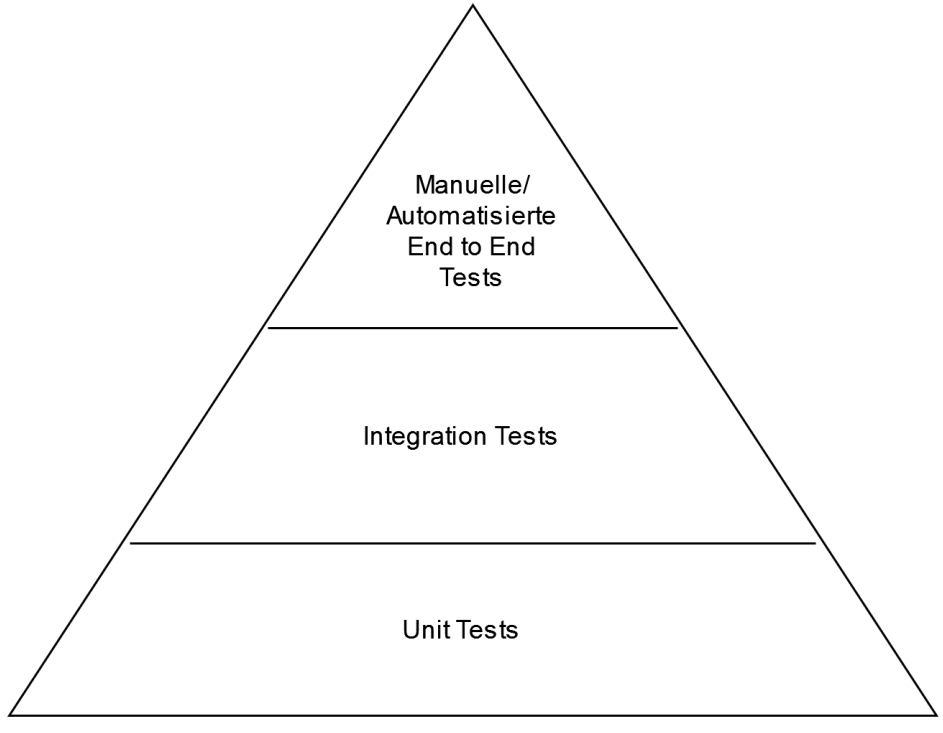
RICH CLIENT REACT TESTING

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

Speaker notes

- Unit Tests müssen automatisiert laufen.
- Wir testen die kleinsten Einheiten des Systems. Z.B. eine Klasse oder eine Component.
- Da wir die alle Einheiten einzeln testen, können wir auf sehr detaillierter Ebene testen.
- Unit Tests haben üblicherweise eine kurze Laufzeit, da nicht eine vollständige Anwendung hochgefahren werden muss.
- Bei einem Frontend wird oft auch kein richtiger Browser verwendet, sondern ein "headless" Browser, der das HTML nicht rendered.
- Unit Tests sollte auch schnell laufen, da sie das schnellste Feedback beim Entwickeln geben.

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

Speaker notes

- Meist wird der Code bei einem Integration-Test in einer Testumgebung getestet.
- Ein Backend oder ein Frontend wird hochgefahren, die Schnittstellen werden gemockt.
- Anschließend werden die Schnittstellen des Frontends/Backends maschinell angesprochen. Z.B. ein Bot, der eine Web API aufruft.
- Da es mehr Fälle geben kann, muss man sich auf wichtige Szenarien beschränken.
- Oft lohnt es sich merkwürdige Fehler die aufgetreten sind über einen Integration-Test zu testen, um den Fehler in Zukunft zu verhindern.
- Meist haben Integration-Tests eine höhere Laufzeit, da erst der Service oder das Frontend hochgefahren werden muss.

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)

Speaker notes

- Test auf der realen Umgebung.
- Computer oder Mensch klickt sich durch die Anwendung und prüft ob sie richtig funktioniert.
- Ein Computer kann nicht entscheiden ob etwas gut oder schlecht aussieht. Daher kommt es beim Styling immer noch auf den Mensch an.
- Besonders bei manuellen Tests ist die Laufzeit sehr hoch, da ein Mensch nur begrenzt schnell klicken kann.
- Außerdem kann man Menschen nicht duplizieren und die Tests parallel ausführen.
- Aber auch für einen Bot dauert es länger das Frontend zu klicken, da immer auf eine richtige Antwort aus dem Backend gewartet werden muss und das Frontend neu rendern muss.

WIESO SCHREIBEN WIR UNIT TESTS?

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

Speaker notes

- Kleine Tests sind wie kleine Components übersichtlicher. Daher sollten wir Unittesten.
- TDD bietet uns bei der Entwicklung schnelles Feedback und das vermeiden von Seiteneffekten beim anpassen von Features.
- Statt ein Feature jedes mal manuell über die UI zu testen, kann man beim TDD einfach schnell die Tests ausführen.
- Components müssen eigenständig getestet sein, da sie auch eigenständig eingesetzt werden können.
- Tests können gut als lebende Dokumentation dienen.

WIE SCHREIBEN WIR UNIT TESTS?

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

Speaker notes

- Über Dependency Injection können wir externe Abhängigkeiten einer Component mocken.
- Mocken heißt, dass wir die reale Implementierung gegen eine Testimplementierung (einen Mock) austauschen.
- Childcomponents können wir auch mocken, da diese ihren eigenen Unittest haben.
- Wir testen die Component nun über alle Schnittstellen. Was kommt rein, was geht raus. Später im Detail mehr.

WIE SCHREIBE ICH GUTE UNIT TESTS?

WAS SOLLTEN WIR IM FRONTEND TESTEN?

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

Speaker notes

- Natürlich sollten wir die Logik und das dynamische Rendering in unseren Components testen.
- Z.B werden die Daten gerendert, die ich hineingebe, wird eine Callback aufgerufen, wenn ein Button gedrückt wird.
- Außerdem sollten wir alle weitere Javascript Logik testen. Wie z.B. ob unser TodoHttpClient die korrekten HttpCalls auslöst.

WIE SOLLTEN WIR EINE COMPONENT TESTEN?

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

Speaker notes

- Nutzer interagieren üblicherweise mit dem HTML und nicht mit dem Javascript Code direkt.
- Unser Test sollte ebenfalls Events im HTML auslösen und nicht direkt die Javascript Funktionen aufrufen.
- Damit sind wir möglichst nahe an der Realität und wir vermeiden mehr Fehler.
- Z.B. könnte die Javascript Funktion korrekt funktionieren, sie wird aber nicht im HTML aufgerufen.
- End to End heißt dabei von den Eingabeparametern und anderen Schnittstellen der Component bis zum HTML und zurück.
- Z.B. wird im HTML das angezeigt, was ich als Parameter hineingegeben habe?
- Z.B. wird die der TodoHttpClient mit den richtigen Parametern aufgerufen, wenn der Save Button geklickt wurde?

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 });
```

Speaker notes

- JavaScript Test werden normalerweise mit Jasmine geschrieben.
- Wir arbeiten in React mit Jest, welches wiederum auf Jasmine aufsetzt und uns eine bequemere API bereitstellt.
- Jasmine liefert verschiedene Tools, um Tests strukturierter aufzubauen und einfacher zu gestalten.
- "describe" beschreibt einen Test und kann mehrere Tests logisch zusammenfassen.
- "beforeAll" bekommt eine Callback Function, die vor allen Tests in diesem "describe" Block ausgeführt wird.
- "beforeEach" bekommt eine Callback Function, die vor jedem Test in diesem "describe" Block ausgeführt wird.
- "it" ist ein einzelner Test. Er hat eine Beschreibung und eine Callback Function in der Testcode stehen sollte.
- "expect" ist quasi ein assert aus Java. Bietet einige Hilfsmethoden wie .toEqual() um das Testen zu vereinfachen.
- "afterEach" und "afterAll" erklären sich glaube ich selbst.

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', () => {});
```

Speaker notes

- Ein Schema sorgt dafür, dass sich andere Entwickler besser zurechtfinden. Und auch man selbst hat irgendwann vergessen was hier in diesem Code passiert.
- Ein korrekter Satz als Testbeschreibung macht es oft einfacher den Testfall zu verstehen. Beim Schreiben von Textnachrichten, lassen wir auch nicht einfach irgendwelche Wörter weg.
- Über gute Testbeschreibungen kann man sehr gut erkennen wie sich Components verhalten sollen und was ihre Aufgabe ist.

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 });
```

Speaker notes

- In dem wir die Testbeschreibung aufteilen und nicht duplizieren, sorgen dir automatisch dafür, dass unser Test eine gute Struktur erhält.
- Zusammengehörige Testfälle für z.B. eine Methode werden im gleichen describe zusammengefasst.

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

Speaker notes

- Wenn der Test nur eine Sache testen, ist im Fehlerfall schneller erkennbar was kaputt ist. Denn es steht ja in der Testbeschreibung.
- Es kann natürlich auch passieren, dass der Test wegen Setup Code fehlschlägt, dieser Setup Code sollte allerdings auch in einem eigenen Test fehlschlagen.

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 });
```

Speaker notes

- Einige Entwickler sehen Codeduplizierung im Test nicht als Problem.
- Manchmal kann es mehr Übersichtlichkeit schaffen Code und Daten zu duplizieren.
- Z.B. kann man Testdaten über mehrere Tests nutzen. Manche Tests wären einfacher zu verstehen, wenn die Testdaten direkt im Test stehen würden.
- Trotzdem gibt es gute Gründe und auch mechanismen, um Codeduplizierung im Test zu vermeiden.
- Setup Code ist oft dupliziert und sollte daher in entsprechende Blöcke ausgelagert werden.
- Parametrisierte Tests helfen, wenn der Testcode dupliziert wird.

REPRODUZIERBAR

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

Speaker notes

- Tests müssen immer laufen, egal was der Kontext ist in dem sie sich befinden.
- Tests können fehlschlagen, wenn sich das Datum ändert und der Testfall statische Daten anstatt dynamischen verwendet.
- Bei uns sind mal Tests am ersten April gefailed. Hier hat sich ein Entwickler einen Spaß erlaubt.

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

Speaker notes

- Wer sich die klassische Syntax ansehen möchte, kann dies natürlich gerne selbst machen: https://reactjs.org/docs/testing.html

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 });
```

Speaker notes

- Das Screen Objekt bietet verschiedene Funktionen, um Inhalt im DOM zu finden.
- getBy... queryBy... und findBy... gibt es jeweils auch für mehrere Elemente heißt dann getByAll...
- Wir können dann nach Text, einer speziellen DataTestId oder nach einer Component suchen.
- Es gibt noch weitere Funktionen. Schaut dazu einfach in die Doku: https://testing-library.com/docs/react-testing-library/cheatsheet/
- expect kennen wir bereits -> ist quasi ein assert (von Java).
- expect bietet uns einige Funktionen, um Werte zu vergleichen, auf gewisse Eigenschaften zu prüfen, etc.

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 });
```

Speaker notes

- Auch hier gibt es natürlich weitere Funktionen. Diese brauchen wir erstmal nicht.

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 });
```

Speaker notes

- Jest Mock Funktionen speichern alle interaktionen mit ihnen.
- Wir können somit prüfen, ob die Funktion entsprechend oft und mit den richtigen Parametern aufgerufen wurde.

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 };
```

Speaker notes

- Wir wollen Components immer isoliert testen.
- Dazu gehören nicht nur Abhängigkeiten, wie z.B. unser TodoHttpClient, sondern auch Child Components.
- Die InputCheckboxGroup Component und die Button Component haben ihre eigenen Unit Tests.
- Wir wollen sie eigentlich nicht mittesten, sondern nur prüfen, ob die ListViewItem Component korrekt funktioniert.
- Daher testen wir nur bis zur Schnittstelle. Das heißt wir prüfen ob die korrekten Parameter in die Component hineingegeben wurden und was passiert, wenn ein Callback von einer Child Component aufgerufen wird.

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 });
```

Speaker notes

- Auch unsere Mock Component bekommt die "props" hineingereicht.
- Da wir prüfen wollen, ob die parameter korrekt sind, schreiben wir sie einfach ins HTML, dann können wir sie später auf ihre Korrektheit prüfen.

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>
    )
);
```

Speaker notes

- Components in denen Routing genutzt wird, müssen innerhalb eines Routers liegen.
- Daher müssen wir einen MemoryRouter um die Component legen.
- Da wir Routing bei mehreren Tests und Components brauchen, sollten wir es abstrahieren.
- Genutzt wird der Wrapper dann in der render Funktion.

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`);
```

Speaker notes

- Da wir prüfen wollen, ob das Routing korrekt ausgelöst wird, müssen wir die navigate Funktion mocken.
- Wir mocken daher den router.useNavigate() und geben eine eigene mock Funktion zurück.
- Mit dieser können wir dann prüfen, ob die richtigen Navigationen ausgelöst wurden.

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);
        }
    }
});
```

Speaker notes

- Über Dependency Injection (genau genommen Context Injection) können wir den TodoHttpClient nutzen.
- Dies machen wir, damit die HTTP Calls abstrahier von verschiedenen Stellen aus getätigt werden können und die Logik dafür nicht in jeder Component dupliziert ist.
- Außerdem können wir dadurch im Testfall den TodoHttpClient einfach mocken, damit keine echten Backendcalls gemacht werden.
- Den TodoHttpClient können wir mit jest mocken oder wie hier manuell mit Javascript.
- Wie ihr sehen könnt, müssen wir natürlich nur die Funktionen mocken, die wir im Test brauchen.

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 }
```

Speaker notes

- Über den Context.Provider können wir anschließend den MockClient verfügbar machen.
- Über den useContext Hook, greift die Component anschließend auf unseren MockClient zu.

CONTEXT MOCKEN IM TEST

- Aufrufe prüfen

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

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

Speaker notes

- Wie auch bei anderen Funktionen, können wir darauf "spyen".
- Anschließend können wir prüfen, ob die Schnittstellen der Methode korrekt aufgerufen werden.
- WIr wollen schließlich prüfen, ob die Component End to End funktioniert.

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 });
```

Speaker notes

- WaitFor ist eine Funktion, der wir einen Callback übergeben. Die Funktion wird so lange aufgerufen, bis der Callback keine Exceptions mehr zurückliefert oder der Timeout abgelaufen ist.
- getByTestId wirft eine Exception, falls die TestId nicht gefunden wurde. Dann wird waitFor erneut aufgerufen.
- Wir können der waitFor Funktion einen Timeout und ein Interval übergeben. Der default für Timeout ist 1000ms und für das Interval 50ms.
- Wir brauchen waitFor, um darauf zu warten, dass die Daten aus dem MockTodoHttpClient zurückgeliefert wurden und anschließend in der Component angezeigt werden.
- Auf unseren Pages brauchen wir daher waitFor, da wir sonst nur "loading" zu sehen bekommen würden.

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));
```

Speaker notes

- Manchmal müssen wir einen Button klicken, der erst sichtbar wird, nachdem etwas erfolgreich vom Backend geladen wurde.
- Die Kombination aus waitFor und fireEvent ist dabei nicht so einfach.
- Wenn wir das fireEvent direkt im waitFor aufrufen, kann es passieren, dass der Button mehrmals geklickt wird, da der Callback öfter ausgeführt werden kann.
- Das gilt natürlich für alle Seiteneffekte.
- Daher sollten wir erst prüfen, ob der Button angezeigt wird und erst danach, außerhalb des waitFor, den Button klicken.

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 }
```

Speaker notes

- Wir sollten nur eine Assertion im waitFor stehen haben, da unser Test im Fehlerfall schneller fehlschlägt, außerdem wird weniger Code wiederholt aufgerufen.
- Die Idee ist wieder: Wir warten bis die UI in einem stabilen Zustand ist und testen anschließend, dass das richtige passiert ist.
- Es ist auch sinnvoll die Assertions ins waitFor zu schreiben, die schneller fehlschlägt.
- Wenn wir "toHaveBeenCalledWith" in die waitFor Funktion gepackt hätten, müssten wir auf den Timeout warten, da die Funktion evtl. noch aufgerufen werden könnte.
- Wird die Funktion zweimal aufgerufen, wird waitFor direkt gestoppt, da die Assertion bereits fehlgeschlagen ist.

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 });
```

Speaker notes

- jest.mock kennen wir bereits. Wir können damit Components mocken, aber auch andere Javascript Objekte, wie z.B. axios.
- Anschließend müssen wir dem Mock Objekt natürlich sagen, wie es sich verhalten soll. Dazu bietet uns Jest verschiedene Möglichkeiten
- mockResolvedValueOnce() sorgt dafür, dass das Mock Objekt diesen Wert genau einmal zurückgibt.
- Außerdem müssen wir definieren, welche Funktion diesen Wert zurückgibt. Daher rufen wir vorher ".get" auf.

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 });
```

Speaker notes

- Wir sollten natürlich verifizieren, dass unser Mock Objekt mit den richtigen Werten aufgerufen wurde, schließlich wollen wir prüfen, ob unser Client auch das richtige macht.
- Dies machen wir wieder wie bei den Mock Funktionen zuvor in den Components. Mit expect können wir die Aufrufe prüfen.

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?

Speaker notes

- Es gibt Unit Tests, Integration Tests und automatisierte/manuelle End to End Tests.
- Sie unterscheiden sich z.B. im Detailgrad und der Ausführungsgeschwindigkeit.
- Es gibt verschiedene Techniken, mit denen ich meine Tests strukturierter aufbauen kann und für anderen Entwickler verständlicher mache.
- Unit Tests sollten die Component End to End testen.
- React und die ReactTestingLibrary bieten uns einigen Support beim Schreiben von Tests.
- Produktivcode muss testbar aufgebaut sein.