Node.js

Begrifflichkeiten

Webseite/Webapplikation: Beinhaltet die Logik der Anwendung. **Web-Server**: Nimmt Anfragen vom Netzwerk entgegen und leitet diese an die entsprechende Webseite weiter. **Server**: Stellt die Hardware und das OS für den Web-Server bereit.

Was ist Node.js?

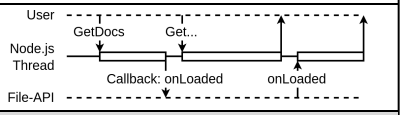
Node.js ist eine **JavaScript Runtime** mit einem riesigen Package Ecosystem. **Vorteile**: Server und Client in selber Sprache (JavaScript), nativer Support von JSON (z.B. für REST APIs), einfaches und schnelles Deployment, modularer Aufbau mit zahlreichen Packages, wenig "Magie".

⇨ 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. GetDocs) 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.

```
fs.readFile('demo.txt', (err, data) => { /* ... */ })
button.addEventListener('click', (event) => { /* ... */ });
```

⇨ Callbacks sind 1:1 Verbindungen, Events sind 1:n Verbindungen.

Promises

Sind eine Alternative zu Callbacks und lösen die Callback-Hell.

```
fs.promises.readFile('demo.txt')
    .then(data => { /* ... */ })
    .catch(err => { /* ... */ })
```
Variante 1

```
try {
    const data = await fs.promises.readFile('demo.txt')
} catch (err) {
    /* ... */
}
```
Variante 2

⇨ 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 Modulsysteme: **CommonJS**: Standard seit Beginn (`module.js` / `module.cjs`). **ESM**: Verwendbar seit ECMAScript 2015 und Node.js 14 (`module.mjs`).

```
function funcStuff() { /* ... */ }
const valueStuff = '...';

module.exports = { funcStuff, valueStuff };

const module = require('./module.js');
module.funcStuff(); module.valueStuff;
```
CommonJS

```
export default function funcStuff() { /* ... */ }
export const valueStuff = '...';

import funcStuff, { valueStuff } from './module.mjs';
funcStuff(); valueStuff;
```
ESM

Auflösungsreihenfolge: **1.** Core-Module (`'fs'`), **2.** Pfade (`'./module.mjs'`). Suche relativ zum aktuellen Pfad, **3.** Filenamen (`'module'`). Suche in `node_modules` und danach bis zum File-System-Root.

⇨ Module werden vom Systemnur 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:** Beinhalten nativen Code. **NVM:** Versionsmanager für Node.js.

Express.js

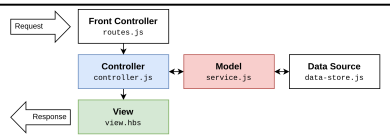
Was ist Express.js

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 "Seperation of Concerns" (Alternativen dazu sind MVVM, MVP etc.)



Middlewares

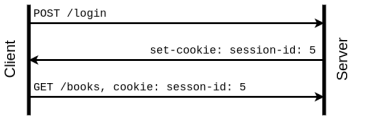
Middlewares sind ein **Stack von Anweisungen**, welche für einen Request ausgeführt werden. Die Reihenfolge der Registrierung bestimmt die Ausführungsreihenfolge.

⇨ Express.js stellt ab V4 viele Middlewares zur Verfügung (zuvor "Connect"-Plugin).
⇨ Beispiele: BodyParser (`bodyParser`), Cookie-Parser (`cookieParser`), Cors, etc.

Authentisierung

Cookies/Sessions

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 authentifizieren.



⇨ Session/Cookies sind nicht Stateless, Tokens hingegen schon.
⇨ Authentisierung: Wer bin ich? (Pin, 2FA, etc.) Autorisierung: Was darf ich?

Tokens

Tokens: Beinhalten alle Informationen für die Authentisierung und Autorisierung eines Benutzers (Ausstelldatum, Ablaufdatum, Benutzerdaten, etc.). Werden vom Server ausgestellt und vom Client bei jedem Request mitgesendet. **Vorteile**: Sind Stateless und Serverübergreifend (vgl. Git-Tokens) **Nachteile**: Können geklaut werden (Lösung: kurzes Ablaufdatum und Invalidierung).

JSON Web Tokens (JWT): Offener Standard für die Erstellung von Tokens. Wird über den HTTP-Header mitgesendet und beinhaltet Header (Algorithmen, etc.), Payload (Daten) und Signatur.

```
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJuYWlIjoiSm9obi-
BEb2UifQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

⇨ Einige Services bieten ihre Tokens öffentlich an (z.B. REST API Tokens)
⇨ Tokens nur über sichere Verbindungen senden

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. **AJAX**: Asynchrones Laden von Webinhalten (Filter, Single Page Applications, etc.). Schwierigkeiten bei SEO und Undo/Redo. Verwendung via `fetch('/page').then(res => res.json()).then(...)`

Verwendung von Express.js

```
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!');
});
```
app.js

```
import express from "express";
import { controller } from './controller.js'
const router = express.Router();

router.get('/books/:id/', controller.getBooks)
router.route('/orders/')
    .get(controller.getOrders)
    .post(controller.postOrders) // Multiple callbacks possible
router.all('/*', controller.default) // Uses pattern matching

export default router;
```
routes.js

```
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();
```
controller.js

```
function customLogSessionMiddleware(req, res, next) {
    console.log(req.session.counter);
    req.session.counter += 1;
    next(); /* Calls next middleware */
}
export default customLogSessionMiddleware;
```
custommiddleware.js

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Example App</title>
</head>
<body>
    {{{body}}}
</body>
</html>
```
views/layouts/main.hbs

```
<h1>{{id}}</h1>
<p>{{concat "book" id}}</p>
<ul>
    {{#each books}}
        <li>{{name}}: {{@root.id}}</li>
    {{/each}}
</ul>
```
views/index.hbs

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`).

```
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) => { /* … */ });
```
data-store.js

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 mitinstalliert 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.

```
let aNumber: number = 1
let aString = 'abc' // Inferred string
let aUndefined: undefined;
let aAny: any = true

aNumber = aString       // NOK
aNumber = aAny          // OK
aAny = aString          // OK
aUndefined = undefined  // OK
aString = undefined     // NOK (in Strict Mode)
```

Komplexe Typen und Typdeklaration

Arrays, **Tupels** und **Enums** wie in anderen Sprachen. **Union Types**: Zusammengesetzte Typen (`boolean | string`). **String/Number Literal Union Types**: Zusammengesetze 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.

```
enum EnumType { A, B, C }
type UnionType = number | undefined
type StrLitUnionType = 'A' | 'B' | 'C'

let aUnionType: UnionType;
let anArray: number[] = [1, 2, 3]
let anUnionArray = [1, 'abc'] // Inferred (number|string)[]
let aTupel: [number, string] = [1, 'abc']
let anEnum = EnumType.A
let aStrLit: StrLitUnionType;

aUnionType = 1          // OK
aUnionType = undefined  // OK
aUnionType = 'abc'      // NOK
anUnionArray[0] = 'abc' // OK
aTupel[0] = 'abc'       // NOK
aStrLit = 'B'           // OK
aStrLit = 'abc'         // NOK
```

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 …`.

```
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
}
```

Funktionen

Wie in anderen Sprachen. Erlauben Defaults und optionale Parameter. Mehrere Signaturen pro Funktion möglich (Function Overloading). Funktionen als Parameter möglich.

```
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
```

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.

Type Assertions: Erlauben Spezialisierung und Generalisierung eines Typs. Im Gegensatz zu Type Casting sind inkompatible Typen nicht erlaubt. **Structural Typing:** JS-Objekte können, wenn sie vom Typ her passen, an TS-Objekte zugewiesen werden.

⇨ Structural Typing verwendet natives "Duck-Typing" aus JavaScript.

```
class AClass {
 #val1: number              // Private: ES2022
 private val2: string       // Private: TS Default
 readonly val3?: number     // Public, Optional, Readonly
 static readonly VAL4 = 1   // Public, Static, Readonly

 constructor(val1: number, val2: string, val3?: number) {
     this.#val1 = val1; this.val2 = val2; this.val3 = val3;
 }
 set val1(val: number) { this.#val1 = val } // ES6
 get val1() { return this.#val1 }           // ES6
}

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> = { valZ: 'abc' }        // NOK
// Type Assertions
let aTypeA= aSubClass as AClass;    // OK
let aTypeB= aSubClass as number;    // NOK
```

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.

```
type Keys = keyof { x: any, y: any } // type Keys = 'x' | 'y'
type TempLit = `my-${Keys}` // type TempLit = 'my-x' | 'my-y'
```

Accessibility

Rechtliches

Seit 2021 gilt in der Schweiz für die öffentliche Verwaltung (Bund, SBB, etc.) der **eCH-0059** Accessibility Standard (Konformitätsstufe AA).

⇨ Wichtig für Menschen mit Seh-, Hör-, kognitiven oder motorischen Behinderungen.

Themen

Farbenblindheit: Informationen nicht nur mit Farben codieren. Stattdessen Mehrfach-Codierung anwenden (Farbe & Form, Farbe & Icon, etc.). Simulation u.a. via Chrome Dev Tools möglich.

Kontrast: Wichtig für Personen mit Sehschwäche oder Alter ü. 50. Kontraststufe 5.7:1 für ü. 50 (AA), 15.9:1 für ü. 80 (AAA). Testing u.a. via Chrome Accessibility Audit oder Wave-Plugin.

Auszeichnung von Medien: Bilder sollten immer einen Alt-Text haben. Art von Bild, Kontext, Inhalt und allenfalls Text als Zitat angeben. (*A comic strip of a confused man saying "Ok".*)

Zoombarkeit: Zoom nicht unterbinden (kein `user-scalable = 0`, etc.). Schrift soll via Browsereinstellungen separat skalierbar sein (Verwendung von `em`/`rem`/`%` statt `px`).

Animationen: Animationen sollten abstellbar sein, um Ablenkungen (z.B. bei ADHS), Epilepsie und Migräne zu verhindern (`@media (prefers-reduced-motion){ }`). Animationen von `opacity` immer in Kombination mit `visibility: visible|hidden` verwenden.

Bedienbarkeit mit der Tastatur: Alle wichtigen Elemente sind in der richtigen Reihenfolge fokussierbar (Tab). Fokus soll sichtbar sein. Dazu Standard-Controls nutzen und Controls nicht mit CSS umsortieren (kein `flex-direction: reverse`, etc.).

Screenreader: Für Unterstützung keine Heading-Levels auslassen, semantische Elemente verwenden (anstatt ARIA-Attribute), versteckte Skip-Links ("Zum Hauptinhalt") am Seitenanfang einbauen, Lang-Attribut setzen, Tabellen-Headings für Zeilen/Spalten verwenden, Captions verwenden, Inputs mit Labels versehen.

Custom-Controls: Nicht selber entwickeln. Bestehende Controls mit angepassten Styling verwenden (Star-Rating aus Radio-Buttons). Drag & Drop vermeiden. Gute Control-Libraries verwenden.

⇨ Automatische Accessibility Tests sind limitiert. Manuelle Tests immer empfohlen.
⇨ Top-5 Probleme: Kontrast, kein Alt-Text, leere Links, keine Labels, leere Buttons.

Wer braucht was?

Sehbehindert: Kontraste, Zoombarkeit, Trennung von Inhalt und Layout. **Hörbehindert**: Audiotranskription, visuelles Feedback, einfache Texte. **Blind**: Gute Dokumentstruktur, Alt-Texte, Tastaturbedienung. **Motorische Einschränkung**: Grosse Schaltflächen, Tastaturbedienung, keine ungewollten, automatischen Aktionen. **Lern- & Aufmerksamkeitsstörung**: Einfache Texte, lesbare Schriftarten, keine ablenkenden Elemente.

⇨ Betrifft 21% der Schweizer Bevölkerung.

Security

OWASP Top 10

A01: Broken Access Control **A02**: Cryptographic Failures **A03**: Injection **A04**: Insecure Design **A05**: Security Misconfiguration **A06**: Vulnerable and Outdated Components **A07**: Identification and Authentication Failures **A08**: Software and Data Integrity Failures **A09**: Security Logging and Monitoring Failures **A10**: Server-Side Request Forgery

Fokusthemen

A01.1: Insecure Direct Object References (IDOR)

Beschreibung: Eine Webseite ist verwundbar, wenn Daten ohne Sicherheitsmechanismen über direkte Referenzen erreichbar sind. **Beispiel**: Die persönlichen Daten eines Nutzers lassen sich über `my-site.ch/:userID` einsehen, ohne dass sichergestellt wird, dass der Aufrufer auch die entsprechenden Rechte hat. Ein Angreifer kann nun z.B. verschiedenen IDs ausprobieren, bis er auf die Seite eines Nutzers kommt.

Lösung: Alle Seiten mit korrekter Berechtigungskontrolle ausstatten. Bei Express.js z.B. via Middlewares, welche die `req.params.id` mit der aktuellen Nutzer-ID vergleichen.

A01.2: Cross-Site Request Forgery (CSRF)

Beschreibung: Eine Webseite ist verwundbar, wenn sie Formulare zusammen mit Cookies verwendet und die Herkunft des Formulars nicht überprüft. **Beispiel**: Ein Angreifer bringt einen eingeloggten User dazu, auf `evil.ch` zu gehen. Auf dieser Seite wird (automatisch) ein Formular an `my-site.ch/deleteAccount` gesendet. Da der Browser die Cookies der realen Webseiten mitsendet, wird die Aktion im Namen des Nutzers ausgeführt.

Lösung: Formulare mit einem CSRF-Token versehen und beim Request überprüfen. Bei Express.js z.B. das `csurf` Plugin verwenden. Alternativ keine Cookies verwenden (sondern z.B. JWT).

A01.3: Replay Attacks

Beschreibung: Eine Webseite ist verwundbar, wenn Aktionen unbeabsichtigt mehrmals ausgeführt werden können.

Beispiel: Eine Webseite belohnt einen Nutzer für jede korrekte Aufgabe mit Punkten. Der Nutzer kann nun eine einzige Aufgabe mehrmals absenden und erhaltet so mehrere Punkte.

Lösung: Keine "generische" Lösung möglich. In diesem Fall z.B. die Aufgaben abspeichern und nur einmal zählen.

A02: Cryptographic Failures

Beschreibung: Eine Webseite ist verwundbar, wenn Sie veraltete oder riskante Kryptoalgorithmen, mangelte Zufallszahlen, hart-kodierte Passwörter oder ähnliches verwendet. **Beispiel**: Eine Webseite sendet die Logindaten eines Benutzers unverschlüsselt und ohne HTTPs an den Server. Ein Angreifer kann nun mit einer Network Sniffing Software die Logindaten abhören.

Lösung: Verwendung von HTTPs. Keine sensitiven Informationen in der URL codieren. Externe Auth-Services nutzen.

A03.1: Cross Site Scripting (XSS)

Beschreibung: Eine Webseite ist verwundbar, wenn ein Angreifer seinen Schadcode so auf der Seite abspeichern kann, dass dieser im Browser eines Nutzers ausgeführt wird. **Beispiel**: Die Eingaben in ein Formular werden ohne "Escaping" an andere Nutzer ausgeliefert. Ein Angreifer kann nun z.B. `<img src='x' onerror='...'>` eingeben und so Code ausführen.

Lösung: "Escaping"/"Encoding" verwenden (`{{content}}` statt `{{{content}}}` in HBS). Eingabebereinigung ("Sanitizing") via Libraries (XSS, DOMPurify) einbauen. CSP-Header setzen. HTTPOnly-Cookies setzen.

A03.2 Remote Code Execution / Injection

Beschreibung: Ein Server ist verwundbar, wenn ein Angreifer den Server dazu bringen kann, seinen Schadcode (z.B. JavaScript, SQL, etc.) auszuführen. **Beispiel**: Die Eingaben in einem Formular werden mittels `eval(…)` vom Server in ein bestimmtes Format konvertiert. Ein Angreifer kann nun z.B. `while(true), process.exit()` (DoS) oder `res.end(fs.readdirSync(…).toString())` ausführen.

Lösung: Niemals `eval(…)`, sondern `parseInt(…)` oder `JSON.parse(…)` verwenden. Eingabebereinigung einbauen. Rechenintensive Tasks auslagern (gegen DDoS-Attacken). Node.js keine Root-Rechte geben. Globale Scopes und Variablen reduzieren.

⇨ `setTimeout(…)` und `setInterval(…)` verhalten sich ähnlich wie `eval(…)`

A07: Identification & Authentication Failure

Beschreibung: Eine Webseite ist verwundbar, wenn sie u.a. Brute-Force-Attacken erlaubt, schlechte Passwörter zulässt oder keine Multi-Faktor-Authentisierung verwendet.

Beispiel: Ein Angreifer bekommt Zugriff auf ein Nutzerkonto, indem er alle möglichen Passwortkombinationen durchprobiert. Da das Passwort 1234 war, ging die Attacke nur wenige Minuten.

Lösung: Authentisierung korrekt umsetzen oder externen Auth-Service nutzen.

Login & Password Handling

Grundsätzlich sollte man, wenn immer möglich, einen externen Auth-Service (z.B. via OAuth) verwenden. Eine Middleware stellt dabei sicher, dass Anfragen auf geschützte Bereiche nur durch autorisierte Benutzer erlaubt sind (z.B. via Token-Validierung).

⇨ Bei grossen Firmen mit eignen CySec-Teams sind eigene Auth-Services in Ordnung.
⇨ Möglichkeiten: OpenID Connect, Oauth, Passwordless (Magic-Links, WebAuthN, TOTP, etc.)

Weiteres

CSP-Header: Mechanismus, um das Laden von Ressourcen auf die angegebenen Domains zu beschränken (`Content-Security-Policy: default-src self trusted-service.com`). **HTTPOnly-, Secure-** und **SameSite-Cookies**: Blockiert den Zugriff von Client-Scripts sowie von anderen Webseiten.

`(app.use(csurf({cookie: true}) und im Formular <input name=_csrf value={{csrfToken}}). Keine Cookies verwenden (stattdessen z.B. JWTs).`

Testing

Testarten

Static: Statische Code-Analyse. **Unit**: Testen einer einzelnen Komponente (Klasse, Modul, etc.). **Integration**: Testen von mehreren Komponenten. **E2E/System**: Testen von allen Komponenten. **Funktional**: Testen von konkreten Anforderungen (Use-Cases). **Regression**: Testen auf potenzielle Fehler nach Code-Änderungen. **Weiteres**: Testen von Security, Usability, Performance, Stress, etc.

⇨ Oft wird dabei von einem SUT (System Under Test) gesprochen.

Struktur von Unit-Tests

Test-Gruppe/Fixture: Beinhaltet die Tests. **Test-Environment**: Umgebung, in welcher die Tests ausgeführt werden. **Doubles/Mocks**: Fake-Klassen fürs Testen.

Phasen: 1. Setup, 2. Exercise, 3. Verify, 4. Teardown

⇨ Achtung: z.T. wird auch das Test-Environment als «Fixture» bezeichnet.

Tools

Assertion Libraries: Erlauben eine einfache Spezifikation von Tests (Chai, Expect.js). **Test-Runners**: Führen die Tests aus (Mocha, Cypress, Jest). **Mocking Libraries**: Erlauben die Erstellung von Mocks (Proxyquire, Sinon.js). **DOM-Handling**: Erlauben das Testen von Web-UIs (Cypress, Puppeteer).

⇨ Assertion Library könnte mit throw new Error (…) ersetzt werden (Sinnlos).

Prinzipen

Alles Testen, das kaputt ging oder gehen könnte. Neuer Code ist immer ‹schuldig›, bis das Gegenteil bewiesen wurde. Vor einem Push ins Repo immer alle Tests laufen lassen. Mocks/Doubles mittels Dependency Injection zulassen.

Test Smells

Hard-to-Test Code, Production Bugs, Fragile Tests (Tests sind zu stark von interner Logik abhängig), Erratic Tests (manchmal erfolgreich, manchmal nicht), Developers not Writing Tests, Assertion Roulette (zu viele Assertions in einem Tests), Test Logic in Production Code, Obscure Tests (zu kompliziert), Slow Tests, Test Code Duplication, Conditional Test Logic (Codeteile werden nicht ausgeführt)

Verwendung (Chai)

Beginnt mit expect(…) .to.equal(1), .to.be.false, to.not.be.undefined, to.deep.equal(array), .to.be.a(number), .to.throw(TypeError), .to.be.an(‹array›).that.does.not.include(3), .to.have.any.key(‹key›)

```
import chai, {expect} from 'chai'
import chaiHttp from 'chai-http';
import jsdom from 'jsdom';
import app from './app.js'

chai.use(chaiHttp);

describe('My Test-Fixture', () => {
    let aNumber;
    beforeEach(() =>{
        aNumber = 1                              // Setup
    })
    it('Unit: Changed value is new value', () => {
        aNumber = 2                              // Exercise
        expect(aNumber).to.equal(2)        // Verify
    })
    it('Integration: Title contains id of book', () => {
        chai.request(app).get('/books/42').end((err, res) => {
            const dom = new jsdom.JSDOM(res.text)
            expect(dom.window.document.querySelector('h1')).to.cor
        });
    })
    afterEach(() => {
        aNumber = 0;                             // Teardown
    })
})
```

Internationalisierung

Begriffe

Internationalisierung (I18N): Programmieren auf eine Art, sodass Lokalisierung möglich wird. **Lokalisierung** (L10N) / **Globalisierung** (G11N): Anpassung des Programms (Sprache, Farbe, etc.) an die Sprachregion. **Übersetzung** (T9N): Übersetzung von Texten und Wörtern. **Locale**: Bezeichnung der Sprachregion (z.B. als String).

⇨ Herausforderungen: Sprache, Wortlänge, Schreibrichtung, Farbbedeutung, etc.

Best Practices

Layout: Elementpositionen an kulturelle Benutzergruppe anpassen (z.B. Leserichtung). Textlänge soll Hierarchie nicht beeinflussen (ungünstige Umbrüche). Layout nicht von der Wortsortierung abhängig machen. **Eingabefelder**: Unterschiede bei Postleitzahl, Telefonnummer, Provinzbezeichnung, etc. beachten. Vollständiger Name anstatt Vor- und Nachname verwenden. Unterschiedliche Datums- und Zahlenformate zulassen.

⇨ Je nach Region/Zielgruppe ist der Lokalisierungsaufwand grösser oder kleiner.
⇨ Nicht alles lässt sich automatisch lokalisieren (Denke: Farbe, Etiquette, etc.).

Umsetzung

Anführungszeichen: Werden bei `<q>…</q>` automatisch basierend auf `lang`-Attribut gesetzt. **Standard**: ES2021 Internationalisierung. **Bibliotheken**: FormatJS, Polyglot.js, i18next.

```
const regions = new Intl.DisplayNames(['en'], { type: 'region' })
regions.of('CH'); // Switzerland

const collator = new Intl.Collator(['de'])
collator.compare('Äh?', 'Zzz') // Smaller (-1)

const dFormatter = new Intl.DateTimeFormat(['en-US'])
dFormatter.format(Date.now()) // 7/9/2023
new Date(Date.now()).toLocaleString(['en-US']) // Alternative

const tFormatter = new Intl.RelativeTimeFormat(['en'])
tFormatter.format(-1, 'week') // 1 week ago

const nFormatter = new Intl.NumberFormat(['de'])
nFormatter.format(1250.50) // 1.250,5

const lFormatter = new Intl.ListFormat(['en'])
lFormatter.format(['Eggs','Milk','Fish']) // Eggs, Milk, and Fish

const plural = new Intl.PluralRules(['en-US'])
plural.select(1) // 'one': e.g. write '1 cat'
plural.select(10) // 'other': e.g. write '10 cats'

const swiss = new Intl.Locale('gsw') // Alternative zum String
```

ES2021 Internat.

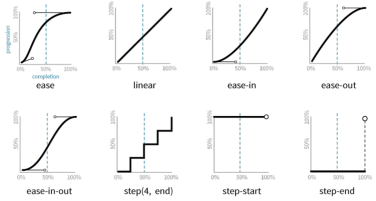
Animationen mit CSS

Transitions

Erzeugen einen weichen Übergang zwischen zwei Zuständen. Die Frames dazwischen werden automatisch ausgerechnet (Tweening).

`transition-property`: CSS-Property zum Animieren. `transition-duration`: Animationsdauer. `transition-timing-function`: Beschleunigungsfunktion. `transition-delay`: Zeitdauer, bevor die Animation startet.

Kurzschreibweise mit `transition`. Erlaubt auch mehrere Übergänge.

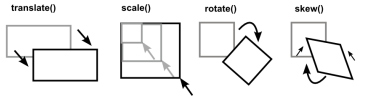



```
h1 {
    transition: all 300ms ease-in 0s,
                          color is linear 0.5s;
}
h1:hover {
    background: red;    /* See 'all' */
    color: white;               /* See 'color' */
}
```

⇨ Alle Animationen werden über eine Cubic Bezier Curve definiert.

Transform

Verändert die Form und Position eines Elements, wenn es angezeigt wird. Kann über Transitions animiert werden. Rotationen sind im Uhrzeigersinn. Der Ursprung kann mittels transform-origin: horizontal vertical; festgelegt werden. Bei 3D-Animationen lässt sich auf dem Parent-Element das perspective-Property setzen (je tiefer der Wert, desto extremer die Perspektive).




```
p { transform: rotate(30deg)          /* rotateX, Y, Z, 3d     */
               translate(20px, 5%)    /* translateX, Y, Z, 3d */
               scale(1.5)             /* scaleX, Y, Z, 3d      */
               skew(15deg);           /* skewX, Y              */
    transform-origin: left top;       /* Or px, %              */
    transition: transform 300ms ease; }

/* Looses translate and scale! */
p:hover { transform: rotate(60deg) skew(0deg); }
```

⇨ Bei mehreren Transforms ist die Anordnung relevant.
⇨ Achtung: Bei Zustandswechsel immer alle gesetzten Transforms wiederholen.

Keyframe Animationen

Erlaubt die Animation von einer Serie von Zuständen.

```
@keyframes my-animation {
    0% { background: red; }
    50% { background: green; }
    100% { background: blue; } }

h1 { animation-name: my-animation;
     animation-duration: 5s;
     animation-timing-function: linear;
     animation-iteration-count: 3;  /* Or infinite */
     animation-direction: normal;   /* Or revert, alternate */ }
```

Die meisten Properties mit quantitativen Werten oder Farben lassen sich animieren. Nicht quantifizierbar sind z.B. `border-style`, `display`, `auto`, `url(…)`, etc.

⇨ Mit Houdini (`@property`) lassen sich neu auch solche Properties animieren.

Weiteres

JS-Animationen: Mächtiger, aber nicht performant (CSS bevorzugen). **SVG-Animationen**: Sehr praktisch, wenn Browser-Support vorhanden.