Kostenloses eBook

LERNEN

redux

Free unaffiliated eBook created from
Stack Overflow contributors.

#redux

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: redux

It is an unofficial and free redux ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official redux.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit Redux

Bemerkungen

Redux ist eine JavaScript-Bibliothek, die Statuscontainer der Flux-basierten Architektur implementiert.

Redux kann in drei Grundprinzipien beschrieben werden:

1. Einzelne Quelle der Wahrheit (Single Store)
2. Zustand ist schreibgeschützt (Aktionen zum Signalisieren einer Änderung erforderlich)
3. Änderungen werden mit reinen Funktionen durchgeführt (Dies erzeugt einen neuen Status entsprechend den Aktionen)

Hauptteile:

- Ladenbauer
- store.dispatch (Aktion)
- Middleware
- Reduzierungen

Redux ist erstaunlich einfach. Es verwendet eine Funktion, die als Reduzierer bezeichnet wird (ein Name, der von der JavaScript-Reduzierungsmethode abgeleitet wird), der zwei Parameter übernimmt: eine Aktion und einen nächsten Zustand.

Der Reduzierer hat Zugriff auf den aktuellen Status (bald vorzeitig), wendet die angegebene Aktion auf diesen Status an und gibt den gewünschten nächsten Status zurück.

Reduzierungen sind als reine Funktionen konzipiert. Das heißt, sie produzieren keine Nebenwirkungen. Wenn Sie die gleichen Eingangswerte 100 Mal an einen Reduzierer übergeben, erhalten Sie den gleichen Ausgangswert 100 Mal. Nichts Seltsames passiert. Sie sind vollständig vorhersehbar. Als jemand mit einer prominenten Haftnotiz "NO SURPRISES" auf meinem Monitor ist dies eine wunderbare Idee, die man sich vorstellen kann.

Reduzierungen speichern den Zustand nicht und sie verändern den Zustand NICHT. Sie sind bestanden und kehren zurück. So sehen Reduzierungen in Aktion aus.
http://redux.js.org/docs/basics/Reducers.html

Referenz: http://redux.js.org/docs/introduction/Motivation.html

Versionen

Versionen	Veröffentlichungsdatum
v3.6.0	2016-09-04
v3.5.0	2016-04-20

Versionen	Veröffentlichungsdatum
v3.4.0	2016-04-09
v3.3.0	2016-02-06
v3.2.0	2016-02-01
v3.1.0	2016-01-28
v3.0.0	2015-09-13
v2.0.0	2015-09-01
v1.0.0	2015-08-14

Examples

Installation oder Setup

Grundinstallation:

Sie können die JavaScript-Datei von redux über diesen Link herunterladen.

Sie können auch Redux mit Bower installieren:

```
bower install https://npmcdn.com/redux@latest/dist/redux.min.js
```

Als nächstes müssen Sie Redux in Ihre Seite aufnehmen:

```
<html>
  <head>
    <script type="text/javascript" src="/path/to/redux.min.js"></script>
  </head>
  <body>
    <div id="app"></div>
    <script>
      // Redux is available as `window.Redux` variable.
    </script>
  </body>
</html>
```

Npm Installation:

Wenn Sie npm verwenden , um redux zu installieren, müssen Sie Folgendes ausführen:

```
npm install redux --save
```

Um redux verwenden zu können, müssen Sie es als Nächstes verwenden (vorausgesetzt, Sie verwenden einen Modul- Bundler wie Webpack):

```
var redux = require('redux');
```

Oder wenn Sie einen Es6-Transpiler wie babel verwenden :

```
import redux from 'redux';
```

Vanilla Redux-Beispiel (ohne Reaktion oder andere)

Sie können die laufende Demo sehen, indem Sie hier klicken .

HTML:

```
<p>
  <span>Counter State</span><br />
  (<em>Will increase each minute</em>):
  <p>
    <span id="counter-state" style="font-weight: bolder"></span>
  </p>
</p>

<p>
  <button id="increment-action">+ Increase +</button>
  <button id="decrement-action">- Decrease -</button>
</p>
```

REDUX LOGIC:

```
// ------------------------ reducer helpers -------------------------
let reducers = {}

let addReducer = (reducers, actionType, reducer) =>
  reducers[actionType] = (state, action) => {
    if (action.type == actionType) {
      return reducer(state, action)
    }
  }

let reducer = (state, action) => {
  if (reducers[action.type]) {
    return reducers[action.type](state, action)
  }
  return state
}


// ------------------------ redux setup -------------------------

const {
  createStore,
  applyMiddleware
  } = Redux


// apply logging middleware (not necessary)
// open the console to see the action logger output on each time new action dispatched
const actionLogger = ({ dispatch, getState }) => next => action => {
```

```
  console.log("action logger: action.type=%s state=%d", action.type, getState())
  return next(action)
}


// ------------------------ reducers ------------------------
// those will be creating new states and returning it,
// depending on the dispatched actions
addReducer(reducers, 'INCREMENT', (state, action) => ++state)
addReducer(reducers, 'DECREMENT', (state, action) => --state)


const DEFAULT_STATE = 0

const store = createStore(
  reducer,
  DEFAULT_STATE,
  applyMiddleware(actionLogger)
);


console.log(createStore)


// ------------------------ rendering ------------------------
let render = () => document.getElementById('counter-state').innerHTML = store.getState()

//
// IMPORTANT BINDING:
//
// store will be dispatching the new state to the render method each time the state changes
//
store.subscribe(render)

//
// render with the state for the first time
//
render()



// ------------------------ redux actions ------------------------

// continously increment the counter (the state) each second
setInterval(() => store.dispatch({type: 'INCREMENT'}), 1000)

// only increment the counter on click to the increase button
document
  .getElementById('increment-action')
  .addEventListener('click', () => store.dispatch({type: 'INCREMENT'}))

// only decrement the counter on click to the decrease button
document
  .getElementById('decrement-action')
  .addEventListener('click', () => store.dispatch({type: 'DECREMENT'}))
```

Erste Schritte mit Redux online lesen: https://riptutorial.com/de/redux/topic/1101/erste-schritte-mit-redux

Kapitel 2: Asynchroner Datenfluss

Examples

Redux-Thunk: Grundlagen

Während redux selbst vollständig synchron ist, können Sie eine Middleware wie `redux-thunk` , um asynchrone Aktionen `redux-thunk` .

> Ein "Thunk" ist ein anderer Name für einen Rückruf. Diese Funktion wird normalerweise als Argument übergeben, das zu einem späteren Zeitpunkt aufgerufen werden soll.

Zur Verwendung wenden Sie die Middleware auf Ihren Redux-Store an:

```
import ReduxThunk from 'redux-thunk';

const store = createStore(
    reducer,
    applyMiddleware(ReduxThunk)
);
```

Auf diese Weise können Sie einen Thunk anstelle eines einfachen Objekts an den `dispatch` . Die Middleware erkennt den Thunk und ruft ihn an. Der Thunk nimmt die `dispatch` des Geschäfts als Parameter an:

```
// an asynchronous action - "thunk"
// This will wait 1 second, and then dispatch the 'INCREMENT' action
const delayedIncrement = dispatch => setTimeout(() => {
    dispatch({
        type: 'INCREMENT'
    });
}, 1000);

// dispatch the thunk.
// note: no () as we're passing the function itself
store.dispatch(delayedIncrement);
```

Redux-thunk mit jQuery.ajax verwenden

```
const loadUser = userId => dispatch => {
    dispatch({ type: 'USER_LOADING' });
    $.ajax('/users/' + userId, {
        type: 'GET',
        dataType : 'json'
    }).done(response => {
        dispatch({ type: 'USER_LOADED', user: response });
    }).fail((xhr, status, error) => {
        dispatch({ type: 'USER_LOAD_ERROR', status, error });
    });
};
```

Um zu verwenden, senden Sie es wie jeder andere Ersteller der Aktion:

```
store.dispatch(loadUser(123));
```

Dies führt dazu, dass eine erste `USER_LOADING` Aktion `USER_LOADING` wird, die zur Anzeige eines `USER_LOADING` verwendet werden kann (falls gewünscht). Nachdem die Antwort empfangen wurde, wird entweder eine `USER_LOADED` Aktion oder eine `USER_LOADED` Aktion `USER_LOAD_ERROR` , abhängig vom Ergebnis der `$.ajax` Anfrage.

Middleware

Wenn Sie `store.dispatch(actionObject)` es synchron behandelt. Das heißt, Reduktionen werden aufgerufen, und Ihre Store-Listener werden benachrichtigt, und Ihre Reaktionsansichten werden für jede ausgelöste Aktion erneut dargestellt.

Mit Middleware können Sie den Versand verzögern oder sogar verschiedene Aktionen in der Mitte ausführen. Durch die Middleware sehen Ihre asynchronen Aktionen synchron aus.

```
const myAsyncMiddleware = (store) => {
    return (next) => {
        return (action) => {
            if(action.type === "ASYNC_ACTION") {
              setTimeout(() => {
                store.dispatch({ type: "ASYNC_ACTION_RESPONSE" });
              }, 1000);
            } else {
              return next(action);
            }
        }
    }
}

const store = createStore(
  reducer,
  applyMiddleware(myAsyncMiddleware)
);
```

Aktionsersteller

Ein weiterer Ansatz für die Asynchronität in Redux besteht in der Verwendung von Aktionserstellern. In Flux sind Aktionsersteller spezielle Funktionen, die Aktionsobjekte konstruieren und auslösen.

```
myActionCreator(dispatch) {
  dispatch({ type: "ASYNC_ACTION_START" });
  setTimeout(() => {
    dispatch({ type: "ASYNC_ACTION_END" });
  }, 1000)
}
```

Verwenden Sie benutzerdefinierte Middleware + Superagent

Dieses Beispiel wurde dieser Boilerplate entnommen.

Benutzerdefinierte Middleware:

```
export default function clientMiddleware() {
  return ({dispatch, getState}) => {
    return next => action => {
      if (typeof action === 'function') {
        return action(dispatch, getState);
      }

      const { promise, types, ...rest } = action; // eslint-disable-line no-redeclare
      if (!promise) {
        return next(action);
      }

      const [REQUEST, SUCCESS, FAILURE] = types;
      next({...rest, type: REQUEST});

      const client = new ApiClient();
      const actionPromise = promise(client);
      actionPromise.then(
        (result) => next({...rest, result, type: SUCCESS}),
        (error) => next({...rest, error, type: FAILURE})
      ).catch((error)=> {
        console.error('MIDDLEWARE ERROR:', error);
        next({...rest, error, type: FAILURE});
      });

      return actionPromise;
    };
  };
}
```

`Superagent` Bibliothek für API-Aufruf `Superagent` :

```
import superagent from 'superagent';
import config from '../config';

const methods = ['get', 'post', 'put', 'patch', 'del'];

function formatUrl(path) {
  const adjustedPath = path[0] !== '/' ? '/' + path : path;
  return adjustedPath;
}

export default class ApiClient {
  constructor(req) {
    methods.forEach((method) =>
      this[method] = (path, { params, data } = {}) => new Promise((resolve, reject) => {
        const request = superagent[method](formatUrl(path));

        if (params) {
          request.query(params);
        }

        if (data) {
          request.send(data);
        }
```

```
        request.end((err, { body } = {}) => err ? reject(body || err) : resolve(body));
      }));
  }
  empty() {}
}
```

Verwenden von Redux-Thunk mit Versprechungen

```
import 'whatwg-fetch';

function checkStatus(response) {
  if (response.status >= 200 && response.status < 300) {
    return response;
  }
  const error = new Error(response.statusText);
  error.response = response;
  throw error;
}

function parseJSON(response) {
  return response.json();
}

function getJSON(endpoint, params) {
  return fetch(endpoint, params)
    .then(checkStatus)
    .then(parseJSON);
}

export function action() {
  return dispatch => getJSON('/example-endpoint')
    .then((result) => {
      dispatch({
        type: GET_SUCCESS,
        result,
      });
    })
    .catch((error) => {
      dispatch({ type: GET_FAILURE, error });
    });
}
```

Asynchroner Datenfluss online lesen: https://riptutorial.com/de/redux/topic/3474/asynchroner-datenfluss

Kapitel 3: Pure Redux - Redux ohne Rahmen

Parameter

Parameter	Beschreibung
Aktion	Es muss ein Objekt mit mindestens der `type` Eigenschaft sein. Jede andere Eigenschaft kann übergeben werden und ist innerhalb der Reduzierfunktion verfügbar.

Bemerkungen

Wenn Sie keine Bundler wie Webpack und Browserify verwenden, ändern Sie die erste Zeile in:

```
const { createStore } = Redux;
```

Oder rufen Sie es einfach direkt aus dem Redux Global an, wenn Sie den Store erstellen:

```
const store = Redux.createStore(counter);
```

Examples

Vollständiges Beispiel

index.html

```
<button id="increment">Increment</button>
<button id="decrement">Decrement</button>
<p id="app"></p>
```

---

index.js

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

function counter(state = 0, action) {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;

    case 'DECREMENT':
      return state - 1;

    default:
      return state;
  }
}
```

```
const store = createStore(counter);

function render() {
  const state = store.getState();
  document.querySelector('#app').innerHTML = `Counter: ${state}`;
}

const incrementButton = document.querySelector('#increment');
const decrementButton = document.querySelector('#decrement');

incrementButton.addEventListener('click', () => {
  store.dispatch({ type: 'INCREMENT' });
});

decrementButton.addEventListener('click', () => {
  store.dispatch({ type: 'DECREMENT' });
});

store.subscribe(() => {
  render();
});

render();
```

Pure Redux - Redux ohne Rahmen online lesen: https://riptutorial.com/de/redux/topic/4079/pure-redux---redux-ohne-rahmen

Kapitel 4: Redux-Apps testen

Examples

Redux + Mokka

Da Redux sehr funktional ist, ist das Testen von Einheiten sehr einfach.

Aktionsersteller:

```
export function showSidebar () {
  return {
    type: 'SHOW_SIDEBAR'
  }
}
```

Aktionstest Unit-Test:

```
import expect from 'expect'
import actions from './actions'
import * as type from './constants'

describe('actions', () => {
  it('should show sidebar', () => {
    const expectedAction = {
      type: type.SHOW_SIDEBAR
    }
    expect(actions.showSidebar()).toEqual(expectedAction)
  })
})
```

Testen eines Redux-Stores mit Mocha und Chai

```
import { expect } from 'chai';
import { createStore } from 'redux';

describe('redux store test demonstration', () => {
  describe('testReducer', () => {
    it('should increment value on TEST_ACTION', () => {
      // define a test reducer with initial state: test: 0
      const testReducer = (state = { test: 0 }, action) => {
        switch (action.type) {
          case 'TEST_ACTION':
            return { test: state.test + 1 };
          default:
            return state;
        }
      });

      // create a redux store from reducer
      const store = createStore(testReducer);

      // establish baseline values (should return initial state)
```

```
      expect(store.getState().test).to.equal(0);

      // dispatch TEST_ACTION and expect test to be incremented
      store.dispatch({ type: 'TEST_ACTION' });
      expect(store.getState().test).to.equal(1);

      // dispatch an unknown action and expect state to remain unchanged
      store.dispatch({ type: 'UNKNOWN_ACTION' });
      expect(store.getState().test).to.equal(1);
    });
  });
});
```

Redux-Apps testen online lesen: https://riptutorial.com/de/redux/topic/4059/redux-apps-testen

Kapitel 5: Reduzierstück

Bemerkungen

Reduzierungen ändern den Anwendungsstatus basierend auf den ausgelösten Aktionen.

Der Zustand ist unveränderlich, das heißt, Reduktionen sollten rein sein: Für dieselbe Eingabe sollten Sie **immer** dieselbe Ausgabe erhalten. Aus diesem Grund ist die Veränderlichkeit in Reduzierstücken verboten.

Examples

Grundminderer

Ein grundlegender Reduzierer würde so aussehen:

```
// Import the action types to recognise them
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';


// Set up a default state
const initialState = {
    error: undefined,
    entities: []
};

// If no state is provided, we take the default state
export default (state = initilState, action) => {

    // Based on the type of action received, we calculate the new state
    switch(action.type) {

        // Set the error
        case ACTION_ERROR:
            // Note that we create a new object, copy the current state into it,
            // and then we make the relevant changes, so we don't mutate the state
            return Object.assign({}, state, { error: action.error });

        // Unset any error, and load the entitites
        case ACTION_ENTITIES_LOADED:
            return Object.assign({}, state, {
                entities: action.entities,
                error: undefined
            };

        // Add only one entity. Again, note how we make a new entities array
        // combining the previous one with the new entity
        // instead of directly modifying it, so we don't mutate the state.
        case ACTION_ENTITY_CREATED:
            return Object.assign({}, state, {
                entities: [action.entity].concat(state.entities)
            }):
```

```
        // If the action is not relevant to this reducer, just return the state
        // as it was before.
        default:
            return state;
    }
};
```

Verwenden Sie unveränderliche

Immutable ist eine großartige Bibliothek, die uns unveränderliche Versionen weit verbreiteter Arten von Sammlungen wie Listen, Stacks, Maps und mehr bietet.

Es vereinfacht die Manipulation des Zustands und macht es einfacher, reine Berechnungen durchzuführen und Mutationen zu vermeiden.

Mal sehen, wie der Basic-Reducer mit den Karten- und Listenstrukturen von Immutable neu geschrieben werden kann:

```
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';

// Import Immutable
import Immutable from 'immutable';


// Set up a default state using a Map, a structure very similar to objects
// Note that states in Redux can be anything, not just objects
const initialState = Immutable.Map({
    error: undefined,
    entities: Immutable.List()
});

export default (state = initialState, action) => {

    switch(action.type) {

        case ACTION_ERROR:
            return state.set('error', action.error);

        case ACTION_ENTITIES_LOADED:
            return state.merge({
                entities: Immutable.List(action.entities)
                error: undefined
            });

        case ACTION_ENTITY_CREATED:
            return state.set('entities', state.entities.push(action.entity));

        default:
            return state;
    }
};
```

Wie Sie vielleicht gesehen haben, wird die Handhabung des unveränderlichen Zustands mit Immutable einfacher.

Basisbeispiel mit ES6-Spread

```
// Import the action types to recognize them
import { ACTION_ERROR, ACTION_ENTITIES_LOADED, ACTION_ENTITY_CREATED } from './actions';

// Set up a default state
const initialState = {
    error: undefined,
    entities: [],
    loading: true
};

// If no state is provided, we take the default state
export default (state = initialState, action) => {

    // Based on the type of action received, we calculate the new state
    switch(action.type) {

        // Set the error
        case ACTION_ERROR:
            // We will create new object with state,
            // which should be produced by error action
            return {
                entities: [],
                loading: false,
                error: action.error
            };

        // Unset any error, and load the entities
        case ACTION_ENTITIES_LOADED:
            return {
                entities: action.entities,
                error: undefined,
                loading: false
            };

        // Add only one entity. We will use spread operator (...) to merge
        // objects properties and to create new entity
        case ACTION_ENTITY_CREATED:
            return {
                ...state,
                entities: [action.entity].concat(state.entities)
            };

        // Every action is processed by each reducer,
        // so we need to return same state if we do not want to mutate it
        default:
            return state;
    }
};
```

Reduzierstück online lesen: https://riptutorial.com/de/redux/topic/6615/reduzierstuck

Kapitel 6: Wie verlinkt man Redux und reagiert?

Syntax

- `<Provider store>`
- `connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])`

Parameter

Streit	Beschreibung
Geschäft	Redux-Shop
mapStateToProps	`(state, ownProps) => resultProps` Benutzer bereitgestelltes Mapping: `(state, ownProps) => resultProps`

Examples

Anbieter

Um Ihren Redux-Store problemlos mit Ihren React-Komponenten zu verknüpfen, können Sie eine zusätzliche Bibliothek verwenden : rea-redux .

Zunächst müssen Sie Ihre App in einen `Provider` , bei dem es sich um eine Komponente handelt, die Ihren Shop übergibt und von untergeordneten Komponenten verwendet wird:

```
import { Provider } from 'react-redux';

/// ... store = createStore()

const App = () => (
  <Provider store={store}>
    <MyComponent>
  </Provider>
)
```

Zuordnungsstatus zu Eigenschaften

Nachdem Sie Ihre App in einen Provider eingebunden haben, können Sie die `connect` verwenden, `connect` Ihre Komponente zu abonnieren, um Änderungen zu speichern und eine Zuordnung zwischen den Eigenschaften des Redux-Status und den Eigenschaften der React-Komponenten bereitzustellen:

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

```
const MyComponent = ({data})  => (
  <div>{data}</div>
);

const mapStateToProps = (state, ownProps) => ({
  data: state.myComponentData
});

connect(mapStateToProps)(MyComponent);
```

Abgeleitete Daten merken

In Ihrem Redux-Store halten Sie die Rohdaten. Manchmal reichen die Rohdaten aus, aber manchmal müssen Sie neue Daten aus den Rohdaten ableiten, häufig durch Kombination von Teilen der Rohdaten.

Ein gebräuchlicher Anwendungsfall zum Ableiten von Daten ist das Filtern einer Liste von Daten basierend auf Kriterien, wobei sowohl die Liste als auch die Kriterien geändert werden können.

Das folgende Beispiel implementiert mapStateToProps und filtert eine Liste von Zeichenfolgen, um diejenigen, die mit einer Suchzeichenfolge übereinstimmen , beizubehalten. Dabei wird eine neue Eigenschaft `filteredStringList` erstellt, die von einer React-Komponente dargestellt werden kann.

```
// Example without memoized selectors
const mapStateToProps(state) => {
    const {stringList, searchString} = state;

    return {
        filteredStringList: stringList
            .filter(string => string.indexOf(searchString) > -1)
    };
}
```

Um die Reduzierungen einfach zu halten, sollten Sie nur die Liste der Daten und die Filterkriterien im Geschäft aufbewahren. Dies bedeutet, dass Sie die Daten zur Lesezeit ableiten müssen (wie im obigen Beispiel).

Das Ableiten von Daten zur Lesezeit wirft zwei Probleme auf:

1. Wenn die gleichen Daten häufig abgeleitet werden, kann dies die Leistung beeinträchtigen.
2. Wenn dieselben Daten in verschiedenen Komponenten benötigt werden, stellen Sie möglicherweise fest, dass Sie den Code zum Ableiten von Daten duplizieren.

Die Lösung ist die Verwendung von Memos, die nur einmal definiert werden. Die Reaktion Gemeinschaft schlägt vor , die NPM - Bibliothek neu wählen memoized Selektoren zu erstellen. Im folgenden Beispiel erzielen wir dasselbe Ergebnis wie im ersten Beispiel, nur mit gespeicherten Selektoren.

```
// Create memoized selector for filteredStringList
import {createSelector} from 'reselect';
```

```
const getStringList = state => state.stringList;

const getSearchString = state => state.searchString;

const getFilteredStringList = createSelector(
    getStringList,
    getSearchString,
    (stringList, searchString) => stringList
        .filter(string => string.indexOf(searchString) > -1)
);

// Use the memoized selector in mapStateToProps
const mapStateToProps(state) => {
    return {
        filteredStringList: getStringList(state)
    };
}
```

Beachten Sie, dass die beiden ersten Selektoren `getStringList` und `getSearchString` *nicht* gespeichert werden, da sie so einfach sind, dass sie keinen Leistungsgewinn bieten. Sie müssen noch erstellt werden, da wir sie als Abhängigkeiten an `createSelector` , damit sie wissen, wann das gespeicherte Ergebnis wiederverwendet wird und wann ein neues Ergebnis berechnet wird.

Der Memo-Selector verwendet die übergebene Funktion als letztes Argument, das an `createSelector` , um die abgeleiteten Daten zu berechnen (in unserem Beispiel die Funktion, die die gefilterte `createSelector` zurückgibt). Wenn bei jedem Aufruf des Memo-Selektors die Abhängigkeiten seit dem letzten Aufruf nicht geändert wurden (in unserem Beispiel `stringList` und `searchString`), gibt der Memo-Selector das vorherige Ergebnis zurück und spart damit die Zeit, die für die Neuberechnung erforderlich ist.

Sie können sich Selektoren (mit oder ohne Memo) als Getter für den Ladezustand vorstellen, genau wie Action-Creators Setter sind.

Weitere Beispiele zur Berechnung abgeleiteter Daten finden Sie im Abschnitt Rezepte der Redux-Dokumentation .

Wie verlinkt man Redux und reagiert? online lesen: https://riptutorial.com/de/redux/topic/6621/wie-verlinkt-man-redux-und-reagiert-

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Redux	1ven, Arijit Bhattacharya, Community, Inanc Gumus, jpdelatorre , Random User, uddhab, Vanuan
2	Asynchroner Datenfluss	Ali Sepehri.Kh, Arijit Bhattacharya, Franco Risso, Ming Soon, rossipedia, Vanuan
3	Pure Redux - Redux ohne Rahmen	Guilherme Nagatomo, Vanuan, Vishnu Y S
4	Redux-Apps testen	Mario Tacke, Shuvo Habib, Vanuan
5	Reduzierstück	Jurosh, Marco Scabbiolo
6	Wie verlinkt man Redux und reagiert?	ArneHugo, Vanuan