Why Test the User Journey?

I recently worked with a team of software developers who were quite proud of the level of testing they had in place on their project. They had a robust set of unit tests, and extensive code coverage reports to match. They were thoughtful about separating out their unit tests (performed in isolation with the tactical use of mocks and stubs) from their more involved, longer running integration tests.

The project was a traditional web site, complete with a login screen, a catalog of items, and the ubiquitous shopping cart. When I asked them how they tested their user interface, their responses ranged from sheepish to indignant.

```
"Well, we have a couple of Selenium tests, but they're pretty brittle. They always
seem to be broken, so we rarely run them."

"We decided that since we test the back end so extensively with unit and integration
tests, the UI tests were redundant and unnecessary."

"Since we use [a popular web framework], we assume that the framework is well-tested.
What bugs could we possibly find that they haven't found already?"

"I click through every screen of the website right before we give our showcase for the
client, just to be sure that nothing blows up in our face."
```

Brittleness. Redundancy. A "testing is for finding bugs"-only mindset. All of these seem to conspire against user interface testing, even among developers who consider themselves test-first advocates. Even the phrase "user interface testing" unwittingly seems to place the emphasis on "interface" rather than "user" in the mind of many developers.

Some of this might be explained by the code-first myopia that many software developers suffer from. Inexplicably, "users" are often reduced to a distant, abstract notion rather than the whole point of the software writing exercise in the first place. The tangible nature of the source code in front of the developer — in many cases, the source code that they just wrote themselves — is far more "real" than any notion of a user trying to use the code that they just composed.

All of which is why I've become a big fan of user journey testing — from the writing of the tests themselves to the words we use to describe the tests.

Because, as you'll see in just a moment, there were a number of bugs lurking in their user interface — bugs that were trivial and obvious to uncover when they simply tested the application by using it as their users would. After months of successfully passing unit tests, the team was surprised by how many undiscovered bugs they still had when they applied a new avenue of testing.

What is a User Journey Test?

A *user journey* is simply the specific steps a user has to perform to accomplish something on your website. If a user needs to buy ingredients for a spaghetti dinner from a grocery store website, the steps might be:

- Visit the website
- Add items (like tomatoes, garlic, Parmesan cheese, and noodles) to the shopping cart
- Click on the Checkout button

So then, a *user journey test* is a programmatic, automated way to perform the same steps that the user would in the same circumstances. Stated another way, user journey tests assure the correctness of your website by simply walking the same paths that your users already do.

The team of developers were already savvy unit testers, so they were excited and intrigued to tackle a new kind of testing that played to their strengths while gently broadening their perspective.

How are User Journey Tests Different than Unit Tests?

For software developers who embrace unit testing, user journey testing should feel largely familiar, while differing in some subtle, interesting ways.

The principal considerations for unit tests are *inputs*, *outputs*, and *assertions*. If you're testing a `CalculateSalesTax` function, the inputs might be a `SalesAmount` and a `Location`. The output is the calculated amount of sales tax. And the assertions should ensure that the expected results match the actual results.

If the `CalculateSalesTax` function accepts `City` and `State` arguments as `Strings`, you can pass in a variety of values and assert that the results match your expectations. If the function expects a `LoggedInUser` to provide these details in a `UserAddress` class, you might have to mock out these objects so that the unit test can pass. Similarly, if the `CalculateSalesTax` function uses a GPS call to determine where the user is physically located at the time of checkout, you'll need to invest some additional time and energy in figuring out how to mock out the GPS and supply arbitrary `Latitude` /`Longitude` points that simulate different users in different locations.

So, you can see how quickly unit tests can become an exercise in simulation and abstractions away from reality. You don't need an actual user in Denver, Colorado to put actual grocery items in their shopping cart just to determine if your `CalculateSalesTax` function is correct.

In comparison, user journey tests may simulate a user's behavior on your website, but that's where the artificialness ends. We expect an actual website to be up and running, and the steps of the user journey tests should be indistinguishable (from the website's perspective) from those of an actual living, breathing user.

The principal consideration for a user journey test is the *steps* a user must take to complete a task. These tests are more interested in the *assurance* that each step of the user journey can be completed successfully, in sequence, rather than individual, explicit *assertions* along the way. An automated collection of passing user journey tests is your measure of success and a measurable indication of the overall health of your website. (No more hurriedly clicking through your website by hand in a mad rush the last moment before you showcase the software in a customer demo.)

A good user journey testing library will still allow you to simulate various aspects of the website interaction — simulating a smartphone on a slow cellular network connection from a fictitious location — but the actual steps performed on the website are more real than not. We can simulate the conditions around the user journey, but the steps of the user journey test itself are the real test of the website's actual functionality.

But perhaps the biggest difference between a unit test and a user journey test is the perspective of the test writer. Unit tests are written from the perspective of a software developer wanting to use a specific function. User journey tests are written from the perspective of a user wanting to use your website. This shift in perspective — putting yourself in the user role, describing the steps the user will take to accomplish a task — brings empathy into the equation. You, as the test writer, are quite literally walking the same path through your website as your users do. And this empathy isn't something that you should strive to mock away.

User Journey Testing in Action

The conscientious team of software developers that you met at the beginning of this story decided to try out a new browser automation library — one that is specifically built for user journey testing — called Taiko. The easy installation — `npm install -g taiko` — and the light, expressive JavaScript-based DSL made it easy to learn and explore. The Taiko REPL — an interactive command-line environment that contains documentation and encourages you to explore Taiko in a live setting — helped the team experiment with Taiko and save their experiments as actual tests that can be run outside of the REPL environment. Taiko dramatically reduces the developer cost of writing user journey tests, and markedly reduces the test brittleness as well.

Here's what a simple "Buy the ingredients for a spaghetti dinner" user journey test looks like in Taiko:

A user journey test written in Taiko

```
openBrowser()
goto('https://thirstyhead.com/groceryworks/')
click('Produce')
click('Vine-Ripened Tomatoes')
click('Purchase')
```

When the developers tried to log into their website using Taiko, they discovered that they couldn't select the `Username` and `Password` fields because they hadn't used the semantically appropriate `<label>` elements with their `<input>` fields. They discovered an accessibility bug with the first test they wrote!

They went on to discover that they couldn't programmatically select items from the catalog because the items all had the same HTML ID due to a copy/paste error. This functionality had already passed numerous unit tests, but none of the unit tests caught this particular bug because they were testing the shopping cart in isolation instead of testing it through the user interface as their users would use it.

A page that they tried to visit using Taiko errored out because they had been testing the pages in isolation and in a simulated environment. When they visited that same page as the user would — as one step in a sequence of many along the path of a specific user journey — they uncovered a tricky state-related bug that they never would’ve found testing the page in isolation.

At the end of my engagement with the project, one of the team members said:

```
"We always knew that testing was important. What we didn't realize is that you
unconsciously test what's important to you.

As software developers, we were deeply invested in the quality of our code by writing
unit tests. But what we failed to do was dedicate that same level of care to the user
experience.

Now that we know more about user journey testing -- and how easy it is! -- we know
that our website's user experience will be as rich and bug-free as the developer
experience."
```