upGrad

#RahoAmbitious

Full-Stack Software Development

upGrad

Course: Advanced Frontend Development Using React

Lecture On: Testing in React

Instructor: Mrigank Kaushik

In the last class, we discussed...

- Redux makes state management centralised, more predictable, debuggable and flexible
- In the Redux architecture, views subscribe to data from the central store, and the only way to change state is to dispatch actions that go through reducer functions to generate a new version of the changed state
- The connect function from the react-redux library is used to connect the view to the store, which re-renders each time the state updates

Poll 1

What handles the transformation of an action to state in Redux?

A. Store
B. View
C. Connect
D. Reducer

Poll 1 (Answer)

What handles the transformation of an action to state in Redux?

A. Store
B. View
C. Connect
D. Reducer

Poll 2

How many maximum arguments can the createStore function take?

A. One
B. Two
C. Three
D. Zero

Poll 2 (Answer)

How many maximum arguments can the createStore function take?

A. One
B. Two
C. Three
D. Zero

Today's Agenda

1. Testing Frontend Code
2. Testing React Apps
3. Jest

Testing Frontend Code

Testing Frontend Code

upGrad

Why Test Frontend Code?

- Detect client-side performance issues
- Validate app behaviour on different browsers and OS
- Improve user experience
- Ensure seamless integration of services

Testing Frontend Code

upGrad

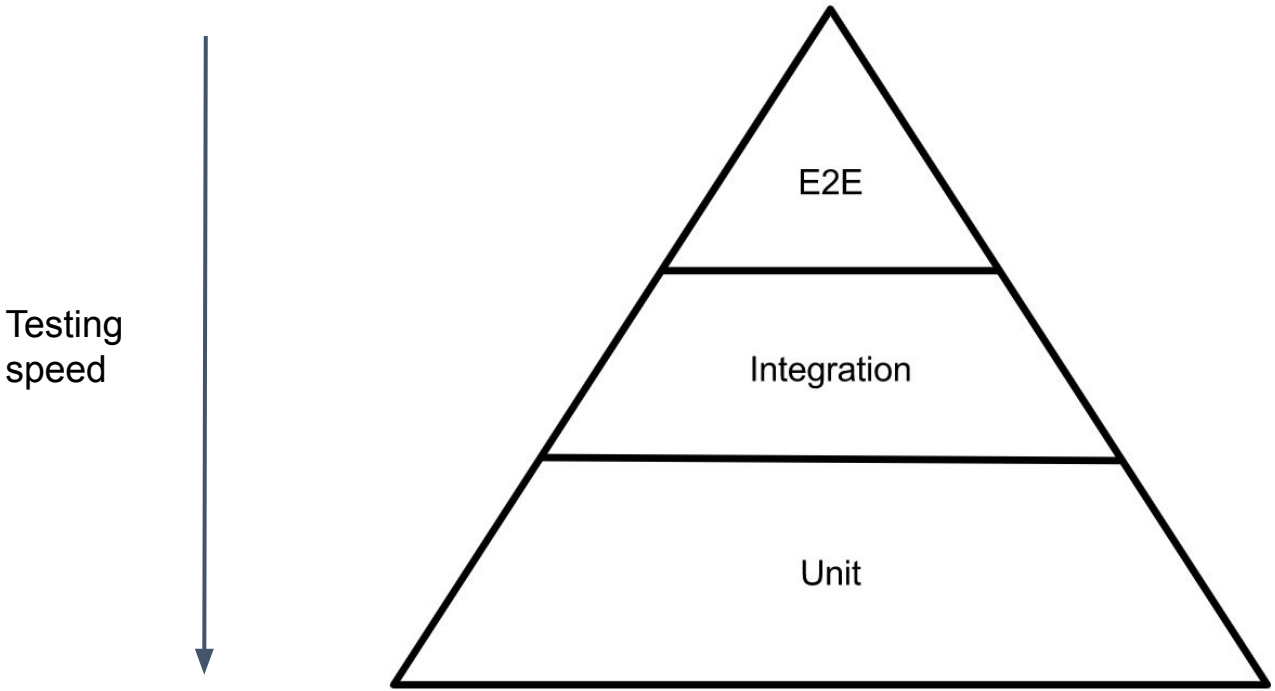
Types of Tests

- Unit tests
- Integration tests
- End-to-end tests
- Acceptance tests
- Accessibility tests
- Performance tests

Testing Frontend Code

upGrad

Test Pyramid



Testing Frontend Code

upGrad

Frontend Testing Tools

- **Cross-browser tests**
  e.g., LambdaTests
- **Snapshot tests**
  e.g., Jest, Ava
- **Visual regression tests**
  e.g., Needle, Wraith

Testing Frontend Code

upGrad

Frontend Testing Tools

- **Test runners**
  e.g., Karma, Jest, Jasmine
- **Structure providers**
  e.g., Jest, Jasmine, Mocha
- **Assertion functions**
  e.g., Jest, Jasmine, Chai, Unexpected
- **Mocks and stubs**
  e.g., Sinon, Enzyme, Jest
- **Code coverage reporters**
  e.g., Parasoft, Jtest, Cobertura

Poll 3

Which of these testing methodologies are specific to frontend development?

A. Snapshot testing
B. System testing
C. Structural testing
D. Cross-browser testing

Poll 3 (Answer)

Which of these testing methodologies are specific to frontend development?

A. Snapshot testing

B. System testing

C. Structural testing

D. Cross-browser testing

Testing React Apps

Testing React Code

upGrad

Methods

- **Render component trees**
  - Runs in a simplified environment
  - Uses a lightweight representation of DOM, does not use the actual DOM
- **Running a complete app**
  - Runs in an actual browser and real DOM
  - Closer to integration/end-to-end testing

Tradeoffs

- **Iteration speed versus realistic environment**
  - The real DOM and browser rendering can be slow
  - A mock simplified environment is usually much faster
- **The extent of mocking**
  - Test only a top-level component as a whole or also test the underlying components

Testing Environment

The rendering surface

- Simulated or mock DOM, e.g., jsdom
- Allows modelling interactions, dispatching events, and observing and asserting on the side effects of these actions
- Does not perform layout or navigation

Testing React Code

upGrad

Common recipes

- Setup/teardown
- Rendering
- Mocking data
- Mocking modules
- Firing DOM events to test interactivity
- Snapshot testing

Poll 4

Which of these is a limitation of a simulated browser environment used for testing like jsdom?

A. They do not support accessing DOM nodes
B. They do not perform layouts
C. They do not support triggering events
D. They do not support rendering React components

Poll 4 (Answer)

Which of these is a limitation of a simulated browser environment used for testing like jsdom?

A. They do not support accessing DOM nodes

B. They do not perform layouts

C. They do not support triggering events

D. They do not support rendering React components

Jest

What is Jest?

Jest is a JavaScript test runner that allows access to a simulated DOM via jsdom

Features of Jest:

- Fast
- Supports asynchronous code
- Generates code coverage
- Allows easy mocking of modules
- Provides rich context of failed tests

Getting started

- Install via NPM

```
npm install jest
```

- Create configuration file

```
jest --init
```

- Run from the command line

```
jest --config=config.json
```

Writing Your First Test

Let's assume you have a file *sum.js* which contains a function sum that adds two numbers, like this:

```
function sum(a, b) {
    return a + b;
}
```

Now, create a test file, called sum.test.js, which you will use to write some tests for the sum function

Writing Your First Test

You will write a test block:

```
test('sum', () => {
    expect(sum(2, 3)).toBe(5);
});
```

The expect function returns an expectation object, against which you will call a matcher to assert that the test code is doing what it is supposed to

Here toBe is a matcher that checks for equality

Matchers

Jest provides different kinds of matchers corresponding to data types

- toBeNull
- toBeUndefined
- toBeDefined
- toBeTruthy
- toBeFalsy

- toBe
- toEqual
- toBeGreaterThan
- toBeGreaterThanOrEqual
- toBeLessThan
- toBeLessThanOrEqual

toBe vs **toEqual**

- They behave in the same way for primitive data types
- For complex types like objects and arrays, toEqual checks for equality in terms of value; however, toBe checks for referential equality, i.e., it will match only if the object being tested is the same object that it is being tested against

```
expect({ foo: 'bar' }).toBe({ foo: 'bar' });     // false
expect({ foo: 'bar' }).toEqual({ foo: 'bar' });  // true
```

Testing negative cases: not

To check for the opposite condition of a matcher, you can use ‘not’, e.g.,

```
expect(1).not.toBe(2);      // true
```

Testing Asynchronous Code

The body of a test block is synchronous. However, if you wish to test asynchronous code such as setTimeout and functions that return a Promise, such as network calls, you can make use of a done callback to specify to Jest that you are testing asynchronous code, and it will not immediately exit once it reaches the last statement within the code block

```
test('test async code', (done) => {
    setTimeout(() => {
      expect(true).toBe(true);
      done();  // mark end of test
    }, 5000);
});
```

Setup and teardown

- If you wish to perform some common (setup) tasks before each test, you can use the beforeEach function
- If you wish to perform some common cleanup (teardown) tasks after each test, you can make use of the afterEach function
- If you wish to perform some setup task just once (and not before each test), you can use the beforeAll function
- Similarly, to perform some common teardown after all tests have finished execution, you can use the afterAll function

Setup and teardown

```
beforeEach(() => {
    // setup code, executed thrice, before each test
});

afterEach(() => {
    // teardown code, executed thrice, after each test
});

beforeAll(() => {
    // setup code, executed only once at the very beginning
});

afterAll(() => {
    // teardown code, executed only once at the end
});

test('test one', () => { ... });
test('test two, () => { ... });
test('test three', () => { ... });
```

Mock functions

To test units of code independently and in isolation, sometimes you want to assume that different modules that your code depends on are working as expected, e.g., an API call--you might not need to make an actual API call over the network but simply provide a mock response that gets consumed by your code

Similarly, you need not bother about the actual implementation of other functions that generate some output that our code consumes and simply use a mock value of the output to test your code alone

Mocking functions

```
const mockFn = jest.fn().mockReturnValue(42);
console.log(mockFn());  // 42

const mockFnImpl = jest.fn().mockImplementation((a, b) => a + b);
console.log(mockFnImpl(2, 3));  // 5
```

Alternately, you could also have simply done this:

```
const mockFnImpl = jest.fn((a, b) => a + b);
console.log(mockFnImpl(2, 3));  // 5
```

Mocking modules

Let's say you want to mock the axios library to return a mock response for an API call:

```
jest.mock('axios');
axios.get.mockResolvedValue({ data: { ... } });
```

Now, any axios.get call in your code will return a Promise that will immediately resolve to the dummy or mock response that you have provided

```
const mockFnImpl = jest.fn((a, b) => a + b);
console.log(mockFnImpl(2, 3));  // 5
```

Poll 5

Which function would you use to write a code to unmount and clean up a React component after every test?

A. beforeAll
B. beforeEach
C. afterAll
D. afterEach

Poll 5 (Answer)

Which function would you use to write a code to unmount and clean up a React component after every test?

A. beforeAll

B. beforeEach

C. afterAll

D. afterEach

Poll 6

If you wish to test a module A, but it has a dependency on another module B, which of the following techniques would you use to test module A independent of B?

A. Snapshot testing

B. End-to-end testing

C. Mocking module B to output hard-coded response

D. Setup and teardown

Poll 6 (Answer)

If you wish to test a module A, but it has a dependency on another module B, which of the following techniques would you use to test module A independent of B?

A. Snapshot testing

B. End-to-end testing

C. Mocking module B to output hard-coded response

D. Setup and teardown

Let's look at this simple counter component:

```
function MyCounter() {
    const [count, setCount] = React.useCount(0);
    const increment = () => setCount(count + 1);

    return (
      <div>
        <div className="count">Count: {count}</div>
        <button className="plus-btn" onClick={increment}>+</button>
      </div>
    );
}
```

Now, let's write a test just to render the component using jsdom:

```
import { render } from 'jsdom';

beforeEach(() => {
  // insert a DOM node that will serve as the root for our component to mount
  const div = document.createElement('div');
  div.setAttribute('id', 'root');
  document.body.appendChild(div);
});
afterEach(() => {
  // clean up DOM
  document.body.innerHTML = '';
});
test('render MyCounter component', () => {
  const root = document.getElementById('root');
  render(<MyCounter />, root);
});
```

Testing React Components

Now, you will add interactivity to your test. Let's click the plus button and check what happens

```
test('render MyCounter component', () => {
    const root = document.getElementById('root');
    render(<MyCounter />, root);
    const plusBtn = document.querySelector('.plus-btn');
    plusBtn.dispatchEvent(new MouseEvent('click', { bubbles: true });
    const counterEl = document.querySelector('.counter');
    expect(counterEl.innerText).toBe('Counter 1');
    plusBtn.dispatchEvent(new MouseEvent('click', { bubbles: true });
    expect(counterEl.innerText).toBe('Counter 2');
});
```

Notice how jsdom behaves just like the actual DOM, and you are able to access elements within it as well as trigger events

Phone Directory

Finally, you will be writing some tests for the phone directory app to ensure it has the correct functionality

For this purpose, you will be mocking the Redux store, and use Enzyme to render the component tree and test whether the correct values are getting reflected in the DOM

Code Reference

Poll 7

Which of the following functions from jsdom is used to render a React component in the simulated DOM environment?

A. createDOM
B. render
C. renderComponent
D. renderElement

Poll 7 (Answer)

Which of the following functions from jsdom is used to render a React component in the simulated DOM environment?

A. createDOM

B. render

C. renderComponent

D. renderElement

Hands-On Exercise

upGrad

Write a test suite to test the app you build in the last session to manage tasks

Automate the creation of a task by simulated user input and assert whether the correct task details are getting added to the list

The stub code is provided here
The solution code is provided here

All the code used in today's session can be found in the link provided below (branch Session13):

https://github.com/upgrad-edu/react-hooks/tree/Session13

Doubt Clearance (5 minutes)

The following tasks are to be completed after today's session:

MCQs

Key Takeaways

- Unit tests, integration tests, end-to-end tests, cross-browser tests, snapshot tests and visual regression tests are the different kinds of front end tests
- Jest is a test runner which offers a rich set of matchers to write assertions, can use jsdom to render and interact with React components in a lightweight, fast and fake browser-like environment
- It is possible to test both synchronous and asynchronous code with Jest

upGrad

#RahoAmbitious

Thank You!