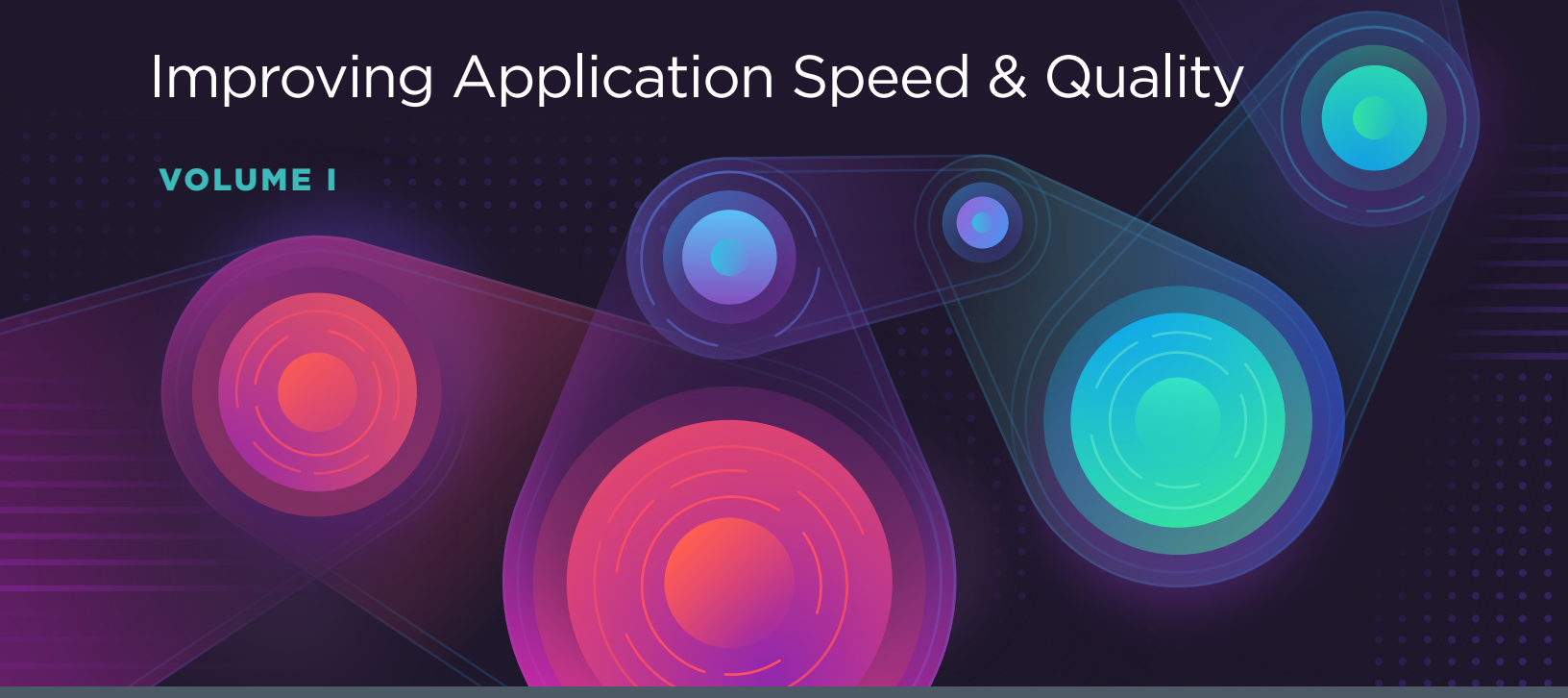
DZone

THE DZONE GUIDE TO

Automated Testing

Improving Application Speed & Quality

VOLUME I



RESEARCH PARTNER SPOTLIGHT

SAUCELABS

Key Research Findings

BY G. RYAN SPAIN
PRODUCTION COORDINATOR, DZONE

DEMOGRAPHICS

434 software professionals completed DZone's 2017 Automated Testing survey. Respondent demographics are as follows:

- 41% of respondents identify as developers or engineers, and 27% identify as developer team leads.
- The average respondent has 14 years of experience as an IT professional. 56% of respondents have 10 years of experience or more; 21% have 20 years or more.
- 40% of respondents work at companies headquartered in Europe; 34% work in companies headquartered in North America.
- 18% of respondents work at organizations with more than 10,000 employees; 20% work at organizations between 1,000 and 10,000 employees; and 28% work at organizations between 100 and 1,000 employees.
- 83% develop web applications or services; 49% develop enterprise business apps; and 36% develop native mobile applications.

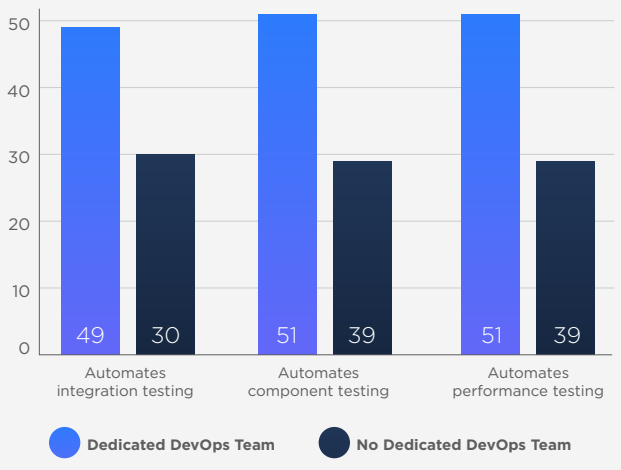
AUTOMATED TESTING

We asked survey respondents which tests in their organization's pipeline(s) are automated and which tests are performed manually. The most popular automated tests were integration (61%), component (58%), and performance (56%). While 22% of respondents automate none of these tests, 17% automate one of the three, 25% automate two, and 36% of respondents automate all three. For manual testing, the most common responses were user acceptance (78%), usability (70%), and story-level tests (63%). Across all manual and automated testing, manual testing had 36% more responses than automated testing. We also asked about a wide array of tools for automated testing. The most popular tools amongst our respondents were JUnit (61%), Selenium (46%), JMeter (45%), SoapUI (29%), and Cucumber (21%). 44% of respondents say their organization's Continuous Integration processes extend into an automated Continuous Delivery pipeline from code check-in to production deployment.

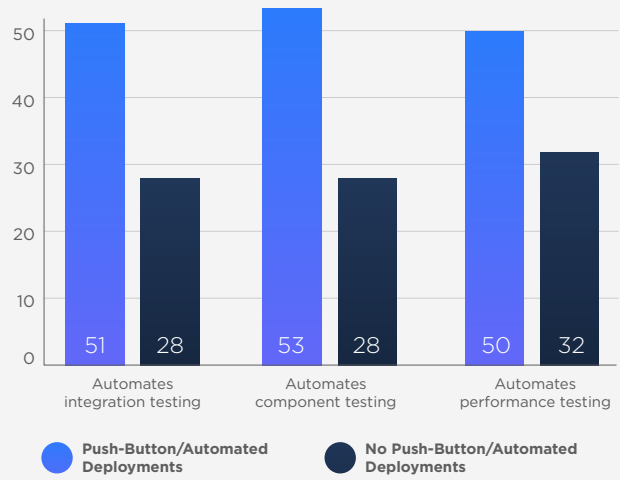
DEVOPS TRENDS

It's no surprise that automated testing and other DevOps practices go hand in hand. 49% of respondents working

Does your organization have a dedicated DevOps team?



Does your organization have either push-button or automated deployments?



at organizations with dedicated DevOps teams said one of that team's goals was introducing automation across the entire SDLC. Looking at the three most popularly automated tests, we found that respondents who said their organization automated these were much more likely to have one of these dedicated DevOps teams. 49% of respondents whose org automates integration tests said they have a DevOps team, as opposed to 30% who said their org does not automate integration tests. For component tests and performance tests, the difference was 51% with dedicated DevOps teams compared to 29% of respondents at orgs not automating these tests. Respondents answering that these tests are automated were also much more likely to say their organization performs push-button or automated deployments; for example, for integration tests, this difference was 51% vs. 28%. These respondents were also significantly more likely to believe their organization has fully achieved Continuous Delivery.

TOOLS AND AUTOMATED TESTING

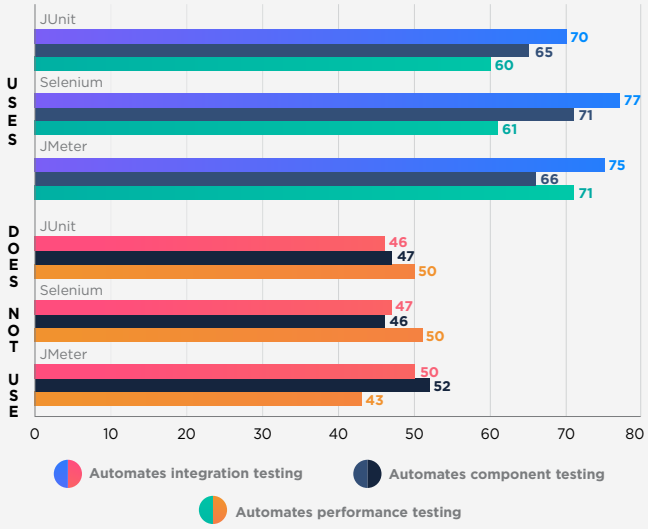
Responses regarding the most popular testing tools were also connected with these commonly automated tests. 70% of respondents whose organization uses JUnit said they automate integration tests, compared to 46% of non-JUnit users. For JMeter this difference was 75% vs. 50%, and for Selenium it was 77% vs. 47%. These trends apply to component and performance testing as well. Performance tests, while not as dramatically different as the others for users of JUnit and Selenium, were automated by 71% of JMeter-users, vs. 43% of non-users. Considering it was more likely for respondents to automate more than one of these popular tests, even starting test automation in one area seems to have an impact on other tests.

TOOLS AND LANGUAGES

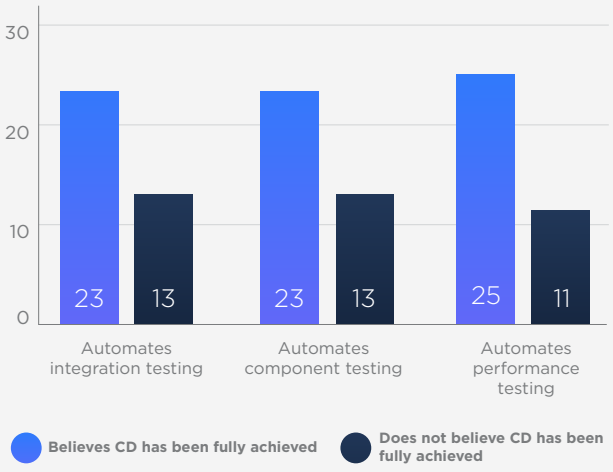
Given how language-specific testing tools are, the popularity of JUnit and JMeter amongst our respondents makes sense. 86% of respondents work at an organization that uses the Java ecosystem, and 62% of respondents work at organizations where Java is the primary programming language. 68% of respondents working at an organization that uses Java at all said their org uses JUnit, and 50% said they use JMeter. Of the respondents who work at primarily Java organizations, 77% said they use JUnit, and 53% said they are using JMeter. So these two open source testing tools are taking hold in the Java world.

**Margin of error calculated with 95% confidence interval*

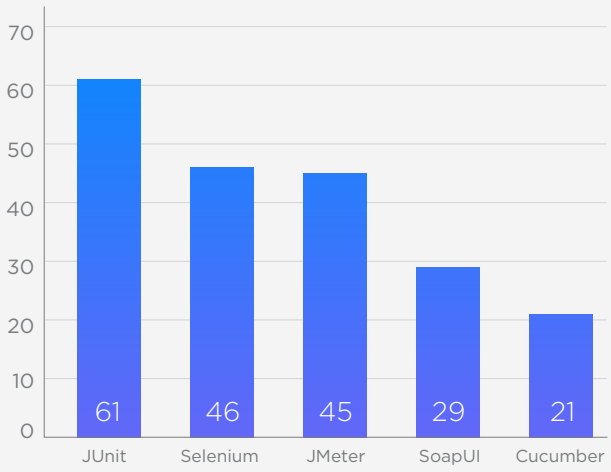
Does your organization automate integration, component, or performance testing?



Do you believe continuous delivery has been achieved in your organization?



Which of the following testing automation tools does your organization use?



THE CROSSROADS OF TESTING

Automated testing is seen as one of the key components of achieving Continuous Delivery in an organization. Thought leaders often suggest to automate everything. However, as of right now, not everything can, or should, be automated, particularly tests that rely on real-user input. So, developers stand at a crossroads: which tests can be automated now, and why? We asked almost 400 developers about which tests they performed manually and which were performed automatically.

MANUAL TESTING

USER ACCEPTANCE TESTS ensure that a user is satisfied with a product's function. These can be used to test user stories and ensure that they have been implemented correctly. Since this is based on the subjective opinions of end users, this is not conducive to automation.

78% of respondents perform manual user acceptance tests.

USABILITY TESTS These are tests that are performed directly with users to determine how easy it is to use a piece of software. Because results rely on the opinions of real users, this cannot easily be automated.

70% of users perform manual usability tests.

POST-DEPLOYMENT TESTS can vary between smoke checks to ensure the application is running and testing any major bugs that become apparent once the application is live. Because of the troubleshooting and unique circumstances involved, this is not an easy task to automate.

63% of DZone's audience perform manual post-deployment tests.

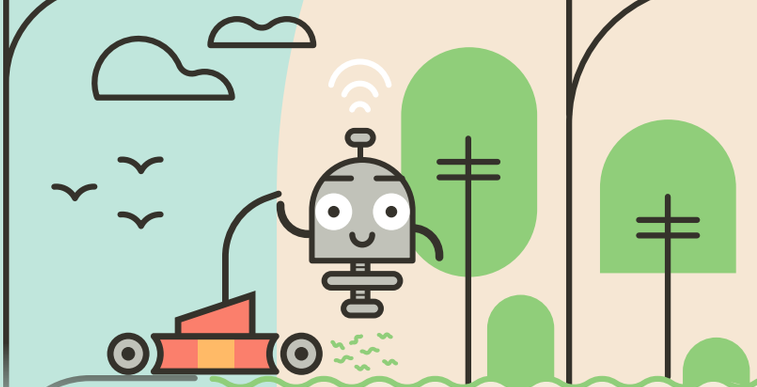
AUTOMATIC TESTING

Also called component tests, **UNIT TESTS** take individual pieces of an application's source code, called units, and ensures they are operating properly. Since these units don't rely on external dependencies, automating them is a relatively straightforward process.

58% of respondents perform automated unit tests.

INTEGRATION TESTS are performed when two pieces of software are combined and tested as a single unit. It should be an easy task to ensure that two pieces of software can communicate between each other, and can be easily automated.

61% of readers perform automated integration tests.



PERFORMANCE TESTS determine the speed or effectiveness of an application or network. Performance tests may be manually performed to determine the source of a performance bottleneck or diagnose an issue, but automated performance testing is useful to get a consistent, up-to-date picture of an application's performance. task to automate.

55% of users automate performance tests, while **45%** perform them manually.

Building Testable Apps

QUICK VIEW

01 Automated testing at the code level isn't something that can be "bolted on" after your code's complete—you need to start with testable code from the beginning.

02 Ensure you have clear acceptance criteria and great communication with your customers.

03 After that you need to focus on clean design, effective dependency management, and good craftsmanship principles like SOLID.

BY **JIM HOLMES**
EXECUTIVE CONSULTANT, **PILLAR TECHNOLOGY**

TAKING YOUR APPLICATIONS' TESTABILITY TO THE NEXT LEVEL

Nearly all successful teams understand at least the fundamentals of testable system code: good automated unit and integration/API tests have finally reached the point of being an accepted—or even required—part of a solid delivery process. What's still missing with many teams is an understanding of how small changes to systems and user interfaces can dramatically improve test automation at the functional level.

Because of its focus on business value, functional system testing is one of the most critical parts of a project's overall quality and value delivery. Wrapping this critical, high-value testing in sensible automation helps project teams ensure they're keeping their overall.

MAKING THE CASE FOR TESTABILITY

Getting testing involved earlier isn't just about reducing the cost of testing later in the project. It's also about making testing easier from the start. Far too many teams miss the fact that making testing easier can be dramatically impacted by making the system itself easier to test. High-performing teams look at testability as one of their fundamental design considerations.

DEVELOPER-LEVEL TESTABILITY

Good software design focuses on simplicity and maintainability. Testability below the UI layer dovetails with good craftsmanship principles and practices like low complexity, dependency injection, and low coupling. Test-driven practices drive out this simplicity by their very nature; however, good design practices keep the system more testable regardless of when test automation is completed.

EASING FUNCTIONAL AUTOMATION WOES

Functional test automation is by its very nature far more complex and brittle than automation at the unit or integration level. Not only are there complexities of logic, dependencies, etc., the very technologies and toolsets for user interfaces drop in additional challenges for automation. These challenges often leave teams with brittle, low-value functional automation tests.

Thankfully, a few simple approaches can greatly improve testability, making it far easier to have high-value, low-maintenance automation suites that check the system's functionality. The approaches listed here start out with simple techniques for interacting with the UI, escalate to simplifying asynchronous situations, and finish off with complex configuration of the system.

STARTING EASY: LOCATORS

Every automated functional test tool, regardless of platform, relies on finding things to interact with on the user interface: buttons, fields, text, controls, etc. The testing tool has to locate those objects to click them, inject text, compare text, and numerous other actions. Exact mechanics for this location process vary greatly across platforms; however, the concepts are the same regardless.

Various properties, attributes, or metadata can be used for

this location process—appropriately, these are referred to as locators.

GOOD IDS

HTML gives one of the simplest mechanics for good locators: the ID attribute. Generally speaking, IDs are fast for tools to locate, they're unique on a page (if the page is valid HTML!), and they're also very easy for developers to customize.

As an example, consider a table used to display contacts from a Customer Relationship Management (CRM) system. Adding an ID to an element on the page is normally a very simple task for a developer working on the page's code. The result may not seem earth-shattering, but it makes all the difference in the world for test automation:

With this simple addition in place, one statement will enable a WebDriver script to quickly locate the table:

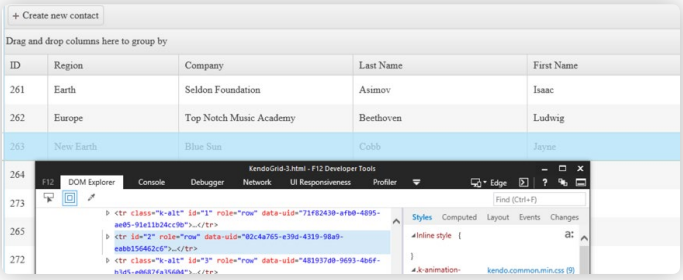
```
WebElement table = browser.FindElement(By.Id("contacts"));
```

Again, the mechanics for implementing the HTML attribute vary by platform. Controls on the page may offer the ability to easily add attributes to their rendered output. Various frameworks offer great control over their output as well.

DEALING WITH DYNAMIC IDS

While HTML IDs are a terrific locator, you can't use them thoughtlessly. Sometimes they're not the best locator, especially if they're dynamically created.

Going back to our example of a contact list, imagine a grid/table control that dynamically creates IDs for each row in the grid.

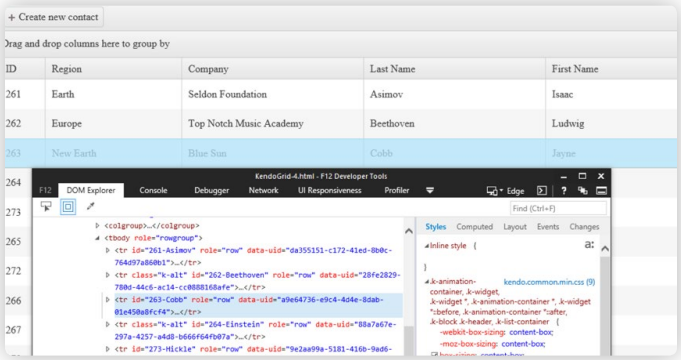


Suppose you're writing a test to confirm data for the contact Jayne Cobb. A WebDriver statement to locate that row, given the above HTML, could look like this:

```
WebElement JayneRow = browser.FindElement(By.Id("2"));
```

That would work well for the first run, but what would happen if the source data changed? The row you're looking for would likely appear elsewhere in the grid—your script would break and your test would fail.

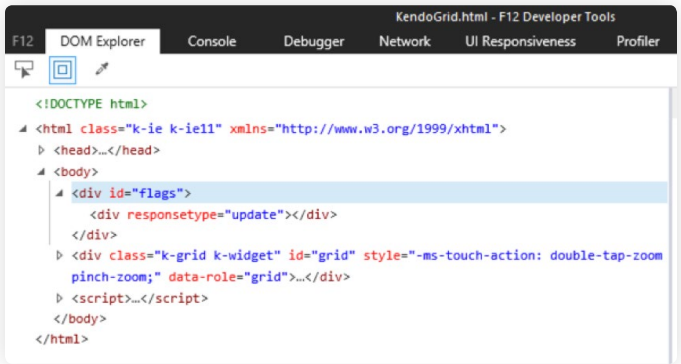
A short conversation between testers and developers can solve this. The underlying system can be altered to render an ID that contains information pertinent and specific to the data you're looking to locate. In the case below, the ID attribute is a composite of the contact's ID and last name.



This dramatically changes the functional test script, enabling us to locate by the desired test row's data:

```
WebElement JayneRow = browser.FindElement(By.
CssSelector("[id$=Cobb"));
var contactSource = new kendo.data.DataSource({
            requestEnd: function (e) {
                var node = document.
getElementById('flags');
                while (node.firstChild) {
                    node.removeChild(node.firstChild);
                }
                var type = e.type;
               $('#flags').append('<div responseType=\'' +
type + '\'/>');
            }
```

This snippet causes an empty DIV element to appear on the page with the specific action (post, update, delete, etc.) as an attribute within the element:



Now it's easy to use your functional testing tool to wait for the actions to complete. Pseudo code for a test using this approach might look like:

```
//Navigate to screen
//Locate and edit an existing contact
WebDriverWait wait.Until(ExpectedConditions.
ElementExists(By.CssSelector("#flags >
div[responseType='update']")));
```

TACKLING THE HARD THINGS: SYSTEM CONFIGURATION AND SERVICES

How do you solve hard problems in testing? Apply the same smart, thoughtful engineering approaches you do for solving hard software problems. Too often teams forget they can make major system changes to ease testing burdens.

Creating software switches to deactivate areas of the system that are hard to test is one of the best ways to improve testability. Creating stub or mock services is another.

Approaches like this take time and hard work, but they make total sense for high-value, high-risk areas of the system.

FEATURE SWITCHES

Experienced test automation folks regularly get asked "How do you write test scripts for CAPTCHA?" A common response is "Don't."

By its very nature, CAPTCHA is meant to prevent automation. This is why it's used for guarding things like user registration or account creation. Trying to write test automation scripts to deal with it is crazy; however, the functional slice of creating a new account absolutely has to be covered. How to deal with this gap?

Easy: Cheat.

Teams can work to create a feature switch within the system itself that can totally disable CAPTCHA and enable the registration/creation process to proceed without having CAPTCHA as part of the flow. As mentioned earlier, this can be hard work, and it brings its own complexity to the situation. However, if the team has decided this particular flow merits the investment, then it's worth it to have a configuration file, API service endpoint, or some other means that turns off CAPTCHA. Obviously similar code will be needed to return the system to its normal state...

This same approach can be used for similar concepts:

- **Third-Party Controls**. Don't write tests for those controls—the vendor should have tested them. If those controls are hard to interact with, then use a feature switch to swap out for an easier route. TinyMCE has long had a history of being hard to automate. Swap it out for a simple text box.
- **Email**. Need to check validity of outgoing mails? Don't ever, EVER use a functional test to log on to Gmail or some other service. Instead, configure an SMTP sink such as NDumbster or similar tool.

STUB OUT ENTIRE SERVICES

What is your functional test flow really dependent on? How many external services do you really need if you're focused on the high-value business related part of your test?

For example, consider a check to ensure your order system enables you to search for a part, add that part to your shopping cart, and proceed to check out. There are a number of ways to consider testing this, but let's take these considerations to work with:

- User must have an account created, and be logged on as that user
- User searches for a specific part known to be in the system
- User adds that resulting part to their shopping cart
- User proceeds to checkout. Part remains in shopping cart.

This test is not about checking out, it's confirming you can search for a part, add it to the cart, and head to checkout. This test is also not about validating searches, it's about adding a result from the search.

There are a number of things we don't need to concern ourselves with as part of this test:

- **Logon**. Tested elsewhere and is not central to the functionality of the search results to cart flow.
- **Part Search**. Again, should be tested elsewhere and isn't central to the flow.

Spending the time to stub or mock out these services would be a perfect use of a team's time. The overall flow is high-value, so it needs wrapping with an automated check. Stubbing the services would let the team write the automated tests to work in any environment the mock could be established in—another great advantage, as external systems often aren't available in lower environments for development and testing.

CLOSING

Functional testing is one of the most critical aspects of software development. It focuses on the end user and business needs. Taking time to make systems more testable at the functional level reaps high rewards for teams disciplined and supported enough to do the work.

Jim Holmes is an Executive Consultant at Pillar Technology, where he works with organizations to improve their software delivery process. He's also the owner/principal of Guidepost Systems, which lets him engage directly with struggling organizations. He has been in various corners of the IT world since joining the US Air Force in 1982. He's spent time in LAN/WAN and server management roles in addition to many years delivering great systems. When not at work you might find Jim in the kitchen with a glass of wine, playing Xbox, hiking with his family, or practicing guitar.

Nine Critical Considerations for Testing Responsive Websites Using Selenium

BY **CARLO CADET** - DIRECTOR PRODUCT MARKETING, **PERFECTO MOBILE**

Responsive website design is becoming a method of choice for many organizations. Among the primary motivations for embracing responsive design are:

- Consistent user experience across all platforms
- Improve marketing results by being mobile friendly
- Lower maintenance cost

One code base across so many platforms and form factors raises the bar on quality and therefore the testing strategy.

In the below checklist, you can find the most critical testing consideration for a RWD that will ensure good UX, and sufficient test automation coverage. All of the below considerations can be automated using Selenium framework or cloud-based tools.

1. VISUAL VALIDATIONS

Does your website look right across all platforms like desktop browsers, smartphones, tablets, and IoT-supported devices?

Does the site look okay in various orientations, like portrait and landscape, as well as in various languages?

2. ENVIRONMENT CONDITIONS

Validate performance across all expected conditions. Factor in variables such as incoming events, background apps, location services, and changing network conditions (Data, Wi-Fi, Airplane mode, etc.)

3. NAVIGATION

Validate CSS breakpoints across different form factors and orientations. When the site is launched across these displays, the navigation and the content of what is being displayed to the user changes (above the fold and beyond the fold content, hamburger menus, etc.).

4. PLATFORM COVERAGE

Analyze web traffic to determine coverage strategy, identifying the mandatory platforms and OS versions to be tested throughout the SDLC.

5. ACCESSIBILITY COMPLIANCE

Assess compliance with accessibility requirements across the market you serve. Using tools like WAVE that can be integrated into your Selenium scripts or be a stand-alone tool for your test engineers, is a good choice (out of few others) to adopt.

6. PERFORMANCE

Performance optimization is key, especially considering Google's recent prioritization of mobile-friendly sites. Performance testing for source and data loading, caching controls, and functional scenarios are proving to be effective tools for achieving critical performance gains.

7. LOCALIZATION

Validate location services scenarios. Design scenarios for both location specific data and the "traveling user".

8. SECURITY

Personalization strategies are driving an increasing quantity of private user data process by sites. Add data privacy scenarios to test suites. This includes authentication rules and types, cleaning of private data upon session termination.

9. DON'T FORGET QUALITY ANALYSIS & VISIBILITY

Testing a RWD site across multiple platforms, means, dealing with large amount of test data. Having a quality dashboard after each test automation execution that is tag-driven for easy filtering enables data-driven and risk-based decisions.

Automated Testing Is Essential To DevOps And Continuous Delivery

Much has been written and said about DevOps, less so about the role of automated testing in a DevOps environment. In order to reap the full benefits of a healthy DevOps practice, organizations must integrate automated software testing into their Continuous Delivery pipelines. It is the only way to ensure that releases occur at both a high frequency, and with a high level of quality.

In order to integrate continuous testing effectively into a DevOps toolchain, look for the following essential features when evaluating an automated testing platform:

- **Support for a variety of languages, tools, and frameworks**. The programming languages and development tools that your DevOps teams use today are likely to change in the future. Look for a testing solution that can support a broad array of languages, tools, and frameworks.
- **Cloud testing**. On-demand cloud-based testing is the most cost-efficient option because it obviates the need to setup and maintain an on-premises test grid that is underutilized most of the time. It also reduces the resource drain associated with identifying and resolving false positives, or failures due to problems in the test infrastructure.
- **The ability to scale rapidly**. Your testing platform should be able to perform tests as quickly as needed, and be able to do so across all required platforms, browsers, and devices. It should also be highly scalable to support as many parallel tests at one time as you require.
- **Highly automated**. DevOps teams achieve their speed and agility in part by automating as much of the software delivery process as possible. Your testing solution should work seamlessly with other components of your toolchain, most notably your CI and collaboration tools.
- **Security**. In a DevOps environment, all members of the team have an important role to play in keeping applications secure. Testing platforms, therefore, need enterprise-grade security features.

A software testing platform that includes these qualities will empower your organization to derive full value from its migration to a DevOps-based workflow by maximizing the agility, scalability and continuity of your software delivery pipeline.

WRITTEN BY LUBOS PAROBEK
VP OF PRODUCTS, **SAUCE LABS**

PARTNER SPOTLIGHT

Automated Testing Platform

SAUCELABS

Sauce Labs accelerates the software development process by providing the world's largest automated testing cloud for mobile and web applications.

CATEGORY
Automated Testing Infrastructure

NEW RELEASES
Daily

OPEN SOURCE
Yes

CASE STUDY

GoDaddy is a growing web hosting and domain registration company that serves more than 16 million customers across the globe. Based in Scottsdale, Arizona, the firm is the world's largest domain name registrar, with more than 70 million domain names under management. To ensure that websites created by its customers stay up and running around the clock, GoDaddy is committed to delivering the highest-quality software to support its backend systems. GoDaddy sought to increase cross-browser coverage and reduce the time and money it took to run its internal software testing environment. The company met its needs by selecting Sauce Labs, and now utilizes dozens of Sauce Labs virtual machines in parallel to run thousands of tests every day. The tests are automatically run as part of a CI/CD pipeline from GoDaddy's Jenkins CI server. The company now continuously tests all popular browser and OS variations for its products and services, and can also easily test more browser variations as needed, thanks to the on-demand nature of Sauce Labs.

STRENGTHS

- Enterprise-grade cloud-based test infrastructure provides instant access to more than 800+ desktop browser/OS combinations, ~200 mobile emulators and simulators, and 1,000+ real devices
- Highly scalable, on-demand platform reduces testing time from hours to minutes when tests are run in parallel
- Optimized for CI/CD workflows, testing frameworks, tools, and services
- Single platform for all your web and mobile app testing needs

NOTABLE CUSTOMERS

- Salesforce
- VISA
- Starbucks
- GoDaddy
- WalMart Labs
- Slack
- Yahoo!

WEBSITE saucelabs.com | **TWITTER** @saucelabs | **BLOG** saucelabs.com/blog

A brief history of web and mobile app testing.

Find out how Sauce Labs can accelerate your testing to the speed of awesome.

For a demo, please visit saucelabs.com/demo
Email sales@saucelabs.com or call (855) 677-0011 to learn more.

SAUCELABS
Testing at the speed of awesome.