TestUtils

Overview

- Provided by React team
- Documented at https://facebook.github.io/react/docs/test-utils.html
- "Shallow renderer" supports testing top level of components without using a DOM implementation
    - can't get details about nested components, but can test their types
- While not related to TestUtils, this section also covers testing Redux reducer functions and using a Redux store to dispatch actions in a test

Installs

- Install the following with "`npm install --save-dev`"
- **`mocha`**
  - "JavaScript test framework for Node.js and the browser"
  - could also use Jasmine, but would need to modify the `test` script in `package.json`
    - see next slide
  - test file naming convention - end with `.spec.js`
- **`expect`**
  - provides test assertion methods
  - see details at https://github.com/mjackson/expect
- **`expect-jsx`**
  - adds JSX-specific test assertion methods to expect
- **`react-addons-test-utils`**
  - "makes it easy to test React components in the testing framework of your choice"

Also, verify that `.babelrc` contains

```
{
  "presets": ["es2015", "react"]
}
```

Running Tests

- Add **`test`** script to **`package.json`**

```
"test": "mocha './test/**/*.spec.js' --compilers js:babel-core/register"
```

- **`npm test`**
- Output

from todo-redux-rest app

```
  dispatch
    ✓ should process a series of actions

  reducer
    ✓ should process add action
    ✓ should process archive action
    ✓ should process delete action
    ✓ should process error action
    ✓ should process setTodos action
    ✓ should process textChange action
    ✓ should process toggleDone action

  TodoHeader
todo-header.js render: entered
    ✓ should have expected content

  TodoList
todo-list.js render: entered
    ✓ should have expected content

  Todo
todo.js render: entered
    ✓ should have expected content
todo.js render: entered
    ✓ should have functioning delete button

  12 passing (217ms)
```

Shallow Renderer

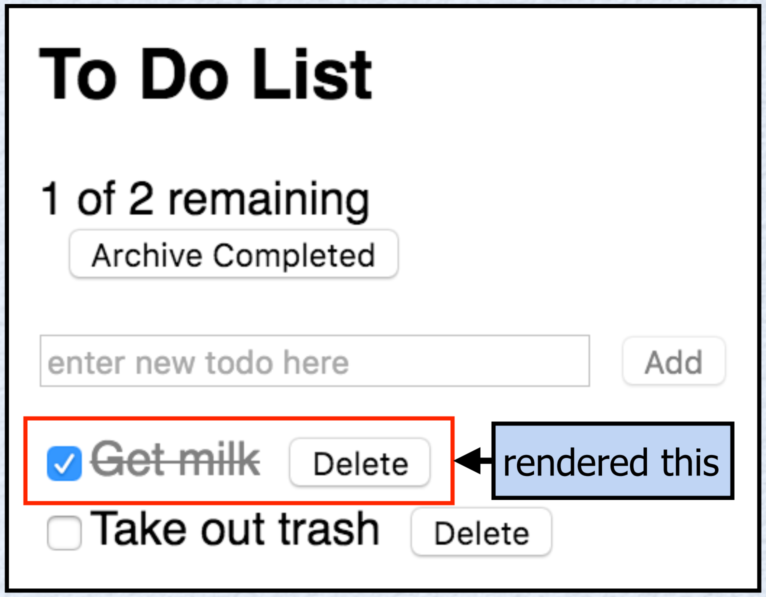
```
import TestUtils from 'react-addons-test-utils';
```

- Create renderer
  - `const renderer = TestUtils.createRenderer();`
- Render a component
  - `renderer.render(component-jsx);`
- Get output
  - `const output = renderer.getRenderOutput();`
- Make assertions about output
  - example `output` value

```
{
  '$$typeof': Symbol(react.element),
  type: 'li',
  key: null,
  ref: null,
  props: { children: [ [Object], [Object], [Object] ] },
  _owner: null,
  _store: {}
}
```

parts in red are what we want to test

see these objects on next slide



To Do List

1 of 2 remaining

rendered this

```
[ // children from previous slide
  {
    '$$typeof': Symbol(react.element),
    type: 'input',
    key: null,
    ref: null,
    props:
     { type: 'checkbox',
       checked: true,
       onChange: [Function: onToggleDone] },
    _owner: null,
    _store: {}
  },
  {
    '$$typeof': Symbol(react.element),
    type: 'span',
    key: null,
    ref: null,
    props: { className: 'done-true', children: 'Get milk' },
    _owner: null,
    _store: {}
  },
  {
    '$$typeof': Symbol(react.element),
    type: 'button',
    key: null,
    ref: null,
    props: { onClick: [Function: onDeleteTodo], children: 'Delete' },
    _owner: null,
    _store: {}
  }
]
```

parts in red are what we want to test

TestUtils.Simulate ...

- “Simulate an event dispatch on a DOM node with optional event data”
- “Simulate has a method for every event that React understands”
- Requires a DOM
  - can use JSDOM
  - **`npm install --save-dev jsdom`**
- Setup steps

```
import jsdom from 'jsdom';
global.document = jsdom.jsdom();
global.window = global.document.defaultView;
```

Finding Nodes

- Use methods in `TestUtils` to find nodes
    - "**scry**" methods return an array of matching nodes
    - "**find**" methods return a single node

    - `scryRenderedDOMComponentsWithClass(ReactComponent tree, string className)`
    - `findRenderedDOMComponentWithClass(ReactComponent tree, string className)`

    - `scryRenderedDOMComponentsWithTag(ReactComponent tree, string tagName)`
    - `findRenderedDOMComponentWithTag(ReactComponent tree, string tagName)`

    - `scryRenderedComponentsWithType(ReactComponent tree, Class componentClass)`
    - `findRenderedComponentWithType(ReactComponent tree, Class componentClass)`

Simulating Clicks

- Can click a node

```
TestUtils.Simulate.click(node);
```

Changing Inputs

- Can simulate changing the value of an input field

```
// Render the form into a DOM Document.
const form = TestUtils.renderIntoDocument(<MyForm {...props}/>);
// Get the first input.
const [input] =
  TestUtils.scryRenderedDOMComponentsWithTag(form, 'input');
// Change the value.
input.value = 'some new value';
TestUtils.Simulate.change(input);
// Make some assertion.
```

- Can also simulate pressing ENTER
- Provide event properties used in component
  - e.g. keyCode, which, etc.

```
animalInput.value = 'giraffe'
TestUtils.Simulate.change(animalInput);
TestUtils.Simulate.keyDown(animalInput,
  {
    key: 'Enter',
    keyCode: 13,
    which: 13
  });
```

Todo Test ...

```
/* global describe, it */
import Immutable from 'immutable';
import React from 'react'; //eslint-disable-line
import TestUtils from 'react-addons-test-utils';
import expect from 'expect';
import Todo from '../public/todo.js';

// Setup JSDOM which is needed by TestUtils.renderIntoDocument.
import jsdom from 'jsdom';
global.document = jsdom.jsdom();
global.window = global.document.defaultView;
```

... Todo Test ...

```
describe('Todo', () => {
  let deletedTodo, iTodo;

  function onDeleteTodo() { // mock
    deletedTodo = true;
  }

  function onToggleDone() {} // mock

  beforeEach(() => {
    deletedTodo = false;
    // Define prop values needed to render a Todo component.
    iTodo = Immutable.fromJS({text: 'Get milk', done: true});
  });

  it('should have expected content', () => {
    // Create a "shallow renderer" that renders only the
    // top-level component and does not require a DOM.
    const renderer = TestUtils.createRenderer();

    // Render a Todo element.
    renderer.render(
      <Todo iTodo={iTodo}
        onDeleteTodo={onDeleteTodo}
        onToggleDone={onToggleDone}/>);
    const output = renderer.getRenderOutput();
```

... Todo Test ...

```
    // Test the rendered output.

    expect(output.type).toBe('li');

    const children = output.props.children; // an Array
    expect(children.length).toBe(3);

    const [input, span, button] = children;

    expect(input.type).toBe('input');
    expect(input.props.type).toBe('checkbox');
    expect(input.props.checked).toBe(true);
    expect(input.props.onChange).toBe(onToggleDone);

    expect(span.type).toBe('span');
    expect(span.props.className).toBe('done-true');
    expect(span.props.children).toBe('Get milk’);

    expect(button.type).toBe('button');
    expect(button.props.children).toBe('Delete');
    expect(button.props.onClick).toBe(onDeleteTodo);
  });
```

... Todo Test

```
  it('should have functioning delete button', () => {
    const todo = TestUtils.renderIntoDocument(
      <Todo iTodo={iTodo}
        onDeleteTodo={onDeleteTodo}
        onToggleDone={onToggleDone}/>);
    const deleteBtn =
      TestUtils.findRenderedDOMComponentWithTag(todo, 'button');
    expect(deletedTodo).toBe(false);
    TestUtils.Simulate.click(deleteBtn);
    expect(deletedTodo).toBe(true);
  });
});
```

expect-jsx

- "Turns React elements into formatted strings"
- https://github.com/algolia/expect-jsx
- Provides these JSX-related assertions
  - `toEqualJSX(jsx)`
  - `toNotEqualJSX(jsx)`
  - `toIncludeJSX(jsx)`
  - `toNotIncludeJSX(jsx)`
- To enable use

```
import expect from 'expect';
import expectJSX from 'expect-jsx';
expect.extend(expectJSX);
```

TodoHeader Test ...

```
/* global describe, it */
import Immutable from 'immutable';
import React from 'react'; //eslint-disable-line
import TestUtils from 'react-addons-test-utils';
import expect from 'expect';
import expectJSX from 'expect-jsx';
import TodoHeader from '../public/todo-header.js';

expect.extend(expectJSX);

describe('TodoHeader', () => {
  it('should have expected content', () => {
    // Define prop values needed to render a TodoHeader component.
    const iTodos = Immutable.fromJS({
      1: {_id: 1, text: 'Get milk', done: true},
      2: {_id: 2, text: 'Take out trash', done: false}
    });
    function onArchiveCompleted() {}

    const renderer = TestUtils.createRenderer();

    // Render a TodoHeader component.
    renderer.render(
      <TodoHeader iTodos={iTodos}
        onArchiveCompleted={onArchiveCompleted}/>);
    const output = renderer.getRenderOutput();
```

… TodoHeader Test …

using expect-jsx

```
// Test the rendered output in pieces.
expect(output).toIncludeJSX(
  <h2>To Do List</h2>);
expect(output).toIncludeJSX(
  <span>1 of 2 remaining</span>);
expect(output).toIncludeJSX(
  <button onClick={onArchiveCompleted}>
    Archive Completed
  </button>);

// Test the rendered output all together.
expect(output).toEqualJSX(
  <div>
    <h2>To Do List</h2>
    <div>
      <span>1 of 2 remaining</span>
      <button onClick={onArchiveCompleted}>
        Archive Completed
      </button>
    </div>
  </div>
);
```

... TodoHeader Test

using expect - more tedious

```
    expect(output.type).toBe('div');
    const children = output.props.children; // an Array
    expect(children.length).toBe(2);

    const [header, div] = children;

    expect(header.type).toBe('h2');
    expect(header.props.children).toBe('To Do List’);

    expect(div.type).toBe('div');
    const divChildren = div.props.children;
    expect(divChildren.length).toBe(2);

    const [span, button] = divChildren;

    const spanChildren = span.props.children;
    expect(spanChildren[0]).toBe(1);
    expect(spanChildren[1]).toBe(' of ');
    expect(spanChildren[2]).toBe(2);
    expect(spanChildren[3]).toBe(' remaining’);

    expect(button.type).toBe('button');
    expect(button.props.children).toBe('Archive Completed');
    expect(button.props.onClick).toBe(onArchiveCompleted);
  });
});
```

TodoList Test ...

```
/* global describe, it */
import Immutable from 'immutable';
import React from 'react'; //eslint-disable-line
import TestUtils from 'react-addons-test-utils';
import expect from 'expect';
import Todo from '../public/todo.js';
import TodoList from '../public/todo-header.js';

describe('TodoList', () => {
  it('should have expected content', () => {
    // Define prop values needed to render a TodoList element.
    const iTodos = Immutable.fromJS([
      {_id: 1, text: 'Get milk', done: true},
      {_id: 2, text: 'Take out trash', done: false}
    ]);
    function onDeleteTodo() {}
    function onToggleDone() {}

    const renderer = TestUtils.createRenderer();

    // Render a TodoList component.
    renderer.render(
      <TodoList iTodos={iTodos}
        onDeleteTodo={onDeleteTodo}
        onToggleDone={onToggleDone}/>);
    const output = renderer.getRenderOutput();
```

... TodoList Test

```
    // Test the rendered output.

    expect(output.type).toBe('ul');

    const children = output.props.children;
    // children is an Immutable Seq, not an Array,
    // due to the way Todos are rendered in todo-list.js.
    expect(children.size).toBe(2);

    let todo = children.first();
    expect(todo.type).toBe(Todo);
    let iTodo = todo.props.iTodo;
    expect(iTodo.get('text')).toBe('Get milk');
    expect(iTodo.get('done')).toBe(true);

    todo = children.last();
    expect(todo.type).toBe(Todo);
    iTodo = todo.props.iTodo;
    expect(iTodo.get('text')).toBe('Take out trash');
    expect(iTodo.get('done')).toBe(false);
  });
});
```

checking the type of a nested, custom component

Testing Redux Reducers

- Doesn’t require any special testing libraries
- Just testing functions that are passed
  a current state object and an action object
  to verify that they return the correct new state object
- Use Immutable if reducers use it to represent state
  - it’s a good idea!
- For a large example, see
  https://github.com/mvolkmann/react-examples/tree/master/todo-redux-rest
- The following is one test from that example
  - **`test/reducer.spec.js`**

Example Reducer Test

```
/* global describe, it */
import Immutable from 'immutable';
import expect from 'expect';
import reducer from '../public/reducer.js';

describe('reducer', () => {
  ...
  it('should process toggleDone action', () => {
    let iState = Immutable.fromJS({
      text: 'foo',
      todos: {
        1: {_id: '1', text: 'Get milk', done: true},
        2: {_id: '2', text: 'Take out trash', done: false}
      }
    });

    // Toggle done flag for "Take out trash" to true.
    const action = {type: 'toggleDone', payload: {_id: '2'}};
    iState = reducer(iState, action);

    expect(iState.get('text')).toBe('foo'); // should not change
    const iTodos = iState.get('todos');
    expect(iTodos.size).toBe(2); // # of todos should not change
    const iTodo = iTodos.get('2'); // keys are strings
    expect(iTodo.get('text')).toBe('Take out trash'); // should not change
    expect(iTodo.get('done')).toBe(true); // should change
  });
  ...
});
```

Testing Redux Dispatch

- Doesn’t require any special testing libraries
- Test that a series of dispatched actions results in the correct state
- Use Immutable if reducers use it to represent state
  - it’s a good idea!
- For a large example, see https://github.com/mvolkmann/react-examples/tree/master/todo-redux-rest
- The following is one test from that example
  - **`test/dispatch.spec.js`**

Example Dispatch Test ...

```
/* global describe, it */
import expect from 'expect';
import reducer from '../public/reducer.js';
import {createStore} from 'redux';

describe('dispatch', () => {
  it('should process a series of actions', () => {
    const store = createStore(reducer);

    const actions = [
      {type: 'addTodo', payload: {_id: '1', text: 'Get milk'}},
      {type: 'addTodo', payload: {_id: '2', text: 'Take out trash'}},
      {type: 'addTodo', payload: {_id: '3', text: 'Make lunch'}},
      {type: 'toggleDone', payload: {_id: '1'}},
      {type: 'deleteTodo', payload: {_id: '3'}},
      {type: 'archiveCompleted'},
      {type: 'textChange', payload: {text: 'I typed this'}},
      {type: 'error', payload: 'Something went wrong'}
    ];
```

... Example Dispatch Test

```
    for (const action of actions) {
      store.dispatch(action);
    }

    const iState = store.getState();
    const iTodos = iState.get('todos');
    expect(iTodos.size).toBe(1); // one deleted and one archived
    const iTodo = iTodos.get('2'); // id of remaining Todo
    expect(iTodo.get('_id')).toBe('2');
    expect(iTodo.get('text')).toBe('Take out trash');
    expect(iState.get('text')).toBe('I typed this');
    expect(iState.get('error')).toBe('Something went wrong');
  });
});
```

Lab ...

- cd to react-examples/gift
- Follow steps in **`README.md`** to build
- Run existing tests by running "**`npm test`**"
  - all should pass unless the changes from
    the last lab in the Overview section are still present
  - if that is the case, update the tests to accommodate
    the display of the number of gifts for the selected person
- Review existing test code in **`test`** directory
  - **`text-entry.spec.js`**
  - **`name-select.spec.js`**
  - **`gift-list.spec.js`**
  - **`gift-app.spec.js`**

… Lab

- Add the following assertions in **`gift-list.spec.js`** see TODO comments
  - current value of the **`select`** matches **`selectedGift`**
    - stored in **`select.props.value`**
  - the **`select`** has three children,
    they are all **`option`** elements,
    their **`key`** is one of the gifts,
    and their text value is the same gift
  - the button **`onClick`** handler is the **`onDelete`** function
    - stored in **`button.props.onClick`**
- Add the following assertions in **`gift-app.spec.js`** see TODO comments
  - type of **`giftList`** is **`GiftList`**
- Hint
  - if you're unsure what data is available to use in assertions,
    add a **`console.log`** and run the tests to see it