# Web-Entwicklung

Jil Zerndt, Lucien Perret
December 2024

## Browser-Technologien

Vordefinierte Objekte

- Die allgemeinen Objekte sind in JavaScript vordefiniert
- Tatsächlich handelt es sich um Funktionen/Konstruktoren
- Die Browser-Objekte existieren auf der Browser-Plattform
- Sie beziehen sich auf das Browser-Fenster, das angezeigte Dokument, oder den Browser selbst
  document
- Repräsentiert die angezeigte Webseit
- Einstieg ins DOM (Document Object Model)
- Diverse Attribute und Methoden, zum Beispiel:

```
1 document.cookie /* Zugriff auf Cookies */
2 document.lastModified /* Zeit der letzten Änderung */
3 document.links /* die Verweise der Seite */
4 |document.images /* die Bilder der Seite */
```

window

- Repräsentiert das Browserfenster
- Zahlreiche Attribute und Methoden, u.a.:
- Alle globalen Variablen und Methoden sind hier angehängt
- Neue globale Variablen landen ebenfalls hier

```
1 window.document /* Zugriff auf Dokument */
2 window.history /* History-Objekt */
3 window.innerHeight /* Höhe des Viewports */
4 window.pageYOffset /* vertikal gescrollte Pixel */
5 window.alert === alert /* -> true */
6 window.setTimeout === setTimeout /* -> true */
7 window.parseInt === parseInt /* true */
```

## navigator

Konsolen-eingabe auf dem folgenden Bild:

```
> navigator.userAgent
"Mozilla/5.0 (Macintosh; Intel Mac OS X 10.15; rv:91.0) Gecko/20100101 Firefox/91.0"
> navigator.language
"de"
> navigator.platform
"MacIntel"
> navigator.onLine
true
location
> location.href
"https://gburkert.github.io/selectors/"
> location.protocol
"https:"
> document.location.protocol
"https:"
```

## JavaScript und DOM

- Element erzeugen: document.createElement
- Attribute erzeugen: document.createAttribute
- Und hinzufügen: .setAttribute
- Element in Baum einfügen: .appendChild

Element auffinden

```
1 let aboutus = document.getElementById("aboutus")
2 let aboutlinks = aboutus.getElementsByTagName("a")
3 let aboutimportant = aboutus.getElementsByClassName("important")
4 let navlinks = document.querySelectorAll("nav a")
```

## Textknoten erzeugen

```
<p>The <img src="img/cat.png" alt="Cat"> in the
<img src="img/hat.png" alt="Hat">.</p>
<p><button onclick="replaceImages()">Replace</button></p>
<script>
    function replaceImages () {
        let images = document.body.getElementsByTagName("img")
        for (let i = images.length - 1; i >= 0; i--) {
                let image = images[i]
                if (image.alt) {
                let text = document.createTextNode(image.alt)
                image.parentNode.replaceChild(text, image)
                }
            }
    }
</script>
```

Elementknoten erzeugen

```
<blockquote id="quote">
    No book can ever be finished. While working on it we learn
</blockquote>
<script>
/* definition of elt ... */
document.getElementById("quote").appendChild(
    elt("footer", "-",
    elt("strong", "Karl Popper"),
    ", preface to the second edition of ",
    elt("em", "The Open Society and Its Enemies"),
    ", 1950"))
</script>
```

Attribut setzen

1
6
8

```
3 let att = document.createAttribute("class")
4 att.value ="democlass"
5 h1.setAttributeNode(att)
7 /* oder kürzer: */
let h1 = document.querySelector("h1")
,h1.setAttribute("class","democlass")
```

Style anpassen

1 Nice text
2

## Event handling

Event abonnieren/entfernen

Mit der Methode addEventListener() wird ein Event abonniert. Mit removeEventListener kann das Event entfernt werden.

```
<button>Act-once button</button>
<script>
    let button = document.querySelector("button")
    function once () {
        console.log("Done.")
        button.removeEventListener("click", once)
    }
    button.addEventListener("click", once)
</script>
```

Wenn ein Parameter zur Methode hinzugefügt wird, wird dieses als das Event-Objekt gesetzt.

```
<script>
    let button = document.querySelector("button")
    button.addEventListener("click", (e) => {
        console.log("x="+e.x+", y="+e.y)
    })
</script\
```

Das Event wird bei allen abonnierten Handlern ausgeführt bis ein Handler stopPropagation() ausführt.

```
<script>
    let button = document.querySelector("button")
    button.addEventListener("click", (e) => {
                console.log("x="+e.x+", y="+e.y)
...             e.stopPropagation()
    })
</script>
```

Viele Ereignisse haben ein Default verhalten. Eigene Handler werden vor Default-Verhalten ausgeführt. Um das Default-Verhalten zu verhindern, muss die Methode preventDefault() ausgeführt werden.

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
    let link = document.querySelector("a")
    link.addEventListener("click", event => {
            console.log("Nope.")
                event.preventDefault()
        })
,/script>
```

## Tastatur-Events

- keydown
- keyup
- Achtung: bei keydown kann das event mehrfach ausgelöst werden

```
<p>Press Control-Space to continue.</p>
<script>
    window.addEventListener("keydown", event => {
            if (event.key == " " && event.ctrlKey) {
                console.log("Continuing!")
            }
    })
</script>
```

Mauszeiger-Events

- Mausklicks:
- mousedown
- mouseup
- click
- dblclick
- Mausbewegung
- mousemove
- Touch-display
- touchstart
- touchmove
- touched

## Scroll-Events

Das Scrollevent hat die Attribute des Event-Objekts: pageYOffset, pageXOffset.

```
window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight
    bar.style.width = `${(pageYOffset / max) * 100}%`
,})
```

Fokus- und Ladeereignisse
- Fokus erhalten / verlieren
- focus
- blur
- Seite wurde geladen (ausgelöst auf window und document.body)
- load
- beforeunload

Jquery
JQuery ist eine freie JavaScript-Bibliothek, die Funktionen zur DOM-Navigation und -Manipulation zur Verfügung stellt.

```
$("button.continue").html("Next Step...")
var hiddenBox = $("#banner-message")
$("#button-container button").on("click", function(event) {
        hiddenBox.show()
    .})
```

| Aufruf | Bedeutung |
|---|---|
| $(Funktion) | DOM ready |
| $("CSS Selektor") .aktion(arg1, ....) .aktion(...) | Wrapped Set |
| | - Knoten, die Sel. erfüllen |
| | - eingepackt in jQuery Obj. |
| | |
| $("HTML-Code") | Wrapped Set<br>- neuer Knoten |
| | - eingepackt in jQuery Obj.<br>- noch nicht im DOM |
| $(DOM-Knoten) | Wrapped Set |
| | - dieser Knoten<br>- eingepackt in jQuery Obj. |

## Web-Grafiken

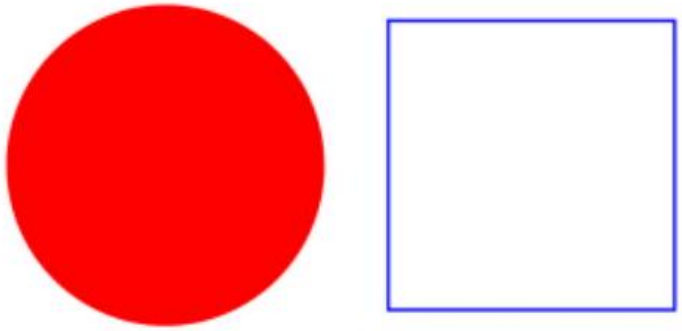
- Einfache Grafiken mit HTML und CSS möglich
- Zum Beispiel: Balkendiagramme
- Alternative für Vektorgrafiken: SVG
- Alternative für Pixelgrafiken: Canvas

SVG
- Basiert wie HTML auf XML
- Elemente repräsentieren grafische Formen
- Ins DOM integriert und durch Scripts anpassbar

Beispiel:

```
p>Normal HTML here.</p>
<svg xmlns="http://www.w3.org/2000/svg">
    <circle r="50" cx="50" cy="50" fill="red"/>
    <rect x="120" y="5" width="90" height="90" stroke="blue" fill="none"/>
</svg>
```

Ausgabe:
Normal HTML here.



JavaScript:

```
1 let circle = document.querySelector("circle")
2 circle.setAttribute("fill","cyan")
```

## Canvas

- Element canvas als Zeichenbereich im Dokument
- API zum Zeichnen auf dem Canvas

```
<canvas></canvas>
<script>
Let cx = document.querySelector("canvas").getContext("2d")
cx.beginPath()
cx.moveTo(50, 10)
cx.lineTo(10, 70)
cx.lineTo(90, 70)
cx.fill()
let img = document.createElement("img")
img.src = "img/hat.png"
img.addEventListener("load" , () => {
  for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10)
  }
})
</script>
```

Canvas Methoden

| Methoden | Beschreibung |
|---|---|
| scale | Skalieren |
| translate | Koordinatensystem verschieben |
| rotate | Koordinatensystem rotieren |
| save | Transformationen auf Stack speichern |
| restore | Letzten Zustand wiederherstellen |

Browser-API
Web Storage
Web Storage speichert Daten auf der Seite des Client.

## Local Storage

Local Storage wird verwendet, um Daten der Webseite lokal abzuspeichern. Die Daten bleiben nach dem Schliessen des Browsers erhalten. Die Daten sind in Developer Tools einsehbar und änderbar.

Die Daten werden nach Domains abgespeichert. Es können pro Webseite etwa 5MB abgespeichert werden.

```
1 localStorage.setItem("username","bkrt")
2 console.log(localStorage.getItem("username")) // -> bkrt
3 localStorage.removeItem("username")
```

Die Werte werden als Strings gespeichert, daher müssen Objekte mit JSON codiert werden:
1 Let user = {name: "Hans", highscore: 234}
2 localStorage.setItem(JSON.stringify(user))

## History

History gibt zugriff auf den Verlauf des akutellen Fensters/Tab.

```
1 function goBack() {
2 window.history.back();
3
    ,}
```

| Methoden | Beschreibung |
|---|---|
| length (Attribut) | Anzahl Einträgte inkl. aktueller Seite. Keine Methode! |
| back | zurück zur letzten Seite |

GeoLocation
Mit der GeoLocation-API kann der Standort abgefragt werden.

```
var options = { enableHighAccuracy: true, timeout: 5000, maximum
function success(pos) {
    var crd = pos.coords
    console.log(`Latitude : ${crd.latitude}`)
    console.log(`Longitude: ${crd.longitude}`)
    console.log(`More or less ${crd.accuracy} meters.`)
}
function error(err) { ... }
navigator.geolocation.getCurrentPosition(success, error, options
```

| Beispiel | |
|---|---|
| $(function0 { ... }); | |
| $(".toggleButton").attr("title"). | |
| $(".toggleButton").attr("title", "click here") | |
| $(".toggleButton").attr((title", "click here", ...)) | |
| $(".toggleButton").attr("title", function0{...}) .css(...) .text(..) on("click", function(event) { ...}) | |
| $("...").addClass(...) .appendTo(SSelektor") | |
| $(«li»-.../li>").length | $(«li>...<li>")[0] |
| $(document.body) | |
| $(this) | |

## Formulare

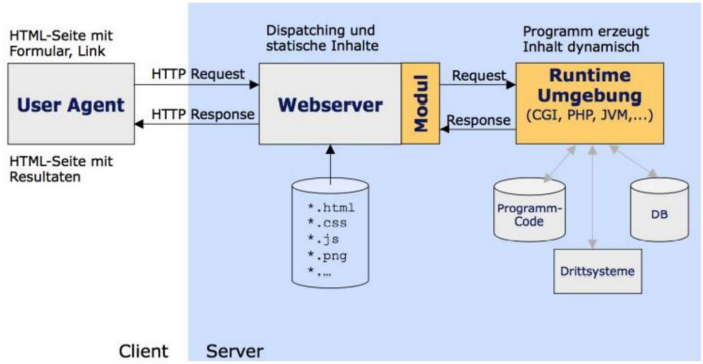
Formulare ermöglichen Benutzereingaben. Sie gilt als Grundlade für Interaktion mit dem Web.
Input types:

- submit, number, text, password, email, url , range , date , search , color


```
<form>
    <fieldset>
        <legend>General information</legend>
        <label>Text field <input type="text" value="hi"></label>
        <label>Password <input type="password" value="hi"></label
        <label class="area">Textarea <textarea>hi</textarea></label>
    </fieldset>
    <fieldset>
        <legend>Additional information</legend>
        <label>Checkbox <input type="checkbox"></label>
        <label>Radio button <input type="radio" name="demo" checked></label>
        <label>Another one <input type="radio" name="demo"></label>
    </fieldset>
    <form>
    <label>Button <button>Click me</button></label>
    <label>Select menu
    <select name="cars">
    <option value="volvo">Volvo</option>
    <option value="saab">Saab</option>
    <option value="fiat">Fiat</option>
    <option value="audi">Audi</option>
    </select>
    </label>
    <input type="submit" value="Send">
</form>
|'
```

Formulare können auch POST/GET Aktionen ausführen:
Action beschreibt das Skript, welches die Daten annimmt. Method ist die Methode die ausgeführt wird.

```
<form action="/login" method="post">
2 ...
3 </form>
```




Formular Events

| Events | Beschreibung |
|---|---|
| **change** | Formularelement geänder |
| **input** | Eingabe in Textfeld |
| **submit** | Formular absenden |

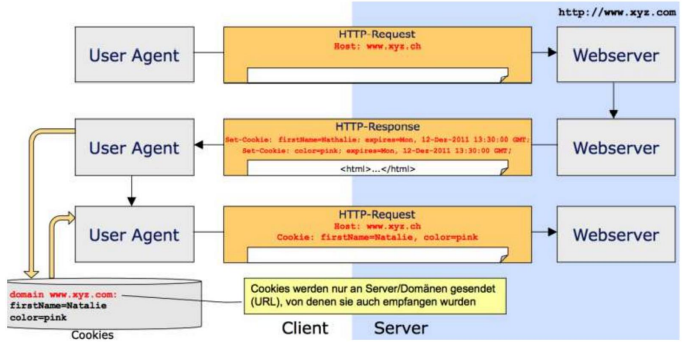
GET/POST-Methode

```
<form action="http://localhost/cgi/showenv.cgi"method="get">
    <fieldset>
        <legend>Login mit GET</legend>
        <label for="login_get">Benutzername:</label>
        <input type="text" id="login_get" name="login"/>
        <label for="password_get">Passwort:</label>
        <input type="password" id="password_get" name="password"/>
        <label for="submit_get"></label>
        <input type="submit" id="submit_get" name="submit" value="Anmelden" />
    </fieldset>
</form>
```

## Cookies und Sessions

Cookies

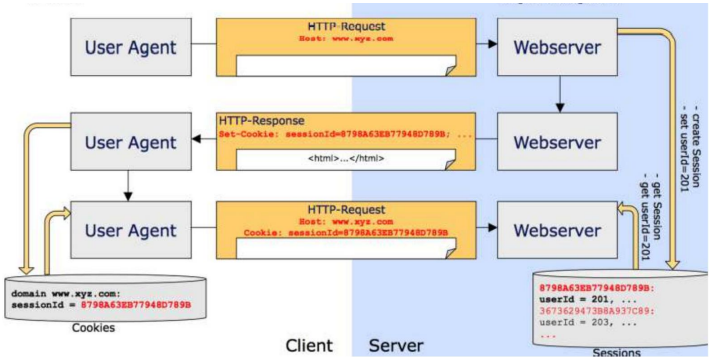
- HTTP als zustandsloses Protokoll konzipiert
- Cookies: Speichern von Informationen auf dem Client
- Response: Set-Cookie -Header, Request: Cookie -Header
- Zugriff mit JavaScript möglich (ausser HttpOnly ist gesetzt)




## Sessions

- Cookies auf dem Client leicht manipulierbar
- Session: Client-spezifische Daten auf dem Server speichern
- Identifikation des Clients über Session-ID (Cookie o.a.)
- Gefahr: Session-ID gerät in falsche Hände (Session-Hijacking)

Ablauf:
http://www.xyz.com




## Fetch API

- HTTP-Requests von JavaScripts
- Geben Promise zurück
- Nach Server-Antwort erfüllt mit Response-Objekt

```
fetch("example/data.txt")
.then(response => {
            console.log(response.status) // -> 200
    console.log(response.headers.get("Content-Type")) // -> text/pla
})
.then(resp => resp.text())
.then(text => console.log(text))
// -> This is the content of data.txt
```

Response Objekt

- headers : Zugriff auf HTTP-Header-Daten Methoden get, keys, forEach , ...
- status: Status-Code
- json() : liefert Promise mit Resultat der JSON-Verarbeitung
- text() : liefert Promise mit Inhalt der Server-Antwort

## UI Bibliothek

### UI Komponenten

## WBE: UI-BIBLIOTHEK

TEIL 1: KOMPONENTEN

## ÜBERSICHT

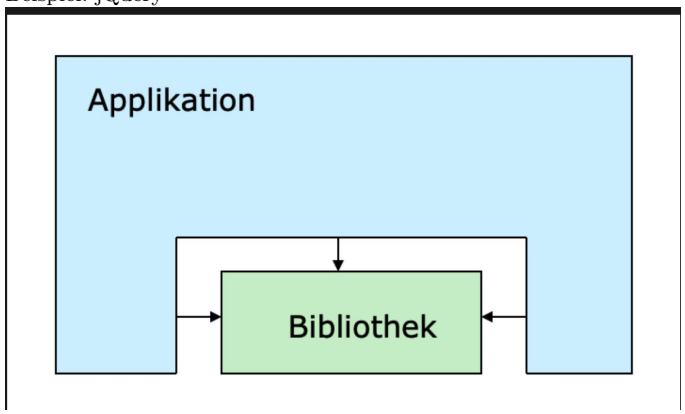
- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- JSX und SJDON
- Eigene Bibliothek: SuiWeb

## ÜBERSICHT

- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- JSX und SJDON
- Eigene Bibliothek: SuiWeb

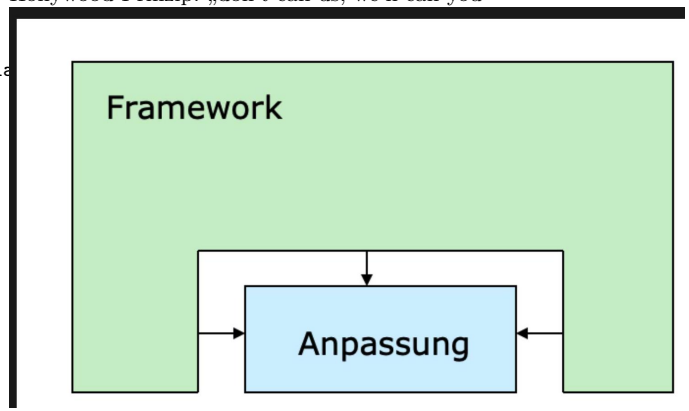
## BIBLIOTHEK

- Kontrolle beim eigenen Programm
- Funktionen und Klassen der Bibliothek verwendet
- Beispiel: jQuery




## FRAMEWORK

- Rahmen für die Anwendung
- Kontrolle liegt beim Framework
- Hollywood-Prinzip: „don't call us, we'll call you"




## ANSÄTZE IM LAUF DER ZEIT

- Statische Webseiten
- Inhalte dynamisch generiert (CGI z.B. Shell Scripts, Perl)
- Serverseitig eingebettete Scriptsprachen (PHP)
- Client Scripting oder Applets (JavaScript, Java Applets, Flash)
- Enterprise Application Server (Java, Java EE)
- MVC Server-Applikationen (Rails, Django)
- JavaScript Server (Node.js)
- Single Page Applikationen (SPAs)

## SERVERSEITE

- Verschiedene Technologien möglich
- Zahlreiche Bibliotheken und Frameworks
- Verschiedene Architekturmuster
- Häufig: Model-View-Controller (MVC)
- Beispiel: Ruby on Rails

## MODEL-VIEW-CONTROLLER (MVC)

Models

- repräsentieren anwendungsspezifisches Wissen und Daten
- ähnlich Klassen: User, Photo, Todo, Note
- können Observer über Zustandsänderungen informieren

## Views

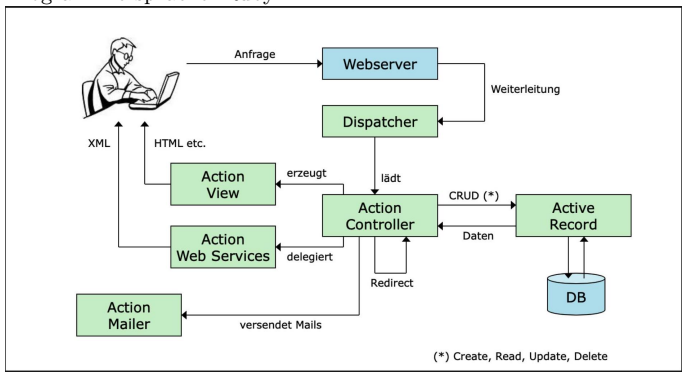
- bilden die Benutzerschnittstelle (z.B. HTML/CSS)
- können Models überwachen, kommunizieren aber normalerweise nicht direkt mit innen

## Controllers

- verarbeiten Eingaben (z.B. Clicks) und aktualisieren Models

## RUBY ON RAILS

- Serverseitiges Framework, basierend auf MVC
- Programmiersprache: Ruby




## Convention over Configuration

https://rubyonrails.org

## FOKUS AUF DIE CLIENT-SEITE

- Programmlogik Richtung Client verschoben
- Zunehmend komplexe User Interfaces
- Asynchrone Serveranfragen, z.B. mit Fetch
- Gute Architektur der Client-App wesentlich
- Diverse Frameworks und Bibliotheken zu diesem Zweck

## SINGLE PAGE APPS (SPAs)

- Neuladen von Seiten vermeiden
- Inhalte dynamisch nachgeladen (Ajax, REST)
- Kommunikation mit Server im Hintergrund
- Ul reagiert schneller (Usability)

## ÜBERSICHT

- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- JSX und SJDON
- Eigene Bibliothek: SuiWeb

## DOM-SCRIPTING

- Zahlreiche Funktionen und Attribute verfügbar
- Programme werden schnell unübersichtlich
- Gesucht: geeignete Abstraktionen

## AUFGABE

- Zum Vergleich der verschiedenen Ansätze
- Liste aus einem Array erzeugen

```
/* gegeben: */
let data = ["Maria", "Hans", "Eva", "Peter"]
```

```
<!-- DOM-Struktur entsprechend folgendem Markup aufzubauen: -->
<ul>
    <li>Maria</li>
    <li>Hans</li>
    <li>Eva</li>
    <li>Peter</li>
</ul>
```

## DOM-SCRIPTING

```
function List (data) {
    let node = document.createElement("ul")
    for (let item of data) {
        let elem = document.createElement("li")
        let elemText = document.createTextNode(item)
        elem.appendChild(elemText)
        node.appendChild(elem)
    }
    return node
}
```

- Erste Abstraktion: Listen-Komponente
- Basierend auf DOM-Funktionen

## DOM-SCRIPTING

```
function init () {
    let app = document.querySelector(".app")
    let data = ["Maria", "Hans", "Eva", "Peter"]
    render(List(data), app)
}
function render (tree, elem) {
    while (elem.firstChild) { elem.removeChild(elem.firstChild) }
    elem.appendChild(tree)
}
```

## DOM-SCRIPTING VERBESSERT

```
function elt (type, attrs, ...children) {
    let node = document.createElement(type)
    Object.keys(attrs).forEach(key => {
        node.setAttribute(key, attrs[key])
    })
    for (let child of children) {
        if (typeof child != "string") node.appendChild(child)
        else node.appendChild(document.createTextNode(child))
    }
    return node
}
```

## DOM-SCRIPTING VERBESSERT

- Damit vereinfachte List-Komponente möglich
- DOM-Funktionen in einer Funktion elt gekappselt

```
function List (data) {
    return elt("ul", {}, ...data.map(item => elt("li", {}, item)))
}
```

## JQUERY

```
function List (data) {
    return $("<ul>").append(...data.map(item => $("<li>").text(item)))
}
function render (tree, elem) {
    while (elem.firstChild) { elem.removeChild(elem.firstChild) }
    $(elem).append(tree)
}
```

- List gibt nun ein jQuery-Objekt zurück
- Daher ist eine kleine Anpassung an render erforderlich

## WEB COMPONENTS

- Möglichkeit, eigene Elemente zu definieren
- Implementiert mit HTML, CSS und JavaScript
- Implementierung im Shadow DOM verstecken

```
<custom-progress-bar class="size">
<custom-progress-bar value="25">
<script>
    document.querySelector('.size').progress = 75;

\section*{REACT.JS}

const List = ({data}) => (

{ data.map(item => ({item})) }

)
const root = createRoot(document.getElementById('app'))
root.render(
<List data={["Maria", "Hans", Ëva", "Peter"]} />
)

- XML-Syntax in JavaScript: JSX
- Muss zu JavaScript übersetzt werden
- https://reactjs.org

\section*{VUE.JS}

https://vuejs.org

var app4 = new Vue({
el: '#app',
data: {
items: [
{ text: 'Learn JavaScript' },
{ text: 'Learn Vue' },
{ text: 'Build something awesome' }
]
}
})



\section*{ÜBERSICHT}
- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- JSX und SJDON
- Eigene Bibliothek: SuiWeb

\section*{EIGENE BIBLIOTHEK}
- Ziel: eigene kleine Bibliothek entwickeln
- Ideen von React.js als Grundlage
- Ist dieser und den folgenden Lektionen schrittweise aufgebaut
- Wir nennen es:
}\author{
SuiWeb \\ Simple User Interface Toolkit for Web Exercises
}

\section*{EIGENE BIBLIOTHEK: MERKMALE}
![](https://cdn.mathpix.com/cropped/2025_01_02_22162ee5453ad023032
- Komponentenbasiert
- Also: User Interface aus Komponenten zusammengesetzt
- Zum Beispiel:

Komponente ArticleList

\section*{EIGENE BIBLIOTHEK: MERKMALE}
![](https://cdn.mathpix.com/cropped/2025_01_02_22162ee5453ad023032
- Datengesteuert
- Input: Daten der Applikation
- Output: DOM-Struktur für Browser

\section*{(data) =>}
![](https://cdn.mathpix.com/cropped/2025_01_02_22162ee5453ad023032

\section*{NOTATION FÜR KOMPONENTEN}
- Gesucht: Notation zum Beschreiben von Komponenten
- Ziel: möglichst deklarativ
- Also nicht: imperativen JavaScript- oder jQuery-Code, der DOM ma
- Verschiedene Möglichkeiten, z.B.
- JSX: in React.js verwendet
- SJDON: eigene Notation

JSX
const Hello = () => (
Hello World
)

- Von React-Komponenten verwendete Syntax
- Komponente beschreibt DOM-Struktur mittels JSX
- HTML-Markup gemischt mit eigenen Tags
- JSX = JavaScript XML
(oder: JavaScript Syntax Extension?)

\section*{JSX INS DOM ABBILDEN}

const domNode = document.getElementById('app')
const root = createRoot(domNode)
root.render()

- Root zum Rendern der Komponente anlegen
- Methode render aufrufen mit Code der gerendert werden soll

\section*{JSX}
- Problem: das ist kein JavaScript-Code
- Sondern: JavaScript-Code mit XML-Teilen
- Muss erst in JavaScript-Code übersetzt werden (Transpiler)
- Browser erhält pures JavaScript
![](https://cdn.mathpix.com/cropped/2025_01_02_22162ee5453ad023032

\section*{JSX: HTML-ELEMENTE}
- HTML-Elemente als vordefinierte Komponenten
- Somit können beliebige HTML-Elemente in Komponenten verwendet we
root.render(A Header)
```

```
\section*{JSX: HTML-ELEMENTE}
- HTML-Tags in Kleinbuchstaben
- Eigene Komponenten mit grossen Anfangsbuchstaben
- HTML-Elemente können die üblichen Attribute haben
- Wenige Ausnahmen, z.B.:
class-Attribut heisst className in JSX

\section*{JSX: KOMPONENTEN}

1 const MyComponent = () => (

My Component Content in my component...
)
root.render(

10 )


\section*{JSX: KOMPONENTEN}

const List = ({data}) => (

{ data.map(item => ({item})) }

)
root.render(
<List data={["Maria", "Hans", Ëva", "Peter"]} />
)

- JavaScript in JSX in \{... \}

\section*{JSX: KOMPONENTEN}
- Funktionen, welche JSX-Code zurückgeben
- Neue Komponente kann dann als Tag im JSX benutzt werden
- Üblicherweise werden Komponenten in eigenen Modulen implementiert und per require importiert

\section*{SJDON}
- Alternative zu JSX, eigene Notation
- SJDON - Simple JavaScript DOM Notation
- Bezeichnung aus einer Semesterendprüfung in WWD (WebPublishing und Webdesign, C. Bürkert, 2011 an der ZHAW)
4. JavaScript-Datenstrukturen, JSON, PHP (12 Punkte)

In einer Ajax-Anwendung soll HTML-Code in einfachen JavaScript-Datenstrukturen aufgebaut und manipuliert werden. Diese können dann im JSON-Format an den Server übertragen und zum Beispiel in einer

Die Notation - nennen wir sie SJDON (Simple JavaScript DOM Notation)

Die Notation - nennen wir sie SJDON (Simple JavaScript DOM Notation) - sei wie folgt definiert:
- Ein Textknoten ist einfach der String mit dem Text.
- Ein Elementknoten ist ein Array, das als erstes den Elementnamen als String enthält und anschliessend die Kindelemente (Text- oder Elementknoten, in der gewünschten Reihenfolge) und Attributbesc
- Attributbeschreibungen sind Objekte deren Attribute und Werte direkt den Attributen und Werten der HTML-Elements entsprechen. Alle Attribute des Elements können in einem Objekt zusammengefasst c
![](https://cdn.mathpix.com/cropped/2025_01_02_22162ee5453ad023032 8g SB.jpg?height=519&width=2313&top_left_y=1119&top_left_x=647)

\section*{VERGLEICH}

/* JSX const element = (
Hello World from SuiWeb )
const element =
["div", {style: "background:salmon"},
["h1", "Hello World"],
["h2", {style: "text-align:right"}, "from SuiWeb"] ]

\section*{ELEMENTE}
- Ein Element wird als Array repräsentiert
- Das erste Element ist der Elementknoten
- String: DOM-Knoten mit diesem Typ
- Funktion: Selbst definierte Komponente

["br"] /* br-Element /
*[ül", ["li", ëins"], ["li", ßwei"]]* / Liste mit zwei Items /
*[App, {name: SSuiWeb"}]* / Funktionskomponente */


\section*{ATTRIBUTE}
- Als Objekte repräsentiert
- Irgendwo im Array (ausser ganz vorne)
- Mehrere solcher Objekte werden zusammengeführt

/* mit style-Attribut, Reihenfolge egal */
["p", {style: "text-align:right"}, "Hello world"]
["p", "Hello world", {style: "text-align:right"}]


\section*{FUNKTIONEN}
- Funktion liefert SJDON-Ausdruck
- Kein \{ . . . \} für JavaScript wie in JSX nötig

const App = ({name}) =>
["h1", "Hi ", name]
const element =
[App, {name: SSuiWeb"}]


\section*{BEISPIEL: LISTENKOMPONENTE}

const MyList = ({items}) =>
[ül", items.map((item) => ["li", item]) ]
const element =
[MyList, {items: ["milk", "bread", ßugar"]}]

- JavaScript-Ausdruck generiert Kind-Elemente für ul
- Kein Problem, JavaScript-Ausdrücke einzufügen SJDON is pure JavaScript

\section*{ÜBERSICHT}
- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
JSX und SJDON
- Eigene Bibliothek: SuiWeb

\section*{ZIEL}
- Bau einer kleinen Web-Bibliothek
- Ausgerichtet an den Ideen von React
- Komponenten in JSX oder SJDON

\author{
Motto: \\ Keep it simple
}

\section*{SuiWeb}
- Simple User Interface Toolkit for Web Exercises
- Kein Mega-Framework
- Keine "full-stack"-Lösung
- Daten steuern Ausgabe der Komponenten
- Komponenten können einen Zustand haben

\section*{KEIN TWO-WAY-BINDING}
![](https://cdn.mathpix.com/cropped/2025_01_02_22162ee5453ad023032
- Ul-Elemente nicht bidirektional mit Model-Daten verbunden
- Daten werden verwendet, um View zu generieren
- Benutzerinteraktionen bewirken ggf. Anpassungen am Model
- Dann wird die View erneut aus den Daten generiert

\section*{AUSBLICK}
- Schrittweiser Aufbau von SuiWeb
- Beispiele im Praktikum

Wichtiger Hinweis: React.js ist ein bekanntes und verbreitetes Fra


\subsection{UI Implementierung}

\begin{itemize}
    \item Interne Repräsentation und das DOM
    \item Komponenten und Properties
    \item Darstellung von Komponenten
    \item Defaults und weitere Beispiele
  \end{itemize}

  \section*{ÜBERSICHT}
  \begin{itemize}
    \item Interne Repräsentation und das DOM
    \item Komponenten und Properties
    \item Darstellung von Komponenten
    \item Defaults und weitere Beispiele
  \end{itemize}

  \section*{RÜCKBLICK}
  \begin{itemize}
    \item Ziel: eigene kleine Bibliothek entwickeln
    \item Komponentenbasiert und datengesteuert
    \item An Ideen von React.js und ähnlicher Systeme orientiert
    \item Motto: „Keep it simple!"
    \item Bezeichnung:
  \end{itemize}

  \section*{SuiWeb}
  Simple User Interface Toolkit for Web Exercises

  \section*{RÜCKBLICK}
  \begin{itemize}
    \item Notation für den Aufbau der Komponenten
    \item JSX in React is verwendet
    \item SJDON: eigene Notation
    \item SuiWeb soll beide Varianten unterstützen
  \end{itemize}

  \begin{verbatim}
  // jsx
  const element = (<h1 title="foo">Hello</h1>)
  // sjdon
  const element = ["h1", {title: "foo"}, "Hello"]
```

## ANSTEHENDE AUFGABEN

- Interne Repräsentation der Komponenten
- Konvertierung von JSX und SJDON in diese Repräsentation
- Abbildung interne Repräsentation ins DOM
- Daten steuern Komponenten: Properties
- Hierarchie von Komponenten
- Komponenten mit Zustand

Anregungen und Code-Ausschnitte aus:
Rodrigo Pombo: Build your own React
https://pomb.us/build-your-own-react/
Zachary Lee: Build Your Own React.js in 400 Lines of Code
https://webdeveloper.beehiiv.com/p/build-react-400-lines-code

## AUSGANGSPUNKT

```
// jsx
/** @jsx createElement */
const element = (<h1 title="foo">Hello</h1>)
// jsx babel output (React < 17)
const element = createElement(
    "h1",
    { title: "foo" },
    "Hello"
)
// sjdon
const element = ["h1", {title: "foo"}, "Hello"]
```

## INTERNE REPRÄSENTATION

```
// jsx babel output
const element = createElement(
    "h1",
    { title: "foo" },
    "Hello"
)


// internal representation
const element = {
    type: "h1",
    props: {
        title: "foo",
        children: ["Hello"],
    },
}
```

## INTERNE REPRÄSENTATION

```
{
    type: "h1",
    props: {
        title: "foo",
        children: ["Hello"], /* noch anzupassen */
    },
}
```

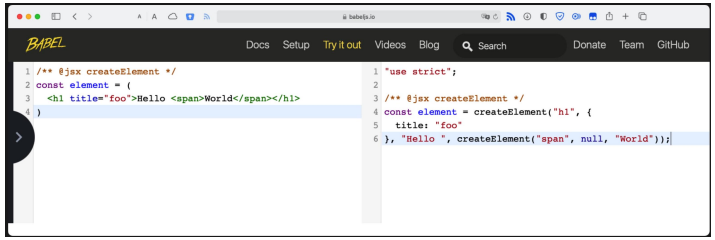
- Element: Objekt mit zwei Attributen, type und props
- type: Name des Elements ("body", "h1", ...)
- props: Attribute des Elements
- props.children: Kindelemente (Array)

## TEXT-ELEMENT

```
{
    type: "TEXT_ELEMENT",
    props: {
        nodeValue: "Hello",
        children: [],
    },
}
```

- Aufbau analog zu anderen Elementen
- Spezieller Typ: "TEXT_eLEMENT"

## VERSCHACHTELTE ELEMENTE




- Mehrere Kindelemente: ab drittem Argument von createElement
- Verschachtelte Elemente: rekursive Aufrufe von createElement

## KONVERTIERUNG VON JSX

```
function createElement (type, props,
                                ...children) {
    return {
        type,
        props: {
            ...props,
            children: children.map(child =>
                typeof child === "object"
                    ? child
                    : createTextElement(child)
            ),
    },
    }
}


function createTextElement (text) {
    return {
        type: "TEXT_ELEMENT",
        props: {
            nodeValue: text,
            children: [],
        },
    }
}
```

## CREATEELEMENT: BEISPIEL

```
// <div>Hello<br></div>
createElement("div", null, "Hello", createElement("br", null))
// returns
{
    type: 'div',
    props: {
        children: [
            {
                type: 'TEXT_ELEMENT',
                    props: { nodeValue: 'Hello', children: [] }
            },
            { type: 'br', props: { children: [] } }
        ]
    }
}
```

## KONVERTIERUNG VON SJDON

```
function parseSJDON ([type, ...rest]) {
    const isObj = (obj) => typeof(obj)==='object' && !Array.isAr
    const children = rest.filter(item => !isObj(item))
    return createElement(type,
        Object.assign({}, ...rest.filter(isObj)),
        ...children.map(ch => Array.isArray(ch) ? parseSJDON(ch)
    )
}
```

- Abbildung auf createElement-Funktion
- Attribute in einem Objekt zusammengeführt
- Kindelemente bei Bedarf (Array) ebenfalls geparst

## ZWISCHENSTAND

- Einheitliche Repräsentation für Elemente unabhängig von der ursprünglichen Syntax (JSX or SJDON)
- Baumstruktur von Elementen
- Text-Elemente mit leerem Array children
- DOM-Fragment im Speicher repräsentiert (virtuelles DOM?)

Zu tun:

- Abbildung der Baumstruktur ins DOM

## RENDER TO DOM

```
function render (element, container) {
    /* create DOM node */
    const dom =
        element.type == "TEXT_ELEMENT"
            ? document.createTextNode("")
            : document.createElement(element.type)
    /* assign the element props */
    const isProperty = key => key !== "children"
    Object.keys(element.props)
        .filter(isProperty)
        .forEach(name => { dom[name] = element.props[name] })
    /* render children */
    element.props.children.forEach(child => render(child, dom))
    /* add node to container */
    container.appendChild(dom)
}
```

## HTML-ELEMENTE

- Komponenten können HTML-Elemente verwenden
- Tagnamen in Kleinbuchstaben
- Gross-/Kleinschreibung ist relevant

- Übliche Attribute für HTML-Elemente möglich
- Wenig Ausnahmen: className statt class

## BEISPIEL


```
import { render } from "./lib/suiweb-1.1.js"
const element =
    ["div", {style: "background:salmon"},
        ["h1", "Hello World"],
        ["h2", {style: "text-align:right"}, "from SuiWeb"] ]
const container = document.getElementById("root")
render(element, container)
```

## ZWISCHENSTAND

- Interne Struktur aufbauen
- Ins DOM rendern

## ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- Darstellung von Komponenten
- Defaults und weitere Beispiele

## FUNKTIONSKOMPONENTEN

```
1 const App = (props) =>
    ["h1", "Hi ", props.name]
4 const element =
5 [App, {name: "foo"}]
```

- App ist eine Funktionskomponente
- Die zugehörige Repräsentation erzeugt keinen DOM-Knoten
- Ergebnis des Aufrufs liefert auszugebende Struktur
- Konvention: eigene Komponenten mit grossen Anfangsbuchstaben

## PROBLEM

- Komponenten in JSX retournieren mittels createElement erzeugte interne Strukturen
- Unter SJDON liefern sie allerdings SJDON-Code, der nach Aufruf der Komponente noch geparst werden muss
- Abhilfe: SJDON-Komponenten erhalten ein Attribut sjdon, welches die Konvertierung (parseSJDON ) ergänzt
- Dieses Attribut lässt sich mit einer kleinen Hilfsfunktion anbringen

## SJDON-KONVERTIERUNG ERWEITERT

```
function useSJDON (...funcs) {
    for (let f of funcs) {
        const fres = (...args) => parseSJDON(f(...args))
        f.sjdon = fres
    }
}
```

- Kann für mehrere Komponentenfunktionen aufgerufen werden, indem sie als Argumente übergeben werden
- Diese werden um das sjdon-Attribut ergänzt

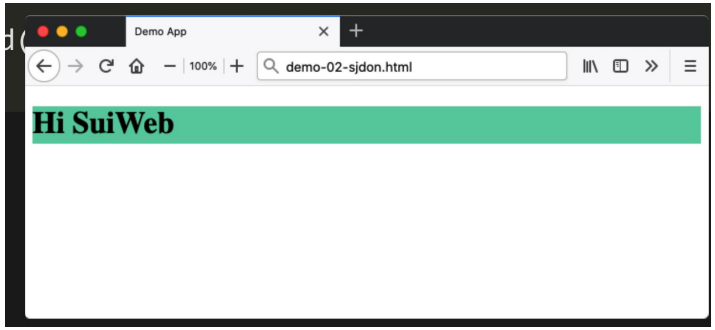
## FUNKTIONSKOMPONENTEN

- Funktion wird mit props-Objekt aufgerufen
- Ergebnis ggf. als SJDON geparst

```
switch (typeof type)
    case 'function': {
        let children
        if (typeof(type.sjdon) === 'function') {
            children = type.sjdon(props)
        } else {
            children = type(props)
        }
        reconcileChildren(...)
        break
    }
}
```

## BEISPIEL

```
const App = (props) =>
    ["h1", {style: "background: mediumaquamarine"}, "Hi ", props.name]
const element =
    [App, {name: "SuiWeb"}]
// notify SuiWeb that the App component returns SJDON
useSJDON(App)
const container = document.getElementById ©\bullet. Domonop
render(element, container)
demo-02-jsx.html demo-02-sjdon.html
```




## WERTE STEUERN UI-AUFBAU

```
const App = () => {
    const enabled = false
    const text = 'A Button'
    const placeholder = 'input value...'
    const size = 50
    return (
        ["section",
            ["button", {disabled: !enabled}, text],
            ["input", {placeholder, size, autofocus: true}] ]
    )
}
```

demo-03-values

## ARRAY ALS LISTE AUSGEBEN

```
const List = ({items}) =>
    ["ul", ...items.map((item) => ["li", item]) ]
const element =
    [List, {items: ["milk", "bread", "sugar"]}]
useSJDON(List)
```

- Die props werden als Argument übergeben
- Hier interessiert nur das Attribut items

demo-04-liste

## OBJEKT ALS TABELLE

```
const ObjTable = ({obj}) =>
    ["table", {style},
        ...Object.keys(obj).map((key) =>
        ["tr", ["td", key], ["td", obj[key]]])]
const style = {
    width: "8em",
    background: "lightblue",
}
const element =
    [ObjTable, {obj: {one: 1111, two: 2222, three: 3333}}]
demo-05-object
```

## VERSCHACHTELN VON ELEMENTEN

```
/* JSX */
<MySection>
    <MyButton>My Button Text</MyButton>
</MySection>
```

- Eigene Komponenten können verschachtelt werden
- MyButton ist mit seinem Inhalt in props.children von MySection enthalten

## VERSCHACHTELN VON ELEMENTEN

```
const MySection = ({children}) =>
    ["section", ["h2", "My Section"], ...children]
const MyButton = ({children}) =>
    ["button", ...children]
const element =
    [MySection, [MyButton, "My Button Text"]]
useSJDON(MyButton, MySection)
```

demo-06-nested

## TEILBÄUME WEITERGEBEN

```
const Main = ({header, name}) =>
    ["div",
```

```
        [...header, name],
        ["p", "Welcome to SuiWeb"] ]
const App = ({header}) =>
    [Main, {header, name: "web developers"}]
const element = [App, {header: ["h2", "Hello "]}]
useSJDON(App, Main)
```

demo-07-subtree

## ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- Darstellung von Komponenten
- Defaults und weitere Beispiele

## DARSTELLUNG

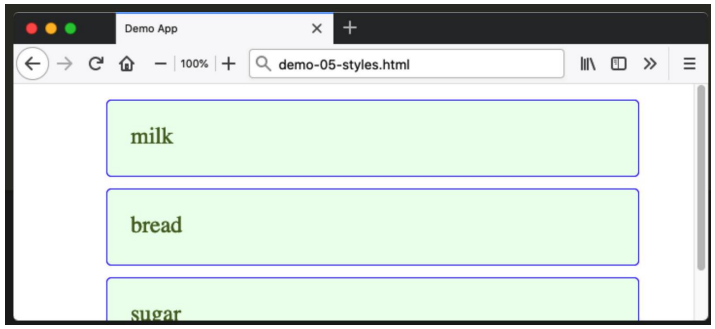
- Komponenten müssen ggf. mehrere Styles mischen können
- Neben Default-Darstellung auch via props eingespeist
- Daher verschiedene Varianten vorgesehen:
- CSS-Stil als String
- Objekt mit Stilangaben
- Array mit Stil-Objekten

## DARSTELLUNG

```
function combineStyles (styles) {
    let styleObj = {}
    if (typeof(styles)=="string") return styles
    else if (Array.isArray(styles)) styleObj = Object.assign({}, ...styles)
    else if (typeof(styles)=="object") styleObj = styles
    else return ""
    let style = ""
    for (const prop in styleObj) {
        style += prop + ":" + styleObj[prop] + ";"
    }
    return style.replace(/([a-z])([A-Z])/g, "$1-$2").toLowerCase()
}
```

## BEISPIEL

```
const StyledList = ({items}) => {
    let style = [styles.listitem, {color: "#556B2F"}]
    return (
        ["ul", ...items.map((item) => ["li", {style}, item]) ]
    )
}
const element =
    [StyledList, {items: ["milk", "bread", "sugar"]}]
const styles = {
    listitem: {
            padding: "1em",
            margin: "0.5em 2em",
            fontSize: "1.5em",
            ... }
}
```




## ÜBERSICHT

- Interne Repräsentation und das DOM
- Komponenten und Properties
- Darstellung von Komponenten
- Defaults und weitere Beispiele

## DEFAULT PROPERTIES

```
const App = () => (
        ["main",
            [MyButton, {disabled: true, text: 'Delete'}],
            [MyButton] ]
)
const MyButton = ({disabled=false, text='Button'}) => (
    ["button", disabled ? {disabled} : {}, text]
)
```

demo-09-defaultprops

## DEFAULT PROPERTIES

- Übergebene Properties überschreiben Defaults
- Selbst zu implementieren (ist einfach, s. Beispiel)
- In React.js können Defaults an Funktion gehängt werden: (in SuiWeb nicht umgesetzt, wäre aber möglich)

```
const MyButton = (props) => { ... }
MyButton.defaultProps = {
    text: 'My Button',
    disabled: false,
}
```

## WEITERES BEISPIEL

```
const MyButton = ({children, disabled=true}) =>
    ["button", {style: "background: khaki", disabled}, ...children]
const Header = ({name, children}) =>
    ["h2", "Hello ", name, ...children]
const App = (props) =>
    ["div",
        [Header, {name: props.name}, " and", ["br"], "web developers"],
        [MyButton, "Start", {disabled:false}],
        [MyButton, "Stop"] ]
useSJDON(App, Header, MyButton)
render([App, {name: "SuiWeb"}], container)
```

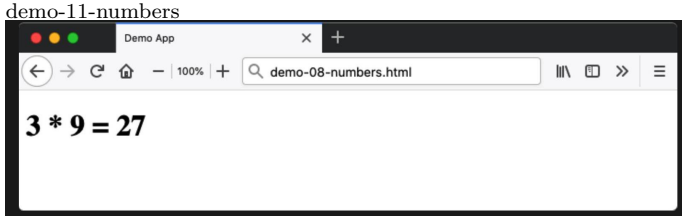
demo-10-children

## ZAHLEN IN PROPS

```
const App = ({num1, num2}) =>
    ["h1", num1, " * ", num2, " = ", num1*num2]
const element = [App, {num1: 3, num2: 9}]
```

- Beim Funktionsaufruf als Zahlen behandelt
- Beim Rendern in Textknoten abgelegt

demo-11-numbers




## AKTUELLER STAND

- Notationen, um Komponenten zu definieren: JSX, SJDON
- Funktionen zur Anzeige im Browser: render-Funktion
- Daten können Komponenten steuern: Argument props
- Ausserdem: Verarbeiten von Styles, Default-Properties
- Also: Ul-Aufbau mit Komponenten
- Was noch fehlt: Mutation, Zustand
  → nächste Woche :)

# UI 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:"
            "Count: " + state]
    )
}
const element = [Counter]
```

## STATE-HOOK: UMSETZUNG

- 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, useState, useSJDON } from "./lib/suiweb-1.1.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], ]
    )
}
```

demo-22-state

## 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

-. Demapop ×+

←→ C ? - 1 100% + + $Q_{\text{demo-09-state.entm}|}$ III

Count 16

Count 19

Count 22

## 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

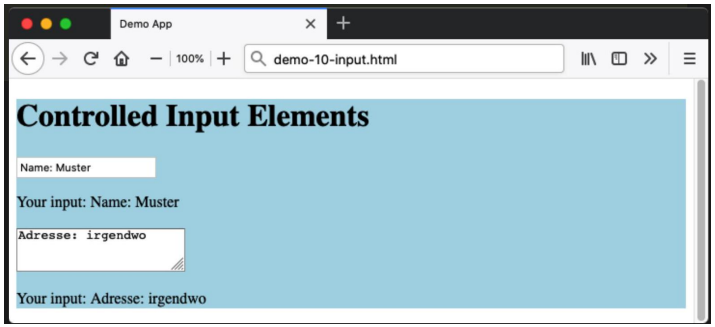
## KONTROLLIERTE EINGABE

- 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

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

5

## CONTAINER-KOMPONENTE

```
const MyContainer = () => {
    let initialState = { items: ["Fetching data..."] }
    let [state, setState] = useState(initialState)
    if (state === initialState) {
        fetchData()
            .then(items => setState(() => ({ items })))
    }
    return (
        [MyList, state]
```

```
    )
}
```

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 })))
    }, []) ...
}
```

- React.js-Beispiel
- Hier ist ein weiteres Beispiel:
  https://suiweb.github.io/docs/tutorial/4-hooks#indexjs

## 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

- Artikel können hinzugefügt werden
- Artikel: Titel, Zusammenfassung
- Klick auf den Titel: Inhalt einund ausblenden
- Klick auf X: Artikel löschen

## AUFTEILUNG IN KOMPONENTEN

### Articles

- Article 1$\underline{x}$

Article 1 Summary

- Article 2$\underline{X}$
- Article 3 X
- Article 4 X

### ArticleList

### ArticleItem

## 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: '',
    }
}
```

## ],

```
title:
summary: ',
```

- Array von Artikeln
- Generierte IDs
- title und summary 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 Artic
        ]
    )
}
```

## 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

## IMPERATIVER ANSATZ

Ergänze alle Code-Teile in denen die Artikelliste erweitert oder verkleinert wird wie folgt:

- Wenn der letzte Artikel gelöscht wird, entferne <ul></ul> und füge einen Button für den Netzwerkzugriff ein
- Wenn der erste Artikel eingefügt wird, entferne den Button und füge ein 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/
Zachary Lee: Build Your Own React.js in 400 Lines of Code
https://webdeveloper.beehiiv.com/p/build-react-400-lines-code
```

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

## 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
}
```

## 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

[div, [h1, p, a], h2]

- 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 = nul1
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 requestIdleCall.back

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

## 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

## 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

## Wrap-up

## ÜBERSICHT

- Von SuiWeb zu React.js
- Ausblick: Weitere Themen rund ums Web
- Abschluss, Feedback
- Anhang: Themenliste WBE

## ÜBERSICHT

- Von SuiWeb zu React.js
- Ausblick: Weitere Themen rund ums Web
- Abschluss, Feedback
- Anhang: Themenliste WBE

## SUIWEB

- SuiWeb ist eine experimentelle Bibliothek
- Angelehnt an die Ideen von React.js
- Es ist Zeit, React.js noch etwas anzusehen

## REACT

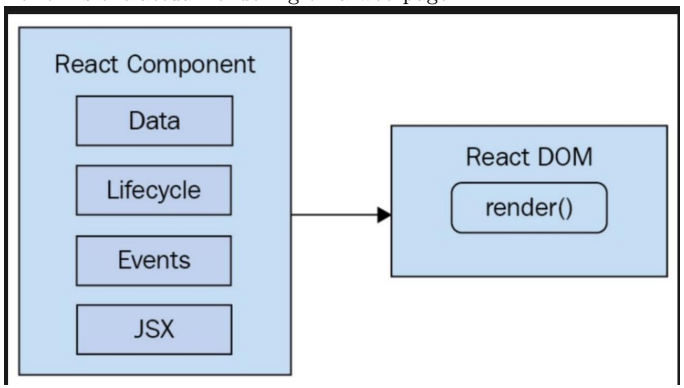
„A JavaScript library for building user interfaces"

- Declarative
- Component-Based
- Learn Once, Write Anywhere

Facebook, Instagram
2013 vorgestellt
https://react.dev

## ZWEI TEILE

- React DOM
- Performs the actual rendering on a web page




- React Component API
- Data to be rendered
- Lifecycle methods
- Events: respond to user interactions
- JSX: syntax used to describe UI structures

## KOMPONENTEN UND KLASSEN

```
// ES5
var HelloComponent = React.createClass({
    render: function() {
        return <div>Hello {this.props.name}</div>
    }
})
// ES6
class HelloComponent extends React.Component {
    render() {
        return <div>Hello {this.props.name}</div>
    }
}
// Function Component
const HelloComponent = (props) => {
    return (<div>Hello {props.name}</div>)
}
```

## KOMPONENTEN

```
const MyComponent = () => (
    <section>
        <h1>My Component</h1>
        <List data={["Maria", "Hans", "Eva", "Peter"]} />
    </section>
)
const List = ({data}) => (
    <ul>
        { data.map(item => (<li key={item}>{item}</li>)) }
    </ul>
)
const root = createRoot(document.getElementById('app'))
root.render(
    <MyComponent />
)
```

## ZUSTAND

```
const Counter = () => {
    const [state, setState] = useState(1)
    const handler = () => setState(c => c + 1)
    return (
        <h1 onclick={handler} style={{userSelect:"none",cursor:'p
            Count {state}
        </h1>
    )
}
const root = createRoot(document.getElementById('counter'))
root.render(
    <Counter />
)
```

## PROPERTIES

```
const MyButton = (props) => {
    const { disabled, text } = props
    return (
            <button disabled={disabled}>{text}</button>
    )
}
const root = createRoot(document.getElementById('counter'))
root.render(
    <main>
        <MyButton text='My Button' disabled=true />
    </main>
)
```

## UND SONST

- Funktions- und Klassenkomponenten unterstützt
- Funktionskomponenten mit Hooks (u.a. State Hook)
- Diverse Optimierungen: virtuelles DOM, Fibers
- Entwicklertools, React Devtools
- Serverseitiges und clienseitiges Rendern
- Komponententechnologie auch für native iOS und Android Apps verwendbar (React Native)

## WAS IST NUN REACT?

- React bildet die View einer Applikation
- Nicht (nur) Framework, sondern in erster Linie Konzept
- Unterstützt das Organisieren von Vorlagen in Komponenten
- Das virtuelle DOM sorgt für schnelles Rendern

## POWER OF COMPONENTS

- Kleinere Einheiten entwickeln
- Weniger Abhängigkeiten
- Einfacher zu verstehen, zu pflegen, zu testen
- Komponentendesign: für genau eine Sache verantwortlich
- Zustand in wenigen Komponenten konzentrieren

## 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

## Existing Frameworks Influenced: All of them

- Angular komplett überarbeitet
- Neue Frameworks entstanden: Vue.js, Svelte, ...
- Entwicklung nativer Mobil-Apps: SwiftUI, Compose
- ...

## ÜBERSICHT

- Von SuiWeb zu React.js
- Ausblick: Weitere Themen rund ums Web
- Abschluss, Feedback
- Anhang: Themenliste WBE

## HAUPTTHEMEN IN WBE

- JavaScript die Sprache (und Node.js)
- JavaScript im Browser
- Client-seitige Web-Apps

## WEITERE THEMEN RUND UMS WEB

Rund ums Web gibt es noch viele spannende Themen...
Ein paar Anregungen sind auf den folgenden Slides zusammengestellt

(ohne Anspruch auf Vollständigkeit)

## HTML und CSS

- Grundlagen: als Vorkenntnisse für WBE
- Skript im Vorbereitungskurs (Moodle)
- Diverse Tutorials (ein paar im Kurs verlinkt)
  ▷ Vorbereitungskurs WBE

## Web-Apps für Mobilgeräte

- Layout für verschiedene Devices (Smartphones, ...)
- Responsives Webdesign (u.a. Bilder)
- Web-APIs für Gerätesensoren
- Apps basierend auf React und Ionic
- React Native / Expo
  ▷ MOBA

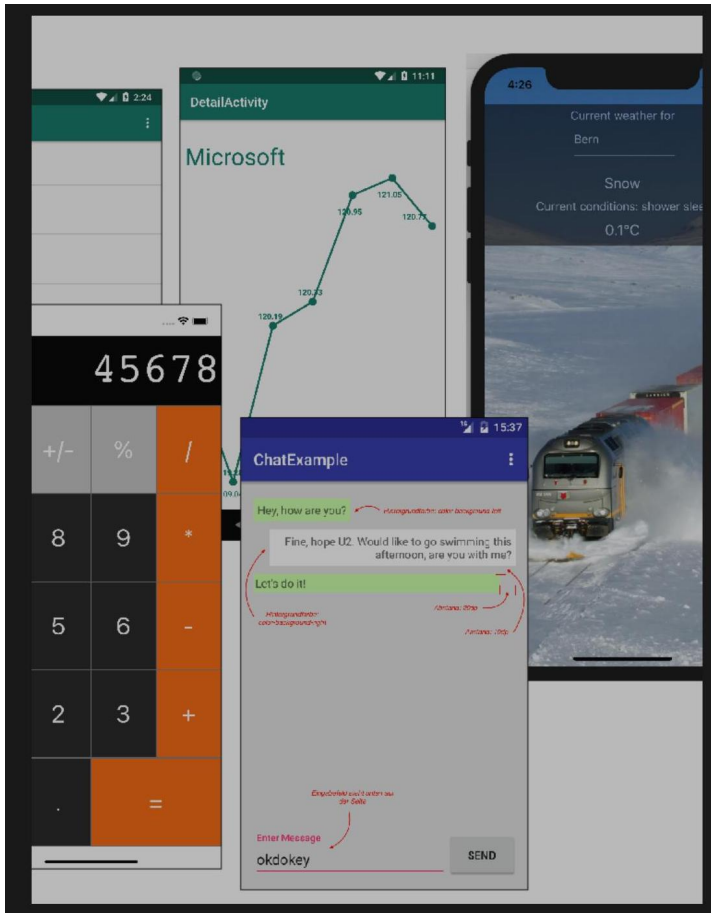
## Mobile Applications (MOBA1/MOBA2)

- Mobile Layouts, CSS Flexbox
- Device APIs, Sensoren
- Web Components, React, Ionic
- React Native

und:

- Android native (Kotlin, Compose)
- iOS native (Swift, SwiftUI)

Info ▷ H. Stormer (stme), G. Burkert (bkrt)




## Apps mit Webtechnologien

- Desktop-Applikationen mit Web-Technologien https://www.electronjs.org https://nwjs.io
- Basis für Applikationen wie VSCode
- Diverse weitere Frameworks in diesem Bereich
- Mobil-Applikationen mit Web-Technologien https://cordova.apache.org https://capacitorjs.com

## WebAssembly (WASM)

- Bytecode zur Ausführung in Webbrowsern
- Ziel: höhere Performanz für Web-Applikationen
- Verschiedene Programmiersprachen kompilieren zu WASM
- Erste Version funktioniert in aktuellen Browsern bereits https://webassembly.org
  ▷ PSPP

## JavaScript-Alternativen

- Werden nach JavaScript „kompiliert"
- TypeScript (Microsoft)
- statisches Typenkonzept
- ReScript (ehemals ReasonML)
- speziell für React-Ansatz geeignet
- funktionaler Ansatz, an OCaml angelehnt
- ClojureScript (Lisp-Dialekt)
- PSPP

## Funktionale Programmierung

- JavaScript ist eine Multiparadigmensprache
- Es eignet sich sehr gut für funktionale Programmierung (higher order functions, partial application, currying, ...)
- In WBE wird dieser Aspekt kaum thematisiert
- PSPP
  ▷ FUP

## Programmiersprachen und -Paradigmen (PSPP)

- Compiler, Bytecodes (WASM)
- Logische Programmierung (Prolog)
- Objektorientierte Programmierung (Smalltalk)
- Funktionale Programmierung (Lisp, Python)
- und: Modulkonzept, Scriptsprachen, Typenkonzepte

Info ▷ G. Burkert (bkrt), K. Rege (rege)

## Design, Usability, ...

- Grafische Gestaltung
- Gestaltungsprinzipien
- Farbenlehre
- Typografie
- Usability
- Barrierefreiheit
  ▷ Vorbereitungskurs WBE (design-usability.pdf)

## Zurück zu JavaScript ...

## DOUGLAS CROCKFORD

## Autor von: JavaScript: The Good Parts

„The idea of putting powerful functions and dynamic objects in the same language was just brilliant. That's the thing that makes JavaScript interesting."

FullStack London 2018
https://www.youtube.com/watch?v=8oGCyfautKo
„My advice to everybody who wants to be a better programmer is to learn more languages. A good programming language should teach you. And in my career the language which has taught me the most was JavaScript."

The Better Parts. JS Fest 2018
https://www.youtube.com/watch?v=XFTOG895C7c

## ÜBERSICHT

- Von SuiWeb zu React.js
- Ausblick: Weitere Themen rund ums Web
- Abschluss, Feedback
- Anhang: Themenliste WBE

## ÜBERBLICK WBE

| Woche | Thema |
|---|---|
| 1 | Einführung, Administratives, das Web im Überblick |
| 2 | JavaScript: Grundlagen |
| 3 | JavaScript: Objekte und Arrays |
| 4 | JavaScript: Funktionen |
| 5 | JavaScript: Prototypen von Objekten |
| 6 | JavaScript: Asynchrones Programmieren |
| 7 | JavaScript: Webserver |
| 8 – 9 | Browser-Technologien: JavaScript im Browser |
| 10 | Browser-Technologien: Client-Server-Interaktion |
| 11 – 13 | Ul-Bibliothek: Komponenten, Implementierung, Einsatz |
| 14 | Abschluss: React, Feedback |

## WBE-ZIELE

In erster Linie:
Solide Kenntnisse in grundlegenden Web-Technologien, speziell JavaScript, denn dies ist die Programmiersprache des Web.

### Grundlagen:

HTML und CSS als Basistechnologien des Web muss man natürlich auch kennen, um mit Webtechnologien entwickeln zu können.

### Ausserdem:

Einen Überblick erhalten über einen für heutige Anforderungen relevanten Ausschnitt aus dem riesigen Gebiet der Web-Technologien.

## ALLGEMEINE BETRACHTUNG

- Themen, welche vertieft behandelt wurden

Grösserer Block in mindestens einer Vorlesung, also nicht nur zwei bis drei Slides dazu, in der Regel auch im Praktikum thematisiert

- Themen welche nebenbei behandelt wurden

Im Sinne von: das gibt's auch, sollte man kennen, wenn man sich mit Webtechnologien beschäftigt, Einarbeitung nach Bedarf

## ALLGEMEINE BETRACHTUNG

- Themen, welche vertieft behandelt wurden

Mit diesen Themen sollte man sich auskennen (ein paar mehr Details im Anhang)

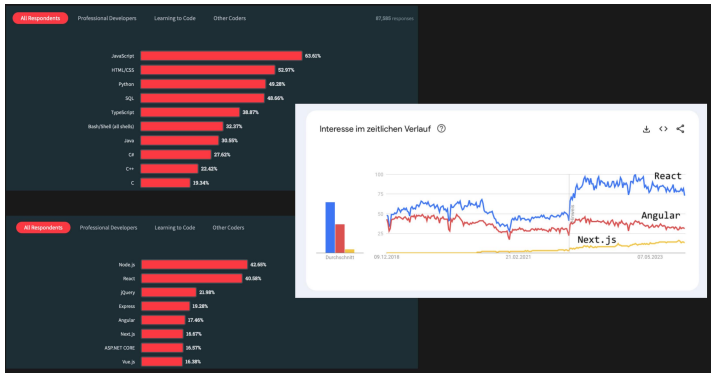
- Themen welche nebenbei behandelt wurden

Hier sollte man wissen, worum es geht, dazu gehören ein paar wesentliche Merkmale der Technologie, des Frameworks oder der Idee, aber Details sind hier nicht das Ziel

## BITTE UM FEEDBACK

- Inhalte?
- Stoffumfang?
- Praktika?
- Art der Durchführung?

## STACKOVERFLOW SURVEY, GOOGLE TRENDS




TIOBE Index for December 2023

| Dec 2023 | Dec 2022 | Change | Programming Language |
|---|---|---|---|
| 1 | 1 | | |
| 2 | 2 | | |
| 3 | 3 | | |
| 4 | 4 | | |
| 5 | 5 | | |
| 6 | 7 | ∧ | JS |
| 7 | 10 | ∧ | |
| 15 | 23 | Λ | |
| 16 | 16 | | |
| 17 | 15 | ✓ | |
| 18 | 20 | ∧ | (19) |
| 38 | | cript | |
| 39 | | | |
| 40 | | | |
| 41 | M | | |

## Schöne Feiertage

## ÜBERSICHT

- Von SuiWeb zu React.js
- Ausblick: Weitere Themen rund ums Web
- Abschluss, Feedback
- Anhang: Themenliste WBE

## ÜBERBLICK

- Ganzes Thema wichtig
- inklusive Unterthema
- Thema teilweise wichtig
- zum Beispiel dieses Unterthema
- Unterthema: Überblick genügt
- Überblick genügt
- Unterthema ebenso

## GRUNDLAGEN: HTML & CSS

- Markup und HTML
- Konzept von Markup verstehen
- Eckpunkte der Entwicklung von HTML kennen
- Aufbau eines HTML-Dokuments
- Grundbegriffe: Element, Tag, Attribut
- Grundlegende Elemente: html, head, title, meta, body, p, div, span, p, img, h1, ..., ul, ol, li
- Weitere Elemente: header, article, ...
- Attribute: contenteditable, data-
- Bild- und Grafikformate, SVG

## GRUNDLAGEN: HTML & CSS

- Darstellung mit CSS
- CSS mit HTML verbinden, CSS-Regeln
- Selektoren
- Einige Eigenschaften, Grössen- und Farbangaben (am besten an Beispielen und Aufgaben orientieren)
- Schriften laden, Transitionen, Transformationen, Animationen
- Weitere Eigenschaften
- Werkzeuge und Hilfsmittel

## GRUNDLAGEN: HTML & CSS

- Das Box-Modell
- overflow, width, height, margin, padding, border
- border-radius, color, background-color
- Farbverläufe, Sprites
- Positionierung und fliessende Boxen
- position, float, clear, display (block, inline, none)

## 1. DAS WEB

- Internet und WWW
- Einige Eckpunkte der Entwicklung kennen
- Client-Server-Architektur
- Konzepte und wesentliche Tools kennen
- User Agents, Webserver
- URI/URL, IP-Adresse, Domain-Name
- Grundzüge des HTTP-Protokolls

## 1. DAS WEB

- Die Sprachen des Web: HTML, CSS, JavaScript
- Vorkenntnisse / Vorkurs
- Web-Standards und APIs
- W3C und WHATWG kennen
- clientseitige vs. serverseitige Technologien

## 2. JAVASCRIPT GRUNDLAGEN

- JavaScript und Node.js
- Einige Eckpunkte der Entwicklung
- Node.js als JavaScript-Laufzeitumgebung
- Node.js Einsatz, REPL, NPM
- console.log
- Werte, Typen, und Operatoren
- Zahlen, typeof, Strings, logische Ausdrücke, ...

## 2. JAVASCRIPT GRUNDLAGEN

- Programmstruktur
- Ausdruck vs. Anweisung
- Syntax, Variablen, Kontrollstrukturen, Kommentare, ...
- Funktionen
- Überblick, mehr später

## 3. JS: OBJEKTE UND ARRAYS

- Objekte
- Objektliterale, Attribute, Methoden, ...
- Methoden von Object: assign, keys, values
- Spezielle Objekte: Arrays
- Array-Literale
- Schleifen über Arrays
- Array-Methoden: slice, concat, Array.isArray
- Weitere Methoden schaut man bei Bedarf nach

## 3. JS: OBJEKTE UND ARRAYS

- Werte- und Referenztypen
- Unterschied verstehen
- Wissen, welche Typen in JS Werte- und Referenztypen sind
- Vordefinierte Objekte, JSON
- Wichtigste vordefinierte Objekte kennen
- Methoden schaut man bei Bedarf nach
- JSON.stringify, JSON.parse

Zum vorletzten Punkt: Unterschied zwischen in eigenem Code verwenden und in bestehendem Code verstehen. Was ein "Hello World".indexOf(İl") bedeutet, sollte man sich schon vorstellen können.

## 4. JS: FUNKTIONEN

- Funktionen definieren
- Definition und Deklaration, Pfeilnotation
- Gültigkeitsbereiche
- Parameter von Funktionen
- Default-, Rest-Parameter, arguments
- Spread-Operator
- Arrays und Objekte destrukturieren
- Funktionen höherer Ordnung
- Arrays: forEach, filter, map, reduce

## 4. JS: FUNKTIONEN

- Closures
- Einsatz von Closures
- Pure Funktionen
- Funktionen dekorieren
- Funktionales Programmieren
- Mehr zu Node.js
- Konsole, Kommandozeilenargumente
- Module in JavaScript
- NPM, NPX

## 5. JS: PROTOTYPEN VON OBJEKTEN

- Prototypen und this
- Bedeutung von this je nach Aufruf
- Strict Mode
- call, apply, bind
- Prototyp eines Objekts, Object. create
- Weitere Methoden (getPrototypeOf, getOwnPropertyNames) schlägt man bei Bedarf nach

## 5. JS: PROTOTYPEN VON OBJEKTEN

- Konstruktoren und Vererbung
- Konstruktorfunktionen, new
- Prototypenkette
- Gewohntere Syntax: Klassen
- class, extends, constructor,...

- Test-Driven Development
- Konzept verstehen
- Jasmine einsetzen können

## 6. JS: ASYNCHRONES PROGRAMMIEREN

- File API
- Unterschied zwischen fs.readFileSync und fs.readFile
- Streams und weitere Methoden
- Reagieren auf Ereignisse
- Event Loop im Überblick
- Modul „events"
- Promises, Async/Await

## 7. JS: WEBSERVER

- Internet-Protokolle
- Internet-Protokoll-Stack
- Protokolle: FTP, SFTP, SSH
- Das HTTP-Protokoll
- Grundlagen des Protokolls
- HTTP-Methoden: GET, POST, PUT, PATCH, DELETE

## 7. JS: WEBSERVER

- Node.js Webserver
- Web-Server, -Client, Streams: Code lesen können
- Beispiel File-Server: Aufbau grob verstehen
- REST APIs
- Konzept verstehen
- Alternative GraphQL
- Express.js
- Für einfache Aufgaben verwenden können
- Reverse Proxy

## 8. BROWSER: JAVASCRIPT

- JavaScript im Browser
- Überblick, ES-Module
- Document Object Model
- Repräsentation im Speicher, Baumstruktur
- Verschiedene Knotentypen, Knoten anlegen
- Array-ähnliche Objekte, Array.from
- Attribute: HTML-Attribute, className, classList, style
- requestAnimationFrame
- Überblick, was möglich ist (Details kann man nachschlagen)
- DOM-Scripting-Code lesen können

## 8. BROWSER: JAVASCRIPT

- Vordefinierte Objekte
- Allgemeine Objekte und Browser-Objekte
- CSS und das DOM
- Layout-Angaben im DOM
- class und style

## 9. BROWSER: JAVASCRIPT

- Event Handling im Browser
- Events registrieren: window .addEventListener
- Event-Handler und Event-Objekt
- Event-Weiterleitung und Default-Verhalten
- Events: click, weitere Events
- Kleiner Exkurs: jQuery
- Bilder und Grafiken
- Weitere Browser-APIs
- WebStorage
- History, Geolocation, Workers

## 10. BROWSER: CLIENT-SERVER

- Formulare
- Element form mit Attributen method, action
- Elemente input, label mit wichtigen Attributen
- Mehr kann man bei Bedarf nachschlagen
- Daten mit GET und POST übertragen
- File-Input, GET und POST in Express
- Cookies, Sessions
- Konzept verstanden

## 10. BROWSER: CLIENT-SERVER

- Ajax und XMLHttpRequest
- Konzept verstanden
- Fetch API
- Verwenden von fetch (Promise)
- jQuery, Axios, CORS

## 11. UI-BIBLIOTHEK (1)

- Frameworks und Bibliotheken
- Unterschied, Eckpunkte der Entwicklung
- Model-View-Controller, Singe-Page Apps
- DOM-Scripting und Abstraktionen
- Verschiedene Ansätze im Überblick
- JSX und SJDON
- Vergleich der Notationen
- Eigene Bibliothek: SuiWeb
- Ziel, Vorgehen

## 12. UI-BIBLIOTHEK (2)

- Erste Schritte
- Interne Datenstruktur, createElement, render
- Ansatz verstehen, Code lesen können
- Komponenten und Properties
- Einsetzen können
- Details wie sie implementiert sind weniger wichtig
- Darstellung von Komponenten
- Defaults und weitere Beispiele

## 13. UI-BIBLIOTHEK (3)

- Zustand von Komponenten
- State-Hook, einsetzen können
- Kontrollierte Eingabe
- Details der Implementierung sind weniger wichtig
- Komponenten-Design
- Container-Componente
- Lifecycle-Methoden, Effect-Hook
- Aufteilen in Komponenten:

Beispiel nachvollziehen können

- Deklarativer vs. imperativer Ansatz

## 13. UI-BIBLIOTHEK (3)

- Ausblick: Optimierungsansätze
- Aufteilen in Arbeitsschritte, asynchrones Abarbeiten
- Render- und Commit-Phasen

## 14. ABSCHLUSS

- Von SuiWeb zu React.js
- Klassenkomponenten
- Weitere Konzepte
- Ausblick: Weitere Themen rund ums Web