Workshop

Testing in Jest

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

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()`
- `toEqual()`
- `toContain()`
- `toBeUndefined()`
- `toBeTruthy()`
- `toThrow()`

→ `toBeGreaterThan()`

→ `toBeLessThan()`

→ `toBeCloseTo()`

You can also create your own matchers.

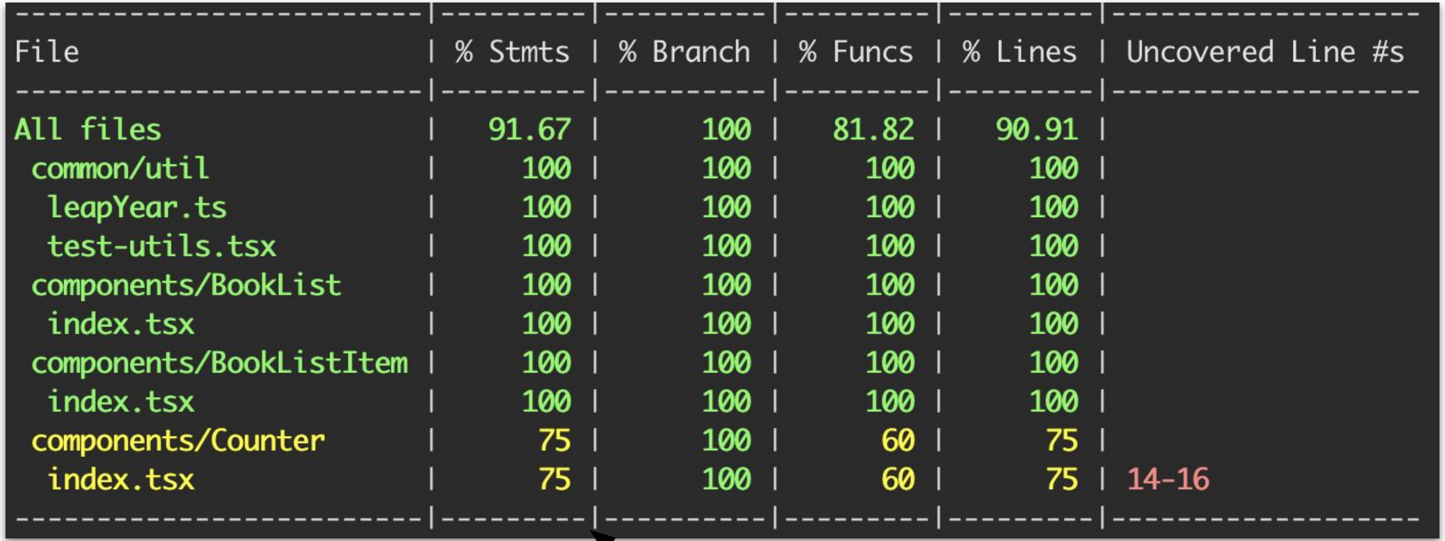
Code coverage

- ***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

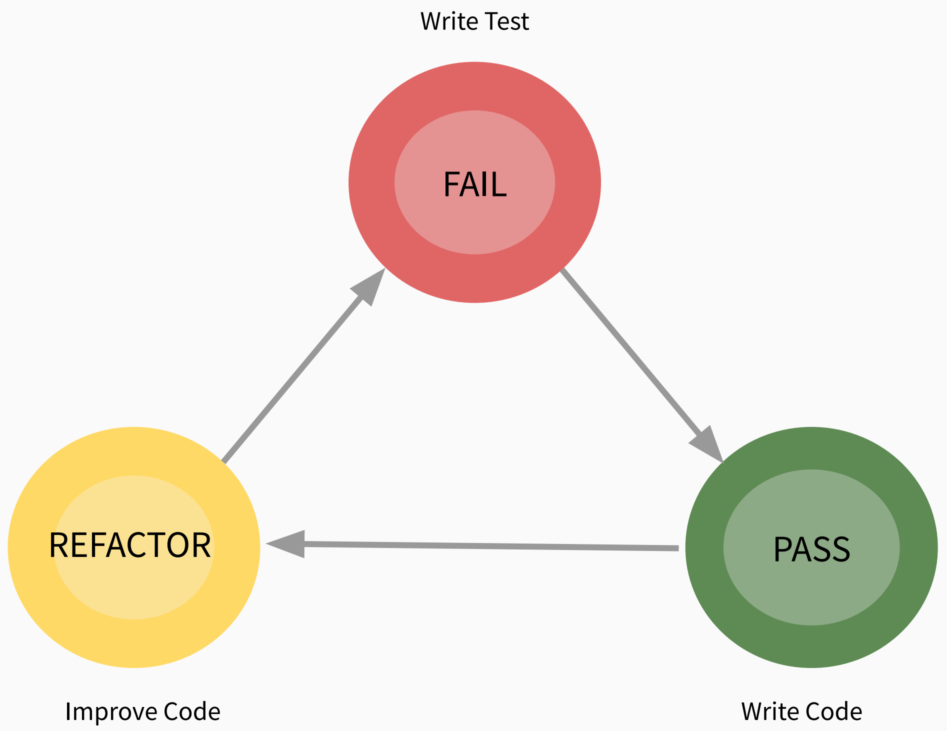
Example output

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)

We teach.

workshops.de