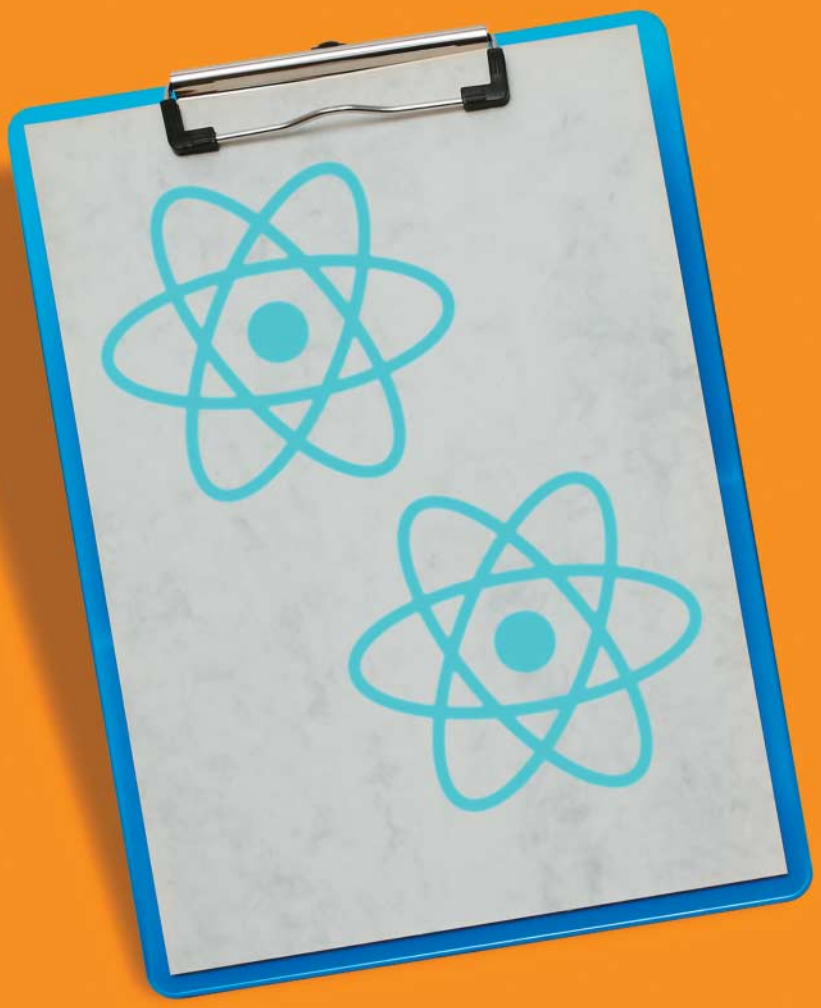
React-Anwendungen mit Jest und Enzyme testen

Seitentest

Manuel Ernst



Mit den Testwerkzeugen Jest und Enzyme können Entwickler einfach Unit-Tests für Webanwendungen erstellen, die das JavaScript-Framework React nutzen.

Das Frontend-Framework React, 2013 von Facebook veröffentlicht, ist neben Googles Angular und dem Open-Source-Projekt Vue.js heute die beliebteste Bibliothek zur Erstellung von Single-Page-Anwendungen. Kernmerkmal ist die hierarchische Gliederung in einzelne Komponenten, dank der Aktionen und Zuständigkeiten sauber getrennt sind: Jede Komponente ist für genau eine Aufgabe zuständig.

Eine besondere Herausforderung bei der Entwicklung grafischer Oberflächen besteht darin, diese Komponenten programmatisch zu validieren. Der generierte Code ist nicht einfach nur ein leicht zu überprüfender Wert, sondern bildet einen beliebig tief geschachtelten Baum. Deshalb sind spezielle Werkzeuge notwendig, um die gewünschten Tests durchzuführen.

Facebook und Airbnb haben das Test-Framework Jest und das Testwerkzeug Enzyme speziell für den Komponententest von React-Anwendungen konzipiert. Jest orchestriert und strukturiert Testsuiten und überprüft gewünschte Eigenschaften. Enzyme ergänzt Jest dabei, führt den zu testenden Frontend-Code aus und stellt die zugehörigen Schnittstellen zur Verfügung. Mit diesen Schnittstellen greift man auf die generierten Elemente zu und interagiert mit ihnen.

Dieser Artikel zeigt, wie sich Unit-Tests für React erstellen lassen. Anhand einer Musterkomponente werden React-Grundprinzipien erläutert, sodass sich die Beispiele auch ohne vertiefte React-Kenntnisse nachvollziehen lassen. Diese Komponente, ein simuliertes Thermometer, dient im weiteren Verlauf dieses Artikels außerdem zur Illustration verschiedener Testtechniken und Vorgehensweisen.

Thermometer und Temperatur

Der Komponente Thermometer wird über eine Property die aktuelle Temperatur zugewiesen, die sich fortlaufend ändert. Aufgabe der Komponente ist es, die aktuelle Temperatur anzuzeigen sowie über die auftretenden Werte Buch zu führen und zusammen mit der aktuellen die niedrigste und die höchste Temperatur im Messzeitraum auszugeben. Der Konstruktor initialisiert den internen Zustand der Komponente, der die niedrigste und die höchste bislang gemessene Temperatur festhält.

getDerivedStateFromProps ist ein spezieller Hook, also eine Schnittstelle, über die man auf den Lifecycle der Komponente Einfluss nehmen kann. Das Framework ruft diesen Hook immer dann auf,

wenn eine Elterninstanz aktualisierte Temperaturwerte übergibt. Sie erhält die neuen Properties als ersten Parameter, der zweite Parameter stellt den aktuellen Zustand dar. Der Rückgabewert der Funktion ist dann der neue State. Im Beispiel des Thermometers gleicht die Komponente die aktuelle Temperatur mit der gespeicherten Minimum- und Maximumtemperatur ab und aktualisiert diese gegebenenfalls.

Die *render*-Methode stellt die bisher gesammelten Werte im Browser dar. Dabei kommt JSX, ein JavaScript-Superset, zum Einsatz. Beim Build der Anwendung übersetzt ein spezieller Compiler JSX in JavaScript. Im Beispiel rendert die Komponente eine Überschrift mit dem Text „Thermometer“ sowie drei Sektionen mit den drei Temperaturwerten. Sobald eine übergeordnete Komponente einen neuen Temperaturwert übergibt, wird der daraus resultierende *state* ermittelt und die Komponente neu dargestellt.

React ist stark komponentenbasiert, die einzelnen Aufgaben und Darstellungen sollen in separate Komponenten ausgelagert werden. Dies ist auch beim Thermometer geschehen: Eine separate Komponente stellt den Temperaturwert innerhalb der Thermometerkomponente dar. Diese erhält über Properties das anzuzeigende Label sowie den Temperaturwert (siehe Listing 2), mit *round* gerundet auf zwei Stellen nach dem Komma. Label und gerundeter Wert stecken in einem *div*-Element, als dessen ID das in Kleinbuchstaben konvertierte Label dient. Dieses Element enthält das Label sowie den gerundeten Temperaturwert inklusive Einheit. Alle gezeigten Codebeispiele und Tests findet man unter ix.de/ix1902048.

Listing 1: Die Thermometerkomponente

```
import React, { Component } from 'react';
import Temperatur from './Temperatur';

export default class Thermometer extends Component {
  constructor() {
    super();
    this.state = {
      min: Infinity,
      max: -Infinity
    };
  }

  static getDerivedStateFromProps(props, state) {
    return {
      min: Math.min(props.temp, state.min),
      max: Math.max(props.temp, state.max)
    };
  }

  render() {
    return (
      <div id="thermometer">
        <h1>Thermometer</h1>
        <Temperatur, label="Min" value={this.state.min} />
        <Temperatur, label="Current" value={this.props.temp} />
        <Temperatur, label="Max" value={this.state.max} />
      </div>
    );
  }
}
```

Listing 2: Die Temperaturkomponente

```
class Temperatur extends React.Component {
  render() {
    const {label, value} = this.props;

    return (
      <div id={label.toLowerCase()} className="temp">
        {label}: {this._round(value)}°C
      </div>
    );
  }

  _round(value) {
    return Math.round(value * 100) / 100;
  }
}
```

Unit-Testing mit Jest

Die Werkzeuge Jest und Enzyme überprüfen die vorgestellten React-Komponenten auf Herz und Nieren. Jest strukturiert die Tests, führt Testsuiten aus und überprüft definierte Vorgaben.

Wenn man Jest in einem Projektverzeichnis startet, sucht der Test-Runner anhand eines festgelegten Musters die Testdateien, die ausgeführt werden sollen. Falls nicht anders definiert, sind das alle Dateien vom Typ *.js und *.jsx, die sich innerhalb eines Ordners mit dem Namen *__tests__* befinden. Von diesen Ordnern kann es beliebig viele an beliebiger Stelle innerhalb des Projektbaums geben. Außerdem sucht Jest standardmäßig nach Dateien mit dem Suffix *spec.js* oder *test.js* – das lässt sich bei Bedarf anpassen.

Um das Testen zu vereinfachen und zu beschleunigen, ermittelt Jest anhand der verwendeten Quellcodeverwaltung (hier Git), welche Dateien sich seit dem letzten Commit geändert haben. Anschließend werden nur die Tests ausgeführt, die mit diesen Dateien in Verbindung stehen. Dieses Verhalten kann man wenn nötig deaktivieren, sodass Jest immer alle Tests absolviert.

Strukturierung von Testsuiten

Jest implementiert eine domänenspezifische Sprache, um Testsuiten und Testfälle zu definieren. Eine Testsuite kann aus einzelnen Tests, aber auch aus mehreren Testuntergruppen bestehen. Die Funktionen *describe(title, handler)* und *it(title, handler)* nehmen zwei Parameter entgegen: einen Titel, der die Testsuite oder den Testfall beschreibt, und eine Handler-Methode, die die Logik enthält, die ausgeführt werden soll. Statt der Funktion *it()* kann man auch deren Alias *test()* verwenden, Parameter und Verhalten sind identisch.

Angenommen, es gilt, eine Methode zu testen, die eine Zahl durch eine andere Zahl teilt. Bei Divisionen ist es wichtig, dass die Parameter den richtigen Typ haben und dass der Teiler ungleich 0 ist. Eine Struktur der Testsuite könnte daher wie in Listing 3 aussehen.

iX-TRACT

- Jest ist ein Werkzeug zur Erstellung von Unit-Tests für JavaScript-Code. Enzyme ermöglicht den Zugriff auf die gerenderten Komponenten des JavaScript-Frameworks React.
- Die Kombination von Jest und Enzyme erlaubt ein bequemes und umfassendes Testen von React-Anwendungen.
- Jest stammt von React-Erfinder Facebook, Enzyme wird von Airbnb entwickelt. Beide Tools sind Open Source unter MIT-Lizenz.

Listing 3: Teststruktur

```
// testsuite 'division'
describe('division', () => {
  describe('parameters', () => {
    it('should throw if divisor is zero', () => {
      // testcode
    });

    it('should throw if parm 1 is not a number', () => {
      // testcode
    });

    it('should throw if parm 2 is not a number', () => {
      // testcode
    });
  });

  it('should divide parm 1 by parm 2', () => {
    // testcode
  });
});
```

Es ist möglich, innerhalb einer Testdatei auf die Untergliederung in Testsuiten zu verzichten und ausschließlich einzelne Testfälle zu definieren. Diese Vorgehensweise ist aber nicht empfehlenswert, da man die Testsuite so nicht mit einem sprechenden Titel versehen kann. Insbesondere bei Projekten mit vielen Tests geht dabei schnell die Übersicht verloren. Abbildung 1 zeigt einen erfolgreichen Durchlauf der Testsuite aus Listing 3.

Assertions prüfen Ergebnisse

Innerhalb eines *it()*-Blocks prüft das Test-Framework über einen Aufruf der Funktion *expect()* vorgegebene Bedingungen ab. *expect()* wird nie für sich alleine ausgeführt, sondern immer in Kombination mit einem sogenannten Matcher, der formuliert, welcher Testwert erwartet wird. Am besten lässt sich dies an einem Beispiel veranschaulichen (Listing 4): Bei der oben erwähnten Divisionsfunktion soll getestet werden, ob das Ergebnis der Division von 10 durch 2 auch 5 ist.

Der Testfall ist so formuliert, dass der Divisionsfunktion die beiden bekannten Werte übergeben werden. Die *expect()*-Funktion überprüft das Ergebnis dieser Division und stellt sicher, dass die Funktion *division()* als Quotient von 10 und 2 auch tatsächlich 5 zurückgibt. Der Part *.toEqual(expected)* ist dabei der Matcher, der das zu testende Ergebnis spezifiziert.

Grundsätzlich sollte man bei Assertions dem „Arrange-Act-Assert"-Pattern folgen, also der expliziten Unterteilung in Testvorbereitung, der eigentlichen Ausführung des zu testenden Codes und der Überprüfung der Ergebnisse. Mehr Details zum AAA-Pattern findet man unter ix.de/ix1902048.

Das Test-Framework enthält eine ganze Reihe verschiedenartiger Matchers. Aus Platzgründen geht dieser Artikel nur auf die wichtigsten ein. Darüber hinaus existieren zahlreiche weitere Matchers, darunter solche, die im Zusammenhang mit Mock-Implementierungen zum Einsatz kommen, Matchers für typsicheres Abprüfen von Werten, zur Zusammenarbeit mit Arrays und viele weitere. Die komplette Liste ist über ix.de/ix1902048 zu finden.

Wichtige Matchers

Der Matcher *.toEqual()* führt einen „Deep Equality"-Test zwischen den beiden Vergleichswerten durch. Geschachtelte Objekte werden dabei in ihrer kompletten Tiefe traversiert und verglichen. Damit der Gleichheitstest erfolgreich ist, müssen alle Keys und alle Values identisch sein.

.toBe() arbeitet ähnlich wie *.toEqual()*, überprüft allerdings bei Referenztypen wie Arrays lediglich die beiden übergebenen Referenzen auf Gleichheit: Sie müssen auf die gleiche Speicherstelle zeigen, sonst schlägt der Test fehl. Außerdem überprüft *.toBe()* primitive Typen wie Strings auf Identität.

.toBeGreaterThan() ist ein Vertreter einer Reihe von Matchers, die auf den Vergleich von Zahlen abzielen. Er überprüft, ob der zu testende Wert größer als ein anderer Zielwert ist. Entsprechend verhalten sich die Matchers *.toBeGreaterThanOrEqual()*, *.toBeLessThan()* und *toBeLessThanOrEqual()*.

Der spezielle Matcher *.toThrow()* gleicht keine Ergebnisse ab, sondern überprüft, ob die übergebene Funktion bei der Ausführung einen Fehler wirft. Listing 5 zeigt, wie man überprüft, ob bei der Divisionsfunktion beim Aufruf mit dem Divisor 0 ein Fehler auftritt.

.not schließlich ist selbst kein Matcher, sondern wird immer im Verbund mit einem anderen Matcher verwendet und kehrt dessen Bedeutung um:

```
expect(result).not.toEqual(not_expected);
```

Enzyme rendert React-Komponenten

Während Jest Testcode strukturiert und ausführt, ist die Bibliothek Enzyme verantwortlich für das Handling von React-Komponenten. Enzyme rendert React-Komponenten in einer Sandbox-Umgebung mit all ihrer zugehörigen Logik, zum Beispiel Hooks. Von einer mit Enzyme gerenderten Komponente kann man zum einen Werte und Zustände abfragen und zum anderen mit der Komponente interagieren. So lässt sich zum Beispiel ein Status-Update simulieren und anschließend sicherstellen, dass die erwarteten Änderungen tatsächlich eintreffen. Auch wenn dieser Artikel Jest und Enzyme im Verbund betrachtet, sind die beiden Bibliotheken unabhängig voneinander und können jede für sich eingesetzt werden.

Enzyme kann React-Komponenten auf drei Arten rendern: Shallow Rendering, Full DOM Rendering und Static Rendering.

Shallow Rendering rendert lediglich das Markup, das die *render()*-Funktion zurückgibt, stellt aber keine untergeordneten Komponenten wie im Thermometer-Beispiel die Kindkomponente Tempe-

```
 PASS  src\division.test.js
  division
    ✓ should divide correctly
    parameters
      ✓ should throw if divisor is zero
      ✓ should throw if parameters are not numbers

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.074s
Ran all test suites related to changed files.
```

Jest stellt die Ergebnisse im Testbericht entsprechend der Gliederung der Testfälle dar. Das ist besonders dann hilfreich, wenn ein Test einmal nicht erfolgreich durchläuft. Im Beispiel wurden die drei Testfälle erfolgreich absolviert (Abb. 1).

ratur dar. Das ist hilfreich, wenn es darum geht, eine Komponente isoliert zu betrachten und dabei sicherzustellen, dass ein Test keine untergeordnete Komponente indirekt beeinflusst. Beim Shallow Rendering bleibt der React-Kontext einer Komponente erhalten. So lassen sich zum Beispiel Property Updates senden. Referenzen auf Kindkomponenten bleiben erhalten, sodass sich diese in einem Test abfragen lassen.

Das Document Object Model (DOM) des Browsers ist eine Programmierschnittstelle, die den Inhalt eines Webseitendokuments repräsentiert und dessen Modifikation ermöglicht. Beim Full DOM Rendering rendert Enzyme die zu testende Komponente mit ihrem kompletten DOM-Baum. Das bedeutet, dass dabei auch referenzierte Kindkomponenten erzeugt werden. Darüber hinaus wird die Komponente in einer Umgebung initialisiert, die die native Browser-API implementiert. Dies ist insbesondere dann hilfreich, wenn es gilt, eine Komponente zu testen, die mit DOM-APIs interagiert. Auch beim Full DOM Rendering bleibt der React-Kontext erhalten.

Beim Static Rendering schließlich wird der React-Kontext über Bord geworfen. Das Resultat ist eine Datenstruktur, die das gerenderte Markup repräsentiert und die Abfragen sowie Querys auf den generierten DOM-Baum zulässt.

Listing 6 generiert die oben eingeführte Temperaturkomponente in allen drei Render-Modi. Die Rückgabewerte der verschiedenen Render-Methoden stellen jeweils einen Wrapper dar, der die Schnittstelle zu den Testkomponenten implementiert.

Zugriff auf genrenderte Komponenten

Hat Enzyme eine Komponente gerendert, ist im ersten Schritt zu definieren, welche Eigenschaften des Ergebnisses überprüft werden sollen. Auf die verschiedenen gerenderten DOM-Elemente (und auch React-Referenzen) greift man über sogenannte Selektoren zu. Diese Selektoren arbeiten genauso wie die native Selektions-API des Browsers. Um ein oder mehrere DOM-Elemente zu selektieren, muss also lediglich einen CSS-Selektor an die *.find()*-Methode des Render-Wrappers übergeben werden. Diese gibt wiederum einen Render-Wrapper zurück, der die Ergebnismenge repräsentiert und auf dem bei Bedarf erneute Abfragen ausgeführt werden können. Listing 7 zeigt einige Beispiele.

Listing 4: Assertions

```
it('should divide', () => {
  // arrange
  const divisor = 10;
  const dividend = 2;

  // act
  const result = division(divisor, dividend);

  // assert
  const expected = 5;
  expect(result).toEqual(expected);
});
```

Listing 5: *.toThrow()*

```
it('should throw if divisor is zero', () => {
  // arrange
  const dividend = 1;
  const divisor = 0;
  const action = () => division(dividend, divisor);

  // act & assert
  const expectedError = 'division by zero';
  expect(action).toThrowError(expectedError);
});
```

Im Unterschied zur nativen DOM-API des Browsers ist es mit Enzyme auch möglich, nach Referenzen auf Kindkomponenten zu selektieren. Im Kontext der eingangs definierten Thermometerkomponente kann man so zum Beispiel auf jedes der gerenderten Temperatur-Kinder zugreifen, um sicherzustellen, dass korrekte Parameter zugewiesen wurden.

Hat man ein DOM-Element oder eine Komponentenreferenz herausgesucht, werden im nächsten Schritt Eigenschaften des oder der Elemente abgefragt. Auch hier existiert eine große Zahl von Möglichkeiten; im Folgenden kommen lediglich die wichtigsten Methoden zu Sprache. Listing 8 zeigt einige Beispiele.

Die *.text()*-Abfrage gibt den Textinhalt eines DOM-Elements zurück. Wenn eine Kindkomponente selektiert wurde, lassen sich über *.props()* und *.prop(key)* React-spezifische Eigenschaften abfragen, hier etwa die übergebenen Properties.

Die Thermometerkomponente rendert drei Temperatur-Kinder mit der Minimal-, Maximal- und der aktuellen Temperatur. *.props()* liefert alle Properties ei-

Listing 6: Rendering mit Enzyme

```
import { shallow, mount, render } from 'enzyme';

const component = <Temperatur
  label='label'
  value={42} />;

// shallow rendering
const shallowWrapper = shallow(component);

// full DOM rendering
const fullDOMWrapper = mount(component);

// static rendering
const staticRendered = render(component);
```

Listing 7: Zugriff auf Selektoren

```
// Selektion via tagName
const resultByTag = shallowWrapper.find('div');

// Selektion via id
const resultById = shallowWrapper.find('#label');

// Selektion via Klassenname
const resultByClass = shallowWrapper.find('.temp');

// kombinierter Selektor
const resultByMisc = shallowWrapper.find('div.temp');
```

Listing 8: Abfrage von Eigenschaften

```
const temperaturWrapper = shallow(
  <Temperatur label="min" temp={42} />);
const tempDiv = temperaturWrapper.find('div');
const text = tempDiv.text();   // Ergebnis: "min: 42°C"
const id = tempDiv.prop('id'); // Ergebnis: "min"

const shallowWrapper = shallow(
  <Thermometer temp={20} />);
// at(1) bezieht sich auf das zweite Element
const current = wrapper.find(Temperatur).at(1);
const properties = current.props(); // Ergebnis:  {label: 'Current', value: 20}
```

Listing 9: Test Cases für die Temperaturkomponente

```
it('should render the correct id', () => {
  // arrange: die properties für Temperatur
  const label = "Temperatur";
  const temperature = 42.229;

  // act: die Temperatur wird gerendert
  const wrapper = shallow(<Temperatur
    label={label} value={temperature} />);

  // assert: über den Enzyme-Selektor 'div' wird
  // die id des Containers extrahiert und überprüft
  const id = wrapper.find('div').prop('id');
  const expected = 'temperatur';
  expect(id).toEqual(expected);
});

it('should render temp and label', () => {
  // arrange: die properties für Temperatur
  const label = "Temperatur";
  const temperature = 42.229;

  // act: die Temperatur wird gerendert
  const wrapper = shallow(<Temperatur
    label={label} value={temperature} />);

  // assert: über den Enzyme-Selektor 'div' wird der
  // Inhalt des Containers extrahiert und überprüft
  const text = wrapper.find('div').text();
  const expected = 'Temperatur: 42.23°C';
  expect(text).toEqual(expected);
});
```

Listing 10: Test Case 1 für die Temperaturkomponente

```
it('should render the min temp', () => {
  // arrange: die Temperatur
  const temperature = 20;

  // act: das Thermometer wird gerendert
  const wrapper = shallow(<Temperatur
    label={label} value={temperature} />);

  // assert: über den Enzyme-Selektor werden die erste
  // Referenz auf eine Temperatur (Minimum) und deren
  // properties ermittelt
  const min = wrapper.find(Temperatur).at(0);
  const expected = {label: 'Min', value: 20};
  expect(min.props()).toEqual(expected);
});
```

Listing 11: Test Case 2 für die Temperaturkomponente

```
describe('render thermo with updates (shallow)', () => {
  let wrapper = null;
  beforeAll(() => {
    // arrange + act: über den beforeAll hook wird zuerst
    // das Thermometer gerendert, anschließend werden
    // property updates ausgeführt
    const temp = 20;
    wrapper = shallow(<Thermometer temp={temp} />);
    wrapper.setProps({temp: 19.1211});
    wrapper.setProps({temp: 21.2221});
    wrapper.setProps({temp: 20.7123});
  });

  it('should render the min temperature', () => {
    const minTemp = wrapper.find(Temperatur).at(0);

    // assert: niedrigste Temperatur übergeben?
    const expected = {label: 'Min', value: 19.1211};
    expect(minTemp.props()).toEqual(expected);
  });

  it('should render the current temperature', () => {
    const currentTemp = wrapper.find(Temperatur).at(1);

    // assert: aktuelle Temperatur übergeben?
    const expected = {label: 'Current', value: 20.7123};
    expect(currentTemp.props()).toEqual(expected);
  });

  it('should render the max temperature', () => {
    const maxTemp = wrapper.find(Temperatur).at(2);

    // assert: höchste Temperatur übergeben?
    const expected = {label: 'Max', value: 21.2221};
    expect(maxTemp.props()).toEqual(expected);
  });
});
```

ner solchen Kindkomponente, im Beispiel wird die zweite mit *.at(1)* ausgewählt. *.prop(key)* arbeitet analog, liest allerdings nur den Wert aus, der dem übergebenen Key entspricht. Damit lassen sich wie in Listing 8 auch Attribute von DOM-Elementen auslesen.

Um das Verhalten einer React-Komponente zu testen, muss man verschiedene Ereignisse im Lifecycle einer Komponente ausführen. Einer der wichtigsten Hooks ist dabei die Reaktion auf aktualisierte Properties. Die Enyzme-Methode *setProps()* setzt an der gerenderten Komponente neue Properties. Das Testing-Tool registriert den Aufruf der Funktion und bewirkt, dass die Komponente automatisch neu gerendert wird. Ebenso werden assoziierte Hooks wie *getDerivedStateFromProps()* ausgeführt. Man kann bei einer gerenderten Komponente über *setState()* direkt den Status setzen oder über *simulate()* Events an DOM-Nodes auslösen.

Jest und Enzyme im Zusammenspiel

Aus der Kombination von Jest und Enzym ergibt sich ein leistungsfähiges Werkzeug für das Frontend-Unit-Testing. Der Temperaturkomponente des Beispiels etwa werden zwei Parameter übergeben, das Label und der aktuelle Temperaturwert. Das Label dient in Kleinbuchstaben als ID für ein zu renderendes *div*, das das Label und den gerundeten Wert inklusive der Einheit enthält.

Daraus ergeben sich zwei Testfälle: Erhält das *div* das richtige Label und werden Label und gerundete Temperatur im *div* korrekt ausgegeben? Listing 9 zeigt, wie Jest und Enzyme dabei zusammenspielen.

Die Thermometerkomponente erhält lediglich einen Temperaturwert. Wird dieser Wert aktualisiert, dann aktualisiert die Komponente den Minimal- und den Maximalwert.

Der erste Test Case in Listing 10 überprüft, ob das Thermometer den gesetzten Temperaturwert an die Temperaturkomponenten weitergibt. Das vorliegende Testbeispiel überprüft nur die Minimaltemperatur, die Tests der weiteren Temperaturwerte erfolgen analog; sie werden hier aus Platzgründen ausgelassen. In einer realen Implementierung würde man natürlich jede Temperaturreferenz überprüfen.

Das Festhalten von Minimal- und Maximaltemperatur ist das zweite wichtige Feature der Thermometerkomponente. Wie oben beschrieben setzt der Testcode auf der gerenderten Komponente wieder-

holt verschiedene Temperaturwerte (siehe Listing 11). Aufgabe des Thermometers ist es, die Extremtemperaturen festzuhalten und die ermittelten Werte an die Kindkomponenten zu übergeben.

Diesen Test hätte man auch über Full DOM Rendering (mount) durchführen können. Damit wäre zum Beispiel eine konkrete Abfrage des gerenderten Gesamtergebnisses möglich. Aus Platzgründen zeigt der Artikel diese Variante nicht, sie ist aber im Beispielcode im Repository implementiert (siehe ix.de/ix1902048).

Snapshots zeigen Veränderungen

Jest bringt ein weiteres praktisches Hilfsmittel mit: die sogenannten Snapshot-Tests. Ein einfacher Schreibbefehl weist Jest an, das Abbild eines übergebenen Objektes anzufertigen. Dieses erzeugte Abbild stellt dabei eine textuelle Repräsentation des übergebenen Objektes dar. Jest speichert dieses Textabbild in einem speziellen Format in einem separaten Verzeichnis. Nachfolgende Testläufe ziehen diese abgespeicherte Textrepräsentation heran und vergleichen sie mit der Darstellung des aktuellen Stands. Da es sich hierbei um reinen Text handelt, ist Jest in der Lage, einen eindeutigen Abgleich zwischen den (möglicherweise) unterschiedlichen Ständen herzustellen und Unterschiede hervorzuheben.

Besonders hilfreich sind Snapshot-Tests, wenn es darum geht, den gerenderten Output einer Komponente zu überprüfen. Würde man bei einer Komponente jeden generierten DOM-Knoten von Hand abgleichen, wäre eine unübersichtliche und fehleranfällige Menge an Testfällen das Resultat. Der Snapshot-Test führt diesen Test auf einem höheren Abstraktionslevel durch: Das Framework visualisiert damit Unterschiede so, dass der Entwickler schnell entscheiden kann, ob eine Abweichung tatsächlich auf einen Fehler zurückzuführen ist oder auf eine gewollte Codeänderung.

Schlägt ein Test fehl, zeigt Jest den fehlerhaften Test an und hebt die Unterschiede zwischen dem Snapshot und dem aktuellen Stand des Prüfobjektes hervor. Auf dieser Grundlage kann der Entwickler entscheiden, ob diese Abweichung gewünscht ist oder ein Fehler vorliegt.

Abbildung 2 zeigt, wie Jest eine Abweichung darstellt, die sich in der Thermometerkomponente ergeben hat: Zum einen hat sich die ID des *div*-Knotens geändert (hier scheint ein Tippfehler passiert zu sein), zum anderen hat die Komponente statt einer Überschrift erster Ordnung eine Überschrift zweiter Ordnung gerendert – wahrscheinlich eine gewollte Änderung. In jedem Fall erhält der Entwickler Feedback zu den Veränderungen und kann entsprechende Maßnahmen ergreifen.

Snapshot vs. aktuelles Objekt: Ein „unified diff", der Unterschiede zwischen Textdateien menschen- und maschinenlesbar auflistet, macht deutlich, inwiefern sich der gespeicherte und der aktuelle Stand unterscheiden (Abb. 2).

```
Received value does not match stored snapshot 1

- Snapshot
+ Received

@@ -1,11 +1,11 @@
 <div
-  id="thermometer"
+  id="thermomete"
 >
-  <h1>
+  <h2>
     Thermometer
-  </h1>
+  </h2>
   <Temperatur
     label="Minimum"
     value ={42}
   />
   <Temperatur
```

Listing 12: Snapshot-Test

```
it('should match the snapshot', () => {
  const temperature = 42;
  const component = <Thermometer temp={temperature} />;
  const rendered = shallow(component);
   expect(rendered).toMatchSnapshot();
});
```

Grundsätzlich sollten Snapshot-Artefakte in die Quellcodeverwaltung der Wahl eingecheckt werden, damit sich Snapshot-Tests automatisiert in einer CI/CD-Umgebung durchführen lassen. Allerdings sollten Snapshot-Abbilder nicht automatisiert erstellt und überschrieben werden, da die explizite Interaktion mit einem Entwickler erwünscht ist: Nur so kann man Fehler eindeutig kontrollieren.

Natürlich haben Snapshot-Tests auch ihre Grenzen. Sie können keinesfalls das detaillierte Testen und Prüfen von Logik und Verhalten ersetzen. Sie stellen jedoch ein effizientes Werkzeug dar, um einen groben Überblick über auftretende Fehler zu erhalten.

Um einen Snapshot-Test durchzuführen, muss man lediglich in gewohnter Weise die Funktion *expect()* verwenden, kombiniert mit dem Matcher *toMatchSnapshot()* (siehe Listing 12). Bei jedem Testlauf wird Jest nun an einem definierten Ort im Dateisystem nachsehen, ob für den Test Case „should match a snapshot" bereits ein Snapshot vorliegt. Ist dies der Fall, vergleicht man die serialisierte Form der gerenderten Komponente *component*. Existiert kein Snapshot, wird dieser erzeugt und gespeichert.

Fazit

Mit den Tools Jest und Enzyme lassen sich umfassende Testsuiten mit wenig Aufwand erstellen. Da bei der Testausführung keine native Browserumgebung zum Einsatz kommt, sind Tests deutlich effizienter und stabiler. Die Simulation hat aber auch Nachteile: Gegenüber einem realen Browser kann es in der simulierten Umgebung Abweichungen geben. Beispielsweise ist die Simulation von DOM-Events in Enzyme bisher nur relativ rudimentär angelegt; es existieren jedoch Pläne, dies in Zukunft auszubauen. Abhängig vom Anwendungsfall kann es also angebracht sein, die Unit-Tests durch ein dediziertes End-To-End-Testing-Framework zu ergänzen. (odi@ix.de)

Manuel Ernst

ist als Senior Software Engineer bei Method Park tätig. Er berät und unterstützt seine Kunden aus der Medizinbranche bei der Entwicklung von mobilen und Web-Apps.

Alle Links: ix.de/ix1902048