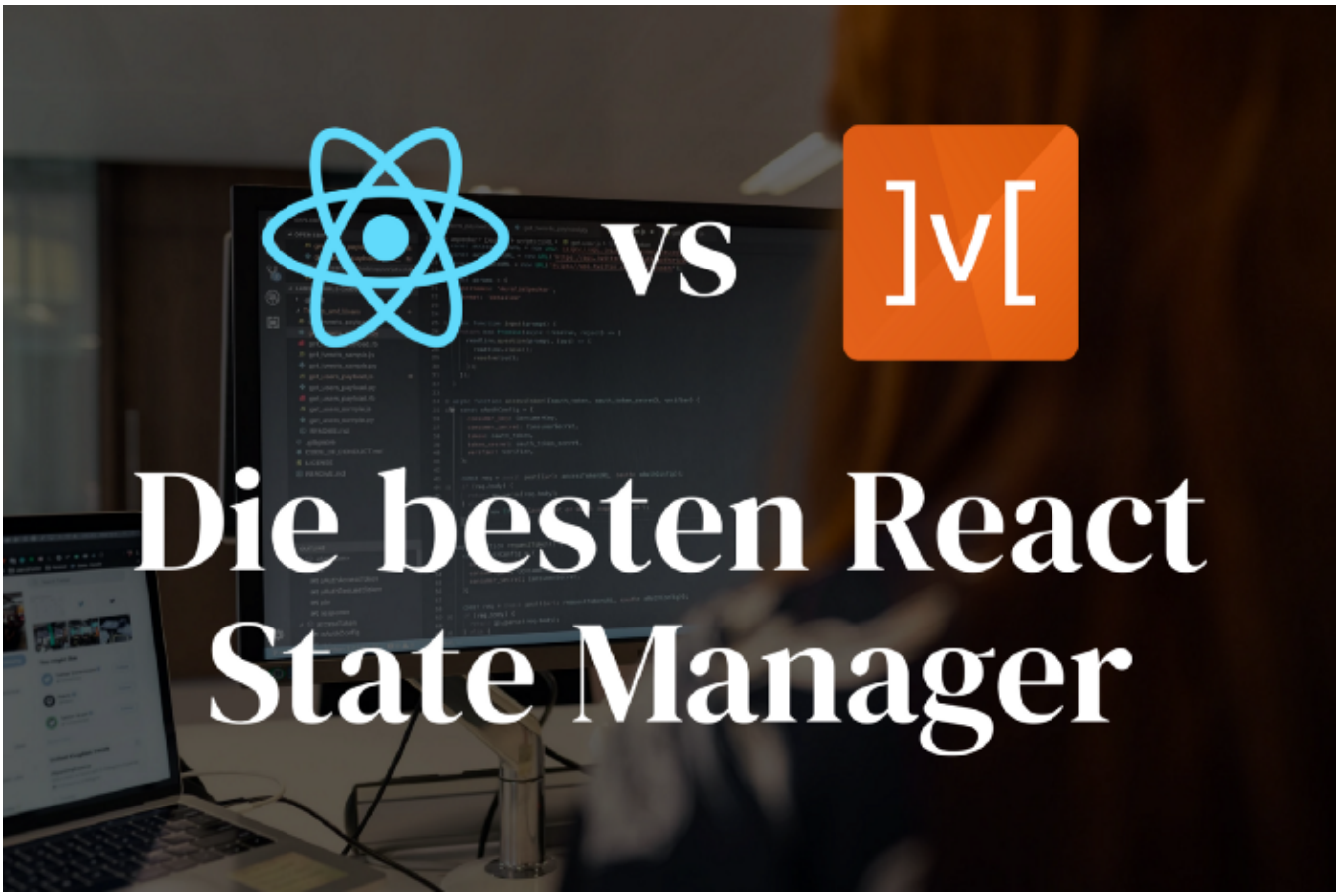
David Minkovski Follow

Jul 28, 2021 · 5 min read ★



Die besten React State Manager

Die meisten React-Entwickler wissen, wie wichtig das lokale und globale Zustandsmanagement für eine saubere Architektur und das Endbenutzererlebnis ist.
React-Entwickler wollen, dass die Zustandsverwaltung einfach, erweiterbar und

Aber es können dennoch Probleme auftreten, wenn der Zustand von vielen Komponenten geteilt werden soll. Dafür müssen Entwickler Tools und Bibliotheken finden, die ihren Anforderungen entsprechen und gleichzeitig hohe Standards erfüllen, die für Anwendungen erforderlich sind.

In diesem Artikel werde ich die beliebtesten State Bibliotheken miteinander zu vergleichen und meinen Favoriten auswählen.

Etwas Context — Reacts eigene API

React verfügt über ein hervorragendes eigenes Werkzeug, um Daten über mehrere Komponenten hinweg bereitzustellen.
Das primäre Ziel von **Context** ist es, unendliche Props-Weitergaben zu vermeiden. Das Ziel ist es, ein Konzept zu entwickeln welches folgende Anwendungsfälle abdeckt: häufige Updates, Umstrukturierung, Einführung neuer Komponenten usw.

Obwohl all dies mit Context machbar ist, benötigt dieses Konzept sehr spezifische und lösungsorienterte Implementierung die viel Zeit erfordert. Der einzige Vorteil von Context besteht darin, dass es nicht von einer Drittanbieterbibliothek abhängt, aber dieser kann den gesamten Aufwand zur Nutzung dieses Konzepts nicht legitimieren.

Darüber hinaus hat React-Teammitglied Sebastian Markbage erwähnt, dass die neue Context API nicht für häufige Updates entwickelt und optimiert wurde.

Existierende Bibliotheken

Auf GitHub gibt es Dutzende von State-Management-Tools (z. B. Redux, MobX, Recoil und Zustand). Die Berücksichtigung jedes einzelnen würde zu endlosen Recherchen, Tests und Vergleichen führen. Aus diesem Grund habe ich meine Auswahl auf zwei Hauptkonkurrenten basierend auf ihrer Popularität, Verwendung und Wartung eingeschränkt.

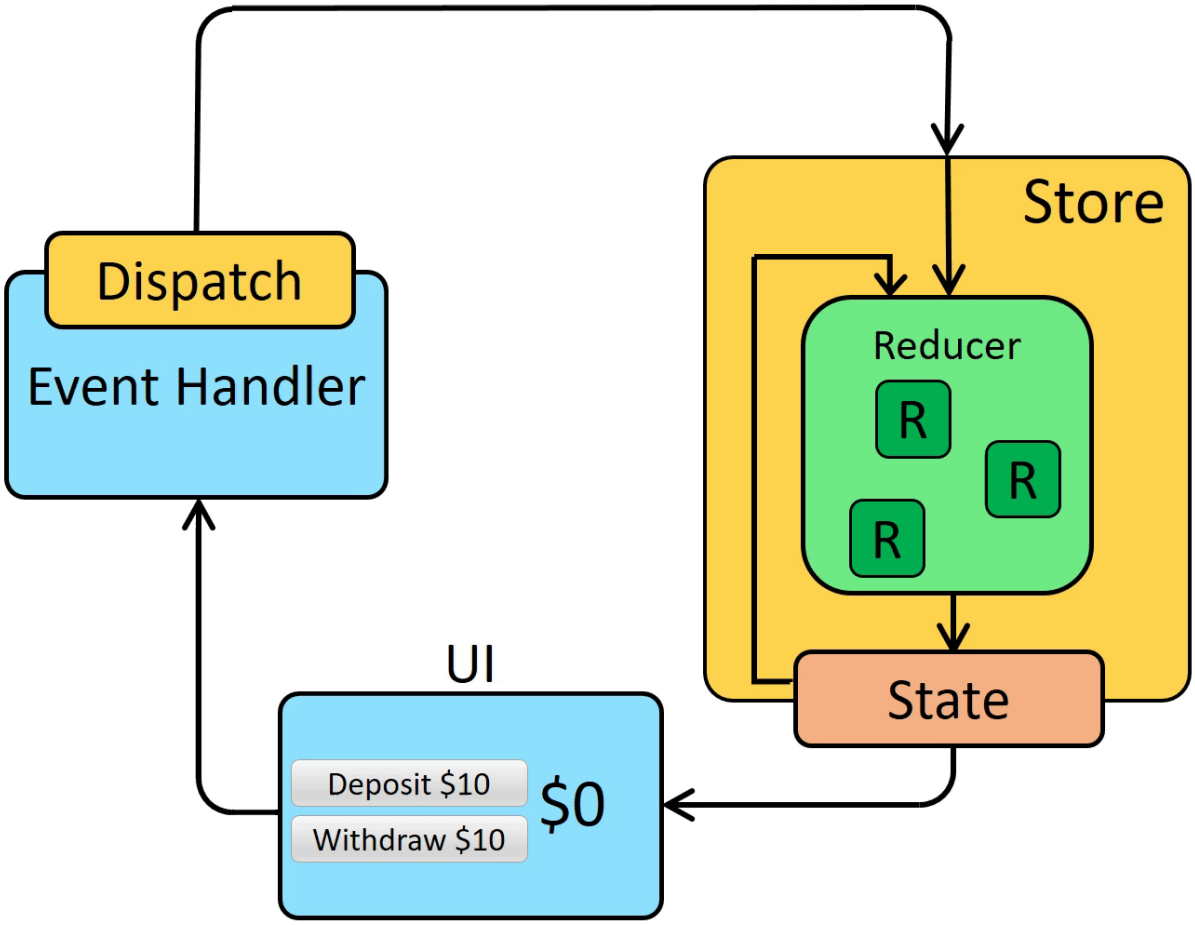
Um den Vergleich deutlich zu machen, werde ich die folgenden Qualitätsmerkmale berücksichtigen:

- Benutzerfreundlichkeit

- Veränderbarkeit
- Wiederverwendbarkeit
- Community

Redux

Redux ist eine State Technologie aus dem Jahr 2015. Es wurde schnell beliebt aufgrund folgender Faktoren. Es gab damals keine ernsthafte Alternative.
Es sorgte für eine Trennung zwischen State und Actions.
React-Redux ermöglichte eine direkte State Verbindung.
Der Mitgründer der bekannten Bibliothek ist Mitglied des React-Kernteams Dan Abramov.



Quelle: https://redux.js.org/tutorials/essentials/part-1-overview-concepts

Code Beispiel

```
// slices/counter.js
import { createSlice } from "@reduxjs/toolkit";

export const slice = createSlice({
  name: "counter",
  initialState: {
    value: 0
  },
  reducers: {
    plus: (state) => {
      state.value += 1;
    },
    minus: (state) => {
      state.value -= 1;
    }
  }
});

export const actions = slice.actions;
export const reducer = slice.reducer;


// store.js
import { configureStore } from "@reduxjs/toolkit";
import { reducer as counterReducer } from "./slices/counter";

export default configureStore({
  reducer: {
    counter: counterReducer
  }
});


// index.js
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import App from './App'
import store from './store'

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

```
import { actions } from "./slices/counter";

const App = () => {
  const count = useSelector((state) => state.counter.value);
  const dispatch = useDispatch();

  return (
    <div>
      <div>
        <button onClick={() =>
dispatch(actions.plus())}>Plus</button>
        <span>{count}</span>
        <button onClick={() =>
dispatch(actions.minus())}>Minus</button>
      </div>
    </div>
  );
};

export default App;
```

Qualitätsmerkmale

Benutzerfreundlichkeit

Redux wird durch Nutzung von react-redux sehr einfach. Sie erstellen einen Reducer, definieren Aktionen und übergeben es an den Store. Dann greifen wir über Hooks in einer Komponente darauf zu.

Wartbarkeit

Redux erfordert kein tiefes Wissen, um es zu verstehen. Die gesamte Logik wird durch die Reducer abgedeckt.

Testbarkeit

Redux besteht aus reinen Funktionen (Aktionen und Reducern) und eignet sich daher hervorragend für Unit-Tests und Integrationstests.

Skalierbarkeit

Redux hat einen globalen State, was die Skalierung erschwert. Es gibt jedoch einige Bibliotheken, die die Erstellung modularer Reducer und Middleware ermöglicht.

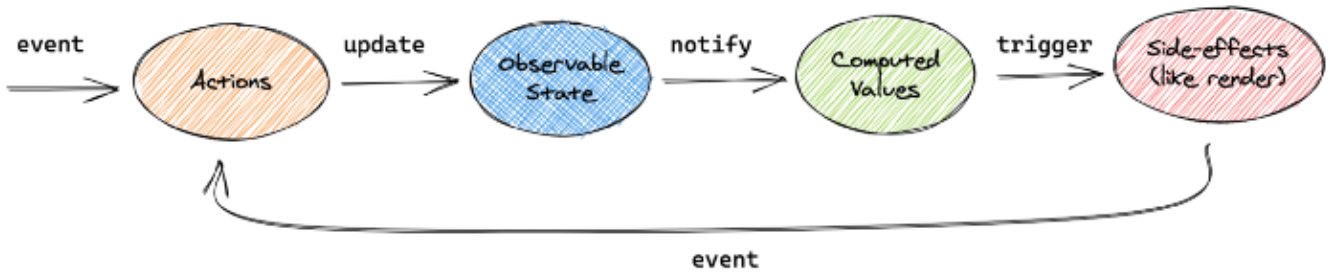
Veränderbarkeit

Wiederverwendbarkeit
Redux ist Framework-unabhängig, daher ist es sehr gut wiederverwendbar.

Community
Redux hat eine große Community mit einer riesigen Wissensbasis aufgebaut. Es gibt über 19.000 beantwortete Fragen auf Stack Overflow.

MobX

MobX ist eine weitere State-Bibliothek und hat 23000 Sterne auf GitHub.
Was es von Redux unterscheidet, ist, dass es dem OOP-Paradigma folgt und sogenannte *Observables* verwendet. MobX wird von einer Gruppe von Open-Source-Entwicklern gepflegt.



Quelle:
https://camo.githubusercontent.com/c8e97f5bf3e6908e68b34fd187d97fac599afc4efc2807a3434782285db46357/68747470733a2f2f6d6f62782e6a732e6f72672f6173736574732f666c6f77322e706e67

In MobX erstellen wir eine JavaScript-Klasse mit einem makeObservable-Aufruf innerhalb des Konstruktors, der ein Observable Store ist (Man könnte auch den @observable Decorator verwenden). Dann deklarieren wir den Zustand und die Methoden der Klasse. Die Komponenten “abonnieren” diesen State. Dann kann man auf den Zustand, berechnete Werte und Aktionen zugreifen.

Ein weiteres wesentliches Merkmal von MobX ist die Veränderlichkeit. Es ermöglicht die automatische Aktualisierung des Status, falls Sie Nebenwirkungen vermeiden möchten.

Code Beispiel

```
  constructor() {
    makeAutoObservable(this);
  }

  plus() {
    this.value += 1;
  }

  minus() {
    this.value -= 1;
  }
}

export default CounterStore;


// index.js
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "mobx-react";
import App from "./App";
import CounterStore from "./stores/counter";

ReactDOM.render(
  <Provider counter={new CounterStore()}>
    <App />
  </Provider>,
  document.getElementById("root")
);


// App.js
import React from "react";
import { inject, observer } from "mobx-react";

const App = inject((stores) => ({ counter: stores.counter }))(
  observer(({ counter }) => {
    return (
      <div>
        <div>
          <button onClick={() => counter.plus()}>Plus</button>
          <span>{counter.value}</span>
          <button onClick={() => counter.minus()}>Minus</button>
        </div>
      </div>
    );
  })
);
```

Benutzerfreundlichkeit
Eine Observable Klasse ist der einzige Einstiegspunkt für die Zustandsverwaltung. Es macht die Verwendung von MobX dadurch sehr einfach.

Wartbarkeit
Die Verwendung von MobX in einem schlecht qualifizierten Team kann leicht zu Datenkonsistenzproblemen führen.

Testbarkeit
Die Stores sind einfache JavaScript-Objekte mit versteckten Funktionen. Das Testen ist genauso einfach wie für jede andere JavaScript-Klasse.

Skalierbarkeit
Beobachtbare Speicher werden logisch in Klassen aufgeteilt.
Es gibt daher keine Schwierigkeiten, MobX zu skalieren.

Veränderbarkeit
MobX ermöglicht die Erstellung von benutzerdefinierten Observables mit eigenem Verhalten. Das macht MobX sehr anpassbar.

Wiederverwendbarkeit
MobX ist Framework-unabhängig, daher ist es sehr gut wiederverwendbar.

Community
Es gibt über 1.000 beantwortete Fragen mit dem mobx-Tag auf Stack Overflow.

Der Favorit

MobX und Redux sind beides große Player auf dem Markt. Was sie voneinander unterscheidet, ist die Lernkurve. MobX erfordert ein grundlegendes Verständnis der reaktiven Programmierung. Wenn die Entwickler nicht ausreichend qualifiziert sind, kann die Anwendung am Ende Code-Inkonsistenzen, Leistungsprobleme und eine deutlich längere Entwicklungszeit verursachen.

Redux hat auch einige Probleme, hauptsächlich im Bezug auf Skalierbarkeit. Es gibt jedoch bewährte Lösungen für diese Probleme.