Web Frameworks - Node, Express & Nest.js

- Werken met de MEAN stack
- De A kennen jullie al
- M=mongo, E=express, N=node
- = de serverside
- Beauty = allemaal gebaseerd op JS technologie
- Tussen de software lagen wordt JSON data uitgewisseld
- Angular <=> Node / Express <=> mongoDB

Web server standaard => webpack web server / kan ook gehost worden

Node.JS

Node.js is een open-source en cross-platform JavaScript-runtime-omgeving. Het laat je toe om JavaScript te gebruiken voor server-side scripting—dit betekent dat je scripts kunt schrijven die op de server draaien en dynamische webpagina's genereren voordat ze naar de webbrowser van de gebruiker worden verzonden. In tegenstelling tot traditionele JavaScript die in browsers draait en voornamelijk gebruikt wordt voor client-side scripting (interacties aan de voorkant van een website), stelt Node.js je in staat om volledige webapplicaties te bouwen in JavaScript. Dit maakt het een krachtig hulpmiddel in de MEAN-stack, waarbij het samenwerkt met MongoDB, Express.js en AngularJS. Node.js is bijzonder efficiënt voor het bouwen van snelle en schaalbare netwerkapplicaties.

```
mkdir NodeDemo
npm init
code .
touch main.js
```

Handige tool: Nodemon

```
npm install -g nodemon
nodemon --inspect
```

Ga naar debug rubriek - Launch.json - node.js attach - "restart": true - hiermee kan je vanuit vscode debuggen en tegelijk code aanpassen.

Je kan dependencies toevoegen via package.js

Node maakt gebruik van CommonJS syntax.

Gebruik module.exports om naar andere modules te exporteren.

Gebruik require("") om een andere module in te laden.

Eenvoudige webserver met Node.js

https://www.digitalocean.com/community/tutorials/how-to-create-a-web-server-in-node-js-with-the-http-module

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  if (req.url === '/json') {
    // JSON-route
    res.statusCode = 200;
    res.setHeader('Content-Type', 'application/json');
    res.end(JSON.stringify({ message: 'Dit is een JSON bericht' }));
  } else if (req.url === '/html') {
    // HTML-route
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html');
    res.end('<html><body><h1>Dit is een HTML pagina</h1></body></html>');
  } else {
    // Standaard route (platte tekst)
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Hallo Wereld\n');
  }
});

server.listen(port, hostname, () => {
  console.log(`Server draait op http://${hostname}:${port}/`);
});
```

De callback-functie definieert hoe de server moet reageren op inkomende verzoeken.

Routing zorgt ervoor dat voor elke url de juiste resource zal worden teruggestuurd naar de client.

Het kan dus met node.js maar het is iets eenvoudiger mbv de express npm module.

Express.js

Express.js is een snel, onopvallend en minimalistisch webframework voor Node.js. Het is ontworpen om webapplicaties en API's gemakkelijker te bouwen met een reeks handige functies die niet standaard aanwezig zijn in Node.js. Express.js stroomlijnt het proces van het afhandelen van verzoeken en reacties, en het biedt handige middelen (middleware) voor het routeren, het beheren van sessies, het verwerken van fouten, en nog veel meer.

Het grootste voordeel van Express.js is dat het ontwikkelaars toelaat om flexibel en snel serverzijde logica te schrijven zonder de vrijheid van Node.js op te geven. Het wordt vaak gebruikt voor het bouwen van webapplicaties, RESTful API's en als onderdeel van de MEAN/MERN stack (MongoDB, Express.js, Angular/React, Node.js). Met zijn minimalistische aanpak kunnen ontwikkelaars de functionaliteiten die ze nodig hebben toevoegen door middel van middleware, waardoor de applicatie licht en efficiënt blijft.

```
const express = require('express');
const app = express();
const port = 3000;

// Route voor platte tekst
app.get('/', (req, res) => {
  res.send('Hallo Wereld');
});

// Route voor JSON
app.get('/json', (req, res) => {
  res.json({ message: 'Dit is een JSON bericht' });
});

// Route voor HTML
app.get('/html', (req, res) => {
  res.send('<html><body><h1>Dit is een HTML pagina</h1></body></html>');
});

// De server starten
app.listen(port, () => {
  console.log(`Server draait op http://localhost:${port}`);
});
```

- https://expressjs.com/
- npm install express
- Huidige versie: 4.18.1

Web API met Express.js

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware om JSON request bodies te verwerken
app.use(express.json());

// Start dataset
let mensen = [
  { id: 1, naam: 'Alice', leeftijd: 30 },
  { id: 2, naam: 'Bob', leeftijd: 25 },
  { id: 3, naam: 'Carol', leeftijd: 35 },
  { id: 4, naam: 'Dave', leeftijd: 40 },
  { id: 5, naam: 'Eve', leeftijd: 28 }
];

// CRUD Operaties

// Create (POST)
app.post('/mensen', (req, res) => {
```

```
  const nieuweMens = {
    id: mensen.length + 1,
    naam: req.body.naam,
    leeftijd: req.body.leeftijd
  };
  mensen.push(nieuweMens);
  res.status(201).send(nieuweMens);
});

// Read (GET)
app.get('/mensen', (req, res) => {
  res.status(200).send(mensen);
});

app.get('/mensen/:id', (req, res) => {
  const mens = mensen.find(m => m.id === parseInt(req.params.id));
  if (!mens) res.status(404).send('Mens niet gevonden');
  res.send(mens);
});

// Update (PUT)
app.put('/mensen/:id', (req, res) => {
  const mens = mensen.find(m => m.id === parseInt(req.params.id));
  if (!mens) {
    res.status(404).send('Mens niet gevonden');
    return;
  }
  mens.naam = req.body.naam;
  mens.leeftijd = req.body.leeftijd;
  res.send(mens);
});

// Delete (DELETE)
app.delete('/mensen/:id', (req, res) => {
  const index = mensen.findIndex(m => m.id === parseInt(req.params.id));
  if (index === -1) {
    res.status(404).send('Mens niet gevonden');
    return;
  }
  mensen.splice(index, 1);
  res.status(204).send();
});

// Server starten
app.listen(port, () => {
  console.log(`Server draait op http://localhost:${port}`);
});
```

We kunnen onze API testen met Postman => https://www.postman.com/

Gebruik van Express Router

Om het voorbeeld te herschrijven met behulp van Express Router, zullen we een router-module aanmaken voor de "mensen" resource. Dit maakt de code beter gestructureerd en gemakkelijker te onderhouden, vooral als de applicatie groeit. De Express Router stelt ons in staat om route-handlers in een apart bestand te scheiden, wat de code netter en overzichtelijker maakt.

App.js:

```
const express = require('express');
const app = express();
const port = 3000;

const mensenRouter = require('./routes/mensen');

app.use(express.json());
app.use('/mensen', mensenRouter);

app.listen(port, () => {
  console.log(`Server draait op http://localhost:${port}`);
});
```

/routes/mensen.js:

```
const express = require('express');
const router = express.Router();

// Start dataset
let mensen = [
  { id: 1, naam: 'Alice', leeftijd: 30 },
  { id: 2, naam: 'Bob', leeftijd: 25 },
  { id: 3, naam: 'Carol', leeftijd: 35 },
  { id: 4, naam: 'Dave', leeftijd: 40 },
  { id: 5, naam: 'Eve', leeftijd: 28 }
];

// CRUD Operaties

// Create (POST)
router.post('/', (req, res) => {
  const nieuweMens = {
    id: mensen.length + 1,
    naam: req.body.naam,
    leeftijd: req.body.leeftijd
  };
  mensen.push(nieuweMens);
  res.status(201).send(nieuweMens);
});

// Read (GET)
router.get('/', (req, res) => {
  res.status(200).send(mensen);
```

```
});

router.get('/:id', (req, res) => {
  const mens = mensen.find(m => m.id === parseInt(req.params.id));
  if (!mens) res.status(404).send('Mens niet gevonden');
  res.send(mens);
});

// Update (PUT)
router.put('/:id', (req, res) => {
  const mens = mensen.find(m => m.id === parseInt(req.params.id));
  if (!mens) {
    res.status(404).send('Mens niet gevonden');
    return;
  }
  mens.naam = req.body.naam;
  mens.leeftijd = req.body.leeftijd;
  res.send(mens);
});

// Delete (DELETE)
router.delete('/:id', (req, res) => {
  const index = mensen.findIndex(m => m.id === parseInt(req.params.id));
  if (index === -1) {
    res.status(404).send('Mens niet gevonden');
    return;
  }
  mensen.splice(index, 1);
  res.status(204).send();
});

module.exports = router;
```

Ophalen van objecten met URL query string

In Express kunnen deze parameters worden opgevraagd via de query property van het request object. We krijgen dan een object met alle ingestelde parameters als property. Hiermee kan dan vervolgens gefilterd worden in de data.

```
router.get('/zoeken', (req, res) => {
  let gefilterdeMensen = mensen;

  if (req.query.naam) {
    gefilterdeMensen = gefilterdeMensen.filter(mens =>
mens.naam.toLowerCase().includes(req.query.naam.toLowerCase()));
  }

  if (req.query.leeftijd) {
    gefilterdeMensen = gefilterdeMensen.filter(mens => mens.leeftijd ===
parseInt(req.query.leeftijd));
  }
```

```
  res.send(gefilterdeMensen);
});
```

In deze route:

- We gebruiken `req.query` om toegang te krijgen tot de query parameters.
- Als er een `naam` query parameter is, filteren we de `mensen`-lijst op die naam. We gebruiken `toLowerCase()` om het zoeken hoofdletterongevoelig te maken.
- Als er een `leeftijd` query parameter is, filteren we de lijst verder op die leeftijd. We zetten de query parameter om naar een getal met `parseInt()`.
- Ten slotte sturen we de gefilterde lijst terug als de response.

Met deze route kun je verzoeken sturen zoals `GET /mensen/zoeken?naam=Alice&leeftijd=30` om mensen te vinden die aan beide criteria voldoen.

POST/PUT met bodyparser npm package

! Kristof: doe ik met app.use(express.json());

In moderne versies van Express (versie 4.16 en later), is het gebruik van de aparte `body-parser` npm package niet echt noodzakelijk. Dit komt omdat Express sinds versie 4.16 de belangrijkste functionaliteit van `body-parser` als ingebouwde middlewares heeft. De methodes `express.json()` en `express.urlencoded()` zijn nu ingebouwd in Express en bieden dezelfde functionaliteit als `body-parser`.

Daarom, voor de meeste standaardtoepassingen, is het gebruik van `express.json()` voor het parsen van JSON request bodies en `express.urlencoded()` voor het parsen van URL-encoded data (handig voor het verwerken van gegevens afkomstig uit HTML-formulieren) voldoende en aanbevolen.

Het gebruik van `body-parser` zou alleen overwogen moeten worden in specifieke situaties waarbij geavanceerde configuratie nodig is die de ingebouwde middlewares van Express niet bieden. Echter, voor de meeste standaard webapplicaties en API's, is het gebruik van de ingebouwde middlewares van Express de voorkeursaanpak en helpt het om de afhankelijkheid van externe packages te verminderen. Dit maakt de applicatie eenvoudiger en vermindert de kans op compatibiliteitsproblemen of beveiligingsissues.

Nest.js

https://docs.nestjs.com/

Nest.js is een modern, schaalbaar server-side JavaScript framework gebouwd op Node.js. Het gebruikt Express.js onder de motorkap, maar introduceert een extra laag van abstractie en structuur om meer georganiseerde en onderhoudbare applicaties te bouwen. Hier zijn enkele kernpunten die Nest.js onderscheiden van het gebruik van enkel Express.js:

1. **TypeScript Support**: Nest.js is gebouwd met TypeScript, wat sterke typisering en object-georiënteerde programmeerprincipes introduceert. Dit helpt bij het schrijven van meer robuuste en foutbestendige code.
2.

3. **Modulaire Structuur**: Nest.js maakt gebruik van modules om de applicatie te organiseren, wat de scheiding van zorgen bevordert en de codebasis overzichtelijker maakt. Dit maakt het gemakkelijker om grote en complexe applicaties te beheren.

4. **Dependency Injection**: Het framework maakt gebruik van een krachtig dependency injection-systeem, vergelijkbaar met dat in Angular. Dit maakt het gemakkelijker om code te testen en te onderhouden.

5. **Ingebouwde Applicatiearchitectuur**: Nest.js biedt een out-of-the-box applicatiearchitectuur die ontwikkelaars begeleidt naar het schrijven van schaalbare, testbare en onderhoudbare code. Dit omvat ingebouwde ondersteuning voor verschillende lagen zoals controllers, services en modules.

6. **Ondersteuning voor Websockets en Microservices**: Nest.js biedt geïntegreerde ondersteuning voor WebSockets en microservice-architecturen, waardoor het bouwen van realtime data-applicaties en gedecentraliseerde systemen eenvoudiger wordt.

7. **Rich Ecosystem**: Het framework biedt een rijk ecosysteem met veel extra modules en pakketten die functies toevoegen zoals ORM-integratie (bijv. TypeORM, Mongoose), security (bijv. Passport), en taakplanning.

8. **Decorator-Based Routing**: Nest.js gebruikt decorators voor routing, wat zorgt voor een declaratieve en intuïtieve manier om routes en hun handlers te definiëren.

Samengevat, hoewel Express.js flexibel en minimalistisch is, voegt Nest.js structuur en abstractie toe, waardoor het bijzonder geschikt is voor grote, enterprise-level applicaties of projecten waar een gestructureerde aanpak gewenst is. Het helpt teams om consistente, schaalbare en onderhoudbare code te schrijven.

Mensen-api herschrijven

Het herschrijven van de Express 'mensen'-API met behulp van Nest.js vereist een fundamenteel andere benadering, omdat Nest.js een framework is dat sterk leunt op TypeScript en het gebruik van decorators voor routing en dependency injection. Nest.js is gebaseerd op Express onder de motorkap, maar voegt een extra laag van structuur en abstractie toe om meer schaalbare en onderhoudbare applicaties te bouwen.

Eerst moet je Nest.js en TypeScript in je project installeren:

- npm i -g @nestjs/cli
- nest new nest-voorbeeld

In een Nest.js-project structuur je code in modules, controllers en services. Voor de 'mensen'-API zou je een `MensenModule`, een `MensenController` en een `MensenService` creëren.

app.module.ts:

```
import { Module } from '@nestjs/common';
import { MensenModule } from './mensen/mensen.module';

@Module({
    imports: [MensenModule],
    controllers: [],
```

```
    providers: [],
})

export class AppModule {}
```

In directory mensen:

mensen.module.ts:

```
import { Module } from '@nestjs/common';
import { MensenController } from './mensen.controller';
import { MensenService } from './mensen.service';

@Module({
    controllers: [MensenController],
    providers: [MensenService],
})

export class MensenModule {}
```

mensen.controller.ts:

```
import { Controller, Get, Param } from '@nestjs/common';
import { MensenService } from './mensen.service';
import { Mens } from './mens.entity';

@Controller('mensen')
export class MensenController {

constructor(private readonly mensenService: MensenService) {}

@Get()
findAll(): Mens[] {
    return this.mensenService.findAll();
}

@Get(':id')
findOne(@Param('id') id: string): Mens {
    const numericId = parseInt(id, 10);
    return this.mensenService.findOne(numericId);
}

}
```

mensen.service.ts:

```
import { Injectable } from '@nestjs/common';
import { Mens } from './mens.entity';

@Injectable()
export class MensenService {

private mensen: Mens[] = [
{ id: 1, naam: 'Alice', leeftijd: 30 },
{ id: 2, naam: 'Bob', leeftijd: 25 },
{ id: 3, naam: 'Carol', leeftijd: 35 },
{ id: 4, naam: 'Dave', leeftijd: 40 },
{ id: 5, naam: 'Eve', leeftijd: 28 },
];

findAll(): Mens[] {
return this.mensen;
}

findOne(id: number): Mens {
return this.mensen.find((mens) => mens.id === id);

}

}
```

mens.entity.ts:

```
export class Mens {

id: number;

naam: string;

leeftijd: number;

}
```

npm run start om nest toepassing te starten!