SAUCELABS

WHITE PAPER

Building Testable Apps

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.

TABLE OF CONTENTS

SAUCELABS

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:

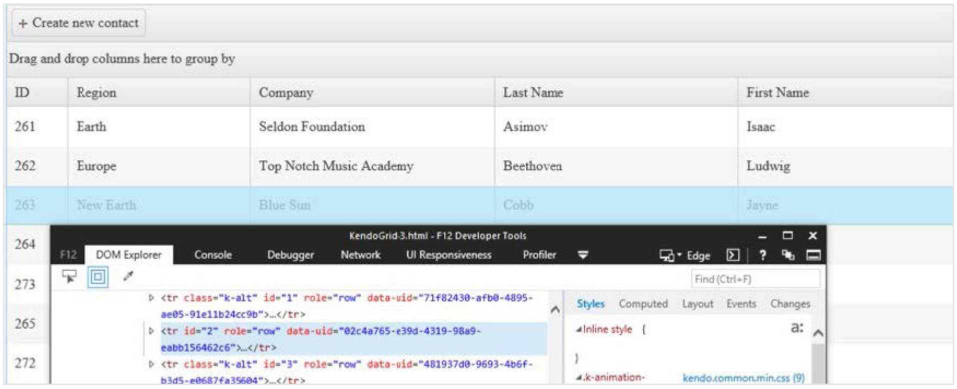
```
WebElement table = browser.FindElement(By.Id(“contacts”));
```

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.



ID	Region	Company	Last Name	First Name
261	Earth	Seldon Foundation	Asimov	Isaac
262	Europe	Top Notch Music Academy	Beethoven	Ludwig
263	New Earth	Blue Sun	Cobb	Jayne
264				
273				
265				
272				

```
<tr class="k-alt" id="1" role="row" data-uid="71f82430-afb0-4895-ae05-91e11b24cc9b">…</tr>
<tr id="2" role="row" data-uid="02c4a765-e39d-4319-98a9-eabb156462c6">…</tr>
<tr class="k-alt" id="3" role="row" data-uid="481937d0-9693-4b6f-b3d5-e0687fa35604">…</tr>
```

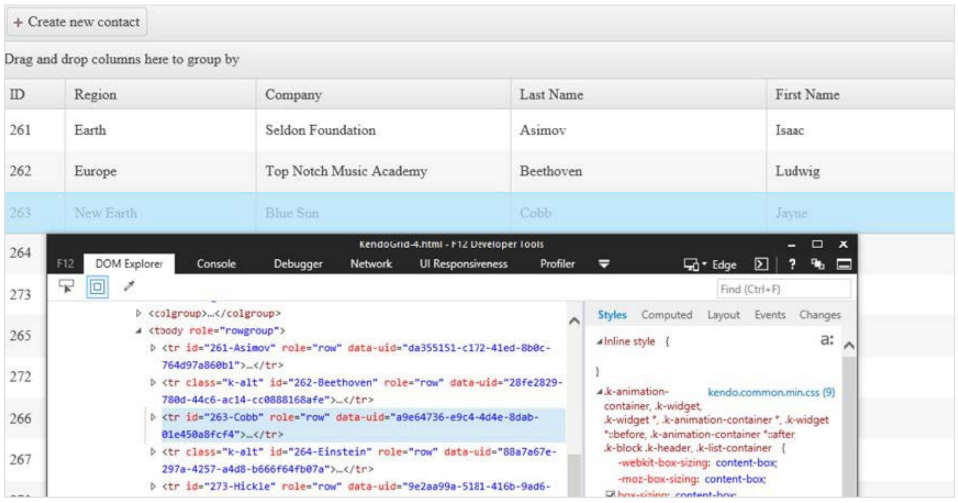
```
Inline style {
}
.k-animation-        kendo.common.min.css (9)
```

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.



ID	Region	Company	Last Name	First Name
261	Earth	Seldon Foundation	Asimov	Isaac
262	Europe	Top Notch Music Academy	Beethoven	Ludwig
263	New Earth	Blue Sun	Cobb	Jayne
264				
273				
265				
272				
266				
267				

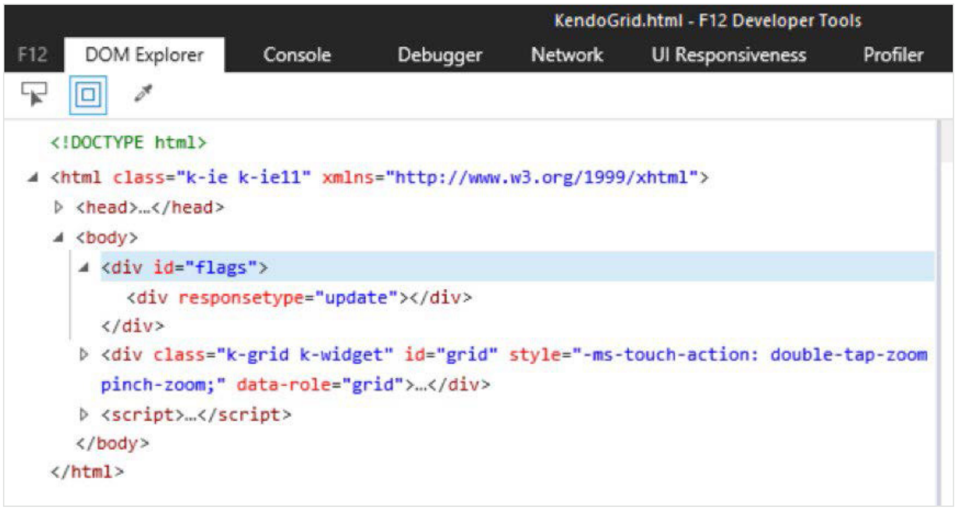
```
▷ <colgroup>…</colgroup>
◢ <tbody role="rowgroup">
  ▷ <tr id="261-Asimov" role="row" data-uid="da355151-c172-41ed-8b0c-
    764d97a860b1">…</tr>
  ▷ <tr class="k-alt" id="262-Beethoven" role="row" data-uid="28fe2829-
    780d-44c6-ac14-cc0888168afe">…</tr>
  ▷ <tr id="263-Cobb" role="row" data-uid="a9e64736-e9c4-4d4e-8dab-
    01e450a8fcf4">…</tr>
  ▷ <tr class="k-alt" id="264-Einstein" role="row" data-uid="88a7a67e-
    297a-4257-a4d8-b666f64fb07a">…</tr>
  ▷ <tr id="273-Hickle" role="row" data-uid="9e2aa99a-5181-416b-9ad6-
```

```
◢Inline style  {
}
◢.k-animation-            kendo.common.min.css (9)
 container, .k-widget,
 .k-widget *, .k-animation-container *, .k-widget
 *::before, .k-animation-container *::after
 .k-block .k-header, .k-list-container  {
    -webkit-box-sizing: content-box;
    -moz-box-sizing: content-box;
```

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:



```
<!DOCTYPE html>
<html class="k-ie k-ie11" xmlns="http://www.w3.org/1999/xhtml">
  <head>…</head>
  <body>
    <div id="flags">
      <div responsetype="update"></div>
    </div>
    <div class="k-grid k-widget" id="grid" style="-ms-touch-action: double-tap-zoom
    pinch-zoom;" data-role="grid">…</div>
    <script>…</script>
  </body>
</html>
```

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 & 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.

ABOUT SAUCE LABS

Sauce Labs is the leading provider of continuous testing solutions that deliver digital confidence. The Sauce Labs Continuous Testing Cloud delivers a 360-degree view of a customer's application experience, ensuring that web and mobile applications look, function, and perform exactly as they should on every browser, OS, and device, every single time. Sauce Labs is a privately held company funded by Toba Capital, Salesforce Ventures, Centerview Capital Technology, IVP, Adams Street Partners and Riverwood Capital. For more information, please visit saucelabs.com.

saucelabs.com/signup/trial

FREE TRIAL