

Translation of user interaction into automated UI tests

João Lopes¹ and Joss Santos²

¹ Jscrambler, S.A, Porto, Portugal

² Jscrambler, S.A, Porto, Portugal

Abstract. Jscrambler’s Quality Assurance (QA) team has been developing a custom, in-house testing engine, which features an elegant UI automated testing solution. However, writing UI end-to-end tests for this engine remains a cumbersome task as the XPath’s of several HTML elements needed for testing must be manually obtained, the number of which can be increasingly high. For this reason, the development and QA teams require a tool that can record user interaction in any website and generate data that allows the re-execution of said interaction in the testing engine. This paper documents the analysis and part of the implementation phase of the development of a browser extension that solves this problem. The developed solution solves the main goal of the project by providing Jscrambler’s development and QA teams a ready to use Chromium extension capable of recording steps in a webpage and generating the data necessary to re-execute these steps in the company’s internal testing engine.

Keywords: Quality Assurance, Test recorder, Automated Tests, Browser extension, Cucumber.

1 Introduction

Jscrambler is a technology company specializing in client-side security for web and mobile applications [1]. All Jscrambler’s services must be fully tested in order to provide their clients with high quality software. To achieve this, an internal testing framework has been developed by the QA team, which enables performance, vulnerability, and integrity testing of any service, including the user interfaces of Jscrambler’s website and product’s dashboards.

Despite being very powerful, writing UI tests for this framework requires a lot of repetitive work. Elements in a page are located through their XPath’s and thus, when writing tests, the developers must manually inspect all the HTML elements being tested in a page, save any data or credentials being submitted, and ensure all navigation is registered. Because the test data is reused, as the number of tests increases and webpages get more complex, keeping track of all this data becomes increasingly difficult and time-consuming.

The Quality Assurance market offers several solutions to this problem, but all of them have panned out to have some pitfalls for Jscrambler’s particular use cases. These solutions can be divided into two general categories: automated testing platforms and browser extension test recorders.

The first category comprises services such as Smartbears' TestComplete, which offers a large set of features such as test recording and playback, integration with continuous delivery (CD) pipelines and auditing tools [2]. However, these platforms are meant to be used individually and do not allow for much customization or integration with other systems.

The second category are browser extensions that allow the user to record his interaction in a webpage and then export the recorded test in the form of a script with hard-coded test data, which will run on Selenium Webdriver [3]. These tools do offer the flexibility that Jscrambler's QA team desires, but by only exporting tests in a one-layer format, maintainability is compromised greatly because all the test logic, data and execution are hardcoded in the same file. Because both solutions had compromises deemed unacceptable, a new tool that could record user interaction and generate tests in a format usable by the existing testing engine was required.

2 Analysis

For UI testing, Jscrambler's automated testing framework defines four entities: Tests, Test Sets, Test Plans and Test Executions. Each Test is a part of a Test Set, which allows the aggregation of tests by feature or functional groups. If a feature or group is deprecated, then the Test Set associated with it is also discarded. Test Sets are then grouped in Test Plans, where each Test Plan represents a version of a given system under test (SUT). Finally, Test Executions are linked to and will execute every Test of every Test Set in a Test Plan. Using these abstractions, test data is completely separated from its execution, allowing its usage in various Tests [4].

In concrete terms, a Test, which can also be referred to as a scenario, is a single script detailing UI interaction or assertions. A Test Set is then a collection of Tests in a Cucumber feature file [5]. These files contain tags that represent different versions or groups of tests – Test Plans – which means that the same Test Set can belong to multiple Test Plans. Finally, a Test Execution is a JavaScript file defining steps and their runtime behavior, utilizing Selenium Webdriver to simulate user actions in the browser. The file structure for this implementation can be seen in Fig. 2.

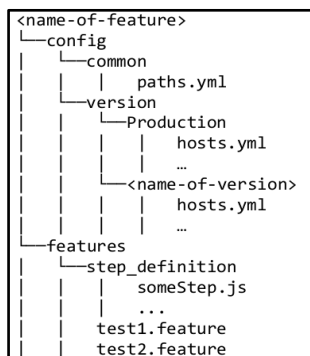


Fig. 2. Jscrambler's QA framework file structure

Files inside the *common* folder are shared between runtime environment versions, whereas files inside the *version* folder are exclusive to a specific runtime environment. All feature files and step definitions are global to the framework. A Cucumber *feature* file represents a Test Set and can be associated with any runtime environment version or Test Plan with the usage of tags. An example can be seen in Fig. 3.

```

1. @SomeFeature @SomeOtherFeature @SomeVersion
2. Feature: Testing
3.   Background:
4.     Given I use User "1"
5.
6. @SomeTestPlan @SomeOtherTestPlan
7. Scenario: Testing youtube opens
8.   Given I create Application "youtube"
9.   Then I am greeting the world

```

Fig. 3. Feature file example

Feature file's scenarios are then mapped to step definitions, which execute the former's logic using Selenium Webdriver. An example of this mapping can be seen in Fig. 4.

```

1. const { Given, Then } = require("cucumber");
2.
3. Given("I use User {string}", async function (userId) {
4.   // create or login User
5. });

```

Fig. 4. Step definition example

A step definition can be anything that Selenium Webdriver and the browser's API permits. To promote code reuse, a shared project was created containing the step definitions most used across testing projects. To be viable, the test recorder being developed had to support these common types: *action steps*, *navigation steps*, *visualization steps*, and *other steps*.

An action step involves clicking on an HTML element or submitting a form; A navigation step handles switching to a new URL or *iFrame*; Visualization steps handle ensuring that a specified HTML element is visible in a page and contains some data; Other steps support saving elements in a variable within the testing context.

To offer some support for context-dependent steps, a fifth step type was also created, named *custom step*, where the user can manually write a step in Gherkin syntax. In this case, the user must also ensure that a matching step definition is present in the destination testing project.

3 Implementation

Browser extensions consist of three components that share data with each other: *the background script*, *content scripts*, and the *popup*. This relationship can be seen in Fig. 5.

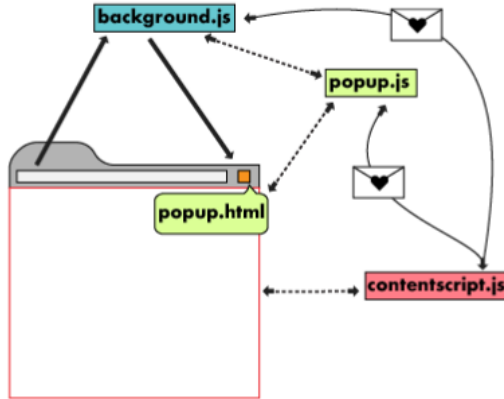


Fig. 5: Browser extension architecture [6]

In this project, the *popup* will be presenting the GUI through which the user can control the recording of tests, view the recorded data and export it. The *background script* will handle creating a browser context menu to make creating a step more friendly and, finally, the *content scripts* will poison functions and inject event listeners in the web pages being tested in order to detect the user's interaction with them. Traditionally, and as recommended by Google in their browser extensions guidelines, extensions should serve as simple enhancements to websites, featuring minimalist UIs that serve a single purpose [7].

The project described in this paper, however, warrants a much more complex GUI. To achieve this, the extension opens in a large pop-up window, and its UI functions as a single page application powered by ReactJS. The user then interacts with it as it would with a normal website.

Usually, a web page's only concern is to render information and then, via some communication protocol such as HTTP, pass user inputs to where domain logic is handled: back-end servers. In this project, however, the chrome extension is standalone in the sense that all the domain logic also lives within the client, thus behaving as an offline rich-client application. This can be achieved by taking advantage of React Hooks [8]. By combining the *useContext* and the *useReducer* hooks, it is possible to have a global, immutable state of the domain that can only be modified via a dispatcher function. This state can then be serialized and saved in the browser's *localStorage*, in order to have some form of persistence. The UML component diagram in Fig. 6 shows the relationship between the presentation logic and the domain logic inside the browser extension.

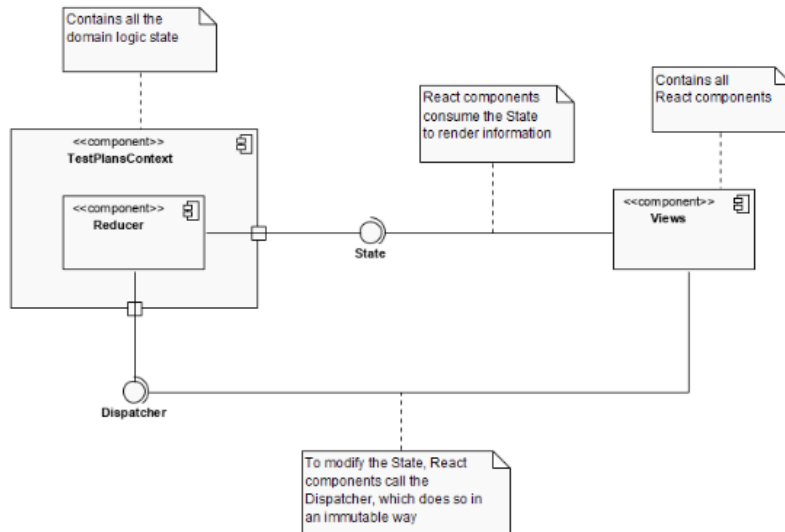


Fig. 6. Domain and presentation logic relationship

As seen in the above diagram, the domain logic is treated as immutable and separate from the presentation logic. With this low coupling level, presentation logic can be safely maintained without touching the domain logic. The latter can also be altered without touching the former, assuming the *Dispatcher* interface does not suffer changes.

3.1. Test view

The main goal for this view was to present all the information of a test in an easy to read and easy to use manner. This can be seen in Fig. 7.

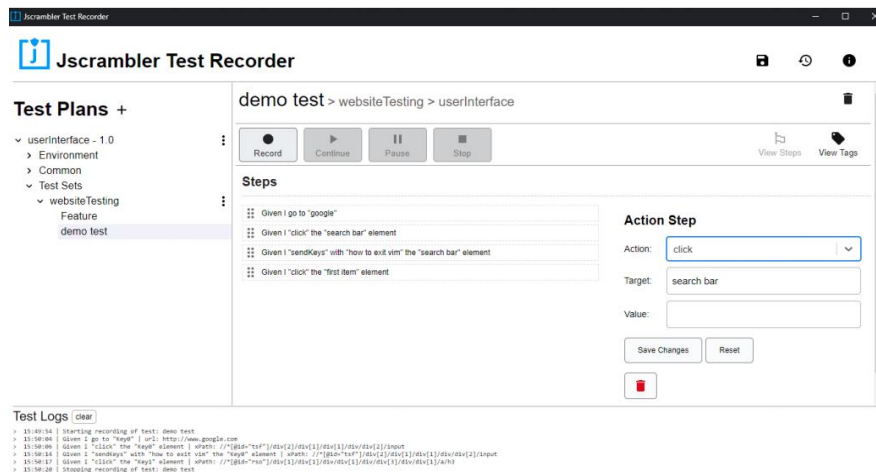
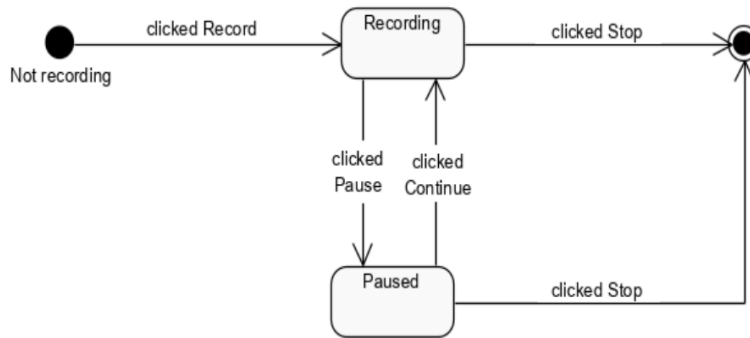


Fig. 7. Edit step view

The scenario above contains four steps, which are sorted by their order of execution. A step can be dragged and dropped to a different order, and by clicking on one a view for editing it appears. New steps can be added to the bottom of the list by clicking the “Record” button and interacting with web pages. The State diagram for a recording session can be seen in Fig. 8.

**Fig. 8.** State diagram for a recording session

All objects mentioned above can be hard deleted. However, to prevent accidental clicks that would require re-recording steps, all delete actions require a confirmation from the user, which is presented via a modal. A session can also be saved to the browser’s local storage with a click, in order to be restored later. Taking advantage of the technologies and methods described above, a cross-platform and easy to distribute test recorder that currently supports all Chromium-based browsers was achieved.

4 Evaluating the solution

The main goal of this project is to make the tester’s workflow easier when creating UI tests. To ensure this goal was met, once a minimum viable product (MVP) was achieved, a usability analysis session was conducted. Usability can be understood as a quality attribute that evaluates how intuitive and easy to use a user interface is. This attribute, according to [9] and [10], is usually associated with the following metrics: Learnability, efficiency, memorability, errors, and satisfaction.

However, different standards of human-computer interaction can attribute different values to these metrics. For this project, the ISO 9241 standard was targeted, which values effectiveness, efficiency, and satisfaction [11]. To determine these metrics for the system being analyzed, the most popular and effective methodology is to run evaluation sessions, where users interact with the system and posteriorly give their opinions and feedback through a questionnaire [12]. The papers [13] and [14] offer comparisons between multiple standard usability analysis questionnaires and, based on their findings, the Usability Metric for User Experience (UMUX) questionnaire was chosen for

this project. As for the number of participants in the analysis, according to [15], in a usability analysis session, 80% of the usability problems in a system are detected with four to five subjects, as additional participants are less likely to reveal new information. Based on this, a total of five participants were asked to participate in the usability analysis session, all of which are developers at Jscrambler. The questionnaire's items can be seen in Table 1.

Table 1. Questionnaire items

Item 1	The system's capabilities meet my demands.
Item 2	Using this system is a frustrating experience.
Item 3	This system is easy to use.
Item 4	I have to spend too much time correcting things with this system.

The results for the questionnaire can be seen in the stacked bar chart in Fig. 9.

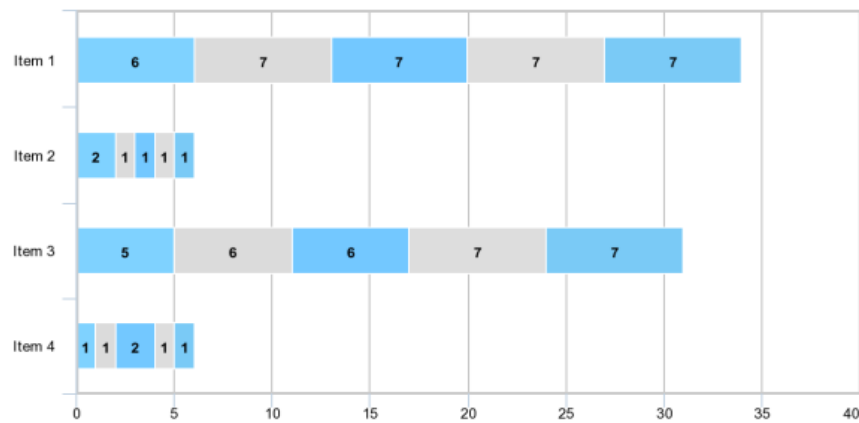


Fig. 9. Questionnaire results

By following the formula in [16] to calculate the UMUX score for the system, an average score of 95 for a maximum score of 100 was achieved. With this result, it can be concluded that the MVP for the test recorded browser extension achieved the goal of helping Jscrambler's QA team create UI tests faster and more reliably.

5 Conclusion

The major goal of creating a tool that allows the development and QA teams to easily create UI tests by recording their actions in websites has been met. The MVP created in this project allows a tester to interact with any website while recording his actions, download the resulting test files, copy-paste them into the Jscrambler's testing engine

and run them. This workflow alone already saves the tester immense time; however, more work is currently under development to skip the “copy-paste” part, namely by creating a cross-platform desktop application that handles all the logic of merging the new recorded data with an existing testing project.

References

- [1] ‘JavaScript Protection and Webpage Monitoring’, *Jscrambler*. <https://jscrambler.com/> (accessed Nov. 13, 2020).
- [2] ‘TestComplete Automated UI Testing Tool | SmartBear’. <https://smartbear.com/product/testcomplete/overview/> (accessed Nov. 13, 2020).
- [3] ‘WebDriver :: Documentation for Selenium’. <https://www.selenium.dev/documentation/en/webdriver/> (accessed Nov. 13, 2020).
- [4] J. Santos and V. Mangano, ‘QA Automation Guide’. May 2019.
- [5] ‘BDD Testing & Collaboration Tools for Teams | Cucumber’. <https://cucumber.io/> (accessed Nov. 13, 2020).
- [6] ‘What are extensions? - Google Chrome’. <https://developer.chrome.com/extensions> (accessed Nov. 13, 2020).
- [7] ‘Extensions Quality Guidelines FAQ - Google Chrome’. https://developer.chrome.com/extensions/single_purpose (accessed Nov. 13, 2020).
- [8] ‘Introducing Hooks – React’. <https://reactjs.org/docs/hooks-intro.html> (accessed Nov. 13, 2020).
- [9] J. Nielsen, *Usability engineering*, Nachdr. Amsterdam: Kaufmann, 2010.
- [10] W. L. in R.-B. U. Experience, ‘Usability 101: Introduction to Usability’, *Nielsen Norman Group*. <https://www.nngroup.com/articles/usability-101-introduction-to-usability/> (accessed Nov. 13, 2020).
- [11] ‘ISO 9141-2:1994(en), Road vehicles — Diagnostic systems — Part 2: CARB requirements for interchange of digital information’. <https://www.iso.org/obp/ui/#iso:std:iso:9141:-2:ed-1:v1:en> (accessed Nov. 13, 2020).
- [12] S. Federici and S. Borsci, ‘Usability evaluation: models, methods, and applications’, 2010, pp. 1–17.
- [13] T. S. Tullis and J. N. Stetson, *UPA 2004 Presentation—Page 1 A Comparison of Questionnaires for Assessing Website Usability*. .
- [14] Ahlem Assila, Houcine Ezzedine, and Káthia Marçal de Oliveira¹, ‘Standardized Usability Questionnaires: Features and Quality Focus | electronic Journal of Computer Science and Information Technology’, Accessed: Nov. 13, 2020. [Online]. Available: <http://ejcsit.uniten.edu.my/index.php/ejcsit/article/view/96>.
- [15] R. A. Virzi, ‘Refining the Test Phase of Usability Evaluation: How Many Subjects Is Enough?’, *Hum. Factors J. Hum. Factors Ergon. Soc.*, vol. 34, no. 4, pp. 457–468, Aug. 1992, doi: 10.1177/001872089203400407.
- [16] Anastacia Valdespino, ‘UMUX (Usability Metric for User Experience)’, *Qualaroo Help & Support Center*. <https://help.qualaroo.com/hc/en-us/articles/360039072752-UMUX-Usability-Metric-for-User-Experience-> (accessed Nov. 13, 2020).