

WBE: UI-BIBLIOTHEK

TEIL 1: KOMPONENTEN

ÜBERSICHT

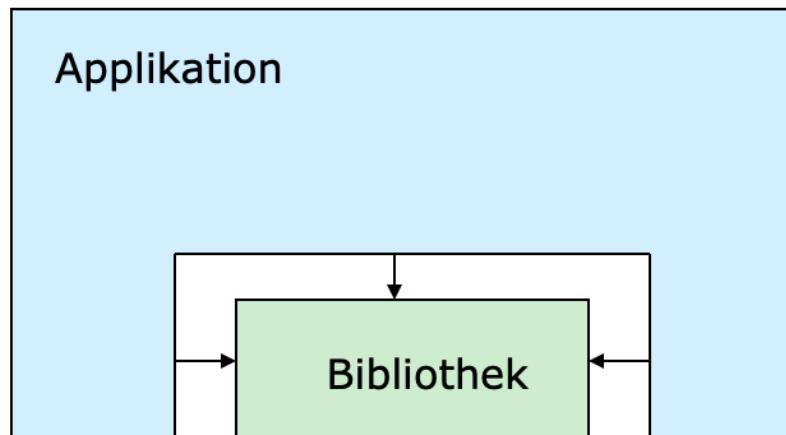
- Frameworks und Bibliotheken
- DOM-Scripting und Abstraktionen
- Aufbau einer UI-Bibliothek

ÜBERSICHT

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

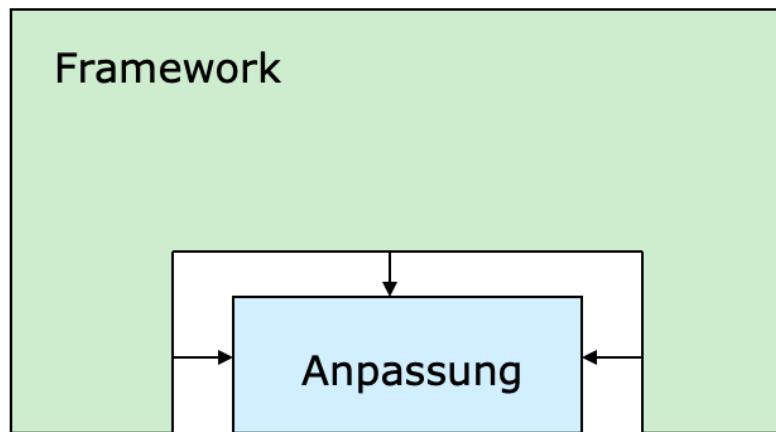
BIBLIOTHEK

- 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)

Speaker notes

Diese Liste ist nicht streng chronologisch: sowohl die genaue Abgrenzung der verschiedenen Ansätze als auch präzise Zeitangaben sind kaum möglich.

SERVERSEITE

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

MODEL-VIEW-CONTROLLER (MVC)

Models

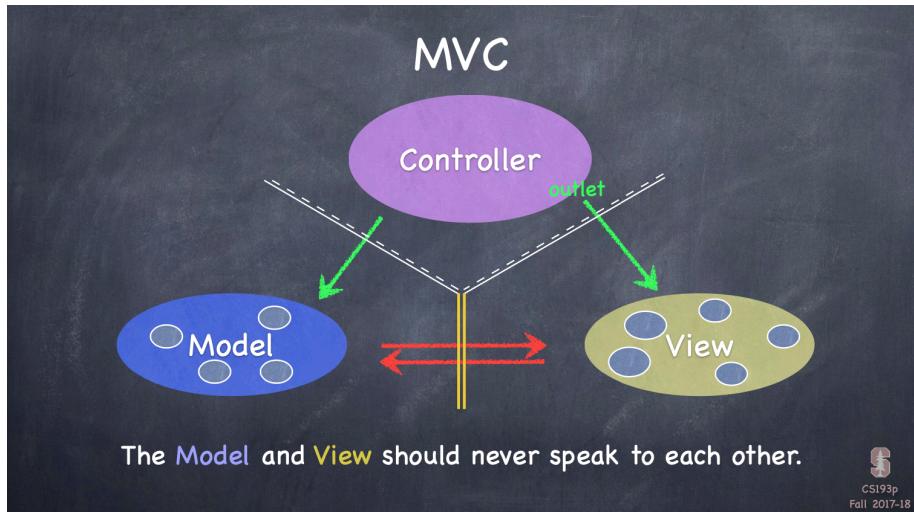
- repräsentieren anwendungsspezifisches Wissen und Daten
- ähnlich Klassen: User, Photo, Todo, Note
- können Observer über Zustandsänderungen informieren

Views

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

Controllers

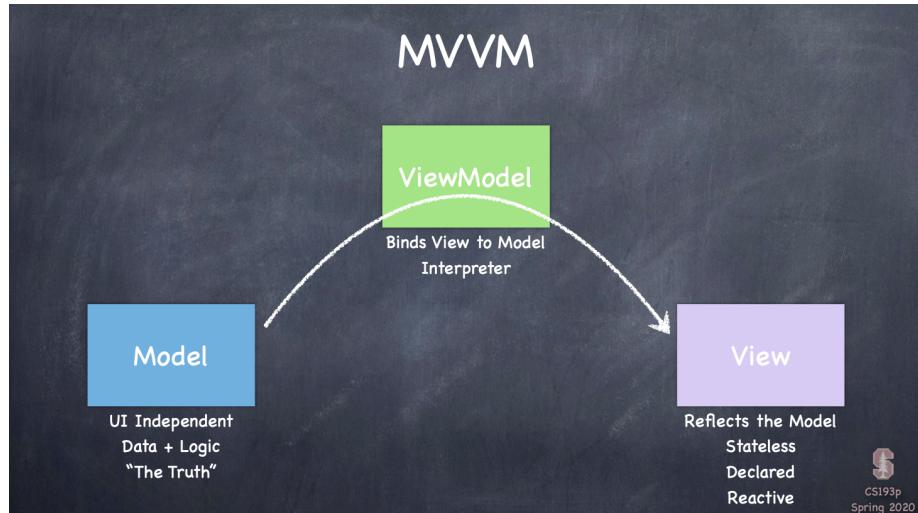
- verarbeiten Eingaben (z.B. Clicks) und aktualisieren Models



Architekturmuster wie MVC werden nicht immer ganz einheitlich aufgefasst. Da findet man im Netz durchaus sehr unterschiedliche Interpretationen. Das Bild stammt aus einem Kurs zur App-Entwicklung für iOS an der Stanford University.

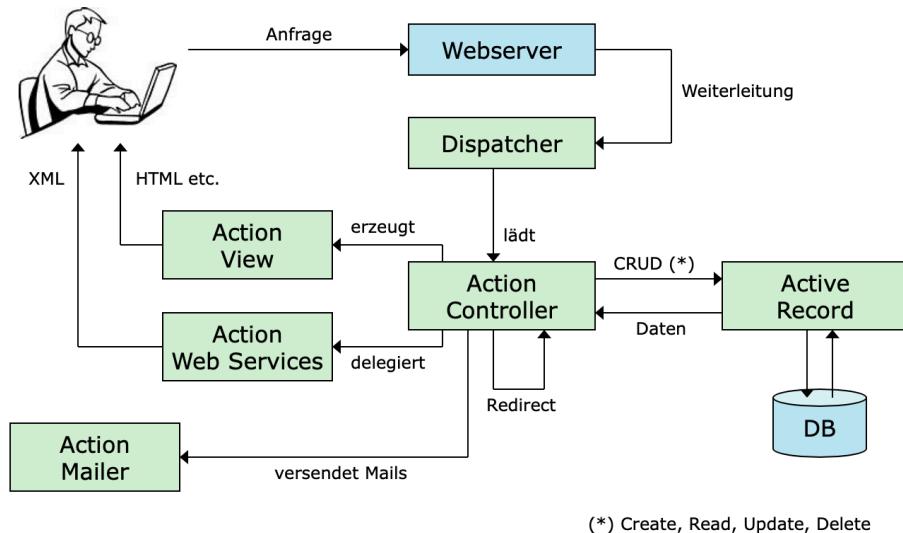
Diverse Varianten:

- Rolle des Controllers angepasst
- MVVM: Model View ViewModel
- MVP: Model View Presenter
(Präsentationslogik im Presenter)



RUBY ON RAILS

- Serverseitiges Framework, basierend auf MVC
- Programmiersprache: Ruby

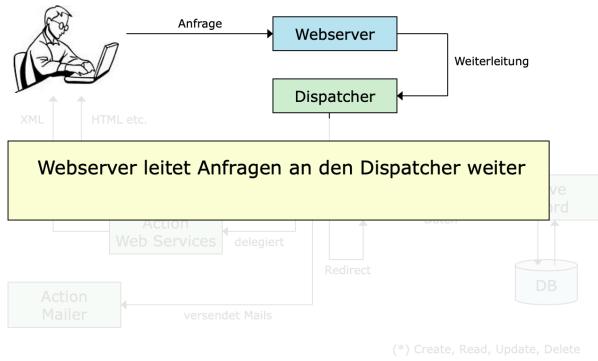


„Convention over Configuration“

<https://rubyonrails.org>

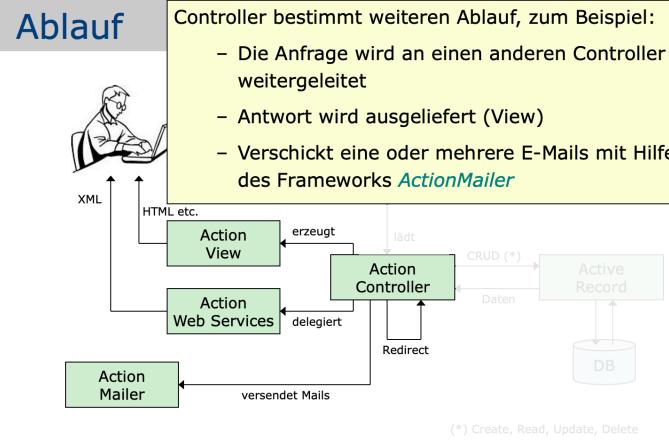
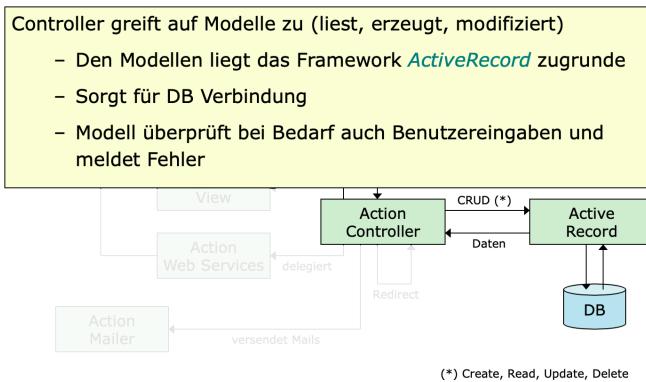
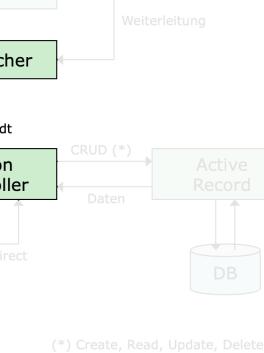
Speaker notes

Zum Ablauf, wie eine Anfrage bearbeitet wird, hier ein paar Bilder. Achtung: Diese Bilder stellen nicht den aktuellen Stand von Ruby on Rails dar, sondern sollen eher zeigen, wie ein prinzipieller Ablauf aussehen könnte. Dies sind Slides des Kurses "Web-Publishing und Webdesign" (WWD) von 2008... :)

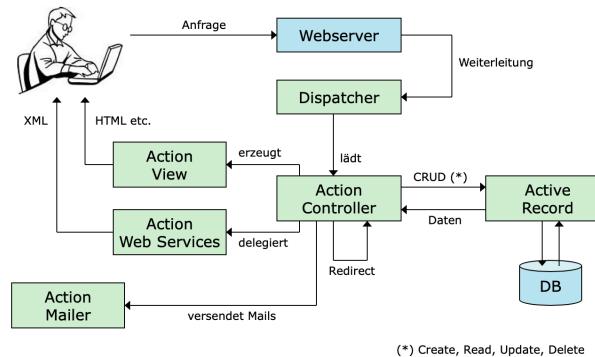
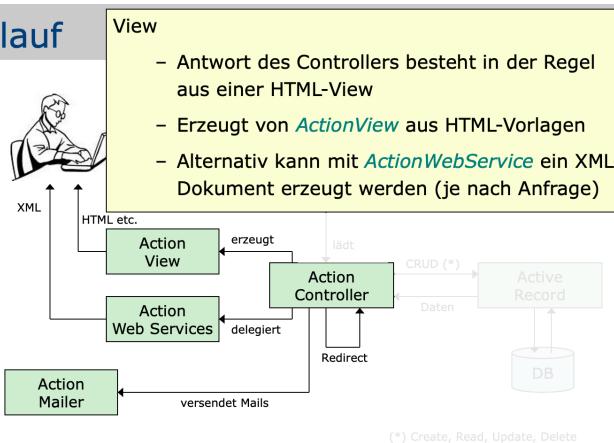


Dispatcher ruft zuständigen Controller auf

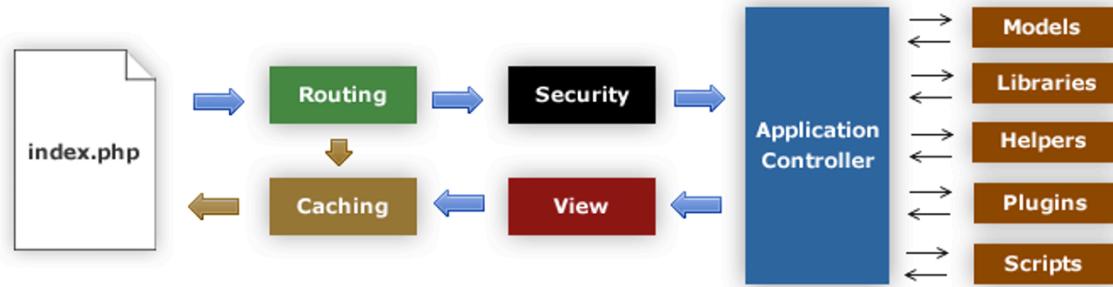
- Die für den Programmablauf zuständigen Skripte
- Keine Konfiguration: Rails ermittelt den Controller aus der URL
- Controller setzt auf dem Framework **ActionController** auf



Ablauf



Auch für PHP gibt es ganz ähnliche Tools, hier zum Beispiel CodeIgniter:



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)

SINGLE PAGE APPS

Der letzte Punkt heisst nicht in jedem Fall, dass Daten schneller verfügbar sind, sie müssen ja immer noch vom Server geladen werden. Es müssen aber nur diejenigen Daten geladen werden, die aktuell benötigt werden und nicht die ganze Seite drum herum. Ausserdem muss der Browser nicht die ganze Seite neu aufbauen. Auf Benutzer-Interaktionen kann so wesentlich unmittelbarer reagiert werden – selbst wenn zunächst nur ein "Daten werden geladen..." o.ä. angezeigt wird.

CLIENT SIDE VS. SERVER SIDE RENDERING

Frameworks wie React haben sich zunächst auf das clientseitige Rendering (CSR) konzentriert. Das heisst, dass dynamische Seitenelemente via JavaScript und DOM-Manipulation auf der Seite platziert werden. das hat aber auch Nachteile:

- Das Laden der ersten Seite einer Applikation dauert länger, da in der Regel eine Reihe von Netzzugriffen erforderlich ist, bis sich die Seite komplett aufgebaut hat.
- Suchmaschinen haben grössere Probleme, den Inhalt der Seiten in den Index aufzunehmen, da sich der Inhalt dynamisch ändern kann. Das kann zu einer schlechteren Platzierung in den Suchergebnissen führen.

Aus diesen Gründen wird heute auch Server Side Rendering (SSR) oder eine Kombination aus SSR und CSR verwendet. Beim SSR wird der Code zum Aufbau der Seite auf dem Server abgearbeitet und dynamisch eine HTML-Seite erstellt, die dann zum Client geschickt wird.

Frameworks wie Next.js, welches auf React basiert, erlauben komponentenbasiertes serverseitiges Rendering, das mit clientseitigem Rendering kombiniert werden kann:

<https://nextjs.org>

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
}
```

Speaker notes

Man könnte `elt` auch noch erweitern. In dieser Variante ist das zweite Argument mit den Attributen fakultativ:

```
function elt (type, ...args) {
  let node = document.createElement(type)
  if (args.length === 0) return node
  let [attrs, ...children] = args
  if (attrs && typeof(attrs) === 'object' && !(attrs instanceof HTMLElement)) {
    Object.keys(attrs).forEach(key => {
      node.setAttribute(key, attrs[key])
    })
  } else if (attrs) {
    children = [attrs, ...children]
  }
  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` gekapselt

```
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

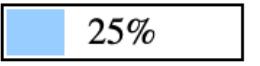
WEB COMPONENTS

- 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>
```



75%



25%

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

WARUM ÜBERHAUPT EIN MINI-REACT SELBER BAUEN?

Die Frage stellt sich natürlich. Man könnte dann doch eigentlich gleich React.js verwenden, oder?

Das Problem ist, dass wir dann sehr schnell in einer Situation landen, in der wir über spezifische React-Features diskutieren. Uns geht es aber nicht um React oder womöglich eine ganz bestimmte React-Version und deren Vorteile und Bugs, sondern um das **Konzept komponentenbasierter User Interfaces**. Das lässt sich besser anhand einer Beispielbibliothek vermitteln, welche auf einige wesentliche Features solcher Bibliotheken reduziert ist.

Oder anders ausgedrückt: es geht nicht darum, React.js zu verstehen, sondern darum, komponentenbasierte User Interfaces zu verstehen. Eine Reihe von Frameworks und Bibliotheken verwenden diesen Ideen als Grundlage. Wenn man die Grundideen verinnerlicht hat, kann man sich relativ leicht in die verschiedenen Frameworks und Bibliotheken einarbeiten.

ES IST ABER EIN PROBLEM ENTSTANDEN

Bibliotheken wie React entwickeln sich rasch weiter. Mit beschränkten Ressourcen konnte die eigene Bibliothek kaum auf dem neuesten Stand gehalten werden. In den Lektionen zur UI-Bibliothek verwenden wir daher einen gemischten Ansatz:

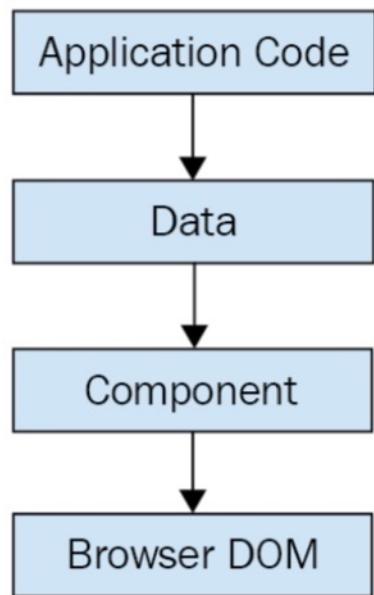
- Es wird weiterhin Hinweise zum Bau eines Mini-Reacts geben
- Für praktische Aufgaben wird aber auf das Original React.js zurückgegriffen

UI-BIBLIOTHEK: MERKMALE

Search...	
<input type="checkbox"/> Only show products in stock	
Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$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

Speaker notes

Der komponentenbasierte, datengesteuerte Ansatz zum Bau von User Interfaces in Kombination mit einer deklarativen Beschreibung der Komponenten führt insgesamt zu einem übersichtlicheren Ansatz bei der Entwicklung von User Interfaces.

Ein *imperativer Stil* der GUI-Programmierung, zum Beispiel mit jQuery, könnte Code wie diesen enthalten:

```
// Imperative GUI Programming
$('.config').prop('checked', false)
$('.namein').attr('placeholder', 'Your name')
```

Im Gegensatz dazu verwendet ein *deklarativer Stil* eine Beschreibung, wie das UI für einen bestimmten Zustand der Applikation aussehen soll. Ein Framework oder Bibliotheksfunktionen kümmern sich dann um die Anzeige und bei Bedarf um deren Anpassung.

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?)

Speaker notes

Die Klammern um das JSX-Markup sind nicht zwingend nötig. Es ist aber üblich, sie einzufügen, um JSX-Markup sauber vom umgebenden JavaScript zu trennen.

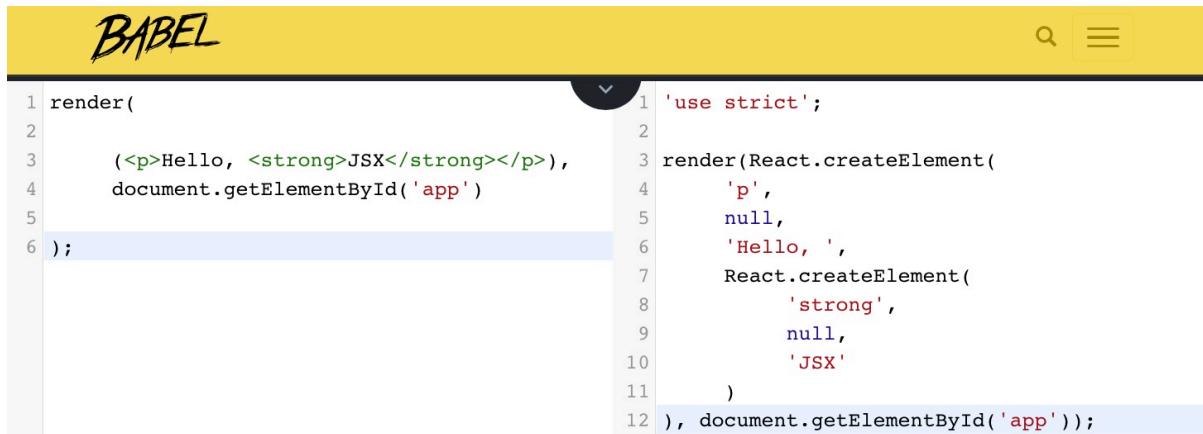
JSX INS DOM ABBILDEN

```
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
- Sonstern: JavaScript-Code mit XML-Teilen
- Muss erst in JavaScript-Code übersetzt werden (Transpiler)
- Browser erhält pures JavaScript



The screenshot shows the Babel REPL interface. On the left, there is a code editor window with a yellow header containing the word "BABEL". The code editor displays the following JSX code:

```
1 render(  
2   (

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


```

On the right, the transpiled JavaScript code is shown:

```
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

Speaker notes

Weiteres Beispiel:

```
root.render(  
  <section>  
    <header>  
      <h1>A Header</h1>  
    </header>  
    <nav>  
      <a href="item">Nav Item</a>  
    </nav>  
    <main>  
      <p>The main content...</p>  
    </main>  
    <footer>  
      <small>&copy; 2021</small>  
    </footer>  
  </section>  
)
```

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 )
```

Speaker notes

React.js unterstützt auch Klassen:

```
class MyComponent extends Component {
  render() {
    // All components have a "render()" method, which
    // returns some JSX markup. In this case, "MyComponent"
    // encapsulates a larger HTML structure.
    return (
      <section>
        <h1>My Component</h1>
        <p>Content in my component...</p>
      </section>
    )
  }
}

root.render(
  <MyComponent>
)
```

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

SJDOM

- Alternative zu JSX, eigene Notation
- SJDOM – 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 **SJDOM** (Simple JavaScript DOM Notation) – sei wie folgt definiert:

Speaker notes

WWD (Web-Publishing und Webdesign) war ab 2001 bkrt's erster Kurs an der ZHAW (die damals noch ZHW hieß). Später ersetzt durch WBD (Webdesign) und WBE (Web-Entwicklung, gleicher Name aber anderer Kurs als heute). Später ersetzt durch eine Folge von Kursen: WEB1, WEB2 und WEB3. Und diese wurden inzwischen ersetzt durch den Kurs WBE.

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><html> <head> <title>Hello Bsp</title> </head> <body> <div id="nav" class="old">Navi</div> <div id="main" class="old">Hello</div> </body> </html></pre>	<pre>["html", ["head", ["title", "Hello Bsp"]], ["body", ["div", {"id": "nav", "class": "old"}, "Navi"], ["div", {"id": "main"}, "Hello", {"class": "old"}]]]</pre>

Speaker notes

WWD – Semesterprüfung FS 11 G. Burkert, 29.06.2011	Name:
Punkte:	maximal: 60
Note: Dauer: 90 Minuten. Hilfsmittel: 4 A4-Seiten mit Notizen zulässig. Bitte die Fragen möglichst direkt auf dem Aufgabenblatt beantworten. Bei den Multiple-Choice-Fragen können auch mehrere Antworten richtig sein. Wichtig: ordentliche und lesbare Darstellung. Auf jede separat abgegebene Seite den Namen schreiben. Viel Erfolg!	

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:

- 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><html> <head> <title>Hello Bsp</title> </head> <body> <div id="nav" class="old">Navi</div> <div id="main" class="old">Hello</div> </body> </html></pre>	<pre>["html", ["head", ["title", "Hello Bsp"]], ["body", ["div", { "id": "nav", "class": "old" }, "Navi"], ["div", { "id": "main" }, "Hello", { "class": "old" }]]]</pre>

Vergleichbare Ansätze:

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

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 */
```

ATTRIBUTES

- 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

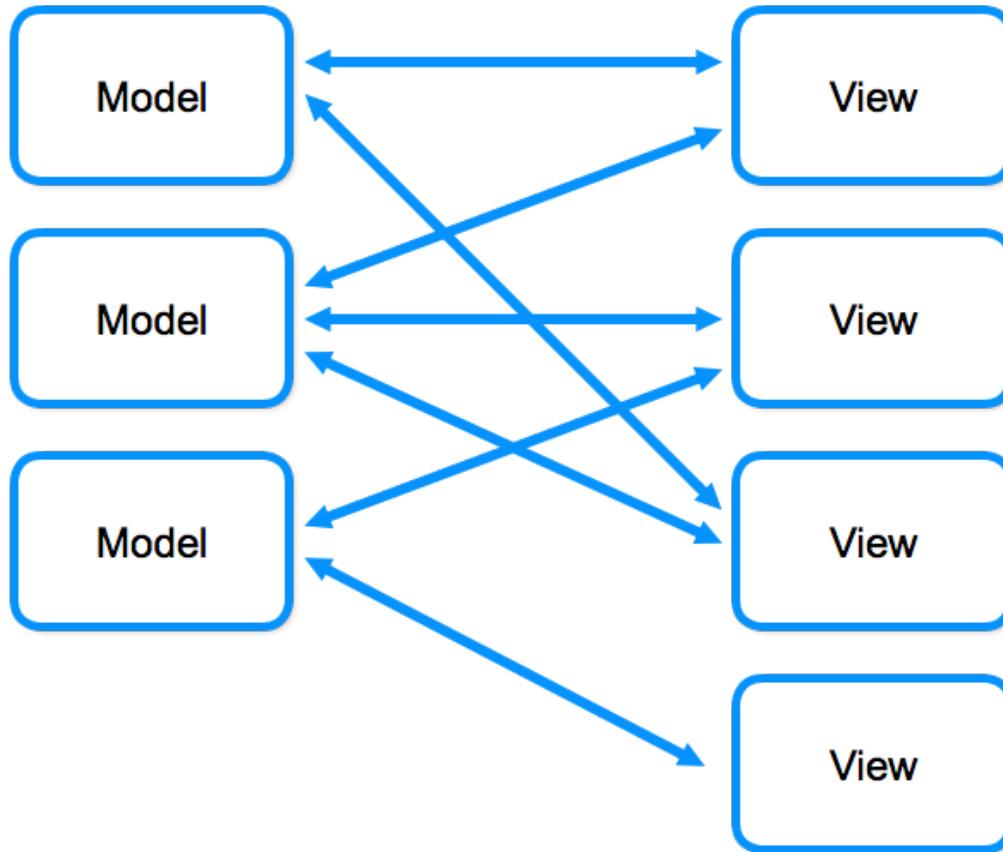
KEIN TWO-WAY-BINDING



- 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

Speaker notes

Two-Way-Binding kann bei grösseren UIs schnell unübersichtlich werden



Beispiel

- Facebook Ads
- Komplexes UI mit vielen Abhängigkeiten der einzelnen UI-Elemente

The screenshot shows the Facebook Audience Insights interface. At the top, there's a search bar and a navigation bar with 'Tom' and 'Home 2'. Below the search bar, the text 'Who do you want your ads to reach?' is displayed.

Target Ads to People Who Know Your Business
You can create a Custom Audience to show ads to your contacts, website visitors or app users. Create a Custom Audience

Audience Definition

Your audience is defined.

Specific Broad

Audience Details:

- Location:
 - United States: Menlo Park (+25 mi) California; Chicago (+10 mi) Illinois; Las Vegas (+50 mi) Nevada
- Not connected to:
 - That's a Pro Tip
- Age:
 - 18 - 65+

Potential Reach: 7,000,000 people

Locations: United States, California
Menlo Park + 25 mi ✓
United States, Illinois
Chicago + 10 mi ✓
United States, Nevada
Las Vegas + 50 mi ✓
Add a country, state/province, city, ZIP or address

Age: 18 - 65+ ✓

Gender: Men Women

Languages: English

Demographics: 16-24, Female

Interests: 18, Interests, Suggestions, Browse

Behaviors: Behaviors, Browse

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>