Understanding the Challenges and Needs of Programmers Writing Web Automation Scripts

Rebecca Krosnick[1], Steve Oney[2,1]
[1]*Computer Science & Engineering*, [2]*School of Information*
University of Michigan | Ann Arbor, MI USA
{rkros, soney}@umich.edu

Abstract—For web scraping and task automation purposes, programmers write scripts to interact with websites. This is similar to writing end-to-end user interface (UI) test automation suites for software, but on third-party websites that the programmer does not own, introducing new challenges. A programmer might know what semantic operations they want their script to perform, but translating this to code can be difficult. The programmer must investigate the website's internal structure, content, and how UI elements behave, and then write code to click, type, and otherwise interact with UI elements. Many tools and frameworks for creating web automation scripts exist but the challenges programmers face in using them remains understudied. We conducted two studies to study how programmers write web automation scripts. The first study focuses on understanding general challenges. The second focuses on the ways website UI context and script feedback can be helpful. We also provide a set of design findings that detail the kinds of context and feedback developers need while writing web automation scripts.

Index Terms—web automation, automation, macros

I. INTRODUCTION

The Web is a rich source of information and services. The vast majority of web content was designed to be accessed by people through web browsers. However, there is tremendous value in providing services that are also computer-accessible. *Web automation macros*—programs that mimic human input to interact with web pages—can help users perform repetitive tasks, test software applications at scale, help users overcome web accessibility issues, and more. Decades of research into web automation has explored how to allow computers to extract information [1] and perform actions [2]–[4] on web content. Although many tools for web automation have been proposed, the fundamental challenges of writing web automation code remains understudied.

The particular challenges and needs of web automation tools are important to understand for several reasons. First, web automation (and related techniques like Robotic Process Automation) are increasingly common as more information and services continue to be digitized. Second, an evidence-backed description of the challenges of web automation can help provide valuable design guidelines to a large and growing body of work into web automation tools. Finally, several aspects of writing web automation code make it meaningfully different from other kinds of programming. Writing web

automation code requires referencing an external data source (a web page) that was designed to be consumed by humans, rather than code. Aspects of interacting with a web page that are second-nature to people—referencing a particular button, handling unexpected content, and dealing with sequentiality and timing—can be challenging to deal with in code. Further, web pages change over time (e.g., through redesigns or internal refactoring) and change with context (e.g., with A/B testing).

This paper contributes an evidence-backed description of the challenges of writing web automation macros. We conducted two studies—one focused on the general challenges of writing web macros and another focused specifically on providing feedback and context—to better understand these challenges. Among other things, our findings include that developers need feedback and UI context about the page elements they are selecting and interacting with. This paper contributes:

- A first study, uncovering the general challenges programmers face when writing web automation scripts in a traditional text editor.
- A web automation IDE prototype that presents UI snapshots and feedback on element selection across multiple execution contexts.
- A second study, understanding where UI feedback and context features can help programmers writing web automation scripts, and where support is still lacking.
- Design implications for future web automation tools.

II. BACKGROUND AND RELATED WORK

A. Background on Web Automation

Web automation is useful for saving time and energy on tedious and repetitive computer tasks (e.g., approving employee payroll, scraping data), testing software systems robustly at scale, and automating web tasks on inaccessible websites for blind users [5]–[8]. Web automation tools mimic human interactions on web pages. In most web automation frameworks, programmers write code that simulates interactions such as clicking and typing in a web browser. In order to specify which UI elements to interact with, programmers typically use XPath [9] or CSS selectors [10]. Both XPath and CSS selectors reference the DOM (Document Object Model) [11]— a tree structure that represents page content. The typical setup for writing web automation scripts consists of a code editor (for writing the automation code) and a web browser with developer tools [12], [13] (for referencing the page's DOM).

B. Web Automation Tools

Selenium [14], Puppeteer [15], and Cypress [16] are three widely-used commercial web automation frameworks at the time of writing. All three frameworks work similarly—programmers write code in these frameworks (which provide functions for simulating user input, referencing the page, and more). Selenium and Puppeteer simply show the real-time execution of the script on the website UI. Cypress is a newer framework that additionally allows the programmer to post hoc inspect the page state before or after any script command and see which elements were selected. In a lab study we test Cypress and a prototype we built to learn what kinds of UI context and feedback programmers find useful.

Some tools have explored Programming-by-Demonstration (PbD) approaches for web automation, in order to save developers the effort of writing scripts manually. PbD and direct manipulation interfaces allow users to specify or edit a program's behavior by providing visual examples [17], [18] or directly manipulating the visual output [19]–[21]. Selenium IDE [22], iMacros [23], Cypress Studio [24] and several research systems (e.g., Koala [3], CoScripter [4], Rousillon [25] and Sugilite [2]) have explored PbD approaches for generating scripts, i.e., by having the user demonstrate their actions on the web UI. However, PbD systems are limited so in order to have precise control, programmers often still want to hand-write their web automation scripts.

Other researchers have proposed making UI automation easier by simplifying the language used to write web automation macros. Koala [3] and CoScripter [4] represent scripts in a language that is close to natural language—for example, "Click 'Add to cart'." Similarly, Sikuli [26] allows programmers to specify elements visually (with screenshots) for desktop-based automation. With these tools, developers would not need to reference the page's internal DOM structure. Instead, the interpreter searches the page for an element that fits the high-level description of the target element. However, scripts generated in these systems are often not as expressive (because the language is limited) or robust as scripts that explicitly reference the internal page DOM.

C. UI Context and Feedback

As we describe in the "Design Implications" section, many of the challenges of writing web automation code can be categorized as the need for UI context or live feedback. Although the specific context and feedback needs for web automation developers are unique, prior research has explored mechanisms for integrating context and feedback into development tools.

Some programming systems [27]–[30] generate storyboards [31] to illustrate the sequence of program actions and their resulting user interface states. Cypress and our prototype that we evaluated in Study 2 similarly offer UI snapshots to explain program behavior.

Kubelka et al. [32] studied the kinds of immediate feedback features programmers use in several languages, including JavaScript. They observed how programmers heavily use the DOM inspector and console to get faster feedback. In our work, we similarly observed how programmers heavily use the DOM inspector and console. We also observed challenges programmers face specific to web automation and then evaluated environments that offer continual or live feedback on UI state sequences and UI element selection.

III. Study 1: Traditional Editor Environment

We conducted a user study to learn the strategies programmers use and the challenges they encounter when writing web automation scripts in a traditional text editor environment.

A. Study Design

We recruited 15 participants (3 female, 12 male; 20–40 years old) from our university and social media. All participants reported substantial experience with JavaScript and querying the DOM with CSS selectors. Six had 2–5 years and nine had at least 5 years of general programming experience. Our participants included eight professional developers, one product designer, five graduate students, and one undergraduate student. All but one participant reported at least some prior experience with creating web automation scripts.

Each session lasted 90 minutes and participants were compensated with a $25 USD (or equivalent) Amazon gift card. We asked participants to use Puppeteer [15] to write a web automation script. Only one participant had prior experience using Puppeteer. We first gave participants a 15 minute tutorial to familiarize them with Puppeteer. During the task we gave participants reference material for Puppeteer and CSS, allowed them to search online, answered questions about syntax, and provided hints if they were stuck for awhile. We gave each participant one of three tasks to work on for 45 minutes (each task was assigned to five participants):

- *Airbnb* or *Google Hotels*: Create a script that searches for hotels. Set a location (text field), check-in and check-out dates (calendar widget), and display matching results.
- *Amazon*: Create a script for identifying an item to purchase. Search for an item (text field), indicate it must be available via Prime (checkbox), find the first result with a "Best Seller" label, and print out the name of the item.

We chose these tasks to observe a variety of scripting steps participants would need to take (e.g., advancing a calendar to the desired month on Airbnb and Google Hotels; querying for appropriate ancestor and descendant DOM elements on Amazon), as well as their element selection strategies for a variety of website DOMs. Although the Airbnb and Google Hotels tasks are semantically very similar, we used both because we noticed the Google Hotels DOM is complex and many participants were stuck in the early stages of the task.

We asked participants to generalize their scripts to support variable input values (i.e., locations, dates, item to purchase) and gave them two test cases to ensure their script worked for. We then conducted a brief interview.

B. Findings

1) Selecting UI elements: In order to correctly select a desired UI element, participants had to choose CSS selectors [10]

that **uniquely** identify the desired element and are **robust** to page state changes and varying user input. This involved inspecting the DOM to understand the relationships between nodes and reasoning about selector specificity, either based on intuition or by testing selectors manually. For many element selection subtasks, choosing appropriate CSS selectors took a few minutes and some iteration (e.g., stacking selectors once the participant realized a particular CSS class was not unique enough), but were not overly difficult. Other element selection subtasks were more challenging, as we describe below:

Sometimes a unique identifier is not robust across sessions. Some websites (e.g., the Google Hotels website in our study) randomly generate the letters/numbers in IDs [33], classes [34], or attributes [35] per page load or browser session. However, some participants did not realize this ahead of time and accidentally chose selectors containing randomly generated strings. Two Google Hotels participants did this when trying to select the calendar element, using the dev tool's "Copy selector" feature to get a unique selector for it (e.g., `#ow28 > div:nth-child(1) > div:nth-child(1) > div:nth-child(1) > div:nth-child(2)`), but this included an ID (e.g., `#ow28`) that was randomly generated per page load. These participants were then puzzled when the selector did not match any elements on the next run, and when they tried "Copy selector" again and got a different selector this time (e.g., `#ow24 > div > div > div:nth-child(1)`). C3 spent 20 minutes and C5 ten minutes unsuccessfully investigating why their selectors were not working before the study facilitator explained that the IDs change per page load.

Fig. 1. To query the Amazon DOM for the first "Best Seller" and get the item's name, (1) query for the first "Best Seller" label, (2) query for its ancestor that represents the full item, and then (3) query for the item name.

Multi-part queries through the DOM hierarchy. To select the item name for the first "Best Seller" on an Amazon results page, it was not possible for participants to query for simply a single selector. The "Best Seller" label (Figure 1, box 1) and the item name (Figure 1, box 3) are neither ancestors nor descendants of one another, but rather both descendants of a common DOM node ancestor (Figure 1, box 2) representing

the item as a whole (which contains the "Best Seller" label, the item name, item image, etc). Four participants took the approach of first searching for the first "Best Seller" label on the page, then querying up through the DOM tree for the node representing the item as a whole, then querying down through this node's descendants to find the item name (Figure 1). For example, one participant's query was `$(".a-badge-text:contains(Best Seller)") [0].closest('div[data-component-type= "s-search-result"]').find("h2 span.a-text- normal")`. This was non-trivial, because it required careful search of the DOM to find a common ancestor for the "Best Seller" label and item name nodes. These four participants took between 7.5 and 22 minutes to investigate, identify, and test their full selector query chain, a testament to the challenge. The fifth Amazon participant took a slightly different approach, first selecting all of the item containers on the page, then looping through to find the first one containing the text "Best Seller", and then planned to query down through this node's descendants to find the item name. This participant spent 13 minutes on this, but ran out of time.

2) Keeping track of DOM nodes: Most commercial websites have extensive and complex DOM trees. In order to write selectors that are correct, robust, and unique, programmers need to account for not only the target DOM node but also other elements in the DOM. For example, they might need to find the common ancestor of two elements or compare different elements to see if they have the same class. Although most browser dev tools make it easy to navigate to one particular element, they often do not help developers understand the relationships between different parts of the DOM.

3) Navigation and timing: Some interactions cause the browser to navigate to a different page (i.e., from the Amazon home page to a search results page). Before trying to interact with a UI element on a new page, the script needs to allow the page to finish loading (e.g., via the `waitForNavigation` [36] command). However, some participants forgot to include a "wait" command and as a result their script failed to find target UI elements on the page. It took the programmer some effort to understand why the UI element could not be interacted with, because when they manually inspect the page, they see the UI element is present.

4) Trouble typing into input fields: Three Google Hotels participants (C2, C3, C6) had trouble with what originally appeared to be a simple subtask – typing a location into a search bar. These participants decided to select the search bar by the selector `.whsOnd.zHQkBf` and programmatically type into it, but when they ran the script they did not see this typing behavior occur and were puzzled. One participant instead searched for a different selector to use, while the other two participants spent significant time (C2: 16 min, C6: 5 min) trying to debug, trying different things like setting the `value` attribute of the element, which also did not work as desired. The reason participants could not type into the element is because there are actually multiple `<input>` elements on the page with the same class, the first two of which correspond

to the location search bar. However, it turns out that the first element is disabled, which none of the participants noticed. In order to successfully type into the search bar, participants either had to click into the first element before typing into it to give it focus, or they had to select the second element matching their selector, which turns out to not be disabled. This second element matching the selector (but not the first) has the attribute selector `[aria-label="Enter your destination"]`, which a fourth participant C4 chose on a whim at the beginning of the task and as a result never ran into the challenges the other participants faced. Relatedly, participant D3 working on the Airbnb task tried typing dates into the "Check in" and "Check out" date elements, but these elements cannot be typed into at all—the user or script has to actually click dates on the calendar. For these challenges in trying to type into or set the `value` of UI elements, participants did not receive explicit feedback from the environment that these actions could not be performed.

5) Interacting with calendar widgets: A large part of the Google Hotels and Airbnb tasks involved appropriately interacting with and querying the calendar widget. The calendar widget only shows two months at a time (the current month and the next month), so if trying to book a hotel for several months in the future, the script will need to advance the calendar to the correct month. Participants had to reason about how to identify if their desired month and date were visible in the calendar, which involved understanding what the DOM looks like. For example, the Airbnb calendar widget only shows two months at a time, but the DOM actually contains four months in total at a time (i.e., the prior and next months are in the DOM but visibly hidden), which impacts the logic the user might use to correctly identify the current months. Participants also had to make sure that relevant UI rendering finished before they performed queries (e.g., that the calendar finished rendering before they queried any of its contents), otherwise the desired DOM nodes might not be present.

6) Feedback loop and debugging: Participants used a combination of different approaches to understand the results of their code. Six participants simply ran their full in-progress Puppeteer script each time they wanted to evaluate its behavior. The other nine participants used a combination of running the full script and executing commands in the browser dev console, shortening their feedback loop. The browser dev console gave them immediate feedback on whether their CSS selectors uniquely matched the elements they intended, whether interacting with an element had the desired effect (e.g., whether clicking on a button causes the page to navigate), and whether an element had particular attributes. In fact, one participant (Amazon task) essentially drafted his entire script in the browser dev console before adapting it to the Puppeteer environment, incrementally writing commands, observing whether they worked, and adjusting as necessary. 11 participants also inserted `console.log` print statements into their scripts to check intermediate values. Six participants used the browser debugger in order to step through their code to identify the source of a problem and be able to inspect the

DOM at a particular page state.

Several participants explicitly commented that the feedback loop for evaluating whether their code worked was slow, not receiving feedback on their code until the next time they actually ran the script. For example, Google Hotels participant C4 said *"I've done a fair amount of testing and I work as a front end Dev for my job. So I'm using selectors all day long. You can see clearly how many mistakes I was making and there's nothing, there's no feedback to go 'you're being a bit of an idiot here'. The computer is terrible at that....The tests are reasonably kind of slow to run. So if you get something wrong, you have to go work out kind of what's gone wrong, that's not obvious. And then you kind of go run the tests again. And by the time you've done all that, it's like well two minutes in my life, I'm never getting back".*

7) Future website changes might break scripts: Six participants noted that even if they find CSS selectors that work today, their script could break at any time if the website owner changes the website's content, layout, or DOM implementation – *"I would say probably in all cases, you just can't be sure if it's going to work tomorrow...I don't know that [selecting by text] is necessarily going to be more stable than just a test ID or class name. Because who knows what they will change first"* (D4 – Airbnb).

As a proxy for testing the robustness of participants' CSS selectors and understanding how website DOMs change over time, we searched for participants' selectors in older versions of the task websites (via the Internet Archive WayBack Machine [37]) to see if they existed there. Some selectors work for website versions from the last several years, for example the `#twotabsearchtextbox` selector for the search bar on the Amazon home page works on websites back until July 2010. However, participants' selectors for other elements do not work for earlier website versions within a year of when we ran our study (October 2020). Of the four Amazon participants who finished creating a selector to select the text for the first "Best Seller" item on the page, three participants' selectors do not work on the January 2020 version because they selected by attribute values or class combinations that previously did not exist. Of the three Google Hotels participants who finished creating a selector for clicking to open the calendar widget, two participants' selectors (`.p0RA.ogfYpf.Py5Hke` and `.DpvwYc.of9kZ`) do not work on the October 2019 version, while another seemingly obscure selector (`.eoY5cb.MphfQd.yJ5hSd`) does work. In fact, `.DpvwYc.of9kZ` actually no longer works on the current Google Hotels website as of the submission of this paper (May 5, 2021). This suggests that writing selectors that are robust across page changes is a significant challenge.

IV. Study 2: Environments that Provide UI Context and Live Feedback

We conducted a study to evaluate the benefits and limitations of web automation environments that provide the programmer UI context and feedback. We evaluated a prototype we built (Figures 2 and 3) and Cypress [16], an increasingly

popular test automation framework. First we describe each environment. Then we describe the study design and results.

A. Prototype

We designed and built a prototype IDE (Figures 2 and 3) for programmers writing web automation scripts, inspired in part by Study 1. The prototype provides live feedback on CSS selectors, integrates UI context within the code editor, and helps users understand script results across different user inputs and for different pages. We built this prototype to provoke new ideas about providing UI context and feedback in web automation tools, and see to what degree programmers find them useful. The prototype includes a code editor on the left, main website view in the center, and UI snapshots which pop out from the right. Chromium dev tools are available for the main website and UI snapshots.
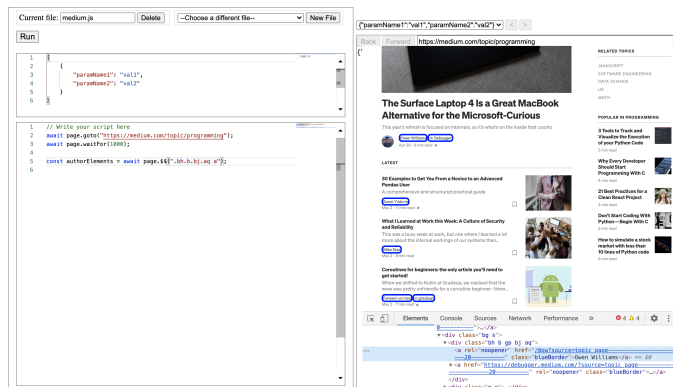
Fig. 2. It is challenging to select an author link on Medium because the <a> element does not have a semantic or specific selector. Instead, the parent <div> has a unique set of classes, so the programmer includes those in the selector – .bh.b.bj.aq a. Our prototype immediately highlights all matching elements on the page with a blue border, and lets the programmer see that they are mistakenly selecting not only author links but also publications.

1) Dynamic element highlighting: When the programmer writes a CSS selector, matching UI elements in the current website view are highlighted with a blue border, as Figure 2 shows. Each time the user edits their code or moves their cursor to a different line, the highlights update to show the matching elements. This gives developers immediate feedback on which elements they are selecting and can help them identify mistakes.

2) UI snapshots: At runtime the tool captures UI "before" and "after" snapshots for each line of code, which the programmer can review to understand the effect of a given line. If the line has a CSS selector, the matching UI elements are highlighted in the snapshots (Figure 3, green borders for elements matching line 14 selector dd.txt).

3) CSS selector validity feedback: An error message is provided and squiggle shown beneath each CSS selector string in the editor to indicate its validity in the context of the runtime page state: a yellow squiggle if the selector is found but not unique (Figure 3, lines 6 and 14) and a red squiggle if the selector is not found. Squiggles are updated live when the user edits a selector, with the selector checked against the UI

"before" snapshot for that line. If snapshots are stale (i.e., earlier parts of the script have been edited since the last run), validity feedback is not shown for CSS selectors on that line.
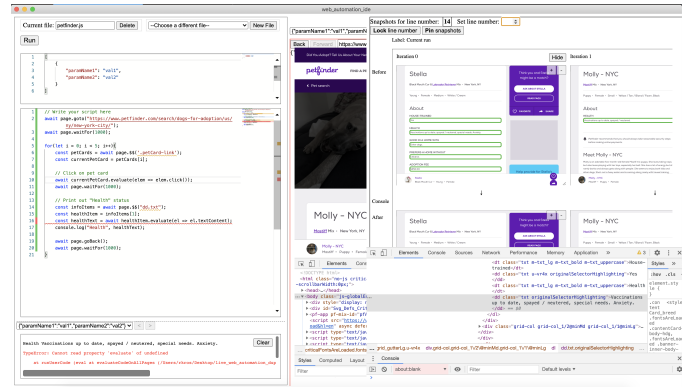
Fig. 3. Our prototype lets users inspect UI snapshots per line of code, across execution contexts. Here, the script has failed in the i=1 iteration of the loop, and the snapshots illustrate why. The UI snapshots for line 14 indicate that Stella (i=0) has five info elements (highlighted with a green border) matching selector dd.txt whereas Molly (i=1) only has one, which explains why the infoItems[1] indexing on line 15 failed for Molly's page.

4) Context and feedback across different runs: The prototype allows programmers to write scripts that contain loops and to run their script simultaneously across different sets of user inputs. This lets programmers test their code to make sure it works across scenarios or pages, and see corresponding UI snapshots and holistic CSS selector feedback for a given line of code in one place (Figure 3). This might help programmers discover that they have written a CSS selector or other logic that works in some scenarios or pages but not all.

5) Implementation: The prototype is implemented as an Electron [38] app, using Monaco editor [39] and Puppeteer [15] as the automation scripting library. It uses rrweb-snapshot [40] to capture and render UI snapshots of the DOM.
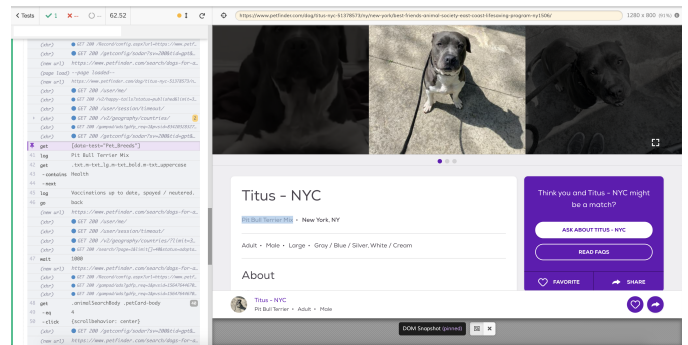
B. Cypress

Fig. 4. Cypress running a script that scrapes data from the Petfinder website. The user can hover or click on a particular command to see the UI state at that point in the execution, here item 40 where [data-test="Pet_Breeds"] is selected. The matched element ("Pit Bull Terrier Mix") is highlighted in the website view on the right.

Cypress [16] is an increasingly popular test automation environment that offers visual context and feedback about

scripts at runtime. With Cypress, programmers write their code in a text editor and when they save their file, the results of their script are automatically updated in a web browser augmented with Cypress UI panes (Figure 4). On the left, the Cypress command log presents the sequence of element selection and interaction commands the script executed. The programmer can hover or click on a given command (e.g., item 40 in Figure 4) to see the website's UI state at that point in the execution in the main browser viewport. For selection commands, the selected element(s) will also be highlighted in the website UI and the number of matched elements indicated on the command log item. Programmers can also use their browser's built-in developer tools as they normally would.

C. Study design

We recruited ten participants (eight male, two female; age 21–56, median age 29) from our university department, social media, and the Future of Coding community to participate in a 90 minute user study. We compensated participants with a $25 USD (or equivalent) Amazon gift card. Participants were all experienced programmers (eight with at least 5 years, two with 2–5 years experience) and all reported being comfortable working with CSS selectors and JavaScript methods for querying the DOM. Four participants reported some but not extensive experience with Cypress. Participants came from a variety of occupations (five professional developers, three PhD students, one undergraduate student, one CTO) and have varying experience with UI automation, ranging from none to more than five years. Each participant completed a web scraping task on each of the two conditions, our prototype and Cypress. The two web scraping tasks were:

1) *Medium*: Create a script that navigates to a Medium topic page[1] and for the first five articles, navigates to the article author's page, prints out the number of followers they have, and then navigates to their "About" page.

2) *Petfinder*: Create a script that navigates to a Petfinder search results page[2] and for the first five dogs, navigates to the dog's page, prints out the dog's breed, and prints out information about the dog's health.

We chose these two websites because they are non-trivial to write scripts for: many elements do not have IDs, classes, or attributes that are semantically meaningful to select by; and there are differences across pages on a given site, either in the content shown or DOM implementation.

We counterbalanced task order and website/tool pairings. Participants were given 25 to 30 minutes per task, with the exception of P1 who was only given 22 minutes for the Cypress task. Before each task, participants watched an eight minute tutorial video about the tool that illustrated how it works and its different UI context and feedback features. We gave them a reference sheet and allowed them to search the web for resources during the task. Due to short task time and to help fill knowledge gaps, during the task we answered

[1]https://medium.com/topic/programming
[2]https://www.petfinder.com/search/dogs-for-adoption/us/ny/new-york-city/

questions they had about Cypress, Puppeteer, and CSS syntax and provided hints if the participant was stuck for awhile. After completing both tasks, we conducted a brief interview.

D. Results

Participants found aspects of both tools useful, in particular the feedback on which UI elements are being selected. We first give an overview of the main challenges participants encountered in writing generalized scripts. We then discuss the kinds of context and feedback participants needed, in what ways the tools provided them, and participants' opinions on specific UI context and feedback features.

1) Challenges: A primary challenge of the tasks was identifying selector logic that generalizes appropriately. Specifically, some of the common challenges were:

Selecting content correctly across pages when it has no semantically meaningful class names and content order varies. Petfinder dog profile pages (e.g., as seen in the snapshots in Figure 3) include information about the dog's health, friendliness, adoption fee, and more, but the exact categories and number of categories shown per dog varies. This information is presented in DOM elements that have no semantically meaningful class or attribute names, making them more challenging to extract. When we asked participants to scrape dogs' health information across pages, six participants tried selecting by a general selector like `dl dd.txt`, which selects text from all information categories, and then indexing into the results list to choose the second item (which, on the first dog's page, corresponds to the health information). A couple participants noted that this might not work, but tried it anyway. Once they ran their script, they got an error and saw that the second dog only has "health" information and no other categories, so their indexing approach is not robust (Figure 3). Three other participants up front chose to select by text value, which was a successful approach – they first selected the element containing the text "Health", then chose its next sibling to retrieve the information itself.

Selecting an element correctly when it has different CSS class names across different pages. The Medium website uses obscure CSS class names that vary across pages. All authors have a "number of followers" element in their page header but the CSS classes are different per author. Many participants constructed their selector for the "number of followers" element using the CSS classes listed on the first author's page (e.g., `.cd.gg.t a`), which then did not work on other author pages. Four participants encountered this problem only after they ran their script and saw a "no elements matched" error. However, two other participants avoided this problem by instead using the more robust and semantic selector `[href$="/followers"]`, selecting elements whose `href` attribute ends in `/followers`. One participant made this choice based on intuition, and the other first chose to review multiple author pages' to compare their DOM trees and check if the class-based selector they chose would generalize, and they saw it did not.

Identifying elements that are clickable. For subtasks that involved clicking, participants were easily able to identify the correct visual element on the page to click on. However, when constructing a CSS selector, some participants selected an ancestor element of the clickable `<a>` element (e.g., because the ancestor has more specific classes or attributes), but the ancestor in some cases actually does not respond to click events. For example, when selecting an author link from the Medium starting page, participants discovered that author links do not have semantically meaningful or specific classes or attributes (Figure 2). The best option is to use the author link's parent's class names (e.g., `.bh.b.bj.aq`) as part of the selector. Five participants used a selector like `.bh.b.bj.aq` but forgot to further query to select the actual `<a>` element, and were therefore confused when clicking `.bh.b.bj.aq` did not navigate to the author's page. Two participants experienced the same problem on Petfinder.

Choosing a selector that is specific enough to select certain elements but not others. As mentioned above, participants used selectors like `.bh.b.bj.aq` and then further queried by `a` to select author links from the Medium website. Many participants therefore simply chose `.bh.b.bj.aq a`, which selects all `<a>` with an ancestor that has classes `.bh.b.bj.aq`. This selects the author link (e.g., text "Owen Williams" in Figure 2), but also incorrectly selects publication links (e.g., text "in Debugger"). Three participants used this selector and only discovered it was too general once they ran their script and saw it navigating not only to author pages but also publication pages. On the other hand, a different participant (P2) realized his selector was too general before he even ran his script, by taking advantage of our prototype's dynamic element highlighting feature. For his first selector attempt `.bh.b.bj.aq a`, author and publication links were highlighted (Figure 2), which is not what he wanted. He then adjusted his selector to be `.bh.b.bj.aq a:first-child` and saw the blue highlighting update to highlight only the author links as he wanted.

2) Element selection context and feedback: Participants appreciated receiving feedback and UI context for the elements their script selected. Seven of ten participants verbally expressed that they found the CSS selector inline feedback squiggles in our prototype useful. Participant P6 said *"I found that really useful, the inline contextual help on that, because that helped me like immediately identify, 'OK, it was running this line, it couldn't find this thing' "*, referring to a CSS selector he wrote that had a typo. A couple participants also noticed that the selector feedback squiggles update when they edit a selector string – *"I'm seeing that when it doesn't match anything, that turns red. If I had known it existed at the beginning I would've used that instead of fiddling around in the console. That feedback is really nice"* (P7). Participants similarly appreciated Cypress's runtime element selection feedback.

Participants also appreciated selected elements being highlighted in the website view and UI snapshots, and used this to identify if they selected the desired elements and make corrections accordingly. For example, with Cypress, participant P2 realized that simply selecting by the text "Health" on the Petfinder website was not specific enough to query the "Health" category, because he saw another instance of "Health" on the page was getting selected instead. With Cypress, P9 realized that her selector for selecting author links on Medium was actually incorrectly selecting publication links some of the time. Seven participants said the dynamic element highlighting our prototype offers in the main website view is useful, commenting on how the dynamic highlighting shortens the feedback loop and provides an easy way for checking if their selector is selecting the right elements. P2 in particular used the dynamic highlighting feature heavily, iteratively writing and adjusting his selector based on the highlighting feedback, and for example realizing he was incorrectly selecting publication links as discussed above in the "Challenges" section and Figure 2.

Participants mentioned additional kinds of live feedback they would like to see. P1 and P6 want to see live UI snapshots that update immediately each time the user edits their code. P2 also wants to see variable and element attribute values evaluated live – *"I would probably like to see what the [hrefs] capture, because I usually spend a lot of time debugging...like what would be evaluated...a bit of like a REPL experience like in dev tools or console"*.

3) Understanding page states: In creating their scripts, participants needed to understand the pages with which their script was interacting. When participants needed to confirm that their script commands worked as intended and navigated to the correct sequence of pages, most participants simply watched their script run in the main website view. One participant (P4) used Cypress's snapshots heavily for understanding unexpected page navigation. She was confused why a certain author was visited twice and used Cypress's snapshots to discover that the order of authors listed on Medium had changed during the course of her script's execution, which was using the live website content. To write generalized element selection logic, participants needed to understand the similarities and differences between different author pages on Medium and different dog pages on Petfinder, and to do this they manually navigated in the main website viewport.

Although participants did not heavily use UI snapshots, several participants commented on how they could be useful. P8 commented on how being able to compare different pages is important – *"I realized that each page might be different, I wondered if that selector from the last page is going to be generalizable...I wonder if there's like a better way than me just manually clicking through [the pages], I was imagining if there's a visual comparison, where I got to select multiple sites at once..."*. We suspect UI snapshots were underutilized by participants because 1) the short task time was not enough to become fully familiar with UI snapshots and internalize where they would be useful and 2) UI snapshots might be a tool that is appropriate for less frequent situations.

4) Traditional debugging approaches: Even with the various UI context and feedback features available, most par-

ticipants still leveraged traditional web UI development and debugging techniques. Seven participants executed selector query commands in the browser dev console to experiment with candidate selectors, check which element(s) they match, and further inspect these elements.

V. DESIGN IMPLICATIONS

As our study results show, writing web automation code presents a unique set of challenges and information needs. We can divide the information needs of web automation developers into roughly two categories: *context* and *feedback*. In this section, we describe our recommendations for the kinds of contextual information and feedback that future web automation tools should provide.

A. Contextual Information

Web automation code references the internal structure of the web page on which it runs. Providing the right context about the target web page can make it easier for developers to write Web automation macros by bridging the "gulf of execution" [41].

- *DOM nodes and values*: The code editor should provide inline access to the values of variables, selected elements, and their attributes, perhaps available on hover. Inline access is important for helping developers early on understand the elements they are selecting and the values their script is producing.
- *UI snapshots*: UI snapshots of each step of the execution should be provided to help programmers understand whether they are selecting the correct elements, whether the expected behavior occurred, and what the page state is after a given command. Many participants in our second study found this helpful.

B. Effective Feedback

Immediate feedback can help developers discover problems in their code early by bridging the "gulf of evaluation" [41]. It may be technically challenging to provide live feedback, as web automation scripts do not run immediately but rather only as quickly as web pages navigate and render. For efficiency, a live feedback tool might keep a copy of the page state per line of code, so that whatever line of code the programmer edits next, the script can be run starting from that particular line.

- *Feedback on selectors*: The code editor should provide inline feedback per element selection command, indicating clearly the number of matching elements and whether any elements are hidden. The exact elements that are selected should be highlighted in a UI snapshot of the corresponding page state. Many participants in our second study found this feedback helpful. UI snapshots should be shown in the periphery of the editor so the programmer can validate their element selection logic as they write it.
- *Feedback on interactivity of elements*: The code editor should give feedback on whether the selected element can be interacted with as the developer intends. For example, if the programmer tries typing into an element

that cannot be typed into, clicking on an element that cannot be clicked, or setting the `value` for an element that has no `value` attribute, the editor should show an error rather than letting the command silently fail. This feedback is important because information about element event handlers is not always clearly visible in browser dev tools. None of the environments we evaluated provide feedback on whether elements can be interacted with as intended, and as a result a few of our participants were puzzled that their interaction commands did nothing.
- *Feedback across pages*: Many participants in our studies wrote element selection logic that worked for one website page but that they later realized did not work for others. Perhaps web automation tools should proactively suggest or prompt users to identify multiple pages that the script should run correctly on. This could help programmers earlier on understand differences across pages and write code that appropriately generalizes.
- *Longitudinal feedback*: Developers cannot anticipate and have no control over how and in what ways third-party websites will change over time. We saw that the DOM for websites from our first study changed within the course of a year, and some participant-chosen selectors would not have worked on those other website versions. It would be valuable for web automation tools to help developers identify when a website has changed in ways that will break their script or cause its behavior to change, and to help developers repair their script accordingly.

VI. DISCUSSION AND LIMITATIONS

Our study design allows us to present a qualitative description of the challenges and needs of programmers writing web automation scripts. However, due to its small participant size and exploratory nature it is lacking quantitative measures of how long certain task types take and whether UI context and feedback features offer speedup. Additional studies with more participants, a broader set of tasks, and longer study sessions would be informative.

UI context and feedback features will *inform* programmers as they develop and debug, but will not actually *write* the code for them. Recent innovations in programming-by-demonstration [2], [25] could help generate automation scripts, but for full control, programmers will still need to reason about and choose navigation logic and CSS selectors themselves.

VII. CONCLUSION

Programmers writing web automation scripts have specialized needs, as they need to interpret third-party websites and programmatically mimic user interactions. Through two user studies, we found that these programmers need contextual information about the UIs they are interacting with and feedback on their element selection and interaction code. We hope our research can help guide the design of future web automation tools. We also believe many of our design implications may be relevant for UI test automation and UI programming.

REFERENCES

[1] M. Craven, A. McCallum, D. PiPasquo, T. Mitchell, and D. Freitag, "Learning to extract symbolic knowledge from the world wide web," Carnegie Mellon University, Tech. Rep., 1998.

[2] T. J.-J. Li, A. Azaria, and B. A. Myers, "Sugilite: creating multimodal smartphone automation by demonstration," in *Proceedings of the 2017 CHI conference on human factors in computing systems*, 2017, pp. 6038–6049.

[3] G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan, "Koala: capture, share, automate, personalize business processes on the web," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007, pp. 943–946.

[4] G. Leshed, E. M. Haber, T. Matthews, and T. Lau, "Coscripter: automating & sharing how-to knowledge in the enterprise," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2008, pp. 1719–1728.

[5] S. Oney, A. Lundgard, R. Krosnick, M. Nebeling, and W. S. Lasecki, "Arboretum and arbility: Improving web accessibility through a shared browsing architecture," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 937–949.

[6] J. P. Bigham, T. Lau, and J. Nichols, "Trailblazer: enabling blind users to blaze trails through the web," in *Proceedings of the 14th international conference on Intelligent user interfaces*, 2009, pp. 177–186.

[7] J. P. Bigham, I. Lin, and S. Savage, "The effects of" not knowing what you don't know" on web accessibility for blind web users," in *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*, 2017, pp. 101–109.

[8] J. P. Bigham and R. E. Ladner, "Accessmonkey: a collaborative scripting framework for web users and developers," in *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, 2007, pp. 25–34.

[9] "Xpath," https://developer.mozilla.org/en-US/docs/Web/XPath/, accessed: 2021-03-20.

[10] "Css selectors," https://developer.mozilla.org/en-US/docs/Learn/CSS/Building_blocks/Selectors, accessed: 2021-03-19.

[11] "Document object model (dom)," https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/, accessed: 2021-06-29.

[12] "Get started with viewing and changing the dom," https://developer.chrome.com/docs/devtools/dom/, accessed: 2021-05-02.

[13] "Dom property viewer," https://developer.mozilla.org/en-US/docs/Tools/DOM_Property_Viewer, accessed: 2021-05-02.

[14] "Selenium," https://www.selenium.dev/, accessed: 2020-09-11.

[15] "Puppeteer," https://pptr.dev/, accessed: 2020-09-18.

[16] "Cypress," https://www.cypress.io/, accessed: 2021-03-19.

[17] R. G. McDaniel and B. A. Myers, "Getting more out of programming-by-demonstration," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 1999, pp. 442–449.

[18] M. R. Frank, P. N. Sukaviriya, and J. D. Foley, "Inference bear: designing interactive interfaces through before and after snapshots," in *Proceedings of the 1st conference on Designing interactive systems: processes, practices, methods, & techniques*, 1995, pp. 167–175.

[19] B. Hempel and R. Chugh, "Semi-automated svg programming via direct manipulation," in *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, 2016, pp. 379–390.

[20] B. Hempel, J. Lubin, and R. Chugh, "Sketch-n-sketch: Output-directed programming for svg," in *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, 2019, pp. 281–292.

[21] B. Victor, "Inventing on principle," https://vimeo.com/36579366/, accessed: 2021-03-20.

[22] "Selenium ide," https://www.selenium.dev/selenium-ide/, accessed: 2020-06-08.

[23] "imacros," https://imacros.net/, accessed: 2020-06-08.

[24] "Cypress studio," https://docs.cypress.io/guides/core-concepts/cypress-studio, accessed: 2021-05-05.

[25] S. E. Chasins, M. Mueller, and R. Bodik, "Rousillon: Scraping distributed hierarchical web data," in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 963–975.

[26] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using gui screenshots for search and automation," in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, 2009, pp. 183–192.

[27] F. Modugno and B. A. Myers, "A state-based visual language for a demonstrational visual shell," in *Proceedings of 1994 IEEE Symposium on Visual Languages*. IEEE, 1994, pp. 304–311.

[28] J. Meskens, K. Luyten, and K. Coninx, "D-macs: building multi-device user interfaces by demonstrating, sharing and replaying design actions," in *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, 2010, pp. 129–138.

[29] P.-Y. Chi, S.-P. Hu, and Y. Li, "Doppio: Tracking ui flows and code changes for app development," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–13.

[30] P.-Y. Chi, Y. Li, and B. Hartmann, "Enhancing cross-device interaction scripting with interactive illustrations," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 2016, pp. 5482–5493.

[31] K. N. Truong, G. R. Hayes, and G. D. Abowd, "Storyboarding: an empirical determination of best practices and effective guidelines," in *Proceedings of the 6th conference on Designing Interactive systems*, 2006, pp. 12–21.

[32] J. Kubelka, R. Robbes, and A. Bergel, "The road to live programming: insights from the practice," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1090–1101.

[33] "Id selectors," https://developer.mozilla.org/en-US/docs/Web/CSS/ID_selectors, accessed: 2021-05-05.

[34] "Class selectors," https://developer.mozilla.org/en-US/docs/Web/CSS/Class_selectors, accessed: 2021-05-05.

[35] "Attribute selectors," https://developer.mozilla.org/en-US/docs/Web/CSS/Attribute_selectors, accessed: 2021-05-05.

[36] "Puppeteer waitfornavigation," https://pptr.dev/#?product=Puppeteer&version=v8.0.0&show=api-pagewaitfornavigationoptions/, accessed: 2021-05-02.

[37] "Internet archive wayback machine," https://archive.org/web/, accessed: 2021-05-05.

[38] "Electron," https://www.electronjs.org/, accessed: 2020-09-18.

[39] "Monacoeditor," https://microsoft.github.io/monaco-editor/, accessed: 2021-05-02.

[40] "rrweb-snapshot," https://github.com/rrweb-io/rrweb-snapshot, accessed: 2020-09-18.

[41] D. A. Norman, *The psychology of everyday things*. Basic books, 1988.

[Larger copy of **Figure 1**] To query the Amazon DOM for the first "Best Seller" and get the item's name, (1) query for the first "Best Seller" label, (2) query for its ancestor that represents the full item, and then (3) query for the item name.

[Larger copy of **Figure 2**] It is challenging to select an author link on Medium because the `<a>` element does not have a semantic or specific selector. Instead, the parent `<div>` has a unique set of classes, so the programmer includes those in the selector – `.bh.b.bj.aq a`. Our prototype immediately highlights all matching elements on the page with a blue border, and lets the programmer see that they are mistakenly selecting not only author links but also publications.
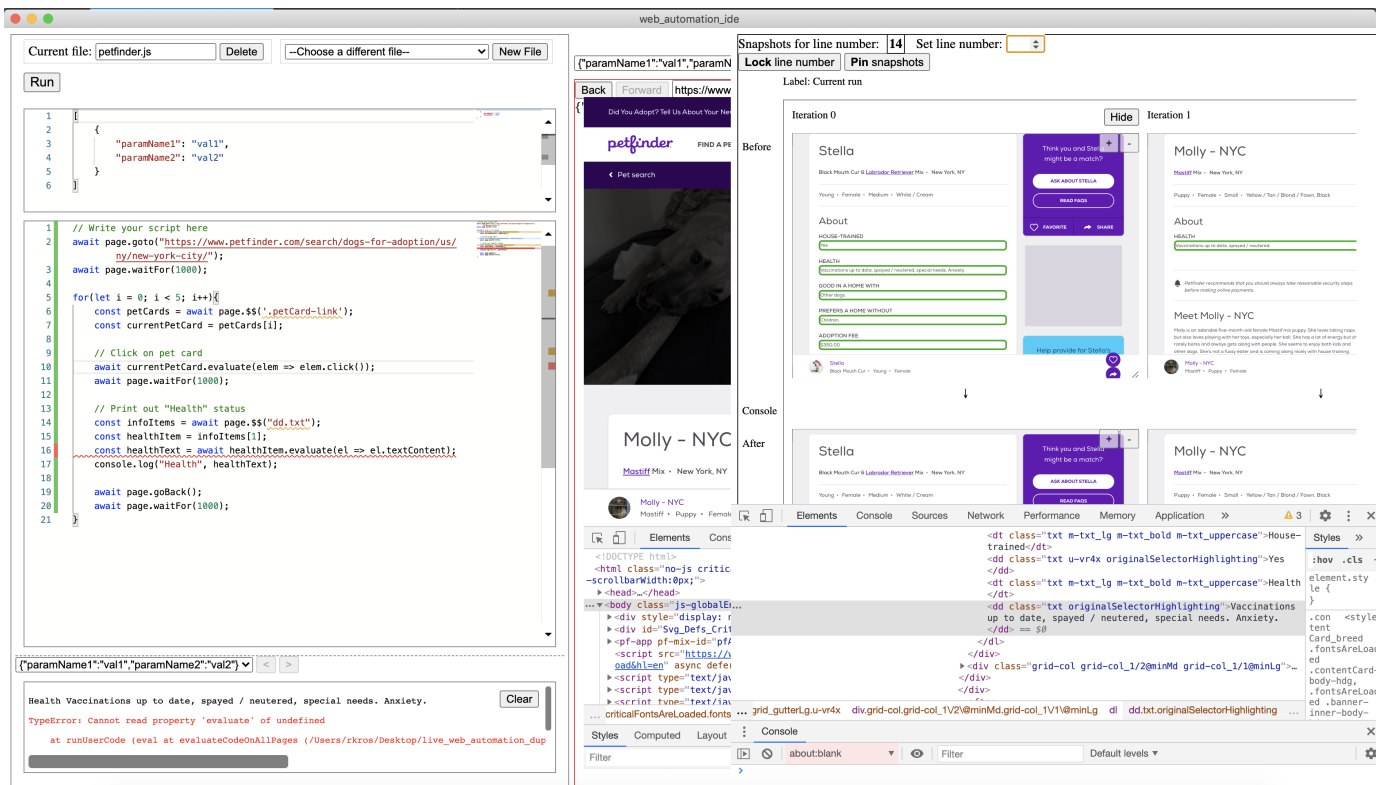
[Larger copy of **Figure 3**] Our prototype lets users inspect UI snapshots per line of code, across execution contexts. Here, the script has failed in the `i=1` iteration of the loop, and the snapshots illustrate why. The UI snapshots for line 14 indicate that Stella (`i=0`) has five info elements (highlighted with a green border) matching selector `dd.txt` whereas Molly (`i=1`) only has one, which explains why the `infoItems[1]` indexing on line 15 failed for Molly's page.
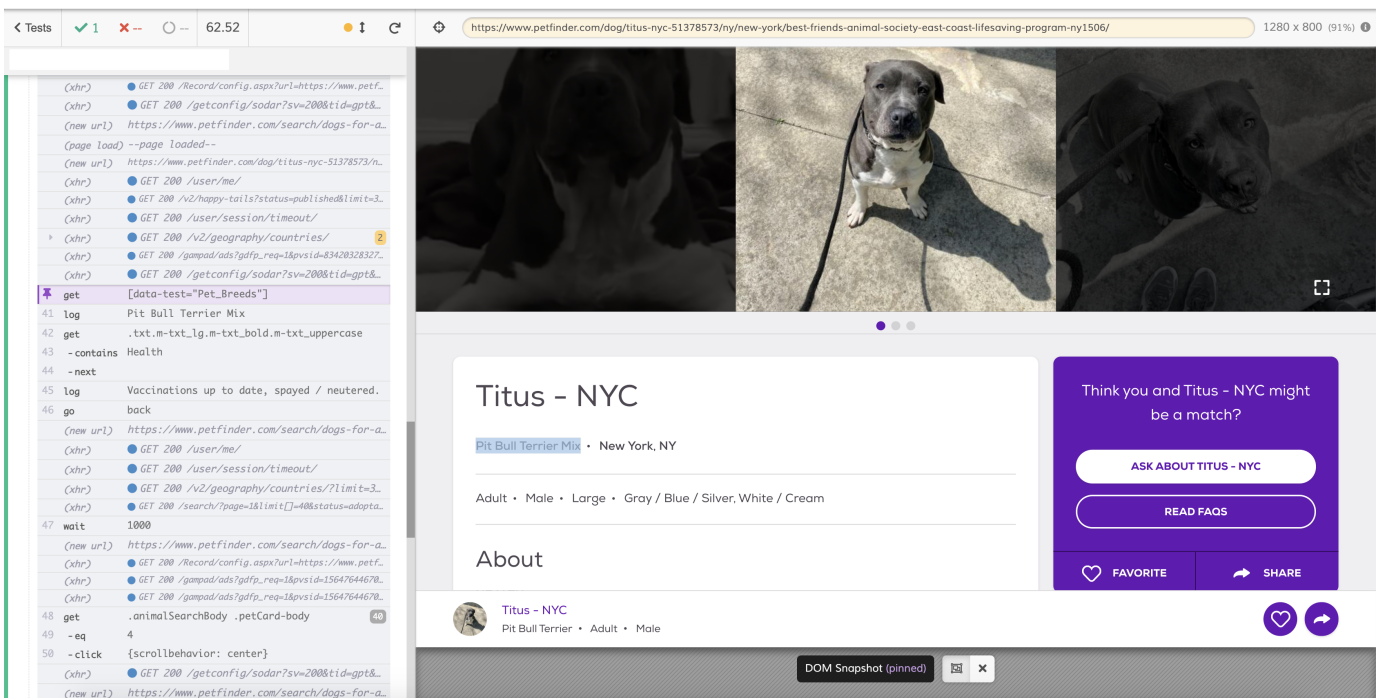
[Larger copy of **Figure 4**] Cypress running a script that scrapes data from the Petfinder website. The user can hover or click on a particular command to see the UI state at that point in the execution, here item 40 where `[data-test="Pet_Breeds"]` is selected. The matched element ("Pit Bull Terrier Mix") is highlighted in the website view on the right.