

Node.js
Begrifflichkeiten
Website/Webapplikation: Beinhaltet die Logik der Anwendung. Web-Server: Nimmt Anfragen vom Netzwerk entgegen und leitet diese an die Website weiter. Server: Stellt Hardware und das OS für den Web-Server bereit.
Was ist Node.js?
Node.js ist eine JavaScript Runtime mit einem riesigen Package Ecosystem. Vorteile: Server und Client in selber Sprache (JavaScript), nativer Support von JSON (z.B. für REST APIs), einfaches und schnelles Deployment, modularer Aufbau mit zahlreichen Pack-ages, wenig "Magie". ⇒ Node.js baut auf Chrome's V8 JavaScript Engine auf. ⇒ Es ist aber kein Web-Framework und keine Programmiersprache
Event-Driven, Non-Blocking I/O-Model
Node.js verwendet nur einen Thread. Wird ein Event ausgelöst (z.B. onclick) nimmt Node dieses entgegen, übergibt es an eine API (z.B. File-API), registriert ein Callback-Event und arbeitet dann weiter. Wird das Callback-Event ausgelöst, wiederholt Node diesen Prozess bis zum Ende. ⇒ Sinnvoll für viele kleine, delegierbare Aufgaben in Single-Threaded Systemen. ⇒ Ab Node.js 12 sind jedoch auch "Worker Threads" möglich.
Callbacks und Events
Callbacks sind Funktionen, welche zu einem späteren Zeitpunkt aufgerufen werden. Events sind Ereignisse, auf welche man sich registrieren kann. <pre>fs.readFile('demo.txt', (err, data) => { /* ... */ }); button.addEventListener('click', (event) => { /* ... */ });</pre> ⇒ Callbacks sind 1:1 Verbindungen, Events sind 1:n Verbindungen.
Promises
Promises sind eine Callback-Alternative. Lösen die Callback-Hell. <pre>fs.promises.readFile('demo.txt') .then(data => { /* ... */ }) .catch(err => { /* ... */ });</pre> <div>Variante 1</div>
<pre>try { const data = await fs.promises.readFile('demo.txt') } catch (err) { /* ... */ }</pre> <div>Variante 2</div> ⇒ Der aktuelle Standard in Node.js sind immer noch Callbacks. ⇒ Viele Module bieten aber eine Promise-Variante zusätzlich an (z.B. fs.promises).
Module
Module sind Code-Pakete, welche Funktionalitäten für andere Module bereitstellen. Node verwendet den Package Manager npm und kennt zwei Systeme: <ul style="list-style-type: none">• CommonJS: Standard seit Beginn (module.js/module.cjs)• ESM: Verwendbar seit ECMAScript 2015 und Node.js 14 (module.mjs) <pre>function funcStuff() { /* ... */ } const valueStuff = '...'; module.exports = { funcStuff, valueStuff }; const module = require('./module.js'); module.funcStuff(); module.valueStuff;</pre> <div>CommonJS</div> <pre>export default function funcStuff() { /* ... */ } export const valueStuff = '...'; import funcStuff, { valueStuff } from './module.mjs'; funcStuff(); valueStuff;</pre> <div>ESM</div>
Auflösungsreihenfolge: 1. Core-Module ('fs'), 2. Pfade ('./module.mjs'). Suche relativ zum aktuellen Pfad, 3. Dateien ('module'). Suche in node_modules und danach bis zum File-System-Root. ⇒ Module werden vom System nur einmal geladen. ⇒ Core-Module: http, url, fs, net, crypto, dns, util, path, os, etc.
Weiteres
package.json: Beinhaltet die Projektinformationen (Name, Scripts, Packages, etc.) und ist zwingend. package-lock.json: Beinhaltet die exakten Package-Abhängigkeiten. Wird automatisch aktualisiert und gehört ins Repo. Native Module: Beinhaltten nativen Code. NVM: Versionsmanager für Node.js.
Express.js
Was ist Express.js
Express.js ist ein bekanntes Web-Framework für Node.js. Es baut auf dem MVC-Pattern auf und verwendet Middlewares für die Request-Bearbeitung.
Model-View-Controller (MVC)
Beschreibt ein Frontend Design Pattern. Model: Beinhaltet die Daten und Datenaufbereitung. View: Stellt die Daten dar. Controller: Verknüpft das Model mit der View. ⇒ Ziel ist "Separation of Concerns" (Alternativen dazu sind M/V/M, M/P/etc.)

Middlewares
Middlewares sind ein Stack von Anweisungen, welche für einen Request ausgeführt werden. Merke: Registrierungsreihenfolge = Ausführungsreihenfolge. ⇒ Express.js stellt ab V4 viele Middlewares zur Verfügung (zuvor "Connect"-Plugin). ⇒ Beispiele: bodyParser (bodyParser), cookieParser (cookieParser), Cors, etc.
Authentisierung
Benutzer (bzw. HTTP-Anfragen) lassen sich im Web über Cookies/Sessions oder Tokens authentifizieren. ⇒ Session/Cookies sind nicht Stateless. Tokens hingegen schon. ⇒ Authentisierung: Wer bin ich? (Ph, 2FA, etc.) Autorisierung: Was darf ich?
Cookies: Repräsentieren ein kleines Stück von Daten. Der Server schreibt die Cookies zum Client, der Client sendet alle seine Cookies bei jedem Request mit. Sessions: Bauen auf Cookies auf. Beim ersten Request wird eine Session-ID vom Server erstellt und als Cookie zum Client gesendet. Der Server kann den Benutzer nun bei jedem Request mit dieser ID identifizieren.
Weiteres
Routing: Zuordnung von Routes (GET /orders) zu Actions (orderController.getAll). Template Engines: Engines für das Definieren und Rendern von HTML-Templates über einen einfachen Syntax.
Verwendung von Express.js
<pre>import express from 'express'; import bodyParser from 'express'; import session from 'express-session'; import path from "path"; import exphbs from 'express-handlebars'; import customLogSessionMiddleware from './custom-middleware.js'; import router from './routes.js'; const app = express(); /* Middlewares */ app.use(express.static(path.resolve('public'))); // Static files app.use(session({ secret: '1234567', resave: false, saveUninitialized: true })); app.use(bodyParser.urlencoded({ extended: false })); // req.body app.use(customLogSessionMiddleware); app.use(router); /* Handlebars */ const hbs = exphbs.create({ extname: '.hbs' }); hbs.handlebars.registerHelper('concat', (a, b) => a + '-' + b); app.engine('hbs', hbs.engine); app.set('view engine', 'hbs'); app.listen(3001, '127.0.0.1', () => { console.log('Example app listening on port 3001!'); });</pre> <div>apps.js</div>
<pre>import express from "express"; import { controller } from './controller.js'; const router = express.Router(); router.get('/books/:id?', controller.getBooks) .route('/orders') .get(controller.getOrders) .post(controller.postOrders) // Multiple callbacks possible router.all('/', controller.default) // Uses pattern matching export default router;</pre> <div>routes.js</div>
<pre>class Controller { default(req, res) { res.render('index', { id: 0 }); } getBooks(req, res) { res.render('index', { id: req.params.id, books: [{ name: 'A' }] }); } getOrders(req, res) { /* ... */ } postOrders(req, res) { /* ... */ } } export const controller = new Controller();</pre> <div>controller.js</div>
<pre>function customLogSessionMiddleware(req, res, next) { console.log(req.session.counter); req.session.counter += 1; next(); /* Calls next middleware */ } export default customLogSessionMiddleware;</pre> <div>custommiddleware.js</div>
<pre><!DOCTYPE html> <html> <head> <meta charset="utf-8"> <title>Example App</title> </head> <body> {{{body}}} </body> </html></pre> <div>views/layouts/main.hbs</div>

<pre><h1>{{id}}</h1> <div>{{concat "book" id}}</div> {{#each books}} {{name}}: {{#root.id}} </each> </div></pre> <div>views/index.hbs</div>
NeDB
NeDB ist eine NoSQL-Datenbank . Alle Daten werden in JSON-Dokumenten abgespeichert. Relationen müssen manuell gesetzt und verwaltet werden (z.B. via doc..._id).
<pre>import Datastore from 'nedb' const db = new Datastore({ filename: './books.db', autoload: true }); db.insert({ name: 'A' }, (err, doc) => { /* ... */ }); db.find({ name: 'A' }, (err, docs) => { /* ... */ }); db.findOne({ name: 'A' }, (err, doc) => { /* ... */ }); db.update({ name: 'A' }, { name: 'B' }, {}, (err, num) => { /* ... */ });</pre> <div>datastore.js</div>
TypeScript
Was ist TypeScript?
TypeScript ist eine Programmiersprache , welche JavaScript mit Typen und weiteren Syntaxelementen ergänzt. Ein Pre-Processor übersetzt TypeScript in JavaScript, d.h. es existiert kein Runtime-Typechecking. Jedes valide JS ist valides TS. ⇒ Die Typen müssen oft installiert werden (npm i -D @types/node) ⇒ Statt Node.js kann auch ts-node verwendet werden (bietet JIT-Compilation von TS)
TS-Config
Strict Mode
noImplicitAny: Keine untypisierten Variablen. alwaysStrict: Verwendet automatisch ein "use strict" im JS-File. strictNullChecks: null und undefined sind nicht mehr Teil der Basistypen (explizite Deklaration nötig). strictFunctionTypes: Strenge Überprüfung von Funktionstypen. strictPropertyInitialization: Klasseneigenschaften müssen initialisiert werden. ⇒ Weitere Einstellungen sind noImplicitThis und strictBindCallApply
Spezifikation
Basistypen
boolean, number, string, null: Wie in anderen Sprachen. undefined: Variable deklariert, aber kein Wert zugewiesen. any: Beliebiger Wert. Zuweisung in beide Richtungen beliebig möglich. unknown: Unbekannter Typ. Zuweisung zu unknown beliebig möglich, Zuweisung von unknown erst nach Type Narrowing. Type Inference: Ohne Typdeklaration wird der Typ automatisch bestimmt. <pre>let aNumber: number = 1 let aString = 'abc' // Inferred string let aUndefined: undefined; let aUnknown: unknown; let aAny: any = true aNumber = aString // NOK aNumber = aAny // OK aAny = aString // OK aUndefined = undefined // OK aString = undefined // NOK (in Strict Mode) aUnknown = aNumber // OK aNumber = aUnknown // NOK if (typeof aUnknown === 'number'){ aNumber = aUnknown // OK }</pre>
Komplexe Typen und Typdeklaration
Arrays, Tupels und Enums wie in anderen Sprachen. Union Types: Zusammengesetzte Typen (boolean string). String/Number Literal Union Types: Zusammengesetzte Typen aus Text- oder Zahlenwerten, ähnlich wie Enums. Wichtig: Keine Type Inference bei Tupeln! Diese werden als 'Union Type Arrays' erkannt. Neue Typen können mit type customType = _ deklariert werden. <pre>enum EnumType { A, B, C } type UnionType = number undefined type StrLitUnionType = 'A' 'B' 'C' let aUnionType: UnionType; let anArray: number[] = [1, 2, 3] let aUnionArray = [1, 'abc'] // Inferred (number string)[] let aTuple: [number, string] = [1, 'abc'] let anEnum = EnumType.A let aStrLit: StrLitUnionType; aUnionType = 1 // OK aUnionType = undefined // OK aUnionType = 'abc' // NOK aUnionArray[0] = 'abc' // OK aTuple[0] = 'abc' // NOK aStrLit = 'B' // OK aStrLit = 'abc' // NOK</pre>
Type Narrowing
TypeScript verwendet Flussanalyse, um die tatsächlichen Typen von Variablen zu bestimmen. Zuweisungen werden so möglich. Achtung: unknown braucht explizites Type Narrowing mit typeof ...

<pre>let aNumberOrString: string number let aUnknown: unknown; let aNumber: number aNumber = aNumberOrString // NOK aNumber = aUnknown // NOK aNumberOrString = 1 // OK aUnknown = 1 // OK aNumber = aNumberOrString // OK aNumber = aUnknown // NOK if (typeof aUnknown === 'number'){ aNumber = aUnknown // OK }</pre>
Funktionen
Wie in anderen Sprachen. Erlauben Default und optionale Parameter. Mehrere Signaturen pro Funktion möglich (Function Overloading). Funktionen als Parameter möglich. <pre>function myFuncA(a: string number = '', b?: number): void {} function myFuncB(a: (num: number) => number): void {} myFuncA('abc', 1) // OK myFuncA('abc') // OK myFuncA() // OK myFuncB(num) => num + 1 // OK</pre>
Klassen und Interfaces
Wie in anderen Sprachen. Properties lassen sich im Konstruktor definieren (automatische Generierung und Initialisierung). Generics sind möglich. Mit readonly<T> können alle direkten Felder einer Klasse T unveränderlich gemacht werden. ⇒ Structural Typing verwendet natives "Duck-Typing" aus JavaScript. <pre>class AClass { #val1: number // Private: ES2022 private val2: string // Private: TS Default readonly val3: number // Public, Optional, Readonly static readonly VAL4 = 1 // Public, Readonly constructor(val1: number, val2: string, val3?: number) { this.#val1 = val1; this.val2 = val2; this.val3 = val3; } set val1(val: number) { this.#val1 = val } // ESE get val1() { return this.#val1 } // ESE interface AInterface<T> { readonly valA: T // Public (Never static, private, ...) valB?: boolean // Public (Never static, private, ...) func(): void // Public (Never static, private, ...) } class ASubClass<T> extends AClass implements AInterface<T> { constructor(public valA: T, private valC: number) { super(1, 'abc') } func() { /* ... */ } } let aClass = new AClass(1, 'abc', 2); let aSubClass = new ASubClass<number>(1, 2); aSubClass.val1 = 1; // Structural Typing let aIntA: AInterface<number> = { valA: 1, func() {} } // OK let aIntB: AInterface<number> = { val2: 'abc' } // NOK // Type Assertions let aTypeA = aSubClass as AClass; // OK let aTypeB = aSubClass as number; // NOK</pre>
Weiters
Globale Variablen aus nicht TS-Files können mit declare let aGlobal :: deklariert werden. Keyof und Template Literal Types erlauben die Generierung von speziellen String Literal Union Types. <pre>type Keys = keyof { x: any, y: any } // type Keys = 'x' 'y' type TempLit = my.\$(Keys) // type TempLit = 'my-x' 'my-y'</pre>