Unit Testing React with React Testing Library

Objectives

In this chapter we will discuss

- The React Testing Library
- Running Unit Tests
- Testing Asynchronous Code
- Building and Running Component Tests
- Snapshot Testing
- Query Functions
- Simulating Events
- Text Matching

1.1 React Testing Framework

- React Projects create with create-react-app come equipped with a testing framework from Facebook that's:
  - Invoked using “npm test”
  - JavaScript Based
  - Open Source BSD 3-clause license
  - Built on top of Jasmine
  - From Facebook
  - Works with other development frameworks as well (e.g. Angular)

1.2 Features

- Support for:
  - Component Testing
  - Mocks
  - Asynchronous testing
  - Spies, Stubs
- Test Runner Features
  - Snapshot testing
  - Code coverage
  - Interactive mode

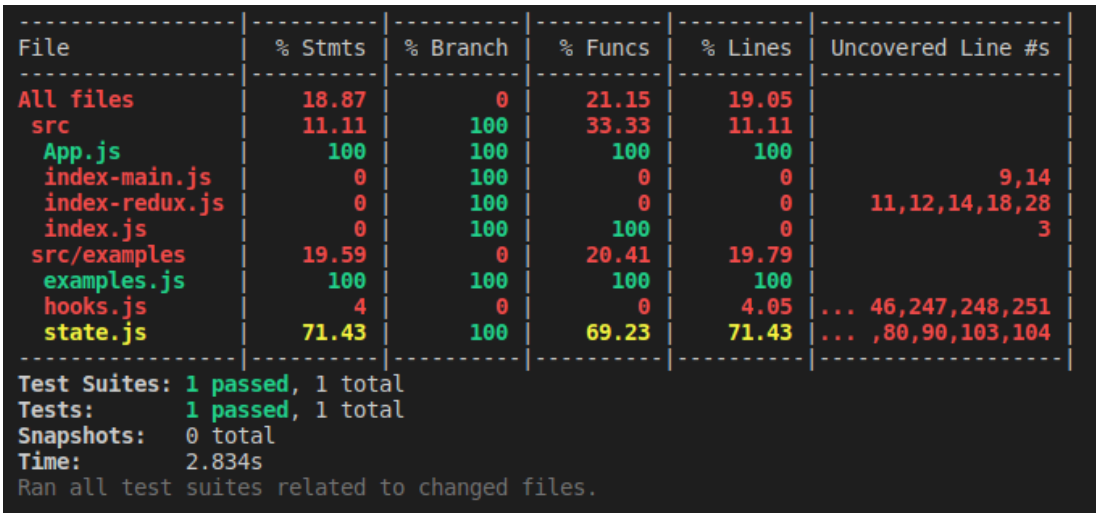
1.3 Snapshot Testing

- Compare old and new user interfaces to detect changes
  - Manual: build user interface and compare visually
  - Automated: generate UI snapshot and compare against existing snapshots
- Snapshots (for React) are a serialized version of the React tree
  - Snapshots should be stored in configuration management (SCM)
- On subsequent runs test runner will render the UI and compare the serialized form (new snapshot) with the old snapshot
- The test will fail if the snapshots are different

1.4 Code Coverage

- The test runner has a built-in code coverage feature
- Add the `--coverage` flag to the npm test script

```
"test": "react-scripts test --coverage",
```

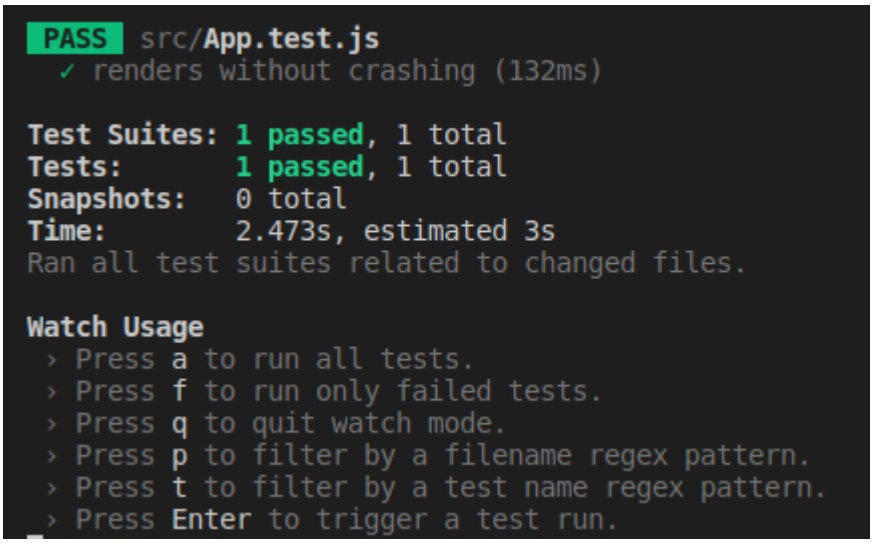


```
-----------------|----------|----------|----------|----------|-------------------|
File             |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
-----------------|----------|----------|----------|----------|-------------------|
All files        |    18.87 |        0 |    21.15 |    19.05 |                   |
 src             |    11.11 |      100 |    33.33 |    11.11 |                   |
  App.js         |      100 |      100 |      100 |      100 |                   |
  index-main.js  |        0 |      100 |        0 |        0 |              9,14 |
  index-redux.js |        0 |      100 |        0 |        0 |    11,12,14,18,28 |
  index.js       |        0 |      100 |      100 |        0 |                 3 |
 src/examples    |    19.59 |        0 |    20.41 |    19.79 |                   |
  examples.js    |      100 |      100 |      100 |      100 |                   |
  hooks.js       |        4 |        0 |        0 |     4.05 |... 46,247,248,251 |
  state.js       |    71.43 |      100 |    69.23 |    71.43 |... ,80,90,103,104 |
-----------------|----------|----------|----------|----------|-------------------|
Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.834s
Ran all test suites related to changed files.
```

1.5 Interactive Mode

One of the test runner's built-in features is its ability to watch for changes and selectively run (or not run tests)

- watch(default) – when a file changes it runs tests only on changed files
- You can also issue commands in interactive mode



Note

One non-trivial projects, test suites can become quite large. The ability to run tests only on changed files without constant reconfiguration makes it more likely that development teams will run tests frequently.

1.6 Projects created with *create-react-app*

- ***create-react-app*** is a command line utility for creating and maintaining React projects
- React projects created with create-react-app include the testing framework by default
- Installing create-react-app tool:

```
npm install -g create-react-app
```

- Create a React project:

```
create-react-app simple-app
```

- Unit tests are placed in the same folder as the files they test:

```
mycomponent.test.js (tests mycomponent.js)
```

- Integration tests can be placed in a designated directory:

```
\tests\myintegration.test.js
```

- Unit tests are run with the following command:

```
npm test
```

1.7 Default App Component Test

- A simple test file(App.test.js') that tests the App component is generated along with each project.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
  ReactDOM.unmountComponentAtNode(div);
});
```

- Executing the test runner will run this test:

```
npm test
```

Note

This test is trivial, it just checks to see that the application renders properly.

1.8 Unit Tests

- Test the smallest 'unit' of an application
  - *function* or *method*
- Unit tests are independent (i.e. you can run one or several)
- Unit tests should be confined to the class being studied
  - *Mocks* represent other classes/services
- Unit tests should be applied before Integration tests

Note

Tests that cross unit boundaries (e.g. access a database, call a RESTful service) are integration tests. During an integration test where multiple *units* are involved it may be difficult to pinpoint precisely where failures occur. Validating the *units* before integration tests simplifies the process.

1.9 Anatomy of a Unit Test

- Natural language *like* syntax
- Filenames based on file being tested ***.spec.js*** or ***.test.js***
- Tests can be located in the source directory or the `__tests__` directory
- Uses a `test()` or `it()` function with a callback to check the condition
- Test name should describe what's being testing:

  "isAvailable() should return false"

- How you would write a basic test:

```
it('isAvailable() returns false', () => {
  expect(isAvailable()).toBeFalsy();
});
```

Note

In JavaScript a value is *falsy* if it is *false, null, undefined, 0, NaN, '', ""*. A value is *truthy* if it is not *falsy*.

The above code uses an arrow function but regular functions also work:

```
it('isAvailable() returns false', function(){
  expect(isAvailable()).toBeFalsy();
});
```

1.10 Common Matchers

Matcher	Meaning
toBe	Strong equality (===), compares obj references
toEqual	Deep equality, compares object properties, (recursive)
toBeCloseTo	For floating point values allows for rounding
toBeTruthy, toBeFalsy	truthy/falsy
toBeNull	Checks for null values
toBeDefined, toBeUndefined	defined/undefined
toMatch	Tests a string against a regular expression
toContain	Array membership
toThrow	Tests for exceptions
not	Used to negate other matchers e.g. `expect(result).not.toBe(3);`

Note

This is only a partial list.

1.11 Combining Tests

- The **`describe()`** function combines tests
  - Uses a callback that calls **`test()`** or **`it()`** functions

```
describe( 'calculator', () => {
  it('should add',() => {
    expect(calc.add(3,2)).toBe(5);
  }
  test('should not divide by zero',() => {
    expect(calc.divide(3,0)).toThrow('divide by 0 error');
  }
});
```

Note

The `test` or `it` syntax may be used interchangeably.

1.12 Running Tests

- Run scripts inside your project

```
npm test
```

- "test" is defined in the package.json file:

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

- Here test is defined to include code coverage statistics:

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

1.13 Testing Promise based async code with 'done'

- Code that returns results asynchronously can be tested using the 'done' function as shown below.
- The 'done' function is always passed as a parameter into the testing function (but its only used when we need to test async code)

```
test('getResult returns with true', (done) => {
  getResult().then(
    (data)=>{
      expect(data).toBe(true);
      done();
    }
  )
});
```

- Here getResult() returns a Promise. We call 'then()' on the Promise and pass in a callback function. We call 'done' as the last line in the callback.
- 'done' lets the testing framework know when it can move on to the next test.

1.14 Setup and Teardown

- The testing framework will the following methods before and after tests.
- Add code to run before/after *all* tests to these methods:
  - `beforeAll(), afterAll()`
- Add code to run before/after *individual* tests to these methods:
  - `beforeEach(), afterEach()`
- Example:

```
beforeEach(
  ()=>{ console.log("in beforeEach");}
);
afterEach(
  ()=>{ console.log("in afterEach");}
);
```

- Note how the code above is placed inside of callback functions

1.15 react-testing-library

- The react-testing-library supports testing of components
- Installing the react-testing-library:

```
npm install --save @testing-library/react
```

- The react-testing-library makes use of jest-dom as well:

```
npm install --save @testing-library/jest-dom
```

- The following imports should be added at the top of you test file

```
import { render } from '@testing-library/react';
import '@testing-library/jest-dom/extend-expect';
```

- More information can be found here:

https://testing-library.com/docs/react-testing-library/intro

1.16 A Simple Component Test

- Here is a simple component:

```
function Stuff() {
    return (
        <div className="box">
            <h3>Components</h3>
            <Stuff1></Stuff1>
        </div>
    );
}
```

- We will test it with this test:

```
test('renders welcome message', () => {
  const { getByText } = render(<Stuff />);
  expect( getByText('Components') ).toBeInTheDocument();
});
```

- The test renders the component to a simulated DOM and then checks if the text “React Examples” appears

1.17 A Simple Snapshot Test

- A snapshot is a serialized(JSON) version of the React tree
- This example renders a component `<Stuff/>` and compares it to a snapshot

```
test('matches snapshot', function(){
  const { asFragment } = render(<Stuff />);
  expect(asFragment()).toMatchSnapshot();
});
```

1.18 Running and Updating SnapShot Tests

- The first time this test is run a snapshot is saved to disk and you will see this message:

```
› 1 snapshot written from 1 test suite.
```

- The next time the test is run a new snapshot will be taken and compared against the one that was saved. If it matches then the test passes. If it does not match then you get this message:

```
› 1 snapshot failed from 1 test suite. Inspect your
code changes or press `u` to update them.
```

- Pressing 'u' will update the saved snapshot with the latest one.
- If the original snapshot was correct then you would adjust your Component code to fix the issue.

1.19 Building Component Tests

- Tests follow these steps:
  - Call render() to return utility functions

```
const { getByText } = render(<Stuff />);
```

  - Simulate events and user interactions (if needed)

```
fireEvent(node: HTMLElement, event: Event)
```

  - Add an assertion statement that tests the results

```
expect(getByText('Title Text')).toBeInTheDocument();
```

- We will go over details of these steps in the next few pages

1.20 Calling Render

- Before calling render you need to import the render function:

```
import { render } from '@testing-library/react'
```

- Render returns one or more properties that can be used to test the component:

```
const { getByText, container } = render(<Stuff />);
```

- You can pass parameters to components in render

```
render(<Stuff2 text='some text' />)
```

- Note that curly brackets are not needed around the text value 'text' (rather than {'text'} like in JSX)

Notes

Documentation for render can be found here:

https://testing-library.com/docs/native-testing-library/api#render

1.21 Render Properties

- Various properties can be returned from render:
  - Utility Properties:
    - baseElement – The base element of the rendered component
    - container – An instance of ReactTestRendererInstance,
    - debug – A method for printing the container (for example, to log)
    - rerender – function to rerender the component from within your test

  - Query Properties:
    - getByText – get the element that holds the given text
    - getAltText – get the element with a given alt attribute value
    - getByTitle – get the element with a given title attribute value

Notes

Documentation for queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-queries#queries

Simulating Events

1.22 Simulating Events

- Before simulating events you will need to import 'fireEvent'

```
import { render, fireEvent } from '@testing-library/react';
```

- fireEvent() can be called like this:

```
fireEvent.click( getByText('Click'), {button: 1});
```

- The second parameter above provides properties for the event object that gets passed when the event is triggered
- Various events can be simulated in this way:
  - `click, submit, keyUp, focus, blur, drag, drop, etc.`

Notes

Documentation for text matching in queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-events

List of events that fireEvent can execute (fireEvent.event-name):

https://github.com/testing-library/dom-testing-library/blob/master/src/events.js

1.23 Testing Results

- Results can be tested with expectations (a type of assertion)
- Format for an expectation:

```
expect(htmlElement).matcher()
```

- The htmlElement can be obtained using a query returned by render()

```
let htmlElement = getByText('some text');
```

- An htmlElement matcher can be used to complete the test:

```
.toBeInTheDocument();
```

- The complete expression:

```
expect(getByText('text...').toBeInTheDocument();
```

Notes

Documentation for extended html Matchers can be found here:

https://github.com/testing-library/jest-dom#custom-matchers

1.24 Using Query Functions

- Query functions are returned by render()

```
const { getByText, queryByTitle } = render(<Stuff />);
```

- When executed they return HTML nodes, null or throw errors
- The results of calling query functions are tested using expections

```
expect(getByText('text...').toBeInTheDocument();
```

- Queries that start with "**get**" (getByText, getByTitle, etc):
  - Return the matching HTML node or array of nodes
  - or, ***Throw an error*** if no elements match
- Queries that start with "**query**" (queryByText, queryByTitle, etc):
  - Return the matching HTML node or array of nodes
  - or, ***return null*** if no elements match

Notes

Documentation covering queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-queries

1.25 Text Matching

- Sometimes you need a query to match only a subset of the element text

```
getByText('substring', {exact:false})
```

- The above can retrieve the following element

```
<p>Find a substring in a larger string</p>
```

- The string passed in can also be a regex (Regular Expression)
- Other examples:

Query	*Matches*
`getByText(container, 'Hello World')`	`Full text 'Hello World'`
`getByText('World', {exact:false})`	`Any string with 'World' in it`
`getByText(/World/)`	`Any string with 'World' in it (regex)`
`getByText(/world/i)`	`Any string with 'World' or 'world' or 'WORLD', etc (regex)`

Notes

Documentation for text matching in queries can be found here:

https://testing-library.com/docs/dom-testing-library/api-queries#textmatch

1.26 Counter Component

- Counter Component

- Code:

```
export function Counter() {
    const [count, setCount] = useState(0);
    return (
      <div>Counter: <span>{count}</span> 
        <button onClick={()=>setCount(count + 1)}>
              Click</button>
      </div>);
}
```

- Clicking on the button increments the counter value. We will create a test to verify this functionality.

1.27 counter-test.js

- This test checks that clicking the button updates the counter:

```
test('clicking button updates counter', () => {
  const { getByText } = render(<Counter />);
  expect(getByText('0')).toBeInTheDocument();
  fireEvent.click(getByText('Click'),{button: 1});
  expect(getByText('1')).toBeInTheDocument();
});
```

- Explanation:
  - Line 01 Calls render() to get the 'getByText' query
  - Line 02 Checks for an element with the default value '0'
  - Line 03 Gets the button and clicks it
  - Line 04 Checks for an element with the updated value '1'

1.28 Summary

In this chapter we discussed

- The React Testing Library
- Running Unit Tests
- Testing Asynchronous Code
- Building and Running Component Tests
- Snapshot Testing
- Query Functions
- Simulating Events
- Text Matching