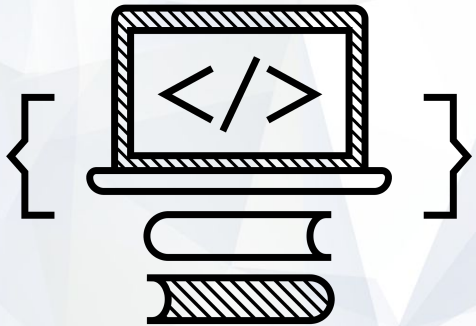
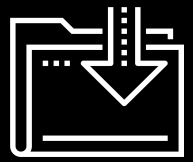


CRA Testing

Course: Java

S1

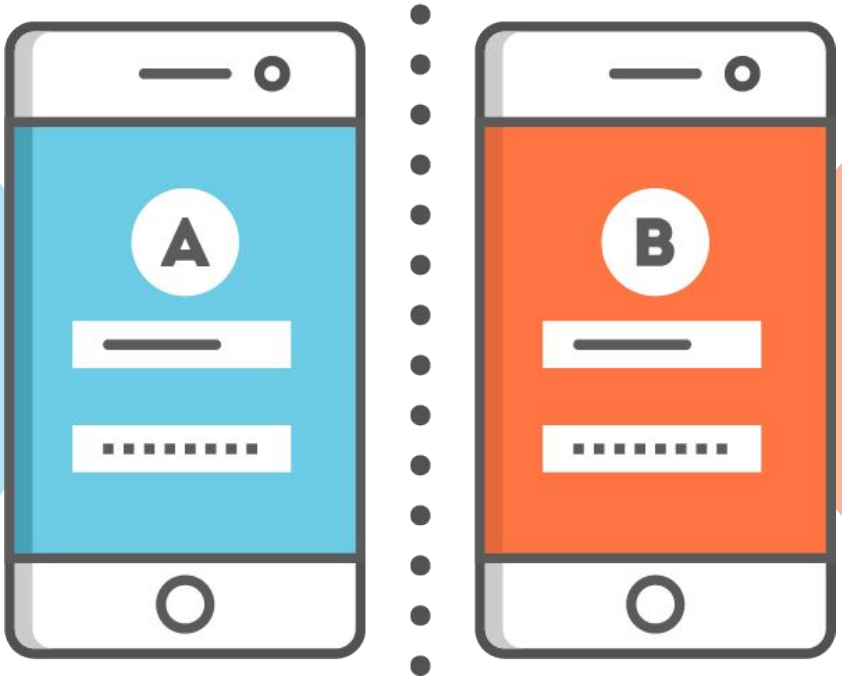


We learned the value of testing in Java. It's no less valuable in JavaScript. Code is code.

CRA Testing

We cannot claim that our code works without evidence. We must either:

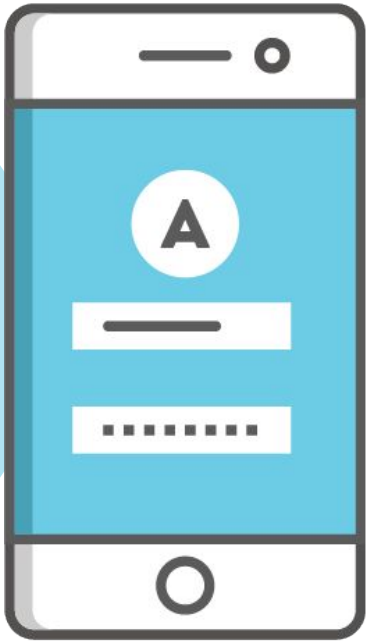
Manually test our code by spinning up the entire application, navigating to the correct component, and hoping that it works this time.



Write unit and integration tests that isolate our component.

CRA Testing (continued)

Option A has many drawbacks.



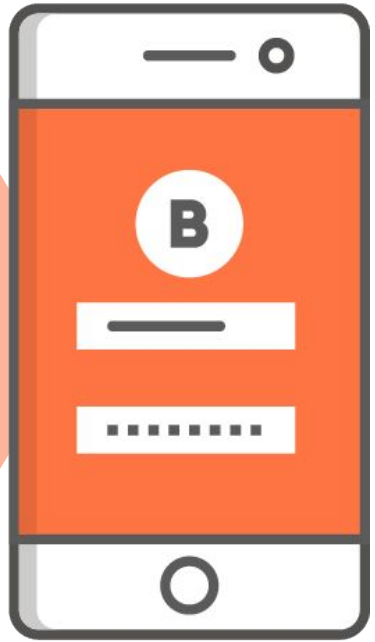
01 It’s slow.

02 It’s not thorough.

03 It has too much context.

CRA Testing (continued)

Option B has many benefits.



01 It’s fast.

02 Each scenario added gives us more confidence.

03 A well-written test isolates our code to confirm which component is causing an issue.

CRA Testing (continued)

Conceptually, Jest and jsdom tests are somewhere between unit tests and integration tests.

We can assert on DOM conditions, like an integration test (see Cypress and Selenium). But, we're not actually interacting with a second process, and we isolate components, like a unit test.

Ultimately, the name doesn't matter.

Jest and jsdom tests help us quickly test at the component level.

Jest

jsdom

Learning Outcomes

By the end of this lesson, you will be able to:

01 Create a Jest test that checks for the existence of an element in the DOM.

02 Interactively develop components and their tests with CRA's `npm test`.

03 Trigger an event in a test, and confirm DOM updates.

04 Use both Testing Library utilities and standard DOM APIs in tests.

05 Mock the `fetch` function.

Testing

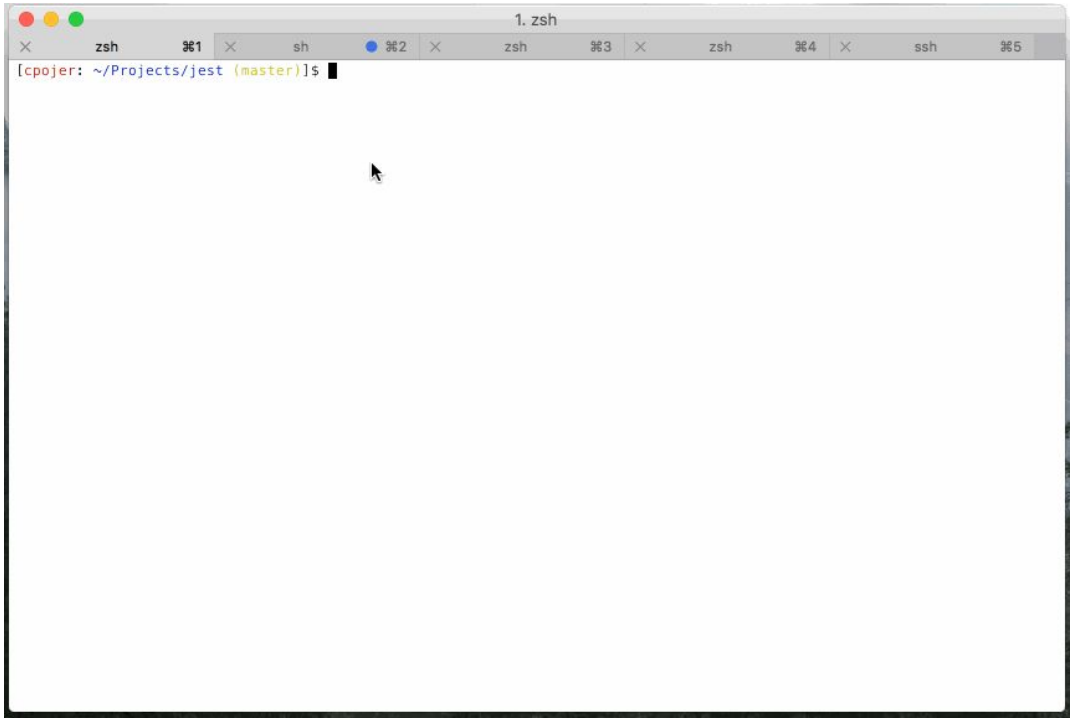
The React project cares about testing, but it doesn’t care which testing framework we use.

React vs. CRA Testing

Create React App chose to use Jest with jsdom integration.

In CRA, `npm test` starts an interactive testing session, just like `npm start` starts an interactive server/browser development session.

CRA can be configured to use a different testing framework.



Quick Start

01 Open `code\testing` with: VS Code

02 Install dependencies with `npm install`.

03 Start interactive testing with `npm test`.

04 The “renders learn react link” test in `App.test.js` should fail.

- Component tests follow the `ComponentName.test.js` naming convention.
- The test runner finds any JavaScript file ending with `.test.js`.
- Execute `npm test` to run tests, and then proceed with interactive test-driven development.

Quick Start (continued)

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

test('renders learn react link', () => {
  render(<App />);
  const linkElement = screen.getByText(/learn react/i);
  expect(linkElement).toBeInTheDocument();
});
```

Console

01 In the test console, press “W” to see options.

02 Press “F” to rerun the failed test.

03 Press “W” for options again.

04 Press “Enter” to run all tests.

05 Press “Q” to exit interactive testing.

```
Watch Usage
 › Press f to run only failed tests.
 › Press o to only run tests related to changed files.
 › Press q to quit watch mode.
 › Press p to filter by a filename regex pattern.
 › Press t to filter by a test name regex pattern.
 › Press Enter to trigger a test run.
```

```
 FAIL  src/App.test.js
  × renders learn react link (37 ms)

  ● renders learn react link

    TestingLibraryElementError: Unable to find an element with the text: /learn react/i. This could be because the text is broken
up by multiple elements. In this case, you can provide a function for your text matcher to make your matcher more flexible.

    <body>
     <!-- lots moare HTML -->
    </body>

      4 | test('renders learn react link', () => {
      5 |   render(<App />);
    > 6 |   const linkElement = screen.getByText(/learn react/i);
        |                              ^
      7 |   expect(linkElement).toBeInTheDocument();
      8 | });
      9 |

      at Object.getElementError (node_modules/@testing-library/dom/dist/config.js:37:19)
      at node_modules/@testing-library/dom/dist/query-helpers.js:90:38
      at node_modules/@testing-library/dom/dist/query-helpers.js:62:17
      at getByText (node_modules/@testing-library/dom/dist/query-helpers.js:111:19)
      at Object.<anonymous> (src/App.test.js:6:30)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
Snapshots:   0 total
Time:        2.55 s
Ran all test suites.
```

Consider this in comparison to Java testing. Are testing frameworks the same? Similar? Not at all similar?

Jest

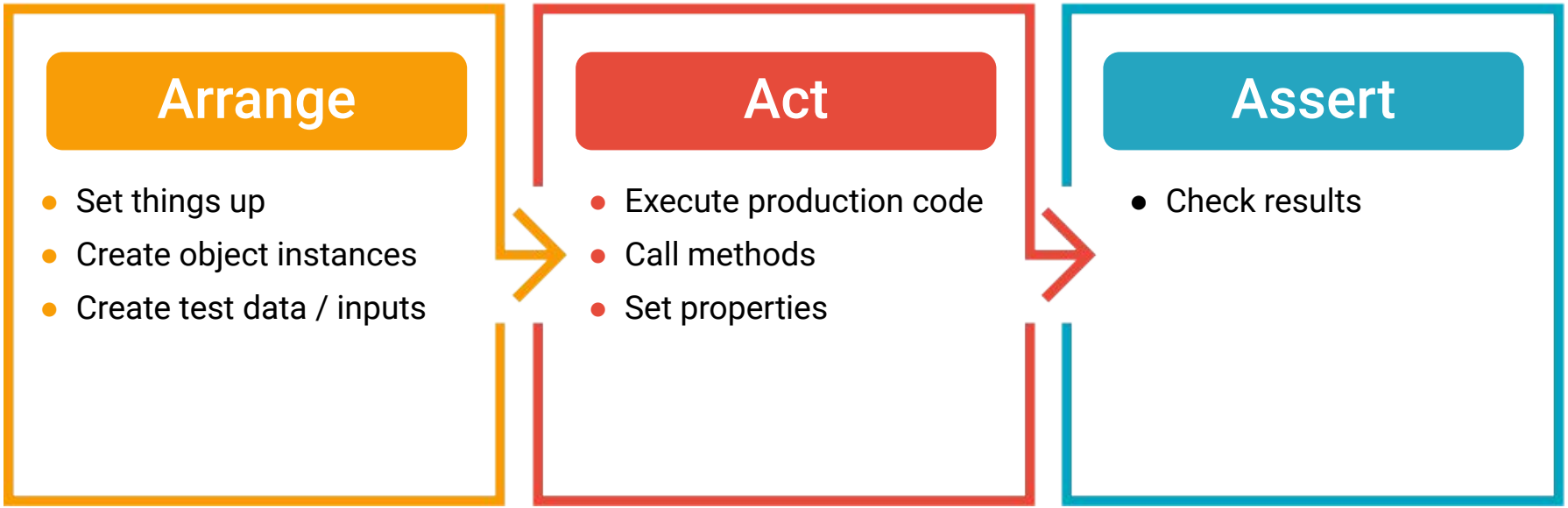
The three A's: Arrange, Act, and Assert

- Jest defines tests with the `test` function.
- The first argument is a description.
- The second is a function with arrangement, action, and assertion.

```
test("5 * 5 equals 25", () => {
   // Arrange
   const firstOp = 5;
   const secondOp = 5;
   // Act
   const actual = firstOp + secondOp;
   // Assert
   const expected = 25;
   expect(actual).toBe(expected);
});
```

Jest

The three A's: Arrange, Act, and Assert



Again, let’s compare to Java testing.

Does Java have set up methods?
Tear down methods?
Testing groups?

Jest API

https://jestjs.io/docs/en/api

- `beforeEach` is test setup to help DRY up test code.
- Create test groups with the `describe` function. Grouped tests are nested inside.
- The API is full-featured, with per-file setup and tear-down functions and many, many assertions.

```
// Scope is isolate per test file.
// obj doesn't leak into global scope.
let obj;

// Set up before each test
beforeEach(() => {
    obj = init();
});

// Solo test
test("obj should behave", () => {
    // Arrange, act, assert
});

// Create test groups with `describe`.
describe("obj", () => {

    test("should do the right thing", () => {
        // Arrange, act, assert
    })

    test("should fail predictably", () => {
        // Arrange, act, assert
    })
});
```

Jsdom

01 Start interactive testing in `code\testing` with `npm test`.

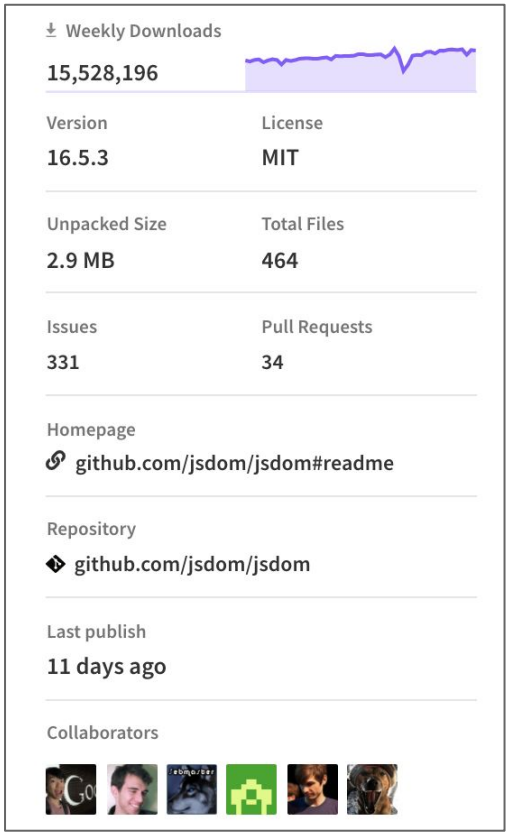
02 Update `App.test.js` to expect an element with the text "Pokedex".

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

test('renders "Pokedex"', () => {
    render(<App />);
    const element = screen.getByText("Pokedex");
    expect(element).toBeInTheDocument();
});
```

03 Go back to the interactive testing window. On save, tests rerun, and all tests should pass.

Jsdom (continued)



Although jsdom isn't the browser's DOM, it's very, very good. The team continues to add test edge cases to ensure compliance.

Jsdom has one significant advantage:
We don't have to fire up a browser to use it. That makes our tests fast.

- Jsdom is a 100% JavaScript DOM implementation.
- There's no visual result.
- It doesn't require a browser.
- CRA is configured with Jest/jsdom integration, test utilities that render JSX, and DOM-specific matchers.

https://www.npmjs.com

Jsdom (continued)

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

test('renders "Pokedex"', () => {
  render(<App />);
  const element = screen.getByText("Pokedex");
  expect(element).toBeInTheDocument();
});
```

```
 PASS  src/App.test.js
  √ renders "Pokedex" (47 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        4.052 s
Ran all test suites.

Watch Usage: Press w to show more.
```

DOM API

01 In `ColorButtonPanel.test.js` in VS Code, focus on the `DOM manipulation example` test.

02 Notice the transition to standard DOM.

```
 // grab the virtual DOM
const dom = result.baseElement;

// then use regular DOM methods
const strongs = dom.querySelectorAll("strong");
const buttons = dom.querySelectorAll("div>button");
```

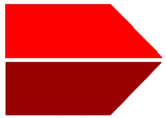
03 Notice matchers other than `toBeInTheDocument`.

```
expect(strongs.length).toBe(3);
expect(buttons.length).toBe(3);
expect(buttons[2].innerHTML.trim()).toBe("YELLOW");
```

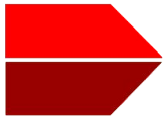
DOM API (continued)

CRA and Jest make it easy to “escape” back to the non-helper DOM API.

Jsdom doesn’t support the full API (`innerText` is usually `undefined`), but it’s sometimes more comfortable to assert conditions using the DOM API, especially for developers who are already comfortable with the DOM.



The Testing Library project provides useful test utilities.



We can also use the DOM API directly.

```
// ColorButtonPanel.test.js
import { render } from '@testing-library/react';
import ColorButtonPanel from './ColorButtonPanel.js';

test("DOM manipulation example", () => {
    const result = render(<ColorButtonPanel />);

    const redButton = result.getByText("RED");
    expect(redButton).toBeInTheDocument();

    // grab the virtual DOM
    const dom = result.baseElement;

    // then use regular DOM methods
    const strongs = dom.querySelectorAll("strong");
    expect(strongs.length).toBe(3);

    const buttons = dom.querySelectorAll("div>button");
    expect(buttons.length).toBe(3);

    expect(buttons[2].innerHTML.trim()).toBe("YELLOW");
});
```


Break

Events, Props, Forms, and Mocking

DOM Events

The Testing Library has more than one way to fire events:

- Firing Events
- user-event

The JavaScript standard `dispatchEvent` method:

01 In `ColorButtonPanel.test.js` in VS Code, focus on the `click changes color` test.

02 We're using regex in `expect(strongs[1].parentNode.innerHTML).toMatch(/.* red.*/)`. Change the argument to a string literal.

```
expect(strongs[1].parentNode.innerHTML).toMatch("red");
```

DOM Events (continued)

The test should fail. Here's why:

- We find a `strong` in the DOM.
- Then we move up to the `strong`'s parent.
- Then we must very flexibly match for the string that we're searching for, "red", in the `innerHTML`.
- A string literal isn't flexible enough.

```
<div>
    <strong>Current Color: </strong> {currentColor}
</div>
```

DOM Events (continued)

03

Fix the test by adding the regex back.

```
expect(strongs[1].parentNode.innerHTML).toMatch(/.* red.*/);
```

Trigger DOM events with the event API, and then confirm the DOM updates appropriately.

The document object is in scope inside Jest tests.

```
test("click changes color", () => {

    render(<ColorButtonPanel />);

    // `document` is in scope inside Jest tests,
    // just like a browser.
    const buttons = document.querySelectorAll("button");
    const strongs = document.querySelectorAll("strong");

    buttons[0].dispatchEvent(new MouseEvent("click", { bubbles: true }));
    expect(strongs[1].parentNode.innerHTML).toMatch(/.* red.*/);

    buttons[1].dispatchEvent(new MouseEvent("click", { bubbles: true }));
    expect(strongs[1].parentNode.innerHTML).toMatch(/.* blue.*/);

    buttons[2].dispatchEvent(new MouseEvent("click", { bubbles: true }));
    expect(strongs[1].parentNode.innerHTML).toMatch(/.* yellow.*/);
});
```

Intercepting Events

It’s possible to implement our own mock callbacks with something like:

```
let clicks = 0;
let args = [];
const callback = () => {
    clicks++;
    args.push(arguments);
};
```

But, we don’t need to spend the time. `jest.fn()` has those features and more.

Plus, it’s built to work with specific mock function matchers.

01

In `ColorButton.test.js` in VS Code, focus on the `click triggers callback` test.

Intercepting Events (continued)

02

Notice the mocked Jest function and how it can be used with the `toHaveBeenCalled` and the `toHaveBeenCalledTimes` matchers.

```
const onClick = jest.fn();
// click
expect(onClick).toHaveBeenCalledTimes(1);
// click
expect(onClick).toHaveBeenCalledTimes(2);
```

If we care only that a callback has been called, use `jest.fn()` to create a dummy function.

Assert on the number of times that it has been called.

```
test("click triggers callback", () => {
    // a "dummy" function to be called by CoinPanelFunc
    const onClick = jest.fn();

    const result = render(<ColorButton onClick={onClick} color="purple" />);
    const button = result.getByText("PURPLE");

    // Click the button.
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));

    // Our dummy function keeps track of the number of calls.
    expect(onClick).toHaveBeenCalledTimes(1);

    // Click it again.
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));

    expect(onClick).toHaveBeenCalledTimes(2);
});
```

Passing Props

Use props for arrangements that test specific scenarios.

01 In **ToDoForm.test.js**, focus on the `should submit` test.

02 Change the initial `ToDo`.

```
 const initial = {
    task: "Pick up corn chips",
    priority: 5,
    done: false
};
```

This causes `expect(task.value).toBe("Wash clothes")` to fail. Update the matcher.

```
expect(task.value).toBe("Pick up corn chips");
```

Passing Props (continued)

03 Once the test works again, explore the following matchers:

```
// Similar to `tbBeCalledTimes`. Mock must be called at least once.
expect(submit).toBeCalled();
// The mock function tracks calls and arguments.
// Here we confirm the argument matches our original ToDo.
expect(submit.mock.calls[0][0]).toMatchObject(initial);
```

- Part of the arrange step is passing sensible props.
- Props can be objects, arrays, functions, and more.
- Jest’s mock functions have many useful features.

https://jestjs.io/docs/mock-function-api

```
test("should submit", () => {

    const submit = jest.fn();
    const initial = {
        task: "Wash clothes",
        priority: 7,
        done: false
    };

    render(<ToDoForm initialToDo={initial} onSubmit={submit} />);
    const task = document.getElementById("task");
    const button = document.querySelector('button[type="submit"]');

    expect(task.value).toBe("Wash clothes");

    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));

    expect(submit).toBeCalled();
    expect(submit.mock.calls[0][0]).toMatchObject(initial);
});
```

Fill Out Forms

It’s a good idea to fill out forms and test their results.

user-event

When we set a value directly, like `input.value = "test text";`, the `onChange` event isn't triggered on the input.

This ruins the controlled component.

Form controls are updated, but state is not.

User event’s `type`, `click`, and so on trigger all appropriate events.

Fill Out Forms (continued)

01 In **ToDoForm.test.js**, focus on the `should fill out form` test.

02 Explore the process:

- Render an initial empty `ToDo`, and share a mock callback as props.
- Grab HTML elements.
- Fill in the task name.
- Fill in the priority.
- Check the “Done” checkbox.
- Click the “Submit” button.
- Confirm that the mock was called.
- Confirm that the actual `ToDo` matches an expected value.

CRA includes Testing Library’s `user-event` package.

Use it to type into inputs, select options, or upload files.

The benefit of `user-event` is that it triggers the component’s `onChange` event.

```
import { render } from '@testing-library/react';
import userEvent from '@testing-library/user-event';

test("should fill out form", () => {

    const submit = jest.fn();

    render(<ToDoForm initialToDo={{
        task: "",
        priority: 0,
        done: false
    }} onSubmit={submit} />);

    const task = document.getElementById("task");
    const priority = document.getElementById("priority");
    const done = document.getElementById("done");
    const button = document.querySelector('button[type="submit"]');

    userEvent.type(task, "Do the dishes");
    userEvent.type(priority, "{backspace}9");
    userEvent.click(done);
    userEvent.click(button);

    expect(submit).toBeCalled();

    const expected = {
        task: "Do the dishes",
        priority: 9,
        done: true
    };

    expect(submit.mock.calls[0][0]).toMatchObject(expected);
});
```

Mocking is the same concept as in Java, but with a different syntax.

Mocking

As applications grow, it's possible to mock services, helper classes, and even modules.

01 In `Pokemon.test.js`, focus on the `should render Weedle` test.

02 Focus on numbered comments.

I. Mock `fetch`. Because the actual call resolves two promises:

```
fetch(`https://pokeapi.co/api/v2/pokemon/${dexNumber}/`)
    .then(response => response.json())
    .then(result => setPokemon(result));
```

...we must do the same:

```
jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({                       // promise 1
        json: () => Promise.resolve(weedle) // promise 2
    })
);
```

Mocking (continued)

II. The component has async data actions, so we must render asynchronously. Without it, our test would finish before the data resolved and we couldn't assert.

III. Since `Pokedex` and `Pokemon` are managed by React Router, we must mock a router. That is a challenge.

To isolate a component, we must mock its context.

Component tests should never fetch data.

`fetch` can be mocked with `jest.spyOn`.

Any component that relies on the React Router for data must be wrapped in a Router.

```
import { act, render } from '@testing-library/react';
import { MemoryRouter, Route } from 'react-router-dom';
import Pokemon from './Pokemon.js';

const weedle = {
    name: "weedle",
    sprites: {
        back_shiny: "https://example.com/12.png"
    }
};

test("should render Weedle", async () => {

    // 1. Replace global.fetch with a mock implementation.
    jest.spyOn(global, "fetch").mockImplementation(() =>
        Promise.resolve({
            json: () => Promise.resolve(weedle)
        }));

    // 2. Since fetch is promise-driven,
    // we must await in our test.
    let result;
    await act(async () => {
        // 3. React Router must be mocked as well.
        result = render(
            <MemoryRouter initialEntries={["/12"]}>
                <Route path={"/:dexNumber"}>
                    <Pokemon />
                </Route>
            </MemoryRouter>
        );
    });

    const h3 = result.getByText("weedle");
    expect(h3).toBeInTheDocument();
});
```

Time to Code

Another Mock Walkthrough

Suggested Time:

10 Minutes

Activity: Testing With Jest

Suggested Time:

30 Minutes

Time's Up! Let's Review.

Questions?


RECAP

Recap

What problem does Jest solve?

What problem does jsdom solve?

What’s the syntax for an assertion in Jest?

How do we mock in Jest?

Learning Outcomes

By the end of this lesson, you will be able to:

01 Create a Jest test that checks for the existence of an element in the DOM.

02 Interactively develop components and their tests with CRA's `npm test`.

03 Trigger an event in a test, and confirm DOM updates.

04 Use both Testing Library utilities and standard DOM APIs in tests.

05 Mock the `fetch` function.


The End