Software Unit Testing

1 Introduction

Businesses normally need to make a profit in order to survive. Peter Drucker (1909 - 2005) introduced the idea of a "profit center" in 1945. [1] A profit center within a company increases the company profits. If we write and sell software, we would want that activity to be a profit center.

If the users encounter errors or problems using the software, they will want the company to correct those deficiencies. To deal with these issues, companies maintain a group or department that provides customer service or technical support. Customer service is NOT normally considered to be a profit center. It is considered to be a cost center. [1] The costs of providing customer service reduce the company profits. If the customer service people are constantly working with customers to resolve problems, this can result in elevated costs to the company which will offset any profits that the company might make. Some companies attempt to limit the costs by providing a limited amount of free support. Extended support would require that the customer subscribe to a plan of paid support.

If we offer software to the public as a product, we are almost certainly competing with other groups or companies that might offer a similar product. How do we convince people to purchase our software and not a competing product? If the user's experience is not positive due to an excessive number of software errors, they might be motivated to seek out a competing product with less errors. So, we should try to eliminate software errors in the product before it goes out to the public. Eliminating software errors is not just about reducing cost in order to maximize profit. Some types of software are subject to regulation. For example, the software found in an aircraft needs to be safe, so the procedures and requirements for developing and testing this type of software have been have been written in standards such as DO-178B and DO-178C, Software Considerations in Airborne Systems and Equipment Certification. [5] [6]

In order to make sure that the software we write is working correctly, we need to test it. In the early days of software development, this meant using the software the same way that a customer would use it and watching for any errors before the product was shipped. The errors would be documented and corrected, then verified that things were now working correctly. If the product was large or complicated, the company might have an entire department just for testing. As time passed, people began to write software tools that would perform automated testing. Automated testing tools reduced the number of testers needed, and some companies were able to have the developers write automated tests at the same time that they were writing new software features. [8] Although automated testing is very helpful, there are still situations when a person must perform manual testing. This usually occurs when there are situations that the test tools cannot adequately assess.

2 Unit Testing versus End-to-End Testing

If we wish to build a brick house, we need to purchase a sufficient quantity of bricks to do so. We would want to purchase high quality bricks that don't start to degrade after just a few years outside exposed to the weather. Modern software is modular, so software modules would be similar to the bricks in a brick house. Just like bricks, we would want our software modules to be high quality. That means that they should perform as expected, without errors. We determine the quality of the software modules by testing them.

We could put all the modules together to form a complete product and test that. This type of testing is called End-to-End testing. But what happens if we discover an error and it's not clear which module caused the problem? Maybe the problem was introduced because of an interaction between one or more modules, not because one of the modules had an error. We could test each of the software modules separately. This is called Unit testing. Unit testing reduces the complexity of the situation. While Unit testing does not guarantee that we won't encounter a problem due to an interaction between modules in the complete product, it's very helpful because we know that any problem that might be encountered has to be within that module. Unit Testing can help us discover problems with the software in the early stages of development. Tests can be written as soon as we complete a software module. If problems are corrected early in the development process, the later stages of development will go more smoothly.

3 Test Coverage

How do we know if we have tested the software completely? Automated test tools monitor the execution of the software and keep a record of which lines of software code were executed and which lines were missed. The test tool can generate a report showing the percentage of the total number of lines of code that were executed. Although some software projects might consider a coverage of 70 to 80% to be acceptable [2], many software projects require that there be a higher level of coverage. For example, Avionics software written in compliance with DO-178B must show that all software requirements (100 %) were covered by testing. [5] A module contains a certain amount of lines of code. Complex modules contain multiple paths through the code. In order to verify that we have checked every path of execution and executed every line of code, we would need to write multiple test cases. Each test case would configure the inputs to the module as required to execute the path and code statements that we wish to verify. After calling the module, the results would be compared with the values that we expected to see. If the actual values that were returned match the expected values, we know that the code is working correctly. The test tool would monitor the execution of the software and keep a record of which lines of software code were executed and which lines were missed. [2] [3] [4] [7]

4 Modern Software Testing Tools

As time passed, there were various attempts to build software tools that could perform software testing. These tools have evolved over several generations. One of the newer testing tools that has been built as an improvement of the previous work is called Cypress. Cypress, which first became available in 2014, is available under the MIT license. [56] The product has already had a decade of development.

Cypress runs in all major browsers. Scripts are written using JavaScript. You can use Cypress to do Behavior-driven development (BDD) and Test-driven development (TDD). [52] [55] Cypress can do End-to-End tests, Integration tests and Unit tests. [54] [59] At this time, Cypress is strictly for testing web applications. It cannot currently test mobile or desktop applications. [53] Cypress runs inside the browser without needing a browser driver. It doesn't have an added layer of complexity due to the driver, so it is faster than Selenium. [52]

Dependencies	
Cypress	42
Cucumber	27
Karma	24
Mocha	20
Jest	4
Jasmine	2

Figure 1: Test tool functionality based on dependencies. [56] [57].

Cypress appears to include a greater number of functionalities than competing testing tools, based on the number of dependencies it requires. Mocha is included within Cypress. So if you know how to write tests using Mocha, you can easily write tests using Cypress. [52] Tests are written in JavaScript, so if you already know JavaScript, you can easily write Cypress tests. [52] [55] You can use the same reporting tools that you find in Mocha. [54] Cypress consists of two parts: you install the free Cypress application on your local machine under Node.js®, and Cypress Cloud records your test runs. [13] Node.js is a free, open-source, cross-platform JavaScript runtime environment that lets you run JavaScript on the server. [49] [50] It is used by millions of developers. [51] Cypress has plugins for integration with GitHub, CircleCI, Jenkins, Slack, and Visual Studio Code. [58]

Another popular product for automated testing is called Selenium. Selenium is supported on Linux, macOS and Windows. [66] Selenium WebDriver is a W3C Recommendation. [62] Selenium requires that you install the binding libraries for the language you are using. The supported languages are C#, Java, JavaScript, Kotlin [64], Pearl, PHP, Python and Ruby. [67] In addition, you must install the browser and the browser driver. [61] [63] The supported browsers are Chrome, Edge, Firefox, Internet Explorer (version 11 only, requires additional configuration) [66], Safari and Opera. [65] [66] Mobile browser drivers are also available for Android and iOS. [68] The browser driver adds an extra layer of complexity, and results in slower test execution. [52] Selenium does not directly allow you to test mobile apps. It is possible to do so, however, through the use of frameworks such as Appium and Selendroid. [69]

Appium provides UI automation for browsers, desktop, mobile and TV. [70] [71] [74] Appium supports multiple programming languages (Java, Python, Ruby, C#). [74] Appium is fairly complex: in order to use Appium you need to install Appium, install a platform driver, and install the client library for the programming language you are using. You may also need to install some plugins. [71]

Selendroid is for testing Android applications. Selendroid tests are written using the Selenium Webdriver client API. [73] Selendroid mostly uses Java. [74] It is reported that tests execute slowly, and that the product is not usable if the computer has less than 4GB of RAM. [73] Selendroid is no longer actively maintained. [74]

5 Test Scenario

For our testing example let's use the children's game "FizzBuzz", which is sometimes used as a coding test. FizzBuzz works like this:

Output the numbers from 1 to 100.

1. If the number is not divisible by either 3 or 5, just display the number.
2. If the number is divisible by 3, display the word "FIZZ" instead of the number.
3. If the number is divisible by 5, display the word "BUZZ" instead of the number.
4. If the number is divisible by both 3 and 5, display the word "FIZZBUZZ" instead of the number.

So, our code must meet the four requirements listed above. In order to do so, the code must make several decisions based on the input value. So we would need a minimum of 4 test cases to do testing based on the requirements. This function has the following code paths:

path 1 - the input IS NOT divisible by EITHER 3 or 5, return the input value
path 2 - if the input is divisible by 3, set the variable divisibleBy3 to true
path 3 - if the input is divisible by 5, set the variable divisibleBy5 to true
path 4 - the input IS divisible by BOTH 3 and 5, return "FIZZBUZZ"
path 5 - the input IS divisible by 3 but IS NOT divisible by 5, return "FIZZ"
path 6 - the input IS NOT divisible by 3 but IS divisible by 5, return "BUZZ"

If we test to meet the requirements, we would expect to see 100% path coverage with the four test cases needed to meet the requirements.

6 Unit Testing - Desktop or Mobile Application

Let's write a procedure that we want to use in a desktop or mobile application. We want to test this procedure to make sure it works correctly. Our example could be written in any one of a number of languages, but we'll write this one in PASCAL. PASCAL is a fairly old language, but many modern languages are descended from it, and share similarities. Our procedure would look like this:

```
procedure fizzBuzz (   counter : integer;
                        VAR resultStr : string   );
begin

  resultStr := inttostr( counter ); (* default - just return the counter *)

  if(counter mod 3 = 0) then (* divisible by 3 *)
  begin
    resultStr := 'FIZZ';
  end;

  if(counter mod 5 = 0) then (* divisible by 5 *)
  begin
    resultStr := 'BUZZ';
  end;

  if( (counter mod 3 = 0) and (counter mod 5 = 0) ) then (* divisible by both 3 and 5 *)
  begin
    resultStr := 'FIZZBUZZ';
  end;

  (* end procedure fizzBuzz *)
end;
```

Figure 2: PASCAL code for the fizzBuzz procedure.

We need to create a test application for the procedure fizzBuzz. Let's call the test application **testFizzBuzz** . We'll compile our test application using the **Free Pascal Compiler** which is online at

https://www.onlinegdb.com/online_pascal_compiler

We also need to write a test procedure inside the application that calls fizzBuzz with an input value and retrieves the result produced by fizzBuzz. Let's call this procedure **testCase** . This procedure would compare the actual result that was returned by fizzBuzz with the result that we were expecting. If they matched, testCase will tell us that the test case passed. If they didn't match, we would be told that the test case failed. The testCase procedure would generate a one line report of the status of that particular test case. The report would look like this:

```
Test case 1   fizzBuzz with input of 1    expecting 1    returned 1    PASSED
Test case 2   fizzBuzz with input of 16   expecting 17   returned 16   FAILED
```

Our testCase procedure would look like this:

```
procedure testCase (   testCaseNumber : integer;
                       inputValue : integer;
                       expectedResultStr : string   );

var
  resultStr : string;

begin

  fizzBuzz (inputValue, resultStr);

  write('Test case ');
  write(inttostr(testCaseNumber));
  write(' ');
  write('fizzBuzz with input of ');
  write(inttostr(inputValue));
  write(' ');
  write('expecting ');
  write(expectedResultStr);
  write(' ');
  write('returned ');
  write(resultStr);

  if(expectedResultStr = resultStr) then
  begin
    write(' ');
    writeLn('PASSED');
  end
  else
  begin
    write(' ');
    writeLn('FAILED');
  end;

  (* end procedure testCase *)
end;
```

Figure 3: PASCAL code for the testCase procedure.

In the Main section of our test application we set up several test cases, as follows:

```
(* testCase (testCaseNumber, inputValue, expectedResultStr) *)
testCase(1, 1, '1');
testCase(2, 2, '2');
testCase(3, 3, 'FIZZ');
testCase(4, 4, '4');
testCase(5, 5, 'BUZZ');
testCase(6,15, 'FIZZBUZZ');
testCase(7, 16, '17'); (* should fail *)
```

Figure 4: PASCAL code for the fizzBuzz procedure test cases.

Test cases 1 through 5 would test the numbers 1 through 5. This would cover 3 of the requirements: NOT divisible by 3 OR 5, divisible by 3 ONLY and divisible by 5 ONLY. Test case 6 would cover the requirement where the input (15) is divisible by BOTH 3 AND 5. Test case 7 is designed to show that our testing code works when a test case fails. These 7 test cases should cover the 4 original requirements. Since we have covered all the requirements, we should have also covered all the paths in this procedure. After compiling our test application, we can run it to see the results.

```
testFizzBuzz
For help, type testFizzBuzz -h.

testFizzBuzz results:
Test case 1    fizzBuzz with input of 1     expecting 1            returned 1            PASSED
Test case 2    fizzBuzz with input of 2     expecting 2            returned 2            PASSED
Test case 3    fizzBuzz with input of 3     expecting FIZZ         returned FIZZ         PASSED
Test case 4    fizzBuzz with input of 4     expecting 4            returned 4            PASSED
Test case 5    fizzBuzz with input of 5     expecting BUZZ         returned BUZZ         PASSED
Test case 6    fizzBuzz with input of 15    expecting FIZZBUZZ     returned FIZZBUZZ     PASSED
Test case 7    fizzBuzz with input of 16    expecting 17           returned 16           FAILED
End of program.
```

Figure 5: Results of the test cases within the test application.

Our procedure has a single numeric input value, and we return a single result in a string. Since we have not placed any restrictions on the input, we could have thousands of possible input values. FizzBuzz only requires that we use the values from 1 to 100, so we would need at least 100 test cases to test every input value that we would expect to encounter. What happens if we input a zero? What happens if we input a value higher than 100? Would they cause an undesired result? Do we need to add code to handle inputs that are out of range? If we were to define the input as a byte, we would still have 256 possible input values (0 to 255). Assuming that we only tested the values 1 to 100, it might be a good idea to test 0 and the values greater than 100, just to be thorough. This would require at least 256 test cases.

7 Unit testing - Web application (JavaScript)

Let's write a JavaScript function that we can use in a Web page or Web application. Function fizzBuzz mimics the fizzBuzz procedure that we looked at earlier. It has a single input parameter. Since we are going to count from 1 to 100, let's call the input parameter "counter". We evaluate the input parameter based on the 4 requirements listed above and return the appropriate result based on which of the requirements match the input. Our function looks like this:

```
// fizzBuzz1.js
function fizzBuzz( counter ) {

   var resultStr = "";        // the result to return
   var divisibleBy3 = false;        // used to improve readability
   var divisibleBy5 = false;        // used to improve readability

   if ( (counter%3) === 0 )        // used to improve readability
   {
      // Begin divisible by 3

      divisibleBy3 = true;

      // End divisible by 3
   }

   if ( (counter%5) === 0 )        // used to improve readability
   {
      // Begin divisible by 5

      divisibleBy5 = true;

      // End divisible by 5
   }

   resultStr = counter;       // default, just return the input value ( the counter )
                              // If none of the conditions match, we want to
                              // return the input without doing anything

   // Begin testing the conditions to determine the correct response.

   if ( divisibleBy3 && divisibleBy5 ) {
      // Begin divisible by both 3 and 5

      resultStr = "FIZZBUZZ";

      // End divisible by both 3 and 5
   }

   if ( divisibleBy3 && !divisibleBy5 ) {
      // Begin divisible by 3 only

      resultStr = "FIZZ";

      // End divisible by 3 only
   }

   if ( !divisibleBy3 && divisibleBy5 ) {
      // Begin divisible by 5 only

      resultStr = "BUZZ";

      // End divisible by 5 only
   }

   return resultStr; // return the result

   // end function fizzBuzz
}
module.exports = { fizzBuzz }
```

Figure 6: JavaScript function fizzBuzz..

7 Instrumentation

We want to create a component or unit test of this module. The test coverage tool keeps track of which lines of the source code were executed. The simplest way to do this would be for the tool to use an array of counters. Every time a line of code executes, the corresponding counter would be incremented. The counters are part of what is called instrumentation. They operate in the background. [23] In order to illustrate the instrumentation, we'll show the it in a different color. Our fizzBuzz function might now be rendered simplistically like this:

```
// fizzBuzz1.js
function fizzBuzz( counter ) {
executionCounter[0]++;

   var resultStr = "";        // the result to return
   executionCounter[1]++;
   var divisibleBy3 = false;        // used to improve readability
   executionCounter[2]++;
   var divisibleBy5 = false;        // used to improve readability
   executionCounter[3]++;

   if ( (counter%3) === 0 )        // used to improve readability
   {
      executionCounter[4]++;
      // Begin divisible by 3

      divisibleBy3 = true;
      executionCounter[5]++;

      // End divisible by 3
   }
   executionCounter[6]++;

   if ( (counter%5) === 0 )        // used to improve readability
   {
      executionCounter[7]++;
      // Begin divisible by 5

      divisibleBy5 = true;
      executionCounter[8]++;

      // End divisible by 5
   }
   executionCounter[9]++;

   resultStr = counter;       // default, just return the input value ( the counter )
   executionCounter[10]++;
                              // If none of the conditions match, we want to
                              // return the input without doing anything

   // Begin testing the conditions to determine the correct response.

   if ( divisibleBy3 && divisibleBy5 ) {
      executionCounter[11]++;
      // Begin divisible by both 3 and 5

      resultStr = "FIZZBUZZ";
      executionCounter[12]++;

      // End divisible by both 3 and 5
   }
   executionCounter[13]++;

   if ( divisibleBy3 && !divisibleBy5 ) {
      executionCounter[14]++;
      // Begin divisible by 3 only

      resultStr = "FIZZ";
      executionCounter[15]++;

      // End divisible by 3 only
   }
   executionCounter[16]++;
```

*** Continues on the next page ***

```
    if ( !divisibleBy3 && divisibleBy5 ) {
        executionCounter[17]++;
        // Begin divisible by 5 only

        resultStr = "BUZZ";
        executionCounter[18]++;

        // End divisible by 5 only
    }
    executionCounter[19]++;

    return resultStr; // return the result
    executionCounter[20]++;

    // end function fizzBuzz
}
executionCounter[21]++;
module.exports = { fizzBuzz }
executionCounter[22]++;
```

Figure 7: JavaScript function fizzBuzz with instrumentation.

8 Cypress testing format

Our Unit test of fizzBuzz using Cypress would look like this:

```
// import fizzBuzz from '../../fizzBuzz1'

describe('Unit Test 1 - fizzBuzz', function () {
    const { fizzBuzz } = require('../../fizzBuzz1');

    before(() => {
        // check if the import worked correctly
        expect(fizzBuzz, 'fizzBuzz').to.be.a('function')
    })

    context('fizzBuzz1.js', function () {

    it('test case 1 - fizzBuzz with input of 1 returns "1" ', function () {
        cy.wrap({ fizzBuzz: fizzBuzz })
            .invoke('fizzBuzz', 1)
            .should('eq', '1')
        })

    })

})
```

Figure 8: Unit test of JavaScript function fizzBuzz written using Cypress.

Let's compare this to the same test written in Mocha.

```
const fizzBuzz = require('../index');
const assert = require('assert');

describe("Testing FizzBuzz", () => {

    it("fizzBuzz with input of 1 returns '1'", () => {
        assert.equal(fizzBuzz(1), "1");
    });

});
```

Figure 9: Unit test of JavaScript function fizzBuzz written using Mocha. [33]

If you're familiar with Jest, the test would look like this:

```
const fizzBuzz = require('./index');

describe("Testing FizzBuzz", () => {

   it("fizzBuzz with input of 1 returns '1'", () => {
      expect(fizzBuzz(1)).toBe("1");
   });

});
```

Figure 10: Unit test of JavaScript function fizzBuzz written using Jest. [33]

We create 7 tests in Cypress which are equivalent to the tests we saw previously. When we run the tests in Cypress, we get the following results:

```
✓ Test 1 - fizzBuzz
   ✓ fizzBuzz1.js
      ✓ test case 1 fizzBuzz with input of 1 expects "1"
      ✓ test case 2 fizzBuzz with input of 2 expects "2"
      ✓ test case 3 fizzBuzz with input of 3 expects "FIZZ"
      ✓ test case 4 fizzBuzz with input of 4 expects "4"
      ✓ test case 5 fizzBuzz with input of 5 expects "BUZZ"
      ✓ test case 6 fizzBuzz with input of 15 expects "FIZZBUZZ"
      × test case 7 fizzBuzz with input of 16 expects "17" *should fail*
         TEST BODY
         1 wrap {fizzBuzz: function() {} }
         2 invoke .fizzBuzz()
         3 -assert expected '16' to equal '17'
         ! AssertionError
         Timed out retrying after 4000ms: expected '16' to equal '17'
```

9 End-to-End testing

Cypress also allows us to do End-to-End testing of the entire product. This type of testing mimics how a real user would interact with the UI. [13] [58] We can do behavior based testing to ensure that the web application responds appropriately to the actions taken by the user.

This page intentionally left blank.

References

[1] T. Segal, “Profit Center: Characteristics vs. a Cost Center, With Examples,” *Investopedia*, 07-Dec-2020. Available: https://www.investopedia.com/terms/p/profitcentre.asp. [Accessed: 31-May-2024].

[2] “Test Coverage Tutorial: Comprehensive Guide With Best Practices.” *lambdatest.com*. Available: https://www.lambdatest.com/learning-hub/test-coverage. [Accessed: 31-May-2024].

[3] H. Shah, “A Detailed Guide on Test Coverage,” *Simform - Product Engineering Company*, 28-Jun-2021. Available: https://www.simform.com/blog/test-coverage/. [Accessed: 31-May-2024].

[4] T. Hamilton, “Test Coverage in Software Testing,” *Guru99*, 13-Apr-2024. Available: https://www.guru99.com/test-coverage-in-software-testing.html. [Accessed: 31-May-2024].

[5] “What is DO-178B?,” *AdaCore*. Available: https://www.adacore.com/industries/avionics/what-is-do-178b. [Accessed: 31-May-2024].

[6] D. Goswami, “Why does your Avionics Software Need DO-178B/C Compliance?,” *Qualitest*, 24-Dec-2020. Available: https://medium.com/qualitest/why-does-your-avionics-software-need-do-178b-c-compliance-4d1cba8c94a0. [Accessed: 31-May-2024].

[7] V. Nanda, “Test Coverage in Software Testing.” *tutorialspoint.com*. Available: https://www.tutorialspoint.com/test-coverage-in-software-testing. [Accessed: 31-May-2024].

[8] “Manual vs Automated Testing | Cypress Testing Tools.” *cypress.io*. Available: https://learn.cypress.io/testing-foundations/manual-vs-automated-testing. [Accessed: 30-May-2024].

[9] “End-to-End Testing: Your First Test with Cypress | Cypress Documentation,” *Cypress Documentation*, 09-May-2024. Available: https://docs.cypress.io/guides/end-to-end-testing/writing-your-first-end-to-end-test. [Accessed: 30-May-2024].

[10] “From Manual Testing to Automated Testing With Cypress - Our Story | Cogworks Blog.” *Cogworks Blog*. Available: https://www.wearecogworks.com/innerworks/cogworks-blog-archive/from-manual-to-automated-testing-with-cypress-our-story. [Accessed: 30-May-2024].

[11] “Best Practices | Cypress Documentation,” *Cypress Documentation*, 11-Apr-2024. Available: https://docs.cypress.io/guides/references/best-practices. [Accessed: 30-May-2024].

[12] “Testing Frameworks for Javascript | Write, Run, Debug | Cypress.” *cypress.io*. Available: https://www.cypress.io/. [Accessed: 30-May-2024].

[13] “Comprehensive Cypress Test Automation Guide | Cypress Documentation,” *Cypress Documentation*, 11-Apr-2024. Available: https://docs.cypress.io/guides/overview/why-cypress. [Accessed: 30-May-2024].

[14] “Key Differences | Cypress Documentation,” *Cypress Documentation*, 14-Dec-2022. Available: https://docs.cypress.io/guides/overview/key-differences. [Accessed: 30-May-2024].

[15] “How Cypress Works | End to end and component testing tools.” *cypress.io*. Available: https://www.cypress.io/how-it-works. [Accessed: 30-May-2024].

[16] G. Hegde, “Getting Started with Cypress Test Automation : Tutorial,” *BrowserStack*. Available: https://browserstack.wpengine.com/guide/cypress-automation-tutorial/. [Accessed: 30-May-2024].

[17] G. Hegde and P. Bhat, “How to perform Component Testing using Cypress,” *BrowserStack*, 03-Oct-2022. Available: https://browserstack.wpengine.com/guide/cypress-component-testing/. [Accessed: 30-May-2024].

[18] K. Pathak, “How to Perform Component Testing using Cypress,” *Medium*, 29-Jan-2023. Available: https://kailash-pathak.medium.com/how-to-perform-component-testing-using-cypress-3bbe74af7492. [Accessed: 30-May-2024].

[19] F. Hric, “Component testing in Cypress: What is it and why it’s important,” *Automated Visual Testing | Applitools*, 15-Dec-2022. Available: https://applitools.com/blog/component-testing-in-cypress-what-is-it-and-why-its-important/. [Accessed: 30-May-2024].

References (continued)

[20] Technocrat, “How to write Test Case in Cypress: (with testing example),” *BrowserStack*. Available: https://browserstack.wpengine.com/guide/how-to-write-test-case-in-cypress/. [Accessed: 30-May-2024].

[21] D. Greene, “Wow! Cypress can run unit tests! 🥳,” *DEV Community*, 13-Nov-2020. Available: https://dev.to/dgreene1/wow-cypress-can-run-unit-tests-15l5. [Accessed: 29-May-2024].

[22] “Recipes | Cypress Documentation,” *Cypress Documentation*, 11-Apr-2024. Available: https://docs.cypress.io/examples/recipes. [Accessed: 29-May-2024].

[23] “Code Coverage | Cypress Documentation,” *Cypress Documentation*, 25-Mar-2024. Available: https://docs.cypress.io/guides/tooling/code-coverage. [Accessed: 29-May-2024].

[24] G. Bahmutov, “Readable Cypress.io tests,” *Better world by better software*, 01-Apr-2019. Available: https://glebbahmutov.com/blog/readable-tests/index.html. [Accessed: 29-May-2024].

[25] “cypress-example-recipes/examples/unit-testing__application-code at master · cypress-io/cypress-example-recipes,” *GitHub*. Available: https://github.com/cypress-io/cypress-example-recipes/tree/master/examples/unit-testing__application-code. [Accessed: 29-May-2024].

[26] “Cypress Testing Framework Tutorial With Examples | Sauce Labs.” *Sauce Labs*. Available: https://saucelabs.com/resources/blog/getting-started-with-cypress. [Accessed: 29-May-2024].

[27] Morrismoses, “Cypress For Test Automation: A Step-by-Step Guide,” *Medium*, 22-Apr-2024. Available: https://medium.com/@morrismoses149/cypress-for-test-automation-a-step-by-step-guide-93cb47163d05. [Accessed: 29-May-2024].

[28] S. Nadeesha, “Making a simple Test Automation Framework with Cypress.io.” *LinkedIn*. Available: https://www.linkedin.com/pulse/making-simple-test-automation-framework-cypressio-supun-nadeesha. [Accessed: 29-May-2024].

[29] “Selenium vs Cypress vs alternatives: choosing the right framework for web testing.” *zebrunner.com*. Available: https://zebrunner.com/blog-posts/what-is-cypress-and-is-it-a-real-alternative-to-selenium. [Accessed: 29-May-2024].

[30] “How to use Cypress Testing Framework?,” *Testbytes Softwares*. Available: https://www.testbytes.net/blog/cypress-testing/. [Accessed: 29-May-2024].

[31] “Reporters | Cypress Documentation,” Mar. 25, 2024. *Cypress Documentation*. Available: https://docs.cypress.io/guides/tooling/reporters. [Accessed: May 29, 2024].

[32] “Mocha - the fun, simple, flexible JavaScript test framework.” *mochajs.org*. Available: https://mochajs.org/. [Accessed: May 29, 2024].

[33] “Jest vs Mocha: Which one Should You Choose?,” *GeeksforGeeks*, Feb. 27, 2024. Available: https://www.geeksforgeeks.org/jest-vs-mocha-which-one-should-you-choose/. [Accessed: May 29, 2024].

[34] “Bundled Libraries | Cypress Documentation,” *Cypress Documentation*, Dec. 15, 2022. Available: https://docs.cypress.io/guides/references/bundled-libraries. [Accessed: May 29, 2024].

[35] “Chai.” *chaijs.com*. Available: https://www.chaijs.com/. [Accessed: May 29, 2024].

[36] D. Vasudevan, “How can you use Jest or Cypress to ensure front-end code quality?” *LinkedIn*. Available: https://www.linkedin.com/advice/3/how-can-you-use-jest-cypress-ensure-front-end-code-k6bnf. [Accessed: May 29, 2024].

[37] “Cypress vs Jest | Top 15 Key Differences,” *Testsigma Blog*, Apr. 02, 2024. Available: https://testsigma.com/blog/cypress-vs-jest/. [Accessed: May 29, 2024].

[38] “Testing Types | Cypress Documentation,” *Cypress Documentation*, Jan. 30, 2023. Available: https://docs.cypress.io/guides/core-concepts/testing-types. [Accessed: May 29, 2024].

References (continued)

[39] “Karma - Spectacular Test Runner for Javascript.” Available: https://karma-runner.github.io/latest/index.html. [Accessed: May 29, 2024].

[40] K. Choksi, “How to write unit tests with Jasmine & Karma?,” *Simform Engineering*, Jun. 01, 2023. Available: https://medium.com/simform-engineering/how-to-write-unit-tests-with-jasmine-karma-f1908bdeb617. [Accessed: May 29, 2024].

[41] Testim, “Karma JS Testing: What, Why, and How to Get Going Right Now,” *AI-driven E2E automation with code-like flexibility for your most resilient tests*, Dec. 11, 2021. Available: https://www.testim.io/blog/karma-js-testing-what-why-and-how-to-get-going-right-now/. [Accessed: May 29, 2024].

[42] “How to Install Cypress for Test Automation,” *BrowserStack*. Available: https://browserstack.wpengine.com/guide/cypress-installation-for-test-automation/. [Accessed: 30-May-2024].

[43] “Code Coverage | Cypress Documentation,” *Cypress Documentation*, 25-Mar-2024. Available: https://docs.cypress.io/guides/tooling/code-coverage. [Accessed: 30-May-2024].

[44] “Open the App with Cypress: Step-by-Step Guide | Cypress Documentation,” *Cypress Documentation*, 19-Apr-2024. Available: https://docs.cypress.io/guides/getting-started/opening-the-app. [Accessed: 30-May-2024].

[45] “End-to-End Testing: Your First Test with Cypress | Cypress Documentation,” *Cypress Documentation*, 09-May-2024. Available: https://docs.cypress.io/guides/end-to-end-testing/writing-your-first-end-to-end-test. [Accessed: 30-May-2024].

[46] “Installing Cypress and writing your first test | Cypress Testing Tools.” Available: https://learn.cypress.io/testing-your-first-application/installing-cypress-and-writing-your-first-test. [Accessed: 30-May-2024].

[47] admin, “Cypress Code Coverage,” *testomat.io*, 07-Feb-2024. Available: https://testomat.io/blog/cypress-code-coverage/. [Accessed: 29-May-2024].

[48] “Setting up Cypress Code Coverage,” *BrowserStack*. Available: https://browserstack.wpengine.com/guide/cypress-code-coverage/. [Accessed: 29-May-2024].

[49] “Node.js — Run JavaScript Everywhere.” *nodejs.org*. Available: https://nodejs.org/en. [Accessed: 31-May-2024].

[50] “Node.js Tutorial.” *W3Schools*. Available: https://www.w3schools.com/nodejs/. [Accessed: 31-May-2024].

[51] “Node.js — Introduction to Node.js.” *nodejs.org*. Available: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs. [Accessed: 31-May-2024].

[52] M. Wankhede, “Automation testing with Cypress, Mocha, and JavaScript,” *TO THE NEW BLOG*, 17-Feb-2023. Available: https://www.tothenew.com/blog/automation-testing-with-cypress-mocha-and-javascript/. [Accessed: 06-Jun-2024].

[53] C. Adams, “Cypress Limitations you should be aware of,” *Codoid*, 12-Jan-2023. Available: https://codoid.com/automation-testing/cypress-limitations-you-should-be-aware-of/. [Accessed: 06-Jun-2024].

[54] “Mocha vs Cypress.io comparison of testing frameworks.” *Knapsack Pro*. Available: https://knapsackpro.com/testing_frameworks/difference_between/mochajs/vs/cypress-io. [Accessed: 06-Jun-2024].

[55] K. Halder, “Automated Testing with Cypress,” *The Startup*, 03-May-2020. Available: https://medium.com/swlh/automated-testing-with-cypress-17bf74bfd97d. [Accessed: 06-Jun-2024].

[56] “Comparing cypress vs. jasmine vs. jest vs. karma vs. mocha,” *NPMCompare*. Available: https://npmcompare.com/compare/cypress,jasmine,jest,karma,mocha. [Accessed: 06-Jun-2024].

[57] “Comparing cucumber vs. cypress vs. jasmine vs. mocha vs. nightwatch,” *NPMCompare*. Available: https://npmcompare.com/compare/cucumber,cypress,jasmine,mocha,nightwatch. [Accessed: 06-Jun-2024].

[58] R. Kumari, “Cypress vs Cucumber - Which One to Choose?,” *Testsigma Blog*, 03-Feb-2024. Available: https://testsigma.com/blog/cypress-vs-cucumber/. [Accessed: 06-Jun-2024].

References (continued)

[59] “Cucumber vs Cypress.io comparison of testing frameworks.” *Knapsack Pro*. Available: https://knapsackpro.com/testing_frameworks/difference_between/cucumber/vs/cypress-io. [Accessed: 06-Jun-2024].

[60] “Selenium,” *Selenium*. Available: https://www.selenium.dev/. [Accessed: 09-Jun-2024].

[61] “WebDriver,” *Selenium*, 29-Mar-2024. Available: https://www.selenium.dev/documentation/webdriver/. [Accessed: 09-Jun-2024].

[62] “WebDriver.” *W3C*, 2018. Available: https://www.w3.org/TR/webdriver1/. [Accessed: 09-Jun-2024].

[63] “Getting started,” *Selenium*. Available: https://www.selenium.dev/documentation/webdriver/getting_started/. [Accessed: 09-Jun-2024].

[64] “Write your first Selenium script,” *Selenium*. Available: https://www.selenium.dev/documentation/webdriver/getting_started/first_script/. [Accessed: 09-Jun-2024].

[65] “Supported Browsers,” *Selenium*, 20-Sep-2022. Available: https://www.selenium.dev/documentation/webdriver/browsers/. [Accessed: 09-Jun-2024].

[66] “Downloads,” *Selenium*. Available: https://www.selenium.dev/downloads/. [Accessed: 09-Jun-2024].

[67] “Selenium WebDriver Tutorial - javatpoint,” *www.javatpoint.com*. Available: https://www.javatpoint.com/selenium-webdriver. [Accessed: 09-Jun-2024].

[68] “WebDriver For Mobile Browsers,” *Selenium*, 12-Jan-2022. Available: https://www.selenium.dev/documentation/legacy/selenium_2/mobile/. [Accessed: 09-Jun-2024].

[69] C. Tozzi , “Can Selenium Be Used for Mobile Testing?” *Sauce Labs Inc.*, 04-Feb-2023. Available: https://saucelabs.com/resources/blog/can-selenium-be-used-for-mobile-testing.
[Accessed: 09-Jun-2024].

[70] “Welcome - Appium Documentation.” *Appium Documentation*. Available: https://appium.io/docs/en/latest/. [Accessed: 10-Jun-2024].

[71] “Appium in a Nutshell - Appium Documentation.” *Appium Documentation*. Available: https://appium.io/docs/en/latest/intro/. [Accessed: 10-Jun-2024].

[72] “Selendroid: Selenium for Android.” *selendroid.io*. Available: http://selendroid.io/. [Accessed: 10-Jun-2024].

[73] “Mobile Testing - Selendroid Framework.” *www.tutorialspoint.com*. Available: https://www.tutorialspoint.com/mobile_testing/mobile_testing_selendroid_framework.htm#:~:text=Selendroid%20is%20a%20test%20automation,the%20Selenium%20Webdriver%20client%20API. [Accessed: 10-Jun-2024].

[74] “Appium vs Selendroid,” *GeeksforGeeks*, 10-Dec-2023. Available: https://www.geeksforgeeks.org/appium-vs-selendroid/. [Accessed: 10-Jun-2024].

This page intentionally left blank.

End of document