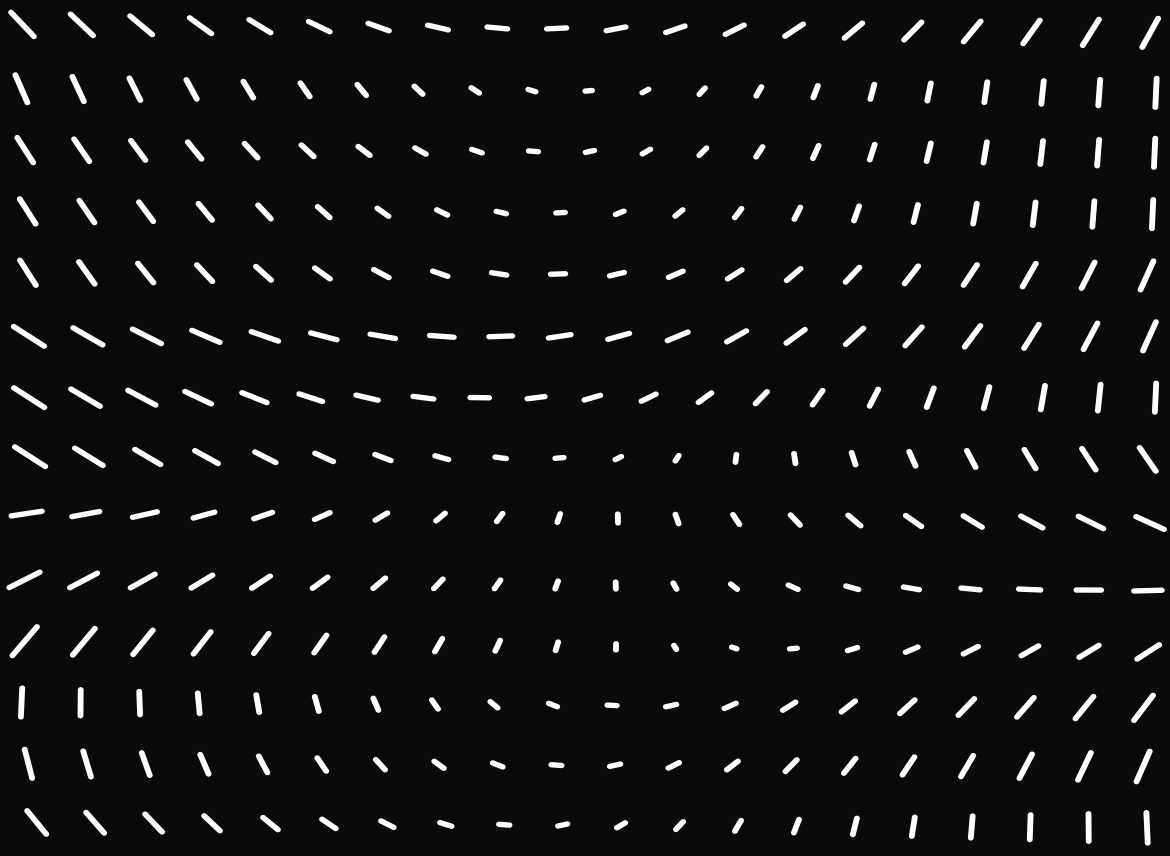
Unit, integration, and e2e testing w/ Max Schwarz



Automated Testing in React

Introduction

1. What is automated testing?
   a. Code that tests our code
      i. Main but important for writing modern applications

** Testing is worthy of a dedicated course all on its own, good to have an intro-level understanding before getting the first job

2. In this course
   a. Explore
      i. What is testing?
      ii. How/why do we do it?
      iii. Unit tests
      iv. Testing React components & building blocks

What & Why?

1. Manual Testing
   a. How we write code to...
      i. implement a feature
      ii. fix a certain issue
      iii. preview/test app in the browser
   b. When?
      i. All the time throughout the SDLC
   c. Why
      i. Enables us to spot check against what the user actually sees
   d. Drawbacks
      i. Error prone
      ii. Hard to test all scenarios
      iii. Breaking changes might slip through to production

      * catch errors way earlier
      * write + ship better code and apps

2. Automated Testing
   a. NOT
      i. replacement for manual testing
   b. How
      i. write "extra" or supplementary code that runs and tests your other code
      ii. automated testing is "code that tests your code"
   c. Pros
      i. since it's automated, can always test even updated code vs. manual tests
      ii. doesn't cost a lot of time
   d. How
      i. test building blocks (units) first
      ii. then test them all together
      iii. Can do this in an automated way whenever you make changes to your code vs. only parts of your app from time to time
   e. Trade-offs
      i. technical but provides lots of coverage

Understanding Different Kinds of Tests

* IMPORTANT: there are different kinds of automated tests w/ their own unique purpose *

1. Three main test categories
   a. Unit test
      i. smallest possible "units" of your app
      ii. Unit - individual building block; functions, components (in isolation)
      iii. Projects may contain dozens to hundreds of unit tests
      iv. Good coverage means testing all the functions and components in your application

* Unit test are considered the most common and important types of test since, if you test all individual units on themselves, then the whole application has better odds of working *

2. Integration tests
   a. What?
      i. verify that units are working together
      ii. test the combination of units
      iii. Example: multiple react components working together
   b. How?
      i. Projects typically only contain 1-2 (can be more)
      ii. The line between unit and integration tests (blurry) e.g. (testing a component that uses another component)

* Integration tests are also generally important, but we use fewer of these than unit tests *

3. End-to-end (e2e) tests
   a. What?
      i. test entire workflows / scenarios
      ii. Example: logging a user in and navigating to a certain page
      iii. aim to reproduce human behavior
      iv. behavioral testing
   b. How?
      i. only a few
      ii. unit + integration tests provide most coverage and reduce need for extensive e2e testing
      iii. e2e is slower and less focused
      iv. harder to consider all possible scenarios when looking at app as a whole

* Important, but partially covered by manual, unit, and integration tests, so don't need many *

What to Test & How

1. Two questions to start
   a. What to test
   b. How to test // What type of code in test?
2. What?
   a. Unit test
      i. the smallest building blocks of your app
      ii. small, focused test that focus on one thing each
      iii. failing for a clear reason → how?
      iv. ANTI-PATTERN: a few (LARGE) tests
3. How?
   a. Test...
      i. success and error cases
      ii. rare (but possible) results

Understanding the technical setup

* Where do we write, test, and execute this code? *
* Need a tool to run tests and assert results * (success / fail)
* In React, need a tool for simulating (rendering) our React app / components *
  ↳ automated tests to interact w/ them
  ↳ simulate the browser

Jest: test runner for unit + int. tests

React Testing Library: simulate / render components

* both ready w/ create-react-app *

Running a first test

1. Files
   a. Test file
      {module}.{test type}.test.js
   b. setupTest.js
      i. set-up work; doesn't need changes
2. Test file
   a. Contains test code
      i. can write tests from scratch
      ii. Convention: name test files like component files w/ .test.js
   b. Test function
      i. Two args: (description, callback (anon fct))
      ii. description – up to you, helps identify this test in the output esp. if tests > 1
      ii. callback – actual testing code
   c. callback can...
      i. render a component w/ .render function imported from "@testing-library/react"
      ii. screen – virtual screen for simulated browser into which a component gets rendered
      iii. identify a rendered element w/ the screen.getByText(/learn react/i) method
           eg/ by RegEx text
      iv. check if in document w/ expect(linkElement) and the .toBeInTheDocument() method

Sample test code

```
1  import { render, screen } from '@testing-library/react';
2  import App from './App';
3
4  test('renders learn react link', () => {
5    render(<App />);
6    const linkElement = screen.getByText(/learn react/i);
7    expect(linkElement).toBeInTheDocument();
8  });
```

3. How to run tests
   a. The script
      i. npm test - runs our automated test
      ii. gives us options in the terminal
   b. Output
      i. test suites pass / fail
      ii. all test + status
      iii. description text
      iv. By default, watches and re-runs tests

Side note: seems like React wants us to export modules using ESM vs. Jest wanting CJS, so instead of using an "experimental" Jest config or a hybrid strategy, I can import and test the component and its various functions in a single "".unit.test.js file

Writing our first test

* Convention: write tests as close as possible to the thing you want to test *

fig 1.) Sample file structure

```
src
├─ components
│  ├─ Greeting.js
│  └─ Greeting.unit.test.js
```

1. Writing the test
   a. test( ) function
      i. globally available, so no need to import
   b. arg 1 is description
      i. briefly describe what it does
      ii. Example: test('renders hello world', () => {})
   c. arg 2 - anonymous function
      i. typically do 3 things.. "The Three As"
2. The Three As
   a. Arrange
      i. set up the test data
      ii. test conditions
      iii. test environment
      * render the component you want to test
      * additional set-up possibly required
   b. Act
      i. do the thing you're testing for (e.g. Simulate a button click)
   c. Assert
      i. Compare execution results w/ expected results

3. Implementing arrange, act, assert pattern
   a. Assert
      i. render(<MyComponent />);
   b. Act
      i. no example
   c. Assert
      i. screen: gives access to virtual DOM
      ii. can use screen to find various elements
4. screen. functions
   a. three types available
      i. get
      ii. find
      iii. query
      * main diff:
        - when they throw errors
        - if they return promises or not
   b. get
      i. throws error if function not found
      ii. query functions won't do this
      iii. find functions return a promise
        * works if element is eventually on screen *
5. screen.getByText( );
   a. Args
      i. arg1 - RegEx or hard coded string
      ii. arg2 - config object

Example: screen.getByText('hello', {exact: true})

if false casing won't matter and will match substrings

   b. getByText output
      i. returns an element
      ii. throws an error if n/a
6. expect( ) // globally available function
   a. Arg
      i. can pass testing result value
      * can be anything eg. (Number, String, DOM node) *
   b. Matchers
      i. expect( ).not.someMatcher
         not: check for opposites
      ii. expect( ).toBeInDocument( );
         checks if argument in document
   c. Choosing the right function
      i. Example: const helloWorld = screen.queryByText('Hello World')
         expect(helloWorld).not.toBeInDocument( );

Grouping Tests Together w/ Test Suites

1. Test suites vs tests
   a. As your application grows...
      i. you'll have dozens or even thousands of tests
      ii. test suites enable you to group and organize your different tests
   b. Organization logic
      i. suite for all units in a component
      ii. suite for all tests belonging to one feature
2. Creating a testing suite
   a. The global `describe()` function
      i. two args
      ii. globally available
   b. arg 1 - description e.g. (category to which your tests belong) ; i.e. (Greeting component)
   c. arg 2 - anonymous function

Testing User Interaction & State

1. Testing state change on user interaction
   a. Testing all possible scenarios
      i. State before interaction
      ii. State after interaction

* Test all possible scenarios *

```jsx
import { useState } from 'react';

const Greeting = () => {
  const [changedText, setChangedText] = useState(false);

  const changeTextHandler = () => {
    setChangedText(true);
  };

  return (
    <div>
      <h2>Hello World!</h2>
      {!changedText && <p>It's good to see you!</p>}
      {changedText && <p>Changed!</p>}
      <button onClick={changeTextHandler}>Change Text!</button>
    </div>
  );
};
```

Fig 2: Simple component example

* Best practice: tests read like sentences; Describe for test suite sets the sentence up and test descriptions complete them *

Example: setting up a test suite to test fig2 state.

```
import { render, screen } from '@testing-library/react';
import Greeting from './Greeting';

describe('Greeting component', () => {
  test('renders good to see you, if button NOT clicked', () => {
    // Arrange
    render(<Greeting />);
    // Act - n/a
    // Assert
    const paragraphEl = screen.getByText(/good to see you/i);
  });
});
```

test 1

```
test('renders "Changed!" if button was clicked', () => {
  // Arrange
  render(<Greeting />);

  // Act - req: import userEvent from "@testing-library/user-event";
  const buttonEl = screen.getByRole('button');
  userEvent.click(buttonEl);

  // Assert
  const outputElement = screen.getByText(/changed!/i);
  expect(outputElement).toBeInTheDocument();
});
```

* What if you introduce an error where you forgot to render the paragraph conditionally? *

```jsx
import { useState } from 'react';

const Greeting = () => {
  const [changedText, setChangedText] = useState(false);

  const changeTextHandler = () => {
    setChangedText(true);
  };

  return (
    <div>
      <h2>Hello World!</h2>
      {/*!changedText &&*/ <p>It's good to see you!</p>
      {changedText && <p>Changed!</p>}
      <button onClick={changeTextHandler}>Change Text!</button>
    </div>
  );
};
```

shouldn't be visible if click button

→ test won't spot this as of now
→ test for click button & paragraph invisible

```
test('hides "It's good to see you" paragraph', () => {
  // Arrange
  render(<Greeting />);

  // Act
  const buttonEl = screen.getByText(/change/i);
  userEvent.click(buttonEl);

  // Assert
  const outputElement = screen.queryByText(/good to see you/i);
  expect(outputElement).toBeNull();
});
```

retrieve null value if not present

Testing connected components

(pg 3)

App.js | Greeting.js | Output.js | Greeting.test.js | {} package

```
6   const [changedText, setChangedText] = useState(false);
7
8   const changeTextHandler = () => {
9     setChangedText(true);
10  };
11
12  return (
13    <div>
14      <h2>Hello World!</h2>
15      {!changedText && <Output>It's good to see you!</Output>}
16      {changedText && <Output>Changed!</Output>}
17      <button onClick={changeTextHandler}>Change Text!</button>
18    </div>
19  );
20 };
21
```

TERMINAL | PROBLEMS | OUTPUT | 1: node

```
PASS src/components/Greeting.test.js
  Greeting component
    ✓ renders "Hello World" as a text (28 ms)
    ✓ renders "good to see" you if the button was NOT clicked (4 ms)
    ✓ renders "Changed!" if the button was clicked (64 ms)
    ✓ does not render "good to see you" if the button was clicked (20 ms)

Test Suites: 1 passed, 1 total
```

```
1  const Output = props => {
2    return <p>{props.children}</p>
3  };
4
5  export default Output;
```

Notice: test suite continues to work w/ this wrapper component taking the text in as props.children

* Because render() really renders the entire component to the virtual DOM (entire component tree), it renders the content of the output component too *
* Could technically call this an integration test because >1 unit (component) is involved
* Because it's just a wrapper component w/o its own logic, integration test isn't really the right term *
* Don't necessarily need to split tests here *

1. When to split unit tests
   a. Related components...
      i. don't have a wrapper / child relationship
      ii. components have significant logic
      iii. components manage state
   b. What to do
      i. test core logic separately
      ii. integration test as needed