REACT

# HOOKS_

10+

**React Hooks & Custom React Hooks**

Eine Übersicht von Lenz Weber

CHEAT SHEET FÜR DEVELOPER

---

## HOOKS IN REACT

Das praktische Nachschlagewerk für den Einsatz von Hooks in React.

MAYFLOWER GMBH

# INHALT

Layout & Produktion
**Mayflower Studio R18**

**VORWORT**
React Hooks & Custom React Hooks
04

**EINSTIEG**
useState
06

**EIGENE HOOKS**
Custom Hooks in React
07

**RULES OF HOOKS**
Die zwei wichtigsten Regeln im Umgang mit Hooks
08

**(LOKAL) STATE HANDLING**
useReducer und useLocalState
10

**SEITENEFFEKTE**
useEffect und useLayoutEffect
16

**MEMOISIERUNG**
useMemo und useCallback
22

**DIE ANDEREN**
useContext, useDebugValue, useRef und useImperativeHandle
26

**AUTOR & EMPFEHLUNGEN**
Infos zum Autor & Links und Tipps mit weiterern Informationen
34

**HINTERGRÜNDIGES**
Aus dem Blog: Das Redux-Jahr 2019 in Retrospektive
36

**ÜBER MAYFLOWER**
Was wir außer React noch alles treiben
38

10+

# REACT HOOKS

EIN EINSTIEG
VON LENZ WEBER

00





## HKS

Hooks sind seit der ReactConf 2018, spätestens aber seit ihrem offiziellen Release mit React 16.8 im Februar 2019 in aller Munde. Oft wird von "React mit Hooks" gesprochen, als ob es so etwas wie "React ohne Hooks" überhaupt noch gäbe.

Dabei sind Hooks einfach ein Teil von React – und so wie sich das Ökosystem gerade entwickelt, wird man um Hooks definitiv nicht mehr herum kommen.

Höchste Zeit also, sich mit diesem Teil von React vertraut zu machen, nicht zuletzt auch aus Eigennutz: Wenn man die praktischen Helferlein einmal eine Weile benutzt hat, möchte man eigentlich nicht mehr ohne.

Dieses Heftchen soll sowohl Anfängern eine Hilfe beim Einstieg in Hooks sein, als auch Fortgeschrittenen als ein Nachschlagewerk (und eine Einordnung) für die obskureren Hooks dienen.

Sei es jetzt die Verwendung von `useState`, oder aber auch die Besonderheiten beim Timing von `useImperativeHandle` – hier ist für jeden was dabei.

# useState

Der `useState`-Hook kann genutzt werden, um in einer Funktionskomponente einen Wert von einem Render zum nächsten zu speichern & mittels Setter zu verändern. Der Hook gibt State und Setter als Array zurück, was in der Regel direkt in zwei Variablen destrukturiert wird.

```
const [myState, setMyState] =
useState("initialValue");
```

Innerhalb einer Komponente können mehrere `useState`-Hooks nacheinander verwendet werden, um verschiedene States zu halten.

## ACHTUNG

Ruft man aus einem Callback mehrere State-Setter direkt nacheinander auf, kann das dazu führen, dass die Komponente mehrmals von React neu gerendert wird! In diesem Falle: siehe `useReducer`



# CUSTOM HOOKS

Das ist eigentlich alles, das man braucht, um einen ersten "custom Hook" zu schreiben. Ein Custom Hook ist einfach nur eine Funktion, deren Namen mit "use" beginnt, die weitere (eigene oder React-originäre) Hooks aufruft.

Hier ein Beispiel, wie man obigen `useState`-Hook schön für die Verwendung in Formularen verpacken kann:

```
function useInputState(initialValue) {
 const [state, setState] = useState(initialValue);
 return {
   value: state,
   onChange(event) {
     setState(event.target.value);
   }
 };
}

function MyInput() {
 const fistName = useInputState("Max");
 const lastName = useInputState("Mustermann");

 return (
   <>
     <p>
       you entered: {fistName.value} {lastName.value}
     </p>
     <p>
       <input {...fistName} />
       <input {...lastName} />
     </p>
   </>
 );
}
```

# DAS WICHTIGSTE

–

Oft wird man custom Hooks vorfinden, die viele andere Hooks aufrufen. Aber was ist hier jetzt eigentlich so besonders an unserer Funktion? Warum bezeichnen wir sie als "custom Hook" und warum muss ihr Name mit "use" beginnen?

## PLUG-IN

Man sollte sich das ESLint-Plugin "eslint-plugin-react-hooks" ins Projekt holen, da das einem hilft, die "Rules of Hooks" durchzusetzen. Und zur Eingangsfrage: Aus diesem Plugin kommt auch die Konvention, custom Hooks mit "use" zu beginnen.

### 1: HOOKS DÜRFEN NUR AUS FUNKTIONS-KOMPONENTEN HERAUS AUFGERUFEN WERDEN

Diese Regel ist relativ einfach zu erklären. Hooks interagieren mit React-Interna, um Werte zur Verfügung zu stellen – abhängig von der Komponente, in der sie aufgerufen werden. Sie außerhalb einer React-Komponente aufzurufen ergibt schlicht keinen Sinn.

### 2: (CUSTOM) HOOKS BEGINNEN IMMER MIT "USE"

Das ist nicht 100 prozentig eine zwingende Hooks-Regel, sondern Konvention, um Regel Eins besser zu erzwingen.

Wenn ich Hooks nur aus Funktionen, deren Name mit "use" beginnt, aufrufe und das nie ausserhalb von React-Funktionskomponenten, laufe ich nicht Gefahr, einen originären React-Hook außerhalb einer React-Funktionskomponente aufzurufen.

Und damit haben wir eine eigene "Klasse" von Funktionen geschaffen: custom Hooks, die mit "use" beginnen.

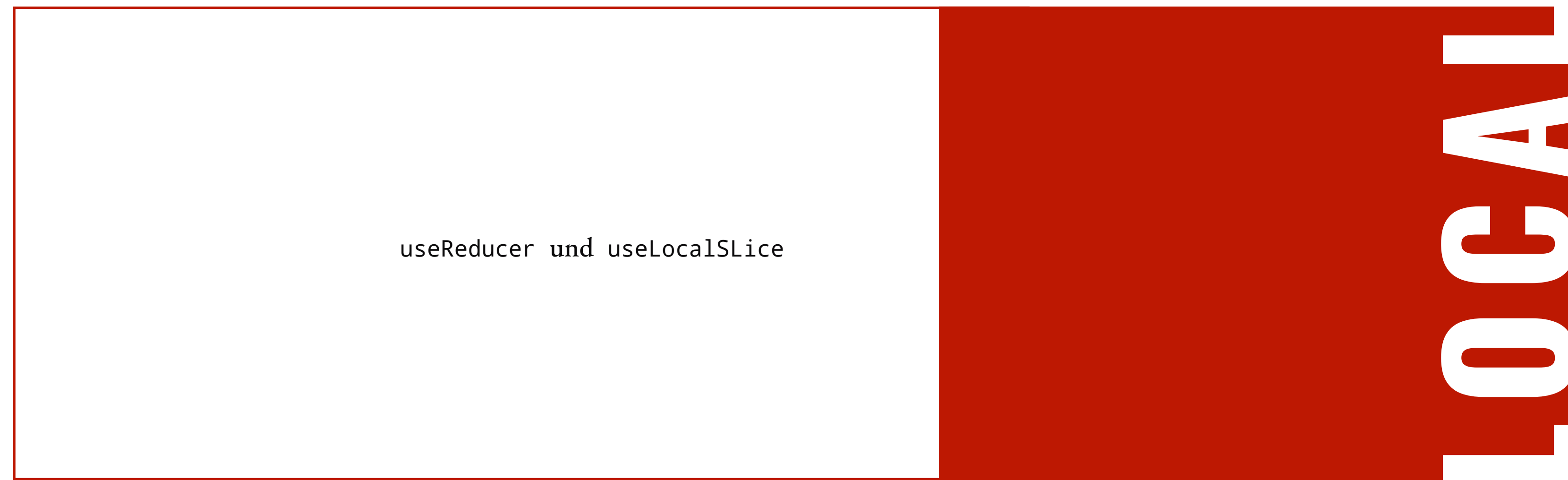
### 3: HOOKS SOLLTEN IMMER AUF DEM "TOP LEVEL" EINER FUNKTION AUFGERUFEN WERDEN

Das bedeutet, man darf Hooks nicht in if-Abfragen oder Schleifen verschachteln. Der Grund liegt auf den ersten Blick nicht 100-prozentig auf der Hand, ergibt sich aber leicht:

Angenommen, man habe eine Komponente, die drei Mal `useState` (mit Werten A, B, C) aufruft. Der zweite Aufruf von `useState` liegt aber hinter einer if-Abfrage. React kann jetzt aber nicht in unseren Code "hineingucken" und feststellen, warum es einmal mit drei States und einmal mit zwei States aufgerufen wurde. Wie ist aus der Sicht von React also das korrekte Verhalten? A und B zurückgeben? A und C? B und C?

Aus diesem Grund: Die Anzahl, der Typ und die Reihenfolge der Hooks, die man aufruft, dürfen sich nie ändern. Wenn man mal wirklich der Meinung ist, einen Hook in einer Schleife für N Elemente aufzurufen, ist das ein gutes Anzeichen dafür, dass man das Verhalten einer Kind-Komponente in der Elternkomponente abbilden möchte. In diesem Fall kann man meist eine Kind-Komponente erstellen, die den entsprechenden Hook nur einmal aufruft – und diese Komponente dann N-mal verwenden.

# (LOCAL) STATE HANDLING



# useReducer

Mit useReducer kann man, ähnlich wie bei Redux , das Reducer-Pattern verwenden: Man definiert eine Reducer-Funktion, die den Übergang zwischen verschiedenen States definiert und "dispatcht" dann nur noch Aktionen, die diese Reducer-Funktion anstößt. So lässt sich Logik besser von der Komponente trennen und man vermeidet das Problem des "mehrfachen Setters" – durch einen Dispatch können mehrere Werte verändert werden.

Zu beachten hierbei: in einem Reducer sollte man nie den alten State verändern, sondern immer eine veränderte Kopie (oder den unveränderten alten State) zurück geben.

```
const [state, dispatch] = useReducer((oldState, action) => {
    if (action.type === "toUpper") {
        return {
            ...oldState,
            userModifications: oldState.userModifications + 1,
            currentValue: oldState.currentValue.toUpper()
        };
    } else if (action.type === "concat") {
        return {
            ...oldState,
            userModifications: oldState.userModifications + 1,
            currentValue: oldState + action.payload
        };
    }
    // hier noch mehr Cases
    else {
        return oldState;
     }
}, { userModifications: 0, currentValue: "initial Value", someThingElse: {} });
```

```
// in einem EventHandler:
dispatch({type: "concat", payload: "additional content"})
```



## PRO-TIP

Wenn man den Reducer innerhalb der Komponente selbst schreibt, hat man direkten Zugriff auf alle Variablen, die im Scope des aktuellen Renders liegen, z.B. die aktuellen Props. Der Reducer wird nicht zum Zeitpunkt des `dispatch`-Calls aufgerufen, sondern während der nächsten Render-Phase, wenn `useReducer` aufgerufen wird.

## INSIDER-INFO

Intern ist `useState` auch nur ein Sonderfall von `useReducer` – mit einem Reducer in der Form von `(oldState, newState) => newState`

# useLocalSlice

`useReducer` ist in allen komplexeren Fällen schon die richtige Wahl. Insbesondere in einer TypeScript-Umgebung fühlt es sich aber etwas merkwürdig an, die Actions von Hand zu schreiben. Außerdem wird der Reducer so zu einem endlosen if-else-Monster.

Hier können wir uns vom "slice"-Pattern aus dem redux-toolkit inspirieren lassen und etwas ähnliches für `useReducer` implementieren. Die Bibliothek heisst "use-local-slice". Das vorangegangene Beispiel sähe dann wie folgt aus:

```
const [state, dispatchAction] = useLocalSlice({
      initialState: { userModifications: 0, currentValue: "initial Value",
      someThingElse: {} },
    reducers: {
        toUpper(draft) {
            draft.userModifications++;
            draft.currentValue = draft.currentValue.toUpper();
        },
        concat(draft, action: { payload: string }) {
            draft.userModifications++;
            draft.currentValue += action.payload;
        }
    }
})

dispatchAction.concat("additional Content");
```

Auch wenn dieses Cheat Sheet größtenteils TypeScript-frei bleiben soll haben wir hier eine kleine TypeScript-Definition rein geschummelt:

Das `action: { payload: string }` oben ist alles, was nötig ist um dafür zu sorgen, dass `dispatchAction.concat` später nur einen String als Argument annimmt. Würde man nur JavaScript schreiben, würde da nur `action` stehen.

## TIP

`useLocalState` wrappt intern alle Reducer in die `produce`-Funktion des immer-Packages. Deshalb kann man hier gefahrlos Werte zuweisen – am Ende wird trotzdem eine veränderte Kopie des State erzeugt und nicht der Original-State verändert. Gerade bei tief geschachtelten Veränderungen wird der Code hierdurch sehr viel lesbarer.

# SEITENEFFEKTE

Klassen bieten Lifecycle-Events an – bei der Einführung von Hooks haben die Entwickler von React bewusst darauf verzichtet, hier nur ein 1:1-Mapping anzubieten.

Statt `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`, und den ganzen in der Zukunft deprecated `unsafe__`-Variationen gibt es jetzt nur noch `useEffect` und `useLayoutEffect`, in Kombination mit dem dependency-Array.

Aber was heißt das jetzt eigentlich?

# useEffect

Werfen wir einen Blick auf den einfachsten Anwendungsfall:

```
useEffect(() => {

       console.log('effect called');
       return () => ('effect cleanup
       called');

})
```

Wird unsere Komponente jetzt gemounted, einmal neu gerendert und wieder geunmounted, haben wir folgenen Konsolen-Output: `effect called` (beim Mount), `effect cleanup called` und `effect called` (beim Render) sowie `effect cleanup called` (beim Unmount).

Das heißt, dass der Cleanup-Callback immer aufgerufen wird, ehe der Effekt ein weiteres Mal ausgeführt wird. Und der Effekt wird momentan immer ausgeführt, also jedes Mal wenn die Komponente gerendert wird.

Wie bekommen wir es jetzt hin, den Effekt nur beim Mount (bzw. den Cleanup nur beim Unmount) auszuführen? Behalten wir die Frage mal im Hinterkopf und schauen uns den optionalen zweiten Parameter von `useEffect` an.

## DAS DEPENDENCY-ARRAY

Und hier kommen wir an den Punkt, an dem Effekte beginnen, von den Lifecycle-Events abzuweichen. Wir können bei Effekten nämlich sehr granular bestimmen, wann genau sie ausgeführt werden.

Hier ein kleines Beispiel:

```
useEffect(() => {

       console.log('component was
       mounted or prop "a" changed,
       current value is ', props.a);

       return () => ('cleaning up,
       last value of "a" was',
       props.a);

}, [props.a])
```

Für diesen Effekt haben wir jetzt spezifiziert, dass er von `props.a` abhängt, d.h. der Effekt wird nur dann ausgeführt, wenn sich der Wert von `props.a` ändert (was natürlich beim Mount auch passiert). Natürlich kann ein Effekt auch von mehreren Werten abhängen – in diesem Fall würden wir einfach mehrere Werte an das Array übergeben.

Damit kommen wir auch wieder zurück zur Frage: wie lösen wir den Effekt/Cleanup nur bei Mount/Unmount aus? Indem man ein leeres Dependency-Array übergibt. Das ändert sich nie, d.h. der Effekt wird exakt einmal ausgeführt.



## ANMERKUNG

Hier können wir keinen 1:1 Vergleich zum Timing von `componentDidMount` und `componentDidUpdate` ziehen – die Lifecycle-Methoden werden während der Layout-Phase des Browsers (also vor dem Paint) aufgerufen, der Effekt erst nach dem Paint. Damit wirkt der Browser responsiver und meistens spielt es keine Rolle. Spielt es doch mal eine Rolle, siehe `useLayoutEffect`.

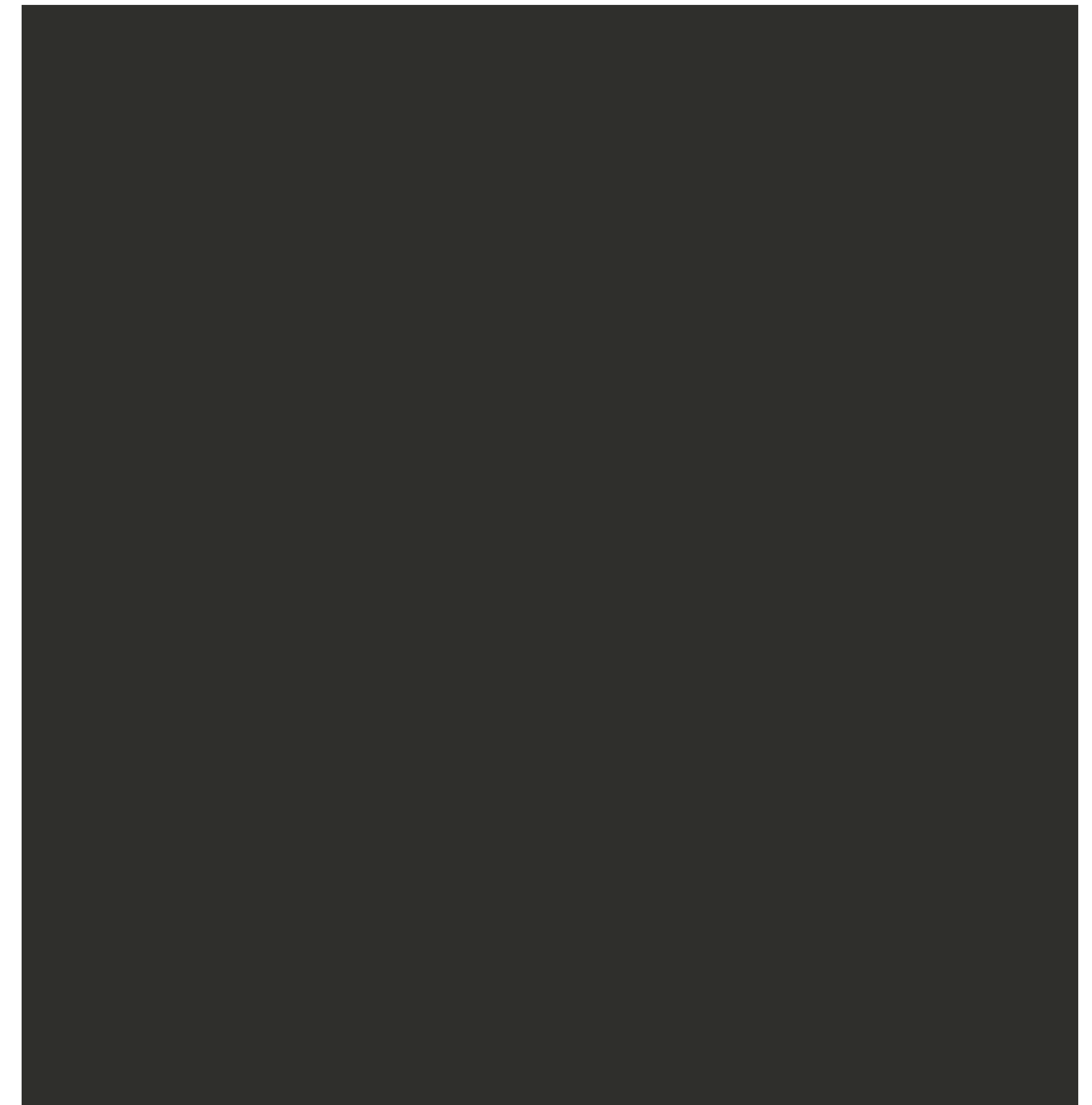
## HINWEISE

Tatsächlich sollte man, wenn man ein Dependency-Array mit Werten übergibt, alle dynamischen Werte (und Funktionen), die innerhalb der Effekt-Funktion benutzt werden, ins deps-Argument schreiben. Das Linter-Plugin "eslint-plugin-react-hooks" forciert dies auch. Gibt man hier einen Wert nicht an, ist das in fast allen Fällen ein Hinweis auf einen möglichen Bug – es gibt da nur sehr wenige Ausnahmen.

`useEffect` wird oft auch genutzt, um aus einer Komponente heraus API-Calls aufzurufen. Siehe dazu die Literaturliste.

# useLayoutEffect

Wie bereits beschrieben: `useEffect` wird nach der paint-Phase ausgeführt. Will man in einem Effekt jetzt noch schnell etwas ausmessen und ggf. darauf reagieren, ehe der Browser zeichnet, muss man das aber schon in der Layout-Phase tun. Hier kann man dann statt dessen `useLayoutEffect` benutzen. Wenn man React seinen Job tun lässt und nicht manuell am DOM manipuliert, sollte man diesen Hook aber fast nie brauchen.



# MEMOISIERUNG

Um Dinge memoisieren zu wollen gibt es grob zwei mögliche Gründe:

- Man hat eine teure Berechnung, um die man manchmal nicht herum kommt, die man aber nicht bei jedem Render ausführen will.
- Man definiert in einer Komponente ein Objekt oder eine Funktion. Wenn man diesen Wert dann an eine `PureComponent` (oder eine in `React.memo` gewrappte Komponente) oder in ein Dependency-Array gibt, ändert sich dennoch bei jedem Render die Objektreferenz – und die Kindkomponente wird neu gerendert, bzw. der Effekt neu ausgeführt. Man hat also eine Motivation, diese Objektreferenz gleich zu halten, solange die Inhalte des Objekts sich nicht verändern.

MEMO

`useMemo` und `useCallback`

# useMemo

Im Falle der teuren Berechnung rufen wir `useMemo` auf wie folgt:

```
const memoizedValue = useMemo( () =>
somethingExpensive(n), [n]);
```

Wir übergeben `useMemo` erst eine unseren Wert berechnende Funktion und dann deren dependecy-Array. Das funktioniert wie schon bei `useEffect`: ändert sich ein Wert im dependency-Array wird der Wert neu berechnet. Sonst wird die Funktion nicht neu ausgeführt und der gespeicherte (memoisierte) Wert der letzten Ausführung wird verwendet.

Möchten wir jetzt einfach nur eine Objektreferenz über mehrere Renderings stabil halten, solange sich keine Werte ändern, geht das recht einfach analog:

```
const stableValueReference = useMemo( () =>
({ a: valueA, b: valueB }), [valueA, valueB]);
```

Auch hier gilt: solange `valueA` und `valueB` gleich bleiben, bleibt `stableValueReference` über mehrere Renderings hinweg stabil eine Referenz auf das selbe Objekt. Ändern sich `valueA` oder `valueB`, bekommen wir ein neues Objekt mit den neuen Werten.



# useCallback

Analog zum nebenstehenden Beispiel könnten wir – falls wir eine stabile Referenz auf eine Funktion und nicht ein Objekt brauchen – auch folgendes schreiben:

```
const stableFunctionReference = useMemo( () => (e) =>
setValue(e.target.value), [setValue]);
```

Das ist aber durch die Funktion höherer Ordnung weder angenehm zu lesen, noch zu schreiben. Deshalb bietet React hier noch `useCallback` an:

```
const stableFunctionReference = useCallback( (e) =>
setValue(e.target.value), [setValue]);
```

Im Prinzip funktionieren beide Varianten genau gleich.

## ZU BEACHTEN

Achtung: Das Wissen um die Existenz von `useCallback` und `useMemo` verleitet einen gerne, alle Werte und alle Callback-Funktionen in `useCallback` und `useMemo` zu wrappen. Das erreicht aber oft keine Performanceoptimierung, sondern das Gegenteil. (Schließlich tauscht man hier Laufzeit gegen Speicher und man hat einen Funktionsaufruf mehr als vorher.)

Hierbei ist zu beachten: im Normalfall rendert React immer alle Kinder neu, unabhängig davon, ob diese die selben Props bekommen wie beim letzten Render. Die Ausnahme hierzu bilden Komponenten wie `PureComponents`, Komponenten die in `React.useMemo` gewrappt sind oder Komponenten, die `shouldComponentUpdate` implementieren. Nur in diesen Fällen sollte man `useMemo`/`useCallback` verwenden, um Referenzen stabil zu halten.

Und natürlich für die Nutzung als Dependency in `useEffect`.

04

# WEITERE HOOKS

—

## DIVERSES

`useContext`, `useDebugValue`, `useRef` und `useImperativeHandle`

# useContext

Dieser Hook ist einfach nur eine simple Alternative dazu, unsere Komponente in einen `Context.Consumer` zu wrappen. Hier ist zu beachten: `useContext` möchte als Argument das komplette Objekt, wie es von `createContext` zurückgegeben wird, mit `Provider` und `Consumer`. Die Anwendung ist dann denkbar einfach:

```
const contextValue =
useContext(MyContext);
```

### ZU BEACHTEN

Zu beachten: wenn sich nicht weiter oben im Komponentenbaum ein Provider für `MyContext` befindet, gibt `useContext` "undefined" zurück. Diesen Fall sollte man abfangen.

# useDebugValue

Dieser Hook hat nur innerhalb von custom Hooks Auswirkungen. Nutzt man ihn innerhalb eines custom Hooks einmal, wird der als Argument übergebene Wert direkt neben dem Hook-Namen in den DevTools angezeigt. Nutzt man ihn mehrmals, kann man den custom Hook aufklappen und sieht ein Array von "DebugValues".

### HINWEIS

Das funktioniert nur, wenn der custom Hook auch noch irgend einen anderen Hook aufruft, der in den React DevTools sichtbar ist, also einen Hook aus der Klasse "Hook mit State" – `useState/useReducer/useMemo/useCallback/useRef` (das könnte sich in zukünftigen Versionen der DevTools aber ändern).

# useRef

In einer Funktionskomponente bieten sich nicht die Möglichkeiten, ein Ref-Objekt zu halten, wie sie in einer Klassenkomponente zur Verfügung stehen, da dort üblicherweise jede Ref als Klassen-Property gehalten würde. Stattdessen bietet sich hier `useRef` an.

```
function MyComponent(props) {

    const myRef = useRef();
    useEffect(() => {

       console.log(myRef.current,
       props.x)

    }, [props.x])

    return (

       <div ref={myRef}>test</div>

    )

}
```

Auf den ersten Blick scheint das Beispiel relativ einleuchtend – wir bekommen ein Ref-Objekt (wie wir es auch mittels `createRef` bekommen würden) via `useRef`, und augenscheinlich sorgt React wie auch bei `useState` dafür, dass wir dieses Ref-Objekt über mehrere Renders hinweg konstant behalten können. Anschließend geben wir es als `ref`-Property an ein DOM-Element. Und dann greifen wir in `useEffect` (also nach dem Paint, zu einem Zeitpunkt, zu dem das DOM gerendert ist und die Referenz korrekt gesetzt sein wird) darauf zu.

Aber dann wird es eigenartig: in diesem Beispiel haben wir ein dependency-Array, aber weder `myRef`, noch `myRef.current` kommen hier vor. Im Kapitel zu `useEffect` hieß es noch, das wäre ein Hinweis auf einen möglichen Bug. Was ist hier also los?

Die Antwort darauf erscheint erst mal ein bisschen merkwürdig: Ref-Objekte existieren außerhalb des normalen Flows einer React-Applikation. Ein Update zu einem Ref-Objekt löst keinen Re-Render aus. Und somit wäre es auch nicht sinnvoll, so etwas wie das Ausführen eines Effekts (was ja immer nach einem Render stattfindet) in irgend einer Art und Weise an den Wert eines Refs zu binden.

Ein Ref-Objekt ist immer eine stabile Objektreferenz – und der Wert von `myRef.current` kann sich zwischen einem möglichen Aufruf von `useEffect` und der Ausführung des Effekts noch verändern.

Das findet hier beispielsweise beim Mount statt: zum Zeitpunkt des `useEffect`-Calls, also in der Render-Phase ist `myRef.current` noch `null`. Zum Zeitpunkt der Ausführung des Effekts, also frühestens in der Layout-Phase (für einen `useLayoutEffect`) existiert das Div bereits im DOM und `myRef.current` zeigt auf unser DOM-Element. Es wäre also vollkommen kontraproduktiv, `myRef` oder eben `myRef.current` als dependency zu verwenden.



## REF-OBJEKTE ALS "KLASSEN-PROPERTY-ERSATZ"

Wo man also in Klassen noch Werte, die irgendwie gehalten werden, aber keinen Re-Render auslösen sollten, noch in Properties speicherte, bietet sich hier mit `useRef` ein neues derartiges Schlupfloch. Man kann einfach Refs mittels `useRef` erhalten und Werte in `ref.current` halten – ähnlich einem Komponenten-State, der aber keinen Re-Render auslöst.

Das kann ein nützlicher Kniff sein, um Werte zu halten – und unter Umständen vorbei am Dependency-Array in einen `useEffect`-Aufruf zu schmuggeln.

## VORSICHT

Während diese Verwendung von `useRef` viele neue Möglichkeiten eröffnet, sollte sie sparsam genutzt werden. Übermäßige Nutzung dieses Patterns ist ein Code Smell und ein gutes Anzeichen dafür, dass man gegen React und nicht mit React arbeitet.

# useImperativeHandle

Das ist einer von diesen Hooks, die man vermutlich fast nie in der freien Wildbahn sieht. Er beantwortet die Frage "wie rufe ich aus einer Elternkomponente eine Methode meiner Kind-Komponente auf?" In der Welt der Klassenkomponenten war das noch recht leicht. Einfach der Komponente eine `ref`-Property mitgeben und diese Ref zeigt dann auf eine Instanz der Klassenkomponente – man kann dann beliebig Klassenmethoden aufrufen. Aber wie macht man das bei einer Funktionskomponente? Funktionskomponenten darf man gar keine `ref`-Property übergeben. Außerdem haben sie keine Klassenmethoden.

Hier spielen `forwardRef` (um der Funktionskomponente eine Ref zu übergeben) und `useImperativeHandle` (um auf dieser Ref der Elternkomponente dann Methoden zur Verfügung zu stellen) zusammen. Das sieht so aus:

```
function MyButtonInner(props, ref) {

 const btnRef = useRef();

 useImperativeHandle(ref, () => ({

   focus: () => {

     btnRef.current.focus();

   }

 }), [ /* eventuelle Deps hier*/ ]);

 return <button ref={btnRef}>focus me
 programmatically!</button>;

}

export const MyButton =
forwardRef(MyButtonInner);
```

Hier verwenden wir also `forwardRef`, um die `ref` als zweiten Parameter an unsere Funktionskomponente weiter zu geben. Und dann `useImperativeHandle`, um `ref.current` einen Wert zuzuweisen – in diesem Fall mit einem leeren Dependency-Array.

Aber warum brauchen wir `useImperative-Handle` hier überhaupt?

Klar, wir könnten auch einfach in der Render-Phase direkt `ref.current` manuell setzen. Aber das wäre höchst problematisch. Was ist, wenn eine Komponente weiter unten im Baum einen Fehler wirft und der komplette Render bis zur nächsten Error Boundary weggeworfen wird? Dann würden in `ref.current` unter Umständen falsche Sachen stehen. Wir dürfen `ref.current` also frühestens in der Commit-Phase setzen. Der nächste Zeitpunkt, der uns hier zur Verfügung steht, wäre bei Einsatz von `useLayoutEffect`. Das ist aber unter Umständen zu spät, falls andere `useLayoutEffect`-Calls früher geschedult sind und unsere Referenz benötigen. Deshalb nutzt man hier den eigenen Hook, nur für diesen Anwendungsfall: `useImperativeHandle`



## ACHTUNG

Imperativer Code in jeder Form arbeitet "um React herum" und sollte nach Möglichkeit nur sehr sparsam oder gar nicht eingesetzt werden! React ist vom Konzept her deklarativ.

# @phry

Lenz Weber arbeitet schon seit über 15 Jahren in den Bereichen Webentwicklung und DevOps. Seit 2016 ist er bei Mayflower dabei und wechselt hier allmählich vom Backend ins Frontend, wobei er sich neuen Herausforderungen stellt und mit allem experimentiert, was ihm unter die Finger kommt.

Wenn er sich nicht gerade mit Webentwicklung auseinandersetzt, ließt er sich in Security-Themen ein (insbesondere im Kontext von Passwörtern), experimentiert mit NixOS oder setzt sich mit diversen Open-Source-Projekten auf GitHub auseinander.

Wenn Du in Kontakt treten magst, ist ein Tweet an @phry vermutlich die beste Idee.

Lenz Weber (@phry) has been working in Web Development and DevOps for 15 years. Since 2016, he's working at Mayflower, shifting his focus from Backend to Frontend, always challenging himself and experimenting with the tools at his disposal.

If he's not doing Web Development, he reads up on Security (special focus on passwords), experiments with NixOS or interacts with random Open Source Projects on GitHub.

If you want to get into contact, a tweet is the best idea.

## WEITERFÜHRENDE INFORMATIONEN

**Video: React Hooks @ React Conf 2018**
https://www.youtube.com/watch?v=dpw9EHDh2bM

**Introducing Hooks**
https://reactjs.org/docs/hooks-intro.html

**Guide to useEffect**
https://overreacted.io/a-complete-guide-to-useeffect/

**useMemo and useCallback**
https://kentcdodds.com/blog/usememo-and-usecallback/

**Fetch Data with React Hooks**
https://www.robinwieruch.de/react-hooks-fetch-data

**Avoid React Hooks Pitfalls**
https://kentcdodds.com/blog/react-hooks-pitfalls



## ABOUT

# LENZ WEBER

## WEB DEVELOPMENT & DEVOPS

# BLOG: DAS REDUX-JAHR 2019 IN RETROSPEKTIVE

Seit der Einführung von Hooks mit React 16.8 haben sich eine ganze Menge Bibliotheken herausgebildet, die Teilaufgaben dessen, was wir früher noch sehr umständlich mit Redux gemacht haben, jetzt elegant mit wenigen Zeilen erledigen. Auch React Context ist nun leicht mit Hooks nutzbar. Das hat dazu geführt, dass viele Entwickler zumindest in Betracht ziehen, ihre globale State-Management-Lösung über den Haufen werfen. Auch komplett neue State-Management-Bibliotheken, wie z.B. Zustand, wurden veröffentlicht. Und mit XState ist das Konzept der State Machines wieder in den Blick der Entwickler gerückt.

Kurzum: Es wurde eine ganze Menge Staub aufgewirbelt und es gibt neue Konkurrenz für den Platzhirsch Redux und seinen alten Gegenspieler MobX. Geschadet hat es nicht, ganz im Gegenteil: Das „offizielle" Redux-Ökosystem scheint nach einigen Jahren des gefühlten Winterschlafs aufgewacht zu sein. Es gab zwar immer wieder Änderungen an den Interna der Implementierung, nicht aber an den APIs an sich.

Tatsächlich hat sich eine Menge getan. Nicht nur an den APIs, sondern auch bei der Kommunikation der Entwickler.

## REACT HOOKS

Während es schon zum Erscheinungsdatum von React 16.8 erste inoffizielle Bibliotheken (*redux-hooks, redux-react-hooks*, um nur zwei zu nennen) gab, um Hooks mit Redux zu nutzen, war es um die offizielle react-redux-Bibliothek erst einmal ziemlich lange still. Das lag nicht etwa daran, dass dem Thema keine Bedeutung zugewiesen wurde. Viel mehr wurden in einer mehrere Monate andauernden, sehr lesenswerten Diskussion verschiedenste APIs und Implementierungsdetails diskutiert. Redux 7.1.0 wurde schließlich die erste Version mit offiziellem Hooks-Support.

## REDUX TOOLKIT

Über Jahre hinweg war das Motto von Redux sinngemäß: „Wir stellen euch die Tools zur Verfügung, das Ökosystem regelt sich von alleine". Das hat zu einem Wildwuchs von mehreren hundert Bibliotheken geführt, die alle irgendwie versuchen, Redux einfacher nutzbar zu machen. Und ein Großteil dieser Bibliotheken mag es auf die eine oder andere Art und Weise geschafft haben. Aber egal wie oft Problem X schon gelöst wurde: Man konnte sicher davon ausgehen, dass die nächste neue Bibliothek wieder einen neuen eigenen Ansatz fuhr.

Infolgedessen gab es bis vor kurzem keinen „offiziellen" Weg, Redux zu verwenden. Ein Entwickler, der sich in seiner eigenen Redux-Codebasis bestens auskennt, kann in der Codebasis eines anderen Teams vor völlig neue Konzepte gestellt sein.

Um diese Situation in den Griff zu bekommen, hat Mark Erikson, der aktuelle Maintainer von Redux, bereits im März 2018 die Entwicklung von Redux-Toolkit (damals noch Redux-Starter-Kit) begonnen. Anfang 2019 wurde das Projekt nach TypeScript portiert und Ende Oktober 2019 in Version 1.0 released. Gegenwärtig sind wir bei Version 1.2.5.

Wer die ganze Geschichte nachlesen möchte, kann das im Blog von Mark Erikson tun. Kurz zusammengefasst stellt RTK viele Helfer zur Verfügung, die Boilerplate-Code massiv reduzieren und (für TypeScript-Entwickler) die Typsicherheit massiv verbessern. Zudem wird mit dem Konzept des *slice* auch das über die Zeit in der Community aufgekommene „ducks"-Pattern offiziell empfohlen.

Allgemein ist Redux mit Redux Toolkit als offizieller Bibliothek deutlich mehr opinionated als früher – man könnte sagen, Redux Toolkit ist für Redux das, was MobX StateTree für MobX ist. Zum Thema „opinionated" aber gleich noch mehr.

## TYPESCRIPT

Guter TypeScript-Support spielte im letzten Jahr für Redux allgemein eine deutlich höhere Rolle als bisher. Das mag auch daran liegen, dass Mark Erikson inzwischen voll in TypeScript eingestiegen ist. Seine Erfahrungen hat er in einem Blogartikel zusammengefasst.

Wie schon erwähnt, wurde das Redux Toolkit schon recht bald in seiner Entwicklung nach TypeScript übersetzt. Einen Großteil der aktuellen Typendefinitionen hat übrigens der Autor dieser Zeilen verbrochen; ich bin seit Mitte 2019 Contributer und seit kurz nach dem 1.0er-Release offiziell mit an Bord.

Aber das war noch nicht das Ende. Nach einer längeren Diskussion auf GitHub und einer überraschend eindeutigen Twitter-Umfrage fiel die Entscheidung, auch den Redux-Core, eine Bibliothek, die man eigentlich als „fertig" bezeichnen konnte, nach TypeScript zu übersetzen.

Die kommende Version 5.0 wird also in TypeScript geschrieben sein. Darüber hinaus befindet sich die Dokumentation in einem großen Overhaul. Aktuell wurde sie für alle drei Bibliotheken um eine spezifische Seite zu TypeScript ergänzt:

- https://redux.js.org/recipes/usage-with-typescript
- https://react-redux.js.org/using-react-redux/static-typing
- https://redux-toolkit.js.org/usage/usage-with-typescript

Diese Dokumentation sollte für fast jeden TypeScript-Nutzer noch den einen oder anderen Aha-Effekt bereit halten. Insbesondere der neue *ConnectedProps*-Typ von React-Redux sollte Typdefinitionen für Klassenkomponenten mit *connect* deutlich angenehmer machen.

## STYLEGUIDE

Als letzte große Neuerung gibt es jetzt einen offiziellen StyleGuide, der sich selbst als eine Sammlung von empfohlenen Patterns und Best Practices versteht und erfrischend „opinionated" daher kommt. Und der hat es in sich, räumt er doch sogar mit vielen Empfehlungen auf, die sich zwar früher mal in den Docs wiedergefunden haben, aber aufgrund von Performanceoptimierungen oder jahrelanger Praxiserfahrung einfach nicht mehr zeitgemäß sind.

Die Lektüre lohnt sich:
https://redux.js.org/style-guide/style-guide

**MEHR TEXTE WIE DIESEN**

auf: **blog.mayflower.de**

# VISION

Wir führen Unternehmen in die Zukunft –
mit modernen Technologien und
agiler Organisation.

# MAYFLOWER

-

Unternehmen in die Zukunft zu bringen, bedeutet für uns, den richtigen Mix aus modernen Technologien und agiler Organisation zu verwenden, um gemeinsam mit unseren Kunden ihre Ziele zu erreichen.

Uns zeichnet ganzheitliches Denken aus, dabei bewahren wir unseren Kern als Tech-Company. Wir arbeiten ganzheitlich auf allen Ebenen, um werthaltige Software von hoher Qualität entwickeln zu können.

Wir verstehen uns als Lotsen & Bergführer. Wir gehen aktiv voran und helfen dem Kunden an unterschiedlichen Stellen, diese Zukunft zu erreichen.

Wir sind pragmatisch, schnell, offen und zuverlässig. Unsere Kunden schätzen dies. Deswegen vertrauen sie uns wichtige Vorhaben an.

Wir sehen immer wieder, dass es nicht ausreicht, einfach nur gute Software zu bauen. Ganz im Gegenteil, gute Software entsteht oft nur dann, wenn alle am Softwareprodukt beteiligten Menschen gut zusammenwirken. Wenn wir feststellen, dass irgendwo ein Rädchen nicht gut läuft, dann wirken wir darauf hin, dass Verbesserungen stattfinden. Ob bei einer Beratung des Kunden-Product-Owners oder eine in Hinblick auf die Infrastruktur: Wir schauen uns alle Ebenen der Organisation an, die auf dem Weg zu werthaltiger Software für den Nutzer involviert sind, und versuchen mit dem Kunden zusammen Verbesserungen zu erreichen. Wir sind diejenigen, die mit "skin in the game" an diesen Verbesserungen mitarbeiten.

---

## 3x HQ

### STANDORTE

Berlin

Würzburg

München

## >80%

### DEVELOPER (X/W/M)

Agile Web Development

Beratung & Konzeption

Trainings & Workshops

DevOps & Infrastructure

## <3

### KULTUR

10% Slacktime

Firmeninternes Barcamp

Faible für Open-Source

Speaker, Maintainer & Core-Developer

Lokalkolorit von Münchner Wies'n bis Berliner Currywurst

Talente, schräge Vögel und souveräne Ansprechpartner*Innen

MAYFLOWER

**BERLIN**
Liegnitzer Straße 15
10999 Berlin

**WÜRZBURG**
Landsteinerstraße 4
97074 Würzburg

**MÜNCHEN**
Landsberger Straße 314
80687 München

**WEB**
mayflower.de

**E-MAIL-ADRESSE**
kontakt@mayflower. de