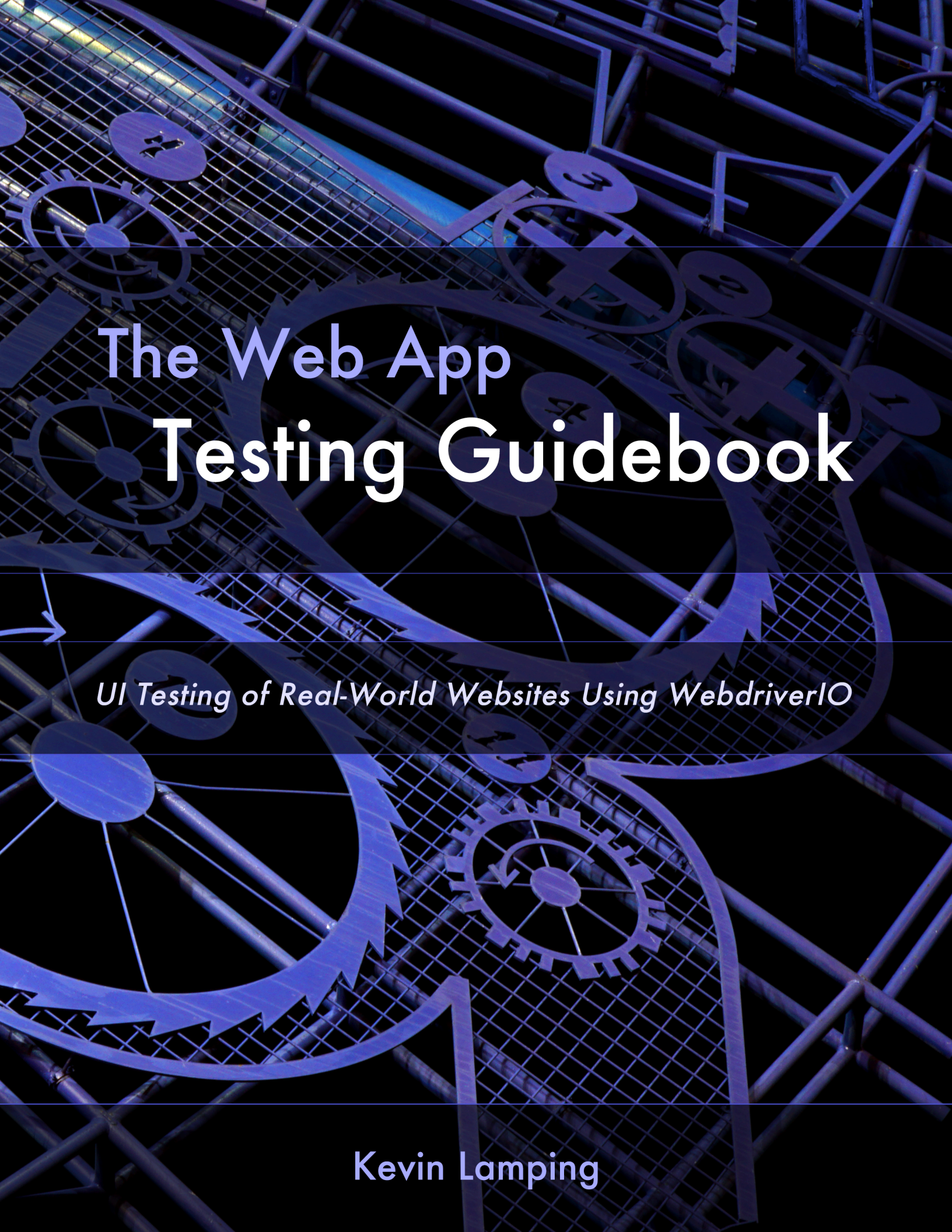


The Web App Testing Guidebook

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 2021-10-26

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?"

Can you paint beautiful art while only knowing how to mix colors and hold a paintbrush?

Is understanding WebdriverIO's 'addValue' command enough to write tests that are effective?

I don't think so.

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: 7.7.3
- Java 8 (optional)

Important: Node v16 includes a breaking change which causes 'sync' mode to malfunction. The examples in the book are therefore not compatible with Node v16. (An update to the book is currently in progress to address this)

Git Repository

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

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)

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

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

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 (12+, but not version 16)
- 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

Important: As mentioned in 'Technical Requirements', Node v16 includes a breaking change which causes 'sync' mode to malfunction. The examples in the book are therefore not compatible with Node v16. Please ensure you install an earlier version of Node than v16.

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

[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/

support and updates after 6 months. For more information on this, have a read through the Node.js release plan[33].

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.

[33] https://github.com/nodejs/Release#release-plan
[34] https://github.com/creationix/nvm
[35] https://github.com/tj/n

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`
- 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.

[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/

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 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).

[42] https://w3c.github.io/webdriver/
[43] https://www.npmjs.com/package/devtools
[44] https://www.npmjs.com/package/selenium-webdriver
[45] https://chromedevtools.github.io/devtools-protocol/

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, 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
- 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:

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

```
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
```

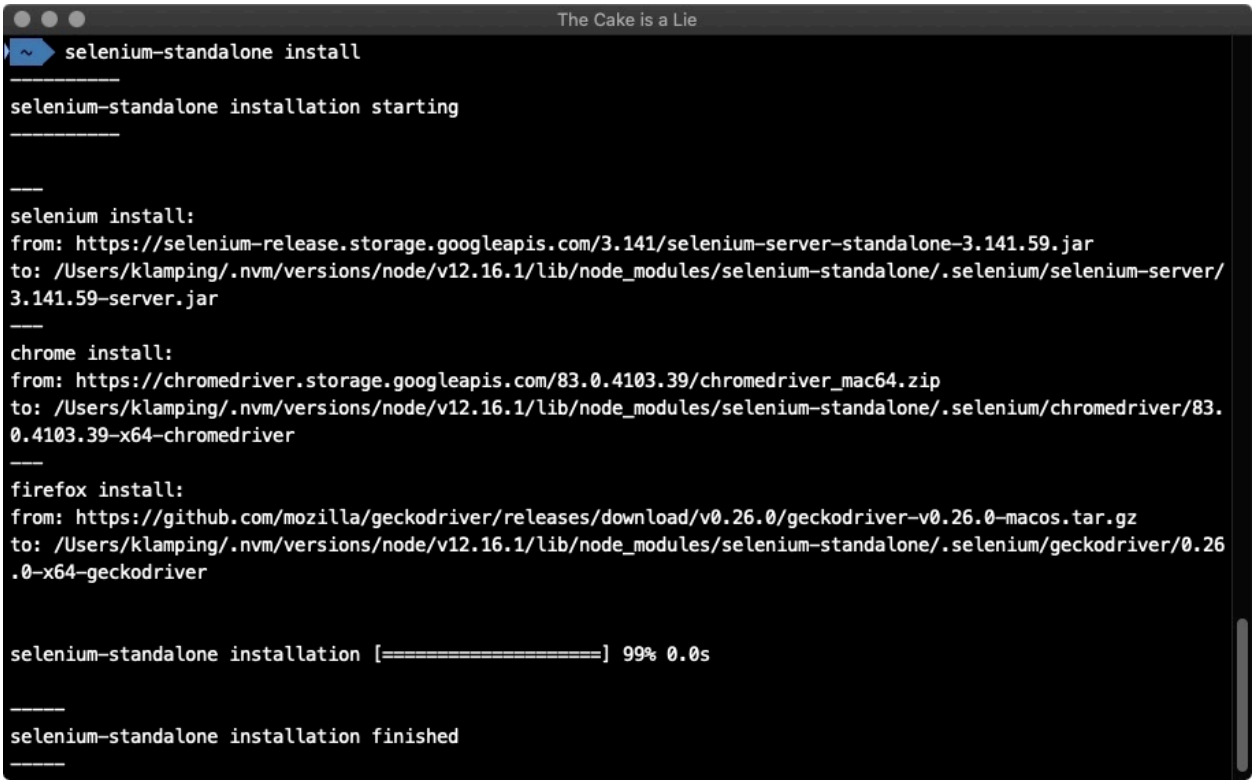
This will make a global command available called `selenium-standalone`. With this command, we can do the following:

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@7
```

Note: We include the `@7` version number so that the version you install is compatible with the examples in this book. If you'd like, you can leave the `@7` part off, but be warned that some code may not work.

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`

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]
```

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

```
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.

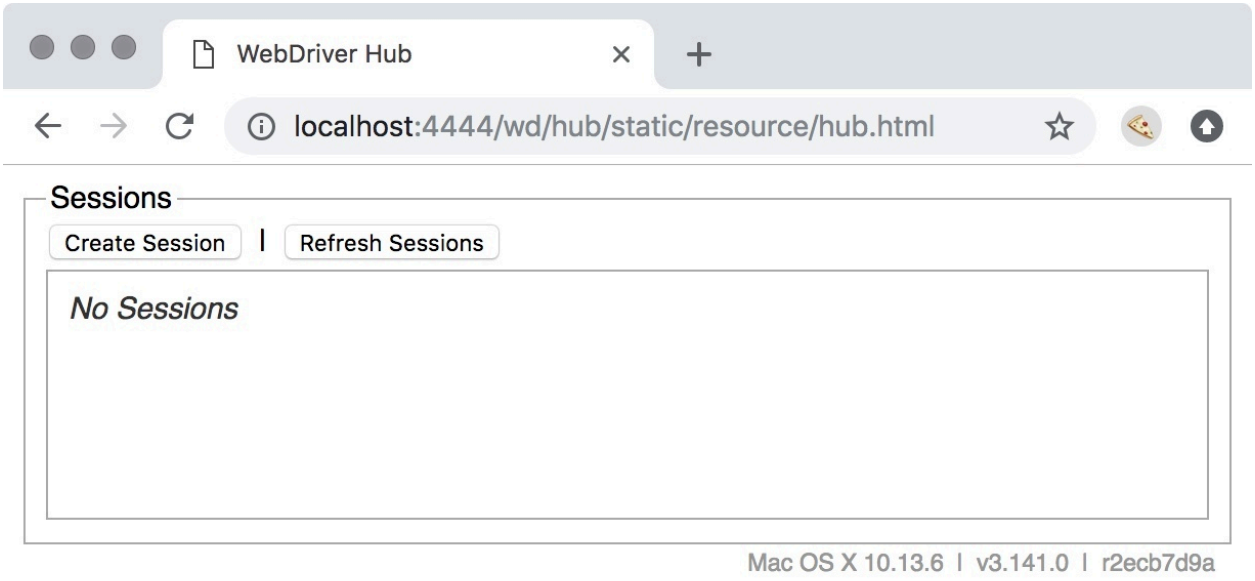
Running Through Selenium Standalone

*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

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

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 `/`).

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.