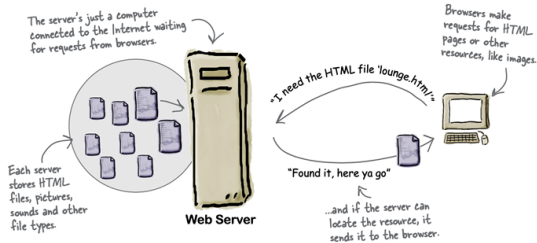


Einführung

WEB-Architektur Client-Server-Modell:

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



Internet vs. WWW

Internet:

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

World Wide Web:

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

Technologien

Client-Seitig → Front-end Entwickler

- Beschränkt auf Browser-Funktionalität
- HTML + CSS + JavaScript
- Browser APIs und Web-Standards

Server-Seitig → Back-end Entwickler

- Freie Wahl von Plattform und Programmiersprache
- Generiert Browser-kompatible Ausgabe
- Beispiele: Node.js, Express, REST APIs

URL-Aufbau URL-Struktur:

```
1 Schema:  
2 //[user[:password]@]host[:port]/path[?query][#fragment]  
3  
4 Beispiel:  
5 http://hans:1234@idk.org:80/demo?land=de&stadt=aa#fuck
```

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

JavaScript

Grundlagen und Datentypen

JavaScript Grundlagen

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

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

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

Datentypen Primitive Datentypen:

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

typeof-Operator

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

Variablenbindung

JavaScript kennt drei Arten der Variablendeklaration:

- `var`
 - Scope: Funktions-Scope
 - Kann neu deklariert werden
 - Wird geholt
- `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)

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

Verzweigungen, Wiederholung und Switch Case

- `if (condition) {...} else {...}`
- `switch (expression) { case x: ... break; default: ... }`
- `for (initialization; condition; increment) {...}`
- `while (condition) {...}`
- `do {...} while (condition)`
- `for (let x of iterable) {...}`

Kontrollstrukturen

```
1 // If-Statement  
2 if (condition) {  
3   // code  
4 } else if (otherCondition) {  
5   // code  
6 } else {  
7   // code  
8 }  
9  
10 // Switch Statement  
11 switch(value) {  
12   case 1:  
13     // code  
14     break;  
15   default:  
16     // code  
17 }  
18  
19 // Loops  
20 for (let i = 0; i < n; i++) { /* code */ }  
21  
22 while (condition) { /* code */ }  
23  
24 do { /* code */ } while (condition);  
25  
26 for (let item of array) {  
27   doSomething(item);  
28 }  
29  
30 for (let key in object) {  
31   doSomething(object[key]);  
32 }
```

Strings und reguläre Ausdrücke

Strings Strings in JavaScript sind:

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

Template Strings (mit Backticks) bieten erweiterte Funktionalität:

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

```
1 // String Interpolation
2 `half of 100 is ${100 / 2}` // "half of 100 is 50"
3
4 // Mehrzeilige Strings
5 `erste Zeile
6 zweite Zeile` // "erste Zeile\nzweite Zeile"
```

String Operationen

```
1 // String Verkettung
2 "con" + "cat" + "enate" // "concatenate"
3
4 // Template String mit Interpolation
5 const name = "World"
6 `Hello ${name}!` // "Hello World!"
7
8 // Mehrzeiliger Template String
9 const text = `
10 Erste Zeile
11 Zweite Zeile
12 `
```

Objekte und Arrays

Objekt vs Array

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

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

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

```
1 let person = {
2   name: "John",
3   age: 30,
4   greet() {
5     return "Hello, I'm" + this.name;
6   }
7 };
8 // Eigenschaften manipulieren/abfragen
9 person.job = "Developer"; // hinzufügen
10 delete person.age; // löschen
11 "name" in person; // true
12 // Objekte zusammenführen
13 Object.assign(person, {city: "Berlin"});
14 // Spread Syntax
15 let clone = {...person};
16 // Destrukturierung
17 let {name, job} = person;
```

JS Arrays spezielle Objekte: dynamische Grösse und Typen

```
1 let arr = [1, 2, 3, 4, 5];
2 // Elemente hinzufügen/entfernen
3 arr.push(6); // [1, 2, 3, 4, 5, 6]
4 arr.pop(); // [1, 2, 3, 4, 5]
5 // Elemente am Anfang hinzufügen/entfernen
6 arr.unshift(0); // [0, 1, 2, 3, 4, 5]
7 arr.shift(); // [1, 2, 3, 4, 5]
8 // Elemente einfügen/entfernen
9 arr.splice(2, 0, 2.5); // [1, 2, 2.5, 3, 4, 5]
10 arr.splice(2, 2); // [1, 2, 4, 5]
11 // Teilarray erstellen
12 arr.slice(1, 3); // [2, 4]
13 // Funktional
14 arr.map(x => x * 2); // [2, 4, 8, 10]
15 arr.filter(x => x > 3); // [4, 5]
16 arr.reduce((acc, x) => acc + x, 0); // 15
17 arr.copyWithin(0, 3, 5); // [4, 5, 8, 10, 5]
18 // Suchen und Testen
19 arr.every(x => x > 0); // true
20 arr.some(x => x > 5); // true
21 arr.find(x => x > 3); // 4
22 arr.findIndex(x => x > 3); // 3
23 // Iteration
24 arr.forEach(x => console.log(x));
25 // Arrays verbinden
26 arr.join(', '); // '1, 2, 3, 4, 5'
27 arr.concat([6, 7]); // [1, 2, 3, 4, 5, 6, 7]
28 // Sortieren/Umkehren
29 arr.sort(); // [1, 2, 3, 4, 5]
30 arr.reverse(); // [5, 4, 3, 2, 1]
```

JSON JavaScript Object Notation:

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

```
1 let obj = {type: "cat", name: "Mimi", age: 3};
2 let json = JSON.stringify(obj);
3 // '{"type":"cat","name":"Mimi","age":3}'
4
5 let parsed = JSON.parse(json);
6 // {type: 'cat', name: 'Mimi', age: 3}
```

Funktionen

Funktionen

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

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

Funktionsdefinition

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

Funktionsdefinitionen

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

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

```
1 function counter() {
2   let count = 0;
3   return {
4     increment: () => ++count,
5     decrement: () => --count,
6     getCount: () => count
7   };
8 }
9
10 const myCounter = counter();
11 myCounter.increment(); // 1
12 myCounter.increment(); // 2
13 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

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

Prototypen von Objekten

Prototypen

- 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

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

Klassen

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

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

Vererbung

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

Getter und Setter

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

Statische Methoden

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

```
1 class Person {
2   static create (name) {
3     return new Person(name)
4   }
5 }
6 let p36 = Person.create("John")
```

Asynchrone Programmierung

File API

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

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

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

FS Funktionen

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

Asynchrone Dateioperationen

```
1 const fs = require('fs')
2 fs.access('test.txt', fs.constants.R_OK |
3   fs.constants.W_OK, (err) => {
4   if (err) {
5     console.error('no access!')
6     return
7   }
8   console.log('can read/write')
9 })
```

Datei-Informationen

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

Dateien lesen und schreiben

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

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

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

Event Loop, Callbacks und Timer

Asynchrone Programmierung

JavaScript verwendet verschiedene Mechanismen für asynchrone Operationen:

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

Event Loop und Threads

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

Der event loop bearbeitet asynchrone tasks in folgender Reihenfolge:

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

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

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

Callbacks und Timer

```
1 // setTimeout
2 setTimeout(() => {
3   console.log('Delayed by 1 second');
4 }, 1000);
5
6 // setInterval
7 const id = setInterval(() => {
8   console.log('Every 2 seconds');
9 }, 2000);
10 clearInterval(id);
11
12 // Event Handler mit Callback
13 element.addEventListener('click', (event) => {
14   console.log('Clicked!');
15 });
```

SetTimeout

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

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

SetInterval

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

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

SetImmediate

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

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

Event-Modul (EventMitter)

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

Listener hinzufügen

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

Event auslösen

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

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

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

Promise Erstellung und Verwendung

```
1 // Promise erstellen
2 const myPromise = new Promise((resolve, reject) => {
3   // Asynchrone Operation
4   setTimeout(() => {
5     if (/* erfolg */) {
6       resolve(result);
7     } else {
8       reject(error);
9     }
10  }, 1000);
11 });
12 // Promise verwenden
13 myPromise
14   .then(result => {
15     // Erfolgsfall
16   })
17   .catch(error => {
18     // Fehlerfall
19   })
20   .finally(() => {
21     // Wird immer ausgeführt
22   });
```

Promise-Konstruktor erhält resolver-Funktion

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

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

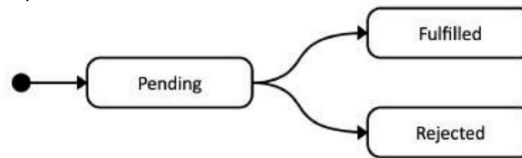
Aufruf neu:

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

Promise-Zustände

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

Nur ein Zustandsübergang möglich und Zustand in Promise-Objekt gekapselt



Promises Verknüpfen

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

Promise.all()

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

Promise.race()

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

Promise.all() und Promise.race()

```
1 // Promise.all
2 Promise.all([promise1, promise2])
3   .then(results => {
4     // Array mit allen Ergebnissen
5   });
6
7 // Promise.race
8 Promise.race([promise1, promise2])
9   .then(firstResult => {
10    // Erstes erfülltes Promise
11  });
```

Async/Await

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

Async/Await

```
1 // Async/Await Syntax
2 async function myAsync() {
3   try {
4     const result = await myPromise;
5     // Erfolgsfall
6   } catch (error) {
7     // Fehlerfall
8   }
9 }
10
11 // Async Funktion
12 async function getData() {
13   try {
14     const response = await fetch(url);
15     const data = await response.json();
16     return data;
17   } catch (error) {
18     console.error('Error:', error);
19   }
20 }
21
22 // Parallele Ausfuehrung
23 async function getMultipleData() {
24   const [data1, data2] = await Promise.all([
25     getData(url1),
26     getData(url2)
27   ]);
28   return { data1, data2 };
29 }
```

Webserver

Server im Internet

File-Transfer (File Server)

Web-Transfer-Protokolle

File-Transfer:

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

HTTP/HTTPS:

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

HTTP-Server

Ports

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

HTTP-Methoden

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

HTTP Status Codes

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

Node.js und Module

Node.js

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

Module System JavaScript verwendet verschiedene Modulsysteme:

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

Module Import/Export

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

Error Handling

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

Module System

```
1 // CommonJS (Node.js)
2 const fs = require('fs');
3 module.exports = { /* ... */ };
4
5 // ES Modules
6 import { function1 } from './module.js';
7 export const variable = 42;
8 export default class MyClass { /* ... */ }
9
10 // package.json
11 {
12   "type": "module",
13   "dependencies": {
14     "express": "^4.17.1"
15   }
16 }
```

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

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

Einfacher Webclient

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

Server und Client mit Streams

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

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


REST API

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

Express.js

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

Installation

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

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

Beispiel: Express Server

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

Routing

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

Test-Driven Development

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

Jasmine Tests

```
1 describe("Calculator", () => {
2   let calc;
3
4   beforeEach(() => {
5     calc = new Calculator();
6   });
7
8   it("should add numbers", () => {
9     expect(calc.add(1, 2)).toBe(3);
10  });
11
12  it("should throw on division by zero", () => {
13    expect(() => {
14      calc.divide(1, 0);
15    }).toThrow();
16  });
17 });
```

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

```
1 // Installation
2 npm install --save-dev jasmine
3
4 // jasmine.json
5 {
6   "spec_dir": "spec",
7   "spec_files": [
8     "**/*[sS]pec.js"
9   ],
10  "helpers": [
11    "helpers/**/*.js"
12  ]
13 }
14
15 // Test ausführen
16 npx jasmine
```

Beispiel (zugehörige Tests)

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

JASMINE: MATCHER

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

JASMINE: TESTS DURCHFÜHREN

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

Browser-Technologien

Vordefinierte Browser-Objekte

Browser-Objekte

Im Browser stehen spezielle globale Objekte zur Verfügung:

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

document-Objekt repräsentiert das aktuelle HTML-Dokument:

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

window-Objekt als globaler Namespace:

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

navigator-Objekt für Browser-Informationen:

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

history-Objekt für Browser-Verlauf:

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

location-Objekt für URL-Informationen:

```
1 location.href // URL der Seite
2 location.hostname // Hostname
3 location.pathname // Pfad
4 location.search // Query-Parameter
5 location.hash // Anker
```

Document Object Model (DOM)

structure:

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

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

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

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

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

DOM Manipulation Grundlegende Schritte zur DOM Manipulation:

1. Element(e) finden:

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

2. Elemente erstellen:

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

3. DOM modifizieren:

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

4. Attribute/Style setzen:

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

DOM Manipulation

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

DOM Navigation Zugriff auf DOM-Elemente:

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


Event Handling

Event Handling Events sind Ereignisse, die im Browser auftreten:

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

Event Handler Grundlegende Event Handling Schritte:

1. Event Listener registrieren:

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

2. Event Handler mit Event-Objekt:

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

Event Listener Event Listener registrieren und entfernen:

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

Event Informationen abfragen und steuern

```
1 // Event Listener hinzufuegen
2 element.addEventListener('click', (event) => {
3   console.log('Clicked!', event);
4
5   // Event Informationen
6   event.type // Art des Events
7   event.target // Ausloesende Element
8   event.currentTarget // Element mit Listener
9
10  // Event Steuerung
11  event.preventDefault(); // Default verhindern
12  event.stopPropagation(); // Bubbling stoppen
13 });
14
15 // Event Listener entfernen
16 element.removeEventListener('click', handler);
```

Event-Objekt

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

```
1 <script>
2   let button = document.querySelector("button")
3   button.addEventListener("click", (e) => {
4     console.log("x="+e.x+", y="+e.y)
5   })
6 </script>
```

Event Bubbling und Capturing

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

stopPropagation()

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

```
1 <script>
2   let button = document.querySelector("button")
3   button.addEventListener("click", (e) => {
4     console.log("x="+e.x+", y="+e.y)
5     e.stopPropagation()
6   })
7 </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.

```
1 <a href="https://developer.mozilla.org/">MDN</a>
2 <script>
3   let link = document.querySelector("a")
4   link.addEventListener("click", event => {
5     console.log("Nope.")
6     event.preventDefault()
7   })
8 ,/script>
```

Event Types

Event Typen Wichtige Event-Kategorien:

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

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

```
1 <p>Press Control-Space to continue.</p>
2 <script>
3   window.addEventListener("keydown", event => {
4     if (event.key == " " && event.ctrlKey) {
5       console.log("Continuing!")
6     }
7   })
8 </script>
```

Maus-Events

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

```
1 let button = document.querySelector("button")
2 button.addEventListener("click", () => {
3   console.log("Button geklickt!")
4 })
```

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

```
1 window.addEventListener("scroll", () => {
2   let max = document.body.scrollHeight -
3     window.innerHeight;
4   let bar = document.querySelector("#scrollbar");
5   bar.style.width = `${(window.pageYOffset / max) * 100}%`;
6 });
```

Focus-Events

Fokus- und Ladeereignisse

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

Jquery

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

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

```
1 $("#button.continue").html("Next Step...")
2 var hiddenBox = $("#banner-message")
3 $("#button-container button").on("click",
4     function(event) {
5         hiddenBox.show()
6     })
```

\$(Funktion) → DOM ready

```
1 $(function() {
2     // Code to run when the DOM is ready
3 });
```

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

```
1 $(".toggleButton").attr("title");
2 // Get the title attribute of elements with class
   'toggleButton'
```

```
1 $(".toggleButton").attr("title", "click here");
2 // Set the title attribute of elements with class
   'toggleButton' to 'click here'
```

```
1 $(".toggleButton").attr({
2     title: "click here",
3     // other attributes
4 });
5 // Set multiple attributes of elements with class
   'toggleButton'
```

```
1 $(".toggleButton").attr("title", function() {
2     // function to set title
3 }).css({
4     // CSS properties
5 }).text("New Text").on("click", function(event) {
6     // click event handler
7 });
```

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

```
1 $(".<li>...</li>").addClass("new-item").appendTo("ul");
2 // Create a new list item, add a class, and append it
   to a list
```

```
1 $(".<li>...</li>").length;
2 // Get the length of the new list item
```

```
1 $(".<li>...</li>")[0];
2 // Get the raw DOM element of the new list item
```

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

```
1 $(document.body);
2 // Wrap the body element in a jQuery object
```

```
1 $(this);
2 // Wrap the current element in a jQuery object
```

Graphics

Web-Grafiken

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

Grafik im Browser

 Zwei Haupttechnologien für Grafiken:

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

SVG

SVG

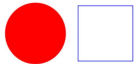
 Scalable Vector Graphics

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

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

Ausgabe:

Normal HTML here.



SVG Grafiken

1. SVG erstellen:

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

2. SVG mit JavaScript manipulieren:

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

Vorteile SVG:

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

SVG mit JavaScript

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

Wichtige SVG Attribute und Methoden

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

Canvas

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

```
1 <canvas></canvas>
2 <script>
3   let cx =
4     document.querySelector("canvas").getContext("2d")
5   cx.beginPath()
6   cx.moveTo(50, 10)
7   cx.lineTo(10, 70)
8   cx.lineTo(90, 70)
9   cx.fill()
10   let img = document.createElement("img")
11   img.src = "img/hat.png"
12   img.addEventListener("load", () => {
13     for (let x = 10; x < 200; x += 30) {
14       cx.drawImage(img, x, 10)
15     }
16   })
17 </script>
```

Canvas Methoden

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

Canvas API

 1. Canvas erstellen:

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

2. Context holen und zeichnen:

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

3. Transformationen:

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

Wichtige Canvas-Methoden:

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

Browser API

Browser-APIs Browser bieten APIs für verschiedene Aufgaben:

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

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

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

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

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

Ausserdem: Synchroner API-Zugriff

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

```
1 sessionStorage.setItem("sessionId", "abc123")
```

Web Storage speichert Daten auf der Client-Seite:

1. Local Storage (Daten speichern):

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

2. Session Storage (nur für aktuelle Session):

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

History gibt zugriff auf Verlauf des aktuellen Fensters/Tab.

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

History API 1. Navigation:

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

2. History Manipulation:

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

3. Auf Änderungen reagieren:

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

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

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

GeoLocation

Mit der GeoLocation-API kann der Standort abgefragt werden.

```
1 var options = { enableHighAccuracy: true, timeout:
2   5000, maximumAge: 0 }
3 function success(pos) {
4   var crd = pos.coords
5   console.log(`Latitude : ${crd.latitude}`)
6   console.log(`Longitude: ${crd.longitude}`)
7   console.log(`More or less ${crd.accuracy} meters.`)
8 }
9 function error(err) { ... }
10 navigator.geolocation.getCurrentPosition(success,
11   error, options)
```

Geolocation API 1. Einmalige Position abfragen:

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

2. Position kontinuierlich überwachen:

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

Web Workers 1. Worker erstellen:

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

2. Worker beenden:

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

Wichtig:

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

Client-Server-Interaktion (Formulare)

Formular Events

Formulare Formulare ermöglichen Benutzereingaben. Sie gilt als Grundlage für Interaktion mit dem Web.

Input types:

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

HTML-Formulare

Formulare ermöglichen Benutzereingaben und Datenübertragung:

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

Formular Handling 1. Formular erstellen:

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

2. Formular Events abfangen:

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

3. Formulardaten verarbeiten:

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

Event Handling für Formulare

Formular Events Wichtige Events bei Formularen:

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

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

```
1 let form = document.querySelector("form");
2 form.addEventListener("submit", event => {
3   event.preventDefault();
4   console.log("Formular abgesendet!");
5 });
```

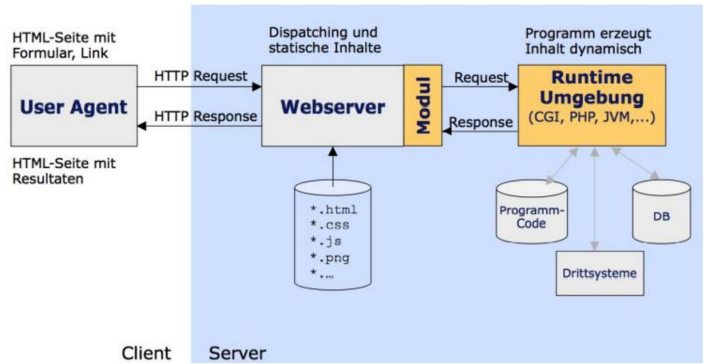
Formular Handling

```
1 // HTML Formular erstellen
2 <form action="/submit" method="POST">
3   <label for="username">Username:</label>
4   <input type="text" id="username" name="username">
5
6   <label for="password">Password:</label>
7   <input type="password" id="password"
8     name="password">
9
10   <button type="submit">Login</button>
11 </form>
12
13 // JavaScript Handler
14 form.addEventListener('submit', (event) => {
15   event.preventDefault(); // Verhindert
16   Standard-Submit
17
18   const formData = new FormData(form);
19   // Zugriff auf Formular-Daten
20   const username = formData.get('username');
21   const password = formData.get('password');
22
23   // Mit Fetch API senden
24   fetch('/submit', {
25     method: 'POST',
26     body: formData
27   });
28 });
```

GET/POST Methode

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

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



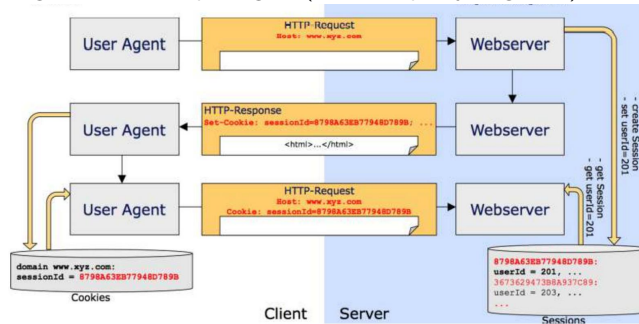
Formulare

```
1 <form>
2   <fieldset>
3     <legend>General information</legend>
4     <label>Text field <input type="text"
5       value="hi"></label>
6     <label>Password <input type="password"
7       value="hi"></label>
8     <label class="area">Textarea
9       <textarea>hi</textarea></label>
10   </fieldset>
11   <fieldset>
12     <legend>Additional information</legend>
13     <label>Checkbox <input type="checkbox"></label>
14     <label>Radio button <input type="radio"
15       name="demo" checked=></label>
16     <label>Another one <input type="radio"
17       name="demo"></label>
18   </fieldset>
19   <form>
20     <label>Button <button>Click me</button></label>
21     <label>Select menu
22     <select name="cars">
23       <option value="volvo">Volvo</option>
24       <option value="saab">Saab</option>
25       <option value="fiat">Fiat</option>
26       <option value="audi">Audi</option>
27     </select>
28     <input type="submit" value="Send">
29   </form>
30 | '
```



HTTP-Cookies

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



Cookies Cookies speichern clientseitig Daten:

```
1 document.cookie = "username=Max; expires=Fri, 31 Dec
2   2025 23:59:59 GMT";
3 console.log(document.cookie);
```

HTTP-Cookies sind kleine Datenpakete:

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

Cookie Handling 1. Cookie setzen:

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

2. Cookies lesen:

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

3. Cookie löschen:

```
1 document.cookie = "username=; expires=Thu, 01 Jan 1970
2   00:00:00 GMT; path="/"
```

Wichtige Cookie-Attribute:

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

Cookie Handling

```
1 // Cookie setzen
2 document.cookie = "username=John Doe; expires=Thu, 18
3   Dec 2024 12:00:00 UTC; path="/";
4
5 // Cookie lesen
6 const cookies =
7   document.cookie.split(';').reduce((acc, cookie)
8     => {
9     const [name, value] = cookie.trim().split('=');
10    acc[name] = value;
11    return acc;
12  }, {});
13
14 // Cookie löschen
15 document.cookie = "username=; expires=Thu, 01 Jan 1970
16   00:00:00 UTC; path="/";
```

Sessions Server-seitige Speicherung von Benutzerdaten:

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

Sessions

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

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

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


AJAX und Fetch API

AJAX Asynchronous JavaScript And XML:

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

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

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

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

```
1 fetch("example/data.txt")
2 .then(response => {
3     console.log(response.status) // -> 200
4     console.log(response.headers.get("Content-Type")) // ->
      text/plain
5 })
6 .then(resp => resp.text())
7 .then(text => console.log(text))
8 // -> This is the content of data.txt
```

Response Objekt

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

XMLHttpRequest und Fetch Moderne Ansätze für HTTP-Requests:

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

Fetch API Grundlagen

```
1 // GET Request
2 fetch('https://api.example.com/data')
3 .then(response => {
4     if (!response.ok) {
5         throw new Error('Network response was not
      ok');
6     }
7     return response.json();
8 })
9 .then(data => console.log(data))
10 .catch(error => console.error('Error:', error));
11
12 // POST Request
13 fetch('https://api.example.com/data', {
14     method: 'POST',
15     headers: {
16         'Content-Type': 'application/json',
17     },
18     body: JSON.stringify({
19         key: 'value'
20     })
21 })
22 .then(response => response.json())
23 .then(data => console.log(data));
24
25 // Mit async/await
26 async function fetchData() {
27     try {
28         const response = await
          fetch('https://api.example.com/data');
29         if (!response.ok) {
30             throw new Error('Network response was not
              ok');
31         }
32         const data = await response.json();
33         return data;
34     } catch (error) {
35         console.error('Error:', error);
36     }
37 }
```

HTTP Requests mit Fetch

1. GET Request:

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

2. POST Request:

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

3. Mit async/await:

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

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

```
1 // Login Request
2 async function login(username, password) {
3   const response = await fetch('/api/login', {
4     method: 'POST',
5     headers: {
6       'Content-Type': 'application/json',
7     },
8     credentials: 'include', // Fuer Cookies
9     body: JSON.stringify({
10       username,
11       password
12     })
13   });
14
15   if (response.ok) {
16     const user = await response.json();
17     // Session Token in localStorage speichern
18     localStorage.setItem('token', user.token);
19   }
20 }
21
22 // Authenticated Request
23 async function getProtectedData() {
24   const token = localStorage.getItem('token');
25   const response = await fetch('/api/protected', {
26     headers: {
27       'Authorization': `Bearer ${token}`
28     }
29   });
30   return response.json();
31 }
```

WebSocket Bidirektionale Echtzeit-Kommunikation:

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

```
1 const ws = new WebSocket('ws://localhost:8080');
2
3 ws.addEventListener('open', () => {
4   console.log('Connected to WebSocket');
5   ws.send('Hello Server!');
6 });
7
8 ws.addEventListener('message', event => {
9   console.log('Received:', event.data);
10 });
11
12 ws.addEventListener('close', () => {
13   console.log('Disconnected from WebSocket');
14 });
```

REST APIs

REST Prinzipien Representational State Transfer:

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

REST API Implementierung mit Fetch API:

```
1 // GET - Daten abrufen
2 fetch('/api/users')
3   .then(response => response.json())
4   .then(users => console.log(users));
5
6 // POST - Neue Ressource erstellen
7 fetch('/api/users', {
8   method: 'POST',
9   headers: {
10     'Content-Type': 'application/json',
11   },
12   body: JSON.stringify({
13     name: 'John',
14     email: 'john@example.com'
15   })
16 });
17
18 // PUT - Ressource aktualisieren
19 fetch('/api/users/123', {
20   method: 'PUT',
21   headers: {
22     'Content-Type': 'application/json',
23   },
24   body: JSON.stringify({
25     name: 'John Updated'
26   })
27 });
28
29 // DELETE - Ressource loeschen
30 fetch('/api/users/123', {
31   method: 'DELETE'
32 });
```

REST API Implementierung mit Express

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

UI-Bibliotheken und Komponenten

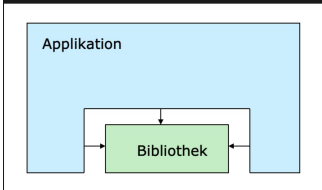
Overview

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

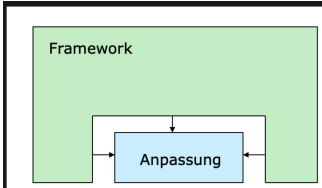
Frameworks und Bibliotheken

Framework vs. Bibliothek

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



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



ANSÄTZE IM LAUF DER ZEIT

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

SERVERSEITE

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

Architektur

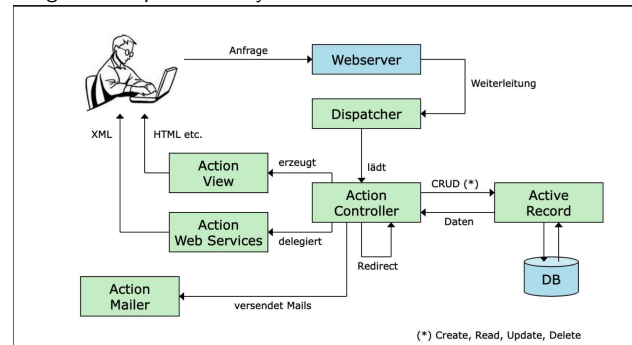
- **MVC (Model-View-Controller):**
 - Model: Repräsentiert Daten und Geschäftslogik, können Observer über Zustandsänderungen informieren
 - View: Bildet UI (z.B. HTML/CSS), kommuniziert mit Controller
 - Controller: Verarbeitet Eingaben (z.B. Clicks), aktualisiert Model
- **Single Page Apps (SPAs):**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

Model-View-Controller (MVC)

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

RUBY ON RAILS

- Serverseitiges Framework, basierend auf MVC
- Programmiersprache: Ruby



FOKUS AUF DIE CLIENT-SEITE

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

Komponentenbasierte Entwicklung Grundprinzipien:

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

DOM-Scripting und Abstraktionen

DOM-SCRIPTING

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

AUFGABE Liste aus einem Array erzeugen:

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

Entsprechendes Markup:

```
1 <ul>
2   <li>Maria</li>
3   <li>Hans</li>
4   <li>Eva</li>
5   <li>Peter</li>
6 </ul>
```

DOM-SCRIPTING

```
1 function List (data) {
2   let node = document.createElement("ul")
3   for (let item of data) {
4     let elem = document.createElement("li")
5     let elemText = document.createTextNode(item)
6     elem.appendChild(elemText)
7     node.appendChild(elem)
8   }
9   return node
10 }
```

DOM-SCRIPTING VERBESSERT

```
1 function elt (type, attrs, ...children) {
2   let node = document.createElement(type)
3   Object.keys(attrs).forEach(key => {
4     node.setAttribute(key, attrs[key])
5   })
6   for (let child of children) {
7     if (typeof child !== "string")
8       node.appendChild(child)
9     else
10      node.appendChild(document.createTextNode(child))
11   }
12   return node
13 }
```

Damit vereinfachte List-Komponente möglich:

```
1 function List (data) {
2   return elt("ul", {}, ...data.map(item => elt("li",
3     {}, item)))
4 }
```

JQUERY

jQuery List Implementation

```
1 function List (data) {
2   return $("

").append(...data.map(item =>
3     $("- ").text(item)))
4 }
5
6 function render (tree, elem) {
7   while (elem.firstChild) {
8     elem.removeChild(elem.firstChild)
9   }
10  $(elem).append(tree)
11 }

```

WEB COMPONENTS

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

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

JSX und SJDON

JSX

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

```
1 // JSX Komponente
2 const Welcome = ({name}) => (
3   <div className="welcome">
4     <h1>Hello, {name}</h1>
5     <p>Welcome to our site!</p>
6   </div>
7 );
8
9 // Verwendung
10 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

```
1 // SJDON Komponente
2 const Welcome = ({name}) => [
3   "div", {className: "welcome"},
4   ["h1", `Hello, ${name}`],
5   ["p", "Welcome to our site!"]
6 ];
7
8 // Verwendung
9 const element = [Welcome, {name: "Alice"}];
```

Vergleich JSX und SJDON

```
1 // JSX
2 const element = (
3   <div style={{background: 'salmon'}}>
4     <h1>Hello World</h1>
5     <h2 style={{textAlign: 'right'}}>
6       from Web Framework
7     </h2>
8   </div>
9 );
10
11 // SJDON
12 const element = [
13   "div", {style: "background:salmon"},
14   ["h1", "Hello World"],
15   ["h2", {style: "text-align:right"},
16     "from Web Framework"]
17 ];
```

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

```
1 const Counter = () => {
2   // State initialisieren
3   const [count, setCount] = useState(0);
4
5   // Event Handler
6   const increment = () => setCount(count + 1);
7   const decrement = () => setCount(count - 1);
8
9   return [
10    "div",
11    ["button", {onclick: decrement}, "-"],
12    ["span", count],
13    ["button", {onclick: increment}, "+"]
14  ];
15 };
16
17 // Komplexere State Objekte
18 const Form = () => {
19   const [state, setState] = useState({
20     username: '',
21     email: '',
22     isValid: false
23   });
24
25   const updateField = (field, value) => {
26     setState({
27       ...state,
28       [field]: value
29     });
30   };
31 };
```

Kontrollierte Eingabefelder

```
1 const InputForm = () => {
2   const [text, setText] = useState("");
3
4   return [
5     "form",
6     ["input", {
7       type: "text",
8       value: text,
9       oninput: e => setText(e.target.value)
10     }],
11     ["p", "Eingabe: ", text]
12   ];
13 };
```

Komponenten-Design

Container Components

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

Component Design Principles

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

Best Practices

 Grundprinzipien für gutes Komponenten-Design:

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

Komponenten-Struktur

```
1 // Container Komponente
2 const UserContainer = () => {
3   const [user, setUser] = useState(null);
4   useEffect(() => {
5     fetchUser().then(setUser);
6   }, []);
7
8   return [UserProfile, {user}];
9 };
10 // Praesentations-Komponente
11 const UserProfile = ({user}) => {
12   if (!user) return ["div", "Loading..."];
13   return [
14     "div",
15     ["h2", user.name],
16     ["p", user.email],
17     [UserDetails, {details: user.details}]
18   ];
19 };
```

Komponenten in SuiWeb

```
1 // Einfache Komponente
2 const MyButton = ({onClick, children}) => [
3   "button",
4   {
5     onclick: onClick,
6     style: "background: khaki"
7   },
8   ...children
9 ];
10 // Komponente mit State
11 const Counter = () => {
12   const [count, setCount] = useState(0);
13   return [
14     "div",
15     ["button",
16       {onclick: () => setCount(count + 1)},
17       `Count: ${count}`
18     ]
19   ];
20 };
```

Container Komponente

```
1 const TodoContainer = () => {
2   const [todos, setTodos] = useState([]);
3   // Daten laden
4   if (todos.length === 0) {
5     fetchTodos().then(data => setTodos(data));
6   }
7   // Event Handler
8   const addTodo = (text) => {
9     setTodos([...todos, {
10       id: Date.now(),
11       text,
12       completed: false
13     }]);
14   };
15   const toggleTodo = (id) => {
16     setTodos(todos.map(todo =>
17       todo.id === id
18       ? {...todo, completed: !todo.completed}
19       : todo
20     ));
21   };
22   // Render Praesentationskomponente
23   return [TodoList, {
24     todos,
25     onToggle: toggleTodo,
26     onAdd: addTodo
27   }];
28 };
29 // Praesentationskomponente
30 const TodoList = ({todos, onToggle, onAdd}) => [
31   "div",
32   [TodoForm, {onAdd}],
33   ["ul",
34     ...todos.map(todo => [
35       TodoItem, {
36         key: todo.id,
37         todo,
38         onToggle
39       }
40     ])
41   ]
42 ];
```

Container Komponenten

```
1 const MyContainer = () => {
2   let initialState = { items: ["Loading..."] };
3   let [state, setState] = useState(initialState);
4   if (state === initialState) {
5     fetchData()
6       .then(items => setState({items}));
7   }
8   return [
9     MyList,
10    {items: state.items}
11  ];
12};
```

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

```
1 const TodoList = () => {
2   const [todos, setTodos] = useState([]);
3   const addTodo = (text) => {
4     setTodos([...todos, {
5       id: Date.now(),
6       text,
7       completed: false
8     }]);
9   };
10   const toggleTodo = (id) => {
11     setTodos(todos.map(todo =>
12       todo.id === id
13       ? {...todo, completed: !todo.completed}
14       : todo
15     ));
16   };
17   return [
18     "div",
19     [TodoForm, {onSubmit: addTodo}],
20     [TodoItems, {
21       items: todos,
22       onToggle: toggleTodo
23     }]
24   ];
25};
```

Optimierungen Möglichkeiten zur Performanzverbesserung:

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

Styling in SuiWeb

Styling in SuiWeb Verschiedene Möglichkeiten für Styles:

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

Styling Best Practices

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

Style Optionen

```
1 // String Style
2 ["div", {style: "color: blue; font-size: 16px"}]
3 // Style Objekt
4 const styles = {
5   container: {
6     backgroundColor: "lightgray",
7     padding: "10px"
8   },
9   text: {
10    color: "darkblue",
11    fontSize: "14px"
12  }
13};
14 // Kombinierte Styles
15 ["div", {
16   style: [
17     styles.container,
18     {borderRadius: "5px"}
19   ]
20}];
```

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

```
1 // Function Component
2 const Welcome = ({name}) => {
3   return <h1>Hello, {name}</h1>;
4 };
5 // State Hook
6 const Counter = () => {
7   const [count, setCount] = useState(0);
8
9   return (
10     <div>
11       <p>Count: {count}</p>
12       <button onClick={() => setCount(count + 1)}>
13         Increment
14       </button>
15     </div>
16   );
17 };
18 // Effect Hook
19 const DataFetcher = () => {
20   const [data, setData] = useState(null);
21
22   useEffect(() => {
23     fetchData().then(setData);
24   }, []);
25
26   return data ? <DisplayData data={data} /> :
27     <Loading />;
28};
```

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

```
1 // Effiziente Listen-Rendering
2 const List = ({items}) => [
3   "ul",
4   ...items.map(item => [
5     "li",
6     {key: item.id}, // Wichtig fuer Performance
7     item.text
8   ])
9 ];
10 // Lazy Loading
11 const LazyComponent = async () => {
12   const module = await import('./Component.js');
13   return module.default;
14};
```


Wrap-up

Überblick des Kurses

Hauptthemen

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

Weiterführende Themen

Modern Web Development

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

JavaScript Ecosystem

 Wichtige Tools und Frameworks:

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

Best Practices

 Wichtige Prinzipien für die Web-Entwicklung:

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

Ressourcen

Weiterführende Materialien

- **Dokumentation:**
 - MDN Web Docs: <https://developer.mozilla.org>
 - React Docs: <https://react.dev>
 - Node.js Docs: <https://nodejs.org/docs>
- **Bücher:**
 - "Éloquent 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