

Workshop

Testing in React

Hint for trainers

- Report each change or addition to the **trainers'** Discord-Channel.
- Tell which Slide is affected, why the change is important and what benefit your change provides.
- Use the [code-highlighting-app](#) if you work with code-snippets.
- Use the following slide if you want to repeat certain topics of the workshop.

**unit vs. integration vs. e2e
testing**

Unit Testing

- code level
- every component can be unit tested (!)
- isolated testing
- Every dependency will be mocked and stubbed

Integration Testing

- code level
- Testing a component with its dependencies
- Takes sometimes a lot of effort to implement
- If isolated unit test doesn't make sense

E2E Testing

- User level (Browser)
- Browser robot
- Assertions against the document

Testing in React

- **Unit** tests (Jest)
- **Component** Testing
 - **Component** tests (with `@testing-library/react`)
 - Components (unit test)
 - Screens (integration test)
 - **Snapshot** tests
- End-to-end tests (Cypress)

Unit Tests with Jest

Jest is a JavaScript test runner



Why / What you'll learn



- Fast with parallel tests
- Zero-Configuration
- Everything you need built-in (e.g. code coverage, mocks, snapshot tests, ...)

Jest

<code>

Test method names should be sentences:

```
describe("BookListItem", () => {  
  test("renders a book from a book prop", () => {  
    // ...  
  });  
});  
  
// ✓ BookListItem renders a book from a book prop
```

Jest

<code>

Test method names should be sentences:

```
test("whether it will rain today", () => {  
  expect(isRaining("today")).toBe(true);  
});
```

Jest Basics

Jest in comparison to “classic tests”:

Test Suite: `describe()` **Test Suites can be nested!**

Test Case: `it()` or `test()`

Setup: `beforeEach()`

Tear Down: `afterEach()`

Assert: `expect()`

Jest Matchers

Matchers replace "assert_equal", "assert_..."

- `toBe()` → `toBeGreaterThan()`
- `toEqual()` → `toBeLessThan()`
- `toContain()` → `toBeCloseTo()`
- `toBeUndefined()`
- `toBeTruthy()`
- `toThrow()`

You can also create your own matchers.

Code coverage

- **statement coverage**: how many of the statements in the script have been executed.
 - 100% statement coverage implies 100% line coverage
- **branch coverage**: how many of the branches of control structures (e.g. if statements) have been executed.
- **function coverage**: how many of the functions defined have been called.
- **line coverage**: how many of lines of source code in the script have been tested.

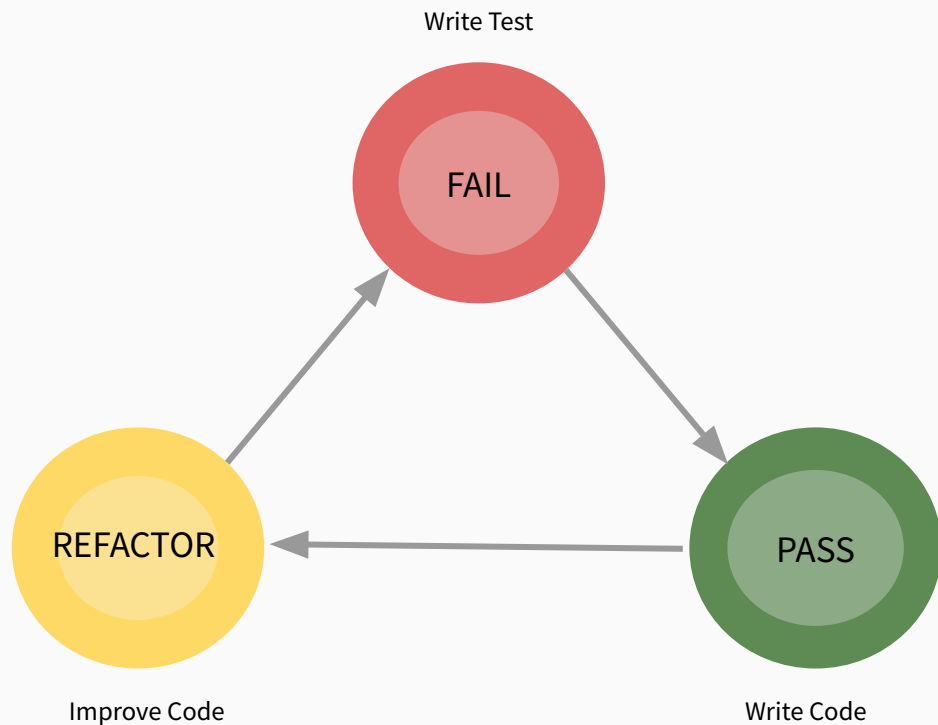
Code coverage

Example output

code coverage comes in colors **green**,
yellow and **red** as a quick visual feedback.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	91.67	100	81.82	90.91	
common/util	100	100	100	100	
leapYear.ts	100	100	100	100	
test-utils.tsx	100	100	100	100	
components/BookList	100	100	100	100	
index.tsx	100	100	100	100	
components/BookListItem	100	100	100	100	
index.tsx	100	100	100	100	
components/Counter	75	100	60	75	
index.tsx	75	100	60	75	14-16

Test Driven Development (TDD)



Test Driven Development (TDD)

1. Write a test case and make sure it fails. (red)
2. Satisfy the test case with minimal effort. (green)
3. Improve/refactor your code...
 - a. Meet general code guidelines.
 - b. Make it readable and comprehensible.
 - c. Remove redundant code.
4. Verify that the test case is still passing. (green)

Task

**TDD (Test driven development)
with Jest**



Testing React - Component Tests

Component tests

- Tests of individual React (Native) components
- *“Unit tests for Components”*
- You can create an instance of the component, pass it props, interact with it.

react-testing-library contains simple and complete React DOM testing utilities that encourage good testing practices.



render

<code>

Render into a container which is appended to `document.body`

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

```
render(<div />)
```

Queries

Queries are the methods that Testing Library gives you to find elements on the page

- `getBy...`
- `queryBy...`
- `findBy...`
- `getAllBy...`
- `queryAllBy...`
- `findAllBy...`

Queries

They behave differently if they can't find one or multiple elements

- | | | |
|---------------------------|---|------------------------------|
| ● <code>getBy...</code> | <code>throw error</code> | ● <code>getAllBy...</code> |
| ● <code>queryBy...</code> | <code>return null</code> <code>return []</code> | ● <code>queryAllBy...</code> |
| ● <code>findBy...</code> | <code>throw error</code> | ● <code>findAllBy...</code> |

query return null when no
matching element is found -> allow
negative assertions*

find also retries asynchronously*

getBy... Queries

<code>

Query argument can be either a string, regex, or a function which returns **true** for a match and **false** for a mismatch.

```
// Matching a string:
screen.getByText('Hello World') // full string match
screen.getByText('llo Worl', {exact: false}) // substring match
screen.getByText('hello world', {exact: false}) // ignore case

// Matching a regex:
screen.getByText(/World/) // substring match
screen.getByText(/world/i) // substring match, ignore case
screen.getByText(/^hello world$/i) // full string match, ignore case
screen.getByText(/Hello W?oRld/i) // substring match, ignore case, searches for "hello world" or "hello orld"

// Matching with a custom function:
screen.getByText((content, element) => content.startsWith('Hello'))
```

Other getBy... query methods

<code>

Examples of queries with different query methods

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

```
// ByText
```

```
render(<MyComponent />)
```

```
const aboutNode = screen.getByText(/about/i)
```

```
// ByLabelText
```

```
render(<MyForm />)
```

```
const nameInputNode = screen.getByLabelText('name')
```

```
// ByPlaceholderText
```

```
const ageInputNode = screen.getByPlaceholderText('age')
```

findBy... Queries

<code>

These queries are used for asynchronous actions or when you need to wait for an element to appear **asynchronously**. Example:

```
import { render, screen } from '@testing-library/react';
import userEvent from '@testing-library/user-event';
import { fetchPosts } from './api';

test('renders a list of posts fetched asynchronously', async () => {
  render(<Posts />);
  const loadButton = screen.getByRole('button', { name: 'Load Posts' });
  userEvent.click(loadButton);

  const postElement = await screen.findByText('Sample Post');
  expect(postElement).toBeInTheDocument();
});
```

Task

Component Tests



Testing React - Snapshots

Snapshots save the complete test
output into a file

react-test-renderer

Renders React components to pure JavaScript objects, without dependencies on the DOM.

react-test-renderer

- Not included in `create-react-app`
- Install it via npm

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

react-test-renderer

<code>

Import the test renderer

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

react-test-renderer

<code>

The result of the renderer is an object

```
const result = renderer.create(  
  <Link page="https://workshops.de/">Workshops.de</Link>  
)  
)  
.toJSON();
```

```
// => { type: 'a',  
// =>   props: { href: 'https://workshops.de/' },  
// =>   children: [ Workshops.de ] }
```

Snapshots

Snapshots save the HTML of a rendered component into a file

Why / What you'll learn



Don't spend more time writing a test than the component itself.

Matcher for Snapshots

<code>

Snapshots use the matcher `.toMatchSnapshot()`

```
it('renders a book item correctly', () => {  
  const book = { title: "React Advanced"  
  
  const tree = ReactTestRenderer.create(  
    <BookListItem book={book} />  
  ).toJSON();  
  expect(tree).toMatchSnapshot();  
});
```

Matches the tree and the saved snapshot

Snapshots and `react-router-dom`

<code>

`MemoryRouter` may be used to provide routing context in tests

```
it(`renders a book item correctly`, () => {  
  const tree = renderer  
    .create(  
      <MemoryRouter>  
        <BookListItem book={book} />  
      </MemoryRouter>  
    )  
    .toJSON();  
  expect(tree).toMatchSnapshot();  
});
```

Snapshots

On first execution the html result is written to a file

__tests__

__snapshots__

BookListItem.test.tsx.snap

BookListItem.test.tsx

```
exports[`components/BookListItem renders a
book item correctly`] = `
<a
  href="/books/undefined"
  onClick={[Function]}
>
  My first book
</a>
`;
```

Snapshots - working with errors

- If something changes you get an error with a diff
- You can now either
 - Fix the bug
 - Update all snapshots with `u` in the console
(or run `yarn test --updateSnapshot`)

Received value does not match stored snapshot 1.

- Snapshot
+ Received

@@ -1,3 +1,3 @@

- React Advanced
+ React Beginner

Snapshots - working with errors

- If something changes you get an error with a diff
- You can now either
 - Fix the bug
 - Update all snapshots with `u` in the console(or run `yarn test --updateSnapshot`)

```
● components/BookList > renders a booklist correctly

expect(received).toMatchSnapshot()

Snapshot name: `components/BookList renders a booklist correctly 1`

- Snapshot - 0
+ Received + 8

@@ -2,17 +2,25 @@
  <li>
    <a
      href="/books/undefined"
      onClick={[Function]}
    >
+   <div
+     className="booklist__item"
+   >
      My first book
+   </div>
    </a>
  </li>
  <li>
    <a
      href="/books/undefined"
      onClick={[Function]}
+   >
+     <div
+       className="booklist__item"
+     >
      My second book
+     </div>
    </a>
  </li>
</ul>

19 |     )
20 |     .toJSON();
> 21 |     expect(tree).toMatchSnapshot();
    |                   ^
22 |   });
23 |
24 |   test("renders all books", () => {

at Object.<anonymous> (src/components/BookList/BookList.test.tsx:21:18)

> 1 snapshot failed.
> 1 snapshot obsolete.
```

Task

Snapshot Tests



Testing Interactivity

fireEvent

<code>

Fire DOM events via the built in utility.

```
import { fireEvent, screen } from "@testing-library/react";

// click event
fireEvent.click(screen.getByText("reset"));

// change event
fireEvent.change(screen.getByPlaceholderText(/email/i), {
  target: { value: "abc@def.gh" },
});
```

@testing-library/user-event

- Companion library for Testing Library simulating user interactions.
- Closer to real interactions in a browser as when using fireEvent.
- Needs to be installed separately:

```
npm install --save-dev @testing-library/user-event
```

```
yarn add --dev @testing-library/user-event
```


@testing-library/user-event

<code>

Fire DOM events via the built in utility.

```
import userEvent from "@testing-library/user-event";

it(`handles a click event`, async () => {
  const user = userEvent.setup()
  const mockOnSave = jest.fn()

  render(<BookForm onSave={mockOnSave} />)

  await user.click(screen.getByRole('button', {name: /save book/i}))
  expect(mockOnSave).toHaveBeenCalled()
});
```

@testing-library/user-event

<code>


Fire DOM events via the built in utility.

```
import userEvent from "@testing-library/user-event";

it(`handles a click event`, async () => {
  const user = userEvent.setup()
  const mockOnSave = jest.fn()

  render(<BookForm onSave={mockOnSave} />)

  await user.click(screen.getByRole('button', {name: /save book/i}))
  expect(mockOnSave).toHaveBeenCalled()
});
```



@testing-library/user-event

<code>

Fire DOM events via the built in utility.

```
import userEvent from "@testing-library/user-event";
```

```
it(`handles a click event`, async () => {
```

```
  const user = userEvent.setup()
```

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

```
  render(<BookForm onSave={mockOnSave} />)
```

Render our component
with a mock event handler.

```
  await user.click(screen.getByRole('button', {name: /save book/i}))
```

```
  expect(mockOnSave).toHaveBeenCalled()
```

```
});
```

@testing-library/user-event

<code>

Fire DOM events via the built in utility.

```
import userEvent from "@testing-library/user-event";
```

```
it(`handles a click event`, async () => {
```

```
  const user = userEvent.setup()
```

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

```
  render(<BookForm onSave={mockOnSave} />)
```

```
  await user.click(screen.getByRole('button', {name: /save book/i}))
```

```
  expect(mockOnSave).toHaveBeenCalled()
```

```
});
```

Perform a click on an element we select from the screen.

@testing-library/user-event

<code>

Fire DOM events via the built in utility.

```
import userEvent from "@testing-library/user-event";

it(`handles a click event`, async () => {
  const user = userEvent.setup()
  const mockOnSave = jest.fn()

  render(<BookForm onSave={mockOnSave} />)

  await user.click(screen.getByRole('button', {name: /save book/i}))
  expect(mockOnSave).toHaveBeenCalled()
});
```

Assert our mock event handler was called.

Task

**Component Tests
with interactivity**



Advanced topics

Jest: Setup and Teardown

<code>

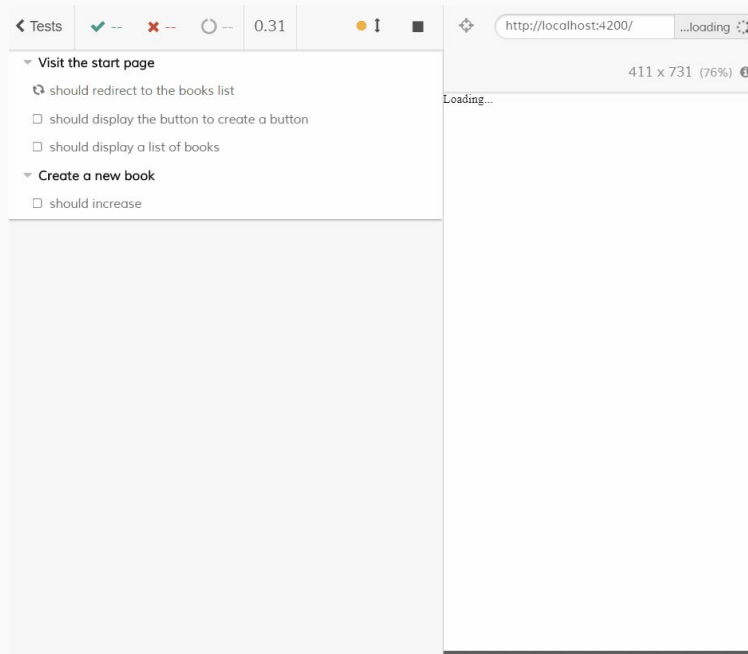
helper functions to setup work that needs to happen before tests run and finishing work that needs to happen after tests run.

```
beforeAll() // is run once before all the tests in a describe  
afterAll()  // is run once after all the tests in a describe
```

```
beforeEach() // is run before each test in a describe  
afterEach()  // is run after each test in a describe
```


End-to-end tests with Cypress

Cypress Test Runner





We teach.

workshops.de