TAIKO

A Code-First Approach to User Journey Testing

Scott Davis

taiko.dev

Taiko

A Code-First Approach to User Journey Testing

Scott Davis

Version 1.0.0

> *I’ve always objected to doing anything over again if I had already done it once.*
>
> — Grace Hopper

Table of Contents

Why Test the User Journey?

I recently worked with a team of software developers who were quite proud of the level of testing they had in place on their project. They had a robust set of unit tests, and extensive code coverage reports to match. They were thoughtful about separating out their unit tests (performed in isolation with the tactical use of mocks and stubs) from their more involved, longer running integration tests.

The project was a traditional web site, complete with a login screen, a catalog of items, and the ubiquitous shopping cart. When I asked them how they tested their user interface, their responses ranged from sheepish to indignant.

```
"Well, we have a couple of Selenium tests, but they're pretty
brittle. They always seem to be broken, so we rarely run them."

"We decided that since we test the back end so extensively with unit
and integration tests, the UI tests were redundant and unnecessary."

"Since we use [a popular web framework], we assume that the framework
is well-tested. What bugs could we possibly find that they haven't
found already?"

"I click through every screen of the website right before we give our
showcase for the client, just to be sure that nothing blows up in our
face."
```

Brittleness. Redundancy. A "testing is for finding bugs"-only mentality. All of these seem to conspire against user interface testing, even among developers who consider themselves test-first advocates. Even the phrase "user interface testing" unwittingly seems to place the emphasis on "interface" rather than "user" in the mind of many developers.

Some of this might be explained by the code-first myopia that many software developers suffer from. Inexplicably, "users" are often reduced to a distant, abstract notion rather than the whole point of the software writing exercise in the first place. The tangible nature of the source code in front of the developer — in many cases, the source code that they just wrote themselves — is far more "real" than any notion of a user trying to use the code that they just composed.

All of which is why I’ve become a big fan of user journey testing — from the writing of the tests themselves to the words we use to describe the tests.

What is a User Journey Test?

A *user journey* is simply the specific steps a user has to perform to accomplish something on your website. If a user needs to buy ingredients for a spaghetti dinner from a grocery store website, the steps might be:

- Visit the website
- Add items (like tomatoes, garlic, Parmesan cheese, and noodles) to the shopping cart
- Click on the Checkout button

So then, a *user journey test* is a programmatic, automated way to perform the same steps that the user would in the same circumstances. Stated another way, user journey tests assure the correctness of your website by simply walking the same paths that your users already do.

How are User Journey Tests Different than Unit Tests?

For software developers who embrace unit testing, user journey testing should feel largely familiar, while differing in some subtle, interesting ways.

The principal considerations for unit tests are *inputs*, *outputs*, and *assertions*. If you're testing a `CalculateSalesTax` function, the inputs might be a `SalesAmount` and a `Location`. The output is the calculated amount of sales tax. And the assertions should ensure that the expected results match the actual results.

If the `CalculateSalesTax` function accepts `City` and `State` arguments as `Strings`, you can pass in a variety of values and assert that the results match your expectations. If the function expects a `LoggedInUser` to provide these details in a `UserAddress` class, you might have to mock out these objects so that the unit test can pass. Similarly, if the `CalculateSalesTax` function uses a GPS call to determine where the user is physically located at the time of checkout, you'll need to invest some additional time and energy in figuring out how to mock out the GPS and supply arbitrary `Latitude`/`Longitude` points that simulate different users in different locations.

So, you can see how quickly unit tests can become an exercise in simulation and abstractions away from reality. You don't need an actual user in Denver, Colorado to put actual grocery items in their shopping cart just to determine if your sales tax calculations are correct.

In comparison, user journey tests may simulate a user's behavior on your website, but that's

where the artificialness ends. We expect an actual website to be up and running, and the steps of the user journey tests should be indistinguishable (from the website's perspective) from those of an actual living, breathing user.

The principal consideration for a user journey test is the *steps* a user must take to complete a task. These tests are more interested in the *assurance* that each step of the user journey can be completed successfully, in sequence, rather than individual, explicit *assertions* along the way. A collection of passing user journey tests is your measure of success.

A good user journey testing library or framework will still allow you to simulate various aspects of the website interaction — simulating a smartphone on a slow cellular network connection from a fictitious location — but the actual steps performed on the website are more real than not. We can simulate the conditions around the user journey, but the steps of the user journey test itself are the real test of the website's actual functionality.

But perhaps the biggest difference between a unit test and a user journey test is the perspective of the test writer. Unit tests are written from the perspective of a software developer wanting to use a specific function. User journey tests are written from the perspective of a user wanting to use your website. This shift in perspective — putting yourself in the user role, describing the steps the user will take to accomplish a task — brings empathy into the equation. You, as the test writer, are quite literally walking the same path through your website as your users do. And this empathy isn't something that you should strive to mock away.

Conclusion

The conscientious team of software developers that you met at the beginning of this story decided try out a new user journey testing library called Taiko. The easy installation — `npm install -g taiko` — and the light, expressive JavaScript-based DSL made it easy to learn and explore. Taiko dramatically reduced the developer cost of writing user journey tests, and markedly reduced the test brittleness as well.

Here's what their "Buy the ingredients for a spaghetti dinner" user journey test looked like in Taiko:

A user journey test written in Taiko

```
openBrowser()
goto('https://thirstyhead.com/groceryworks/')
click('Produce')
click('Vine-Ripened Tomatoes')
click('Purchase')
```

When the developers tried to log into their website using Taiko, they discovered that they couldn't select the `Username` and `Password` fields because they hadn't used the semantically appropriate `<label>` elements with their `<input>` fields. They discovered an accessibility bug with the first test they wrote!

They went on to discover that they couldn't programmatically select items from the catalog because the items all had the same HTML ID due to a copy/paste error. A page that they tried to visit using Taiko errored out because they had been testing the pages in isolation and in a simulated environment. When they visited that same page as the user would — as one step in a sequence of many along the path of a specific user journey — they uncovered a tricky state-related bug that they never would've found testing the page in isolation.

At the end of my engagement with the project, one of the team members said:

```
"We always knew that testing was important. What we didn't realize is
that you unconsciously test what's important to you.

As software developers, we were deeply invested in the quality of our
code by writing unit tests. But what we failed to do was dedicate
that same level of care to the user experience.

Now that we know more about user journey tests, we know that our
website's user experience will be as rich and bug-free as the
developer experience."
```

Introduction

Testing has been important to me as a professional software engineer for as long as I can remember.

Now, I'm not suggesting that I've been writing tests since the very beginning, or that I always write tests, or even that the tests I write are particularly good. But my first professional development gigs as a software consultant were in Java in the late 1990s, and a new project called *JUnit* — a unit testing framework written by Erich Gamma and Kent Beck — was really taking off.

What intrigued me about JUnit at the time was that it was not just a simple testing library — it was a key part of a philosophy called *Test-Driven Development* (**TDD**), which in turn was a key part of a larger system of agile practices called *Extreme Programming* (**XP**). Not surprisingly, Kent Beck (along with co-author Martin Fowler) wrote a book about all of these practices called *Planning Extreme Programming* — one of the first books I read on the subject.

Despite the "Extreme" qualifier in XP, the practices recommended by Kent and Martin in their book seemed quite sensible and practical:

- Customers pick the features to be added
- Programmers add the features so that they are completely ready to be used
- Programmers and customers write and maintain automated tests to demonstrate the presence of these features

This felt like such a common sense strategy to me at the time that I couldn't fully grasp why all software developers didn't use this approach. If I drop my car off at the repair shop and say, "When I drive above 55 miles per hour, I hear a loud clanking", I fully expect the mechanic to:

- Drive my car above 55 miles per hour so that they can hear (and verify) the clanking sound
- Fix the clanking
- Demonstrate to me, when I pick up my car after the repair, that the clanking is gone by driving above 55 miles per hour with me in the car

Now, if you've been programming for a while, you might be thinking, "That clanking is a bug, not a feature!" And while you're technically correct, what different behavior would you expect if I dropped my car off and said instead, "I'd like you to upgrade my sound system" or "I'd like you

to install a new sun roof"? I'd expect the same sequence of events. Wouldn't you?

What is User Journey Testing?

Suppose my client says to me, "I need a website for a software conference I'm running. I'd like to have a page that lists all of the speakers. When you click on a speaker, I'd like that to lead to a page with their biography and a list of their talks." What they just described to me is a *user journey*.

So, I understand the feature they're asking for. I can add that feature with relatively little effort. But how can I demonstrate the new feature I just added?

As the developer of the feature, I probably manually go through the sequential steps of "Go to the Speakers Page; Click on a Speaker; Verify that I end up on a page with the Speaker's biography and list of talks" tens, if not hundreds, of times during the development process. After all, I want to be fully convinced that the process works before I demonstrate it to my client.

But manual testing can be time consuming and prone to error if not done consistently. What if I could automate the user journey? What if I could write a little bit of code that tests the user journey in a consistent, repeatable manner? Something like this:

A user journey test written in Taiko

```
openBrowser()
goto('https://thirstyhead.com/conferenceworks/speakers/')
click('Dr. Rebecca Parsons')
highlight('About')
highlight('Talks')
screenshot({path:'speakerListTest-screenshot.png'})
```

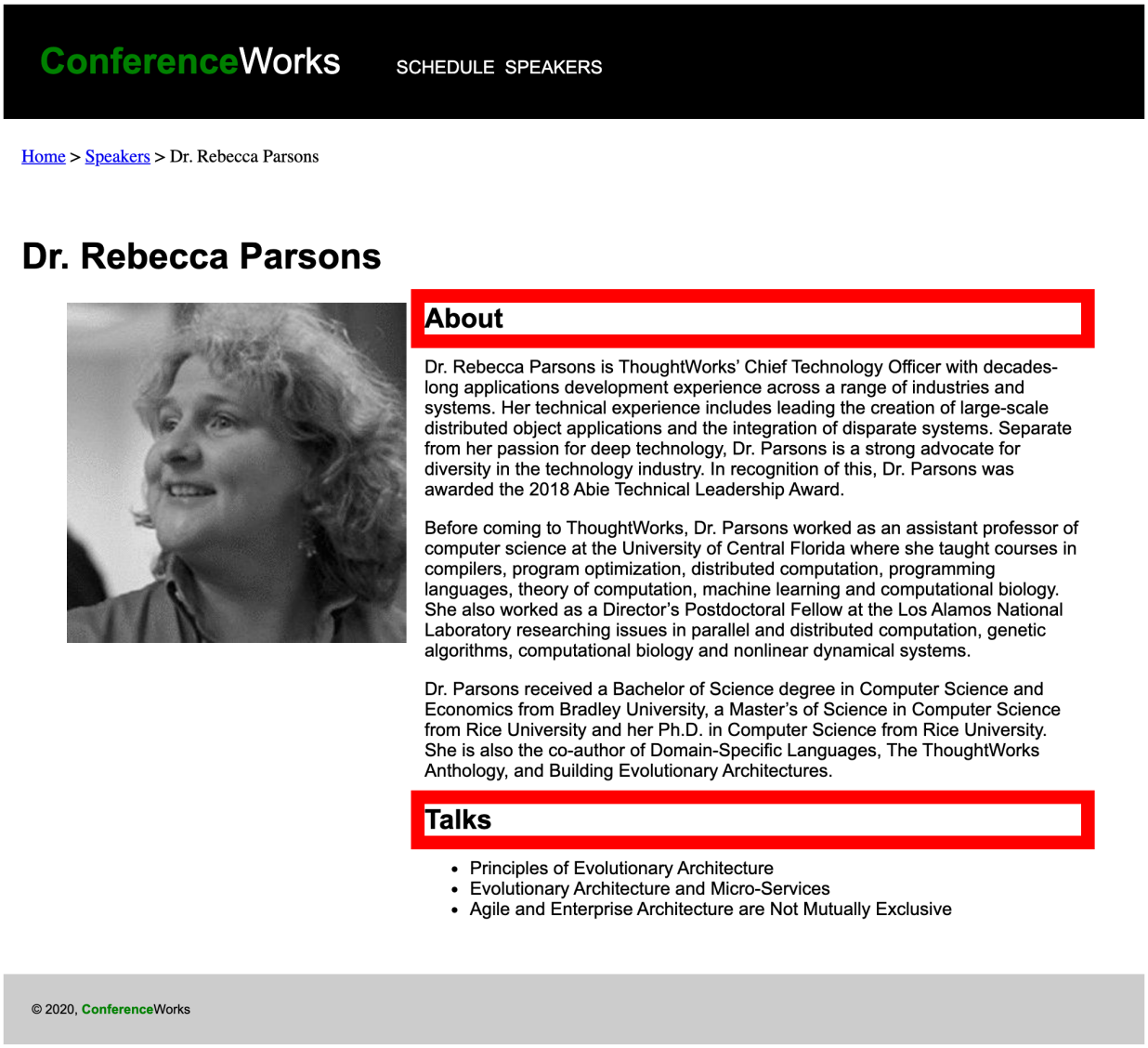

Figure 1. The resulting screenshot from the user journey test

Taiko is an open source Node.js library for testing modern web applications. It is a purpose-built *Domain Specific Language* (**DSL**) for writing user journey tests. Anything that your user can do on your website can be automated using Taiko. So, if your user does this on their *login journey*:

- Go to the Login page
- Click the 'Username' field
- Write in the user name
- Click the 'Password' field

- Write in the password
- Click the Submit button

…you can automate that with Taiko like this:

Automating the login user journey with Taiko

```
openBrowser()
goto('https://thirstyhead.com/conferenceworks/login')
click('Username')
write('suzi@q.com')
click('Password')
write('wordpass')
click('Submit')
```

Of course, TDD is no more tied to JUnit than user journey testing is tied to Taiko. User journey testing is a practice — a discipline — that can be implemented in a variety of different languages, using a variety of different libraries. If you can practice TDD by using a library other than JUnit (say, *NUnit* for .NET languages, or *Test::Unit* for Ruby), then you can certainly write user journey tests using a library other than Taiko. But I'll continue to use Taiko here whenever I need to explain a concept in code.

If you'd like to follow along and run the Taiko tests yourself, installing Taiko is as simple as typing `npm install -g taiko`. Once Taiko is installed, you can type `taiko` at the command prompt to enter the interactive REPL and explore on your own. Anything that you type in the Taiko REPL can be exported to modern JavaScript by typing `.code` to see the code on screen, or by typing `.code mytest.js` to save the JavaScript to the current working directory. After that, you can type `taiko mytest.js` to run the code outside of the REPL by hand or, say, in your *Continuous Delivery* (**CD**) pipeline.

Reconsidering the Test Pyramid

One of the most important aspects of unit testing is, well, the *unit* of code being tested. More specifically, the size of the unit. The goal of unit testing is to focus on the smallest cohesive hunk of code that you can tease apart from the rest of the application in isolation. I often say that unit tests "test the bricks, not the building" because, after all, you can't trust the building if you don't trust the bricks.

If your unit of code interacts with a database, or a file system, or a remote web service, it's common to *mock* or *stub* out those services with a fast, in-memory doppelgänger that behaves

just like the original service does, but without the latency and brittleness that depending on an external service might introduce.

Author Mike Cohn, in his book *Succeeding with Agile*, introduced a powerful visual metaphor for this with the *Test Pyramid*.

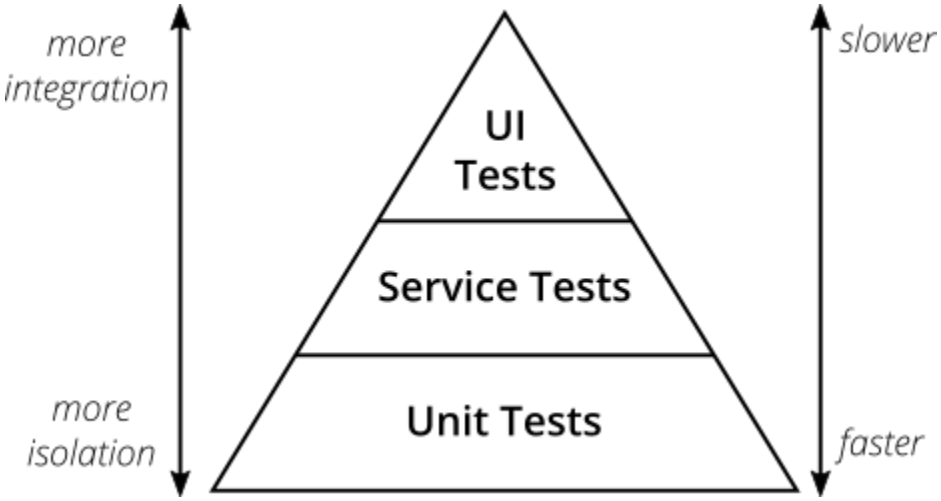

Figure 2. Mike Cohn's Test Pyramid, from Succeeding with Agile

Under this rubric, developers are encouraged to write as many unit tests as they can, because they are the fastest, most stable, most repeatable types of tests. Service Tests (or, more popularly in subsequent years, *Integration Tests*, since they integrate with the databases and web services that unit tests intentionally mock out) still offer value, but they are typically slower to run, and more prone to brittleness due to circumstances beyond the control of the test itself. According to the Test Pyramid, you should write fewer of them than unit tests.

At the tip of the pyramid are *UI Tests* (User Interface Tests). Since all of the pieces of the application must be up and running, properly configured and secured — and since many of those services may be out of the immediate control of the individual developer, or there may be a lack of a proper testing environment that accurately mirrors the production environment — these tests are visually deemed "least important" in the hierarchy of tests.

Arguing against the validity of the Test Pyramid, even in a book that is all about those tests that live on the vanishing tip of the pyramid as it recedes from view, is a futile battle. Especially since I personally agree with the message of the Test Pyramid — that is, if I'm a developer who is sitting furthest away from that mythical *user* that everyone else seems to insist exists. Writing a test on behalf of someone who I most likely will never meet is a tall order to fill. On the other hand, writing tests for my fellow developers — developers who I deal with every day; developers who will be depending on the validity of my code so that they can trust in the validity of their own code — is a crucial and essential goal.

This myopic view of the development process as a whole isn't myopic in the least if you're a brick builder. But everyone else actively involved in the process who is further "up the pyramid", towards the user and the finished software product, might take issue with their role (and their tests) being deemed "less important".

The Test Spectrum

Consider, for a moment, the legion of software development professionals who deal with the user directly and repeatedly. The group of software developers who are just as dedicated to the validity of the software application being developed. The group of professionals who want to apply the same engineering rigor of testing to the *User Experience* as thoughtful developers do to the *Developer Experience*.

This change in perspective might benefit from a different visual metaphor. Let's call it the *Test Spectrum*.

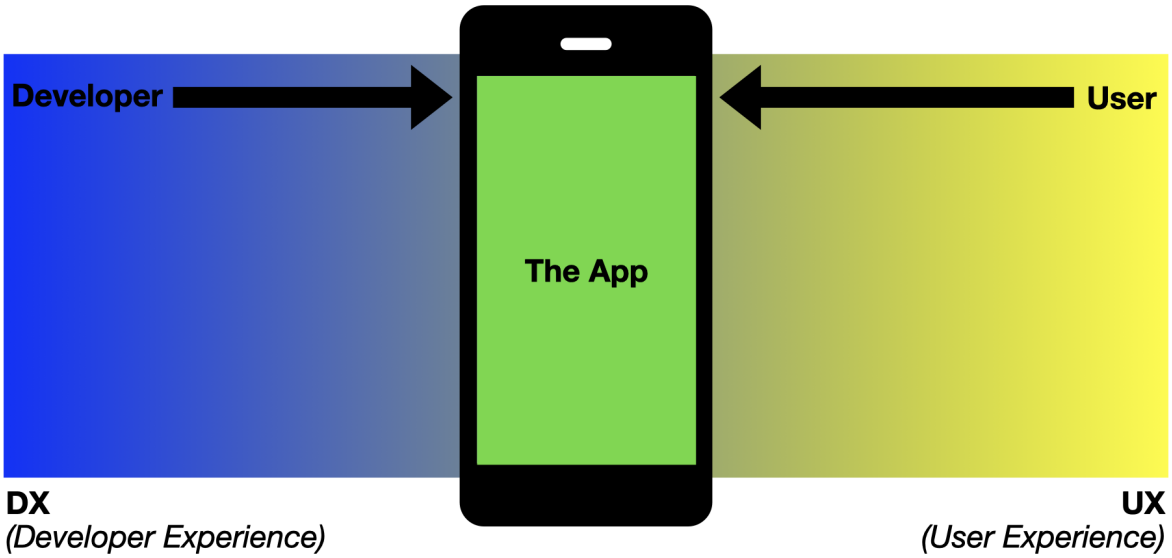

Figure 3. A new visual metaphor for software development that places the app at the center of focus: the Test Spectrum

First of all, let's place the application at the center of our model. A finished, correctly working app is the highest priority of both the developer and the user. As the user describes what they want the app to do, the developer converts their vision into working code. The application, therefore, is both the fulcrum of the user-developer relationship as well as proof of its success.

The application is also an opaque boundary between the two worlds. Source code, and the test suite that measures its success, is quite literally written in a foreign, unintelligible language to the end user. A symphonygoer can tell you in great detail what they enjoy about the music, but they may or may not be able to point to the specific passage in the sheet music that brings them such joy.

So, with this new perspective in mind, let’s place unit tests on the Test Spectrum.

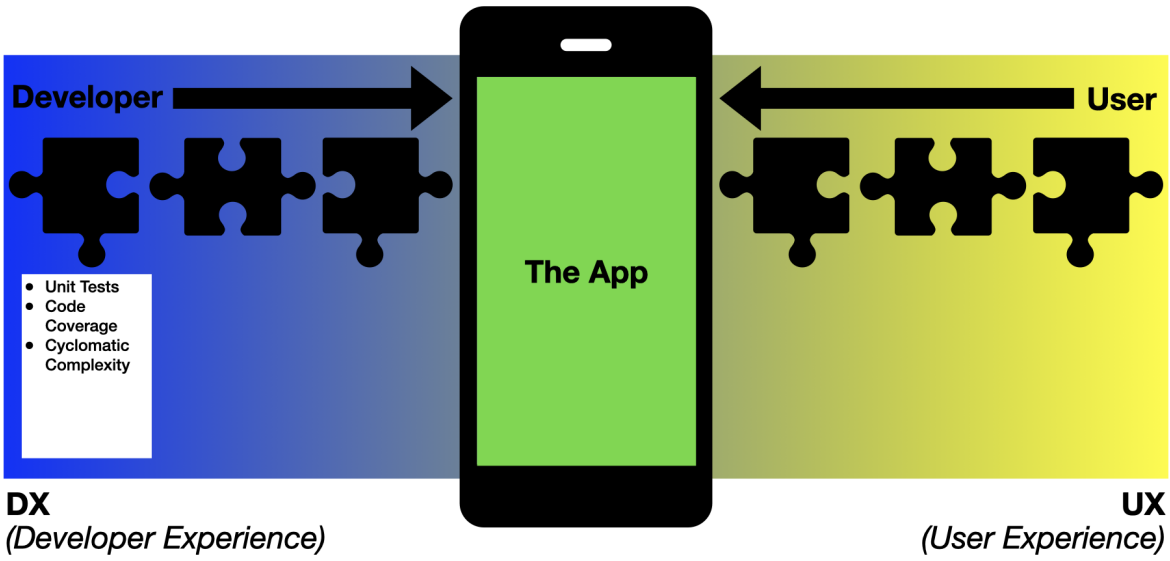

Figure 4. Unit tests on the Test Spectrum

In our new visual metaphor, we can see that unit tests are about as far away from the User Experience as the spectrum allows. This doesn’t mean that unit tests are unimportant; instead, it shows us who the unit tests are most important to. As Neal Ford, co-author of *Fundamentals of Software Architecture* and *Building Evolutionary Architectures* says, "Testing is the engineering rigor of software development."

The Test Spectrum also visually indicates that unit tests are just one piece of the testing puzzle.

Without a solid suite of unit tests, software developers cannot have subsequent conversations about *code coverage* — how much of the codebase is *covered* or tested by unit tests — and *cyclomatic complexity* — how complex the codebase is, which can suggest that *hidden bugs* might be masked by the accidental complexity of the code being tested.

These conversations are crucially important to me as a software developer, from a developer’s perspective. But these tests don’t speak to the user experience. Unit tests aren’t shipped with the

finished app. The user can't run them directly. While the user definitely benefits from a solid suite of unit tests in an abstract way, much like a symphonygoer benefits from a cellist applying bowstring wax before the performance and practicing their musical scales, the presence or absence of unit tests, let alone the intrinsic quality of them, is invisible to the end user.

User Journey Testing with Taiko

So, what does speak to the user experience, and affect the user directly? The user interface, of course! From the user's perspective, the UI *is the app*, just like the API *is the app* from the developer's perspective. The user isn't adding Strings to an Array, or even CatalogItems to a ShoppingCart object when they use the app — they are adding bananas to their basket.

And what might a test look like, from the user's perspective?

Adding bananas to the basket with Taiko

```
openBrowser()
goto('https://thirstyhead.com/groceryworks/')
click('Produce')
click('Bananas')
click('Purchase')
```

These are the steps the user would take, quite possibly in a language similar to (but not identical to) what they would use to describe their user journey to someone else. The Taiko DSL isn't meant to be plain English, but hopefully it is readable to the non-programmer.

Taiko is, in fact, well-formed JavaScript. It is an example of an *internal DSL* — "internal" to and consistent with the programming language that it is written in — as opposed to an *external DSL* which has its own personal syntax rules separate from its source programming language.

If you want to capture a user journey in something even closer to the language the user used to describe the steps, you might be interested in another open source testing tool called *Gauge*. Gauge allows you to describe the steps of your test in a language called *Markdown*, which is as close to plain English as I've been able to find.

Here's what a Gauge test might look like:

Adding bananas to the basket with Gauge

```
## Buying Bananas
* goto "GroceryWorks"
* click "Produce"
* click "Bananas"
* click "Purchase"
```

And here's another way that you could represent the same user journey in Gauge:

Another way to add bananas to the basket with Gauge

```
## Buying Bananas
* visit the shopping website
* click on the "Produce" menu item in the sidebar
* select "Bananas" from the list of produce items
* press the "Purchase" button on the shopping cart
```

Gauge and Taiko work quite well together. All you have to do is associate the steps in Gauge with the underlying code in Taiko, and you have a set of user journey tests expressed in a language that any non-programming user should recognize and understand.

Since our focus here is on Taiko, I'll leave Gauge behind for the time being. But if Gauge looks interesting to you, I encourage you to learn more about it at https://gauge.org/.

Placing User Journey Tests on the Test Spectrum

Regardless of which language we use to express our user journey tests, where do you think they might show up on the Test Spectrum?

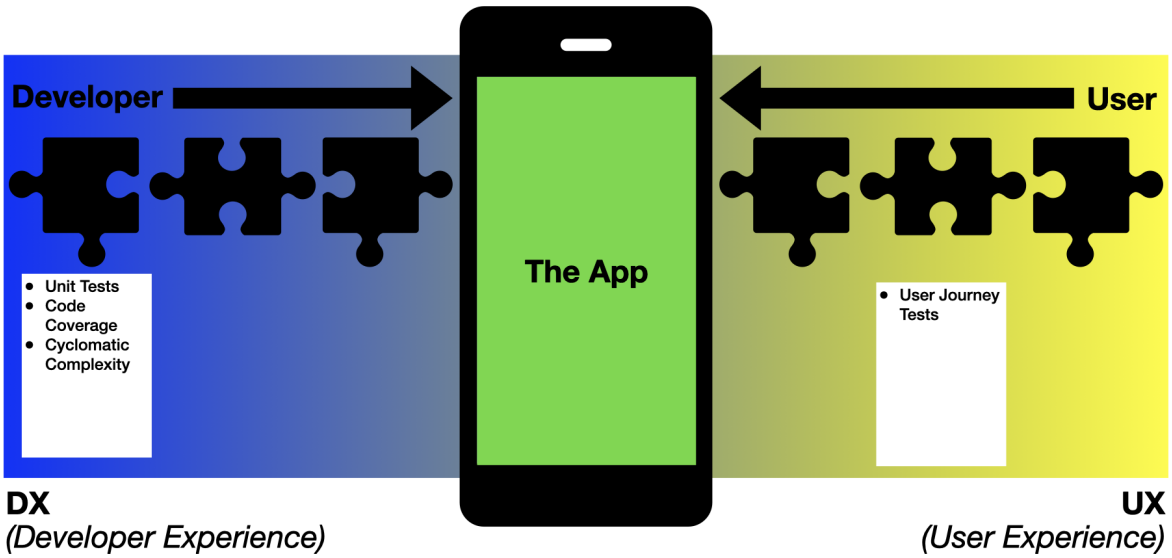

Figure 5. User journey tests on the Test Spectrum

As you can see, unit tests and user journey tests both exercise the app in important ways. The only difference is that unit tests exercise the app from the developer's perspective, while user journey tests exercise the app from the user's perspective. Unit tests are written in the language of the developer, while user journey tests are written in the language of the user. Unit tests are for the benefit of the developer, while user journey tests are for the benefit of the user.

And why didn't I place user journey tests to the far right of the Test Spectrum like I did with unit tests to the far left? Well, there are a number of important types of tests that aren't automated, or even automatable. *Manual tests* live at the far right — tests that are run by hand instead of by software.

For example, *usability tests* give the user a task to perform like, "Buy the ingredients you'd need to make a spaghetti dinner" while usability experts watch and evaluate how easy it is to complete the task. Another example is *accessibility tests*, where a user who has low vision or complete vision loss is encouraged to make the same user journey through the app to purchase the ingredients for a spaghetti dinner.

Since user journey tests are automated, they are one step closer to the developer experience than manual tests are. Similarly, integration tests are one step closer to the user experience than unit tests are.

If you filled in the Test Spectrum with all of the puzzle pieces, it might look something like this:

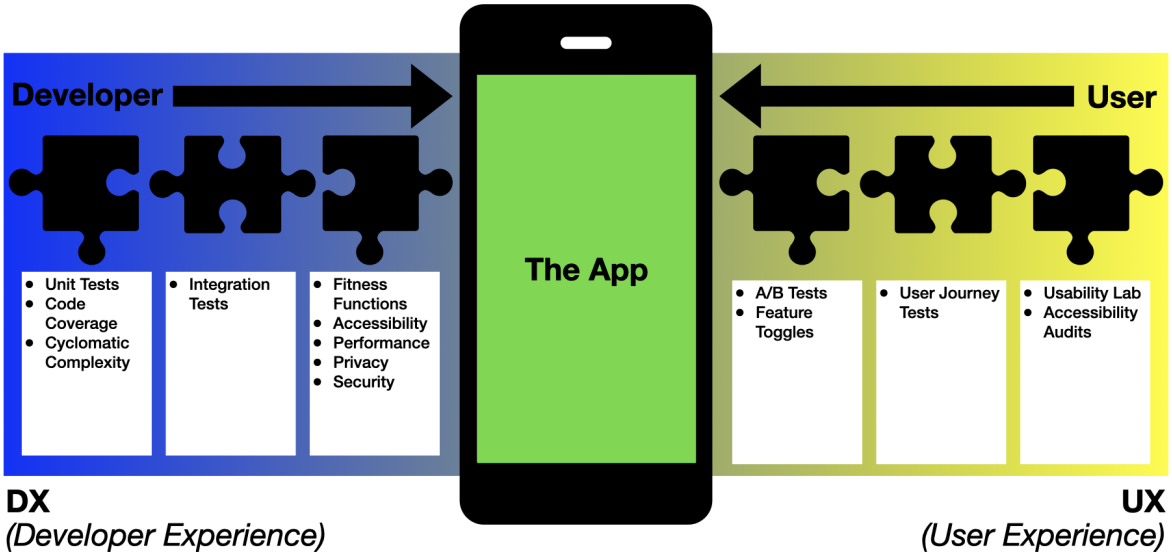

Figure 6. A variety of tests on the Test Spectrum

Note that this is highly subjective, and not meant to be a recipe for you to follow down to the letter. It's meant to be a way for you to think about tests. Is this particular type of test closer to the finished app, or farther away? Whose experience does this type of test affect most?

For example, *fitness functions* test the health of the application's architecture rather than a low-level API. If your app doesn't meet minimum *accessibility* requirements, that's a showstopper bug that the developers need to fix. If it doesn't meet the minimum expectations you put into your *performance budget*, that is something the developers need to be notified of. These, along with *privacy* and *security*, form the *APPS* suite of fitness functions that I, as a Web Architect, typically put in place to ensure that the app in question is qualified and ready to be released to production. And just like unit tests, these fitness functions are opaque to the end user.

Now consider *A/B tests* — a programming technique that shows one version of a feature (A) to a select group of users, while another group of users are exposed to the (B) version of the same feature. An example of this might be allowing 1% of your user base to log in with their Twitter account (a new feature that you'd like to evaluate) while the remaining 99% log in with their existing username and password. Since this directly affects the user experience, I've placed it along the UX spectrum, but closest to the app and the developers.

Empathy and the Test Spectrum

Another way that you can evaluate where the tests belong on the Test Spectrum is through the prism of *empathy*. When I’m in TDD-mode — writing my tests first and watching 'em fail, then writing the code to make 'em pass — I’m rarely testing for bugs. After all, the code doesn’t exist yet! What kind of pessimist with low self esteem would I have to be to write tests while thinking, "Yup, this is the kind of bug I always write…"?

Instead, I’m writing my first set of tests to take the API out for a spin. I’m quite literally the first user of my code when I write my tests, so I’m constantly asking myself, "How does this API feel? Did I name it well? Do the arguments make sense? Are they in a logical order?"

This is developer empathy. I’m putting myself in the shoes of other developers who will eventually use my code. How does that make me feel? Do I trust my code enough to share it with others?

On the UX side of the equation, a set of practices called *Design Thinking* resembles agile and XP practices in striking ways. Both are iterative processes. Both recommend making tiny changes, evaluating their effectiveness, and then repeating the process again.

And the first step in Design Thinking is empathy.

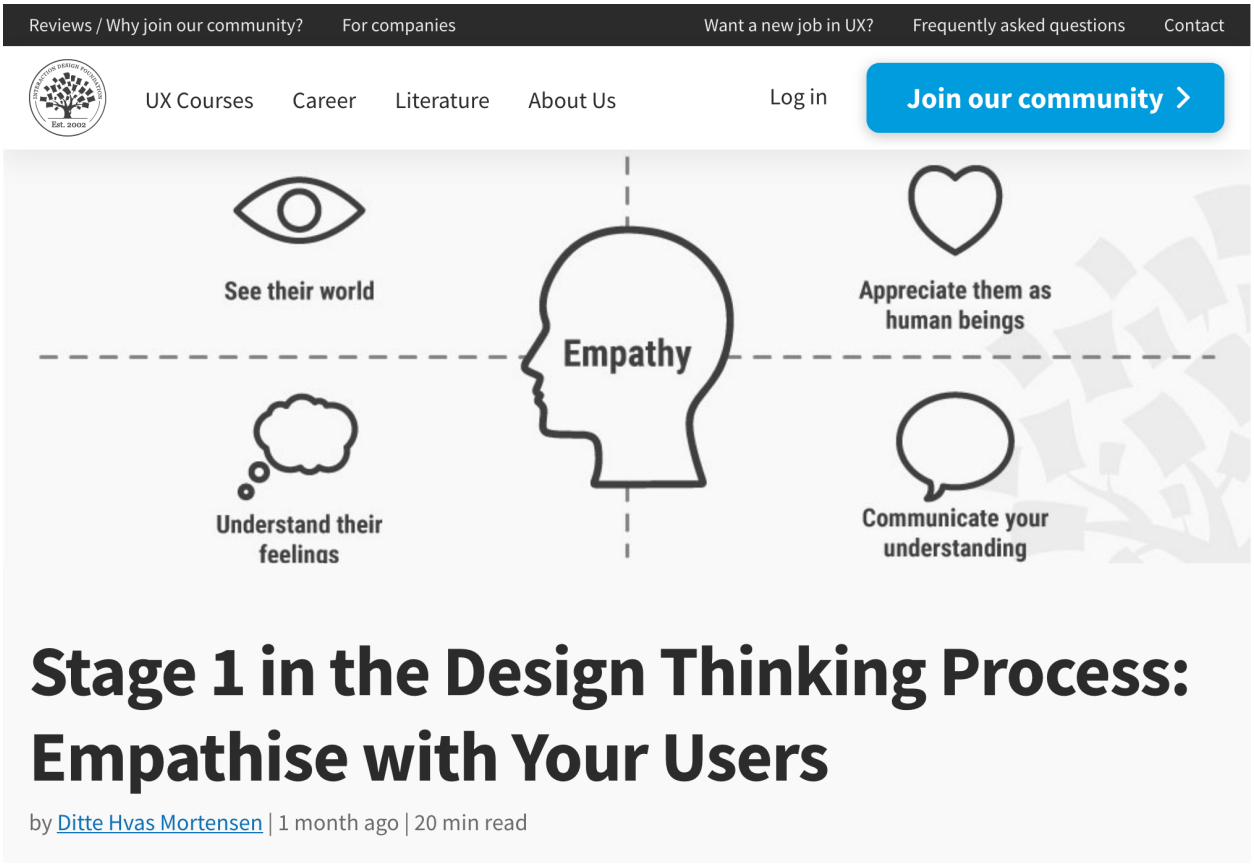

Figure 7. Empathy is Stage 1 of the Design Thinking process

So, when I'm writing user journey tests, I'm rarely looking for bugs, either. I'm taking the UI out for a spin. Are there too many clicks to get the bananas into the basket? Do I have to log in before I start putting things in my shopping cart, or can I log in just-in-time, right before I need to provide payment and shipping details? I ask myself, "Is this a user experience that I'd enjoy?" in the same way I ask myself if the API I'm writing is something that I'd enjoy as a developer.

Just like I'd do with TDD, I can write my user journey tests before I've written the UI. I run the user journey test, and then watch it fail when I try to reach the website in question. I write just enough HTML to make the test pass, and then it fails when I try to click "Produce". I can add a "Produce" item to the sidebar to allow the next step to pass and keep going until I've successfully implemented the entire user journey.

As a matter of fact, when I teach classes and workshops on web development, I've started including Taiko tests with each of the labs that I ask my students to complete. They can run the test at the beginning of the lab to see what I'm asking them to do, watch it fail, and then chip away at it until they know that they're unambiguously done and unambiguously correct.

Just like I've done for years with my other programming classes and unit test-driven labs.

The one thing that might feel odd about writing user journey tests, when compared to unit tests, is the apparent lack of assertions. At least, it certainly did to me initially.

When I’m teaching my students about unit tests, we focus on inputs, outputs, and assertions. If I add three things to my shopping cart in a test, my assertion might be something like `assert shoppingCart.itemCount() === 3`. Most people think of things like JUnit as a "testing library", while, in fact, it really is more of an "assertion library".

There are a number of different testing/assertion libraries written in JavaScript, and you are welcome to incorporate any of them into your Taiko testing routine. After all, a Taiko test is just a well-formed JavaScript program that runs in Node.js. If I need an assertion library, I’ll typically type `const assert = require('assert')` at the top of my Taiko test. But I do that less often than you might think.

Why is that? Because each step of a user journey test is an implicit assertion. I’m validating that the user journey works as I expect it to, and any step along the way that fails means that my assumption about the journey is flawed.

If I’m adding bananas to my basket, I’m confident that some developer deep in the depths of that opaque codebase has written a unit test to make sure that `shoppingCart.itemCount()` is valid and ready to be used in production. With Taiko, I’m not testing the bricks; I’m testing the building.

Installation and Configuration

Installing Taiko couldn’t be easier. It’s a single command: `npm install -g taiko`. But there’s plenty more that you can do to configure and customize Taiko once it’s installed.

Install Taiko

```
$ npm install -g taiko

/Users/scott/.nvm/versions/node/v12.14.1/bin/taiko ->
/Users/scott/.nvm/versions/node/v12.14.1/lib/node_modules/
  taiko/bin/taiko.js

> taiko@1.0.7 install
/Users/scott/.nvm/versions/node/v12.14.1/lib/node_modules/taiko
> node lib/install.js

Downloading Chromium r724157 - 117.6 Mb [====================] 100%
0.0s

> taiko@1.0.7 postinstall
/Users/scott/.nvm/versions/node/v12.14.1/lib/node_modules/taiko
> node lib/documentation.js

Generating documentation to lib/api.json
+ taiko@1.0.7
added 73 packages from 114 contributors in 50.835s
```

When you install Taiko, notice that you get a known-compatible version of Chromium installed as well. Chromium is an open-source, bare-bones web browser that, as you might've guessed by the name, is the core of the Google Chrome browser. Interestingly, Chromium is also the foundation of the Opera browser, the Microsoft Edge browser, and many others. Chromium-based browsers make up roughly two-thirds of the browser market, so using Chromium with Taiko covers the widest possible swath of typical web users.

Run the Taiko REPL

```
$ taiko

Version: 1.0.7 (Chromium:81.0.3994.0)
Type .api for help and .exit to quit

> openBrowser()
 ¬ Browser opened
> goto('wikipedia.org')
 ¬ Navigated to URL http://wikipedia.org
> click('Search')
 ¬ Clicked element matching text "Search" 1 times
> write('User (computing)')
 ¬ Wrote User (computing) into the focused element.
> press('Enter')
 ¬ Pressed the Enter key
> click('Terminology')
 ¬ Clicked element matching text "Terminology" 1 times
> closeBrowser()
 ¬ Browser closed
> .exit
```

The Taiko REPL (Read Evaluate Print Loop) is an interactive terminal shell that allows you to experiment with a live browser. When you type `openBrowser()`, a browser window should open on your computer. When you type `goto('wikipedia.org')`, you should end up on the Wikipedia website.

The Taiko REPL is the perfect way to experiment with Taiko whether you are brand new to the DSL or an experienced user. Once you are confident that your code works (because you've just watched it work), you can save it and run it outside of the REPL, either manually or as a part of your automated CD pipeline.

Save Code from the Taiko REPL

```
$ taiko

> openBrowser()
 ¬ Browser opened
> goto('wikipedia.org')
 ¬ Navigated to URL http://wikipedia.org
> closeBrowser()
 ¬ Browser closed
> .code

const { openBrowser, goto, closeBrowser } = require('taiko');
(async () => {
    try {
        await openBrowser();
        await goto('wikipedia.org');
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();

// If you provide a filename,
//   .code saves your code to the current directory
> .code visit-wikipedia.js
```

At any point in the Taiko REPL, you can type `.code` to see what the JavaScript will look like once you run your Taiko code outside of the REPL. Notice that this is modern asynchronous JavaScript — every command will `await` completion before moving on to the next step.

If you'd like to save this code for running outside of the REPL, simply provide a filename like `.code visit-wikipedia.js`. This will save the JavaScript code to the current directory.

Run Taiko Code Outside of the REPL

```
$ taiko visit-wikipedia.js

 ¬ Browser opened
 ¬ Navigated to URL http://wikipedia.org
 ¬ Browser closed
```

When you type `taiko` without a filename, it launches the Taiko REPL. When you type `taiko visit-wikipedia.js`, it runs the Taiko commands in the file.

You might have noticed that typing `openBrowser()` in the Taiko REPL actually opens a browser that you can see. By default, running Taiko commands outside of the REPL runs the browser in "headless mode". This means that the browser isn't actually shown on screen, but its behavior in headless mode is identical to its behavior with a visible browser. This is ideal for running Taiko commands in an automated server environment where there most likely isn't a screen to display the progress.

If you'd like to see the browser when running Taiko commands outside of the REPL, type `taiko --observe visit-wikipedia.js`. The `--observe` command-line flag, in addition to showing the browser, also inserts a 3 second (3000 millisecond) delay between steps to make them easier to observe. If you'd like to adjust this delay, use the `--wait-time` command-line flag — `taiko --observe --wait-time 1000 visit-wikipedia.js`.

Get Command-Line Help

```
$ taiko --help

Usage: taiko [options]
       taiko <file> [options]

Options:
  -v, --version                    output the version number

  -o, --observe                    enables headful mode and runs
                                   script with 3000ms delay by
                                   default. pass --wait-time
                                   option to override the default
                                   3000ms

  -l, --load                       run the given file and start the
                                   repl to record further steps.

  -w, --wait-time <time in ms>     runs script with provided delay

  --emulate-device <device>        Allows to simulate device
                                   viewport.
                                   Visit https://github.com/getgauge/
                                   taiko/blob/master/lib/devices.js
                                   for all the available devices

  --emulate-network <networkType>  Allow to simulate network.
                                   Available options are GPRS,
                                   Regular2G, Good2G, Regular3G,
                                   Good3G, Regular4G, DSL,
                                   WiFi, Offline

  --plugin <plugin1,plugin2...>    Load the taiko plugin.

  --no-log                         Disable log output of taiko

  -h, --help                       display help for command
```

There are a number of command-line flags that affect Taiko at runtime. `--observe` and `--wait-time` allow you to see the browser as the Taiko commands are performed. (Normally, Taiko runs in "headless mode" at the command-line.)

You can use `--emulate-device` and `--emulate-network` to simulate smartphone usage.

`--load` allows you to preload the Taiko REPL with commands stored in a file.

`--plugin` allows you to load Taiko plugins that extend native behavior.

Run Taiko in an Alternate Browser

```
$ TAIKO_BROWSER_PATH=/Applications/Opera.app/Contents/MacOS/Opera
  taiko visit-wikipedia.js

¬ Browser opened
 ¬ Navigated to URL http://wikipedia.org
 ¬ Browser closed
```

When you install Taiko, it ships with a known-good version of Chromium — one that won't auto-update and inadvertently break your tests. But you might want to use Taiko to drive an alternate Chromium-based browser, like Google Chrome, Opera, or Microsoft Edge. To do so, simply create a `TAIKO_BROWSER_PATH` environment variable that contains the path to the browser you'd like Taiko to use.

NOTE

Taiko uses the Chrome DevTools Protocol (CDP) to communicate with the browser. This is the same protocol that the Google Chrome DevTools use, as well as Lighthouse (for reporting) and Puppeteer (a similar tool to Taiko written by Google). As of this writing, neither Firefox nor Safari support CDP-based communications. For an alternate way to drive non-CDP browsers, look at the WebDriver[1] W3C initiative.

Emulate a Smartphone

```
$ taiko --observe
        --emulate-device 'iPhone X'
        --emulate-network 'Regular3G'
        visit-wikipedia.js

 ¬ Browser opened with viewport iPhone X
 ¬ Device emulation set to iPhone X
 ¬ Set network emulation with values "Regular3G"
 ¬ Navigated to URL http://wikipedia.org
 ¬ Device emulation set to iPhone X
 ¬ Browser closed
```

When you run Taiko on your desktop computer, it opens a desktop browser and runs at full network speed. If you'd like Taiko to emulate a different kind of device, use the `--emulate-device` and `--emulate-network` command-line flags.

To find the available values for these flags, type `taiko --help`.

For a better understanding of what these flags do, you can look at the JavaScript files that supply the values in devices.js[2] and networkConditions.js[3] on GitHub[4].

Here is the code for `iPhone X` device emulation:

```
  'iPhone X': {
    userAgent:
      'Mozilla/5.0 (iPhone; CPU iPhone OS 11_0 like Mac OS X)
AppleWebKit/604.1.38 (KHTML, like Gecko) Version/11.0 Mobile/15A372
Safari/604.1',
    viewport: {
      width: 375,
      height: 812,
      deviceScaleFactor: 3,
      isMobile: true,
      hasTouch: true,
      isLandscape: false,
    },
  },
```

The emulation code sets a device-specific User-Agent string, and adjusts the size and characteristics of the screen.

Here is the code for `Regular3G` network emulation:

```
  Regular3G: {
    offline: false,
    downloadThroughput: (750 * 1024) / 8,
    uploadThroughput: (250 * 1024) / 8,
    latency: 100,
  },
```

The emulation code throttles download and upload speeds, as well as adding some artificial latency.

[1] https://www.w3.org/TR/webdriver2/
[2] https://github.com/getgauge/taiko/blob/master/lib/data/devices.js
[3] https://github.com/getgauge/taiko/blob/master/lib/data/networkConditions.js
[4] https://github.com/getgauge/taiko

Working with the Browser

In this chapter, you'll learn how to open and close a browser, open and close tabs, and take a screenshot.

Open and Close a Browser

In the REPL

```
> openBrowser()
 ¬ Browser opened
> closeBrowser()
 ¬ Browser closed
```

In a script

```
const { openBrowser, closeBrowser } = require('taiko');
(async () => {
    try {
        await openBrowser()
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

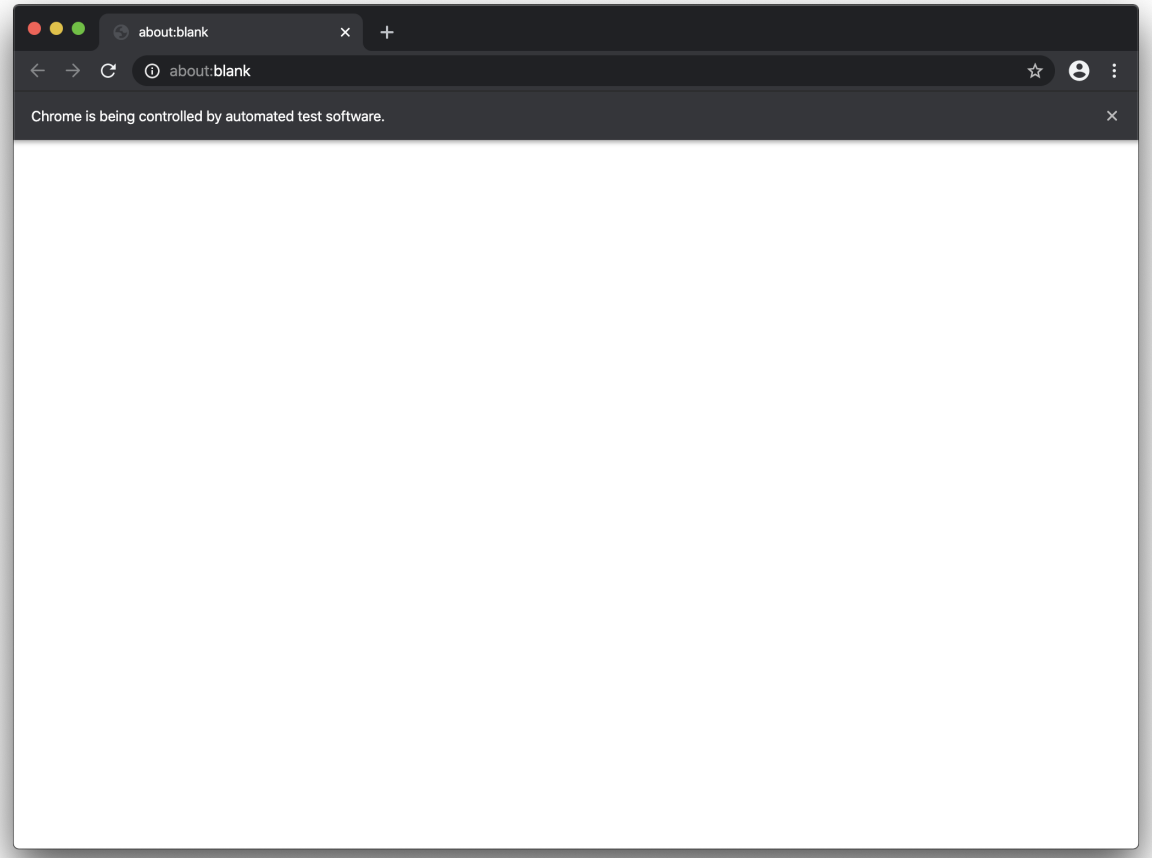

Figure 8. `openBrowser` *opens a new browser window with a single empty new tab.*

Every Taiko action assumes that you have an open, active browser window as the result of an `openBrowser` call. You'll also want to close the browser window at the end of your Taiko script by calling `closeBrowser`.

> **NOTE** If you are typing these examples yourself in the Taiko REPL, you can type `.code` to view the script output, or type `.code name-of-your-file.js` to save the code to a filename of your choice in the current working directory.

The script example shows you one way to structure your code in a standard JavaScript `try/catch/finally` block. The `finally` block ensures that the browser window closes at the end of the script run, regardless of whether the run was successful (`try`) or encountered errors along the way (`catch`).

NOTE

All Taiko actions are asynchronous. When running Taiko in a script outside of the REPL, be sure to mark the function as `async` and preceed each Taiko action with `await` to ensure that it has fully completed before the next Taiko action is called.

Open a Browser with a Specific Window Size

In the REPL

```
> openBrowser({args:['--window-size=1024,768']})
 ¬ Browser opened
```

In a script

```
const { openBrowser, closeBrowser } = require('taiko');
(async () => {
    try {
        await openBrowser({args:['--window-size=1024,768']});
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

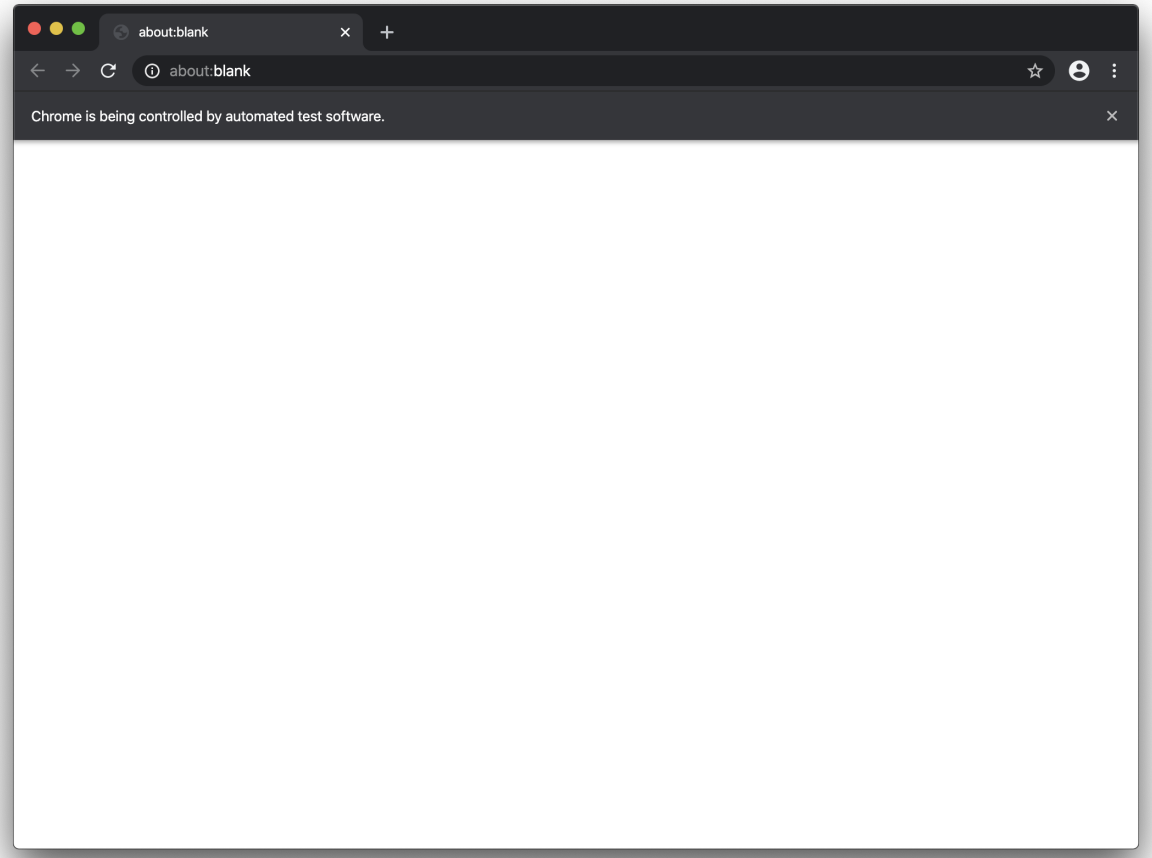

Figure 9. `openBrowser` *accepts any Chrome command line switches, including* `--window-size` *and* `--window-position`

If you are testing your website across multiple platforms (desktop, tablet, smartphone, smart TV, etc.), then you'll need the ability to test across multiple window sizes. The `openBrowser` action accepts a JSON argument with an array of `args`. Any command line switch that you'd normally pass into Chrome can be passed into `openBrowser` using the `args` array.

NOTE

You can pass in a comma-separated list of command line switches to `args`. For example, `openBrowser({args:['--window-size=1024,768', '--window-position=2048,0']})`. For a full list of Chrome command line switches, see https://peter.sh/experiments/chromium-command-line-switches/.

Goto a URL

In the REPL

```
> openBrowser()
 ¬ Browser opened
> goto('https://thirstyhead.com/conferenceworks/')
 value: {
  url: 'https://thirstyhead.com/conferenceworks/',
  status: { code: 200, text: '' }
}
```

In a script

```
const { openBrowser, goto, closeBrowser } = require('taiko');
(async () => {
    try {
        await openBrowser();
        await goto('https://thirstyhead.com/conferenceworks/');
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

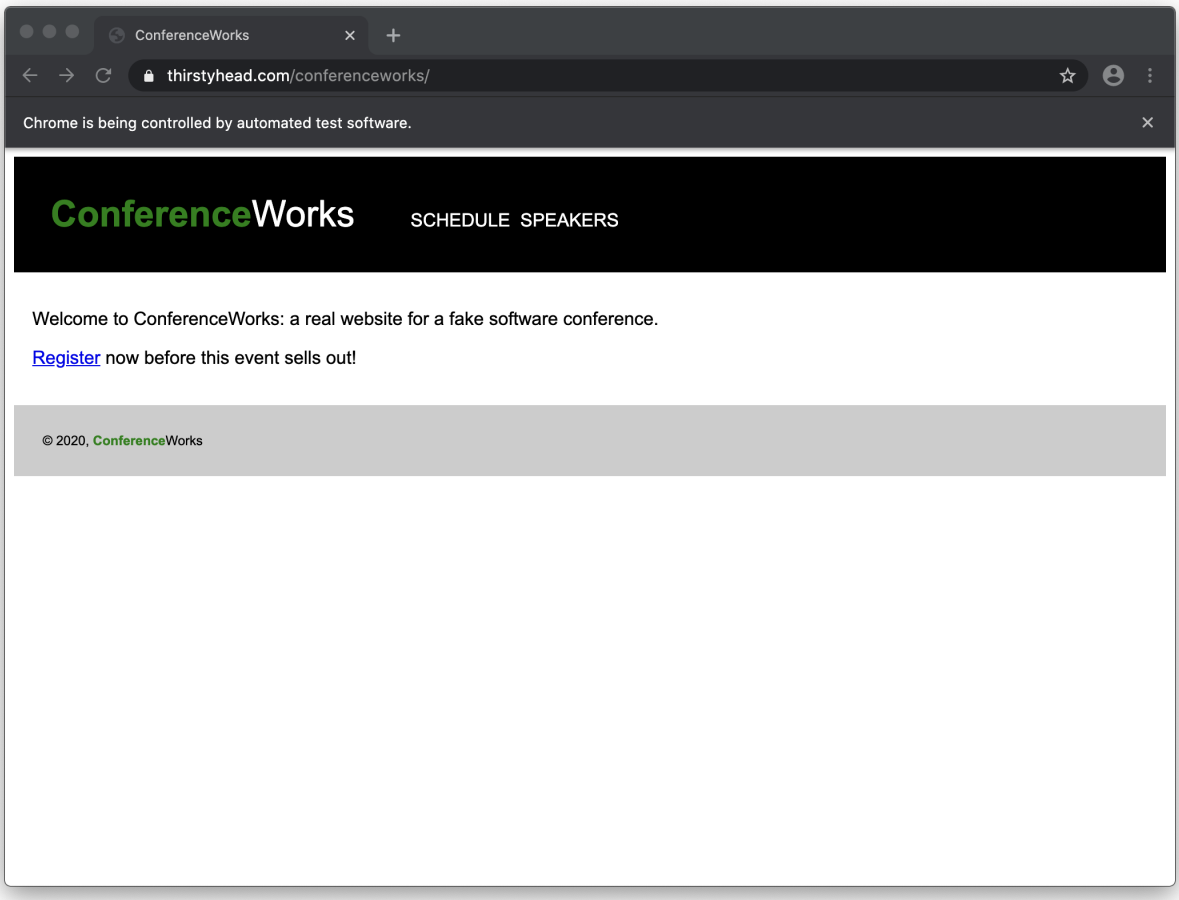

Figure 10. `goto(URL)` *visits the URL, just like a user typing the URL into the address bar.*

Once you have a browser window open, you'll almost certainly want to visit a website by using the `goto(URL)` action. This action returns a `value` object that contains the `url` you visited, as well as a `status` object that represents the HTTP response from the website.

The `goto(URL)` action accepts any partial URL fragment that the underlying browser does. For example, if you type `goto('thirstyhead.com/conferenceworks')`, notice that three separate HTTP GET requests are sent:

1. The first HTTP response is a `301` redirect to upgrade the request from an unsecure `http` address to a secure `https` one.
2. The second HTTP response is another `301` redirect, this time to include the trailing `/` in the URL (indicating that `conferenceworks` is a directory instead of a file).
3. The third HTTP response is a `200`, showing us the final successful HTTP request for the implicit `index.html` file in the `/conferenceworks/` directory.

`goto(URL)` *accepts URL fragments and follows HTTP redirects.*

```
> openBrowser()
 ¬ Browser opened
> goto('thirstyhead.com/conferenceworks')
 value: {
  redirectedResponse: [
    {
      url: 'http://thirstyhead.com/conferenceworks',
      status: { code: 301, text: 'Moved Permanently' }
    },
    {
      url: 'https://thirstyhead.com/conferenceworks',
      status: { code: 301, text: '' }
    }
  ],
  url: 'https://thirstyhead.com/conferenceworks/',
  status: { code: 200, text: '' }
}
```

NOTE	This series of `HTTP` redirects is the normal behavior of the Chromium browser, and of all browsers in general.

Click a Link

In the REPL

```
> openBrowser()
 ¬ Browser opened
> goto('https://thirstyhead.com/conferenceworks/')
 ¬ Navigated to URL https://thirstyhead.com/conferenceworks/
> click('Register')
 ¬ Clicked element matching text "Register" 1 times
> goBack()
 ¬ Performed clicking on browser back button
> goForward()
 ¬ Performed clicking on browser forward button
> click('Home')
 ¬ Clicked element matching text "Home" 1 times
```

In a script

```
const { openBrowser, goto, click, goBack, goForward, closeBrowser } =
require('taiko');
(async () => {
    try {
        await openBrowser();
        await goto('https://thirstyhead.com/conferenceworks/');
        await click('Register');
        await goBack();
        await goForward();
        await click('Home');
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

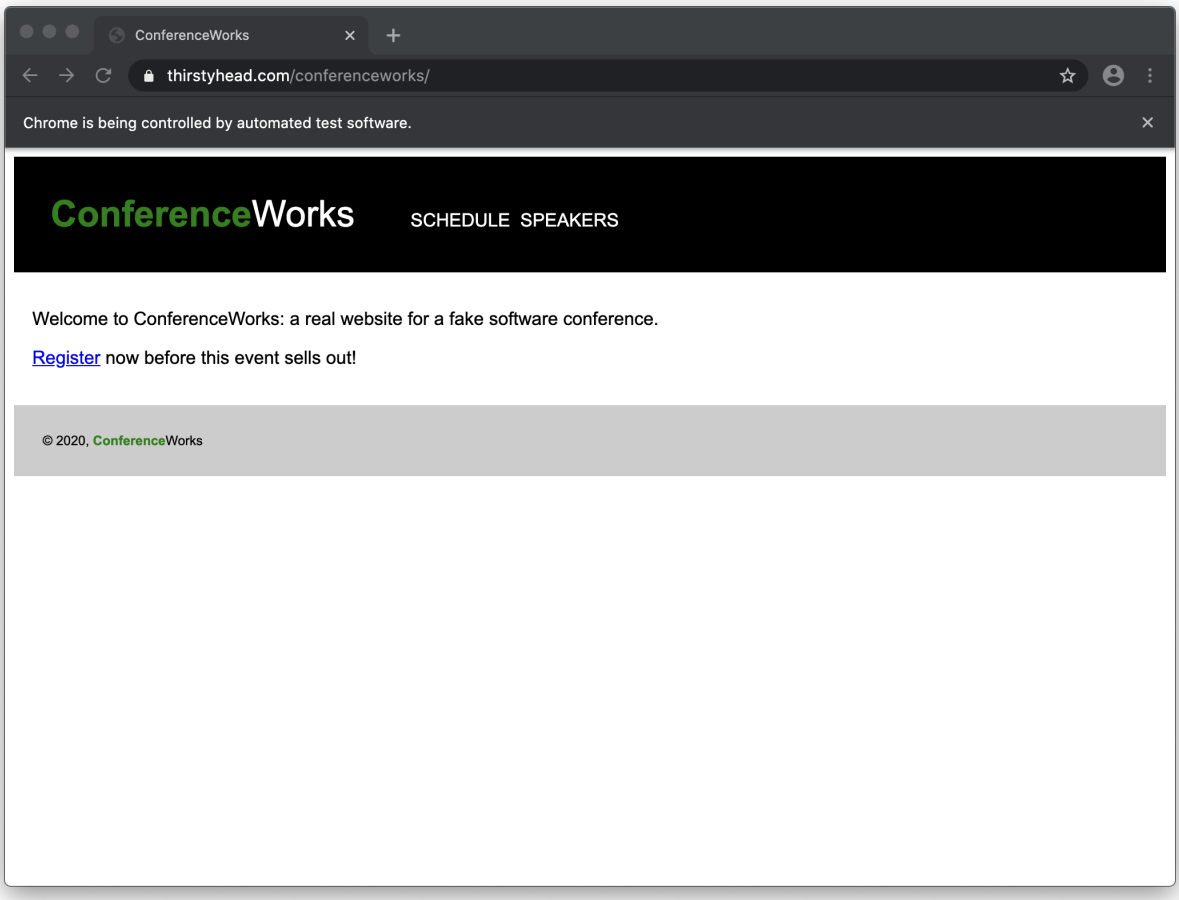

Figure 11. `click` *emulates a user clicking on a link like* `Register`*, or tabbing to it and pressing* `Enter`.

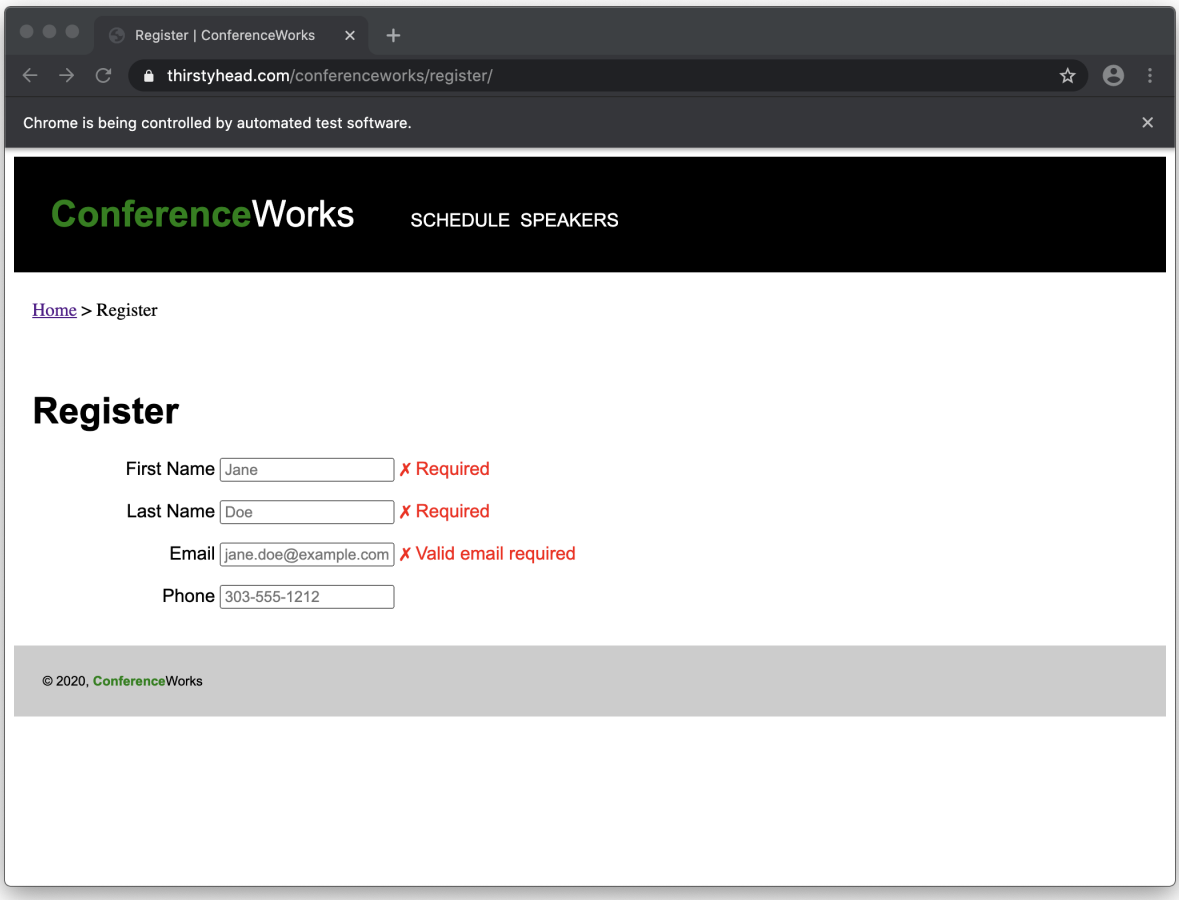

Figure 12. The new web page after the `Register` *link is clicked on the previous page.*

Using the `click(SELECTOR)` action emulates the user clicking on the selected element. You can also use the `goBack` and `goForward` actions to emulate the user clicking on the `Back` and `Forward` browser buttons.

NOTE

Taiko has a sophisticated Smart Selector algorithm that allows you to interact with the web page just like a user would by using *what the user sees on screen* rather than *what the web developer sees from a source code perspective*. While you can use detailed CSS or XPath selectors, that can lead to brittle tests if the underlying source code changes without changing the visible user experience.

For example, while `click('Register')` and `click($('body > main > p:nth-child(2) > a'))` are both functionally equivalent, the former is more readable, better represents the user's interaction with the web page, and ultimately will be more maintainable over time.

Semantic and Proximity Selectors

`click(SELECTOR)` eagerly matches the first item on the page. If you have multiple elements on the page — all with 'Register' as a visual indicator — the first thing you should do is re-evaluate your design. After that, you can refine your selector with semantic selectors like `click(link('Register'))` or `click(button('Register'))`.

Here's a list of semantic selectors:

- button
- checkBox
- color
- dropDown
- fileField
- image
- link
- listItem
- radioButton
- range
- tableCell
- text
- textBox
- timeField

Taiko also provides proximity selectors, like `toRightOf` and `below`. Here's a list of proximity selectors:

- above
- below
- toLeftOf
- toRightOf
- near

The Taiko actions `click('Register')` and `click(link('Register', toLeftOf(text('now before this event sells out'))))` are functionally equivalent.

Smart Selectors and Shadow DOM

Sometimes, a user can see an element on screen that isn't selectable programmatically by Taiko. A common example of this is when a web developer includes a Web Component that uses a Shadow DOM. As the name implies, a Shadow DOM is a separate DOM tree that is hidden from the main DOM, as well as any JavaScript outside of the Web Component. (For more information on Shadow DOM, see 'Using Shadow DOM' on MDN.)

The ConferenceWorks website uses a Web Component named `<cw-header>` to encapsulate and reuse the header across multiple pages. This header contains two links: `SCHEDULE` and `SPEAKERS`. Since Shadow DOM makes these links invisible to JavaScript outside of the Web Component, they are invisible to Taiko as well.

Shadow DOM elements are invisible to Taiko's Smart Selectors

```
> openBrowser()
 ¬ Browser opened
> goto('https://thirstyhead.com/conferenceworks/')
 ¬ Navigated to URL https://thirstyhead.com/conferenceworks/
> click('SPEAKERS')
 ¬ Error: Element with text SPEAKERS not found, run `.trace` for more
info.
> link('SPEAKERS').exists()
 value: false
 ¬ Does not exists
```

In this case, you can simply use `goto('https://thirstyhead.com/conferenceworks/speakers/')` in your script instead of attempting (and failing, due to the Shadow DOM contract with the browser) to click on the link programmatically.

Open and Close a Tab

In the REPL

```
> openBrowser()
 ¬ Browser opened
> goto('https://thirstyhead.com/conferenceworks/')
 ¬ Navigated to URL https://thirstyhead.com/conferenceworks/

> openTab()
 ¬ Opened tab with URL http://about:blank
> closeTab()
 ¬ Closed current tab matching about:blank

> const cwPageTitle = title()
> cwPageTitle
 value: 'ConferenceWorks'
> openTab('https://thirstyhead.com/groceryworks/')
 ¬ Opened tab with URL https://thirstyhead.com/groceryworks/
> const gwURL = currentURL()
> gwURL
 value: 'https://thirstyhead.com/groceryworks/'
> switchTo(cwPageTitle)
 ¬ Switched to tab matching ConferenceWorks
> closeTab(gwURL)
 ¬ Closing last target and browser.
```

In a script

```
const { openBrowser, goto, openTab, closeTab, title, currentURL,
switchTo, closeBrowser } = require('taiko');
(async () => {
    try {
        await openBrowser();
        await goto('https://thirstyhead.com/conferenceworks/');
        await openTab();
        await closeTab();
        const cwPageTitle = title();
        cwPageTitle;
        await openTab('https://thirstyhead.com/groceryworks/');
        const gwURL = currentURL();
        gwURL;
        await switchTo(cwPageTitle);
        await closeTab(gwURL);
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

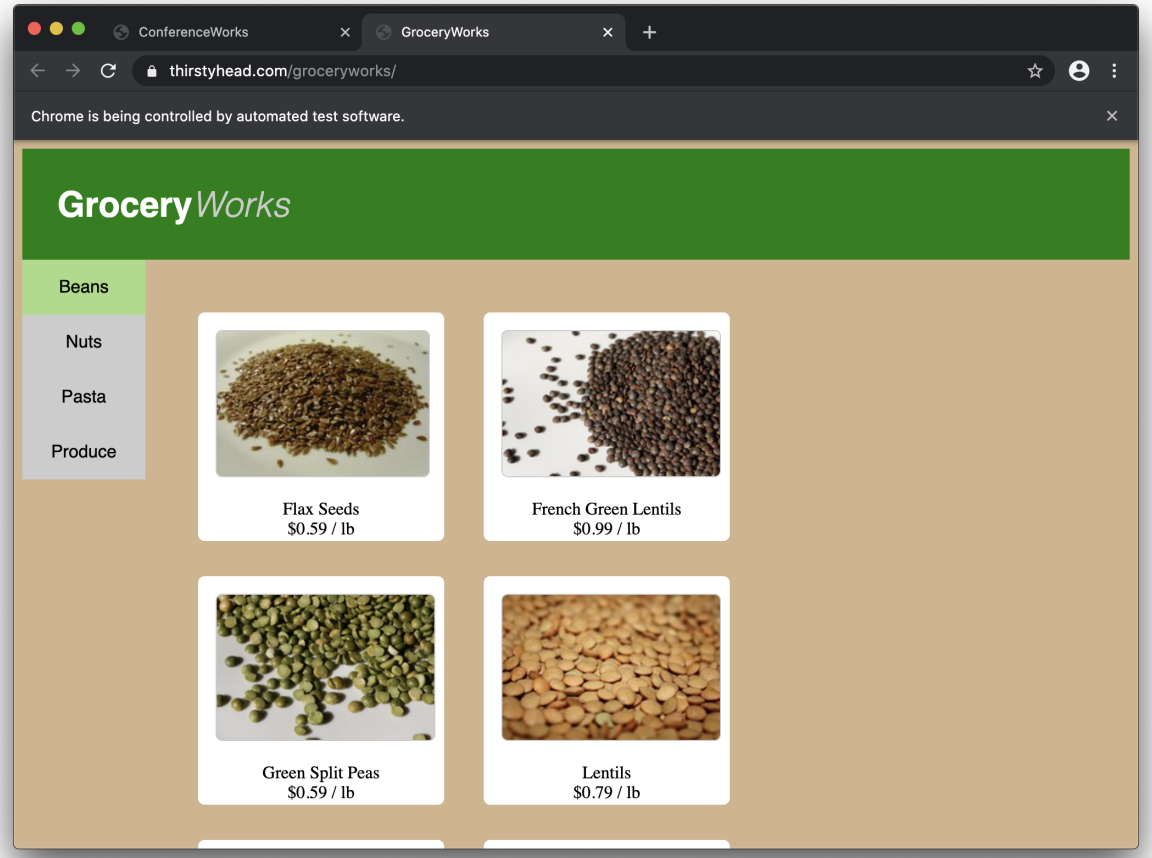

Figure 13. Taiko actions `openTab` *and* `closeTab` *allow you to open and close new browser tabs.*

As your app grows in complexity, your user might need to have multiple browser tabs open to accomplish certain tasks. The Taiko actions `openTab` and `closeTab` emulate the user opening and closing new tabs.

By default, `openTab()` opens a new, blank tab. If you'd like to open the tab to a specific URL, simply pass in the URL as an argument: `openTab('https://thirstyhead.com/groceryworks/')`.

As you begin working with tabs in Taiko, you'll quickly discover that being able to grab and store the `title()` of the tab and the `currentURL()` will be quite helpful. This is especially true when it comes to closing tabs. The Taiko action `closeTab()` closes the current tab, unless you pass in the target tab title `closeTab('GroceryWorks')` or the target tab URL `closeTab('https://thirstyhead.com/groceryworks/')`.

Open and Close an Incognito Window

In the REPL

```
> openBrowser()
 ¬ Browser opened
> openIncognitoWindow('https://thirstyhead.com/conferenceworks/',
                      {name:'New Incognito Window'})
 ¬ Incognito window opened with name New Incognito Window
> closeIncognitoWindow('New Incognito Window')
 ¬ Window with name New Incognito Window closed
```

In a script

```
const { openBrowser, openIncognitoWindow, closeBrowser,
closeIncognitoWindow } = require('taiko');
(async () => {
    try {
        await openBrowser();
        await openIncognitoWindow(
          'https://thirstyhead.com/conferenceworks/',
          {name:'New Incognito Window'});
        await closeIncognitoWindow('New Incognito Window');
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

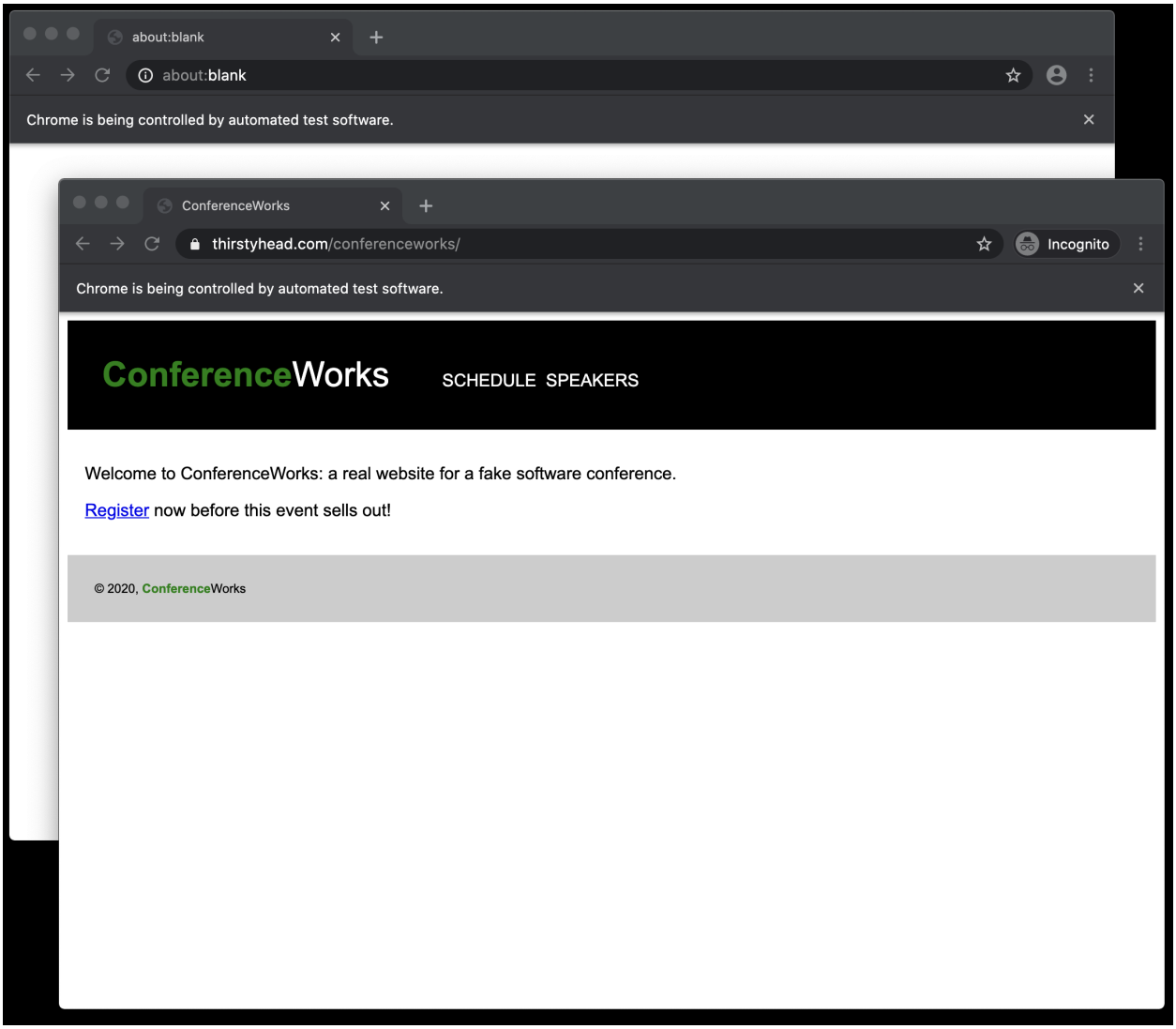

Figure 14. Open a new Incognito window with `openIncognitoWindow`.

The Taiko action `openIncognitoWindow` allows you to run your scripts in an Incognito window instead of a standard window. Two arguments are required to open a new Incognito window — a URL and a window name: `openIncognitoWindow('https://thirstyhead.com/conferenceworks/', {name:'New Incognito Window'})`.

The window name is especially important, because it is required to close an Incognito window: `closeIncognitoWindow('New Incognito Window')`.

You should probably store the window name in a constant or variable so that you can use it later to close the Incognito window.

Be sure to store the name of your new Incognito window so that you can close it later

```
> openBrowser()
 ¬ Browser opened

> const windowName = 'Private Window'
> const windowURL = 'https://thirstyhead.com/conferenceworks/'
> openIncognitoWindow(windowURL, {name:windowName})
 ¬ Incognito window opened with name Private Window
> closeIncognitoWindow(windowName)
 ¬ Window with name Private Window closed
```

Take a Screenshot

In the REPL

```
> openBrowser({args:['--window-size=1024,768']})
 ¬ Browser opened
> goto('https://thirstyhead.com/conferenceworks/')
 ¬ Navigated to URL https://thirstyhead.com/conferenceworks/
> click('Register')
 ¬ Clicked element matching text "Register" 1 times

> screenshot({path:'form-before-entry.png'})
 ¬ Screenshot is created at form-before-entry.png
> click('First Name')
 ¬ Clicked element matching text "First Name" 1 times
> write('Suzi')
 ¬ Wrote Suzi into the focused element.
> click('Last Name')
 ¬ Clicked element matching text "Last Name" 1 times
> write('Q')
 ¬ Wrote Q into the focused element.
> click('Email')
 ¬ Clicked element matching text "Email" 1 times
> write('suzi@q.org')
 ¬ Wrote suzi@q.org into the focused element.
> click('Phone')
 ¬ Clicked element matching text "Phone" 1 times
> write('3035551212')
 ¬ Wrote 3035551212 into the focused element.
> screenshot({path:'form-after-entry.png'})
 ¬ Screenshot is created at form-after-entry.png
```

In a script

```
const { openBrowser, goto, click, screenshot, write, closeBrowser } =
require('taiko');
(async () => {
    try {
        await openBrowser({args:['--window-size=1024,768']});
        await goto('https://thirstyhead.com/conferenceworks/');
        await click('Register');
        await screenshot({path:'form-before-entry.png'});
        await click('First Name');
        await write('Suzi');
        await click('Last Name');
        await write('Q');
        await click('Email');
        await write('suzi@q.org');
        await click('Phone');
        await write('3035551212');
        await screenshot({path:'form-after-entry.png'});
    } catch (error) {
        console.error(error);
    } finally {
        await closeBrowser();
    }
})();
```

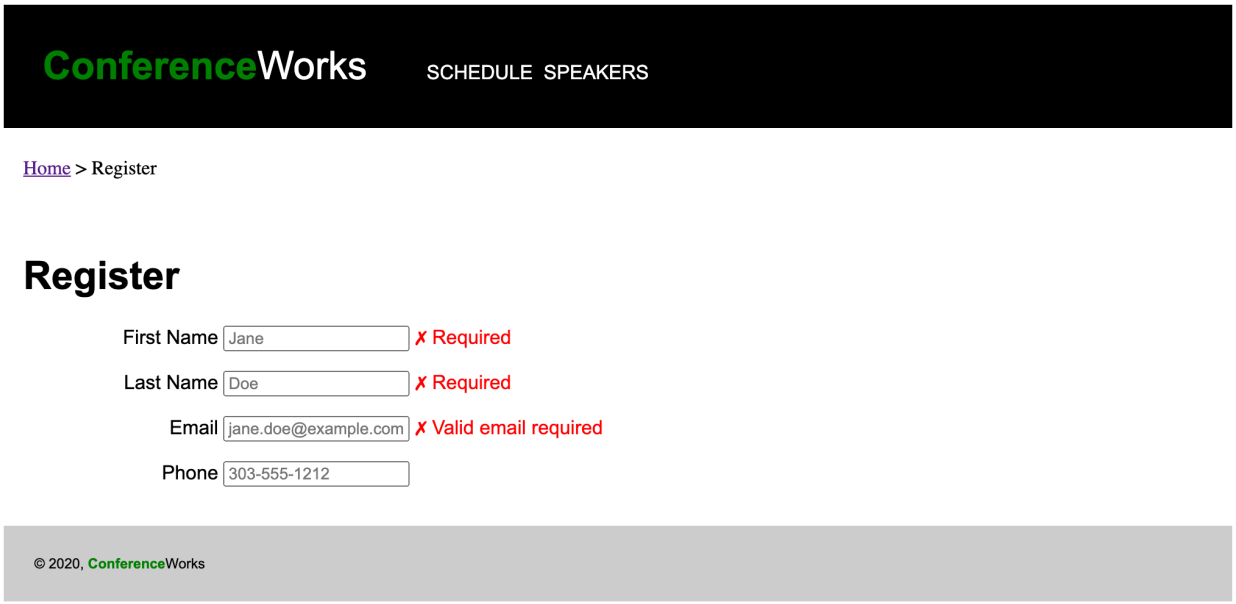

Figure 15. `form-before-entry.png` *captured using Taiko action* `screenshot()`

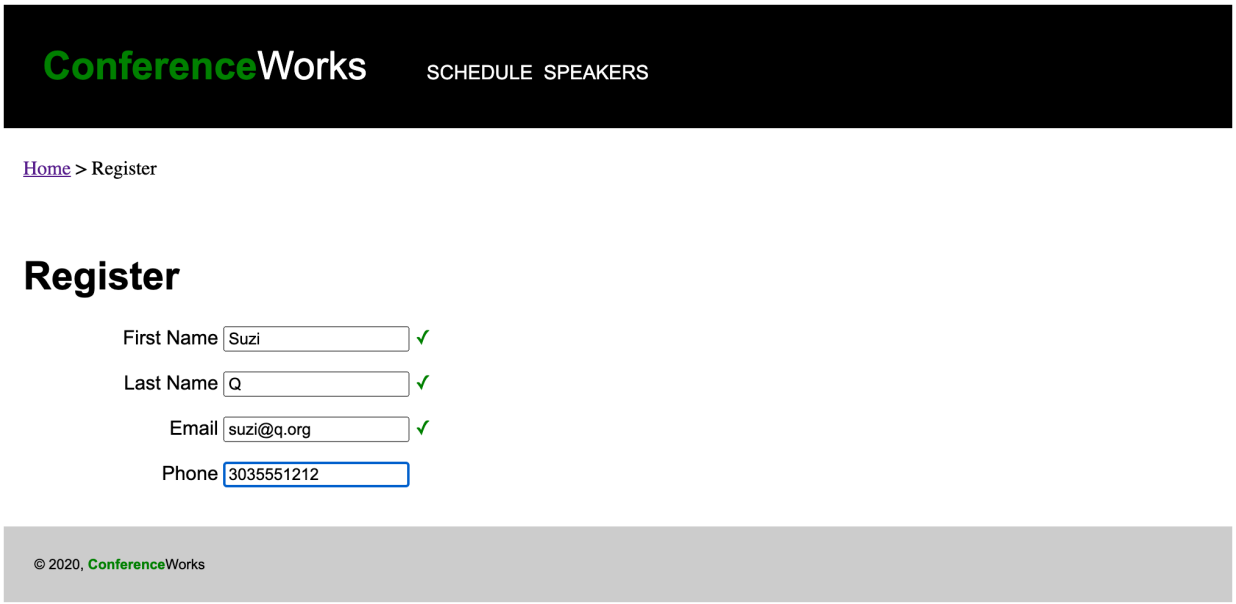

Figure 16. `form-after-entry.png` *captured using Taiko action* `screenshot()`

The ability to capture screenshots at key points in your Taiko script helps illustrate the User Journey you are automating. The `screenshot()` action with no arguments creates a PNG image in the current directory named `Screenshot-1589490638953.png`. The last half of the filename is a timestamp.

You'll almost certainly want to give your screenshot a more descriptive name, like `screenshot({path:'form-before-entry.png'})` or `screenshot({path:'form-after-entry.png'})`. In this example, we are capturing a screenshot of an HTML form before data entry begins, and then another screenshot after data entry is complete.