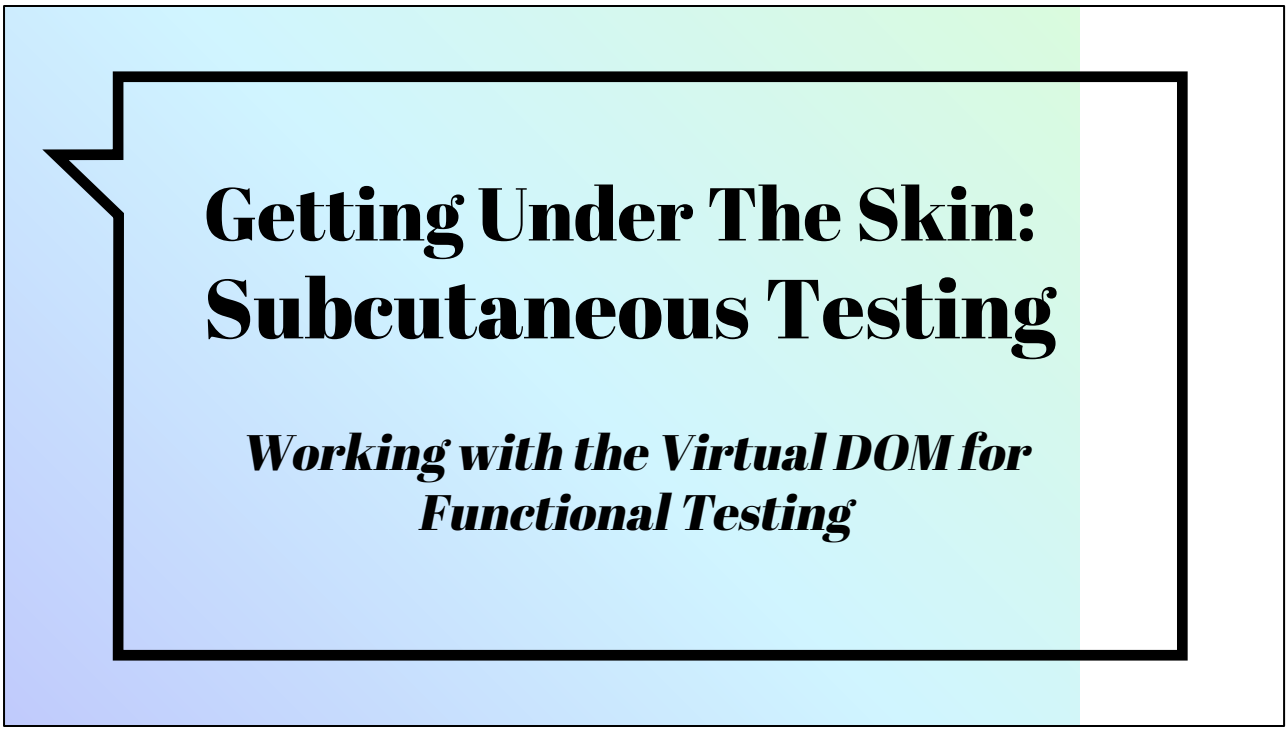
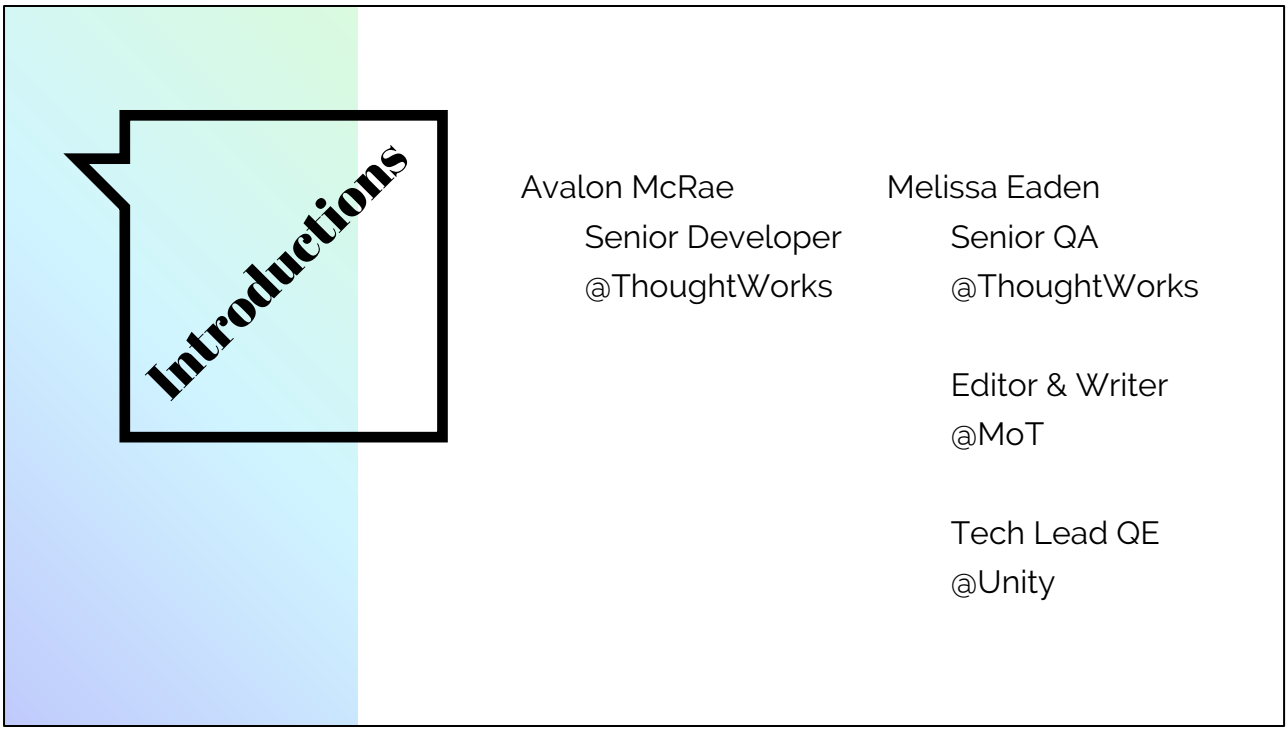


Is everyone having a good day so far?

Thank you for being here!



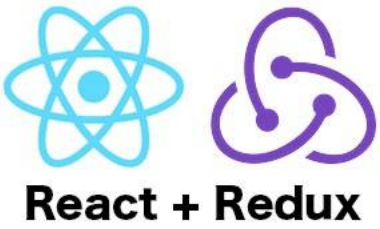
Avalon & Melissa

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

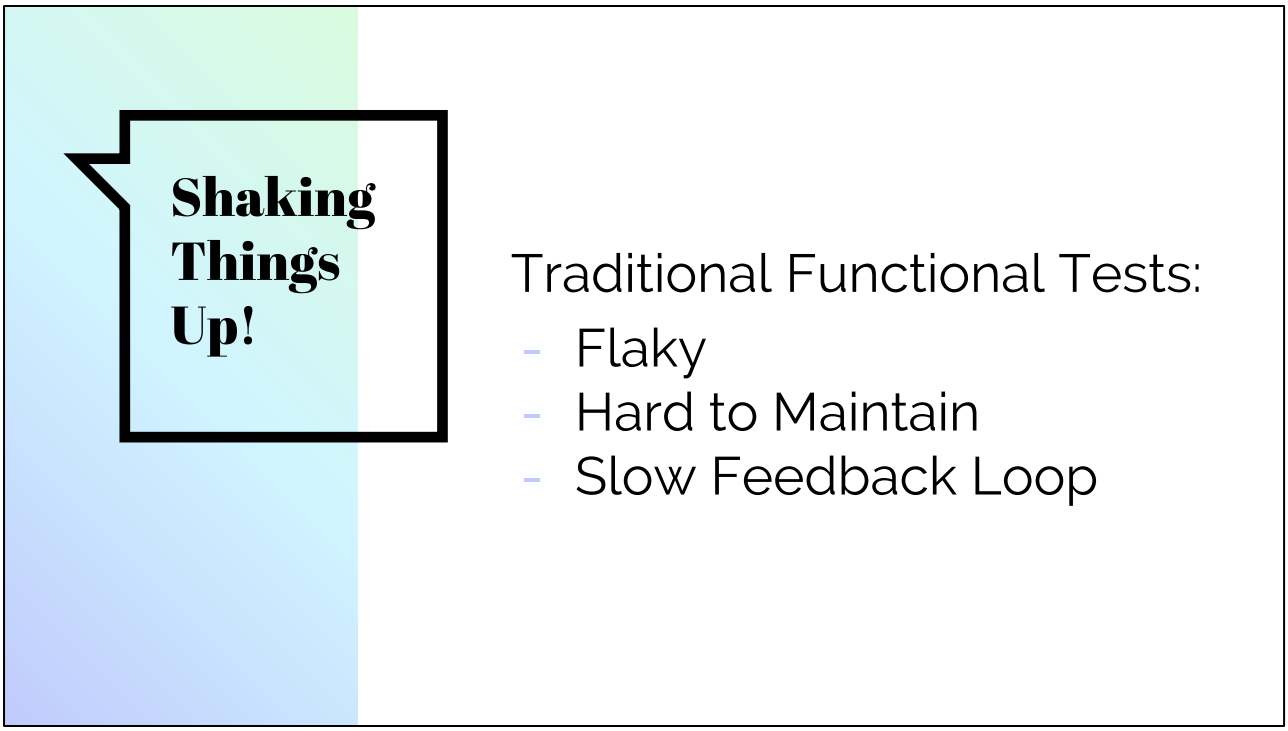


Melissa

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.



Melissa

It had the problems a lot of traditional functional testing frameworks have.

It was flaky 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.

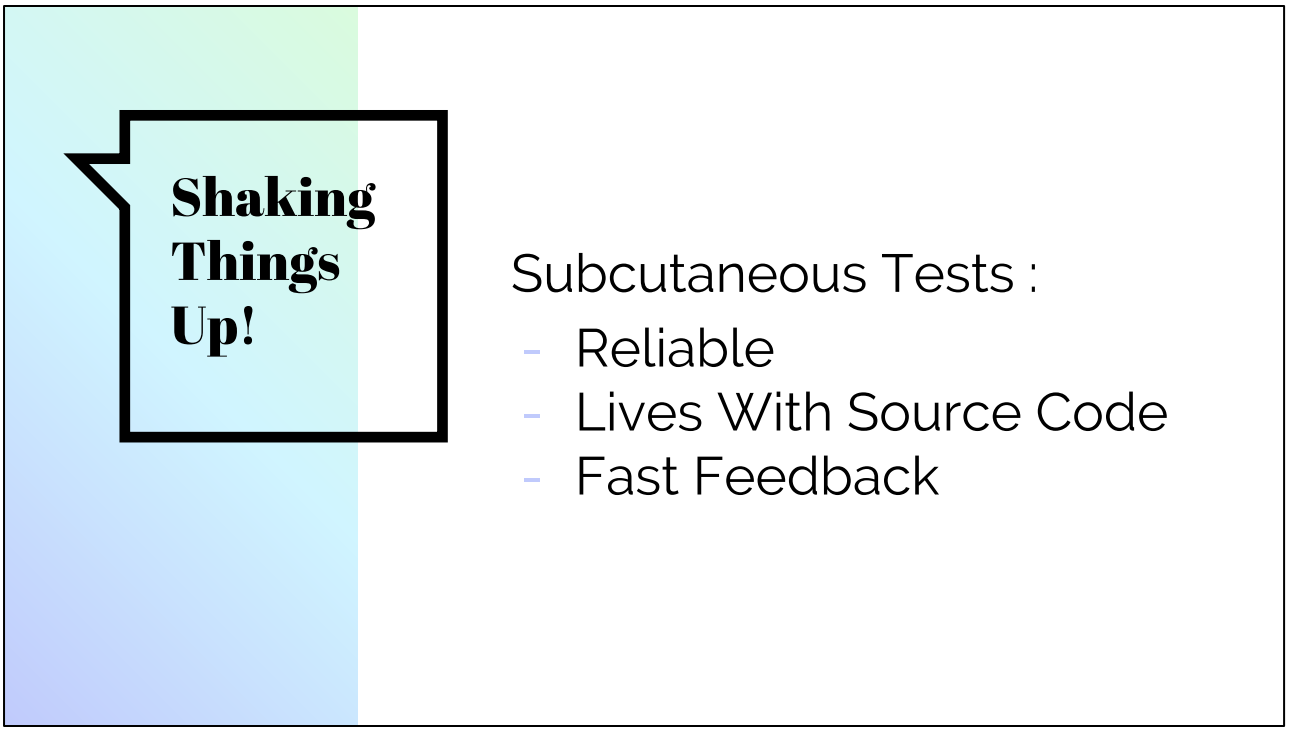
It was hard to maintain (that's from several folks perspective who worked for the client, not just our team), it was a slow feedback loop with very little valuable information coming back to the teams involved.

Most teams didn't rely on them for information because of this.

The footprint of the functional tests were extremely large and mostly unreliable.

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.



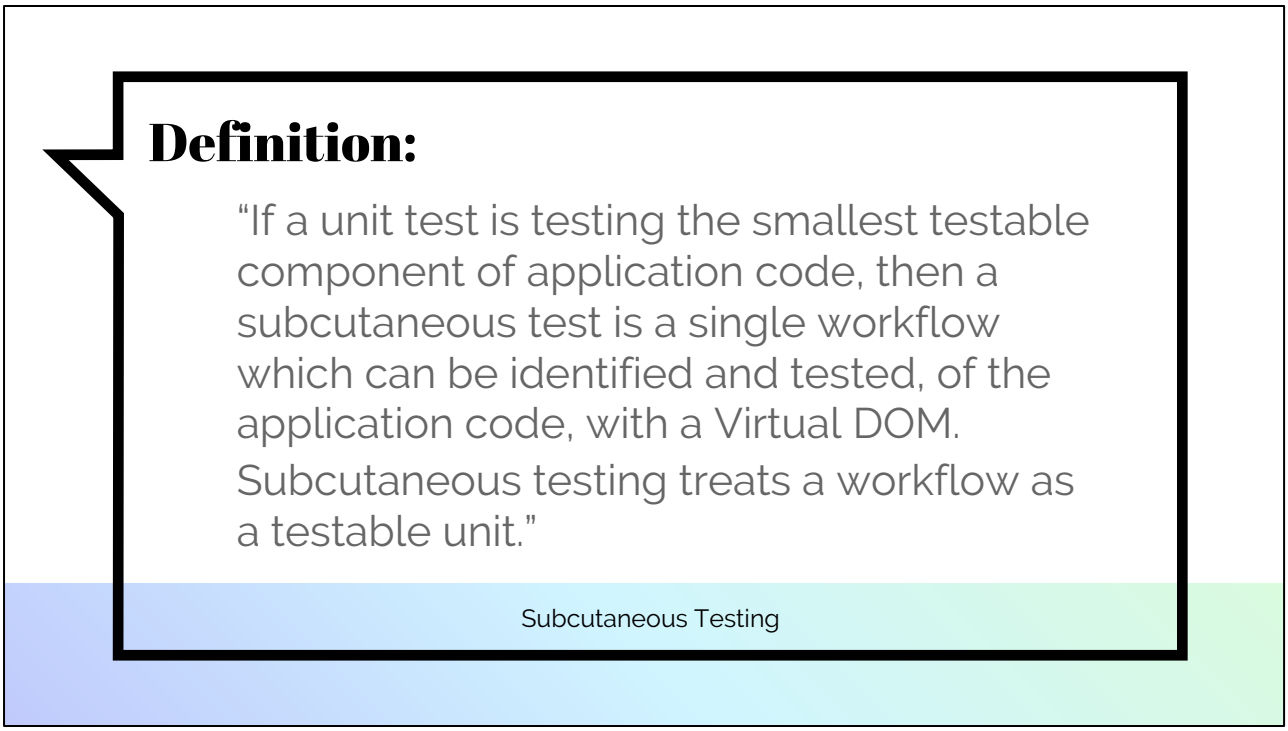
Avalon

Reliable -> don't experience selector flakiness of "element not found", ids or class names changing etc.

Lives with source code -> 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)

So now that we've talked about *why* we want to do these types of tests, let's talk about what these tests would actually look like...



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 and came up with the one you are reading now.

The most important parts of this definition I would point out are "with a Virtual DOM" and "Subcutaneous testing treats a workflow as a testable unit."

An Example: Happy Doggy Grooming

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

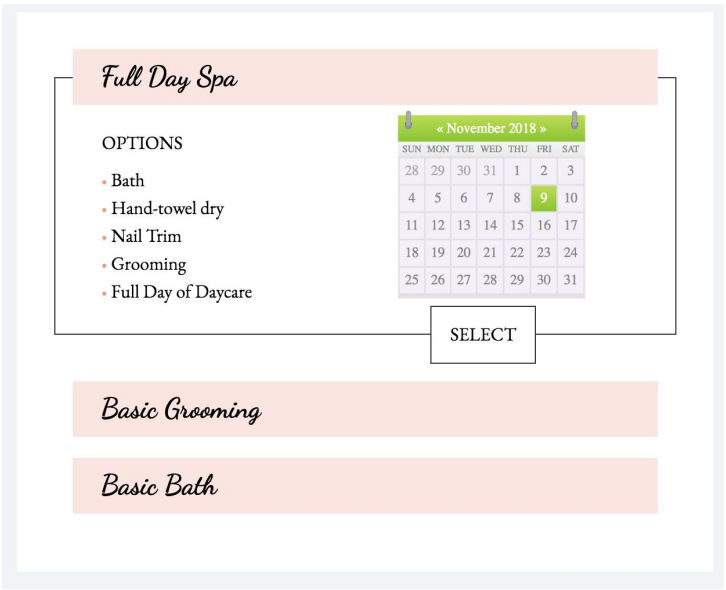
This one is a theoretical application mocked up to give you a sense of what the team was trying to accomplish.

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



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

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.

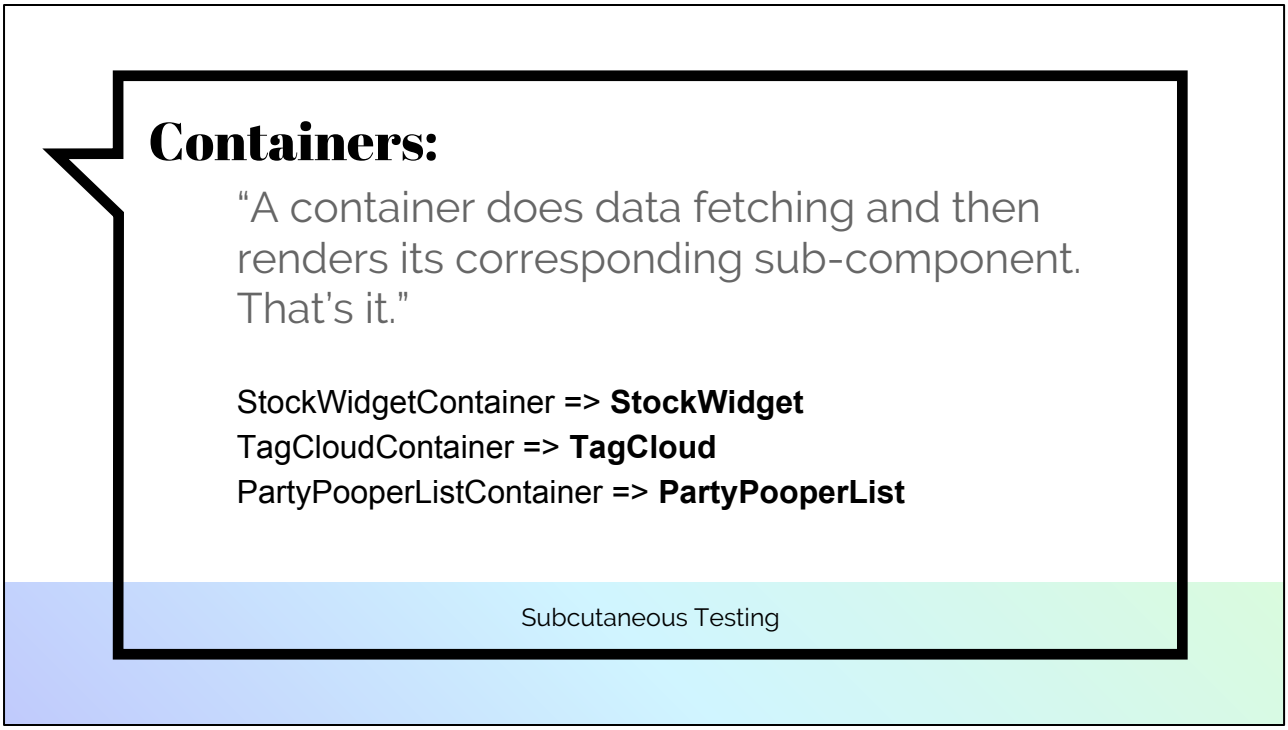
I've called a workflow a testable unit for a reason.

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 testing a behavior inside of a container. Only that container is mounted

in the Virtual DOM and tested.

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



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.

An Example: Happy Doggy Grooming

Melissa

An example of this is validation of error messages for the form. This is very easily handled by unit tests because the container/component combination lets us test the responses very easily.

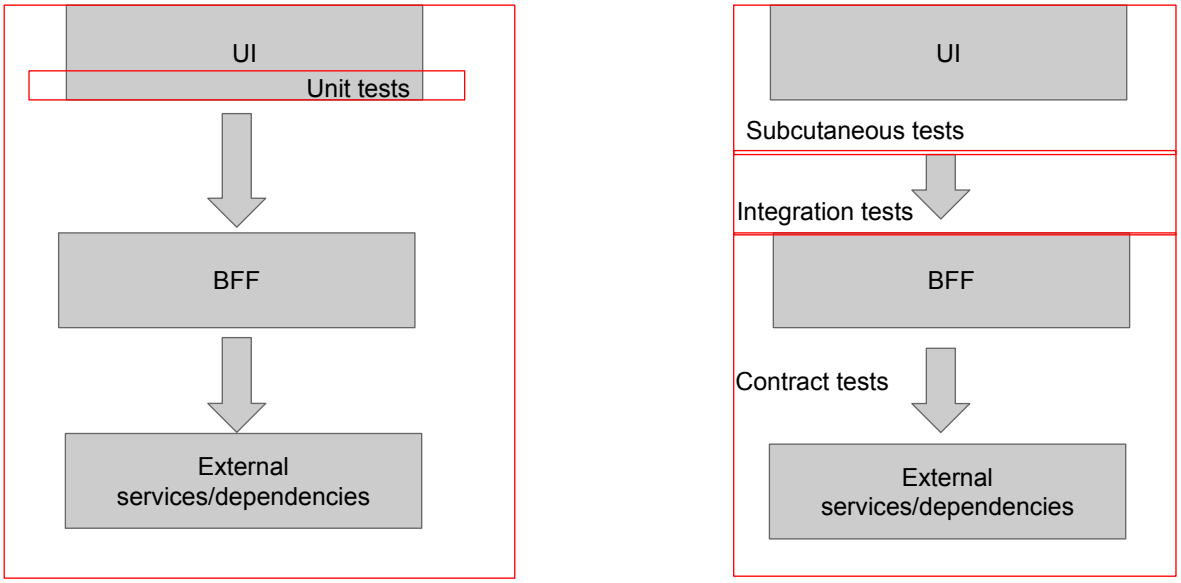
For subQ tests interactions between containers. The workflow is unit, which requires the that entire application is mounted in the Virtual DOM.

For Example, when you select an appointment option, the date picker is there, you click select, then verify the information then click submit.

With the Virtual DOM, tests can be focused on functional workflows. This allowed the team to separate functionality from compatibility issues very quickly.

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 is an example architecture diagram showing the newer sort of testing approach that Melissa and I have tried on our last couple of projects. 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.
Mention unit tests don't need to delve into this too much.

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.

Quick Primer on React

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called reconciliation.

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 react uses something called the virtual DOM. All of the code to create the UI of the application

DOM stands for *Document Object Model*. *Elements* of HTML become *nodes* in the DOM.

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

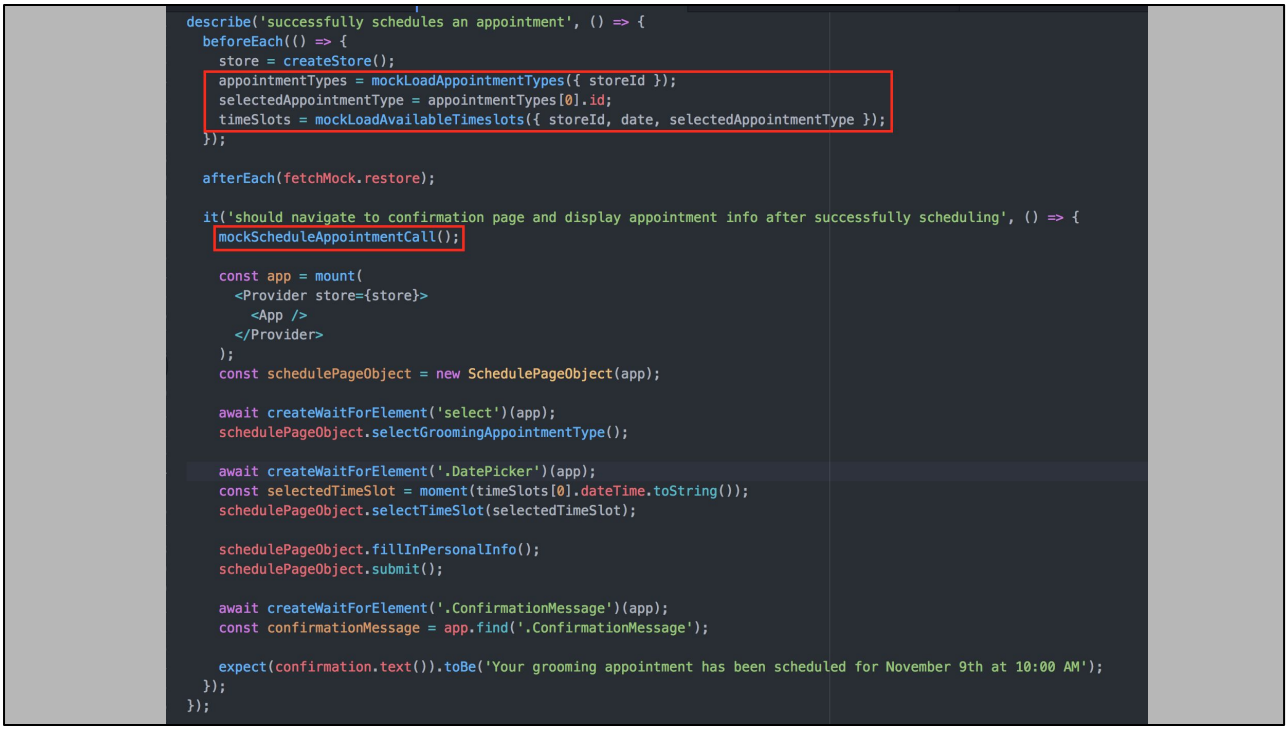
Render a React element into the DOM

```
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
If you do not want to run your tests inside of a browser, the recommended approach to using `mount` is to depend on a library called jsdom which is essentially a headless browser implemented completely in JS.

We are using enzyme (a testing library) to mount our entire app via jsdom, which is essentially a headless browser implemented completely in javascript

Using the same testing framework as the unit tests to create end-to-end (or workflow) tests which are testable as a unit



```
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');
  });
});
```

This it 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

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';
const apiResponse = {
  createdDateTime: '2018-08-01 11:24:29',
  firstName: 'Jill',
  lastName: 'Johnson',
  confirmationCode: 'ABC321',
  phone: '1233334444',
  email: 'jilljohnson@email.com',
  appointmentReason: 'Grooming',
  appointmentReasonId: 1,
  storeId: '881',
  appointmentDateTime: '2018-08-03 10:00:00',
  appointmentId: '5de33b0d-92b6-4e52-87d5-31f357cab055'
};

export default (response = apiResponse) => {
  fetchMock.post('/schedule', response);
  return response;
}
```

If showing a builder pattern speak to flexibility of mocks

Page Object Model

```
import { DatePicker } from 'react-kuic';
import BasicInfoForm from '../BasicInfoForm/BasicInfoFormContainer';
import BasicInfoFormPageObject from './basicInfoFormPageObject';

class SchedulerPageObject {
  constructor(container) {
    this.container = container;
  }

  selectBox() {
    return this.container.find('select');
  }

  selectedReasonId() {
    return this.selectBox().props().value;
  }

  appointmentComments() {
    return this.container.find('.PleaseSpecifyFieldContainer').find('input');
  }

  updateAppointmentComments(value) {
    this.container.find('.PleaseSpecifyFieldContainer').find('input')
      .simulate('change', { target: { value } });
  }

  datePicker() {
    return this.container.find(DatePicker);
  }

  basicInfoForm() {
    return new BasicInfoFormPageObject(this.container.find(BasicInfoForm));
  }

  selectReasonById(reasonId) {
    this.selectBox().props().onChange({ target: { value: reasonId } });
  }
```

```
it('should see success message after successful appointment subm
  mockScheduleAppointment();

  const app = mount(
    <Provider store={ store }>
      <Layout params={ { businessName, locationSourceSystemId } }
    </Provider>
  );

  const schedulerPageObject = new SchedulerPageObject(app);

  await createWaitForElement('select')(app);
  schedulerPageObject.selectReasonById(reasonId);

  await createWaitForElement(DatePicker)(app);
  const selectedSlot = moment(slots[0].dateTime.toString());
  schedulerPageObject.selectTimeSlotByDateTime(selectedSlot);

  schedulerPageObject.fillInBasicInfo();
  schedulerPageObject.submitForm();

  await createWaitForElement('.Success-Message')(app);
  const success = app.find('.Success-Message');

  expect(success.text()).toBe('Your appointment has been success
});
```

a model used a lot in traditional functional tests, but one big difference here is that we can reuse these methods in our unit tests if we want to.

What types of things can we select?

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');
  });
});
```

NPM package that we used in conjunction with javascript's async/await functionality to deal with promises in a more reasonable way, a little clear then setTimeout

Similar pattern to selenium tests using waitForElement

Example Test Case

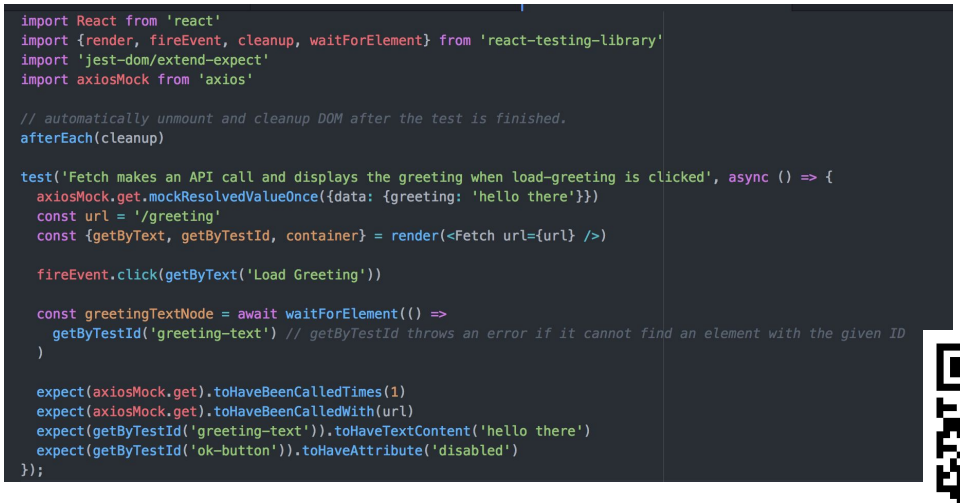
```
describe('successfully schedules an appointment', () => {
  beforeEach(() => {
    store = createStore();
    appointmentTypes = mockLoadAppointmentTypes({ storeId });
    selectedAppointmentType = appointmentTypes[0].id;
    timeSlots = mockLoadAvailableTimeslots({ storeId, date, selectedAppointmentType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

  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(confirmation.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, but again we are leveraging jsdom to…. (fill in more specfiics)

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 axiosMock from 'axios'

// automatically unmount and cleanup DOM after the test is finished.
afterEach(cleanup)

test('Fetch makes an API call and displays the greeting when load-greeting is clicked', async () => {
  axiosMock.get.mockResolvedValueOnce({data: {greeting: 'hello there'}})
  const url = '/greeting'
  const {getByText, getByTestId, container} = render(<Fetch url={url} />)

  fireEvent.click(getByText('Load Greeting'))

  const greetingTextNode = await waitForElement(() =>
    getByTestId('greeting-text') // getByTestId throws an error if it cannot find an element with the given ID
  )

  expect(axiosMock.get).toHaveBeenCalledTimes(1)
  expect(axiosMock.get).toHaveBeenCalledWith(url)
  expect(getByTestId('greeting-text')).toHaveTextContent('hello there')
  expect(getByTestId('ok-button')).toHaveAttribute('disabled')
});
```

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.

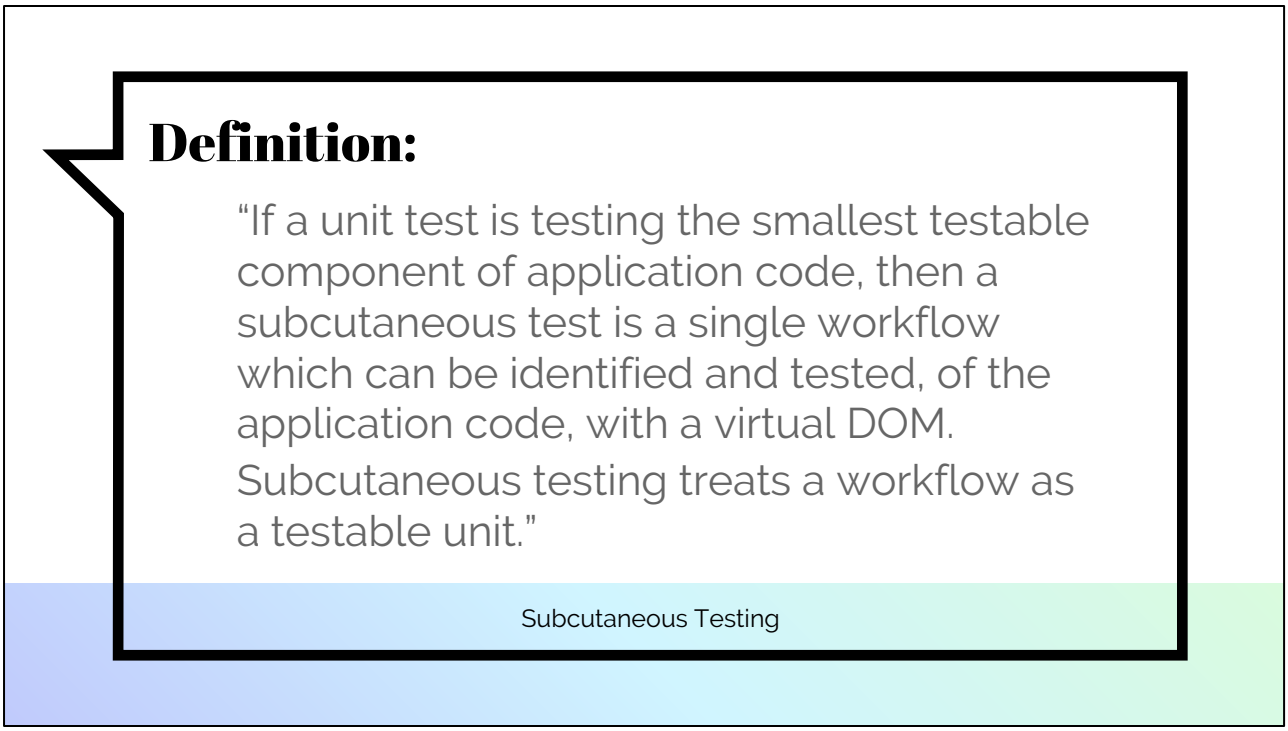
React testing library's whole philosophy is centered around what we're trying to do
"While you can follow these guidelines using enzyme itself, enforcing this is harder because of all the extra utilities that enzyme provides (utilities which facilitate testing implementation details). " -> from their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

Better segue to next slide!



Melissa

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

Here is the definition again.

This kind of testing 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.

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.

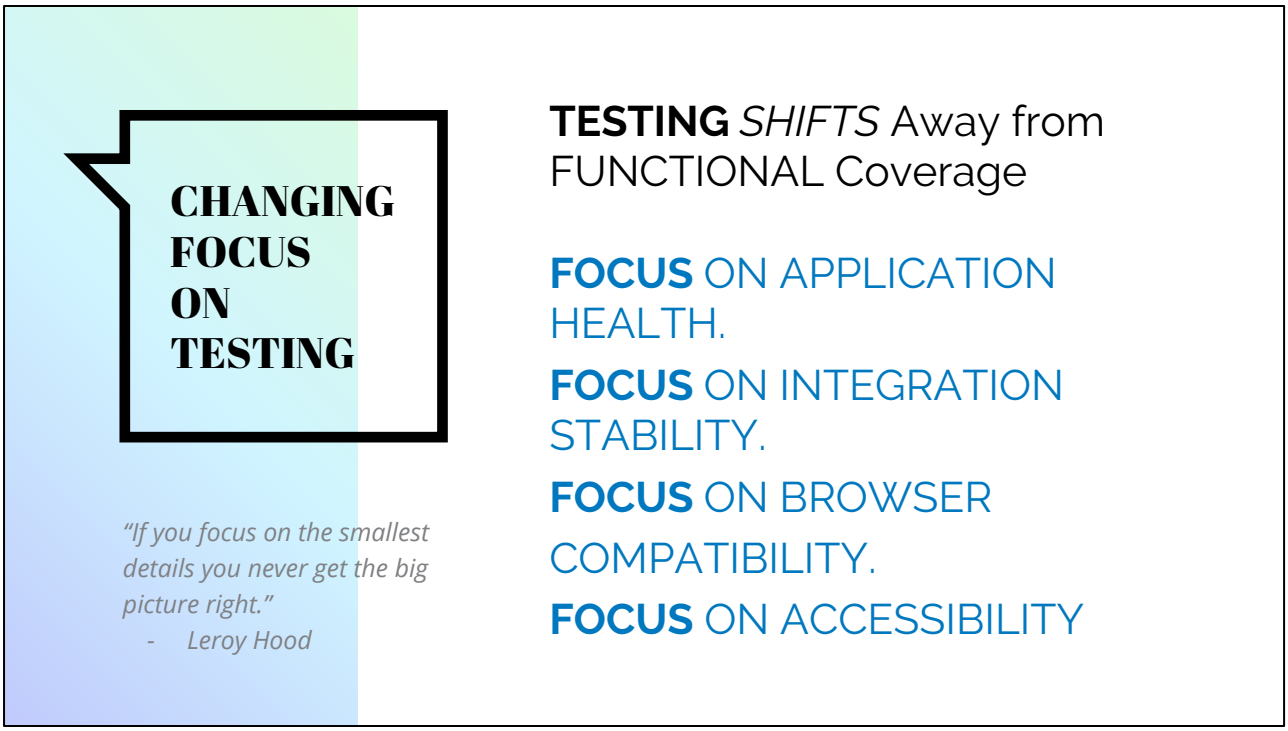
The test could be as simple as starting on one web page and clicking a button and verifying it moved to another web page.

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.



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?
- Does your code framework or libraries have a Virtual DOM?
- 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.

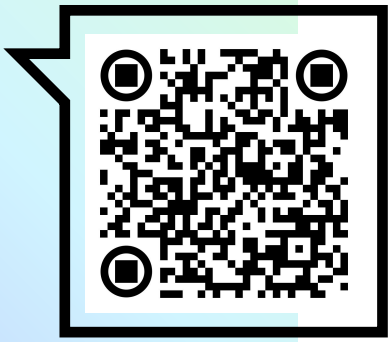
Not all applications have a UI. Your UI might only be a text bar and a button integrated into another app.

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.

The Virtual DOM is a must. React and Vue are the only JS libraries/frameworks which I know have a Virtual DOM.

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

Avalon McRae:
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

Add link to melissa's article after publication