TAIKO

A Code-First Approach

Scott Davis

taiko.dev

Table of Contents

Introduction

Taiko is a JavaScript-based Domain Specific Language (DSL) for automatically driving your web browser just like a typical user does. If a user goes to your website, clicks on a link, fills in some form fields, and clicks the "submit" button, you can script up that behavior in Taiko and replay it in a reliable, automated way.

What is User Journey Testing?

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?

So then, 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*.

I now 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'})
```

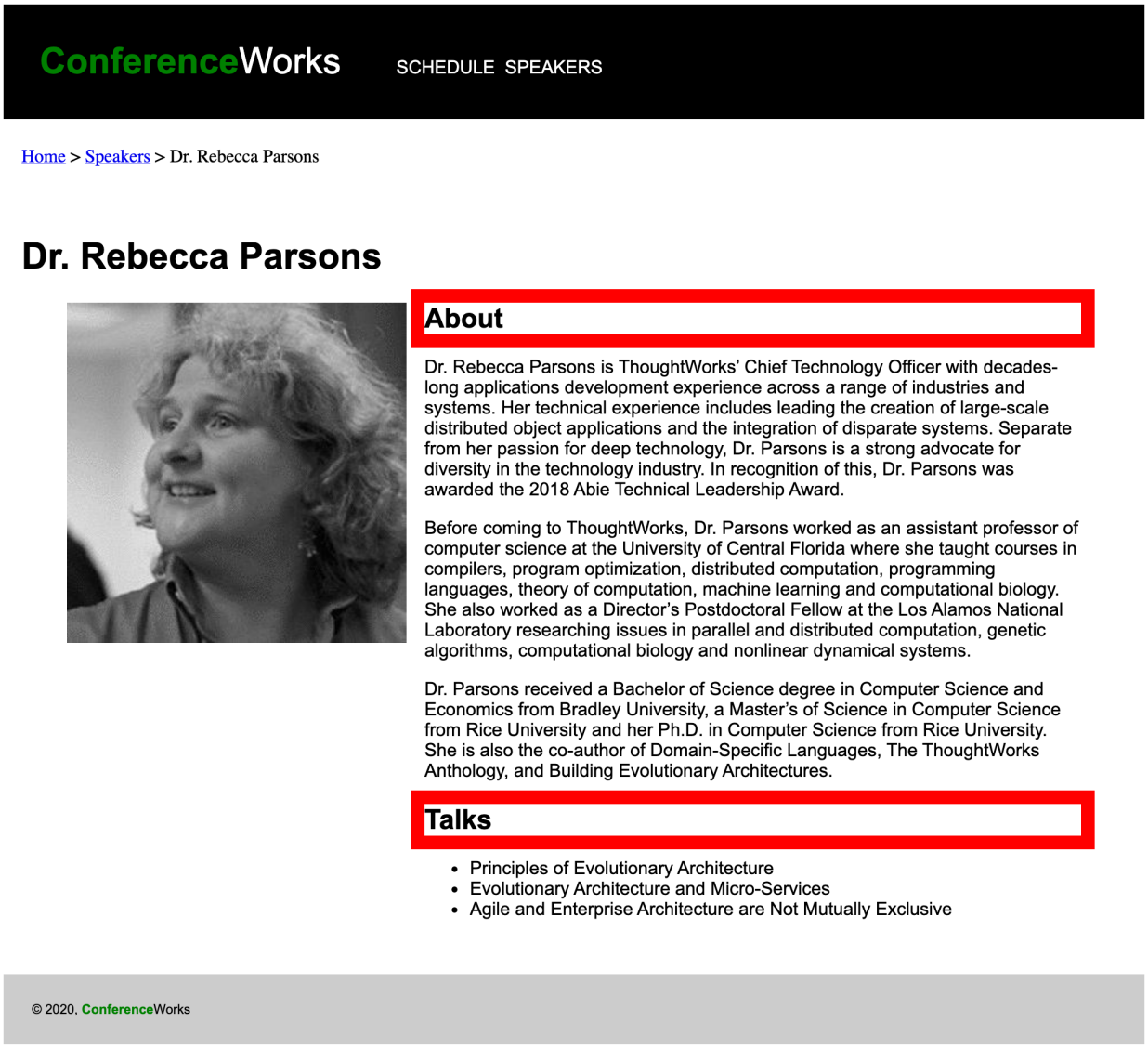

Figure 1. The resulting screenshot from the User Journey test

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 `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.

Where do User Journey Tests live on the Testing Spectrum?

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*.

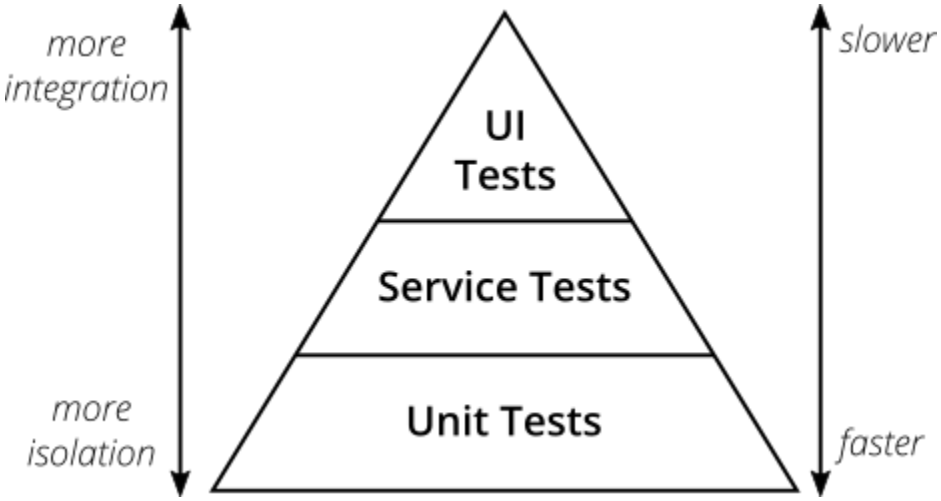

Figure 2. Mike Cohn's Test Pyramid, from Succeeding with Agile

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.

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 absolute 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".

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.

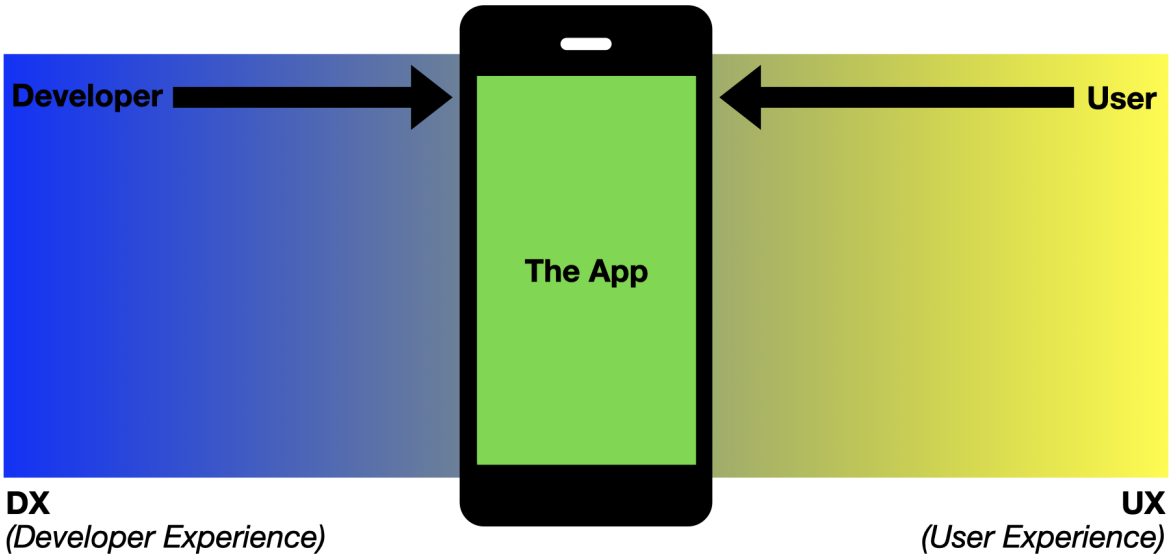

Figure 3. A new visual metaphor for software development that places the app as the center of focus

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

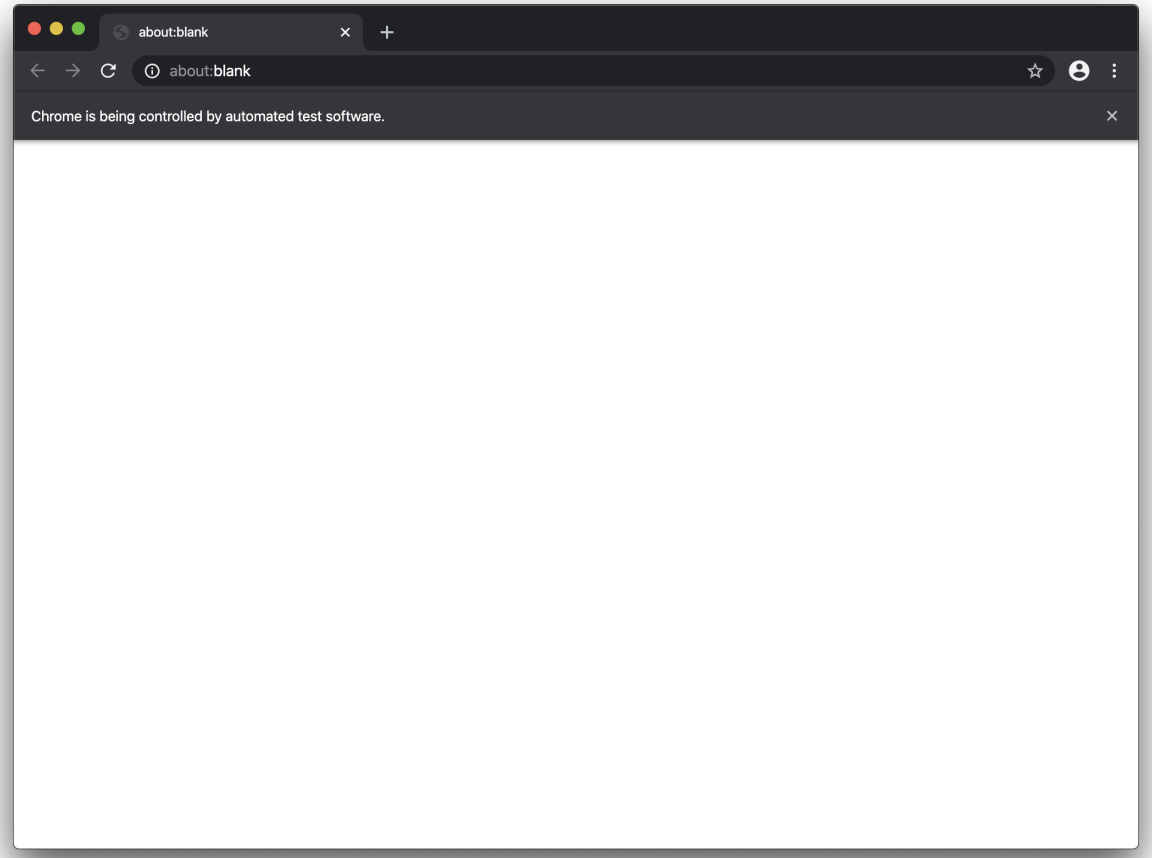

Figure 4. `openBrowser` *opens a new browser window with a single empty new tab.*

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

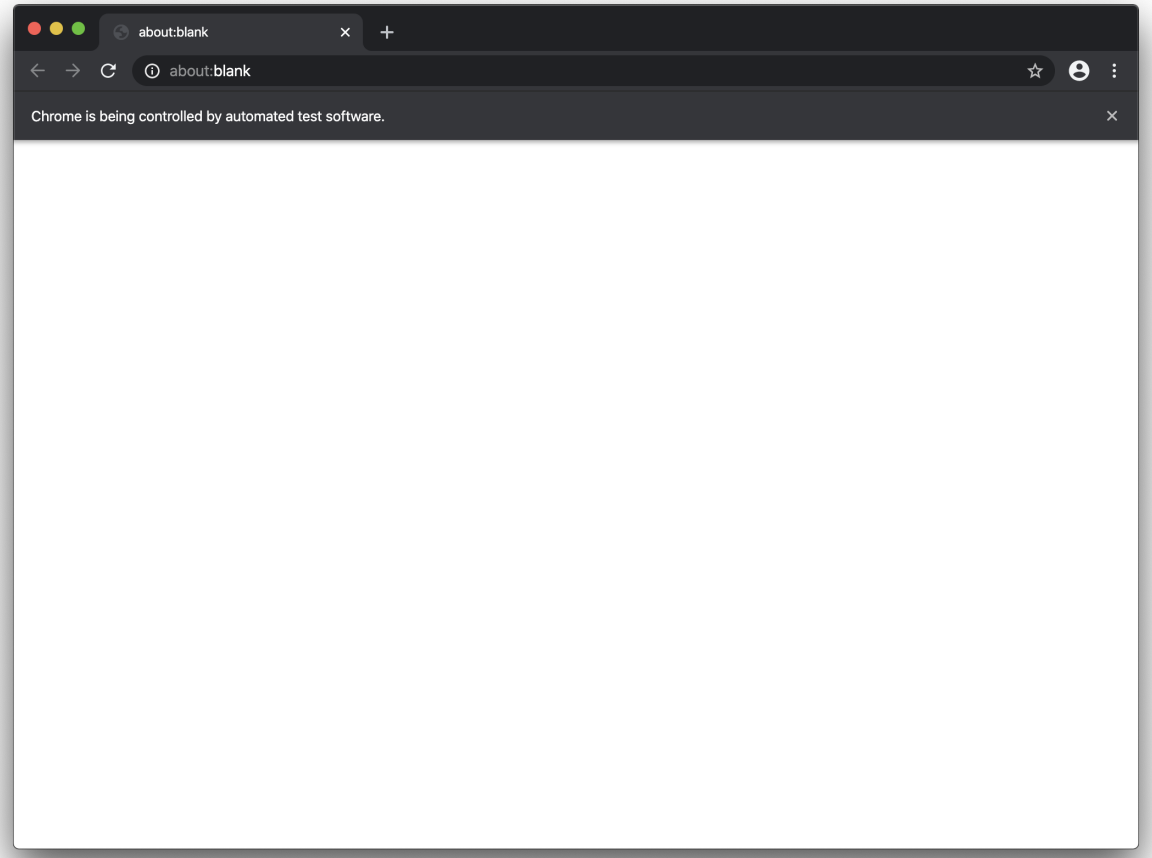

Figure 5. `openBrowser` *accepts any Chrome command line switches, including* `--window-size` *and* `--window-position`

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

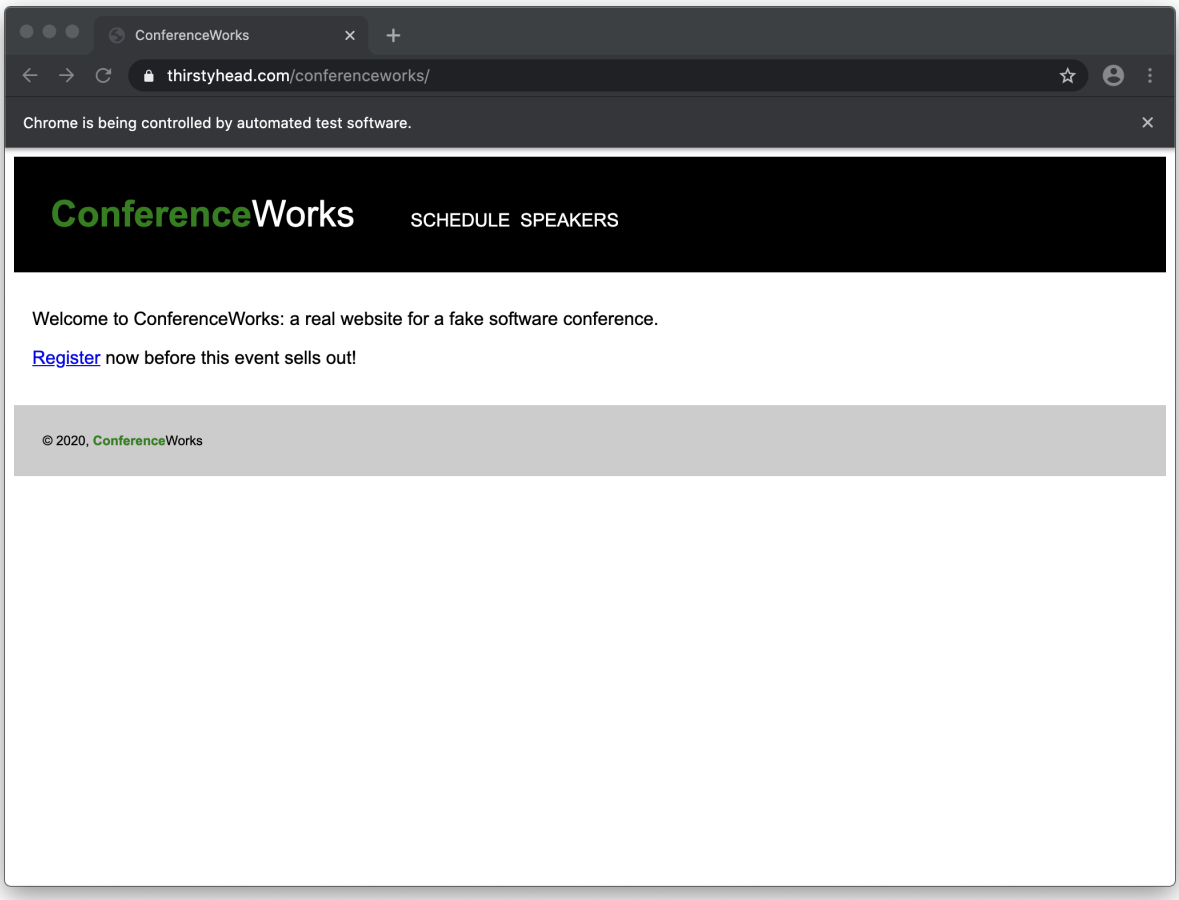

Figure 6. `goto(URL)` *visits the URL, just like a user typing the URL into the address bar.*

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

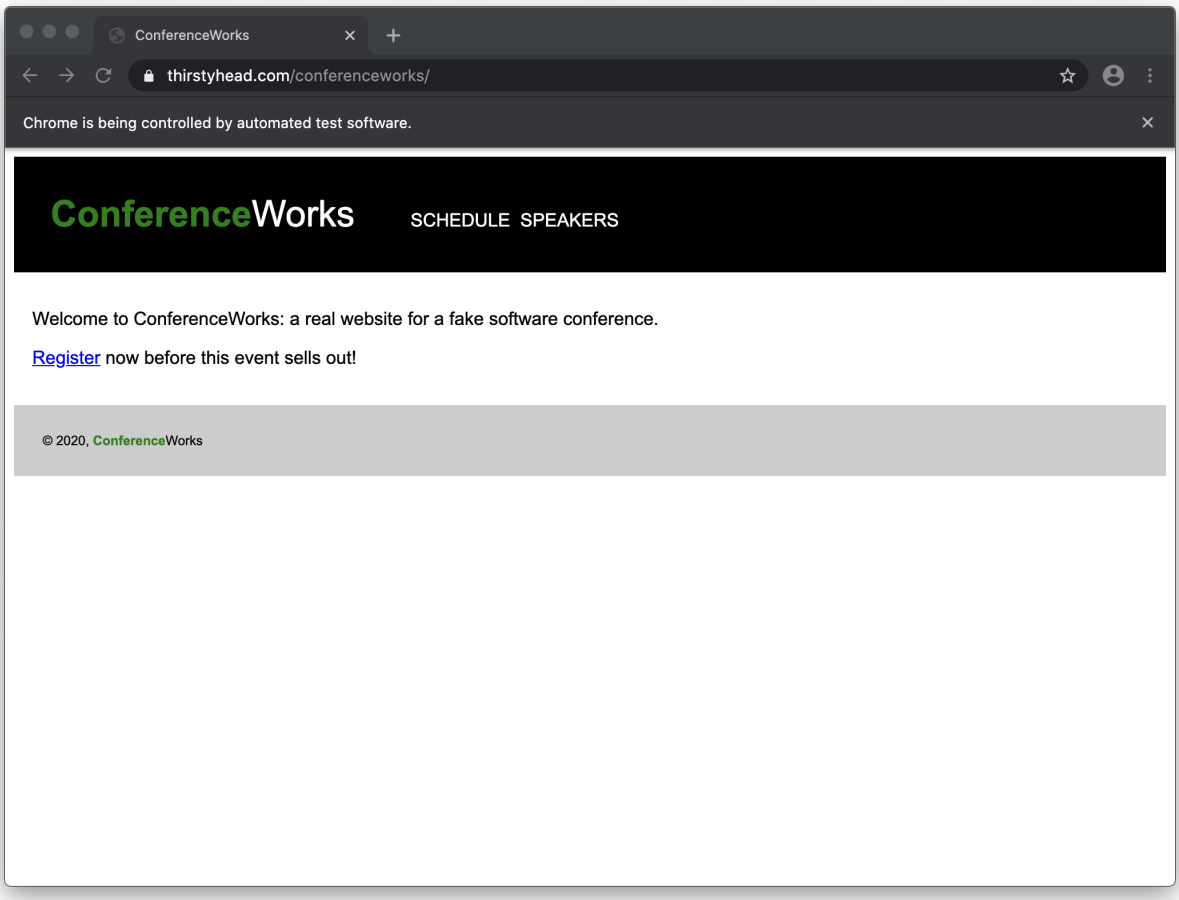

Figure 7. `click` *emulates a user clicking on a link like* `Register`*, or tabbing to it and pressing* `Enter`.

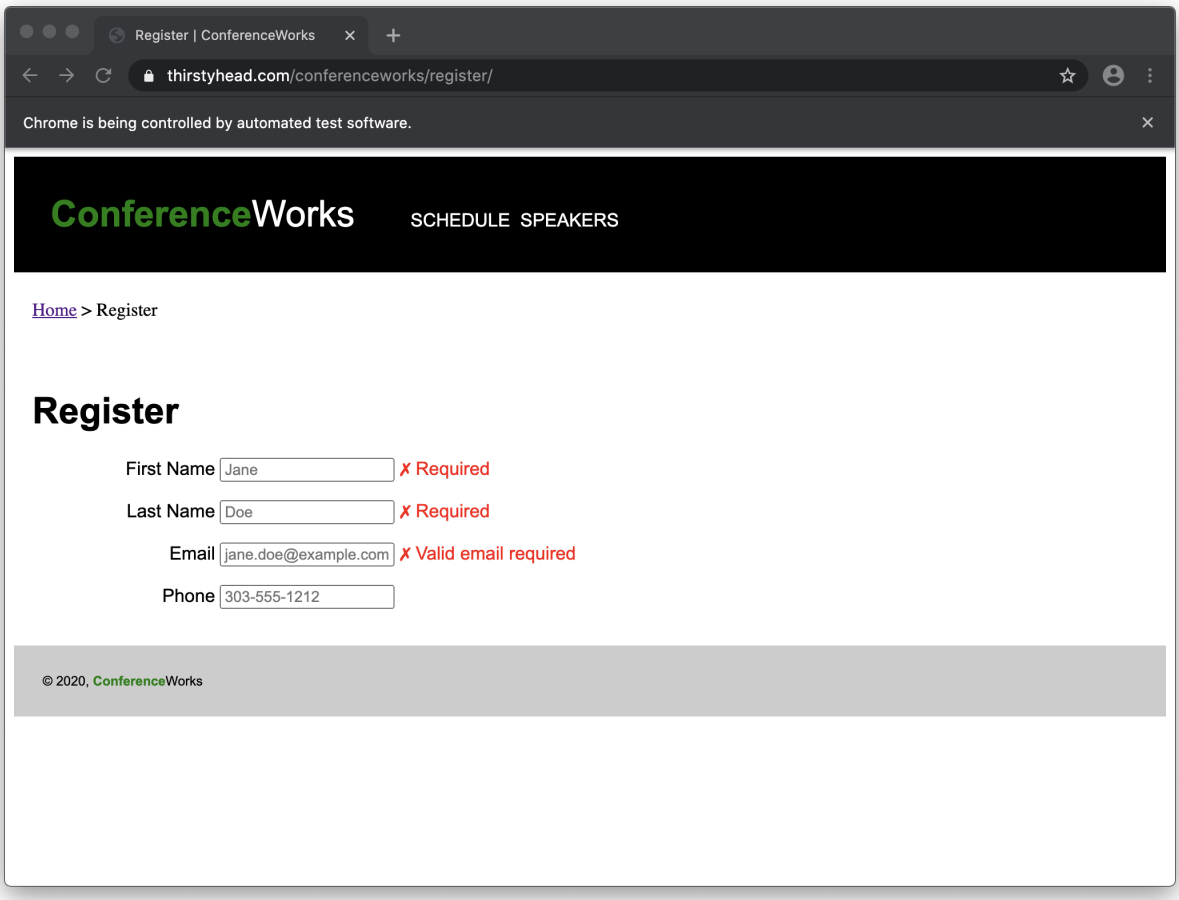

Figure 8. The new web page after the `Register` *link is clicked on the previous page.*

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

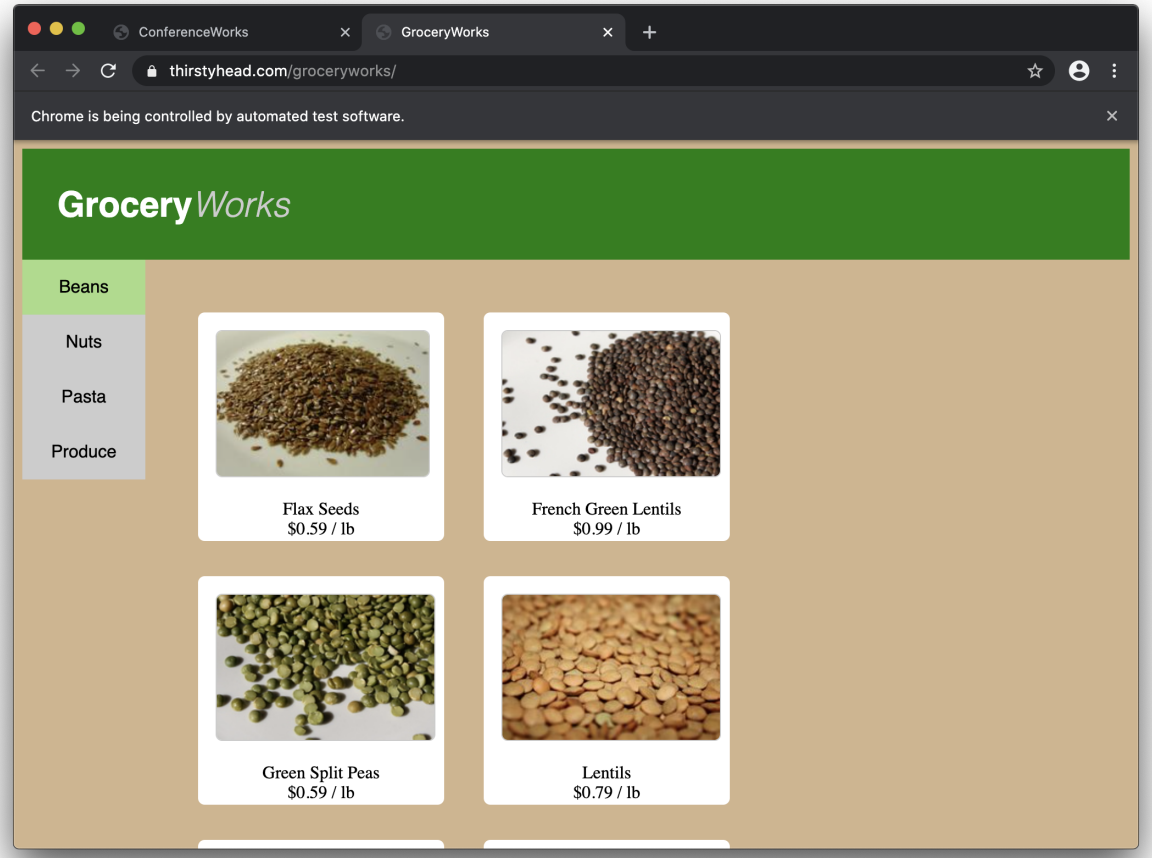

Figure 9. Taiko actions `openTab` *and* `closeTab` *allow you to open and close new browser tabs.*

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

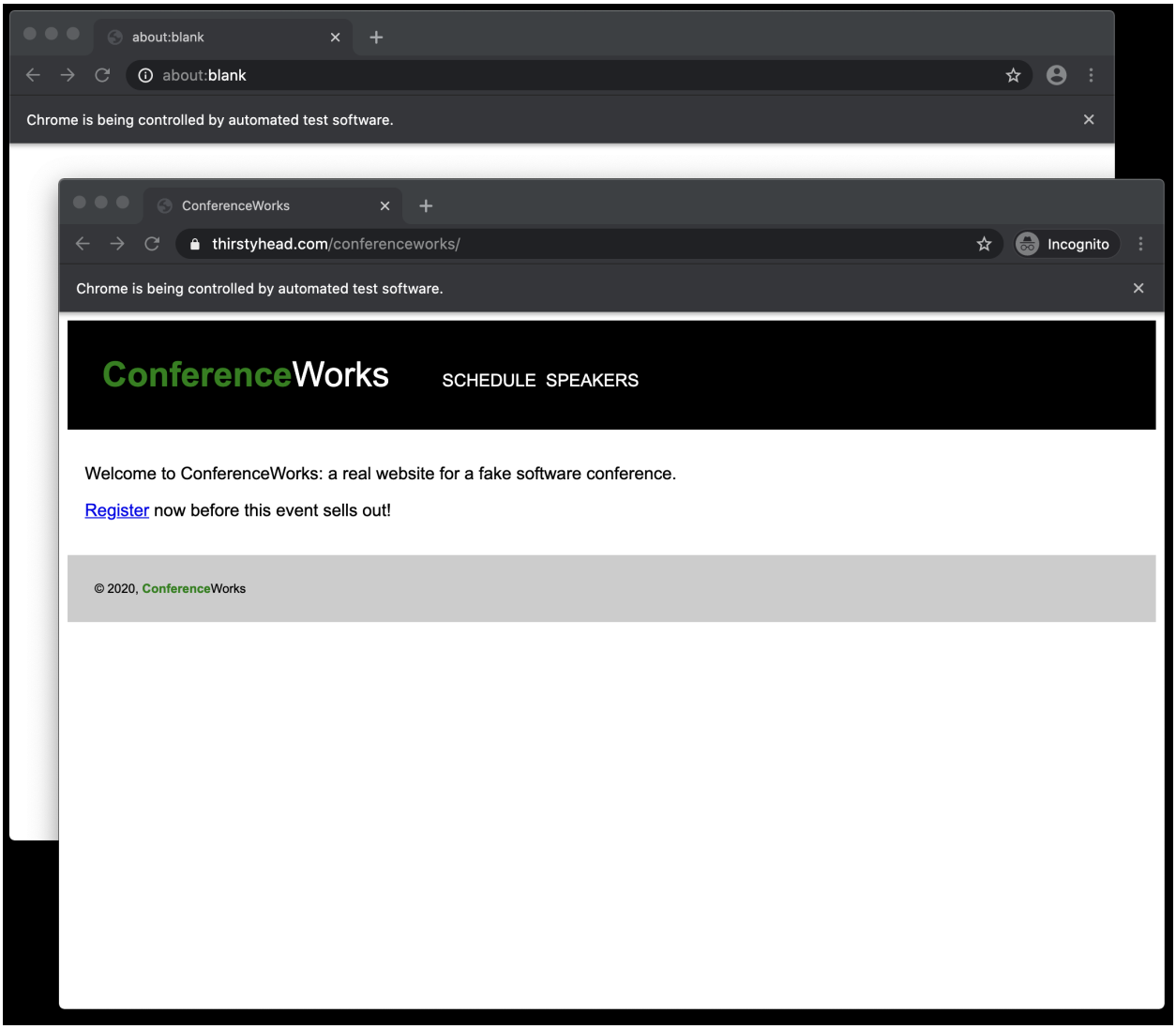

Figure 10. Open a new Incognito window with `openIncognitoWindow`.

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

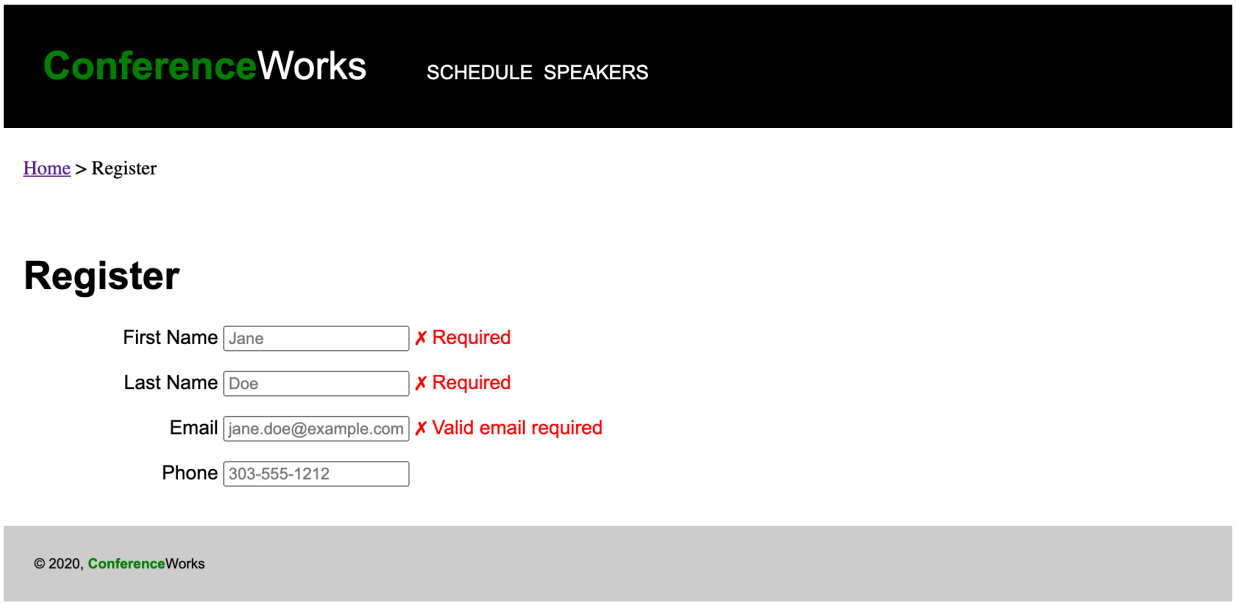

Figure 11. `form-before-entry.png` *captured using Taiko action* `screenshot()`

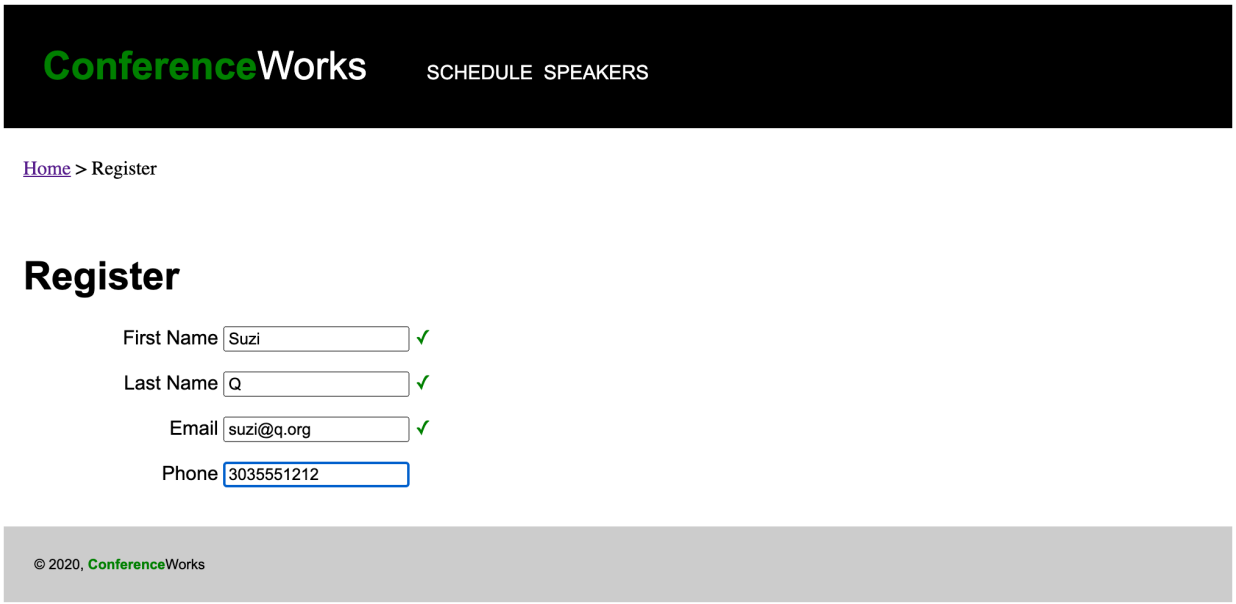

Figure 12. `form-after-entry.png` *captured using Taiko action* `screenshot()`

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.

Working with Forms

In this chapter, you'll learn how to work with forms.

Write in a Text Field

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
```

Taiko makes it easy to interact with HTML forms. You can `click()` on `<label>` elements, `write()` into the associated `<input>` fields, and then `press('Tab')` to move to the next field. To submit the form, you can `click('Submit')` (because the button is an `<input type='submit'>` element), `click(button('Register'))` to more accurately model what the user sees on the screen in this example, or even `press('Enter')`.

If Taiko doesn't interact with your HTML forms in the same way as shown here, it's worth understanding how this example was built. Chances are you might be using a web framework that behaves differently than the standards-based, browser-native, pure HTML and CSS example presented here. Another possibility is your HTML elements might be lacking some of the semantics that modern browsers expect you to be using.

The `Register` page of ConferenceWorks offers a simple HTML form with four text input fields and a submit button (labeled `Register` in this example). The text input fields take advantage of HTML's native declarative field validation, so no JavaScript is necessary to ensure that required fields are populated if the `required` attribute is present on the `<input>` elements. Declarative CSS rules like `input:required:valid` and `input:required:invalid` show and hide the corresponding `<span>` elements next to the `<input>` fields that contain the validation messages.

Using the `required` *attribute on* `<input>` *elements in coordination with* `input:required:valid` *and* `input:required:invalid` *CSS rules*

```
<style>
  .valid{
    color: green;
    display: none;
  }

  .invalid{
    color: red;
    display: none;
  }

  input:required:valid + .field-message .valid{
    display: inline;
  }

  input:required:invalid + .field-message .invalid{
    display: inline;
  }
</style>

<p>
  <label for="firstname">First Name</label>
  <input id="firstname"
         name="firstname"
         type="text"
         placeholder="Jane"
         required>
  <span class="field-message">
    <span class="valid">&check;</span>
    <span class="invalid">&cross; Required</span>
  </span>
</p>
```

The `First Name` and `Last Name` fields are simple `<input type='text'>` fields. The `Email` field layers on a bit of semantics (and additional validation) by specifying `<input type='email'>`. This ensures that the field conforms to the most basic requirements of an email address: `user@hostname`. If you'd like to enforce a stricter validation rule for email addresses — for instance, requiring a `.com` or `.org` at the end of the address — you can provide a `pattern` attribute on the `<input>` field with a declarative Regular Expression (RegEx).

The `Password` field is a text field with a different set of semantics and behavior. By using `<input type='password'>`, the browser masks all input with dots instead of the actual

text of the password, and doesn't allow the `value` to be copied out of the field and pasted somewhere else to reveal the secret content. Operating system-based password managers, as well as third-party password managers, all hook into password fields based on the semantic `<type='password'>` attribute.

If clicking on your `<label>` element doesn't change the focus to the corresponding `<input>` element, chances are good the two aren't associated with each other. For example, well-written, semantically correct, accessible HTML form fields link the `< label>` to the `<input>` using the `for` attribute of the `<label>` and the `id` of the `<input>`.

Associating a label with an input field using the `for` *attribute*

```
<label for="firstname">First Name</label>
<input id="firstname" name="fname" type="text" placeholder="Jane"
required>
```

*Click a Checkbox or a Radio Button

```
checkBox
radioButton
```

*Select from a Dropdown

```
dropDown
```

*Upload a File

```
fileField
attach
```

*Adjust a Time Field

```
timeField
```

*Pick from a Color Field

```
color
```

*Adjust a Range Field

```
range
```

Performing Mouse and Tap Actions

In this chapter, you’ll learn how to perform mouse and tap actions.

*Click or Tap

```
click
tap
```

*Doubleclick

```
doubleClick
```

*Right click

```
rightClick
```

*Hover

```
hover
```

*Drag and Drop

```
dragAndDrop
```

*Perform a Mouse Action

```
mouseAction
```

Working with Alerts and Dialog Boxes

In this chapter, you’ll learn how to work with alerts and dialog boxes.

*Dismiss Alert Boxes

```
alert
accept
dismiss
```

*Answer Prompts

```
prompt
```

*Answer Confirmations

```
confirm
```

*Set Up beforeUnload Events

```
beforeunload
```

Mocking and Emulation

In this chapter, you’ll learn how to mock network calls and emulate devices and networks.

*Intercept Network Calls

```
intercept
clearIntercept
```

*Emulate a Smartphone

```
emulateDevice
```

*Emulate a Slow Network Connection

```
emulateNetwork
```

*Emulate a Timezone

```
emulateTimezone
```

*Emulate a GPS Location

```
setLocation
```

*Configure the Viewport

```
setViewPort
```

*Set Global Configurations

```
setConfig
getConfig
```

*Set Cookies

```
setCookies
getCookies
deleteCookies
```

*Override Permissions

```
overridePermissions
clearPermissionOverrides
```

*Get Chrome Remote Interface (CRI) Client

```
client
```

*Run a Custom Script on a Selected Element

```
evaluate
```

*Wait for an Element or Condition

```
waitFor
```