Testing Strategies for React and Redux

By kumar303

Posted on April 24, 2018 in Featured Article and Web Developers

When the Firefox Add-ons team ported addons.mozilla.org to a single page app backed by an API, we chose React and Redux for powerful state management, delightful developer tools, and testability. Achieving the *testability* part isn't completely obvious since there are competing tools and techniques.

Below are some testing strategies that are working really well for us.

Testing must be fast and effective

We want our tests to be lightning fast so that we can ship high-quality features quickly and without discouragement. Waiting for tests can be discouraging, yet tests are crucial for preventing regressions, especially while restructuring an application to support new features.

Our strategy is to only test what's necessary and only test it once. To achieve this we test each *unit* in isolation, faking out its dependencies. This is a technique known as unit testing and in our case, the unit is typically a single React component.

Unfortunately, it's very difficult to do this safely in a dynamic language such as JavaScript since there is no *fast* way to make sure the fake objects are in sync with real ones. To solve this, we rely on the safety of static typing (via Flow) to alert us if one component is using another incorrectly — something a unit test might not catch.

A suite of unit tests combined with static type analysis is very fast and effective. We use Jest because it too is fast, and because it lets us focus on a subset of tests when needed.

Testing Redux connected components

The dangers of testing in isolation within a dynamic language are not entirely alleviated by static types, especially since third-party libraries often do not ship with type definitions (creating them from scratch is cumbersome). Also, Redux-connected components are hard to isolate because they depend on Redux functionality to keep their properties in sync with state. We settled on a strategy where we trigger all state changes with a real Redux store. Redux is crucial to how our application runs in the real world so this makes our tests very effective.

As it turns out, testing with a real Redux store is fast. The design of Redux lends itself very well to testing due to how actions, reducers, and state are decoupled from one another. The tests give the right feedback as we make changes to application state. This also makes it feel like a good fit for testing. Aside from testing, the Redux architecture is great for debugging, scaling, and especially development.

Consider this connected component as an example: (For brevity, the examples in this article do not define Flow types but you can learn about how to do that here.)

```
import { connect } from 'react-redux';
import { compose } from 'redux';

// Define a functional React component.
export function UserProfileBase(props) {
  return (
    <span>{props.user.name}</span>
  );
}

// Define a function to map Redux state to properties.
```

```
function mapStateToProps(state, ownProps) {
  return { user: state.users[ownProps.userId] };
}

// Export the final UserProfile component composed of
// a state mapper function.
export default compose(
  connect(mapStateToProps),
)(UserProfileBase);
```

You may be tempted to test this by passing in a synthesized user property but that would bypass Redux and all of your state mapping logic. Instead, we test by dispatching a real action to load the user into state and make assertions about what the connected component rendered.

```
import { mount } from 'enzyme';
import UserProfile from 'src/UserProfile';

describe('<UserProfile>', () => {
  it('renders a name', () => {
    const store = createNormalReduxStore();
    // Simulate fetching a user from an API and loading it into
    store.dispatch(actions.loadUser({ userId: 1, name: 'Kumar'

    // Render with a user ID so it can retrieve the user from s
    const root = mount(<UserProfile userId={1} store={store} />

    expect(root.find('span')).toEqual('Kumar');
  });
});
```

Rendering the full component with Enzyme's `mount()` makes sure `mapStateToProps()` is working and that the reducer did what this specific component expected. It simulates what would happen if the real application requested a user from the API and dispatched the result. However, since

`mount()` renders all components including nested components, it doesn't allow us to test `UserProfile` in isolation. For that we need a different approach using shallow rendering, explained below.

Shallow rendering for dependency injection

Let's say the `UserProfile` component depends on a `UserAvatar` component to display the user's photo. It might look like this:

```
export function UserProfileBase(props) {
  const { user } = props;
  return (
    <div>
      <UserAvatar url={user.avatarURL} />
      <span>{user.name}</span>
    </div>
  );
}
```

Since `UserAvatar` will have unit tests of its own, the `UserProfile` test just has to make sure it calls the interface of `UserAvatar` correctly. What is its interface? The interface to any React component is simply its properties. Flow helps to validate property data types but we also need tests to check the data values.

With Enzyme, we don't have to *replace* dependencies with fakes in a traditional dependency injection sense. We can simply infer their existence through shallow rendering. A test would look something like this:

```
import UserProfile, { UserProfileBase } from 'src/UserProfile';
import UserAvatar from 'src/UserAvatar';
import { shallowUntilTarget } from './helpers';

describe('<UserProfile>', () => {
  it('renders a UserAvatar', () => {
```

```
    const user = {
      userId: 1, avatarURL: 'https://cdn/image.png',
    };
    store.dispatch(actions.loadUser(user));

    const root = shallowUntilTarget(
      <UserProfile userId={1} store={store} />,
      UserProfileBase
    );

    expect(root.find(UserAvatar).prop('url'))
      .toEqual(user.avatarURL);
  });
});
```

Instead of calling `mount()`, this test renders the component using a custom helper called `shallowUntilTarget()`. You may already be familiar with Enzyme's `shallow()` but that only renders the first component in a tree. We needed to create a helper called `shallowUntilTarget()` that will render all "wrapper" (or higher order) components until reaching our target, `UserProfileBase`.

Hopefully Enzyme will ship a feature similar to `shallowUntilTarget()` soon, but the implementation is simple. It calls `root.dive()` in a loop until `root.is(TargetComponent)` returns true.

With this shallow rendering approach, it is now possible to test `UserProfile` in isolation yet still dispatch Redux actions like a real application.

The test looks for the `UserAvatar` component in the tree and simply makes sure `UserAvatar` will receive the correct properties (the `render()` function of `UserAvatar` is never executed). If the properties of `UserAvatar` change and we forget to update the test, the test might still pass, but Flow will alert us about the violation.

The elegance of both React and shallow rendering just gave us dependency injection for free, without having to inject any dependencies! The key to this testing strategy is that the implementation of `UserAvatar` is free to evolve on

its own in a way that won't break the `UserProfile` tests. If changing the implementation of a unit forces you to fix a bunch of unrelated tests, it's a sign that your testing strategy may need rethinking.

Composing with children, not properties

The power of React and shallow rendering really come into focus when you compose components using children instead of passing JSX via properties. For example, let's say you wanted to wrap `UserAvatar` in a common `InfoCard` for layout purposes. Here's how to compose them together as children:

```
export function UserProfileBase(props) {
  const { user } = props;
  return (
    <div>
      <InfoCard>
        <UserAvatar url={user.avatarURL} />
      </InfoCard>
      <span>{user.name}</span>
    </div>
  );
}
```

After making this change, the same assertion from above will still work! Here it is again:

```
expect(root.find(UserAvatar).prop('url'))
  .toEqual(user.avatarURL);
```

In some cases, you may be tempted to pass JSX through properties instead of through children. However, common Enzyme selectors like `root.find(UserAvatar)` would no longer work. Let's look at an example of passing `UserAvatar` to `InfoCard` through a `content` property:

```
export function UserProfileBase(props) {
  const { user } = props;
  const avatar = <UserAvatar url={user.avatarURL} />;
  return (
    <div>
      <InfoCard content={avatar} />
      <span>{user.name}</span>
    </div>
  );
}
```

This is still a valid implementation but it's not as easy to test.

Testing JSX passed through properties

Sometimes you really can't avoid passing JSX through properties. Let's imagine that `InfoCard` needs full control over rendering some header content.

```
export function UserProfileBase(props) {
  const { user } = props;
  return (
    <div>
      <InfoCard header={<Localized>Avatar</Localized>}>
        <UserAvatar url={user.avatarURL} />
      </InfoCard>
      <span>{user.name}</span>
    </div>
  );
}
```

How would you test this? You might be tempted to do a full Enzyme `mount()` as opposed to a `shallow()` render. You might think it will provide you with better test coverage but that additional coverage is not necessary — the `InfoCard`

component will already have tests of its own. The `UserProfile` test just needs to make sure `InfoCard` gets the right properties. Here's how to test that.

```
import { shallow } from 'enzyme';
import InfoCard from 'src/InfoCard';
import Localized from 'src/Localized';
import { shallowUntilTarget } from './helpers';

describe('<UserProfile>', () => {
  it('renders an InfoCard with a custom header', () => {
    const user = {
      userId: 1, avatarURL: 'https://cdn/image.png',
    };
    store.dispatch(actions.loadUser(user));

    const root = shallowUntilTarget(
      <UserProfile userId={1} store={store} />,
      UserProfileBase
    );

    const infoCard = root.find(InfoCard);

    // Simulate how InfoCard will render the
    // header property we passed to it.
    const header = shallow(
      <div>{infoCard.prop('header')}</div>
    );

    // Now you can make assertions about the content:
    expect(header.find(Localized).text()).toEqual('Avatar');
  });
```

This is better than a full `mount()` because it allows the `InfoCard` implementation to evolve freely so long as its properties don't change.

Testing component callbacks

Aside from passing JSX through properties, it's also common to pass callbacks to React components. Callback properties make it very easy to build abstractions around common functionality. Let's imagine we are using a `FormOverlay` component to render an edit form in a `UserProfileManager` component.

```
import FormOverlay from 'src/FormOverlay';

export class UserProfileManagerBase extends React.Component {
  onSubmit = () => {
    // Pretend that the inputs are controlled form elements and
    // their values have already been connected to this.state.
    this.props.dispatch(actions.updateUser(this.state));
  }

  render() {
    return (
      <FormOverlay onSubmit={this.onSubmit}>
        <input id="nameInput" name="name" />
      </FormOverlay>
    );
  }
}

// Export the final UserProfileManager component.
export default compose(
  // Use connect() from react-redux to get props.dispatch()
  connect(),
)(UserProfileManagerBase);
```

How do you test the integration of `UserProfileManager` with `FormOverlay`? You might be tempted once again to do a full `mount()`, especially if you're testing integration with a third-party component, something like Autosuggest. However, a full `mount()` is not necessary.

Just like in previous examples, the `UserProfileManager` test can simply check the properties passed to `FormOverlay`. This is safe because `FormOverlay` will have tests of its own and Flow will validate the properties. Here is an example of testing the `onSubmit` property.

```
import FormOverlay from 'src/FormOverlay';
import { shallowUntilTarget } from './helpers';

describe('<UserProfileManager>', () => {
  it('updates user information', () => {
    const store = createNormalReduxStore();
    // Create a spy of the dispatch() method for test assertior
    const dispatchSpy = sinon.spy(store, 'dispatch');

    const root = shallowUntilTarget(
      <UserProfileManager store={store} />,
      UserProfileManagerBase
    );

    // Simulate typing text into the name input.
    const name = 'Faye';
    const changeEvent = {
      target: { name: 'name', value: name },
    };
    root.find('#nameInput').simulate('change', changeEvent);

    const formOverlay = root.find(FormOverlay);

    // Simulate how FormOverlay will invoke the onSubmit proper
    const onSubmit = formOverlay.prop('onSubmit');
    onSubmit();

    // Make sure onSubmit dispatched the correct ation.
```

This tests the integration of `UserProfileManager` and `FormOverlay` without relying on the implementation of `FormOverlay`. It uses sinon to spy on the

`store.dispatch()` method to make sure the correct action is dispatched when the user invokes `onSubmit()`.

Every change starts with a Redux action

The Redux architecture is simple: when you want to change application state, dispatch an action. In the last example of testing the `onSubmit()` callback, the test simply asserted a dispatch of `actions.updateUser(...)`. That's it. This test assumes that once the `updateUser()` action is dispatched, everything will fall into place.

So how would an application like ours actually update the user? We would connect a saga to the action type. The `updateUser()` saga would be responsible for making a request to the API and dispatching further actions when receiving a response. The saga itself will have unit tests of its own. Since the `UserProfileManager` test runs without any sagas, we don't have to worry about mocking out the saga functionality. This architecture makes testing very easy; something like redux-thunk may offer similar benefits.

Summary

These examples illustrate patterns that work really well at addons.mozilla.org for solving common testing problems. Here is a recap of the concepts:

- We dispatch real Redux actions to test application state changes.
- We test each component only once using shallow rendering.
- We resist full DOM rendering (with `mount()`) as much as possible.
- We test component integration by checking properties.
- Static typing helps validate our component properties.
- We simulate user events and make assertions about what action was dispatched.

Want to get more involved in Firefox Add-ons community? There are a host of ways to contribute to the add-ons ecosystem – and plenty to learn, whatever your skills and level of experience.

About kumar303

Kumar hacks on Mozilla web services and tools for various projects, such as those supporting Firefox Add-ons. He hacks on lots of random open source projects too.

- farmdev.com/
- @kumar303
- Facebook

More articles by kumar303...

One comment

Gabriel Micko

Awesome post, thank you!

April 24th, 2018 at 13:16

Comments are closed for this article.