Versie 1.0.0 2023-12-20

Spring

Spring Initializr

- https://start.spring.io/
- Maven vs Gradle Groovy vs Gradle Kotlin
- artifact vs name
- JPA, Web, PostgreSQL, Lombok

Spring-project maken met database

- in main/resources zit een application.properties
- die moet er ongeveer uitzien als:

```
spring.datasource.url=jdbc:postgresql://localhost:5432/books
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.jpa.show-sql=true

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```

- als je commit naar Git zou ik aanraden de gebruikersnaam en wachtwoord e.d. op te slaan als environment-variabele/systeemvariabele, bv PG_BOOKS, PG_USERNAME, PG_PASSWORD. application.properties wordt dan

```
spring.datasource.url=${PG_BOOKS}
spring.datasource.username=${PG_USERNAME}
spring.datasource.password=${PG_PASSWORD}
spring.jpa.show-sql=true

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto = update
```

Het seeden van een database

- Vaak wil je data in je database hebben, om je programma te kunnen uittesten.
- Die *kan* je erin zetten met SQL-scripts door een file data.sql in de resources te zetten.
- Het kan echter ook met Java-code, bijvoorbeeld met een seeder-klasse:

```
@Component
@RequiredArgsConstructor
public class Seeder implements CommandLineRunner {

    private final ApplicationRepository applicationRepository;

    @Override
    public void run(String... args) throws Exception {
        if (applicationRepository.count() == 0) {
            applicationRepository.save(new Application("game", "secret"));
        }

    }
}
```

Basic controller commands

- @RestController
- @RequestMapping
- api/v1/movies - patroon
- @GetMapping

HTTP Basics

- requests vs responses
- start line - headers - empty line - body
- verbs: GET PUT POST DELETE PATCH (en OPTIONS, later voor CORS)
- status code: 200s (success) 300s (redirects) 400s (client error) 500s (server error)
- HTTP verb in start line request
- status code in start line response

Basaal JPA-gebruik

- @Entity
- @Id
- @GeneratedValue
- id type Long or UUID
- interface XRepository extends JpaRepository <X, TypeOfId> {}

Basaal Lombok-gebruik

- @DefaultConstructor // nodig voor @Entity
- @RequiredArgsConstructor // handig voor injectie in Controllers en Services
- @Getter
- @Setter
- NOOT: vermijd @Data

Basic Dependency Injection

- normaal geeft een “stereotype annotatie” als @Component (of @Controller, @RestController, @Service) aan dat de klasse geinjecteerd kan worden.
- echter: @Component enzo is zinloos bij een interface. Interfaces die bepaalde interfaces extenden (zoals JpaRepository en CrudRepository) worden door Spring/JPA automatisch geinjecteerd (of beter gezegd, door Spring gemaakte klassen die die interface annoteren)
- Je kunt alleen dependencies injecten IN klassen die een ‘stereotype annotation’ hebben (@Component/@Repository/@Service/@Controller/@RestController)
- Er zijn drie soorten Dependency Injection mogelijk in Spring: setter injection (bijna nooit gebruikt), @Autowired boven fields (gebruikt in demos), en injectie via de constructor. Die laatste staat final fields toe en geeft meer flexibiliteit voor bv unittests, en krijgt in professionele omgevingen voor zover ik begrijp meestal de voorkeur.
- Als er maar één constructor is, wordt die door Spring automatisch @AutoWired.

Basic code-structuur

- altijd nodig: entity, repository, controller.
- discussiepunt: horizontale versus vertikale packages. Sommige groepen gebruiken packages als ‘controllers’, ‘repositories’, ‘models’ en ‘services’ (horizontaal: alles in dezelfde logische laag zit in hetzelfde package). Andere groepen gebruiken packages voor bv ‘movies’, ‘actors’, ‘reviews’, in elk package zit de entity, de repository en de controller. Dat worden ook wel ‘vertikale packages’ genoemd. Omdat je in de praktijk vaak bezig bent heen en weer te springen tussen de controller en de repository en data model ervan, hebben vertikale packages mijn persoonlijke voorkeur, maar het is vermoedelijk geen grote handicap om horizontale packages te gebruiken.
- discussiepunt: het gebruik van services. In principe kan een controller rechtstreeks een repository geinjecteerd krijgen en data naar de database schrijven en inlezen. Maar in sommige gevallen krijg je dan allerlei gedupliceerde code of hulpmethoden in de controller, of wordt de controller erg groot en ingewikkeld. Voor dat probleem worden ook wel services gebruikt: de controller roept een service aan, die op zijn beurt de repository aanroept. De discussie is in dit geval niet of services nuttig kunnen zijn, maar of je *altijd* een service moet gebruiken ook als het de code groter, ingewikkelder en moeilijker veranderbaar maakt. Sommige programmeurs vinden de voordelen van een uniforme structuur opwegen tegen de nadelen. Ik ben echter iemand die ooit moest werken aan een project waar de uniforme structuur voldoende lagen had voor zelfs de meest ingewikkelde usecase, dus ik ben zelf minder enthousiast. Ook omdat “code is not an asset, it’s a liability”. Mogelijk dat je in een klein project en beginfasen beter services beter kan vermijden tenzij een controller echt te complex wordt of omdat je fouten maakt omdat instantiatie van een object ofzo relatief complex is. Maar als je overal een service voor wil maken is dat zeker niet objectief fout, al kan het je ontwikkel- en debugsnelheid verminderen.

Endpoints:

- basisstructuur: /api/v1/items
  - API: geeft aan dat dit endpoint geen mooie webpagina’s (HTML) voor menselijke consumptie geeft, maar door computers te lezen data in JSON of andere formaten
  - v1: als je ooit een nieuwe versie van de API maakt kan je dat v2 noemen, en programma’s die v1 gebruiken zullen niet gelijk gaan falen (dus geen boze collega’s/klanten)
  - companies: normaal geef je een endpoint in het meervoud. Conventies zijn: /companies: geeft alle companies, /companies/4 geeft de company met id 4, POST /companies voegt een bedrijf toe, DELETE companies/4 verwijdert bedrijf 4, PATCH of PUT /companies/4 update de informatie van company 4.
- De endpoints in een @RestController geven per default een 200 OK-code terug. Maar vaak wil je dat veranderen:
  - GET (all) geeft altijd 200 terug, zelfs al kan het request body een lege lijst bevatten
  - GET (by id)/DELETE/PUT/PATCH geven een 404 (not found) terug als het gezochte item niet gevonden is.
  - Je kunt ervoor kiezen een PUT/PATCH/POST een 400 (Bad Request) terug te laten geven als er data ontbreekt, inconsistent of ongeldig is.
  - Een succesvolle PUT of DELETE kan een No Content (204) teruggeven
  - Een succesvolle PATCH kan je (net als GET) een 200 laten teruggeven met het nieuwe item in de body.
  - Een successvolle POST hoort officieel een Location-header te hebben, die kan je maken via

```
@PostMapping
private ResponseEntity<Void> createMovie(@RequestBody Movie newMovie, UriComponentsBuilder ucb) {
   Movie savedMovie = cashCardRepository.save(newMovie);
   URI locationOfNewMovie = ucb
            .path("movies/{id}")
            .buildAndExpand(savedMovie.id())
            .toUri();
   return ResponseEntity.created(locationOfNewMovie).body(savedMovie);
}
```

(het is meestal ook handig om het nieuwe object terug te geven, vaak heeft de code die het POST-request doet tenminste de id van het nieuwe object nodig, soms ook andere data die is ingevoegd door de backend)

- Andere statuscodes dan 200 regel je door de methode een ResponseEntity te laten returnen. “No Content” kan via `ResponseEntity<Void>`, een item teruggeven kan via `ResponseEntity<Movie>` en dergelijke.

- In de methode zelf kan je de ResponsEntity maken via ResponseEntity.noContent()/.created()/.body(item)/.notFound(). Soms heb je .build() nodig om de ResponseEntity af te maken.

- waarden uit het HTTP-request krijgen:

    - Uit het pad: `@GetMapping("{id}") public ResponseEntity<Item> getById(@PathVariable long id) {...`
    - uit de body body: @PostMapping public ResponseEntity<Item> post(@RequestBody Item item) { …
    - van de params (`api/v1/movies?name-includes="up"&release-date-after=2000`):

```
@GetMapping
Iterable<Movie> getAll(@RequestParam(required=false, name="name-includes" String nameIncludes, @RequestParam(required=false, n
```

DTOs

- vaak wil je je databaseobjecten niet letterlijk naar de frontend sturen:
    - sommige informatie is overbodig en wil je niet meesturen om dataverbruik van je server te besparen
    - sommige informatie is mogelijk vertrouwelijk
    - sommige informatie is in een formaat dat handig is voor de database, maar niet voor de frontend.
    - Vaak wil je niet dat alle clients breken als je de structuur van het object in de database verandert.
    - Soms heb je problemen dat een object niet letterlijk kan worden verstuurd omdat het ‘recursieve’ relaties bevat, een bedrijf kan een lijst werknemers hebben, en elke werknemer kan naar zijn of haar werkgever (dat bedrijf) wijzen. Als de controller een bedrijf naar de front-end wil sturen, stuurt het eerst de bedrijfsnaam, dan een werknemer, dan het bedrijf waar die werknemer werkt, en dan dus weer de bedrijfsnaam, dan dezelfde werknemer…
- Een aantal van deze problemen kunnen in eenvoudige gevallen worden opgelost met annotaties in de Entityklasse, zoals @JsonIgnore (stuur dit niet naar de frontend, en als de frontend een waarde stuurt, let daar dan niet op en zet hem op 0), @JsonBackReference(stuur dit niet naar de frontend, maar als de frontend een waarde stuurt, lees die dan wel in!), en @JsonManagedReference(wordt normaal gecombineerd met een @JsonBackReference aan de andere kant. Bijvoorbeeld een Movie zal een @JsonManagedReference hebben naar een `Collection<Role>`, maar de Role zal een @JsonBackReference hebben naar Movie)
- Maar flexibeler is een DTO, een Data Transfer Object. Normaal een klasse (of, in modern Java, een record) waarmee je kunt aangeven wat er precies naar de frontend gestuurd wordt. Bijvoorbeeld dat een review, die naar een User wijst, niet het hele User-object zal bevatten, maar alleen de name, die dan kan worden weergegeven met bijvoorbeeld “username”. Voorbeeld:

```
public record ReviewDto(String username, int rating, String text) {}
```

Paginatie en sorteren

- meestal wil je niet letterlijk een ‘getAll’-endpoint maken; duizenden items per keer over het netwerk sturen is duur, maakt je applicatie traag, en kan het geheugen van de client/browser overbelasten.
- Net zoals bij Amazon of Google krijg je normaal slechts 10, 20 tot maximaal 50 resultaten te zien; als je meer wilt, vraag je een de volgende pagina op.
- Normaal werkt dat doordat de URL parameters krijgt als ?page=1&size=20, wat dan zegt dat je de *tweede* pagina opvraagt, aannemende dat de resultaten in pagina’s van 20 zijn opgedeeld.
- Vaak kan je ook nog een sort-parameter toevoegen, bijvoorbeeld page=1&size=20&sort=name,desc
- Spring’s JpaRepository (NIET de CrudRepository) heeft ingebouwde methoden om pagina’s terug te geven. Code als

```
@GetMapping
public Iterable<Book> getAll(Pageable pageable) { // Pageable van import org.springframework.data.domain.Pageable;
    return bookRepository.findAll(pageable); // voorbeeld: http://localhost:8080/api/v1/books?page=0&size=10&sort=title,de
}
```

werkt gewoon. page en size krijgen default waarden als je ze niet invult (page=0, size=20)

- Vaak wil je meer controle, bijvoorbeeld dat de defaultsortering bijvoorbeeld op datum of op prijs of op naam is, en dat niemand zomaar 5000 items in 1x kan opvragen. Dat doe je door het Pageable-object te vertalen naar een ander PageRequest:

```
@GetMapping
public Iterable<Book> getAll(Pageable pageable) {
    return bookRepository.findAll(
            PageRequest.of(
```

```
                    pageable.getPageNumber(),
                    Math.min(pageable.getPageSize(), 3),
                    pageable.getSortOr(Sort.by("title"))));
}
```

@Transactional

- Als je naar een database schrijft, zeker bij een complexe operatie die meerdere stappen heeft, is de @Transactional-annotatie handig
- @Transactional boven een methode zetten betekent dat als de methode een unchecked exceptie gooit, dat de databasetransactie zal worden teruggerold tot de begintoestand.
- Let er dus op dat als je een *checked* exceptie gooit, Transactional niets zal terugrollen.
- Transactional zal ook niet werken als je in de methode zelf de unchecked exceptie opvangt.
- Vaak zetten mensen @Transactional boven een klasse, dat voegt automatisch @Transactional boven alle methoden toe.

@CrossOrigin

- als je in een browser een front-end-app hebt, die een verzoek doet aan een backend, kan de browser zeggen dat er een CORS-error is omdat een preflight-request werd geweigerd door de backend.
- Wat de browser doet is een HTTP OPTIONS-request aan de backend sturen (dat is de ‘preflight’ request) met iets als: de applicatie op poort 5173 wil data van je hebben. Mag dat? Als de backend applicatie ‘nee’ antwoordt of een verkeerd antwoord geeft, rapporteert de browser die CORS error.
- Je kunt dit probleem voorkomen door boven je @RestController-klasse een @CrossOrigin-annotatie te zetten, liefst met het toegestane pad voor client-applicaties, of toegestane paden. Iets als bijvoorbeeld @CrossOrigin("http://localhost:5173")
- Let wel dat je NIET in de SecurityFilterConfiguration (als je die hebt) CORS disablet, want dan geeft Spring geen net antwoord meer aan de browser en krijg je alsnog CORS-errors.

Diversen

- jar vs war-files: JAR is Java-Archive, WAR is Web-Archive, dat waren files die vooral vroeger werden gemaakt om ergens op een application server te draaien. Tegenwoordig hebben de meeste Spring-applicaties een ingebouwde server (dus bv Tomcat), dus WAR wordt tegenwoordig veel minder gebruikt.
- CrudRepository vs JpaRepository: Een JpaRepository extends het CrudRepository, je kunt dus JpaRepository overal gebruiken waar je een CrudRepository hebt. Maar programmeurs vinden dat vaak niet netjes, die willen vaak het ‘minimale contract’. Basaal: als je een entity wil/moet pagineren, gebruik dan JpaRepository. Als dat (nog) niet hoeft, gebruik CrudRepository.

BONUS

- @ComponentScan
- @Bean
- @Configuration

JavaScript

Interactie met de gebruiker

- console.log
- console.table
- confirm (boolean)
- alert
- prompt (string)

Hoe je waarden toekent

- a = 5
- ken const en let (en vermijd var)

Datatypen (eenvoudig)

- string (JavaScript heeft GEEN char)
- number (soort double, er zijn geen integers in JS)
- boolean

Speciale waarden

- undefined
- null

Operatoren:

- alle Java-operatoren zitten in JavaScript: (+ - / %, &&, || ! , ++, --, +=, -=, %= etc.)
- booleaanse operatoren betekenen vaak net iets anders:
  - ze werken op ALLE waarden (niet alleen op booleans)
  - omdat elke waarde in JavaScript 'truthy' of 'falsy' is
  - 0, "", NaN, undefined, null, false zijn allemaal falsy. De rest van de mogelijke waarden is dus truthy.
  - ! zet een truthy/falsy waarde om in false resp. true
  - && returnt de eerste falsy waarde en anders de laatste waarde (en geeft dus voor booleans precies hetzelfde resultaat als Java doet)
  - || returnt de eerste truthy waarde en anders de laatste waarde (en geeft dus voor booleans precies hetzelfde resultaat als Java doet)
- extra operator: ** voor machtsverheffen (12 ** 2 // 144)
- let wel dat != en == in JavaScript typeconversie doen. Bv `0== ""` is true. Gebruik dus liever === en !== ipv == en !=.

Aanhalingstekens

'' vs "": mag allebei (er is toch geen verschil tussen String en char)

Puntkomma

; is vaak onnodig (maar wel voor de zekerheid)

Datatypen (complex)

- arrays (a=[1,2,'hallo'])
- objects (person = {name: "Wim", grade: 5, isExpelled: false})
- als je al waarden hebt, kan het korter:
  const name = Wim
  const age = 12
  const person = { name, age } // {name: "Wim", age: 12}
- opvragen van waarden in object: a[5], person.name, person["name"]

String Interpolation

- Java/oud JavaScript: 'Hello ' + name;
- modern JavaScript: `Hello ${name};`

Array methods and fields

- length
- push(value): voegt aan einde toe
- pop(): verwijdert van einde (en returnt verwijderde waarde)
- shift: verwijder van begin
- unshift: voeg toe aan begin
- zie verder https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

Functies

- versie 1:
  - simpel: function greet(name) { return `Hello ${name};` }
  - complex: function register(name) { names.push(name); return `${name} has been registered.;` }
- versie 2:
  - simpel: const hello = function(name) { return `Hello ${name};` }
  - complex: const register = function (name) { names.push(name); return `${name} has been registered.;` }
- versie 3:
  - simpel:
    - 1 parameter const hello = name => `Hello ${name}`
    - 0 parameters : const greet = () => `Hello unknown person!;`
    - 2+ parameters: const add = (first, second) => first + second;
  - complex: const register = name => { names.push(name); return `${name} has been registered.;` }
- teveel argumenten? teveel worden genegeerd
- te weinig argumenten? rest wordt undefined

Controlestructuren

- if., while, do-while, for, switch statement: Zelfde als in Java (maar dan met truthy)
- foreach/enhanced for:
  - for (const a in arr) => [0, 1, 2] // indexes
  - for (const a of arr) => [‘a’,‘b’,‘c’] // values
- try catch throw zoals in Java (maar catch(e) ipv catch(Exception e) omdat JavaScript geen typen heeft).
- JavaScript heeft geen ‘throws’
- break en continue zijn hetzelfde als in Java (inclusief labeled breaks/continues!)

Objecten en arrays kopiëren

```
const newArray = [...myArray, value]
arr1 = ['a','c','f']
arr2 = [...arr1, 'q'] // ['a','c','f', 'q']

const newObj = {...oldObj, key: value, key2: value2}
oldObj = {name: "Piet", pet: 'dog'}
newObj = {...oldObj, pet: 'cat', age: 4}
```

Keys en values uit objecten krijgen

```
Object.keys(obj)
Object.values(obj);
Object.entries(obj); // array van [key, value]
```

Destructuring

- arrays:

```
arr= ['aap','noot','mies','wim', 'zus']
const [first, second, ...rest] = arr;
// first = 'aap'
// second = 'noot'
// rest = ['mies','wim', 'zus']
```

-objects:

```
menu = {breakfast: 'waffles', lunch:'salad', dinner: 'steak'};
const {breakfast, dinner: supper} = menu;
// breakfast = 'waffles'
// supper = 'steak' // rename
```

-functies:

```
const p = {"name": "Clovis", occupation: "king" };

function greet({name}) {
    console.log(`Hi ${name}!`);
}

greet(p); // outputs "Hi Clovis!"
```

Oppassen bij …

- Als je een object wil teruggeven met een lambda

```
const makeNewTim = sweaterColor => { name: "Tim", sweaterColor} // WERKT NIET, JAVASCRIPT DENKT DAT {} een codeblok vormen
const makeNewTim = sweaterColor => ({ name: "Tim", sweaterColor})
```

Map en filter

- map: transformeer array met X waarden in andere array met X waarden
  [1,5,6].map(number => number ** 2) // [1,25,36]
- filter: uit array met X waarden maak een array van 0 tot X waarden
  [1,5,6].filter(number => number % 2) // [1,5]

Omgaan met ontbrekende waarden en methoden

- ?.
  - voor velden die er mogelijk niet zijn:
    - PROBLEEM: console.log(menu.midnightSnack.length) ERROR
    - BETER: console.log(menu.midnightSnack?.length) // undefined
  - voor methoden die er mogelijk niet zijn
    - PROBLEEM: console.log(input.length()) // ERROR if input is a number, like 3 Uncaught TypeError: input.length is not a function

        - BETER: console.log(input.length?.()) // undefined
- ?? const snack = menu.midnightSnack ?? “No snack!”
- ??= menu.midnightSnack ??= “icecream”

Importeren en exporteren

```
// in test.js
const myFunc = () => `MyFunc called!`;
export myFunc;
export default myValue = 5; // only one export default per file

// in testuser.js
import { myFunc } from 'test.js';
import myValue from 'test.js'; // geen {} nodig wegens default export
```

Promises

Sommige functies geven een promise terug: als ze klaar zijn, voer code uit in ‘then’
fetch(`http://localhost:8080/api/v1/items`)
.then(response => response.json())
.then(actualData => setItems(actualData))
.catch(err => console.log(`An error has occurred: ${err.message}.`)
console.log(“fetch called”) // eerst wordt fetch gestart, dan ‘fetch called’ geprint, en als de fetch klaar is krijg je pas dat setItems(actualData) wordt aangeroepen, typisch NA de console.log.

—BONUS
varargs (Java public static void main(String… args)
JavaScript: function addAll(…values) // roep aan als addAll(1,2,3,5)

Als je een array wil geven aan een functie met meerdere argumenten
function plus(first, second) { return first+second }
plus(…[1,2]); // wordt aangeroepen als plus(1,2)

datatypes:
bigint (4n)

React

Aanmaken React-app

- Vite het handigst (al hoef je `npm create vite@latest` niet uit je hoofd te kennen)
- met Vite werkt +SWC het snelste (compileert je code sneller!)
- JavaScript of TypeScript: JavaScript is iets makkelijker en veelzijdiger, al heeft TypeScript meer fans. Mijn aanraden: ga alleen over op TypeScript als je JavaScript voldoende goed beheerst, want TypeScript is feitelijk vermomd JavaScript met extra complexiteit die je misschien beter niet tegelijkertijd wilt leren.
- npm create-react-app is verouderd, heeft allemaal security leaks. Gebruik liever vite (of eventueel NextJs)

Componenten

- zijn (tegenwoordig) functies
- NOTE: op internet zie je nog wel oude klasse-gebaseerde componenten met een render-methode; zijn tegenwoordig overbodig
- componenten geven een JSX-body terug Hello from me! . Moet 1 (top-level) element zijn

```
return <ol>{toDos.map(toDo => <li key={toDo.item}>{toDo.item}</li>)}</ol>
  <AddToDo addToDo={addToDo} />
```

werkt niet (2 elementen)

```
return <>
  <ol>{toDos.map(toDo => <li key={toDo.item}>{toDo.item}</li>)}</ol>
  <AddToDo addToDo={addToDo} />
</>
```

Werkt dus wel (1 hoofdelement)

- de naam van componenten begint met een hoofdletter (waarschijnlijk omdat componenten eerst klassen waren)
- Vite eist dat componenten in een .jsx-file staan (maar niet alle builders doen dat, .js is meestal ook toegestaan).
- component ziet eruit als `const Hello = () => <p>Hello React</p>`
- een Hello-component gebruik je in andere files/JSX als <Hello />

JSX

- staat voor “JavaScript Syntax Extension”
- je kan werken met <></> (voorkomt dat je een div moet aanmaken speciaal omdat alles gewrapt moet worden in 1 top-level component)
- Lijkt enorm op JavaScript, met paar uitzonderingen
    - class => className
    - {} voor JavaScript-code: hallo print gewoon hallo, {hallo} print de waarde van de variable hallo
    - for => htmlFor
    - onclick, onsubmit etc => onClick, onSubmit

Lijsten

- Vaak wil je meerdere data van hetzelfde soort afbeelden, zoals meerdere items in een todo-list
- in HTML: `<ol><li>first</li><li>second</li></ol>`
    - “ol” is voor “ordered list” - een geordende lijst, punten aangegeven met 1, 2, 3
    - “ul” `<ul>` is voor “unordered list” - een “ongeordende lijst”, punten aangegeven met bullet points
- in React kun je een sequentie maken van alle typen elementen, hoeft geen ol/ul/li te bevatten! In plaats van ol of ul kun je <> of `<div>` of wat dan ook gebruiken, of zelfs niets! En in plaats van li kun je alles gebruiken, zelfs je eigen componenten!
- elke (JSX) component in een lijst die je maakt moet een unieke key property hebben voor optimalisatie van rendering, het is niet ‘best practice’ om de positie in de lijst te gebruiken. Normaal gebruik je het id (uit een database of desnoods random gegenereerd) of een naam of een andere unieke eigenschap.
- normaal maak je gebruik van een array van items en dan een map:
    - `<ol>{items.map(item => <li key={item.name}>{item.name}</li>)}</ol>`
    - `{items.map(item =><p key={item.name}>This item is: {item.name}</p>}`

Data doorgeven aan kindcomponent

- in parent component: <Item item={item} />
- child component:
    - optie 1: `const Item = props => <p>{props.item.name}</p>`
    - optie 2: `const Item =({item}) => <p>{item.name}</p>`

Hoe ga je om met een buttonclick of andere verandering?

1. Maak binnen de component (normaal boven de JSX die je returnt) een functie die doet wat je wilt,
   `const sayHello = () => alert("Hi!");`
2. zorg dat de knop of andere component een onClick krijgt met die functie als argument:
   `<button onClick={sayHello}>Greet me!</button>`
3. je hoeft in theorie geen aparte functie te maken, je kunt de code ook tussen de {} zitten. Maar het wordt dan makkelijk onoverzichtelijk
   `<button onClick={() => alert("Also hi!")}>Greet me!</button>`
4. let op dat ik hier NIET <button onClick={alert(“Also hi!”)}>Greet me!</button> gebruik. Als je dat doet wordt de alert onmiddellijk uitgevoerd als de pagina opent, en niet als je klikt! IN REACT, ALS EEN ONCLICK OFZO HET NIET LIJKT TE DOEN, CHECK OF JE DE JUISTE VORM VAN DE METHODE GEBRUIKT:
    - functie zonder argumenten: `<button onClick={sayHello}>Greet me!</button>`
    - functie mèt argumenten: `<button onClick={() => sayHello(name)}>Greet me!</button>`

Maar hoe voer je ingewikkeldere data in dan een knop klikken? Bijvoorbeeld tekst?

```
-gebruik controlled components... Al vraag je je misschien af wat die zijn? Beschouw de volgende code

import { useState } from 'react';

const Input = () => {
    const [item, setItem] = useState("");

    const change = event => {
        setItem(event.target.value);
    }

    const submit = event => {
        event.preventDefault();
        alert(`You typed '${item}'!`);
    }

    return <form onSubmit={submit}>
        <input type="text" value={item} onChange={change}></input>
        <input type="submit"></input>
    </form>
}
```

Je ziet dat ik hier een tekst-input-component gebruik (`<input type="text" ... />`). En ik wil iets met de waarde doen die dat oplevert als de gebruiker op een knop drukt.

Wat ik dus gebruik zijn:

1. Het invoerveld zelf `<input type="text" ...>`
2. Een variabele voor de waarde die in het tekstveld zit, hier heet die variabele `item`.
3. Die variabele maak ik met behulp van een `useState` commando van React. De useState("") betekent dat als de Input-component voor het eerst wordt getoond, item de waarde “” krijgt. `useState` zelf betekent dat telkens als de waarde van value verandert de component opnieuw wordt getekend. Althans, alleen als de waarde via `setItem` wordt veranderd code als `item="hallo"` zorgt er niet voor dat de component opnieuw wordt getekend (en een ‘state’-waarde als item veranderen via = wordt bij mijn weten ook als ‘bad practice’ gezien bij React)
4. Het input-veld heeft een onChange-property die verwijst naar een functie, hier dus de ‘change’-functie.
5. De ‘change’-functie verandert de waarde van item door `setItem` aan te roepen. Zoals je ziet krijgt de functie die aan onChange wordt toegekend automatisch een ‘event’-parameter mee, die met event.target.value de waarde die nu in het tekstveld staat ophaalt. Als dit wat verwarrend klinkt: de browser produceert een ‘event’ als je op bijvoorbeeld een knop drukt. De event.target is wijst naar de component die het event afvuurde. In dit geval dus de knop. De event.target.value verwijst naar de value-property in diezelfde component, in dit geval dus dat ding die “value={item}”, die dus in het begin de waarde “item” heeft maar momenteel (omdat je wat hebt ingetypt) bijvoorbeeld iets als item+‘a’ (als je ‘a’ had ingetypt)
6. onSubmit={submit} betekent dat als je op de submit-knop drukt (die aangegeven is met `<input type="submit"></input>`), de ‘submit’-methode wordt aangeroepen.
7. De submit-methode krijgt van de browser ook een event mee, vandaar die parameter voor de submit-functie.
8. Browsers hebben als standaard (‘default behavior’) dat als je op de submit-knop drukt, een pagina opnieuw wordt geladen, dus alle data wordt ververst. Dat wil je hier niet, want dan wordt alles weer op de beginstand gezet, dus item wordt dan weer “”. Daarom roep je eerst de .preventDefault() aan op het event.
9. Daarna kun je iets doen, in dit geval bijvoorbeeld de waarde afbeelden in een alert-box.

Maar hoe kan je handig data invoeren als je meer dan 1 veld hebt, moet je dan 4 of 5 change-methoden schrijven?

- Als je meerdere input-velden hebt gebruik je meestal een object in plaats van van losse waarden

```
const updateNewItem = event => {
    const { name, value } = event.target;
    setNewToDo({ ...newToDo, [name]: value });
}

return <form onSubmit={submit}>
    <input type="text" name="item" value={newToDo.item} onChange={updateNewItem} />
    <input type="number" name="importance" value={newToDo.importance} onChange={updateNewItem} />
    <input type="number" name="urgency" value={newToDo.urgency} onChange={updateNewItem} />
    <input type="submit" />
</form>
```

Data doorgeven van kind aan oudercomponent

- dit doe je door in de oudercomponent een functie te maken, en die door te geven

Data laden van een API of andere startup-logica

- useEffect voor wat moet worden uitgevoerd bij het re-renderen van een component
- useEffect heeft als laatste argument een array die aangeeft bij welke veranderingen in de state het weer wordt aangeroepen ([] betekent volgens mij als de props verandert)
- axios/fetch

```
useEffect(() => {
  axios("http://localhost:8080/api/v1/todos").then(result => setToDos(result.data))
}, [])

const addToDo = newToDo =>
  axios.post("http://localhost:8080/api/v1/todos", newToDo).then(result => setToDos([...toDos, result.data]));

const addToDo2 = async newToDo => {
  const result = await axios.post("http://localhost:8080/api/v1/todos", newToDo)
  setToDos([...toDos, result.data])
}
```

BONUS:
-hoe zorg je dat verschillende paden naar verschillende componenten leiden (handig voor bookmarks)
-hoe maak je een link?
-hoe zorg je dat een button een andere pagina/component activeert
-hoe extraheer je path variabelen

// Let op: undefined controlled/uncontrolled