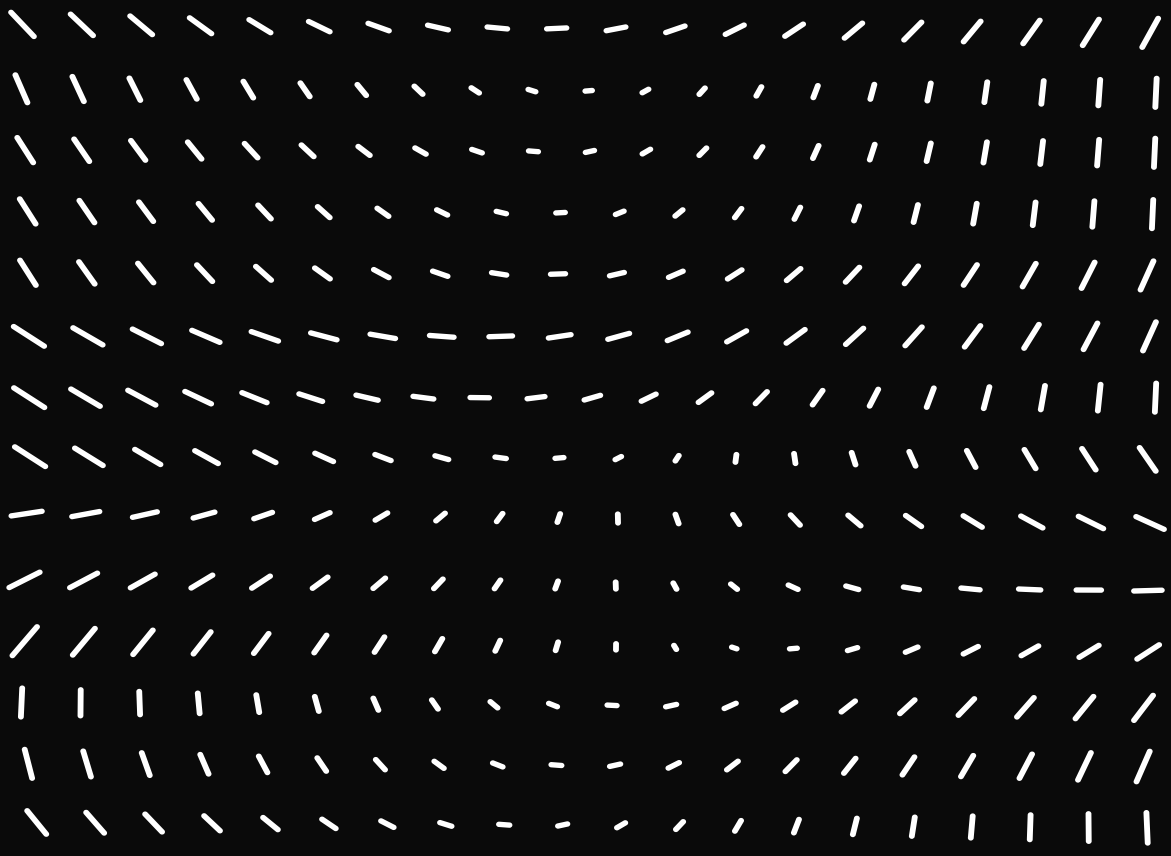
React Testing Crash Course w/ Traversy Media



Mitchell Intro

1. Learning outcomes
   a. Goals
      i. test front apps efficiently and effectively
      ii. Use modern testing tools
   b. First 1s) big ideas
      i. Why you should test
      ii. What to test
      iii. Unit tests
      iv. Integration tests
      v. End-to-end (e2e) tests

} React testing library + Cypress

Why you should test

1. Overview
   a. Goal: check whether an application behaves as expected
   b. Safeguards
      i. against bugs and regressions
      ii. Courage for new features, refactoring, dependency updates
      iii. Quick, automated check for app functionality
      iv. Supplements manual testing

What you should test

1. Prioritization / Order (ND "Golden Rule")
   a. High value features ← this order mostly works and reflects common approaches in testing community
      i. bring most value or $$$ to an app
      ii. e.g. [view + purchase products on AMZ, playing music on Spotify, user can read + send email w/ Gmail]
      iii. focus on happy paths - sunny day testing where you test for functionality required to pass the test (just get the thing to work)
   b. Edge cases in high-value features
      i. focus on sad paths - testing for unexpected or unwanted behaviors
      ii. e.g. [max order = 100; user attempts 1000]
   c. Things that break easily
      i. list of features you notice break every now and then during development
      ii. sensitive features, functions, modules, etc.
   d. Basic React Component Testing
      * focus on most to least important *

Ex 1.) Component precedence (simple)
1. Components in high-value features
2. Components that are heavily reused

i. User interactions // onClick, onChange

ii. Conditional rendering // state changes e.g. (props, useState)
↳ causes UI to rerender

iii. Utils & Hooks // functions used throughout various components; custom Hooks

e. Anti-pattern

i. DON'T test implementation details

↳ not reliable or representative of true user behavior

* Breakdown of testing types *

Unit tests

→ small, chunks of self-contained code

→ e.g. in functional programming; test functions

Integration testing

→ whether multiple units in your apps are working correctly together

→ combining unit tests into a larger test

End-to-end test

→ test from one end (front end) to the other (back end)

→ recording user interactions in the browser

* Best practice: bottom-up because "bottom-of-list" tests are more concise and easier to understand *

→ lots of easy, concise tests building to a few "harder", more expensive tests.

Unit tests

* React Testing Library comes out-of the-box w/ Create-react-app *

1. Testing a component

a. set-up

i. import { render, screen } from "@testing-library/react"

ii. import the component you're testing

iii. also remember to create the test file as close to the component as possible

b. component that takes props

i. pass example props in the render function component argument depending on what you're testing

Eg 2) passing props to unit

```
test('component does a thing', () => {
  render(<MyComponent someProp={{}}/>);
```

* If you don't make any assertions in your test, the test will automatically pass *

2. screen.debug();

a. RTL will console.log the...

i. HTML of your app's body

3. screen.getByRole();

a. searches app for roles based on arg (string)

i. e.g. (header, button, etc.)

4. screen.getByRole(/button/i, {name: /pay/i})

a. testing for button enabled

fig 3.) toBeEnabled()

```
// Assert expected behavior
expect(screen.getByRole(/button/i, {name: /pay/i})).toBeEnabled
```

b. Interesting behavior

* A component's expected behavior may occur after a brief period of time e.g. (HTTP, Promises, library/module/package behaviors)

Use async/await to test for something to return before testing an assertion

↳ use findByBlahBlah since getBy functions throw errors here

* Best practice *

If you're making an assertion...

First do it the opposite way and then make sure the test passes

For example

→ testing a button is disabled...

→ test if it's enabled first for faulty behavior, then write a test for the desired behavior (disabled)

.toBeEnabled() → .toBeDisabled()

6. RTL Recommended testing priority

a. getByRole()

i. can be used to query every element exposed in accessibility tree

ii. name option - filter returned elements by accessible name

* Best Practice: this is top preference for just about anything *

↳ unless your UI is inaccessible

fig 4.) getByRole example

getByRole('button', {name: /submit/i})

b. form elements → getByLabelText

* Honestly, just navigate to the RTL docs

→ Core API > About Queries > Queries accessible...

7. Mimicking user interaction

a. Set-up

i. import userEvent from "@.../user-event"

b. Act - form filled, test button enabled

i. userEvent.type(screen.getByPlaceholderText(/i), "")

↑ type input value

c. Assert - check button w/

i. expect(screen.getByRole('button', {name: /i})).toBe...();

ii. ...Disabled → ...Enabled

* Key idea: writing tests should be easy; about translating user interactions → code