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-Middleware:

```
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. → Wiederspricht 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 Authorisierung 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
myInferedNumVar1 = 'hi';
myStringVar = myInferedNumVar1;
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 privateünd 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 dies not
     exist on type IPoint
```

Custom Types:

Non-Nullable Type Check:

4 Responsive Design

5 Testing

6 Security

7 Accessibility

8 Animation

9 UX Research, Information Architecture

10 Internationalization

11 Dev-Ops