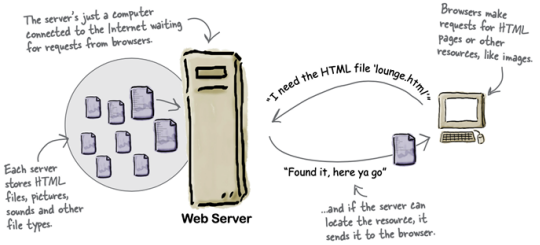
# Web-Entwicklung

Jil Zerndt, Lucien Perret
December 2024

**WEB-Architektur** **Client-Server-Modell:**
- Browser (Client) sendet Anfragen an Server
- Server verarbeitet Anfragen und sendet Antworten
- Kommunikation über HTTP/HTTPS (Port 80/443)




**Technologien**

**Client-Seitig** → Front-end Entwickler
- Beschränkt auf Browser-Funktionalität
- Basistechnologien: HTML (Struktur), CSS (Darstellung), JavaScript (Verhalten)
- Browser APIs und Web-Standards

**Server-Seitig** → Back-end Entwickler
- Freie Wahl von Plattform und Programmiersprache
- Generiert Browser-kompatible Ausgabe
- Beispiele: Node.js, Express, REST APIs

**Internet vs. WWW**

**Internet:**
- Weltweites Netzwerk aus vielen Rechnernetzwerken
- Verschiedene Dienste: E-Mail, FTP, WWW, etc.
- Basis-Protokolle: TCP/IP

**World Wide Web:**
- Service, der auf dem Internet aufbaut
- Entwickelt von Tim Berners-Lee am CERN (1990er)
- Basiert auf: HTTP, HTML, URLs

**Web-Standards**
- W3C (World Wide Web Consortium)
- WHATWG (Web Hypertext Application Technology Working Group)
- HTML Living Standard
- Browser-Hersteller (Chrome, Firefox, Safari, etc.)

## JavaScript

**Web-Konsole** JavaScript Console im Browser:
- `console.log(message)`: Gibt eine Nachricht aus
- `console.clear()`: Löscht die Konsole
- `console.trace(message)`: Stack trace ausgeben
- `console.error(message)`: stderr ausgeben
- `console.time()`: Timer starten
- `console.timeEnd()`: Timer stoppen

API-Dokumentation: `https://nodejs.org/api/console.html`

**Datentypen** JavaScript kennt folgende primitive Datentypen:
- `number`: 64-Bit Floating Point (IEEE 754)
  - `Infinity`: 1/0
  - `NaN`: Not a Number (0/0)
- `bigint`: Ganzzahlen beliebiger Größe (mit n am Ende)
- `string`: Zeichenketten in ", oder "
- `boolean`: `true` oder `false`
- `undefined`: Variable deklariert aber nicht initialisiert
- `null`: Variable bewusst ohne Wert
- `symbol`: Eindeutiger Identifier

**typeof-Operator**

Mit `typeof` kann der Typ eines Wertes ermittelt werden:

```
typeof 42           // 'number'
typeof 42n          // 'bigint'
typeof "text"       // 'string'
typeof true         // 'boolean'
typeof undefined    // 'undefined'
typeof null         // 'object' (!)
typeof {}           // 'object'
typeof []           // 'object'
typeof (() => {})   // 'function'
typeof Infinity     // 'number'
typeof NaN          // 'number' !!
typeof 'number'     // 'string'
```

**Variablenbindung**

JavaScript kennt drei Arten der Variablendeklaration:
- `var`
  - Scope: Funktions-Scope (Global oder Lokal)
  - Eigenschaften: Kann neu deklariert werden
- `let`
  - Scope: Block-Scope (innerhalb von {})
  - Eigenschaften: Moderne Variante für veränderliche Werte
- `const`
  - Scope: Block-Scope (innerhalb von {})
  - Eigenschaften: Wert kann nicht neu zugewiesen werden

**Operatoren**
- Arithmetische Operatoren: $+, -, *, /, \%, ++, --$
- Zuweisungsoperatoren: $=, + =, - =, * =, / =, \% =, ** =, <<=, >>=, >>>=, \& =, ^=, | =$
- Vergleichsoperatoren: $==, ===, ! =, ! ==, >, <, >=, <=$
- Logische Operatoren: $\&\&, ||, !$
- Bitweise Operatoren: $\&, |, ^, <<, >>, >>>$
- Sonstige Operatoren: `typeof`, `instanceof`

**Vergleichsoperatoren** JavaScript unterscheidet zwei Arten von Gleichheit:
- `==` und `!=`: Mit Typumwandlung
- `===` und `!==`: Ohne Typumwandlung (strikt)

```
5 == "5"      // true  (Typumwandlung)
5 === "5"     // false (keine Typumwandlung)
null == undefined    // true
null === undefined   // false
```

**Verzweigungen, Wiederholung und Switch Case**
- `if (condition) {...} else {...}`
- `switch (expression) { case x: ... break; default: ... }`
- `for (initialization; condition; increment) {...}`
- `while (condition) {...}`
- `do {...} while (condition)`
- `for (let x of iterable) {...}`

**Kontrollstrukturen**

```
// If-Statement
if (condition) {
    // code
} else if (otherCondition) {
    // code
} else {
    // code
}

// Switch Statement
switch(value) {
    case 1:
        // code
        break;
    case 2:
        // code
        break;
    default:
        // code
}

// Loops
for (let i = 0; i < n; i++) { }
while (condition) { }
do { } while (condition);
for (let item of array) { }
for (let key in object) { }
```

**Funktionsdefinition**
- `function name(parameters) {...}`
- `const name = (parameters) => {...}`
- `const name = parameters => {...}`
- `const name = parameters => expression`

**Funktionsdefinitionen** JavaScript kennt verschiedene Arten, Funktionen zu definieren:

```
// Funktionsdeklaration
function add(a, b) {
    return a + b;
}

// Funktionsausdruck
const multiply = function(a, b) {
    return a * b;
};

// Arrow Function
const subtract = (a, b) => a - b;

// Arrow Function mit Block
const divide = (a, b) => {
    if (b === 0) throw new Error('Division by zero');
    return a / b;
};
```

## Objects and Arrays

### Objekt vs Array

| Was | Objekt | Array |
|---|---|---|
| Art | Attribut-Wert-Paare | Sequenz von Werten |
| Literalnotation | werte = { a: 1, b: 2} | liste = [1, 2, 3] |
| Ohne Inhalt | werte = {} | liste = [] |
| Elementzugriff | werte["a" ] oder werte.a | liste[0] |

**Json** JavaScript Object Notation

- Daten-Austauschformat, nicht nur für JavaScript
- Orientiert an Notation für JavaScript-Objektliterale

https://www.json.org/json-en.html

```
> JSON.stringify({type: "cat", name: "Mimi", age: 3})
'{"type":"cat", "name":"Mimi", "age":3}'
> JSON.parse('{"type": "cat", "name": "Mimi", "age":
    3}')
{type: 'cat', name: 'Mimi', age: 3}
```

**JS-Objekte** sind Sammlungen von Schlüssel-Wert-Paaren:

- Eigenschaften können dynamisch hinzugefügt/entfernt werden
- Werte können beliebige Typen sein (auch Funktionen)
- Schlüssel sind immer Strings oder Symbols

### Objekte erstellen und manipulieren

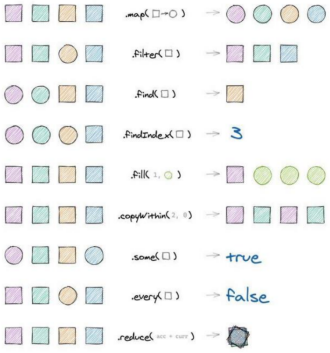
```
// Objekt-Literal
const person = {
    name: "Alice",
    age: 30,
    greet() {
        return "Hello, I am" + this.name;
    }
};

// Eigenschaften manipulieren
person.job = "Developer";     // hinzufuegen
delete person.age;            // loeschen
"name" in person;             // true
```

**Arrays** Arrays in JavaScript sind spezielle Objekte für geordnete Sammlungen:

- `push()`, `pop()`: Ende hinzufügen/entfernen
- `unshift()`, `shift()`: Anfang hinzufügen/entfernen
- `splice()`: Elemente einfügen/entfernen
- `slice()`: Teilarray erstellen
- `map()`, `filter()`, `reduce()`: Funktional
- `forEach()`: Iteration über Elemente
- `indexOf()`, `lastIndexOf()`: Index suchen
- `concat()`: Arrays zusammenfügen
- `sort()`, `reverse()`: Sortieren/Umkehren

```
const arr = [1, 2, 3];
arr.push(4);               // [1,2,3,4]
arr.pop();                 // [1,2,3]
arr.unshift(0);            // [0,1,2,3]
arr.shift();               // [1,2,3]
```




Achtung: draw new!!!

## Funktionen und funktionale Programmierung

### Funktionen

- Funktionen sind spezielle, aufrufbare Objekte
- Man kann ihnen jederzeit Attribute oder Methoden hinzufügen
- Sie haben bereits vordefinierte Methoden

```
> const add = (x, y) => x + y
> add.doc = "This function adds two values"
> add(3,4)
7
> add.doc
'This function adds two values'
```

### Modulsystem in JavaScript

- `import` und `export` für Module
- `export default` für Standardexport
- `import {name} from 'module'` für benannte Exports
- `import * as name from 'module'` für alle Exports

```
const car = {                        //car-lib.js
    brand: 'Ford',
    model: 'Fiesta'
}
module.exports = car
const car = require('./car-lib') //other js file
```

## Prototypen von Objekten

### Prototypen

- Die meisten Objekte haben ein Prototyp-Objekt.
- Dieses fungiert als Fallback für Attribute und Methoden.

```
>Object.getPrototypeOf(Math.max)==Function.prototype
true
>Object.getPrototypeOf([])==Array.prototype
true
>Object.getPrototypeOf(Function.prototype)==Object.prototype
true
>Object.getPrototypeOf(Array.prototype)==Object.prototype
true
```

### Prototypen-Kette Call, apply, bind

- Weitere Argumente von call : Argumente der Funktion
- Weiteres Argument von apply : Array mit den Argumenten
- Erzeugt neue Funktion mit gebundenem this

```
function Employee (name, salary) {
    Person.call(this, name)
    this.salary = salary
}
Employee.prototype = new Person()
Employee.prototype.constructor = Employee
let e17 = new Employee("Mary", 7000)
console.log(e17.toString()) // Person with name 'Mary'
console.log(e17.salary) // 7000
```

### Klassen

- Klassen sind syntaktischer Zucker für Prototypen
- Klassen können Attribute und Methoden enthalten
- Klassen können von anderen Klassen erben

```
class Person {
    constructor (name) {
        this.name = name
    }
    toString () {
        return `Person with name '${this.name}'
    }
}
let p35 = new Person("John")
console.log(p35.toString()) // Person with name 'John'
```

### Vererbung

```
class Employee extends Person {
    constructor (name, salary) {
        super(name)
        this.salary = salary
    }
    toString () {
        return `${super.toString()} and salary
            ${this.salary}
    }
}
let e17 = new Employee("Mary", 7000);
console.log(e17.toString()) /* Person with name 'Mary'
    and salary 7000 */
console.log(e17.salary) /* 7000 */
```

### Getter und Setter

```
class PartTimeEmployee extends Employee {
    constructor (name, salary, percentage) {
        super(name, salary)
        this.percentage = percentage
    }
    get salary100 () { return this.salary * 100 /
        this.percentage}
    set salary100 (amount) { this.salary = amount *
        this.percentage / 100 }
}
let e18 = new PartTimeEmployee("Bob", 4000, 50)
console.log(e18.salary100) /* -> 8000 */
e18.salary100 = 9000
console.log(e18.salary) /* \ 4500 */
```

## Filesystem

**Pfade der Datei** Um Pfad-informationen einer Datei zu ermitteln muss man dies mit require('path') machen.

```
const path = require('path')
const notes = '/users/bkrt/notes.txt'
path.dirname(notes) /* /users/bkrt */
path.basename(notes) /* notes.txt */
path.extname(notes) /* .txt */
path.basename(notes, path.extname(notes)) /* notes */
```

**File API** Mit require('fs') wird auf die File-Api zugegriffen. Die File-Api bietet Funktionen zum Lesen und Schreiben von Dateien.

### FS Funktionen

- `fs.access`: Zugriff auf Datei oder Ordner prüfen
- `fs.mkdir`: Verzeichnis anlegen
- `fs.readdir`: Verzeichnis lesen, liefert Array von Einträgen
- `fs.rename`: Verzeichnis umbenennen
- `fs.rmdir`: Verzeichnis löschen
- `fs.chmod`: Berechtigungen ändern
- `fs.chown`: Besitzer und Gruppe ändern
- `fs.copyFile`: Datei kopieren
- `fs.link`: Besitzer und Gruppe ändern
- `fs.symlink`: Symbolic Link anlegen
- `fs.watchFile`: Datei auf Änderungen überwachen

### Datei-Informationen

```
const fs = require('fs')
fs.stat('test.txt' , (err, stats) => {
    if (err) {
    console.error(err)
    return
    }
    stats.isFile() /* true */
    stats.isDirectory() /* false */
    stats.isSymbolicLink() /* false */
stats.size /* 1024000 = ca 1MB */
})
```

### Dateien lesen und schreiben

```
const fs = require('fs')
fs.readFile('/etc/hosts',"utf8", (err, data) => {
        if (err) throw err
        console.log(data)
})

const content = 'Node was here!'
fs.writeFile('/Users/bkrt/test.txt', content, (err) =>
    {
    if (err) {
        console.error(`Failed to write file: ${err}`)
        return
    } // file written successfully
})
```

## Asynchrone Programmierung

### Asynchrone Programmierung

JavaScript verwendet verschiedene Mechanismen für asynchrone Operationen:

- Callbacks: Traditioneller Ansatz
- Promises: Moderner Ansatz für strukturiertere asynchrone Operationen
- Async/Await: Syntaktischer Zucker für Promises

## Callbacks und Timers

**Callbacks** Ein Callback ist eine Funktion, welche als Argument einer anderen Funktion übergeben wird und erst aufgerufen wird, wenn das Ereignis eingetreten ist. In der folgenden Abbildung wird die KlickFunktion vom Button mit der Id «Button» abonniert.

```
document.getElementById('button').addEventListener('click',
    () => {
//item clicked
})
```

### SetTimeout

- Mit setTimeout kann Code definiert werden, der zu einem späteren Zeitpunkt ausgeführt werden soll
- Eintrag in die Timer-Liste, auch wenn Zeit auf 0 gesetzt wird
- Kann mit clearTimeout entfernt werden

```
setTimeout(() => {
    /* runs after 50 milliseconds */
}, 50)
```

### SetInterval

- Callback alle n Millisekunden in die Callback Queue eingefügt
- Kann mit clearInterval beendet werden

```
const id = setInterval(() => {
// runs every 2 seconds
}, 2000)
clearInterval(id)
```

### SetImmediate

- Callback wird in die Immediate Queue eingefügt
- Wird nach dem aktuellen Event-Loop ausgeführt

```
setImmediate(() => {
    console.log('immediate')
})
```

## Events und Promises

### Event-Modul (EventMitter)

- EventEmitter verwaltet Liste von Listeners zu bestimmten Events
- Listener für das Event können hinzugefügt oder entfernt werden
- Event kann ausgelöst werden → Listener werden informiert

### Listener hinzufügen

```
const EventEmitter = require('events')
const door = new EventEmitter()

door.on('open', () => {
    console.log('Door was opened')
})
```

### Event auslösen

```
door.on('open', (speed) => {
    console.log(`Door was opened, speed: ${speed ||
        'unknown'}`)
})

door.emit('open')
door.emit('open', 'slow')
```

**Promises** Ist ein Platzhalter für einen Wert, der erst später voraussichtlich verfügbar sein wird. Funktion mit Promise:

```
function readFilePromise(file) {
    let promise = new Promise(function
        resolver(resolve, reject) {
        fs.readFile(file, "utf8", (err, data) => {
            if (err) reject(err);
            else resolve(data);
        });
    });
    return promise;
}
```

Gibt nun ein Promise-Object zurück

### Promise-Konstruktor erhält resolver-Funktion

Rückgabe einer Promise: potentieller Wert kann später erfüllt oder zurückgewiesen werden

- Rückgabe einer Promise: potentieller Wert
- kann später erfüllt oder zurückgewiesen werden

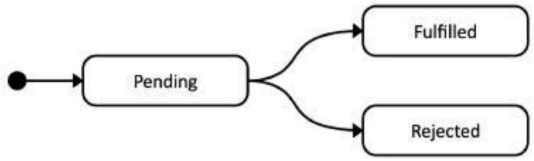
Aufruf neu:

```
readFilePromise('/etc/hosts')
    .then(console.log)
    .catch(() => {
        console.log("Error reading file")
    })
```

### Promise-Zustände

- pending: Ausgangzustand
- fulfilled: erfolgreich abgeschlossen
- rejected: ohne Erfolg abgeschlossen

Nur ein Zustandsübergang möglich und Zustand in Promise-Objekt gekapselt




### Promises Verknüpfen

- Then-Aufruf gibt selbst Promise zurück
- Catch-Aufruf ebenfalls, per Default erfüllt
- So können diese Aufrufe verkettet werden
- Promise, welche unmittelbar resolved wird: Promise.resolve (...)
- Promise, welche unmittelbar rejected wird: Promise.reject (...)

### Promise.all()

- Erhält Array von Promises
- Erfüllt mit Array der Result, wenn alle erfüllt sind
- Zurückgewiesen sobald eine Promise zurückgewiesen wird

### Promise.race()

- Erhält Array von Promises
- Erfüllt sobald eine davon erfüllt ist
- Zurückgewiesen sobald eine davon zurückgewiesen wird

### ASYNC/AWAIT

```
/* Bekanntes Beispiel */
const readHosts =() => {
    readFilePromise('/etc/hosts')
        .then(console.log)
        . catch(() => {
            console.log("Error reading file")
        })
}
/* Mit async/await */
const readHosts = async () => {
    try {
        console.log(await
            readFilePromise('/etc/hosts'))
    }
    catch (err) {
        console.log("Error reading file")
    }
}
```

Beispiel 2:

```
function resolveAfter2Seconds (x) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(x)
        }, 2000)
    })
}
async function add1(x) {
    var a = resolveAfter2Seconds(20)
    var b = resolveAfter2Seconds(30)
    return x + await a + await b
}
add1(10).then(console.log)
```

### Promise Erstellung und Verwendung

```
// Promise erstellen
const myPromise = new Promise((resolve, reject) => {
    // Asynchrone Operation
    setTimeout(() => {
        if (/* erfolg */) {
            resolve(result);
        } else {
            reject(error);
        }
    }, 1000);
});

// Promise verwenden
myPromise
    .then(result => {
        // Erfolgsfall
    })
    .catch(error => {
        // Fehlerfall
    })
    .finally(() => {
        // Wird immer ausgefuhrt
    });

// Async/Await Syntax
async function myAsync() {
    try {
        const result = await myPromise;
        // Erfolgsfall
    } catch (error) {
        // Fehlerfall
    }
}
```

### Module System

JavaScript verwendet verschiedene Modulsysteme:

- CommonJS (Node.js): `require/module.exports`
- ES Modules: `import/export`

### Module Import/Export

```
// CommonJS (Node.js)
const fs = require('fs');
module.exports = { /* ... */ };

// ES Modules
import { function1, function2 } from './module.js';
export const variable = 42;
export default class MyClass { /* ... */ }
```

### Error Handling

```
try {
    // Code der Fehler werfen konnte
    throw new Error('Something went wrong');
} catch (error) {
    // Fehlerbehandlung
    console.error(error.message);
} finally {
    // Wird immer ausgefuhrt
    cleanup();
}
```

## Webserver

Die Standard-Ports von einem Webserver sind 80 und 443. Der Webserver wartet auf eine Anfrage vom Client.

### Server im Internet

- Wartet auf Anfragen auf bestimmtem Port
- Client stellt Verbindung her und sendet Anfrage
- Server beantwortet Anfrage

### Ports

| Port | Service |
|---|---|
| **20** | FTP - Data |
| **21** | FTP - Control |
| **22** | SSH Remote Login Protocol |
| **23** | Telnet |
| **25** | Simple Mail Transfer Protocol (SMTP) |
| **53** | Domain Name System (DNS) |
| **80** | HTTP |
| **443** | HTTPs |

### File-Transfer File Server

Um Dateien auf einem File-Server auszutauschen, werden die Protokolle FTP (File Transfer Protocol) und SFTP (SSH File Transfer Protocol) verwendet.

## HTTP

### HTTP-Requests

- GET: Ressource laden
- POST: Informationen senden
- PUT: Ressource anlegen, überschreiben
- PATCH: Ressource anpassen
- DELETE: Ressource löschen

### HTTP-Response Codes

| Code | Beschreibung |
|---|---|
| **1xx** | Information (101 Switching protocols) |
| **2xx** | Erfolg (200 OK) |
| **3xx** | Weiterleitung (301 Moved permanently) |
| **4xx** | Fehler in Anfrage (403 Forbidden, 404 Not Found) |
| **5xx** | Server-Fehler (501 Not implemented) |

## Einfacher Webserver (Node.js)

### Node.js Webserver

```
const {createServer} = require("http")
Let server = createServer((request, response) => {
    response.writeHead(200, {"Content-Type":
        "text/html"})
    response.write(`
        <h1>Hello!</h1>
        <p>You asked for
            <code>${request.url}</code></p>`)
    response.end()
})
server.listen(8000)
console.log("Listening! (port 8000)")|
```

### Einfacher Webclient

```
const {request} = require("http")
let requestStream = request({
    hostname: "eloquentjavascript.net",
        path: "/20_node.html",
        method: "GET"
        headers: {Accept: "text/html"}
}, response => {
        console.log("Server responded with status
            code", response.statusCode)
})
requestStream.end()
```

### Server und Client mit Streams

```
const {createServer} = require("http")
createServer((request, response) => {
    response.writeHead(200, {"Content-Type":
        "text/plain"})
        request.on("data", chunk =>
            response.write(chunk.toString().toUpperCase()))
        request.on("end" , () => response.end())
    }).listen(8000)
```

```
const {request} = require("http")
Let rq = request({
    hostname: "localhost",
    port: 8000,
    method: "POST"
}, response => {
    response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
})
rq.write("Hello server\n")
rq.write("And good bye\n")
rq.end()
```

### REST API

- REST: Representational State Transfer
- Zugriff auf Ressourcen über ihre Adresse (URI)
- Kein Zustand: jede Anfrage komplett unabhängig
- Kein Bezug zu vorhergehenden Anfragen
- Alle nötigen Informationen in Anfrage enthalten
- Verwenden der HTTP-Methoden: GET , PUT , POST , ...

### Express.js

Express.js ist ein minimales, aber flexibles Framework für Web-apps. Es hat zahlreiche Utilities und Erweiterungen. Express.js basiert auf Node.js. → http://expressjs.com

### Installation

- Der Schritt npm init fragt eine Reihe von Informationen (Projektname, Version, ...) zum Projekt ab
- Als Entry Point ist hier index.js voreingestellt
- Das kann zum Beispiel in app.js geändert werden.

```
$ mkdir myapp
$ cd myapp
$ npm init
$ npm install express --save
```

### Beispiel: Express Server

```
const express = require('express')
const app = express()
const port = 3000
app.get('/', (req, res) => {
        res.send('Hello World!')
})
app.listen(port, () => {
    console.log(`Example app listening at
        http://localhost:${port}`)
}})
```

### Routing

```
app.get('/', function (req, res) {
    res.send('Hello World!')
})
app.post('/', function (req, res) {
    res.send('Got a POST request')
})
app.put('/user', function (req, res) {
    res.send('Got a PUT request at /user')
})
app.delete('/user', function (req, res) {
    res.send('Got a DELETE request at /user')
})
```

### Beispiel (zugehörige Tests)

```
/* PlayerSpec.js - Auszug */
describe("when song has been paused", function() {
    beforeEach(function() {
        player.play(song)
        player.pause()
    })
    it("should indicate that the song is currently
         paused", function() {
        expect(player.isPlaying).toBeFalsy()
        /* demonstrates use of 'not' with a custom
             matcher */
        expect(player).not.toBePlaying(song)
    })
    it("should be possible to resume", function() {
        player.resume()
        expect(player.isPlaying).toBeTruthy()
        expect(player.currentlyPlayingSong)
            .toEqual(song)
    })
})
```

### JASMINE: MATCHER

```
expect([1, 2, 3]).toEqual([1, 2, 3])
expect(12).toBeTruthy()
expect("").toBeFalsy()
expect("Hello planet").not.toContain("world")
expect(null).toBeNull()
expect(8).toBeGreaterThan(5)
expect(12.34).toBeCloseTo(12.3, 1)
expect("horse_ebooks.jpg")
    .toMatch(/\w+.(jpg|gif|png|svg)/i)
```

### JASMINE: TESTS DURCHFÜHREN

```
$ npx jasmine
Randomized with seed 03741
Started
......
5 specs, 0 failures
Finished in 0.014 seconds
Randomized with seed 03741
    (jasmine --random=true --seed=03741)
```

# Client-Server Interaktion

## Formulare

**HTML-Formulare** Formulare ermöglichen Benutzereingaben und Datenübertragung:

- `<form>` Element mit `action` und `method`
- `method="GET"`: Daten in URL (sichtbar)
- `method="POST"`: Daten im Request-Body (unsichtbar)
- Verschiedene Input-Typen: text, password, checkbox, radio, etc.

### Formular Handling

```
<!-- HTML Form -->
<form action="/submit" method="POST">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username">

    <label for="password">Password:</label>
    <input type="password" id="password"
        name="password">

    <button type="submit">Login</button>
</form>

<!-- JavaScript Handler -->
form.addEventListener('submit', (event) => {
    event.preventDefault(); // Verhindert
        Standard-Submit

    const formData = new FormData(form);
    // Zugriff auf Formular-Daten
    const username = formData.get('username');
    const password = formData.get('password');
});
```

**Formular Events** Wichtige Events bei Formularen:

- `submit`: Formular wird abgeschickt
- `reset`: Formular wird zurückgesetzt
- `change`: Wert eines Elements wurde geändert
- `input`: Wert wird gerade geändert
- `focus`: Element erhält Fokus
- `blur`: Element verliert Fokus

## AJAX und Fetch API

**AJAX** Asynchronous JavaScript And XML:

- Asynchrone Kommunikation mit dem Server
- Kein vollständiges Neuladen der Seite nötig
- Moderne Alternative: Fetch API
- Datenformate: JSON, XML, Plain Text

### Fetch API Grundlagen

```
// GET Request
fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));

// POST Request
fetch('https://api.example.com/data', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({
        key: 'value'
    })
})
    .then(response => response.json())
    .then(data => console.log(data));

// Mit async/await
async function fetchData() {
    try {
        const response = await
            fetch('https://api.example.com/data');
        const data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('Error:', error);
    }
}
```

## Cookies und Sessions

**Cookies** HTTP-Cookies sind kleine Datenpakete:

- Werden vom Server gesetzt
- Im Browser gespeichert
- Bei jedem Request mitgesendet
- Haben Name, Wert, Ablaufdatum und Domain

### Cookie Handling

```
// Cookie setzen
document.cookie = "username=John Doe; expires=Thu, 18
    Dec 2024 12:00:00 UTC; path=/";

// Cookie lesen
const cookies =
    document.cookie.split(';').reduce((acc, cookie)
    => {
    const [name, value] = cookie.trim().split('=');
    acc[name] = value;
    return acc;
}, {});

// Cookie loeschen
document.cookie = "username=; expires=Thu, 01 Jan 1970
    00:00:00 UTC; path=/;";
```

**Sessions** Server-seitige Speicherung von Benutzerdaten:

- Session-ID wird in Cookie gespeichert
- Daten bleiben auf dem Server
- Sicherer als Cookies für sensible Daten
- Temporär (bis Browser geschlossen wird)

## REST APIs

**REST Prinzipien** Representational State Transfer:

- Zustandslos (Stateless)
- Ressourcen-orientiert
- Einheitliche Schnittstelle
- Standard HTTP-Methoden

### HTTP-Methoden

| Methode | Verwendung |
|---|---|
| GET | Daten abrufen |
| POST | Neue Daten erstellen |
| PUT | Daten aktualisieren (komplett) |
| PATCH | Daten aktualisieren (teilweise) |
| DELETE | Daten löschen |

### REST API Implementierung mit Express

```
const express = require('express');
const app = express();
app.use(express.json());

// GET - Alle Benutzer abrufen
app.get('/api/users', (req, res) => {
    res.json(users);
});

// GET - Einzelnen Benutzer abrufen
app.get('/api/users/:id', (req, res) => {
    const user = users.find(u => u.id ===
        parseInt(req.params.id));
    if (!user) return res.status(404).send('User not
        found');
    res.json(user);
});

// POST - Neuen Benutzer erstellen
app.post('/api/users', (req, res) => {
    const user = {
        id: users.length + 1,
        name: req.body.name
    };
    users.push(user);
    res.status(201).json(user);
});

// PUT - Benutzer aktualisieren
app.put('/api/users/:id', (req, res) => {
    const user = users.find(u => u.id ===
        parseInt(req.params.id));
    if (!user) return res.status(404).send('User not
        found');

    user.name = req.body.name;
    res.json(user);
});

// DELETE - Benutzer loeschen
app.delete('/api/users/:id', (req, res) => {
    const user = users.find(u => u.id ===
        parseInt(req.params.id));
    if (!user) return res.status(404).send('User not
        found');

    const index = users.indexOf(user);
    users.splice(index, 1);
    res.json(user);
});
```

### HTTP Status Codes

| Code | Bedeutung |
|---|---|
| 200 | OK - Erfolgreich |
| 201 | Created - Ressource erstellt |
| 400 | Bad Request - Fehlerhafte Anfrage |
| 401 | Unauthorized - Nicht authentifiziert |
| 403 | Forbidden - Keine Berechtigung |
| 404 | Not Found - Ressource nicht gefunden |
| 500 | Internal Server Error - Serverfehler |

## Browser APIs und DOM

### Vordefinierte Objekte

**Browser Objekte** Im Browser stehen spezielle globale Objekte zur Verfügung:

- `window`: Browserfenster und globaler Scope
- `document`: Das aktuelle HTML-Dokument
- `navigator`: Browser-Informationen
- `location`: URL-Informationen
- `history`: Browser-Verlauf

**Document Object Model (DOM)** Das DOM ist eine Baumstruktur, die das HTML-Dokument repräsentiert:

- Jeder HTML-Tag wird zu einem Element-Knoten
- Text innerhalb von Tags wird zu Text-Knoten
- Attribute werden zu Attribut-Knoten
- Kommentare werden zu Kommentar-Knoten

**DOM Navigation** Zugriff auf DOM-Elemente:

```
// Element ueber ID finden
const elem = document.getElementById('myId');

// Elemente ueber CSS-Selektor finden
const elem1 = document.querySelector('.myClass');
const elems = document.querySelectorAll('div.myClass');

// Navigation im DOM-Baum
elem.parentNode          // Elternknoten
elem.childNodes          // Alle Kindknoten
elem.children            // Nur Element-Kindknoten
elem.firstChild          // Erster Kindknoten
elem.lastChild           // Letzter Kindknoten
elem.nextSibling         // Naechster Geschwisterknoten
elem.previousSibling     // Vorheriger Geschwisterknoten
```

**DOM Manipulation** Elemente erstellen und manipulieren:

```
// Neues Element erstellen
const newDiv = document.createElement('div');
const newText = document.createTextNode('Hello');
newDiv.appendChild(newText);

// Element einfuegen
parentElem.appendChild(newDiv);
parentElem.insertBefore(newDiv, referenceElem);

// Element entfernen
elem.remove();
parentElem.removeChild(elem);

// Attribute manipulieren
elem.setAttribute('class', 'myClass');
elem.getAttribute('class');
elem.removeAttribute('class');

// HTML/Text Inhalt
elem.innerHTML = '<span>Text</span>';
elem.textContent = 'Nur Text';
```

### Events

**Event Handling** Events sind Ereignisse, die im Browser auftreten:

- Benutzerinteraktionen (Klicks, Tastatureingaben)
- DOM-Änderungen
- Ressourcen laden
- Timer

**Event Listener** Event Listener registrieren und entfernen:

```
// Event Listener hinzufuegen
element.addEventListener('click', function(event) {
    console.log('Clicked!', event);
});

// Mit Arrow Function
element.addEventListener('click', (event) => {
    console.log('Clicked!', event);
});

// Event Listener entfernen
const handler = (event) => {
    console.log('Clicked!', event);
};
element.addEventListener('click', handler);
element.removeEventListener('click', handler);
```

**Wichtige Event-Typen**

- Maus: `click`, `dblclick`, `mousedown`, `mouseup`, `mousemove`
- Tastatur: `keydown`, `keyup`, `keypress`
- Formular: `submit`, `change`, `input`, `focus`, `blur`
- Dokument: `DOMContentLoaded`, `load`, `unload`
- Fenster: `resize`, `scroll`

**Event Bubbling und Capturing**

```
// Bubbling (default)
element.addEventListener('click', handler);

// Capturing
element.addEventListener('click', handler, true);

// Event-Ausbreitung stoppen
element.addEventListener('click', (event) => {
    event.stopPropagation();
});

// Default-Verhalten verhindern
element.addEventListener('click', (event) => {
    event.preventDefault();
});
```

### Browser Storage

**Storage APIs** Browser bieten verschiedene Möglichkeiten zur Datenspeicherung:

- `localStorage`: Permanente Speicherung
- `sessionStorage`: Temporäre Speicherung (nur für aktuelle Session)
- `cookies`: Kleine Datenpakete, die auch zum Server gesendet werden
- `indexedDB`: NoSQL-Datenbank im Browser

**LocalStorage Verwendung**

```
// Daten speichern
localStorage.setItem('key', 'value');
localStorage.setItem('user', JSON.stringify({
    name: 'John',
    age: 30
}));

// Daten abrufen
const value = localStorage.getItem('key');
const user = JSON.parse(localStorage.getItem('user'));

// Daten loeschen
localStorage.removeItem('key');
localStorage.clear();  // Alles loeschen
```

## Canvas und SVG

**Grafik im Browser** Zwei Haupttechnologien für Grafiken:

- Canvas: Pixel-basierte Grafik
- SVG: Vektor-basierte Grafik

**Canvas Grundlagen**

```
const canvas = document.querySelector('canvas');
const ctx = canvas.getContext('2d');

// Rechteck zeichnen
ctx.fillStyle = 'red';
ctx.fillRect(10, 10, 100, 50);

// Pfad zeichnen
ctx.beginPath();
ctx.moveTo(10, 10);
ctx.lineTo(50, 50);
ctx.stroke();

// Text zeichnen
ctx.font = '20px Arial';
ctx.fillText('Hello', 10, 50);
```

**SVG Manipulation**

```
// SVG-Element erstellen
const svg = document.createElementNS(
    "http://www.w3.org/2000/svg",
    "svg"
);
svg.setAttribute("width", "100");
svg.setAttribute("height", "100");

// Kreis hinzufuegen
const circle = document.createElementNS(
    "http://www.w3.org/2000/svg",
    "circle"
);
circle.setAttribute("cx", "50");
circle.setAttribute("cy", "50");
circle.setAttribute("r", "40");
circle.setAttribute("fill", "red");

svg.appendChild(circle);
```

# Browser-Technologien

## Vordefinierte Objekte

**Browser-Objekte** Browser-Objekte existieren auf der Browser-Plattform und referenzieren verschiedene Aspekte:

- **document:** Repräsentiert die aktuelle Webseite, Zugriff auf DOM
- **window:** Repräsentiert das Browserfenster, globale Funktionen/-Methoden
- **navigator:** Browser- und Geräteinformationen
- **location:** URL-Manipulation und Navigation

**document-Objekt** Wichtige Methoden des document-Objekts:

```
// Element finden
document.getElementById("id")
document.querySelector("selector")
document.querySelectorAll("selector")

// DOM manipulieren
document.createElement("tag")
document.createTextNode("text")
document.createAttribute("attr")

// Event Handler
document.addEventListener("event", handler)
```

**window-Objekt** Das window-Objekt als globaler Namespace:

```
// Globale Methoden
window.alert("message")
window.setTimeout(callback, delay)
window.requestAnimationFrame(callback)

// Eigenschaften
window.innerHeight  // Viewport Hoehe
window.pageYOffset  // Scroll Position
window.location     // URL Infos
```

## DOM (Document Object Model)

**DOM Manipulation** Grundlegende Schritte zur DOM Manipulation:

1. Element(e) finden:

```
let element = document.getElementById("id")
let elements = document.querySelectorAll(".class")
```

2. Elemente erstellen:

```
let newElem = document.createElement("div")
let text = document.createTextNode("content")
newElem.appendChild(text)
```

3. DOM modifizieren:

```
// Hinzufuegen
parent.appendChild(newElem)
parent.insertBefore(newElem, referenceNode)

// Entfernen
element.remove()
parent.removeChild(element)

// Ersetzen
parent.replaceChild(newElem, oldElem)
```

4. Attribute/Style setzen:

```
element.setAttribute("class", "highlight")
element.style.backgroundColor = "red"
```

## Event Handling

**Event Handler** Grundlegende Event Handling Schritte:

1. Event Listener registrieren:

```
element.addEventListener("event", handler)
element.removeEventListener("event", handler)
```

2. Event Handler mit Event-Objekt:

```
element.addEventListener("click", (event) => {
  console.log(event.type)     // Art des Events
  console.log(event.target)   // Ausloesendes Element
  event.preventDefault()      // Default verhindern
  event.stopPropagation()    // Bubbling stoppen
})
```

Wichtige Event-Typen:

- Mouse: click, mousedown, mouseup, mousemove
- Keyboard: keydown, keyup, keypress
- Form: submit, change, input
- Document: DOMContentLoaded, load
- Window: resize, scroll

## Formulare

### Formular Handling

1. Formular erstellen:

```
<form action="/api/submit" method="post">
  <input type="text" name="username">
  <input type="password" name="password">
  <button type="submit">Login</button>
</form>
```

2. Formular Events abfangen:

```
form.addEventListener("submit", (e) => {
  e.preventDefault()
  // Eigene Verarbeitung
})
```

3. Formulardaten verarbeiten:

```
const formData = new FormData(form)
fetch("/api/submit", {
  method: "POST",
  body: formData
})
```

## Fetch API

### HTTP Requests mit Fetch

1. GET Request:

```
fetch("/api/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error(error))
```

2. POST Request:

```
fetch("/api/create", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify(data)
})
```

3. Mit async/await:

```
async function getData() {
  try {
    const response = await fetch("/api/data")
    const data = await response.json()
    return data
  } catch (error) {
    console.error(error)
  }
}
```

## Web Storage

### Local Storage

1. Daten speichern:

```
// Speichern
localStorage.setItem('key', 'value')
localStorage.setItem('user', JSON.stringify({name:
    'Max'}))

// Lesen
const value = localStorage.getItem('key')
const user = JSON.parse(localStorage.getItem('user'))

// Loeschen
localStorage.removeItem('key')
localStorage.clear()  // Alles loeschen
```

2. Session Storage (nur für aktuelle Session):

```
sessionStorage.setItem('key', 'value')
sessionStorage.getItem('key')
sessionStorage.removeItem('key')
```

Wichtig zu beachten:

- Limit ca. 5-10 MB pro Domain
- Nur Strings speicherbar (JSON für Objekte)
- Synchroner API-Zugriff

## Cookies

### Cookie Handling

1. Cookie setzen:

```
document.cookie = "username=Max; expires=Fri, 31 Dec
    2024 23:59:59 GMT; path=/"
```

2. Cookies lesen:

```
function getCookie(name) {
  const value = `; ${document.cookie}`
  const parts = value.split(`; ${name}=`)
  if (parts.length === 2) return
      parts.pop().split(';').shift()
}
```

3. Cookie löschen:

```
document.cookie = "username=; expires=Thu, 01 Jan 1970
    00:00:00 GMT; path=/"
```

Wichtige Cookie-Attribute:

- expires/max-age: Gültigkeitsdauer
- path: Gültigkeitspfad
- secure: Nur über HTTPS
- httpOnly: Kein JavaScript-Zugriff
- samesite: Cross-Site-Cookie-Verhalten

## Web Graphics

### SVG Grafiken

1. SVG erstellen:

```
<svg width="200" height="200">
  <circle cx="100" cy="100" r="50" fill="red"/>
  <rect x="20" y="20" width="50" height="50"
      fill="blue"/>
</svg>
```

2. SVG mit JavaScript manipulieren:

```
const circle = document.querySelector('circle')
circle.setAttribute('fill', 'green')
circle.setAttribute('r', '60')

// Event Listener fuer SVG-Elemente
circle.addEventListener('click', () => {
  circle.setAttribute('fill', 'yellow')
})
```

Vorteile SVG:

- Skalierbar ohne Qualitätsverlust
- Teil des DOM (manipulierbar)
- Gute Browser-Unterstützung
- Event-Handler möglich

### Canvas API

1. Canvas erstellen:

```
<canvas id="myCanvas" width="200"
    height="200"></canvas>
```

2. Context holen und zeichnen:

```
const canvas = document.getElementById('myCanvas')
const ctx = canvas.getContext('2d')

// Rechteck zeichnen
ctx.fillStyle = 'red'
ctx.fillRect(10, 10, 100, 100)

// Pfad zeichnen
ctx.beginPath()
ctx.moveTo(10, 10)
ctx.lineTo(100, 100)
ctx.stroke()

// Text zeichnen
ctx.font = '20px Arial'
ctx.fillText('Hello', 50, 50)

// Bild zeichnen
const img = new Image()
img.onload = () => ctx.drawImage(img, 0, 0)
img.src = 'image.png'
```

3. Transformationen:

```
// Speichern des aktuellen Zustands
ctx.save()

// Transformationen
ctx.translate(100, 100)  // Verschieben
ctx.rotate(Math.PI / 4)  // Rotieren
ctx.scale(2, 2)          // Skalieren

// Zeichnen...

// Wiederherstellen des gespeicherten Zustands
ctx.restore()
```

Wichtige Canvas-Methoden:

- clearRect(): Bereich löschen
- save()/restore(): Kontext speichern/wiederherstellen
- translate()/rotate()/scale(): Transformationen
- drawImage(): Bilder zeichnen
- getImageData()/putImageData(): Pixel-Manipulation

## Browser APIs

### Geolocation API

1. Einmalige Position abfragen:

```
navigator.geolocation.getCurrentPosition(
  (position) => {
    console.log(position.coords.latitude)
    console.log(position.coords.longitude)
    console.log(position.coords.accuracy)
  },
  (error) => {
    console.error(error.message)
  },
  {
    enableHighAccuracy: true,
    timeout: 5000,
    maximumAge: 0
  }
)
```

2. Position kontinuierlich überwachen:

```
const watchId = navigator.geolocation.watchPosition(
  positionCallback,
  errorCallback,
  options
)

// Ueberwachung beenden
navigator.geolocation.clearWatch(watchId)
```

### History API

1. Navigation:

```
// Navigation
history.back()       // Eine Seite zurueck
history.forward()    // Eine Seite vor
history.go(-2)       // 2 Seiten zurueck
```

2. History Manipulation:

```
// Neuen Eintrag hinzufuegen
history.pushState(
  {page: 1},            // State-Objekt
  '',                   // Title (meist ignoriert)
  '/neue-url'           // URL
)

// Aktuellen Eintrag ersetzen
history.replaceState(
  {page: 2},
  '',
  '/andere-url'
)
```

3. Auf Änderungen reagieren:

```
window.addEventListener('popstate', (event) => {
  console.log(event.state)     // State-Objekt
  console.log(location.href)   // Aktuelle URL
})
```

### Web Workers

1. Worker erstellen:

```
// main.js
const worker = new Worker('worker.js')

worker.postMessage({data: someData})

worker.onmessage = (e) => {
  console.log('Nachricht vom Worker:', e.data)
}

// worker.js
self.onmessage = (e) => {
  // Daten verarbeiten
  const result = doSomeHeavyComputation(e.data)
  self.postMessage(result)
}
```

2. Worker beenden:

```
worker.terminate()  // Im Hauptthread
self.close()        // Im Worker
```

Wichtig:

- Worker laufen in separatem Thread
- Kein Zugriff auf DOM
- Kommunikation nur über Nachrichten
- Gut für rechenintensive Aufgaben

# UI-Bibliotheken und Komponenten

## Frameworks und Bibliotheken

**Framework vs. Bibliothek**

- **Bibliothek**: Kontrolle beim eigenen Programm, Funktionen werden nach Bedarf verwendet
- **Framework**: Rahmen für die Anwendung, Kontrolle liegt beim Framework ("Hollywood-Prinzip")

**Komponenten-basierte Entwicklung** Grundprinzipien:

- UI wird in wiederverwendbare Komponenten aufgeteilt
- Komponenten können verschachtelt werden
- Datenfluss von oben nach unten (props)
- Zustand wird in Komponenten verwaltet
- Deklarativer Ansatz: UI als Funktion des Zustands

## JSX und SJDON

**JSX** JSX ist eine Erweiterung der JavaScript-Syntax:

- Erlaubt HTML-ähnliche Syntax in JavaScript
- Muss zu JavaScript kompiliert werden
- Wird hauptsächlich in React verwendet

### JSX Syntax

```
// JSX Komponente
const Welcome = ({name}) => (
    <div className="welcome">
        <h1>Hello, {name}</h1>
        <p>Welcome to our site!</p>
    </div>
);

// Verwendung
const element = <Welcome name="Alice" />;
```

**SJDON** Simple JavaScript DOM Notation:

- Alternative zu JSX
- Verwendet reine JavaScript Arrays und Objekte
- Kein Kompilierungsschritt nötig

### SJDON Syntax

```
// SJDON Komponente
const Welcome = ({name}) => [
    "div", {className: "welcome"},
    ["h1", `Hello, ${name}`],
    ["p", "Welcome to our site!"]
];

// Verwendung
const element = [Welcome, {name: "Alice"}];
```

## SuiWeb Framework

**SuiWeb Grundkonzepte** Simple User Interface Toolkit for Web Exercises:

- Komponentenbasiert wie React
- Unterstützt JSX und SJDON
- Datengesteuert mit Props und State
- Vereinfachte Implementierung für Lernzwecke

### Komponenten in SuiWeb

```
// Einfache Komponente
const MyButton = ({onClick, children}) => [
    "button",
    {
        onclick: onClick,
        style: "background: khaki"
    },
    ...children
];

// Komponente mit State
const Counter = () => {
    const [count, setCount] = useState(0);

    return [
        "div",
        ["button",
            {onclick: () => setCount(count + 1)},
            `Count: ${count}`
        ]
    ];
};
```

### Container Komponenten

```
const MyContainer = () => {
    let initialState = { items: ["Loading..."] };
    let [state, setState] = useState(initialState);

    if (state === initialState) {
        fetchData()
            .then(items => setState({items}));
    }

    return [
        MyList,
        {items: state.items}
    ];
};
```

**State Management** Zustandsverwaltung in SuiWeb:

- `useState` Hook für lokalen Zustand
- State Updates lösen Re-Rendering aus
- Asynchrone Updates werden gequeued
- Props sind read-only

**Styling in SuiWeb** Verschiedene Möglichkeiten für Styles:

- Inline Styles als Strings
- Style-Objekte
- Arrays von Style-Objekten
- Externe CSS-Klassen

### Styling Beispiele

```
// String Style
["div", {style: "color: blue; font-size: 16px"}]

// Style Objekt
const styles = {
    container: {
        backgroundColor: "lightgray",
        padding: "10px"
    },
    text: {
        color: "darkblue",
        fontSize: "14px"
    }
};

// Kombinierte Styles
["div", {
    style: [
        styles.container,
        {borderRadius: "5px"}
    ]
}]
```

## Komponenten-Design

**Best Practices** Grundprinzipien für gutes Komponenten-Design:

- Single Responsibility Principle
- Trennung von Container und Präsentation
- Vermeidung von tiefer Verschachtelung
- Wiederverwendbarkeit fördern
- Klare Props-Schnittstelle

### Komponenten-Struktur

```
// Container Komponente
const UserContainer = () => {
    const [user, setUser] = useState(null);

    useEffect(() => {
        fetchUser().then(setUser);
    }, []);

    return [UserProfile, {user}];
};

// Praesentations-Komponente
const UserProfile = ({user}) => {
    if (!user) return ["div", "Loading..."];

    return [
        "div",
        ["h2", user.name],
        ["p", user.email],
        [UserDetails, {details: user.details}]
    ];
};
```

**Event Handling** Behandlung von Benutzerinteraktionen:

- Events als Props übergeben
- Callback-Funktionen für Events
- State Updates in Event Handlern
- Vermeidung von direkter DOM-Manipulation

### Event Handling Beispiel

```
const TodoList = () => {
    const [todos, setTodos] = useState([]);

    const addTodo = (text) => {
        setTodos([...todos, {
            id: Date.now(),
            text,
            completed: false
        }]);
    };

    const toggleTodo = (id) => {
        setTodos(todos.map(todo =>
            todo.id === id
                ? {...todo, completed: !todo.completed}
                : todo
        ));
    };

    return [
        "div",
        [TodoForm, {onSubmit: addTodo}],
        [TodoItems, {
            items: todos,
            onToggle: toggleTodo
        }]
    ];
};
```

**Optimierungen** Möglichkeiten zur Performanzverbesserung:
- Virtuelles DOM für effizientes Re-Rendering
- Batching von State Updates
- Memoization von Komponenten
- Lazy Loading von Komponenten

# UI Bibliothek

## Frameworks und Bibliotheken

### Unterschied Framework vs. Bibliothek

- **Bibliothek**:
  - Kontrolle beim eigenen Programm
  - Funktionen/Klassen der Bibliothek werden verwendet (z.B. jQuery)
- **Framework**:
  - Rahmen für die Anwendung
  - Kontrolle liegt beim Framework
  - Hollywood-Prinzip: "don't call us, we'll call you"

## Architektur

- **MVC (Model-View-Controller)**:
  - Model: Repräsentiert Daten und Geschäftslogik
  - View: Bildet UI, kommuniziert mit Controller
  - Controller: Verarbeitet Eingaben, aktualisiert Model
- **Single Page Apps (SPAs)**:
  - Vermeidet Neuladen von Seiten
  - Inhalte dynamisch nachgeladen (Ajax, REST)
  - Bessere Usability durch schnellere UI-Reaktion

## JSX und SJDON

### JSX (JavaScript XML)

- XML-Syntax in JavaScript
- Muss zu JavaScript transpiliert werden
- HTML-Tags in Kleinbuchstaben
- Eigene Komponenten mit Grossbuchstaben
- JavaScript-Code in geschweiften Klammern {...}
- Beispiel:

```
const List = ({data}) => (
    <ul>
        {data.map(item => (
            <li key={item}>{item}</li>
        ))}
    </ul>
)
```

### SJDON (Simple JavaScript DOM Notation)

- Alternative zu JSX, reines JavaScript
- Array-basierte Notation
- Erstes Element ist Elementtyp
- Attribute als Objekte
- Beispiel:

```
const List = ({data}) =>
    ["ul", ...data.map(item =>
        ["li", {key: item}, item]
    )]
```

## Komponenten

### React/SuiWeb Komponenten

- Funktionskomponenten definieren:

```
const MyComponent = ({prop1, prop2}) => (
    ["div",
        ["h1", prop1],
        ["p", prop2]
    ]
)
```

- Props sind readonly
- Zustand mit useState Hook:

```
const [state, setState] = useState(initialValue)
```

- Event Handler definieren:

```
const handler = () => setState(newValue)
```

## State Management

### Zustandsverwaltung

- Kontrollierte Eingabefelder:

```
const [text, setText] = useState("")
["input", {
    value: text,
    oninput: e => setText(e.target.value)
}]
```

- Container Components:
  - Verwalten Zustand
  - Holen Daten (z.B. API-Calls)
  - Geben Daten via props weiter
- Effect Hook für Seiteneffekte:

```
useEffect(() => {
    // Nach Rendering ausgeführt
    fetchData().then(...)
}, [dependencies])
```

## Best Practices

- Komponenten klein und wiederverwendbar halten
- Zustand in wenigen Container-Komponenten konzentrieren
- Props für Datenweitergabe nach unten
- Events für Kommunikation nach oben
- Module für bessere Separation of Concerns
- Deklarativer statt imperativer Code

# Wrap-up

## React vs SuiWeb

### React.js Key Features

- JavaScript library for building user interfaces
- Released by Facebook in 2013
- Key principles:
  - Declarative
  - Component-Based
  - Learn Once, Write Anywhere
  - Virtual DOM for efficient rendering
  - Unidirectional data flow

**React Components** Component patterns frequently used in React:

1. Function Components (modern approach):

```
const HelloComponent = (props) => {
    return (<div>Hello {props.name}</div>)
}
```

2. Class Components (legacy):

```
class HelloComponent extends React.Component {
    render() {
        return <div>Hello {this.props.name}</div>
    }
}
```

3. State Management with Hooks:

```
const Counter = () => {
    const [state, setState] = useState(1)
    const handler = () => setState(state + 1)
    return (
        <button onClick={handler}>Count: {state}</button>
    )
}
```

## Course Overview

### Main Topics

1. JavaScript Core Concepts
   - Basic language features
   - Objects and Arrays
   - Functions and Prototypes
   - Asynchronous Programming
2. Browser Technologies
   - DOM Manipulation
   - Events
   - Web APIs
3. Client-Side Applications
   - Component Design
   - UI Libraries/Frameworks
   - Modern Web Development

## Advanced Web Topics

### Further Areas of Web Development

- Mobile Development
  - Responsive Design
  - React Native
  - Progressive Web Apps
- Performance Optimization
  - WebAssembly (WASM)
  - Code Splitting
  - Client-side Caching
- Alternative Technologies
  - TypeScript
  - WebAssembly
  - Functional Programming

## Course Goals

### Key Learning Outcomes

- Solid understanding of JavaScript fundamentals
- Proficiency in core web technologies
- Understanding of modern web development practices
- Foundation for advanced web development topics

Depth of Topics Topics covered in the course are divided into:

- In-depth topics: Require thorough understanding and practical application
- Overview topics: Require basic understanding of concepts and use cases

Detailed lists of what falls into each category are provided in the course materials.

## Web Development Landscape

### Current State of Web Development

- JavaScript remains dominant web programming language
- React continues as leading UI library
- Growing ecosystem of tools and frameworks
- Increasing focus on performance and user experience
- Emergence of new patterns and technologies

## Weekly Schedule

## Additional Resources

For further learning:

- HTML/CSS: `https://developer.mozilla.org/docs/Web`
- React: `https://react.dev`
- JavaScript: `https://eloquentjavascript.net`
- TypeScript: `https://www.typescriptlang.org`
- WebAssembly: `https://webassembly.org`