JavaScript refresher en React

Inhoud

- JavaScript opfrissen
- React-concepten

Inhoud: JavaScript opfrissen

- Zal (te) snel gebeuren
- Ik verwacht niet dat je alles onthoudt
- Doel is dat als je iets nodig hebt "Zei Wubbo niet zoiets als" en het dan kan opzoeken
- Als je iets in Java kan, kan je vaak makkelijk vertalen naar JavaScript
- Wil je dingen nalezen? …
- Goed je eigen JavaScript-encyclopedie te schrijven

Inhoud: React-concepten

- Maken React-app
- Basis-opbouw (index, root, App)
- Componenten (function en class) + JSX
- State – en waarom dat de essentie van React is
- Props
- Informatie aan ouder-componenten teruggeven
- Tussendoor: oefeningen!
- Als tijd (anders komende dagen) geavanceerder react: context, store..

JavaScript opfrissen

- JavaScript en Java
- ECMAScript?
- IO
- JavaScript-valkuilen
- Basistypen JavaScript
- Const/let/var
- String interpolation
- Declaring functions
- Classes
- Manipulating objects and arrays
- Destructuring assignment
- Repetition & choice
- Importing and Exporting
- *this* and binding
- Promises
- Array methods

- If time:
- Error handling
- Iterators

ECMAScript (JavaScript of JScript)

- JavaScript: december 1995 door Netscape (Java toen hotste programmeertaal wegens demo in 1994)
- JScript: augustus 1996, Microsoft (EEE?). Gebruikt in IE
- ECMAScript: december 1996, als gemeenschappelijke standaard
  - "ECMAScript was always an unwanted trade name that sounds like a skin disease." - Brendan Eich
- Standaard heet ECMAScript (handig voor wiki over nieuwste ontwikkelingen) - https://en.wikipedia.org/wiki/ECMAScript_version_history
- Bijna elke programmeur noemt het JavaScript
- Microsoft heeft tegenwoordig TypeScript

JavaScript en Java

- As het in Java kan, heeft JavaScript meestal hetzelfde of bijna hetzelfde
- Date, Math, Random, Regex, String-methoden...
- Al zijn er uitzonderingen!
  - geen BigDecimal, werk in centen
  - Geen streams maar iterators (en meestal gewoon arrays)
  - Geen maps maar objecten

IO

- Output
  - console.log(text)
  - alert(text)

- Input
  - const reply = prompt("question", initialValue);
  - const agree = confirm("Do you want to give me all your money?");

JavaScript-valkuilen

- " en ' zijn min of meer hetzelfde
- Gebruikt Java een boolean (if, while), JavaScript gebruikt truthy/falsy (false, 0, Nan, "", null, undefined are falsy, rest truthy)
- != en == doen niet altijd wat je verwacht! Gebruik liever !== en ===
- && en || werken wat raar: geven eerste falsey of truthy value terug, anders tweede waarde.
- ; KAN je in JavaScript weglaten, maar is niet verstandig (https://www.freecodecamp.org/news/lets-talk-about-semicolons-in-javascript-f1fe08ab4e53/)

Typen JavaScript

- Primitive
  - boolean
  - number (=double)
  - string (!)
  - undefined
  - null*
  - bigint
  - symbol

- Reference/Object
  - Object
    - Map<string, anyType>
    - Create
      - const hobby = "gardening";
      - function f() { console.log("hello"); }
      - Const obj = { hello: "world", g() { console.log("bye"), hobby, f, list: [1,2,3], obj2: {a: 3, b: 5});
    - access values with a.b or a[b]
  - Array
    - Object with keys like "0", "1", "2"
    - Create: arr = [1,'hello', true];
  - function

Hoe werkt een object? Ongeveer zo

```
class Entry {
    String key;
    Object value;
}

class Object {
    List<Entry> entries;
```

```
void add(entry) {
   index = entries.keys.indexOf(entry.key);
    if (index != undefined)
       entries[index].value = entry.value;
    else entries.add(entry);
}
```

Variabelen en constanten

- Vroeger scripting: a = 5;
- Veel bugs, toen "use strict" en var
- Bugs/onintuïtief wegens hoisting, nu
  - const (als Java's final var) - kan meestal (90%?)
  - let (als Java's var) - tweede keus

String interpolation

- In JavaScript kan "hello " + name + "!";
- Maar ook alternatief: `hello ${name}`
- Voorkomt rottige spatiëringsfouten
- Ook gelijk goed voor multiline strings
- En support " EN ' in string!
- In welke taal je ook gaat gebruiken, zoek string interpolation op
  - Maakte er in elk geval blitz mee bij jonge Python-programmeur

Declaring functions: function statements, function expressions, arrow functions

GEEN argumententypen

GEEN returntype

```
function hello(name) {
    console.log(`Hello ${name}`);
}
```

```
const hello = function(name) {
    console.log(`Hello ${name}`);
}
```

```
const hello = name
=> console.log(`Hello ${name}`);
```

() nodig bij 0 of >1 argument

{} nodig bij >1 statement

```
const greet = (title, lastName) => { if
(!title || !lastName) return "Greeting
failed!"; console.log(`Greetings, ${title}
${lastName}`); return "greeting
succeeded";}
```

Fun with function arguments

```
function sum(...numbers) {
    let sum = 0;
    for (const number of numbers) {
        sum += number;
    }
    return sum;
}

function priorities(first, second) {
    console.log(`First I'll ${first}, then I'll ${second}.`);
}
```

```
sum(1,2,3) // 6
priorities(['eat', 'sleep', 'code']) //
 First I'll eat,sleep,code, then I'll
undefined.

priorities(...['eat', 'sleep', 'code'])
// First I'll eat, then I'll sleep.
```

Fun with function arguments

```
function menu({breakfast, dinner: supper, ...rest}) {
   Console.log(`First ${breakfast}, then ${supper}, and somewhen ${rest.lunch}.`);
}

function greet(name = "Fred") {
   console.log(`Hello, ${name}!");
}
```

```
const schedule = { breakfast: "bread", lunch: "soup", dinner: "beef"};

menu(schedule);
// First bread, then beef, and somewhen soup.

greet("Annika"); // Hello, Annika!
greet(); // Hello, Fred!
```

Classes

```
class Square extends Figure {
    constructor(side) {
        super();
        this.side = side;
    }

    get area() {
        return side * side;
    }

    describe() {
        console.log(`I'm a square with side ${this.side}.`);
    }
}
```

- A "class" in JavaScript is actually a function creating an object!
- Always need "this." Except when defining methods and fields
- Can use get, set, static, # (for private)
- Created in Java-look-alike way, but can be totally distorted!

Manipulating objects and arrays

```
const arr = [4, 5, 6];

const person = { name: "Hanja",
hobby: "hiphop" };

Object.keys(arr); // returns ['0', '1',
  '2']

Object.keys(person); // returns
  ['name', 'hobby']

Object.values(arr); // returns [4, 5, 6]

Object.values(person); // returns
  ['Hanja', 'hiphop']
```

- Object.entries(arr); // returns [['0', 4], ['1', 5], ['2', 6]]
- Object.entries(person); // returns [['name', 'Hanja'], ['hobby', 'hiphop']]

- delete person.hobby // person wordt {name: 'Hanja'}

Copying objects and arrays

- WRONG (ARRAY)
  - const a1 = [1, 2, 3];
  - const a2 = a1;
  - a2.push(-5); // modifies a1 too!
- WRONG (OBJECT)
  - const p1 = {name: "Brenda", hobby: "beet-growing" }
  - const p2 = p1;
  - p2.hobby = "beagle-breeding";
    - Modifies p1 too!
- RIGHT (ARRAY)
  - const b1 = [1, 2, 3];
  - const b2 = [...b1];
  - b2.push(-5);
- RIGHT (OBJECT)
  - const q1 = {name: "Brenda", hobby: "beet-growing" }
  - const q2 = {...q1};
  - q2.hobby = "beagle-raising";
  - OR: const q3 = {...q1, hobby: "beagle-raising" };

Repetition and choice

- Repetition
  - for, while, do-loops: same as Java, but any value instead of only boolean as condition (truthy/falsy)
  - for (const a in arr) // prints keys (0, 1, 2...) of arrays and objects
  - for (const b of arr) // prints values ('a', 'f', 'g') of arrays and objects
  - In = indices, of = for falues (ehm... values)

- Choice
  - If, ?:, switch statement: like Java, but based on truthyness
  - No real equivalent for switch expression, can occasionally use

  - const choices = { yes: "I agree!", no: "You should reconsider...", maybe: "I should explain it to you again..." };

  - const answer = prompt("Do you agree with proposal 362?");
  - alert(choices[answer] || "Please answer with 'yes', 'no' or 'maybe'!")

Destructuring assignment

- Same as in parameter lists!
- const [first, second] = myList; // for arrays
- const [first, second, ...rest] = myList; // for arrays
- const { name, hobby: job } = someFriend; // for objects
- const { name, hobby: job, ...rest } = someFriend; // for objects

Imports and exports

```
class GreatClass {}
const greatFunc = () => console.log("Hello world!");
const greatNumber = 1;
const greatString = "Hello!";
export { greatFunc, greatNumber as greaterNumber, greatString };
export default GreatClass;
```

Imports and exports: imports

```
import {greatFunc, greaterNumber as greatestNumber } from
   './great_module';
import GreatClass from './great_module';
import * as gm from './great_module';


console.log(gm.greatString);
console.log(greatestNumber);
```

'this' and binding

- "this" normally only works in methods (non-arrow-functions in objects or classes)
- Can make it work in functions, though!

```
const ralph = { firstName: "Ralph", lastName: "Wiggum" };
const introduce = function() {
    console.log(`Hi, I'm ${this.firstName}!`);
}
introduce(); // shows "Hi, I'm undefined!"
introduce.call(ralph); // displays "Hi, I'm Ralph!"
const ralphIntroduce = introduce.bind(ralph);
ralph.introduce(); // Uncaught TypeError: ralph.introduce is not a function
ralphIntroduce(); // displays "Hi, I'm Ralph!"
```

'this' and binding

- Note: if a function has arguments, you can use
  - functionName.call(object, arg1, arg2, arg3...)
  - functionName.apply(object, arrayOfArgs)

  - this.myfunc = this.myFunc.bind(this);

Promises

- Produce some result at some time
- .then() and .catch() for normal result and errors. Can be chained

```
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}.`)
```

Functional programming: filter/map/reduce

```
const dwarves = ["Happy", "Bashful", "Sneezy", "Grumpy", "Sleepy", "Dopey", "Doc"];

const dDwarves = dwarves.filter(dwarf => dwarf.startsWith("D"));
console.log(dDwarves); // lists ['Dopey', 'Doc']

const bigDwarves = dwarves.map(dwarf => dwarf.toUpperCase());
console.log(bigDwarves); // lists ['HAPPY', 'BASHFUL', 'SNEEZY', 'GRUMPY', 'SLEEPY',
   'DOPEY', 'DOC']

const quickDwarves = dwarves.reduce((soFar, current) => soFar+current, "");
console.log(quickDwarves); // outputs "HappyBashfulSneezyGrumpySleepyDopeyDoc"
```

Arrays have lots of useful methods!

- Add to/remove from end or start: push, pop, shift, unshift
- Change sequence in array or copy: sort, reverse, toSorted, toReversed
- Check if value(s) are present: value=includes, indexOf, lastIndexOf; predicate: every, some, find, findLast, findIndex, findLastIndex
- Make sublist: slice
- Delete/replace sublist: splice (toSpliced)
- Fill
- To single string: join
- Filter, flat, map, foreach, flatMap, reduce, reduceRight
- keys(), values(), entries()
- at(-1) for last element...
- length

Restant...

- ?? (zoals ||, maar geeft alleen tweede waarde als eerste waarde null of undefined is)
- ?. (person?.occupation?.location: geeft undefined als iets undefined is, geen error)
- ** : 2 **5 === 32 (alternatief voor Math.pow)
- async / await: alternatief voor promises
- (ook boel andere dingen als typed arrays, maar weet niet hoe nuttig die zijn)

JavaScript opfrissen

- IO
- JavaScript-valkuilen
- Basistypen JavaScript
- Const/let/var
- String interpolation
- Declaring functions
- Classes
- Manipulating objects and arrays
- Destructuring assignment
- Repetition & choice
- Importing and Exporting
- *this* and binding
- Promises

- If time:
- Error handling
- Iterators

React!

Nieuwe react app

- npm create vite@latest

Index.html // resource loaded by default

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>My title</title>
</head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Main.jsx // why JSX?

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'


ReactDOM.createRoot(document.getElementById('root')).render(
   <React.StrictMode>
      <App />
   </React.StrictMode>,
)
```

App.jsx

```
import './App.css'
import { useEffect, useState } from 'react';
const App = () => {
  const [items, setItems] = useState([]);
  useEffect(() => {
  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}.`))
  }, []);
  return <ol>{items.map(item => <li key={item.id}>{item.name}</li>)}</ol>
}
export default App;
```

State!

- React werd beroemd wegens zijn snelheid
- Door alleen scherm te updaten als het nodig was
- Twee aspecten
  - In 1x updaten via Virtual DOM (kladblok)
  - State!

State

- "state" was de naam van een field in een React component
- Tegenwoordig gebruiken we normaal functional components, met 1 of meer useState()
- ZODRA iets van de state verandert, worden component en children gererenderd
- Maar om daarvoor te zorgen mogen we niet zomaar state = newState gebruiken, maar methodes

State en useState

- const [items, setItems] = useState([]);

- items: om de waarden te krijgen (for (item of items) …
- setItems: om de waarden van items te veranderen (setItems(["cabbage", "pears", "computers"])
- [] (argument useState): beginwaarde van items

useEffect?

```
useEffect(() => {
 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}.`))
 }, []);
```

- Normaal voor initialisatie
- Wordt uitgevoerd als de component voor de eerste keer wordt gerenderd, EN/OF als een van de variabelen in het tweede argument verandert

JSX: om te onthouden

- JSX begint bij de eerste tag (zoals <p> of <>)
- Je kan JavaScript erin schrijven door {} te gebruiken rond de JavaScript
- Is niet helemaal html: onclick => onClick, class => className

Waarden doorgeven naar kinderen: props

- Wat als je waarden wilt doorgeven aan een child component?
- LIJKT op HTML: key1="value1" key2="value2" key3
- Key zonder expliciete waarde krijgt waarde *true*
- Maakt in feite object <MyComponent {...obj} /> werkt ook!
- Het object dat je doorgeeft wordt het eerste (en normaal enige) argument van de component
- Die parameter wordt traditioneel *props* genoemd (van properties)

Props naar kinderen: voorbeeld

```
/* App.jsx */ return (
   <div className="App">
    <ToDoList items={toDoList} />
   </div>)
---
```

```
/* ToDoList.jsx */  const ToDoList = props
=>
   <ul>{props.items.map(item =>
  <ToDoItem key=  {item.description} item
  ={item}  />
   )
   }</ul>
```

Key=?

- React wil efficiënt updaten
- Dus alleen dingen die veranderen opnieuw tekenen
- Elk element in lijst geassocieerd met unieke key
- Best practices: GEEN index, maar unieke waarde

En hoe geef je waarden TERUG aan ouders?

- Maak één van de doorgegeven properties een functie die de staat van de parent aanpast!

Passing back-example

```
 const addItem = text =>
   setToDoList([...toDoList, { description: text, done: false }]);

return (
   <div className="App">
    <ListInput returnItem={addItem}/>
   </div>)
```

```
const ListInput = props => {
   const [value, setValue] = useState("");

   const updateMe = event => setValue(event.target.value)

   const submit = event => {
      event.preventDefault();
      props.returnItem(value);
      setValue("");
   }

   return (
      <form onSubmit={submit}>
         <input type="text" value={value} onChange={updateMe}/>
         <input type="submit" value="Add this item!" />
      </form>
   )
}
```

Controlled component

```
const ListInput = props => {
  const [value, setValue] = useState("");

  const updateMe = event => setValue(event.target.value)

  const submit = event => {
    event.preventDefault();
    props.returnItem(value);
    setValue("");
  }

  return (
    <form onSubmit={submit}>
      <input type="text" value={value} onChange={updateMe}/>
      <input type="submit" value="Add this item!" />
    </form>
  )
}
```

- Bij form: zorg dat onChange de state update
- Typisch: event => setX(event.target.value)
- Voor meerdere velden: kijk naar https://www.pluralsight.com/guides/handling-multiple-inputs-with-single-onchange-handler-react

Gotchas

- onChange={updateMe} werkt goed als functie geen argumenten heeft
- Wel argumenten?
  - onChange={updateMe(1)} // wordt 1x aangeroepen, bij laden component
  - onChange={() => updateMe(1)} // werkt wel goed!

Iets over Componenten-functies en Componenten-klassen

- Traditioneel werd React geschreven met klassen (class App extends Component { } die een render() methode hadden die JSX teruggaf
- Klassen zie je vaak in oude code of code die iets heel speciaals moet doen (lifecycle hooks)
- Klassen hebben een paar tricky dingen als functies moeten binden
- Om React 'meester' te worden moet je ook met React-klassen kunnen werken
- Maar ik hoop dat functies voor de meeste dingen genoeg zijn

Inhoud: React-concepten

- Maken React-app
- Basis-opbouw (index, root, App)
- Componenten (function en class) + JSX
- State – en waarom dat de essentie van React is
- Props
- Informatie aan ouder-componenten teruggeven
- Tussendoor: oefeningen!
- Als tijd (anders komende dagen) geavanceerder react: klassen, context, store..