Snapshot Testing: Benefits and Drawbacks

Jason Cheatham | May 19, 2020

REACT INTERN TESTING JEST ENTERPRISE APPS DEVELOPMENT SNAPSHOT

Snapshot testing has become very popular for front end-development over the last few years. The term has almost become synonymous with Jest and React, but it can be used to test more than just components. This article provides a brief overview of what snapshot testing is, what it isn't, and how it might be helpful for your project.

What is Snapshot Testing?

Snapshot testing is a type of "output comparison" or "golden master" testing. These tests prevent regressions by comparing the current characteristics of an application or component with stored "good" values for those characteristics. Snapshot tests are fundamentally different from unit and functional tests. While these types of tests make assertions about the correct behavior of an application, snapshot tests only assert that the output now is the same as the output before; it says nothing about whether the behavior was correct in either case.

Consider the two login forms below.

Original login form:

Login

username

password

Click me

Current login form:

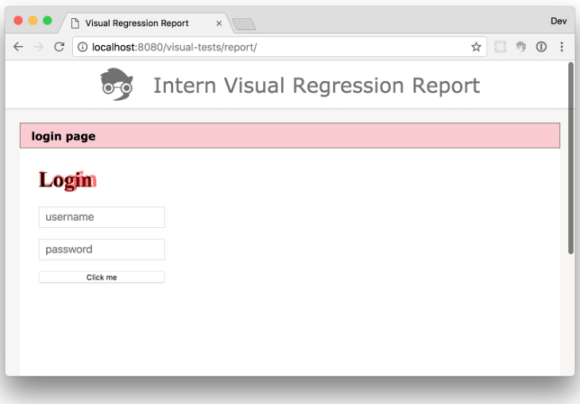
Login

username

password

Click me

The one on the left was the original, and the one on the right was the form after some recent styling changes. An image-based snapshot test would notice the change and highlight it, as in the report from Intern's visual-plugin below:



The test doesn't say whether the old or new stylings were correct, it just highlights to the tester that something has changed.

A variety of characteristics can be measured, but front-end tests typically focus on two: data (serializable JavaScript values) and images. For example, Depicted compares an image of a rendered page or component against a stored image of the rendered entity and flags any "perceptual" differences. Approval Tests compare JavaScript values, such as the output of application functions, to stored good values, similar to how unit tests

often work. Jest's snapshot tests also work for serializable JavaScript values, but they are most commonly applied to the DOM-based render trees of React components.

Snapshot testing isn't a new concept. The term has traditionally referred to visual regression testing, where a literal snapshot of a rendered app or page is compared to a stored image. However, Jest's render tests are what really brought the term into the mainstream for modern front-end developers, and this is what "snapshot testing" typically means in the JavaScript world.

How Snapshot Testing Works

Snapshot tests work by recording some characteristic of a system (e.g., taking a snapshot), and then later comparing that stored snapshot to the current value of the characteristic. For Jest-style tests, that characteristic is typically a serialized render tree:

```
const elem = renderer.create(<MyComponent label="foo">).toJSON();
expect(elem).toMatchSnapshot();
```

The first time a test is run, the toMatchSnapshot expectation saves the data it receives to a file. In this case, that's elem, a serialized render tree, which might look like:

```
<my-component
  label="foo"
  className="component"
/>
```

During later test runs, the current value of elem is compared to the stored "good" one. The test fails if the two values are different.

Other data-focused snapshot testing tools work similarly. The same test implemented using the snap-shot-it Mocha plugin would look like:

```
const elem = renderer.create(<MyComponent label="foo">).toJSON();
snapshot(elem);
```

Snapshots will eventually get out of sync with the components they represent and will have to get updated. Tools generally make this easy. For example, Jest snapshots can be updated by running

```
jest --updateSnapshot
```

Updating snapshots is very easy, making them very maintainable, but this can be both a blessing and a curse (more on that later).

The Benefits of Snapshot Testing

Writing tests can be a huge time sink. When Jest's snapshot testing feature was first announced, the developers said that "... engineers frequently told us that they spend more time writing a test than the component itself." This led to many developers saying they simply stopped writing tests entirely.

Snapshot tests can help out quite a bit in that situation because they're typically much shorter and easier to write than traditional unit tests. This snapshot test:

```
const elem = renderer.create(<MyComponent label="foo">).toJSON();
expect(elem).toMatchSnapshot();
```

is quite a bit simpler than this unit test:

```
const elem = renderer.create(<MyComponent label="foo">);
expect(elem).toHaveProperty('type', 'my-component');
expect(elem).toHaveProperty('props.label', 'foo');
expect(elem).toHaveProperty('props.className', 'base-component');
```

Snapshot tests are also easy to keep up to date as developers generally just need to run a single command to get the testing system to record new snapshots. This is certainly much easier than needing to edit many test files to bring tests back in sync with reality.

Tools for Snapshot Testing

There are quite a few tools that can be used for snapshot testing front-end code.

Several tools take snapshots of serializable data. Jest, as mentioned before, has built-in support for snapshot testing, and is frequently used to test React components. Cypress supports snapshot testing via plugins, such as the official @cypress/snapshot. Approval Tests supports snapshot testing for a number of languages, including JavaScript. The snap-shot-it JavaScript library adds snapshot testing capabilities to JavaScript-based BDD testing frameworks such as Mocha.

Many front-end tools focus on visual snapshots rather than data. Intern supports simple visual regression testing with its visual-plugin. The Jest-Image-Snapshot plugin adds visual snapshotting to Jest. Storybook, a UI development system, doesn't support snapshot testing itself, but it provides a rendering platform that many other tools use. Applitools performs "visual AI testing". It compares images more like a human would, ignoring imperceptible differences such as minor font and image rendering variances between different browsers, browser versions, and browsers on different operating systems.

Most of the tools mentioned above are for local testing, but there are also several cloud-based services for visual snapshot testing. These tools take care of managing the various browsers tests may run against, and also store the test snapshots. Chromatic is based around the Storybook. Percy can work with Storybook, and with a range of other testing systems.

The Drawbacks of Snapshot Testing

While snapshot tests are easier to write and keep up to date than traditional unit or functional tests, and they can be an effective tool for preventing regressions in an application, they do have several potential drawbacks.

A significant disadvantage is that they're tightly coupled to an application's output, making them very fragile.

Any changes, even to insignificant parts of the output, can cause snapshot tests to fail. Developers then must (or at least should) manually verify that everything is still working properly and update the snapshots.

This leads to another potential problem with snapshot tests: they don't actually indicate anything about the expected output, just that it shouldn't change. Unlike unit and functional tests, snapshot tests don't contain focused, meaningful assertions or expectations. A developer who has to manually verify that the output is still "good" may run into trouble when the failing test is for a part of the app he or she isn't familiar with, because a snapshot test doesn't indicate what parts of the output are important.

Snapshot tests aren't inherently well suited to dynamic content. A "random quote of the day" component will frequently fail snapshot tests since the random quote in the output usually won't match the stored component data. Tools can deal with this problem by letting users mark areas with dynamic content.

For example, Jest provides "asymmetric matchers" that can be used when creating snapshots to identify dynamic elements. In the snippet below, some properties in the userData value may vary from test session to test session. The `expect.any` directive tells Jest to accept any value of a given type rather than the specific value captured when the snapshot was originally created.

```
expect(userData).toMatchSnapshot({
    createdAt: expect.any(Date),
    id: expect.any(Number),
  });
```

Certain types of dynamic content can be even more problematic. For example, JavaScript and CSS animations change the visual representation of a page or component over time. Tools such as Percy and Chromatic have techniques for dealing with dynamic content, although some developer intervention may still be required.

Another potential drawback to snapshot tests is storage requirements. Snapshots must be stored somewhere; typically they're checked into the project repository. Depending on what characteristic is being recorded, snapshots can be quite large. Render trees are text-based, and generally diff and compress well, but an extensive suite of screenshot-based snapshot tests can easily consume tens of megabytes of space. This is another area where cloud-based services can provide some assistance by storing snapshots for visual tests.

Conclusion

Snapshot tests are easy to create and maintain, and they are a great way to check that your application's behavior isn't changing unexpectedly during development. However, they're not a replacement for unit and functional tests, which verify that an application is working correctly, not just that it hasn't changed.

Do you need help improving your current approach to testing and software architecture? Are you looking to explore how to best leverage snapshot testing as part of your automated testing strategy? Contact us to discuss how we can help you improve the reliability of your development workflow with robust automated testing.

Recent Posts

The Definitive TypeScript 5.0 Guide

Originally published October 2018. Updated March 2023. This article describes the features and...

Read More

Building a Serverless Chat Application with Supabase

Modern times have seen an explosion in services providing a multitude of serverless possibilities, but...

Read More

Using Redux-Saga to Write a Game Loop

Redux-Saga is an intuitive side effect manager for Redux.

Read More

Sign up to receive our latest articles on JavaScript, TypeScript, and all things software development!

Your Email Address