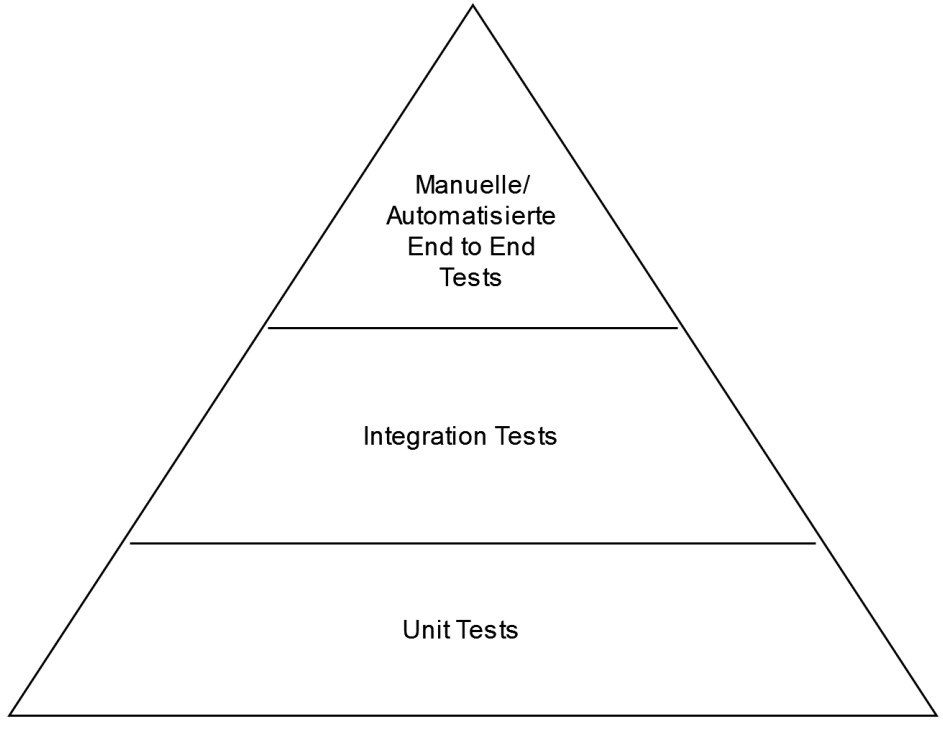
RICH CLIENT REACT TESTING

LERNZIELE

- Welche verschiedenen Arten von Tests gibt es?
- Wie schreibe ich gute Unit Tests (in Javascript)?
- Wie schreibe ich Unit Tests für ein React Frontend?

ARTEN VON TESTS

TEST PYRAMIDE



UNIT TESTS

- automatisierte Tests
- Testen der kleinsten Einheiten
- auf sehr detaillierter Ebene
- kurze Laufzeit

INTEGRATION TESTS

- automatisierte Tests
- Testen zusammenhängender Teile der Anwendung
  - ein Backend Service (ohne Frontend)
  - ein Frontend (ohne Backend)
- weniger Detailtiefe
- Fokus liegt auf
  - wichtigen Szenarien
  - interessanten Edge-Cases
  - Fehlern die aufgetreten sind
- etwas längere Laufzeit

MANUELLE/AUTOMATISIERTE END TO END UI TESTS

- manuelle oder automatisierte Tests
- Testen über das richtige UI
- Testen der gesamten Software
- Styling erfordert manuelle Tests
- lange Laufzeit (besonders für einen Mensch)

WIESO SCHREIBEN WIR UNIT TESTS?

- kleine Tests sind übersichtlicher
- test driven development
  - schnelles Feedback
  - vermeidet Seiteneffekte
- Components vielseitig Einsetzbar
- lebende Dokumentation

WIE SCHREIBEN WIR UNIT TESTS?

- Component muss isoliert werden
- z.B. mit Dependency Injection
- Schnittstellen werden "gemockt"
- Childcomponents werden "gemockt"

WIE SCHREIBE ICH GUTE UNIT TESTS?

WAS SOLLTEN WIR IM FRONTEND TESTEN?

- Logik in unseren Components
- dynamisches rendering in Components
- weitere Javascript Logik (TodoHttpClient)

WIE SOLLTEN WIR EINE COMPONENT TESTEN?

- Nutzer interagieren mit Buttons und Textfeldern ...
- ... nicht mit Javascript Funktionen
- am besten immer End to End

TESTING MIT JAVASCRIPT (JASMINE)

```
 1 describe('ich bin eine Beschreibung', () => {
 2     beforeAll(() => {});
 3
 4     beforeEach(() => {});
 5
 6     it('ich bin ein Test', () => {
 7         expect(actual).toEqual(expect) // quasi ein assert
 8     });
 9
10     afterEach(() => {});
11
12     afterAll(() => {});
13 });
```

SAUBERER AUFBAU VON JAVASCRIPT TESTS

TESTBESCHREIBUNG

- sollte einem Schema folgen
  - z.B. "Object ... should ... when"
  - gerne auch andere Schema's
  - viele gehen in ähnliche Richtung
- oft hilft es einen Satz zu bilden
- Testbeschreibung als lebende Doku

```
1 it('ComponentUnderTest should show element-card
2             when element-data is not empty', () => {});
```

TESTBESCHREIBUNG

- sollte wenig Duplizierungen enthalten
- damit entsteht eine saubere Struktur

```
 1 describe('ComponentUnderTest', () => {
 2   describe('updateData()', () => {
 3     it('should update data when data is not empty',
 4         () => {});
 5
 6     it('should not update data when data is empty',
 7         () => {});
 8
 9     ...
10   });
11
12   ...
13 });
```

SINGLE RESPONSIBILITY PRINCIPLE

- jeder Test sollte nur eine Sache testen
  - am besten ein "expect" pro Test
  - macht es einfacher eine Testbeschreibung zu finden
- im Fehlerfall ist das Problem schneller erkannt

CODEDUPLIZIERUNG (IN TESTS)

- ist ein kontroverses Thema
- kann trotzdem vermieden werden
- setup Code kann in "beforeEach"/"beforeAll"
- parametrisierte Tests
  - gleicher Test mit unterschiedlichen Parametern
  - weniger Codeduplizierung

```
1 it.each([
2   {input: 'input1', expected: 'expected1'},
3   {input: 'input2', expected: 'expected2'},
4   ...
5 ])('should to something', ({input, expected}) => {
6    // code for testing
7 });
```

REPRODUZIERBAR

- Tests müssen reproduzierbar sein
- "date.now()"?
  - Produktivcode ist abhängig vom aktuellen Datum
  - Testcode muss damit auch vom aktuellen Datum abhängig sein
  - typischer Aprilscherz

UNIT TESTS IN REACT

REACT TESTING LIBRARY

- wir benutzen die React Testing Library
- natives Testing enthält sehr viel Boilerplate
- die Testing Library stellt auch eine einfachere API bereit

RENDER()

- zum Rendern der Component

```
1 it('some test', () => {
2     render(<Button dataTestId={buttonDataTestId}
3                    label={buttonLabel}/>);
4 });
```

SCREEN

- zum Abrufen von gerenderten Inhalten

```
1  it('some test', () => {
2      render(<Button dataTestId={buttonDataTestId}
3                     label={buttonLabel}/>);
4
5      expect(screen.getByText(buttonLabel))
6          .toBeInTheDocument();
7  });
```

SCREEN FUNKTIONEN

- verschiedene Funktionen, um Inhalt zu suchen
- getBy... wirft einen Fehler wenn (Element != 1)
- queryBy... gibt null zurück
- findBy... gibt ein Promise zurück

```
 1 it('some test', () => {
 2     ...
 3
 4     expect(screen.getByRole(Button))
 5         .toBeInTheDocument();
 6
 7     expect(screen.getByText(buttonLabel))
 8         .toBeInTheDocument();
 9
10     expect(screen.getByTestId(buttonDataTestId))
11         .toBeInTheDocument();
12 });
```

FIREEVENT

- hilft uns beim triggern von Events

```
 1 it('some test', () => {
 2       ...
 3
 4       fireEvent.click(screen.getByTestId(buttonDataTestId));
 5
 6       fireEvent.change(screen.getByTestId(inputFieldDataTestI
 7                         { target: {value: 'new text'}, });
 8
 9       ...
10 });
```

JEST.FN()

- mocken von Funktionen

```
 1 it('some test', () => {
 2     const onClick = jest.fn();
 3
 4     render(<Button dataTestId={buttonDataTestId}
 5                    label={buttonLabel}/>);
 6
 7     fireEvent.click(screen.getByTestId(buttonDataTestId));
 8
 9     expect(onClick).toHaveBeenCalledTimes(1);
10 });
```

PRAXIS: BUTTON TEST

- schreibt einen Test für die Button Component
- was sollten wir testen?
- https://gitlab.com/dhbw_webengineering_2/rich_client_react_test
- Branch: step_0-button_test

NEXT STEP: TESTEN EINES LIST VIEW ITEMS

- isoliertes Testen?
- Child Components haben eigene Tests

```
1 export default function ListViewItem({ todo, onShowDetail, d
2     return (
3         <div className='list-view-item' data-testid={dataTes
4             <p className='list-view-item--title'>{todo.title
5             <InputCheckboxGroup className='list-view-item--c
6             <Button className='list-view-item--' label='Deta
7         </div>
8     );
9 };
```

CHILD COMPONENTS MOCKEN

- jest.mock erlaubt es uns Imports zu mocken
- wir überschreiben nun die Component

```
1 jest.mock('../../molecules/.../InputCheckboxGroup', () => {
2     return function DummyInputCheckboxGroup(props) {
3         return <div>{props.id},{props.checked.toString()}</d
4     }
5 });
```

PRAXIS: LIST VIEW ITEM TEST

- schreibt einen Test für die ListViewItem Component
- was muss getestet werden?
- mockt die Child Components
- Branch: step_1-list_view_item_test

ROUTING IM TEST

- Router muss vorhanden sein

```
export const reactRouterTestWrapper = (ui) => {
    return (
        <MemoryRouter>
            <Routes>
                <Route path="/*" element={ui} />
            </Routes>
        </MemoryRouter>
    )
}
```

```
render(
    reactRouterTestWrapper(
        <SomeComponent></SomeComponent>
    )
);
```

ROUTING IM TEST

- navigate sollte gemockt werden
- das Routing sollten wir prüfen

```
1 const navigate = jest.fn();
2
3 beforeEach(() => {
4     jest.spyOn(router, 'useNavigate')
5         .mockImplementation(() => navigate);
6 });
```

```
1 expect(navigate).toHaveBeenCalledTimes(1);
2 expect(navigate).toHaveBeenCalledWith(`/list`);
```

CONTEXT MOCKEN IM TEST

- TodoHttpClient muss gemockt werden
- für isoliertes Testing

```
let todoHttpClientMock;

beforeEach(() => {
    todoHttpClientMock = {
        getTodoById(_) {
            return Promise.resolve(todo);
        },
        saveTodo(todo) {
            return Promise.resolve(todo);
        }
    }
});
```

CONTEXT MOCKEN IM TEST

- TodoHttpClient wird über den Context provided

```
1 function renderWithContextProvider(ui) {
2     render(
3         <TodoHttpClientContext.Provider
4                 value={todoHttpClientMock}>
5             {ui}
6         </TodoHttpClientContext.Provider>
7     );
8 }
```

CONTEXT MOCKEN IM TEST

- Aufrufe prüfen

```
1  saveTodoSpy = jest.spyOn(todoHttpClientMock, 'saveTodo');
```

```
1  expect(saveTodoSpy).toHaveBeenCalledTimes(1);
2  expect(saveTodoSpy).toHaveBeenCalledWith(todo);
```

WAITFOR

- warten auf asynchronen Code
- z.B. bei Backendcalls

```
1 it('should render', async () => {
2     render(<DetailView dataTestId={detailViewDataTestId} />)
3
4     await waitFor(() => {
5         expect(screen.getByTestId(testId)).toBeInTheDocument
6     });
7 });
```

WAITFOR MIT FIREEVENT

- Button muss sichtbar sein, bevor er geklickt werden kann
- die Kombination ist tricky
- FireEvent darf nicht in WaitFor aufgerufen werden

```
1 await waitFor(() => {
2     expect(screen.getByTestId(buttonId)).toBeInTheDocument()
3 });
4 fireEvent.click(screen.getByTestId(buttonId));
```

ASSERTIONS IN WAITFOR

- nur einzelne Assertions erlaubt
- führt zu schnellerer Testausführung

```
1 await waitFor(() => {
2     expect(saveSpy).toHaveBeenCalledTimes(1);
3 });
4 expect(saveSpy).toHaveBeenCalledWith({ ...todo, done: true }
```

PRAXIS: DETAIL PAGE TEST

- schreibt einen Test für die Detail Page Component
- was sollten wir testen?
- Branch: step_2-detail_page_test

TODOHTTPCLIENT?

- ausgelagerte Logik muss auch getestet werden
- axios muss gemockt werden

```
1 jest.mock("axios");
2
3 it('should get all todos', async () => {
4     axios.get.mockResolvedValueOnce({data: mockTodos});
5
6     ...
7 });
```

AUFRUFE VERIFIZIEREN

```
1 it('should get all todos', async () => {
2     ...
3
4     expect(axios.get).toHaveBeenCalledTimes(1);
5     expect(axios.get).toHaveBeenCalledWith(url, headers);
6 });
```

PRAXIS: TODO HTTP CLIENT TEST

- Test schreiben für den Client
- was wollen wir testen?
- Branch: step_3-todo-http-client

LERNZIELE

- Welche verschiedenen Arten von Tests gibt es?
- Wie schreibe ich gute Unit Tests (in Javascript)?
- Wie schreibe ich Unit Tests für ein React Frontend?