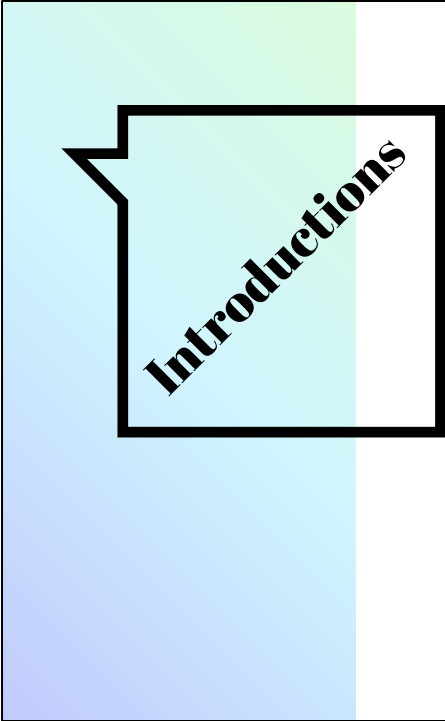


# Getting Under The Skin: Subcutaneous Testing

*A Working Example Using JSDOM  
for Functional Testing*

Is everyone having a good day so far?

Thank you for being here!



# Introductions

Avalon McRae  
Senior Developer  
@ThoughtWorks

Melissa Eaden  
Editor & Writer  
@Ministry of  
Testing

Tech Lead QE  
@Unity Tech

Avalon & Melissa

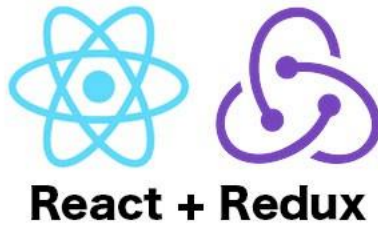
Hi we're Avalon McRae and Melissa Eaden.

Avalon - Sr. Full Stack Developer @ThoughtWorks

Melissa - Formerly a Sr QA @ThoughtWorks and now The Content Editor & Writer @  
Ministry of Testing and a Tech Lead QE @Unity

# Tech Stack

---



Melissa

Avalon and I were working together on a retail scheduling product. This particular product would allow customers to go online and schedule appointments for various services, send them an email with appointment details and a confirmation code.

This was a one page application written in React/Redux libraries which had to interact with other APIs and Applications. In fact, the only way to get to this Single page application was through other applications.

The client the team was working with had a lot of GUI automation in a traditional functional testing framework for the external application our application was embedded in.



## **Shaking Things Up!**

### Traditional Functional Tests:

- Flaky/Poor reliability
- Hard to Maintain
- Slow Feedback Loop

Melissa

The client test framework had the problems a lot of traditional functional testing frameworks have if you are testing everything and the kitchen sink.

It wasn't reliable.

With lots of failing tests remaining un-addressed either due to actual failures, or test script issues, but it was hard to tell because the tests runs were rarely green, and because it took a significant amount of time to run them, most people opted out of running the test suite altogether.

Because of the constant failures, it wasn't a very nice suite to maintain and pretty much everyone grumbled, including client QA folks, about maintaining it. Tests would be commented out with the idea that someone would fix them. More often than not, no one would get back to that test.

This all led to slow feedback loops for everyone involved in the development process. QAs couldn't get ahead on testing, devs were unaware of issues until late in the testing phases, and the process of testing leaned more and more into the manual area for regressions since doing so was faster than fixing whatever was broken.

It's true that some of this might be better addressed organizationally, but those large ship-turning changes take time to fix, takes time to change behavior, and we only had

three months to build our app.

With those pressures in mind, our team decided to take a different approach with the functional testing model by leveraging the React/Redux library application we built.



## Shaking Things Up!

### Subcutaneous Tests :

- Reliable
- Fast Feedback
- Lives With Source Code

#### Avalon

##### Reliable

-Don't experience the same types of flakiness seen in traditional, browser-based functional testing such as "element not found" selector flakiness or timeouts

##### Lives with source code:

- Re-using the same stack and even some of the test set up
- Living closer to the code

-We actually ran these together with our unit tests so they're easier to keep maintained by developers and less likely to get out of date

you can run these as part of the same task as unit tests if you want to (you don't have to). A lot of react projects are using jest now - webdriverio doesn't support jest so we had to switch between testing frameworks and assertion libraries between unit and functional before this. Can share some of the test setup you use for unit tests (stubs of api calls, selectors, etc).

Fast feedback -> on my current project we have 13 of these tests that run in 7 seconds (most selenium suites i've had have taken at least a few minutes, plus having to re-run for flakiness)

Our team saw a huge benefit from these tests, we're running the functional tests many times a day both locally during development and in our pipeline  
So now that we've talked about \*why\* we want to do these types of tests, let's talk about what these tests actually are in a little bit more detail...



## Definition:

"If a unit test is testing the smallest testable component of application code, then a subcutaneous test is a single workflow which can be identified and tested, of the application code. Subcutaneous testing treats a workflow as a testable unit."

Subcutaneous Testing

Melissa

When we started talking about Subcutaneous testing as a team, we felt the need to come up with a definition that fit what the team was practicing.

There have been a few mentions of Subcutaneous testing over the years, even as far back as 2010. It's not a new concept, but it is definitely gaining traction where teams are using React/Redux to build applications.

Our team worked through several versions of this definition and came up with the one you are reading now.

The most important part of this definition I would point is: "Subcutaneous testing treats a workflow as a testable unit."



## An Example: Happy Doggy Grooming

Human's Name	<input type="text" value="Sally Fields"/>	✓
Dog's Name	<input type="text" value="Fisher"/>	✓
E-Mail	<input type="text" value="lheartdogs@gmail.com"/>	✓
Phone #	<input type="text" value="(512)555-5555"/>	✓

*Full Day Spa*

*Basic Grooming*

*Basic Bath*

Melissa

As I mentioned before our team was working on a small, embedded scheduling application.

This one is a theoretical application mocked up to give you a sense of what the team was trying to accomplish.

The tech stack for this theoretical app was created with react/redux/css/html and we have restful api call to submit information

Imaging this embedded app is somewhere on the landing page for Happy Doggy Grooming.

You have filled out your personal information, and your dog's information. Next you'll be selecting a service.

## An Example: Happy Doggy Grooming

*Full Day Spa*

OPTIONS

- Bath
- Hand-towel dry
- Nail Trim
- Grooming
- Full Day of Daycare

« November 2018 »

SUN	MON	TUE	WED	THU	FRI	SAT
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

SELECT

*Basic Grooming*

*Basic Bath*

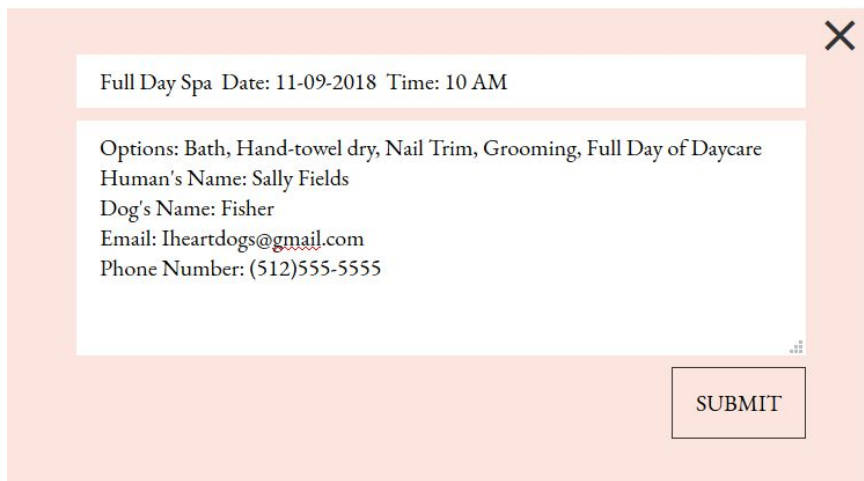
Melissa

Here it shows an option was selected.

Then a date was selected for the doggy day spa appointment.

Then by clicking select, another screen is presented.

## An Example: Happy Doggy Grooming



Full Day Spa Date: 11-09-2018 Time: 10 AM

Options: Bath, Hand-towel dry, Nail Trim, Grooming, Full Day of Daycare

Human's Name: Sally Fields

Dog's Name: Fisher

Email: Iheartdogs@gmail.com

Phone Number: (512)555-5555

SUBMIT

Melissa

The modal shows all the information from the previous screen. By clicking submit, the user should be presented with a confirmation message, receive an email, see a confirmation page... whatever combination of interactions the business requests. On the back end, the API is sending the information to the appropriate places to be processed.

This is a common workflow which could be automated through a typical JavaScript/webdriver, or selenium framework.

Depending on the browser someone is using, it might be a big pain to maintain several workflows, especially if other workflows needed to be checked like cancellations, modifications, or special requests.

In Subcutaneous testing: A Workflow of the app equals a testable unit, or a test case, very much like a typical GUI functional test.

I've called a workflow a testable unit for a reason.

While subcutaneous testing uses the same testing framework as the unit tests there is a distinct difference in how to approach a unit test versus a subcutaneous test.

Unit tests are mostly testing a behavior inside of a container. Only that container is

mounted and tested with a specific test in mind.

If you are unfamiliar with what a container is, here is a quick explanation...

## Containers:

“A container does data fetching and then renders its corresponding sub-component.”

StockWidgetContainer => **StockWidget**

TagCloudContainer => **TagCloud**

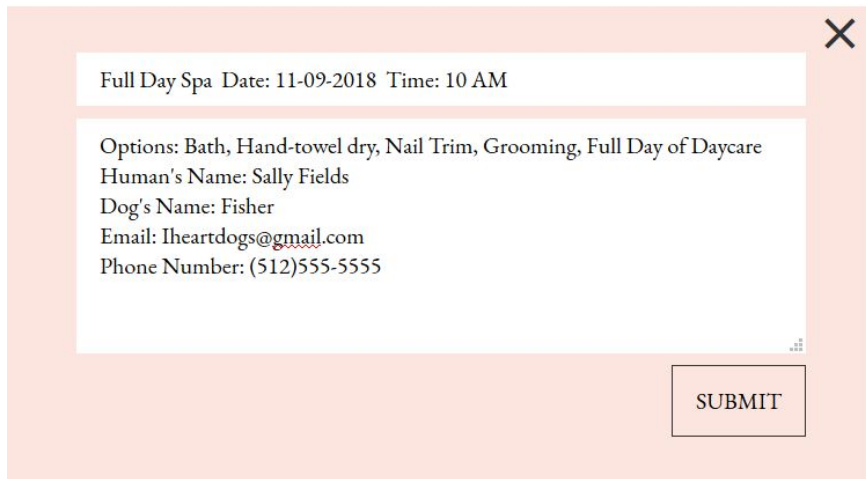
PartyPooperListContainer => **PartyPooperList**

Subcutaneous Testing

Melissa

Corresponding means basically a component with the same name. This allows separation of rendering and data fetching which is one of the benefits of react. Which in turn provides us with the building blocks for doing subcutaneous testing.

## An Example: Happy Doggy Grooming



Full Day Spa Date: 11-09-2018 Time: 10 AM

Options: Bath, Hand-towel dry, Nail Trim, Grooming, Full Day of Daycare

Human's Name: Sally Fields

Dog's Name: Fisher

Email: lheartdogs@gmail.com

Phone Number: (512)555-5555

SUBMIT

Melissa

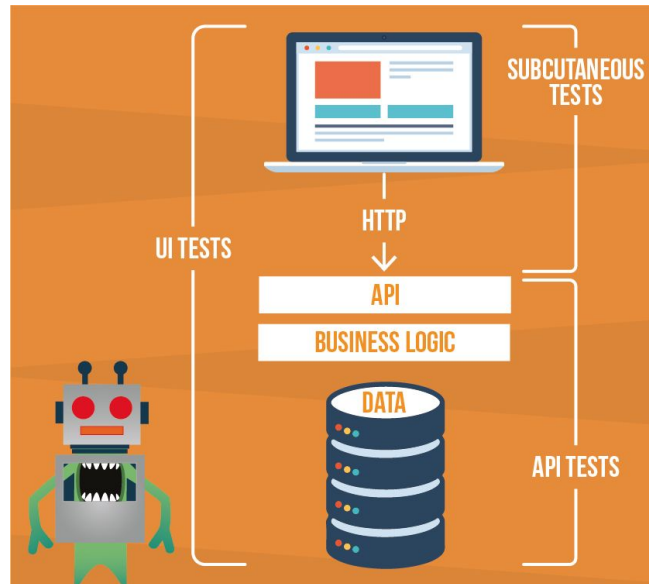
Subcutaneous tests can test the interactions between the various containers of a React application. Since we are treating the workflow as a unit, instead of mounting one container, we mount the whole application using a neat library called JS DOM.

For Example, when you select an appointment option, the date picker is there, you click select, then verify the information (which is the screen you are looking at) then click submit.

With the Subcutaneous test, JS DOM allows for a partial render of the elements we are clicking on, like a user would, like a typical selenium functional test would, but without the added complication of browser/webdriver compatibility issues.

Avalon is going to walk through our overall technical approach to testing and show you how we created these tests and where they live in the general testing approach.

## Technical Approach



### Avalon

Here we have a simple architectural diagram mirroring the architecture of our domain example. On the left is a more traditional approach to a testing pyramid - we have unit tests that cover a small sliver of the UI and then functional tests that test the entire vertical, as well as unit and integration tests for the back end (we won't go in to this part too much)

Speak to issues with unit testing vs. integration and trying to debug at a functional level ?

On the right is the approach that we used to incorporate subcutaneous testing. These tests cover the whole UI and the integration between front end components, but they stop there. For testing the back end and our contract with the BFF, we have integration tests, contract tests, unit tests, etc.

**Notice that we're still covering the entire vertical of the application, but what we're proposing is that UI tests can be split up from the dependency on the back end the way we often split up integration tests. Reiterating what we mentioned earlier, this can improve run time and decrease flakiness of these UI tests.**

**If you still want to have a test that tests a vertical slice of integration of the entire application, you can have one happy path functional test that spins up the browser, hits the real services, etc. - but the point is that not every single UI/user flow test we have needs to do this.**

## Leveraging jsdom as a headless browser

---

- What is jsdom?
- jsdom vs PhantomJS
- React Router

```
const App = <all app code>
ReactDOM.render(<App />)
```

### Avalon

The reason we are able to create these tests that simulate interacting with the actual pages without fully starting the app and having to bring up the browser is because we are using something called jsdom.

[JSDOM](#) is a JavaScript based headless browser that can be used to create a realistic testing environment.

Since enzyme's [mount](#) API requires a DOM, JSDOM is required in order to use mount if you are not already in a browser environment (ie, a Node environment)

### Jsdm vs phantom js

-jsdom is more lightweight - it does not perform layout or rendering and it does not support page changes

-jsdom is much quicker to run

-can't take screenshots because it isn't actually rendering

So how we can talk about testing a user workflow without page changes? Well, the reason this is able to work here is because we are using react router, which does in-memory routing instead of browser based routing.

Testing all of the code to create the UI of the application

So, while HTML is a text, the DOM is an in-memory representation of this text.



Render a React element into the DOM

```

describe('successfully schedules an appointment', () => {
  beforeEach(() => {
    store = createStore();
    appointmentTypes = mockLoadAppointmentTypes({ storeId });
    selectedAppointmentType = appointmentTypes[0].id;
    timeSlots = mockLoadAvailableTimeSlots({ storeId, date, selectedAppointmentType });
  });

  afterEach(fetchMock.restore);

  it('should navigate to confirmation page and display appointment info after successfully scheduling', () => {
    mockScheduleAppointmentCall();

    const app = mount(
      <Provider store={store}>
        <App />
      </Provider>
    );
    const schedulePageObject = new SchedulePageObject(app);

    await createWaitForElement('select')(app);
    schedulePageObject.selectGroomingAppointmentType();

    await createWaitForElement('.DatePicker')(app);
    const selectedTimeSlot = moment(timeSlots[0].dateTime.toString());
    schedulePageObject.selectTimeSlot(selectedTimeSlot);

    schedulePageObject.fillInPersonalInfo();
    schedulePageObject.submit();

    await createWaitForElement('.ConfirmationMessage')(app);
    const confirmationMessage = app.find('.ConfirmationMessage');

    expect(confirmation.text()).toBe('Your grooming appointment has been scheduled for November 9th at 10:00 AM');
  });
});

```

## Avalon

(walk through test like domain example - note that it mirrors the flow of a happy path functional test)

Here we are using enzyme (a very widely used testing library for React) to mount our entire application via jsdom, which is essentially a headless browser implemented completely in javascript that is automatically configured for you if you're using jest (or pretty simple to configure for other tools like mocha, etc).

We are mounting our entire application, including the router, all of our logic with our actual redux store. We are essentially using the same testing frameworks and tools as the unit tests to create end-to-end (or workflow) tests which are testable as a unit

So as we went through implementing this example, it's a newer approach and we definitely went through some cycles of learnings and gotchas, so Melissa and I wanted to walk through step by step how we came to this example and some of the decisions behind our approach.

```

describe('successfully schedules an appointment', () => {
  beforeEach(() => {
    store = createStore();
    appointmentTypes = mockLoadAppointmentTypes({ storeId });
    selectedAppointmentType = appointmentTypes[0].id;
    timeSlots = mockLoadAvailableTimeSlots({ storeId, date, selectedAppointmentType });
  });

  afterEach(fetchMock.restore());

  it('should navigate to confirmation page and display appointment info after successfully scheduling', () => {
    mockScheduleAppointmentCall();

    const app = mount(
      <Provider store={store}>
        <App />
      </Provider>
    );
    const schedulePageObject = new SchedulePageObject(app);

    await createWaitForElement('select')(app);
    schedulePageObject.selectGroomingAppointmentType();

    await createWaitForElement('.DatePicker')(app);
    const selectedTimeSlot = moment(timeSlots[0].dateTime.toString());
    schedulePageObject.selectTimeSlot(selectedTimeSlot);

    schedulePageObject.fillInPersonalInfo();
    schedulePageObject.submit();

    await createWaitForElement('.ConfirmationMessage')(app);
    const confirmationMessage = app.find('.ConfirmationMessage');

    expect(confirmationMessage.text()).toBe('Your grooming appointment has been scheduled for November 9th at 10:00 AM');
  });
});

```

This is the ONLY thing we are stubbing/mocking - no mock store or anything like that.

This is something we \*chose\* to do - don't necessarily have to do this.

## Mocking API calls

---

- With Javascript (e.g. fetch-mock, moxios,nock)
  - Allows you to parameterize your mocks and their responses more easily
  - Can reuse these mocks in unit tests
- With Over the Wire Test Doubles (e.g. mountebank, wiremock)
  - Don't need to add any code in your tests to mock api calls
  - Stubs are reusable for local development
  - More complexity
    - May require a few different stubs for each endpoint
    - More difficult to set up

This is the ONLY thing we are stubbing/mocking - no mock store or anything like that  
Mocking your HTTP client (axios, fetch)

Mountebank is a tool that provides over the wire test doubles, allows you to point your application under test to mountebank instead of the real dependency (in this case, our BFF)

Don't need to add any code in your tests to mock api calls, just need to change your configuration to point your api calls to the port where mb is running

Another thing that is valuable about using mountebank is that you can also use those stubs for other purposes, such as running the app locally. For instance, if we want to test that the api returns a certain status code.

## An example using fetch-mock

---

```
import fetchMock from 'fetch-mock';

class ScheduleAppointmentStubBuilder {
  constructor() {
    this.response = {
      firstName: 'Jill',
      lastName: 'Johnson',
      confirmationCode: 'ABCDEF',
      phone: '12333344444',
      email: 'jilljohnson@email.com',
      appointmentType: 'Grooming',
      storeId: '323432',
      appointmentDateTime: '2018-11-10T15:10:07+00:00',
      appointmentId: '60bc3f20-9eb8-4895-ae9f-fc5317159ea1'
    }
  }

  withAppointmentType(type) {
    this.response.appointmentType = type;
    return this;
  }

  withStoreId(storeId) {
    this.response.storeId = storeId;
    return this;
  }

  build() {
    fetchMock.post('/schedule', this.response);
    return response;
  }
}
```

If showing a builder pattern speak to flexibility of mocks

# Page Object Model

```
2 import BasicInfoForm from '../BasicInfoForm/BasicInfoFormContainer';
3 import BasicInfoFormPageObject from './basicInfoFormPageObject';
4
5 class SchedulerPageObject {
6   constructor(container) {
7     this.container = container;
8   }
9
10  selectBox() {
11    return this.container.find('select');
12  }
13
14  selectedReasonId() {
15    return this.selectBox().props().value;
16  }
17
18  appointmentComments() {
19    return this.container.find('.PleaseSpecifyFieldContainer').find('input');
20  }
21
22  updateAppointmentComments(value) {
23    this.container.find('.PleaseSpecifyFieldContainer').find('input')
24      .simulate('change', { target: { value } });
25  }
26
27  datePicker() {
28    return this.container.find(DatePicker);
29  }
30
31  basicInfoForm() {
32    return new BasicInfoFormPageObject(this.container.find(BasicInfoForm));
33  }
34
35  selectReasonById(reasonId) {
36    this.selectBox().props().onChange({ target: { value: reasonId } });
37  }
38}
```

a model used a lot in traditional functional tests

The main difference here is that we are using enzyme selectors, which select in jsdom rather than selectors that traverse the actual dom. This really just translates to us calling `app.find(<class name>)` instead of `document.getElementsByClassName`

Another nice thing is that we can reuse these methods in our unit tests if we want to. So, if we want to change a class name or something that we're selecting on, we might only need to change it in one place for both unit and functional tests.

# Enzyme-wait and async/await

```
describe('successfully schedules an appointment', async () => {
  beforeEach(() => {
    store = createStore();
    appointmentTypes = mockLoadAppointmentTypes({ storeId });
    selectedAppointmentType = appointmentTypes[0].id;
    timeSlots = mockLoadAvailableTimeSlots({ storeId, date, selectedAppointmentType });
  });

  afterEach(fetchMock.restore);

  it('should navigate to confirmation page and display appointment info after successfully scheduling', () => {
    mockScheduleAppointmentCall();

    const app = mount(
      <Provider store={store}>
        <App />
      </Provider>
    );
    const schedulePageObject = new SchedulePageObject(app);

    await createWaitForElement('select')(app);
    schedulePageObject.selectGroomingAppointmentType();

    await createWaitForElement('.DatePicker')(app);
    const selectedTimeSlot = moment(timeSlots[0].dateTime.toString());
    schedulePageObject.selectTimeSlot(selectedTimeSlot);

    schedulePageObject.fillInPersonalInfo();
    schedulePageObject.submit();

    await createWaitForElement('.ConfirmationMessage')(app);
    const confirmationMessage = app.find('.ConfirmationMessage');

    expect(confirmation.text()).toBe('Your grooming appointment has been scheduled for November 9th at 10:00 AM');
  });
});
```

Because we're walking through a user flow with multiple api calls, page changes, etc. we have to deal with more async behavior and promises.

We found a small, easy to use library called enzyme wait. NPM package that we used in conjunction with javascript's async/await functionality to deal with promises in a more reasonable way, a little cleaner than setTimeout, especially when chaining a lot of async behavior

Similar pattern to selenium tests using waitForElement

## Example Test Case

```
describe('successfully schedules an appointment', () => {
  beforeEach(() => {
    store = createStore();
    appointmentTypes = mockLoadAppointmentTypes({ storeId });
    selectedAppointmentType = appointmentTypes[0].id;
    timeSlots = mockLoadAvailableTimeslots({ storeId, date, selectedAppointmentType });
  });

  afterEach(fetchMock.restore);

  it('should navigate to confirmation page and display appointment info after successfully scheduling', () => {
    mockScheduleAppointmentCall();

    const app = mount(
      <Provider store={store}>
        <App />
      </Provider>
    );
    const schedulePageObject = new SchedulePageObject(app);

    await createWaitForElement('select')(app);
    schedulePageObject.selectGroomingAppointmentType();

    await createWaitForElement('.DatePicker')(app);
    const selectedTimeSlot = moment(timeSlots[0].dateTime.toString());
    schedulePageObject.selectTimeSlot(selectedTimeSlot);

    schedulePageObject.fillInPersonalInfo();
    schedulePageObject.submit();

    await createWaitForElement('.ConfirmationMessage')(app);
    const confirmationMessage = app.find('.ConfirmationMessage');

    expect(confirmationMessage.text()).toBe('Your grooming appointment has been scheduled for November 9th at 10:00 AM');
  });
});
```

Walk through each step again, note how similar this looks to a selenium test

- We can use enzyme to traverse the virtual dom just like we would traverse the HTML dom in a functional test (we're just doing `app.find(<class name>)` instead of `document.getElementsByClassName`)

Looks very similar to a javascript selenium test - you are able to test the same user flow, the same code, the same code coverage, the only thing that's missing is the actual browser

You can see from the example that we can get a fairly simple example up and running and pretty quickly, but we found we needed to do some workarounds from what we wanted to do. One of the reasons for this is that we found that Enzyme's utilities and selectors are focused around testing via implementation detail.



# Newer Tools & Opportunities: React Testing Library

```
import React from 'react'
import {render, fireEvent, cleanup, waitForElement} from 'react-testing-library'
import 'jest-dom/extend-expect'
import App from 'App.js'

beforeEach(() => {
  store = createStore();
  appointmentTypes = mockLoadAppointmentTypes({ storeId });
  selectedAppointmentType = appointmentTypes[0].id;
  timeSlots = mockLoadAvailableTimeSlots({ storeId, date, selectedAppointmentType });
})

afterEach(cleanup)

test('should navigate to confirmation page and display appointment info after successfully scheduling', async () => {
  mockScheduleAppointmentCall()

  const {getByText} = render(
    <Provider store={store}>
      <App />
    </Provider>
  )

  const appointmentTypeDropdown = await waitForElement(() =>
    getByText('Select Appointment Type')
  )
  fireEvent.change(appointmentTypeDropdown, { target: { value: 'Full Dog Grooming'}})

  const datePicker = await waitForElement(() =>
    getByText('Select Appointment Time')
  )
  const selectedTimeSlot = moment(timeSlots[0].dateTime.toString());
  fireEvent.change(datePicker, { target: value { selectedTimeSlot }})

  fireEvent.click(getByText('Submit'))

  await waitForElement(() =>
    getByText('Your appointment is confirmed!')
  )
  expect(confirmationMessage).toHaveTextContent('Your grooming appointment has been scheduled for November 9th at 10:00 AM')
})
```



## Avalon

You can see from the example that we can get a fairly simple example up and running and pretty quickly, but there are some workarounds required. Why is this? Enzyme's utilities are focuses around testing via implementation detail.

React testing library's whole philosophy is centered around what we're trying to do "While you can follow these guidelines using enzyme itself, enforcing this is harder because of all the extra utilities that enzyme provides (utilities which facilitate testing implementation details). " -> from their docs

Don't need enzyme wait as the library has a build in 'waitForElement' method may be easier to introduce these tests with enzyme since most people are already using enzyme - migrating to react testing library would be a bigger effort

-----

- You can see from the example that we can get a fairly simple example up and running and pretty quickly, but there are some workarounds required. Why is this? Enzyme's utilities are focuses around testing via implementation detail.

- React testing library's whole philosophy is centered around what we're trying to do
- "While you *can* follow these guidelines using enzyme itself, enforcing this is harder because of all the extra utilities that enzyme provides (utilities which facilitate testing implementation details). " -> from their

- docs
- Don't need enzyme wait as the library has a built in 'waitForElement' method
- may be easier to introduce these tests with enzyme since most people are already using enzyme - migrating to react testing library would be a bigger effort

“React attaches an event handler on the document and handles some DOM events via event delegation (events bubbling up from a target to an ancestor). Because of this, your node must be in the document.body for fireEvent to work with React. This is why render appends your container to document.body. This is an alternative to simulating Synthetic React Events via [Simulate](#). The benefit of using fireEvent over Simulate is that you are testing real DOM events instead of Synthetic Events. This aligns better with [the Guiding Principles](#). ([Also Dan Abramov told me to stop use Simulate](#)).”

## **Definition:**

"If a unit test is testing the smallest testable component of application code, then a subcutaneous test is a single workflow which can be identified and tested, of the application code. Subcutaneous testing treats a workflow as a testable unit."

Subcutaneous Testing

Melissa

We've talked about all the great things Subcutaneous testing can do.

Here is the definition again.

This isn't a silver bullet. Testing is only as good as you build it. While it covers a great many things, it has blind spots.

When you know what those are, you can use this as another tool in the testing tool belt.

## What Subcutaneous Will Not Cover

---

- Third Party Integrations (ads, graphics, CMS)
- Application Integrations (if one application references to another application)
- Compatibility
- Information Dependencies (APIs, DBs, Other Apps)

Melissa

Here are some of the things our team realized subcutaneous would not cover.

For our application it wasn't an issue, but if you are dealing with a more complex application or multiple applications, these are definitely things to consider.

Anything that the app might call via a third party would not be covered.

This would include presentation widgets, ads, or graphics, like ones managed by a CMS.

To make the tests self-contained the team mocked the API calls needed to run the tests.

It wasn't necessary to test the content being returned from the APIs, because API tests were covering the JSON format and general content type and content requirements. (ie - the business logic wasn't in the front end of the app)

Proxies, protocols which handle network traffic for various environments, can be verified via API tests on Dark Prod or Prod. Or Monitoring. Generally, if there is something wrong with any proxies everyone knows it immediately, and because the tests are not being run through actual HTTP or HTTPS protocols, it would have no idea if a proxy wasn't working.

Compatibility testing becomes a separate entity from the functional testing. With advances in AI and Machine learning, this could take on a whole new twist!

Interactions with other applications or various dependencies, like login information, could be handled by a very light weight webdriver framework running an integration check.

The test could be as simple as starting on one web page and clicking a button and verifying it moved to another web page.

What could have been a complex script is reduced to three or four lines of code because functionality no longer needs to be tested at the GUI level.

Even then, it might not be worth the time spent writing, maintaining, or checking a test for such a small application junction if it's already pretty stable.

Teams would need to look at what would be necessary on a case-by-case basis.

This allows teams, especially testers, to shift their focus away from the constant demand of the functional problems to broader application concerns.



## CHANGING FOCUS ON TESTING

*"If you focus on the smallest details you never get the big picture right."*

- Leroy Hood

**TESTING** *SHIFTS* Away from  
FUNCTIONAL Coverage

**FOCUS** ON APPLICATION  
HEALTH.

**FOCUS** ON INTEGRATION  
STABILITY.

**FOCUS** ON BROWSER  
COMPATIBILITY.

**FOCUS** ON ACCESSIBILITY

Melissa

When you stop worrying that an application is functional because the tests are helping the developers make it functional, testers can start looking at other things involving the application.

Monitoring, Observation, Analytics all become more valuable and things that start to drive testing rather than being by-products of the application.

Testing can shift to making sure the cracks between applications are addressed and helping build and maintain contract tests and integration tests. This becomes really important a micro architecture setup.

Testing different browsers and devices with some exploratory testing or simple checks.

I'm focused on visual anomalies that cropped up. There are tools for these kinds of things now too.

I didn't have to execute the whole workflow. I could skip along and check what I need to.

I applied more focus to accessibility testing for mobile and web.

Tools have made this easier, but with functional testing covered, I actually had time to use them instead of worrying about whether the app was broken.

The team experienced all of these shifts. It was amazing to not feel bogged down with constantly checking the functionality.

Our team had reliable feedback in a timely manner which let us create a pretty robust application without the undue weight of constantly checking a GUI testing framework or adding to one.

## Is Subcutaneous Testing For You?

---

- Does your app have a UI?
- Do you have strong backend contract/integration test coverage?
- Can you identify a continuous workflow contained in app code?

Melissa

Having talked about all those amazing experiences, it's only fair to bring up things you should have to make all this wonderful magic happen.

Not all applications have a UI. Your UI might only be a text bar and a button integrated into another app.

Backend testing which includes contract, integration, and database testing  
These become invaluable with Subcutaneous testing as you are focused on functionality, not whether a call is being made to an API.

If you are working with JavaScript, I imagine this paradigm will continue forward.

If you have to go through several apps to complete workflow, this might only grant partial benefits, but even that might be enough to make it worth the effort.





# Thank You!

Melissa Eaden:

[melthetester@gmail.com](mailto:melthetester@gmail.com)

Avalon McRae:

[avalonmcrae@gmail.com](mailto:avalonmcrae@gmail.com)

Thank you for your time!

If we have time, we are willing to answer questions.

---

## Beneficial Info

---

- Mountebank: <http://www.mbtest.org/>
- Enzyme-wait: <https://github.com/etiennedi/enzyme-wait>
- Containers: <https://medium.com/@learnreact/container-components-c0e67432e005>
- Enzyme rendering methods:  
<https://medium.com/@Yohanna/difference-between-enzymes-rendering-methods-f82108f49084>
- 

Add link to melissa's article after publication