

Testen im Frontend

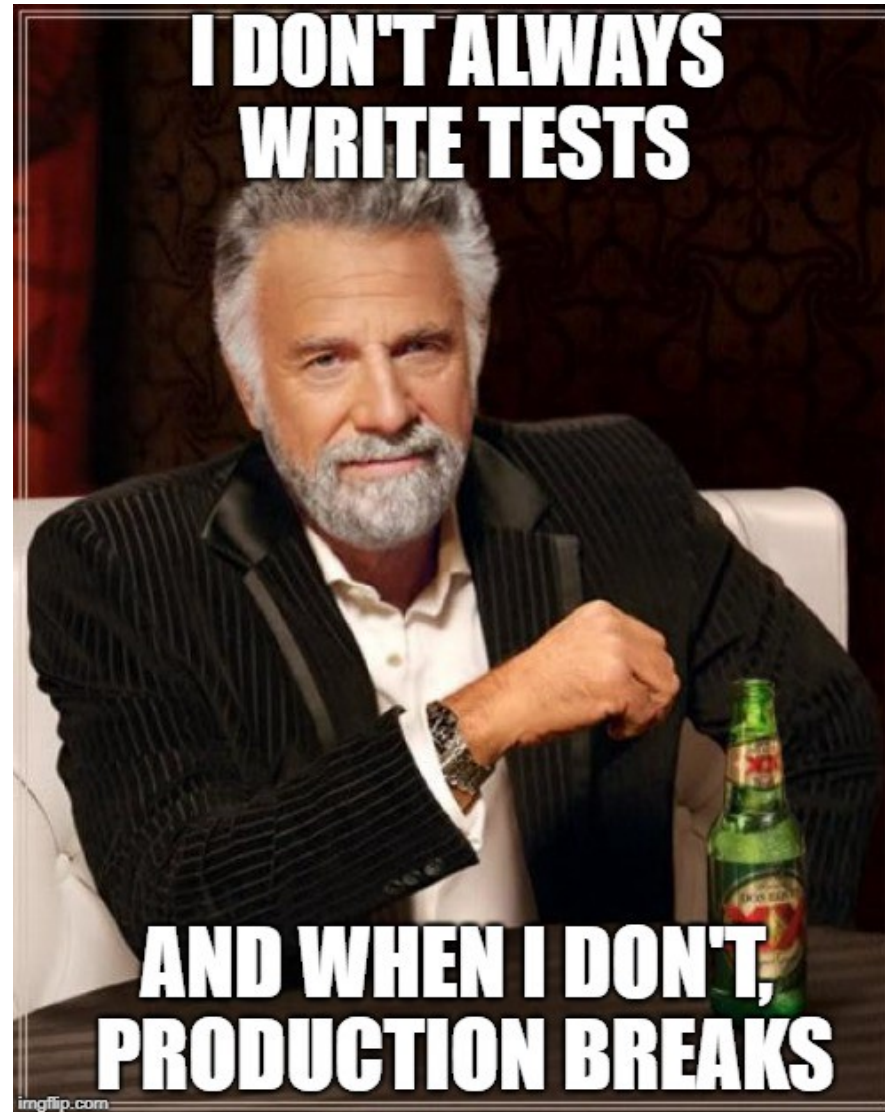
Pablo Klaschka



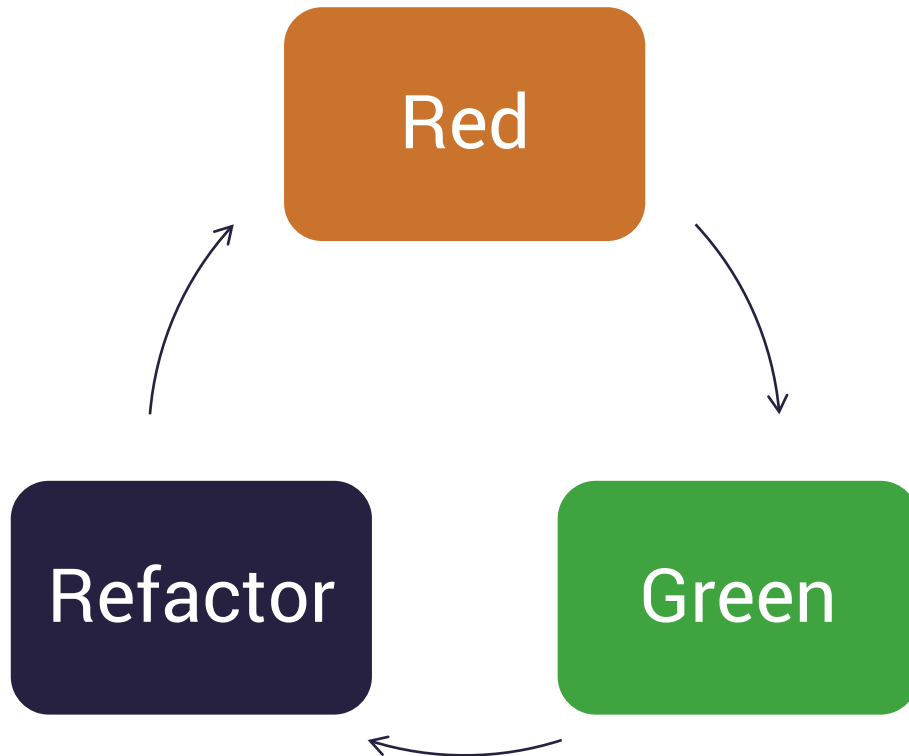
Warum Testen? Ist doch so viel Arbeit

Ihr erinnert euch schon, dass wir Software für ein Raumflug-Projekt schreiben? Die sollte einigermaßen zuverlässig funktionieren. Und bei kurzfristigen Probleme ist es ziemlich viel Wert, durch Tests ohne hohes Risiko schnelle Eingriffe im Code vornehmen zu können, ohne dass am Ende alles auseinanderfällt! Testen spart Zeit, glaubts oder glaubts nicht.





TDD Workflow



1. Tests für gewünschtes Verhalten schreiben. Ohne Implementierung failen diese => Red
2. Implementierung schreiben. Am Ende funktioniert es => Tests erfolgreich => Green
3. Refactor => Something Breaks
4. The cycle restarts
5. Beim Launch sollte alles Green sein (Ground Station: Go)!



JEST: Testing Framework für JavaScript

```
myFunction.spec.js

describe('myFunction', () => {
  it('should return 1', () => {
    expect(myFunction()).toBe(1)
  })
})
```

- JUnit für JavaScript
- auch mit Typescript nutzbar
- Einfach in der Handhabung
- gut anpassbar und erweiterbar
- „Englisch“: „Describe component it should do something so expect x to be y“

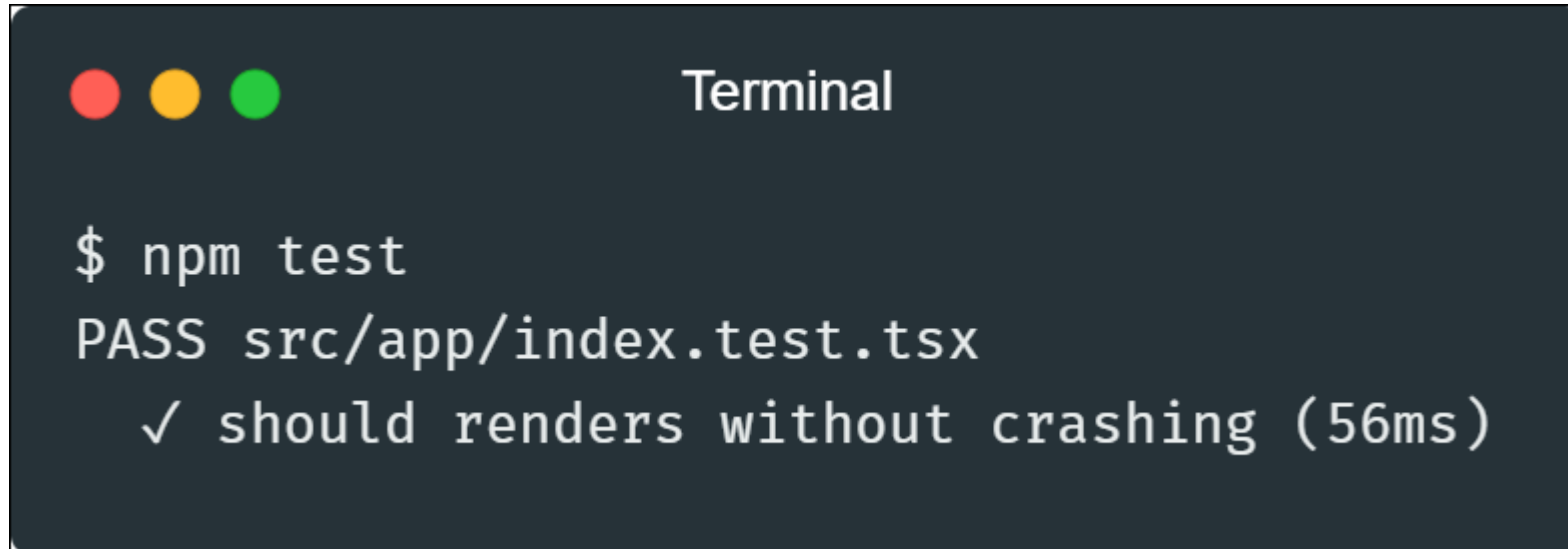


Grundregeln von Tests in React

- Tests befinden sich neben getesteten Dateien in `.spec.ts`-Dateien
Zum Beispiel: `my-func.ts` wird getestet in Datei `my-func.spec.ts`
- Es soll ausschließlich diese eine Datei getestet werden. Externe (importierte) Funktionalität sollte durch Mocks ersetzt werden



Ausführen von Tests in React-Apps



```
$ npm test
PASS src/app/index.test.tsx
  ✓ should renders without crashing (56ms)
```



Mocks

```
__mocks__/example-api.ts

// Calls cb (callback) function with "API"-Data
export default function getAPIData(cb: function) {
  // Simulate delay in API of 500ms:
  setTimeout(() => cb({
    // Fake data
    weather: 'clear',
    temperature: 32
  }), 500);
}
```

- ermöglicht isoliertes Testen (Fehlschlagen eines Tests bedeutet Fehler in exakt diesem Modul)
- Mocks “simulieren” das Verhalten von Modulen (bzw. ihren exportierten Members) ohne tatsächliche Anbindung an DBs, APIs, etc.
- Ein Mock für a.js liegt in `__mocks__/a.js` und wird im Test durch `jest.mock('./a')` aktiviert



Snapshot Testing

- Schnelle Möglichkeit, funktionierenden Code abzusichern
- Vergleicht Wert mit früherem, abgespeicherten, commiteten Wert
- `expect(value).toMatchSnapshot()`
- Besonders nützlich für das Testen von komplexeren Rückgabewerten (wie die von Components), wo eine Überprüfung der commiteten Snapshot-Werte beim ersten Testen einfacher ist



Und wie testet man jetzt React?

```
component.spec.ts

import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

- Mit dem react-test-renderer
- Ersetzt ReactDOM (statt in die DOM rendern wir das Component jetzt in den Test Renderer)
- vgl. <https://reactjs.org/docs/test-renderer.html>



Und was ist mit dem Frontend als Ganzes

Unit-Tests

- Testen einzelne, isolierte Komponenten/Dateien
- Interagieren nicht mit externen Komponenten wie Datenbanken u.Ä.
- Bilden die Applikation auf funktionaler, nicht aber auf Anwender-Ebene ab

E2E-Tests

- Testen die App als Ganzes
- Bilden die App aus Anwendersicht ab (erst anmelden, dann den Button klicken, dann expected dass Dashboard X geladen wurde)
- Mit Cypress umsetzbar
- Fehler sind nicht einfach lokalisierbar, Tests testen „nur“ Funktionieren der Gesamt-App



Cypress Example (Quoted from Cypress docs)

```
Cypress Example Test

describe('My First Test', () => {
  it('Gets, types and asserts', () => {
    cy.visit('https://example.cypress.io')

    cy.contains('type').click()

    // Should be on a new URL which includes '/commands/actions'
    cy.url().should('include', '/commands/actions')

    // Get an input, type into it and verify that the value has been updated
    cy.get('.action-email')
      .type('fake@email.com')
      .should('have.value', 'fake@email.com')
  })
})
```

- Visit: <https://example.cypress.io>
- Find the element with content: type
- Click on it
- Get the URL
- Assert it includes: /commands/actions
- Get the input (text field) with the .action-email class
- Type `fake@email.com` into the input
- Assert the input reflects the new value

<https://docs.cypress.io/guides/getting-started/writing-your-first-test.html#Step-4-Make-an-assertion>



Vielleicht wäre das ja was fürs SED?

```
sed.spec.ts

describe('SED', () => {
  describe('Mechanical Drawing', () => {
    it('should include the box', () => {
      expect(drawings).toContain('the box');
    });

    it('should highlight the box', () => {
      expect(box.highlighted).toBeTruthy();
      expect(box.highlightColor.brightness).toBeGreaterThan(100);
    });
  });
});
```

