# nodejs

**Web Server:** HTTP Anfragen/Anfragen annehmen/absenden, Actions ausführen basierend auf URL (z.B Statische Inhalte ausliefern, Dynamische Inhalte rendern)

Graphical user interface, application

Description automatically generated

**Wieso NodeJS:** Client/Server in JS, Ideal für Datenschnittstellen wie REST, Läuft überall, einfach zu deployen, Sehr schnelle Entwicklung möglich, Sehr modularer Aufbau

## nodejs grundlagen

**Asynchrone Programmierung:** Funktionen können als Parameter übergeben und später aufgerufen werden

function myFunc(a, b, fn) {  
 *setTimeout*(function () {  
 fn(a + b);  
 }, 1000);  
}  
myFunc(2, 4, console.log);  
myFunc(10, 3, console.error);

**Callback-Hell:** Verschachtelte asynchrone Funktionen. Lässt sich mit Promises oder async/await lösen.

**Event:** Callbacks sind 1 : 1 Verbindungen. Events sind 1 : \* Verbindungen

**I/O Modules:** HTTP/HTTPS, URL, File System, Console, UDP/Net, Crypto

## Module

Ein Modul kann Funktionalität und Werte anderen Modulen zu Verfügung stellen. Ein Modul kann von anderen Modulen exportierte Funktionalität und Werte nutzen. NodeJS verwendet 2 Module Systeme (CommonJS und ESM)

**NPM:** Node verwendet für die Module Verwaltung npm.

**Export:** Wird nur einmal durchlaufen. Das Module wird «ge-cached» und bei jeder Nachfrage wieder zurückgegeben.

export default {count: add, get: get};  
function add() { return ++counter; }

**Import:**

import counterA from './counter.mjs'; *// Named*import {add, get} from './counter.mjs';

**Resolve-Reihenfolge:**

1. Core Module z.B import fs from 'fs';
2. Falls der mit «.\», «..\» oder «\» startet z.B from './counter.mjs’;
3. Falls ein «Filename» angeben wurde z.B from 'counter';

**Package.json:** Beinhaltet die Information zum Projekt. Wird benötigt um ein Modul zu installieren/publishen.

**Package-lock.json:** Beschreibt den exakten Abhängigkeitsgraph vom Projekt. Garantiert, dass immer die gleichen Abhängigkeiten installiert werden. Beschleunigt die Installation.

# express

## controller

**Beispiel:**

import express from 'express';  
const app = *express*();  
app.listen(3000, function () {  
 console.log('Example app on port 3000!');  
});

**Middleware:** Express nutzt Middleware für die Request Bearbeitung. Middleware ist ein Stack von Anweisungen welche für ein Request ausgeführt wird.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Middleware 1 |  |  |
| Request |  | Middleware 2 |  | Response |
|  |  | Middleware 3 |  |  |

**Middleware registrieren:** Mit app.use(…) wird eine neue Middleware registriert. Die Reihenfolge der Registrierung bestimmt die Ausführungsreihenfolge.

import express from 'express';  
import *bp* from 'body-parser';  
const app = *express*();  
const router = *express*.*Router*();  
app.use(*express*.static(\_\_dirname + '/public'));  
app.use(bp.urlencoded({extended: false}));  
app.use(router);

**Middleware-Sammlung Connect:** Connect definiert die Middleware Logik und eine Sammlung von Middlewares (z.B body-parser, cookie-parser)

**Router-Middleware:** Middleware befindet sich auf dem Express Objekt.

* **router.all(path, [callback, ...] callback)**
  + Wird unabhängig vom der HTTP Methode aufgerufen. Dynamische Werte möglich (/order/:id/)
* **router.METHOD(path, [callback, ...] callback)**
  + Wird aufgerufen, falls die jeweilige HTTP Methode verwendet wurde

router.get('/', function (req, res) {  
 res.send('hello world');  
});

* **router.route(path)**
  + Kann benutzt werden um für einen Path verschiedene Methoden zu gruppieren

app.route('/book').get(function (req, res) {  
 res.send('Get a random book');  
}).post(function (req, res) {  
 res.send('Add a book');  
})

**Static-Middleware:** Statische Files ausliefern. Mehrere static-routes möglich

**Custom-Middleware:** Hat 3 Parameter request, response und next. Next kann aufgerufen werden, um die nächste Middleware aufzurufen.

function myLogger(options) {  
 options = options ? options : {};  
 return function *myInLogger*(req, res, next) {  
 console.log(req.method + ":" + req.url);  
 next();  
 }  
}

**Error-Middleware:** Bearbeitet Errors, welche von den Middlewares generiert wurden. Muss 4 Parameter haben und als letztes registriert werden. Wird aufgerufen, falls ein Error-Objekt dem Next-Callback übergeben wird.

app.use(function (err, req, res, next) {  
 console.error(err.stack);  
 res.status(500).send('Something broke!');  
});

## model

**Ziel:** Die Daten sollten in einem Modul verwaltet und abgespeichert werden. (Z.b mit Array, JSON, NoSQL oder SQL-Db).

**NoSQL DB mit nedb:** NoSQL Datenbanken sind Dokumenten-basierend. Jedes Dokument beinhaltet alle Daten, welche notwendig sind. Relationen können über «keys» manuell erstellt werden.

**Beispiel:**

import Datastore from '@seald-io/nedb';  
const db = new Datastore({  
 filename: './data/order.db',   
 autoload: true });

db.insert(order, function (err, newDoc) {  
 console.log(" insert");  
 if (callback) {  
 callback(err, newDoc);  
 } });

db.find({}, function (err, docs) {  
 callback(err, docs); });

db.update({\_id: id},   
 {$set: {"state": "DELETED"}},   
 {returnUpdatedDocs: true},   
 function (err, numDocs, doc) {  
 callback(err, doc); });

## view

Mit Template Engines kann HTML-Code generiert werden. «Einfache» Sprache um (komplexere) Templates zu unterstützen. Single Page Frameworks und Web-Frameworks nutzen Template Engines um Templates zu rendern. Trennt Controller und View.

**Beispiel:**

res.render("template", {  
 description: "List of some Movies",   
 items: songs});

<figure>  
 <ul>  
 {{#each items}}  
 <li>  
 <h3>{{title}}</h3>  
 <p>{{artist}}</p>  
 </li>  
 {{/each}}  
 </ul>  
 <figcaption>  
 <p>{{description}}</p>  
 </figcaption>  
</figure>

## session

**Cookie:** Ein Cookie repräsentiert ein kleines Stück Information. Der Server schreibt ein Cookie auf einen Client (set-cookie: …). Client schickt alle Cookies für aktuelle Seite zurück and Server (cookie: …).

Diagram

Description automatically generated

**Session:** HTTP-Stateless umgehen. Daten werden Server-seitig einem Benutzer zugeordnet.

**Authentifizierung:** Wer bin ich? (Passwort, Handy, Biometrie)

**Autorisierung:** Was darf ich?

**Token:** Ziel ist ein Stateless Server. Bei jeder REST Anfrage muss ein Token mitgegeben werden (Signatur, Ausstell-/Ablaufdatum). Vorteil ist dass jede Anfrage zu einem beliebigen Server gesendet werden kann. Nachteil ist dass Token geklaut werden kann.

**Token Generierung:** Bei einer Route müssen die persönlichen Daten angegeben werden. Falls korrekt, wird ein Token generiert und an den Client geschickt. Verschiedene Libraries wie OAuth, Express-JWT, etc.

## ajax

**Motivation:** User Experience verbessern mit Auto-Complete in Formularen. Nachträgliches Laden von Daten und Updaten von Website.

**Nachteile:** State der Applikation ist nicht bekannt. Undo/Redo-Stack (History) wird oft vergessen.

**Fetch:** Neue Variante von XMLHttpRequest. Fetch Response bietet Methoden an um die Daten vom Body zu interpretieren. Cookies werden nicht automatisch mitgesendet.

*fetch*('/login', {  
 method: 'POST',  
 headers: {'Content-Type':'application/json'},  
 body: JSON.stringify({  
 email: "admin@admin.ch",   
 pwd: "123456"})  
}).then(function (res) {  
 console.log(res);  
})

**SOP/CORS:** Same-Origin-Policy erlaubt XMLHttpRequest nur zur Origin. Cross-Origin Resource Sharing ist ein Mechanismus, um Cross-Site-Requests zu ermöglichen. Der Ziel-Server kann dem Client den Zugriff erlauben.

Diagram

Description automatically generated

# typescript (TS)

**Motivation:** TS hilft mittels «statischer Analyse», entdeckt Typos, erlaubt Spezifikation von Typen für Variablen/Parameter etc.

**Typ-Inferenz:** boolean, number, string, null, undefined und any. Any kann beliebigen Wert annehmen und kann einer beliebig anderen Variable zugewiesen werden.

**Strict Mode:** Keine untypisierten Variablen, null und undefined ist nicht mehr Teil der Basistypen und können auch nicht mehr zugewiesen werden.

**Variablendeklarationen:** Globale Variablen aus nicht TS-Files können mit dem Keyword declare deklariert werden. Variablen können als unknown deklariert werden, dessen Typ später definiert wird.

declare let myMagicVar: string;

Auf Tupeln wird keine Typen-Inferenz angewendet, bei Enums wird es. Bessere Alternative zu Enums sind String Literal Union Type.

let myInferredNumArray = [1, 2, 3];  
let myNotInferredTupel = [1, "abcd"];

enum StrColor {*Red* = "red", *Green* = "green"};  
let c: Color = StrColor.*Green*;  
type StrLitColor = "red" | "green";  
let c3: StrLitColor = "green";

**Union Type:**

type StringOrNumberType = string | number;   
let myVar2: StringOrNumberType;

**Funktionen:** Function Overloading erlaubt. Parameter können Default Value und optional sein

function f(sn: number | string = "", ns?: number): string {  
 return *String*(sn) + *String*(ns || "");  
}

*// Funktion als Parameter*function f2(numArray: number[],  
 numFun: (prev: number,  
 curr: number) => number): number {  
 return numArray.reduce(numFun);  
}

**Klassen:** Properties der Instanzen und der Klasse (static) werden im Kontext der Klasse definiert. Methoden und Properties können mit den Zusätzen "private" und "readonly" versehen werden.

class SportsCar {  
 constructor(  
 public make: string,   
 public color: SportsCarColor)   
 { *//kein code nötig }*}

**Interfaces:** Interfaces (Typen) können in Deklaration von Klassen genutzt werden und darf mehr als ein Interface implementieren. Readonly Variablen können nicht verändert werden und «?» bedeutet optional.

interface IPoint {  
 readonly y?: number;  
}  
class Item {  
 constructor(public description: string) {}  
}  
class POI extends Item implements IPoint {  
 constructor(public x: number,   
 public y: number,   
 description: string,   
 public likes?: number) {  
 super(description);  
 }  
}

**Typen vs Interface:** Vielfach lassen sich stukturierte Typen sowohl als Interface als auch als Type definieren. Aber nur ein «Interface» kann unter gleichen Namen erweitert werden und nur ein «Type» kann Werte spezifizieren.

**Type Assertion:** Type Assertion erlauben Spezialisierung und Generalisierung des Typs, aber kein Casting auf einen inkompatiblen Typ. Mit keyof lässt sich ein Ty paus einem anderen Typen ableiten

const myCanvas = document.getElementById("ina") as HTMLCanvasElement;

const x = "hello" as number; *// nicht erlaubt*

type Point = { x: number; y: number };   
type PK = keyof Point;

**Template Literal Types:** Mit Template Literal Types lassen sich «literal» Types aus anderen Typen ableiten.

type Shipper = 'UPS' | 'FEDEX' | 'DHL';   
type Type = 'Overnight' | 'Priority' | 'Economy';  
type PostMethod = `${Shipper}-${TrackingType}`;

**Generics:** Mit Generics lassen sich Funktionen und Strukturen beschreiben die Wiederverwendung mittels ‘Composition’ ermöglichen.

interface PointList<ItemType> {  
 itemList: ItemType[];  
}  
  
function filteredPointList<ItemType>(  
 pList: PointList<ItemType>,   
 inclPred: (item: ItemType) => boolean): ItemType[] {  
 return pList.itemList.filter(inclPred);  
}

# responsive layout

**Definition:** Web-Seiten mit responsivem Layout sind sowohl flexibel, als auch adaptiv

* Flexibel: Zusätzlicher Platz wird durch Verbreiterung der Elemente ausgefüllt. Dynamisches (grössenadaptives) Layout welches sich ohne Media-Queries umsetzen lassen
* Responsive: Zusätzlich Platz wird durch Umordnung und zusätzliche Elemente ausgefüllt. Dynamisches Layout welches für unterschiedliche Geräte, Display-Grössen und Medien separates m Layouts definiert. Umsetzung mit Media Queries.

Shape

Description automatically generated

**Vorgehensmodell für Responsive Layout:**

* Graceful Degradation bedeutet, dass eine Anwendung mit einem Grundgerüst voller Funktionalität in modernen Browsern erstellt wird und dann die Layers zu entfernen, um sicherzustellen, dass sie mit älteren Browsern funktioniert
* Progressive Enhancement ist das Gegenteil von Graceful Degradation. Anstatt alle Funktionen von Anfang an zu entwickeln, wird eine Webseite auf der Grundlage der von allen Browsern (und Browserversionen) unterstützten Funktionen erstellt. Dann werden fortgeschrittene Funktionen wie Ebenen hinzugefügt.

Timeline

Description automatically generated

## responsive layout

Spezifisches CSS für verschiedene Medien definieren.

**Typen:**

@media screen { ... }   
@media print { ... }

**Dimensionen:**

@media ([width|min-width|max-width]: 375px){...}  
@media ([height|min-height|max-height]: 7px){...}

**Mobile First:** Zuerst Layout für alle Screens und den kleinsten Screen (ohne Media-Query), dann nacheinander mit aufsteigenden @media CSS für die (Layout)-Anpassungen mit mehr Bildschirmbreite

@media screen and (min-width: 30em */\*480px\*/*) {  
 body:before {  
 content: "480px+";  
 }  
 .fourEighty {…}  
}  
@media screen and (min-width: 600px) {  
 body:before {…}  
 .sixHundred {…}  
}  
@media screen and (min-width: 700px) {…}

**Media/Feature Queries:**

@media (orientation: landscape) { ... }  
@supports not (display: grid) {  
 div {float: right;}  
}

**Media Operatoren:** Kombinierbar mit and, or, not und only.

@media (min-width: 20em) and (max-width: 30em){…}

**Mobile Geräte:** Anweisung ist wichtig, um die “Intelligenz“ mobiler Browser zu unterbinden. Nicht angemessen für responsive Sites

<meta name="viewport" content="width=device-width,initial-scale=1">

Graphical user interface, diagram, application

Description automatically generated

**Custom Properties:** In CSS können ‘nicht-standard’ Properties definiert werden. Starten mit 2 Minuszeichen. Custom Properties können anstatt konstanter Werte mit var(…) genutzt werden (auch in calc())

**Beispiel:**

.demo-root {  
 --color-1: red;  
 --color-2: blue;  
 */\*Werte Definition im Vorfahren\*/* --bg: linear-gradient(to right,

var(--color-1),

var(--color-2));  
}  
.box {  
 width: 200px;  
 height: 70px;  
 */\*Werte Nutzung im Nachfahren\*/* background: var(--bg);  
 color: white;  
}  
.variant {  
 */\*Anpassung Formel Werte im Nachfahren\*/* --color-1: blue;  
 --color-2: red;  
}

**Custom Properties JS:** Custom Properties können mit JS ausgelesen und verändert werden wie CSS Property-Werte.

const styles = *getComputedStyle*(document.documentElement);  
const red = *String*(styles.getPropertyValue('--red')).trim();

**Style Switching mit Data-Attribute:** Custom Properties können unter anderem zum Style-Switching genutzt werden. Die Kombination mit html-Data-Attribut und Regeln mit Attriut-Selektor vereinfachen das JS

:root {  
 */\*colors\*/* --color-light-red: #ffeaea;  
 --color-dark-violet: #47027e;  
}  
  
body[data-style='light'] {  
 --color: var(--color-dark-violet);  
 --bg-color: var(--color-light-red);  
}  
  
*/\*variables / switching\*/*body[data-style='dark'] {  
 --color: var(--color-light-red);  
  
 --bg-color: var(--color-dark-violet);  
}  
  
*/\*applying variables\*/*body {  
 color: var(--color);  
 background-color: var(--bg-color);  
}

document.querySelector('fieldset')  
 .addEventListener('change', (event) => {  
 const style = event.target.value;  
 document.body.setAttribute('data-style', style);  
 });

# accessibility

**Farbkontrast:** Wichtig für Personen über 50

Text

Description automatically generated

**Accessbility:**

* Bilder sollen immer einen Alt-Text haben
* Zoom sollte nicht unterbunden werden
* Animationen sollten abstellbar sein (Verringerung von Ablenkungen und Verhinderung von Epilepsie und Migräne)
* Alle wichtigen Input Elemente sind fokussierbar in der richtigen Reihenfolge mit Tastatur
* Screen Reader soll keine Heading Levels auslassen, Semantic Elements richtig nutzen, Skip-Links am Anfang der Seite und lang Attribut korrekt setzen
* Entwickler sollten die Verwendung des korrekten semantischen HTML-Elements der Verwendung von ARIA vorziehen
* Tabellen sollten mit Headings for Rows und Columns ausgestattet sein und Captions haben

# security

**OWASP Top 10:**

|  |  |
| --- | --- |
| Cross-Site-Scripting | Replay Attack |
| Remote Code Execution | Cryptographic Failures |
| Insecture Direct Object References | Identification & Authentication Failure |
| Cross-Site Request Forgery |  |

**Cross Site Scripting (XSS):** Website besitzt eine XSS-Verwundbarkeit, wenn es möglich ist den Server so zu manipulieren, dass Schadcode (JavaScript) eines Angreifers an Nutzer ausgeliefert wird und im Browser dieser Nutzer ausgeführt wird. Gegenmassnahme mit Input Sanitation.

**Remote Code Execution:** Webserver besitzt eine Code Injection "Vulnerability" wenn ein Angreifer den Server dazu bringen kann vom Angreifer eingeschleusten Code zum Ausführen zu bringen.

**Broken Access Control:** Beim behandeln von Formular-Submission-Requests (GET und POST) sollte überprüft werden, dass dies von einem vom Server für den Nutzer «frisch» ausgelieferte Formulare stammt.

**Cryptographic Failures:** Passwort oder Token wird nicht verschlüsselt übertragen. Unabhängig davon ob Query-Body/Request-Parameter.

**Identification & Authentication Failure:** Bei Problemen bei der Authentisierung und dem Session Management können externe Angreifer oder Angreifer mit einem validen Login auf Informationen zugreifen, welche nicht für sie bestimmt sind

**Sign-up Form Best Practices:**

* Use meaningful HTML elements: form, input, label and button
* Label each input with a label
* Use element attributes to access built-in browser features: type, name, autocomplete, required.
* Use autocomplete="new-password" and id="new-password" for the password input in a sign-up form, and for the new password in a reset-password form.
* Provide Show password functionality.
* Don't double-up inputs. Don’t force users to enter emails or passwords twice.

**Content Security Policy (CSP):** CSPs ermöglichen die Ausführung von bösartigem Code auf Webseiten zu verhindern. Quellen von Ressourcen (Skripten, Bilder etc.) können eingeschränkt werden. CSPs werden im HTTP Header definiert

**Cross-Origin Resource Sharing (CORS):** CORS Header ermöglichen Ressourcen (wie z.B. Bilder, Skripte oder Daten) von einem anderen Ursprungsort als der eigenen “Origin” zu laden.

# testing

**Unit Tests:** Getestet werden einzelne "Units" wie Klassen, Module (meist ein File). Automation ist relativ einfach. Herausforderung ist die Isolation der Units, asynchrone Operationen, Testdatengenerierung (z.B. Fuzzing)

**Integrationstests:** Getestet wird das Zusammenspiel 2 oder mehr "Units". Automation meist möglich. Herausforderungen ist die Isolation der Units, asynchrone Operationen, Simulation Browser & Benutzerinteraktion, Test mit Datenbank, Testdatengenerierung (z.B. Fuzzing)

**Funktionstests:** Getestet wird ob sich das System entsprechend spezifizierte funktionale Anforderungen (Use-Cases, User Stories, ...) verhält. Automation möglich mit speziellen Tools

**(Visuelle) Regressionstests:** Getestet wird ob Veränderungen im Code zu (unerwarteten) Änderungen im Verhalten (oder UI) führen. Automation möglich mit speziellen Tools

**Funktionale Systemtests:** Getestet wird das Zusammenspiel aller Systemkomponenten in der Zielumgebung Automation meist nur in Teilen möglich. Herausforderungen: Realistische aber vorhersagbare Umgebung

## tools

**Test-Runner:** Ein Rahmen der Tests entgegennimmt, ausführt und die Resultate anzeigt. Beispiele: Ava CLI, Jasmine, Jest, Mocha, Cypress

**Assertion Library:** Code zur Ausführung einzelner Tests (Unterstützung Testing Patterns) Beispiele: Assert, Ava Power-Assert, Chai, Expect.js

**Mocking Library:** Separierung von Units / Erstellung von Mocks etc. Beispiele: Expect.js, Proxyquire, Sinon.js

**DOM Handling :** Cypress, JSDom, Puppeteer (Headless), Playwright, Storybook (Regression), Enzyme (React)

**Mocha API:** Aufruf einer umfassenden Funktion describe() mit zwei Argumenten.

describe('Array', function () { *// Test Suite* describe('#indexOf()', function () {  
 beforeEach(function () {  
 this.testArray = [1, 2, 3]; /*/ Test Setup* });

*// Test Case*  
 it('return -1 when value not here', function () {

*// Assertion* expect(this.testArray.indexOf(4)).to.equal(-1); });  
 });  
});

**Unit Testing:** Alles testen das kaputt gehen könnte. Alles testen das kaputt gegangen war. Neuer Code ist sus bis unschuldigTest bewiesen.