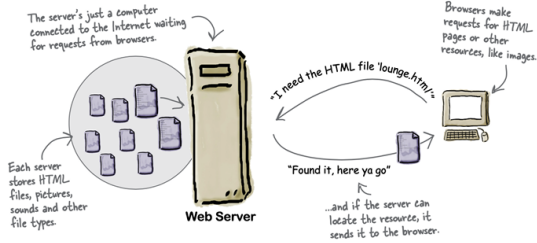


WEB-Architektur Client-Server-Modell:

- 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-OperatorMit `typeof` kann der Typ eines Wertes ermittelt werden:

```
1 typeof 42           // 'number'
2 typeof 42n          // 'bigint'
3 typeof "text"       // 'string'
4 typeof true         // 'boolean'
5 typeof undefined    // 'undefined'
6 typeof null         // 'object' (!)
7 typeof {}           // 'object'
8 typeof []           // 'object'
9 typeof (() => {})     // 'function'
10 typeof Infinity    // 'number'
11 typeof NaN         // 'number' !!
12 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)

```
1 5 == "5"           // true  (Typumwandlung)
2 5 === "5"          // false (keine Typumwandlung)
3 null == undefined  // true
4 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

```
1 // If-Statement
2 if (condition) {
3   // code
4 } else if (otherCondition) {
5   // code
6 } else {
7   // code
8 }
9
10 // Switch Statement
11 switch(value) {
12   case 1:
13     // code
14     break;
15   case 2:
16     // code
17     break;
18   default:
19     // code
20 }
21
22 // Loops
23 for (let i = 0; i < n; i++) { }
24 while (condition) { }
25 do { } while (condition);
26 for (let item of array) { }
27 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:

```
1 // Funktionsdeklaration
2 function add(a, b) {
3     return a + b;
4 }
5
6 // Funktionsausdruck
7 const multiply = function(a, b) {
8     return a * b;
9 };
10
11 // Arrow Function
12 const subtract = (a, b) => a - b;
13
14 // Arrow Function mit Block
15 const divide = (a, b) => {
16     if (b === 0) throw new Error('Division by zero');
17     return a / b;
18 };
```

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-Objektlitereale

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

```
1 > JSON.stringify({type: "cat", name: "Mimi", age: 3})
2 '{"type":"cat", "name":"Mimi", "age":3}'
3 > JSON.parse('{"type": "cat", "name": "Mimi", "age":
4 3}')
5 {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

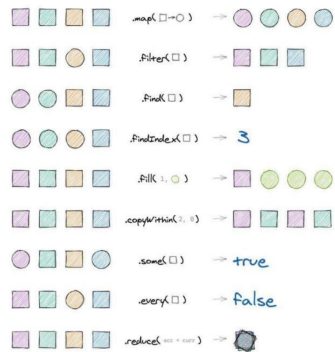
Objekte erstellen und manipulieren

```
1 // Objekt-Literal
2 const person = {
3     name: "Alice",
4     age: 30,
5     greet() {
6         return "Hello, I am" + this.name;
7     }
8 };
9
10 // Eigenschaften manipulieren
11 person.job = "Developer"; // hinzufügen
12 delete person.age; // löschen
13 "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

```
1 const arr = [1, 2, 3];
2 arr.push(4); // [1,2,3,4]
3 arr.pop(); // [1,2,3]
4 arr.unshift(0); // [0,1,2,3]
5 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

```
1 > const add = (x, y) => x + y
2 > add.doc = "This function adds two values"
3 > add(3,4)
4 7
5 > add.doc
6 '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

```
1 const car = { //car-lib.js
2     brand: 'Ford',
3     model: 'Fiesta'
4 }
5 module.exports = car
6 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

```
1 function Employee (name, salary) {
2     Person.call(this, name)
3     this.salary = salary
4 }
5 Employee.prototype = new Person()
6 Employee.prototype.constructor = Employee
7 let e17 = new Employee("Mary", 7000)
8 console.log(e17.toString()) // Person with name 'Mary'
9 console.log(e17.salary) // 7000
```

Klassen

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

```
1 class Person {
2     constructor (name) {
3         this.name = name
4     }
5     toString () {
6         return `Person with name '${this.name}'`
7     }
8 }
9 let p35 = new Person("John")
10 console.log(p35.toString()) // Person with name 'John'
```

Vererbung

```
1 class Employee extends Person {
2     constructor (name, salary) {
3         super(name)
4         this.salary = salary
5     }
6     toString () {
7         return `${super.toString()} and salary
8             ${this.salary}
9     }
10 }
11 let e17 = new Employee("Mary", 7000);
12 console.log(e17.toString()) /* Person with name 'Mary'
    and salary 7000 */
13 console.log(e17.salary) /* 7000 */
```

Getter und Setter

```
1 class PartTimeEmployee extends Employee {
2   constructor (name, salary, percentage) {
3     super(name, salary)
4     this.percentage = percentage
5   }
6   get salary100 () { return this.salary * 100 /
7     this.percentage }
8   set salary100 (amount) { this.salary = amount *
9     this.percentage / 100 }
10 }
11 let e18 = new PartTimeEmployee("Bob", 4000, 50)
12 console.log(e18.salary100) /* -> 8000 */
13 e18.salary100 = 9000
14 console.log(e18.salary) /* \ 4500 */
```

Filesystem

Pfad der Datei Um Pfad-Informationen einer Datei zu ermitteln muss man dies mit `require('path')` machen.

```
1 const path = require('path')
2 const notes = '/users/bkrt/notes.txt'
3 path.dirname(notes) /* /users/bkrt */
4 path.basename(notes) /* notes.txt */
5 path.extname(notes) /* .txt */
6 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

```
1 const fs = require('fs')
2 fs.stat('test.txt', (err, stats) => {
3   if (err) {
4     console.error(err)
5     return
6   }
7   stats.isFile() /* true */
8   stats.isDirectory() /* false */
9   stats.isSymbolicLink() /* false */
10 stats.size /* 1024000 = ca 1MB */
11 })
```

Dateien lesen und schreiben

```
1 const fs = require('fs')
2 fs.readFile('/etc/hosts', 'utf8', (err, data) => {
3   if (err) throw err
4   console.log(data)
5 })
6
7 const content = 'Node was here!'
8 fs.writeFile('/Users/bkrt/test.txt', content, (err) => {
9   if (err) {
10     console.error(`Failed to write file: ${err}`)
11     return
12   } // file written successfully
13 })
```

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.

```
1 document.getElementById('button').addEventListener('click',
2   () => {
3     //item clicked
4   })
```

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

```
1 setTimeout(() => {
2   /* runs after 50 milliseconds */
3 }, 50)
```

SetInterval

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

```
1 const id = setInterval(() => {
2   // runs every 2 seconds
3 }, 2000)
4 clearInterval(id)
```

SetImmediate

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

```
1 setImmediate(() => {
2   console.log('immediate')
3 })
```

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

```
1 const EventEmitter = require('events')
2 const door = new EventEmitter()
3
4 door.on('open', () => {
5   console.log('Door was opened')
6 })
```

Event auslösen

```
1 door.on('open', (speed) => {
2   console.log(`Door was opened, speed: ${speed ||
3     'unknown'}`)
4 })
5 door.emit('open')
6 door.emit('open', 'slow')
```

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

```
1 function readFilePromise(file) {
2   let promise = new Promise(function
3     resolver(resolve, reject) {
4       fs.readFile(file, 'utf8', (err, data) => {
5         if (err) reject(err);
6         else resolve(data);
7       });
8     });
9   return promise;
10 }
```

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

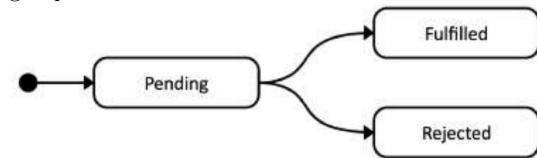
Aufruf neu:

```
1 readFilePromise('/etc/hosts')
2   .then(console.log)
3   .catch(() => {
4     console.log("Error reading file")
5   })
```

Promise-Zustände

- pending: Ausgangszustand
- 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

ASYNCH/AWAIT

```
1 /* Bekanntes Beispiel */
2 const readHosts = () => {
3   readFilePromise('/etc/hosts')
4     .then(console.log)
5     .catch(() => {
6       console.log("Error reading file")
7     })
8 }
9 /* Mit async/await */
10 const readHosts = async () => {
11   try {
12     console.log(await
13       readFilePromise('/etc/hosts'))
14   }
15   catch (err) {
16     console.log("Error reading file")
17   }
18 }
```

Beispiel 2:

```
1 function resolveAfter2Seconds(x) {
2   return new Promise(resolve => {
3     setTimeout(() => {
4       resolve(x)
5     }, 2000)
6   })
7 }
8 async function add1(x) {
9   var a = resolveAfter2Seconds(20)
10  var b = resolveAfter2Seconds(30)
11  return x + await a + await b
12 }
13 add1(10).then(console.log)
```

Promise Erstellung und Verwendung

```
1 // Promise erstellen
2 const myPromise = new Promise((resolve, reject) => {
3   // Asynchrone Operation
4   setTimeout(() => {
5     if (/* erfolg */) {
6       resolve(result);
7     } else {
8       reject(error);
9     }
10  }, 1000);
11 });
12
13 // Promise verwenden
14 myPromise
15   .then(result => {
16     // Erfolgsfall
17   })
18   .catch(error => {
19     // Fehlerfall
20   })
21   .finally(() => {
22     // Wird immer ausgeführt
23   });
24
25 // Async/Await Syntax
26 async function myAsync() {
27   try {
28     const result = await myPromise;
29     // Erfolgsfall
30   } catch (error) {
31     // Fehlerfall
32   }
33 }
```

Module System JavaScript verwendet verschiedene Modulsysteme:

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

Module Import/Export

```
1 // CommonJS (Node.js)
2 const fs = require('fs');
3 module.exports = { /* ... */ };
4
5 // ES Modules
6 import { function1, function2 } from './module.js';
7 export const variable = 42;
8 export default class MyClass { /* ... */ }
```

Error Handling

```
1 try {
2   // Code der Fehler werfen konnte
3   throw new Error('Something went wrong');
4 } catch (error) {
5   // Fehlerbehandlung
6   console.error(error.message);
7 } finally {
8   // Wird immer ausgeführt
9   cleanup();
10 }
```

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

```
1 const {createServer} = require("http")
2 let server = createServer((request, response) => {
3   response.writeHead(200, {"Content-Type":
4     "text/html"})
5   response.write(`
6     <h1>Hello!</h1>
7     <p>You asked for
8       <code>${request.url}</code></p>`)
9   response.end()
10 })
server.listen(8000)
console.log("Listening! (port 8000)")
```

Einfacher WebClient

```
1 const {request} = require("http")
2 let requestStream = request({
3   hostname: "eloquentjavascript.net",
4   path: "/20_node.html",
5   method: "GET"
6 }, {headers: {Accept: "text/html"}
7 }, response => {
8   console.log("Server responded with status
9     code", response.statusCode)
10 })
requestStream.end()
```

Server und Client mit Streams

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

```
1 const {request} = require("http")
2 let rq = request({
3   hostname: "localhost",
4   port: 8000,
5   method: "POST"
6 }, response => {
7   response.on("data", chunk =>
8     process.stdout.write(chunk.toString()));
9 })
10 rq.write("Hello server\n")
11 rq.write("And good bye\n")
12 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.

```
1 $ mkdir myapp
2 $ cd myapp
3 $ npm init
4 $ npm install express --save
```

Beispiel: Express Server

```
1 const express = require('express')
2 const app = express()
3 const port = 3000
4 app.get('/', (req, res) => {
5   res.send('Hello World!')
6 })
7 app.listen(port, () => {
8   console.log(`Example app listening at
9     http://localhost:${port}`)
10 })
```

Routing

```
1 app.get('/', function (req, res) {
2   res.send('Hello World!')
3 })
4 app.post('/', function (req, res) {
5   res.send('Got a POST request')
6 })
7 app.put('/user', function (req, res) {
8   res.send('Got a PUT request at /user')
9 })
10 app.delete('/user', function (req, res) {
11   res.send('Got a DELETE request at /user')
12 })
```

Jasmine (Testing)

Beispiel (zugehörige Tests)

```
1  /* PlayerSpec.js - Auszug */
2  describe("when song has been paused", function() {
3      beforeEach(function() {
4          player.play(song)
5          player.pause()
6      })
7      it("should indicate that the song is currently
8         paused", function() {
9          expect(player.isPlaying).toBeFalsy()
10         /* demonstrates use of 'not' with a custom
11            matcher */
12         expect(player).not.toBePlaying(song)
13     })
14     it("should be possible to resume", function() {
15         player.resume()
16         expect(player.isPlaying).toBeTruthy()
17         expect(player.currentlyPlayingSong)
18             .toEqual(song)
19     })
20 })
```

JASMINE: MATCHER

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

JASMINE: TESTS DURCHFÜHREN

```
1  $ npx jasmine
2  Randomized with seed 03741
3  Started
4  .....
5  5 specs, 0 failures
6  Finished in 0.014 seconds
7  Randomized with seed 03741
8  (jasmine --random=true --seed=03741)
```


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:

```
1 // Element ueber ID finden
2 const elem = document.getElementById('myId');
3
4 // Elemente ueber CSS-Selektor finden
5 const elem1 = document.querySelector('.myClass');
6 const elems = document.querySelectorAll('div.myClass');
7
8 // Navigation im DOM-Baum
9 elem.parentNode // Elternknoten
10 elem.childNodes // Alle Kindknoten
11 elem.children // Nur Element-Kindknoten
12 elem.firstChild // Erster Kindknoten
13 elem.lastChild // Letzter Kindknoten
14 elem.nextSibling // Naechster Geschwisterknoten
15 elem.previousSibling // Vorheriger Geschwisterknoten
```

DOM Manipulation Elemente erstellen und manipulieren:

```
1 // Neues Element erstellen
2 const newDiv = document.createElement('div');
3 const newText = document.createTextNode('Hello');
4 newDiv.appendChild(newText);
5
6 // Element einfüegen
7 parentElem.appendChild(newDiv);
8 parentElem.insertBefore(newDiv, referenceElem);
9
10 // Element entfernen
11 elem.remove();
12 parentElem.removeChild(elem);
13
14 // Attribute manipulieren
15 elem.setAttribute('class', 'myClass');
16 elem.getAttribute('class');
17 elem.removeAttribute('class');
18
19 // HTML/Text Inhalt
20 elem.innerHTML = '<span>Text</span>';
21 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:

```
1 // Event Listener hinzufuegen
2 element.addEventListener('click', function(event) {
3     console.log('Clicked!', event);
4 });
5
6 // Mit Arrow Function
7 element.addEventListener('click', (event) => {
8     console.log('Clicked!', event);
9 });
10
11 // Event Listener entfernen
12 const handler = (event) => {
13     console.log('Clicked!', event);
14 };
15 element.addEventListener('click', handler);
16 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

```
1 // Bubbling (default)
2 element.addEventListener('click', handler);
3
4 // Capturing
5 element.addEventListener('click', handler, true);
6
7 // Event-Ausbreitung stoppen
8 element.addEventListener('click', (event) => {
9     event.stopPropagation();
10 });
11
12 // Default-Verhalten verhindern
13 element.addEventListener('click', (event) => {
14     event.preventDefault();
15 });
```

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

```
1 // Daten speichern
2 localStorage.setItem('key', 'value');
3 localStorage.setItem('user', JSON.stringify({
4     name: 'John',
5     age: 30
6 }));
7
8 // Daten abrufen
9 const value = localStorage.getItem('key');
10 const user = JSON.parse(localStorage.getItem('user'));
11
12 // Daten loeschen
13 localStorage.removeItem('key');
14 localStorage.clear(); // Alles loeschen
```

Canvas und SVG

Grafik im Browser Zwei Haupttechnologien für Grafiken:

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

Canvas Grundlagen

```
1 const canvas = document.querySelector('canvas');
2 const ctx = canvas.getContext('2d');
3
4 // Rechteck zeichnen
5 ctx.fillStyle = 'red';
6 ctx.fillRect(10, 10, 100, 50);
7
8 // Pfad zeichnen
9 ctx.beginPath();
10 ctx.moveTo(10, 10);
11 ctx.lineTo(50, 50);
12 ctx.stroke();
13
14 // Text zeichnen
15 ctx.font = '20px Arial';
16 ctx.fillText('Hello', 10, 50);
```

SVG Manipulation

```
1 // SVG-Element erstellen
2 const svg = document.createElementNS(
3     "http://www.w3.org/2000/svg",
4     "svg"
5 );
6 svg.setAttribute("width", "100");
7 svg.setAttribute("height", "100");
8
9 // Kreis hinzufuegen
10 const circle = document.createElementNS(
11     "http://www.w3.org/2000/svg",
12     "circle"
13 );
14 circle.setAttribute("cx", "50");
15 circle.setAttribute("cy", "50");
16 circle.setAttribute("r", "40");
17 circle.setAttribute("fill", "red");
18
19 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:

```
1 // Element finden
2 document.getElementById("id")
3 document.querySelector("selector")
4 document.querySelectorAll("selector")
5
6 // DOM manipulieren
7 document.createElement("tag")
8 document.createTextNode("text")
9 document.setAttribute("attr")
10
11 // Event Handler
12 document.addEventListener("event", handler)
```

window-Objekt Das window-Objekt als globaler Namespace:

```
1 // Globale Methoden
2 window.alert("message")
3 window.setTimeout(callback, delay)
4 window.requestAnimationFrame(callback)
5
6 // Eigenschaften
7 window.innerHeight // Viewport Hoehe
8 window.pageYOffset // Scroll Position
9 window.location // URL Infos
```

DOM (Document Object Model)

DOM Manipulation Grundlegende Schritte zur DOM Manipulation:

1. Element(e) finden:

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

2. Elemente erstellen:

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

3. DOM modifizieren:

```
1 // Hinzufuegen
2 parent.appendChild(newElem)
3 parent.insertBefore(newElem, referenceNode)
4
5 // Entfernen
6 element.remove()
7 parent.removeChild(element)
8
9 // Ersetzen
10 parent.replaceChild(newElem, oldElem)
```

4. Attribute/Style setzen:

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

Event Handling

Event Handler Grundlegende Event Handling Schritte:

1. Event Listener registrieren:

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

2. Event Handler mit Event-Objekt:

```
1 element.addEventListener("click", (event) => {
2   console.log(event.type) // Art des Events
3   console.log(event.target) // Ausloesendes Element
4   event.preventDefault() // Default verhindern
5   event.stopPropagation() // Bubbling stoppen
6 })
```

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:

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

2. Formular Events abfangen:

```
1 form.addEventListener("submit", (e) => {
2   e.preventDefault()
3   // Eigene Verarbeitung
4 })
```

3. Formulardaten verarbeiten:

```
1 const formData = new FormData(form)
2 fetch("/api/submit", {
3   method: "POST",
4   body: formData
5 })
```

Fetch API

HTTP Requests mit Fetch 1. GET Request:

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

2. POST Request:

```
1 fetch("/api/create", {
2   method: "POST",
3   headers: {
4     "Content-Type": "application/json"
5   },
6   body: JSON.stringify(data)
7 })
```

3. Mit async/await:

```
1 async function getData() {
2   try {
3     const response = await fetch("/api/data")
4     const data = await response.json()
5     return data
6   } catch (error) {
7     console.error(error)
8   }
9 }
```


Web Storage

Local Storage 1. Daten speichern:

```
1 // Speichern
2 localStorage.setItem('key', 'value')
3 localStorage.setItem('user', JSON.stringify({name:
4   'Max'}))
5
6 // Lesen
7 const value = localStorage.getItem('key')
8 const user = JSON.parse(localStorage.getItem('user'))
9
10 // Loeschen
11 localStorage.removeItem('key')
12 localStorage.clear() // Alles loeschen
```

2. Session Storage (nur für aktuelle Session):

```
1 sessionStorage.setItem('key', 'value')
2 sessionStorage.getItem('key')
3 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:

```
1 document.cookie = "username=Max; expires=Fri, 31 Dec
2   2024 23:59:59 GMT; path=/"
```

2. Cookies lesen:

```
1 function getCookie(name) {
2   const value = `; ${document.cookie}`
3   const parts = value.split('; ${name}=')
4   if (parts.length === 2) return
5     parts.pop().split(';').shift()
6 }
```

3. Cookie löschen:

```
1 document.cookie = "username=; expires=Thu, 01 Jan 1970
2   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:

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

2. SVG mit JavaScript manipulieren:

```
1 const circle = document.querySelector('circle')
2 circle.setAttribute('fill', 'green')
3 circle.setAttribute('r', '60')
4
5 // Event Listener fuer SVG-Elemente
6 circle.addEventListener('click', () => {
7   circle.setAttribute('fill', 'yellow')
8 })
```

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

Canvas API 1. Canvas erstellen:

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

2. Context holen und zeichnen:

```
1 const canvas = document.getElementById('myCanvas')
2 const ctx = canvas.getContext('2d')
3
4 // Rechteck zeichnen
5 ctx.fillStyle = 'red'
6 ctx.fillRect(10, 10, 100, 100)
7
8 // Pfad zeichnen
9 ctx.beginPath()
10 ctx.moveTo(10, 10)
11 ctx.lineTo(100, 100)
12 ctx.stroke()
13
14 // Text zeichnen
15 ctx.font = '20px Arial'
16 ctx.fillText('Hello', 50, 50)
17
18 // Bild zeichnen
19 const img = new Image()
20 img.onload = () => ctx.drawImage(img, 0, 0)
21 img.src = 'image.png'
```

3. Transformationen:

```
1 // Speichern des aktuellen Zustands
2 ctx.save()
3
4 // Transformationen
5 ctx.translate(100, 100) // Verschieben
6 ctx.rotate(Math.PI / 4) // Rotieren
7 ctx.scale(2, 2) // Skalieren
8
9 // Zeichnen...
10
11 // Wiederherstellen des gespeicherten Zustands
12 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:

```
1 navigator.geolocation.getCurrentPosition(
2   (position) => {
3     console.log(position.coords.latitude)
4     console.log(position.coords.longitude)
5     console.log(position.coords.accuracy)
6   },
7   (error) => {
8     console.error(error.message)
9   },
10  {
11    enableHighAccuracy: true,
12    timeout: 5000,
13    maximumAge: 0
14  }
15 )
```

2. Position kontinuierlich überwachen:

```
1 const watchId = navigator.geolocation.watchPosition(
2   positionCallback,
3   errorCallback,
4   options
5 )
6
7 // Ueberwachung beenden
8 navigator.geolocation.clearWatch(watchId)
```

History API 1. Navigation:

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

2. History Manipulation:

```
1 // Neuen Eintrag hinzufuegen
2 history.pushState(
3   {page: 1}, // State-Objekt
4   '', // Title (meist ignoriert)
5   '/neue-url' // URL
6 )
7
8 // Aktuellen Eintrag ersetzen
9 history.replaceState(
10  {page: 2},
11  '',
12  '/andere-url'
13 )
```

3. Auf Änderungen reagieren:

```
1 window.addEventListener('popstate', (event) => {
2   console.log(event.state) // State-Objekt
3   console.log(location.href) // Aktuelle URL
4 })
```

Web Workers 1. Worker erstellen:

```
1 // main.js
2 const worker = new Worker('worker.js')
3
4 worker.postMessage({data: someData})
5
6 worker.onmessage = (e) => {
7   console.log('Nachricht vom Worker:', e.data)
8 }
9
10 // worker.js
11 self.onmessage = (e) => {
12   // Daten verarbeiten
13   const result = doSomeHeavyComputation(e.data)
14   self.postMessage(result)
15 }
```

2. Worker beenden:

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

Wichtig:

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

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

```
1 <!-- HTML Form -->
2 <form action="/submit" method="POST">
3   <label for="username">Username:</label>
4   <input type="text" id="username" name="username">
5
6   <label for="password">Password:</label>
7   <input type="password" id="password"
8     name="password">
9
10  <button type="submit">Login</button>
11 </form>
12
13 <!-- JavaScript Handler -->
14 form.addEventListener('submit', (event) => {
15   event.preventDefault(); // Verhindert
16   // Standard-Submit
17
18   const formData = new FormData(form);
19   // Zugriff auf Formular-Daten
20   const username = formData.get('username');
21   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

```
1 // GET Request
2 fetch('https://api.example.com/data')
3   .then(response => response.json())
4   .then(data => console.log(data))
5   .catch(error => console.error('Error:', error));
6
7 // POST Request
8 fetch('https://api.example.com/data', {
9   method: 'POST',
10  headers: {
11    'Content-Type': 'application/json',
12  },
13  body: JSON.stringify({
14    key: 'value'
15  })
16 })
17   .then(response => response.json())
18   .then(data => console.log(data));
19
20 // Mit async/await
21 async function fetchData() {
22   try {
23     const response = await
24       fetch('https://api.example.com/data');
25     const data = await response.json();
26     console.log(data);
27   } catch (error) {
28     console.error('Error:', error);
29   }
30 }
```

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

```
1 // Cookie setzen
2 document.cookie = "username=John Doe; expires=Thu, 18
   Dec 2024 12:00:00 UTC; path=/";
3
4 // Cookie lesen
5 const cookies =
   document.cookie.split(';').reduce((acc, cookie)
   => {
6     const [name, value] = cookie.trim().split('=');
7     acc[name] = value;
8     return acc;
9 }, {});
10
11 // Cookie loeschen
12 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

```
1 const express = require('express');
2 const app = express();
3 app.use(express.json());
4
5 // GET - Alle Benutzer abrufen
6 app.get('/api/users', (req, res) => {
7     res.json(users);
8 });
9
10 // GET - Einzelnen Benutzer abrufen
11 app.get('/api/users/:id', (req, res) => {
12     const user = users.find(u => u.id ===
13         parseInt(req.params.id));
14     if (!user) return res.status(404).send('User not
15         found');
16     res.json(user);
17 });
18
19 // POST - Neuen Benutzer erstellen
20 app.post('/api/users', (req, res) => {
21     const user = {
22         id: users.length + 1,
23         name: req.body.name
24     };
25     users.push(user);
26     res.status(201).json(user);
27 });
28
29 // PUT - Benutzer aktualisieren
30 app.put('/api/users/:id', (req, res) => {
31     const user = users.find(u => u.id ===
32         parseInt(req.params.id));
33     if (!user) return res.status(404).send('User not
34         found');
35
36     user.name = req.body.name;
37     res.json(user);
38 });
39
40 // DELETE - Benutzer loeschen
41 app.delete('/api/users/:id', (req, res) => {
42     const user = users.find(u => u.id ===
43         parseInt(req.params.id));
44     if (!user) return res.status(404).send('User not
45         found');
46
47     const index = users.indexOf(user);
48     users.splice(index, 1);
49     res.json(user);
50 });
```

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

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

Übungsaufgaben

JavaScript Grundlagen

Datentypen und Operatoren **Aufgabe 1:** Was ist die Ausgabe folgender Ausdrücke?

```
1 typeof NaN
2 typeof []
3 typeof null
4 typeof undefined
5 [] == false
6 null === undefined
7 "5" + 3
8 "5" - 3
```

Lösung:

```
1 "number" // NaN ist vom Typ number
2 "object" // Arrays sind Objekte
3 "object" // null ist historisch ein Objekt
4 "undefined" // undefined ist ein eigener Typ
5 true // [] wird zu 0 konvertiert
6 false // === vergleicht auch Typen
7 "53" // String-Konkatenation
8 2 // Numerische Subtraktion
```

Funktionen und Scoping **Aufgabe 2:** Was ist die Ausgabe dieses Codes?

```
1 let x = 1;
2 const f = () => {
3   let x = 2;
4   return {
5     getX: () => x,
6     setX: (val) => { x = val; }
7   };
8 };
9 const obj = f();
10 console.log(x); // ?
11 console.log(obj.getX()); // ?
12 obj.setX(3);
13 console.log(obj.getX()); // ?
14 console.log(x); // ?
```

Lösung:

```
1 // Globales x bleibt 1
2 // Closure hat Zugriff auf lokales x
3 // Lokales x wird auf 3 gesetzt
4 // Globales x bleibt unverändert
```

DOM und Events

DOM Manipulation **Aufgabe 3:** Erstellen Sie eine Funktion, die eine ToDo-Liste verwaltet.

```
1 function createToDoList(containerId) {
2   // Container finden
3   const container =
4     document.getElementById(containerId);
5
6   // Input und Liste erstellen
7   const input = document.createElement('input');
8   const button = document.createElement('button');
9   const list = document.createElement('ul');
10
11   // Button konfigurieren
12   button.textContent = 'Add';
13   button.onclick = () => {
14     if (input.value.trim()) {
15       const li = document.createElement('li');
16       li.textContent = input.value;
17       list.appendChild(li);
18       input.value = '';
19     }
20   };
21
22   // Elemente zusammenfügen
23   container.appendChild(input);
24   container.appendChild(button);
25   container.appendChild(list);
26 }
```

Event Handling **Aufgabe 4:** Implementieren Sie einen Klick-Zähler mit Event Delegation.

```
1 document.getElementById('container').addEventListener('click', (e) => {
2   if (e.target.matches('button')) {
3     const count =
4       parseInt(e.target.dataset.count) || 0
5     + 1;
6     e.target.dataset.count = count;
7     e.target.textContent = `Clicked ${count}
8       times`;
9   }
10 });
```

Client-Server Kommunikation

Fetch API **Aufgabe 5:** Implementieren Sie eine Funktion für API-Requests.

```
1 async function apiRequest(url, method = 'GET', data =
2   null) {
3   const options = {
4     method,
5     headers: {
6       'Content-Type': 'application/json'
7     }
8   };
9
10  if (data) {
11    options.body = JSON.stringify(data);
12  }
13
14  try {
15    const response = await fetch(url, options);
16    if (!response.ok) {
17      throw new Error(`HTTP error:
18        ${response.status}`);
19    }
20    return await response.json();
21  } catch (error) {
22    console.error('API request failed:', error);
23    throw error;
24  }
```

Formular-Validierung **Aufgabe 6:** Erstellen Sie eine Formular-Validierung.

```
1 function validateForm(formId) {
2   const form = document.getElementById(formId);
3
4   form.addEventListener('submit', (e) => {
5     e.preventDefault();
6
7     const formData = new FormData(form);
8     const errors = [];
9
10    // Email validieren
11    const email = formData.get('email');
12    if (!email.includes('@')) {
13      errors.push('Invalid email');
14    }
15
16    // Passwort validieren
17    const password = formData.get('password');
18    if (password.length < 8) {
19      errors.push('Password too short');
20    }
21
22    if (errors.length === 0) {
23      // Form submission logic
24      console.log('Form valid, submitting...');
25      form.submit();
26    } else {
27      alert(errors.join('\n'));
28    }
29  });
30 }
```

UI-Komponenten

SuiWeb Komponente **Aufgabe 7:** Erstellen Sie eine Counter-Komponente mit SuiWeb.

```
1 const Counter = () => {
2   const [count, setCount] = useState(0);
3
4   return [
5     "div",
6     ["h2", `Count: ${count}`],
7     ["button",
8       {onclick: () => setCount(count + 1)},
9       "Increment"
10    ],
11    ["button",
12      {onclick: () => setCount(count - 1)},
13      "Decrement"
14    ]
15  ];
16};
```

Container Component **Aufgabe 8:** Implementieren Sie eine UserList-Komponente.

```
1 const UserList = () => {
2   const [users, setUsers] = useState([]);
3   const [loading, setLoading] = useState(true);
4
5   if (loading) {
6     fetchUsers()
7       .then(data => {
8         setUsers(data);
9         setLoading(false);
10      })
11     .catch(error => {
12       console.error(error);
13       setLoading(false);
14     });
15   }
16
17   if (loading) {
18     return ["div", "Loading..."];
19   }
20
21   return [
22     "div",
23     ["h2", "Users"],
24     ["ul",
25       ...users.map(user =>
26         ["li", `${user.name} (${user.email})`]
27       )
28     ]
29   ];
30};
```

Theoriefragen

Konzeptfragen 1. Erklären Sie den Unterschied zwischen == und === in JavaScript.

Antwort: == vergleicht Werte mit Typumwandlung, === vergleicht Werte und Typen ohne Umwandlung.

2. Was ist Event Bubbling?

Antwort: Events werden von dem auslösenden Element durch den DOM-Baum nach oben weitergeleitet.

3. Was ist der Unterschied zwischen localStorage und sessionStorage?

Antwort: localStorage persistiert Daten auch nach Schließen des Browsers, sessionStorage nur während der Session.

4. Erklären Sie den Unterschied zwischen synchronem und asynchronem Code.

Antwort: Synchroner Code wird sequentiell ausgeführt, asynchroner Code ermöglicht parallele Ausführung ohne Blockierung.

Praktische Aufgaben

Implementierungsaufgaben 1. Implementieren Sie eine Funktion zur Deep Copy von Objekten.

2. Erstellen Sie eine Funktion, die prüft ob ein String ein Palindrom ist.

3. Implementieren Sie eine debounce-Funktion.

4. Erstellen Sie eine Komponente für einen Image Slider.

Debugging-Aufgaben 1. Finden Sie den Fehler im folgenden Code:

```
1 const getData = () => {
2   fetch('api/data')
3     .then(response => response.json())
4     .then(data => {
5       return data;
6     });
7 }
8 // Warum kommt undefined zurueck?
```

Antwort: Die Funktion hat kein explizites return Statement. Sie sollte entweder async/await verwenden oder die Promise zurückgeben.