

NILS HARTMANN

Moderne

React

Pattern

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

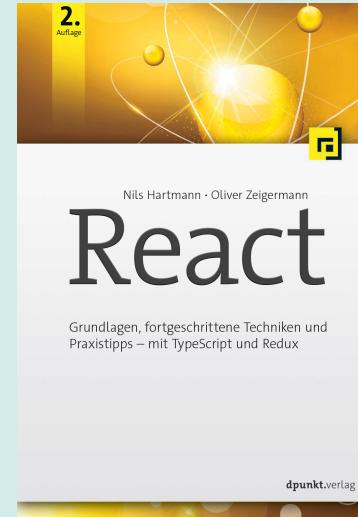
NILS HARTMANN

nils@nilshartmann.net

Freiberuflicher Entwickler, Architekt, Trainer aus Hamburg

Java
JavaScript, TypeScript
React
GraphQL

Trainings & Workshops



<https://reactbuch.de>

HTTPS://NILSHARTMANN.NET

HINTERGRUND: RENDER PROPERTIES

Pattern: Render Properties

HINTERGRUND: RENDER PROPERTIES

Render Properties

Klassiker: Das children-Property

```
function Page(props) {  
  return <main className="Container">  
    {props.children}  
  </main>  
}
```

HINTERGRUND: RENDER PROPERTIES

Render Properties

Klassiker: Das children-Property

```
function Page(props) {  
  return <main className="Container">  
    {props.children}  
  </main>  
}
```

```
function App() {  
  return <Page>  
    <h1>Hello World!</h1>  
  </Page>  
}
```

HINTERGRUND: RENDER PROPERTIES

Render Properties

Beispiel: Layout-Komponente, zwei Children benötigt!

```
function Layout(props) {  
  return <div className="Layout">  
    <div className="Left"> </div>  
    <div className="Right"> </div>  
  }  
}
```

HINTERGRUND: RENDER PROPERTIES

Render Properties: Ein Property, das JSX-Elemente entgegen nimmt

- Genau wie children-Property, nur selbst definiert

```
function Layout(props) {  
  return <div className="Layout">  
    <div className="Left">{props.left}</div>  
    <div className="Right">{props.right}</div>  
  }  
}
```

HINTERGRUND: RENDER PROPERTIES

Render Properties: Ein Property, das JSX-Elemente entgegen nimmt

- Genau wie children-Property, nur selbst definiert

```
function Layout(props) {  
  return <div className="Layout">  
    <div className="Left">{props.left}</div>  
    <div className="Right">{props.right}</div>  
  }  
}
```

```
function BlogListPage(props) {  
  return <Layout left={<BlogList header="Newest Posts"/>}  
                right={BlogListSidebar />}>  
}
```

HINTERGRUND: RENDER PROPERTIES

Render Properties: Informationen für Child-Komponenten

- Was machen wir, wenn die Oberkomponente Informationen für die Unterkomponenten hat

```
function BlogListPage(props) {  
  return <DataLoader url="...">  
    <BlogList posts={...}" />  
  </DataLoader>  
}
```

👉 Beispiel: DataLoader hat Informationen
für die Kind-Komponente (hier: BlogList)

RENDER PROPERTIES

Beispiel: Generische DataLoader-Komponente

- "Infrastruktur"-Komponente, die Daten laden implementiert

```
function DataLoader(props) {  
  const state = React.useState({ loading: true, data: null });  
  
  React.useEffect( () => {  
    // vereinfacht  
    fetch(this.props.url)  
      .then(data => setState({data, loading: false}));  
  }  
}  
}  
}
```

RENDER PROPERTIES

Beispiel: Generische DataLoader-Komponente

- "Infrastruktur"-Komponente, die Daten laden implementiert
- Besonderheit: Komp

```
function DataLoader(props) {  
  const state = React.useState({ loading: true, data: null });  
  
  React.useEffect( () => {  
    // vereinfacht  
    fetch(this.props.url)  
      .then(data => setState({data, loading: false}));  
  }  
  
  // Kind-Komponente rendern und Properties (loading, data)  
  // übergeben....  
  return .... ? // WIE? WAS?  
}
```

RENDER PROPERTIES

Render Property als Funktion

- Function-as-a-Child (statt statischer Komponente!)
- Callback-Funktion liefert dann die Komponente zurück

```
function DataLoader(props) {  
  const state = React.useState({ loading: true, data: null });  
  
  React.useEffect( () => {  
    // vereinfacht  
    fetch(this.props.url)  
      .then(data => setState({data, loading: false}));  
  }  
  
  // props.children ist eine Funktion!  
  return props.children(  
    loading: state.loading,  
    data: state.data  
  );  
}
```

RENDER PROPERTIES

Beispiel: DataLoader-Komponente - Verwendung

- Function-as-a-Child (statt statischer Komponente!)
- Callback-Funktion liefert dann die Komponente zurück

```
function BlogListPage(props) {  
  
  return <DataLoader url="http://api/posts">  
    {  
      ({ loading, data }) =>  
        loading ? <LoadingIndicator /> :  
          <BlogList posts={data}>  
    }  
  </DataLoader>  
}
```

RENDER PROPERTIES

Beispiel: Mehrere Funktionen als Kind-Elemente

RENDER PROPERTIES

Beispiel: Mehrere Funktionen als Kind-Elemente

```
function BlogListPage(props) {  
  
  return  
    <ApiConfiguration>  
      { config =>  
        <DataLoader url={config.url + "/posts"}>  
          { ({ loading, data }) =>  
            loading ? <LoadingIndicator /> :  
              <BlogList posts={data}>  
                {  
                  </DataLoader>  
                }  
              </ApiConfiguration>  
      }  
}
```

RENDER PROPERTIES

Beispiel: Mehrere Funktionen als Kind-Elemente

Lesbarkeit? 😬

Verständlichkeit (des Konzeptes)? 😳

```
function BlogListPage(props) {  
  
  return  
    <ApiConfiguration>  
      { config =>  
        <DataLoader url={config.url + "/posts"}>  
          { ({ loading, data }) =>  
            loading ? <LoadingIndicator /> :  
              <BlogList posts={data}>  
            }  
          </DataLoader>  
        }  
    </ApiConfiguration>  
}
```

Hooks als Alternative

HOOKS ALS ALTERNATIVE

Hooks API als Alternative

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>

HOOKS ALS ALTERNATIVE

Beispiel DataLoader: Alternative zum Render Property

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

```
function useApi(url) {  
  const [loading, setLoading] = useState(true);  
  const [data, setData] = useState(null);  
  
  useEffect(() => {  
    const controller = new AbortController();  
    const signal = controller.signal;  
  
    fetch(url, { signal })  
      .then(response => response.json())  
      .then(data => setData(data))  
      .catch(error => console.error(error));  
  
    return () => controller.abort();  
  }, [url]);  
  
  return { loading, data };  
}
```

HOOKS ALS ALTERNATIVE

Beispiel DataLoader: Alternative zum Render Property

- Es kann eigener State definiert werden

```
function useApi(url) {  
  const [ apiState, setApiState ] =  
    React.useState({ loading: false, data: null });  
  
  // ...  
  
  return apiState;  
}
```

HOOKS ALS ALTERNATIVE

Beispiel DataLoader: Alternative zum Render Property

Code ähnlich wie beim DataLoader...

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

HOOKS ALS ALTERNATIVE

Beispiel DataLoader: Alternative zum Render Property

...aber einfacher zu verwenden...

```
function BlogListPage() {  
  const { loading, data } = useApi("http://api/posts");  
  
  if (loading) {  
    return <LoadingIndicator />  
  }  
  
  return <BlogList posts={data} />  
}
```

HOOKS ALS ALTERNATIVE

Beispiel DataLoader: Alternative zum Render Property

...auch bei mehreren Hooks

```
function BlogListPage() {  
  const { config } = useConfiguration();  
  const { loading, data } = useApi(`${config.url}/posts`);  
  
  if (loading) {  
    return <LoadingIndicator />  
  }  
  
  return <BlogList posts={data} />  
}
```

CUSTOM HOOKS ZUR VERWALTUNG VON STATE

Komplexer Zustand

Noch ein Zustand 🤯

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

CUSTOM HOOKS ZUR VERWALTUNG VON STATE

Komplexer Zustand

...und noch ein 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);  
  
    fetch(url) // vereinfacht  
      .then(res => {  
        setLoading(false);  
        setData(res.json());  
      }).catch(e => setError(e))  
  }, [url]);  
  ...
```

Noch komplexer: Fehlerzustand!

Kein setData, ... ist das gewollt?
Was passiert, wenn wir error nicht
zurücksetzen?

Was passiert, wenn wir
vergessen, loading **oder** error zurück
zusetzen?

...oder hier?

CUSTOM HOOKS ZUR VERWALTUNG VON STATE

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(res => {  
        setLoading(false);  
        setData(res.json());  
      }).catch(e => setError(e))  
  }, [url]);  
  ...  
}
```

Wie häufig wird hier gerendert? 🤔

CUSTOM HOOKS ZUR VERWALTUNG VON STATE

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(res => {  
        setLoading(false);  
        setData(res.json());  
      }).catch(e => setError(e))  
  }, [url]);  
  ...  
}
```



👉 Diese "Teilzustände" sind nicht unabhängig!

CUSTOM HOOKS ZUR VERWALTUNG VON STATE

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

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

USERREDUCER HOOK

useReducer: Arbeiten mit komplexem Zustand

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

USERREDUCER HOOK

useReducer: Redux für Komponenten?

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"  
}
```

USERREDUCER HOOK

useReducer: Redux für Komponenten?

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

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

USERREDUCER HOOK

useReducer: Redux für Komponenten?

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

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

USERREDUCER HOOK

useReducer: Redux für Komponenten?

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!");  
  }  
}
```

USERREDUCER HOOK

useReducer: Redux für Komponenten?

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;
}
```

USERREDUCER HOOK

useReducer: Konsequenzen

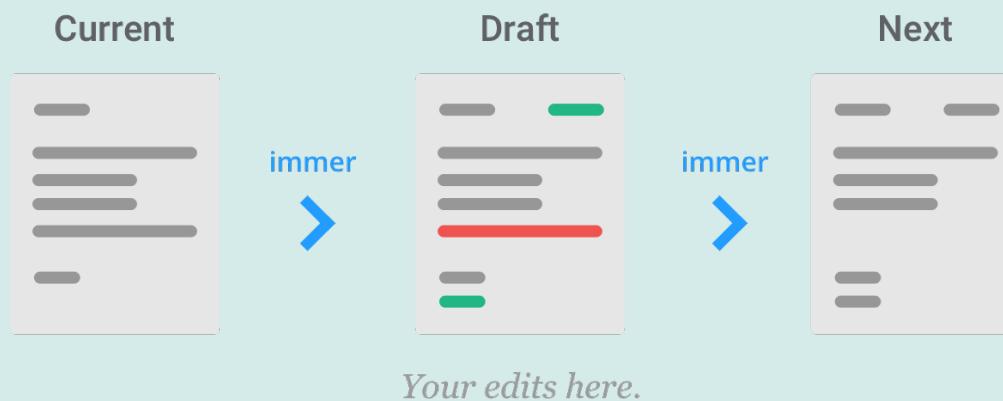
- Reducer Standard-JavaScript-Funktion, d.h. gut test- und wiederverwendbar, nicht React-spezifisch
- Komplette Logik zur Behandlung des Zustandes an einer zentralen Stelle
- Bei späterer Migration nach Redux können sie weiterverwendet werden
- dispatch von Actions Code-intensiv
- Arbeiten mit immutable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

USERREDUCER HOOK

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: reducer-Funktion mit immer

```
import produce from "immer";

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

state ist ein Proxy,
oldState bleibt
unverändert!

Sieht aus wie
"mutable Code",
keine zusätzliche
API notwendig

KOMMUNIKATION VON KOMPONENTEN

State über Komponentengrenzen

Idee: Wiederverwendbare Form-Komponente

```
function LoginForm(props) {  
  
    return (<Form> ← Hält Zustand,  
    Validierungen etc  
        </Form>);  
}
```

KOMMUNIKATION VON KOMPONENTEN

State über Komponentengrenzen

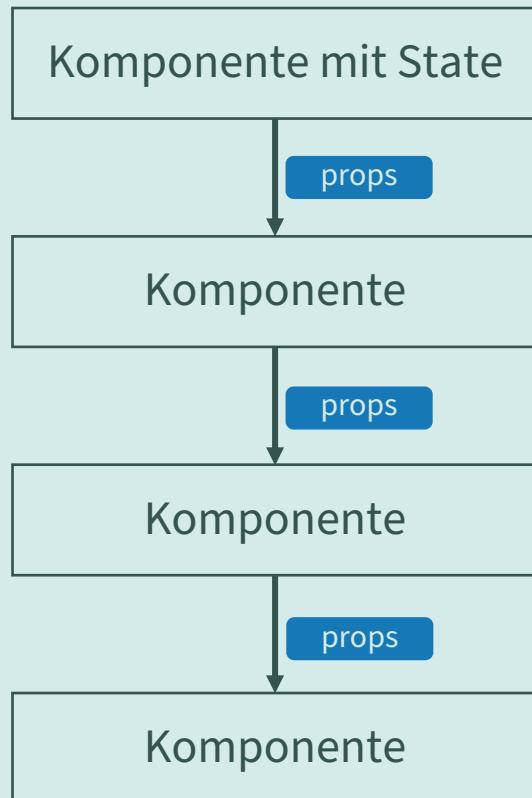
Idee: Wiederverwendbare Form-Komponente

```
function LoginForm(props) {  
  
  return (<Form>  
    <input value={???}  
      onChange={e => ???} />  
  
    <input value={???}  
      onChange={e => ???} />  
  
    <button onClick={  
      () => props.doLogin(???)}>Login</button>  
  </Form>);  
}
```

🤔 Wie kommen die Elemente an
ihre Werte und die Callbacks?

REACT CONTEXT

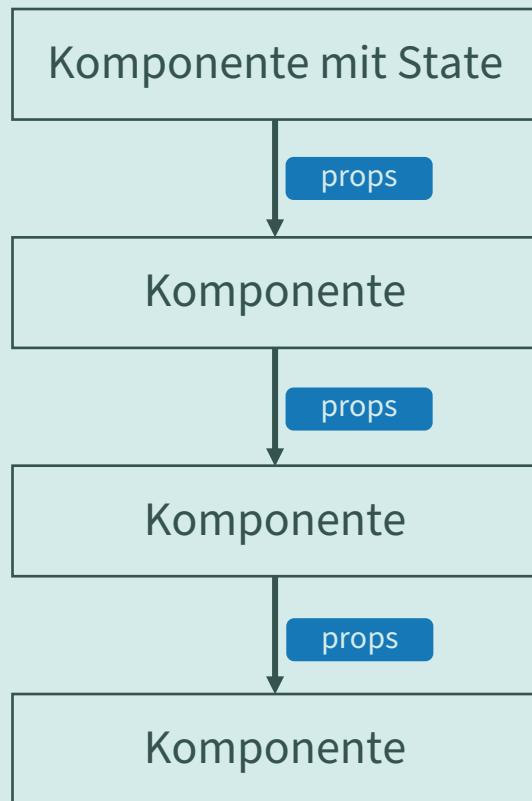
Hintergrund: React Context



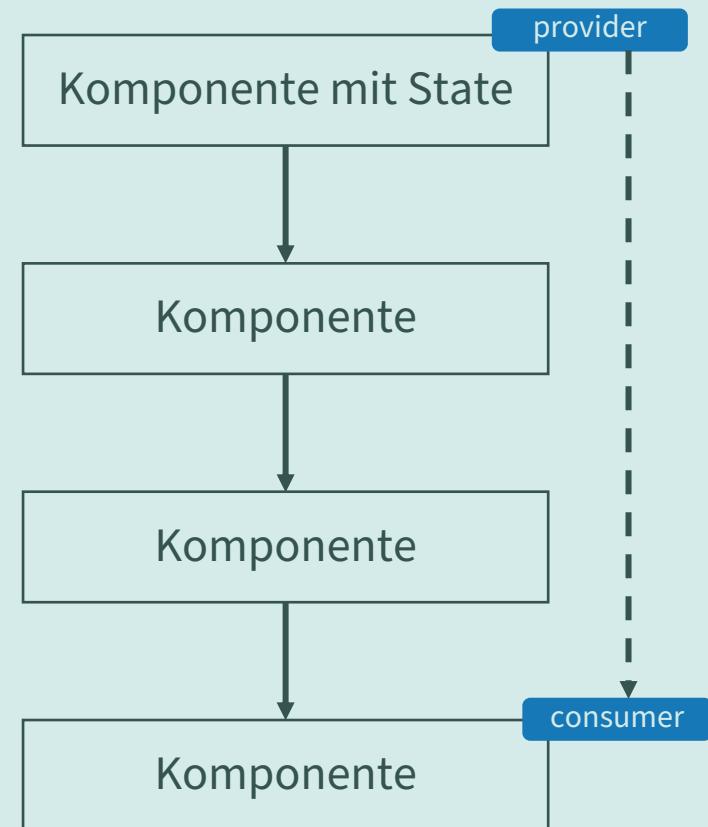
Klassisch: State und Callbacks werden per Properties durchgereicht

REACT CONTEXT

React Context: Stellt Werte innerhalb einer Komponentenhierarchie zur Verfügung



State und Callbacks werden per Properties durchgereicht



State und Callbacks werden mit Context bereitgestellt

KOMMUNIKATION VON KOMPONENTEN

State über Komponentengrenzen

Beispiel Context – Provider-Komponente

```
const FormContext = React.createContext();

function Form(props) {
  const [formState, setFormState] = React.useState({ ... });

  return <FormContext.Provider value={{ formState }}>
    {props.children}
  </form>;
}
```



React-Element als Children

KOMMUNIKATION VON KOMPONENTEN

State über Komponentengrenzen

Beispiel: Ein Consumer

```
function TextField({name}) {  
  const { formState } = React.useContext(FormContext);  
  
}  
}
```

KOMMUNIKATION VON KOMPONENTEN

State über Komponentengrenzen

Beispiel: Ein Consumer

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

KOMMUNIKATION VON KOMPONENTEN

State über Komponentengrenzen

Beispiel Context - Zugriff

```
function LoginForm(props) {  
  return <Form> ← Provider  
    <TextField ... />  
    <TextField ... /> ← Consumer  
    <Button ...>  
  </Form>;  
}  
  
Alle Komponenten unterhalb  
des Providers haben Zugriff auf  
das bereitgestellte Objekt
```

Globale Daten

GLOBALE DATEN

Beispiel: angemeldeter Benutzer

This post has been
published at
03.01.2020 by **you**
and already
received **27** likes

Welcome, **Nils Hartmann**
[Logout](#)

03.01.2020

Routing Solutions for React

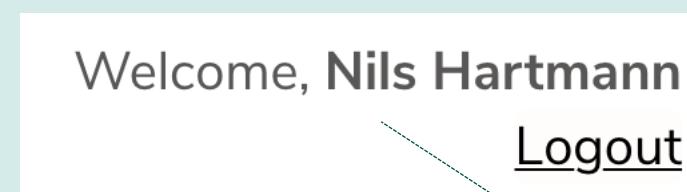
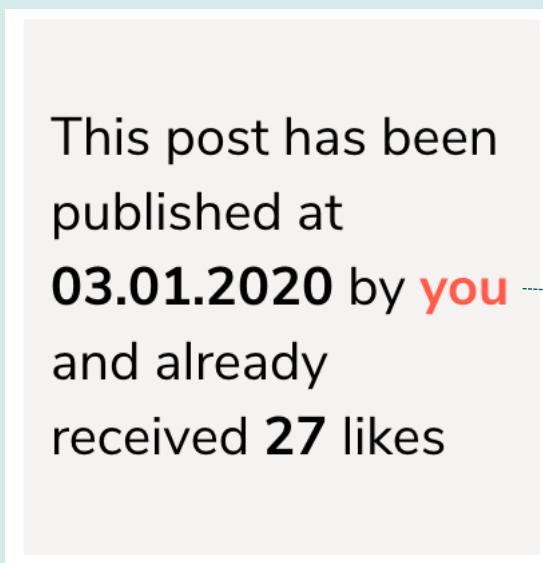
Your Post!

Read more

GLOBALE DATEN

Beispiel: angemeldeter Benutzer

Ansatz 1: React Context



User-Objekt mit Daten und Funktionen liegt in einem Context



REACT CONTEXT

Ansatz: React Context

- Analog zum Form-Beispiel, nur für andere Art von Daten

```
function AuthContextProvider(props) {  
  const [ user, setUser ] = React.useState(...);  
  
  function login(username, password) {  
    loginViaHttp(...).then(response => setUser(response.user));  
  }  
  
  function logout() { setUser(null); }  
  
  return (  
    <AuthContext.Provider value={ {  
      user, login, logout ----- Bereitgestellte Werte und Callback-Funktionen  
    } }>  
      { children }  
    </AuthContext.Provider>  
  );  
}
```

REACT CONTEXT

Idee: Custom Hook - "Fachlicher" Zugriff auf Context

Versteckt, die Tatsache, dass es sich um einen Context handelt

```
function useAuth() {  
  // AuthContext ist Implementierungsdetail  
  
  const auth = useContext(AuthContext);  
  return auth;  
}  
  
function CurrentUser(props) {  
  const { user } = useAuth();  
  
  return ...;  
}
```

REACT CONTEXT

useReducer & useContext – Redux Light?

Wir können den AuthContext mit useReducer implementieren

Bereitgestellte Funktionen dispatchen dann Actions

REACT CONTEXT

useReducer & useContext – Redux Light?

Wir können den AuthContext mit useReducer implementieren

Bereitgestellte Funktionen dispatchen dann Actions

```
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

Wir können den AuthContext mit useReducer implementieren

Bereitgestellte Funktionen dispatchen dann Actions

- 👍 Globaler Zustand (kann ganz oben in der Hierarchie eingefügt werden)
 - Strukturierung nach Geschmack möglich (Zusand 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

Wir können den AuthContext mit useReducer implementieren

Bereitgestellte Funktionen dispatchen dann Actions

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

- 👎 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)

Redux? 😊

GLOBALE DATEN

Beispiel: angemeldeter Benutzer

Ansatz 2: Redux

This post has been published at **03.01.2020** by **you** and already received **27** likes

Welcome, **Nils Hartmann**
[Logout](#)

User-Objekt mit Daten und Funktionen liegt im **globlen Redux Store**

03.01.2020

Routing Solutions for React

[Read more](#)

Your Post!

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>  
}
```

GLOBALER ZUSTAND

Beispiel: Custom Hook

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

Beispiel: Custom Hook

Analog zum Context-Beispiel

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

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

AUSBLICK: REDUX

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

AUSBLICK: REDUX

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

```
import { createSlice } from '@reduxjs/toolkit'

const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo(state, action) {
      const { id, text } = action.payload
      state.push({ id, text, completed: false })
    },
    toggleTodo(state, action) {
      const todo = state.find(todo => todo.id === action.payload)
      if (todo) {
        todo.completed = !todo.completed
      }
    }
  }
})

export const { addTodo, toggleTodo } = todosSlice.actions

export default todosSlice.reducer
```

Quelle: Redux Toolkit Dokumentation

AUSBLICK: REDUX

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

```
import { createSlice } from '@reduxjs/toolkit'

const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo(state, action) {
      const { id, text } = action.payload
      state.push({ id, text, completed: false })
    },
    toggleTodo(state, action) {
      const todo = state.find(todo => todo.id === action.payload)
      if (todo) {
        todo.completed = !todo.completed
      }
    }
  }
})

export const { addTodo, toggleTodo } = todosSlice.actions

export default todosSlice.reducer
```

behandelt 'addTodo' /
'toggleTodo'-Actions
(kein switch mehr erforderlich)

AUSBLICK: REDUX

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

```
import { createSlice } from '@reduxjs/toolkit'

const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo(state, action) {
      const { id, text } = action.payload
      state.push({ id, text, completed: false })
    },
    toggleTodo(state, action) {
      const todo = state.find(todo => todo.id === action.payload)
      if (todo) {
        todo.completed = !todo.completed
      }
    }
  }
})
export const { addTodo, toggleTodo } = todosSlice.actions
export default todosSlice.reducer
```

erzeugt automatisch
Action Creator-Funktionen

AUSBLICK: REDUX

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

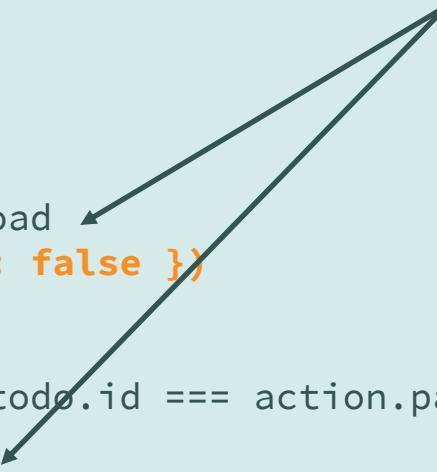
```
import { createSlice } from '@reduxjs/toolkit'

const todosSlice = createSlice({
  name: 'todos',
  initialState: [],
  reducers: {
    addTodo(state, action) {
      const { id, text } = action.payload
      state.push({ id, text, completed: false })
    },
    toggleTodo(state, action) {
      const todo = state.find(todo => todo.id === action.payload)
      if (todo) {
        todo.completed = !todo.completed
      }
    }
  }
})

export const { addTodo, toggleTodo } = todosSlice.actions

export default todosSlice.reducer
```

ImmerJS transparent
eingebunden



Suspense

RENDERN UNTERBRECHEN

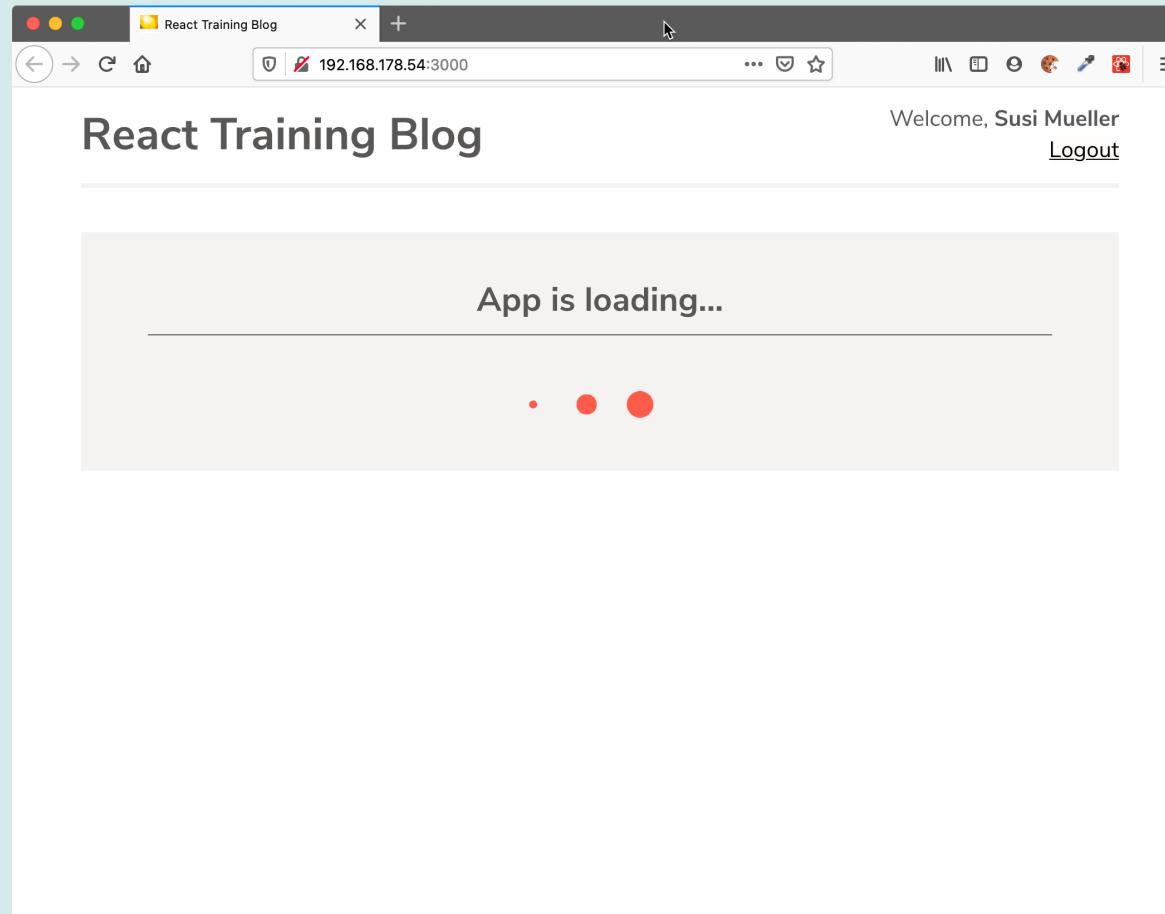
SUSPENSE

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"));  
Dynamic Import  
class App {  
  render() {  
    return <Switch>  
      <Route path="/login">  
        <LoginPage />  
      </Route>  
      // ...weitere Seiten...  
    </Switch>  
  }  
}
```

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

AUSBLICK

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

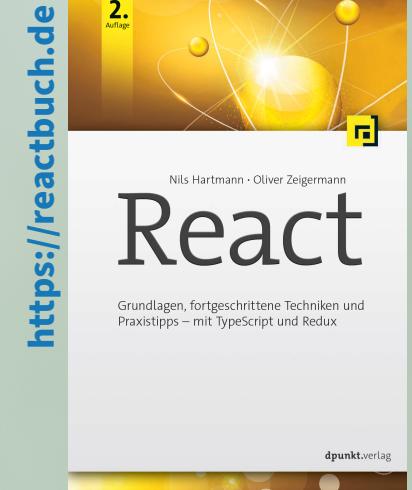
CONCURRENT REACT

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
 - Konzepte/Bibliotheken zum Vorladen von Daten

NILS HARTMANN

<https://nilshartmann.net>



vielen Dank!

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

Fragen & Kontakt: nils@nilshartmann.net

NILS@NILSHARTMANN.NET