Automated Visual Regression Testing with BackstopJS

Angela Riggs
riggs.ang@gmail.com

Abstract

During the software development life cycle, there are many types of testing that are done to ensure the quality of work. While back-end testing is fairly standardized, testing your front-end work is more difficult - verifying implementation across a variety of devices, or making sure new style changes don't introduce style regressions across the site.

But how do you test front-end regressions? Manual testing is an option for smaller sites, but much less efficient as websites become bigger and more complex. After failing to scale exploratory testing and trying other tools that weren't the right fit, we finally landed on BackstopJS, an open-source npm package that allowed us to create a system for automated visual regression testing.

In implementing this tool, we had to overcome some challenges as we began to meet our initial goal of verifying front-end quality. Once BackstopJS was in place, new challenges came up that needed to be solved as well:

- Refining a broad testing tool into a more specific approach
- Ramping up a testing tool into the software development workflow
- Lowering the barrier of use to encourage other teams and developers to adopt it

As these challenges were solved, the result was a robust process for automated visual regression testing that is easily customized and iterated, and offers the ability to catch and fix mistakes before they become our clients' problem.

Biography

As a QA Engineer, my role is at the intersection of Testing, DevOps, Architect, and Scrum Master. I approach my work with determination and a positive energy, holding myself and my team to a high standard. I have an enthusiasm for learning and advancing, and appreciate that my field calls on my curiosity and attention to detail. I understand the importance of acknowledging and balancing competing needs to ensure individual, team, and project success.

I believe in people over process, but also enjoy creating useful foundational processes that promote clear understandings around development workflow. I take pride in both my technical and interpersonal skills, and work hard to be a leader who offers empathy, support, and thoughtful problem-solving to my team and my company.

Outside of work, I enjoy meandering through nature, or through the aisles of Powell's. I also have an enthusiasm for karaoke, and long debates about what can truly be categorized as a sandwich.

1 Catching Visual Regressions

Automated visual regression testing started as a solution to a specific problem for our engineering team. One of our websites was experiencing content loss and style regressions on deploys to staging and production. Unfortunately, we became aware of these challenges because the client would find them and tell us. We needed a better way to catch these regressions so we could rollback, fix the issue, and redeploy in a timely manner.

Our initial solution was to deploy during low-traffic periods, and to follow up the deploys immediately with exploratory testing. This was inefficient and mostly ineffective, as the website had so many pages and sub-pages that it just wasn't reasonable to test this way and expect to catch regressions.

We needed a way to easily check large portions of the website for regressions, and we needed something pretty quick. Some of the initial options required too much configuration before tests could be run, or necessitated writing out all of the tests for each page of the website manually. The time and effort to do that meant that it wasn't much of an improvement from the exploratory testing.

After researching a few options, BackstopJS stood out as the best solution for our problem. It was quick to set up, configure, and start running. It also offered a visual report that compared before and after screenshots side-by-side, which made it easy to see where any differences were.

Most of the setup involved creating a list of URLs for BackstopJS to iterate through for its screenshots. Because the regressions were inconsistent, 60-70 URLs were included in the configuration file. For each URL, a screenshot of the header, footer, and main content section were captured. Before a deploy, the reference command would be run, which captured the present-state screenshots for staging and production environments. The developer would deploy to staging, and the post-deploy BackstopJS command would be run to capture screenshots of the updated site and compare for style regressions. Once it was verified that the deploy went well - or the regressions had been fixed - the developer would deploy to production, and BackstopJS would be run on production to check for regressions.

With this tool in place, style regressions could be caught and fixed before they became a problem for our client. But there were a few more challenges ahead that needed to be solved in order to get the best use out of automated visual regression testing.

2 Narrowing the Scope

The initial problem had been solved, but the current setup required a lot of overhead to run the tests. Because so many URLs were being tested, there were around 400 screenshots per run. This broad sweep approach worked for catching regressions, but there were a few drawbacks. The screenshots included a lot of false negatives, which was tedious to parse through and verify whether a regression had occurred. The tests also took quite a long time to run, and would actually time out and fail about halfway through unless npm's "no timeout" argument was included in the command.

From start to finish - running the references, deploying, running the tests, and reviewing - the process took about 45 minutes to an hour. This hardly made it more worthwhile than manually testing for regressions. A more effective workflow was needed to reduce the time and effort involved, and to take advantage of this automated testing tool.

Looking through the website, there were a handful of templates repeated across the site. This was helpful - the number of URLs in the configuration could be reduced, and still have all of the templates represented in the testing. The height value in the viewport was adjusted to 3000px, and each URL was given a single CSS selector to trigger the screenshot. The height change meant that the captured screenshot was tall enough to cover the whole page, instead of getting coverage via multiple CSS selectors.

These configuration changes drastically reduced the overall running time, removed the risk of timing out, and reduced the number of screenshots per run from 500 to under 100. This iteration made automated visual regression testing much more useful for wider use, and it was set up on all of our projects. Instead of just being a reaction to bad deploys on a single project, it was implemented as a smoke test for all deploys to staging and production.

3 Lowering the Barrier for Use

With the overall configuration improved, there was one challenge left to tackle. Although running the tests had become more efficient, maintaining and updating the configuration was still problematic. Each environment required its own configuration file, which meant four separate files to update and keep in sync. The extra effort here made it hard to iterate the tests as greenfield sites were built, which prevented widespread adoption of BackstopJS during feature development.

To solve this, the configuration setup was changed to use a single JavaScript file instead of the default JSON format. This format change allowed for arguments to be defined and passed in to the BackstopJS commands. It also allowed for an environment argument to be passed in for the URL to test, instead of having to reference separate configuration files. Alongside this change, there was a file with an array of URL paths to iterate over, which was also included as an argument in the BackstopJS commands.

These changes made the workflow much more convenient to use, but had an unexpected side effect that still presented a blocker for developers. Because the setup was very customized, it required a number of arguments to be passed in for executing the tests. This resulted in lengthy and cumbersome commands. To improve this experience, a Makefile was created for running BackstopJS. This took the commands from something like `backstop reference --configPath=backstop.js --pathFile=paths --env=prod` to `make prod-reference`. This was much more intuitive, and much easier to remember.

With these final improvements to the configuration and setup, the barrier of use was lowered for developers, which meant they were willing to add it to their local development workflow. Now, style regressions had the potential to be caught long before any deploys to production!

4 Successfully Transitioning to Automated Testing

With BackstopJS in place, we were able to reliably and efficiently test for style regressions. However, we learned that introducing the tool wasn't a magic solution. Although it was the right tool for the job, there was still work to be done to make it an effective part of the general workflow.

The first introduction of BackstopJS did offer automated testing as an alternative to inefficient manual testing, but the way it was used didn't really end up saving time or effort. By minimizing how much BackstopJS actually needed to test, we were able to reap the benefits of having automated testing in place of the former manual testing. And because we took the time up front to adapt the configuration and keep it simple to run, automated visual regression testing was adopted for use throughout the development workflow, from feature work to deploys.

The challenges we solved and the lessons we learned while adopting BackstopJS can be applied to many processes of researching and adopting any new testing tool. Replacing manual testing with automated testing is not a one-size-fits-all solution. Not all tools are the right solution for your problem, and even the right solution may not work right out of the box. Transitioning to automated testing requires iteration and feedback, and buy-in from the people who will be using or benefiting from it. Find the right tool, figure out how to make it work for your needs, and make it easy to people to use.