course you could just do `$$('ul li')` in this case, but I needed to show off the other way so hush.

[97] https://developer.mozilla.org/en-US/docs/Web/XPath
[98] https://docs.scrapy.org/en/xpath-tutorial/topics/xpath-tutorial.html
[99] https://devhints.io/xpath
[100] http://yizeng.me/2014/03/23/evaluate-and-validate-xpath-css-selectors-in-chrome-developer-tools/
[101] https://skeptoid.com/blog/2013/01/28/one-weird-trick-to-rule-them-all/

2.2.5 Custom Data Attributes for Testing

If you do have the ability to customize the HTML, I highly recommend adding custom HTML attributes to help identify the elements you need to select. This is a popular practice in the industry and can really help provide the specificity you need.

It works by adding a custom attribute to your HTML components which is used solely for testing selectors. HTML5 introduced a formal "data" attribute type[102] that we can use in our tests.

For example, let's edit that HTML I showed earlier to use data attributes:

```
<ul class="items" dropdown-menu id="main-menu" data-qa-id="main-menu">
    <li class="item" data-qa-id="main-menu-item">
        <a ng-click="showProjects()" data-qa-id="main-menu-item-link">
            Projects
            <span class="icon" data-qa-id="main-menu-item-icon"></span>
        </a>
    </li>
    <li class="item" data-qa-id="main-menu-item">
        <a ng-click="showArchive()" data-qa-id="main-menu-item-link">
            Archive
            <span class="icon" data-qa-id="main-menu-item-icon"></span>
        </a>
    </li>
</ul>
```

In this example, I added a `data-qa-id` attribute to all of the elements. It wasn't necessary for some (for example, the `id` of the main menu is the same as the `data-qa-id`), but useful overall. When targeting an element, we'll now rely on our data attribute: `$('[data-qa-id="main-menu"]')`.

The real benefit is that it's much less likely to be altered over the life of the project. Class names may change inadvertently by a front-end dev refactoring the HTML, or an ID may be changed by a JavaScript dev disgruntled by the current name. A custom attribute named with `qa-id` is much less likely to be changed without warning.

We're going to take full advantage of this during our test writing, as the HTML currently available on our test site doesn't quite allow us the specificity we'll need for long-term stability.

2.2.6 Avoiding Poorly Built Selectors

We've covered a fair amount of strategies for selectors, and for good reason. Poorly chosen selectors are a prime candidate for flaky tests.

[102] https://developer.mozilla.org/en-US/docs/Learn/HTML/Howto/Use_data_attributes

Many times I see selectors that look like:

```
$('//*[@id="stream-item-1130810481587957760"]/div[1]/div[2]/a/span[1]/strong')
```

The problem with this type of specificity is that almost any change to the HTML will break your tests. An extra HTML element thrown in or one removed will cause chaos in your code.

Instead, adding a custom data attribute as mentioned above can really limit the effect of an HTML restructuring. Even if an element is completely restructured, so long as the `data-id` stays intact it should still work.

When it comes to selecting elements, I like to follow these general guidelines:

- If I can change the HTML, add a custom data attribute for testing purposes only (this way they don't inadvertently get changed)
- If I don't have control of the HTML, come up with selectors that are specific, but not overly tied to the HTML structure. For example, using simple parent-child relationships, `:nth-child` selectors or text based ones. It's not perfect, but it gets the job done.

2.2.7 Chapter Challenge

The Conduit website doesn't have the best options when it comes to selectors (we'll work on that), but it's manageable. Try creating selectors for the following site elements:

- The "An interactive learning project from Thinkster. Code & design licensed under MIT." text in the page footer
- All "Tags" in the "Popular Tags" sidebar
- All "Tags" in the "Popular Tags" sidebar (using "Popular Tags" header text). Hint: use `text()` and `following-sibling`

2.3 Testing the Login Page

2.3.1 Is This Thing Working?

In our first test, we ran some basic commands to validate that our Conduit site loaded correctly and that you could navigate to the "Sign In" page. While it's a good start, the test doesn't provide much value. It would be much easier just to use a Website Uptime Monitoring service if that's all you're really checking.

But that's not the only thing we want to do with our tests. Website monitoring is important, but we know many bugs lurk deep inside the pages themselves. These nefarious bugs hide out in the corners of closets at the end of long, dark hallways. To get there, you've got to do more than just check that the building is still standing.

Hopefully writing these tests won't be as spooky as I've alluded to (although I have seen some scary looking code in my life). In fact, writing our next set of tests will only take a few commands, so let's get into it!

I always like starting my test writing by listing the names of the tests I'm going to run, then commenting on what each step will be. That way I don't have to worry about specific code to use, but still get a good idea of how the test will be set up.

For our login page, create a new file called `login.js` that goes inside the same `test/specs` folder that your navigation file went into. Then, build out the file with our comments:

```
// Login Page:
// should let you log in
// should error with a missing username
// should error with a missing password
```

We can certainly add more test cases, but these three are a good start. Don't worry about how to test those error scenarios; all we're thinking about right now is that first one.

One more thing about commenting like this. I really like this approach, as it lets me focus on brainstorming all the page requirements. While it would be ideal to get an official list of requirements for every page you're testing, too often testers are left on their own.

By ignoring the code specifics at the start, you allow your mind to really think about the page itself. Conversely, once you start writing the code for your tests, your brain narrows its focus to the test specifics, restricting your view of the overall page. Keep this in mind when you're in the middle of a project and feel your wheels spinning, trying to get your tests working. Maybe you need to step back and look at the bigger picture of what you're really trying to achieve.

Let's go back to our test. With our comments laid out, let's start translating them into code. First, we'll set up our describe/it block:

```
describe('Login Page', function () {
    it('should let you log in', function () {
    });
    // should error with a missing username
    // should error with a missing password
});
```

I took the text straight from the comments and turned them into the structure for my first test. Don't think we're done with comments! In fact, that's our next step.

Similar to how we wrote the comments for the overall test scenarios, we want to do the same thing with our individual test steps. Let's think through what this "should log in" test will do:

- go to the login page
- enter a valid username in the "email" input
- enter a valid password in the "password" input
- click the 'Sign In' button
- assert that we're logged in

Notice how I'm not specific on what username we're using, what the input selectors are going to be, or how we're going to assert that we're logged in. Again, right now, it's all about brainstorming how the test will run, ignoring any specifics that may narrow your vision.

Let's throw those comments into the file:

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
      // go to the login page
      // enter a valid username in the "email" input
      // enter a valid password in the "password" input
      // click the 'Sign In' button
      // assert that we're logged in
    });

    // should error with a missing username
    // should error with a missing password
});
```

With the outline made, let's turn this into real code.

Step 1: Go to the Login Page

In our navigation test, we already achieved the effect of going to the sign in page, so we could just copy that code over and re-use it for this step. But I think that's a bad idea.

First, it's going to take extra time to load the homepage, then find the 'Sign In' link, then click it. WebDriver tests are already slower than most other tests, so let's not waste time taking unnecessary steps.

Second, if the "Sign In" link changes or altered, it could easily break this entire set of tests. Even if the "Sign In" page was working entirely fine, your whole file would be failing due to outside circumstances.

Finally, if you do want to validate that your links work as expected, have that be it's own separate tests. Don't mix responsibilities here. This step is better saved for an individual "links" test, maybe even one that doesn't rely on WebdriverIO to run (and therefore may execute faster).

With those reasons in mind, the best approach is to use a simple `browser.url` call to load the `login` url we referenced in our previous assertion. Adding that to the code, it looks like:

```
it('should let you log in', function () {
    // go to the login page
    browser.url('./login');

    // enter a valid username into the "email" input
    // enter a valid password into the "password" input
    // click the 'Sign In' button
    // assert that we're logged in
});
```

Remember that we prepend our page URL with `./`, so that WebdriverIO knows to add the `baseUrl` to the start of the URL.

Step 2: Enter a Valid Username in the "email" Input

Now that we're on the login page, it's time to fill out our form. There are two fields to enter: email and password.

WebdriverIO gives us four ways to interact with the value of text fields:

- addValue
- setValue
- clearValue
- getValue

The names define what they do to the value of the input, which is the text entered into the field. You might be asking though; What's the difference between `addValue` and `setValue`? `addValue`, as you might guess, adds a text value to the input. This means that it appends your value to the end of any text that's already there. So if you have a field with a value of "do", and run the `addValue` command with text of "nut", you'd get "donut" (yum). `setValue`, on the other hand, won't do this (so you'd just get "nut"). Instead, it will set the value to the exact text you specify. It does this by combining the `clearValue` and `addValue` command, running them in that order.

Typically, I use the `setValue` command, as it's less likely to cause test issues. However, it does run an extra command, which adds just a little bit of time to your tests, so there is benefit to using `addValue` by itself.

Those are the commands, now let's learn how to use them. `clearValue` and `getValue` are called as such:

```
$('#inputId').clearValue();
const theValue = $('#inputId').getValue();
```

For `addValue` and `setValue` though, you need to pass in the text that you want to use:

```
$('#inputId').addValue('do');
$('#inputId').setValue('donut');
```

Altogether, we can modify the form input value again and again:

```
$('#inputId').addValue('do'); // value is 'do'
$('#inputId').addValue('nut'); // value is 'donut'
const theValue = $('#inputId').getValue(); // 'theValue' is 'donut'
$('#inputId').clearValue(); // value is empty
$('#inputId').addValue('do'); // value is 'do'
$('#inputId').setValue('nut'); // value is 'nut'
```

Now that we've covered these actions, let's put them to use. We're only going to be using `setValue` for our needs.

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
        browser.url('./login');

        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');
    });
});
```

For our element selectors, we're using a combination of element type (`input`) with a CSS attribute selector (`name`). For reference, here's what the HTML on the page looks like:

```
<form>
    <fieldset class="form-group">
        <input
            type="email"
            placeholder="Email"
            class="form-control form-control-lg"
            autocomplete="off"
        />
    </fieldset>
    <fieldset class="form-group">
        <input
            type="password"
            placeholder="Password"
            class="form-control form-control-lg"
            autocomplete="off"
        />
    </fieldset>
    <button class="btn btn-lg btn-primary pull-xs-right">Sign in</button>
</form>
```

We tell WebdriverIO what value we want in that input field by passing it as the lone argument in the `setValue` command. First we pass in the email address, then we send over the password.

Step 3: Clicking the 'Sign in' Button

The last step we take is to click the `Sign In` button. We need to be able to select the button to process the click on. We've done this before, by clicking on the 'Sign In' link in the site head. Let's do something similar here.

The one problem is that the text-based selector style we used before only works for links, and the type of element we want to click here is a button. Well actually, it only work *by default* for links. If you recall from the last chapter (no judgement if you don't, it was a dense one), we can pass in a custom element type to tell it to look only for the text on an element that matches the correct type.

For our needs, we're going to look for a button with the text of 'Sign In': `$('button=Sign in')`

Here's what the full file looks like:

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
        browser.url('./login');

        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');

        $('button=Sign in').click();
    });
});
```

Running Our Test

We're still missing a piece of our test (the assertion), but it's a good idea to try out what we have so far. Writing code is always prone to errors, so running your tests early is a good way to catch issues before you get too deep into an approach.

To run our test, we could use the normal `npx wdio` command, however this would run both test files. We're only interested in running the login file.

If you recall from Chapter 1.2.6, there was a special argument you could provide to the `npx wdio` command that allows you to run a specific file. That argument was called `spec`, and it allowed you to override the spec files run during your testing.

To use it, append it to your command like `npx wdio --spec=myFile`. Remember the `myFile` can be either the exact path to the file you want to test, or a filename filter. So even though we've named our file `login.js`, we can type just `npx wdio --spec=login`.

If you took the time right now to run this command, you'll see your browser pop up, open the login page, fill in the values, then submit and close the browser. Looking at the logs, you'll notice it says that our tests have passed.

Well, they certainly didn't error out, but did they really pass? The test closed before we could see the login actually complete. That's because the test ends the moment the `Sign in` button is clicked. It doesn't wait for anything else to happen, so the form still could have failed and we wouldn't know.

Let's fix that with an assertion!

Step 4: Asserting That We're Logged In

As always with testing, we can assert any number of details:

- No login error messages should show on the page
- The URL shouldn't contain 'login'
- The URL should just be 'http://localhost:8080/'
- The login form should not exist on the page
- The homepage should be showing
- The top navigation should have the following items: 'Home', 'New Article', 'Settings', and our username which links to our profile page
- The top navigation should *not* have the following items: 'Sign In', 'Sign Up'
- Check both of the two previous items

All of these conditions potentially demonstrate that we're successfully logged in. Which one should we choose, or should we choose all of them? Honestly, it's up to you and your team to decide. While additional assertions ensure you're in the valid state, they also add complexity and dependencies to the test.

What happens if the redirect after login changes to go to a different URL than the homepage? Or what if the 'settings' link is purposefully removed from the navigation? These are both alterations that will break our test, requiring upkeep to maintain that green checkmark status. A little bit of upkeep should be expected with automation, but it's smart not to tie yourself down to too many requirements that don't really add value.

For this example, we're going to check the first item in that list, that no error messages should show. If you were to inspect the HTML of the login page after trying to login with invalid credentials, you might see this:

```
<ul class="error-messages">
    <li>email or password is invalid</li>
</ul>
```

What we want to do is validate that an `<li>` inside the `error-messages` class doesn't exist (hopefully meaning no errors occurred). We can check for that with this assertion:

```
expect($('.error-messages li')).not.toBeExisting();
```

Notice the `not` in there. I mentioned it before, but to reiterate, that flips the assertion to do the opposite. In this case, it checks that the element matching the selector `.error-messages li` does not exist. If an error was existing on the page for some reason, it would fail. It's not perfect, but it's good enough for now.

Or is it? (VSauce fans should appreciate that)

Here's the full test:

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
        browser.url('./login');

        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');

        $('button=Sign in').click();

        expect($('.error-messages li')).not.toBeExisting();
    });
});
```

If you were to run this test, it would pass (which is what we want, right?)

But there's a hidden problem. What if we skip setting a password. The test should fail, right?

Nope! It passes just fine, even though we shouldn't be logged in and an error should have been shown.

Why is this?

2.3.2 Slow It Down

The reason it still passed has to do with how the site was built and the assertion we're making. Normally a form button submission triggers a standard page load, which WebDriver knows how to handle. But in this case (and this is true for many web apps out there), the form submission is handled through a special VueJS route, which doesn't go through the normal browser methods. WebDriver isn't aware of this action and thinks everything is ready to go immediately, so runs the check.

The problem is, the form is still processing the submission when the check is run, and no error messages are shown yet. So the test thinks all is fine because of this and happily passes.

How do we work around this fact? Teach WebDriver how to wait!

The simplest type of wait available, and the one I **least** recommend, is a `pause`. A `pause` is an arbitrary time delay that stops execution of your test until said amount of time passes.

In this situation, we could add a one second pause to our code between clicking the button and running our assertion. Because the server should have responded with the login result by then, our assertion should pass (or fail if there's a bug).

Let's try it out:

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
        browser.url('./login');

        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        // $('input[type="password"]').setValue('wdiodemo');

        $('button=Sign in').click();

        browser.pause(1000);

        expect($('.error-messages li')).not.toBeExisting();
    });
});
```

Note: I commented out the 'password' line to make the login fail.

The command we use is `browser.pause` and we pass in the total number of milliseconds we want to wait (1000 milliseconds = 1 second for you math nerds).

If you run the test again, you'll notice it takes a bit, but it does fail. With our pause in place, when the `expect($('.error-messages li')).not.toBeExisting();` is run, that element does exist. The `expect-webdriverio` library keeps retrying for this element to disappear, but it's not going anywhere, so the wait ends up timing out and failing the test.

That is, unless you're on a very slow connection. If your internet service isn't super speedy, you may have noticed that it still passes.

But if you have a decent provider then things seem fine. You feel no pain right now, and that's just not right. Let's resolve that.

2.3.3 Network Throttling

Yes, we're going to take a quick detour in our test writing to learn about how to slow down our internet. Why would we ever want to do that?

Well, aside from torturing unruly kids, slowing your connection down is great practice to help catch issues that users without great service providers may face. And it's not that difficult to do.

In our `wdio.conf.js` file, scroll down to the `before` hook section. It's about 150 lines down, commented out, and looks like:

```
/* Gets executed before test execution begins. At this point you can access to
 * all global variables like `browser`. It is the perfect place to define custom
 * commands.
 * @param {Array.<Object>} capabilities list of capabilities details
 * @param {Array.<String>} specs List of spec file paths that are to be run
 */
// before: function (capabilities, specs) {
// },
```

This is a "hook" that runs before our tests start running, but after the test session is created. This means we can mess with the browser at this point to make funny things happen.

The funny thing we're going to make happen is that we're going to artificially slow down our connection. This is done through the `throttle` command[103]. This is what it looks like:

```
// throttle to Regular 3G
browser.throttle('Regular 3G');

// disable network completely
browser.throttle('Offline');

// set custom network throughput
browser.throttle({
    offline: false,
    downloadThroughput: (200 * 1024) / 8,
    uploadThroughput: (200 * 1024) / 8,
    latency: 20
});
```

As you see, you can either pass in a network preset via a matching string, or a custom object defining a specific settings.

This feature uses Chrome DevTools capabilities to enable such behavior. Therefore it can only be supported where such an interface is available which is Chrome, Firefox Nightly and Chromium Edge right now.

For the custom settings, you can adjust those numbers as you like. The larger the latency, the longer the delay before requests are handled. The higher the throughput, the more data can cross the wire at any given time. Both of those combine to impact overall page load speed.

Let's slow our network down by setting the latency to `1000`. This will ensure that all requests take at least a second to process. We'll also include the other three required settings, but keep them in a state that shouldn't impact speed.

In our `before` hook (which you should uncomment if you haven't), add these custom settings:

[103] https://webdriver.io/docs/api/browser/throttle.html

```
before: function (capabilities, specs) {
    browser.throttle({
        latency: 1000,
        offline: false,
        downloadThroughput: 1000000,
        uploadThroughput: 1000000
    });
},
```

Save the file and run your test again. Suddenly the `browser.pause` statement isn't so effective. The site still takes too long to process the login request and we're stuck checking an assertion a bit too early (and have a test passing when it shouldn't).

What to do now? Well, it's unfortunately common practice to just increase that `pause` amount a little bit more, increasing the length of the test but preventing problems due to slow connections. But you're always stuck playing this game of "am I waiting long enough," usually losing.

Plus, every pause you add to your test slows it down just a little bit more. If you've got to do this for every page load, you can easily have tests that take 30 minutes to run due to all the unnecessary waiting.

What we really want to do is wait just the right amount of time and not a second more.

2.3.4 Waiting With Waits

The reason `browser.pause` is inefficient is that it ignores anything the page does during that time. Until that timer has gone off, it's not listening.

It would be much more intelligent if we kept an eye on the page and reacted to its updates. With WebdriverIO, we can do that very effectively.

WebdriverIO comes with five commands built around this idea. They are:

1. waitForExist
2. waitForEnabled
3. waitForDisplayed
4. waitForClickable
5. waitUntil

The `waitFor` commands do as they say, and wait for an element to:

1. Exist on the page
2. Be enabled (for inputs that have a `disabled` HTML attribute)

3. Be "displayed", which is a property that we'll put a definition to in a bit
4. Be "clickable" (element exists, is visible, is within viewport (if not try to scroll to it), its center is not overlapped with another element, is not disabled)

Sidenote: These commands are actually used under the hood by the `expect-webdriverio` library to help with our assertions.

The last command, `waitUntil`, is a generic function used to define custom waits. For example, a common scenario is to wait until an element has specific text before moving on. We'll get into real code examples in a little bit, but for now we're going to jump back to the `waitForExist` command.

As mentioned, the `waitForExist` command will run until one of two things happens:

- The element being searched for is found in the DOM
- The `waitForTimeout` timeout amount is met (this amount is defined in your `wdio.conf.js` file)

If the latter happens, an error will be thrown and your test will fail, as this likely means something is wrong with the website. Hopefully though, the element is found and your test will continue.

So, what are we going to wait for? Well, it can be any number of things unique to the "logged in" experience. There are a couple items you only see if you're logged in to the website:

- The 'Settings' and profile links in the main navigation
- The 'Your Feed' tab in the main content area

Let's go with that 'Settings' link. Using a text-based selector, we can wait for the link to exist on the page:

```
$('=Settings').waitForExist();
```

This will tell WebdriverIO to wait until the following HTML element is on the page:

```
<a>Settings</a>
```

The `<a>` tag can contain any number of attributes; that doesn't matter. But it must be an `<a>` tag and it must have `Settings` as its textual content (per how the link text selector works).

Throwing that code into our test, this is how the whole thing looks:

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
        browser.url('./login');

        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');

        $('button*=Sign in').click();

        $('=Settings').waitForExist();

        // Get the URL of the page, which should no longer include 'login'
        expect(browser.getUrl()).not.toContain('/login');
    });
});
```

Now the test passes if we do login successfully and fails if we don't.

Oh, it's a good idea to go back to your `wdio.conf.js` file and comment out that `browser.throttle` line. There's no reason to keep our tests running slow anymore.

2.3.5 Waiting With Inverse Waits

Let's look at an alternative approach we can take, just to learn a little bit more about these `wait` commands.

Instead of waiting for an element to appear, we can conversely wait for one to disappear. This is done through the same command by passing in a special option.

Instead of waiting for the 'Settings' link to appear, instead we'll wait for the "Sign In" button to stop existing, (it disappears from the DOM as soon as we navigate away from the login page). The command almost looks the same as before, but now we're passing in an options object with a property of `reverse` set to true:

```
$('button*=Sign in').waitForExist({ reverse: true });
```

`waitForExist` (and the other `waitFor*` commands), takes a single argument, which is an object of several possible options:

1. An override to the default `timeout` amount. Since we don't want to override it, we can just leave it out and WebdriverIO will stick with the default.

2. A boolean flag that will reverse how the `waitFor` command works. In this case, if that flag is `true`, it instead waits for the selector to **not** match any elements (the default value is `false`). We want this, so we add it as an option
3. A `timeoutMsg` to show if the `timeout` is met, which whill be logged out upon that error showing.
4. An `interval` to wait between checking for the element to exist. The `waitForExist` command run the `isExisting` command multiple times if necessary, and this `interval` defines how long to wait until running that `isExisting` command again if the previous one failed.

Here it is in the full test:

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
        browser.url('./login');

        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');

        const $signIn = $('button*=Sign in');
        $signIn.click();
        $signIn.waitForExist({ reverse: true });

        expect($('.error-messages li')).not.toBeExisting();
    });
});
```

One thing to notice: I stored the `Sign in` button element for better re-use. Then I use that refence in the next two steps. By doing this, we only have to define our selector once, making updates to that selector changes much easier (this will be a common theme as we get into Page Objects).

Also note the `$` prefix of our variable name. That's not required in any way, but I find it helpful to differentiate between WebdriverIO element references and other variables that might be stored. For example, if you were to see `$message` and `message`, it's not a stretch to guess that `$message` is a WebdriverIO element reference, while the other has to do with actual text.

Well, it took a little bit of trial and error, but eventually we achieved a good solution! You should get to use this if you're going to continue down the path of testing. Nothing is as simple as it seems.

We still have two more tests that we want to write for our login page, but we're going to save that for the next chapter.

2.3.6 Chapter Challenge

- Update the 'waitFor' code to instead use `waitForDisplayed`

- Instead of waiting for the “Sign in” button to stop existing, try waiting for the “Your Feed” tab to appear.

2.4 Custom Functions, Page Objects, and Actions

2.4.1 You've Got Me Hooked

If you recall from the previous chapter, at the beginning we had set out to write three tests. They were:

```
// Login Page:
// should let you log in
// should error with a missing username
// should error with a missing password
```

We completed the first, but haven't touched the other two. Let's get to that now.

Both tests are very similar. In fact, they're almost like the first test we wrote, except we will be omitting certain steps and validating the page didn't change.

Using the "test comments" practice we used in our first test, here's what our two tests will do:

```
// - go to the login page
// - enter either a valid username or a valid password into the corresponding input
// - click the 'Sign In' button
// - assert that correct error message is shown and we aren't logged in
```

If you remember, a couple of those steps are the same as what we've already coded. And if you're lazy like me, you don't want to code anymore than you already have to.

Well, let's avoid that extra work by moving our 'go to login page' step to a common location.

MochaJS provides a feature called "hooks" that allow you to run custom test code around your tests. Each `describe` block can have any or all of the following hooks implemented:

- before
- beforeEach
- afterEach
- after

`before` and `after` are run before/after the test suite. So if you have five tests, `before` would run before the set of tests, and `after` after the tests.

`beforeEach` and `afterEach`, however, execute around each test. Here's how the execution flow runs for a set of tests:

- before
- beforeEach
- test 1
- afterEach
- beforeEach
- test2
- afterEach
- after

As you can see, `beforeEach` and `afterEach` run multiple times based on the number of tests written. `before` and `after` only run at the start/end of the test suite.

Okay, enough explanation, let's see this in action. In our first test, we load the login page by running `browser.url('./login');`. This is the code we're going to move to our hook.

Should we use a `before` or `beforeEach` hook though? That's a good question to ask, because there are benefits/drawbacks to each. The benefit to using a `beforeEach` hook is that we can be sure our page is "fresh" for our test. Changes made in a previous test won't impact this one (mostly... sometimes this isn't true).

On the other hand, because `beforeEach` runs for every test, every step you take in it requires one more command to be run, extending the test execution time.

For us, the decision here is pretty easy. We're going to go with a `beforeEach` hook because we definitely need a fresh page for each of our tests (we don't want an error from our `missing username` test to contaminate our `missing password` test.

With that, let's move our step over. This is what the code looks like:

test/specs/login.js

```
describe('Login Page', function () {
    beforeEach(function () {
        browser.url('./login');
    });

    it('should let you log in', function () {
        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');

        const $signIn = $('button*=Sign in');
```

```
        $signIn.click();

        $signIn.waitForExist({ reverse: true });

        expect($('.error-messages li')).not.toBeExisting();
    });
});
```

Not much has changed, but we've set ourselves up to do a little less work in the long run.

2.4.2 Getting the Error Text

As mentioned, there isn't much different between these two new tests we're writing and our previous test. That means I won't be covering what we've already talked about. So, on that note, here's what our two new tests are almost going to look like:

```
it('should error with a missing username', function () {
    $('input[type="password"]').setValue('wdiodemo');

    $('button*=Sign in').click();

    // assert that error message is showing
});

it('should error with a missing password', function () {
    $('input[type="email"]').setValue('demo@learnwebdriverio.com');

    $('button*=Sign in').click();

    // assert that error message is showing
});
```

In our tests, we enter either the username or the password, but not both (to trigger showing an error message).

As you can see, the only work left to do is assert that the error message is showing. How do we go about this?

We have a couple of decent options:

- Re-use the error message selector we used in our first test (`$('.error-messages li')`), then validate using the `toHaveText` assertion to check our message.

- Use a 'text-based selector' to check for the specific error message (e.g., `$("li*=email can't be blank")`)

Using the second method, we'd combine our text-based selector with a `toBeExisting` assertion that runs after our test actions are executed. Here's what one test would look like:

```
it('should error with a missing username', function () {
    $('input[type="password"]').setValue('wdiodemo');

    $('button*=Sign in').click();

    expect($("li*=email can't be blank")).toBeExisting();
});
```

Wait, shouldn't we still have the `waitForExist` command that waits for the 'Sign in' button to disappear? No, because that button stays on the page when the login fails to complete.

One drawback to this approach is, what if the error message does show, but the error text has slightly changed? Our test will fail and we'll be left thinking the login is completely broken (when really, it still validates correctly, it's just a different message).

Alternatively, using the first approach as mentioned, we can assert the text inside of the error message. While it will still throw an error if the message changes, we'll have a much better clue as to what's going on.

We'll do this by using `toHaveText`, which uses the `getText` command behind the scenes to check the text of the element with the expected page.

The next two snippets of code are essentially the same:

```
expect($('.error-messages li')).toHaveText(`email can't be blank`);
// note the use of backticks due to the apostrophe in "can't"


const errorText = $('.error-messages li').getText();
expect(errorText).toEqual(`email can't be blank`);
```

As you can probably guess, `getText` gets the visible text of an element. Notice that I mentioned "visible." If the element is hidden, the `getText` command will return an empty string. A great example of this is an item in a dropdown menu. Since that item isn't visible until you open up the dropdown menu, `getText` will return an empty string. If you were to open up the dropdown menu, then run `getText`, it would successfully return the text value.

There is a workaround to this issue: The `getHTML` command. Just like `getText`, `getHTML` returns the HTML value of an element, including the text inside it. Do note that as of this writing, there is no `toHaveHTML` assertion to go alongside it.

Let's look again at the HTML for our error message:

```
<ul class="error-messages">
  <li>email can't be blank</li>
</ul>
```

Assuming that all elements are visible, here's what our `getText` and `getHTML` commands would return:

```
$('.error-messages li').getText();
// returns "email can't be blank"

$('.error-messages li').getHTML();
// returns "<li>email can't be blank</li>"

$('.error-messages').getText();
// returns "email can't be blank"

$('.error-messages').getHTML();
// returns "<ul class="error-messages"><li>email can't be blank</li></ul>"
```

First, notice how getting the text of the parent element returned the same value as getting the text of the individual element itself. That's by design, and you can use that knowledge to your advantage, so keep it in mind.

Second, the `getHTML` included the `<li>` tags in the response. It would be a more valid test if we ignored the HTML of the page and only returned the text content. We can do that!

`getHTML` accepts a single parameter, which is a boolean flag to include the selector element tag or not. By default it's `true`, and therefore includes that tag. If we pass in `false`, we get different results:

```
$('.error-messages li').getHTML(false);
// returns "email can't be blank"

$('.error-messages').getHTML(false);
// returns "<li>email can't be blank</li>"
```

In the first command, now we get just the text without HTML. However, because that boolean flag only works on the main element, any child tags will still be included in the results, regardless of what we pass in. This means that `getHTML` still has its limitations, but I still like keeping the command in my back pocket for times when I need to get the text of a hidden element.

Since our error message shouldn't be hidden, let's stick with the `getText` route. As mentioned before, we'll use the `toHaveText` assertion to achieve this. As an added benefit, because `toHaveText` automatically retries if it fails, we don't need to add any `waitFor` commands since it's essentially built-in to it:

```
it('should error with a missing username', function () {
    $('input[type="password"]').setValue('wdiodemo');

    $('button*=Sign in').click();

    // assert that error message is showing
    expect($('.error-messages li')).toHaveText(`email can't be blank`);
});
```

Great, we now have a fully-built test ready for running.

2.4.3 Only One More Thing I Don't Want to Skip

Before running our test though, I want to talk about a quick tip with Mocha. There are two special abilities to it that make it easier to work on specific tests.

Say we're working on a test, but it's failing and we don't really have time to fix it right now. Instead of deleting or commenting out the entire test, you can add `.skip` to either the `describe` or `it` function call, and it will skip the test. Here are some examples:

```
describe('All tests', function () {
    it('will run', function () {});

    it.skip('will be skipped', function () {});

    describe.skip('Skipped Suite', function () {
        it('will be skipped', function () {});
        it('will also be skipped', function () {});
    });
});
```

Alternatively, say you only want to run a single test (or single set of tests). You can append `.only` to the `it` or `describe` function call to have only that test (and any other tests with `only` marked) run:

```
describe('All tests', function () {
    it('will be skipped', function () {});

    it.only('will run', function () {});

    describe.only('Only Suite', function () {
        it('will run', function () {});
        it('will also run', function () {});
    });
});
```

Notice how we can have multiple `only` tests and all of them marked as such will run? That's intentional, and you can use it to run only a few tests, instead of having to go in and run each test individually, moving the `.only` between them (or add `.skip` to all the tests you don't want running).

Okay, so we can use this new knowledge to run just the single new test that we've written. Here's what our updated file will look like:

test/specs/login.js

```
describe('Login Page', function () {
    beforeEach(function () {
        browser.url('./login');
    });

    it('should let you log in', function () {
        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');

        const $signIn = $('button*=Sign in');
        $signIn.click();

        $signIn.waitForExist({ reverse: true });

        expect($('.error-messages li')).not.toBeExisting();
    });

    it.only('should error with a missing username', function () {
        $('input[type="password"]').setValue('wdiodemo');

        $('button*=Sign in').click();

        // assert that error message is showing
```

```
        expect($('.error-messages li')).toHaveText(`email can't be blank`);
    });

    it('should error with a missing password', function () {
        $('input[type="email"]').setValue('demo@learnwebdriverio.com');

        $('button*=Sign in').click();

        // assert that error message is showing
    });
});
```

After running our `npx wdio --spec=login.js` command from the terminal again, you should see the green checkmark next to your passing test. It's always a good idea to make sure your assertion actually works, so try changing the expected text or commenting out the `click` command to validate the test fails if that error message doesn't show.

Since the missing password test is so similar to the missing email test, I'm going to skip over explaining it and just post the test:

```
it('should error with a missing password', function () {
    $('input[type="email"]').setValue('demo@learnwebdriverio.com');

    $('button*=Sign in').click();

    // assert that error message is showing
    expect($('.error-messages li')).toHaveText(`password can't be blank`);
});
```

2.4.4 Custom Test Functions

While we did some work to reduce repetition in our code so far by using the `beforeEach` hook, there's still more we can do. If you look at the first couple steps in all three tests, we do two things:

1. Fill out the form fields
2. Click the "Sign In" button

Shouldn't we move these to the `beforeEach` function? Well no, because the specifics in each individual test vary just enough for them to not fit a common `beforeEach` function.

Rather, we're going to create a custom function that will handle the unique parts of our login steps in a common way. This function will take an email and password value, then click the 'Sign in' button. We'll call this function from inside our tests.

Here's what the function is going to look like:

```
function login(email, password) {
    $('input[type="email"]').setValue(email);
    $('input[type="password"]').setValue(password);

    $('button*=Sign in').click();
}
```

We're using the exact code in our first test, so let's look at how that will look replaced:

```
it('should let you log in', function () {
    login('demo@learnwebdriverio.com', 'wdiodemo');

    $('button*=Sign in').waitForExist({ reverse: true });

    expect($('.error-messages li')).not.toBeExisting();
});
```

In place of those commands, we call the `login` function, passing in our username and password values. This simplifies our test a bit, while keeping all the same functionality (although we do now have duplicate selectors for our 'Sign in' button, but we'll fix that soon).

So what do we do for the other tests where we don't have a value for one of our fields? Well, we just pass in an empty string:

```
it('should error with a missing username', function () {
    login('', 'wdiodemo');

    // assert that error message is showing
    expect($('.error-messages li')).toHaveText(`email can't be blank`);
});
```

WebdriverIO will take that empty string and set the value of the corresponding field to it. It's technically an extra command being run, but it helps reduce repetition in our codebase so it's worth it.

Altogether, here's what the updates look like:

test/specs/login.js

```
function login(email, password) {
    $('input[type="email"]').setValue(email);
    $('input[type="password"]').setValue(password);

    $('button*=Sign in').click();
}

describe('Login Page', function () {
    beforeEach(function () {
        browser.url('./login');
    });

    it('should let you log in', function () {
        login('demo@learnwebdriverio.com', 'wdiodemo');

        $('button*=Sign in').waitForExist({ reverse: true });

        expect($('.error-messages li')).not.toBeExisting();
    });

    it('should error with a missing username', function () {
        login('', 'wdiodemo');

        // assert that error message is showing
        expect($('.error-messages li')).toHaveText(`email can't be blank`);
    });

    it('should error with a missing password', function () {
        login('demo@learnwebdriverio.com', '');

        // assert that error message is showing
        expect($('.error-messages li')).toHaveText(`password can't be blank`);
    });
});
```

So with this custom function, we've moved browser commands into a common area, reducing the redundancy of our code, which is good. That said, it's also made the codebase more complex. Instead of being able to find everything you need for a test within it, we now have to track down this `login` function and figure out what it does.

As with all things programming, this is a tradeoff we make. Every step to reduce the verbosity of our tests increases the complexity of it. Be mindful of this as we move into the next section. Perhaps,

for the sake of simplicity in a basic test suite, the following knowledge may be overdoing it in some circumstances.

2.4.5 Basics of Page Objects

The function we built was pretty neat, and opens up the idea of moving implementation specifics outside the tests and into common areas more available for reuse. Page Objects build on that idea by providing a nomenclature for defining a page and the actions you can take on it.

We defined a "login" function, but what if we could define the "login" page itself. The specific elements on that page, and the various actions you could take upon it. That's where Page Objects slide in.

Despite the "login" function reducing some code duplication, we still had to restate the `Sign in` button selector in our first test and the 'Error Message' selector in all three. Wouldn't it be nice if we could create a common area to store all of our element selectors?

Well, a basic system would be to define each selector as a constant, then reference that constant in our tests:

```
const emailSelector = 'input[type="email"]';
const passwordSelector = 'input[type="password"]';
const signInSelector = 'button*=Sign in';

function login(email, password) {
    $(emailSelector).setValue(email);
    $(passwordSelector).setValue(password);

    $(signInSelector).click();
}
```

This is an improvement, but it's only okay as far as code style goes. We still have to call the `$()` function a bunch, which makes the code a little bit more clunky. How about we switch those selectors to be element references?

```
const $email = $('input[type="email"]');
const $password = $('input[type="password"]');
const $signIn = $('button*=Sign in');

function login(email, password) {
    $email.setValue(email);
    $password.setValue(password);

    $signIn.click();
}
```

Isn't that better? No! It's much worse! If you tried running that code, you'd get several failures.

The reason is that we're trying to run the `$()` function outside its normal living space (this has to do with the `wdio-sync` library, but I won't go into those details). Because the constants are defined outside the WebdriverIO context (which only exists within the specific Mocha functions), it errors out with a message of `$email.setValue is not a function`.

So what we really need is "just-in-time" references. We want to store the code needed to get the element reference, but only run that code when we're inside our tests. What if we write some cute little functions that do that?

```
const $email = function () {
    return $('input[type="email"]');
};
const $password = function () {
    return $('input[type="password"]');
};
const $signIn = function () {
    return $('button*=Sign in');
};

function login(email, password) {
    $email().setValue(email);
    $password().setValue(password);

    $signIn().click();
}
```

Okay, so that code works, but it sure is ugly. Having to call our element references as a function every time we want to access them? Certainly there's a cleaner way of achieving the same effect…

Surprise, there is! Okay… maybe it's not a surprise. I worked too hard to set myself up for all of this.

Regardless, newer versions of JavaScript let us achieve this with a much cleaner syntax. Using the `class` syntax (introduced in ECMAScript 2015 for those keeping track), we can define `get` functions that look like standard variables, but act like functions.

First, we need to define a new `class` to hold all of our authentication related information:

```
class Auth {
}
```

Then, we define our `get` functions inside it:

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }
}
```

The `get` functions mirror our earlier element reference functions, so it should look familiar. But we don't access them by calling `Auth.$email()`. There is an extra step we need to take to make it all work.

When you define a `class`, you're stating how something "should" work. It's not the same as defining a variable or constant, which are able to be referenced right away. No, with a class, you first have to "instantiate" it.

Instantiate is a big fancy term meaning "the creation of a real instance or particular realization of an abstraction or template such as a class."

Not any clearer? Well, it basically means we're creating a new instance of something. In this case, it's the class we've just defined. We have to make an instance of it before putting it to work.

So we need to "instantiate" our class before we can use it. That's pretty easy:

```
const auth = new Auth();
```

We call `new` on our Auth class like it's a function, and store it as an `auth` constant. Now we have a real reference that we can use in our test.

Collectively, this is how our code now looks using classes:

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }
}

const auth = new Auth();

function login(email, password) {
    auth.$email.setValue(email);
    auth.$password.setValue(password);

    auth.$signIn.click();
}
```

We got rid of the function call for each of our element references, and prepended everything with our `auth` instantiated class. If you're counting text characters, we've actually gone up, but it's a lot cleaner and more self-documenting. All the elements related to authentication are stored inside the Auth instance.

One added benefit of this approach is that we can use this `auth` instance throughout our tests. Here's what the fully updated test suite looks like:

test/specs/login.js

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }
}

const auth = new Auth();

function login(email, password) {
    auth.$email.setValue(email);
    auth.$password.setValue(password);

    auth.$signIn.click();
}

describe('Login Page', function () {
    beforeEach(function () {
```

```
        browser.url('./login');
    });

    it('should let you log in', function () {
        login('demo@learnwebdriverio.com', 'wdiodemo');

        auth.$signIn.waitForExist({ reverse: true });

        expect(auth.$errorMessages).not.toBeExisting();
    });

    it('should error with a missing username', function () {
        login('', 'wdiodemo');

        // assert that error message is showing
        expect(auth.$errorMessages).toHaveText(`email can't be blank`);
    });

    it('should error with a missing password', function () {
        login('demo@learnwebdriverio.com', '');

        // assert that error message is showing
        expect(auth.$errorMessages).toHaveText(`password can't be blank`);
    });
});
```

Notice how the tests themselves no longer store the selector information. All of those details are inside our page object. This separation of concerns[104] helps reduce maintenance efforts (fewer places to change things) and improves readability. Instead of our test code being cluttered with selectors, they have more understandable references.

2.4.6 Naming Patterns

Speaking of which, how should you name your variables? Overall, that's an individual/team choice, but there are a few things you'll want to consider.

One option is to include the element type in the variable name (e.g., `$emailInput`, `$passwordInput`). While it's nice to know we're working with an input element, there are couple drawbacks to this approach:

[104] https://en.wikipedia.org/wiki/Separation_of_concerns

1. It's more verbose. You can remediate this by using shorter names: `inp` vs `input` or `lbl` instead of `label`. But that can be confusing to folks unfamiliar with the nomenclature.
2. Not all elements line up nicely with an element name. For example, container elements. Are you going to name it `accordionContainer` or `accordionDiv`? Are you going to name a list as `todoList` or `todoUl`?

I'm not saying including element information is a bad idea, only that you need to be consistent with your naming pattern.

Another consideration is how you'll name types of components. For example, if you're testing a to-do list, you might have the following HTML:

```
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
</ul>
```

Here's what the page object may look like:

```
class TodoList {
    get $container () { return $('ul'); }
    get $$items () { return $$('li'); }
}
```

But it could also be:

```
class TodoGroup {
    get $parent () { return $('ul'); }
    get $$tasks () { return $$('li'); }
}
```

Both are valid and work well, but the naming scheme is definitely different between the two. Inconsistent naming patterns can cause confusion for newcomers to the codebase, so be clear with the pattern you define from the start.

2.4.7 Page Actions

We were able to move our selectors out of our tests and into our page object; can we do the same with the login function? It's technically part of our "Auth" flow, and it would be great to group it inside of our Auth page object for reuse across our tests.

Yes, we certainly can move it, and that's just what we're going to do next.

JavaScript classes, which are what we've been using so far to define our page object, allow you to define functions on them (as well as the 'getters' we've already used for our element references). You add these functions in a similar format to normal functions that live in the wild.

Instead of declaring our function on its own, we include it inside the curly braces that define our class. Let's update our class to declare a "login" function:

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    login (email, password) {
        // login actions go here
    }
}

const auth = new Auth();
```

I didn't include the steps in the function yet, as there's a slight tweak we need to make to the login code. Recall how the login function referenced the `auth` instance we created. For example: `auth.$signIn.click()`

If we try a straight copy/paste of that, the code will run successfully, but a bug will be lurking beneath the surface.

When our function commands reference `auth`, they're talking about the specific `auth` instance that's later created in our code. Our page object has no real control over what this instance is named, so any changes to that name will break the code.

For instance, if the instantiation is changed to `const authentication = new Auth();`, our login function will error out with `auth is not defined`, as `auth` no longer exists. This means our function is brittle and likely to break with outside code changes. It's a bad situation to put yourself in.

How do we work around this issue? Well, conveniently, JavaScript classes give us a pre-defined `this` keyword that we can use to reference any instance of our class. We don't need to declare it anywhere, it exists as part of the normal runtime. Similar to how we don't define a `browser`, it already exists when we run our code.

JavaScript `this` can be a bit confusing, and we'll get into scenarios in the future where that's true. In our code right now though, just imagine that `this` is a reference to the overall class. Let's add the commands into our function, replacing `auth` with `this` as necessary:

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    login (email, password) {
        this.$email.setValue(email);
        this.$password.setValue(password);

        this.$signIn.click();
    }
}
```

Now the `const auth = new Auth();` instantiation can be named anything and our code will still work. By using the magic of `this`, we alleviate the need to know what our instance will be named.

There's one last item to complete in our transition to page actions: we need to update our `login` reference in our test code. Since we want to call the `login` function that belongs to our `auth` instance, we can refer to it in the same way we've accessed our element reference: by prepending the instance name (i.e., `auth`) to our function name.

So:

```
login('demowdio@example.com', 'wdiodemo');
```

becomes:

```
auth.login('demowdio@example.com', 'wdiodemo');
```

Now when we run the code, it will look for the `login` function on the `auth` page object instance. Here's what the entire file now looks like:

test/specs/login.js

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    login (email, password) {
        this.$email.setValue(email);
        this.$password.setValue(password);
```

```
        this.$signIn.click();
    }
}

const auth = new Auth();

describe('Login Page', function () {
    beforeEach(function () {
        browser.url('./login');
    });

    it('should let you log in', function () {
        auth.login('demo@learnwebdriverio.com', 'wdiodemo');

        auth.$signIn.waitForExist({ reverse: true });

        expect(auth.$errorMessages).not.toBeExisting();
    });

    it('should error with a missing username', function () {
        auth.login('', 'wdiodemo');

        // assert that error message is showing
        expect(auth.$errorMessages).toHaveText(`email can't be blank`);
    });

    it('should error with a missing password', function () {
        auth.login('demo@learnwebdriverio.com', '');

        // assert that error message is showing
        expect(auth.$errorMessages).toHaveText(`password can't be blank`);
    });
});
```

We've now fully built-out our page object with element references and custom functions. This framework will be used throughout our tests moving forward, so you'll be prepared to see a lot of this. It might feel a bit over-complicated right now, but it opens up opportunities for extension in the future that we'll be covering in the next chapter.

2.4.8 Improving Our Login Detection

We've gotten rid of the code duplication between our tests, but there's one final step we can take to reduce future duplication. When we login, we have a `waitForExist` command that pauses execution until the page has updated (i.e., `auth.$signIn.waitForExist({ reverse: true });`). Right now we're only using it once, but we'll be using this login function a lot in the future when we tests pages that require user login.

But we can't just move that `waitForExist` command into our login function, because then our other two tests will fail as the `Sign in` button never disappears in those flows. How do we get around this?

I mentioned the `waitUntil` function a while back; now it's time to take advantage of it.

As I mentioned, in our three tests, there are two different conditions we wait for:

Either:

1. For the login button to stop existing: `auth.$signIn.waitForExist({ reverse: true });`
2. For an error message to exist: `auth.$errorMessages.waitForExist();`

On the outside, it doesn't make sense to combine these, as one is for a successful login and the other is for a failed login. But what if we could say, "Wait for either this *or* that?"

This is where a custom `waitUntil` command comes in. It's your universal weapon if you want to wait on some specific scenario. It expects a condition and waits until that condition is fulfilled with a truthy value.

With it, we can say, "Wait for the login button to disappear or for an error message to exist." To use the `waitUntil` method, call it and pass in a function that returns a true or false value, depending if the condition is met. WebdriverIO will keep waiting/calling that function until it either returns true, or reaches the timeout limit.

For our login method, we can write our function as follows:

```
browser.waitUntil(function () {
    const signInExists = auth.$signIn.isExisting();
    const errorExists = auth.$errorMessages.isExisting();

    return !signInExists || errorExists;
});
```

Our return statement will return true for either of the following conditions (`||` means 'or', if you're not familiar with it):

- The 'Sign in' button no longer exists (`!` reverses the value of `signInExists`)

- An error exists

Because `waitUntil` automatically retries, our function will continue to return false until either of those conditions is met (checking every half a second by default). If that condition isn't met by our `waitForTimeout` amount, it will fail saying `waitUntil condition timed out after 10000ms`.

Well, that message isn't very helpful. We know that `waitUntil` failed, but unless you were expecting the failure (which you likely weren't), you're going to have to dig in the code to figure out what condition it was.

We can help that by passing in a custom error message. The `waitUntil` function signature looks like:

```
browser.waitUntil(condition, { timeout, timeoutMsg, interval });
```

The `condition` is our function, which is required. After that is an options object. This object is almost the same as what we have for `waitForExist` (`reverse` isn't included, as it's not needed).

The `timeoutMsg` option is the custom error message we want to show in case of failure. We can leave out the rest so they'll go with the defaults.

Here's what this looks like:

```
browser.waitUntil(
    function () {
        const signInExists = auth.$signIn.isExisting();
        const errorExists = auth.$errorMessages.isExisting();

        return !signInExists || errorExists;
    },
    { timoutMsg: 'The "Sign in" button still exists and an error never appeared' }
);
```

Perfect! Now we can remove that `waitForExist` statement in our test code and add our new `waitUntil` function to our `login` function:

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    login (email, password) {
        this.$email.setValue(email);
        this.$password.setValue(password);
```

```
        this.$signIn.click();

        // wait until either the sign in button is gone or an error appears
        browser.waitUntil(
            () => {
                const signInExists = this.$signIn.isExisting();
                const errorExists = this.$errorMessages.isExisting();

                return !signInExists || errorExists;
            },
            {
                timoutMsg:
                    'The "Sign in" button still exists and an error never appeared'
            }
        );
    }
}
```

Two important things to point out that I changed here.

1. I updated `auth.` to `this.` So `auth.$signIn.isExisting()` is now `this.$signIn.isExisting()`. That's inline with the changes we made earlier.
2. I used a `fat arrow` function definition, instead of the previous `function () {}`. That's because our `waitUntil` function will be called with a different `this` value if we don't use it.

Remember how I said `this` can be tricky? Well, here's your example. If we used the following code...

```
browser.waitUntil(
    function () {
        const signInExists = this.$signIn.isExisting();
        const errorExists = this.$errorMessages.isExisting();

        return !signInExists || errorExists;
    },
    { timoutMsg: 'The "Sign in" button still exists and an error never appeared' }
);
```

...the `this` in our code wouldn't refer to our page object, but rather the WebdriverIO `browser` instance (so it would be the same as calling `browser.$signIn.isExisting()`, which doesn't work). By using a fat arrow function, we bind `this` to the page object, allowing us to say `this.$signIn.isExisting();`.

For more details on fat arrow functions, have a read through this in-depth article by Mozilla[105].

Here's what our fully updated `login.js` file looks like:

test/specs/login.js

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    login(email, password) {
        this.$email.setValue(email);
        this.$password.setValue(password);

        this.$signIn.click();

        // wait until either the sign in button is gone or an error appears
        browser.waitUntil(
            () => {
                const signInExists = this.$signIn.isExisting();
                const errorExists = this.$errorMessages.isExisting();

                return !signInExists || errorExists;
            },
            {
                timoutMsg:
                    'The "Sign in" button still exists and an error never appeared'
            }
        );
    }
}

const auth = new Auth();

describe('Login Page', function () {
    beforeEach(function () {
        browser.url('./login');
    });

    it('should let you log in', function () {
        auth.login('demo@learnwebdriverio.com', 'wdiodemo');
```

[105] https://hacks.mozilla.org/2015/06/es6-in-depth-arrow-functions/

```
        expect(auth.$errorMessages).not.toBeExisting();
    });

    it('should error with a missing username', function () {
        auth.login('', 'wdiodemo');

        // assert that error message is showing
        expect(auth.$errorMessages).toHaveText(`email can't be blank`);
    });

    it('should error with a missing password', function () {
        auth.login('demo@learnwebdriverio.com', '');

        // assert that error message is showing
        expect(auth.$errorMessages).toHaveText(`password can't be blank`);
    });
});
```

2.4.9: Separating Files

We've finished with our Page Object/Actions, but there's still one last thing to do. As we move forward with new tests, it will be really helpful to have this Auth page object available elsewhere. It's a very good idea to separate out your page objects from your test files, so let's do that.

Grab all of your `class` code and move it to a new file that looks like:

test/pageObjects/Auth.page.js

```
class Auth {
    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    login(email, password) {
        this.$email.setValue(email);
        this.$password.setValue(password);

        this.$signIn.click();

        // wait until either the sign in button is gone or an error appears
```

```
        browser.waitUntil(
            () => {
                const signInExists = this.$signIn.isExisting();
                const errorExists = this.$errorMessages.isExisting();

                return !signInExists || errorExists;
            },
            {
                timoutMsg:
                    'The "Sign in" button still exists and an error never appeared'
            }
        );
    }
}

module.exports = Auth;
```

In addition to the moved code, at the very bottom, we added an 'export' for our class via Node's export system. We did this by attaching our class to the `module.export` value (Node.js technical docs on `module.export`[106]). Without this, our class wouldn't get exported from the file, which would be bad.

Now we need to save our file. First, we need a new folder to save our page object in. If we save it to our specs folder, WebdriverIO will think it's a test spec file and try running it as so. Instead, let's save it outside the `test/specs` folder, to a specific page objects folder at `test/pageObjects`.

For the filename, I like to use a leading capital letter, signifying that we've exported a single class. I also like to append `.page` in my filename, to let others know that this is a page object definition. So my filename will be saved to `test/pageObjects/Auth.page.js`.

Finally, back in our test file, we need to import our page object for usage. This is what the top of the file will now look like:

```
const Auth = require('../pageObjects/Auth.page');

const auth = new Auth();
```

With everything moved around and hooked up, you can run your tests one more time to ensure it's all plugged in correctly.

In the next chapter, we're going to use our new Auth page object to help ourselves get logged in, so we can test all sorts of authenticated functionality.

[106] https://nodejs.org/api/modules.html#modules_exports_shortcut

2.4.10 Chapter Challenge

- Write a set of tests for the register account page that checks the following requirements:
  - Requires username, email, and password
  - errors if username is already taken
  - errors if email is not a valid format (e.g., missing `@`)
  - errors if email has already been taken
  - Takes you to the home page once you register

2.5 Sharing Common Page Object Functionality

2.5.1 Testing the Post Editor

Now that we've got our login tests running, the next bit of functionality we want to write tests for is our Post Editor page.

This page is crucial to the user interaction on the site, as it's the interface between their ideas and their posts. As a user, nothing is more frustrating than wanting to share content but having an error that prevents you from doing so.

Over the next couple chapters, we'll be working on writing tests for this page. We're going to check that the following requirements are met:

- it should load the page properly
- it should let you publish a new post
- it should alert you if you try navigating away from the page without saving your post

In this chapter, we're going to validate the page load. This will include the following steps:

1. Load the login page
2. Login with a valid user
3. Load the Post Editor page
4. Assert the URL is correct
5. Assert the page fields are correct

While we've done similar work in the past with our Login tests, we're going to advance our abilities by investigating how to share common functionality across our tests.

To get things started, let's lay out the basic structure for our new editor test. Create a new file in your `specs` folder named `editor.js`. In it, we'll insert our `require` statements and `describe/it` blocks:

test/specs/editor.js

```
const Auth = require('../pageObjects/Auth.page');

const auth = new Auth();

describe('Post Editor', function () {
    it('should load page properly', function () {
        // Load the login page
        // Login with a valid user
        // Load the Post Editor page
        // Assert the URL is correct
        // Assert the page fields are correct
    });
});
```

This structure is very similar to what we've seen with our Login page, which means that it's a good thing we've made our Auth page object available for reuse. Let's take advantage of it by adding the login code to our test:

```
it('should load page properly', function () {
    // Load the login page
    browser.url('./login');
    // Login with a valid user
    auth.login('demo@learnwebdriverio.com', 'wdiodemo');
    // Load the Post Editor page
    // Assert the URL is correct
    // Assert the page fields are correct
});
```

Thinking ahead, we'll want to be logged in for all our tests in this file. This is similar to how, in our login tests, we re-loaded the login page for a test using Mocha's `beforeEach` hook.

But we don't want to waste time logging in for each individual test. It would work to login at the beginning of our test suite, then re-use that session for each test. Therefore, we're going to use a `before` hook instead, which will run the login code once at the beginning. Since WebdriverIO uses the same browser session for the entire file, we don't need to login again (when we load the editor page, it will use the authentication information stored in the browser's session).

Using the same code as our login page, we can add in a `before` hook that will look like:

```
describe('Post Editor', function () {
    before(function () {
        // Load the login page
        browser.url('./login');
        // Login with a valid user
        auth.login('demo@learnwebdriverio.com', 'wdiodemo');
    });
    it('should load page properly', function () {
        // Load the Post Editor page
        // Assert the URL is correct
        // Assert the page fields are correct
    });
});
```

The next step in our test is to load the 'Editor' page. We'll do this with a simple `browser.url('./editor')` command. But where should we put this? As mentioned in the previous chapter, including the `url` command in our `beforeEach` hook is my preferred approach, as it ensures we have a clean page between all our test runs. If we added to the `before` hook, changes we make in one test might impact the state of the page in another.

Here's what the code looks like with both a `before` and a `beforeEach` hook:

```
describe('Post Editor', function () {
    before(function () {
        // Load the login page
        browser.url('./login');
        // Login with a valid user
        auth.login('demo@learnwebdriverio.com', 'wdiodemo');
    });
    beforeEach(function () {
        // Load the Post Editor page
        browser.url('./editor');
    });
    it('should load page properly', function () {
        // Assert the URL is correct
        // Assert the page fields are correct
    });
});
```

2.5.2 Has the Page Loaded Properly?

Our first test solely consists of validating that the page has loaded correctly. This is a very common test type seen across the industry as it's a common starting ground for test cases.

In fact, this type of test has a common name: "Smoke" tests. They're called that because they check for "smoke," or a quick sign that something is on fire. Their goal is to quickly reveal simple failures before moving on to more in-depth testing.

While the idea may seem simple from the outside, there are a lot of questions that this type test brings up. Here are a few:

1. How much of the page do we test? Testing all elements on the page is impractical, so where do we draw the line?
2. How much maintenance does this introduce? Every element we check introduces maintenance when that element is changed in the future.
3. Should WebdriverIO even be testing this? Are there tools better suited for this type of job?

And here's my opinion:

1. We should only test what we're going to interact with in our other tests, as we want to ensure they're on the page before interacting with them. I admit this isn't the strongest argument, but I like it as a guideline. The counter argument to that is: if the elements are missing, WebdriverIO will throw errors about the element not existing in our other tests. My reply is that we can quickly identify the problem by having a specific test to check that our page elements exist.
2. We can use page objects to reduce future maintenance, but yes, it still exists. We always need to be careful of testing too much.
3. While these 'element check' tests are certainly something WebdriverIO can do, it might not be the best way to do it. Since you're looking at the page when it initially loads, you don't need a tool that interacts with the page, just one that reads it.

An alternative to WebdriverIO is to use a "snapshot" testing tool like Jest[107]. It works with web frameworks like React to render specific components and ensure they render correctly.

Overall, be careful writing a lot of tests like this. Most of the time, it's apparent right away that an element is missing, so having this sort of test only adds extra maintenance. Also, I've seen testers focus solely on this type of testing, when most of the bugs in the software appear when interacting with the website through clicks and form entry. Focus on the items most likely to break, not static data that's mostly reliable.

With all of that said, I want to continue with using this as an example as it gives us a chance to hone our Page Object creation skills from the previous chapter.

[107] https://jestjs.io/docs/en/snapshot-testing

Our test is going to check the URL, then check for the existence of the following five form items: Title, Description, Body, Tags and Publish.

As we've already covered creating page objects, I'm going to show just the code needed to move forward:

Create a new `Editor.page.js` file inside the `pageObjects` folder:

test/pageObjects/Editor.page.js

```
class Editor {
    get $title () { return $('[data-qa-id="editor-title"]'); }
    get $description () { return $('[data-qa-id="editor-description"]'); }
    get $body () { return $('[data-qa-id="editor-body"]'); }
    get $tags () { return $('[data-qa-id="editor-tags"]'); }
    get $publish () { return $('[data-qa-id="editor-publish"]'); }
}

module.exports = Editor;
```

Then back in our test:

test/specs/editor.js

```
const Auth = require('../pageObjects/Auth.page');
const Editor = require('../pageObjects/Editor.page');

const auth = new Auth();
const editor = new Editor();

describe('Post Editor', function () {
    before(function () {
        browser.url('./login');
        auth.login('demo@learnwebdriverio.com', 'wdiodemo');
    });
    beforeEach(function () {
        browser.url('./editor');
    });
    it('should load page properly', function () {
        expect(browser).toHaveUrl('editor', { containing: true });
        expect(editor.$title).toBeExisting();
        expect(editor.$description).toBeExisting();
        expect(editor.$body).toBeExisting();
        expect(editor.$tags).toBeExisting();
        expect(editor.$publish).toBeExisting();
```

```
    });
});
```

To recap, we create a new Editor class with our element references, then require that file inside our test and use those element references to check for their existence.

You can give your tests a quick run with `npx wdio --spec=editor` and validate that it passes.

2.5.3 Storing Common Credentials as Test Fixtures

Before moving on, I'd like to review the fact that we have common login credentials shared between our two test files. If the username/password ever change, this is going to mean extra work, as we'll have to make updates in multiple places. Before we get too far down the road of redefining these credentials in even more files, let's look at how we can store them in a common file for reference in our tests.

Similar to how we store our page objects in separate files, we can do the same with our login credentials. Let's create a new file, and in it, define an object which will store information on multiple users:

test/fixtures/users.js

```
const users = {
    user1: {
        email: 'demo@learnwebdriverio.com',
        password: 'wdiodemo'
    }
};

module.exports = users;
```

Reviewing this code, we create a `users` object, and attach the individual user information as a property on this object. We only have one user defined right now (we'll add more later). This user (`user1`) property is itself an object, with `email` and `password` property/value pairs.

We use the standard `module.exports` to pass back the `users` object to whatever file requires it.

Now we need to save this file and load it in our test. But where should we save it? It's not a page object, so including it in that folder wouldn't make sense. Same goes for including it in the 'test' folder.

What we need to do is create a new folder for static test data. Luckily, there's a name for that: 'fixtures'. A 'test fixture' is a 'fixed' bit of data that stays the same between test runs, so that test runs are repeatable. Here, our user information will always have the same values, hence they're a 'fixture'. Other examples of fixtures are:

- Files used for testing file upload functionality
- "Known" data for validating page components like tables or dynamic lists (e.g., when I sort by the 'Name' column, I expect 'Aaron' to be the first row)
- "Known" data for validating form submissions (e.g., when I submit a new post titled 'My Awesome Post', I expect to be taken to a page with 'My Awesome Post' as the title)

There are other types of test fixtures out there, and we'll cover instances of 'fixtures' that aren't actually static. But for now, how about we get back to our users file?

Let's create a `fixtures` folder to live next to our `specs` and `pageObjects` folder. After creating it, you should have a folder structure like this:

```
./
./wdio.conf.js
./test/fixtures/
./test/specs/
./test/pageObjects/
```

We'll name our file `users.js`, as it contains our user information. Now, back in our `editor.js` file, we can import that data using `const { user1 } = require('../fixtures/users');`.

If you're not familiar with 'object destructuring', that syntax might look a little strange. If we imported the entire `users` object via `const users = require('../fixtures/users');`, we'd need to get our `user1` property out of it like so: `const user1 = users.user1`.

Instead, we use object destructuring[108] to automatically create that `user1` constant when we require the file. It's the same result, just a little bit more concise.

To use our `user1` constant, we update our `auth.login` code to pass in the property values: `auth.login(user1.email, user1.password);`

Here's what the fully updated file looks like:

```
const Auth = require('../pageObjects/Auth.page');
const Editor = require('../pageObjects/Editor.page');
const { user1 } = require('../fixtures/users');

const auth = new Auth();
const editor = new Editor();

describe('Post Editor', function () {
    before(function () {
        browser.url('./login');
        auth.login(user1.email, user1.password);
```

[108] https://exploringjs.com/es6/ch_destructuring.html

```
    });
    beforeEach(function () {
        browser.url('./editor');
    });
    it('should load page properly', function () {
        expect(browser).toHaveUrl('editor', { containing: true });
        expect(editor.$title).toBeExisting();
        expect(editor.$description).toBeExisting();
        expect(editor.$body).toBeExisting();
        expect(editor.$tags).toBeExisting();
        expect(editor.$publish).toBeExisting();
    });
});
```

2.5.4 Using and Updating the Auth Login Function

Fair warning: This next section will not focus on testing much at all. It's entirely unnecessary and you'll likely fine skipping over it. If you're like me, though, you're very curious to read more about it.

Our `auth.login` function takes two parameters: a username and a password. We pass those values in as part of our 'user' object, which is to be expected.

But why not just pass in the entire user object and let the function handle separating things out? The reason here is that "we initially coded the function to work this way, when we were just hardcoding the values via strings."

I mention all of this because it's quite common to follow a pattern due to always having done it that way. In order to preserve brainpower, we develop reusable patterns based on a specific way of thinking. We then re-use these standards elsewhere, even when a better option exists, out of pure habit.

In this instance, we follow the "two parameters" pattern, even when it would be simpler to pass in only the object now that we've switched to that.

But, and this is why I prefaced this entire section, it almost entirely doesn't matter whether we pass in a single object or two strings. From a code re-usability standpoint, they're almost the same. In fact, passing in two strings is more clear from an upfront readability standpoint, as someone reading the tests will understand what's inside the `user1` object without having to trace it down.

It's all one big trade-off.

For the sake of my argument though, let's switch to passing in an object, in an effort to save effort down the road. First, we need to update our function call to pass in the object, instead of the parts. That's a simple change, as we just delete half the function call:

```
auth.login(user1);
```

This is the big payoff of this approach. It's a much more concise login call.

Now for the slightly more complicated part. In our `Auth.page.js` file, we need to update our `login` function to accept an object with two properties. It used to be trickier, but with 'named parameters', we can do a little trick that looks like:

```
login ({ email, password} ) {
```

All we did was add curly braces around what are now our property names. If you're looking for more information on how this works, Wes Bos has a video on how these "Named Parameters" work[109]. This trick can be helpful in your day-to-day coding, so I think it's well worth knowing.

Okay, one last bit before our transition is complete... while it works to pass in the `user1` object for our `editor.js` file, our `login.js` file isn't as easy. There, we're still using the strings to test different credentials.

Since we need to pass in missing details, sending over the full object won't work. Instead, we can update our functions to send in partial data where necessary.

For the first test in the `login.js` file, we can switch over to using the `user1` like we've done in our `editor.js` file. For the other two, we need to define a new object to pass over, using the relevant `user1` property, plus a blank property:

```
const Auth = require('../pageObjects/Auth.page');
const { user1 } = require('../fixtures/users');

const auth = new Auth();

describe('Login Form', function () {
    beforeEach(function () {
        browser.url('./login');
    });

    it('should let you log in', function () {
        auth.login(user1);

        expect(auth.$errorMessages).not.toBeExisting();
    });

    it('should error with a missing username', function () {
        auth.login({
```

[109] https://www.youtube.com/watch?v=c2PGgkCIjEA

```
            email: '',
            password: user1.password
        });

        expect(auth.$errorMessages).toHaveText(`email can't be blank`);
    });

    it('should error with a missing password', function () {
        auth.login({
            email: user1.email,
            password: ''
        });

        expect(auth.$errorMessages).toHaveText(`password can't be blank`);
    });
});
```

Now we've got all our files updated to a new way of thinking and can get back to building out our new Editor test.

2.5.5 Abstracting the Page Load

Just kidding! If you haven't figured it out by now, I really dislike implementation details (e.g., element selectors) in my tests. And right now, there are too many URLs!

For one, we're repeating the `browser.url('./login')` command in both our login and editor tests. While it's not a huge problem right now, it will grow every time we add a new test. Soon enough, we'll have this URL scattered throughout our tests, and any update to the path will require fixing every file we have. That just won't do.

Just like we added element selector information to our page object, we can also add the page URL to it. But we won't do it through a 'getter'; instead we'll create a `load` function that we'll call to run our `browser.url` command:

```
class Login {
    load() {
        browser.url('./login');
    }
}
```

```
before(function () {
    auth.load();
    auth.login(user1);
});
```

Now, we could call that `load` function from inside our login function, and I'm really tempted to do just that. But, I don't want to distract us yet again with yet another micro-optimization, especially when we're already in the middle of one. So I'll leave it up to you for that (in the chapter challenges of course):

We can do the same thing for our Editor page object:

```
class Editor {
    load() {
        browser.url('./editor');
    }
}
```

And in the `beforeEach` hook of our `editor.js` file:

```
beforeEach(function () {
    editor.load();
});
```

You know, we're going to follow this `load` function pattern a fair bit for all of our future page objects. It sure would be nice not to have to write the same `load` function over and over...

2.5.6 Common Page Objects

Which brings us to our next project: Using `generic` classes to provide standard functionality across our page objects. Sounds fun so let's dive in!

The `class` keyword we've been using to build out our page objects comes with the ability to `extend` a base class. It allows you to build off of another class to extend its capabilities.

Think of it like a smartphone. Everyone buys the same standard phone, then they 'extend' that phone's capabilities by customizing details and adding specific apps they want to use. In that same vein, we're going to create a 'base' page object, and customize/extend it for each specific page we want to define.

As I mentioned, it would be nice to avoid redefining that `load` function across our objects. To do this, we're going to create a `Generic` page object class, then `extend` it as needed.

In our `Generic` class, we'll define a `load` function, similar to the one we defined in our `Login` and `Editor` objects:

```
class Generic {
    load() {
        browser.url(???);
    }
}

module.exports = Generic;
```

But how do we know what URL to load? (Hence the ???)

Well, we can pass that information in through a `constructor` function. Constructors are called on classes when they're instantiated, so if I were to instantiate a class like so...

```
const genericPage = new Generic('something');
```

...I could get that value passed in via my constructor function:

```
class Generic {
    constructor (information) {
        console.log(information);
    }
}
```

Now when I run that initial code, it will log out `something` to the console. `constructor` is a special function that will be run whenever the class is instantiated. You don't have to call it in your code; it's just part of the built-in class system.

With that knowledge, we can pass in the URL:

```
class Generic {
    constructor (path) {
        console.log(path); // will log out './login', which is passed in below
    }
}

const genericPage = new Generic('./login');
```

This path value can be stored to the class's `this` value and then used in the `load` function (similar to how we access element references):

```
class Generic {
    constructor(path) {
        this.path = path;
    }

    load() {
        browser.url(this.path); // navigates to './login', which is passed in below
    }
}

const genericPage = new Generic('./login');
```

2.5.7 Time to 'extend' Our Generic Page

That's all fine and dandy, but we're not using 'Generic' pages. We want this to work with our Auth and Editor page objects. We need these classes to 'extend' our Generic page object functionality, which is where the `extends` keyword comes in.

We can define a class to 'extend' another class, similar to the phone analogy before. Both Editor and Auth will extend the base functionality of Generic, including that wonderful `load` function.

```
class Generic {
    constructor(path) {
        this.path = path;
    }

    load() {
        browser.url(this.path); // navigates to './login', which is passed in below
    }
}

class Auth extends Generic {
    // element references and login function left out for brevity
}

const auth = new Auth('./login');
```

Now our Auth class has that `load` function built in, so our `beforeEach` hook now looks like:

```
beforeEach(function () {
    auth.load();
});
```

That's nice, but I don't like passing in the login path via the class instantiation line. We're going to need to use the Auth class in several tests, and it'll be the same issue that we're defining that path in every test. What if we instead store that path information inside the Auth class itself?

Well, in the same way we defined a constructor function in our Generic page, we can define that in our Auth page as well. And instead of it accepting a path, we just set the `this.path` value ourselves:

```
class Auth extends Generic {
    constructor () {
        this.path = './login';
    }

    // element references and login function left out for brevity
}
```

Great, now we can instantiate our Auth page without the path: `const auth = new Auth();`. This all works, but it isn't very elegant. We now have a constructor function defined in the Generic class that isn't really being used. I'd rather have my Auth class pass the path to the Generic constructor function, similar to how we pass in the path via instantiation.

2.5.8 The 'super' Keyword

Well, there's one more trick up the Class sleeve. From within our Auth class constructor, we can 'call up' (or 'call down', not really sure) to the Generic constructor using the `super` keyword:

```
class Auth extends Generic {
    constructor () {
        super('./login');
    }
    // element references and login function left out for brevity
}
```

With this code, when Auth is instantiated, its `constructor` function will be called. This function will call the Generic's constructor via the `super` keyword, passing in our desired path. The Generic constructor function will take over, storing the path information to `this.path`. When `auth.load()` is then called, it will use that stored information and load our login page as desired. We can do the same thing with our Editor class:

```
class Editor extends Generic {
    constructor () {
        super('./editor');
    }
    // element references left out for brevity
}
```

Phew, that's a lot of information to wrap your brain around. If you're looking for alternative explanations, I highly suggest loading up YouTube and watching any of the top videos found when searching "javascript class constructor super". Here are a few:

- JavaScript ES6 / ES2015 - Classes and Inheritance[110]
- Classes in JavaScript with ES6 - p5.js Tutorial[111]
- Class keyword - Object Creation in JavaScript P7 - Fun Fun Function[112]

These videos dive into the idea deeper than I get into here and are great to help solidify what we just talked about. We're going to be using this pattern a fair bit moving forward, so it's worth it to spend time learning about the topic.

2.5.9 Implementing Our Generic Page Object

We've gone through a lot of theory and have reworked the code several times. Let's look at what our final implementation of this idea is going to be. The first item on our list is to create the file for our Generic page object. We'll name the file `Generic.page.js`, and store it in our `page_objects` folder.

The contents of that file will be:

```
class Generic {
    constructor(path) {
        this.path = path;
    }
    load() {
        browser.url(this.path);
    }
}

module.exports = Generic;
```

Next up, we need to update our Auth and Editor files to import the Generic page object and extend it:

[110] https://www.youtube.com/watch?v=RBLIm5LMrmc
[111] https://www.youtube.com/watch?v=T-HGdc8L-7w
[112] https://www.youtube.com/watch?v=Tllw4EPhLiQ

```
const Generic = require('./Generic.page');

class Auth extends Generic {
    constructor() {
        super('./login')
    }

    get $email () { return $('input[type="email"]'); }
    get $password () { return $('input[type="password"]'); }
    get $signIn () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    login({ email, password }) {
        this.$email.setValue(email);
        this.$password.setValue(password);

        this.$signIn.click();

        // wait until either the sign in button is gone or an error has appeared
        browser.waitUntil(
            () => {
                const signInExists = this.$signIn.isExisting();
                const errorExists = this.$errorMessages.isExisting();

                return !signInExists || errorExists;
            },
            {
                timoutMsg:
                    'The "Sign in" button still exists and an error never appeared'
            }
        );
    }
}

module.exports = Auth;
```

```
const Generic = require('./Generic.page');

class Editor extends Generic {
    constructor() {
        super('./editor')
    }

    get $title () { return $('[data-qa-id="editor-title"]'); }
    get $description () { return $('[data-qa-id="editor-description"]'); }
    get $body () { return $('[data-qa-id="editor-body"]'); }
    get $tags () { return $('[data-qa-id="editor-tags"]'); }
    get $publish () { return $('[data-qa-id="editor-publish"]'); }
}

module.exports = Editor;
```

With those adjustments made, we can run our tests again, which now use `auth.load()` and `editor.load()`, and validate they all pass with flying colors.

2.5.10 Using Our Page Object's Path in Tests

If you had a close eye on the editor test, you may have noticed this line:

```
expect(browser).toHaveUrl('editor', { containing: true });
```

If we've defined that path value in our `Editor` page object, why aren't we using it in our tests? Let's get to fixing that!

Similar to how we can access element references in our tests (e.g., `editor.$title`), we can also access properties defined to the `this` value. That means we can access that `path` property via `editor.path`. So we could update our test to be:

```
expect(browser).toHaveUrl(editor.path, { containing: true });
```

But there's a problem with that. If we were to run the test, it would fail with the error message: `expected 'http://localhost:8080/editor' to include './editor'`

The reason for this is that `path` is set to `./editor` and not just `editor`. Why the extra `.`? Well if you remember back to Chapter 2.1, we needed to set our path to be relative to the `baseUrl`. Using the `.` does just that.

That means we can't change our `path` property definition, because it would break our tests if we have a `baseUrl` with a built-in path. That leaves us with a few options:

1. Manually trim off the `.` from the start of the path: `editor.path.substr(1);`
2. Create a second property that doesn't have a period: `this.path = '/editor';`
3. Use the NodeJS `URL` module to parse the path

I don't like Option #1 because our test now needs to "do work." It's not just consuming information, it's modifying it to be in a specific format. I heavily shy away from this, as it can easily introduce bugs into your test code, which we want to avoid at all costs. No one wants to write tests for their tests.

Option #2 isn't great as it redefines the same information. It adds an extra line of code that offers very little value.

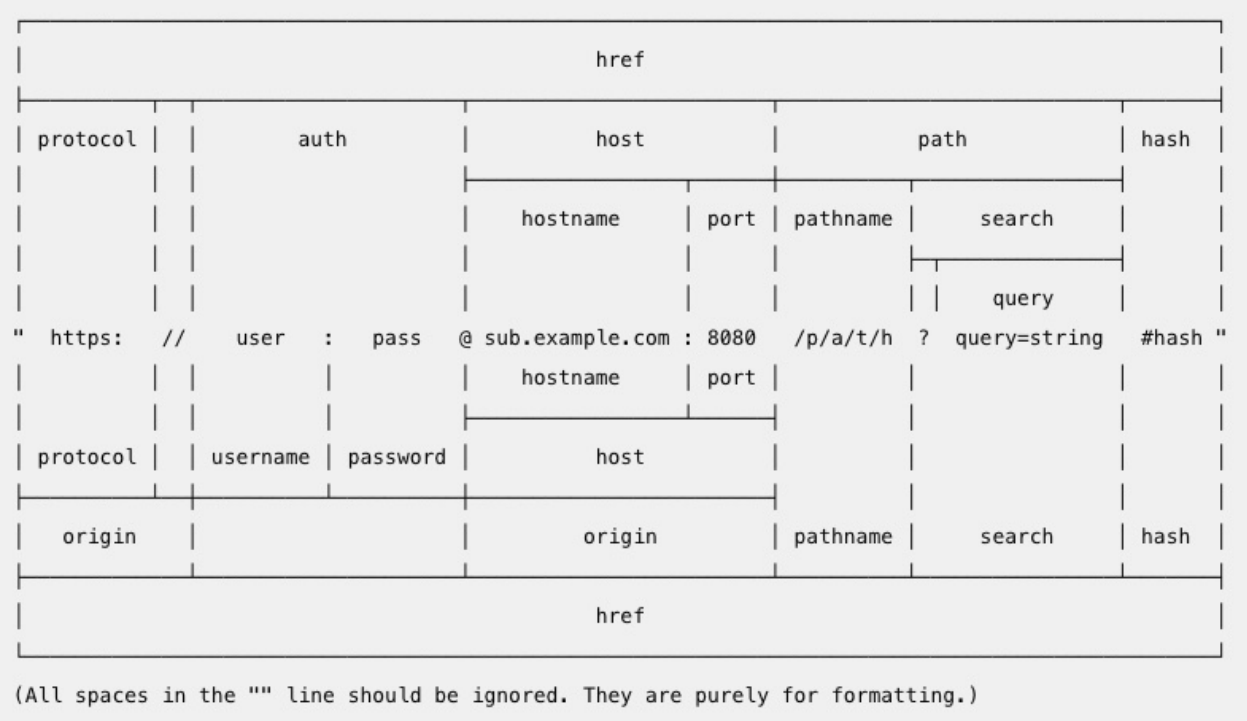
Instead, let's try out Option #3 and see if we can get some added benefit from the work.

2.5.11 Taking Advantage of NodeJS's URL Utility

NodeJS comes with several built-in utilities, some of which we've already tried out (e.g., the `Assert` module). As I mentioned, there's a `URL` module[113] we can use to define and parse URLs.

URLs may seem simple, but they can be very complex. Take a look at the example structure provided by the NodeJS website[114]:

[113] https://nodejs.org/dist/latest-v12.x/docs/api/url.html
[114] https://nodejs.org/api/url.html#url_url_strings_and_url_objects



Structuring Diagram of a URL

All of that can go into a URL, and trying to break a URL string into those separate parts is complicated. But this is where the URL module helps out. We can pass in that very long, complex URL string to the module, and it will return a `URL Object` that has all of those properties defined.

For example:

```
const url = require('url');

const myURL = new URL(
    'https://user:pass@sub.example.com:8080/p/a/t/h?query=string#hash'
);
console.log(myURL.hash);
// prints '#hash'

console.log(myURL.pathname);
// Prints '/p/a/t/h'
```

Any property defined in the chart above can be accessed as a property on our URL.

Another neat bit about the URL constructor is that it doesn't need to be a single string that we pass in. We can send it parts of our URL, and it will piece together everything into a full object:

```
const myURL = new URL('/foo', 'https://example.org/');
// https://example.org/foo
```

In this example, we send over the `path` and `host` in separate strings, and the URL constructor joins them correctly. This matches up with our tests, as we have our `path` in our page object and the `host` stored as our `baseUrl` in our config. Now we just need to combine those two parts to make our URL object.

We want to store the URL object in our Generic page as `this.url`. This is almost the same as `this.path`, except we'll be constructing our object instead of passing it in as a string.

To get the `baseUrl` from inside our page object, we can reference `browser.config`. This is a store of most of the options we defined inside our `wdio.conf.js` file, including our `baseUrl` property. It's really handy to know about it for specific circumstances, so commit it to memory.

Pulling this all together, our Generic page object now looks like:

```
const { URL } = require('url');

class Generic {
    constructor(path) {
        this.path = path;

        // store the url by combining specific page path with WDIO base url
        // using the NodeJS URL utility
        this.url = new URL(path, browser.config.baseUrl);
    }
    load() {
        browser.url(this.path);
    }
}

module.exports = Generic;
```

Note that we added the `require` statement to the top of the file, and that our `load` function stayed the same. All we're doing here is adding a new property for our Editor and Auth pages to use. I think it's neat that we can add this new property to our Generic page, and magically have it available in our other pages.

2.5.12 Finishing the URL Implementation

Okay, now that we've updated our page objects, let's put this to use in our test. Our assertion is checking that the URL is correct for the page we want to be on. It was doing a partial check before,

validating that the path exists inside the returned URL. But now that we can build out the full url, why not do a strict match to really make sure we're on the same page?

The URL object that we've constructed contains an `href` property that will return the entire URL. We can compare that against what's returned from the browser to complete our assertion: `expect(browser).toHaveUrl(editor.url.href);`

Notice that we're omitting the `{ containing: true }` option from before. Again, this is due to our ability to combine the `path` and `baseUrl` values to make a fully qualified URL. Here's what the fully updated `editor.js` file now looks like:

test/specs/editor.js

```
const Auth = require('../pageObjects/Auth.page');
const Editor = require('../pageObjects/Editor.page');
const { user1 } = require('../fixtures/users');

const auth = new Auth();
const editor = new Editor();

describe('Post Editor', function () {
    before(function () {
        auth.load();
        auth.login(user1);
    });
    beforeEach(function () {
        editor.load();
    });
    it('should load page properly', function () {
        expect(browser).toHaveUrl(editor.url.href);
        expect(editor.$title).toBeExisting();
        expect(editor.$description).toBeExisting();
        expect(editor.$body).toBeExisting();
        expect(editor.$tags).toBeExisting();
        expect(editor.$publish).toBeExisting();
    });
});
```

Well, that sure was a lot to go through for a simple test. It's worth it though. By extending more generic classes, we can continue to grow our automation test suite without running into issues with code duplication, which can create maintenance nightmares in the long run.

One final note: you can have JavaScript classes that extend classes that extend classes. Right now, we have a Generic class, but we could also have a SlightlyLessGeneric class that's extended by specific page types. We'll take a look at that in a few chapters. Next up: learning how to test complex inputs.

2.5.13 Chapter Challenge

- Move the Auth's `load` function call to be inside the `login` function.
- Add element references to the Generic page for common items like the site navigation and page footer

2.6 Testing Complex Inputs

2.6.1 Testing the Publish Action

It's nice that we can now confirm that our post editor page is loading correctly with all the proper form fields available. That said, it would be a lot more useful to confirm that those form fields actually work.

It's not difficult to manually load a page and validate fields exist, but it takes effort and time to fill out the fields and check the submission functionality. That "little bit of effort" can be just enough to make developers say "my change shouldn't have broken that" and not regression test their work.

I'm honestly not criticizing this "lazy" behavior; just admitting that I've been that developer, and anything I can do to make up for that laziness is appreciated. Maybe that's why I like test automation so much.

That idea brings us to our next test, which is that the Editor page `should let you publish a new post`. In the test, we'll do three things:

- Set a value for the 'Title'
- Set a value for the 'Description'
- Set a value for the 'Body'

There is a "Tags" field we can fill out as well, but we'll save that for later, as it requires a little bit of extra knowledge that I don't want to get into just yet.

Using our `editor` page object, we can set all three values calling the `setValue` command we used before:

```
it('should let you publish a new post', function () {
    editor.$title.setValue('Test Title');
    editor.$description.setValue('Test Description');
    editor.$body.setValue('Test Body');
});
```

Except we can't. While the `setValue` command works for the first two fields, the 'Body' field throws an error: `invalid element state: Element must be user-editable in order to clear it.`

The reason it's mad is that we can only call `setValue` on 'editable' elements (e.g., `<input>` or `<textarea>` elements). But our selector is for the `<div>` that contains that textarea.

This is an example of how complex form fields (in this case, a custom Markdown editor), can increase the complexity of our tests. Here, we're using the third-party `mavon-editor` component[115] available for Vue-based applications.

And because it's a third-party component, we're unable to access the HTML for that `<textarea>` element, keeping us from attaching our `qa-id` attribute directly to it. If you're curious, this is what the Vue "HTML" looks like:

```
<mavon-editor
    :toolbars="{
        bold: true,
        italic: true,
        header: true,
        underline: true,
        strikethrough: true
    }"
    v-model="article.body"
    placeholder="Write your article (in markdown)"
    data-qa-id="editor-body"
/>
```

Again, the `<textarea>` is hidden behind the `mavon-editor` component, denying us the ability to modify it. Thankfully, the solution is pretty simple. In our `$body` element reference, change our selector to point specifically to the textarea inside the element with the proper `data-qa-id`:

```
get $body () { return $('[data-qa-id="editor-body"] textarea'); }
```

Update this selector and our test now passes.

Many web apps face a similar issue with popular third-party tools like ACE editor or CodeMirror. Working with them will require investigating the underlying HTML and finding the correct input field. There's no one-size-fits-all solution here, you just have to handle it one instance at a time.

If you're unfamiliar with using the Chrome DevTools to inspect the generated HTML of a website, have a read through their documentation[116] to become familiar with the very useful functionality it provides.

2.6.2 Key Commands

I mentioned before that we can add tags to our posts, but that it's also more complicated to do. That's because adding a tag takes two actions:

[115] https://www.npmjs.com/package/mavon-editor
[116] https://developers.google.com/web/tools/chrome-devtools/dom

1. Set the value of the tag you want
2. Press the 'enter' key to trigger a "tag" being formed

We have yet to cover keyboard actions, so let's tackle that right now.

At any point in your automation, you can send a 'key' event to the browser by using the `browser.keys` command. For example, if I wanted to trigger the keyboard shortcut for the `?` character, I would use `browser.keys('?')`. This would be useful for testing if typing the `?` in the browser triggered a "keyboard shortcuts" popup to appear.

We're going to use this command to send an `enter` key event. But that's a bit different from a normal alphanumeric key. You wouldn't just say `browser.keys('Enter')`, would you?

Yes, you would!

You can use characters like "Left arrow" or "Backspace", or in our instance "Enter", and WebdriverIO takes care of translating them into the unicode characters needed to represent those keys. The W3C has a list of all supported keywords[117] available for your reference.

So, to get back to our tags field, this is what the code will look like:

```
editor.$tags.setValue('Tag1');
editor.$tags.keys('Enter');
```

Notice that, in this instance, we use `editor.$tags.keys('Enter')`. While `browser.keys('Enter')` would work the same, specifying the element we want to type in helps clarify the goal of our code.

2.6.3 Submitting and Validating the Result

The last steps in our test are to click the submit button, then validate our results.

Tackling the clicking of the button is a simple command:

```
editor.$publish.click();
```

Asserting the article was submitted correctly is a whole different story, all together (*Airplane!* fans respond: "It's a whole different story").

There are many levels of validation we can take; one being to validate the generated URL matches the title of our post. So if the title we set was `Test Title`, then we'd expect the URL to contain `articles/test-title` (as it would be converted to a URL-friendly format).

Next, we'll run our assertion that the URL is correct:

[117] https://w3c.github.io/webdriver/#keyboard-actions

```
expect(browser).toHaveUrl('articles/test-title', { containing: true });
```

This is similar to the assertion we used in our Login test.

While we could end here, our test has "polluted" the data on the site and we should take care to clean that up. We created an artifact in our test: a new article. If we were to run this test a second time, the URL would actually be something like `articles/test-title-dpywzg`. That extra bit at the end was added to avoid conflicting URLs.

While this is good that the website handles duplicate post titles, it's a good idea to delete the article at the end of our test to avoid unwanted side effects. Thankfully, that's pretty simple to do, considering the site offers us a 'delete' button on the article page itself.

So, let's add one last command to our test to delete the newly created article, just to be good stewards of the site:

```
$('button*=Delete Article').click()
```

Let's take a look at the entire test now:

```
it('should let you publish a new post', function () {
    editor.$title.setValue('Test Title');
    editor.$description.setValue('Test Description');
    editor.$body.setValue('Test Body');

    editor.$tags.setValue('Tag1');
    editor.$tags.keys('Enter');

    editor.$publish.click();

    // expect to be on new article page
    expect(browser).toHaveUrl('articles/test-title', { containing: true });

    // to avoid making a lot of articles, let's just click the delete button to
    // clean it up. We'll talk about a better way to clean it later on.
    $('button*=Delete Article').click();
});
```

Yes, we should move our article page selectors to a separate page object, but let's let that detail go for now. There's plenty of time for that later. I want to look at how we can improve other bits.

2.6.4 Form Submission Functions

Recall back in our Login page object, we wrote a function inside it to handle the login flow. Let's do the same thing for our article submission. This way, we have a common function to call when we want to test anything related to the editor page.

Similar to our login function, we'll create a `submitArticle` function which will take an object with the following properties:

- title
- description
- body
- tags

The code inside our function will look almost entirely the same as what we already have in our test. However, we want to make one small adjustment.

For the tag field, it would be nice to allow for multiple tags. We can accomplish this by using the `forEach` function available in arrays. This means the `tags` property will need to be passed in as an array, so our function call would look like:

```
editor.submitArticle({
    title: 'Test Title',
    description: 'Test Description',
    body: 'Test Body',
    tags: ['Tag1']
});
```

In that example, we only pass in a single tag, but still use an array for it. This way our function does not need to handle multiple data types. Here is what that function is going to look like:

```
submitArticle({ title, description, body, tags }) {
    this.$title.setValue(title);
    this.$description.setValue(description);
    this.$body.setValue(body);
    tags.forEach((tag) => {
        this.$tags.setValue(tag);
        this.$tags.keys('Enter');
    });
    this.$publish.click();
}
```

Reviewing it, we accept a single object as an argument, which has the four required properties we mentioned. Then, we use the `setValue` command three times for our text fields.

Next, we use the `forEach` function to loop through the tags passed in. If you're unfamiliar with this method, check out the MDN documentation on `forEach`[118].

Inside this loop, we tag the specific tag text we want to use, set the value of the form field to that, then press enter to trigger the tag being formed.

Finally, we end the function by clicking the publish button, which submits the form.

Combined with our page object, here's how the entire thing looks:

test/specs/Editor.page.js

```
const Generic = require('./Generic.page');

class Editor extends Generic {
    constructor () {
        super('./editor')
    }

    get $title () { return $('[data-qa-id="editor-title"]'); }
    get $description () { return $('[data-qa-id="editor-description"]'); }
    get $body () { return $('[data-qa-id="editor-body"] textarea'); }
    get $tags () { return $('[data-qa-id="editor-tags"]'); }
    get $publish () { return $('[data-qa-id="editor-publish"]'); }

    submitArticle({ title, description, body, tags }) {
        this.$title.setValue(title);
        this.$description.setValue(description);
        this.$body.setValue(body);
        tags.forEach((tag) => {
            this.$tags.setValue(tag);
            this.$tags.keys('Enter');
        });
        this.$publish.click();
    }
}

module.exports = Editor;
```

And to update our test, we'll replace the relevant lines of code with our new function call:

[118] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

```
it('should let you publish a new post', function () {
    editor.submitArticle({
        title: 'Test Title',
        description: 'Test Description',
        body: 'Test Body',
        tags: ['Tag1']
    });

    // expect to be on new article page
    expect(browser).toHaveUrl('articles/test-title', { containing: true });

    // to avoid making a lot of articles, let's just click the delete button to
    // clean it up. We'll talk about a better way to clean it later on.
    $('button*=Delete Article').click();
});
```

Great, things are a bit cleaner in our test, and any updates/improvements to the editor form can all be handled from inside the Editor page object.

2.6.5 ChanceJS to Create Data

While not required, it can be useful (and a little fun) to mix in some random generation into our tests. When working with data entry, using the same data is simpler, but very restricted in scope of testing.

In our previous test, we're only validating those four pieces of data. It would be more useful to validate random strings with each test run, possibly revealing a bug in the system for when the article is titled "sfdeljknesv[119]".

One quick way to introduce some randomization to your tests is to use the `Date.now()` method[120] available natively in NodeJS. This function returns the current number of milliseconds elapsed since January 1, 1970. Every time you run your test, you'll get a larger number, as more milliseconds have elapsed since that last run.

While that's helpful, it's really only testing a simple numeric input. The numbers may change, but the data format doesn't. What we need is a full-featured utility that allows us to request random types of data. That's where a tool like ChanceJS[121] can help out.

Honestly, this may be overkill, and more for fun than anything else, but it's a good example of how we can use the NodeJS ecosystem to extend our test functionality.

To use ChanceJS, we need to do several things.

[119] https://twitter.com/sempf/status/514473420277694465
[120] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/now
[121] https://chancejs.com/

1. Install the library via `npm i chance`
2. Require the library
3. Instantiate it
4. Call the specific function needed to generate the random data

Assuming you've done the first step, we'll add the next two to the top of our test file:

```
const Auth = require('../pageObjects/Auth.page');
const Editor = require('../pageObjects/Editor.page');
const { user1 } = require('../fixtures/users');

const auth = new Auth();
const editor = new Editor();

// Load Chance
const Chance = require('chance');

// Instantiate Chance so it can be used
const chance = new Chance();
```

Then, in our `editor.submitArticle` function call, update our static strings to Chance function calls:

```
editor.submitArticle({
    title: chance.sentence({ words: 3 }),
    description: chance.sentence({ words: 7 }),
    body: chance.paragraph({ sentences: 4 }),
    tags: [chance.word(), chance.word()]
});
```

Actually, this won't work. In our URL assertion, we test the URL of the page with the assumption that our title will be `Test Title` (the generated URL of the post page is based off of the title). With a randomized title, we won't know what to assert against.

What we need to do is store the article information as a separate object and send it into `submitArticle`. That's easy enough to do:

```
const articleDetails = {
    title: chance.sentence({ words: 3 }),
    description: chance.sentence({ words: 7 }),
    body: chance.paragraph({ sentences: 4 }),
    tags: [chance.word(), chance.word()]
};

editor.submitArticle(articleDetails);
```

Then in our URL assertion, reference the `title` property stored in our `articleDetails` object. Except... we need to convert the title to the URL version. Here's what that looks like:

```
const slug = articleDetails.title
    .toLowerCase()
    .replace(/ /g, '-')
    .replace(/[^\w-]+/g, '');

// expect to be on new article page
expect(browser).toHaveUrl(`articles/${slug}`, { containing: true });
```

Yuck. Not only do we need to run two `replace` calls, we're also introducing "logic" into our tests, and having to match code functionality between our site and our tests. All things I generally shy away from, as they can quickly introduce bugs into our tests, which is never fun.

2.6.6 Checking Page Data

Instead of checking the URL, let's check the content of the page to see that it matches our submitted parts.

Let's create a new 'Article' page object with our article element references. We won't have a 'constructor' function because we don't have a set URL and can't easily auto-generate one.

test/pageObjects/Article.page.js

```
const Generic = require('./Generic.page');

class Article extends Generic {
    get $container () { return $('[data-qa-id="article-page"]'); }
    get $title () { return $('[data-qa-id="article-title"]'); }
    get $body () { return $('[data-qa-id="article-body"]'); }
    get $tags () { return $('[data-qa-id="article-tags"]'); }
    get $edit () { return $('[data-qa-id="article-edit"]'); }
    get $delete () { return $('[data-qa-id="article-delete"]'); }
```

```
}

module.exports = Article;
```

We honestly don't need to extend the Generic page object, as we're not using any of its functionality. But I'm going to keep it just for consistency sake.

We'll have to look at that 'tags' element reference in a bit, because it's actually multiple tags, not a single string. When we test that, we'll handle a single tag or multiple, but for now let's ignore that little detail.

Save that file in your `pageObjects` folder as `Article.page.js` and load it into your `editor` test file along with the Editor page object:

test/specs/editor.js

```
const Auth = require('../pageObjects/Auth.page');
const Editor = require('../pageObjects/Editor.page');
const Article = require('../pageObjects/Article.page');
const { user1 } = require('../fixtures/users');

const auth = new Auth();
const editor = new Editor();
const article = new Article();
```

Then we're going to update our test to reference our new page object, including replacing the URL check with a couple element text checks:

```
it('should let you publish a new post', function () {
    const articleDetails = {
        title: chance.sentence({ words: 3 }),
        description: chance.sentence({ words: 7 }),
        body: chance.paragraph({ sentences: 4 }),
        tags: [chance.word(), chance.word()]
    };

    editor.submitArticle(articleDetails);

    expect(article.$title).toHaveText(articleDetails.title);
    expect(article.$body).toHaveText(articleDetails.body);

    // to avoid making a lot of articles, let's just click the delete button to
    // clean it up. We'll talk about a better way to clean it later on.
    article.$delete.click();
});
```

Now we're not dependent on some tricky logic to use the URL for validation of our post contents. Plus, it's more important for the information on the page to be correct than the URL be in some specific format. I'd much rather know that our title isn't saving correctly than if our URL isn't right.

2.6.7 Testing Multiple Tags

I talked early about how the 'tags' element reference wasn't quite usable in its current state. This is due to the fact that we can have multiple tags, but are targeting a single reference. So if we run `getText` on that element, we'd get all the tags in a single string (e.g., 'tag1 tag2').

Instead, let's update our selector to target each tag individually, that way we can run the `getText` function on each separate tag, and save them all as an array for later comparison.

On a side note, you can still do the assertion using this combined string. It would look something like: `expect(article.$tags).toHaveText(articleDetails.tags.join(' '));`.

So the first thing is to add a new element reference, which looks like:

```
get $tags () { return $('[data-qa-id="article-tags"]'); }
get $$tags () {
    return $$('[data-qa-id="article-tags"] [data-qa-type="article-tag"]');
}
```

We use `$$` to target all elements that match (not just the first), and we add a second bit of information to our selector to match the specific tag (not just the container of the tags).

Now we need to extract the text out of the returned elements. We **can't** run `article.$$tags.getText()`, because that function only works on a single element reference. `article.$$tags[0].getText()` would work, since it's a single element, but then we'd only be getting that single bit of information when we really want all of it.

If you recall, we used the built-in `forEach` array functionality to add our tags to the editor page. In a similar way, we can use the built-in `map` function to pull information out of our array. `map` is very similar to `forEach`. The main difference is that `map` allows you to take the data in the array and return something else from it.

Let's look at an example using some simple math. Say we have an array of numbers:

```
const myAwesomeNumbers = [3, 4, 7];
```

Now we want to double each of those numbers (e.g., 3 -> 6).

Using `forEach`, we could `console.log` out the information:

```
myAwesomeNumbers.forEach(awesomeNumber => {
    console.log(awesomeNumber * 2);
});
```

This would log out the following three lines:

```
6
8
14
```

But that's not very useful if we want to use those doubled numbers in our code. That's where `map` comes in. If we replace `forEach` with `map`, and then return the result instead of logging it out, we'll get a new array of our doubled amounts:

```
const myAwesomeNumbersDoubled = myAwesomeNumbers.map(awesomeNumber => {
    return awesomeNumber * 2;
});
```

With this, `myAwesomeNumbersDoubled` is `[ 6, 8, 14 ]` and `myAwesomeNumbers` stays the same.

Instead of working with numbers, we can work with element references. Using this same technique, let's take our array of `tag` elements, and return the result of calling `getText` on them:

```
const tags = article.$$tags.map($tag => {
    return $tag.getText();
});
```

Now tags will be an array of whatever text is there (e.g., ['tag1', 'tag2']), which is going to match what we have defined in `articleDetails.tags` and create a handy assertion:

```
const tags = article.$$tags.map($tag => {
    return $tag.getText();
});
expect(tags).toEqual(articleDetails.tags);
```

Since this tags text information is likely to be used in other tests, let's move the mapping function to our `Article` page object. We'll store it as `tags`, but you could use `tagsText` if you prefer to be explicit about its contents. Here's what our fully updated Article looks like:

test/pageObjects/Article.page.js

```
const Generic = require('./Generic.page');

class Article extends Generic {
    get $container () { return $('[data-qa-id="article-page"]'); }
    get $title () { return $('[data-qa-id="article-title"]'); }
    get $body () { return $('[data-qa-id="article-body"]'); }
    get $tags () { return $$('[data-qa-id="article-tags"]'); }
    get $$tags () {
        return $$('[data-qa-id="article-tags"] [data-qa-type="article-tag"]');
    }
    get $edit () { return $('[data-qa-id="article-edit"]'); }
    get $delete () { return $('[data-qa-id="article-delete"]'); }

    get tags () { return this.$$tags.map($tag => $tag.getText()); }
}

module.exports = Article;
```

And here's how the entire Editor test looks with all our changes:

test/specs/editor.js

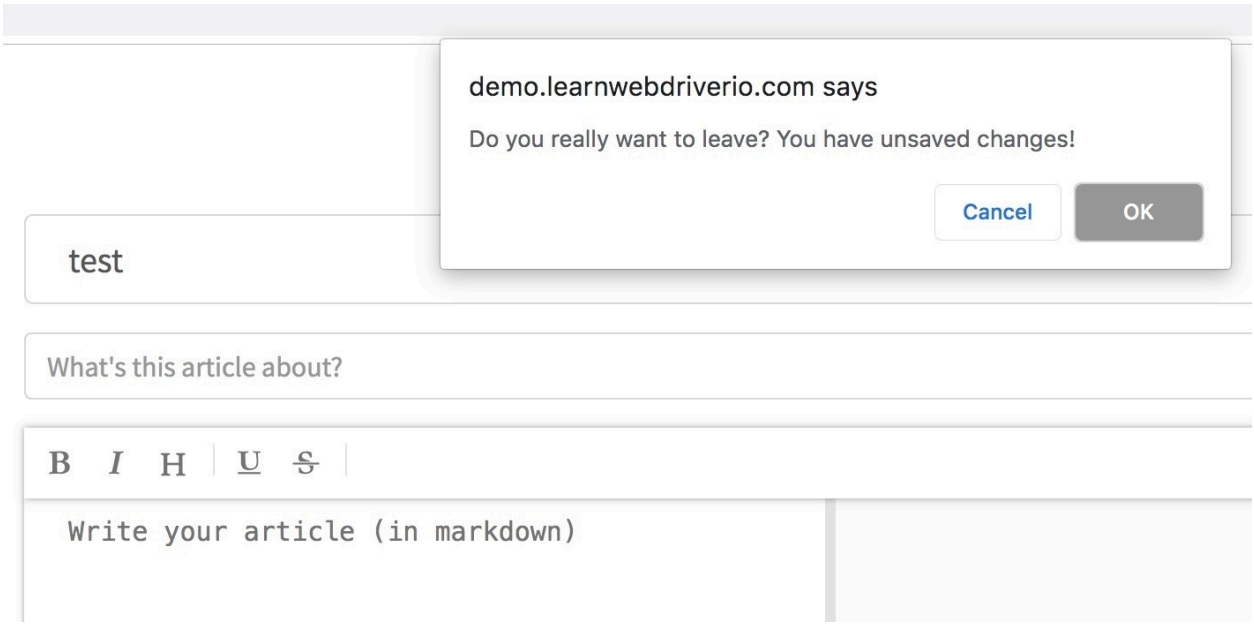
```
const Auth = require('../pageObjects/Auth.page');
const Editor = require('../pageObjects/Editor.page');
const Article = require('../pageObjects/Article.page');
const { user1 } = require('../fixtures/users');

const auth = new Auth();
const editor = new Editor();
const article = new Article();

// Load Chance
const Chance = require('chance');

// Instantiate Chance so it can be used
const chance = new Chance();

describe('Post Editor', function () {
    before(function () {
        auth.load();
        auth.login(user1);
    });
```

```
    beforeEach(function () {
        editor.load();
    });
    it('should load page properly', function () {
        expect(browser).toHaveUrl(editor.url.href);
        expect(editor.$title).toExist();
        expect(editor.$description).toExist();
        expect(editor.$body).toExist();
        expect(editor.$tags).toExist();
        expect(editor.$publish).toExist();
    });

    it('should let you publish a new post', function () {
        const articleDetails = {
            title: chance.sentence({ words: 3 }),
            description: chance.sentence({ words: 7 }),
            body: chance.paragraph({ sentences: 4 }),
            tags: [chance.word(), chance.word()]
        };

        editor.submitArticle(articleDetails);

        expect(article.$title).toHaveText(articleDetails.title);
        expect(article.$body).toHaveText(articleDetails.body);
        expect(article.tags).toEqual(articleDetails.tags);

        // to avoid making a lot of articles, let's just click the delete button to
        // clean it up. We'll talk about a better way to clean it later on.
        article.$delete.click();
    });
});
```

2.6.8 Checking Alerts When Leaving With Unsaved Changes

One of the most frustrating experiences on a website for a user occurs when they've spent a valuable amount of time typing up content, only to have that content viciously erased by an errant page navigation. Whether it's accidentally hitting the back button, unintentionally refreshing the page, or inadvertently going to another site, nothing will fill your user's eyes with tears quicker than causing them losing their work.

To avoid this tragic outcome, website owners now implement "alerts" that pop up when you try leaving certain pages. These alerts will inquire about your intentions and whether they were purposeful or not:



Screenshot of Alert box appearing upon page navigation

It's a minor annoyance for users who don't mind losing their work, but appreciated by clumsier folks like me. We're going to write a few tests checking that these alerts appear on our editor page.

There are two scenarios we want to test:

1. An action is performed outside the page, causing a page navigation event (e.g., clicking the 'refresh' button)
2. An action is performed inside the page, causing a page navigation event (e.g., clicking a link)

Since these two tests are related, we're going to group them under a specific `describe` block:

```
describe('Post Editor', function () {
    ... other tests and such here ...

    describe('"Unsaved Changes" alerts', function () {
    });
});
```

We'll have two tests:

1. it should alert you when using browser navigation

2. it should alert you when clicking a link

In the first test, we'll need to trigger a browser navigation event. We can do this using any of the following commands:

- `browser.url('someotherurl')`
- `browser.back()`
- `browser.forward()` (although this will only work if we've just navigated "back" from a page)
- `browser.refresh()`

The one I'll use is the `refresh` command. By the way, these commands come as part of the standard WebDriver library[122] and are not specific to WebdriverIO.

Before doing that though, I need to edit one of the fields to trigger the "unsaved changes" state to exist (we'll do that in the `beforeEach` hook since it'll be needed for both tests).

Here's what it looks like now:

```
describe('"Unsaved Changes" alerts', function () {
    beforeEach(function () {
        editor.$title.setValue('Unsaved Change');
    });

    it('should alert you when using browser navigation', function () {
        // try refreshing the page
        browser.refresh();

        // validate alert is showing
        // ??
    });
});
```

Pretty basic. Now let's figure out how to check for the alert. There are four "alert" related commands available in the WebDriver standard:

- dismissAlert
- acceptAlert
- getAlertText
- sendAlertText

[122] https://webdriver.io/docs/api/webdriver.html

For this test, we're just going to try accepting the alert. If you were to read the documentation for that command, you would see that it throws an error if the alert doesn't exist: "The Accept Alert command accepts a simple dialog if present, otherwise error."

We can use that error to validate that the alert is shown (if no error is thrown when calling `acceptAlert`, then an alert was shown and accepted). Normally, code throwing errors would trigger a complete failure condition and halt any other execution. But `expect-webdriverio`, and other assertion libraries, come with the ability to handle errors thrown in a gracious manner.

There is a `toThrow` method[123] available in ExpectJS that works like this: You call `expect()` and pass in a function to run which may or may not throw an error. You finish your assertion with `toThrow()` or `not.toThrow`, depending on what you expect. When the time comes, ExpectJS (which is what `expect-webdriverio` is built off of) runs this function (handling any thrown errors with grace) and asserts your expected outcome. All in all, it looks like this:

```
expect(() => browser.acceptAlert()).not.toThrow();
```

Remember, we can't do: `expect(browser.acceptAlert()).not.toThrow();` because if `browser.acceptAlert()` did throw an error, it would happen before ExpectJS is ready to handle it. That's why we need to wrap our command in a function.

Here's what our entire test looks like:

```
describe('"Unsaved Changes" alerts', function () {
    beforeEach(function () {
        editor.$title.setValue('Unsaved Change');
    });

    it('should alert you when using browser navigation', function () {
        // try refreshing the page
        browser.refresh();

        // validate alert is showing
        expect(() => browser.acceptAlert()).not.toThrow();
    });
});
```

One thing to know about our assertion is that, since we are actually calling the `browser.acceptAlert()` command, it accepts the alert and our `browser.refresh()` command is fully executed. If we were to have called `browser.dismissAlert()` instead, the `refresh` would not have occurred (it would have been rejected), leaving our page in an "unsaved changes" state. This could cause trouble if a follow-on test tried reloading the page (which it does as part of the `before` hook in our parent

[123] https://jestjs.io/docs/en/using-matchers#exceptions

`describe` block). If you try to navigate via WebdriverIO when an alert is showing, you'll get this error: `unexpected alert open: {Alert text : }`

Okay, on to our second test. In it, we're going to click a link ('Home'), then get the alert text (using `getAlertText`) and validate it's correct.

Why didn't we get the alert text in the previous test? Well, to oversimplify, it was a different type of alert, and running `getAlertText` on it would result in a useless empty string. It's a quirk in how this website works. One type of navigation triggers a `confirm` alert to show (which `getAlertText` can return a valid string from), and the other hooks into BeforeUnloadEvent[124], which causes a dialog box to appear which cannot be read by `getAlertText`.

So, we'll trigger our link navigation through a simple click `$('=Home').click();`

Then we'll get the alert text: `const alertText = browser.getAlertText();`

With that text, we can run our assertion: `expect(alertText).toEqual('Do you really want to leave? You have unsaved changes!');`

And as a final clean-up step, we'll accept the alert: `browser.acceptAlert();`

All together, it looks like:

```
it('should warn you when trying to change URL', function () {
    // try going to the homepage
    $('=Home').click();

    const alertText = browser.getAlertText();

    expect(alertText).toEqual(
        'Do you really want to leave? You have unsaved changes!'
    );

    // accept the alert to avoid it from preventing further tests from executing
    browser.acceptAlert();
});
```

Handling alerts isn't a common part of testing work in my experience, but it is needed from time to time, so it's good to know about.

One final note: you can't use these commands to enter "basic auth" credentials. To do that, you need to pass in the credentials via your `browser.url` call:

`browser.url('http://username:password@example.com/');.`

[124] https://developer.mozilla.org/en-US/docs/Web/API/BeforeUnloadEvent

2.6.9 A Not-So-Random Chance

After all this work, there's one last bit of optimization we should do to prepare ourselves for future testing. Earlier we imported the Chance library to add some randomness to our tests. But what happens when that randomness causes a failure due to some specific data. If you ran the test again, you'd use different data which may not cause the error to occur.

We need to add a way to re-run our tests using the same data from Chance. They allow us to do this by passing in a "seed"[125], which is used to create the random data. Here's a code example:

```
// Load Chance
const Chance = require('chance');

// Instantiate Chance so it can be used, passing in a 'seed' for repeatable data
const chance1 = new Chance('webdriverio');
const chance2 = new Chance('webdriverio');

chance1.integer(); // will always output '2055889922031616'
chance2.integer(); // will also always output '2055889922031616' since it's using th\
e same seed
```

Since both copies of Chance had the same seed ('webdriverio'), they will both generate the same random number sequence each time they're called.

So now we can repeat our data used in our tests, except now we've removed the randomness by running the same data through each time.

Instead, let's set up a system where we create a Chance instance using a random seed, log that seed out to our console, then allow us to pass in that seed in the next test.

The 'seed' can be anything, so let's just use a random number generated by JavaScript:

```
const seed = Math.random();
console.log(`ChanceJS Seed: ${seed}`);
const chance = new Chance(seed);
```

Now when our test is run, it will log something out like `ChanceJS Seed: 0.023341019330085366`

Next, we need to be able to pass in a previously used seed. There are a couple ways of doing that, but I prefer to use environment variables since they're easy to access and set. Let's add in a check for the `SEED` environment variable being set, and have it use that instead, if set:

[125] https://chancejs.com/usage/seed.html

```
if (!process.env.SEED) {
    // store as a string since that's how the SEED environment variable is passed in
    process.env.SEED = Math.random().toString();
}
console.log(
    `ChanceJS Seed: ${process.env.SEED} - Pass in using 'SEED=${process.env.SEED}'`
);
const chance = new Chance(process.env.SEED);
```

So if no `SEED` variable is set, we'll create our own. We'll always log out the value for documentation purposes (you never know when you'll need that information).

Now when we run our `editor.js` test, we'll get that seed data logged out, which we could use again by running:

```
SEED=0.00726358915055525 npx wdio --spec=editor
```

On the Windows command line, this looks like:

```
cmd /C "set SEED=0.00726358915055525 && npx wdio --spec=editor"
```

So long as that `SEED` value is set and stays the same, the editor test will always generate the same "random" data, making it much easier to reproduce any defects in the site.

2.6.10 Global Chance via WebdriverIO Configuration Hooks

Okay, that's some nice functionality, but I'd hate to have to copy/paste it into every test file that I want to use Chance in. I'd rather have a Chance instantiation set up automatically for all my test files in a single location.

This is where WebdriverIO configuration hooks can help out. These are defined at the bottom of our `wdio.conf.js` file and can be used to step into the test process in order to enhance it and/or build services around it.

We already touched on this with our `browser.throttle` call in the `before` hook. The `before` hook function gets called right before test execution begins (hence the name). This is right about the time we want to define our Chance instance. But if we used our normal `const chance = new Chance(process.env.SEED);` code, that `chance` constant would only be available inside the `before` function and not in our tests.

It turns out there's a `global` object in NodeJS[126] that you can add properties to which will be stored for any other file run in that same Node process. Adding a property to it is the same as any other object: `global.someVar = 'some value';`.

In this case, we'll set `global.chance` to our seeded Chance instance:

```
before: function (capabilities, specs) {
    // browser.throttle({
    //     latency: 1000,
    //     offline: false,
    //     downloadThroughput: 1000000,
    //     uploadThroughput: 1000000
    // });

    // store Chance globally so all tests can use it with the specific seed
    // In order to avoid chance re-using the same seed for each test file,
    // we create a Chance instance using the base seed,
    // plus the path of the file (specs[0])
    global.chance = new Chance(process.env.SEED + specs[0]);
}
```

What about the other code needed to make this run? That needs to go at the top of our configuration file so that we're only defining our SEED value once per test run.

```
const Chance = require('chance');

// Instantiate Chance so it can be used
if (!process.env.SEED) {
    // store as a string since that's how the SEED environment variable is passed in
    process.env.SEED = Math.random().toString();
}
console.log(
    `ChanceJS Seed: ${process.env.SEED} - Pass in using 'SEED=${process.env.SEED}'`
);

exports.config = {
  ... configs here (including the `before` hook) ...
}
```

Finally, make sure you clean up the `editor.js` file to remove the Chance configuration at the top. You can leave the chance data calls the same (Node looks on the `global` object by default if it doesn't find `chance` in the local scope. Alternatively, if you want to self-document that `chance` is coming from the `global` object, update your code to look like this:

[126] https://nodejs.org/api/globals.html#globals_global

```
const articleDetails = {
    title: global.chance.sentence({ words: 3 }),
    description: global.chance.sentence({ words: 7 }),
    body: global.chance.paragraph({ sentences: 4 }),
    tags: [global.chance.word(), global.chance.word()]
};
```

That wraps up all our work for the Editor page. Form inputs are one of the biggest features of a website that are prone to bugs, so having good test coverage for them is crucial.

2.6.11 Chapter Challenge

We have a test for how to create an article, but what about editing one? Write a new set of tests to validate you can edit an article. Ensure that:

- The fields populate with the existing article contents
- The article page shows the updated contents after saving the edits

Hint: You can use the same `Editor.page.js` file as it's actually the exact same page as the "New Article" one (just prefilled with the contents of the article to edit).

2.7 Managing User Sessions

2.7.1 Thinking Through a New Type of Test

Last chapter, we wrote assertions to check for content that we submitted through our Editor page.

The test flow was:

1. Generate data
2. Submit data through website
3. Validate data on next page matched data generated in step 1

This worked well because we knew what data to match against because we just used it. Now let's look at a situation where we're tasked with validating data that isn't something we just submitted: the homepage on our demo site.

On the homepage, it lists out the latest articles from all users. How do we write a test that validates it has the "latest" article shown first? Won't that content change on a regular basis as people create new articles?

This is the challenge we're going to tackle in this chapter. We'll get to an answer in a little bit, but first there is some setup work to do.

We already have a very basic homepage test set up, which we wrote back in Chapter 2.1 (it's the `navigation.js` file). We need to get rid of it!

Why? Well, the two things it validates (page title and link functionality) aren't all that useful in terms of functional testing. While it would be helpful to know that our page title is incorrect, it's likely not worth the cost of test upkeep. Every title change will require an update to our test, and if the title is wrong... well, it's not that bad of a thing.

For the links, that's likely better handled through a specific tool that checks for dead links, like the W3C's link checker[127]. It'll run much faster and be a lot easier to set up than writing tests for every single link on our site.

So, let's just delete that entire `navigation.js` file and get ready to set up a new homepage test file with a brand new page object. We'll start with the page object, saved in a new `Home.page.js` file in our `pageObjects` folder:

[127] https://validator.w3.org/checklink

```
const Generic = require('./Generic.page');

class Home extends Generic {
    constructor() {
        super('./');
    }
}

module.exports = Home;
```

This matches the other page objects we've created in that we extend the Generic page object and define our page URL, using `./` to represent a URL without any suffix (so just the `baseUrl` alone). Now let's get our test going. The first test we'll write will validate the page has loaded correctly. How do we want to do this?

In our `editor` test, we checked that certain form fields existed. What's on our homepage that we can validate?

Well, several components:

- The site header (site title and global navigation)
- A 'banner' with the site name and description
- The 'global feed' (and 'your feed' tab when logged in)
- The 'popular tags' section
- The site footer

While the site header should appear on every page, we haven't validated its existence in any of our tests yet (aside from the navigation test that we just deleted). But, I'm not a huge fan of running this sort of validation.

If the header is missing from your homepage, you're going to notice before any automation is run. It's such an important part of the site that it's likely to go missing.

Also, the code for the header of the site likely won't change that frequently. Redesigns of the main navigation are often limited due to the impact these changes have. So you're going to be writing a test that will be run hundreds, likely thousands, of times for code that is never updated.

Now, on the other side of the argument, header content can be much more unique than what our demo site has going on. A basic example would be the header containing some sort of "notification" functionality. You definitely should have tests that validate those notifications are working as expected.

But should you write "notification" tests for every page of your site? I say that's major overkill. I would instead go with a strategy of testing the notification system on a very simple page (to avoid introducing unneeded complexity to the system), and assume it works across the site. Sure, you're

risking the system being broken on a specific page (or set of pages), but you're also not wasting hours of time on redundant testing.

Overall, a decision like this should be made by weighing the following:

- How important is the functionality?
- How much effort does it take to write the tests?
- How much time will it consume keeping the tests up to date?

Given all that, I do want to demonstrate how we can go about including site-wide functionality in our tests. So for the sake of example, we're going to test that our homepage has the following items:

- Site header (The 'bar' across the top of the page)
- Site footer (The 'bar' at the bottom of the page)
- Site navigation (The links in the site header)

We'll use a standard set of `toBeExisting` checks in our test file:

test/specs/home.js

```
const Home = require('../pageObjects/Home.page');

const home = new Home();

describe('Homepage', function () {
    before(function () {
        // load the page
        home.load();
    });

    it('should load properly', function () {
        // check that top nav/footer exist
        expect(home.$siteHeader).toBeExisting();
        expect(home.$siteFooter).toBeExisting();
        expect(home.$siteNav).toBeExisting();
    });
});
```

Now, normally, we'd add these element references to our "`home`" page object. But since these are elements that will be on every page, it's smarter to add them to our Generic page object.

We can do this in a manner similar to how we've added a generic 'load' method on that same object:

test/pageObjects/Generic.page.js

```
const { URL } = require('url');

class Generic {
    constructor(url) {
        this.url = url;

        // store the fullUrl by combining specific page url with WDIO base url
        // using NodeJS URL utility
        this.fullUrl = new URL(url, browser.options.baseUrl);
    }
    load() {
        browser.url(this.url);
    }
    get $siteHeader () { return $('[data-qa-id="site-header"]'); }
    get $siteNav () { return $('[data-qa-id="site-nav"]'); }
    get $siteFooter () { return $('[data-qa-id="site-footer"]'); }
}

module.exports = Generic;
```

By including these references in the Generic page object, anything that extends off of it will have those references (hence `home.$siteHeader` in the test file). This is useful for sharing common page template details across all your files.

2.7.2 Checking Those Feeds

In my list of "components on the homepage," I mentioned that there are two 'feeds' available. The 'Global' feed, which shows up for all visitors, and a specialized feed for users who are logged in. Let's add these to our test suite.

Right now, we're only going to validate that the tab appears (and not the content inside the tab, we'll talk about that soon though). So our test will be a single line of code that validates against the 'feed' tabs:

```
it('should only show the global feed tab', function () {
    expect(home.feedTabsText).toEqual(['Global Feed']);
})
```

What's `feedTabsText`? Well, it will be an array containing the text content of the tab titles. There should be two possible outcomes:

- Anonymous user: `['Global Feed']`
- Logged-in user: `['Your Feed', 'Global Feed']`

To get the text of the tabs in an array format, we need to get the tab elements themselves. Since we've already taken a similar approach before with the 'tags' on the article page, let's skip right to what the code will look like:

```
get $$feedTabs () { return $$('[data-qa-id="feed-tabs"] [data-qa-type="feed-tab"]') }
get feedTabsText () { return this.$$feedTabs.map($tab => $tab.getText().trim()); }
```

Add those two lines to your `Home.page.js` file, along with the updates to the homepage test file. And while you're in that test file, update it to include the fact that these tests are written for the "anonymous" view. Let's do that by updating our test file to have two nested `describe` blocks, one for the Anonymous view, and the other for when you're logged in:

test/specs/home.js

```
const Home = require('../pageObjects/Home.page');

const home = new Home();

describe('Homepage', function () {
    describe('Anonymous', function () {
        before(function () {
            // load the page
            home.load();
        });

        it('should load properly', function () {
            // check that top nav/footer exist
            expect(home.$siteHeader).toBeExisting();
            expect(home.$siteFooter).toBeExisting();
            expect(home.$siteNav).toBeExisting();
        });

        it('should only show the global feed tab', function () {
            expect(home.feedTabsText).toEqual(['Global Feed']);
        });
    });

    describe('Logged In', function () {
        // we'll fill this out soon
    });
});
```

To be fair, we could separate these two files instead. I'm keeping them together for simplicity sake, but feel free to separate them if you wish.

2.7.3 Session Management

For our logged in tests, we should, you know, log in. Let's copy over what we've done for our editor test to get us into a logged in state:

test/specs/home.js

```
const Home = require('../pageObjects/Home.page');
const Auth = require('../pageObjects/Auth.page');
const { user1 } = require('../fixtures/users');

const home = new Home();
const auth = new Auth();

describe('Homepage', function () {
    // ... anonymous tests hidden for brevity

    describe('Logged In', function () {
        before(function () {
            auth.load();
            auth.login(user1);

            home.load();
        });
    });
});
```

And then add our test that checks the tabs are correct:

```
it('should show both feed tabs', function () {
    expect(home.feedTabsText).toEqual(['Your Feed', 'Global Feed']);
});
```

Now, if you were to run this test file, everything would pass (hopefully). But there's a big problem with it. WebdriverIO (really, it's Mocha behind the scenes), runs tests from the top of the file to the bottom. So tests at the top get executed before tests at the bottom. Thus, our test timeline would look like this:

- Load the homepage

- Run `Anonymous - should load properly` test
- Run `Anonymous - should only show the global feed tab` test
- Login
- Load the homepage
- Run `Logged In - should show both feed tabs` test

If we were to rearrange our code so that the 'logged in' tests were at the top of the file, we'd end up running our 'anonymous view' tests from a 'logged in' state. That would cause our feed tab assertion to fail, as it would show both tabs instead of just the 'global' one.

Why is that? Well, WebdriverIO runs each test file in a separate instance. But the individual tests in each file share that instance. So any 'session' data you set in a test will be set in all subsequent tests for that file.

In our original setup with the anonymous view first, this didn't matter, as the 'logged in' tests ran after everything else. But if someone were to rearrange the order for some reason, they'd be left scratching their head as to why these tests are now failing.

It'd be smart to avoid this, so we should clean up after ourselves (as anyone with good manners would agree). To accomplish that, we'll utilize the `after` hook inside our test file to logout:

```
describe('Logged In', function () {
    before(function () {
        auth.load();
        auth.login(user1);

        home.load();
    });
    after(function () {
        // run steps to log out
    });
});
```

With that set up, let's decide how we want to log out. I say "decide" because we have a couple of options:

- Logout via the website (click the 'settings' main nav link, then the 'logout' button)
- Programmatically delete the user information from our browser session

The advantage of the second choice is that it's much faster, as we don't have to wait for any page loads (which is the slowest part of functional testing). The disadvantage is that you have to figure out how to delete that user information, and it's going to be different for every site you test.

That's okay, because I'm going to let you cheat here and tell you that the user logging information is stored inside the browser's Local Storage system[128]. This is a data store built into browsers where developers can host data specific to that user; in this case, the unique `token` used as identification for the user.

Now just how do we get rid of this thing? Well, there is a `clearLocalStorage` command[129] available in WebdriverIO, but unfortunately it's not fully supported (e.g., the `wdio-chromedriver-service` throws an error with that command, although running it through a Selenium server seems to work fine).

As a workaround, we're going to directly call the browser's `Local Storage` API[130]. We'll accomplish this by using the `execute` command[131], which is a very useful utility to know about.

2.7.4 Executing Arbitrary Scripts in the Browser

WebdriverIO offers a lot of functionality out of the box, but there are times when our test requires running code inside the browser itself. The `execute` function allows us to run JavaScript code on the page we're testing, from inside the browser. It works by injecting our snippet of code into the page for execution, similar to how you would run any other JavaScript function from inside a browser window.

This is extremely useful for those cases where Selenium doesn't have a built in option for the functionality you need. For example, you can use it to change the DOM of the page, adding or removing a class on an HTML element. Or you could use it to check that a certain file has been loaded on the page, or even that a specific JavaScript variable has been set.

Or in our case, to use the browser's Local Storage API to clear our session information. Here's a basic example of execute in action:

```
browser.execute(function () {
    window.alert('This call is coming from inside the browser!');
});
```

Wait, where did the `window` object come from? Is that part of WebdriverIO? Nope!

If you're unfamiliar with JavaScript in the browser, `window` is a global object that the browser maintains. It's similar to how WebdriverIO gives you a `browser` object to use. We're able to access it, because the function we defined in our `execute` call is run inside the browser.

It can be confusing to understand where our scripts are being run, so I'm going to add two 'log' statements; one will run inside our node.js instance, and the other will run inside the browser:

[128] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API
[129] https://webdriver.io/docs/api/jsonwp.html#clearlocalstorage
[130] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage
[131] https://webdriver.io/docs/api/browser/execute.html

```
console.log('outside the browser.'); // this is run by Node.js
browser.execute(function () {
    console.log('Inside the browser.'); // this is run by the browser
    window.alert('This call is coming from inside the browser!');
});
```

If you were to run that script, you'd see the 'outside the browser' output, but not the second log. Where did it go?

Well, despite both lines using similar `console.log` code, they're run in two entirely different ways. Both Node.js and browsers support a common `console` API, so the function names and formats are the same between the two. This makes it much easier for developers to switch from writing browser code to Node.js code, and vice versa.

So when our second `console.log` line is run, it's logging to the browser's console, not the console we're using to run our test. That's why we don't see it anywhere.

Getting back to our original goal, which is to clear the browser's Local Storage data, we can do that by sending `window.localStorage.clear()` to the browser via `execute`. Here's how that looks in the context of our Home test file:

```
describe('Homepage', function () {
    // ... anonymous tests hidden for brevity

    describe('Logged In', function () {
        before(function () {
            auth.load();
            auth.login(user1);

            home.load();
        });
        // ... some tests here ...
        after(function () {
            browser.execute(function () {
                window.localStorage.clear();
            });
        });
    });
});
```

That's some pretty useful functionality that we'll likely use again in other tests. How about we move it to a function in our Auth page object?

```
class Auth extends Generic {
    // ... other code removed for brevity ...
    clearSession() {
        browser.execute(function () {
            window.localStorage.clear();
        });
    }
}
```

And updating our `after` test hook:

```
after(function () {
    auth.clearSession();
});
```

Now we can organize our Home test file however we'd like, knowing that the session information won't be shared between our suites.

2.7.5 Speeding up the login via APIs

If we can clear the session information to emulate a user logging out, can we programmatically add session data in? Like, perhaps an auth token…

Why yes, yes we can. And we will!

What I'm proposing is that we skip logging in via the UI and instead log in via an API call.

Now, let me say that taking this approach is in no way necessary. The code will also vary greatly between different sites, sometimes being more complicated than it's worth. I'm mentioning it here for two reasons:

- This will likely speed up almost every test you run by three to five seconds
- It makes for a good starting example for a more complicated technique we'll be trying later

We're going to do the following with our code:

- Send an API request to the website's authentication service
- The service will respond with a 'token', which is what we deleted in the previous section
- We take that 'token' and inject it into our Local Storage, using the same `execute` command from before

Way back at the start of this book we talked about APIs. We learned that WebdriverIO works by sending API requests to a WebDriver server and handles the responses. We're going to do a very similar thing.

If you're unfamiliar with the concept of an API, there are many useful explanations already out there, including:

- What is an API?[132]
- What is an API Gateway?[133]
- An Introduction to APIs[134]
- What exactly IS an API?[135]
- Introduction to web APIs[136]

The concept of an API is important to grasp, so I recommend spending some time learning about the general structure before diving into this lesson. Or maybe read through this section, then read any of those links I provided to help cement what's going on.

I'm going to move forward assuming you're knowledgeable about the basics of an API. I'm also going to give you the following "insider" information about authentication on our website:

- Authentication requests are sent to the `users/login` endpoint
- They are sent as a POST request, with a JSON payload of `{ user: { email: 'email@address.com', password: 'hunter2' } }`
- In the JSON response, you'll receive a user object with a `token` property
- That `token` is added to the browser's Local Storage under the `id_token` key

These details vary from site to site, so keep that in mind if you want to try this technique out on a different website. You'll need to figure out how to authenticate and where to store your token.

WebdriverIO does not have a built-in way to make POST API requests, but that's okay because it really doesn't need to. Instead, we can use one of the many popular Node.js HTTP request libraries out there, including:

- Got[137] (what we'll use)
- node-fetch[138]
- Axios[139]
- Request[140]

[132] https://dev.to/mercier_remi/what-is-an-api-4ao9
[133] https://dev.to/bearer/what-is-an-api-gateway-42i6
[134] https://zapier.com/learn/apis/chapter-1-introduction-to-apis/
[135] https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f
[136] https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction
[137] https://github.com/sindresorhus/got
[138] https://github.com/bitinn/node-fetch
[139] https://github.com/axios/axios
[140] https://github.com/request/request

Whichever one you use is fine. Figure out what fits best for your needs (or just try one and see how it goes). We're going to use 'Got', which is what WebdriverIO uses behind the scenes for its own needs.

First thing, install Got as a dependency in your local project folder by running `npm i got`.

With that, let's keep things organized by creating a new file called `Api.js`, and storing it in a new folder called `utils` in our project root.

In that file, a new class will be defined which will store all of our API interactions. The purpose of this book is not to teach you how to write an API client, so I'm going to let you cheat and just give you the code needed for it:

utils/Api.js

```
const got = require('got');

class Api {
    constructor(prefixUrl) {
        this.client = got.extend({
            prefixUrl,
            responseType: 'json'
        });
    }

    getAuthToken({ email, password }) {
        return this.client
            .post('/users/login', {
                json: { user: { email, password } }
            })
            .then((response) => response.body.user.token);
    }
}

module.exports = Api;
```

A brief overview of the file is this:

- It exports an "Api" class, which can be constructed with a `prefixUrl` which should match our root API endpoint ('http://localhost:3000')
- The class has a `getAuthToken`, which will accept user credentials and return a promise which will resolve with the token data

Here's an example of a basic Node.js script using this new file:

temporary-example.js

```
const Api = require('./utils/Api');

(async () => {
    const api = new Api('http://localhost:3000/api/');
    const token = await api.getAuthToken({
        email: 'demowdio@learnwebdriverio.com',
        password: 'wdiodemo'
    });
    console.log(token);
})();
```

Now we need to convert this script to something WebdriverIO can use. Back in our `home.js` test file, we're going to update the 'Logged In' `before` function. For reference, this is what the function currently runs:

```
before(function () {
    auth.load();
    auth.login(user1);

    home.load();
});
```

Instead, we need to:

1. Require our Api file
2. Instantiate a new Api with the correct url: `const api = new Api('http://localhost:3000/api/');`
3. Call the `getAuthToken` function via `browser.call` (since that function is promise-based and we need to wait for it to complete)
4. Load the page in an unauthorized state (we can't set `localStorage` until after we have the page loaded)
5. Set the token via `browser.execute` and the `window.localStorage.setItem` function[141]
6. Reload the homepage, now that we have our token set

Here's what all of that looks like:

[141] https://developer.mozilla.org/en-US/docs/Web/API/Storage/setItem

```
before(function () {
    // 1. Require our Api file
    const Api = require('../../utils/Api');

    // 2. Instantiate a new Api instance with the url of our Api
    const api = new Api('http://localhost:3000/api/');

    // 3. Call the `getAuthToken` function
    const token = browser.call(() => {
        return api.getAuthToken(user1);
    });

    // 4. Load the page in an unauthorized state
    home.load();

    // 5. Set the token
    browser.execute((browserToken) => {
        window.localStorage.setItem('id_token', browserToken);
    }, token);

    // 6. Reload the homepage
    home.load();
});
```

Now, if you were to run this test, you'd see everything load basically in the same way, except it would skip the login page step. Because we're retrieving our token via the API, we don't need to go through the login page steps.

The next step is to move all of that API/token business to a common area, so that we can re-use it in our other files. We could add it to our Auth page object (similar to how we added the `clearSession` function), but I want to propose an alternative location: the `wdio.conf.js` file.

2.7.6 Global Custom Commands

Start things off in the configuration file by adding the `const Api = require('./utils/Api');` require statement to the top of the file. This can go before or after the `Chance` code.

Then, scoot on down to our `before` hook, which if you recall from earlier, we used to store a global reference to our Chance instance.

After the `global.chance = new Chance(process.env.SEED + specs[0]);` line in that hook, let's store the instantiated API object to the global scope:

```
global.api = new Api('http://localhost:3000/api/');
```

That's nice, in case we need to reference this API instance in other tests. But it still doesn't get the token retrieval and storage functionality out of our test.

What we're going to do to achieve this is use the `browser.addCommand` command[142]. If you want to extend the browser instance with your own set of commands, this is the method to use.

To use it, pass in the name of the command you want to define, and then a function containing the code the command should run (which can be anything).

We're going to define a new `loginViaApi` command with the following code:

```
browser.addCommand('loginViaApi', function (user) {
    const token = browser.call(() => {
        return global.api.getAuthToken(user);
    });

    // load the base page so we can set the token
    browser.url('./');

    // inject the auth token
    browser.execute((browserToken) => {
        window.localStorage.setItem('id_token', browserToken);
    }, token);
});
```

Add that right after your `global.api` line and save the configuration file. Then jump back over to `home.js` and update the `before` hook we were previously editing to use this new command:

```
before(function () {
    browser.loginViaApi(user1);

    home.load();
})
```

That sure is a lot less code in our hook than before! And we can use this command anywhere in our tests, without having to specifically load it, making our tests a bit cleaner than before.

The `addCommand` command is something I don't use often, but it definitely comes in handy when necessary.

To recap, using an API to login does take extra knowledge about how API's work, including specifics surrounding how the API and authentication of the site you're testing works. But, it can really come in handy, not only for authenticating users, but adding, editing and deleting data, which is something we'll take a deeper look into soon.

[142] https://webdriver.io/docs/customcommands.html

2.7.7 Chapter Challenge

- Rather than use the `addCommand` command, update the `Auth` page object to have the `loginViaApi` functionality
- Update the `editor.js` test to use this new function (but not the `login.js` file, since we actually want to test the login page)

2.8 Creating Page Components

2.8.1 Checking the Active Tab

We've got our login and logout functionality worked out for our tests. Now it's time to make use of that effort.

By default, both logged in and out views should show the 'global' tab by default. Let's write a test that checks which tab is "active" on page load:

```
it('should default to showing the "global" feed', function () {
    // get all tabs with an 'active' class, check only one returns with correct text
})
```

How do we know which is the active tab? We have a few choices:

- get the tab with the 'active' class and validate the text is correct
- get the 'Global Feed' tab and check that the 'class' attribute has an 'active' class
- get all tabs with an 'active' class, check that only one returns, validate the text

Looking over the options, I prefer the last one as it includes the chance that there could be multiple 'active' tabs (which would definitely be a bug).

Now, using our previous knowledge, let's update the `Home` page object to include our new `activeFeedTabText` element reference:

test/pageObjects/Home.page.js

```
const Generic = require('./Generic.page');

class Home extends Generic {
    constructor() {
        super('./');
    }
    get $$feedTabs() {
        return $$('[data-qa-id="feed-tabs"] [data-qa-type="feed-tab"]');
    }
    get feedTabsText() {
        return this.$$feedTabs.map(($tab) => $tab.getText().trim());
    }
    get activeFeedTabText() {
```

```
        return this.$$feedTabs.$$('.active').map(($tab) => $tab.getText().trim());
    }
}

module.exports = Home;
```

It's similar to the existing `feedTabsText` property, except it only gets element text with an `active` class.

Now it's time to use it in our test. Usage is similar to the previous test we wrote checking that both tabs exist:

```
it('should default to showing the global feed', function () {
    // get all tabs with an 'active' class, check only one returns with correct text
    expect(home.activeFeedTabText).toEqual(['Global Feed']);
})
```

With that written, there's a little bit of optimization we can do to our Home page object (I'm all about optimization if you haven't caught on). In our `feedTabsText` and `activeFeedTabText` references, we use the same `map` function that we used in our Article page object.

Maybe it's a good idea to move that to a utility file? One could argue that this is a bit of an over-optimization, but let's continue, even just to demonstrate the concept of utility files.

So, in a new file, we'll define an exported object with a single function property that acts as our mapped method:

utils/functions.js

```
module.exports = {
    getTrimmedText: $el => $el.getText().trim()
}
```

We'll save this in the `utils` folder we created for our API file, naming it `functions.js`. To use this in our page object, we need to require it, then pass it to our `map` function:

```
const Generic = require('./Generic.page');
const { getTrimmedText } = require('../utils/functions');

class Home extends Generic {
    constructor() {
        super('./');
    }
    get $feedsContainer() {
        return $('[data-qa-id="feed-tabs"]');
    }
    get $$feedTabs() {
        return this.$feedsContainer.$$('[data-qa-type="feed-tab"]');
    }
    get feedTabsText() {
        return this.$$feedTabs.map(getTrimmedText);
    }
    get activeFeedTabText() {
        return this.$feedsContainer
            .$$('[data-qa-type="feed-tab"] .active')
            .map(getTrimmedText);
    }
}

module.exports = new Home();
```

Again, this isn't entirely necessary, but it can reduce some of the duplication in your code.

2.8.2 Switching Tabs

So we've validated that both tabs show, and that `Global` is the default tab being displayed. Now let's check that we can switch between the two tabs. We can do that with our existing page object, by finding the tab with the text we want and clicking on it:

```
it('should let you switch between global and personal feeds', function () {
    // get the tab that has `Your Feed` as text
    const yourFeedTab = home.$$feedTabs.find(
        ($tab) => $tab.getText() === 'Your Feed'
    );
    // Click on that tab
    yourFeedTab.click();
    // Validate 'active' tab is now `Your Feed`
    expect(home.activeFeedTabText).toEqual(['Your Feed']);
    // get the tab that has `Global` as text
    const globalFeedTab = home.$$feedTabs.find(
        ($tab) => $tab.getText() === 'Global Feed'
    );
    // Click on that tab
    globalFeedTab.click();
    // validate again
    expect(home.activeFeedTabText).toEqual(['Global Feed']);
});
```

There are a couple problems with this code. First off, it's a bit verbose to have to find the tab text each time we want to switch. It would be nice not to have to do that each time we switch tabs.

And there's a bug in this code (it's a tricky one too). If you run these tests with limited bandwidth, it will click the tab and then check for the text, but because of network delays, the tab will not have actually switched yet. So you'll get a failure that the active tab text isn't correct.

We could add a custom wait statement after each click, but that would clog up an already busy test case. Instead, let's look at some alternative approaches.

One option is to get all the tabs, map their text to an object, and reference it through that. For example:

```
const tabs = home.$$feedTabs.reduce(function (accTabs, tab) {
    accTabs[tab.getText()] = tab;

    return accTabs;
}, {});

// click on 'Your feed' tab
tabs['Your Feed'].click();
// // validate 'active' tabs are correct
expect(home.activeFeedTabText).toEqual(['Your Feed']);
// // click 'Global' tab
tabs['Global Feed'].click();
```

```
// // validate again
expect(home.activeFeedTabText).toEqual(['Global Feed']);
```

The above is definitely a more complex coding example (understanding the `reduce` function is worthy of a PhD in my opinion), so I won't dive into details here as it revolves more around JavaScript coding than testing. It also has its own drawback, especially if two tabs share the same name. I simply wanted to mention it as an alternative.

How about another option? We'll create a `clickTab` function on the Home page object, which looks like:

test/pageObjects/Home.page.js

```
const Generic = require('./Generic.page');
const { getTrimmedText } = require('../../utils/functions');

class Home extends Generic {
    constructor() {
        super('./');
    }
    get $feedsContainer() {
        return $('[data-qa-id="feed-tabs"]');
    }
    get $$feedTabs() {
        return this.$feedsContainer.$$('[data-qa-type="feed-tab"]');
    }
    get feedTabsText() {
        return this.$$feedTabs.map(getTrimmedText);
    }
    get activeFeedTabText() {
        return this.$feedsContainer
            .$$('[data-qa-type="feed-tab"] .active')
            .map(getTrimmedText);
    }

    clickTab(tabText) {
        const tabToClick = this.$$feedTabs.find(
            ($tab) => $tab.getText() === tabText
        );
        tabToClick.click();
        browser.waitUntil(
            () => {
                return this.activeFeedTabText[0] === tabText;
            },
```

```
            { timeoutMsg: 'Active tab text never switched to desired text' }
        );
    }
}

module.exports = new Home();
```

Notice the custom `waitUntil` function we added after the tab click. This tells WebdriverIO to wait until the active tab text has switched to our desired value. Again, this is necessary to avoid test failures when we run these on a slower network.

Now, to use it in our test:

```
it('should let you switch between global and personal feeds', function () {
    // click on 'Your feed' tab
    home.clickTab('Your Feed');
    // // validate 'active' tabs are correct
    expect(home.activeFeedTabText).toEqual(['Your Feed']);
    // click 'Global' tab
    home.clickTab('Global Feed');
    // validate again
    expect(home.activeFeedTabText).toEqual(['Global Feed']);
});
```

Let's move forward with this solution, as it's a good mix of providing a utility method and not complicating our code too much.

2.8.3 Dynamic Content Testing

Now that we've validated the proper tabs exist on the page, and that we can switch between them, let's focus now on what those tabs contain.

We'll split our testing four ways:

General feed tests (functionality common in 'Global', 'Personal', and 'Tag' tabs)

- should order by most recent article first
- should allow you to follow a user

Global feed tests

- should show articles from all users

Personal Feed

- should show only articles from followed users

Tag feed

- should only show when on a 'Tag' page
- should show all articles with that tag

The first set of tests we'll write are for the general/common functionality. It may not seem like it, but it's going to be a bit tricky to undertake. To explain, examine that first requirement: "should order by most recent article first."

How are we to validate this? Certainly we know how to get the content of the article preview, but how do we know that this article is the most recent? If this is a live site we're testing, the most recent article will be changing on a regular basis. Even if we were to create an article and try to quickly test that it's 'first,' we still risk someone else creating a newer article in the time it takes to execute our script.

On top of that, this sort of "random" failure is likely to drive you mad, as it would be difficult to reproduce consistently, making it frustrating to debug. This is one of the many troubles with running automation against a production/live site and why I generally shy away from it. Plus, all these fake articles we're creating can be confusing for real users.

Instead, I'm going to move forward with the assumption that we're testing on the "development" server. We still may run into this same issue as testers create fake articles, but it's less likely. Regardless, I prefer alternative approaches than what I've described.

Those approaches include the following:

A. Deploy the test site with existing content that's always the same (this is good if you can easily deploy your site, so you can create individual servers for your specific test run)
B. Load content via a direct database script before running tests
C. Inject the content via an API call before/during test run
D. Use a tool like Fiddler to intercept the requests being made by the browser to inject 'fake' content for the site to use
E. Use a "sandbox" type site that allows you to test components individually outside of normal use
F. In your script, read the database/API to get the value that should be displayed on the page (I generally dislike this strategy though)

That's a fair number of options; too many to cover in detail. When choosing from that list, think about what best fits your needs.

For example, if you go with option D, you're no longer validating that the database/server is responding with the correct data. You're now only checking that the front-end of the site is making the proper request and handling the response in the right way. This may be beneficial though, as it reduces the risks of "overtesting."

Think about this… if your database/API is broken for any reason (or not yet written), you're unable to run your tests even though the front-end is still ready for them. By "mocking" out this interface, you avoid that scenario. Also, the API should be independently tested and not rely on functional browser tests to validate its behavior.

There's a lot of wiggle room out there on what features WebdriverIO should be responsible for testing. After many years, I've found it best to focus on browser functionality/interaction. It's far less likely that a page title fails to appear on a page than it is to have some advanced JavaScript-based interaction broken. It's also a lot more difficult to test that JavaScript functionality, as it requires clicks and form entries and all sorts of actions.

That's the sort of work I prefer to have my tests focus on. So I'm definitely interested in seeing how we can take shortcuts in less necessary areas to focus on where WebdriverIO shines.

So, with all that said, for now, we're going to take the first approach, which is to have the site have specific content already on it for us to check against (we will look at option C later on).

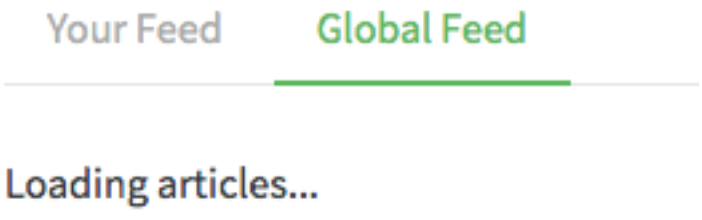
2.8.4 Waiting to Load

One thing I did not mention with these tabs is that they're loaded "asynchronously" from the page itself. The HTML and text inside the feed is retrieved separately from the rest of the page, meaning that it may not be available on page load.

This is beneficial for the user, as it allows them to get new tab content without having to reload the entire page. It's for us though as we have to write specific code to handle this unknown wait time.

This won't be too difficult, and we'll use our old friend `waitForExist()` to make it happen.

When the page first loads, the following loading text is shown:



"Loading Articles…" text screenshot

That text is removed from the page once the content is loaded, meaning we can use it's absence as an indicator that the content has completed loading.

All that's great, except the name of the command we're going to use is `waitForExist()`, not `waitForNotExist()`. How do we achieve the opposite effect of the name of the command?

Well, all `waitFor` commands allow you to switch the condition to the opposite by passing in a boolean flag via an `options` object.

Here's the usage information from the WebdriverIO documentation:

```
$(selector).waitForExist({ timeout, reverse, timeoutMsg, interval })
```

`timeout` represents the time in milliseconds to wait. We don't want to change that from the default, so we'll leave it out.

`reverse` should be set to `true` if it instead waits for the selector to not match any elements (which is what we want).

`timeoutMsg` and `interval` will be omitted, since we don't need to change their defaults.

Bringing that all together, we can wait for the "Loading Articles..." text to disappear with the following code:

```
$('div=Loading Articles...').waitForExist({ reverse: true });
```

A quick note: It can be tricky to figure out the HTML of these loading indicators because they often disappear before you can inspect them. Thankfully, the built-in browser development tools often have a way to artificially throttle your connection speed (think back to how we did this in Chapter 2.3.3 with `browser.throttle`). Here's how to do that in Chrome[143].

So with the network throttled to a slow speed, you can reload the page, then switch the setting to 'offline' and have an indefinite time to inspect the HTML. When you've figured out what you need, set the condition back to 'online' and continue with your work.

While I used the actual text in the previous example, there's also a `data-qa-id` attribute on that div, which is what we'll use for our updated page object:

[143] https://developers.google.com/web/tools/chrome-devtools/network/reference#throttling

test/pageObjects/Home.page.js

```
class Home extends Generic {
    // ... some contents omitted for brevity ...

    get $articleLoadingIndicator() {
        return $('[data-qa-id="article-loading-indicator"]');
    }

    clickTab(tabText) {
        const tabToClick = this.$$feedTabs.find(
            ($tab) => $tab.getText() === tabText
        );
        tabToClick.click();
        browser.waitUntil(
            () => {
                return this.activeFeedTabText[0] === tabText;
            },
            { timeoutMsg: 'Active tab text never switched to desired text' }
        );
        this.$articleLoadingIndicator.waitForExist({ reverse: true });
    }
}
```

Now we can ensure that when we switch tabs, we also wait for the feed content to load. This is good, but we need to do this exact same thing when the page initially loads as well. But before we get there, let's take a quick detour...

2.8.5 Custom Page Components

I can't recall the first time I came across the idea of a UI component. HTML had always defined a bare set of components, mainly revolving around form fields. These components encapsulated a common set of functionality that could be customized on a case-by-case basis.

But at some point (probably very early on), developers weren't happy with the limitations posed by the standard set of HTML elements. Instead, they hoped to extend the bare minimum and create their own, more functional (and better designed) interfaces.

Early projects used the jQuery plugin system which allowed developers to construct common interfaces such as tabs, tooltips, dropdown menus, and more. Those individual efforts were soon morphed into the jQuery UI library, providing devs with a set of widgets for implementation across their sites.

In the past ten years, we've seen major growth in this arena through the concept of Pattern Libraries (alternatively known as UI Libraries, Design Systems, or Style Guides, although the pedantic readers will enjoy arguing the differences between them). JavaScript frameworks have adapted and now all major libraries support the concept of components; in fact, a major effort is underway to standardize all libraries under the umbrella of "Web Components."

None of this is necessarily important to testing, but I do find it interesting.

Can we follow this pattern in our tests, building out not page objects, but page components? Can we mimic the ease-of-use and reusability of these patterns? The Magic 8-Ball I just shook says, "Without a doubt." (Well, actually, it said, "Reply hazy. Try again." about five times until I made up the response I wanted.)

Let's start with the basic idea of what a page component should be: a simple reference to a page element. From that, we'll extend it to our individual needs:

test/pageObjects/components/Component.js

```
/*
 * This is the most basic form of a component.
 * There isn't much to it, but it's a good starting point
 */

class Component {
    // 'selector' is the selector for the main element that all other selectors for \
that component are based upon.
    // This can be the main container element, or an input element that has a specif\
ic id
    // 'selector' can be either a selector, or a wdio element
    constructor(selector, options = {}) {
        this.selector = selector;
        this.options = options;
    }

    // The $origin element can be used to move up via xpath:
    // const $originParent = this.$origin.$('..');
    get $origin() {
        // allow 'selector' to be a WDIO element
        return typeof this.selector === 'string' ? $(this.selector) : this.selector;
    }
}

module.exports = Component;
```

This component is what we'll build all our other components off of. It's not entirely necessary, but I find it helpful to have a common starting point. The `$origin` element reference can be changed as

you see fit; some people prefer `$element` or `$container`. I use 'origin' because it's a more flexible name than the other two examples.

Assuming you want to follow me down this path of an uber-generic page component, save this file as `Component.js` to a new `components` folder inside your `page_objects` folder.

2.8.6 The Feed Page Component

Now comes the time to build out our first real component, which will define the structure of a "Feed".

In it, we'll load the component file we just created, define two article references (the 'articles' inside our feed, and the loading indicator we talked about earlier), and finally include a `waitForLoad` function which we moved from the Home page object.

Here's how it looks:

test/pageObjects/components/Feed.js

```
const Component = require('./Component');

class Feed extends Component {
    get $$articles() {
        return this.$origin.$$('[data-qa-type="article-preview"]');
    }
    get $articleLoadingIndicator() {
        return this.$origin.$('[data-qa-id="article-loading-indicator"]');
    }
    waitForLoad() {
        this.$articleLoadingIndicator.waitForExist({ reverse: true });
    }
}
```

A few things:

1. We don't need to define a constructor, as that's handled by the Component class we've extended.
2. In `$$articles` and `$articleLoadingIndicator`, we reference `this.$origin`, which comes from the parent `Component` class.
3. We've included a `waitForLoad` function, which is copied over from our previous Homepage work. We'll delete that Homepage code in a second.

With this new class (saved to `Feed.js` in that same `components` folder), we can update our Home page object to reference it.

First, include the `Feed` file at the top:

```
const Feed = require('./components/Feed');
```

Then, let's add a new reference in our object, getting the currently shown 'feed':

```
get currentFeed () { return new Feed('[data-qa-type="article-list"]'); }
```

How does this selector work? Well, if you were to inspect the HTML on the page as you switch tabs, you'd notice that the code for the article list is completely switched out every time a tab is clicked, meaning that there's only ever one element that matches the `'[data-qa-type="article-list"]'` selector. That's why we can use that single selector to find our feed, even though they're technically two different items.

With our `currentFeed` defined, we can use it and the `waitForLoad` function on the Feed class, to make our Home page wait until the feed is loaded to continue.

The Generic page object has a `load` function already defined that runs the `browser.url` command. We need to extend it in our Home page object to add this code: `this.currentFeed.waitForLoad();`

Say we were to define a new `load` function in the Home page object:

```
class Home extends Generic {
    load () {
        this.currentFeed.waitForLoad();
    }
}
```

When `home.load();` is called in our test, it would fail. Instead of loading the URL properly, it would just wait for the current feed to load. That's because we've overwritten the Generic load function with our own.

To work around this, in our Home `load` function, we want to first call the Generic `load` function, then run our custom code. Fortunately, thanks to JavaScript's Class functionality, that's as simple as calling `super.load();`.

If you recall from our constructor, we do something similar by calling `super()` itself. This is a similar, but different, way of doing this. `super()` calls the constructor function of the parent class, whereas `super.load()` calls the `load` function of the parent class. You can use this with any function on the parent class.

Going back to the Home page object, we'll add that `super.load()` call to our `load` function, and also add the `this.currentFeed.waitForLoad();` call to our `clickTab` function so that it also waits for the feed content to load before continuing on:

test/pageObjects/Home.page.js

```
load () {
    super.load();
    this.currentFeed.waitForLoad();
}

clickTab (tabText) {
    const tabToClick = this.$$feedTabs.find(
        ($tab) => $tab.getText() === tabText
    );
    tabToClick.click();
    browser.waitUntil(
        () => {
            return this.activeFeedTabText[0] === tabText;
        },
        { timeoutMsg: 'Active tab text never switched to desired text' }
    );
    this.currentFeed.waitForLoad();
}
```

2.8.7 Testing Feeds

Now that we've got all the waiting in place that we want, it's time to check the contents of our feeds. We're going to focus just on the `Personal Feed` data for now, as it's a simpler example than the Global feed.

In the `Logged In` test suite, we'll define a new child `describe` block for the 'Personal Feed'.

```
describe('Personal Feed', function () {

});
```

In the `before` hook, we'll make sure we're on the `Your Feed` tab.

```
before(function () {
    // ensure we're on the personal feed tab
    if (home.activeFeedTabText !== 'Your Feed') {
        home.clickTab('Your Feed');
    }
});
```

Then, we'll write our test that checks that the number of items in the feed is correct (which will be 1, from a special account we follow that only has a single article). We use the 'toHaveChildren' check[144], which looks at the amount of fetched elements using `$$` command and compares them to the number passed in.

```
it('should show most recent articles from people you follow', function () {
    expect(home.currentFeed.$$articles).toHaveChildren(1);
});
```

Overall, it looks like:

```
describe('Homepage', function () {
    describe('Logged In', function () {
        before(function () {
            browser.loginViaApi(user1);

            home.load();
        });

        // tests hidden here for brevity

        describe('Personal Feed', function () {
            before(function () {
                // ensure we're on the personal feed tab
                if (home.activeFeedTabText !== 'Your Feed') {
                    home.clickTab('Your Feed');
                }
            });

            it('should show most recent articles from people you follow', function (\
) {
                expect(home.currentFeed.$$articles).toHaveChildren(1);
            });
        });
```

[144] https://webdriver.io/docs/api/expect.html#tohavechildren

```
        after(function () {
            auth.clearSession();
        });
    });

    // tests here hidden for brevity
});
```

2.8.8 Testing Feed Content

The next question on my mind: is "How can we validate that the content, not just the length, is correct?" It could be pulling in the wrong article, and checking only the number of articles listed would give us a false positive result.

We should do due diligence and check the content of the article preview, which takes us to the next component we'll create. A 'Feed' contains a number of Article Previews. Each preview has the following data:

- Author
- Date Published
- 'Favorite' button
- Title
- Preview
- 'Read More' Link
- Tags

Let's use this outline to create a new page component, similar to how we setup the 'Feed' component:

test/pageObjects/components/ArticlePreview.js

```
const Component = require('./Component');

class ArticlePreview extends Component {
    get $author () { return this.$origin.$('[data-qa-type="author-name"]') }
    get $date () { return this.$origin.$('[data-qa-type="article-date"]') }
    get $title () { return this.$origin.$('[data-qa-type="preview-title"]') }
    get $description() {
        return this.$origin.$('[data-qa-type="preview-description"]');
    }
    get $readMoreLink () { return this.$origin.$('[data-qa-type="preview-link"]') }
    get $favorite () { return this.$origin.$('[data-qa-type="article-favorite"]') }
```

```
    get $$tags () { return this.$origin.$$('[data-qa-type="tag-list"] li') }
}

module.exports = ArticlePreview;
```

We'll save this as `ArticlePreview.js` in our `pageObjects/components` folder. Then, in our `Feed.js` file in that same folder, we'll require it and define a list of feeds that we can reference in our tests.

To do this, we'll do two things:

1. Get all the container elements of each element preview using the `$$` command (which we already have defined)
2. Map through those elements to generate dynamic `ArticlePreview` classes, similar to how we made our `currentFeed` reference

Here's what the updated file looks like:

test/pageObjects/components/Feed.js

```
const Component = require('./Component');
const ArticlePreview = require('./ArticlePreview');

class Feed extends Component {
    get $$articles () { return this.$origin.$$('[data-qa-type="article-preview"]') }
    get articles() {
        return this.$$articles.map(($article) => new ArticlePreview($article));
    }
    get $articleLoadingIndicator() {
        return this.$origin.$('[data-qa-id="article-loading-indicator"]');
    }
    waitForLoad () {
        this.$articleLoadingIndicator.waitForExist({ reverse: true });
    }
}

module.exports = Feed;
```

Finally, we'll add a new test that checks the title of the first article in the 'Personal' feed:

```
it('should show most recent article first', function () {
    expect(home.currentFeed.articles[0].$title).toHaveText('An Article');
});
```

So, to recap:

- We have a 'Feed' component, which is used to more easily access the current feed being shown in the page
- Inside that component, we have a reference to multiple 'ArticlePreview' components, which give us easier access to the various parts of an article preview

2.8.9 Validating Multiple Properties Easier

It's nice to know the title is correct, as it's a positive sign that the correct data is being displayed. But if we have reference to the full article details, why not check against that as well?

While we could copy/paste the test code from before, it would be more helpful to create a helper function that gets all the article information on the page for us.

In our `ArticlePreview` class, let's create a `getDetails` function that retrieves the text in a test-friendly format for asserting against:

```
getDetails() {
    // this is important, because `getText` will return an empty string if the
    // article preview is outside the browser's viewport
    this.$origin.scrollIntoView(true);
    return {
        author: this.$author.getText().trim(),
        date: this.$date.getText().trim(),
        title: this.$title.getText().trim(),
        description: this.$description.getText().trim(),
        tags: this.$$tags.map($tag => $tag.getText().trim())
    }
}
```

Then, we'll update our test to reference the new `getDetails` function, validating the details. The assertion we'll use is `toMatchObject`[145], which is part of ExpectJS and allows us to compare two objects with each other (here, it's the object returned from `getDetails`, with an object of expected data on the page).

[145] https://jestjs.io/docs/en/expect#tomatchobjectobject

```
it('should show most recent article first', function () {
    const firstArticleDetails = home.currentFeed.articles[0].getDetails();
    expect(firstArticleDetails).toMatchObject({
        author: 'singlearticleuser',
        date: 'May 1, 2020',
        title: 'An Article',
        description: 'A Single Article',
        tags: []
    });
});
```

As you see, we can dynamically create multiple page components on the fly for our use. Now, this is a very powerful feature, but understand that it comes with a cost. Creating each dynamic component does take a half-second of time, so if you have hundreds of components to create, perhaps this isn't the best approach.

Also, it complicates the code a bit. Anyone new to the codebase would need to look into what the 'Feed' and 'ArticlePreview' components do in order to really understand the test. As will all things in code, by trying to reduce the complexity, you end up introducing new complexities. Sometimes you just can't win. :D

2.8.10 Chapter Challenge

Use the API to get the tags to be shown on the 'Popular Tags' section on the homepage.

With that data, add a test that asserts those tags are properly shown.

Hint: The endpoint to get the tags is be `/tags/` (e.g., https://conduit-api.learnwebdriverio.com/api/tags/)

2.9 Generating Data for Testing via APIs

In our last set of tests, I mentioned how difficult it can be to test the content on a page if that content isn't static. If we're testing a shared website, someone may change the content we're meaning to validate against, causing our tests to fail.

While much of this pain can be avoided by running all tests against a local server, we will still be faced with some problems:

1. Getting a local server set up is not always an option
2. Testing on staging/shared development servers will likely be required, as we need to know that the code deployed to it correctly
3. When you have a large test suite, the content created during test runs can cause conflicts between your tests

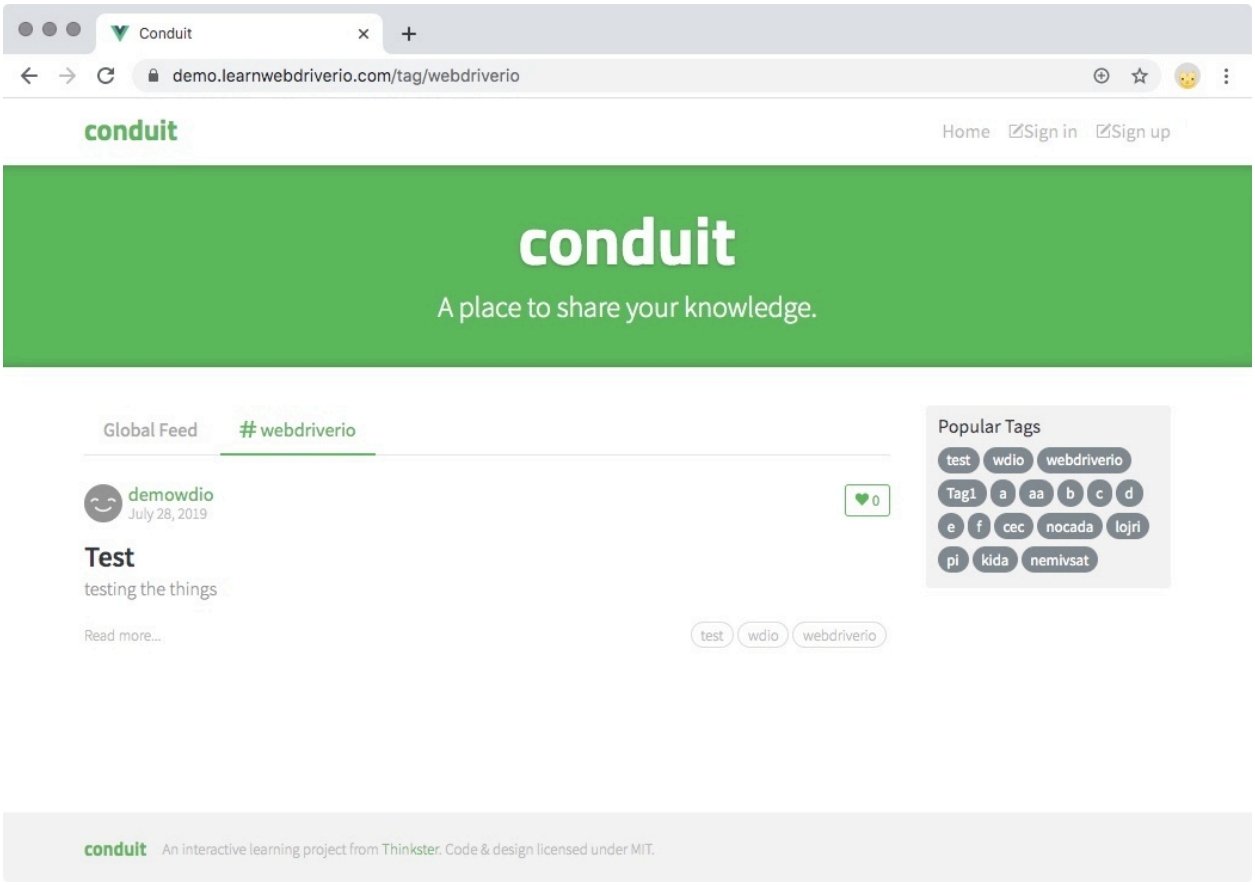
I can't say I have a solution for all of this, but I do have some strategies to address it. To recap the options I provided in the last chapter, here are some options:

A. Deploy the test site with existing content that's always the same (this is good if you can easily deploy your site, so you can create individual servers for your specific test run).
B. Load content via a direct database script before running tests.
C. Inject the content via an API call before/during test run.
D. Use a tool like Fiddler to intercept the requests being made by the browser to inject 'fake' content for the site to use.
E. Use a "sandbox" type site that allows you to test components individually outside of normal use.
F. In your script, read the database/API to get the value that should be displayed on the page. I generally dislike this strategy though.

In this chapter, we're going to look at Option C, which has involves creating content before our test run via the website's API. We did something similar to this for our API login strategy, but this time we'll be creating new data, instead of retrieving information for an already existing user.

2.9.1 The 'Tag' Page

When an article is created, a user can add 'tags' to that article to mark it as a specific type of post. Then, later on, a user can limit their feed results by a specific tag. That's the page we're going to be testing now.



Example 'WebdriverIO' Tag page

To represent this page in our code, let's create a new "tag" page. If you look at this page, you'll notice that it's almost exactly the same as the homepage, just with an extra tab for the specific tag. What we can do is, instead of duplicating the Home page object, just extend off of it.

test/pageObjects/Tag.page.js

```
const Home = require('./Home.page');

class Tag extends Home {
    constructor(tagName) {
        super('./tag/' + tagName);
    }
}

module.exports = Tag;
```

In this file, we require the 'Home' page object, create a new 'Tag' class that extends home, then update our constructor to build out the proper URL.

There's one issue with this code though. In our Home page object, the constructor hard-codes the URL passed in: `super('./')`

In order for our Tag page object to work, we need to update this code to allow for a different URL to be passed in. Thanks to optional parameters in JavaScript, making this update requires only a couple of small changes in our `Home.page.js` file:

```
// set default URL to homepage, but allow for a custom URL to be passed in
constructor (url = './') {
    super(url);
}
```

So now we set a default URL to use, but also allow for any child class to define their own.

With our page objects updated, let's move on to our next step.

2.9.2 Creating a New Article/Tag via the API

There isn't a singular "Tag" page, but rather, it's a type of page. To view one, you must have a tag defined to search with. We could do this for our test in multiple ways:

1. Have a predefined tag, similar to how we have a predefined user
2. Go to the "Create Article" page and generate a new article with a specific tag via UI automation
3. Create an article with a specific tag via the API

Option #1 is decent, but requires that all test sites (e.g., local and staging) have the same data. It's also difficult to document, and someone may change this data not knowing they're breaking tests.

Option #2 is the slowest approach and burdens our test code with unnecessary steps. I avoid this option whenever possible.

Option #3 is the method we're going to try now. By generating the data "just in time," we ensure that it's always ready to go for our test.

To tackle this, we need to update our API file to add a new `createArticle` function. This function will accept user and article details (the post will be attributed to that specific user).

Here's what the updated `Api.js` file will look like:

utils/Api.js

```
const axios = require('axios');

class Api {
    constructor(baseURL) {
        this.api = axios.create({
            baseURL,
            Accept: 'application/json, text/plain, */*',
            'Content-Type': 'application/json'
        });
    }

    getAuthToken({ email, password }) {
        return this.api
            .post('/users/login', { user: { email, password } })
            .then((res) => res.data.user.token);
    }

    async createArticle(user, details) {
        const token = await this.getAuthToken(user);
        const response = await this.client.post('articles', {
            json: {
                article: details
            },
            headers: {
                Authorization: `Token ${token}`
            }
        });
        return response.body.article;
    }
}

module.exports = Api;
```

We define our `createArticle` function as an `async` function, as it will use the `await` keyword to more easily handle the promise returned from the `getAuthToken` function.

But why do we need to use `async` here instead of letting WebdriverIO handle it? Well, there are limitations to the WebdriverIO's `sync` module, namely that it only works for WebdriverIO-specific functions. Any asynchronous functions that we create outside WebdriverIO need to be handled in more traditional ways (e.g., async/await).

Moving forward, the first step in our function is to get the proper Authorization token to use when we make our API post. We'll await the response from the internal `getAuthToken` function call.

Once that returns, we make a new request, this time POSTing to the `articles` API endpoint. We pass along a JSON body following the format of the API (a single `article` property with a JSON representation – `details` – of the desired article). We also pass along authorization headers using the token we received. This will let the API know which user to associate this article with.

Finally, we'll return just the `body.article` from the response. This isn't necessary, but will make our test code a little bit cleaner.

2.9.3 Writing Our Tag Page Test

Speaking of the test code, now it's time to write our test. It will consist of the following:

Before hook:

1. Dynamically create a new article (including the tag we're going to test) using ChanceJS
2. Post that article to the API using the function we just wrote
3. Create a new Tag page object based on the tag we just generated
4. Load the page for our tests

Tests:

- check that we're on the tag tab
- check that it loaded only articles with that tag (should be just the one since it's a unique tag)
- check that it loaded the correct article preview details

We've covered everything we need to know to build out these tests, so let's jump right into the code:

test/specs/tag.js

```
const { user1 } = require('../fixtures/users');
const Tag = require('../pageObjects/Tag.page');

describe('Tag Feed', function () {
    let articleDetails, tagName, tagPage, articleResponse;

    before(function () {
        articleDetails = {
            title: chance.sentence({ words: 3 }),
            description: chance.sentence({ words: 7 }),
            body: chance.paragraph({ sentences: 2 }),
            tagList: [chance.word({ length: 30 })]
        };
```

```
        tagName = articleDetails.tagList[0];

        // create the article we need to get the specific tag
        articleResponse = browser.call(() => {
            return global.api.createArticle(user1, articleDetails);
        });

        tagPage = new Tag(tagName);

        // load the page
        tagPage.load();
    });

    it('should have tag tab', function () {
        // check that we're on the tag tab
        expect(tagPage.feeds.$$activeFeedTabs[0]).toHaveText(tagName, {
            trim: true
        });
    });

    it('should load only articles for that tag', function () {
        expect(tagPage.currentFeed.$$articles).toHaveChildren(1);
    });

    it('should load correct article preview details', function () {
        const firstArticleDetails = tagPage.currentFeed.articles[0].getDetails();

        expect(firstArticleDetails).toMatchObject({
            title: articleDetails.title,
            description: articleDetails.description
        });
    });
});
```

Hopefully this is starting to look familiar by now. The one idea we haven't covered before is dynamically generating the `Tag` page in our `before` hook. But there's nothing too special about this approach; we're merely changing where we create it.

2.9.4 Cleaning Up Our Mess

One drawback to this "on the fly" approach to creating test data is that every test run generates new content. Over time, after many test runs, our website will be flooded with random articles. This isn't always a negative, as it can inadvertently help test that the site works well even with a lot of data on it. But, there are likely much better ways to validate that.

With that said, let's look at how we can "clean up" after our test, by making an API call to delete the article we used for our test. Back in the `Api.js` file, let's add a new function that takes the `slug` of an article and makes a request to delete it. We also still need to pass in the user so that we have a proper auth token in order for our request to delete.

utils/Api.js

```
async deleteArticle(user, slug) {
    const token = await this.getAuthToken(user);
    return this.client.delete(`articles/${slug}`, {
        headers: { Authorization: `Token ${token}` }
    });
}
```

Again, I won't get into the details on the Conduit API much, because it's specific to this instance. The exact code you'll need to write will depend heavily on the website that you're testing. We also could do better than retrieving the user token every time we need it (e.g., store it somewhere), but again, I didn't want to get too bogged down with technical implementations here.

Now to call this `deleteArticle` function. We only want to delete the article after all our tests are completed, so we'll make that call in the `after` Mocha hook:

```
after(function () {
    browser.call(() => {
        return global.api.deleteArticle(user1, articleResponse.slug);
    });
});
```

Note that we get the generated slug from the API response, which was stored back in our `before` hook.

With that added, we now have a set of tests that generate the needed data via API calls, then delete that data after running, to ensure no artifacts are left. This is just one small example, but hopefully you find this pattern of data management useful.

2.9.5 Exercises

Create a set of tests for the user profile page (e.g., https://demo.learnwebdriverio.com/@singlearticleuser/). These tests should be responsible for creating a unique profile with a article that shows on the "My Article" feed.

The test should validate that:

- Username is shown at top of the page
- You can follow/unfollow the user if you are logged in
- Two tabs are shown: 'My Articles' and 'Favorited Articles'
  - The content for those tabs is correct
  - It's up to you whether you want to write the API code to "favorite" an article

3.0 Chapter Challenge Code Solutions

Each chapter includes a challenge or two for you to tackle on your own. But it's nice to check your work, so the following are solutions for each challenge.

Chapter 2.1: Assertions

In your existing test, add a check that validates that when you click on the "Conduit" logo in the top left of the site, it returns you to the homepage.

In the test we're adding to, we're on the login page. Now we need to click that logo, which can be done using the link text selector like we used for the 'Sign Up' button: `$('=conduit')`

This does need to be lowercase, because that's how it is on the page itself.

Next, we want to write our assertion. The page title doesn't actually change between pages, so that's not a good option (we could still be on the sign up page and it would pass since the title for that page is the same as the homepage).

Instead, we'll just validate that we're no longer on the login url, by using the `.not` flip in our assertion. All together, our updated file looks like so:

test/specs/navigation.js

```
describe('Homepage', function () {
    it('should load properly', function () {
        // load the page
        browser.url('./');

        // Get the title of the homepage, should be 'Conduit'
        expect(browser).toHaveTitle('Conduit');

        // Click the 'Sign in' navigation link
        $('=Sign in').click();

        // Get the URL of the sign in page. It should include 'login'
        expect(browser).toHaveUrl('/login', { containing: true });

        $('=conduit').click();

        expect(browser).not.toHaveUrl('/login', { containing: true });
```

```
    });
});
```

Chapter 2.2: Selectors

Page Footer Text

The developers of the site handed us a gift with this one, placing a unique `class` on the containing element called `attribution`. We can use that to write a simple CSS selector of:`$('.attribution')`

All 'Tags' in the 'Popular Tags' sidebar

Similar to the previous challenge, we have a class name of `sidebar` that we can use for this selector. We can combine it with a descendent selector using the `a` tag to target the tags: `$$('.sidebar a')`

Why not use the `tag-list` class instead? Think about what other elements might have this class name... If you look closely at the main feed, you'll see each article preview also has a `tag-list` class used. If we wanted to still use it, we'd need to be more specific with our selector and limit it to the `sidebar` container: `$$('.sidebar .tag-list a')`

All 'Tags' in the 'Popular Tags' sidebar (using 'Popular Tags' header text). Hint: use `text()` and `following-sibling`

This last one was the trickiest by far, and requires some XPath magic. First, we need to get the heading element by the text inside it. We can do this two ways:

- Using a full text match: `$('//p[text()="Popular Tags"]')`
- Using a partial text match: `$('//p[contains(text(), "Popular Tags")]')`

From there, we need to navigate over to the links. We can use the `following-sibling` modifier to move over to the `div` (`following-sibling::div`). From there, we can get all the links using the `a` tag.

All together, this is what your selector may look like:

```
$$('//p[contains(text(), "Popular Tags")]/following-sibling::div//a')
```

Chapter 2.3: Waiting Another Way

Update the 'waitFor' code to instead use `waitForDisplayed`

This is a pretty simple change, just replacing `Exist` with `Displayed` in your command name: `$signIn.waitForDisplayed({ reverse: true });`. Everything else stays the same.

Instead of waiting for the "Sign in" button to stop existing, try waiting for the "Your Feed" tab to appear.

We can use a simple text selector to target the "Your Feed" tab/link, then use the `waitForDisplayed` command to wait for it to appear: `$('=Your Feed').waitForDisplayed();`.

The full file would look like:

test/specs/login.js

```
describe('Login Page', function () {
    it('should let you log in', function () {
        browser.url('./login');

        $('input[type="email"]').setValue('demo@learnwebdriverio.com');
        $('input[type="password"]').setValue('wdiodemo');

        $('button*=Sign in').click();

        $('=Your Feed').waitForDisplayed();

        expect($('.error-messages li')).not.toBeExisting();
    });
});
```

Chapter 2.4: Registration Tests

Write a set of tests for the register account page

This is a pretty lengthy exercise, so I'm not going to detail the steps involved (that's what the chapter was for). Here is one possible solution though:

test/pageObjects/Register.page.js

```
const Generic = require('./Generic.page');

class Register extends Generic {
    constructor() {
        super('./register');
    }

    get $username () { return $('input[name="email"]'); }
    get $email () { return $('input[name="email"]'); }
    get $password () { return $('input[type="password"]'); }
```

```
    get $register () { return $('button*=Sign in'); }
    get $errorMessages () { return $('.error-messages li'); }

    submit({ username, email, password }) {
        this.$username.setValue(username);
        this.$email.setValue(email);
        this.$password.setValue(password);

        this.$register.click();

        // wait until either the register button is gone or an error has appeared
        browser.waitUntil(
            () => {
                const buttonExists = this.$signUpButton.isExisting();
                const errorExists = this.$errorMessages.isExisting();

                return !buttonExists || errorExists;
            },
            {
                timeoutMsg:
                    'The "Sign Up" button is not gone and an error never appeared'
            }
        );
    }
}

module.exports = Register;
```

test/specs/register.js

```
const Register = require('../pageObjects/Register.page');

const register = new Register();

const username = 'newuser';
const email = 'newuser@learnwebdriverio.com';
const password = 'hunter2';

describe('Register Page', function () {
    beforeEach(function () {
        register.load();
    });
```

```
    it('should let you register', function () {
        // use a timestamp to generate "unique" username/email every test run
        const uniqueUsername = username + Date.now();

        register.submitForm({
            username: uniqueUsername,
            email: uniqueUsername + '@learnwebdriverio.com',
            password
        });

        // Get the URL of the page, which should no longer include 'register'
        expect(browser).not.toHaveUrl(register.url.href);
    });

    it('should error with a missing username', function () {
        register.submitForm({
            username: '',
            email,
            password
        });

        expect(register.$errorMessages).toHaveText(`username can't be blank`);
    });

    it('should error with an already registered username', function () {
        register.submitForm({
            username: 'demouser',
            email,
            password
        });

        expect(register.$errorMessages).toHaveText(`username is already taken.`);
    });

    it('should error with a missing email', function () {
        register.submitForm({
            username,
            email: '',
            password
        });

        expect(register.$errorMessages).toHaveText(`email can't be blank`);
    });
```

```
    it('should error with an already registered email', function () {
        register.submitForm({
            username,
            email: 'demo@learnwebdriverio.com',
            password
        });

        expect(register.$errorMessages).toHaveText(`email is already taken.`);
    });

    it('should error with an invalid email', function () {
        register.submitForm({
            username,
            email: 'invalid',
            password
        });

        expect(register.$errorMessages).toHaveText(`email is invalid`);
    });

    // uh-oh, this test fails because there's a bug on the register page!
    it.skip('should error with a missing password', function () {
        register.submitForm({
            username,
            email,
            password: ''
        });

        expect(register.$errorMessages).toHaveText(`password can't be blank`);
    });
});
```

Chapter 2.5: More Sharing

Move the Auth's `load` function call to be inside the `login` function.

Since we need to have the Login page opened in order to login, we can ensure we're there by calling the `login.open()` function in our login function. You can also remove it from the `editor.js` test file too!

First, we'll delete the `beforeEach` function from our `login.js` test file (since it won't be needed anymore). Then, update the login function in our page object to have `this.load()` at the start. Note that we don't use `login.load`, since we're inside the login page element.

Your updated function will look like:

test/pageObjects/Auth.page.js

```
login({ email, password }) {
    this.load();

    this.$email.setValue(email);
    this.$password.setValue(password);

    this.$signIn.click();

    // wait until either the sign in button is gone or an error has appeared
    browser.waitUntil(
        () => {
            const signInExists = this.$signIn.isExisting();
            const errorExists = this.$errorMessages.isExisting();

            return !signInExists || errorExists;
        },
        {
            timoutMsg:
                'The "Sign in" button still exists and an error never appeared'
        }
    );
}
```

Add element references to the Generic page for common items like the site navigation and page footer

You can add element references in the exact same way as other page objects:

test/pageObjects/Generic.page.js

```
const { URL } = require('url');

class Generic {
    constructor (path) {
        this.path = path;

        // store the url by combining specific page path with WDIO base url
        // using the NodeJS URL utility
        this.url = new URL(path, browser.options.baseUrl);
    }
    load() {
        browser.url(this.path);
    }
    get $siteHeader () { return $('[data-qa-id="site-header"]'); }
    get $siteNav () { return $('[data-qa-id="site-nav"]'); }
    get $siteFooter () { return $('[data-qa-id="site-footer"]'); }
}

module.exports = Generic;
```

Then, in any Page Object based off of the Generic one, reference it as if it were already there: `editor.$siteNav.isExisting();`

Chapter 2.6: Validating 'Edit Article' Functionality

First, we need an article to edit. The pre-existing "Demo Article" is a good option, although it would be useful to create a unique article just for testing purposes. Which one you choose is up to you, but I went ahead and went with creating a new one.

I also went with setting up and entirely new test file. You could try and include these tests in your `editor.js` file, but for simplicity sake I used a separate one.

Also, to keep things simpler, I skipped making an edits to the tags. While I could have deleting the existing tags and added new ones, that code was a little more complicated than what we've covered so far, so I left it off.

We need to add two new references in our `Editor.page.js` file, for the "Tags List" items that are created when adding tags. That code is:

test/pageObjects/Editor.page.js

```
get $$tagsListItems () { return $$('.tag-list .tag-pill'); }
get tagsListItems () { return this.$$tagsListItems.map($tag => $tag.getText()); }
```

Then, for the full set of tests, I created a new file called `articleEdit.js` and added the following code to it:

test/specs/articleEdit.js

```
const Auth = require('../pageObjects/Auth.page');
const Editor = require('../pageObjects/Editor.page');
const Article = require('../pageObjects/Article.page');
const { user1 } = require('../fixtures/users');

const auth = new Auth();
const editor = new Editor();
const article = new Article();

describe('Existing Article Editor', function () {
    let articleDetails;

    before(function () {
        auth.login(user1);

        // create an article for editing
        editor.load();

        articleDetails = {
            title: global.chance.sentence({ words: 1 }),
            description: global.chance.sentence({ words: 2 }),
            body: global.chance.paragraph({ sentences: 1 }),
            tags: [global.chance.word(), global.chance.word()]
        };

        editor.submitArticle(articleDetails);
        article.waitForLoad();

        article.$edit.click();
        editor.$title.waitForDisplayed();
    });

    it('should populate fields with article details', function () {
        expect(browser).toHaveUrl('editor', { containing: true });
```

```
        expect(editor.$title).toHaveValue(articleDetails.title);
        expect(editor.$description).toHaveValue(articleDetails.description);
        expect(editor.$body).toHaveValue(articleDetails.body);
        expect(editor.tagsListItems).toEqual(articleDetails.tags);
    });

    it('should update article after editing', function () {
        const updatedDetails = {
            title: global.chance.sentence({ words: 1 }),
            description: global.chance.sentence({ words: 2 }),
            body: global.chance.paragraph({ sentences: 1 }),
            tags: [] // don't add any new tags
        };

        editor.submitArticle(updatedDetails);

        article.waitForLoad();

        expect(article.$title).toHaveText(updatedDetails.title);
        expect(article.$body).toHaveText(updatedDetails.body);

        // tags should be same since we didn't change them
        expect(article.tags).toEqual(articleDetails.tags);
    });
});
```

To be honest, this isn't the prettiest of code samples, but it gets the job done!

Chapter 2.7: Move 'loginViaApi' Command to the Auth Page Object

There are two steps to this change:

- Move the function from the `wdio.conf.js` file `before` hook to a new function inside the Auth.page.js file:

test/pageObjects/Auth.page.js

```
class Auth extends Generic {
    // ... other Auth page object stuff goes here

    loginViaApi(user) {
        const token = browser.call(() => {
            return global.api.getAuthToken(user);
        });

        // load the base page so we can set the token
        browser.url('./');

        // inject the auth token
        browser.execute((browserToken) => {
            window.localStorage.setItem('id_token', browserToken);
        }, token);
    }
}
```

- Update references in the test to switch `browser.loginViaApi` to `Auth.loginViaApi`.

Personally, I prefer the custom command style, but I did want to demonstrate another way you can make this happen.

Chapter 2.8: Validating the "Popular Tags" Block

Use the API to get the tags to be shown on the 'Popular Tags' section on the homepage.

In your `Api.js` file, add a new function to make the API call. You don't need to worry about the auth token, as it's not required.

utils/Api.js

```
async getTags() {
    const tagsResponse = await this.client.get(`tags`);
    // return just the array, not the full response
    return tagsResponse.tags;
}
```

Next, add the 'popularTags' parts to your Home page object:

test/pageObjects/Home.page.js

```
get $$popularTags() {
    return $$('//p[text()="Popular Tags"]/following-sibling::div/a');
}
get popularTags () { return this.$$popularTags.map(getTrimmedText) }
```

The selector is a bit complex, due to the lack of a `data-qa` attribute on either the container or the tags themselves. So, we rely on some xPath to grab that "Popular Tags" heading, then get the tags from that point.

With that, add a test that asserts those tags are properly shown. In your `home.js` test file, in the `Anonymous` section, add the following:

test/specs/home.js

```
it('should show "Popular Tags"', function () {
    // get the tags that should exist from the API
    const apiTags = browser.call(() => {
        return global.api.getTags();
    });

    expect(home.popularTags).toEqual(apiTags);
});
```

Note that this test does have the chance it will fail, if the tags haven't loaded on the page yet. I'm not as concerned here, because we already have a wait for the articles to load, plus we have our own delay in waiting for the API to respond with the expected tags.

It's also difficult to add a proper wait here, since there is no "loading" indicator in that popular tags box. Sometimes you just can't be perfect :)

Chapter 2.9: Public User Profile Page Tests

Create a set of tests for the user profile page (e.g., https://demo.learnwebdriverio.com/@singlearticleuser/). These tests should be responsible for creating a unique profile with a article that shows on the "My Article" feed.

The test should validate that:

- Username is shown at top of the page
- You can follow/unfollow the user
- Two tabs are shown: 'My Articles' and 'Favorited Articles'

- The content for those tabs is correct
- It's up to you whether you want to write the API code to have a "favorited" article

Create a new Profile page object:

test/pageObjects/Profile.page.js

```
const Generic = require('./Generic.page');
const Feeds = require('./components/Feeds');

class Profile extends Generic {
    constructor(username) {
        super('./@' + username);
    }

    get feeds () { return new Feeds('[data-qa-id="feed-tabs"]'); }
    get followToggle () { return $('[data-qa-id="follow-toggle"]'); }
    get $username () { return $('[data-qa-id="profile-username"]'); }

    load () {
        super.load();
        this.feeds.currentFeed.waitForLoad();
    }
}

module.exports = Profile;
```

As you can see, the Profile page uses a new "Feeds" Component, which has shared 'feeds' functionality between it and the Home/Tags page:

test/pageObjects/Feeds.page.js

```
const Component = require('./Component');
const { getTrimmedText } = require('../../utils/functions')
const Feed = require('./Feed');

class Feeds extends Component {
    get $feedsContainer () { return $(this.selector); }
    get $$feedTabs() {
        return this.$feedsContainer.$$('[data-qa-type="feed-tab"]');
    }
    get feedTabsText () { return this.$$feedTabs.map(getTrimmedText); }
    get activeFeedTabText() {
        return this.$feedsContainer
```

```
            .$$('[data-qa-type="feed-tab"] .active')
            .map(getTrimmedText);
    }
    get currentFeed () { return new Feed('[data-qa-type="article-list"]'); }

    clickTab(tabText) {
        const tabToClick = this.$$feedTabs.find(
            ($tab) => $tab.getText() === tabText
        );
        tabToClick.click();
        browser.waitUntil(
            () => {
                return this.activeFeedTabText[0] === tabText;
            },
            { timeoutMsg: 'Active tab text never switched to desired text' }
        );
        this.currentFeed.waitForLoad();
    }
}

module.exports = Feeds;
```

Since we're using a new shared component, let's replace that code from the Home Page Object:

test/pageObjects/Home.page.js

```
const Generic = require('./Generic.page');
const { getTrimmedText } = require('../utils/functions')
const Feeds = require('./components/Feeds');

class Home extends Generic {
    constructor (url = './') {
        super(url)
    }
    get feeds () { return new Feeds('[data-qa-id="feed-tabs"]'); }
    get $$popularTags() {
        return $$('//p[text()="Popular Tags"]/following-sibling::div/a');
    }
    get popularTags () { return this.$$popularTags.map(getTrimmedText) }

    load () {
        super.load();
        this.feeds.currentFeed.waitForLoad();
    }
```

```
}

module.exports = Home;
```

Now we need to update the Home and Tag tests to use this new component path:

test/specs/home.js

```
const Home = require('../pageObjects/Home.page');
const Auth = require('../pageObjects/Auth.page');
const { user1 } = require('../fixtures/users');

const home = new Home();
const auth = new Auth();

describe('Homepage', function () {
    describe('Logged In', function () {
        before(function () {
            browser.loginViaApi(user1);

            home.load();
        });

        it('should show both feed tabs', function () {
            expect(home.feeds.feedTabsText).toEqual(['Your Feed', 'Global Feed']);
        });

        it('should default to showing the global feed', function () {
            // get all tabs with an 'active' class,
            // check only one returns with correct text
            expect(home.feeds.activeFeedTabText).toEqual(['Global Feed']);
        });

        it('should let you switch between global and personal feeds', function () {
            // click on 'Your feed' tab
            home.feeds.clickTab('Your Feed');
            // validate 'active' tabs are correct
            expect(home.feeds.activeFeedTabText).toEqual(['Your Feed']);
            // click 'Global' tab
            home.feeds.clickTab('Global Feed');
            // validate again
            expect(home.feeds.activeFeedTabText).toEqual(['Global Feed']);
            // click on 'Your feed' tab
            home.feeds.clickTab('Your Feed');
```

```
            // validate 'active' tabs are correct
            expect(home.feeds.activeFeedTabText).toEqual(['Your Feed']);
        });

        describe('Personal Feed', function () {
            before(function () {
                // ensure we're on the personal feed tab
                if (home.feeds.activeFeedTabText !== 'Your Feed') {
                    home.feeds.clickTab('Your Feed');
                }
            });

            it('should show most recent articles from people you follow',
                function () {
                    expect(home.feeds.currentFeed.$$articles).toHaveChildren(1);
                }
            );

            it('should show most recent article first', function () {
                const firstArticleDetails = home.feeds.currentFeed.articles[0]
                    .getDetails();
                expect(firstArticleDetails).toMatchObject({
                    author: 'singlearticleuser',
                    date: 'May 1, 2020',
                    title: 'An Article',
                    description: 'A Single Article',
                    tags: []
                });
            });
        });

        after(function () {
            auth.clearSession();
        });
    });
    describe('Anonymous', function () {
        before(function () {
            // load the page
            home.load();
        });

        it('should load properly', function () {
            // check that top nav/footer exist
```

```
            expect(home.$siteHeader).toBeExisting();
            expect(home.$siteFooter).toBeExisting();
            expect(home.$siteNav).toBeExisting();
        });

        it('should only show the global feed tab', function () {
            expect(home.feeds.feedTabsText).toEqual(['Global Feed']);
        });

        it('should show "Popular Tags"', function () {
            // get the tags that should exist from the API
            const apiTags = browser.call(() => {
                return global.api.getTags();
            });

            expect(home.popularTags).toEqual(apiTags);
        });
    });
});
```

test/specs/tag.js

```
const { user1 } = require('../fixtures/users');
const Tag = require('../pageObjects/Tag.page');

describe('Tag Feed', function () {
    let articleDetails, tagName, tagPage, articleResponse;

    before(function () {
        articleDetails = {
            title: chance.sentence({ words: 3 }),
            description: chance.sentence({ words: 7 }),
            body: chance.paragraph({ sentences: 2 }),
            tagList: [chance.word({ length: 30 })]
        };

        tagName = articleDetails.tagList[0];

        // create the article we need to get the specific tag
        articleResponse = browser.call(() => {
            return global.api.createArticle(user1, articleDetails);
        });
```

```
        tagPage = new Tag(tagName);

        // load the page
        tagPage.load();
    });

    after(function () {
        browser.call(() => {
            return global.api.deleteArticle(user1, articleResponse.slug);
        });
    });

    it('should have tag tab', function () {
        // check that we're on the tag tab
        expect(tagPage.feeds.$$activeFeedTabs[0]).toHaveText(tagName, {
            trim: true
        });
    });
    it('should load only articles for that tag', function () {
        expect(tagPage.feeds.currentFeed.$$articles).toHaveChildren(1);
    });
    it('should load correct article preview details', function () {
        const firstArticleDetails = tagPage.feeds.currentFeed.articles[0]
            .getDetails();

        expect(firstArticleDetails).toMatchObject({
            title: articleDetails.title,
            description: articleDetails.description
        });
    });
});
```

Okay, with all that written, let's focus back on writing the actual tests.

In our profile test, we need to set up some data:

- A user profile to validate
- That user should have an article published for us to check out

Add the following function to the API util, allowing us to create a new user/profile via the API:

utils/Api.js

```
async createProfile(profile) {
    const response = await this.client.post('users', {
        json: {
            user: profile
        }
    });
    return response.body.user;
}
```

We need to make one small update to the `Feed.page.js` component, which is adding an element reference for the text that shows if no articles are available (i.e., 'No articles are here... yet.'). We'll use a text-based selector to match a div with that text. The line of code will be `get $noArticlesText () { return this.$origin.$('div=No articles are here... yet.'); }`.

Here's what the full file looks like:

test/pageObjects/components/Feed.js

```
const Component = require('./Component');
const ArticlePreview = require('./ArticlePreview');

class Feed extends Component {
    get $$articles () { return this.$origin.$$('[data-qa-type="article-preview"]') }
    get articles() {
        return this.$$articles.map(($article) => new ArticlePreview($article));
    }
    get $articleLoadingIndicator() {
        return this.$origin.$('[data-qa-id="article-loading-indicator"]');
    }
    get $noArticlesText() {
        return this.$origin.$('div=No articles are here... yet.');
    }
    waitForLoad () {
        this.$articleLoadingIndicator.waitForExist({ reverse: true });
    }
}

module.exports = Feed;
```

Now, we'll create a new `profile.js` test file, and call the needed API utils to create the user and article (along with sketching out the tests we want to write), then run our tests:

test/specs/profile.js

```
const Profile = require('../pageObjects/Profile.page');

describe('User Profile Page', function () {
    let profile, profileResponse, articleDetails;

    before(function () {
        profile = {
            email: `test+${Date.now()}@learnwebdriverio.com`,
            password: 'testwdio',
            username: chance.word({ length: 15 })
        };

        // create the user/profile we need to for our test
        profileResponse = browser.call(() => {
            return global.api.createProfile(profile);
        });

        // create a new article for the new user
        articleDetails = {
            title: chance.sentence({ words: 3 }),
            description: chance.sentence({ words: 7 }),
            body: chance.paragraph({ sentences: 2 })
        };
        // create the article we need to get the specific tag
        articleResponse = browser.call(() => {
            return global.api.createArticle(profile, articleDetails);
        });

        profilePage = new Profile(profile.username);

        // load the page
        profilePage.load();
    });

    after(function () {
        browser.call(() => {
            return global.api.deleteArticle(profile, articleResponse.slug);
        });
    });

    it('should show username at top of the page', function () {
        expect(profilePage.$username).toHaveText(profile.username);
```

```
    });

    it('should show username at top of the page', function () {
        expect(profilePage.$username).toHaveText(profile.username);
    });

    it('should allow you to follow and unfollow the user if you are logged in',
    function () {
        expect(profilePage.$followToggle).toHaveText(`Follow ${profile.username}`, {
            trim: true
        });

        profilePage.$followToggle.click();

        expect(profilePage.$followToggle).toHaveText(
            `Unfollow ${profile.username}`,
            { trim: true }
        );

        profilePage.$followToggle.click();

        expect(profilePage.$followToggle).toHaveText(`Follow ${profile.username}`, {
            trim: true
        });
    });

    it('should show "My Articles" and "Favorited Articles" tabs with correct content\
',
    function () {
        // check that the tabs are correct
        expect(profilePage.feeds.feedTabsText).toEqual([
            'My Articles',
            'Favorited Articles'
        ]);
        expect(profilePage.feeds.activeFeedTabText).toEqual(['My Articles']);

        // // check that the content for the 'My Articles' tab is correct
        expect(profilePage.feeds.currentFeed.$$articles).toHaveChildren(1);

        const articleDetails = profilePage.feeds.currentFeed.articles[0]
            .getDetails();

        // format created date to format of "August 1, 2020"
```

```
        const months = [
            'January',
            'February',
            'March',
            'April',
            'May',
            'June',
            'July',
            'August',
            'September',
            'October',
            'November',
            'December'
        ];
        const createdDate = new Date(articleResponse.createdAt);
        const formattedDate = `${
            months[createdDate.getMonth()]
        } ${createdDate.getDate()}, ${createdDate.getFullYear()}`;

        expect(articleDetails.author).toEqual(profile.username);
        expect(articleDetails.date).toEqual(formattedDate);
        expect(articleDetails.title).toEqual(articleResponse.title);
        expect(articleDetails.tags).toEqual(articleResponse.tagList);

        // switch tabs and run again
        profilePage.feeds.clickTab('Favorited Articles');

        expect(profilePage.feeds.currentFeed.$$articles).toHaveLength(0);
        expect(profilePage.feeds.currentFeed.$noArticlesText).toBeExisting();
    });
});
```

3.1 Final Thoughts

First off, if you've made it here, a huge thank you for your efforts. I really do appreciate the time you've taken to read what I have to say about WebdriverIO, and to learn the technology behind so many automated test suites available today. I hope the content has been informative, and helped you to evaluate new ways that you can write better tests.

While there is more content I'd love to cover (e.g., Visual Regression Testing), it simply won't fit within the bounds of this book. I do post regular articles to my blog, and videos to my Front-end Testing channel.[146], so check those out if you're looking to continue learning.

Although the book is "complete", I do welcome any and all feedback on this book. You can reach out through the contact form on my website[147], send a private note via the Leanpub contact form[148], or email me at kevin@learnwebdriverio.com. I really hope to hear from you (even just to say, "Hi!").

Finally, a huge thanks to the entire WebdriverIO community, but especially the maintainers and project committers who make WebdriverIO possible. Their contributions are used around the world by the community to make websites better, and I can't thank everyone who contributes enough.

I also want to give a special thanks to Michael Gilbert for his tireless efforts in editing this book. He has found countless typos and grammar mistakes and this book is much better because of it. Thank you so much!

I hope you've found this journey helpful and have learned new skills to put to use. I'm honored to have had you as a reader and thank you for taking the time to read through these chapters. And always remember, have fun testing!

[146] https://youtube.com/user/medigerati
[147] https://blog.kevinlamping.com/contact-me/
[148] https://leanpub.com/webapp-testing-guidebook/email_author/new