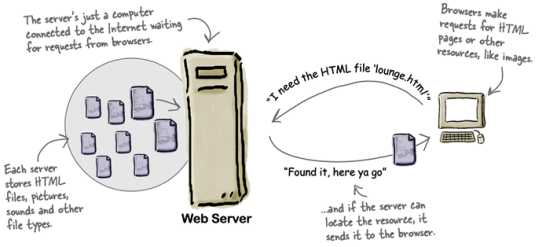
Web-Entwicklung

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
- Entwickelt von Tim Berners-Lee am CERN (1990er)
- Basiert auf: HTTP, HTML, URLs
- Wichtige Applikations- und Informationsplattform
- Unzählige Technologien und Spezifikationen

Web-Standards

- **W3C (World Wide Web Consortium)**
  - Standardisiert Technologien des WWW
  - Gegründet 1994 am MIT
  - Gründer und Vorsitzender: Tim Berners-Lee
- **WHATWG**
  - Web Hypertext Application Technology Working Group
  - Gründer: Apple, Mozilla, Opera, später Microsoft, Google
  - HTML Living Standard

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

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

Entwicklung Historische Entwicklung von Web-Technologien:

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

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

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

URL-Aufbau URL-Struktur:

```
scheme://[user[:password]@]host[:port]/path[?query][#fragme

Beispiel:
http://hans:geheim@example.org:80/demo/example.cgi?land=de&
```

- 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

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)

Grundlagen und Datentypen

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;
    case 2:
        // code
        break;
    default:
        // code
}

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

Objekte und Arrays

Objekt vs Array

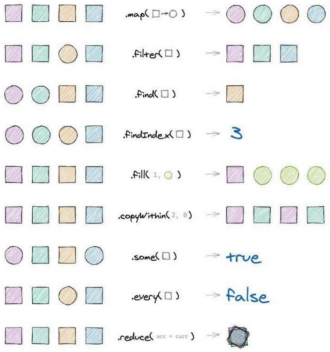
Was	Objekt	Array
Art	Attribut-Wert-Paare	Sequenz von Werten
Literalnotation	werte = {a: 1, b: 2}	liste = [1,2,3]
Ohne Inhalt	werte = {}	liste = []
Elementzugriff	werte["a"] oder werte.a	liste[0]

Objekt-Manipulation

```
// Objekt erstellen
let person = {
    name: "John",
    age: 30,
    greet() {
        return `Hello, I'm ${this.name}`;
    }
};

// Eigenschaften manipulieren
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;
```

Array-Methoden Wichtige Array-Operationen:

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



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}
```

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

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

Objekte erstellen und manipulieren

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

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

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 */
```

Filesystem

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

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

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

FS Funktionen

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

Datei-Informationen

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

Dateien lesen und schreiben

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

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

Asynchrone Programmierung

Asynchrone Programmierung

JavaScript verwendet verschiedene Mechanismen für asynchrone Operationen:

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

Event Loop

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

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')
})
```

Events und Promises

Event-Modul (EventMitter)

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

Listener hinzufügen

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

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

Event auslösen

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

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

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

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

Gibt nun ein Promise-Object zurück

Promises

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

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

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

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

Promise-Konstruktor erhält resolver-Funktion

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

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

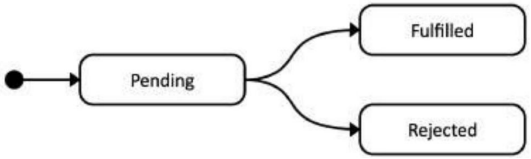
Aufruf neu:

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

Promise-Zustände

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

Nur ein Zustandsübergang möglich und Zustand in Promise-Objekt gekapselt



Promises Verknüpfen

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

Promise.all()

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

Promise.race()

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

Async/Await

```
// 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 };
}
```

ASYNC/AWAIT

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

Beispiel 2:

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

Promise Erstellung und Verwendung

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

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

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

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

- 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')
})
```

Testing mit Jasmine

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)
```

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

document-Objekt Wichtige Methoden des document-Objekts:

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

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

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

window-Objekt Das window-Objekt als globaler Namespace:

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

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

Document Object Model (DOM)

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 Navigation Zugriff auf DOM-Elemente:

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

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

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

DOM Manipulation Elemente erstellen und manipulieren:

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

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

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

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

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

DOM Navigation

```
// Element finden
const elem = document.getElementById('myId');
const elements = document.querySelectorAll('.myClass');
const firstMatch =
    document.querySelector('div.myClass');

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

DOM Manipulation

```
// 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';
```

Events

Event Handling Events sind Ereignisse, die im Browser auftreten:

- User Events: Klicks, Tastatureingaben
- Form Events: submit, change, input
- Document Events: DOMContentLoaded, load
- Window Events: resize, scroll
- Custom Events

Event Handling Events sind Ereignisse, die im Browser auftreten:

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

Event Handling Ereignisse wie Mausklicks oder Tastatureingaben können mit Event-Handlern behandelt werden:

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

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>
```

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\
```

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>
```

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

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

Event Handler Grundlegende Event Handling Schritte:
1. Event Listener registrieren:

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

2. Event Handler mit Event-Objekt:

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

Wichtige Event-Typen:

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

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 Listener

```
// 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 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

Wichtige Event-Typen

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

Event Bubbling und Capturing

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

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

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

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

Jquery

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

```
$("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

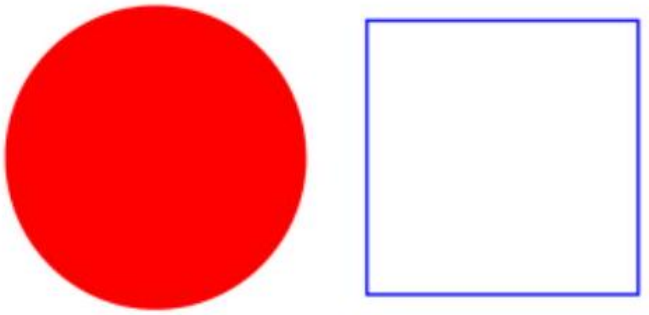
SVG und Canvas

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 mit JavaScript

```
let circle = document.querySelector("circle")
circle.setAttribute("fill","cyan")
```

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 und SVG

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 Grafiken 1. SVG erstellen:

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

2. SVG mit JavaScript manipulieren:

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

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

Vorteile SVG:

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

Canvas API 1. Canvas erstellen:

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

2. Context holen und zeichnen:

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

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

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

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

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

3. Transformationen:

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

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

// Zeichnen...

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

Wichtige Canvas-Methoden:

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

Canvas API

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

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

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

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

// Transformationen
ctx.save();
ctx.translate(100, 100);
ctx.rotate(Math.PI / 4);
ctx.scale(2, 2);
ctx.restore();
```

SVG Manipulation

```
// 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>

// SVG mit JavaScript
const circle = document.querySelector('circle');
circle.setAttribute('fill', 'green');
circle.setAttribute('r', '60');

// Event Handler
circle.addEventListener('click', () => {
    circle.setAttribute('fill', 'yellow');
});
```

Grafik im Browser Zwei Haupttechnologien für Grafiken:

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

Canvas Grundlagen

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

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

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

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

SVG Manipulation

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

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

svg.appendChild(circle);
```

Browser API

Web Storage
Web Storage speichert Daten auf der Seite des Client.

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 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 5MB abgespeichert werden.

```
1 localStorage.setItem("username","bkrt")
2 console.log(localStorage.getItem("username")) // ->
    bkrt
3 localStorage.removeItem("username")
```

Die Werte werden als Strings gespeichert, daher müssen Objekte mit JSON codiert werden:
1 Let user = {name: "Hans", highscore: 234}
2 localStorage.setItem(JSON.stringify(user))

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

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

History History gibt zugriff auf den Verlauf des akutellen Fensters/Tab.

```
1 function goBack() {
2 window.history.back();
3
    ,}
```

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

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)
```

Storage APIs Browser bieten verschiedene Möglichkeiten zur Datenspeicherung:

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

LocalStorage Verwendung

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

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

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

Web Storage

Storage APIs Browser bieten verschiedene Speichermöglichkeiten:

- localStorage: Permanente Speicherung
- sessionStorage: Temporär für aktuelle Session
- cookies: Kleine Datenpakete, auch für Server
- indexedDB: NoSQL-Datenbank im Browser

Local Storage

```
// 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
```

Local Storage 1. Daten speichern:

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

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

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

2. Session Storage (nur für aktuelle Session):

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

Wichtig zu beachten:

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

Client-Server-Interaktion (Formulare)

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 Attributen
- GET: Daten in URL (sichtbar)
- POST: Daten im Request-Body (unsichtbar)
- Verschiedene Input-Typen

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
})
```

Formular Handling

```
// 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();

    const formData = new FormData(form);
    const username = formData.get('username');

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

Formular Handling

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

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

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

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

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

Formular Events Wichtige Events bei Formularen:

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

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

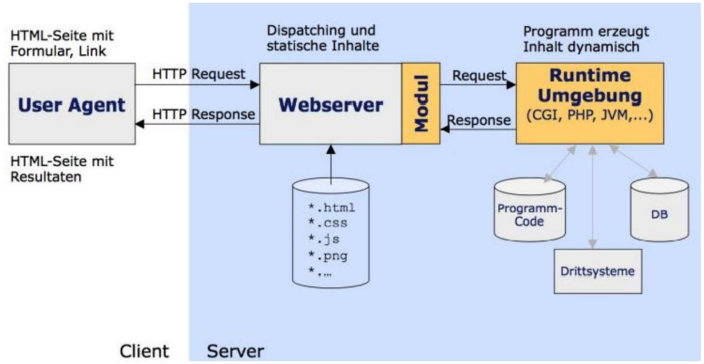
- submit, number, text, password, email, url , range , date , search , color

```
<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>
|'
```

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>
```



Formular Events

Events	Beschreibung
change	Formularelement geänder
input	Eingabe in Textfeld
submit	Formular absenden

GET/POST-Methode

```
<form action="http://localhost/cgi/showenv.cgi"method="get">
    <fieldset>
        <legend>Login mit GET</legend>
        <label for="login_get">Benutzername:</label>
        <input type="text" id="login_get" name="login"/>
        <label for="password_get">Passwort:</label>
        <input type="password" id="password_get" name="password"/>
        <label for="submit_get"></label>
        <input type="submit" id="submit_get" name="submit" value="Anmelden" />
    </fieldset>
</form>
```

Event Handling für Formulare

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

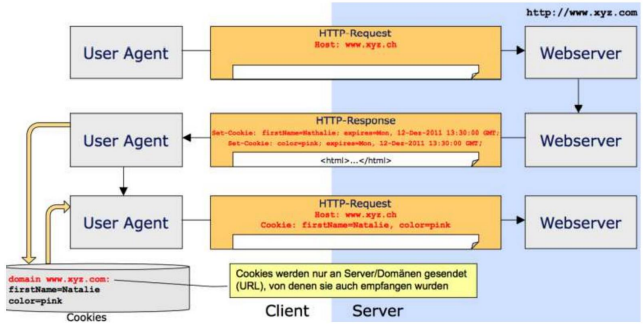
```
let form = document.querySelector("form");
form.addEventListener("submit", event => {
    event.preventDefault();
    console.log("Formular abgesendet!");
});
```

Cookies und Sessions

Cookies Cookies speichern clientseitig Daten:

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

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 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=/;";
```

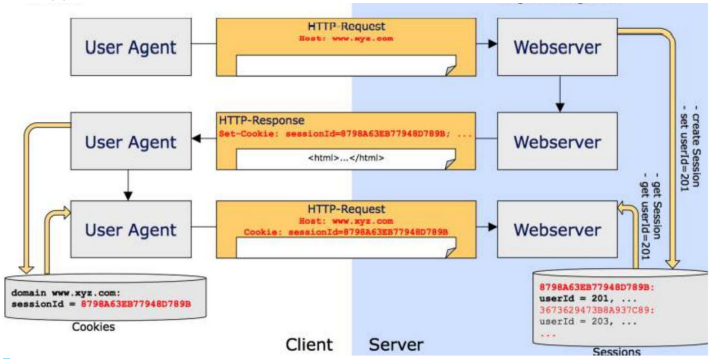
Sessions Server-seitige Speicherung von Benutzerdaten:

- 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)

Ablauf:
http://www.xyz.com



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);
```

Cookies

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

// Cookies 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=/;";
```

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))
```

Fetch API

- 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

Fetch API Grundlagen

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

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

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

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

```javascript
// 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 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

HTTP Requests mit Fetch

1. GET Request:

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

2. POST Request:

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

3. Mit async/await:

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

REST API Implementierung

```javascript
// 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'
});
```

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

```javascript
// 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

```javascript
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

HTTP-Methoden

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

REST API Implementierung mit Express

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

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

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

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

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

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

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

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

HTTP Status Codes

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

Browser APIs

Geolocation API 1. Einmalige Position abfragen:

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

2. Position kontinuierlich überwachen:

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

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

History API 1. Navigation:

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

2. History Manipulation:

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

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

3. Auf Änderungen reagieren:

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

Web Workers 1. Worker erstellen:

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

worker.postMessage({data: someData})

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

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

2. Worker beenden:

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

Wichtig:

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

UI-Bibliotheken und Komponenten

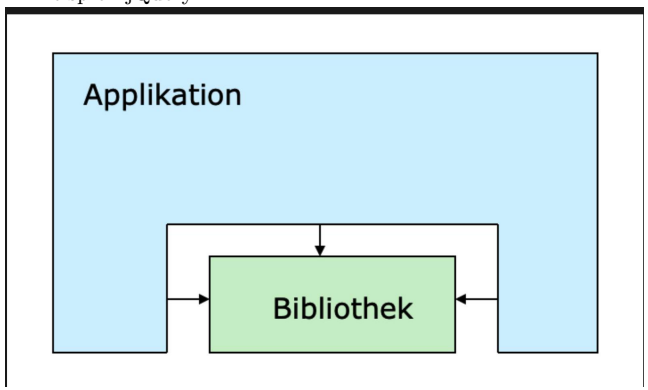
ÜBERSICHT

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

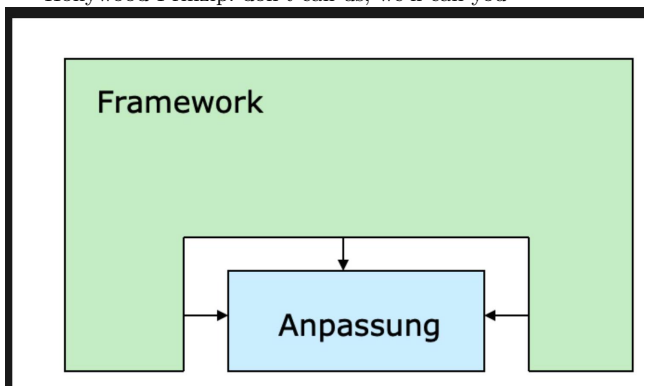
Frameworks und Bibliotheken

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

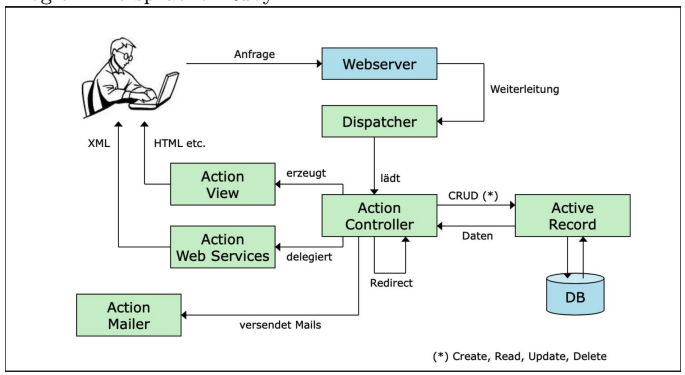
- Verschiedene Technologien möglich
- Zahlreiche Bibliotheken und Frameworks
- Verschiedene Architekturmuster
- Häufig: Model-View-Controller (MVC)
- Beispiel: Ruby on Rails

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

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

- Zum Vergleich der verschiedenen Ansätze
- Liste aus einem Array erzeugen

```
/* gegeben: */
let data = ["Maria", "Hans", "Eva", "Peter"]
<!-- DOM-Struktur entsprechend folgendem Markup aufzubauen: -->
<ul>
    <li>Maria</li>
    <li>Hans</li>
    <li>Eva</li>
    <li>Peter</li>
</ul>
```

DOM-SCRIPTING

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

- Erste Abstraktion: Listen-Komponente
- Basierend auf DOM-Funktionen

DOM-SCRIPTING

```
function init () {
    let app = document.querySelector(".app")
    let data = ["Maria", "Hans", "Eva", "Peter"]
    render(List(data), app)
}
function render (tree, elem) {
    while (elem.firstChild) { elem.removeChild(elem.firstChild) }
    elem.appendChild(tree)
}
```

DOM-SCRIPTING VERBESSERT

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

DOM-SCRIPTING VERBESSERT

- Damit vereinfachte List-Komponente möglich
- DOM-Funktionen in einer Funktion elt gekappselt

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

JQUERY

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

- List gibt nun ein jQuery-Objekt zurück
- Daher ist eine kleine Anpassung an render erforderlich

WEB COMPONENTS

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

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

\section*{REACT.JS}

const List = ({data}) => (

{ data.map(item => ({item})) }

)
const root = createRoot(document.getElementById('app'))
root.render(
<List data={["Maria", "Hans", Ëva", "Peter"]} />
)

- XML-Syntax in JavaScript: JSX
- Muss zu JavaScript übersetzt werden
- https://reactjs.org

\section*{VUE.JS}

https://vuejs.org

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

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

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
            }
        ])
    ]
];
```

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

Komponenten in SuiWeb

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

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

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

Container Komponenten

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

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

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

Komponenten-Design

Best Practices Grundprinzipien für gutes Komponenten-Design:

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

Komponenten-Struktur

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

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

    return [UserProfile, {user}];
};

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

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

Event Handling Behandlung von Benutzerinteraktionen:

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

Event Handling Beispiel

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

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

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

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

Optimierungen Möglichkeiten zur Performanzverbesserung:

- Virtuelles DOM für effizientes Re-Rendering
- Batching von State Updates
- Memoization von Komponenten
- Lazy Loading von Komponenten

Styling in SuiWeb

Styling in SuiWeb Verschiedene Möglichkeiten für Styles:

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

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"}
    ]
}]
```

Styling Best Practices

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

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;
};
```

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

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

```jsx
// 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 />;
};
```

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

Übungsaufgaben

JavaScript Grundlagen

Datentypen und Operatoren

Aufgabe 1: Was ist die Ausgabe folgender Ausdrücke?

```
typeof NaN
typeof []
typeof null
typeof undefined
[] == false
null === undefined
"5" + 3
"5" - 3
```

Lösung:

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

Funktionen und Scoping

Aufgabe 2: Was ist die Ausgabe dieses Codes?

```
let x = 1;
const f = () => {
    let x = 2;
    return {
        getX: () => x,
        setX: (val) => { x = val; }
    };
};
const obj = f();
console.log(x);        // ?
console.log(obj.getX()); // ?
obj.setX(3);
console.log(obj.getX()); // ?
console.log(x);        // ?
```

Lösung:

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

DOM und Events

DOM Manipulation

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

```
function createTodoList(containerId) {
    // Container finden
    const container =
        document.getElementById(containerId);

    // Input und Liste erstellen
    const input = document.createElement('input');
    const button = document.createElement('button');
    const list = document.createElement('ul');

    // Button konfigurieren
    button.textContent = 'Add';
    button.onclick = () => {
        if (input.value.trim()) {
            const li = document.createElement('li');
            li.textContent = input.value;
            list.appendChild(li);
            input.value = '';
        }
    };

    // Elemente zusammenfuegen
    container.appendChild(input);
    container.appendChild(button);
    container.appendChild(list);
}
```

Event Handling

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

```
document.getElementById('container').addEventListener('clic
    (e) => {
        if (e.target.matches('button')) {
            const count = (
                parseInt(e.target.dataset.count) || 0
            ) + 1;
            e.target.dataset.count = count;
            e.target.textContent = `Clicked ${count}
                times`;
        }
    }
);
```

Client-Server Kommunikation

Fetch API

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

```
async function apiRequest(url, method = 'GET', data =
    null) {
    const options = {
        method,
        headers: {
            'Content-Type': 'application/json'
        }
    };

    if (data) {
        options.body = JSON.stringify(data);
    }

    try {
        const response = await fetch(url, options);
        if (!response.ok) {
            throw new Error(`HTTP error:
                ${response.status}`);
        }
        return await response.json();
    } catch (error) {
        console.error('API request failed:', error);
        throw error;
    }
}
```

Formular-Validierung

Aufgabe 6: Erstellen Sie eine Formular-Validierung.

```
function validateForm(formId) {
    const form = document.getElementById(formId);

    form.addEventListener('submit', (e) => {
        e.preventDefault();

        const formData = new FormData(form);
        const errors = [];

        // Email validieren
        const email = formData.get('email');
        if (!email.includes('@')) {
            errors.push('Invalid email');
        }

        // Passwort validieren
        const password = formData.get('password');
        if (password.length < 8) {
            errors.push('Password too short');
        }

        if (errors.length === 0) {
            // Form submission logic
            console.log('Form valid, submitting...');
            form.submit();
        } else {
            alert(errors.join('\n'));
        }
    });
}
```

UI-Komponenten

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

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

    return [
        "div",
        ["h2", `Count: ${count}`],
        ["button",
            {onclick: () => setCount(count + 1)},
            "Increment"
        ],
        ["button",
            {onclick: () => setCount(count - 1)},
            "Decrement"
        ]
    ];
};
```

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

```
const UserList = () => {
    const [users, setUsers] = useState([]);
    const [loading, setLoading] = useState(true);

    if (loading) {
        fetchUsers()
            .then(data => {
                setUsers(data);
                setLoading(false);
            })
            .catch(error => {
                console.error(error);
                setLoading(false);
            });
    }

    if (loading) {
        return ["div", "Loading..."];
    }

    return [
        "div",
        ["h2", "Users"],
        ["ul",
            ...users.map(user =>
                ["li", `${user.name} (${user.email})`]
            )
        ]
    ];
};
```

Theoriefragen

Konzeptfragen **1. Erklären Sie den Unterschied zwischen == und === in JavaScript.**
Antwort: == vergleicht Werte mit Typumwandlung, === vergleicht Werte und Typen ohne Umwandlung.
2. Was ist Event Bubbling?
Antwort: Events werden von dem auslösenden Element durch den DOM-Baum nach oben weitergeleitet.
3. Was ist der Unterschied zwischen localStorage und sessionStorage?
Antwort: localStorage persistiert Daten auch nach Schließen des Browsers, sessionStorage nur während der Session.
4. Erklären Sie den Unterschied zwischen synchronem und asynchronem Code.
Antwort: Synchroner Code wird sequentiell ausgeführt, asynchroner Code ermöglicht parallele Ausführung ohne Blockierung.

Praktische Aufgaben

Implementierungsaufgaben **1. Implementieren Sie eine Funktion zur Deep Copy von Objekten.**
2. Erstellen Sie eine Funktion, die prüft ob ein String ein Palindrom ist.
3. Implementieren Sie eine debounce-Funktion.
4. Erstellen Sie eine Komponente für einen Image Slider.

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

```
const getData = () => {
    fetch('api/data')
        .then(response => response.json())
        .then(data => {
            return data;
        });
}
// Warum kommt undefined zurueck?
```

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

Example Exercises

JavaScript Fundamentals

Basic Array Manipulation Write a function that takes an array of numbers and returns a new array containing only the even numbers, doubled.

```
// Example solution
function processArray(numbers) {
    return numbers
        .filter(num => num % 2 === 0)
        .map(num => num * 2);
}

// Test
console.log(processArray([1, 2, 3, 4, 5, 6])); // [4, 8, 12]
```

Closure Implementation Create a function that generates unique IDs with a given prefix. Each call should return a new ID with an incrementing number.

```
// Example solution
function createIdGenerator(prefix) {
    let counter = 0;
    return function() {
        counter++;
        return `${prefix}${counter}`;
    };
}

// Test
const generateUserId = createIdGenerator('user_');
console.log(generateUserId()); // "user_1"
console.log(generateUserId()); // "user_2"
```

Async Programming Write an async function that fetches user data from two different endpoints and combines them. Handle potential errors appropriately.

```
async function getUserData(userId) {
    try {
        const [profile, posts] = await Promise.all([
            fetch(`/api/profile/${userId}`).then(r =>
                r.json()),
            fetch(`/api/posts/${userId}`).then(r =>
                r.json())
        ]);

        return {
            ...profile,
            posts: posts
        };
    } catch (error) {
        console.error('Failed to fetch user data:',
            error);
        throw new Error('Failed to load user data');
    }
}
```

DOM Manipulation

Dynamic List Creation Write a function that takes an array of items and creates a numbered list in the DOM. Add a button to each item that removes it from the list.

```
function createList(items, containerId) {
    const container =
        document.getElementById(containerId);
    const ul = document.createElement('ul');

    items.forEach((item, index) => {
        const li = document.createElement('li');
        li.textContent = `${index + 1}. ${item} `;

        const button =
            document.createElement('button');
        button.textContent = 'Remove';
        button.onclick = () => li.remove();

        li.appendChild(button);
        ul.appendChild(li);
    });

    container.appendChild(ul);
}
```

Component Implementation

Form Component Create a form component in SuiWeb that handles user input with validation and submits data to a server.

```
const UserForm = () => {
    const [formData, setFormData] = useState({
        username: '',
        email: ''
    });
    const [errors, setErrors] = useState({});

    const validate = () => {
        const newErrors = {};
        if (!formData.username) {
            newErrors.username = 'Username is
                required';
        }
        if (!formData.email.includes('@')) {
            newErrors.email = 'Valid email is
                required';
        }
        setErrors(newErrors);
        return Object.keys(newErrors).length === 0;
    };

    const handleSubmit = async (e) => {
        e.preventDefault();
        if (!validate()) return;

        try {
            await fetch('/api/users', {
                method: 'POST',
                headers: {'Content-Type':
                    'application/json'},
                body: JSON.stringify(formData)
            });
        } catch (error) {
            setErrors({submit: 'Failed to submit
                form'});
        }
    };

    return [
        "form",
        {onsubmit: handleSubmit},
        ["div",
            ["label", {for: "username"}, "Username:"],
            ["input", {
                id: "username",
                value: formData.username,
                oninput: (e) => setFormData({
                    ...formData,
                    username: e.target.value
                })
            }],
            errors.username && ["span", {class:
                "error"}, errors.username]
        ],
        ["div",
            ["label", {for: "email"}, "Email:"],
            ["input", {
                id: "email",
                type: "email",
                value: formData.email,
                oninput: (e) => setFormData({
                    ...formData,
                    email: e.target.value
                })
            }],
            errors.email && ["span", {class: "error"},
                errors.email]
```

API Implementation

REST API with Express Create a simple REST API for a todo list with Express.js, including error handling and basic validation.

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

let todos = [];

// Get all todos
app.get('/api/todos', (req, res) => {
    res.json(todos);
});

// Create new todo
app.post('/api/todos', (req, res) => {
    const { title } = req.body;

    if (!title) {
        return res.status(400).json({
            error: 'Title is required'
        });
    }

    const todo = {
        id: Date.now(),
        title,
        completed: false
    };

    todos.push(todo);
    res.status(201).json(todo);
});

// Update todo
app.patch('/api/todos/:id', (req, res) => {
    const { id } = req.params;
    const { completed } = req.body;

    const todo = todos.find(t => t.id ===
        parseInt(id));

    if (!todo) {
        return res.status(404).json({
            error: 'Todo not found'
        });
    }

    todo.completed = completed;
    res.json(todo);
});

app.use((err, req, res, next) => {
    console.error(err);
    res.status(500).json({
        error: 'Internal server error'
    });
});

app.listen(3000);
```

State Management Implement a shopping cart component that manages products, quantities, and total price calculation.

```
const ShoppingCart = () => {
    const [items, setItems] = useState([]);

    const addItem = (product) => {
        setItems(current => {
            const existing = current.find(
                item => item.id === product.id
            );

            if (existing) {
                return current.map(item =>
                    item.id === product.id
                        ? {...item, quantity:
                            item.quantity + 1}
                        : item
                );
            }

            return [...current, {...product, quantity:
                1}];
        });
    };

    const removeItem = (productId) => {
        setItems(current =>
            current.filter(item => item.id !==
                productId)
        );
    };

    const total = items.reduce(
        (sum, item) => sum + item.price *
            item.quantity,
        0
    );

    return [
        "div",
        ["h2", "Shopping Cart"],
        ["ul",
            ...items.map(item => [
                "li",
                ["span", `${item.name} x
                    ${item.quantity}`],
                ["span", `$${item.price *
                    item.quantity}`],
                ["button",
                    {onclick: () =>
                        removeItem(item.id)},
                    "Remove"
                ]
            ])
        ],
        ["div", `Total: $${total.toFixed(2)}`]
    ];
};
```

Browser APIs and Events

Custom Event System Implement a publish/subscribe system using browser events.

```
class EventBus {
    constructor() {
        this.eventTarget = new EventTarget();
    }

    publish(eventName, data) {
        const event = new CustomEvent(eventName, {
            detail: data,
            bubbles: true
        });
        this.eventTarget.dispatchEvent(event);
    }

    subscribe(eventName, callback) {
        const handler = (e) => callback(e.detail);
        this.eventTarget.addEventListener(eventName,
            handler);
        return () => {
            this.eventTarget.removeEventListener(eventName,
                handler);
        };
    }
}

// Usage
const bus = new EventBus();
const unsubscribe = bus.subscribe('userLoggedIn',
    (user) => {
    console.log(`Welcome, ${user.name}!`);
});

bus.publish('userLoggedIn', { name: 'John' });
unsubscribe(); // Cleanup
```

Drag and Drop Implement a simple drag and drop system for list items.

```
function initDragAndDrop(containerId) {
    const container =
        document.getElementById(containerId);
    let draggedItem = null;

    container.addEventListener('dragstart', (e) => {
        draggedItem = e.target;
        e.target.classList.add('dragging');
    });

    container.addEventListener('dragend', (e) => {
        e.target.classList.remove('dragging');
    });

    container.addEventListener('dragover', (e) => {
        e.preventDefault();
        const afterElement =
            getDragAfterElement(container, e.clientY);
        if (afterElement) {
            container.insertBefore(draggedItem,
                afterElement);
        } else {
            container.appendChild(draggedItem);
        }
    });

    function getDragAfterElement(container, y) {
        const draggableElements = [
            ...container.querySelectorAll('li:not(.dragging)')
        ];

        return draggableElements.reduce((closest,
            child) => {
            const box = child.getBoundingClientRect();
            const offset = y - box.top - box.height /
                2;

            if (offset < 0 && offset > closest.offset)
                {
                return { offset, element: child };
            }
            return closest;
        }, { offset: Number.NEGATIVE_INFINITY
            }).element;
    }
}
```

Data Manipulation and Algorithms

Deep Object Comparison Implement a function that deeply compares two objects for equality.

```
function deepEqual(obj1, obj2) {
    // Handle primitives and null
    if (obj1 === obj2) return true;
    if (obj1 == null || obj2 == null) return false;
    if (typeof obj1 !== 'object' || typeof obj2 !==
        'object')
        return false;

    const keys1 = Object.keys(obj1);
    const keys2 = Object.keys(obj2);

    if (keys1.length !== keys2.length) return false;

    return keys1.every(key => {
        if (!keys2.includes(key)) return false;
        return deepEqual(obj1[key], obj2[key]);
    });
}

// Test
const obj1 = {
    a: 1,
    b: { c: 2, d: [3, 4] },
    e: null
};
const obj2 = {
    a: 1,
    b: { c: 2, d: [3, 4] },
    e: null
};
console.log(deepEqual(obj1, obj2)); // true
```

Custom Promise Implementation Create a simplified version of the Promise API.

```
class MyPromise {
    constructor(executor) {
        this.state = 'pending';
        this.value = undefined;
        this.handlers = [];

        const resolve = (value) => {
            if (this.state === 'pending') {
                this.state = 'fulfilled';
                this.value = value;
                this.handlers.forEach(handler =>
                    this.handle(handler));
            }
        };

        const reject = (error) => {
            if (this.state === 'pending') {
                this.state = 'rejected';
                this.value = error;
                this.handlers.forEach(handler =>
                    this.handle(handler));
            }
        };

        try {
            executor(resolve, reject);
        } catch (error) {
            reject(error);
        }
    }

    handle(handler) {
        if (this.state === 'pending') {
            this.handlers.push(handler);
        } else {
            const cb = this.state === 'fulfilled'
                ? handler.onSuccess
                : handler.onFail;
            if (cb) {
                try {
                    const result = cb(this.value);
                    handler.resolve(result);
                } catch (error) {
                    handler.reject(error);
                }
            }
        }
    }

    then(onSuccess, onFail) {
        return new MyPromise((resolve, reject) => {
            this.handle({
                onSuccess: onSuccess || (val => val),
                onFail: onFail || (err => { throw err;
                    }),
                resolve,
                reject
            });
        });
    }

    catch(onFail) {
        return this.then(null, onFail);
    }
}

// Usage
new MyPromise((resolve, reject) => {
    setTimeout(() => resolve('Success!'), 1000);
```

Component Testing

Unit Testing Components Write tests for a form component using Jasmine.

```
describe('UserForm Component', () => {
    let form;

    beforeEach(() => {
        form = new UserForm();
    });

    it('should initialize with empty values', () => {
        expect(form.state.username).toBe('');
        expect(form.state.email).toBe('');
        expect(Object.keys(form.state.errors)).toHaveSize(0)
    });

    it('should validate email format', () => {
        form.state.email = 'invalid-email';
        const isValid = form.validate();

        expect(isValid).toBe(false);
        expect(form.state.errors.email)
            .toContain('Valid email is required');
    });

    it('should submit form with valid data', async ()
        => {
        form.state.username = 'testuser';
        form.state.email = 'test@example.com';

        spyOn(window, 'fetch').and.returnValue(
            Promise.resolve({ ok: true })
        );

        await form.handleSubmit();

        expect(window.fetch).toHaveBeenCalledWith(
            '/api/users',
            jasmine.any(Object)
        );
        expect(form.state.errors).toEqual({});
    });
});
```