12 Flux-Architektur am Beispiel von Redux

12.1 Einführung

Sobald deine Anwendung größer wird und du vielleicht auch mit mehreren Entwicklern daran arbeitest, kann eine einheitliche Architektur der Übersichtlichkeit und Wartbarkeit dienen. Flux[1] ist im React-Bereich der De-facto-Standard für eine solche Architektur und ist zusammen mit React bei Facebook entstanden.

Redux

Redux[2] wiederum ist eine Implementierung dieser Flux-Architekturidee. Also: Flux ist die Architekturidee, Redux ist eine Implementierung dieser Idee.

Redux hebt sich für uns durch seine hohe Verbreitung, seine Einfachheit und seine fortschrittlichen Konzepte von anderen Flux-Implementierungen ab. Redux wird ebenso wie der React Router von der React-Community entwickelt. Dan Abramov, der Hauptentwickler von Redux, legte Anfang 2015 den Grundstein für die Entwicklung mit seinem Blogpost[3], in dem er erklärt, wofür und wie man Flux nutzen sollte.

1. *https://facebook.github.io/flux/docs/overview.html*
2. *http://redux.js.org/*
3. *https://medium.com/@dan_abramov/the-case-for-flux-379b7d1982c6*

Redux im Überblick

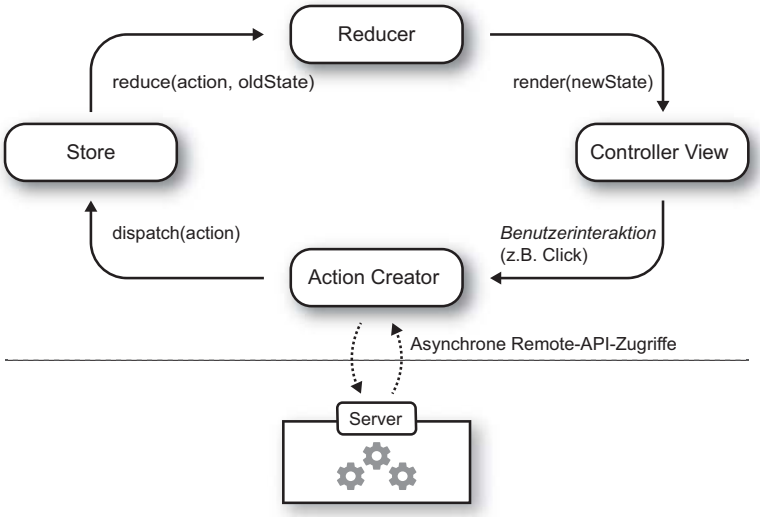


Abb. 12–1
Flux-Architektur von Redux umgesetzt

In Abbildung 12–1 siehst du eine Übersicht über die Flux-Architektur, wie sie in Redux umgesetzt ist.

Controller-Views

Unsere Vote-Anwendung haben wir bereits so aufgeteilt, dass die Funktionalität und der Zustand der Anwendung auf nur wenige Komponenten verteilt sind. Diese Komponenten haben wir Controller genannt.

In Redux verlieren auch diese Komponenten, hier *Controller-Views* genannt, die Verantwortung für den Zustand der Anwendung und für die Funktionalität, die durch Benutzerinteraktionen ausgelöst wird.

Action-Creators

Stattdessen wandert die Funktionalität in *Action-Creators*. Diese Funktionen werden nach einer Benutzerinteraktion aufgerufen und erzeugen eine fachliche *Action*. Diese Action enthält Informationen darüber, wie sich der Zustand der Anwendung ändern soll (z.B. welche Umfrage um wie viele Stimmen erhöht werden soll). Ein Action-Creator ist auch die einzige Stelle, in der asynchrone Aktivitäten (z.B. Serverzugriffe) ausgeführt werden. Das Erzeugen der Action wird dann so lange zurückgehalten, bis die asynchrone Aufgabe erledigt wurde und zum Beispiel die Antwort vom Server eingetroffen ist.

Store

Der Zustand der Anwendung wiederum wird zentral über einen einzigen *Store* gemanagt, der die über Action-Creators erzeugten Actions entgegennimmt und an sogenannte Reducer verteilt (dispatched).

Reducer

Ein *Reducer* ist eine Funktion, die für die (fachliche) Verarbeitung einer Action zuständig ist. Eine Anwendung besteht typischerweise aus mehreren Reducers, die nach Fachlichkeit aufgeteilt sind. Sobald im Store eine Action eintrifft, wird diese samt des bisherigen Zustands an alle Reducer weitergeleitet. Daraus produzieren die Reducer einen neuen Zustand, den sie an den Store zurückgeben. Mit diesem neuen Zustand werden dann die React-Komponenten neu gerendert.

Datenfluss in eine Richtung

Der Daten- und Kontrollfluss in einer reduxbasierten Anwendung fließt also immer in eine Richtung. Direkte Interaktion zwischen einzelnen Komponenten durch Event-Listener wie in vielen anderen UI-Architekturen gibt es in Redux (und auch in Flux) nicht.

Flux bei Facebook

Flux ist bei Facebook als eine Antwort auf Schwierigkeiten bei der Entwicklung ihrer Hauptanwendung entstanden. Aufgrund vieler Abhängigkeiten zwischen unterschiedlichen Teilen der Anwendung war der Fluss der Daten durch die Anwendung nicht mehr durchschaubar. Gewisse Fehlerklassen konnten immer wieder beobachtet und nicht dauerhaft behoben werden. Flux ist eine Architekturidee, die eine sehr klare Trennung der an der Anwendung beteiligten Bestandteile und einen sehr klaren Fluss von Daten und Kontrolle in eine Richtung fordert. Eine solche Architektur verspricht eine hohe Verständlichkeit der Anwendung und gibt einen klaren Leitfaden für die Entwicklung.

Flux ist kein Framework und keine Bibliothek, sondern nur ein Konzept, eine Idee, ein Muster. Facebook selbst liefert für Flux nur etwas Utility-Code (https://facebook.github.io/flux/docs/flux-utils.html) aus.

12.2 Hands-on: Eine Redux-Anwendung

Wie schon im vorherigen Kapitel entwickeln wir in diesem Hands-on-Teil wieder eine sehr einfache Anwendung. Dabei verwenden wir nur Redux selber, aber weder den React Router noch Universal Rendering. Dadurch bekommst du einen einfachen Einstieg, selbst wenn du dich nur für Flux oder Redux interessierst.

In den folgenden Abschnitt zeigen wir dir weiterführende Redux-Konzepte (z.B. den Umgang mit dem React Router) dann anhand unserer Vote-Beispielanwendung.

Ausgangssituation: Die HelloMessage-Anwendung

Die Grundlage für unser Redux-Beispiel ist die HelloMessage-Anwendung aus dem vorherigen Abschnitt 11.1 (ohne Universal Rendering). Dieses Beispiel ist allerdings um einen »Reset«-Button erweitert, der den Inhalt des Grußeingabefelds löscht. Die Verarbeitung der entsprechenden Events erfolgt in den beiden Methoden updateGreeting bzw. resetGreeting, so dass die Komponente zunächst wie folgt aussieht:

Listing 12–1
HelloMessage.js mit Reset-Button

```
import React from 'react';

export default class HelloMessage extends React.Component {
  constructor(props) {
    super(props);
    this.state = {greeting: this.props.greeting};
  }

  resetGreeting() {
    this.setState({greeting: ''});
  }

  updateGreeting(greeting) {
    this.setState({
      greeting
    });
  }

  render() {
    const { greeting } = this.state;
    return <div>
      <input
        onChange={event=> this.updateGreeting(event.target.value)}
        value={greeting}
      />
      <p>Hello, {greeting}</p>
      <button
        onClick={() => this.resetGreeting() }>
        Clear
      </button>
    </div>;
  }
}
```

Der Code für die Anzeige im Browser ist wieder trivial:

Listing 12–2
main.js

```
import React from 'react';
import ReactDOM from 'react-dom';

import HelloMessage from './HelloMessage';

const greeting = 'World';
const mountNode = document.getElementById('mount');
ReactDOM.render(<HelloMessage greeting={greeting}/>, mountNode);
```

Abb. 12–2
Der Ausgangszustand der Beispielanwendung

Du findest diesen Ausgangszustand unseres Beispiels im Repository in dem Verzeichnis `steps/12_redux/hellomessage_redux/00-no-redux`. Von dort kannst du es mit `npm start` ausführen, die Anwendung läuft dann unter *http://localhost:8080*. Auf dieser Grundlage werden wir in den folgenden Schritten die Anwendung auf Redux umstellen.

Schritt 1: Actions und Action-Creators

In dem Beispiel oben haben wir die beiden Methoden `updateGreeting` und `resetGreeting`, um den Zustand der Anwendung zu setzen, der ebenfalls in der `HelloWorld`-Komponente gehalten wird. Wie wir später noch genauer sehen werden, ist eine Idee von Redux, den Zustand der Anwendung nicht in einzelnen Komponenten, sondern zentral an einer Stelle zu halten. Dasselbe gilt auch für Aktionen, die sich auf diesen Zustand auswirken und ihn verändern.

Wir ziehen daher zunächst die `updateGreeting`- und `resetGreeting`-Methoden aus der React-Komponente heraus und verschieben sie in ein eigenes Modul in der neuen Datei `actions.js`:

Listing 12–3
Das Modul actions.js

```
export const UPDATE_GREETING = 'UPDATE_GREETING'; // A
export const RESET_GREETING = 'RESET_GREETING';

export function updateGreeting(greeting) { // B
    return { // C
        type: UPDATE_GREETING, // D
        greeting // E
    };
}

export function resetGreeting() { // F
  return {
    type: RESET_GREETING
  };
}
```

Eine Action in Redux ist ein Objekt, das zumindest aus einem Property mit dem Namen `type` besteht. Dieses Property legt den Typ der Aktion fest. In Zeile *A* definieren wir einen solchen neuen Action-Typ als einfache String-Konstante, die wir in Zeile *D* als `type` für die Action verwenden.

Action-Objekte

Neben dem `type` kann das Action-Objekt noch weitere Informationen enthalten, die für die fachliche Verarbeitung der Aktion notwendig sind. Für das Aktualisieren des Grußes wird der neue Gruß benötigt und als `greeting`-Property im Action-Objekt abgelegt (Zeile *E*). Beim Zurücksetzen des Grußes hingegen werden keine weiteren Informationen benötigt, so dass diese Action nur aus dem `type`-Property besteht.

Action-Creator

Funktionen, die ein Action-Objekt erzeugen, werden in Redux Action-Creator genannt. In unserem Fall sind das die beiden Funktionen `updateGreeting` (Zeile *B*) und `resetGreeting` (Zeile *F*). Anders als die ursprünglichen Methoden, die in der `HelloMessage`-Komponente definiert waren, verändern die Action-Creator jedoch nicht den Zustand der Anwendung. Sie erzeugen stattdessen nur eine Action, die andere Teile der Anwendung noch ausführen müssen. Dies geschieht mit Reducers, die wir im folgenden Schritt ansehen.

Schritt 2: Logik in Reducers implementieren

Zustand verändern

Reducer sind dazu da, anhand von den eben gesehenen Actions einen Zustand in einen neuen Zustand zu überführen. In unserem Beispiel wäre der alte Zustand der bisherige Inhalt des Eingabefelds. Der neue Zustand ist dann der künftige Inhalt des Felds, also ergänzt um das gerade vom Benutzer eingegebene Zeichen.

Reducer

Reducer sind seiteneffektfreie Funktionen: Sie erhalten zwei Eingabeparameter (den derzeitigen Anwendungszustand sowie die Action, die gerade ausgelöst wurde) und erzeugen daraus eine Ausgabe (den neuen Anwendungszustand).

Obwohl sie den Teil der Anwendungslogik beschreiben, der den Zustand verändert, sind sie daher dennoch sehr einfach zu verstehen und zu testen. Die beiden von uns benötigten Reducer, die unsere beiden Actions bearbeiten, implementieren wir in der neuen Datei *store.js*:

Listing 12–4
Reducer in der Datei store.js

```
import {UPDATE_GREETING, RESET_GREETING} from './actions'; // A

function greetingReducer(state = 'World', action) { // B
  switch (action.type) { // C
    case UPDATE_GREETING:
      return action.greeting; // D
    case  RESET_GREETING:
      return ''; // E
    default:
      return state; // F
  }
}
```

Der Reducer wird als einfache Funktion in Zeile *B* definiert und bekommt den bisherigen Zustand (`state`) sowie eine Action (`action`)

als Parameter übergeben. Daraus erzeugt er einen neuen Zustand und gibt diesen zurück.

Action-Verarbeitung

Dazu ermitteln wir zunächst in Zeile C den übergebenen Action-Typ anhand des `type`-Propertys und führen davon abhängig die konkrete fachliche Logik aus. Im Falle der `UPDATE_GREETING`-Action verwenden wir dazu die in dem Action-Objekt enthaltenen Informationen (`greeting`-Property), um den Zustand auf den neuen Gruß zu setzen. Die `RESET_GREETING`-Bearbeitung ist noch einfacher: Hier setzen wir den Zustand einfach auf einen Leerstring (Zeile *E*).

Mehrere Reducer

In einer realistischen Anwendung gäbe es wahrscheinlich mehr als zwei Actions und auch mehrere Reducer. Da sich nicht jeder Reducer für jede Action interessiert, gibt ein Reducer für Actions, an denen er nicht interessiert ist, einfach den alten Zustand zurück (Zeile *F*).

Initialer Zustand

Der Zustand der Anwendung wird ausschließlich von Redux verwaltet. Du brauchst den Zustand also nicht in deinen Reducers abzuspeichern.

ES6: Default-Parameter werden im Anhang im ES6-Kapitel erklärt.

Für den Fall, dass (beim Starten der Anwendung) noch gar kein Zustand vorliegt, können wir einfach einen Default-Zustand setzen. Dafür verwenden wir einen ES6-Default-Parameter und setzen den initialen Zustand auf »World« in Zeile *B*. Bei der allerersten Ausführung des Reducers wird `state` also mit dem String »World« vorbelegt und wir brauchen bei der Verarbeitung der Action keine weitere Sonderbehandlung durchzuführen für den Fall, dass es noch keinen vorherigen Zustand gibt.

Im nächsten Schritt sehen wir uns an, wie Redux den Zustand der Anwendung in einem Store verwaltet.

Schritt 3: Store und State

Unsere Anwendung reagiert jetzt auf Benutzerinteraktionen, indem sie mit einem Action-Creator eine Action erzeugt, die anzeigt, was gerade passiert ist. Mit einem Reducer wird dann abhängig von der Action aus dem alten Zustand ein neuer Zustand erzeugt.

Store

Dieser Zustand wird bei Redux zentral in einem einzigen Store gehalten. Das gilt auch, wenn es mehrere Reducer geben sollte. Dabei ist der Store strukturiert und für jeden Bereich darin ist genau ein Reducer zuständig.

Aus welchen Teilzuständen der Store besteht und welcher Reducer dafür jeweils zuständig ist, definierst du zunächst mit der Redux-Funktion `combineReducers`. Danach erzeugst du mit `createStore` den zentralen Store. Das sehen wir uns gleich genauer an.

Redux installieren

Da wir an dieser Stelle zum ersten Mal Code von Redux verwenden, müssen wir Redux zunächst installieren. Bitte installiere jetzt das `redux-npm`-Paket in der Version 3 in dem Beispielprojekt:

```
npm install redux@3 --save
```

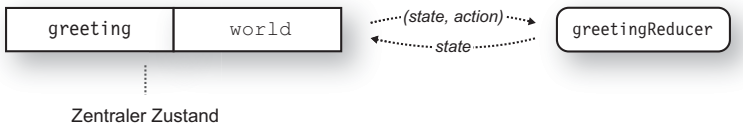
Danach kannst du das im vorherigen Schritt angelegte Modul *store.js* mit dem folgenden Code erweitern, um den Store mit der gewünschten Konfiguration zu erzeugen:

Listing 12–5
Die erweiterte Datei store.js

```
import { createStore, combineReducers } from 'redux';

function greetingReducer(state = 'World', action) { . . . }

const store = createStore(combineReducers({ // A
    greeting: greetingReducer // B
}));

export default store;
```

In Zeile *A* wird mit der Funktion `createStore` der zentrale Store erzeugt. Dieser erwartet als Parameter eine Reducer-Funktion. In unserem Beispiel hätte es ausgereicht, an dieser Stelle die `greetingReducer`-Funktion zu übergeben. Der Store hätte dann ausschließlich aus dem Zustand bestanden, den die Funktion zurückliefert (also dem eingegebenen Gruß).

Abb. 12–3
Zentraler Zustand in der HelloMessage-Anwendung



Die combineReducers-Funktion

In einer größeren Anwendung werden zur Strukturierung sehr schnell mehrere Reducer verwendet. Die createStore-Methode erwartet aber genau eine Reducer-Funktion. Daher können mehrere Reducer mit der Funktion combineReducers zu einem Reducer zusammengefasst werden. Die combineReducers–Funktion erwartet ein Objekt, das den Namen eines Teilzustands (Schlüssel) auf die entsprechende Reducer-Funktion abbildet (Wert).

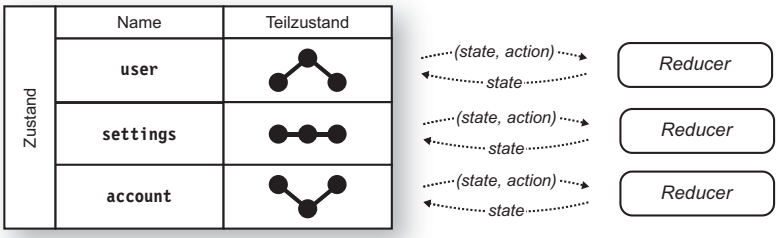


Abb. 12–4
(Fiktiver) Zentraler Zustand mit mehreren Teilzuständen (user, settings, account) und ihren entsprechenden Reducers

In unserem Beispiel (Zeile *B*) gibt es einen Teilzustand mit dem Namen `greeting`, der von unserem `greetingReducer` verwaltet wird.

Das bedeutet, dass der `greetingReducer` von Redux mit dem unter `greeting` abgelegten Teilzustand aufgerufen wird, wenn eine Action zu verarbeiten ist. Das Ergebnis des Reducers wird dann auch wieder unter diesem Teilzustand abgelegt.

Schritt 4: Die einzelnen Teile verbinden

Wir haben gesehen, dass Komponenten in Redux Aufgaben abgeben. Der Zustand wandert aus den Komponenten heraus in einen einzigen, zentralen Store. Die Logik wird in Action-Creators und Reducers ausgelagert.

Wir müssen jetzt dafür sorgen, dass unsere `HelloMessage`-Komponente Zugriff auf diese Dinge bekommt, damit sie einerseits den Gruß aus dem Zustand lesen und anderseits nach einer Benutzerinteraktion eine entsprechende Aktion auslösen kann.

Schritt 4a: Zugriff auf den Zustand

Zunächst wollen wir uns ansehen, wie die `HelloMessage`-Komponente Zugriff auf den benötigten Zustand bekommt.

Zustand über Properties

Der Zustand wird über Properties aus dem zentralen Zustand in die Komponenten hineingereicht. Oftmals arbeitet eine Komponente aber nicht mit dem gesamten Zustand, sondern nur auf einem der Teilzustände. Welcher Teilzustand für eine Komponente relevant ist, kannst du über eine Mapping-Funktion festlegen. Dieser Funktion wird der komplette Zustand übergeben. Sie liefert dann ein Objekt zurück, in dem angegeben wird, welcher Teil des Zustands unter welchem Property-Namen der Komponente zur Verfügung gestellt werden soll.

Funktion zum Mappen von Zustand auf Properties

Wir nennen die Mapping-Funktion für unsere `HelloMessage`-Komponente `mapStateToProps` und legen diese neben der bestehenden Komponentenklasse auch in der Datei `HelloMessage.js` ab:

Listing 12–6
Mapping des Zustands auf Properties (HelloMessage.js)

```
function mapStateToProps(state) { // A
  return {
    greeting: state.greeting // B
  };
}
```

Der Mapping-Funktion wird von Redux der komplette Zustand über den Parameter `state` übergeben (Zeile *A*).

In der Funktion beschreiben wir dann in Zeile *B*, dass der Teilzustand `greeting` aus dem kompletten Zustand als Property in die `HelloMessage`-Komponente hereingereicht werden soll. Der Name des Propertys soll ebenfalls `greeting` lauten, so dass der entsprechende Zustand in der Komponente über `this.props.greeting` zur Verfügung steht.

Schritt 4b: Action-Creators einbinden

Wir haben jetzt definiert, welchen Teilzustand wir in unserer Komponente benötigen und unter welchem Property dieser zur Verfügung stehen soll.

Actions statt UI-Logik

Aber auch die UI-Logik verschwindet aus der Komponente selbst. Stattdessen soll die Komponente nur noch Actions auslösen. Dazu stehen die Action-Creator zur Verfügung. Auf diese braucht unsere Komponente ebenfalls Zugriff.

Zur Erinnerung hier noch einmal die oben extrahierte `updateGreeting`-Funktion, die in `actions.js` definiert ist:

```
export function updateGreeting(greeting) {
    return {
        type: UPDATE_GREETING,
        greeting
    };
}
. . .
```

Verteilen von Actions: die dispatch-Methode

Das Ergebnis dieses Action-Creators ist eine Action, die wir durch Redux an alle Reducer verteilen lassen. Die Verteilung einer Action funktioniert über die `dispatch`[4]-Methode aus dem Redux-Store. Sie erwartet als Parameter eine Action. Die Action wird dann allen Reducers nacheinander mit dem jeweiligen alten Teilzustand zur Verarbeitung angeboten. Die zurückgegebenen Teilzustände bilden danach den neuen Gesamtzustand der Anwendung.

4. *http://redux.js.org/docs/api/Store.html#dispatch*

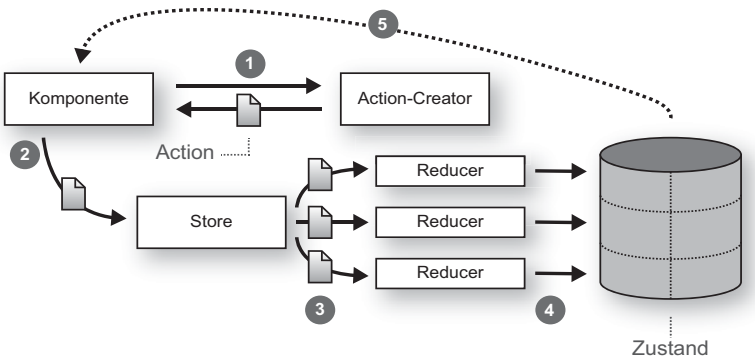


Abb. 12–5
Dispatchen einer Action

Eine Komponente erzeugt mit einem Action-Creator eine Action (1) und übergibt diese zur Verteilung an den zentralen Store (2). Der Store gibt die Action an alle bekannten Reducer weiter (3). Die Reducer verarbeiten die Action und geben jeweils einen neuen Teilzustand als Ergebnis der Verarbeitung zurück. Die Ergebnisse aller Reducer werden im neuen Gesamtzustand abgelegt (4). Der neue Gesamtzustand wird der Komponente über Properties (5) übergeben. Die Komponente rendert sich mit den aktualisierten Properties.

Das Erzeugen einer Action und die Übergabe an den Dispatcher könnten demnach wie folgt aussehen:

```
import store from './store'; // A
import { updateGreeting } from './actions'; // B

const action = updateGreeting('Hiho'); // C
store.dispatch(action); // D
```

In Zeile *A* wird der Store importiert. Er enthält die Konfiguration aller Reducer. In Zeile *B* wird der Action-Creator `updateGreeting` importiert, der in Zeile *C* dann aufgerufen wird. Die dabei erzeugte Action wird in Zeile *D* schließlich über die erwähnte `dispatch`-Methode verteilt.

Das Verteilen ist notwendig, weil Actions für sich noch nichts bewirken. Sie müssen erst durch die `dispatch`-Methode geleitet werden, damit sie an die Reducer weitergeleitet werden können und letztlich eine Zustandsänderung bewirken. In Redux ist dies die einzige Möglichkeit, den Zustand zu ändern.

Action-Creators an die Komponente binden

Den oben gezeigten Code könnten wir so auch in unserer Komponente verwenden, allerdings ist er auf Dauer recht sperrig. Daher ist es üblich, Action-Creators an die Properties der Komponente zu binden, ähnlich, wie wir das im vorherigen Schritt auch beim Zustand gesehen haben. Die *gebundenen* Action-Creators kümmern sich dann auch gleich um das Verteilen der erzeugten Action, so dass in der Komponente kein expliziter `dispatch`-Aufruf erforderlich ist.

Dazu legen wir ebenfalls in der `HelloMessage.js`-Datei eine weitere Funktion mit dem Namen `mapDispatchToProps` an:

Listing 12–7 *Die Funktion mapDispatchToProps (HelloMessage.js)*

```
import { bindActionCreators } from 'redux';
import * as Actions from './actions';

function mapDispatchToProps(dispatch) { // A
  return bindActionCreators(Actions, dispatch); // B
}
```

Der Funktion wird in Zeile *A* von Redux die `dispatch`-Funktion übergeben. In Zeile *B* benutzen wir die Redux-Funktion `bindActionCreators`[5], welche Action-Creators mit der `dispatch`-Funktion aus dem Redux-Store verknüpft. Dazu werden der Funktion die Action-Creators übergeben (in unserem Fall `Actions`, die wir aus dem `actions`-Modul importiert haben). Außerdem wird der Funktion die `dispatch`-Funktion übergeben, mit deren Hilfe Actions, die über einen Action-Creator erzeugt wurden, verteilt werden sollen.

Schritt 5: Integration mit React

Wir haben jetzt Zustand und Logik aus unserer Komponente in Store bzw. Action-Creators und Reducers verschoben und beschrieben, welchen Teilzustand und welche Action-Creators wir in unserer Komponente benötigen. Jetzt fehlt uns nur noch, wie wir die `HelloMessage`-Komponente mit Redux verknüpfen, so dass Redux der Komponente die geforderten Properties auch zur Verfügung stellen kann.

react-redux

Redux ist in zwei Bibliotheken aufgeteilt: eine allgemeine (*redux*), die wir in den vorherigen Schritten schon verwendet haben. Diese ist von React unabhängig. Darüber hinaus gibt es noch eine zweite Bibliothek (*react-redux*), die die speziellen Bindungen an React enthält.

Installieren von react-redux.

Bevor wir fortfahren, musst du nun also auch *react-redux* installieren. Wir benutzen für unsere Anwendung die Version 4:

```
npm install react-redux@4 --save
```

Die Bibliothek *react-redux* stellt die Funktion `connect`[6] zur Verfügung, mit der eine React-Komponente mit Redux gekoppelt werden kann. Dabei werden die oben entwickelten Funktionen `mapStateToProps` und `mapDispatchToProps` verwendet, um Redux mitzuteilen, welche Properties unsere Komponente benötigt.

Die `HelloMessage`-Komponente sieht nun wie folgt aus. Den entsprechenden `connect`-Aufruf findest du am Ende der Datei `HelloMessage.js`:

5. *http://redux.js.org/docs/api/bindActionCreators.html*
6. *http://redux.js.org/docs/basics/UsageWithReact.html*

Listing 12–8
HelloMessage mit Redux verbunden

```
import React from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';

import * as Actions from './actions';

class HelloMessage extends React.Component {
  render() {
    const {
      greeting, updateGreeting, resetGreeting
    } = this.props;  // A

    return <div>
      <input
        onChange={event=> updateGreeting(event.target.value)} // B
        value={greeting} />
      <p>Hello, {greeting}</p>
      <button
        onClick={() => resetGreeting()}> // C
        Clear
      </button>
    </div>;
  }
}

function mapStateToProps(state) { // D
  return {
    greeting: state.greeting
  };
}

function mapDispatchToProps(dispatch) { // E
  return bindActionCreators(Actions, dispatch);
}

export default connect // E
  (mapStateToProps, mapDispatchToProps)(HelloMessage);
```

Innerhalb unserer Komponente können wir über Properties auf die gebundenen Action-Creators und den Zustand zugreifen (Zeile *A*). Die Action-Creators lassen sich dann wie »gewöhnliche« Funktionen ausführen, die dabei erzeugten Actions werden automatisch verteilt (Zeile *B* und *C*).

In Zeile *D* und *E* siehst du zur Erinnerung die beiden Funktionen, die den Gesamtzustand der Anwendung (`mapStateToProps`) sowie die gebundenen Action-Creators (`mapDispatchToProps`) auf Properties für die Komponente mappen.

Die connect-Funktion

In Zeile *E* verbinden wir schließlich die Komponente mit Redux: Technisch gesehen erzeugt dort die Funktion `connect` aus der ursprünglichen Komponente `HelloMessage` eine neue React-Komponente. Diese neue Komponente ummantelt `HelloMessage`, um ihr weitere Fähigkei-

ten zu verleihen: in unserem Fall die Fähigkeit, Action-Creators und State an Properties zu binden und die Komponenten bei Änderung des Zustands neu zu rendern.

Higher-Order-Komponenten

Eine solche Komponente nennt man Higher-Order-Komponente, also eine Komponente höherer Ordnung. Diese Higher-Order-Komponente exportieren wir anstatt der ursprünglichen Komponente. Für den Rest der Anwendung, die Verwender der Komponente, ergibt sich daraus keinerlei Änderung.

Schritt 6: Der Redux Provider

Die Higher-Order-Komponente, die wir mit `connect` erzeugt haben, braucht den zentralen Store, um diese Ummantelung korrekt auszuführen. Der Store wird für die `dispatch`-Funktion benötigt und außerdem für das Mapping des Zustands auf die Properties der Komponente.

Um an den Store zu gelangen, verwendet die `connect`-Methode den React Context (den wir auch schon in Kapitel 10 gesehen haben).

Das Ablegen des Stores im React Context übernimmt die Redux-Komponente `Provider`, die wir als Top-Level-Komponente für unsere Anwendung einbinden – ebenfalls ein ähnliches Verfahren wie beim React Router.

Unsere vollständige `main.js`-Datei sieht mit dem Provider nun wie folgt aus:

Listing 12–9
Die fertige main.js-Datei mit dem Redux Provider

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';

import HelloMessage from './HelloMessage';

const mountNode = document.getElementById('mount');
ReactDOM.render(
  <Provider store={store}> // A
    <HelloMessage /> // B
  </Provider>,
  mountNode
);
```

Den in Schritt 3 erzeugten Store geben wir in Zeile *A* als Property in den Provider, der diesen im React Context hält und seinen Kindern zur Verfügung stellt. Unsere Anwendung besteht nur aus einer einzigen Kind-Komponente (Zeile *B*), unsere `HelloMessage`-Komponente. Dabei handelt es sich übrigens nicht um die ursprüngliche `HelloMessage`-Komponente, sondern um die Higher-Order-Komponente, die durch connect erzeugt und dann exportiert wurde. Diese Komponente holt sich den

store aus dem React Context und nutzt ihn für den Zustand und die Action-Creators.

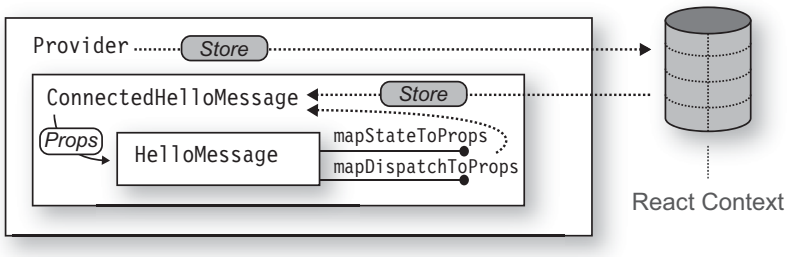


Abb. 12–6
Zusammenspiel von Provider, Higher-Order-Komponente und Komponente

Fertige Redux-Anwendung

Damit haben wir nun unsere kleine Anwendung komplett auf Redux umgebaut und dabei alle wichtigen Redux-Konzepte kennengelernt. Diese Konzepte zeichnen sich allerdings erst in größeren Anwendungen wirklich aus. Für eine so kleine Anwendung wie in unserem Beispiel bedeutet der Einsatz von Redux zu viel Overhead und ist sicherlich nicht gerechtfertigt.

Das fertige Beispiel findest du in `steps/12_redux/hellomessage-redux/01-redux`. Du kannst es dort mit `npm start` ausführen.

Abb. 12–7
Das fertige Beispielprojekt

12.3 Redux und Vanilla Flux

Wir nutzen in diesem Kapitel das Redux-Framework, das in manchen Punkten von der ursprünglich von Facebook vorgeschlagenen Flux-Architektur abweicht. In diesem Abschnitt stellen wir dir die ursprüngliche Flux-Idee (Vanilla Flux) vor und erklären, wo es Abweichungen gibt und warum.

Unidirectional Dataflow

Die zentrale Idee von Flux ist es, dass Daten grundsätzlich nur in einer Richtung durch eine Anwendung fließen. Das soll die Fehleranfälligkeit verringern und die Nachvollziehbarkeit der Anwendung erhöhen. In vielen anderen Komponentenarchitekturen interagieren die Komponenten durch Event-Listener direkt miteinander, was schnell