

WBE: UI-BIBLIOTHEK

TEIL 3: EINSATZ

ÜBERSICHT

- Zustand von Komponenten
- Komponenten-Design
- Optimierungsansätze

ÜBERSICHT

- Zustand von Komponenten
- Komponenten-Design
- Optimierungsansätze

ZUSTAND

- Komponenten sollen auch einen Zustand haben können
- In React möglich, zum Beispiel mit als Klassen implementierten Komponenten
- Neuere Variante: **Hooks**, in diesem Fall: **State-Hook**

STATE-HOOK IN REACT

```
const [stateVar, setStateVar] = useState(initialValue)
```

- `useState` liefert Zustand und Update-Funktion
- Initialwert wird als Argument übergeben
- Zustandsänderung führt zum erneuten Rendern der Komponente

STATE-HOOK IN REACT

```
const Counter = () => {  
  const [state, setState] = useState(1)  
  const handler = () => setState(c => c + 1)  
  
  return (  
    ["h1", {onclick:handler, style:{userSelect:"none",cursor:"pointer"}}],  
    "Count: " + state  
  )  
}  
  
const element = [Counter]
```

STATE-HOOK: UMSETZUNG

- Änderungen in `updateFunctionComponent` nötig
- Aktuelles Element erhält ein Attribut `hooks` (Array)
- Beim Aufruf der Komponente wird `useState` aufgerufen
- Dabei: Hook angelegt mit altem Zustand oder Initialwert
- Ausserdem wird `setState` definiert:
 - Aufrufe in einer Queue im Hook speichern
 - Re-render des Teilbaums anstossen
- Nächster Durchgang: alle Aktionen in Queue ausführen

STATE-HOOK IN SUIWEB

- State hooks sind auch in SuiWeb umgesetzt
- <https://suiweb.github.io/docs/tutorial/4-hooks>

BEISPIEL: EVENT

```
import { render, parseSjdon, createElement, useState }  
from "../lib/suiweb.min.js"  
  
const Counter = () => {  
  const [state, setState] = useState(1)  
  const handler = () => setState(state + 1)  
  
  return (  
    ["h1", {onclick:handler, style:{userSelect:"none",cursor:"pointer"}}],  
    ["Count: " + state]  
  )  
}  
  
const element = [Counter]
```

[demo-21-state](#) →

BEISPIEL: TIMER (TEIL 1)

```
const App = () => {  
  let initialState = {  
    heading: "Awesome SuiWeb (Busy)",  
    content: "Loading...",  
    timer: null,  
  }  
  
  let [state, setState] = useState(initialState)  
  
  if (!state.timer) {  
    setTimeout(() => {  
      setState({ heading: 'Awesome SuiWeb', content: 'Done!',  
        timer: true, })  
    }, 3000)  
  } ...  
}
```

BEISPIEL: TIMER (TEIL 2)

```
const App = () => {  
  ...  
  const { heading, content } = state  
  
  return (  
    [ "main",  
      [ "h1", heading],  
      [ "p", content] ]  
  )  
}
```

demo-22-state →

Speaker notes

Timer im Komponentenzustand sind eine etwas knifflige Angelegenheit. Nach drei Sekunden wird der Zustand geändert. Das führt zu einem erneuten Rendering der Komponente. Dabei muss darauf geachtet werden, dass dadurch nicht ein neuer Timer gestartet wird. Das würde nämlich wieder zu einem Re-rendering führen, erneut würde ein Timer gestartet usw.

Damit beim vom Timer ausgelösten Rendering nicht ein neuer Timer gestartet wird, ergänzen wir den Zustand hier um ein Flag, das anzeigt, ob wir bereits einen Timer haben.

BEISPIEL: TIMER

- Komponente zunächst mit Default-Zustand angezeigt
- Nach 3 Sekunden wird der Zustand aktualisiert
- Diese Änderung wird im UI nachgeführt

Das UI wird einmal deklarativ spezifiziert. Über die Zeit kann sich der Zustand der Komponente ändern. Um die Anpassung des DOM kümmert sich die Bibliothek.

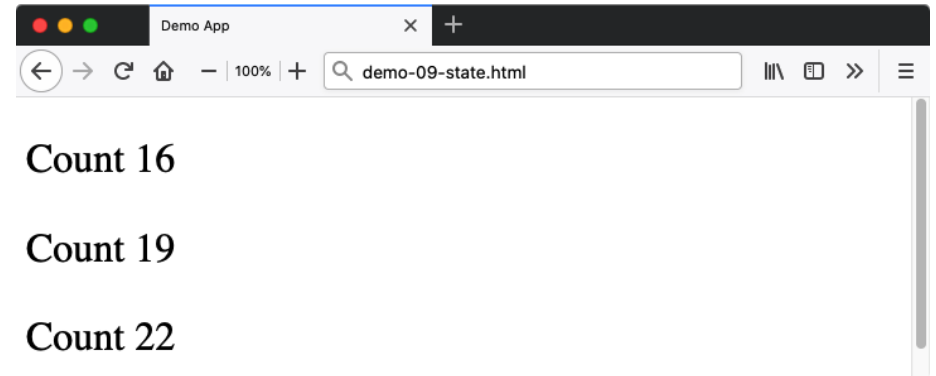
BEISPIEL: ZÄHLER (TEIL 1)

```
const Counter = (props) => {  
  let [count, setCount] = useState(props.count)  
  setTimeout(() => setCount(count+1), 1000)  
  
  return (  
    ["p",  
      {style: "font-size:2em"},  
      "Count ", count ]  
  )  
}
```

BEISPIEL: ZÄHLER (TEIL 2)

```
const App = (props) =>
  ["div",
    [Counter, {count: 1, key: 1}],
    [Counter, {count: 4, key: 2}],
    [Counter, {count: 7, key: 3}] ]
```

demo-23-state →



ZUSTAND UND PROPERTIES

- Komponente kann einen Zustand haben (`useState`-Hook)
- Properties werden als Argument übergeben (`props`-Objekt)
- Zustand und Properties können Darstellung beeinflussen
- Weitergabe von Daten (aus Zustand und Properties) an untergeordnete Komponenten wiederum als Properties

KONTROLLIERTE EINGABE

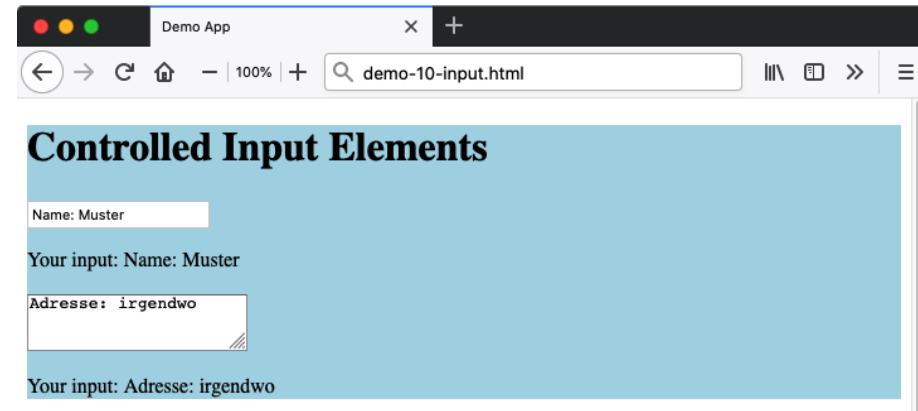
- Zustand bestimmt, was in Eingabefeld angezeigt wird
- Jeder Tastendruck führt zu Zustandsänderung
- Problem: beim Re-Render geht der Fokus verloren
- In SuiWeb nur unbefriedigend gelöst: Index des Elements und Cursor-Position werden gespeichert

KONTROLLIERTE EINGABE

```
const App = ({init}) => {
  let [text, setText] = useState(init)
  let [otherText, setOtherText] = useState("")

  const updateValue = e => {
    setText(e.target.value)
  }
  const updateOtherValue = e => {
    setOtherText(e.target.value)
  }
  return (
    [ "div", {style: "background: lightblue"},
      [ "h1", "Controlled Input Elements"],
      [ "input", {oninput: updateValue, value: text}],
      [ "p", "Your input: ", text ],
      [ "textarea", {oninput: updateOtherValue}, otherText],
      [ "p", "Your input: ", otherText ] ] )
  )
}
const element = [App, {init: "Name"}]
```

demo-24-input →



KONTROLLIERTE EINGABE

- Ermöglicht es, nur bestimmte Eingaben zu erlauben
- Beispiel: nur Ziffern und Dezimalpunkt erlaubt

```
const updateValue = e => {  
  const inp = e.target.value  
  const reg = /^\\d+\\.?\\d*$/  
  if (reg.test(inp)) setText(inp)  
  else setText(text)  
}
```

ÜBERSICHT

- Zustand von Komponenten
- Komponenten-Design
- Optimierungsansätze

CONTAINER-KOMPONENTE

- Daten-Verwaltung von Daten-Darstellung trennen
- Container-Komponente zuständig, Daten zu holen
- Daten per `props` an Render-Komponenten weitergegeben
- Übliches Muster in React-Applikationen

BEISPIEL

```
1  /* Utility function that's intended to mock a service that this
2  /* component uses to fetch it's data. It returns a promise, just
3  /* like a real async API call would. In this case, the data is
4  /* resolved after a 2 second delay. */
5
6  function fetchData() {
7      return new Promise((resolve) => {
8          setTimeout(() => {
9              resolve([ 'First', 'Second', 'Third' ])
10          }, 2000)
11      })
12  }
```

CONTAINER-KOMPONENTE

```
1  const MyContainer = () => {  
2  
3    let initialState = { items: ["Fetching data..."] }  
4    let [state, setState] = useState(initialState)  
5  
6    if (state === initialState) {  
7      fetchData()  
8      .then(items => setState(() => ({ items })))  
9    }  
10  
11    return (  
12      [MyList, state]  
13    )  
14  }
```

[demo-25-container](#) →

EFFECT HOOK

- Container-Komponenten haben verschiedene Aufgaben
- Zum Beispiel: Timer starten, Daten übers Netz laden
- In React unterstützen Klassen-Komponenten zu diesem Zweck verschiedene **Lifecycle-Methoden**, u.a.:
 - `componentDidMount`: Komponente wurde gerendert
 - `componentWillUnmount`: Komponente wird gleich entfernt
- In Funktionskomponenten: **Effect Hooks**
- Funktionen, die nach dem Rendern ausgeführt werden

<https://react.dev/learn/synchronizing-with-effects>

EFFECT HOOK

```
const MyContainer = () => {  
  // after the component has been rendered, fetch data  
  useEffect(() => {  
    fetchData()  
      .then(items => setState(() => ({ items })))  
  }, []) ...  
}
```

- SuiWeb unterstützt Effect Hooks
- Hier ist ein weiteres Beispiel:
<https://suiweb.github.io/docs/tutorial/4-hooks#indexjs>

Tatsächlich ersetzen Effect Hooks drei Lifecycle-Methoden:

- `componentDidMount`
- `componentDidUpdate`
- `componentWillUnmount`

Normalerweise wird der Effect ausgeführt, wenn eine Komponente zum ersten Mal rerendert wird (`componentDidMount`), aber auch nach jedem Update (`componentDidUpdate`). Um dies feiner zu steuern, kann `useEffect` als zweites Argument ein Array von Werten mitgegeben werden. Der Effect wird dann nur ausgeführt, wenn seit dem letzten Rendern mindestens einer der Werte im Array verändert wurde. Wenn ein leeres Array übergeben wird, hat dies zur Folge, dass der Effect nur nach dem ersten Mounten der Komponente ausgeführt wird.

Die als Effect übergebene Funktion kann als Return-Wert eine Cleanup-Funktion liefern. Diese wird dann beim Entfernen der Komponente ausgeführt, womit auch `componentWillUnmount` durch Effect Hooks unterstützt wird.

<https://react.dev/learn/synchronizing-with-effects>

MONOLITHISCHE KOMPONENTEN

- Design-Entscheidung: wie viel UI-Logik in einer Komponente?
- Einfaches UI in einer einzelnen Komponente realisieren?
- Damit: weniger Komponenten zu entwickeln und pflegen
- Und: weniger Kommunikation zwischen Komponenten

Aber:

- Wenig änderungsfreundlich
- Kaum Wiederverwendung von Komponenten

BEISPIEL-ANWENDUNG

Articles

<input type="text" value="Title"/>	<input type="text" value="Summary"/>	<input type="button" value="Add"/>
------------------------------------	--------------------------------------	------------------------------------

- [Article 1](#) ✕

Article 1 Summary

- [Article 2](#) ✕
- [Article 3](#) ✕
- [Article 4](#) ✕

- Artikel können hinzugefügt werden
- Artikel: Titel, Zusammenfassung
- Klick auf den Titel: Inhalt ein- und ausblenden
- Klick auf ✕: Artikel löschen

AUFTEILUNG IN KOMPONENTEN

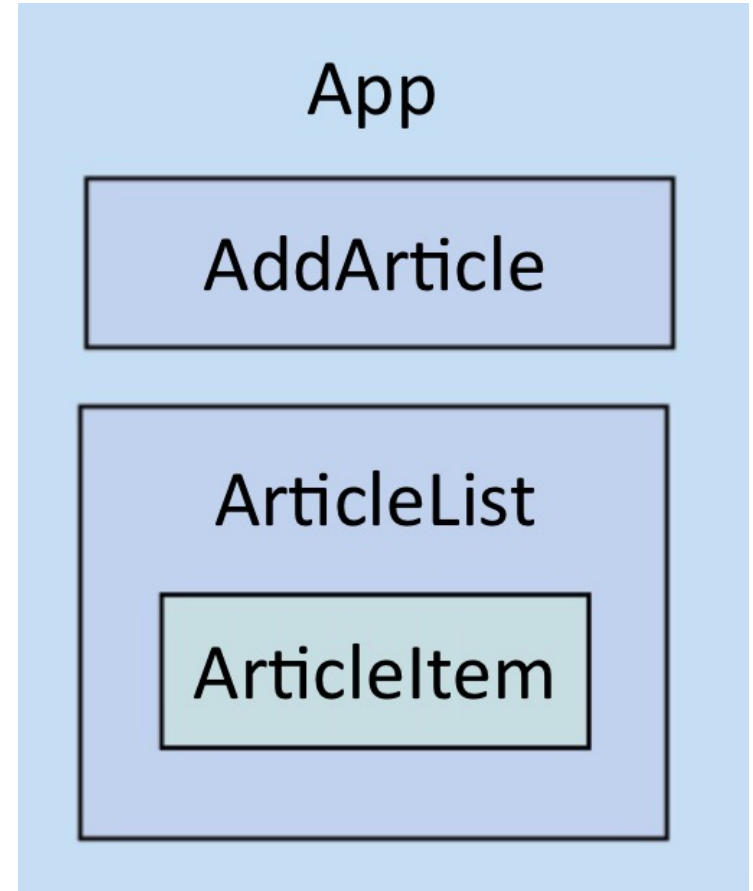
Articles

<input type="text" value="Title"/>	<input type="text" value="Summary"/>	<input type="button" value="Add"/>
------------------------------------	--------------------------------------	------------------------------------

- [Article 1](#) ✕

Article 1 Summary

- [Article 2](#) ✕
- [Article 3](#) ✕
- [Article 4](#) ✕



AUFTEILUNG IN KOMPONENTEN

```
const App = () => {  
  
  let initialState = { ... }  
  let [state, setState] = useState(...)  
  
  const onChangeTitle = e => { ... }  
  const onChangeSummary = e => { ... }  
  const onClickAdd = e => { ... }  
  const onClickRemove = (id) => { ... }  
  const onClickToggle = (id) => { ... }
```

```
  return (  
    [ "section",  
      [AddArticle, {  
        name: "Articles",  
        title: state.title,  
        summary: state.summary,  
        onChangeTitle,  
        onChangeSummary,  
        onClickAdd,  
      }],  
      [ArticleList, {  
        articles: state.articles,  
        onClickToggle,  
        onClickRemove,  
      }]  
    ]  
  )  
}
```

AUFTEILUNG IN KOMPONENTEN

- Komponente `App` kümmert sich um den Zustand
- Sie enthält: Event Handler zum Anpassen des Zustands
- Ausgabe übernehmen `AddArticle` und `ArticleList`
- Diese bekommen dazu den Zustand und die Handler in Form von Properties übergeben

APPLIKATIONSZUSTAND

```
const App = () => {  
  let initialState = {  
    articles: [  
      {  
        id: cuid(),  
        title: 'Article 1',  
        summary: 'Article 1 Summary',  
        display: 'none',  
      },  
      ...  
    ],  
    title: '',  
    summary: '',  
  }  
  ...  
}
```

- Array von Artikeln
- Generierte IDs
- und für Eingabefelder (controlled input)

EREIGNISBEHANDLUNG

```
const App = () => {  
  let initialState = { ...}  
  let [state, setState] = useState(initialState)  
  
  const onChangeTitle = e => {  
    setState({...state, title: e.target.value})  
  }  
  const onClickRemove = (id) => {  
    let articles = state.articles.filter(a => a.id !== id)  
    setState({...state, articles})  
  }  
  /*...*/  
  return (...)  
}
```

AUFTEILUNG IN KOMPONENTEN

```
const AddArticle = ({name, title, summary,
  onChangeTitle, onChangeSummary, onClickAdd}) => (

  ["section",
    ["h1", name],
    ["input", { placeholder: "Title", value: title,
      oninput: onChangeTitle }],
    ["input", { placeholder: "Summary", value: summary,
      oninput: onChangeSummary }],
    ["button", { onclick: onClickAdd }, "Add"] ]
  )
```

AUFTEILUNG IN KOMPONENTEN

```
const ArticleList = ({articles, onClickToggle, onClickRemove}) => (  
  ["ul", ...articles.map(i => (  
    [ArticleItem, {  
      key: i.id,  
      article: i,  
      onClickToggle,  
      onClickRemove} ]))]  
)
```

demo-26-design →

AUFTEILUNG IN KOMPONENTEN

- Zustand in wenigen Komponenten konzentriert
- Andere Komponenten für den Aufbau des UI zuständig
- Im Beispiel: Zustandsobjekt enthält kompletten Applikationszustand (inkl. Inhalt der Eingabefelder)
- Event Handler passen diesen Zustand an und basteln nicht am DOM herum

MODULE

- Komponenten können in eigene Module ausgelagert werden
- Zusammen mit komponentenspezifischen Styles
- Sowie mit lokalen Hilfsfunktionen

Separation of Concerns

- Wo sollte getrennt werden?
- Zwischen Markup und Styles und Programmlogik?
- Zwischen Komponenten?

MODULE

```
import { ArticleItem } from "./ArticleItem.js"

const ArticleList = ({articles, onClickToggle, onClickRemove}) => (
  ["ul", ...articles.map(i => (
    [ArticleItem, {
      key: i.id,
      article: i,
      onClickToggle,
      onClickRemove} ]))]
)

export { ArticleList }
```

demo-27-modules →

NETZWERKZUGRIFF

- Letztes Beispiel erweitert
- Falls Artikelliste leer: Button zum Laden vom Netz
- Dazu stellt unser [Express-REST-Service](#) unter der id `articles` eine Artikelliste mit ein paar Mustereinträgen zur Verfügung

NETZWERKZUGRIFF

```
const App = () => {  
  let [state, setState] = useState(initialState)  
  
  /* ... */  
  
  return (  
    ["section",  
      [AddArticle, { ... } ],  
  
      state.articles.length !== 0  
      ? [ArticleList, {articles: state.articles, onClickToggle, onClickRemove}]  
      : ["p", ["button", {onclick: onLoadData}, "Load Articles"]]  
    )  
  )  
}
```


NETZWERKZUGRIFF

```
// Load articles from server
const onLoadData = () => {
  let url = 'http://localhost:3000/'
  fetch(url + "api/data/articles?api-key=wbeweb", {
    method: 'GET',
  })
    .then(response => response.json())
    .then(articles => setState({...state, articles}))
    .catch(() => {alert("Network connection failed")})
}
```

demo-28-network →

Speaker notes

Zum Laden braucht's hier natürlich den aktiven REST-Service. Dieser ist in `lib-express` und wird so gestartet:

```
$ npm install  
$ node index.js
```

IMPERATIVER ANSATZ

- Ergänze alle Code-Teile in denen die Artikelliste erweitert oder verkleinert wird wie folgt:
- Wenn der letzte Artikel gelöscht wird, entferne `<u1></u1>` und füge einen Button für den Netzwerkzugriff ein
- Wenn der erste Artikel eingefügt wird, entferne den Button und füge ein `<u1></u1>` mit dem ersten `/` ein
- usw.

DEKLARATIVER ANSATZ

- Wenn die Artikelliste leer ist, wird ein Button ausgegeben
- Ansonsten wird die Artikelliste ausgegeben

Wir ändern nur den Zustand... 👍

HAUPTKONZEPTE

- Klarer und einfacher Datenfluss:
 - Daten nach unten weitergegeben (props)
 - Ereignisse nach oben weitergegeben und dort behandelt
- Properties werden nicht geändert, Zustand ist veränderbar
- Zustand wird von Komponente verwaltet
- Es ist von Vorteil, die meisten Komponenten zustandslos zu konzipieren

ÜBERSICHT

- Zustand von Komponenten
- Komponenten-Design
- Optimierungsansätze

OPTIMIERUNGSANSÄTZE

- SuiWeb ist nicht für den produktiven Einsatz gedacht
- Im Folgenden werden Optimierungsansätze beschrieben
- Diese sind in SuiWeb nur teilweise implementiert
- Angelehnt an:

Rodrigo Pombo: Build your own React

<https://pomb.us/build-your-own-react/>

Die Optimierungen werden hier nur grob skizziert und gehören nicht zum WBE-Pflichtstoff. Bei Interesse bitte angegebene Quellen konsultieren.

OPTIMIERUNG

Problem:

Die render-Funktion blockiert den Browser, was besonders beim Rendern grösserer Baumstrukturen problematisch ist

Abhilfe:

- Zerlegen der Aufgabe in Teilaufgaben
- Aufruf mittels `requestIdleCallback`
- Achtung: experimentelle Technologie
- React selbst verwendet dafür mittlerweile **ein eigenes Paket**

„FWIW we’ve since stopped using requestIdleCallback...”

<https://github.com/facebook/react/issues/11171>

Speaker notes

FWIW: for what it's worth

Eine Floskel, die in verschiedenen Situationen verwendet wird, etwa im Sinne von „tatsächlich kann man sagen dass“ oder „ich meine es ist so dass“.

OPTIMIERUNG

```
let nextUnitOfWork = null

function workLoop (deadline) {
  let shouldYield = false
  while (nextUnitOfWork && !shouldYield) {
    nextUnitOfWork = performUnitOfWork(
      nextUnitOfWork
    )
    shouldYield = deadline.timeRemaining() < 1
  }
  requestIdleCallback(workLoop)
}

requestIdleCallback(workLoop)

function performUnitOfWork (nextUnitOfWork) {
  // TODO
}
```

Speaker notes

Hinweis: `window.requestIdleCallback` ist eine experimentelle Technologie und funktioniert derzeit nicht in allen Browsern. Zur Abhilfe kann folgendes Script verwendet werden:

```

window.requestIdleCallback = window.requestIdleCallback || function(handler) {
  let startTime = Date.now();

  return setTimeout(function() {
    handler({
      didTimeout: false,
      timeRemaining: function() {
        return Math.max(0, 50.0 - (Date.now() - startTime));
      }
    });
  }, 1);
}

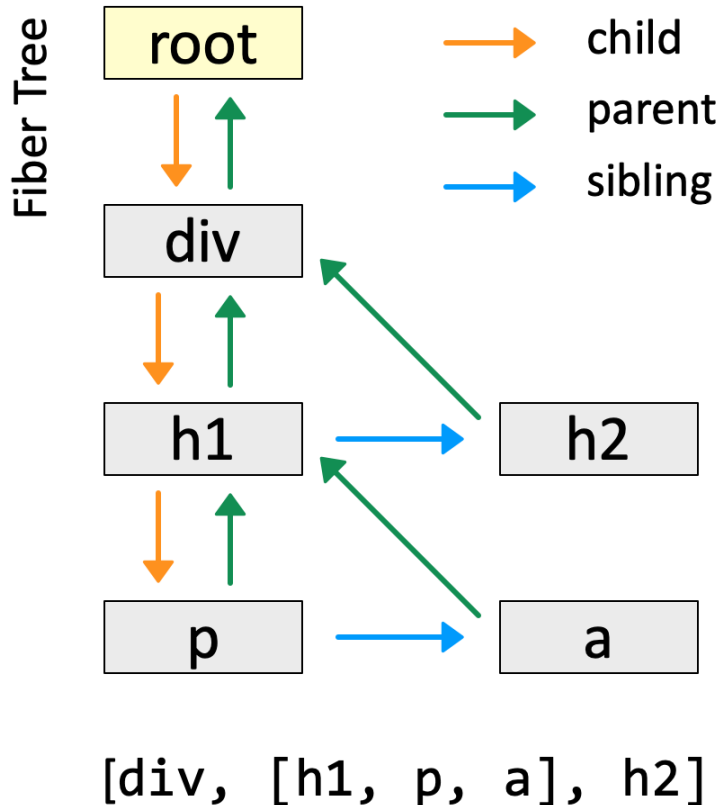
window.cancelIdleCallback = window.cancelIdleCallback || function(id) {
  clearTimeout(id);
}

```

OPTIMIERUNG: FIBERS

- Offen: wie wird das Rendern in Teilaufgaben zerlegt?
- Datenstruktur: **Fiber Tree**
- Ziel: einfaches Auffinden des nächsten Arbeitsschritts
- Fiber heisst eigentlich Faser
- Terminologie hier: **Arbeitspaket**
(eigentlich: Unter-/Teilauftrag)

FIBERS: DATENSTRUKTUR



- Elemente geeignet verlinkt
- Jedes Arbeitspaket kennt
 - erstes Kind
(first child)
 - nächstes Geschwister
(next sibling)
 - übergeordnetes Element
(parent)

FIBERS: NÄCHSTER SCHRITT

- Kind falls vorhanden
- sonst: nächstes Geschwister falls vorhanden
- sonst: Suche nach oben bis Element mit Geschwister
- sonst: fertig

FIBERS: IMPLEMENTIERUNG

- Funktion `render` aufgeteilt
- Legt nun erstes Arbeitspaket fest
- In `createDom` wird DOM-Knoten mit Attributen angelegt

```
let nextUnitOfWork = null
```

```
function render (element, container) {  
  // erstes Arbeitspaket festlegen  
}
```

```
function workLoop (deadline) {  
  // Arbeitspakete bearbeiten  
}
```

FIBERS: IMPLEMENTIERUNG

- Noch offen: `performUnitOfWork`
- Bearbeitet aktuellen Auftrag und liefert nächsten Auftrag
- Dieser wird im while gleich bearbeitet, falls Browser idle
- Sonst im nächsten `requestIdleCallback`

```
function performUnitOfWork (fiber) {  
  // TODO add dom node  
  // TODO create new fibers  
  // TODO return next unit of work  
}
```


Speaker notes

```
function createDom (fiber) {  
  // create DOM node  
  const dom =  
    fiber.type == "TEXT_ELEMENT"  
      ? document.createTextNode("")  
      : document.createElement(fiber.type)  
  
  // assign the element props  
  const isProperty = key => key !== "children"  
  Object.keys(fiber.props)  
    .filter(isProperty)  
    .forEach(name => {  
      dom[name] = fiber.props[name]  
    })  
  
  return dom  
}
```

- Erzeugt einen DOM-Knoten
- je nach fiber.type Text oder Element
- ergänzt ihn um alle fiber.props (ausser children)
- gibt Knoten zurück

FIBERS: IMPLEMENTIERUNG

```
function performUnitOfWork(fiber) {  
  // TODO add dom node  
  // TODO create new fibers  
  // TODO return next unit of work  
}
```

- Knoten anlegen (`createDom`) und ins DOM einhängen
- Für jedes Kindelement Arbeitspaket (Fiber) anlegen
- Referenzen eintragen (`sibling`, `parent`, `child`)
- Nächstes Arbeitspaket suchen und zurückgeben

```
function performUnitOfWork(fiber) {  
  if (!fiber.dom) {  
    fiber.dom = createDom(fiber)  
  }  
  if (fiber.parent) {  
    fiber.parent.dom.appendChild(fiber.dom)  
  }  
  const elements = fiber.props.children  
  let index = 0  
  let prevSibling = null  
  
  while (index < elements.length) {  
    const element = elements[index]  
  
    const newFiber = {  
      type: element.type,  
      props: element.props,  
      parent: fiber,  
      dom: null,  
    }  
  }
```

```

    if (index === 0) {
      fiber.child = newFiber
    } else {
      prevSibling.sibling = newFiber
    }
    prevSibling = newFiber
    index++
  }
  if (fiber.child) {
    return fiber.child
  }
  let nextFiber = fiber
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling
    }
    nextFiber = nextFiber.parent
  }
}

```

ZWISCHENSTAND

- Browser kann nun UI-Aufbau unterbrechen
- Möglich bei Unterbrechung: unvollständiges UI
- Abhilfe: alle DOM-Knoten erzeugen aber noch nicht zusammenhängen
- Am Schluss alles zusammenhängen

AUFTEILUNG IN ZWEI PHASEN

Erste Phase:

- Fibers anlegen
- DOM-Knoten anlegen (`dom`-Attribut)
- Properties hinzufügen
- Fibers verlinken: `parent`, `child`, `sibling`

Zweite Phase:

- DOM-Teil der Fibers (`.dom`) ins DOM hängen

Implementierung: s. Step V: Render and Commit Phases

<https://pomb.us/build-your-own-react/>

ABGLEICH MIT LETZTER VERSION

- Ziel: nur Änderungen im DOM nachführen
- Referenz auf letzte Version des Fiber Tree: `currentRoot`
- Jedes Fiber erhält Referenz auf letzte Version: `alternate`
- Nach der Aktualisierung wird aktuelle zur letzten Version
- Unterscheidung von `UPDATE`- und `PLACEMENT`-Fibers
- Ausserdem eine Liste der zu löschenden Knoten

Didact: (Rodrigo Pombo)

<https://codesandbox.io/s/didact-6-96533>

QUELLEN

- Adam Boduch: React and React Native
Second Edition, Packt Publishing, 2018
Packt Online Shop
- Rodrigo Pombo: Build your own React
<https://pomb.us/build-your-own-react/>

VERWEISE

- Rodrigo Pombo: Build your own React
<https://pomb.us/build-your-own-react/>
- SuiWeb - An Educational Web Framework (Inspired by React)
<https://github.com/suiweb/suiweb>

