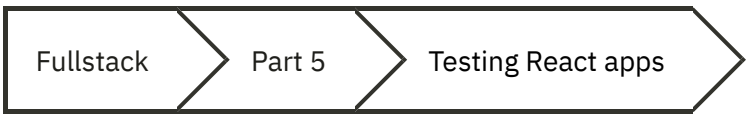


c Testing React apps

There are many different ways of testing React applications. Let's take a look at them next.

Tests will be implemented with the same Jest testing library developed by Facebook that was used in the previous part. Jest is actually configured by default to applications created with create-react-app.

In addition to Jest, we also need another testing library that will help us render components for testing purposes. The current best option for this is react-testing-library which has seen rapid growth in popularity in recent times.

Let's install the library with the command:

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

Let's first write tests for the component that is responsible for rendering a note:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important'
    : 'make important'

  return (
    <li className='note'>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Notice that the *li* element has the CSS classname *note*, that is used to access the component in our tests.

Rendering the component for tests

We will write our test in the *src/components/Note.test.js* file, which is in the same directory as the component itself.

The first test verifies that the component renders the contents of the note:

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

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const component = render(
    <Note note={note} />
  )

  expect(component.container).toHaveTextContent(
    'Component testing is done with react-testing-library'
  )
})
```

After the initial configuration, the test renders the component with the `render` method provided by the react-testing-library:

```
const component = render(
  <Note note={note} />
)
```

Normally React components are rendered to the *DOM*. The render method we used renders the components in a format that is suitable for tests without rendering them to the DOM.

`render` returns an object that has several properties. One of the properties is called *container*, and it contains all of the HTML rendered by the component.

In the expectation, we verify that the component renders the correct text, which in this case is the content of the note:

```
expect(component.container).toHaveTextContent(
  'Component testing is done with react-testing-library'
)
```

Running tests

Create-react-app configures tests to be run in watch mode by default, which means that the `npm test` command will not exit once the tests have finished, and will instead wait for changes to be made to the code. Once new changes to the code are saved, the tests are executed automatically after which Jest goes back to waiting for new changes to be made.

If you want to run tests "normally", you can do so with the command:

```
CI=true npm test
```

NB: the console may issue a warning if you have not installed Watchman. Watchman is an application developed by Facebook that watches for changes that are made to files. The program speeds up the execution of tests and at least starting from macOS Sierra, running tests in watch mode issues some warnings to the console, that can be removed by installing Watchman.

Instructions for installing Watchman on different operating systems can be found on the official Watchman website: https://facebook.github.io/watchman/

Test file location

In React there are (at least) two different conventions for the test file's location. We created our test files according to the current standard by placing them in the same directory as the component being tested.

The other convention is to store the test files "normally" in their own separate directory. Whichever convention we choose, it is almost guaranteed to be wrong according to someone's opinion.

Personally, I do not like this way of storing tests and application code in the same directory. The reason we choose to follow this convention is that it is configured by default in applications created by create-react-app.

Searching for content in a component

The react-testing-library package offers many different ways of investigating the content of the component being tested. Let's slightly expand our test:

```
test('renders content', () => {
```

```
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const component = render(
    <Note note={note} />
  )

  // method 1
  expect(component.container).toHaveTextContent(
    'Component testing is done with react-testing-library'
  )

  // method 2
  const element = component.getByText(
    'Component testing is done with react-testing-library'
  )
  expect(element).toBeDefined()

  // method 3
  const div = component.container.querySelector('.note')
  expect(div).toHaveTextContent(
    'Component testing is done with react-testing-library'
  )
})
```

The first way uses method *toHaveTextContent* to search for a matching text from the entire HTML code rendered by the component.
toHaveTextContent is one of many "matcher"-methods that are provided by the jest-dom library.

The second way uses the getByText method of the object returned by the render method. The method returns the element that contains the given text. An exception occurs if no such element exists. For this reason, we would technically not need to specify any additional expectation.

The third way is to search for a specific element that is rendered by the component with the querySelector method that receives a CSS selector as its parameter.

The last two methods use the methods *getByText* and *querySelector* to find an element matching some condition from the rendered component. There are numerous similiar query methods available.

Debugging tests

We typically run into many different kinds of problems when writing our tests.

The object returned by the render method has a debug method that can be used to print the HTML rendered by the component to the console. Let's try this out by making the following changes to our code:

```
test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const component = render(
    <Note note={note} />
  )

  component.debug()

  // ...
})
```

We can see the HTML generated by the component in the console:

```
console.log node_modules/@testing-library/react/dist/index.js:90
  <body>
    <div>
      <li
        class="note"
      >
        Component testing is done with react-testing-library
        <button>
          make not important
        </button>
      </li>
    </div>
  </body>
```

It is also possible to search for a smaller part of the component and print its HTML code. In order to do this, we need the `prettyDOM` method that can be imported from the *@testing-library/dom* package that is automatically installed with react-testing-library:

```
import React from 'react'
import '@testing-library/jest-dom/extend-expect'
import { render } from '@testing-library/react'
import { prettyDOM } from '@testing-library/dom'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }
```

```
  const component = render(
    <Note note={note} />
  )
  const li = component.container.querySelector('li')

  console.log(prettyDOM(li))
})
```

We used the selector to find the *li* element inside of the component, and printed its HTML to the console:

```
console.log src/components/Note.test.js:21
  <li
    class="note"
  >
    Component testing is done with react-testing-library
    <button>
      make not important
    </button>
  </li>
```

Clicking buttons in tests

In addition to displaying content, the *Note* component also makes sure that when the button associated with the note is pressed, the `toggleImportance` event handler function gets called.

Testing this functionality can be accomplished like this:

```
import React from 'react'
import { render, fireEvent } from '@testing-library/react'
import { prettyDOM } from '@testing-library/dom'
import Note from './Note'

// ...

test('clicking the button calls event handler once', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const mockHandler = jest.fn()

  const component = render(
    <Note note={note} toggleImportance={mockHandler} />
```

```
  )

  const button = component.getByText('make not important')
  fireEvent.click(button)

  expect(mockHandler.mock.calls).toHaveLength(1)
})
```

There's a few interesting things related to this test. The event handler is `mock` function defined with Jest:

```
const mockHandler = jest.fn()
```

The test finds the button *based on the text* from the rendered component and clicks the element:

```
const button = component.getByText('make not important')
fireEvent.click(button)
```

Clicking happens with the `fireEvent` method.

The expectation of the test verifies that the *mock function* has been called exactly once.

```
expect(mockHandler.mock.calls).toHaveLength(1)
```

`Mock objects and functions` are commonly used stub components in testing that are used for replacing dependencies of the components being tested. Mocks make it possible to return hardcoded responses, and to verify the number of times the mock functions are called and with what parameters.

In our example, the mock function is a perfect choice since it can be easily used for verifying that the method gets called exactly once.

Tests for the *Togglable* component

Let's write a few tests for the *Togglable* component. Let's add the *togglableContent* CSS classname to the div that returns the child components.

```
const Togglable = React.forwardRef((props, ref) => {
  // ...
```

```
  return (
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisibility}>
          {props.buttonLabel}
        </button>
      </div>
      <div style={showWhenVisible} className="togglableContent">
        {props.children}
        <button onClick={toggleVisibility}>cancel</button>
      </div>
    </div>
  )
})
```

The tests are shown below:

```
import React from 'react'
import '@testing-library/jest-dom/extend-expect'
import { render, fireEvent } from '@testing-library/react'
import Togglable from './Togglable'

describe('<Togglable />', () => {
  let component

  beforeEach(() => {
    component = render(
      <Togglable buttonLabel="show...">
        <div className="testDiv" />
      </Togglable>
    )
  })

  test('renders its children', () => {
    expect(
      component.container.querySelector('.testDiv')
    ).toBeDefined()
  })

  test('at start the children are not displayed', () => {
    const div = component.container.querySelector('.togglableContent')

    expect(div).toHaveStyle('display: none')
  })

  test('after clicking the button, children are displayed', () => {
    const button = component.getByText('show...')
    fireEvent.click(button)

    const div = component.container.querySelector('.togglableContent')
    expect(div).not.toHaveStyle('display: none')
```

```
  })

})
```

The `beforeEach` function gets called before each test, which then renders the *Togglable* component into the `component` variable

The first test verifies that the *Togglable* component renders its child component `<div className="testDiv" />`.

The remaining tests use the `toHaveStyle` method to verify that the child component of the *Togglable* component is not visible initially, by checking that the style of the *div* element contains `{ display: 'none' }`. Another test verifies that when the button is pressed the component is visible, meaning that the style for hiding the component *is no longer* assigned to the component.

The button is searched for once again based on the text that it contains. The button could have been located also with the help of a CSS selector:

```
const button = component.container.querySelector('button')
```

The component contains two buttons, but since `querySelector` returns the *first* matching button, we happen to get the button that we wanted.

Let's also add a test that can be used to verify that the visible content can be hidden by clicking the second button of the component:

```
test('toggled content can be closed', () => {
  const button = component.container.querySelector('button')
  fireEvent.click(button)

  const closeButton = component.container.querySelector(
    'button:nth-child(2)'
  )
  fireEvent.click(closeButton)

  const div = component.container.querySelector('.togglableContent')
  expect(div).toHaveStyle('display: none')
})
```

We defined a selector that returns the second button `button:nth-child(2)`. It's not a wise move to depend on the order of the buttons in the component, and it is recommended to find the elements based on their text:

```
test('toggled content can be closed', () => {
  const button = component.getByText('show...')
  fireEvent.click(button)

  const closeButton = component.getByText('cancel')
  fireEvent.click(closeButton)

  const div = component.container.querySelector('.togglableContent')
  expect(div).toHaveStyle('display: none')
})
```

The `getByText` method that we used is just one of the many queries *react-testing-library* offers.

Testing the forms

We already used the fireEvent function in our previous tests to click buttons.

```
const button = component.getByText('show...')
fireEvent.click(button)
```

In practice we used the *fireEvent* to create a *click* event for the button component. We can also simulate text input with *fireEvent*.

Let's make a test for the *NoteForm* component. The code of the component is as follows

```
import React, { useState } from 'react'

const NoteForm = ({ createNote }) => {
  const [newNote, setNewNote] = useState('')

  const handleChange = (event) => {
    setNewNote(event.target.value)
  }

  const addNote = (event) => {
    event.preventDefault()
    createNote({
      content: newNote,
      important: Math.random() > 0.5,
    })

    setNewNote('')
  }
```

```
  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}

export default NoteForm
```

The form works by calling the `createNote` function it received as props with the details of the new note.

The test is as follows:

```
import React from 'react'
import { render, fireEvent } from '@testing-library/react'
import '@testing-library/jest-dom/extend-expect'
import NoteForm from './NoteForm'

test('<NoteForm /> updates parent state and calls onSubmit', () => {
  const createNote = jest.fn()

  const component = render(
    <NoteForm createNote={createNote} />
  )

  const input = component.container.querySelector('input')
  const form = component.container.querySelector('form')

  fireEvent.change(input, {
    target: { value: 'testing of forms could be easier' }
  })
  fireEvent.submit(form)

  expect(createNote.mock.calls).toHaveLength(1)
  expect(createNote.mock.calls[0][0].content).toBe('testing of forms could be easier' )
})
```

We can simulate writing to *input* fields by creating a *change* event to them, and defining an object, which contains the text 'written' to the field.

The form is sent by simulating the *submit* event to the form.

The first test expectation ensures, that submitting the form calls the `createNote` method. The second expectation checks, that the event handler is called with the right parameters - that a note with the correct content is created when the form is filled.

Test coverage

We can easily find out the coverage of our tests by running them with the command

```
CI=true npm test -- --coverage
```

```
PASS src/components/Note.test.js
PASS src/components/NoteForm.test.js
PASS src/components/Togglable.test.js
------------------|----------|----------|----------|----------|-------------------|
File              |  % Stmts | % Branch |  % Funcs |  % Lines | Uncovered Line #s |
------------------|----------|----------|----------|----------|-------------------|
All files         |    20.18 |    27.78 |    15.38 |    20.95 |                   |
 src              |        0 |        0 |        0 |        0 |                   |
  App.js          |        0 |        0 |        0 |        0 |... 11,125,131,134 |
  index.js        |        0 |      100 |      100 |        0 |                 6 |
 src/components   |    66.67 |     62.5 |       60 |    66.67 |                   |
  Footer.js       |        0 |      100 |        0 |        0 |           3,4,10 |
  LoginForm.js    |        0 |      100 |        0 |        0 |          5,12,38 |
  Note.js         |      100 |       50 |      100 |      100 |                 4 |
  NoteForm.js     |      100 |      100 |      100 |      100 |                   |
  Notification.js |        0 |        0 |        0 |        0 |           3,4,5,8 |
  Togglable.js    |    90.91 |      100 |    66.67 |    90.91 |                15 |
 src/services     |        0 |      100 |        0 |        0 |                   |
  login.js        |        0 |      100 |        0 |        0 |           2,4,5,6 |
  notes.js        |        0 |      100 |        0 |        0 |... 20,21,24,25,26 |
------------------|----------|----------|----------|----------|-------------------|

Test Suites: 3 passed, 3 total
Tests:       7 passed, 7 total
Snapshots:   0 total
Time:        7.418s
Ran all test suites.
```

A quite primitive HTML report will be generated to the *coverage/lcov-report* directory. The report will tell us the lines of untested code in each component:

You can find the code for our current application in its entirety in the *part5-8* branch of this Github repository.

Exercises 5.13.-5.16.

5.13: Blog list tests, step1

Make a test which checks that the component displaying a blog renders the blog's title and author, but does not render its url or number of likes by default

Add CSS-classes to the component to help the testing as necessary.

5.14*: Blog list tests, step2

Make a test which checks that the blog's url and number of likes are shown when the button controlling the shown details has been clicked.

5.15*: Blog list tests, step3

Make a test which ensures that if the *like* button is clicked twice, the event handler the component received as props is called twice.

5.16*: Blog list tests, step4

Make a test for the new blog form. The test should check, that the form calls the event handler it received as props with the right details when a new blog is created.

If, for example, you set an *input* element's id attribute as 'author':

```
<input
  id='author'
  value={author}
  onChange={() => {}}
/>
```

You can access the contents of the field with

```
const author = component.container.querySelector('#author')
```

Frontend integration tests

In the previous part of the course material, we wrote integration tests for the backend that tested its logic and connected the database through the API provided by the backend. When writing these tests, we made the conscious decision not to write unit tests, as the code for that backend is fairly simple, and it is likely that bugs in our application occur in more complicated scenarios than unit tests are well suited for.

So far all of our tests for the frontend have been unit tests that have validated the correct functioning of individual components. Unit testing is useful at times, but even a comprehensive suite of unit tests is not enough to validate that the application works as a whole.

We could also make integration tests for the frontend. Integration testing tests the collaboration of multiple components. It is considerably more difficult than unit testing, as we would have to for example mock data from the server. We chose to concentrate making end to end tests to test the whole application, which we will work on in the last chapter of this part.

Snapshot testing

Jest offers a completely different alternative to "traditional" testing called snapshot testing. The interesting feature of snapshot testing is that developers do not need to define any tests themselves, it is simply enough to adopt snapshot testing.

The fundamental principle is to compare the HTML code defined by the component after it has changed to the HTML code that existed before it was changed.

If the snapshot notices some change in the HTML defined by the component, then either it is new functionality or a "bug" caused by accident. Snapshot tests notify the developer if the HTML code of the component changes. The developer has to tell Jest if the change was desired or undesired.

If the change to the HTML code is unexpected it strongly implies a bug, and the developer can

About course
Course contents
FAQ
Partners

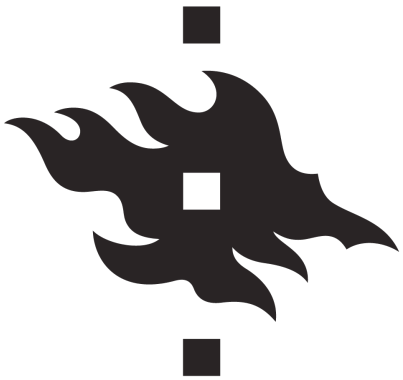
become aware of these potential issues easily thanks to snapshot testing.

Challenge

Propose changes to material

Part 5b
Previous part

Part 5d
Next part



UNIVERSITY OF HELSINKI

HOUSTON