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

> *The human is more important than the technology. Make things more human.*
>
> — Steve "Woz" Wozniak

> *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 mindset. 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.

Because, as you'll see in just a moment, there were a number of bugs lurking in their user

interface — bugs that were trivial and obvious to uncover when they simply tested the application by using it as their users would. After months of successfully passing unit tests, the team was surprised by how many undiscovered bugs they still had when they applied a new avenue of testing.

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.

The team of developers were already savvy unit testers, so they were excited and intrigued to tackle a new kind of testing that played to their strengths while gently broadening their perspective.

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 `CalculateSalesTax` function is 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. An automated collection of passing user journey tests is your measure of success and a measurable indication of the overall health of your website. (No more hurriedly clicking through your website by hand in a mad rush the last moment before you showcase the software in a customer demo.)

A good user journey testing library 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.

User Journey Testing in Action

The conscientious team of software developers that you met at the beginning of this story decided to try out a new browser automation library — one that is specifically built for user

journey testing — called Taiko. The easy installation — `npm install -g taiko` — and the light, expressive JavaScript-based DSL made it easy to learn and explore. The Taiko REPL — an interactive command-line environment that contains documentation and encourages you to explore Taiko in a live setting — helped the team experiment with Taiko and save their experiments as actual tests that can be run outside of the REPL environment. Taiko dramatically reduces the developer cost of writing user journey tests, and markedly reduces the test brittleness as well.

Here's what a simple "Buy the ingredients for a spaghetti dinner" user journey test looks 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. This functionality had already passed numerous unit tests, but none of the unit tests caught this particular bug because they were testing the shopping cart in isolation instead of testing it through the user interface as their users would use it.

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 testing -- and how easy it is! -- we know that our website's user experience will be as rich and bug-free as the developer experience."

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

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

+ 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()
 &check; 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();
    }
})();
```

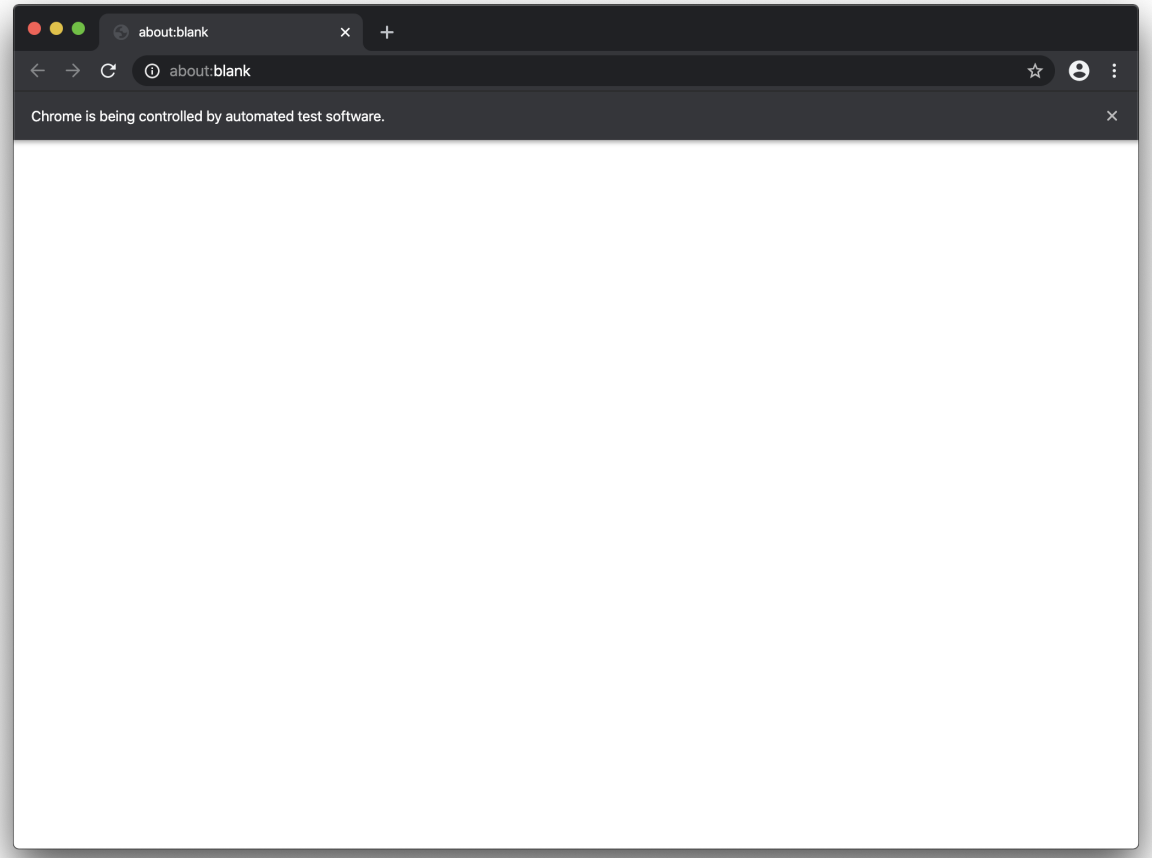

Figure 1. `openBrowser` *opens a new browser window with a single empty new tab.*

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();
    }
})();
```

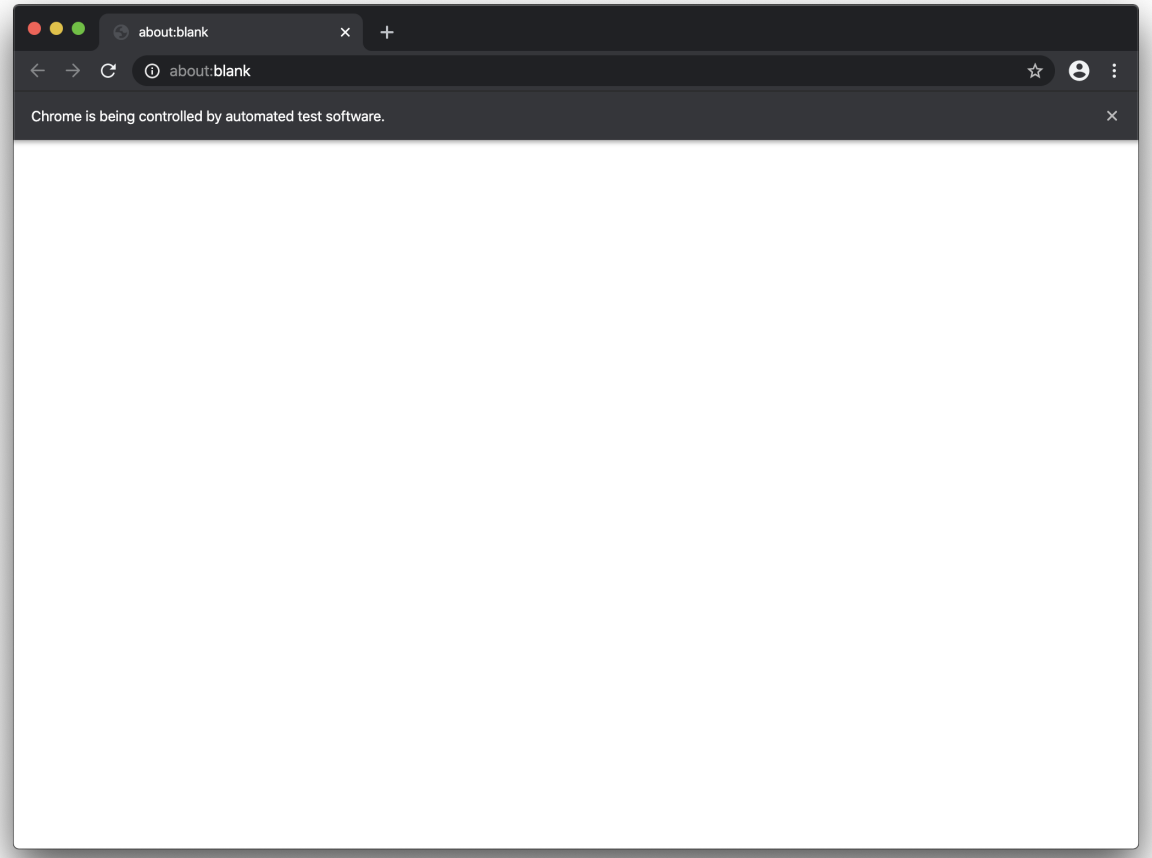

Figure 2. `openBrowser` *accepts any Chrome command line switches, including* `--window-size` *and* `--window-position`

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();
    }
})();
```

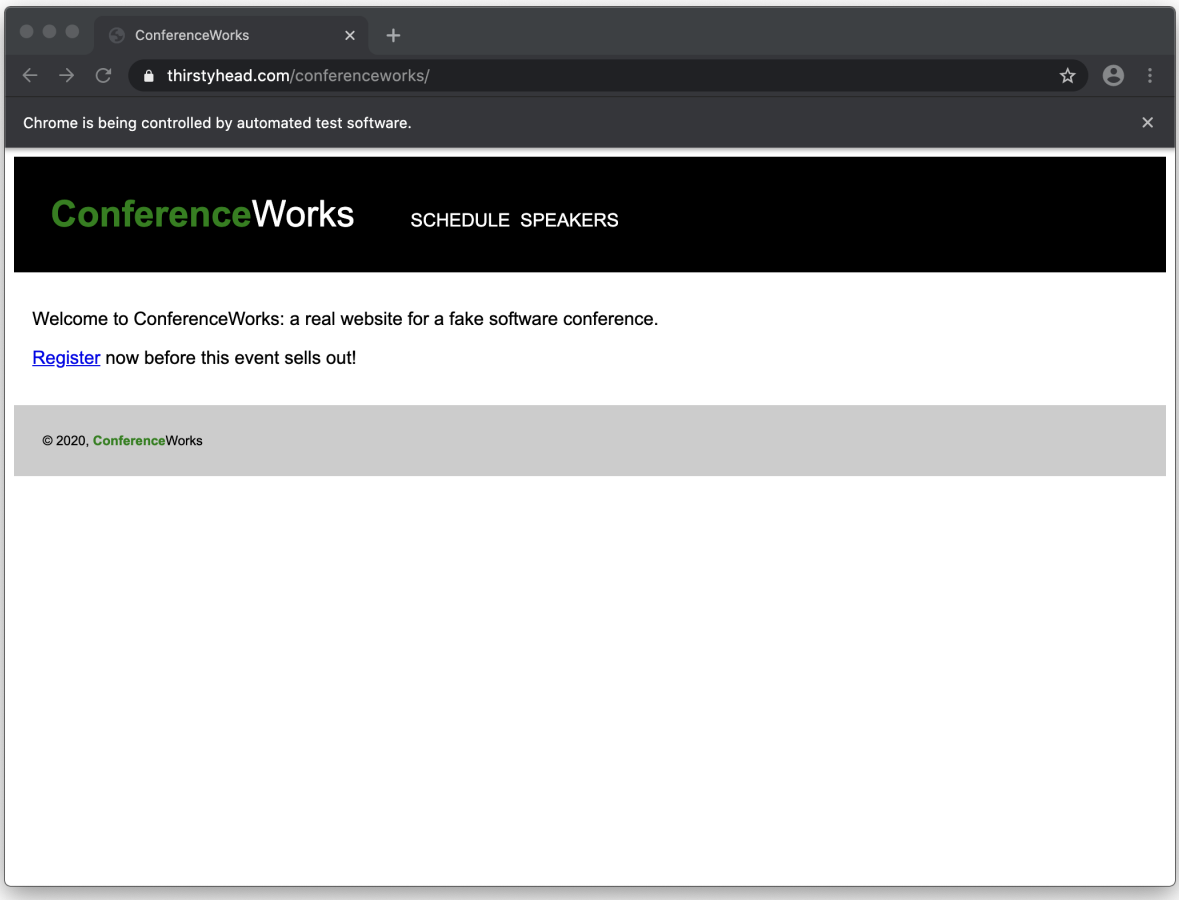

Figure 3. `goto(URL)` *visits the URL, just like a user typing the URL into the address bar.*

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();
    }
})();
```

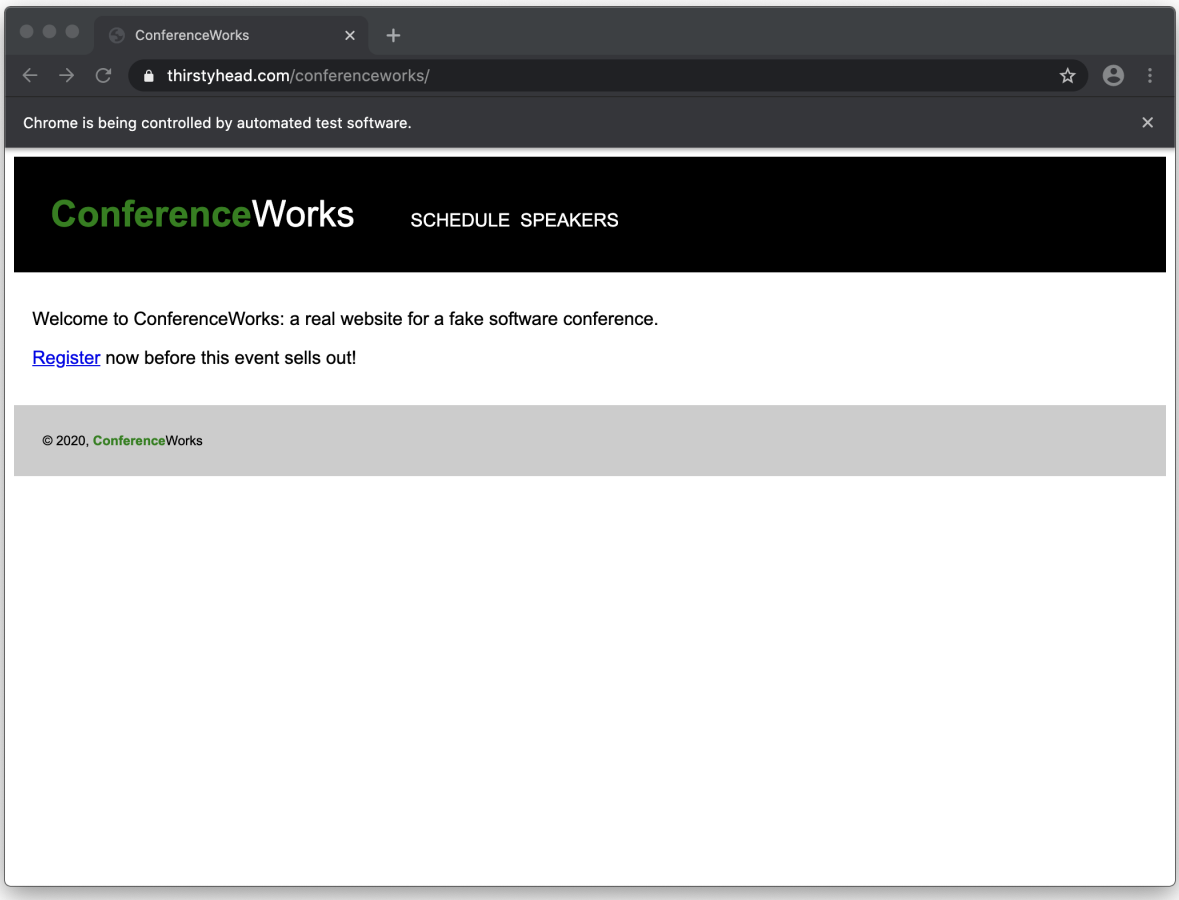

Figure 4. `click` *emulates a user clicking on a link like* `Register`*, or tabbing to it and pressing* `Enter`.

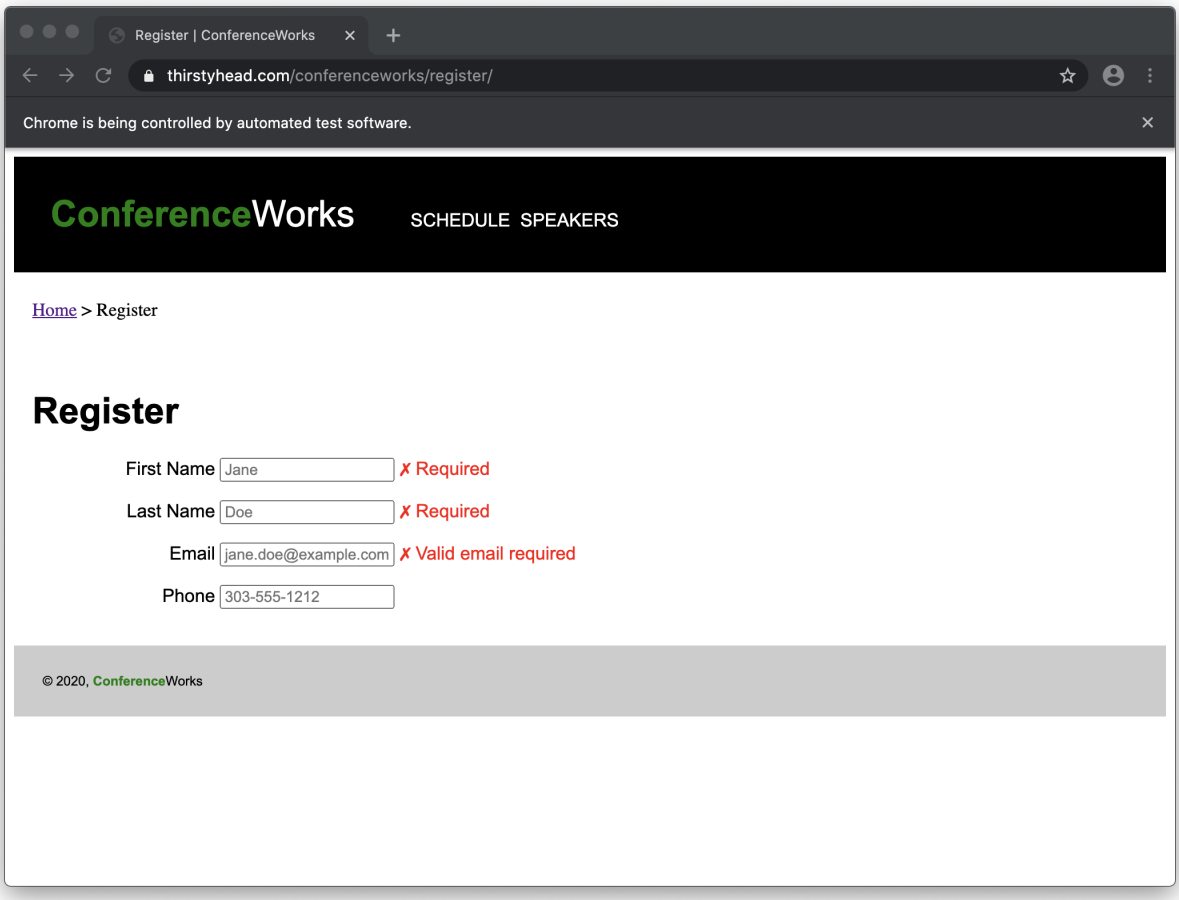

Figure 5. The new web page after the `Register` *link is clicked on the previous page.*

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();
    }
})();
```

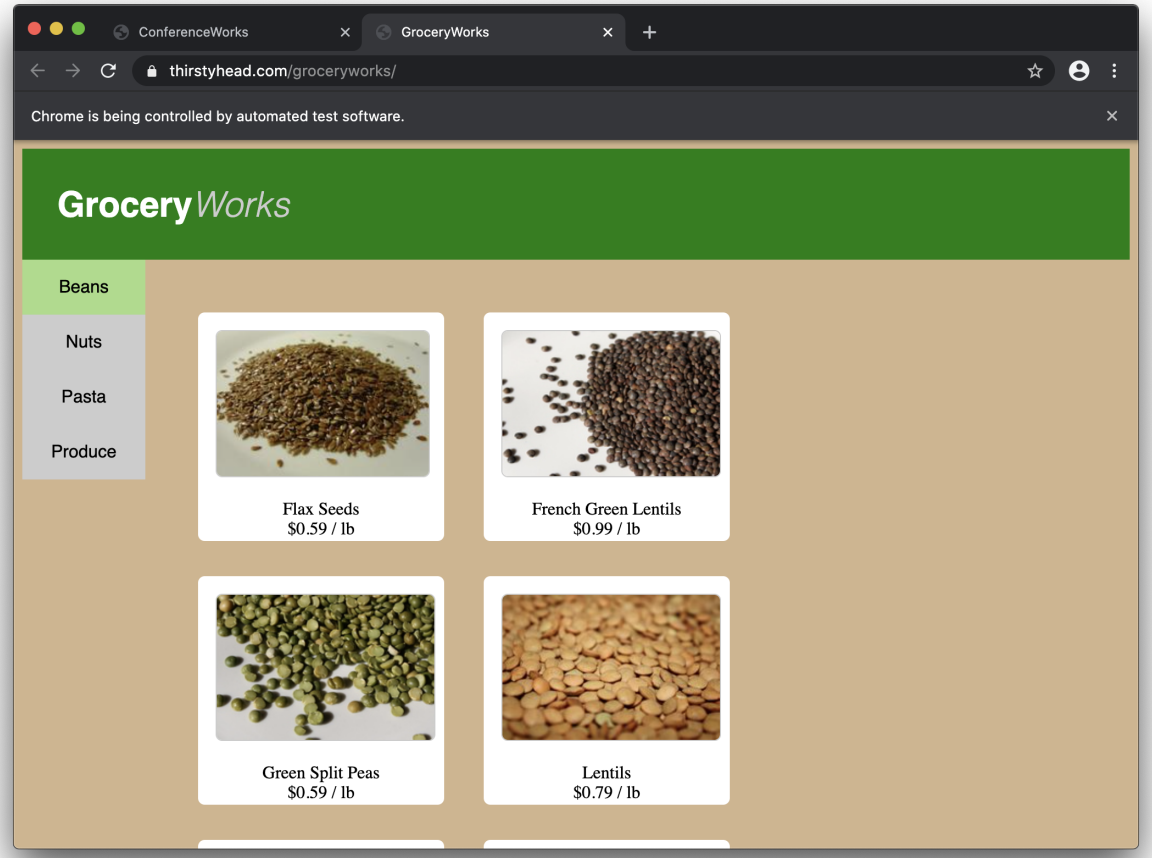

Figure 6. Taiko actions `openTab` *and* `closeTab` *allow you to open and close new browser tabs.*

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();
    }
})();
```

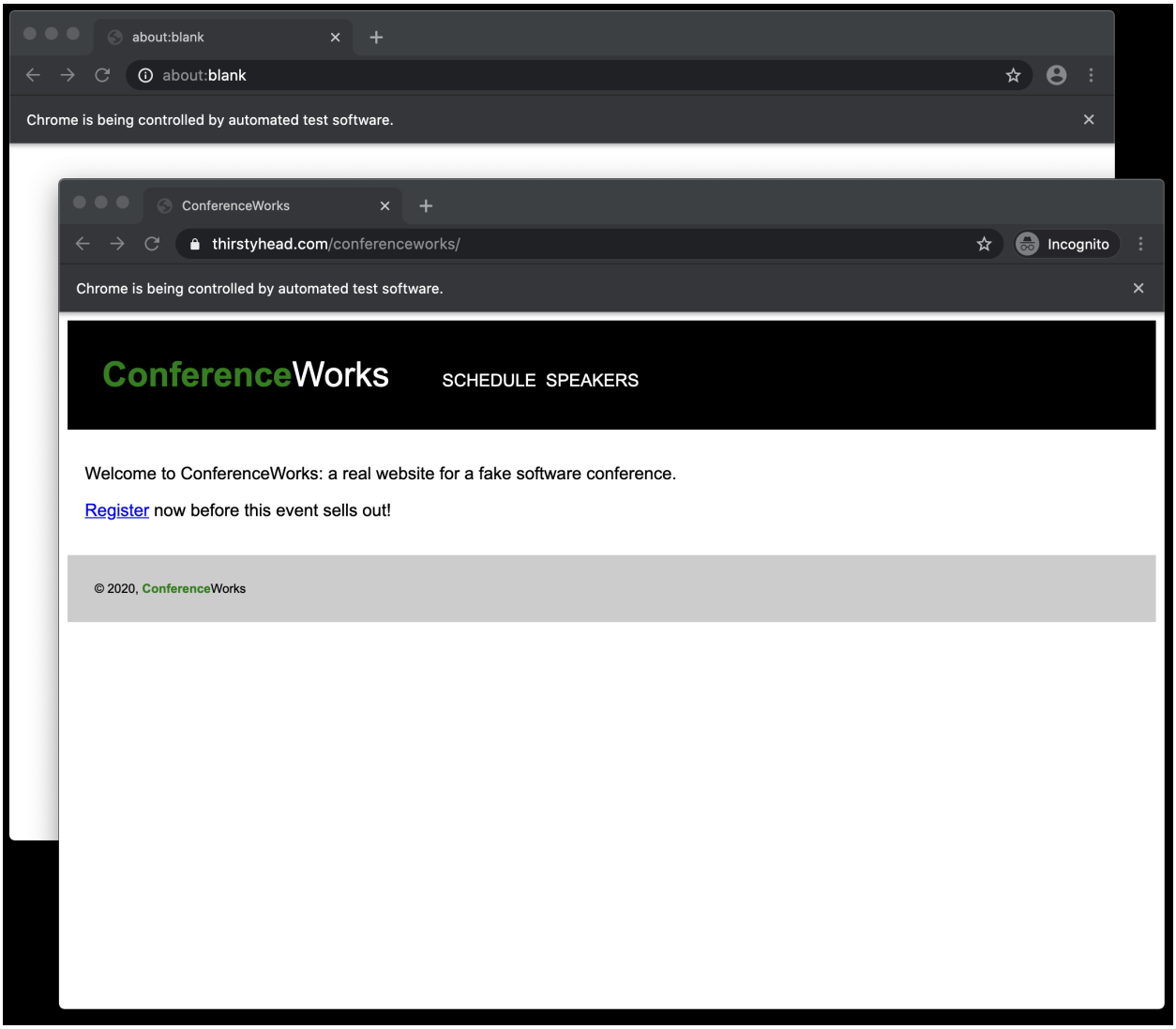

Figure 7. Open a new Incognito window with `openIncognitoWindow`.

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();
    }
})();
```

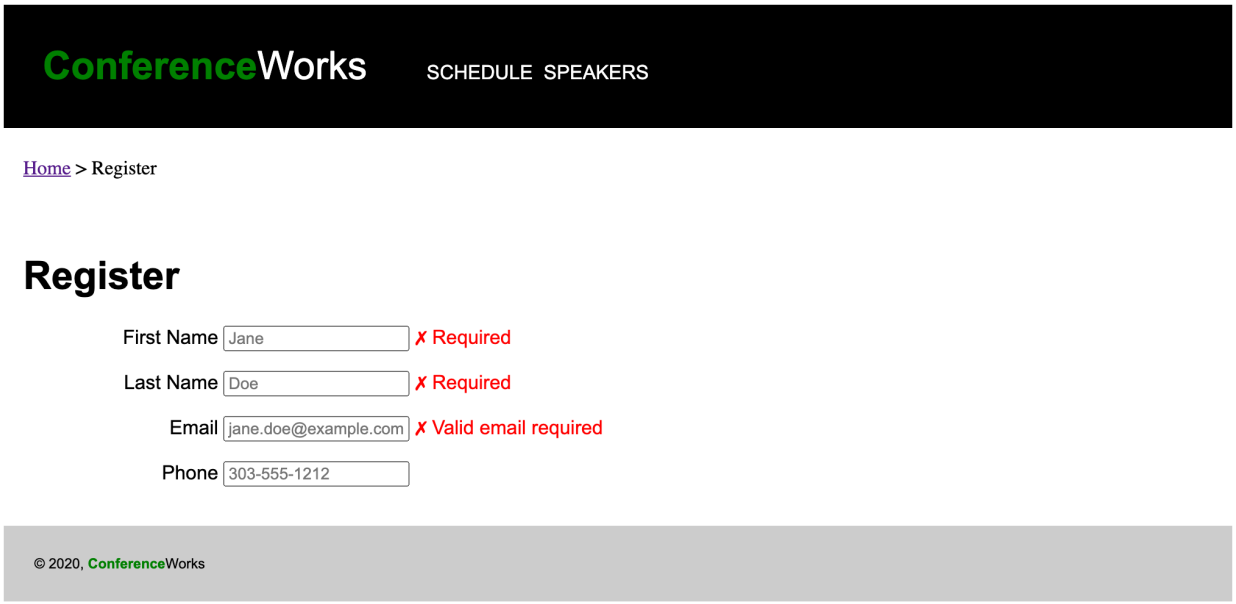

Figure 8. `form-before-entry.png` *captured using Taiko action* `screenshot()`

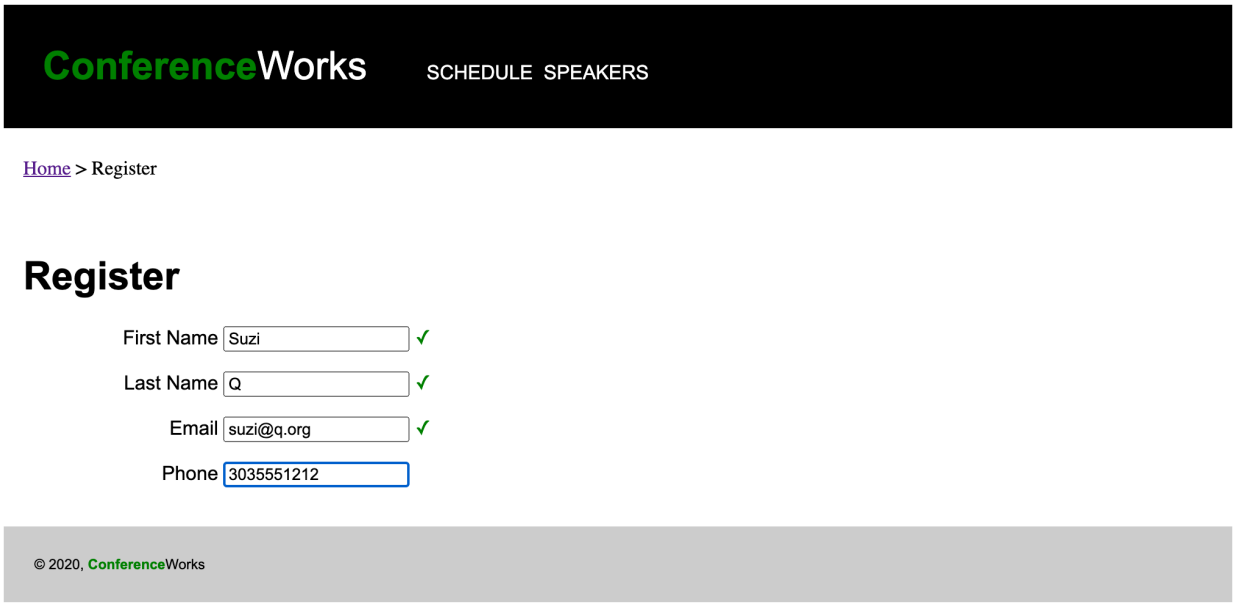

Figure 9. `form-after-entry.png` *captured using Taiko action* `screenshot()`

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.