

```
1 Node JS
Was muss ein Webserver können?
● HTTP Anfragen annehmen
● Actions ausführen basierend auf der Anfrage URL
● HTTP Antworten absenden
Request: Methoden GET, PUT, POST, ...
Response: Methoden writeHead, setHeader, statusCode, statusMessage, write, end
response.writeHead(200, { 'Content-Length': body.length, 'Content-Type': 'text/plain' });
response.setHeader("Content-Type", "text/html");
response.statusCode = 404;
response.statusMessage = 'Not found';
response.write("Data");
response.end("Data");
```

Module:
Node verwendet für die Module Verwaltung npm.

Import/Export

```
export router; // Variable
import router from './file.js';
export {function, otherFunction}; // several Functions
import controller from './controller.js';
export const noteService = new NoteService(); // Class
import {noteService} from './noteServices.js';
import express from "express"; // ES6
```

package.json:

- Beinhaltet die Informationen zum Projekt
- Wird benötigt um es zu publishen
- Wird benötigt um Module zu installieren
- Definiert Skripts (bsp: npm run test)

```
{ "name": "my_package",
  "description": "",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1" },
  "repository": {
    "type": "git",
    "url": "https://github.com/mgfeller/my_package.git" },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/mgfeller/my_package/issues" },
  "homepage": "https://github.com/mgfeller/my_package"
```

2 Express

Server starten:

```
import http from "http";
import express from "express";
const app = express();
const server = http.createServer(app);
const hostname = '127.0.0.1'; const port = '3000';
server.listen(port, hostname, () => {
  console.log(`Running at http://${hostname}:${port}`);
});
```

JSON (JavaScript Object Notation):

- Ist ein Daten-Austauschformat
- Wird verwendet um Daten zu senden und speichern
- Hat im Web XML verdrängt
- Wird oft mit AJAX verwendet
- Datentypen: String, Number, Boolean, Array, Object, null
- JSON-Helper: JSON.parse & JSON.stringify

MVC-Pattern:

- Model: Daten und Datenaufbereitung
- Controller: Verknüpft die View mit den Daten
- View: Darstellen der Daten

Middleware:

- Wird für Request Bearbeitung gebraucht
- Middleware ist ein Stack von Anweisungen welche für einen Request ausgeführt wird
- Neue Middleware registrieren mit: app.use(...);

2.1 Routing

Router-Middleware:

```
// Middleware befindet sich auf dem Express Objekt
import express from "express";
const router = express.Router;
// HTTP Methoden (get, put, post, delete)
router.get('/', function(req, res) {
  res.send('hello world'); });
// Mehrere Methoden auf selbem Link mit .route
app.route('/book') // oder router.route(...)
  .get(function(req, res) {res.send('Get a book')});
  .post(function(req, res) {res.send('Add a book')});
```

BodyParser

```
import bodyParser from "body-parser";
app.use(bodyParser.json());
Static-Middleware:
● Aufgabe: Statische Files ausliefern
● Nutzen wie folgt:
  app.use(express.static(__dirname + '/public'));
  app.use(express.static(path.join(path.resolve(), 'public')));
● Es sind mehrere static-routes möglich
Custom-Middleware:
Hat 3 Parameter: request, response, next
function myDummyLogger( options ){
  options = options ? options : {};
  return function myInnerDummyLogger(req, res, next) {
    console.log(req.method + ":" + req.url);
    next(); } }
app.use(myDummyLogger());
Error-Middleware:
● Bearbeitet Errors, welche von Middlewares generiert werden
● Hat 4 Parameter: error, request, response & next
● Sollte als letzte Middleware 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!'); });
```

2.2 Model

Ziel: Die Daten sollten in einem Module verwaltet und abgespeichert werden. Möglichkeiten: In Memory (array), JSON, NoSQL-Datenbanken (nedb), Sql-Datenbanken, Oracle-datenbank.

```
// beispiel: nedb
import Datastore from "nedb";
const db = new Datastore({ filename: './data/order.db',
  autoload: true });
// insert
db.insert(order, function(err, newDoc) {
  if(callback) {
    callback(err, newDoc); } });
// search: findOne oder findAll
db.findOne({ _id: id }, function (err, doc) {
  callback(err, doc); });
// update
db.update({_id: id}, {$set: {"state": "DELETED"}}, {},
  function (err, doc) {
    publicGet(id, callback); });
```

2.3 View

Ziel: Trennen von Controller und View mittels **Template Engine**. Express bietet eine render Methode an: app.render(view, [locals], callback);

```
// view engine setup
app.set('views', path.join(path.resolve(), 'views'));
app.set('view engine', 'hbs');
```

2.4 Session & Security

Session:
Beim ersten "Connect" vom Client wird eine Session-Id erstellt und als Cookie zum Client geschickt. Die Session-Daten werden auf dem Server abgespeichert. → Widerspricht REST

Nutzen: HTTP-Stateless umgehen und z.B. Login Status von user abspeichern, oder allgemein Daten Server-Seitig einem Benutzer zuordnen. Ermöglicht tracking.

```
// Cookie verwenden:
app.use(require("cookie-parser")());
// Session benötigt Cookies
app.use(session({ secret: '1234567', resave: false,
  saveUninitialized: true}));
```

2.5 Rest & Ajax

Token:
Ziel: Stateless Server. Idee: Bei jeder Anfrage muss für die **Autorisierung ein Token mitgegeben werden. Vorteil:** Jede Anfrage kann zu einem beliebigem Server gesendet werden. **Nachteile:** Was passiert wenn der Token geklaut wird? → Ablaufdatum kurz setzen, Token invalidieren.
JWT-Token: import jwt from 'express-jwt';

2.6 Web Sockets

Das klassische Model vom Request-Response hat 2 Probleme: Der Server kann keine Nachricht an den Client schicken. Jede Anfrage öffnet eine neue Verbindung.
Dieses Model erschwert es real-time Apps zu machen (Games, Chats). Lösung: WebSockets ermöglichen "bi-directional", "always-on" Kommunikation.

3 Typescript

Variablendeklarationen mit Basistypen:

- Variablen können Typdeklaration erhalten. Ohne Typdeklaration wird die "Type-Inferenz" verwendet.
- Variablen können explizit als **any** deklariert werden. → Beliebige Werte dürfen zugewiesen werden. Entsprechend geht auch die Zuweisung an jede andere Variable.
- Globale Variablen aus nicht TS-Files können mit dem Keyword **declare** deklariert werden.
- Basis-Typen: **boolean, number, string, null, undefined**

// Beispiele

```
let myAnyVar1: any = 1;
let myInferredNumVar1 = 1;
let myNumberVar: number = 1;
let myStringVar: string = 'cdef';
declare let myMagicVar: string;
// allowed
myAnyVar1 = 'hi';
myNumberVar = myAnyVar1; // might come as surprise
// not allowed
myInferredNumVar1 = 'hi';
myStringVar = myInferredNumVar1;
myStringVar = 1;
myNumberVar = 'hi';
```

Variablendeklarationen mit komplexen Typen:

- Typescript erlaubt die Deklaration von **Arrays, Tupels und Enums**

- Bei Tupeln wird keine Type-Inferenz angewendet
- Enums können wie Basistypen genutzt werden
- Default-Repräsentation: Integers (Strings möglich)
- Alternative: String Literal Type

// Beispiele

```
let myInferredNumArray = [1, 2, 3];
let myNotInferredTupel = [1, 'abcd'];
let myNumArray: number[] = [1, 2, 3];
let myTupel: [number, string] = [1, 'abcd']
enum Color {Red, Green, Blue};
enum StrColor {Red = "red", Green = "green"};
type StrLitColor = "red" | "green";
let c: Color = Color.Green;
let myTupel2: [Color, number] = [Color.Green, 1];
// allowed
myNotInferredTupel[0] = 2;
myNotInferredTupel[1] = 2; // not inferred
myTupel[1]='hi';
//not allowed
myInferredNumArray[2] = 'hi';
myInferredNumArray[4] = 'hi';
myTupel[0]='hi';
myTupel[1]= 2;
```

Funktionsdeklarationen:

- Spezifizierbar: Typen der Parameter + Typ des Rückgabewertes
- Mehr als eine erlaubte Signatur pro Funktion möglich !
- Funktionsparameter können optional sein (? direkt nach dem Namen der Variablen)

```
// Beispiele
function add(s1: string, s2: string): string;
function add(n1: number, n2: number): number;
function add(n1, n2) {
  return n1 + n2; }
function combineFunction(sn: number | string = "", ns?: number): string {
  return String(sn) + String(ns || ""); }
// allowed
let myNum: number = add(1, 2);
combineFunction(1);
combineFunction('hi', 3);
// not allowed
let myStr: string = add(1, 2);
let myNum: number = add("kk", 2);
combineFunction(1, 'hi');
```

Funktionen als Parameter:

- Funktionsparameter können Funktionen sein
- Signatur dieser Parameter kann auch deklariert werden

```
// Beispiele
function numberApplicator(numArray: number[], numFun: (prevRes: number, current: number) =>number): number{
  return numArray.reduce(numFun); }
function concatFunction(s1: string, s2: string): string {
  return s1 + s2; }
// allowed
let myNum2: number = numberApplicator([1, 2, 3, 4], add);
// not allowed
numberApplicator([1, 2, 3, 4], concatFunction);
```

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 Counter {
  private _doors: number;
  public static readonly WOOD_FACTORS = {'oak': 80, 'pine': 20};
// public static readonly MIN_DOOR_COUNT = // code
constructor({doors = 2}: {doors?: number} = {}) {
  this.doors = doors; }
set doors(newDoorCount: number) {
  if (newDoorCount >= Counter.MIN_DOOR_COUNT && newDoorCount <= Counter.MAX_DOOR_COUNT) {
    this._doors = newDoorCount; }
  else { throw 'Counter can only have ... ' ; }
}
get doors() {
  return this._doors; } }
```

Interfaces:

- Interfaces (Typen) können in Deklaration von Klassen genutzt werden: Eine Klasse darf mehr als ein Interface implementieren
- Sie können in Deklaration von Variablen und Funktionsparametern genutzt werden: Casting ist möglich und Structural-Typing ("Duck-Typing") wird von TypeScript unterstützt.

```
interface IPoint {
  readonly x: number; readonly y: number; }
interface ILikableItem { likes?: number; }
class DescribableItem {
  constructor(public description: string){} }
class PointOfInterest extends DescribableItem implements IPoint, ILikableItem {
  constructor(public x: number, public y: number, description: string, public likes?:number) {
    super(description); } }
// Allowed
let p: IPoint = new PointOfInterest(1, 2, "home");
let p2: PointOfInterest = p as PointOfInterest;
p2.description = "hi";
let p3: IPoint = {x: 3, y: 4}; // duck-typing
let p4: any = p3;
p4.description = "hi" // any can set anything
let p5: PointOfInterest = p3 as PointOfInterest;
p5.description = "hi";
// Not allowed
p.description; // error: Property description does not exist on type IPoint
```

4 Responsive Design

Flexibles vs Responsives Layout:

Flexible: Dynamisches (größenadaptives) Layout welches sich ohne Media-Queries umsetzen lassen. **Responsive:** Dynamisches Layout welches für unterschiedliche Geräte, Bereiche von Display-Größen und unterschiedliche Medien separates ein Layouts definiert. → Umsetzung mit Media Queries

Graceful Degradation: Baseline of full functionality available in modern browsers and then taking the layers off to ensure it works with older browsers.

Progressive Enhancement: Baseline of the features supported by all browsers and advanced features added like layers.

Mobile first: Base Layout und Design sind für Mobile

4.1 Responsive Web layout

Media Queries:

Typische Trigger Punkte: (besser em verwenden)

- 480px/30em: Smartphones
- 768px/48em: Tablets
- 992px/62em: Desktops

```
// Typen
@media screen { ... }
@media print { ... }
// Dimensionen
@media ({width | min-width | max-width | 375px { ... }
  @media ({height | min-height | max-height | 667px {...}
//Zustände
@media (orientation: landscape) { ... }
// Features
@supports not (display: grid) { div { float: right; } }
// Sonstiges
@media (hover: hover) { ... }
@media (pointer: fine | coarse | none) { ... }
@media (any-pointer: fine | coarse | none) { ... }
```

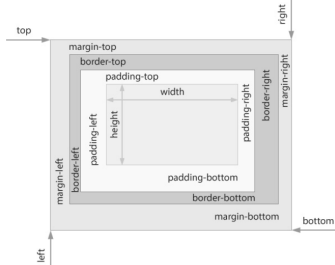
```
@media (min-resolution: 300dpi) { ... }
@media (min-color: 1) { ... }
// Operatoren
@media (min-width: 20em) and (max-width: 30em) { ... }
@media (max-width: 10em), (min-width: 20em) and (
    max-width: 30em), (min-width: 40em) { ... }
@media not screen { ... }
@media not screen and (min-width: 20em) { ... }
@media only screen { ... }
Link referenz mit media Attribut wird nur geladen wenn Bedingung erfüllt:
```

```
<head> ...
<link rel="stylesheet" href="LargeScreenLayout.css"
media="(min-width: 30em)">
```

</head>
View Port:
Mobile Geräte benötigen Viewport Meta-Tag Spezifikation, für die Skalierung:
`<meta name="viewport" content="width=device-width, initial-scale=1">`

4.2 Flexible Layout

Box-Model



Box-Sizing Model gilt nur, wenn display **block** oder **inline-block**
Zentrieren von Elementen:
max-width: 50em;
margin 0 auto;

CSS Funktionen:

```
width: calc(100vw - 5em); // 100vw = 100% der View Width
width: min(500px, 100vw - 5em);
width: max(400px, 100vw - 5em);
width: clamp(400px, 100vw - 5em, 500px); // Links: untere Grenze, Mitte: Bevorzugt, Rechts: obere Grenze
```

Was bedeutet 100%?

Parent's height: height, top
Parent's width: width, left, margin-top, margin-left, padding-top, padding-left
Self's height: translate-top
Self's width: translate-left
Scrolling/Overflow: overflow: visible | hidden | scroll
visible: Text über Box hinaus
hidden: Text abgeschnitten/nur in Box
scroll: Scrollbar (oft erst bei Interaktion)

Werte von Position:

- absolute: Element aus dem Element-Fluss entfernt
 - Eigenschaften: top, left, bottom, right, width, height (Werte relativ zum ersten Parent mit position: relative/absolute)
 - Erlaubt Überlappung von Elementen
- fixed: Immer am gleichem Ort
- sticky: Box bleibt am oberen (oder unteren) Rand des Fensters haften
- static: Standard-Positionierung im Fluss
- relative: Platz im Fluss bleibt reserviert

4.2.1 Flexbox

- display: flex;
- Betrifft alle direkten Kind-Elemente = "Flex-Items"
- Inline Styling ist nicht empfohlen
- Alle Flex-Items verhalten sich wie "inline blocks" → Können height und width definieren

Grösse:

Flex-Items definieren individuell wie sie mit verfügbarem Platz in der "Main-Axis" umgehen
flex-grow: Verhältnis wie der Platz verteilt wird → Default: 0 (nicht grösser werden)
flex-shrink: Verhältnis, wie die Elemente kleiner werden wenn zu wenig platz → Default: 0
Kurzform: flex: [flex-grow][flex-shrink][flex-basis] (default: 0, 0, auto)
→ Wenn alle Flex-Items **flex-grow: 0** und **margin: auto** haben, ist eine Verteilung des leeren Platzes notwendig. Wird auf dem Container definiert: **justify-content: flex-start | flex-end | center | space-between | space-around**

Wrap:

→ Definiert auf Container und wrappt eine Elemente sinnvoll.

flex-wrap: wrap ignoriert flex-shrink Definitionen des flex-Items. Pro Zeile Verteilung entsprechend flex-grow oder justify-content
Order: order: 1

- Flex Elemente werden entsprechend "source order" platziert
- Verwendet, um reihenfolge der Elementa anzupassen
- Kann auch negativ sein (Default: 0)
- Nicht "accessible": reihenfolge für Screen reader ändert nicht

Flex-Direction:
flex-direction: row | row-reverse | column | column-reverse

- Default: row
- Ändert die Haupt-Layoutrichtung
- Bei flex-direction: column braucht Container eine Höhe
- Bei row-reverse und column-reverse kennen CSS Selektoren nur die source order

Höhe, Breite, Ausrichtung:

Default: FlexBox-Items füllen den Platz horizontal (Main Axis) und vertikal (Cross Axis) aus (flex-direction: row)
Main-Axis: Für Items: Attribut **flex**, für Container: Attribut **justify-content**
Cross-Axis: Für Container: **align items: stretch | flex-start | flex-end | center | baseline**, für Items: **align-self**

Summary:

Container Eigenschaften:

- Hauptachse (main axis) wählen:
flex-direction: row | column
- Mehrzeilig erlaubt? (wrap): flex-wrap: nowrap | wrap

- Alignment entlang der Hauptachse wählen (wenn nötig):
justify-content: flex-start | flex-end | center | space-around | space-between
- Alignment entlang der Querachse (cross axis) wählen (wenn nötig):
align-items: stretch | flex-start | flex-end | center

Item Eigenschaften:

- Dynamische Grössenveränderung (Hauptachse):
flex: [flex-grow] [flex-shrink] [flex-basis] (z.B. 0 0 0)
- Alignment des Elements entlang der Querachse (cross axis):
align-self: flex-start | flex-end | center | stretch | auto

4.2.2 CSS Grid

Auf Grid Container: display: grid

Grösse:

fr: Freier Platz wird aufgeteilt, fr Spalten können nicht schmaler als das längste Wort werden.
min-content: Soviel Platz in der Breite wie das längste Wort benötigt. **max-content:** Soviel Platz in der Breite wie der gesamte Text auf einer Zeile benötigt. **Template:**
Definition: grid-template-columns, grid-template-rows, grid-template-areas
Platzierung: grid-[column][row]-[start][end]: number
Kurzform: grid-column: 1/5, grid-row: 1/2 (start/end)
Alle 4: grid-area: Y1/X1/Y2/X2;

// Beispiele:

```
grid-template-columns: auto 7em 1fr minmax(2em, 20em);
grid-row-start: 1; grid-row-end: 2;
grid-area: 1/1/2/5
```

Alignment:

Y-Achse

Default: align-items: stretch; (Items nehmen die ganze Höhe der Zeile an.)

Alternativen:

- Oben: align-items: start;
- Mitte: align-items: center;
- Unten: align-items: end;
- Anpassung per Item statt auf Container: align-self

X-Achse:

default: justify-items: stretch; (Items nehmen die ganze Breite der Zelle ein.)

Alternativen:

- Links: justify-items: start;
- Mitte: justify-items: center;
- Rechts: justify-items: end;
- Anpassung per Item statt auf Container: justify-self

5 Testing

Unit Tests: Testen einzelner Units (Klassen, Module). Automation relativ einfach. **Herausforderung:** Isolation der Units, asynchrone Operationen, Testdatengenerierung
Integrationstests: Testet das Zusammenspiel von 2 oder mehr Units. Automation meist möglich. **Herausforderungen:** Isolation der Units, asynchrone Operationen, Simulation Browser & Benutzerinteraktion, Test mit Datenbank, Testdatengenerierung

Funktionale tests: Testen ob sich das System nach den Anforderungen (Use-Cases, User Stories) verhält.
(Visuelle) Regressionstests: Testen, ob Veränderungen im Code zu Änderungen im Verhalten führen. Für beide: Automation möglich mit speziellen Tools
Funktionale Systemtests: Testet das Zusammenspiel aller Systemkomponenten in der Zielumgebung. Automation nur in Teilen möglich. **Herausforderungen:** Realistische aber vorher-sagbare Umgebung
Weitere Systemtests: Load (Stress), Performance, Endurance Test, Chaos Testing, Security Tests, Usability Tests

5.1 Tools

Test-Runner: Nimmt Tests entgegen, führt diese aus und zeigt die resultate an: Ava CLI, jasmine, **Mocha**
Assertion Library: Code zur Ausführung einzelner Tests: Assert, Ava Power-Assert, Expect.js, should.js, **Chai**
Mocking Library: Separierung von Units/Erstellung von Mocks etc: **Proxyquire**, **Sinon.js**
DOM Handling: Puppeteer, Storybook, Enzyme

// Mocha API

```
describe("Array", function() {
  describe("#indexOf()", function() {
    beforeEach(function() {
      this.testArray = [1, 2, 3];
    });
    it("should return -1 when the value is not present", function() {
      const foundIndex = this.testArray.indexOf(4);
      if (!(foundIndex === -1)) {
        throw new Error("Expected to receive -1 (not found)" + "but received the following" + foundIndex);
      }
    });
  });
});
```

6 Security

XSS (Cross-Site Scripting):

Definition: Den Server so manipulieren, dass Schadcode (JavaScript) an Nutzer (Opfer) ausgeliefert wird und im Browser dieser Nutzer ausgeführt wird.

Gegenmassnahmen:

- XSS oder DOMPurify Library
- "Encoding" bei der Darstellung (Output) von Nutzer-Input
- Content Security Policy (CSP) im Header setzen
- Bei Cookies soweit möglich das HTTPOnly Flag setzen

Code-Injection/Remote Code Execution:

Definition: Den Server dazu bringen, dass eingeschleuster Code ausgeführt wird.

Gegenmassnahmen:

- NICHT eval(), sondern parseInt(), JSON.parse() nutzen
- Globale Scopes und Variablen reduzieren.
- Rechenintensive Tasks mit childprocess.spawn auslagern, um (D)DOS Attacken zu erschweren
- Node NICHT als root Prozess starten

Stored Broken Authentication:

Ziel: Bereiche eines Web-Sites mit benutzer-spezifischen Informationen sollten nur für authentifizierte Nutzer zugreifbar sein.

Gegenmassnahmen:

- Keine geheimen Informationen in query-parametern
- https (TLS) nutzen
- Authentication-Service nutzen
- Session Timeout sinnvoll setzen
- Formulare beim Ausliefern mit einem Token versehen

Stored Broken Access Control: Ziel: Bereiche eines Web-Sites mit benutzer-spezifischen Informationen sollten nur für autorisierte Nutzer zugreifbar sein (read)

Gegenmassnahmen:

- Sicher stellen, dass der eingeloggte Nutzer berechtigt ist.
- Token mit einmaliger Gültigkeit

7 Accessibility

8 Animation

9 UX Research, Information Architecture

10 Internationalization

11 Dev-Ops