

NILS HARTMANN

Fortgeschrittene

React

Pattern

Slides: <https://react.schule/enterjs-2020-react>

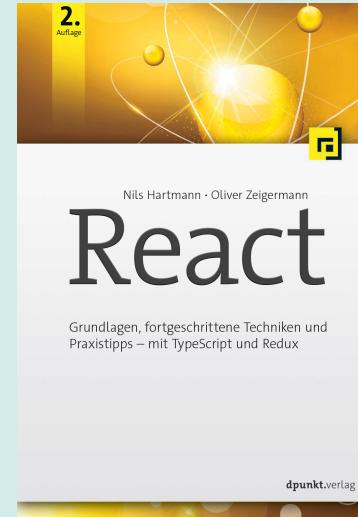
# NILS HARTMANN

nils@nilshartmann.net

**Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg**

Java  
JavaScript, TypeScript  
React  
GraphQL

**Trainings & Workshops**



<https://reactbuch.de>

**HTTPS://NILSHARTMANN.NET**

# Custom Hooks

ZIEL: (INFRASTRUKTUR-)CODE WIEDERVERWENDEN

## Custom Hooks

With Hooks, you can extract stateful logic from a component so it can be tested independently and reused. **Hooks allow you to reuse stateful logic without changing your component hierarchy.** This makes it easy to share Hooks among many components or with the community.

<https://reactjs.org/docs/hooks-intro.html>

## CUSTOM HOOKS

### Beispiel: Hook zum Laden von Daten

- Name, Signatur und Rückgabe eines Hooks kann frei gewählt werden

```
function useApi(url) {  
  const [data, setData] = useState(null);  
  
  useEffect(() => {  
    const fetchData = async () => {  
      const response = await fetch(url);  
      const data = await response.json();  
      setData(data);  
    };  
    fetchData();  
  }, [url]);  
  
  return data;  
}
```

## CUSTOM HOOKS

### Beispiel: Hook zum Laden von Daten

- Custom Hook darf andere Hooks verwenden (z.B. setState)

```
function useApi(url) {  
  const [ data, setData ] = React.useState(null);  
  
  return data;  
}
```

### Beispiel: Hook zum Laden von Daten

- ...oder useEffect

```
function useApi(url) {  
  const [ data, setData ] = React.useState(null);  
  
  React.useEffect( () => {  
    fetch(url) // vereinfacht  
      .then(res => res.json())  
      .then(data => setData(data))  
  }, [url]  
);  
  
  return data;  
}
```

## CUSTOM HOOKS

### Verwendung

...aber einfacher zu verwenden...

```
function ChatPage() {  
  const { data } = useApi("http://api/posts");  
  
  if (!data) {  
    return <LoadingIndicator />  
  }  
  
  return <Chat messages={data} />  
}
```

## Testen von Hooks

<https://react-hooks-testing-library.com/>

```
import { renderHook } from '@testing-library/react-hooks';

import useApi from '../useApi';

it('should work', async () => {
  const { result, waitForNextUpdate } = renderHook( () => useApi("...") );

  expect(result.data).toBe(null);

  // für asynchrone Hooks:
  await waitForNextUpdate();

  expect(result.data).toBe(/* ... */);
});
```

**Komplexer**

**Zustand**

**ZIEL: ZUSTAND KONSISTENT HALTEN**

# KOMPLEXER ZUSTAND

## Erinnerung: useApi-Hook

```
function useApi(url) {  
  const [ data, setData ] = React.useState(null);  
  
  React.useEffect( () => {  
  
    fetch(url) // vereinfacht  
      .then(data => {  
        setData(data);  
      })  
    }, [url]);  
  
  return data;  
}
```

# KOMPLEXER ZUSTAND

## Der `useApi`-Hook etwas realistischer

Ein zweiter Zustand, für den Request Status

```
function useApi(url) {  
  const [ loading, setLoading ] = React.useState(false);  
  const [ data, setData ] = React.useState(null);  
  
  React.useEffect( () => {  
    setLoading(true);  
  
    fetch(url) // vereinfacht  
      .then(data => {  
        setLoading(false);  
        setData(data);  
      })  
  }, [url]);  
  
  return { loading, data };  
}
```

# KOMPLEXER ZUSTAND

## Der `useApi`-Hook etwas realistischer

Ein zweiter Zustand, für den Request Status

```
function useApi(url) {  
  const [ loading, setLoading ] = React.useState(false);  
  const [ data, setData ] = React.useState(null);  
  
  React.useEffect( () => {  
    setLoading(true); ← Fehleranfällig!  
    fetch(url) // vereinfacht  
      .then(data => {  
        setLoading(false); ← Kein setData, ... ist das gewollt?  
        setData(data);  
      })  
  }, [url]);  
  
  return { loading, data };  
}
```

Was passiert, wenn wir vergessen würden, loading zurück zusetzen?

Was passiert im Fehlerfall?

# KOMPLEXER ZUSTAND

## Der `useApi`-Hook etwas realistischer

...und noch ein Zustand: für die Fehler 🤦‍♂️ 🤦‍♂️

```
function useApi(url) {  
  const [ loading, setLoading ] = React.useState(false);  
  const [ error, setError ] = React.useState(null);  
  const [ data, setData ] = React.useState(null);  
  
  React.useEffect( () => {  
    setLoading(true);  
    setError(null); ← Kein setData, ... ist das gewollt?  
    fetch(url) // vereinfacht  
      .then(data => {  
        setLoading(false); ← Was passiert, wenn wir error nicht  
        setData(data);     zurücksetzen?  
      }).catch(e => setError(e)) ← Was passiert, wenn wir  
    }, [url]);           ← vergessen, loading oder error zurück  
  ...                  ← zusetzen?  
}
```

**Noch komplexer: Fehlerzustand!**

# KOMPLEXER ZUSTAND

## Komplexer Zustand

```
function useApi(url) {  
  const [ loading, setLoading ] = React.useState(false);  
  const [ error, setError ] = React.useState(null);  
  const [ data, setData ] = React.useState(null);  
  
  React.useEffect( () => {  
    setLoading(true);  
    setError(null);  
    setData(null);  
    fetch(url) // vereinfacht  
      .then(data => {  
        setLoading(false);  
        setData(data);  
      }).catch(e => setError(e))  
  }, [url]);  
  ...  
}
```



👉 Diese "Teilzustände" sind nicht unabhängig!

# KOMPLEXER ZUSTAND

## Komplexer Zustand

Objekte bei "komplexem" Zustand

```
function useApi(url) {  
  const [ apiState, setApiState ] =  
    React.useState({ loading: false, data: null, error: null });  
  
  React.useEffect( () => {  
    setApiState({loading: true});  
  
    fetch(url) // vereinfacht  
      .then(res => setApiState({data: res}))  
      .catch(err => setApiState({error: err}));  
  }, [url]);  
  
  return apiState;  
}
```

Ein "logischer" Zustand

## KOMPLEXER STATE

### Einfacher State oder komplexer State?

#### Empfehlung:

- **Einfachen State** für unabhängige Werte verwenden (z.B. Felder im Eingabefeld)
- **Komplexen State** für Werte, die in der Regel gemeinsam geändert werden und bei denen keine inkonsistenten Zustände entstehen sollen

### Einfacher State oder komplexer State?

#### Problem:

- **Je komplexer der Zustand, desto komplexer dessen Verwaltung**
  - Soll der Zustand in der Komponente verbleiben?
  - Was ist mit Testen?
  - Was ist mit Wiederverwendbarkeit (außerhalb von React z.B.)

# ARBEITEN MIT KOMPLEXEM ZUSTAND

## useReducer Hook

Schritt 1: Reducer-Funktion (state, action) => newState

# ARBEITEN MIT KOMPLEXEM ZUSTAND

## useReducer Hook

Schritt 1: Reducer-Funktion (state, action) => newState

Actions sind einfache JavaScript-Objekte

Beispiel: Lebenszyklus eines API Requests

```
const action = {  
  type: "LOAD_FINISHED", ----- Type  
  response: "... " ----- Payload  
}
```

```
const action = {  
  type: "LOAD_FAILED",  
  error: "..."  
}
```

```
const action = {  
  type: "FETCH_START"  
}
```

# ARBEITEN MIT KOMPLEXEM ZUSTAND

## useReducer Hook

Schritt 1: Reducer-Funktion (state, action) => newState

```
function apiReducer(oldState, action) {  
  switch (action.type) {  
    case "FETCH_START":  
  
  }  
}
```

# ARBEITEN MIT KOMPLEXEM ZUSTAND

## useReducer Hook

Schritt 1: Reducer-Funktion (state, action) => newState

```
function apiReducer(oldState, action) {  
  switch (action.type) {  
    case "FETCH_START":  
      return { ...oldState, loading: true };  
  
  }  
}  
}
```

# ARBEITEN MIT KOMPLEXEM ZUSTAND

## useReducer Hook

Schritt 1: Reducer-Funktion (state, action) => newState

```
function apiReducer(oldState, action) {  
  switch (action.type) {  
    case "FETCH_START":  
      return { ...oldState, loading: true, error: null };  
    case "LOAD_FAILED":  
      return { loading: false, error: action.error };  
  
    case "LOAD_FINISHED":  
      return { data: action.response };  
  
    default:  
      return throw new Error("Invalid action!");  
  }  
}
```

# ARBEITEN MIT KOMPLEXEM ZUSTAND

## useReducer Hook

### Schritt 2: Verwenden

```
function apiReducer() { ... }

function useApi(url) {
  const [state, dispatch] = React.useReducer(apiReducer);

  React.useEffect( () => {
    dispatch({ type: "FETCH_START" });

    fetch(...)
      .then(response => dispatch({type: "LOAD_FINISHED", response })
  }, []);

  return state;
}
```

## ARBEITEN MIT KOMPLEXEM ZUSTAND

### **useReducer:** Konsequenzen

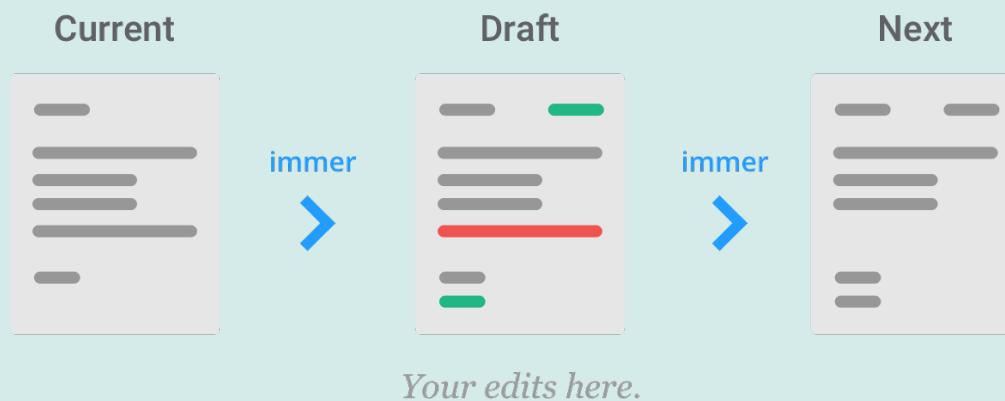
- Reducer Standard-JavaScript-Funktion, d.h. gut test- und wiederverwendbar, nicht React-spezifisch
- Komplette Logik zur Behandlung des Zustandes an einer zentralen Stelle
- Bei späterer Migration nach Redux können sie weiterverwendet werden
- dispatch von Actions Code-intensiv
- Arbeiten mit immutable State anstrengend

# ARBEITEN MIT KOMPLEXEM ZUSTAND

**immer** erlaubt mutable Code zu schreiben, der "normal" aussieht

<https://immerjs.github.io/immer/docs/introduction>

*Immer (German for: always) is a tiny package that allows you to work with immutable state in a more convenient way. It is based on the copy-on-write mechanism.*



## USERREDUCER HOOK

### Beispiel: immer

Funktioniert überall, wo mit Objekten gearbeitet wird (State, Reducer, ...)

```
// ES6: Anpassen eines Objektes in einer Liste an einem Objekt...
function handleMailChange(contactId, newMail) {
  setUser({
    ...user,
    contacts: user.contacts.map(c =>
      c.id === contactId ? { ...c, mail: newMail } : c
    )
  });
}
```

# USERREDUCER HOOK

## Beispiel: immer

Funktioniert überall, wo mit Objekten gearbeitet wird (State, Reducer, ...)

```
import produce from "immer";

// ES6: Anpassen eines Objektes in einer Liste an einem Objekt...
function handleMailChange(contactId, newMail) {
  setUser({
    ...user,
    contacts: user.contacts.map(c =>
      c.id === contactId ? { ...c, mail: newMail } : c
    )
  });
}

// immer
function handleMailChange(contactId, newMail) {
  setUser(produce(draft => {
    const ix = draft.contacts.findIndex(c => c.id === contactId);
    draft.contacts[ix].type = newType;
  )));
}
```

# Globale Daten

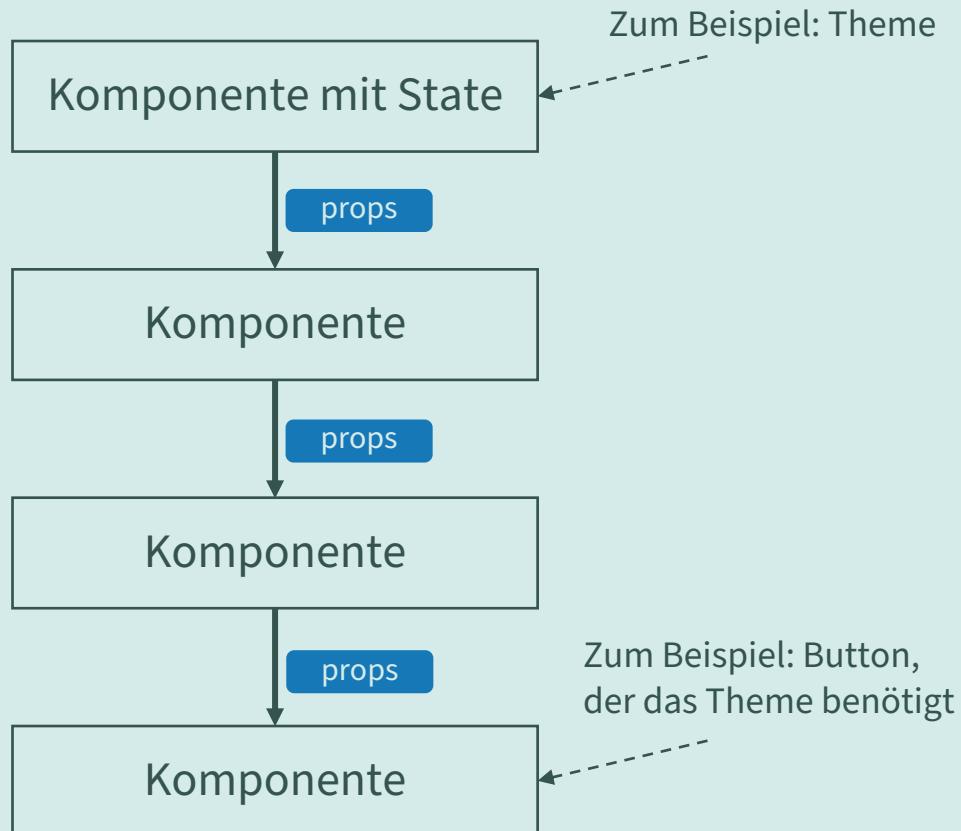
ZIEL: ZUSAMMENSPIEL VON KOMPONENTEN

## GLOBALE DATEN

### Beispiele für globale Daten ("Klassiker")

- Angemeldeter Benutzer mit seinen Rollen
- Aktuelles Theme
- Geladene Daten (Cache)

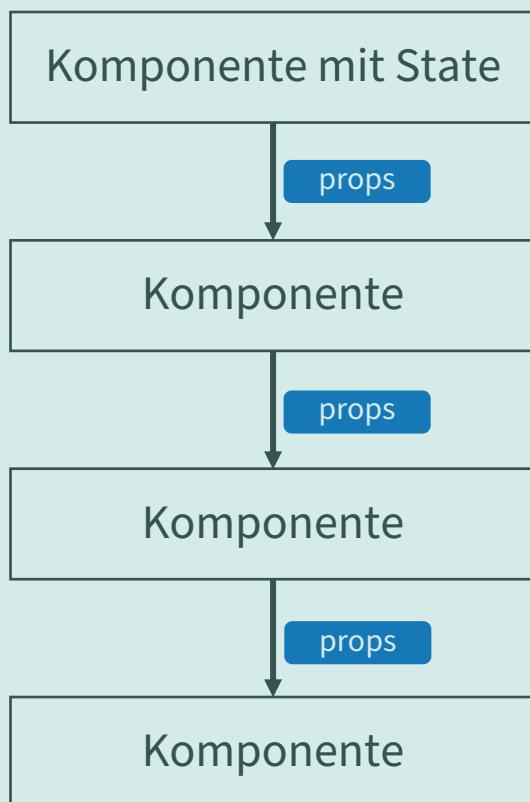
## Globale Daten: Durchreichen von Properties ("Klassiker")



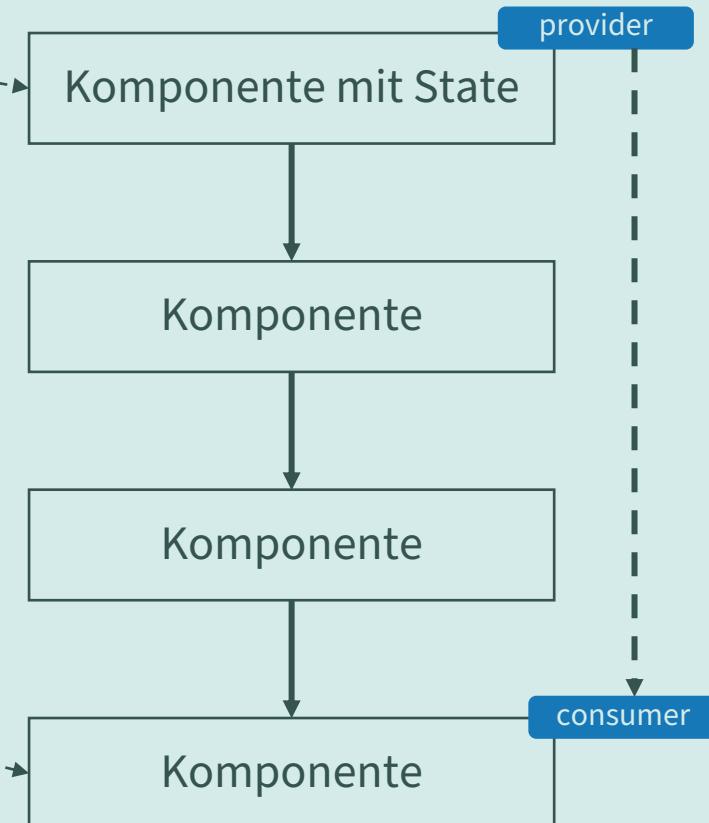
Klassisch: State und Callbacks werden per Properties durchgereicht

# GLOBALE DATEN

**React Context:** Stellt Werte innerhalb einer Komponentenhierarchie zur Verfügung



Zum Beispiel: Theme



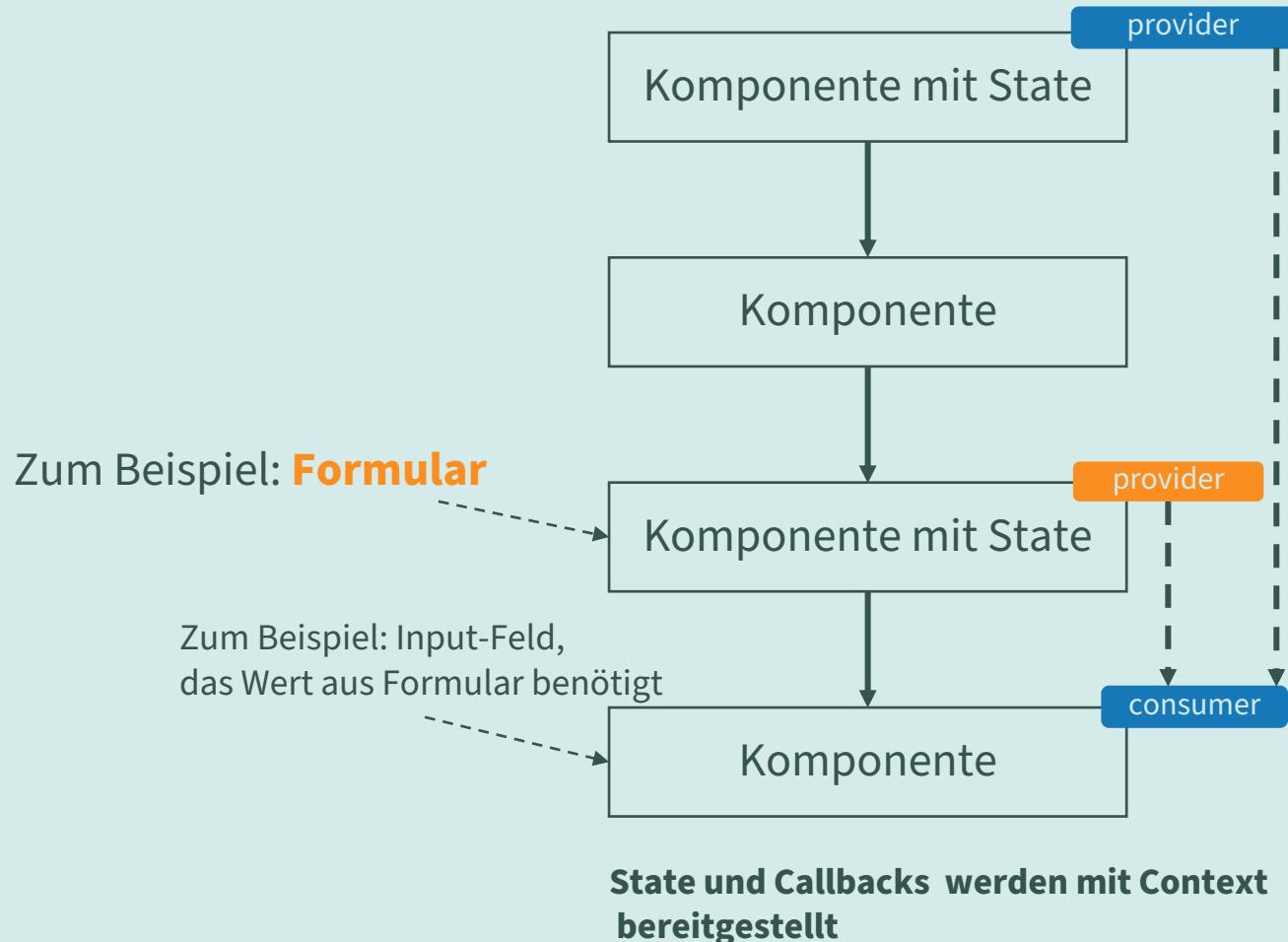
State und Callbacks werden per Properties durchgereicht

Zum Beispiel: Button,  
der das Theme benötigt

State und Callbacks werden mit Context  
bereitgestellt

# GLOBALE DATEN

**React Context:** Muss nicht "ganz" global sein, auch mehrere Contexte sind möglich



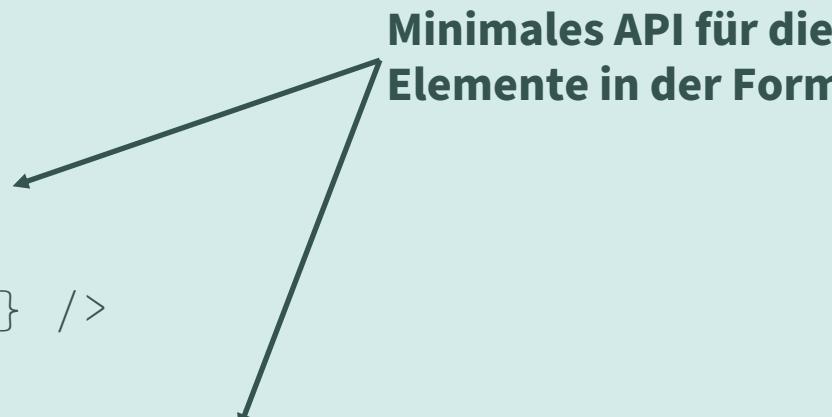
## Beispiel: Form-Komponente

```
function LoginForm({onLogin}) {  
  return (<Form>←  
    </Form>);  
}
```

**Hält Zustand aller Felder,  
Validierungen, Feedback,  
etc.**

## Beispiel: Form-Komponente

```
function LoginForm({onLogin}) {  
  
  return (<Form>  
    <TextField name={user} />  
  
    <TextField name={password} />  
  
    <SubmitButton  
      onSubmit={form => onLogin(form.user, form.password)}>  
      Login  
    </SubmitButton>  
  </Form>);  
}
```



Minimales API für die Elemente in der Form

## Beispiel: Form-Komponente

```
function Form(props) {  
  const [formState, setFormState] = React.useState({....});  
}  
  
Zum Beispiel:  
für jedes Feld ein Eintrag im Objekt  
zu jedem Eintrag gehört Wert, Status, ...  
  
{  
  username: { value: ..., error: ... },  
  password: { value: ..., error: ... }  
}
```



## Beispiel: Form-Komponente

Die Form-Komponente ist ein Context-Provider

```
const FormContext = React.createContext();  
  
function Form(props) {  
  const [formState, setFormState] = React.useState({ ... });  
  
  const formContext = { ... };  
  }  
  
  Zum Beispiel: Objekt mit einem Eintrag  
  pro Form-Feld, das den aktuellen Wert  
  enthält, setter-Funktion, ...  

```

## Beispiel: Form-Komponente

Ein Consumer

```
function TextField({name}) {  
  const formState = React.useContext(FormContext);  
  
  return <div>  
    <input value={formState[name].value}  
          onChange={formState[name].onChange} />  
    <div>{formState[name].validationMessage}</div>  
  </div>;  
}
```

## Beispiel: Form-Komponente

Ein Consumer... und noch einer!

```
function TextField({name}) {  
  const formState = React.useContext(FormContext);  
  
  return <input value={formState[name].value}  
            onChange={formState[name].onChange} />  
}  
  
function TextArea({name}) {  
  const formState = React.useContext(FormContext);  
  
  // formState[name] verwenden  
}
```

## Beispiel: Form-Komponente

Ein Consumer... und noch eine eigene Komponente!

```
function TextField({name}) {  
  const formState = React.useContext(FormContext);  
  
  return <input value={formState[name].value}  
            onChange={formState[name].onChange} />  
}
```

```
function TextArea({name}) {  
  const formState = React.useContext(FormContext);  
  
  // formState[name] verwenden  
}
```

```
function MeineFormKomponente({name}) {  
  const formState = React.useContext(FormContext);  
  
  // formState[name] verwenden  
}
```

## Beispiel: Form-Komponente

Ein Consumer... und noch eine eigene Komponente!

```
function TextField({name}) {  
  const formState = React.useContext(FormContext);  
  
  return <input value={formState[name].value}  
            onChange={formState[name].onChange} />  
}
```

```
function TextArea({name}) {  
  const formState = React.useContext(FormContext);  
  
  // formState[name] verwenden  
}
```

```
function MeineFormKomponente({name}) {  
  const formState = React.useContext(FormContext);  
  
  // formState[name] verwenden  
}
```

- 👉 Redundanter Code
- 👉 Implementierungsdetail (Context)
- 👉 Zugriff nur auf ein Feld



## Beispiel: Form-Komponente

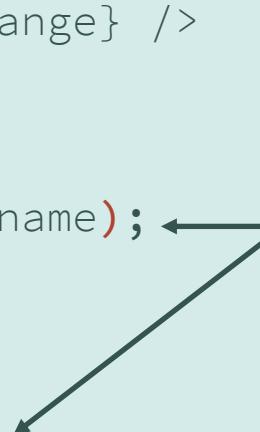
### Custom Hook für Context

```
function useFieldState(fieldName) {  
  const { formState } = React.useContext(FormContext);  
  
  return formState[fieldName];  
}
```

## Beispiel: Form-Komponente

### Custom Hook für Context

```
function useFieldState(fieldName) {  
  const { formState } = React.useContext(FormContext);  
  
  return formState[fieldName];  
}  
  
function InputField({name}) {  
  const { value, onChange, validationMessage } = useFieldState(name);  
  return <input value={value} onChange={onChange} />  
}  
  
function TextArea({name}) {  
  const { value, onChange } = useFieldState(name);  
  // ...  
}  
  
function MeineKomponente({name}) {  
  const { value, validationMessage } = useFieldState(name);  
}
```



👉 Versteckt den Kontext  
👉 "fachliche" API

🤓

# Globale Daten

JETZT WIRKLICH GLOBAL!

## REACT CONTEXT

**useReducer & useContext** – Redux Light?

Wir einen Context mit useReducer implementieren

Bereitgestellte Funktionen dispatchen dann Actions

# REACT CONTEXT

## useReducer & useContext – Redux Light?

Beispiel: Angemeldeter Benutzer

```
function authReducer(state, action) { ... }

function AuthContextProvider(props) {
  const [state, dispatch] = React.useReducer(authReducer);

  return (
    <AuthContext.Provider value={{
      user: state.user,
      login(u,p) { dispatch({type: "LOGIN", ...}) },
      logout() { dispatch({type: "LOGOUT", ...}) }
    }}>
      { children }
    </AuthContext.Provider>
  );
}
```

## REACT CONTEXT

**useReducer & useContext** kombiniert

- 👍 Globaler Zustand (kann ganz oben in der Hierarchie eingefügt werden)
- 👍 Strukturierung nach Geschmack möglich (Zustand pro Anwendungsteil möglich)
- 👍 Geschäftslogik jetzt aus den Komponenten raus (dank reducer-Funktion)
- 👍 Technik ist Transparent für Consumer (kein dispatch-Aufruf...)

## REACT CONTEXT

### useReducer & useContext kombiniert

- 👎 Performance bei häufigen Änderungen?
- 👎 Redux erlaubt feingranulare Auswahl, wann gerendert werden soll
- 👎 Actions, die von mehreren Reducern verarbeitet werden sollen?
- 👎 Tooling (Visualisierung der Änderungen am globalen Zustand, TT Debugging)
- 👎 Middlewares

Redux? 😊

## REDUX HOOKS API

### Beispiel: Redux mit Hooks API ("modern")

Keine connect-HOC mehr!

```
function UserProfile(props) {  
  const username = useSelector(state => state.auth.username) ;  
  const dispatch = useDispatch() ;  
  
  return <div>  
    <h1>{username}</h1>  
    <button onClick={() => dispatch(logout())}>Logout</button>  
  </div>  
}
```

Bessere Metapher: "auswählen"  
statt "mapStateToProps"

## REDUX HOOKS API

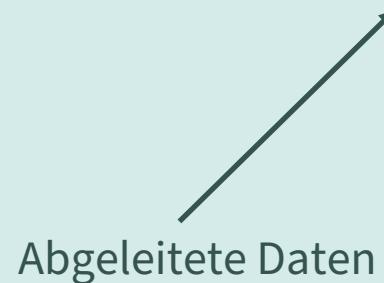
### Beispiel: Custom Hook

```
function useUsername() {  
  return useSelector(state => state.auth.username);  
}
```

# REDUX HOOKS API

## Beispiel: Custom Hook

```
function useUsername() {  
  return useSelector(state => state.auth.username);  
}  
  
function useIsLoggedIn() {  
  return useSelector(state = state.auth.username !== null);  
}
```



Abgeleitete Daten

## Beispiel: Custom Hook

```
function UserProfile(props) {  
  const username = useUsername();  
  
  return <div>  
    <h1>{username}</h1>  
  </div>  
}
```



Rendert nur neu, wenn sich der Username im globalen State verändert hat

# GLOBALER ZUSTAND

## Beispiel: Custom Hook

```
function UserProfile(props) {  
  const username = useUsername();  
  
  return <div>  
    <h1>{username}</h1>  
  </div>  
}
```

```
function LoginButton(props) {  
  const isLoggedIn = useIsLoggedIn();  
  
  return isLoggedIn ? <button>Logout</button>  
    : <button>Login</button>  
}
```

Rendert nur neu, wenn anderer Wert zurückgegeben wird

### Redux Toolkit <https://redux-toolkit.js.org/>

*The official, opinionated, batteries-included toolset for efficient Redux development*

- Vereinfachter Reducer und Actions-Code
- Spart viel Boilerplate-Code
- Out-of-the-box:
  - Wahnsitzig guter TypeScript-Support
  - Thunk Actions, Immer, Re-select

# Suspense

RENDERN UNTERBRECHEN

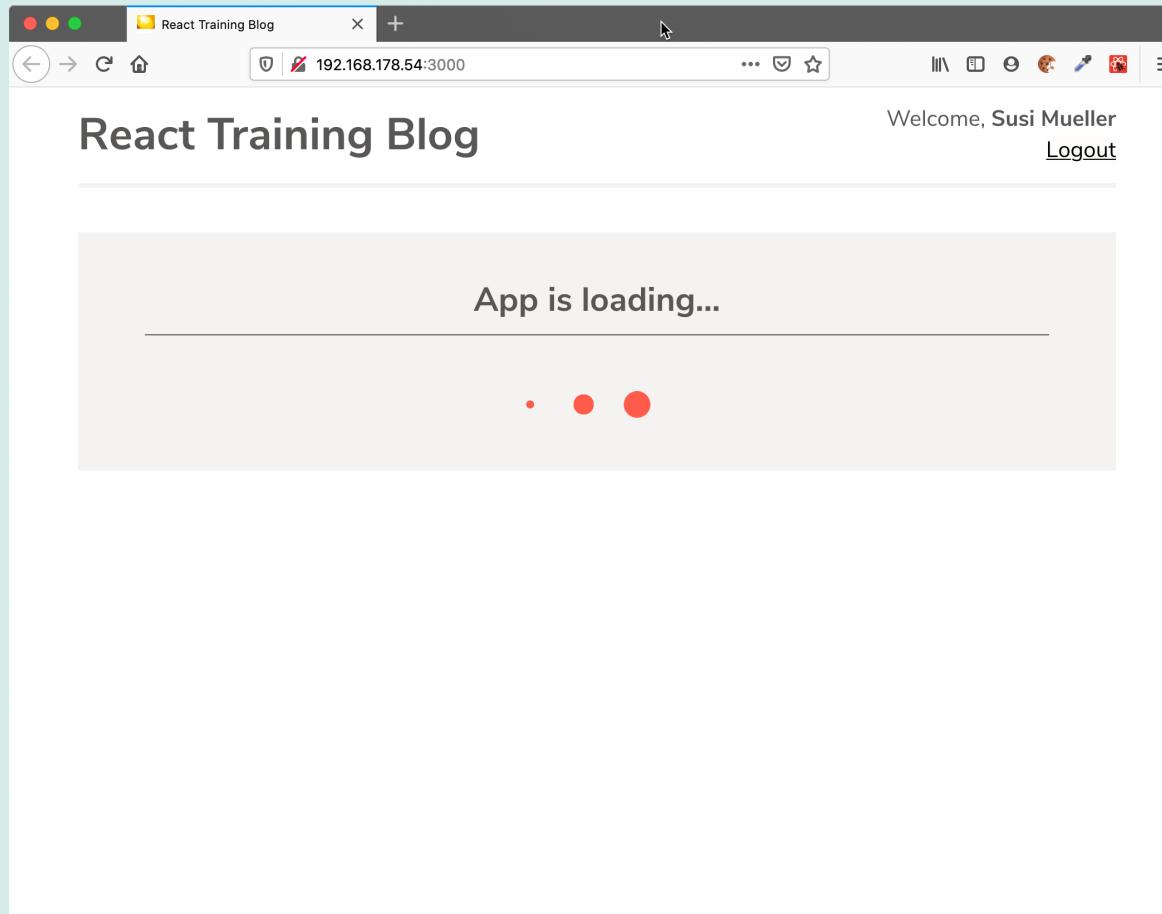
## SUSPENSE

**Suspense:** React kann das Rendern von Komponenten unterbrechen, während (asynchron) Daten geladen werden

- Funktioniert aktuell für **Code Splitting (=> Code nachladen)**
- **Künftig** auch zum **Laden von beliebigen Daten** (z.Zt. experimentell)

## DEMO: LAZY UND SUSPENSE

- Mit dynamic Imports wird Code erst bei Bedarf geladen



# SUSPENSE

## React.lazy: Code splitting with Suspense

```
const LoginPage = React.lazy(() => import("./login/LoginPage"));  
class App {  
  render() {  
    return <Switch>  
      <Route path="/login">  
        <LoginPage />  
      </Route>  
      // ...weitere Seiten...  
    </Switch>  
  }  
}
```

JS Dynamic Import

# SUSPENSE

**React.Suspense:** Zeigt "Fallback"-Komponente an

Bis Komponente geladen ist, muss Spinner o.ä. angezeigt werden

```
const LoginPage = React.lazy(() => import("./login/LoginPage"));

class App {
  render() {
    return <React.Suspense fallback={<LoadingIndicator />}>
      <Switch>
        <Route path="/login">
          <LoginPage />
        </Route>
        // ...weitere Seiten...
      </Switch>
    </React.Suspense>
  }
}
```

# Concurrent Mode & Suspense for Data Fetching

ZIEL: FLÜSSIGERES RENDERN

# Introducing Concurrent Mode (Experimental)

**Caution:**

This page describes **experimental features that are not yet available in a stable release**.

Don't rely on experimental builds of React in production apps. These features may change significantly and without a warning before they become a part of React.

This documentation is aimed at early adopters and people who are curious. If you're new to React, don't worry about these features — you don't need to learn them right now.

<https://reactjs.org/concurrent>

# Introducing Concurrent Mode (Experimental)

## Caution:

This page describes **experimental features that are not yet available in a stable release**.

Don't rely on experimental builds of React in production apps. These features may change significantly and without a warning before they become a part of React.

This documentation is aimed at early adopters and people who are curious. **If you're new to React, don't worry about these features** — you don't need to learn them right now.

<https://reactjs.org/concurrent>

SEPTEMBER 2020 😮

## Concurrent Mode 1

- Rendern ist eine "non-blocking" Operation
  - Es kann **immer** auf User-Interaktionen reagiert werden
- Updates können priorisiert werden

## Concurrent Mode 2

- Komponenten können u.a. vor-gerendert werden, ohne sofort sichtbar zu sein
  - Zum Beispiel beim **Laden von Code und Daten**
  - Verhindert überflüssige Warte- und Zwischen-Zustände
  - Komponenten müssen "etwas" haben, woher sie ihre Daten beziehen (gibt's aber noch nicht)
  - Erst wenn Komponente alle **gewünschten** Daten hat, wird sie angezeigt

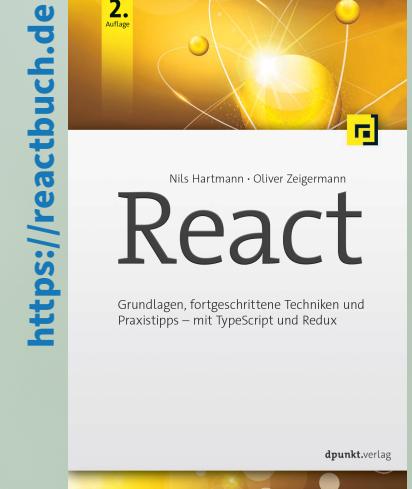
# CONCURRENT REACT

## Concurrent Mode: Aktueller Stand

- Experimentelle Version verfügbar, wird von FB produktiv eingesetzt
- Hat Veränderungen auf die Anwendungsarchitektur
  - Transitionen
  - Vorladen von Daten
- Ökosystem muss darauf vorbereitet sein
  - Router, React Query und Relay haben (experimentellen) Support
  - Konzepte/Bibliotheken zum Vorladen von Daten

**NILS HARTMANN**

<https://nilshartmann.net>



# vielen Dank!

Slides: <https://react.schule/enterjs-2020-react>

Fragen & Kontakt: [nils@nilshartmann.net](mailto:nils@nilshartmann.net)

**NILS@NILSHARTMANN.NET**