JavaScript Test Automation Frameworks

Six Essential Frameworks for Creating Automated Tests

JUSTIN ALBANO
SOFTWARE ENGINEER, IBM

CONTENTS

Web development has changed dramatically over the last three decades. At the advent of the Web, nearly all applications consisted of static content rendered in one or two browsers. Since that time, the Web has evolved into a colossal collection of dynamic web pages and applications rendered in half-a-dozen or more browsers on behalf of billions of users.

In addition to dramatic changes in web applications, JavaScript test frameworks have also morphed to keep pace with this ever-evolving ecosystem. In this Refcard, we will delve into the conceptual underpinnings of modern JavaScript test automation frameworks and explore six of the most popular frameworks available today. Along this journey, we will compare the different design and architecture decisions of each framework and how well these various trade-offs are suited for different requirements and contexts.

JAVASCRIPT TEST AUTOMATION FUNDAMENTALS

To understand JavaScript automated testing frameworks, we must first have a foundational understanding of how to approach testing in the general sense, including the purpose and structure of test cases and the mindset we use to create effective test cases.

A GENERAL APPROACH TO TESTING

A test framework is primarily responsible for running a **test suite** or a collection of **test cases**. Each test case is designed to exercise a single, logical unit of behavior in an application and ensure that the targeted behavior operates as expected. Generally, test cases are structured in three parts:

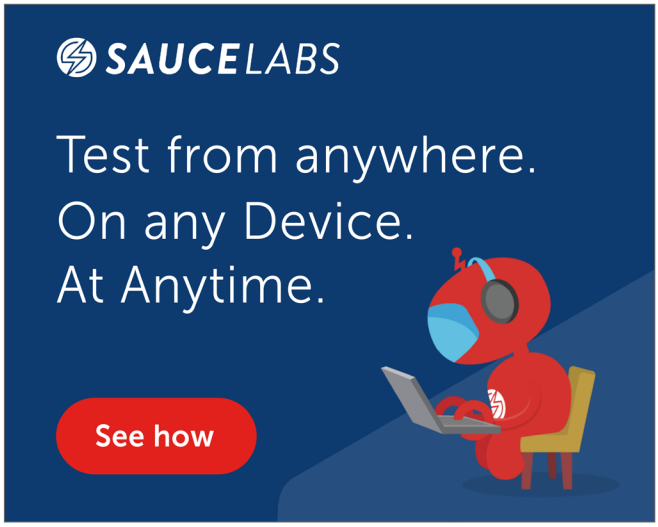
1. **Preconditions** – Assertions that must be true before the test can execute
2. **Command** – The logic of the test case that is intended to verify some behavior
3. **Postconditions** – Assertions that must be true once the test is complete

Written formally, using Hoare Logic, a test case can be expressed as:

$$\{P\}\ C\ \{Q\}$$

where:

- $\{P\}$ are the preconditions of the test case
- $\{C\}$ is the command to be executed
- $\{Q\}$ are the postconditions of the test case that we expect to be to be true



"Yeah, we don't need visual testing."

— Soon-to-be former employee

Your customers expect quality and consistency from your software. With Sauce Visual, you can find UI regressions before they do.

- Automatically detect inconsistencies across browsers, devices, and resolutions
- Programmatic comparisons avoid false positives
- Run functional and visual tests simultaneously
- Ship more frequently with lower risk

UI/Visual testing

One component of the Sauce Labs end-to-end testing solution

- Cross-browser testing
- Live testing
- Low-code testing NEW
- Mobile beta testing NEW
- Mobile app testing
- API testing NEW
- Error reporting NEW

Check out all of our testing solutions, schedule a demo, or start your FREE trial today

SAUCELABS

Develop with confidence

We can write tests in natural language — called Behavior-Driven Development (BDD) — using the following clauses:

1. **Given** – The environment and context of the test case
2. **When** – The logic that is under test
3. **Then** – The assertions about the results of the test

For example:

> Given I have addends of 3 and 5
> When I sum them
> Then the result is 8

Lastly, we can translate our test cases into code, as seen in the pseudocode example below of our previous BDD test case:

```
// Given
a = 3
b = 5

// When
result = a + b

// Then
assert result == 8
```

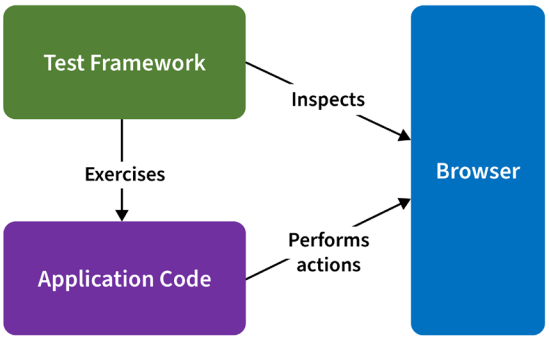
Since our test case is now expressed as code, we can execute it the same way we execute our application code. In practice, this means executing all, or a specific subset, of our test cases each time we make changes to our application. This ensures that **regressions** — bugs that cause previously working components to fail — are not introduced in our application.

Reporting is another important responsibility of a test framework. Apart from simply denoting whether each test passed or failed, frameworks are also responsible for describing why a test case failed. For example, if the value of `result` in our previous test case was `7`, we expect that our report would include a description of the failure, informing us that the actual result of the test, `7`, does not match our expected result, `8`.

TESTING JAVASCRIPT APPLICATIONS

The generalized approach to testing and test frameworks above holds true for nearly all programming languages and ecosystems. While JavaScript can be executed outside of a browser using Node.Js (Node), a vast majority of the JavaScript applications we see today are browser-based. This is unsurprising, as JavaScript has a significant advantage over nearly all other programming languages in this realm: All modern browsers include a JavaScript engine that allows JavaScript code to be executed natively within the browser, which brings its own set of advantages and challenges.

This tight relationship between JavaScript and the browser allows us to programmatically execute basic actions we take for granted, such as clicking a button or hovering over a specific Hypertext Markup Language (HTML) element. Due to JavaScript's capability to interact with the browser and execute these actions, JavaScript test frameworks must also be able to inspect the results of these actions, as seen in the following diagram:



For example, if we click a button, we may want to assert that a different HTML element changes to a specific color, a dialog box is displayed, or a Representational State Transfer (REST) request is made to some back-end service. Additionally, we may want to take screenshots or videos of a test case's results — or the entire execution of a test case — to ensure that the results match our aesthetic specifications (e.g., graphic designs).

Since users can choose from a host of different browsers (e.g., Chrome, Firefox, Safari, Edge), we may also want to ensure that our application behaves and appears as expected in multiple browsers. This adds an additional requirement to our test framework: interoperability with different browsers.

Conceptually, each browser has its own application programming interface (API) that allows an application to inspect the results of an interaction with the browser. Over the years, as more and more browsers with vastly different APIs became popular, standards were devised that abstract the details of each browser and allow test frameworks to interact with browsers in an agnostic fashion.

These advancements have led to three different types of JavaScript test frameworks:

1. **Standardized** – Uses a standard API, which ensures interoperability with all browsers that support the standard, but it can be inefficient and lack support for browser-specific features
2. **Non-Standardized** – Uses browser-specific protocols that have not been standardized, which may be more efficient and feature-rich, but it may not support all browsers

3. **Proprietary** – Uses a customized mechanism — such as proxying or code injection — to interact with browsers, which can be efficient and feature-rich, but it is more complex and may introduce quirky limitations

Each of these categories has its advantages and disadvantages, and the frameworks that use them excel and deteriorate in different ways. In the following section, we will look at six of the more prominent frameworks, exploring how they utilize one of the three approaches above to allow us to create automated tests in JavaScript.

ESSENTIALS OF TEST AUTOMATION FRAMEWORKS

As JavaScript automated testing frameworks have matured over the last two decades, six have risen as the most popular and can be categorized by their protocols:

1. World Wide Web Consortium (W3C) WebDriver – WebDriverIO and Nightwatch
2. Chrome DevTools Protocol (CDP) – Puppeteer and Playwright
3. **Proprietary** – TestCafe and Cypress

In this section, we will delve into how we can install and run each framework, walk through an exploratory example of a simple test case, and compare each framework to understand which is the best fit for our use cases. **Note:** The examples that follow assume Node and the Node Package Manager (NPM) are already installed.

WEBDRIVERIO

WebDriverIO is an open-source project that utilizes the W3C WebDriver standard to interact with browsers in a truly cross-compatible manner. All modern web browsers, including Chrome, Firefox, Safari, and Edge, support the WebDriver standard, which means that WebDriverIO supports nearly all browsers out of the box. Unfortunately, the Web has changed significantly since the WebDriver standard was devised, and it lags in its support for modern web development and testing.

To supplement deficiencies of the WebDriver standard, WebDriverIO also supports CDP, which enables the framework to use the standardized WebDriver interactions, while also allowing us to perform additional, more modern interactions with the browser. This CDP interaction occurs over WebSockets to the browser, which means WebDriverIO can interact with the browser without the need for a patched browser or external service.

To run a WebDriverIO test, we must first create a new Node project:

```
mkdir webdriverio-examples
cd webdriverio-examples/
npm init -y
```

Note that the `-y` flag — "yes" — accepts all prompts during the initialization process, removing the need for manual interaction. Next, we must install the WebDriverIO Command Line Interface (CLI) tool through NPM and configure the project using the `wdio config` command:

```
npm i --save-dev @wdio/cli
npx wdio config -y
```

The configuration command creates a `wdio.conf.js` file that contains the WebDriverIO configuration for our project. Within this configuration file, we can see the following field:

```
specs: [
    './test/specs/**/*.js'
]
```

This `specs` field specifies the location of our test case specifications. By default, a test file, `example.e2e.js`, will be created under the `./test/specs` directory. We can run this test using the following command:

```
npx wdio run wdio.conf.js
```

The `example.e2e.js` test will open a browser window and attempt to authenticate using a login form. Once the test is complete, we can see the successful results of our test case in the WebDriverIO output:

```
Spec Files: 1 passed, 1 total (100% completed) in
00:00:08
```

For more information, see the following WebDriverIO resources:

- What is WebDriver.IO?
- Automation Protocol
- Getting Started
- Boilerplate Projects
- NPM

NIGHTWATCH

Nightwatch is an open-source project similar to WebDriverIO — both use the WebDriver standard to interact with the browser — but Nightwatch focuses more on usability. This focus on ease-of-use results in cleaner syntax, selection through JavaScript and CSS or XPath, and support for Page Object Models (POMs).

The usability of Nightwatch is also reflected in how quickly a test case can be created. Similar to WebDriverIO, we must first create a new Node project:

```
mkdir nightwatch-examples
cd nightwatch-examples/
npm init -y
```

To install Nightwatch, we can run the following command:

```
npm i --save-dev nightwatch geckodriver \
  chromedriver
```

We can then run an example script, **ecosia.js** — contained in **node_modules/nightwatch/examples/tests/** — by running the following command:

```
npx nightwatch \
  node_modules/nightwatch/examples/tests/ecosia.js
```

The **ecosia.js** script performs the following operations:

1. Opens the Ecosia Search Engine — https://www.ecosia.org/
2. Asserts that the title is **Ecosia**
3. Asserts that the search box is visible
4. Enters **nightwatch** into the search box
5. Asserts that the **Search** button is visible
6. Clicks the **Search** button
7. Asserts that the results contain **Nightwatch.js**

After we run this test case, we can see that all four of our assertions have passed:

```
[Ecosia.org Demo] Test Suite
============================
i Connected to localhost on port 4444 (9932ms).
  Using: firefox (87.0) on linux 5.4.0-71-generic
platform.

✓ Running Demo test ecosia.org:

✓ Element <body> was visible after 466
milliseconds.
✓ Testing if the page title contains 'Ecosia'
(11ms)
✓ Testing if element <input[type=search]> is
visible (72ms)
✓ Testing if element <button[type=submit]> is
visible (56ms)
✓ Testing if element <.mainline-results> contains
text 'Nightwatch.js' (3446ms)

OK. 5 assertions passed. (7.359s)
```

Compared to WebDriverIO, Nightwatch is quick to start and test cases are simple to implement. This begs the question: Why use WebDriverIO, then? What Nightwatch gains in simplicity and agility, it sacrifices in control. To reduce the burden on developers, Nightwatch abstracts a large portion of the details about the interaction of the framework with the browser.

WebDriverIO, on the other hand, exposes many of these details, which allows us to fine-tune and tweak our test configuration as needed. In general, Nightwatch should be preferred when simplicity and ease-of-use are a priority, while WebDriverIO should be used when a greater level of control is required.

For more information, see the following Nightwatch resources:

- Getting Started
- Developer Guide
- GitHub
- NPM

PUPPETEER

Puppeteer is an open-source project maintained by the Google Chrome DevTools team that provides an abstracted API that interacts with Chrome and Chromium browsers over CDP. By default, Puppeteer runs headlessly, but it can be configured to run using Chrome or Chromium — in the same manner as WebDriverIO and Nightwatch — as well.

To get started with Puppeteer, we must create a new Node project:

```
mkdir puppeteer-examples
cd puppeteer-examples/
npm init -y
```

Next, we need to install the Puppeteer packages:

```
npm i --save-dev puppeteer
```

Note that the **puppeteer** package installs Puppeteer and an accompanying version of the Chromium browser.

If we want to forgo the installation of Chromium, we can install the **puppeteer-core** package instead. Regardless of our selection, we can create a test script, **example.js** — from the official Puppeteer Usage page — that opens https://example.com/ and takes a screenshot of the browser window:

```
const puppeteer = require('puppeteer');

(async () => {
    const browser = await puppeteer.launch();
    const page = await browser.newPage();
    await page.goto('https://example.com');
    await page.screenshot({ path: 'example.png'
});

    await browser.close();
})();
```

We can then run this script using the following command:

```
node example.js
```

Once the command completes, we can see a screenshot of https://example.com/ in the current directory:

This is only a small fraction of the capability that Puppeteer contains. Puppeteer also allows us to programmatically perform nearly all the actions that we can perform manually in a browser, including:

- Providing input
- Interacting with forms
- Navigating Single-Page Applications (SPAs)
- Obtaining performance timing information
- Testing Chrome extensions

For more information, see the following Puppeteer resources:

- GitHub
- Google Web Developer page
- NPM

PLAYWRIGHT

Playwright is the spiritual successor to Puppeteer. It was created by Microsoft in 2020 (by the same team that created the original Puppeteer at Google) with the goal of bringing the same rich functionality supported by Puppeteer to all mainstream browsers. Instead of utilizing standard APIs for Firefox and WebKit browser engines, Playwright patches these engines to enable support for its APIs. This allows Playwright to support Chromium, Firefox, and WebKit on Windows, Linux, and MacOS at the expense of requiring that test cases be executed in a different browser than the one a user will experience (i.e., a patched browser).

To create a new test case, we must create a new Node project:

```
mkdir playwright-examples
cd playwright-examples/
npm init -y
```

Once we have created the new project, we can install the `playwright` package. We must also install the Playwright dependencies using the `npx playwright install-deps` command.

This command ensures that the browsers we require — Chromium, Firefox, and WebKit — are also installed and ready for use. We can install these packages using the following commands:

```
npm i --save-dev playwright
sudo npx playwright install-deps
```

We can now create a new test case, `example.js` — provided by Playwright — with the following content:

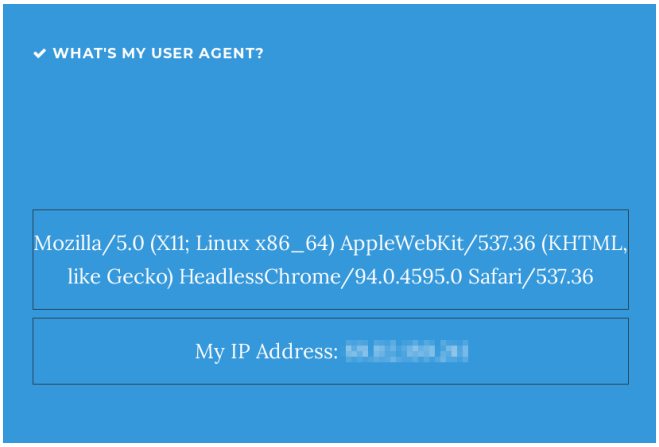
```
const playwright = require('playwright');

(async () => {
  for (const browserType of ['chromium',
'firefox', 'webkit']) {
    const browser = await playwright[browserType].
launch();
    const context = await browser.newContext();
    const page = await context.newPage();
    await page.goto('http://whatsmyuseragent.
org/');
    await page.screenshot({ path: `example-
${browserType}.png` });
    await browser.close();
  }
})();
```

This test case opens http://whatsmyuseragent.org/ in Chrome, Firefox, and Safari, and it takes a screenshot of the web page in each browser. We can run this test case using the following command:

```
node example.js
```

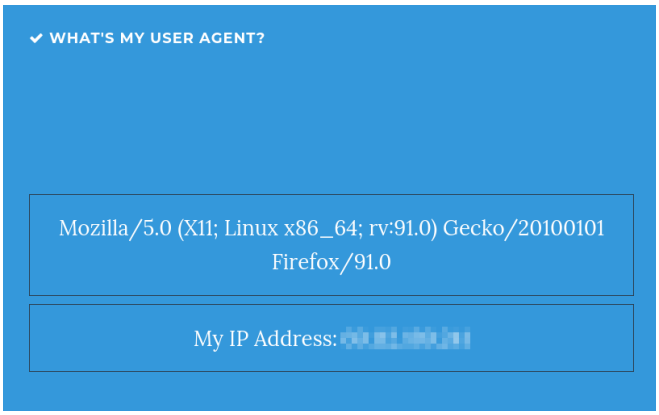
Once execution completes, we can see the following three screenshots (note the contents of each page differs based on the browser in which it was loaded):

Screenshot 1: Chromium

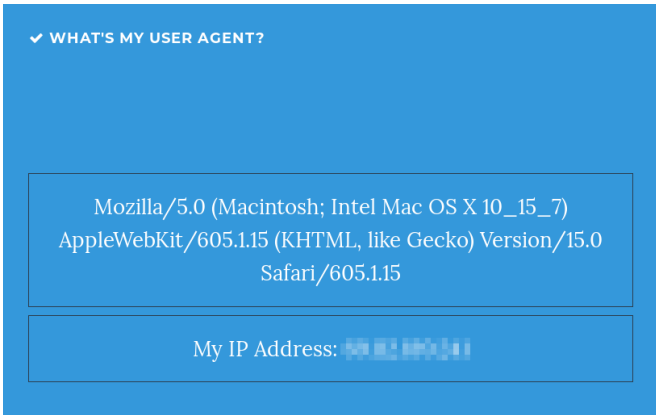


SEE SCREENSHOTS 2 AND 3 ON NEXT PAGE

Screenshot 2: Firefox



Screenshot 3: WebKit



Although Playwright extends the features of Puppeteer to a wider array of browsers, it does not replace Puppeteer. While it does offer a greater degree of interoperability, Playwright requires that patched browser releases be used — which introduces a disparity between the browsers used by our applications' customers and the browsers we use for testing.

For more information, see the following Playwright resources:

- GitHub
- NPM

TESTCAFE

TestCafe is an open-source project that acts as a proxy server and injects the code used for testing. In practice, TestCafe executes a test case that exercises a developer-specified URL, which is changed by TestCafe to the URL of the proxy server, injecting the HTML and JavaScript necessary to execute the test. This page is then delivered to the browser and the test case is executed. When more advanced interaction is needed, developers can use Client Scripts and Client Functions. TestCafe also automatically waits for page objects to load, which reduces the burden on developers when accessing certain elements on a web page.

TestCafe is a powerful tool, and its architecture differs greatly from the previous frameworks we have seen. This proxy architecture ensures that tests are executed quickly and a vast array of browsers are supported. The catch is that events — such as a button click — are not executed natively in a browser, which differs from the user experience and can lead to disparity in testing.

For example, a button may not be clickable when rendered by a browser, but it is possible that a TestCafe test case can click the button anyway.

To run a TestCafe test case, we must first create a new Node project:

```
mkdir testcafe-examples
cd testcafe-examples/
npm init -y
```

We can then install the `testcafe` package using the following command:

```
npm i --save-dev testcafe
```

Next, we can create a simple test case, `example.js` — provided by TestCafe — that enters text into an input field and clicks a submit button:

```
import { Selector } from 'testcafe';

fixture `Getting Started`
    .page `http://devexpress.github.io/testcafe/
example`;

test('My first test', async t => {
    await t
        .typeText('#developer-name', 'John Smith')
        .click('#submit-button');
});
```

Finally, we can execute our test case using the following command:

```
npx testcafe chrome example.js
```

The `testcafe` command accepts two arguments: (1) the alias of the browser to use and (2) the test script to execute. In our case, we want to execute our test case in Chrome, so we supply an alias of `chrome`.

A complete list of browser aliases can be found on the Test Cafe Browsers page.

Once the test successfully completes, the `testcafe` tool outputs the following, denoting that our test case, `My first test`, has passed:

```
Running tests in:
 - Chrome 93.0.4577.63 / Linux 0.0

 Getting Started
 ✓ My first test

 1 passed (8s)
```

For more information, see the following TestCafe resources:

- Getting Started
- Guides
- GitHub
- NPM

CYPRESS

Cypress is an open-source project that runs test cases directly within the browser. Unlike other test frameworks that make remote requests to the API of the browser or a standardized API, Cypress runs as a Node server process, which means that it runs in the same event-loop as our application under test. This has many distinct advantages over alternative approaches, including:

- Faster test execution
- Access to the network layer and web traffic
- Ability to execute operating system tasks (e.g., taking screenshots or video)
- Access to useful debugging information

Like TestCafe, Cypress also supports automatic waiting, which removes the need for developers to add artificial waits for page elements to load. This architecture does have some drawbacks, though, including:

- Supports only Chrome-based and Firefox browsers
- Struggles with certain web security issues (although workarounds are available)
- Constrains each test to a single origin
- Lacks multi-tab support

Although Cypress brings a lot to the table in terms of its continuity with the application under test, this ability comes with some sacrifices. Cypress can be a great automated test framework, but its limitations should be well understood.

To run a Cypress test, we must first create a new Node project:

```
mkdir cypress-examples
cd cypress-examples/
npm init -y
```

Once we create a new project, we can install the `cypress` package:

```
npm i --save-dev cypress
```

With the `cypress` package installed, we can create a new test in the file, `cypress/integration/example_spec.js`:

```
mkdir -p cypress/integration/
touch cypress/integration/example_spec.js
```

Once we create the `example_spec.js` file, we can add the following content — provided by Cypress:

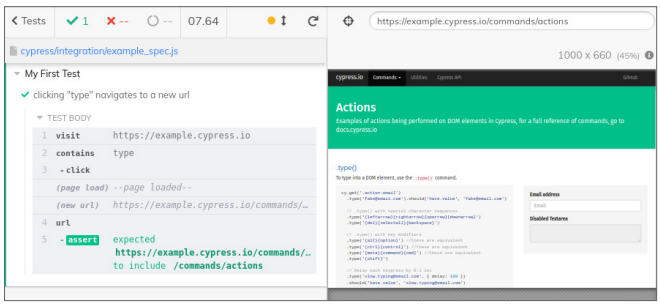
```
describe('My First Test', () => {
  it('clicking "type" navigates to a new url', ()
=> {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()

    cy.url().should('include', '/commands/
actions')
  })
})
```

Next, we can open the Cypress App using the following command:

```
npx cypress open
```

With the dashboard open, we can click on our `example_spec.js` file to open it:

Opening our `example_spec.js` file (by clicking on it) will open a browser window and execute our test case:



Apart from running our tests visually, we can also execute them using the command line, which removes the need to open our test case in a browser.

For more information, see the following Cypress resources:

- Getting Started Guide
- GitHub
- NPM

CONCLUSION

The Web has changed dramatically over the last 30 years, and the JavaScript automated test ecosystem has evolved alongside it to better suit the needs of modern web application development.

Currently, three general approaches have solidified:

1. Using standards such as WebDriver
2. Using non-standard APIs for each browser, such as CDP
3. Using proprietary approaches such as proxy servers and pure Node code

Regardless of the technique used, each JavaScript automated test framework has its advantages and sacrifices. As we have seen in this Refcard, the better we understand where each framework excels and falls short, the more equipped we are to select the test framework that best fits our environment and context.

WRITTEN BY JUSTIN ALBANO,
SOFTWARE ENGINEER, IBM

Justin Albano is a Software Engineer at IBM responsible for building software-storage and backup/recovery solutions for some of the largest worldwide companies, focusing on Spring-based REST API and MongoDB development. When not working or writing, he can be found practicing Brazilian Jiu-Jitsu, playing or watching hockey, drawing, or reading.

DZone, a Devada Media Property, is the resource software developers, engineers, and architects turn to time and again to learn new skills, solve software development problems, and share their expertise. Every day, hundreds of thousands of developers come to DZone to read about the latest technologies, methodologies, and best practices. That makes DZone the ideal place for developer marketers to build product and brand awareness and drive sales. DZone clients include some of the most innovative technology and tech-enabled companies in the world including Red Hat, Cloud Elements, Sensu, and Sauce Labs.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC 27709

888.678.0399 | 919.678.0300

Copyright © 2021 Devada, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means of electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.