Lab – React Testing

COS 420/520 - Introduction to Software Engineering

Goals for This Lab

- Look at simple examples of React testing
- Discuss testing strategies
- Learn what to Google once this lecture period ends

Types of Testing

- Unit Testing - Test a single function or component.
- Integration Testing - Test multiple units in combination.
- System Testing - Test all units that make up a system / subsystem.
    - E.g. account creation system, messaging system, file upload system
- End-to-End Testing - Test an application in full with all functionality and components in place.

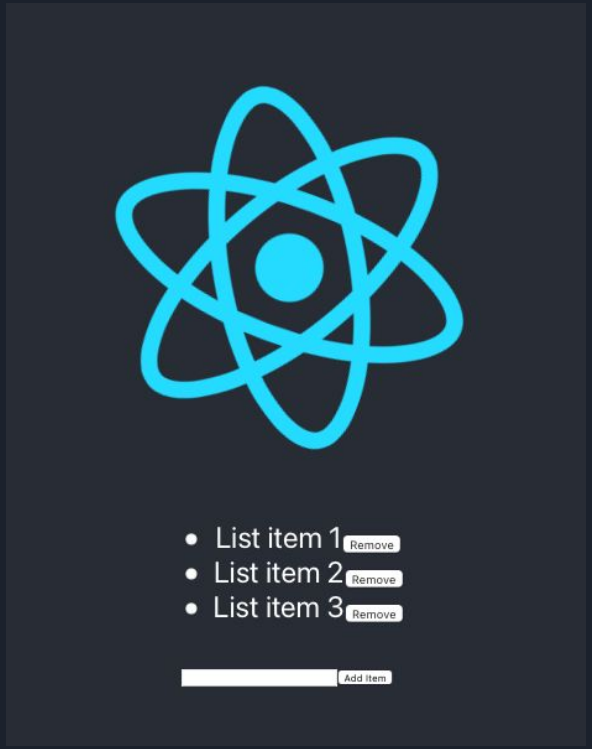
React Testing Tools

- JEST - The most commonly used React/JS testing framework, created by the creators of React. Includes everything needed to begin testing React
- React Testing Library - A set of helper functions, components, and tools to simplify and standardize React testing.
- Mocha, Chai, Jasmine, Enzyme, etc. - Specialized testing frameworks which handle different aspects of testing, such as simulating rendering and providing assertion methods.

Lab Environment - Assumptions & Preparation

- This lab builds off of the React web development lab from February: https://github.com/SKaplanOfficial/COS420-React-Lab.
- I assume you have NodeJS and NPM installed and configured.
- Clone the git repository and run npm install to view the code we are working with.
    - Since we used create-react-app, JEST and all other necessary dependencies will be installed.

The Application To Be Tested



Running Tests

- Recall: package.json defines script to be run with npm run [script].
- Create-React-App added a test script for us! So we can do npm run test.
- JEST won't automatically run tests until we begin making changes and saving files.

```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

```
No tests found related to files changed since last commit.
Press `a` to run all tests, or run Jest with `--watchAll`.

Watch Usage
 › Press a to run all tests.
 › Press f to run only failed tests.
 › 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.
```

Adding Tests

- For each React component (e.g. App, List, ListItem, ListControl), you can create a test file (e.g. App.test.js).
  - JEST will detect these files and run all tests in each file.
  - Dividing tests into separate files helps with component testing — a mix between unit and integration testing.
- For this lab, we use only App.test.js for the sake of simplicity.
  - This file already contains a test, but it fails if we try to manually run it.

Adding Tests

- For each React component (e.g. App, List, ListItem, ListControl), you can create a test file (e.g. App.test.js).
  - JEST will detect these files and run all tests in each file.
  - Dividing tests into separate files helps with component testing — a mix between unit and integration testing.
- For this lab, we use only App.test.js for the sake of simplicity.
  - This file already contains a test, but it fails if we try to manually run it.

Current App.test.js

- In the web dev lab, we modified the application without updating the associated test. The test fails because we removed the text that it checks for.

```
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  });
```

Structure of a Test

```
test('name of test', () => {
    // prep code
    expect(value).assertion();
});
```

Assertion Methods

- JEST offers *many* assertion methods, all of which are useful in different contexts.
- Some examples include toBe(value), toEqual(value), toBeTruthy(), toHaveStyle(style), toBeNull(), and toBeInTheDocument().
- A useful reference for these assertions is the JEST documentation site: https://jestjs.io/docs/expect

Arbitrary Tests

- Checking hard-coded values as we do here is generally not useful, but the structure is the same regardless.
- All of these tests should pass.

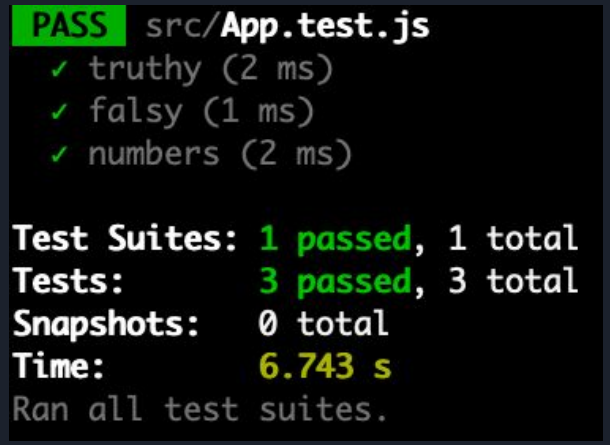
```
/* Arbitrary Tests */
test('truthy', () => {
  expect(true).toBeTruthy;
});

test('falsy', () => {
  expect(false).toBeFalsy;
});

test('numbers', () => {
  expect(3).toBe(3);
  expect(3).toEqual(3);
  expect(3).toBeGreaterThan(2);
  expect(3).toBeLessThan(4);
});
```

Arbitrary Test Results

- If you keep npm run test open in a terminal tab, JEST will run these new tests automatically.
- The overall test time can depend on many factors, such as what other applications are currently running.



Unit Tests

- Unit tests handle one function, with no limits on how complex that function is.
- Some React developers consider component testing and unit testing as the same, even though components might involve multiple functions.
- The goal of unit testing is to make sure that *a single* item (function, class, component, etc) behaves as expected.

```
/* Basic Unit Test */
const greeting = (name) => {
  return "Hello, " + name + "!";
}

test('function_greeting', () => {
  const val = greeting("world");
  expect(val).toBe("Hello, world!");
})
```

```
PASS src/App.test.js
  ✓ function_greeting (3 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        9.017 s
Ran all test suites.
```

Integration Tests

- The goal of integration testing is to make sure that units behave as expected when combined in defined ways.
- Testing a combination of units includes testing the units individually.
- Integration testing can speed up the testing process, but it can overlook internal issues. For example, what if mult() returned a negative value, but pow() set x to -mult(x, x)?

```
/* Basic Integration Test */
const mult = (x, y) => {
  return x * y;
}

const pow = (x, exp, iters) => {
  for (var i = 0; i < iters; i++) {
    x = mult(x, x);
  }
  return x;
}

test('pow', () => {
  // Compute ((3^2)^2)^2
  const val = pow(3, 2, 3);
  expect(val).toBe(6561);
});
```

```
PASS src/App.test.js
  ✓ pow (2 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        8.768 s
Ran all test suites.
```

UI Tests I

- UI testing simulates browser rendering and aims to ensure UI elements are present.
- The React Testing Library provides methods to navigate through React's virtual DOM, allowing you to easily select and interact with UI elements.
- Some examples include getByTestId(), getByText(), and getByRole().
- A useful cheatsheet can be found here: https://testing-library.com/docs/react-testing-library/cheatsheet/

UI Tests II

- The overall format of a UI test is:
  a. Render a component
  b. Get element objects
  c. Make assertions about those element objects.

```
PASS src/App.test.js
  ✓ list exists (87 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        7.417 s
Ran all test suites.
```

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

test('list exists', () => {
  render(<App />);

  const elem1 = screen.getByText('List item 1');
  const elem2 = screen.getByText('List item 2');
  const elem3 = screen.getByText('List item 3');

  expect(elem1).toBeInTheDocument();
  expect(elem2).not.toBeNull();
  expect(elem3).toBeInTheDocument();

  expect(elem3).toHaveStyle('display: list-item');
});
```

Event Tests I

- UI tests handle the look of an application, but only reveal visual issues.
- Event testing can be used to ensure user-lead events, such as filling out a form or clicking a button, function and behave as expected.
- The React Testing Library includes a user testing helper which makes simulating these events very straightforward.
- The overall process is to get an element object, simulate some action on that object, then make assertions about the state of that object or the application as a whole.

Event Tests II

```
/* Event Testing */
test('add item', () => {
  render(<App />);

  const textfield = screen.getByTestId("new_item_text");
  const submit_btn = screen.getByTestId("item_submit");
  userEvent.type(textfield, "Another item");
  userEvent.click(submit_btn);

  const elem4 = screen.getByText("Another item");

  expect(elem4).toBeInTheDocument();
  expect(elem4).toHaveStyle('display: list-item');
});

test('remove item', () => {
  render(<App />);
  const remove_btn = screen.getByText("List item 1").getElementsByTagName("button")[0];

  userEvent.click(remove_btn)

  const elem1 = screen.queryByText('List item 1');
  expect(elem1).toBeNull();
});
```

Event Tests III

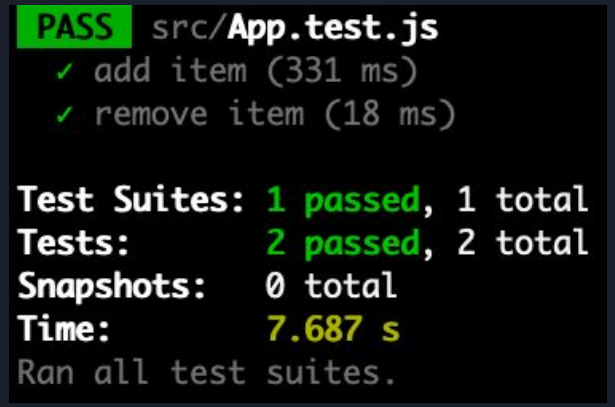
```
test('remove item', () => {
  render(<App />);
  const remove_btn = screen.getByText("List item 1").getElementsByTagName("button")[0];

  userEvent.click(remove_btn)

  const elem1 = screen.queryByText('List item 1');
  expect(elem1).toBeNull();
});
```

Event Test Results

- Any UI test or event test is likely to take longer due to the need to simulate rendering.
    - Running a full system test can be very slow as a result!



Mock Function Tests I

- JEST provides ways to mock/simulate functions.
- Mock functions are useful when you want to conduct unit tests on components that rely on other functions.
- By providing a mock function, you focus the test on the higher-level component.

```
/* Mock Functions */
const call_api = (value, api) => {
  for (var i = 0; i < 10; i++) {
    api(value);
  }
}

test("prep for api", () => {
  const fake_api = jest.fn();
  call_api("Hello", fake_api);
  expect(fake_api).toHaveBeenCalledTimes(10);
});
```

Mock Function Tests II

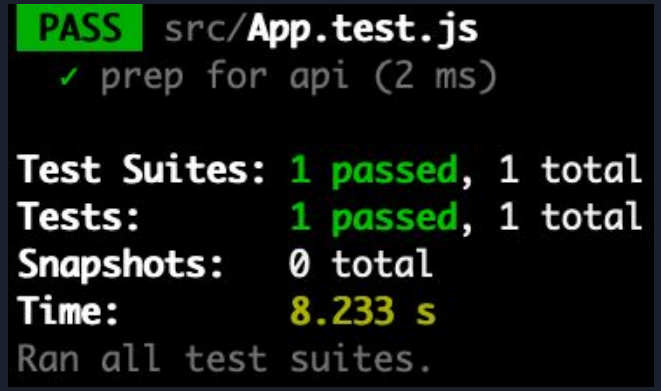
- Mock functions are useful when you want/need to avoid calling a function, e.g. if there are API rate limitations, database speed limitations, or security concerns.

```
/* Mock Functions */
const call_api = (value, api) => {
  for (var i = 0; i < 10; i++) {
    api(value);
  }
}

test("prep for api", () => {
  const fake_api = jest.fn();
  call_api("Hello", fake_api);
  expect(fake_api).toHaveBeenCalledTimes(10);
});
```

Mock Function Test Results

- Mock functions can help speed up system tests by abstracting away subsystems that can be tested separately and/or are known to function properly already.



Resources I

All code for this lab can be found on GitHub:

https://github.com/SKaplanOfficial/COS420-React-Lab

Feel free to reach out to me with any questions at:

stephen.kaplan@maine.edu