

**WBE: UI-BIBLIOTHEK**

**TEIL 1: KOMPONENTEN**

# ÜBERSICHT

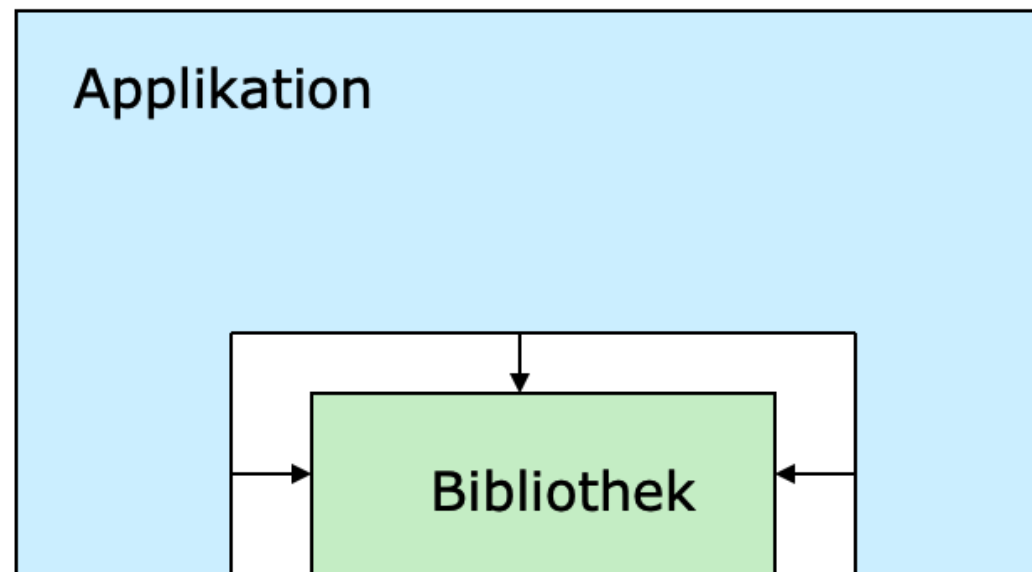
- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- Aufbau einer UI-Bibliothek

# ÜBERSICHT

- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- Aufbau einer UI-Bibliothek

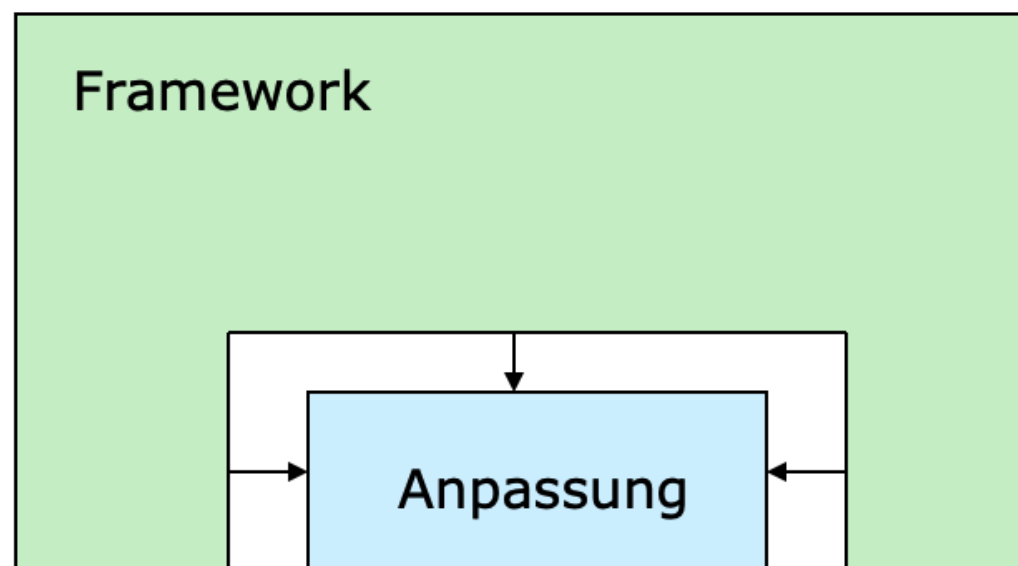
# BIBLIOTHEK

- 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 ihnen

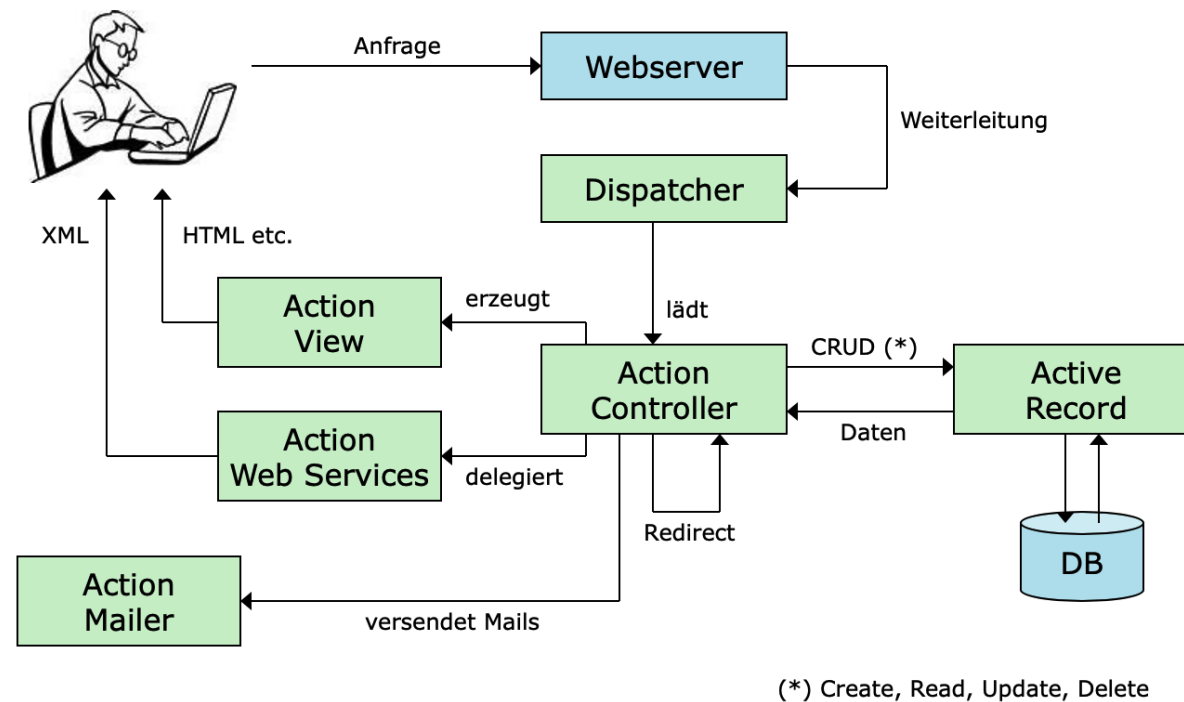
## Controllers

- 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
- UI reagiert schneller (Usability)

# ÜBERSICHT

- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- Aufbau einer UI-Bibliothek

# 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

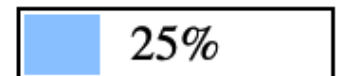
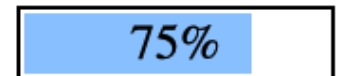
```
function List (data) {  
  return $("<ul>").append(...data.map(item => $("<li>").text(item)))  
}  
  
function render (tree, elem) {  
  while (elem.firstChild) { elem.removeChild(elem.firstChild) }  
  $(elem).append(tree)  
}
```

- `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;  
</script>
```



# REACT.JS

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

- XML-Syntax in JavaScript: JSX
- Muss zu JavaScript übersetzt werden
- <https://react.dev>

# VUE.JS

```
<div id="app">
  <ol>
    <li v-for="item in items">
      {{ item.text }}
    </li>
  </ol>
</div>
```

<https://vuejs.org>

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

# ÜBERSICHT

- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- Aufbau einer UI-Bibliothek

# AUFBAU EINER UI-BIBLIOTHEK

- Ziel: React-Ansatz für den Bau von UIs verstehen
- Dazu sinnvoll: Mini-React einmal selber bauen
- In dieser und den folgenden Lektionen einige Hinweise dazu
- Dabei ist auch eine kleine Bibliothek entstanden:

SuiWeb

Simple User Interface Toolkit for Web Exercises

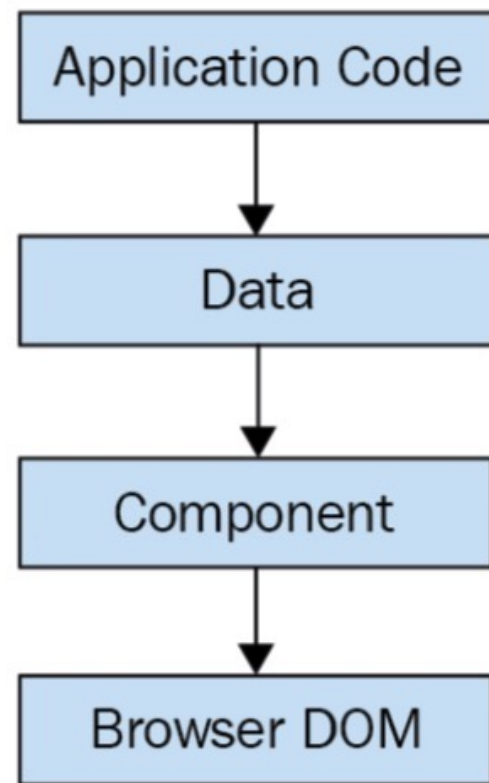


# UI-BIBLIOTHEK: MERKMALE

Name	Price
<b>Sporting Goods</b>	
Football	\$49.99
Baseball	\$9.99
<b>Basketball</b>	\$29.99
<b>Electronics</b>	
iPod Touch	\$99.99
<b>iPhone 5</b>	\$399.99
Nexus 7	\$199.99

- **Komponentenbasiert**
- Also: User Interface aus Komponenten zusammengesetzt
- Zum Beispiel:  
Komponente `ArticleList`

# UI-BIBLIOTHEK: MERKMALE



- Datengesteuert
- Input: Daten der Applikation
- Output: DOM-Struktur für Browser

(data) => view

# NOTATION FÜR KOMPONENTEN

- Gesucht: Notation zum Beschreiben von Komponenten
- Ziel: möglichst deklarativ
- Also nicht: imperativen JavaScript- oder jQuery-Code, der DOM manipuliert
- Verschiedene Möglichkeiten, z.B.
  - **JSX**: in React.js verwendet
  - **SJDON**: eigene Notation

# JSX

```
const Hello = () => (  
  <p>Hello World</p>  
)
```

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

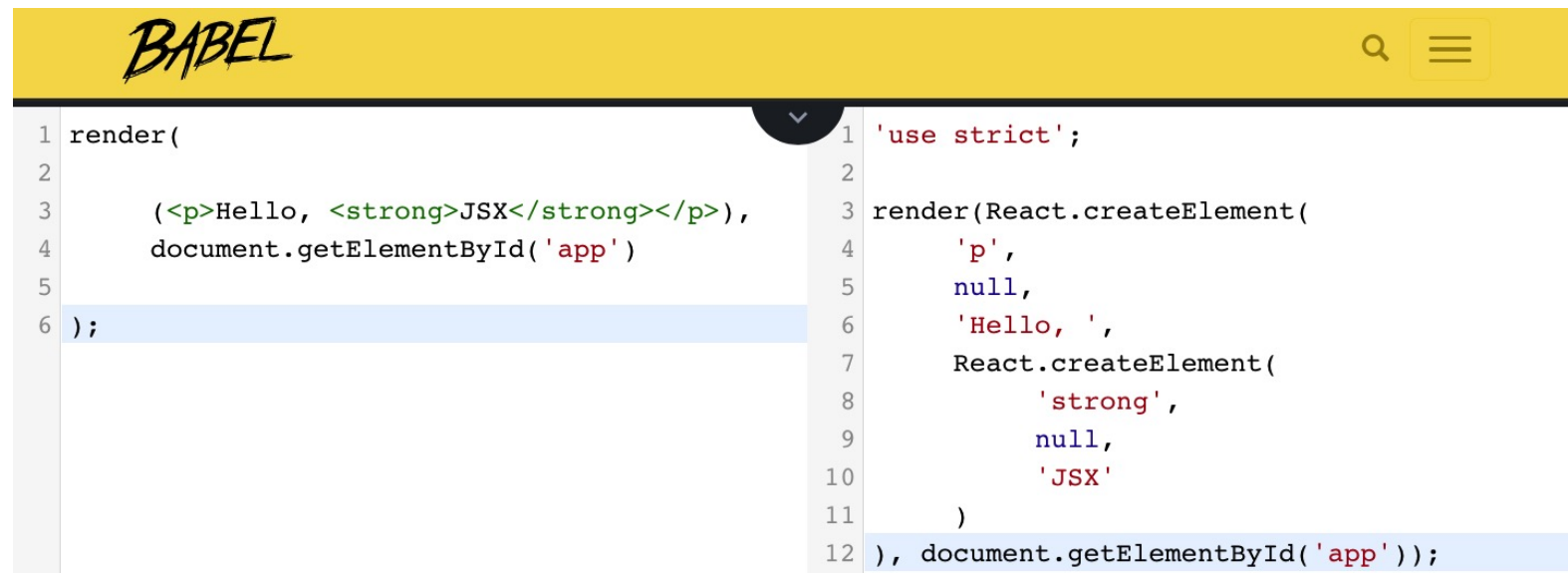
# JSX INS DOM ABBILDEN

```
const domNode = document.getElementById('app')
const root = createRoot(domNode)
root.render(<Hello />)
```

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

# 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



The screenshot shows the Babel REPL interface. The left pane contains the input code, which is a function `render` that takes a JSX element and an ID, and uses `document.getElementById` to find the element. The right pane shows the output code, which is the same function but with the JSX element converted into a call to `React.createElement`. The Babel logo is in the top left, and search and menu icons are in the top right.

```
1 render(  
2  
3   (<p>Hello, <strong>JSX</strong></p>),  
4   document.getElementById('app')  
5  
6 );
```

```
1 'use strict';  
2  
3 render(React.createElement(  
4   'p',  
5   null,  
6   'Hello, ',  
7   React.createElement(  
8     'strong',  
9     null,  
10    'JSX'  
11  )  
12 ), document.getElementById('app'));
```

<https://babeljs.io/repl/>

# JSX: HTML-ELEMENTE

- HTML-Elemente als vordefinierte Komponenten
- Somit können beliebige HTML-Elemente in Komponenten verwendet werden

```
root.render(  
  <section>  
    <header>  
      <h1>A Header</h1>  
    </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

# JSX: KOMPONENTEN

```
1  const MyComponent = () => (  
2    <section>  
3      <h1>My Component</h1>  
4      <p>Content in my component...</p>  
5    </section>  
6  )  
7  
8  root.render(  
9    <MyComponent />  
10 )
```

# JSX: KOMPONENTEN

```
1 const List = ({data}) => (  
2   <ul>  
3     { data.map(item => (<li key={item}>{item}</li>)) }  
4   </ul>  
5 )  
6  
7 root.render(  
8   <List data={["Maria", "Hans", "Eva", "Peter"]} />  
9 )
```

- JavaScript in JSX in `{ . . . }`

# 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 bei Bedarf importiert

# SJDON

- Alternative zu JSX, eigene Notation
- **SJDON – Simple JavaScript DOM Notation**
- Bezeichnung aus einer Semesterendprüfung in WWD (Web-Publishing und Webdesign, G. Burkert, 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 Datenbank gespeichert werden. Schliesslich lässt sich aus den Strukturen auf relativ einfache Weise wieder HTML-Code generieren.

Die Notation – nennen wir sie **SJDON** (Simple JavaScript DOM Notation) – sei wie folgt definiert:

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 Attributbeschreibungen für den Elementknoten.
- Attributbeschreibungen sind Objekte deren Attribute und Werte direkt den Attributen und Werten des HTML-Elements entsprechen. Alle Attribute des Elements können in einem Objekt zusammengefasst oder auf mehrere Objekte verteilt werden.

Beispiel HTML	Mögliche SJDON-Repräsentation
<pre>&lt;html&gt;   &lt;head&gt;     &lt;title&gt;Hello Bsp&lt;/title&gt;   &lt;/head&gt;   &lt;body&gt;     &lt;div id="nav" class="old"&gt;Navi&lt;/div&gt;     &lt;div id="main" class="old"&gt;Hello&lt;/div&gt;   &lt;/body&gt; &lt;/html&gt;</pre>	<pre>[ "html",   [ "head",     [ "title", "Hello Bsp" ]   ],   [ "body",     [ "div", { "id": "nav", "class": "old" },       "Navi" ],     [ "div", { "id": "main" }, "Hello",       { "class": "old" } ]   ] ]</pre>

# VERGLEICH

```
1  /* JSX */
2  const element = (
3    <div style="background:salmon">
4      <h1>Hello World</h1>
5      <h2 style="text-align:right">from SuiWeb</h2>
6    </div>
7  )
8
9  /* SJDON */
10 const element =
11   ["div", {style: "background:salmon"},
12    ["h1", "Hello World"],
13    ["h2", {style: "text-align:right"}, "from SuiWeb"] ]
```

# 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 */  
["ul", ["li", "eins"], ["li", "zwei"]] /* Liste mit zwei Items */  
[App, {name: "SuiWeb"}] /* Funktionskomponente */
```



# 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"}]
```

# FUNKTIONEN

- Funktion liefert SJDON-Ausdruck
- Kein `{ . . . }` für JavaScript wie in JSX nötig

```
const App = ({name}) =>  
  ["h1", "Hi ", name]
```

```
const element =  
  [App, {name: "SuiWeb"}]
```

# BEISPIEL: LISTENKOMPONENTE

```
const MyList = ({items}) =>
  ["ul", ...items.map(item => ["li", item]) ]

const element =
  [MyList, {items: ["milk", "bread", "sugar"]}]
```

- JavaScript-Ausdruck generiert Kind-Elemente für `ul`
- Kein Problem, JavaScript-Ausdrücke einzufügen

SJDON is pure JavaScript 😊

# ZIEL

- Bau einer kleinen Web-Bibliothek
- Ausgerichtet an den Ideen von React
- Komponenten in JSX oder SJDON

Motto:  
Keep it simple

Hinweis: nach unserer Terminologie handelt es sich eher um eine Bibliothek als um ein Framework.

# ZIEL

- Kein Mega-Framework
- Keine "full-stack"-Lösung
- Daten steuern Ausgabe der Komponenten
- Komponenten können einen Zustand haben

# KEIN TWO-WAY-BINDING



- UI-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

# AUSBLICK

- Schrittweiser Aufbau der Bibliothek
- Beispiele im Praktikum

Wichtiger Hinweis: **React.js** ist ein bekanntes und verbreitetes Framework und **JSX** eine bekannte Notation. **SJDON** und **SuiWeb** sind eigene Entwicklungen und ausserhalb WBE unbekannt... 😎

# QUELLEN

- React – A JavaScript library for building user interfaces  
<https://react.dev>
- Adam Boduch: React and React Native  
Second Edition, Packt Publishing, 2018  
[Packt Online Shop](#)



# JSX UND ALTERNATIVEN

- Draft: JSX Specification  
<https://facebook.github.io/jsx/>
- Babel – a JavaScript compiler  
<http://babeljs.io>
- Eigene Notation: SJDON  
<https://github.com/gburkert/sjdon>

## Alternativen:

- HyperScript – Create HyperText with JavaScript  
<https://github.com/hyperhype/hyperscript>
- Hiccup – library for representing HTML in Clojure  
<https://github.com/weavejester/hiccup>

