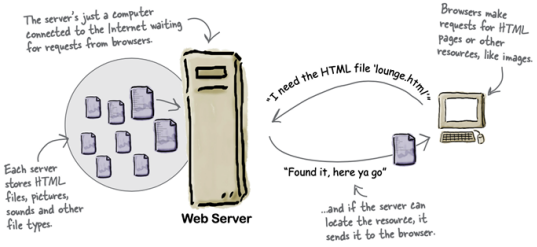
Web-Entwicklung

Jil Zerndt, Lucien Perret
January 2025

Einführung

WEB-Architektur **Client-Server-Modell:**

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



Internet vs. WWW

Internet:

- Weltweites Netzwerk aus vielen Rechnernetzwerken
- Ursprünglich: ARPANET (1969: vier Knoten)
- Als Internet ab 1987 bezeichnet (ca. 27 000 Knoten)
- Verschiedene Dienste: E-Mail, FTP, WWW, etc.
- Basis-Protokolle: TCP/IP

World Wide Web:

- Service, der auf dem Internet aufbaut
- Basiert auf: HTTP, HTML, URLs
- Wichtige Applikations- und Informationsplattform
- Unzählige Technologien und Spezifikationen

Technologien

Client-Seitig → Front-end Entwickler

- Beschränkt auf Browser-Funktionalität
- HTML + CSS + JavaScript
- 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

URL-Aufbau URL-Struktur:

```
Schema:
//[user[:password]@]host[:port]/path[?query][#fragment]

Beispiel:
http://hans:1234@idk.org:80/demo?land=de&stadt=aa#fuck
```

- Scheme: Protokoll (http, https, ftp, etc.)
- User/Password: Optional für Authentifizierung
- Host: Domain oder IP-Adresse
- Port: Optional, Standard ist 80/443
- Path: Pfad zur Ressource
- Query: Optional, Parameter
- Fragment: Optional, Ankerpunkt im Dokument

JavaScript

Grundlagen und Datentypen

JavaScript Grundlagen

- Veröffentlicht 1995 für Netscape Navigator 2.0
- Entwickelt von Brendan Eich
- Dynamisches Typenkonzept
- Objektorientierter und funktionaler Stil möglich
- Wichtigste Programmiersprache für Webanwendungen
- Läuft im Browser und serverseitig (Node.js)

Web-Konsole JavaScript Console im Browser und Node.js:

- `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

Datentypen 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

```
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
  - Kann neu deklariert werden
  - Wird gehoistet
- `let`
  - Scope: Block-Scope
  - Moderne Variante für veränderliche Werte
  - Keine Neudeklaration im gleichen Scope
- `const`
  - Scope: Block-Scope
  - Wert kann nicht neu zugewiesen werden
  - Referenz ist konstant (Objekte können modifiziert 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;
    default:
        // code
}

// Loops
for (let i = 0; i < n; i++) { /* code */ }

while (condition) { /* code */ }

do { /* code */ } while (condition);

for (let item of array) {
    doSomething(item);
}

for (let key in object) {
    doSomething(object[key]);
}
```

Strings und reguläre Ausdrücke

Strings Strings in JavaScript sind:

- Sequenz von 16-Bit-Unicode-Zeichen
- Kein spezieller char-Typ vorhanden
- Definition mit einfachen ('...') oder doppelten ("...") Anführungszeichen möglich
- Escape-Sequenzen mit \:
  - \n für Zeilenumbruch
  - \\ für Backslash
  - \' und \" für Anführungszeichen
- Verkettung mit + Operator

Template Strings (mit Backticks) bieten erweiterte Funktionalität:

- Definition mit Backticks (` ...`)
- Mehrzeilige Strings möglich
- String-Interpolation mit $...
- Backslash wird als \interpretiert (außer vor \`, \$ und Leerzeichen)

```
// String Interpolation
`half of 100 is ${100 / 2}`  // "half of 100 is 50"

// Mehrzeilige Strings
`erste Zeile
zweite Zeile`  // "erste Zeile\nzweite Zeile"
```

String Operationen

```
// String Verkettung
"con" + "cat" + "enate"  // "concatenate"

// Template String mit Interpolation
const name = "World"
`Hello ${name}!`  // "Hello World!"

// Mehrzeiliger Template String
const text = `
  Erste Zeile
  Zweite Zeile
`
```

Objekte und 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]

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

```
let person = {
    name: "John",
    age: 30,
    greet() {
        return "Hello, I'm" + this.name;
    }
};
// Eigenschaften manipulieren/abfragen
person.job = "Developer";     // hinzufuegen
delete person.age;            // loeschen
"name" in person;             // true
// Objekte zusammenfuehren
Object.assign(person, {city: "Berlin"});
// Spread Syntax
let clone = {...person};
// Destrukturierung
let {name, job} = person;
```

JS Arrays spezielle Objekte: dynamische Grösse und Typen

```
let arr = [1, 2, 3, 4, 5];
// Elemente hinzufuegen/entfernen
arr.push(6);     // [1, 2, 3, 4, 5, 6]
arr.pop();       // [1, 2, 3, 4, 5]
// Elemente am Anfang hinzufuegen/entfernen
arr.unshift(0); // [0, 1, 2, 3, 4, 5]
arr.shift();     // [1, 2, 3, 4, 5]
// Elemente einfuegen/entfernen
arr.splice(2, 0, 2.5); // [1, 2, 2.5, 3, 4, 5]
arr.splice(2, 2);      // [1, 2, 4, 5]
// Teilarray erstellen
arr.slice(1, 3); // [2, 4]
// Funktional
arr.map(x => x * 2); // [2, 4, 8, 10]
arr.filter(x => x > 3); // [4, 5]
arr.reduce((acc, x) => acc + x, 0); // 15
arr.copyWithin(0, 3, 5); // [4, 5, 8, 10, 5]
// Suchen und Testen
arr.every(x => x > 0); // true
arr.some(x => x > 5); // true
arr.find(x => x > 3); // 4
arr.findIndex(x => x > 3); // 3
// Iteration
arr.forEach(x => console.log(x));
// Arrays verbinden
arr.join(', '); // '1, 2, 3, 4, 5'
arr.concat([6, 7]); // [1, 2, 3, 4, 5, 6, 7]
// Sortieren/Umkehren
arr.sort(); // [1, 2, 3, 4, 5]
arr.reverse(); // [5, 4, 3, 2, 1]
```

JSON JavaScript Object Notation:

- Daten-Austauschformat, nicht nur für JavaScript
- Basiert auf JavaScript-Objektliteralen
- Methoden: `JSON.stringify()` und `JSON.parse()`

```
let obj = {type: "cat", name: "Mimi", age: 3};
let json = JSON.stringify(obj);
// '{"type":"cat","name":"Mimi","age":3}'

let parsed = JSON.parse(json);
// {type: 'cat', name: 'Mimi', age: 3}
```

Funktionen

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'
```

Funktionsdefinition

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

Funktionsdefinitionen

```
// 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;
};
```

Parameter und Arguments

- Default-Parameter: `function f(x = 1) {}`
- Rest-Parameter: `function f(...args) {}`
- Destrukturierung: `function f({x, y}) {}`
- arguments: Array-ähnliches Objekt mit allen Argumenten

Funktionale Konzepte

- Funktionen sind First-Class Citizens
- Können als Parameter übergeben werden
- Können von Funktionen zurückgegeben werden
- Closure: Zugriff auf umgebenden Scope
- Pure Functions: Keine Seiteneffekte

Closure Beispiel

```
function counter() {
    let count = 0;
    return {
        increment: () => ++count,
        decrement: () => --count,
        getCount: () => count
    };
}

const myCounter = counter();
myCounter.increment(); // 1
myCounter.increment(); // 2
myCounter.decrement(); // 1
```

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

- Jedes Objekt hat ein Prototyp-Objekt
- Prototyp dient als Fallback für Properties
- Vererbung über Prototypenkette
- `Object.create()` für Prototyp-Vererbung

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 */
```

Statische Methoden

- Statische Methoden gehören zur Klasse, nicht zur Instanz
- Werden mit `static` deklariert

```
class Person {
    static create (name) {
        return new Person(name)
    }
}
let p36 = Person.create("John")
```

Asynchrone Programmierung

File API

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

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 */
```

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

Asynchrone Dateioperationen

```
const fs = require('fs')
fs.access('test.txt', fs.constants.R_OK |
    fs.constants.W_OK, (err) => {
    if (err) {
        console.error('no access!')
        return
    }
    console.log('can read/write')
})
```

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
})
```

Streams Streams sind Sequenzen von Daten, die in Teilen verarbeitet werden.

- `Readable`: Datenquelle
- `Writable`: Datenziel
- `Duplex`: Beides
- `Transform`: Daten umwandeln

Event Loop, Callbacks und Timer

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

Event Loop und Threads

- JavaScript ist single-threaded
- Event Loop verarbeitet asynchrone Operationen
- Call Stack für synchronen Code
- Callback Queue/Task Queue für asynchrone Callbacks
- Microtask Queue für Promises und process.nextTick

Der event loop bearbeitet asynchrone tasks in volgender Reihenvolge:

1. Call Stack | Alle Funktionsaufrufe kommen auf den call stack und werden mit first in last out abgehandelt.
2. Microtask Queue | Alle promises (.then/.catch.finally/await) werden als Microtask in die Microtask Queue eingefügt. Ist der Call Stack lehr, checkt der event loop die Microtask Queue und verschiebt existierende Microtasks auf den Call Stack um abgearbeitet zu werden.
3. Callback Queue/Task Queue | Alle callback Funktionen werden als task in der Task Queue abgelegt. Wenn der Call Stack und die Microtask Queue beide lehr sind checkt der Event Loop die Task Queue und verschiebt die existierenden Tasks nacheinander auf den callstack um abgehandelt zu werden.

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
})
```

Callbacks und Timer

```
// setTimeout
setTimeout(() => {
    console.log('Delayed by 1 second');
}, 1000);

// setInterval
const id = setInterval(() => {
    console.log('Every 2 seconds');
}, 2000);
clearInterval(id);

// Event Handler mit Callback
element.addEventListener('click', (event) => {
    console.log('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')
})
```

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;
}
```

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
    });
```

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

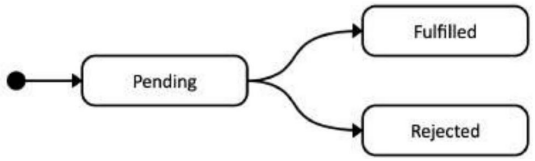
Aufruf neu:

```
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

Promise.all() und Promise.race()

```
// Promise.all
Promise.all([promise1, promise2])
    .then(results => {
        // Array mit allen Ergebnissen
    });

// Promise.race
Promise.race([promise1, promise2])
    .then(firstResult => {
        // Erstes erfuelltes Promise
    });
```

Async/Await

- Syntaktischer Zucker für Promises
- Ersetzt Promise-Verkettung durch sequentielle Ausführung
- `async` markiert Funktion als asynchron
- `await` wartet auf Promise-Resolution
- `try/catch` für Fehlerbehandlung
- `Promise.all` kann durch parallele `await` ersetzt werden
- `await` gibt Wert von Promise zurück
- `await` kann nur in async-Funktionen verwendet werden

Async/Await

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

// Async Funktion
async function getData() {
    try {
        const response = await fetch(url);
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error:', error);
    }
}

// Parallele Ausfuehrung
async function getMultipleData() {
    const [data1, data2] = await Promise.all([
        getData(url1),
        getData(url2)
    ]);
    return { data1, data2 };
}
```

Webserver

Server im Internet

File-Transfer (File Server)

Web-Transfer-Protokolle

File-Transfer:

- FTP (File Transfer Protocol)
- SFTP (SSH File Transfer Protocol)
- Anwendungen mit GUI und Kommandozeile

HTTP/HTTPS:

- Standard-Ports: 80/443
- Request-Response Modell
- Stateless Protokoll
- HTTPS: Verschlüsselte Übertragung mittels SSL/TLS

HTTP-Server

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

HTTP-Methoden

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

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

Node.js und Module

Node.js

- JavaScript Runtime basierend auf V8
- Event-driven und non-blocking I/O
- Großes Ökosystem (npm)
- Ideal für Netzwerk-Anwendungen
- REPL für interaktive Entwicklung

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();
}
```

Module System

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

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

// package.json
{
    "type": "module",
    "dependencies": {
        "express": "^4.17.1"
    }
}
```

NPM Commands

Wichtige npm Befehle:

- `npm init`: Projekt initialisieren
- `npm install`: Abhängigkeiten installieren
- `npm install -save package`: Produktiv-Dependency
- `npm install -save-dev package`: Entwicklungs-Dependency
- `npm run script`: Script ausführen
- `npm update`: Packages aktualisieren

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 und Express.js

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')
})
```

Jasmine (Testing)

Test-Driven Development

- Tests vor Implementation schreiben
- Red-Green-Refactor Zyklus
- Tests als Spezifikation
- Bessere Code-Qualität
- Einfacheres Refactoring

Jasmine Tests

```
describe("Calculator", () => {
    let calc;

    beforeEach(() => {
        calc = new Calculator();
    });

    it("should add numbers", () => {
        expect(calc.add(1, 2)).toBe(3);
    });

    it("should throw on division by zero", () => {
        expect(() => {
            calc.divide(1, 0);
        }).toThrow();
    });
});
```

Jasmine Matchers

- toBe(): Strikte Gleichheit (===)
- toEqual(): Strukturelle Gleichheit
- toContain(): Array/String enthält Element
- toBeDefined(), toBeUndefined()
- toBeTruthy(), toBeFalsy()
- toBeGreaterThan(), toBeLessThan()
- toMatch(): RegExp Match
- toThrow(): Exception wird geworfen

Jasmine Setup

```
// Installation
npm install --save-dev jasmine

// jasmine.json
{
    "spec_dir": "spec",
    "spec_files": [
        "**/*[sS]pec.js"
    ],
    "helpers": [
        "helpers/**/*.js"
    ]
}

// Test ausfuehren
npx jasmine
```

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)
```

JavaScript Exam Preparation

Key Concepts for Exam Essential topics to focus on:

- Data types and type coercion
- Variable scoping (var, let, const)
- Functions (declaration vs expression)
- Objects and prototypes
- Asynchronous programming
- Promise chains and async/await
- Event loop understanding
- Array methods and manipulation

Common Pitfalls Watch out for these tricky areas:

- typeof null returns 'object'
- Hoisting behavior differences between var and let
- this keyword behavior in different contexts
- Promise resolution order
- Event loop execution order
- Closure scope understanding
- Array method return values

Sample Multiple Choice Questions

Data Types and Coercion

What is the output of the following code?

```
console.log(typeof typeof 42);
```

- a) "number"
- b) ßtringcorrect
- c) ündefined"
- d) öbject"

Explanation: typeof 42 returns "number", and typeof "number"returns ßtring"

Variable Scope

What is logged to the console?

```
for(var i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 1);
}
```

- a) 0, 1, 2
- b) 3, 3, 3 - correct
- c) undefined, undefined, undefined
- d) Error

Explanation: var creates one shared binding, loop finishes before timeouts execute

Promises

What is the output sequence?

```
console.log('1');
Promise.resolve().then(() => console.log('2'));
setTimeout(() => console.log('3'), 0);
console.log('4');
```

- a) 1, 2, 3, 4
- b) 1, 4, 2, 3 - correct
- c) 1, 4, 3, 2
- d) 1, 2, 4, 3

Explanation: Microtasks (Promises) execute before macrotasks (setTimeout)

MC exercises

Objects and Prototypes

What is logged?

```
const proto = { value: 42 };
const obj = Object.create(proto);
obj.value = undefined;
console.log('value' in obj);
```

- a) false
- b) true - correct
- c) undefined
- d) 42

Explanation: 'in' operator checks own and inherited properties

Array Methods

What is returned?

```
[1, 2, 3].map(x => x * 2).filter(x => x > 4);
```

- a) [6] - correct
- b) [4, 6]
- c) [2, 4, 6]
- d) []

Explanation: map creates [2,4,6], filter keeps only values > 4

Common Exam Patterns Look for questions about:

- Type coercion in comparisons (== vs ===)
- Scope and closure behavior
- Promise and async execution order
- Object property inheritance
- Array method chaining results
- Event loop and task queue order
- this context in different situations

Critical Code Patterns

```
// Closure Example
function counter() {
    let count = 0;
    return () => ++count;
}

// Promise Chain
Promise.resolve(1)
    .then(x => x + 1)
    .then(x => Promise.resolve(x + 1))
    .then(console.log); // 3

// Event Loop Order
console.log(1);
setTimeout(() => console.log(2), 0);
Promise.resolve().then(() => console.log(3));
console.log(4);
// Outputs: 1, 4, 3, 2

// Prototype Chain
function Animal(name) {
    this.name = name;
}
Animal.prototype.speak = function() {
    return `${this.name} makes a sound.`;
};
```

Key Exam Strategies

- Read code examples carefully for edge cases
- Consider asynchronous execution order
- Check for scope and closure effects
- Remember type coercion rules
- Understand prototype chain inheritance
- Know common array method behaviors
- Consider the event loop for timing questions

more examples

Typische JavaScript-Aufgaben

```
// 1. Type Checking und Konvertierung
typeof(someVariable)                    // Typ pruefen
Number("123")                           // String zu Number
String(123)                             // Number zu String
Boolean(expression)                     // zu Boolean
Array.isArray(arr)                      // Array pruefen

// 2. Array Manipulation
array.push(item)                        // am Ende
    hinzufuegen
array.pop()                             // letztes Element
    entfernen
array.unshift(item)                     // am Anfang
    hinzufuegen
array.shift()                           // erstes Element
    entfernen
array.splice(start, deleteCount)        // Elemente
    entfernen
array.slice(start, end)                 // Teil-Array
    erstellen

// 3. Object Handling
Object.keys(obj)                        // Array von Keys
Object.values(obj)                      // Array von Values
Object.entries(obj)                     // Array von [key,
    value] Paaren
Object.assign({}, obj1, obj2)           // Objekte
    zusammenfuehren
{...obj1, ...obj2}                      // Spread Operator

// 4. String Manipulation
str.split(delimiter)                    // String zu Array
arr.join(delimiter)                     // Array zu String
str.substring(start, end)               // Teilstring
str.replace(search, replace)            // Ersetzen
str.trim()                              // Whitespace
    entfernen
```

Typische Prüfungsaufgaben

```
// Was ist die Ausgabe?
console.log(typeof null)            // "object"
console.log(typeof undefined)       // "undefined"
console.log(typeof [])              // "object"
console.log(typeof (() => {}))      // "function"

// Vergleiche
console.log(null == undefined)      // true
console.log(null === undefined)     // false
console.log([1,2] == [1,2])         // false
console.log("5" == 5)               // true
console.log("5" === 5)              // false

// Array-Methoden
let 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]
```

Funktions-Patterns

```
// 1. Default Parameter
function greet(name = 'User') {
    return `Hello ${name}!`;
}

// 2. Rest Parameter
function sum(...numbers) {
    return numbers.reduce((a, b) => a + b, 0);
}

// 3. Closure
function counter() {
    let count = 0;
    return {
        increment: () => ++count,
        getCount: () => count
    };
}

// 4. IIFE (Immediately Invoked Function Expression)
const counter = (() => {
    let count = 0;
    return {
        increment: () => ++count,
        getCount: () => count
    };
})();

// 5. Callback Pattern
function fetchData(callback) {
    setTimeout(() => {
        callback('Data');
    }, 1000);
}
```

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)
```

Browser-Technologien

Vordefinierte Browser-Objekte

Browser-Objekte

Im Browser stehen spezielle globale Objekte zur Verfügung:

- `window`: Browserfenster und globaler Scope
  - `window.innerHeight`: Viewport Höhe
  - `window.pageYOffset`: Scroll Position
  - `window.location`: URL Manipulation
- `document`: Das aktuelle HTML-Dokument
- `navigator`: Browser-Informationen
- `history`: Browser-Verlauf
- `location`: URL-Informationen

document-Objekt repräsentiert das aktuelle HTML-Dokument:

```
// Element finden
document.getElementById("id")
document.querySelector("selector")
document.querySelectorAll("selector")
// DOM manipulieren
document.createElement("tag")        // Element
    erstellen
document.createTextNode("text")      // Textknoten
    erstellen
document.createAttribute("attr")     // Attribut
    erstellen
// Event Handler
document.addEventListener("event", handler)
```

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
```

navigator-Objekt für Browser-Informationen:

```
navigator.userAgent  // Browser User-Agent
navigator.language   // Browser Sprache
navigator.onLine     // Online-Status
```

history-Objekt für Browser-Verlauf:

```
history.length  // Anzahl Eintraege
history.back()  // Zurueck zur letzten Seite
history.forward()  // Vorwaerts zur naechsten Seite
```

location-Objekt für URL-Informationen:

```
location.href  // URL der Seite
location.hostname  // Hostname
location.pathname  // Pfad
location.search  // Query-Parameter
location.hash  // Anker
```

Document Object Model (DOM)

structure:

- element auffinden
- textknoten erzeugen
- elementknoten erzeugen
- attribute setzen
- style anpassen

DOM Struktur Das DOM ist eine Baumstruktur des HTML-Dokuments:

- Jeder HTML-Tag wird zu einem Element-Knoten
- Text innerhalb von Tags wird zu Text-Knoten
- Attribute werden zu Attribut-Knoten
- NodeType Konstanten:
  - 1: Element Node (ELEMENT_NODE)
  - 3: Text Node (TEXT_NODE)
  - 8: Comment Node (COMMENT_NODE)

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 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"
```

DOM Manipulation

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

// 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.classList.add('newClass');
elem.classList.remove('oldClass');

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

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'); //
    Erstes Element
const elems =
    document.querySelectorAll('div.myClass'); // Alle
    Elemente

// 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
```

Event Handling

Event Handling Events sind Ereignisse, die im Browser auftreten:

- Benutzerinteraktionen
- DOM-Änderungen
- Ressourcen laden
- Timer, usw,

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
})
```

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);
```

Event Informationen abfragen und steuern

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

    // Event Informationen
    event.type            // Art des Events
    event.target          // Ausloesende Element
    event.currentTarget   // Element mit Listener

    // Event Steuerung
    event.preventDefault();  // Default verhindern
    event.stopPropagation(); // Bubbling stoppen
});

// Event Listener entfernen
element.removeEventListener('click', handler);
```

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

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

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();
});
```

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

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

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

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

Event Types

Event Typen Wichtige Event-Kategorien:

- **Maus:** click, dblclick, mousedown, mouseup, mousemove, mouseover
- **Tastatur:** keydown, keyup, keypress
- **Formular:** submit, change, input, focus, blur
- **Dokument:** DOMContentLoaded, load, unload
- **Fenster:** resize, scroll, popstate
- **Drag & Drop:** dragstart, drag, dragend, drop

Tastatur-Events keydown (Achtung: kann mehrmals ausgelöst werden) und keyup:

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

Maus-Events

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

```
let button = document.querySelector("button")
button.addEventListener("click", () => {
    console.log("Button geklickt!")
})
```

Scroll-Events Das Scrollevent enthält Attribute wie pageYOffset und pageXOffset.

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

Focus-Events
Fokus- und Ladeereignisse

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

Jquery

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

- Einfache DOM-Manipulation
- Event-Handling
- Animationen
- AJAX-Requests
- Plugins
- Cross-Browser-Unterstützung

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

$(Funktion) → DOM ready

```
$(function() {
  // Code to run when the DOM is ready
});
```

$("CSS Selektor").aktion(...) → Wrapped Set
Knoten, die Sel. erfüllen, eingepackt in ein jQuery-Objekt

```
$(".toggleButton").attr("title");
// Get the title attribute of elements with class
    'toggleButton'
```

```
$(".toggleButton").attr("title", "click here");
// Set the title attribute of elements with class
    'toggleButton' to 'click here'
```

```
$(".toggleButton").attr({
  title: "click here",
  // other attributes
});
// Set multiple attributes of elements with class
    'toggleButton'
```

```
$(".toggleButton").attr("title", function() {
  // function to set title
}).css({
  // CSS properties
}).text("New Text").on("click", function(event) {
  // click event handler
});
```

$("HTML-Code") → Create new elements (Wrapped Set) neuer Knoten erstellen und in ein jQuery-Objekt einpacken, noch nicht im DOM

```
$("<li>...</li>").addClass("new-item").appendTo("ul");
// Create a new list item, add a class, and append it
    to a list
```

```
$("<li>...</li>").length;
// Get the length of the new list item
```

```
$("<li>...</li>")[0];
// Get the raw DOM element of the new list item
```

Wrapped Set from DOM node dieser Knoten in ein jQuery-Objekt eingepackt

```
$(document.body);
// Wrap the body element in a jQuery object
```

```
$(this);
// Wrap the current element in a jQuery object
```

Graphics

Web-Grafiken

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

Grafik im Browser Zwei Haupttechnologien für Grafiken:

- Canvas: Pixel-basierte Grafik
  - Gut für komplexe Animationen
  - Direkte Pixel-Manipulation
  - Keine DOM-Struktur
- SVG: Vektor-basierte Grafik
  - Skalierbar ohne Qualitätsverlust
  - Teil des DOM
  - Event-Handler möglich

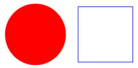
SVG

SVG Scalable Vector Graphics

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

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

Ausgabe:
Normal HTML here.



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

SVG mit JavaScript

```
// 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);
```

Wichtige SVG Attribute und Methoden

- setAttribute(): Attribut setzen
- getAttribute(): Attribut abfragen
- removeAttribute(): Attribut entfernen
- createElementNS(): Element erstellen
- appendChild(): Element hinzufügen
- removeChild(): Element entfernen
- querySelector(): Element finden
- querySelectorAll(): Alle Elemente finden
- addEventListener(): Event-Handler hinzufügen
- removeEventListener(): Event-Handler entfernen
- dispatchEvent(): Event auslösen
- createEvent(): Event erstellen

Canvas

Canvas Das <canvas>-Element bietet eine Zeichenfläche (API) für Pixelgrafiken:

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

Canvas Methoden

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

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 API

Browser-APIs Browser bieten APIs für verschiedene Aufgaben:

- `localStorage`: Permanente Speicherung
- `sessionStorage`: Daten temporär speichern (aktuelle Session)
- `cookies`: Kleine Datenpakete, auch für Server
- `indexedDB`: NoSQL-Datenbank im Browser
- `history`: Browser-Verlauf
- `location`: URL-Informationen
- `navigator`: Browser-Informationen
- `geolocation`: Standort abfragen

Local Storage Mit `localStorage` können Daten auf dem Client gespeichert werden:

```
localStorage.setItem("username", "Max")
console.log(localStorage.getItem("username")) // -> Max
localStorage.removeItem("username")
```

Local Storage wird verwendet, um Daten der Webseite lokal abzuspeichern. Die Daten bleiben nach dem Schliessen des Browsers erhalten. Die Daten sind in Developer Tools einsehbar und änderbar.
Die Daten werden nach Domains abgespeichert. Es können pro Webseite etwa 5-10MB abgespeichert werden.
Die Werte werden als Strings gespeichert, daher müssen Objekte mit JSON codiert werden:

```
Let user = {name: "Hans", highscore: 234}
localStorage.setItem(JSON.stringify(user))
```

Ausserdem: Synchroner API-Zugriff

Session Storage `sessionStorage` speichert Daten nur für die Dauer der Sitzung:

```
sessionStorage.setItem("sessionID", "abc123")
```

Web Storage speichert Daten auf der Client-Seite:
1. Local Storage (Daten speichern):

```
// 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
```

2. Session Storage (nur für aktuelle Session):

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

History gibt zugriff auf Verlauf des akutellen Fensters/Tab.

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

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
})
```

Cookies Cookies sind kleine Datenpakete, die im Browser gespeichert werden:

- `document.cookie`: Cookies lesen/schreiben
- `expires`: Ablaufdatum
- `path`: Pfad, für den das Cookie gültig ist
- `domain`: Domain, für die das Cookie gültig ist
- `secure`: Nur über HTTPS

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

```
var options = { enableHighAccuracy: true, timeout:
     5000, maximumAge: 0 }
function success(pos) {
    var crd = pos.coords
    console.log(`Latitude : ${crd.latitude}`)
    console.log(`Longitude: ${crd.longitude}`)
    console.log(`More or less ${crd.accuracy} meters.`)
}
function error(err) { ... }
navigator.geolocation.getCurrentPosition(success,
     error, options)
```

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)
```

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

Client-Server-Interaktion (Formulare)

Formular Events

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

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

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. 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
})
```

Event Handling für Formulare

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

Default-Verhalten Das Default-Verhalten von Formularen kann mit `preventDefault()` unterbunden werden.

```
let form = document.querySelector("form");
form.addEventListener("submit", event => {
    event.preventDefault();
    console.log("Formular abgesendet!");
});
```

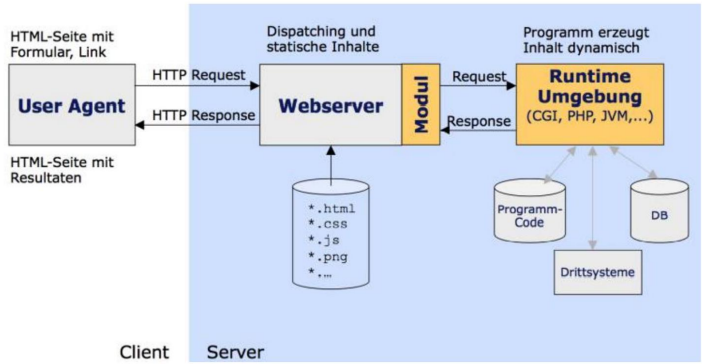
Formular Handling

```
// HTML Formular erstellen
<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');

    // Mit Fetch API senden
    fetch('/submit', {
        method: 'POST',
        body: formData
    });
});
```

GET/POST Methode

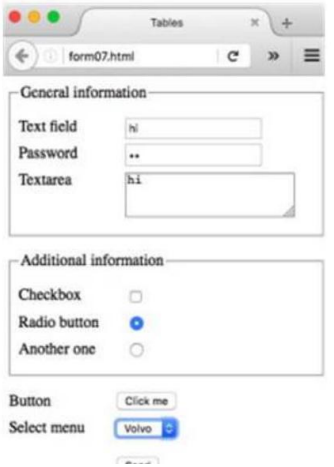
Formulare können auch POST/GET Aktionen ausführen:
Action beschreibt das Skript, welches die Daten annimmt. Method ist die Methode die ausgeführt wird.

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



Formulare

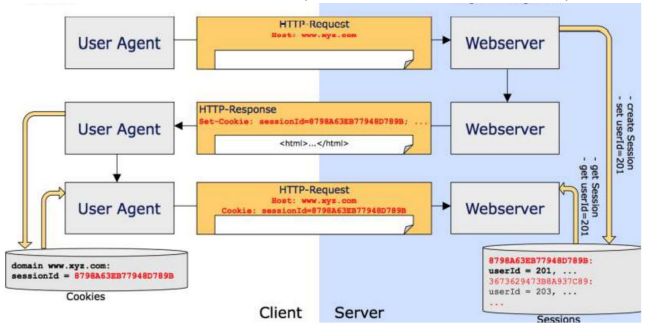
```
<form>
  <fieldset>
      <legend>General information</legend>
      <label>Text field <input type="text"
          value="hi"></label>
      <label>Password <input type="password"
          value="hi"></label>
      <label class="area">Textarea
          <textarea>hi</textarea></label>
  </fieldset>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



Cookies und Sessions

HTTP-Cookies

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



Cookies Cookies speichern clientseitig Daten:

```
document.cookie = "username=Max; expires=Fri, 31 Dec
    2025 23:59:59 GMT";
console.log(document.cookie);
```

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:

```
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

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)

Sessions

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

Sessions Sessions speichern serverseitig Daten und nutzen eine Session-ID für die Zuordnung:

```
// Beispiel: Session-Handling mit Express.js
req.session.user = "Max";
console.log(req.session.user);
```

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 Mit der Fetch-API können HTTP-Requests ausgeführt werden:

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

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

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

Response Objekt

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

XMLHttpRequest und Fetch Moderne Ansätze für HTTP-Requests:

- XMLHttpRequest: Älterer Ansatz, komplexer
- Fetch API: Moderner Ansatz, Promise-basiert
- Unterstützung für verschiedene Datenformate
- CORS (Cross-Origin Resource Sharing)

Fetch API Grundlagen

```
// GET Request
fetch('https://api.example.com/data')
    .then(response => {
        if (!response.ok) {
            throw new Error('Network response was not
                ok');
        }
        return 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');
        if (!response.ok) {
            throw new Error('Network response was not
                ok');
        }
        const data = await response.json();
        return data;
    } catch (error) {
        console.error('Error:', error);
    }
}
```

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)
  }
}
```

CORS (Cross-Origin Resource Sharing) Sicherheitsmechanismus für domainübergreifende Requests:

- Verhindert unauthorized Zugriffe
- Server muss CORS-Header setzen
- Preflight Requests für bestimmte Anfragen
- Wichtige Header:
  - Access-Control-Allow-Origin
  - Access-Control-Allow-Methods
  - Access-Control-Allow-Headers

Sessions und Authentication

```
// Login Request
async function login(username, password) {
    const response = await fetch('/api/login', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        credentials: 'include',  // Fuer Cookies
        body: JSON.stringify({
            username,
            password
        })
    });

    if (response.ok) {
        const user = await response.json();
        // Session Token in localStorage speichern
        localStorage.setItem('token', user.token);
    }
}

// Authenticated Request
async function getProtectedData() {
    const token = localStorage.getItem('token');
    const response = await fetch('/api/protected', {
        headers: {
            'Authorization': `Bearer ${token}`
        }
    });
    return response.json();
}
```

WebSocket Bidirektionale Echtzeit-Kommunikation:

- Permanente Verbindung
- Geringer Overhead
- Ideal für Chat, Live-Updates
- Events: open, message, close, error

```
const ws = new WebSocket('ws://localhost:8080');

ws.addEventListener('open', () => {
    console.log('Connected to WebSocket');
    ws.send('Hello Server!');
});

ws.addEventListener('message', event => {
    console.log('Received:', event.data);
});

ws.addEventListener('close', () => {
    console.log('Disconnected from WebSocket');
});
```

REST APIs

REST Prinzipien Representational State Transfer:

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

REST API Implementierung mit Fetch API:

```
// GET - Daten abrufen
fetch('/api/users')
    .then(response => response.json())
    .then(users => console.log(users));

// POST - Neue Ressource erstellen
fetch('/api/users', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({
        name: 'John',
        email: 'john@example.com'
    })
});

// PUT - Ressource aktualisieren
fetch('/api/users/123', {
    method: 'PUT',
    headers: {
        'Content-Type': 'application/json',
    },
    body: JSON.stringify({
        name: 'John Updated'
    })
});

// DELETE - Ressource loeschen
fetch('/api/users/123', {
    method: 'DELETE'
});
```

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);
});
```

Browser Technologies Exam Preparation

Key Browser API Concepts Essential topics to focus on:

- DOM structure and manipulation
- Event handling and bubbling
- Browser storage mechanisms (localStorage, cookies, sessions)
- Window and document objects
- Browser events lifecycle
- Form handling and validation
- Canvas vs SVG
- AJAX and Fetch API

Common Pitfalls Watch out for these tricky areas:

- Event propagation order (capturing vs bubbling)
- Storage API limitations and restrictions
- DOM traversal methods and returns
- Event delegation vs direct binding
- Canvas context state management
- Async nature of fetch requests
- CORS and same-origin policy

Sample Multiple Choice Questions

DOM Manipulation

What is returned by this code?

```
document.querySelectorAll('div').length;
document.getElementsByTagName('div').length;
// When a new div is added dynamically
```

- a) Both stay the same
- b) QuerySelectorAll stays same, getElementsByTagName updates - correct
- c) Both update
- d) Both throw error

Explanation: getElementsByTagName returns a live NodeList, querySelectorAll returns a static NodeList

Event Handling

In what order are the events logged?

```
<div onclick="console.log('1')">
    <button onclick="console.log('2')">Click</button>
</div>
```

- a) 1, 2
- b) 2, 1 - correct
- c) Just 2
- d) Just 1

Explanation: Events bubble up from target to ancestors by default

Local Storage

What happens here?

```
localStorage.setItem('number', 42);
console.log(typeof localStorage.getItem('number'));
```

- a) "number"
- b) ßtring correct
- c) öbject"
- d) ündefined"

Explanation: localStorage always stores values as strings

MC

Canvas vs SVG

Which statement is true?

- a) SVG elements can have event listeners - correct
- b) Canvas elements maintain separate elements for shapes
- c) SVG is pixel-based
- d) Canvas is better for complex animations

Explanation: SVG elements become part of the DOM and can have event listeners attached

Fetch API

What is logged?

```
fetch('/api/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(err => console.log('Error'));
// When server returns 404
```

- a) Error
- b) undefined
- c) null
- d) The response body - correct

Explanation: fetch() only rejects on network failure, not HTTP errors

Common Exam Patterns Look for questions about:

- DOM method return types (NodeList vs HTMLCollection)
- Event propagation and prevention
- Storage API value types and limitations
- Differences between Canvas and SVG
- Asynchronous operations with fetch
- CORS and security restrictions
- Form validation and submission

Critical Code Patterns

```
// Event Delegation
document.getElementById('parent').addEventListener('click',
    (e) => {
    if (e.target.matches('.child')) {
        console.log('Child clicked');
    }
});

// Storage with Objects
const user = { name: 'John', age: 30 };
localStorage.setItem('user', JSON.stringify(user));
const savedUser =
    JSON.parse(localStorage.getItem('user'));

// Async Form Submission
form.addEventListener('submit', async (e) => {
    e.preventDefault();
    const formData = new FormData(form);
    try {
        const response = await fetch('/api/submit', {
            method: 'POST',
            body: formData
        });
        if (!response.ok) throw new Error('Network
            response was not ok');
        const result = await response.json();
    } catch (error) {
        console.error('Error:', error);
    }
});
```

Key Exam Strategies

- Understand DOM live vs static collections
- Know event propagation phases
- Remember storage API limitations
- Understand async operation order
- Know security restrictions (CORS, same-origin)
- Recognize Canvas vs SVG use cases
- Consider browser compatibility issues

more examples

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

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

Class und Prototype Patterns

```javascript
// 1. Class Definition
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        return `${this.name} makes a sound`;
    }
}

// 2. Inheritance
class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }

    speak() {
        return `${this.name} barks`;
    }
}

// 3. Getter/Setter
class Circle {
    constructor(radius) {
        this._radius = radius;
    }

    get radius() {
        return this._radius;
    }

    set radius(value) {
        if (value <= 0) throw new Error('Invalid
            radius');
        this._radius = value;
    }

    get area() {
        return Math.PI * this._radius ** 2;
    }
}

// 4. Static Methods
class MathUtils {
    static add(x, y) {
        return x + y;
    }

    static multiply(x, y) {
        return x * y;
    }
}
```

Error Handling Patterns

```javascript
// 1. Try-Catch mit spezifischen Errors
try {
    JSON.parse(invalidJson);
} catch (error) {
    if (error instanceof SyntaxError) {
        console.error('Invalid JSON');
    } else {
        console.error('Unknown error:', error);
    }
}

// 2. Custom Error Classes
class ValidationError extends Error {
    constructor(message) {
        super(message);
        this.name = 'ValidationError';
    }
}

// 3. Async Error Handling
async function fetchData() {
    try {
        const response = await fetch(url);
        if (!response.ok) {
            throw new Error(`HTTP error:
                ${response.status}`);
        }
        return await response.json();
    } catch (error) {
        console.error('Fetch failed:', error);
        throw error; // Re-throw fuer weitere
            Verarbeitung
    }
}

// 4. Promise Error Chain
fetchData()
    .then(processData)
    .then(saveData)
    .catch(error => {
        if (error instanceof NetworkError) {
            retryOperation();
        } else {
            logError(error);
        }
    })
    .finally(() => cleanup());
```

Module Patterns

```javascript
// 1. ES6 Module Export/Import
// math.js
export const add = (x, y) => x + y;
export const multiply = (x, y) => x * y;
export default class Calculator {
    // ...
}

// main.js
import Calculator, { add, multiply } from './math.js';
import * as mathUtils from './math.js';

// 2. CommonJS Module Pattern (Node.js)
// utils.js
module.exports = {
    formatDate(date) {
        return date.toISOString();
    },
    calculateTotal(items) {
        return items.reduce((sum, item) => sum +
            item.price, 0);
    }
};

// app.js
const utils = require('./utils.js');

// 3. Revealing Module Pattern
const counterModule = (() => {
    let count = 0;

    const increment = () => ++count;
    const decrement = () => --count;
    const getCount = () => count;

    return {
        increment,
        decrement,
        getCount
    };
})();
```

REST API Implementation

```
// Express REST API Example
const express = require('express');
const app = express();
app.use(express.json());

// GET Collection
app.get('/api/users', async (req, res) => {
    try {
        const users = await User.find();
        res.json(users);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// GET Single Resource
app.get('/api/users/:id', async (req, res) => {
    try {
        const user = await
            User.findById(req.params.id);
        if (!user) {
            return res.status(404).json({ error: 'User
                not found' });
        }
        res.json(user);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});

// POST New Resource
app.post('/api/users', async (req, res) => {
    try {
        const user = new User(req.body);
        await user.save();
        res.status(201).json(user);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
});

// PUT Update Resource
app.put('/api/users/:id', async (req, res) => {
    try {
        const user = await User.findByIdAndUpdate(
            req.params.id,
            req.body,
            { new: true }
        );
        res.json(user);
    } catch (error) {
        res.status(400).json({ error: error.message });
    }
});
```

UI-Bibliotheken und Komponenten

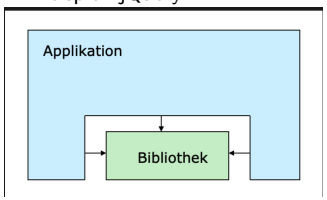
Overview

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

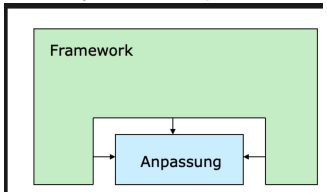
Frameworks und Bibliotheken

Framework vs. Bibliothek

- **Bibliothek**:
  - Kontrolle beim eigenen Programm
  - Funktionen werden nach Bedarf verwendet
  - Beispiel: jQuery



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



ANSÄTZE IM LAUF DER ZEIT

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

SERVERSEITE

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

Architektur

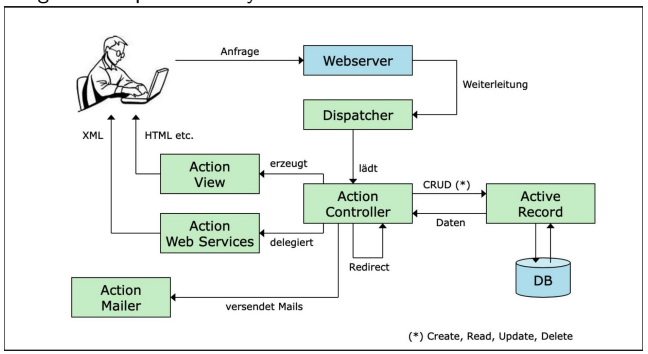
- **MVC (Model-View-Controller)**:
  - Model: Repräsentiert Daten und Geschäftslogik, können Observer über Zustandsänderungen informieren
  - View: Bildet UI (z.B. HTML/CSS), kommuniziert mit Controller
  - Controller: Verarbeitet Eingaben (z.B. Clicks), aktualisiert Model
- **Single Page Apps (SPAs)**:
  - Vermeidet Neuladen von Seiten
  - Inhalte dynamisch nachgeladen (Ajax, REST)
  - Bessere Usability durch schnellere UI-Reaktion

Model-View-Controller (MVC)

- **Models**
  - Repräsentieren anwendungsspezifisches Wissen und Daten
  - Ähnlich Klassen: User, Photo, Todo, Note
- **Views**
  - Bilden die Benutzerschnittstelle
  - Meist HTML/CSS basiert
- **Controllers**
  - Verarbeiten Eingaben
  - Aktualisieren Models
  - Steuern View-Aktualisierung

RUBY ON RAILS

- Serverseitiges Framework, basierend auf MVC
- Programmiersprache: Ruby



FOKUS AUF DIE CLIENT-SEITE

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

Komponentenbasierte Entwicklung Grundprinzipien:

- UI in wiederverwendbare Komponenten aufteilen
- Klarer Datenfluss (Props down, Events up)
- Deklarativer Ansatz
- Komponenten können verschachtelt werden
- Zustandsverwaltung in Komponenten
- Container vs. Präsentations-Komponenten

DOM-Scripting und Abstraktionen

DOM-SCRIPTING

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

AUFGABE Liste aus einem Array erzeugen:

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

Entsprechendes Markup:

```
<ul>
    <li>Maria</li>
    <li>Hans</li>
    <li>Eva</li>
    <li>Peter</li>
</ul>
```

DOM-SCRIPTING

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

DOM-SCRIPTING VERBESSERT

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

Damit vereinfachte List-Komponente möglich:

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

JQUERY

jQuery List Implementation

```
function List (data) {
    return $("<ul>").append(...data.map(item =>
        $("<li>").text(item)))
}

function render (tree, elem) {
    while (elem.firstChild) {
        elem.removeChild(elem.firstChild)
    }
    $(elem).append(tree)
}
```

WEB COMPONENTS

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

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

JSX und SJDON

JSX

- XML-Syntax in JavaScript
- Muss zu JavaScript transpiliert werden
- HTML-Tags in Kleinbuchstaben
- Eigene Komponenten mit Großbuchstaben
- JavaScript-Ausdrücke in {...}

```
// 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 pure JavaScript Arrays und Objekte
- Kein Kompilierungsschritt nötig
- Array-basierte Notation

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

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

Vergleich JSX und SJDON

```
// JSX
const element = (
    <div style={{background: 'salmon'}}>
        <h1>Hello World</h1>
        <h2 style={{textAlign: 'right'}}>
            from Web Framework
        </h2>
    </div>
);

// SJDON
const element = [
    "div", {style: "background:salmon"},
    ["h1", "Hello World"],
    ["h2", {style: "text-align:right"},
        "from Web Framework"]
];
```

SuiWeb Framework

SuiWeb Grundkonzepte Simple User Interface Toolkit for Web Exercises:

- Komponentenbasiert wie React
- Unterstützt JSX und SJDON
- Datengesteuert mit Props und State
- Vereinfachte Implementation für Lernzwecke
- Props sind read-only
- State für veränderliche Daten

State Management

State Management Zustandsverwaltung in SuiWeb:

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

State Hook

- Zustandsverwaltung in Funktionskomponenten
- Initialisierung mit useState Hook
- State Updates lösen Re-Rendering aus
- Asynchrone Updates werden gequeued

State Verwaltung

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

    // Event Handler
    const increment = () => setCount(count + 1);
    const decrement = () => setCount(count - 1);

    return [
        "div",
        ["button", {onclick: decrement}, "-"],
        ["span", count],
        ["button", {onclick: increment}, "+"]
    ];
};

// Komplexere State Objekte
const Form = () => {
    const [state, setState] = useState({
        username: '',
        email: '',
        isValid: false
    });

    const updateField = (field, value) => {
        setState({
            ...state,
            [field]: value
        });
    };
};
```

Kontrollierte Eingabefelder

```
const InputForm = () => {
    const [text, setText] = useState("");

    return [
        "form",
        ["input", {
            type: "text",
            value: text,
            oninput: e => setText(e.target.value)
        }],
        ["p", "Eingabe: ", text]
    ];
};
```

Komponenten-Design

Container Components

- Trennung von Daten und Darstellung
- Container kümmern sich um:
  - Datenbeschaffung
  - Zustandsverwaltung
  - Event Handling
- Präsentationskomponenten sind zustandslos

Component Design Principles

- Single Responsibility Principle
- DRY (Don't Repeat Yourself)
- KISS (Keep It Simple, Stupid)
- Lifting State Up
- Props down, Events up
- Komposition über Vererbung

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}]
    ];
};
```

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 Komponente

```
const TodoContainer = () => {
    const [todos, setTodos] = useState([]);
    // Daten laden
    if (todos.length === 0) {
        fetchTodos().then(data => setTodos(data));
    }
    // Event Handler
    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
        ));
    };
    // Render Praesentationskomponente
    return [TodoList, {
        todos,
        onToggle: toggleTodo,
        onAdd: addTodo
    }];
};
// Praesentationskomponente
const TodoList = ({todos, onToggle, onAdd}) => [
    "div",
    [TodoForm, {onAdd}],
    ["ul",
        ...todos.map(todo => [
            TodoItem, {
                key: todo.id,
                todo,
                onToggle
            }
        ])
    ]
];
```

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}
    ];
};
```

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

Styling in SuiWeb

Styling in SuiWeb Verschiedene Möglichkeiten für Styles:

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

Styling Best Practices

- Konsistente Styling-Methode verwenden
- Styles in separaten Objekten/Modulen
- Wiederverwendbare Style-Definitionen
- Responsive Design beachten
- CSS-Klassen für komplexe Styles

Style Optionen

```
// 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"}
    ]
}]
```

Von SuiWeb zu React

React.js Kernkonzepte

- JavaScript-Bibliothek für User Interfaces
- Entwickelt von Facebook (2013)
- Hauptprinzipien:
  - Deklarativ
  - Komponentenbasiert
  - Learn Once, Write Anywhere
  - Virtual DOM für effizientes Rendering
  - Unidirektionaler Datenfluss

React Components

```
// Function Component
const Welcome = ({name}) => {
    return <h1>Hello, {name}</h1>;
};
// State Hook
const Counter = () => {
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count +
                1)}>
                Increment
            </button>
        </div>
    );
};
// Effect Hook
const DataFetcher = () => {
    const [data, setData] = useState(null);

    useEffect(() => {
        fetchData().then(setData);
    }, []);

    return data ? <DisplayData data={data} /> :
        <Loading />;
};
```

Performance Optimierung

Rendering Optimierung

- Virtuelles DOM für effizientes Re-Rendering
- Batching von State Updates
- Memoization von Komponenten
- Lazy Loading
- Key Prop für Listen-Elemente

Performance Best Practices

```
// Effiziente Listen-Rendering
const List = ({items}) => [
    "ul",
    ...items.map(item => [
        "li",
        {key: item.id},  // Wichtig fuer Performance
        item.text
    ])
];
// Lazy Loading
const LazyComponent = async () => {
    const module = await import('./Component.js');
    return module.default;
};
```

UI Libraries and Components Exam Preparation

Key UI Development Concepts

Essential topics to focus on:

- Framework vs Library differences
- Component lifecycle
- State management
- Props vs State
- Virtual DOM
- JSX and SJDON syntax
- Component composition
- Event handling in components

Common Pitfalls

Watch out for these tricky areas:

- State mutation vs immutable updates
- Props drilling
- Component re-rendering conditions
- Event handler binding
- Asynchronous state updates
- JSX conditional rendering
- Component key prop usage

Sample Multiple Choice Questions

Framework vs Library

Which statement is true?

- a) Libraries control the application flow
- b) Frameworks are more flexible than libraries
- c) Frameworks follow the "Hollywood Principle correct
- d) Libraries require more boilerplate code than frameworks

Explanation: Frameworks follow "Don't call us, we'll call you"principle

State Updates

What will be logged?

```
const [count, setCount] = useState(0);
setCount(count + 1);
setCount(count + 1);
console.log(count);
```

- a) 2
- b) 1
- c) 0 - correct
- d) undefined

Explanation: State updates are asynchronous, and the log shows the current state value

JSX Syntax

Which is correct JSX?

```
// Options:
a) <div class="container">Hello</div>
b) <div className="container">Hello</div>
c) <div classname="container">Hello</div>
d) <div class-name="container">Hello</div>
```

- a) Option a
- b) Option b - correct
- c) Option c
- d) Option d

Explanation: JSX uses className instead of class for CSS classes

MC

SJDON Syntax

What's the equivalent SJDON for this JSX?

```
<div className="header">
    <h1>Title</h1>
</div>
```

```
a) ["div", {className: "header"}, ["h1", "Title"]] -
    correct
b) ["div", "header", ["h1", "Title"]]
c) ["div", {"class": "header"}, ["h1", "Title"]]
d) ["div", ["h1", "Title"], {className: "header"}]
```

Explanation: SJDON uses arrays with element, props, and children

Component Props

What happens here?

```
function Welcome(props) {
    props.name = "Alice";
    return <h1>Hello, {props.name}</h1>;
}
```

- a) Renders "Hello, Alice"
- b) Throws an error - correct
- c) Renders nothing
- d) Renders with original prop value

Explanation: Props are read-only and cannot be modified

Common Exam Patterns

Look for questions about:

- Component lifecycle methods
- State update batching
- Event handling in components
- Virtual DOM diffing
- Props vs State usage
- Component composition patterns
- Performance optimization

Critical Code Patterns

```
// Component with State
const Counter = () => {
    const [count, setCount] = useState(0);

    // Correct state update
    const increment = () => {
        setCount(prev => prev + 1);
    };

    return ["div", {},
        ["button", {onclick: increment}, "Increment"],
        ["span", {}, count]
    ];
};

// Props and Event Handling
const Button = ({onClick, children}) => [
    "button",
    {
        onclick: onClick,
        className: "button"
    },
    children
];

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

    if (!user) {
        fetchUser().then(setUser);
    }

    return [UserProfile, {user}];
};
```

Key Exam Strategies

- Understand component lifecycle flow
- Know state update behavior
- Recognize correct JSX/SJDON syntax
- Understand prop immutability
- Know component composition patterns
- Remember event handling patterns
- Understand state vs props usage

more examples

Warum SuiWeb und nicht gleich React? Konzeption von WBE vor ein paar Jahren mit zwei Kollegen (beide IT-Firma aufgebaut, einer in Glarus, einer in Zürich). Sie hatten die Idee, ein eigenes Framework aufzubauen und nicht ein weit verbreitetes Tool zu nehmen, und zwar aus verschiedenen Gründen:

- Vielleicht der wichtigste Grund: Wir sollen Konzepte vermitteln, nicht Tools oder Bibliotheken. Also: die Idee hinter React, Vuew, Svelte, ... verstehen, was vie/ mehr ist, als ein bestimmtes Tool einsetzen zu können.
- Dabei auch: verstehen, wie so eine Bibliothek implementiert werden kann, indem Kernkomponenten selbst implementiert oder zumindest verstanden werden. Auch das ermöglicht ein tieferes Verständnis der zugrunde liegenden Konzepte.
- Einer der Kritikpunkte am Ökosystem rund um zum Beispiel React ist, dass viele fortgeschrittene Konzepte und Muster verwendet werden, die Einsteiger schnell einmal überfordern können. Solchen Problemen können wir mit einem eigenen, schlanken Werkzeug besser aus dem Weg gehen.
- Tools kommen und gehen. Wir haben schon zu oft erlebt, dass Projekte auf dem aktuell gehypten Framework aufgesetzt haben, welches kurz darauf in der Versenkung verschwunden ist. Google Web Toolkit (GWT) anyone?
- Ein eigenes Tool erlaubt auch, mit alternativen Notationen (SJDON) und Umsetzungen (etwa bei den Hooks) experimentieren zu können.

Konzeptionell orientiert sich SuiWeb allerdings an React.js, speziell was die Funktionskomponenten und Hooks angeht.

FAQ: Warum SJDON und nicht JSX?

- SJDON ist eine alternative Notation für die Darstellung von DOM-Strukturen, die wir in SuiWeb verwenden.
- SJDON ist eine Art Lisp-Syntax für DOM-Strukturen, die sich sehr gut für hierarchische Strukturen eignet.
- SJDON ist eine Art S-Expression, die sich sehr gut für die Verarbeitung durch JavaScript eignet.
- SJDON ist eine Art JSON, die sich sehr gut für die Verarbeitung durch JavaScript eignet.
- Im Gegensatz zu JSX ist SJDON reines JavaScript und kann direkt durch JavaScript verarbeitet werden, ohne vorgängigen Build Step mit Tools wie Babel.
- Das vereinfacht auch Live Demos im Unterricht, wo beim Editieren von SJDON-Strukturen direkt ein Live Preview angezeigt wird (ok, geht auch mit JSX, aber mit etwas mehr Aufwand, Live Server, etc.).
- Für Leute mit Lisp-Erfahrung ist SJDON eh die natürliche Art und Weise, hierarchische Strukturen darzustellen...

Promise Patterns

```javascript
// 1. Promise Erstellen
const myPromise = new Promise((resolve, reject) => {
    // Async operation
    if (success) {
        resolve(result);
    } else {
        reject(new Error('Failed'));
    }
});

// 2. Promise Verkettung
fetchUser(id)
    .then(user => fetchUserPosts(user.id))
    .then(posts => displayPosts(posts))
    .catch(error => console.error(error));

// 3. Promise.all fuer parallele Ausfuehrung
Promise.all([
    fetchUser(id),
    fetchPosts(id),
    fetchComments(id)
])
    .then(([user, posts, comments]) => {
        // Alle Promises erfuellt
    })
    .catch(error => {
        // Ein Promise wurde abgelehnt
    });

// 4. Promise.race
Promise.race([
    fetch('/endpoint1'),
    fetch('/endpoint2')
])
    .then(firstResult => {
        // Schnellste Response verarbeiten
    });

// 5. Async/Await Pattern
async function fetchData() {
    try {
        const user = await fetchUser(id);
        const posts = await fetchUserPosts(user.id);
        return posts;
    } catch (error) {
        console.error(error);
    }
}
```

Event Loop - Typische Fragen

```javascript
// Was ist die Reihenfolge der Ausgaben?
console.log('Start');

setTimeout(() => {
    console.log('Timeout 1');
}, 0);

Promise.resolve()
    .then(() => console.log('Promise 1'))
    .then(() => console.log('Promise 2'));

setTimeout(() => {
    console.log('Timeout 2');
}, 0);

console.log('End');

// Ausgabe:
// Start
// End
// Promise 1
// Promise 2
// Timeout 1
// Timeout 2
```

DOM Manipulation Patterns

```javascript
// 1. Elemente finden
const elem = document.getElementById('myId');
const elems =
    document.getElementsByClassName('myClass');
const elem = document.querySelector('.class #id');
const elems = document.querySelectorAll('div.class');

// 2. Element erstellen und einfuegen
const newElem = document.createElement('div');
newElem.textContent = 'New Element';
newElem.classList.add('myClass');
parentElem.appendChild(newElem);

// 3. Event Handling
element.addEventListener('click', (e) => {
    e.preventDefault();  // Default verhindern
    e.stopPropagation(); // Bubbling stoppen
});

// 4. Attribute manipulieren
element.setAttribute('class', 'newClass');
element.getAttribute('class');
element.hasAttribute('class');
element.removeAttribute('class');

// 5. Style manipulieren
element.style.backgroundColor = 'red';
element.classList.add('highlight');
element.classList.remove('old');
element.classList.toggle('active');
```

Node.js Server Patterns

```
// 1. Basic HTTP Server
const http = require('http');
const server = http.createServer((req, res) => {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
});
server.listen(8000);

// 2. File System Operations
const fs = require('fs').promises;

async function readAndProcessFile(filename) {
    try {
        const data = await fs.readFile(filename,
            'utf8');
        await fs.writeFile('output.txt',
            processData(data));
    } catch (error) {
        console.error('Error:', error);
    }
}

// 3. Express Route Handler
app.get('/api/items', async (req, res) => {
    try {
        const items = await db.getItems();
        res.json(items);
    } catch (error) {
        res.status(500).json({ error: error.message });
    }
});
```

Wrap-up

Überblick des Kurses

Hauptthemen

1. JavaScript Grundlagen
   - Sprache und Syntax
   - Objekte und Arrays
   - Funktionen und Prototypen
   - Asynchrone Programmierung
   - Node.js und Module
2. Browser-Technologien
   - DOM Manipulation
   - Events und Event Handling
   - Web Storage
   - Canvas und SVG
   - Client-Server Kommunikation
3. UI-Bibliotheken
   - Komponentenbasierte Entwicklung
   - JSX und SJDON
   - State Management
   - SuiWeb Framework

Weiterführende Themen

Modern Web Development

- Mobile Development
  - Responsive Design
  - Progressive Web Apps
  - React Native
- Performance
  - WebAssembly (WASM)
  - Code Splitting
  - Service Workers
- Alternative Technologien
  - TypeScript
  - Svelte
  - Vue.js

JavaScript Ecosystem Wichtige Tools und Frameworks:

- **Build Tools:**
  - Webpack
  - Vite
  - Babel
- **Testing:**
  - Jest
  - Testing Library
  - Cypress
- **State Management:**
  - Redux
  - MobX
  - Zustand

Best Practices Wichtige Prinzipien für die Web-Entwicklung:

- Clean Code
  - DRY (Don't Repeat Yourself)
  - KISS (Keep It Simple, Stupid)
  - Single Responsibility Principle
- Performance
  - Lazy Loading
  - Code Splitting
  - Caching Strategien
- Security
  - HTTPS
  - CORS
  - Content Security Policy

Ressourcen

Weiterführende Materialien

- **Dokumentation:**
  - MDN Web Docs: `https://developer.mozilla.org`
  - React Docs: `https://react.dev`
  - Node.js Docs: `https://nodejs.org/docs`
- **Bücher:**
  - Ëloquent JavaScript"von Marijn Haverbeke
  - "You Don't Know JS"von Kyle Simpson
  - "JavaScript: The Good Parts"von Douglas Crockford
- **Online Kurse:**
  - freeCodeCamp
  - Frontend Masters
  - Egghead.io

Kursabschluss Wichtige Lernergebnisse:

- Solides Verständnis von JavaScript
- Beherrschung der Browser-APIs
- Komponentenbasierte Entwicklung
- Moderne Web-Entwicklungspraktiken
- Basis für fortgeschrittene Themen

Integration and Final Concepts Exam Preparation

Key Integration Concepts Essential topics to focus on:

- Full-stack JavaScript applications
- Modern web development workflow
- Client-server architecture
- REST API principles
- Security considerations
- Performance optimization
- Testing strategies
- Best practices and patterns

Common Integration Patterns Key patterns to remember:

- MVC architecture
- Component-based development
- Event-driven architecture
- Asynchronous communication
- State management patterns
- Module bundling
- Progressive enhancement
- Responsive design

Final Exam Strategies

- Connect concepts across different areas
- Understand full request/response cycle
- Know security best practices
- Recognize performance optimization techniques
- Understand testing approaches
- Know common architectural patterns
- Remember REST API principles

Cross-Cutting Concerns Key aspects that span multiple areas:

- Error handling strategies
- Authentication and authorization
- Data management
- State synchronization
- Loading and error states
- Component communication
- API integration patterns

Sample Multiple Choice Questions

Architecture

Which statement about SPAs is correct?

- a) They always require a page reload for navigation
- b) They can't work without a backend server
- c) They manage routing on the client side - correct
- d) They don't support deep linking

Explanation: SPAs handle routing client-side without full page reloads

REST API

Which HTTP method is idempotent?

- a) POST
- b) PUT - correct
- c) PATCH
- d) None of the above

Explanation: PUT operations can be repeated without changing the result

Security

What prevents XSS attacks?

- a) CORS
- b) Content Security Policy - correct
- c) HTTPS
- d) Same-origin policy

Explanation: CSP helps prevent cross-site scripting attacks

Performance

What improves initial load time?

- a) Using more components
- b) Code splitting - correct
- c) Including all JavaScript in one file
- d) Disabling caching

Explanation: Code splitting allows loading only necessary code initially

Testing

Which testing approach tests components in isolation?

- a) End-to-end testing
- b) Integration testing
- c) Unit testing - correct
- d) System testing

Explanation: Unit tests focus on testing individual components in isolation

Critical Integration Patterns

```
// REST API Integration
const api = {
    async getUser(id) {
        const response = await
            fetch(`/api/users/${id}`);
        if (!response.ok) throw new Error('User not
            found');
        return response.json();
    },

    async updateUser(id, data) {
        const response = await
            fetch(`/api/users/${id}`, {
            method: 'PUT',
            headers: {
                'Content-Type': 'application/json'
            },
            body: JSON.stringify(data)
        });
        return response.json();
    }
};

// Component with API Integration
const UserProfile = () => {
    const [user, setUser] = useState(null);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);

    useEffect(() => {
        api.getUser(1)
            .then(setUser)
            .catch(setError)
            .finally(() => setLoading(false));
    }, []);

    if (loading) return ["div", {}, "Loading..."];
    if (error) return ["div", {}, error.message];
    return ["div", {}, user.name];
};

// Error Boundary Pattern
class ErrorBoundary {
    constructor() {
        this.state = { hasError: false };
    }

    static getDerivedStateFromError(error) {
        return { hasError: true };
    }

    render() {
        if (this.state.hasError) {
            return ["div", {}, "Something went wrong"];
        }
        return this.props.children;
    }
}
```