Hitting a Moving Target: Testing Javascript Animations in React with Jest

Author: Charles Rickarby

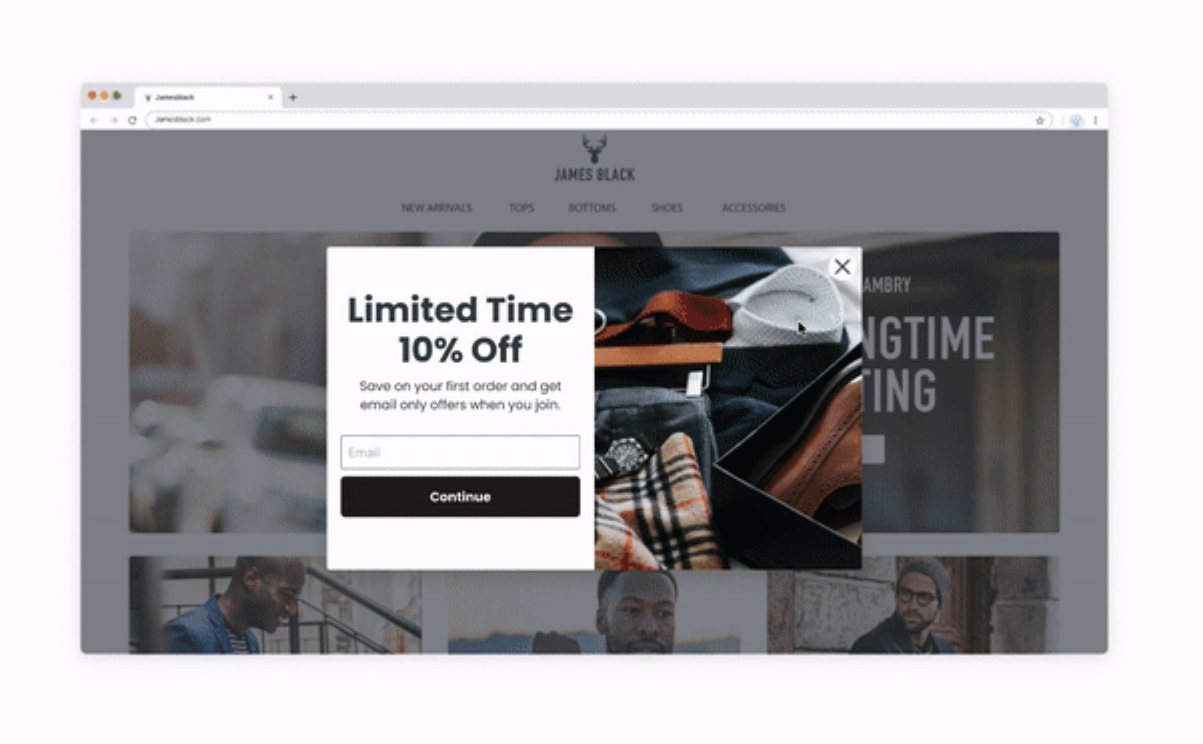
Claps: 48

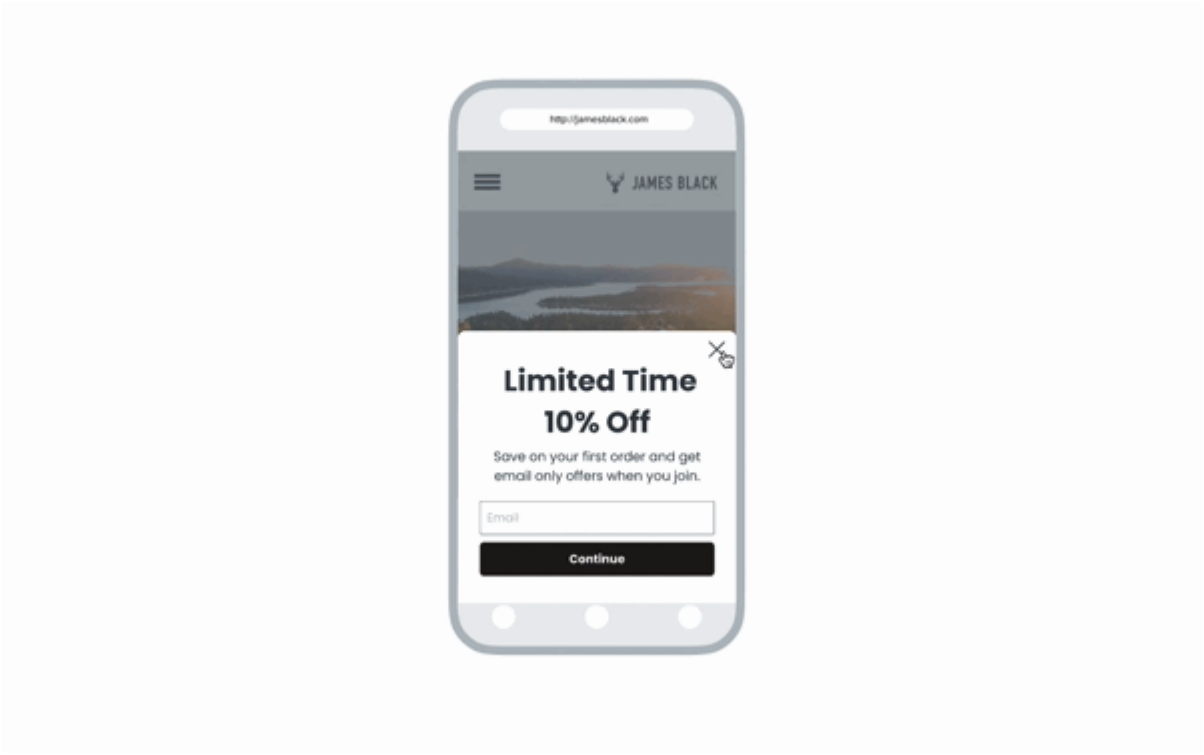
Date: Mar 6

We (engineers) are seeing more and more animations in our projects. If your TikTok For You page is anything like mine, it contains a surprising number of Figma and InDesign tutorials on creating modern user interfaces chock full of animations. These are delightful to users, but as an engineer, they give me a bit of a spook. In this post Iâ€™m going to outline what I ran into trying to write automated tests for animations, and the workarounds I found.

As a full stack engineer, animations are far from my day to day. Last year, however, I had the opportunity to work on a couple of features for the Klaviyo Signup Forms product where I got to dust off the animation skills I hadnâ€™t used since I first started web development.

Last year Klaviyo released our Form Teaser feature. Teasers are small widgets that appear before and/or after a bigger form on the page. They give visitors to our customersâ€™ sites an unobtrusive way to interact with forms.





A big part of this feature was that it needed to look like a native experience for site visitors, and be *delightful* to use, and that landed on animations.

There are two main types of animations that youâ€™ll see in web development: css animations and javascript animations. There are better articles out there that explain their differences and advantages, but at a high level, itâ€™s a good idea to opt for javascript animations over CSS animations when you want to build in more advanced effects such as stop, pause, rewind, re-run, or if you need a control of when and what to animate. Thinking about how teasers animate, we do a lot of these things (stop, start, repeat, rewind) so for flexibilityâ€™s sake we opted for javascript animations. Our javascript animation still relies on CSS keyframes in order to tell the components what to do while animating, but we control when the animations start using javascript.

Thereâ€™s nothing overly unique about how we built the animations. We store state as to whether or not a teaser or form is animating at any given time in a Redux store, so we can fire off other events based on that. The setting of that state is done through APIs on the elements animating:

`onAnimationStart` (the react wrapper around animationstart) â€” This event fires once when an elementâ€™s animation starts, and we use this to set various state in our Redux store so we know in other parts of the app whether or not the form or teaser was currently in animation.

`onAnimationEnd` (the react wrapper around animationend) â€” This event fires when a given element stops animating.

Most of this is well documented and therefore straightforward to implement. However, with front end work, sometimes the trickiest thing is writing the automated tests.

Hereâ€™s a simplified snippet of a test that I thought would just work right off the bat. The purpose of this test is to show that a teaser that shows on the page is visible once the form has closed. We check for the teaser by looking for the text contained within it, â€˜Get 15% Off.â€™

```
describe('A Teaser rendered after the form', () => {
  let form: RenderResult;
  const handlerMock = jest.fn();
  beforeAll(async () => {
    form = await utils.renderFormUsingWholeTriggeringSuite();
    // Wait for the form to render
    await form.findByPlaceholderText(EXPECTED_PLACEHOLDER);
  });
  it('switches to teaser when form is closed', async () => {
    // Expect the teaser to not be in the view
    expect(form.queryByText('Get 15% Off.')).not.toBeInTheDocument();
    const closeButton = await form.findByText(/Close form \d+/); // Close
    userEvent.click(closeButton);
    // Expect the teaser to be in the view
    await waitFor(() => {
      expect(screen.getByText('Get 15% Off.')).toBeVisible();
    });
    // Expect the form not to be in the view
    expect(
      screen.queryByPlaceholderText(EXPECTED_PLACEHOLDER)
    ).not.toBeVisible();
  });
});
```

But the test failed on the line that expects â€˜Get 15% Off.â€™ to be visible. After a bit of debugging, setting breakpoints, and using console logs, I came to the realization that this test was failing because the animations were never completing as expected.

At the time of writing this blog, searching â€œjavascript animation testing in Jestâ€¦ yields a couple of stack overflow hits, a forum question from a site that I havenâ€™t heard of before, but then it rolls into normal questions about testing in Jest and React, nothing animation specific. Thatâ€™s why Iâ€™m writing this!

Testing in Jest is, for the most part, a great way to validate your front end components and behavior, with few caveats. One of those caveats is that when testing in Jest, youâ€™re *not actually testing on the DOM*. Instead youâ€™re testing on Jestâ€™s version of what it thinks a DOM looks like, which most of the time is fine. One of those few times itâ€™s *not* fine is when it comes to animations.

Jestâ€™s DOM is headless and does not run animations like an actual browser. Part of what makes it such an efficient testing tool is that it doesnâ€™t need to actually render anything, but this becomes an issue when you want to test interactions that involve animations. For example, to test this sequence:

- Load page
- Verify teaser loads after delay on page
- Click teaser
- Verify that teaser animates out and form animates in

Weâ€™ve gotten around other animation issues in other testing tools (looking at you Cypress) by overriding the css property animation duration to 0s for all elements on the page, but that only works if the animation youâ€™re trying to evaluate happens on page load or isnâ€™t triggered via JavaScript. Itâ€™s one of those things that you expect to â€œjust workâ€¦ because thatâ€™s what a normal browser does but that wasnâ€™t the case with these tests.

Thankfully, there is a way to still be able to trigger these animations and test the state of our code. You just need to manually fire DOM events (we use react-testing-library).

Letâ€™s illustrate with some pseudo code.

```
import { fireEvent } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
// First we can get the teaser from the Jest DOM
const teaser = document.querySelector('#my-beautiful-teaser');
// Fire the click event on the teaser and the necessary animation events o
userEvent.click(teaser);
fireEvent.animationStart(teaser);
fireEvent.animationEnd(teaser);
expect(teaser).not.toBeVisible();
```

Now, you may be wondering, why did we fire two events? Do we really care if the animation ends? And really the question here is, it depends on what youâ€™re testing. For our case, we donâ€™t remove the teaser from the view until the animation has completed. Otherwise youâ€™d get this jarring strange animation on the end user side, where the animation would start and then the teaser would just vanish before it animated all the way out of view. Not great. However, if youâ€™re testing some other element that is supposed to appear or disappear just as animation of this element starts, then youâ€™re all set, and only need one event.

Lets see if we can apply those same principles to the test I mentioned above:

```
describe('A Teaser rendered after the form', () => {
  let form: RenderResult;
  const handlerMock = jest.fn();
  beforeAll(async () => {
    form = await utils.renderFormUsingWholeTriggeringSuite();
    await form.findByPlaceholderText(EXPECTED_PLACEHOLDER);
  });
  it('switches to teaser when form is closed', async () => {
```

```
    // Expect the teaser to not be in the view
    expect(form.queryByText('Get 15% Off.')).not.toBeInTheDocument();
    const closeButton = await form.findByText(/Close form \d+/);
    // Get a reference to the form's animation div
    const animationDiv = form.getByTestId('FLYOUT');
    // Close the form
    userEvent.click(closeButton);
    // Run the div's onAnimationEnd event
    fireEvent.animationEnd(animationDiv);
    // Get a reference to the teaser's animation div
    const animatedTeaser = await form.findByTestId('animated-teaser');
    // Run the teaser's onAnimationEnd event
    fireEvent.animationEnd(animatedTeaser);
    // Expect the teaser to be in the view
    await waitFor(() => {
      expect(screen.getByText('Get 15% Off.')).toBeVisible();
    });
    // Expect the form not to be in the view
    expect(
      screen.queryByPlaceholderText(EXPECTED_PLACEHOLDER)
    ).not.toBeVisible();
  });
});
```

And running that now, we see that it passes since weâ€™ve added in the animation events on the correct HTML elements. ðŸŽ‰

Here are examples of what we test: Once the teaser starts animating, does the form start animating? Once the form completes animation, is the teaser animation complete? Do all the correct events fire and was the state set properly by the teaser and form animations starting and completing? In this style of tests, we focus on testing state, and the presence/absence of various elements on the page versus visually testing what an element looks like when itâ€™s animating. In other words, we test functionality, not the look at each keyframe.

Knowing how to test animations within your future projects may seem like a novelty, but itâ€™s becoming important as web apps rely more on animation in order to provide a delightful experience.