Visual Regression Testing

A picture is worth a thousand sheep.

Graham Murdoch - 16 December 2019

Don't count sheep. Test regressions.

Today we are going to talk about Visual Regression Testing. We will consider the benefits, the availability of existing solutions, and, for the more adventurous folks out there, how to set up a visual snapshot testing pipeline for your own project.

Regressions

The purpose of regression testing is to verify that changes to source code do not have unforeseen consequences. This is a fairly common occurrence in software development: you change one thing and unintentionally break something else in the process.

When we talk about visual regression testing, we are concerned with preventing unintentional changes to our application's visuals. If we change the styles of a `Button` component, how can we be sure that we haven't messed up the styles of our `ButtonGroup`? The most common strategy for this sort of assertion is known as a **visual snapshot test**.

Snapshots

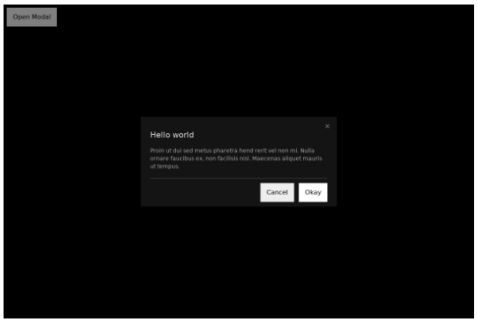
A snapshot test runs a process and compares the output of that process to a previous test run's output (the baseline). If the outputs match, the test passes. If the outputs do not match, the test fails and you either need to update your baseline snapshot or fix something in your source code.

Because snapshot tests are assertions of equality, snapshots need to be serializable, comparable, and deterministic. The format of the snapshot is flexible, it can be inlined into the test as a string or stored as a file— it can be a DOM tree, a JSON blob or even an image.

Visual

This is where the *visual* in "visual snapshot testing" comes from. We use images as our snapshots and compare those images across test runs using an **image comparison library**. When an image differs from our baseline image, we either update the baseline or fix the issue in source.

In Base Web, we render each of our components in a variety of states- we capture an image of each state and save all of these as baseline snapshots. Whenever we make a change to source we can run our visual snapshot tests to ensure that all of our components' visual states look as expected.



An example snapshot of a modal.

Value Added

As we will see in a bit, introducing a suite of visual snapshots into your project comes with some maintenance burden. Taking deterministic screenshots requires running tests in a consistent environment across test runs. Test runs can add significant time to your CI pipeline. Additionally, keeping hundreds or even thousands of images under source control can dramatically increase the storage footprint of your repository.

If you are familiar with snapshot testing you may also wonder why we need to bother with images at all. Why not simply capture the state of the React tree as with traditional snapshot tests? Are visual snapshots worth all the overhead?

For a UI component library such as Base Web the tradeoff is certainly worth making. Here are some of the reasons why we find them essential to our testing strategy:

Catch visual regressions

This is the first and most obvious reason to use visual snapshot tests. When we make a change to source code we want to be sure we have not broken any styles. For a UI library such as Base Web, styling is immensely important — it is one of our

For a UI library such as Base Web, styling is immensely important— it is one of our main responsibilities. Having coverage over our component visuals means better quality components and better sleep at night for us maintainers.

An example of two snapshots catching a regression. Swipe the divider back and forth to see the difference.

Test what users see

Traditional snapshot tests will certainly warn you when something in your code has changed, but the change detected is going to be your React implementation, not the result of that implementation which is what you care about.

Obvious updates

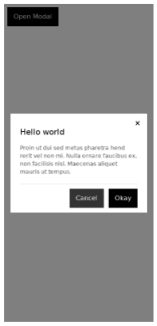
It is a lot easier to decide to update a snapshot when you can see what the before and after look like. Many traditional snapshots have been erroneously updated because a developer didn't understand what had changed. Anyone who has worked with snapshots in the past has seen a diff with hundreds or thousands of changed lines in snapshots. An image tells the full story.

Centralized coverage

Most testing strategies focus on coverage for application state and behavior. For instance, a typical UI unit test might make assertions about a rendering function's output or an end-to-end test might verify that certain elements appear on the page. But we care as much, if not more, about how our components *look*. If a color is off or something is not aligned, we want to catch that before dependent apps can consume it. By focusing on thorough visual coverage on the Base Web library, downstream projects can focus on testing what matters to them.

Verifying appearances across browsers & device sizes

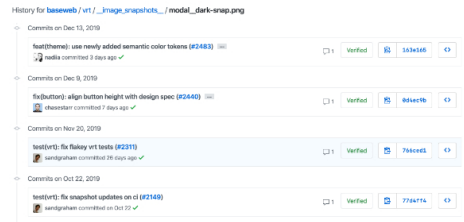
One of the potential benefits of setting up visual tests is to be able to run them on different browsers and devices. Depending on your support requirements, catching an environment-specific regression before it ships is a huge win.



An example of a mobile snapshot.

A component changelog

One of the benefits of having visual snapshots under version control is that you get a component changelog for free. By looking at the history of a snapshot you can track every visual change that's been made to the component. If your project has very thorough visual tests you can even track props and functionality through snapshot histories. The snapshots can also function as a contract to dependent projects— as a supplement to documentation, snapshots can give someone an exact picture of the state of every component.



Viewing the git history of a snapshot shows me the component's visual history.

There may be other reasons to set up visual snapshot tests, but so far these have been enough to warrant the investment for our team.

Solutions

There are a variety of products and libraries out there which will help you with visual test coverage in your codebase. Although we ended up piecing together a custom system using open source tools, it seems we should briefly touch on what other options you have before digging in yourself.

Paid

A whole host of paid services exist to serve your visual testing needs— too many to detail here. With all of these paid solutions, you do not have to worry about storing or comparing images yourself. The service runs a suite of comparison tests and then

flags failures for you to review in their UI. Generally, these runs are triggered from and report their status to, your CI pipeline. As might be expected from a premium service, these all offer entire workflows for managing your project's visual tests.

If you are maintaining even a small component library there is a decent chance that you are already using **Storybook** in your project to develop components in isolation. Most services provide libraries or integrations for using your already existing Storybook scenarios for visual snapshots. This is extremely convenient as you receive free test coverage every time you create a new Storybook scenario.

Also of interest, most services allow you to capture images of scenarios in different environments such as IE11 or iOS Safari. This is not a trivial (or free) thing to set up for your project- so if cross-browser coverage is important for you, choosing a paid solution might be an easy decision to make.

In terms of cost, the industry charges per snapshot. If you have 50 scenarios and test each one in 3 environments (Chrome, IE11, iOS Safari), a single test run will generate 150 snapshots. Generally, you pay for a monthly allotment of snapshots with any "overdraft" snapshots having a flat cost per snapshot. Ultimately one of the reasons we built our solution is that we generate over half a million snapshots a month running our visual test suite on every PR commit. This is well above the monthly allotments quoted by most services and can become quite expensive.

Free

If you want to add visual tests to your codebase there are also a few open source solutions that you can use for free, albeit with some setup. The three below were the ones that seemed most promising for Base Web:

- BackstopJS
- Loki
- Differencify

All of these tools give you an API for visiting pages, taking screenshots, asserting comparisons against a baseline, and updating snapshots as needed. These tools assume that you will store your image snapshots in Git. Also of great import, these tools provide a Docker image for running the tests in a CI pipeline, where environment consistency is required for deterministic screenshots.

These three tools, in particular, are easy to get started with and provide most of the basic functionality you need to get started with visual testing. If you are considering using snapshot tests it makes sense to try one of these out first and see if it works for you.

We tried each of these solutions with Base Web, and while it was simple to get started, we ended up needing more control over the test runs. What's more, the ready-made containers that came with these projects were proving difficult to integrate with our existing CI pipeline— so some of the out-of-the-box utility just wasn't there for us.

Looking at all of these solutions so far, you can begin to see that they all attempt to provide an entire workflow or framework for visual testing. This is why there are so many paid services. Visual testing requires pulling together many different solutions. Getting these right is essential for a happy and productive developer workflow.

Just so, we decided to roll out our solution, tailored for our needs. We already had a CI job set up for running end-to-end tests against our Storybook scenarios, so it didn't seem *that bad* to extend this functionality for generating visual snapshots.

Down to business

Let's go ahead and dive into how our solution works. After a high-level overview, we can drill into some of the implementation details, cover the tradeoffs with our current approach, and finally, we can consider some potential next steps.

The workflow

1 - Open a pull request

When a pull request is opened against `origin/master`, or a commit is pushed to a branch with an open pull request, our CI pipeline (Buildkite) begins a series of parallelized jobs. One of these jobs is responsible for running our visual snapshot tests. We will call this job `vrt` (Visual Regression Tests).

2 - Deploy scenarios

One of the initial steps in the pipeline is spinning up a simple static server that runs each of our Storybook scenarios on a separate page. This is tested against by `vrt` as well as our end-to-end (`e2e`) test job.

3 - Take screenshots of each scenario

When the Storybook scenario site is built, `vrt` initiates a suite of Jest tests focused on capturing visual snapshots. We use `playwright` to visit each scenario page, where we execute a collection of snapshot tests. We take one snapshot of a component at desktop size, one snapshot for mobile, one for our dark theme, and, if specified through a configuration file, we can run multiple "interactions" on a given scenario before taking snapshots.

4 - Compare screenshots to baseline

We use `jest-image-snapshot` for comparing a screenshot taken with our `playwright` instance against our baseline image snapshot. If a snapshot does not match our baseline, we update the snapshot locally within our Docker instance.

5 - Check for updates

After all the snapshot tests finish, `vrt` checks to see if any snapshots were updated locally. If none were updated, the job passes our `vrt` check. If there were updates, `vrt` commits those changes to a new branch, pushes the branch to GitHub, and then opens a pull request from the `vrt` branch into the original PR branch. The original branch owner is pinged for review and a comment is added to the original PR with information about the new visual snapshot PR. The `vrt` step is then failed to indicate that changes are required to the original PR branch before merging into `master`.

6 - Review & merge updates

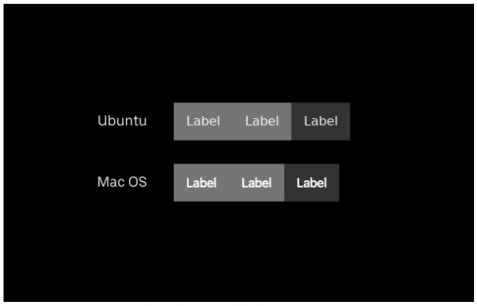
The `vrt` PR contains updated or new visual snapshot images. These can be easily reviewed in the GitHub interface. If the new snapshots look as expected, the original PR author can squash and merge the `vrt` PR into their branch. If the images look off the original author can amend their changes. Any new commit to the original PR branch will trigger a new `vrt` build, which if it passes will close any open `vrt` PR and delete the appropriate branch. If the new build does not pass, the `vrt` branch will be updated accordingly.

Implementation

There are a lot of different pieces needed to get the above system to work. Below, we will go over some of the tools and strategies you will need, as well as some of the gotchas you might encounter when building out a custom visual testing solution.

Docker

One of the first things you will encounter when adding visual tests to your codebase is that the environment of a test run matters. Fonts and colors are not rendered in the same way across all operating systems. The discrepancy is enough that a pixel diffing algorithm will consider images generated in each operating system as incompatible, even if they have rendered the same underlying source code. The result is that if someone runs the test suite on Mac and checks in the generated snapshots, all of the tests will fail when run on Windows or Linux.



The same snapshot rendered on both Mac OS and Ubuntu.

The solution for this is to always run the tests in the same environment, which for most projects means running tests in a Docker container. Our CI pipeline was already configured to use Docker & Linux so this became our "environment of truth". Note that it is also possible to set up your CI pipeline to always run tests on a Mac, Windows, or other environment but Linux is probably the most common choice.

Playwright

An essential tool to acquaint yourself with is **Playwright**. This is a library for automating the usage of Chrome via the Chrome Devtools Protocol. Playwright is what allows us to visit our various scenario pages, take screenshots of our components— it also allows us to script interactions on the page so we can capture states otherwise unreachable.

A static site

We need to render our components somewhere so that we can visit them with Playwright and generate screenshots. As part of our CI pipeline, we build a small static site that can render every Storybook scenario in our codebase. We use query parameters to find the `name` of the scenario to render as well as which `theme` to use (either light or dark). Once this site is built, we run visual regression tests as well as end-to-end tests against it.

```
const image = await page.screenshot(); // generate a new image with playwr
expect(image).toMatchImageSnapshot(); // compare to baseline image
```

If you already use snapshots in your codebase there is almost nothing new to learn here. The only difference from normal inline snapshots is that you will have image files added to your project. These will need to be kept under version control as baselines for future tests.

For a JS/React project, this is a very low overhead way to add visual tests to your codebase.

The test run

All of our snapshot tests are run under one Jest `describe` block. It essentially looks like this:

```
describe('Visual Regression Tests', () => {
  getAllScenarios().forEach((scenario) => {
    describe(scenario, () => {
      it('desktop', async () => {
        await setupDesktop();
        await snapshot();
      });
      it('mobile', async () => {
        await setupMobile();
        await snapshot();
      });
      it('dark', async () => {
        await setupDark();
        await snapshot();
      });
      getAllInteractionsForScenario(scenario).forEach((interaction) => {
        it(interaction.name, async () => {
          await interaction.behavior();
          await snapshot();
        });
      });
    });
  });
});
```

This is an abbreviated version of the real code, but it shows the structure of our test run. One of the key bits here is the `getAllScenarios()` function— this will return every Storybook scenario file in the codebase. The result is that you do not have to add any configuration to add a test- **if there is a Storybook scenario it will automatically become a visual regression test**.

That said, there *is* a configuration file for when you need to modify the behavior of a specific snapshot test. The configuration object is a little clunky but it lets you target a specific scenario and do things like skip the scenario or add additional interaction snapshots based on that scenario.

The config object looks something like this:

```
const config = {
  'input-password': {
    interactions: [
      {
        name: 'togglesMask',
        behavior: async (page) => {
          const toggleSelector = `[data-e2e="mask-toggle"]`;
          await page.$(toggleSelector);
          await page.click(toggleSelector);
        },
      },
    ],
  },
};
```

Each scenario can have an array of `interactions` assigned to it. The `behavior` for an `interaction` is a function that takes a **Playwright Page** instance and returns a Promise that when resolved has arranged the UI into the desired state for a snapshot.

In the above example, we will generate a new snapshot based on the `input-password` scenario. The behavior function will run before a snapshot is saved as `input-password__togglesMask-snap.png`.

Allowing for interactions makes it possible to test specific regressions or otherwise hard to reach states. In some cases adding interaction to a visual snapshot can replace complicated integration tests.

Screenshot dimensions

We went through a few iterations of tinkering with screenshot dimensions. At first, we only took a screenshot of the root element on the page. This had the advantage of capturing as small as possible an area- leading to smaller images. We had to abandon this approach, however, due to a few edge cases:

For one, any component that uses CSS to absolutely position elements, such as a modal or dropdown, will no longer contribute to the size of the root element. This means that most of the relevant UI will be left out of the final screenshot. This could be solved with some configuration parameters, but we wanted things to work well by default.

Another issue was that some elements would expand past the desired viewport width. Because of this, we weren't getting a real picture of what a mobile user would see and screenshots varied in width from snapshot to snapshot. The solution was to "clamp" the screenshot to a fixed width while capturing the maximum height of the page.

With Playwright that looks something like this:

```
// Use Chrome Devtools Protocol to get the scroll height of the page.
const client = await page.target().createCDPSession();
const metrics = await client.send('Page.getLayoutMetrics');
const height = Math.ceil(metrics.contentSize.height);
const image = await page.screenshot({
  clip: {
    x: 0,
    y: 0,
    width: VIEWPORT_WIDTH[viewport],
    height: height,
  },
});
```

The result is that every screenshot has the same dimensions for the given viewport (mobile/desktop) across test runs. This makes comparisons for developers a little bit easier, but it also shows us a more realistic simulation of what an end-user might see on their device.

The added size to each screenshot is fairly limited. The white/black areas can be compressed fairly efficiently so most of the blank space does not contribute to the size of the file. Even though the overall dimensions of each image increased by about 40% the storage footprint only increased by around 10%.

Flakiness

Snapshot tests should be deterministic. That is, they should yield the same result if run in the same environment with the same inputs. Yet even after nailing down a consistent environment, there can be variations in the UI that lead to flaky test results. When your test runs take 5-10 minutes, you don't want to have flaky tests be the reason your tests did not pass.

Here are the two main things that produced flakiness in our snapshots:

Animations

If you have any components that make use of transitions or animations you are going to want to shut those down. You don't want a millisecond difference in rendering time to affect the outcome of the test. Thankfully, it should be as simple as adding a little CSS to your page:

```
*,
*::before,
*::after {
  -moz-transition: none !important;
  transition: none !important;
  -moz-animation: none !important;
  animation: none !important;
  caret-color: transparent !important;
}
```

Asynchronicity

When testing components that require interactions, you often need to wait for the UI to change before you can proceed to the next step in the interaction. It might be tempting to simply pad steps with arbitrary wait times, but this is a sure way to add flakiness to your test.

```
await page.waitForTimeout(250); // waits for 250ms before proceeding
```

This should be a last resort as it will almost certainly flake at some point. When it does someone will probably just increase the wait duration- so over time these paddings accumulate. These can add up to make your test run unbearably slow. The

preferable approach is to wait for assertions on the page to pass. Take this example from one of our interaction snapshots:

```
const config = {
  'select-search-single': {
    interactions: [
      {
        name: 'open',
        behavior: async (page) => {
          const inputSelector = `[data-baseweb="select"]`;
          const dropdownSelector = `[role="listbox"]`;
          await page.waitForSelector(inputSelector);
          await page.click(inputSelector);
          await page.waitForSelector(dropdownSelector);
        },
      },
    ],
  },
};
```

Page, has numerous methods for awaiting events or assertions. While it is still possible for an assertion to time out while waiting, this rarely happens in practice. This approach takes a little more time to get right, but it guarantees you are not wasting time on your tests and greatly reduces the chances your tests will flake.

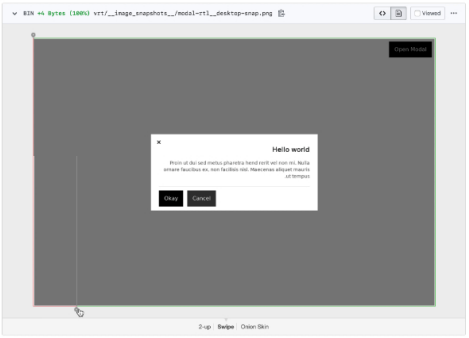
Developer UX

Having a visual test suite pass or fail as part of CI is a great start, but what do you do when a test fails? If you have access to the CI environment you can read the test run's output, but even knowing which tests failed is not especially useful if you can't *see* why. Consider also that sometimes you need to update snapshots or add new ones. If our CI test runs only report successes and failures we are back to running tests locally to update the snapshots.

We implemented a simple Node script to improve our workflow for viewing diffs and updating snapshots. The result is a much more seamless developer experience.

The script runs our visual snapshot tests with the update flag. If nothing is updated, the job has passed. If any snapshots were created or updated we raise a new pull request into the original branch with those updates. The author of the original pull request can then view new and updated snapshots in an isolated diff.

GitHub makes this process even better due to a few built-in image comparison tools. Not only can you see the images side by side, but you can also use two alternative methods of comparison known as "Swipe" and "Onion Skin".



An example of GitHub's image comparison interface.

Once the snapshot pull request is merged in, any new commits to the original branch will have the updated snapshots. It might seem a bit strange that tests are not always compared against `master`, but this is a useful feature because it allows developers to incrementally update snapshots as they work on a branch.

When the original pull request is merged into `master` it will contain any updated snapshots as part of its diff. We now have a changelog of the work associated with the visual changes that change caused and conversely, because the snapshots were in an isolated PR, we can also look at visual changes and easily find the pull request that they were associated with.

Finally, this workflow is much more inclusive for external contributors (of which there are over one hundred so far!). In the past, we used a paid solution for snapshot testing which required a privileged account for viewing snapshots. Our main team of contributors would have to log into that service and either relay information or fix it ourselves. Now external contributors can add and update snapshots without depending on anyone else or even having to learn anything new. After all, the whole system is built with plain old Git and GitHub. If you can merge in a pull request you can work with snapshots.

The future

So far we've covered how our current visual testing solution works and looked at some of the implementation details. What we have now is useful and, after working out most of the kinks, fairly seamless. That said, there is still a lot of room for improvement. Here are some of the things we are considering for the future.

Devices

The first and perhaps largest issue with our current solution is that our tests only run within a Chrome browser. While we can modify the viewport size and do some rough simulations of a mobile device with HTML meta tags, there is no denying that we aren't testing the real thing. We have to manually investigate regressions using real devices or a service such as BrowserStack. It would be ideal to have snapshots generated against mobile Safari and Internet Explorer 11 so that every major browser engine is accounted for.

Speed

The snapshot tests are the longest-running task in our CI pipeline, adding about 5-8 minutes to the test run. While this is run in parallel with other tasks, it would be ideal to halve or quarter this duration. To do this we would likely need to move tests out of Jest so that we could better parallelize the job. Jest is a convenient test runner, and we already have it set up in our project, but it is not necessarily the best tool for the job of running many side-effects (screenshots, reads/writes) laden tests.

Git

Our snapshots currently add about 10MB to our Git repository. It hasn't resulted in any noticeable difference in our workflow but as the project grows we might need to make use of Git LFS or move images out of Git entirely.

Downstream

Base Web is used by hundreds of internal projects at Uber. When we make a change we can use visual snapshots to verify everything looks good on our end, but we have no idea how our changes affect downstream projects unless we run those projects manually. It would be amazing if we could leverage the same system we have built for Base Web with dependent projects downstream. A project might implement a small suite of scenarios for their app which could be snapshotted and verified against every time we bump `baseui`.

A New Service

Moving images out of Git, speeding up test runs, investing in cross-browser testing, allowing other projects to use our tools... all of these goals point towards a more generalized service for visual snapshot testing. Essentially we want the benefits of one of the paid visual testing services, but scalable for hundreds of projects and with the option of privacy for internal projects.

This is likely the next step for our visual testing solution: Taking all of the logic spread out across GitHub, CI, Docker, and duck-tape utility scripts and making that easily runnable or accessible for other projects. By getting more projects onboard we also make the investment in new and improved features more valuable to the company. Ideally, the solutions we create can also be leveraged by the community, so stay tuned if you are interested in this space!

Conclusion

The purpose of this post is to give anyone interested in Visual Regression Testing an overview of their options and some reasons why the effort might be worthwhile. There are many factors to consider when evaluating what will work best for your project. By going over existing options, as well as the implementation and tradeoffs of our custom solution, we hope that you might be able to save some time in your own endeavors.

Thanks for reading!