Unit Testing to Integration Testing

Learn how integration testing can be useful for a react app.

Testing source code is essential to programming and should be seen as a mandatory exercise for serious developers. We want to verify our source code's quality and functionality before using it in production. The testing pyramid will serve as our guide.

The testing pyramid includes end-to-end tests, integration tests, and unit tests. **Unit tests** are used for small, isolated blocks of code, such as a single function or component. **Integration tests** help us figure out if these units work well together. An **end-to-end test** simulates a real-life scenario, such as the login flow in a web application. Unit tests are quick and easy to write and maintain; end-to-end tests are the opposite.

We want to have many unit tests covering our functions and components. After that, we can use several integration tests to make sure the most important functions and components work together as expected. Finally, we may need a few end-to-end tests to simulate critical scenarios. In this learning experience, we will cover **unit and integration tests**, along with a component specific testing technique called **snapshot tests**. **E2E tests** will be part of the exercise.

Since there are many testing libraries, it can be challenging to choose one as a beginner to React. We will use Jest by Facebook as a testing framework to avoid making this tutorial too opinionated. Most of the other testing libraries for React use Jest as foundation, so it's a good introduction.

Unit to Integration Testing

Often the lines between unit and integration tests are unclear. Testing the `List` component with its `Item` component could be considered an integration test, but it could also be a unit test for two tightly coupled components. In this section, we start with unit testing and move towards integration testing. Everything in between is a spectrum between both.

Let's start with a pseudo test in your *src/App.test.js* file:

```
describe('something truthy', () => {
  it('true to be true', () => {
    expect(true).toBe(true);
  });
});
```

Fortunately, create-react-app comes with Jest. You can run the test using the interactive create-react-app test script on the command line. The output for all test cases will be presented in your command line interface.

```
npm test
```

Jest matches all files with a *test.js* suffix in its filename when its command is run. Successful tests are displayed in green; failed tests are displayed in red:

```
describe('something truthy', () => {
  it('true to be true', () => {
    expect(true).toBe(false);
  });
});
```

src/App.test.js

Tests in Jest consist of **test suites** (`describe`), which are comprised of **test cases** (`it`), which have **assertions** (`expect`) that turn out green or red:

```
// test suite
describe('truthy and falsy', () => {
  // test case
  it('true to be true', () => {
    // test assertion
    expect(true).toBe(true);
  });

  // test case
  it('false to be false', () => {
    // test assertion
    expect(false).toBe(false);
  });
});
```

The "it"-block describes one test case. It comes with a test description that returns success or failure. We can also wrap this block into a "describe"-block that defines our test suite with many "it"-blocks for one specific component. Both blocks are used to organize your test cases. Note that the `it` function is known in the JavaScript community as a single-test case function; in Jest, however, `it` is often used as an alias `test` function.

```
describe('something truthy', () => {
  test('true to be true', () => {
    expect(true).toBe(false);
  });
});
```

To use React components in Jest, we require a utility library for rendering components in a test environment:

```
npm install react-test-renderer --save-dev
```

Also, before you can test your first components, you have to export them from your *src/App.js* file:

```
export default App;

export { SearchForm, InputWithLabel, List, Item };
```

Import them along with the previously installed utility library in the *src/App.test.js* file:

```
import React from 'react';
import renderer from 'react-test-renderer';

import App, { Item, List, SearchForm, InputWithLabel } from './App';
```

Write your first component test for the `Item` component. The test case renders the component with a given `item` using the utility library:

```
import React from 'react';
import renderer from 'react-test-renderer';

import App, { Item, List, SearchForm, InputWithLabel } from './App';

describe('Item', () => {
  const item = {
```

```
      title: 'React',
      url: 'https://reactjs.org/',
      author: 'Jordan Walke',

      num_comments: 3,
      points: 4,
      objectID: 0,
  };

  it('renders all properties', () => {
    const component = renderer.create(<Item item={item} />);

    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );
  });
});
```

App.test.js

Information about a component's or element's attributes are available via the `props` property. In the test assertion, we find the anchor tag (`a`) and its `href` attribute, and perform an equality check. If the test turns out green, we can be sure the anchor tag's `href` attribute is set to the correct `url` property of the `item` . In the same test case, we can add more test assertions for the other `item` 's properties:

```
describe('Item', () => {
  const item = {
    title: 'React',
    url: 'https://reactjs.org/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  };

  it('renders all properties', () => {
    const component = renderer.create(<Item item={item} />);

    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );
    expect(
      component.root.findAllByType('span')[1].props.children
    ).toEqual('Jordan Walke');
  });
});
```

Since there are multiple `span` elements, we find all of them and select the second one (index is `1` , because we count from `0`) and compare its React `children` prop to the item's `author` . This test isn't thorough enough, though. Once the order of `span` elements in the `Item` component changes, the test fails. Avoid this flaw by

changing the assertion to:

```
describe('Item', () => {
  const item = { ... };

  it('renders all properties', () => {
    ...

    expect(
      component.root.findAllByProps({ children: 'Jordan Walke' })
        .length
    ).toEqual(1);
  });
});
```

The test assertion isn't as specific anymore. It just tests whether there is one element with the item's `author` property. You can apply this technique for all the other properties of the `item` yourself. Otherwise, leave it for later as exercise.

We tested whether the Item component renders as text or HTML attributes (`href`), but we didn't test the callback handler. The following test case makes this assertion by simulating a click event via the `button` element's `onClick` attribute:

```
describe('Item', () => {
  const item = { ... };

  it('renders all properties', () => {
    ...
  });
  it('calls onRemoveItem on button click', () => {
    const handleRemoveItem = jest.fn();

    const component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );

    component.root.findByType('button').props.onClick();

    expect(handleRemoveItem).toHaveBeenCalledTimes(1);
    expect(handleRemoveItem).toHaveBeenCalledWith(item);

    expect(component.root.findAllByType(Item).length).toEqual(1);
  });
});
```

Jest lets us pass a test-specific function to the Item component as prop. These test specific functions are called **spy**, **stub**, or **mock**; each is used for different test scenarios. The `jest.fn()` returns us a *mock* for the actual function, which lets us capture when it's called. As a result, we can use Jest assertions like `toHaveBeenCalledTimes`, which lets us assert a number of times the function has

`Item` component's unit test is complete because we tested input (`item`) and output (`onRemoveItem`). The two shouldn't be confused with input (arguments) and output (JSX) of the function component, which were also tested. One last improvement makes the test suite for the Item component more concise by giving it a shared setup function:

```
describe('Item', () => {
  const item = { ... };
  const handleRemoveItem = jest.fn();

  let component;
  beforeEach(() => {
    component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );
  });
  it('renders all properties', () => {
    expect(component.root.findByType('a').props.href).toEqual(
      'https://reactjs.org/'
    );

    ...
  });

  it('calls onRemoveItem on button click', () => {
    component.root.findByType('button').props.onClick();

    ...
  });
});
```

App.test.js

A common setup (or teardown) function in tests removes duplicated code. Since the component must be rendered for both test cases, and the `props` are the same for both renderings, we can share this code in a common setup function. From there, we'll move on to testing the `List` component:

```
...

describe('Item', () => {
  ...
});

describe('List', () => {
  const list = [
    {
      title: 'React',
      url: 'https://reactjs.org/',
      author: 'Jordan Walke',
```

```
    num_comments: 3,
      points: 4,
      objectID: 0,
    },
    {
      title: 'Redux',
      url: 'https://redux.js.org/',
      author: 'Dan Abramov, Andrew Clark',
      num_comments: 2,
      points: 5,
      objectID: 1,
    },
  ];

  it('renders two items', () => {
    const component = renderer.create(<List list={list} />);

    expect(component.root.findAllByType(Item).length).toEqual(2);
  });
});
```

App.test.js

The test checks straightforward whether two `Item` components are rendered for the two items in the list. You could continue testing the `List` component by checking whether each callback handler (`onRemoveItem`) is called for each Item component, which would have a similar solution to the previous `Item` component's test. Is this test still a unit test or already an integration test?

Keeping this question in the room, we will move on to the `SearchForm` with `InputWithLabel` component:

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  let component;

  beforeEach(() => {
    component = renderer.create(<SearchForm {...searchFormProps} />);
  });

  it('renders the input field with its value', () => {
    const value = component.root.findByType(InputWithLabel).props
      .value;

    expect(value).toEqual('React');
  });
});
```

App.test.js

In this test, we assert whether the `InputWithLabel` component receives the correct prop from the `SearchForm` component. Essentially the test stops before the `InputWithLabel` component, because it only tests the interface (props) of it. Arguably it's still a unit test because the underlying implementation details of the `InputWithLabel` component could change without changing the interface. You can change the test to make it work through to the `InputWithLabel` component's input field because all child components and its elements are rendered too:

```
describe('SearchForm', () => {
  ...

  it('renders the input field with its value', () => {
    const value = component.root.findByType('input').props.value;

    expect(value).toEqual('React');
  });
});
```

App.test.js

This is our first integration test between the `SearchForm` and `InputWithLabel` components, which aren't as tightly coupled as the `List` and `Item` components. The `InputWithLabel` component can be used in other components (highly reusable), whereas the `Item` component is essentially a non-reusable part of the `List` component.

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  ...

  it('changes the input field', () => {
    const pseudoEvent = { target: 'Redux' };

    component.root.findByType('input').props.onChange(pseudoEvent);

    expect(searchFormProps.onSearchInput).toHaveBeenCalledTimes(1);
    expect(searchFormProps.onSearchInput).toHaveBeenCalledWith(
      pseudoEvent
    );
  });

  it('submits the form', () => {
    const pseudoEvent = {};
```

```
    component.root.findByType('form').props.onSubmit(pseudoEvent);


    expect(searchFormProps.onSearchSubmit).toHaveBeenCalledTimes(1);
    expect(searchFormProps.onSearchSubmit).toHaveBeenCalledWith(
      pseudoEvent
    );
  });

});
```

Like the `Item` component, the last two tests asserted the component's callback handler(s). All input (non function props) and output (callback handler function) props are tested for the `SearchForm` component's interface and integration with the `InputWithLabel` component.

You can test edge cases like a disabled button as well. The `update()` method on the rendered test component helps us provide new `props` to the component at stake:

```
describe('SearchForm', () => {
  const searchFormProps = {
    searchTerm: 'React',
    onSearchInput: jest.fn(),
    onSearchSubmit: jest.fn(),
  };

  let component;

  beforeEach(() => {
    component = renderer.create(<SearchForm {...searchFormProps} />);
  });

  ...

  it('disables the button and prevents submit', () => {
    component.update(
      <SearchForm {...searchFormProps} searchTerm="" />
    );

    expect(
      component.root.findByType('button').props.disabled
    ).toBeTruthy();
  });
});
```

Now we'll move one level higher in our application's component hierarchy. The App component fetches the list data, which is provided to the List component. After importing the App component, a naive test would look like this:

```
describe('App', () => {
  it('succeeds fetching data with a list', () => {
    const list = [
      {
        title: 'React',
        url: 'https://reactjs.org/',
        author: 'Jordan Walke',
        num_comments: 3,
        points: 4,
        objectID: 0,
      },
      {
        title: 'Redux',
        url: 'https://redux.js.org/',
        author: 'Dan Abramov, Andrew Clark',
        num_comments: 2,
        points: 5,
        objectID: 1,
      },
    ];

    const component = renderer.create(<App />);

    expect(component.root.findByType(List).props.list).toEqual(list);
  });
});
```

App.test.js

In the actual `App` component, a third-party library (axios) is used to make a request to a remote API. This API returns data we can't foresee in the test, so we have to mock it instead. Jest provides mechanisms that mock entire libraries and their methods. In this case, we want to mock the `get()` method of axios to return our desired data:

```
import React from 'react';
import renderer from 'react-test-renderer';

import axios from 'axios';

jest.mock('axios');

...

describe('App', () => {
  it('succeeds fetching data with a list', () => {
    const list = [ ... ];

    const promise = Promise.resolve({
      data: {
        hits: list,
      },
    });

    axios.get.mockImplementationOnce(() => promise);
```

```
    const component = renderer.create(<App />);

    expect(component.root.findByType(List).props.list).toEqual(list);
  });
});
```

The test reads synchronously, but we still have to deal with the asynchronous data. The component should re-render when its state updates. We can perform this with our utility library and async/await:

```
describe('App', () => {
  it('succeeds fetching data with a list', async () => {

    const list = [ ... ];

    const promise = Promise.resolve({
      data: {
        hits: list,
      },
    });

    axios.get.mockImplementationOnce(() => promise);
    let component;

    await renderer.act(async () => {
      component = renderer.create(<App />);
    });

    expect(component.root.findByType(List).props.list).toEqual(list);
  });
});
```

App.test.js

Instead of rendering the App component, we mocked the response from the remote API by mocking the method that fetches the data. To stay on the *happy path,* we told the test to treat the component as an asynchronously updating component. You can apply a similar strategy to the *unhappy path*:

```
describe('App', () => {
  it('succeeds fetching data with a list', async () => {
    ...
  });

  it('fails fetching data with a list', async () => {
    const promise = Promise.reject();

    axios.get.mockImplementationOnce(() => promise);

    let component;
```

```
    await renderer.act(async () => {
      component = renderer.create(<App />);
    });

    expect(component.root.findByType('p').props.children).toEqual(
      'Something went wrong ...'
    );
  });
});
```

The data fetching and integration with a remote API is tested now. We moved from focused unit tests for single components to tests with multiple components and their integration with third-parties like `axios` and remote APIs.

Snapshot Testing

Jest also lets you take a **snapshot** of your rendered component, run it against future captures, and be notified of changes. Changes can then be accepted or denied depending on the desired outcome. This mechanism complements unit and integration tests well, since it only tests the differences of the rendered output without heavy maintenance costs. To see it in action, extend the `Item` component test suite with your first snapshot test:

```
describe('Item', () => {
  ...

  beforeEach(() => {
    component = renderer.create(
      <Item item={item} onRemoveItem={handleRemoveItem} />
    );
  });

  ...

  test('renders snapshot', () => {
    let tree = component.toJSON();
    expect(tree).toMatchSnapshot();
  });

});
```

Run your tests again and observe how they succeed or fail. Once we change the output of the render block in `Item` component in the *src/App.js* file, by changing the structure of the returned HTML, the snapshot test fails. We can then decide whether to update the snapshot or to investigate the Item component.

Jest stores snapshots in a folder so it can validate the difference against future snapshot tests. Users can share these snapshots across teams for version control (e.g. git). Running a snapshot test for the first time creates the snapshot file in your project's folder. When the test is run again, the snapshot is expected to match the version from the latest test run. This is how we make sure the DOM stays the same.

The complete code is given below:

```
    □ □ □□ □   ã□  F   □□ □   □ )□       □   9□  5□ @@ □    °□  n□   □PNG
□
    IHDR   □    □□□   (-□S    äPLTE""""""""""""""""""2PX=r□)7;*:>H□¤-BGE□□8do5Xb6[eK□®K□¯1MU9gs3S
□
    IHDR   □    □□□   ×©ÍÊ  □ePLTE"""""""""""""""""""""""2RZN¢¹J□«3R[J□¬)59YÁÞ0KS4W`Q«ÄL□²%+-0JR
?^q÷ñíÛ□ï.},□ìsæÝ_TttÔ¾ □1#□□/(ì□-[□□□è`□è`Ì□ÚïÅðZ□d5□□□□?ÎebZ¿Þ□i.Ûæ□□□ìqÎ□+1°□}Â□5ù  ïçd
□
    IHDR          □□   D¤□Æ  □APLTE    """""""""""""""""""""""""""""2RZVºÖ_ÔôU·Ñ=r□$()'25]ÎíC□□0LS<o}X
□
    IHDR   @   @□□   □·□ì  □:PLTE    """"""""""""""""""""""""""""""""""""""""""""""""""""""""""
¢ßqÇ8Ù□´□mKË±mÆ¶mÛü·yi!è□ÎªYÏuë ÀÏ_Àï?í÷□ý+ò□□ÄA□|□ù{□□´?¿□_En□).□JËD¤<□
©¬¢Z\Ts©R*□(□  ¯©□J□□□□u□X/□4J□9□¡5·DEµ4kÇ4□&ì¥V4Ú□¡®Ð□□¯□vsf:àg,□¢èBC»î$¶□ºÍùî□□á□@□ô□I_
-ê>Û□º«¢XÕ¢î}ß¨ëÛÑ;□ÃöN´□ØvÅý□Î¸ÿ1 □ë×ÄO@&v/Äþ_□ö\ô□Ç\í.□□¾+0□□;□□□!□fÊ□¦´Ó%Â JY·O□Â□'/Å]_□
```

Exercises:

- Confirm the changes from the last section.
- Add one snapshot test for each of all the other components.
- Read more about testing React components.
  - Read more about Jest and Jest for React for unit, integration and snapshot tests.
- Read more about E2E tests in React.
- While you continue with the learning experience in the upcoming sections, keep your tests green and add new tests whenever you feel the need for it.