WebPuppet – A Tiny Automated Web UI Testing Tool

Ricardo Queirós
CRACS – INESC-Porto LA & uniMAD, ESMAD/P. Porto, Portugal

Abstract

One of the most important phases in the Web development cycle is testing. There are several types of tests, different approaches to their use and a wide range of tools. However, most of them are not open source, require coding and do not have a pedagogical nature. This article introduces WebPuppet as an automated Web UI testing tool. The tool is distributed as a small Node package and can be easily integrated into any learning environment in the web development domain. In addition, it does not require coding in any language, just use a very simple domain-specific language that will generate a test script to run in client applications. In order to exemplify its use, a simple test scenario based on a login page is presented.

2012 ACM Subject Classification Applied computing → Computer-managed instruction; Applied computing → Interactive learning environments; Applied computing → E-learning

Keywords and phrases User Interface Testing, Web development, DOM

Digital Object Identifier 10.4230/OASIcs.ICPEC.2022.10

1 Introduction

Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do. There are several types of tests that can be performed in a Web product ranging from unit, functional to end-to-end and integration. Despite the usefulness of all these types, User interface (UI) testing is one of the most important in the Web development cycle. In order to validate whether applications have the desired aesthetics and functionalities, Quality Assurance (QA) professionals should test all interface components. This action will not only improve the software quality but also ensures a rich experience for end users while using the Web application.

UI testing plays a significant role before an application is released to production. UI testing is centered around two main things. First, checking how the application handles user actions carried out using the keyboard, mouse, and other input devices. Second, checking whether visual elements (e.g. text boxes, checkboxes, radio buttons, menus, toolbars, colors, fonts, and others) are displayed and working correctly.

The test can be performed manually or with an automated testing tool. There are several tools that support UI testing (e.g. Parasoft Selenic[1], Katalon[2], Selenium IDE[3], Mabl[4], Perfecto[5]). Most of these tools have great features such as recording abilities, locators recommendations, BDD support, CI/CD integration, self-healing capabilities and many supported languages. Although these features, some of them are proprietary, too complex and not pedagogical-driven. Regardless of the method and tool used, the goal is to ensure all UI elements meet the requested specifications.

[1] `https://www.parasoft.com/products/parasoft-selenic`
[2] `https://katalon.com/`
[3] `https://www.leapwork.com/discover/selenium-automation`
[4] `https://www.mabl.com`
[5] `https://www.perfecto.io/`

This paper presents WebPuppet as an automated Web UI Testing tool. The real motivation for its creation was to test the UI of small web applications created by students in web development courses. In this way, the teacher does not need to manually create tests that are always time consuming and error-prone and can thus concentrate on making more differentiating and impactful exercises. In its genesis, the tool receives as input an instance with all test cases based on a domain-specific language that will generate a test script to be executed later in all student applications. As an output, a report is generated for the student with the identification of all errors and possible solutions.

The remainder is organized as follows. Section 2 provides a background on UI testing approaches. Section 3 presents the WebPuppet tool and its domain-specific language. Section 4 presents a simple test scenario where the tool can be used. Finally, Section 5 summarizes the contributions of this work and points to future directions.

2 UI testing approaches

A test is a code which can run automatically in a client application to validate a user interface in order to meet design and logic requirements. Often we group a set of related tests in a test case and organize a set of test cases with different priorities and dependencies in a test scenario. A test case can have tests of two types:

- **Simulation tests** – those responsible for simulating the user's behavior;
- **Validation tests** – those responsible for checking, after we simulated some user behavior, if the system worked properly.

In the literature several solutions can be found to perform tests on graphical interfaces which resort on computer vision [1], web interface matching algorithms [2], structural comparison (comparing the trees resulting from the two HTML documents) [4], or with very different goals such as detecting phishing sites [3]. The goal of this section is not to compare the different solutions found, but rather compare the different approaches that can be used to automate Web UI testing. There are three main UI testing approaches, namely: manual testing, record/playback and keyword/data-driven scripting. The following subsections detail each one.

2.1 Manual testing

A common way of testing the UI of a Web application is to directly write code in a programming language like JavaScript, Java, PHP or C++. Often this will be the same programming language used to write the application that is being tested. Using this approach, a human tester (or a team) performs a set of operations to check whether the application is functioning correctly and that the graphical elements are conformed to the documented requirements.

Several frameworks allow the creation of tests for various platforms. The most notable examples are Selenium (for Web applications), Appium (for mobile), and Microsoft Coded UI (for Windows applications).

Despite being an flexible approach easy to implement, manual-based testing has some drawbacks such as it can be time-consuming, and the test coverage is extremely low. Additionally, the quality of testing in this approach depends excessively on the knowledge and capabilities of the human tester or testing team.

2.2 Record-and-Playback Testing

This approach resorts to automation tools which records all tasks, actions, and interactions with the application. The recorded steps are then reproduced, executed, and compared with the expected behavior. For further testing, the replay phase can be repeated with various data sets.

Using record-and-playback testing the simulation part of the script is relatively easy to capture just performing the relevant user actions and the system creates a script. However, the validation part is more difficult to achieve. After simulating the user actions, for each element to check on the screen, it is necessary to explicitly add steps to the script in order to identify these elements in the interface, and compare their values to expected values. As obvious, it is impossible to record these validations because they are not user actions. Thus, it is mandatory to define them one by one using the automation system's GUI.

This approach typically generate test scripts behind the scenes in simple scripting languages like VBScript. Often, advanced users manipulate the code directly to make small adjustments.

2.3 Keyword/Data-Driven Testing

Other test frameworks support the definition of a set of "keywords" which specify user actions. This approach calls **keyword testing**. A human tester can specify these keywords and a script will be generated that performs the desired actions on the system under test. Listing 1 shows a small example:

Listing 1 Keyword Scripting Test example.

```
Open Browser To Login Page
Input Username david
Input password 16485
Submit Credentials
```

In this case, the first line tell that the script should navigate to the login screen and enter certain user credentials. A script will be generated that knows how to navigate to the login screen, type in the data provided and submit the form.

Using this approach, the simulation part of the script will be handled by one keyword that defines the user action (e.g. "login page" in the example above). The validation part of your script will require multiple keywords and multiple values of expected data for each part of the UI to validate. From the tester's perspective it is just using the keywords, and don't need to deal with the code. However, this means that typically few validation options are available, and adding more will require help from the responsible for the generation script.

A variant of this approach, called **data-driven testing** (Figure 1), is when the same test needs to be repeated several times with different data values or different user operations.

Data-driven testing (DDT) is data that is external to your functional tests, and is loaded and used to extend the automated test cases. The same test case cab be taken and run it with as many different inputs, thus getting better coverage from a single test.

The advantage of this testing approach is that the code that simulates user operations is not repeated in every test script, rather it is defined in one place, and a tester can use it as a building block in multiple test cases. This reduces the code writing and also the maintenance required as the application under test changes.

One of the most famous testing frameworks which uses this approach is the open source Robot Framework. There are others test automation frameworks which provides a mix between keyword-driven or data-driven testing functionality.

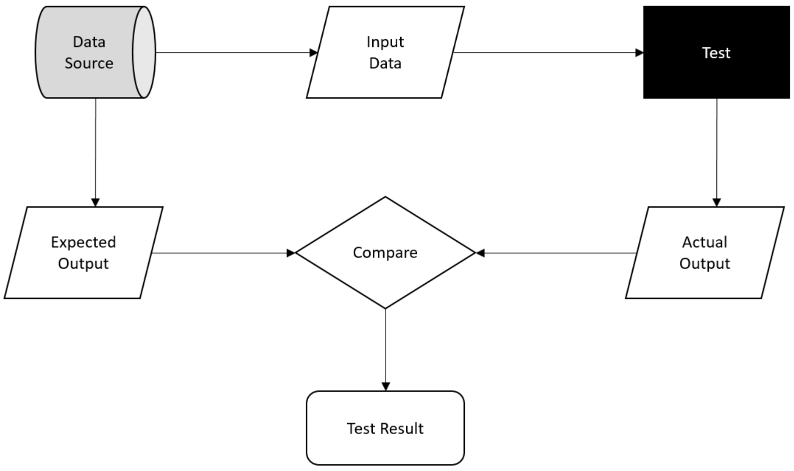

Figure 1 Data-driven Testing.

3 WebPuppet

WebPuppet is a small tool for automating web application UI tests. Its creation arises from the need to support the evaluation of students' performance in the creation of Web applications in learning environments. The architecture of WebPuppet is straightforward and is composed of the following components:

- The editor – Web-based component with a GUI for the definition of test scenarios.
- The engine – client component responsible for the generation of the test script that will run on the client application.

3.1 The Editor

The editor is a Web-based component for the creation of a test scenario. A test scenario is a document that explains how the application under test will be used in real life. A simple test scenario could be: "users will successfully sign in with a valid username and password". In this scenario, we can have tests for multiple GUI events (e.g. provide a valid username and password combination, enter an invalid username, hit the login button). These tests are grouped in test cases for logic organization and dependency.

A test scenario is defined as a domain-specific language (DSL) formalized with a JSON Schema. As said before, a test scenario is composed by one or more test cases. Listing 2 presents how a test case is described.

Listing 2 Test case schema.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "description": "A schema to formalize a Test Case",
 "type": "object",
 "properties": {
   "id": {"type": "integer"},
   "description": {"type": "string"},
   "depends": {"type": "array", "items": {"type": "integer"}},
   "tests": {"type":"array", "items": {"$ref": "#/definitions/Test"}}
 },
 "required": ["id", "description", "depends", "tests"]
}
```

The `id` property is the test case identifier. The `description` property describes in natural language the test case. The `depends` property refers to the test case(s) that this test case depends on. Finally, the `tests` property is an array with all the tests that compose a test case.

A test (Listing 3) is the smallest unit in the DSL and will actually contain the test to be performed in the client code.

Listing 3 Test schema.

```
{
 "$schema": "http://json-schema.org/draft-04/schema#",
 "description": "A schema to formalize a single test",
 "type": "object",
 "properties": {
   "id": { "type": "integer" },
   "description": { "type": "string" },
   "selector": { "type": "string" },
   "action": { "enum": ["VALIDATE", "GET", "FILL", "CLICK"] },
   "operator": { "enum": ["=", "!=", ">", "<","..."] },
   "value": { "type":"string" },
   "error": {"type": "string" }
 },
 "required": ["id", "selector", "action"]
}
```

The `id` property is the test identifier. The `description` property describes in natural language the test. The `selector` property is a CSS selector responsible to find the element to test in the DOM tree. The `action` property is a enumeration of all the actions that can be made in the element selected:

- `VALIDATE` – verify if certain DOM element or attribute exist;
- `GET` – get a value from a DOM element or attribute and compare it with a specific value. In this case two more actions should be defined:
  - `OPERATOR` – the operator to use in the comparison;
  - `VALUE` – the value to compare.
- `FILL` – inject text in a selected text box or select an item in a selected combo box;
- `CLICK` – click in a selected button, radio button or checkbox.

The `error` property is a string with a feedback to be delivered to the student when the test is not passed.

3.2 The Engine

The engine is a JavaScript file that will receive as input an instance of a test scenario formalized in the previous DSL and will generate a test script to be executed in the client application code created by the student. All tests will run according to predefined dependencies and the generated feedback will be presented to the student.

A test script is code that can be run automatically to perform a test on a user interface. The code will typically do the following one or more times: (1) Identify input elements in the UI, (2) Simulate user input, (3) Identify output elements, (4) Assert that output value is equal to expected value, and (5) Write the result of the test to a log.

The engine uses Puppeteer to simulate user actions. Puppeteer is a Node library that provides a high-level API to control headless Chrome or Chromium browsers over the DevTools Protocol. It can also be configured to use full (non-headless) Chrome or Chromium.

In Table 1, we present some of the functions of the WebPuppet engine.

Table 1 WebPuppet engine functions.

Function	Description
TestScenario loadTestScenario(JSON scenario)	Creates a new test scenario based on a WebPuppet DSL scenario
TestCase TestScenario.getTestCase(int id)	Get a test cased based on a given id
List<Test> TestCase.getTests()	Obtains the tests of a specific test case
Object TestScenario.run()	Execute all test cases automatically and returns a JSON object with the feedback

In this moment, only the first and the last functions are implemented. This means that currently the test scenario must be created manually.

4 UI testing scenario

This section presents a typical test scenario of the login function on a website. In this scenario, we can do the following:

1. Load the website homepage
2. Locate the “username” and “password” text-boxes and the “submit” button in the home screen;
3. Type the username “ricardo” and password “12345”
4. identify the “submit” button and click it
5. Wait and locate the title of the Welcome screen that appears after login
6. Read the title of the Welcome screen.
7. Assert that the title text is “Welcome ricardo”.
8. If title text is as expected, record that the test passed. Otherwise, record that the test failed.

It is important to distinguish two important parts of the test scenario:

- The **simulation** part of the script (steps 1, 3 and 4) are responsible for simulating the user’s behavior;
- The **validation** part of the script (steps 2, 5–8) are responsible for checking, after we simulated some user behavior, if the system worked properly.

This scenario should be defined through the DSL mentioned in the previous section. Listing 4 presents a snippet of a test instance, more precisely, the fulfilment of the login form with specific data and the verification, after submission, of the authenticated user name. If the verification is not successful, the student receives automatic feedback based on the value of the `error` property.

Listing 4 Test instance.

```
{
 "id": "3",
 "description": "Fill the login form elements
    and inspect the name of the logged user",
 "depends": "1",
 "tests": [{
   "id": "1",
   "selector": "#username",
   "action": "FILL",
   "value": "ricardo"
  }, {
   "id": "2",
   "selector": "#password",
   "action": "FILL",
   "value": "12345"
  }, {
   "id": "3",
   "selector": "#loginbtn",
   "action": "CLICK",
   "value": "new"
   }, {
    "id": "4",
    "selector": ".usertext",
    "action": "GET",
    "operator": "=",
    "value": "Ricardo Queiros",
    "error": "authentication failed"
    }
  ]
}
```

Using these basic elements of GUI automation, simulation and validation, allows testing even in very complex multi-step operations. In complex test scripts these elements will repeat themselves several times, each time for a different part of the user's workflow.

5 Conclusion

This article introduces an automated UI testing tool called WebPuppet. The purpose of the tool is not to compete with existing ones, but to simplify the testing process as much as possible and orient it towards a more pedagogical nature by defining a simple feedback system for each of the tests.

The tool can be easily integrated in learning environments where students are challenged to create Web applications and used to test their solutions. This way, the tool can relief the burden of manual evaluation of each Web application bu the teacher.

Right now the tool is in beta stage, but it can already be installed and used as an npm package (the package manager for the Node JavaScript platform). As future work it is intended:

- to allow the automatic creation of the test scenario through an API;
- to support the task recording feature in the browser.

References

1 Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. Gui testing using computer vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1535–1544, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1753326.1753555.

2 Marco Primo and José Paulo Leal. Matching User Interfaces to Assess Simple Web Applications. In Pedro Rangel Henriques, Filipe Portela, Ricardo Queirós, and Alberto Simões, editors, *Second International Computer Programming Education Conference (ICPEC 2021)*, volume 91 of *Open Access Series in Informatics (OASIcs)*, pages 7:1–7:6, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/OASIcs.ICPEC.2021.7.

3 Gaurav Varshney, Manoj Misra, and Pradeep K. Atrey. A survey and classification of web phishing detection schemes. *Security and Communication Networks*, 9(18):6266–6284, 2016. doi:10.1002/sec.1674.

4 Jiří Štěpánek and Monika Šimková. Comparing web pages in terms of inner structure. *Procedia - Social and Behavioral Sciences*, 83:458–462, 2013. 2nd World Conference on Educational Technology Research. doi:10.1016/j.sbspro.2013.06.090.