



IK WIL

Javascript Advanced

Introductie

- Voorstelrondje
- Agenda
- Dagindeling
- Cursusmateriaal
- Doel van deze cursus

Agenda (dag 1)

1. Classes en overerving
2. JSON
3. Modules
4. Promises
5. Observables en operators met RXJS

Agenda – vervolg (dag 2)

6. HTTP

7. Offline Applicaties

8. Lazy Loading

9. Web Sockets

10. PWA introductie

Dagindeling

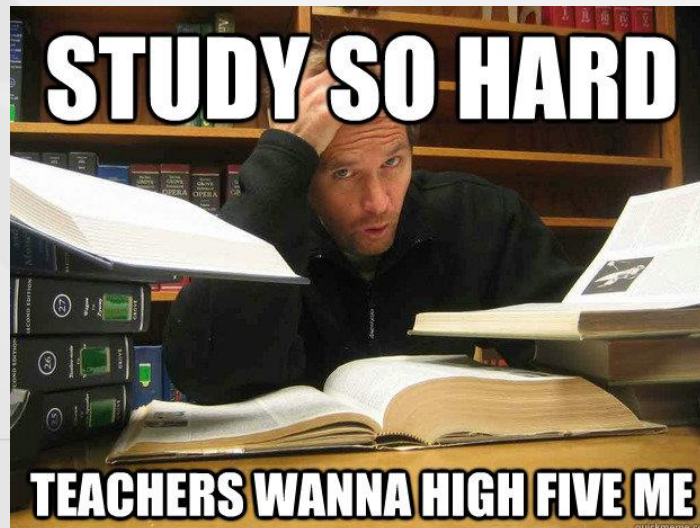
- 8:45 - Begin cursusdag
 - Introductie
 - H1 Hoofdstukinhoud
 - theorie
 - opdrachten maken
 - opdrachten bespreken
- 10:30-10:45 Koffiepauze
- 12:00-12:45 Lunchpauze
- 14:30-14:45 Koffiepauze
--16:00 Einde cursusdag

Cursusmateriaal

- Online documentatie: mijn.vijfhart.nl
- Chrome webbrowser
 - Chrome development tools
- RAD tool: codepen.io of [JSFiddle](http://jsfiddle.net)
- Eigen tools naar keuze: VS Code, WebStorm, Atom, etc.

Doel van de cursus

Geavanceerde webapplicaties kunnen maken met JavaScript door toepassing van classes, de fetch API, Promises, Observables en Web Sockets. Daarnaast het verhogen van de gebruiksvriendelijkheid door het toepassen van Offline mogelijkheden, Lazy Loading en het kennen van de concepten van een PWA..



Classes en overerving

Leerdoelen

- Kunnen benoemen van de voordelen van classes
- Kunnen benoemen wat het Factory Design Pattern is
- Zelf classes kunnen maken en overerving kunnen toepassen

Praktijk toepassing

- Beschrijven van data in vele contexten
- Omgevingen waarbij door het gebruik van prototype de code lastig te beheren is

Inhoud hoofdstuk

- Classes
- Access modifiers
- Factory Design Pattern
- Overerving
- Static modifier

Classes

- Sinds ES6 nieuwe schrijfwijze voor het maken van een constructor functie d.w.v. een Class
- Aanmaak m.b.v. Class keyword, gevolgd door (evt) een constructor, attributen en methoden (functies)
- Bevordert leesbaarheid en overeenkomst met andere programmeertalen
- Beperkte mogelijkheden wat betreft access modifiers; geen private, etc.
- Factory design pattern niet mogelijk

Classes

- Constructor functie 'vroeger':

```
function Persoon(naam, leeftijd, hobby)
{
    this.naam = naam;
    this.leeftijd = leeftijd;
    this.hobby = hobby;
}

Persoon.prototype.zegIets = function() {
    return "ik vertel graag iets over mijn hobby: " + this.hobby;
}

var p1 = new Persoon("Amy", 18, "Paard rijden");
var p2 = new Persoon("Jacob", 20, "Rondjes rennen");
```

<https://codepen.io/5hart/pen/VMaZxy>

Prototype

- Ieder object dat gemaakt is met een constructor functie heeft een property genaamd prototype
- In prototype zit een link naar een ander object
- Bovenste object in JS is 'Object'. Erboven zit alleen 'null'; deze heeft géén prototype
- Zoeken naar properties gaat altijd eerst bij eigen properties, daarna bij de properties van het object in het prototype v/h huidige object en zo verder omhoog.
- <https://codepen.io/5hart/pen/abZoRBo>
- Bonus vb met literal notatie:
<https://codepen.io/teacherStijn/pen/oNNPBpO>

Classes

- Class notatie nu:

```
class Persoon {  
    constructor (naam, leeftijd, hobby)  
    {  
        this.naam = naam;  
        this.leeftijd = leeftijd;  
        this.hobby = hobby;  
    }  
  
    zegIets() {  
        return "ik vertel graag iets over mijn hobby: " +  
this.hobby;  
    }  
  
    var p1 = new Persoon("Amy", 18, "Paard rijden");  
    var p2 = new Persoon("Jacob", 20, "Rondjes rennen");
```

Classes

- Prototype wordt nog gewoon gebruikt achter de schermen:
<https://codepen.io/5hart/pen/abZoRBo>
- Zowel de oude als de nieuwe notatie zijn middels setters en getters uit te breiden. (zie evt boek op pagina 110)

<https://codepen.io/5hart/pen/YbyqWw>

Let op: beware of recursie: [link](#)!

Access modifiers

- Access modifiers niet mogelijk in een JavaScript class. Achter de schermen wordt immers nog steeds van een constructor function gebruik gemaakt.
- Properties zijn wel geconditioneerd in te stellen middels getters.
- Afschermen variabelen wordt nog veelal gedaan door gebruik te maken van *closures*.

Factory Design Pattern

- Een patroon wat een nieuwe instantie (object) van een klasse (of ouderwets: constructor functie) oplevert middels de aanroep van een methode, zónder het woord 'new' dus.
- Doordat we niet met private properties kunnen werken in een klasse, daarin niet zomaar mogelijk.
- Tussen oplossing:
<https://codepen.io/5hart/pen/aMggMj>

Let op: gebruik functie beschrijvingen zonder woord 'function' mag in een object 😊.

Oefening

Maak een 'Tafel' klasse met minimaal een drietal eigenschappen.

Overerving

- Middels overerving kunnen we eigenschappen verkrijgen van een zogeheten *superklasse* en code besparen.
- We kunnen een klasse definiëren als een *subklasse* van een andere klasse middels het keyword *extends*.

Tafel voorbeeld zonder overerving:

<https://codepen.io/5hart/pen/wObYVw>

Tafel voorbeeld mét overerving:

<https://codepen.io/5hart/pen/qvGQdK>

Extra vb:

<https://codepen.io/stijnjanssen/pen/oNNMoMv>

Static modifier

- In JavaScript kunnen we een methode (functie) *static* maken.
- Dit betekent dat deze methode bij de klasse (de blauwdruk) hoort, niet bij een specifieke instantie (object) van de klasse.
- Heeft *niets* te maken met aanpasbaarheid (doen we middels *const*, niet eenvoudig in class te doen).

Voorbeeld static methode:

<https://codepen.io/5hart/pen/YgbmyX>

Static modifier

- Een recente toevoeging is de mogelijkheid static eigenschappen te gebruiken:

<https://javascript.info/static-properties-methods>

Opdrachten

- Kies een hobby die je hebt of iets wat je leuk vindt en schrijf dit uit naar een klasse, met attributen en methoden.
- Denk aan: een sport (voetbalclub, formule 1 teams), een vervoersmiddel (auto, fiets, brommer, motor), een cursus (HTML, SQL, Java) of bijvoorbeeld een verzameling (kaarten, postzegels).
- Maak één klasse, met minimaal twee attributen en minimaal twee methoden.
- *Optioneel:* schrijf ook een constructor, die twee parameters accepteert. Maak óók een subklasse onder deze klasse indien mogelijk.

JSON

Leerdoelen

- Kunnen benoemen van de redenen voor het gebruik van JSON
- Een JSON object kunnen maken

Praktijk toepassing

- Klaarzetten van actuele weerinformatie die vanuit sensoren op een web API wordt gezet
- Verzamelen van automerk specifieke gegevens voor een vergelijkingssite
- Inlezen van gegevens van Marktplaats om zelf op een website te gebruiken

Inhoud hoofdstuk

- JSON
- JSON in de browser

JSON

- JavaScript Object Notation
- Taal om gegevens mee uit te wisselen
- In 1999 opgezet als standaard, naast talen zoals XML.
- Minder beschrijvend dan XML, minder code dan XML.
- Schrijfwijze bijna identiek aan JavaScript objecten; herkenbaar en eenvoudig te combineren met andere ECMASCRIPT talen.
- Inmiddels in afgeleide vorm ook veel toegepast in databases zoals [*MongoDB*](#).

JSON

- Voorbeeld van één JSON object, met een eigenschap 'gameLijst', die een verzameling van objecten bevat:

```
{  
  "gameLijst": [  
    {  
      "titel": "Zelda",  
      "genre": "RPG"  
    },  
    {  
      "titel": "Gran Turismo",  
      "genre": "Racing"  
    }  
  ]  
}
```

JSON in de browser

- JSON is platte tekst, op een vastgelegde wijze geschreven
- Eigenschappen én waarden staan tussen dubbele quotes
- Om JavaScript objecten weg te schrijven naar JSON formaat (serialisatie) en andersom (deserialisatie), zijn in veel programmeertalen methoden geschreven.
- In JavaScript gebruiken we respectievelijk `JSON.stringify()` en `JSON.parse()`

JSON in de browser

Voorbeeld:

<https://codepen.io/stijnjanssen/embed/wOKgOW>

Meer naslag over het JSON formaat is te vinden in het boek op pagina 77 en op: [MDN](#).

Opdrachten

- Maak een JSON bestand op basis van een hobby of interesse die je in de opdrachten van het vorige hoofdstuk hebt gemaakt als klasse.
- Maak een *array* van JSON objecten, volgens de opbouw van je klasse.
- In je JSON objecten mogen geen functies voorkomen.
- Optioneel: i.p.v. dit met de hand te doen... zou je natuurlijk ook een functie kunnen gebruiken...?

Modules

Leerdoelen

- Twee of meer redenen voor het gebruik van modules kunnen benoemen
- De onderliggende concepten van een module kunnen benoemen
- Een eigen module kunnen maken

Praktijk toepassing

- jQuery, React en andere JavaScript libraries
- Functionaliteit van een willekeurig systeem onderbrengen onder één *namespace*

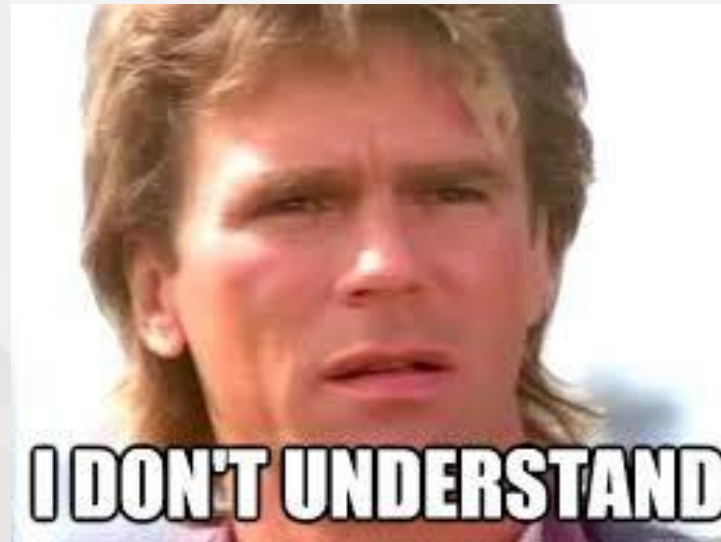
Inhoud hoofdstuk

- Closures
- IIFE's
- Zelfbouw modules
- ECMAScript modules

Closures

- Een functie die een functie of object retourneert, waarbij het geretourneerde toegang blijft houden tot eigenschappen van de buitenste functie.
- Bedoeld om bij elkaar behorende code in één functie te stoppen en deze later aan te kunnen roepen, waarbij de interne variabelen bekend blijven, maar niet van buitenaf te benaderen zijn.
- Zorgt dus ook voor opgeschoonde globale scope.

Closures



Closures

- Opbouw is als:

```
function buitenste() {  
  let waarde = 0;  
  
  return function(){  
    waarde++;  
    alert(waarde);  
  }  
}
```

- Opslaan resultaat (dus binnenste functie) in een var:

```
let draaien = buitenste();  
  
draaien(); // 1  
draaien(); // 2  
...
```

- *Waarde* blijft benaderbaar voor de binnenste functie!
- Buitenste() is nu vaker aan te roepen! (wil je dat?)

IIFE's

- Een functie die direct gedraaid wordt
- Maar eenmalig gedraaid kan worden
- Vaak een closure
- Geeft vaak een *object* terug (functie mag ook)
- De kern van JavaScript zelfbouw modules

IIFE's

Voorbeeld:

```
persoonsBeheer = (function(){  
  ...  
  return {  
    ...  
  }  
})();
```

<https://codepen.io/5hart/pen/VGLBYN>

Door de ronde haken **()** om de functie heen én de ronde haken direct achter de functie, wordt hij direct, éénmalig, gedraaid.

JavaScript zelfbouw modules

- Combinatie van een closure en een IIFE
- Bedoeld voor herbruikbare applicatie functionaliteit, onder een duidelijke namespace
- Dient direct aangeroepen te worden (IIFE, dus met haken ()) om direct bruikbaar te zijn

Voorbeeld:

<https://codepen.io/stijnjanssen/embed/JzYOOZ>

Voorbeeld met HTTP calls:

<https://codepen.io/5hart/pen/jljgVG/>

JavaScript zelfbouw modules

- Geen mogelijkheden voor afhankelijkheden tussen stukken code
- Alle benodigdheden dienen binnen dezelfde scope in dezelfde module te zitten
- Hiervoor zijn ECMAScript modules bedacht

ECMAScript modules

- Sleutelwoorden 'import' en 'export' maken functies en objecten respectievelijk als verwijzing bruikbaar of juist naar buiten kenbaar.
- type="module" bij de <script> tag om aan te geven dat we met een module te maken hebben.
- Named exports en [default](#) exports (wel of geen klassenaam bij het import statement plaatsen)
- De code dient te draaien op een (al dan niet lokale) webserver(!)

(e.v.t. v.b. code **import_export.zip**)

Meer over modules lees je hier:

https://exploringjs.com/es6/ch_modules.html#sec_module-loader-api

Opdrachten

Schrijf een zelf gemaakte module, die entries van specifieke objecten in een array beheert. Deze entries zijn objecten van je in het hoofdstuk gemaakte klasse

Gebruik als begin een notatie als volgt:

```
let mijnKlasseBeheer = (function() {  
  let data = [];  
  
  return {  
    voegToe: function(element){ },  
    verwijder: function(element){ }  
  
    // Enzovoorts  
  };  
})();
```

Maak binnen de module minimaal functies om:
elementen toe te voegen, elementen te verwijderen én een
functie die de array teruggeeft als JSON array.

Promises

Leerdoelen

- Kunnen benoemen wat een Promise is
- De verschillen tussen een callback functie en een Promise kunnen benoemen
- Één of meer voordelen van het gebruik van een Promise kunnen benoemen

Praktijk toepassing

- Het doen van asynchrone acties als reactie op elkaar en daarbij de code véél overzichtelijker te houden
- Het maken van een serververzoek, die (uiteeraard) geheel asynchroon dient te gebeuren

Inhoud hoofdstuk

- Promises
- Async / await

Promises

- Maken het beter mogelijk om te werken met asynchrone acties in JavaScript
- Gebruikt een `.then()` en een `.catch()` methode om code uit te voeren wanneer de Promise gelukt is of niet gelukt is

```
mijnPromise$.then(result=>console.log(result))  
                .catch(err=>console.log(`oops! ${err}`));
```

Promises

Bij handmatig maken van een Promise:

- Schrijf functie die meegegeven wordt aan de Promise
- Deze functie dient twee parameters (wat straks functies zijn), te accepteren en deze ook in de code uit te voeren onder bepaalde voorwaarden

Voorbeeld:

<https://codepen.io/stijnjanssen/pen/BaaOLbP>

<https://codepen.io/5hart/pen/VRowME>

Bekijk ook:

https://developers.google.com/web/fundamentals/primers/promises#promisifying_xmlhttprequest

Promises

In reactie op een Promise, kunnen we een door gaan met het resultaat in een vervolg Promise. Dit noemt met *Promise chaining*.

- Voordeel is dat in tegenstelling tot het gebruik van callback functies, de code overzichtelijker wordt.
- Chaining wordt veel gebruikt bij het reageren op HTTP calls met een andere HTTP call en daarop weer met een andere HTTP call, enzovoorts.

Voorbeeld: <https://codepen.io/5hart/pen/KEOpKG>

Bonus content: [The Coding Train](#) over Promises.

Async / await

- Deze sleutelwoorden kunnen het werken met Promises eenvoudiger en vaak overzichtelijker maken
- **Async** kunnen we als keyword vóór een functie beschrijving plaatsen. Hierdoor zal deze functie een Promise opleveren. Ook al staat er bijvoorbeeld: `return "toffe tekst"` in, dan zal dit automatisch in een Promise [verpakt](#) worden!
- **Await** zal ervoor zorgen dat de volgende code pas uitgevoerd wordt zodra de hierachter opgegeven Promise voltooid is.

Let op: await zelf mag je enkel in een async functie gebruiken.

Een mooie vergelijking tussen Promises en async/await lees je [hier](#).
Meer info over asynchroon programmeren in pag. 181 t/m 188 van het boek.

Bekijk evt [deze](#) uiterst informatieve pagina van Google Expert [Jecelyn Yeen](#).

Opdrachten

In een volgend hoofdstuk gaan wij met de fetch API, en daarmee Promises, werken.

Daarom bij dit hoofdstuk, geen opgaven.



Observables en operators met RXJS

Leerdoelen

- Kunnen benoemen van de voordelen van een Observable boven een Promise
- Kunnen benoemen van twee of meer nuttige RXJS operators

Praktijk toepassing

- Opzetten van een soort Service Bus, waarmee data eenvoudig tussen componenten heen en weer gestuurd kan worden (Subject en Publisher concept)
- Datastromen aanpassen, opsplitsen, samenvoegen, annuleren, enzovoorts

Inhoud hoofdstuk

- Observables
- RXJS
- RXJS Operators

Observables

- Een observable is meer dan een Promise.
- Een observable is iets wat je kunt 'volgen'.
- Zie het als een datastroom. In de echte wereld bijvoorbeeld een **tijdschrift abonnement**. Je kunt je daarop abonneren.

Je weet niet wanneer je het tijdschrift krijgt, maar wél dat je het krijgt. Ook verschillende mensen kunnen zich abonneren. Je kunt ieder moment in het jaar instappen wanneer je dat wilt, wil je eerdere uitgaven ook ontvangen? Geen probleem.

Observables

Verschillen met Promises:

Promise	Observable
Abonnement stopt bij binnenkomst data	Meermalig data te ontvangen
Abonnement is niet tussentijds te stoppen	Tussentijds stoppen is mogelijk
Modificatie van de datastroom mogelijk, met eigen code	Modificatie van de datastroom mogelijk ook met behulp van vele operatoren
Gaat hoe dan ook een keer 'af'	Stuurt pas data bij subscribe() (abonnement)

Veel gebruikte toepassing:

- Subject als Observable (na klikken knop, item op de [lijn](#))

RXJS

- De ReactiveX library, of kortweg RX, maakt het mogelijk om te werken met Observables (en subtypen daarvan).
- Deze library is voor veel programmeertalen te gebruiken.
- De JavaScript variant heet RXJS.

RXJS

- De installatie van RXJS is vrij eenvoudig, maar we hebben ook nodejs nodig. Hiervoor kunnen we de laatste LTS versie downloaden.

Daarna beginnen we de NPM installatie vanuit een MS-DOS venster:

```
npm install rxjs
```

(gebruik eventueel de vlag `-save` om deze afhankelijkheid in je project op te slaan!)

Of kijk voor installatie op: [rxjs](#)

- Bekijken welke versie je hebt geïnstalleerd kan snel door het commando:

```
npm list
```

RXJS

- Een alternatieve en eenvoudigere manier om met RXJS te oefenen is via de website:
<https://stackblitz.com>
- Zelf een Observable maken (met de hand) kan, maar vaak worden factory methoden (zoals 'of' en 'from' gebruikt):

<https://stackblitz.com/edit/rxjs-rox6sc>

RXJS - Operators

Je kunt na het abonneren nog extra acties, zogeheten *operators* uitvoeren op de datastroom, denk aan:

- meerdere abonnementen combineren tot één
- een brievenbus filter op de bus plakken, die alléén de zomer editie weigert
- een schredder in je brievenbus, die het abonnement opsplitst in meerdere abonnementen
- iedere oneven maand je tijdschrift een rode kaft geven

RXJS - Operators

- Werk je met RXJS versie 5.5 of nieuwer? Dan moet er om alle operators die je (tezamen) op een Observable toepast, een [pipe\(\)](#) operator.

Voorbeeld:

```
import { of } from 'rxjs';
import { map, tap } from 'rxjs/operators';

of(1,2,3).pipe(
  // 'x' aangepast naar tekst met x erin, let op komma:
  map(x => `Prachtige waarde: ${x*2}`),
  // 'tap' biedt mogelijkheid tot uitvoeren actie
  // die niets met 'x' doet.
  tap(x => console.log('Ik log alleen even!' + x));
);

// vervolg...
```

RXJS - Operators

// vervolg...

Hierna willen we ergens in onze code nog abonneren op de datastroom. Dit doen we middels de methode 'subscribe'. Deze methode accepteert twee functies, één 'goed' en een 'fout' functie. Dit kan bijvoorbeeld direct achter de pipe() operator middels:

```
.subscribe(  
  data=>{ console.log(data);},  
  error=>{ console.log(error);}  
);
```

Of dit kan als we het resultaat van de pipe() opslaan in een Observable (na bovenaan ook: import { Observable } from 'rxjs' getypt te hebben:

```
let oefenObs$ = of(1,2,3).pipe(...);  
oefenObs$.subscribe(...);
```

RXJS - Operators

Uitstekende naslag, met name voor de operators is te vinden op:

<https://rxjs-dev.firebaseapp.com/guide/operators>

Tot slot, een zéér uitgebreide toelichting op *reactive programming* met Observables:

[The Introduction to Reactive Programming you've been missing.](#)

(onderste is een aanrader!)

Opdrachten

Ga naar: <https://stackblitz.com/>

Klik op:

START A NEW APP

gevolgd door



RxJS
TypeScript

- Maak zelf een eenvoudige Observable aan middels de RXJS library.
- Gebruik hiervan de *of()* factory methode om de daadwerkelijke Observable aan te maken.
- Abonneer op je eigen Observable

Optioneel:

- Gebruik één of meer operators die je Observable datastroom aanpassen

HTTP

Leerdoelen

- Kunnen beschrijven wat HTTP verzoeken zijn
- Kunnen maken van een HTTP verzoek middels het XMLHttpRequest object
- Één of meer voordelen kunnen benoemen van het gebruik van de Fetch API
- Een HTTP verzoek kunnen maken, middels de Fetch API

Praktijk toepassing

- Googelen
- Informatie ophalen van en sturen naar een server
- Dus: héél erg veel!

Inhoud hoofdstuk

- HTTP
- HTTP requests en responses
- Fetch API

HTTP

- Bijna alle verzoeken van via het Internet gebeuren middels het HTTP en HTTPS protocol
- Voorbeelden van verzoeken zijn:
 - Google actie
 - Afbeelding op een webpagina bekijken
 - CSS of JavaScript bestand inladen
 - Verbinding maken met een bijv. (JSON) API
- HTTP bericht bestaat altijd uit een *header* (regels en opties) en een *body* (inhoud)

HTTP requests en responses

Header content:

- startregel met daarin het protocol
- de statuscode
- het soort verzoek
- 0 of meer opties die je als *key-value* paren kunt meegeven als je een verzoek verstuurt, zoals de gebruikte user-agent, eventuele encoding en authorisatie

Bijvoorbeeld:

HTTP/1.1 206 Partial content

Date: Wed, 8 May 2019 13:25:24 GMT

Last-Modified: Wed, 8 May 2019 11:58:08 GMT

Content-Type: image/gif

HTTP requests en responses

Overwegingen bij opstellen HTTP bericht

- Versturen en ontvangen we synchroon of asynchroon?
- Gebruiken we POST, GET, PUT/PATCH of DELETE als methode?
- Welk content-type ([MIME-type](#)/gegevenssoort) is de data die we versturen en/of ontvangen?
- Moeten we gebruik maken van authenticatie en hoe steekt deze in elkaar?
- Gebruiken we libraries zoals OAuth(open source), Auth0, Permit, Feathers of eigen geschreven systeem?
- Additionele header informatie al dan niet opgeven zoals encoding (GZIP), referer (waar kom ik vandaan), enz.
- Moeten we een bericht versturen (of ontvangen) van of naar een ander domein? Dan lopen we mogelijk tegen CORS ([Cross-Origin Resource Sharing](#)) issues aan.
- Gebruiken we [JSONP](#) om externe content in onze pagina te laden en hiermee CORS issues te omzeilen, maar onze pagina daarmee ook kwetsbaar maken?

HTTP requests en responses

Werken met HTTP berichten

- Gebruik een tool zoals: F12->Network, HTTP Trace (extensie Google) of *Postman*.
- Meer details over HTTP lees je in het boek op pagina 311 t/m 314.

HTTP requests en responses

Maken van een HTTP bericht

- In JavaScript zit de mogelijkheid om een HTTP call te doen middels gebruik van het zogeheten XMLHttpRequest object.
- We moeten middels callback functies de goed- (en evt foutsituaties afvangen
- Werken middels dit object is vrij omslachtig en fout gevoelig
- We kunnen zelf headers toevoegen met behulp van de setRequestHeader() functie die een 'key' en een 'value' verwacht;

<https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/setRequestHeader>

HTTP requests en responses

Voorbeeld:

<https://codepen.io/5hart/embed/bLPPBg>

We zien dat:

- We middels een callback functie het 'readystatechange' event afvangen
- Énkel bij een status gelijk aan 200 en een readyState gelijk aan 4, is het bericht goed verzonden en de respons ontvangen.
- Uiteraard zijn nog veel meer status [codes](#) en [readyState](#) waarden af te vangen

HTTP requests en responses

Uiteraard kunnen we ook POST, PUT, PATCH en DELETE berichten versturen.

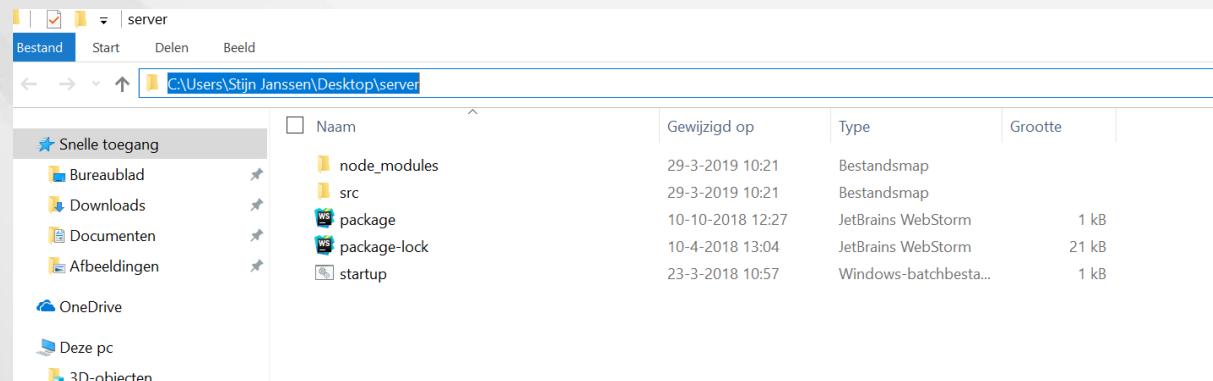
Bekijk e.v.t. de voorbeeld omgeving door:

- de file [HTTP_server.zip](#) te downloaden; Dit is een NodeJS server en een HTML bestand.
- Dan mag je naar de [NodeJS website](#) en daar de LTS versie van NodeJS downloaden. Dit hebben wij nodig om onze oefen server te kunnen draaien.
- Doorloop de installatie van de NodeJS software (volgende, volgende, enz).

HTTP requests en responses

(vervolg):

- Pak nu het gedownloadde .zip bestand uit naar een directory waar je goed bij kunt.
- Ga vervolgens naar een MS-DOS venster (cmd) door in de volgende blauwe balk op je scherm 'cmd' in te typen, gevolgd door 'enter'.

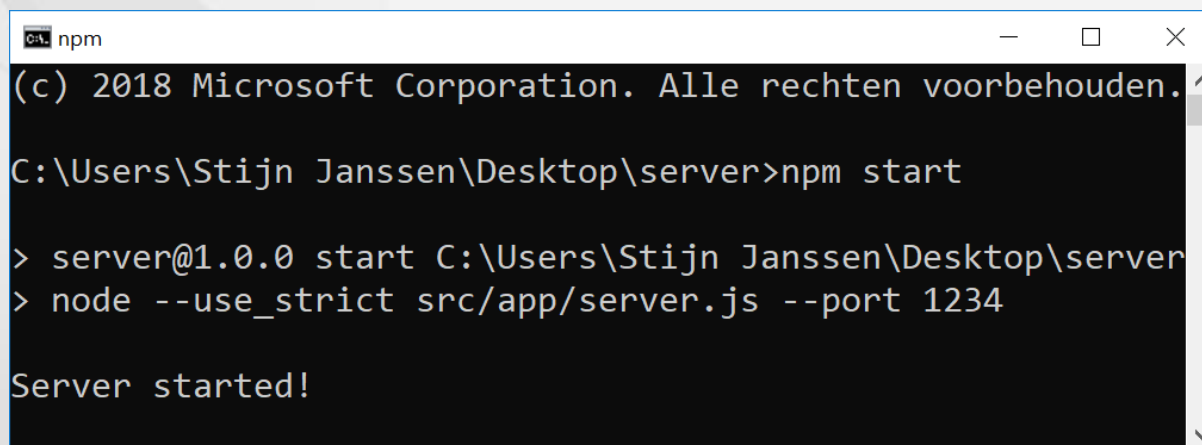


- Er is nu een MS-DOS venster geopend in deze directory

HTTP requests en responses

(vervolg):

- Geef nu het commando: `npm install`
- Wacht even ... ;)
- Geef nu het commando: `npm start`
- Je zou nu dit venster moeten zien:



```
npm
(c) 2018 Microsoft Corporation. Alle rechten voorbehouden.
C:\Users\Stijn Janssen\Desktop\server>npm start

> server@1.0.0 start C:\Users\Stijn Janssen\Desktop\server
> node --use_strict src/app/server.js --port 1234

Server started!
```

Tip: volg onze [NodeJS](#) cursus om zelf een dergelijke server te kunnen bouwen én meer.

HTTP requests en responses

(vervolg):

- Ga naar de directory `/src/app` en open daar het bestand `'testpagina.html'`. Zet de console open.
- Kijk hoe je verzoek eruit ziet. Bekijk ook de console (MS-DOS) venster voor output van de server.

HTTP berichten met de fetch API

- Afhandelen van berichten middels het XMLHttpRequest object kan vrij omslachtig / langdradig / onoverzichtelijk kan zijn
- Er zijn er tal van libraries en [wrappers](#) geschreven.
- Een JavaScript eigen feature is het gebruik van de fetch API met de fetch() methode
- Deze methode werkt met Promises.
- Dit maakt het gebruik van HTTP berichten een stuk overzichtelijker.

HTTP berichten met de fetch API

In dit codevoorbeeld reduceren wij de code van onze GET call uit vorige voorbeeld aanzienlijk:

```
fetch('http://localhost:1234/random')
  .then(data=>{
    document.getElementById("merkwijzig").value =
JSON.parse(data).merk;
    document.getElementById("typewijzig").value =
JSON.parse(data).type;
  })
);
```

HTTP berichten met de fetch API

Een POST bericht zouden we nog specifiek de methode voor moeten opgeven middels een 'options' object:

```
fetch('http://localhost:1234/versturen',  
{  
  method: "POST",  
  body: JSON.stringify(data)  
})  
.then(...);
```

Zelf een [Headers](#) object maken is ook een mogelijkheid. Dit werkt alleen bij de fetch API.

Opdrachten

Maak een eenvoudige webpagina die middels het XMLHttpRequest object alle 100 berichten van de URL: <https://jsonplaceholder.typicode.com/posts> ophaalt én op het scherm toont.

Zorg er (natuurlijk) voor dat je pagina geladen is wanneer je iets wilt tonen (bijv) middels:

```
window.onload = function(){  
    // hier komt je code  
}
```

Opdrachten

Herschrijf de vorige opgave naar het gebruik van de fetch API

Optioneel: toon de waarden in een tabel (één item per rij).

Offline applicaties

Leerdoelen

- Kunnen benoemen waarom het belangrijk is je applicatie offline beschikbaar te maken
- Kunnen tonen door code te schrijven of een applicatie online of offline is
- Kunnen benoemen van twee of meer cache gebruik overwegingen
- Kunnen schrijven van code dat gegevens uitleest en wegschrijft naar de localStorage
- Kunnen maken en uitlezen van gegevens van een IndexedDB

Praktijk toepassing

- Kunnen plaatsen van items in je winkelwagen, terwijl er even geen Internet verbinding is.
- Gebruik van iets oudere data (in cache) wanneer de Internet verbinding is weggevallen en de data niet héél actueel hoeft te zijn.
- Kunnen tonen van een keurige melding wanneer verkeersinformatie niet actueel is, gezien er een probleem lijkt te zijn met de Internet verbinding



Inhoud hoofdstuk

- onLine property
- Caching
- Cookies
- Session- en localStorage
- IndexedDB

onLine property

- We kunnen in JavaScript kijken of de browser waar we in draaien een Internet connectie heeft.
- Middels de *navigator.onLine* eigenschap.
- Is 'true' indien we connectie hebben en 'false' indien dat niet het geval is.

Volgens het MDN:

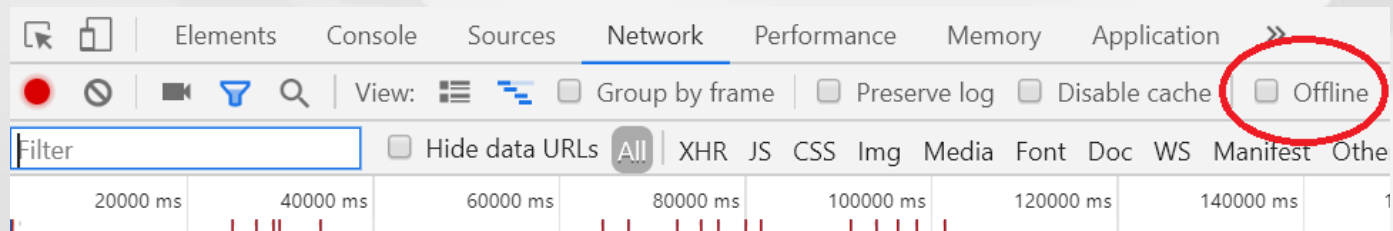
"The navigator.onLine attribute must return false if the user agent will not contact the network when the user follows links or when a script requests a remote page (or knows that such an attempt would fail)..."

onLine property

- Vaak wordt de online en offline events afgevangen, die afgaan bij wijzigen van deze property:

Te zien op: <https://codepen.io/5hart/embed/VNJeez>

Het simuleren van bovenstaand voorbeeld kan hier:



Caching

- Caching is het principe waar bestanden van de server op het device (pc, laptop, tablet, mobiel) van de client worden opgeslagen.
- Hierdoor hoeven deze bestanden niet meer gedownload te worden
- kunnen ze direct van de harde schijf van de client worden ingeladen.

We doorlopen:

- een aantal belangrijke overwegingen
- caching met behulp van het cache manifest
- caching middels de cache API in een PWA.

Cache overwegingen

- Zal de webpagina veel via mobiele devices benaderd gaan worden, óók via de wat langzamere (tot 3G) verbindingen, dan is meer cachen niet per definitie beter
- Wat te denken van een pagina met veel scripts; dienen deze direct geladen te worden of pas als de pagina al bruikbaar is voor de eindgebruiker?
- Dienen alle afbeeldingen van een foto album al direct ingeladen te worden?
- Zijn er afbeeldingen, stukken JavaScript of CSS code die pas hoeven ingeladen te worden na het optreden van een event?
- In hoeverre dient je pagina functioneel te zijn wanneer de netwerk connectie wegvalt?
- Welke zaken zijn het belangrijkste om zo spoedig mogelijk ingeladen te zijn bij een vervolg bezoek?

Cache Manifest

Om bestanden te cachen wordt vaak een appCache manifest bestand gebruikt volgens een opbouw zoals deze, échter sinds de komst van Service Workers is deze file [DEPRECATED!](#)

CACHE MANIFEST

Gemaakt door Vijfhart-IT

Commentaar regels beginnen met een hash-teken

CACHE:

index.html

images/groot_plaatje.jpg

scripts/jquery.js

NETWORK:

database_verzoeken.php

FALLBACK:

Index.html Index_fb.html

Tot slot is er ook nog de mogelijkheid via HTTP Headers te [cachen](#).

Cache API

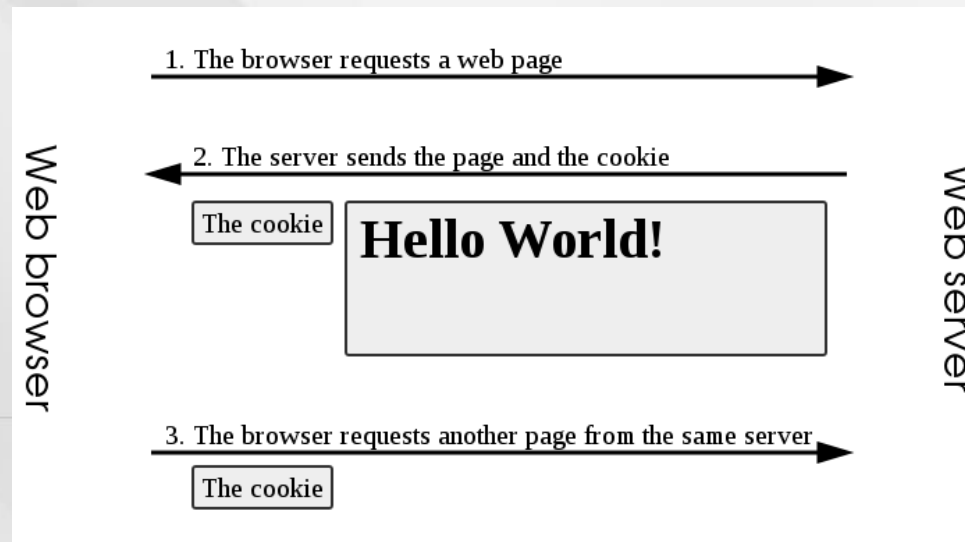
- Bij gebruik appCache Manifest én geregistreerde service worker heeft, zal de service worker het cachen regelen middels de Cache API.
- Deze API is zowel te gebruiken binnen een gewoon browser tabblad (en het window object) én in Service Workers.
- Tijd over? Video van Google over het werken met de cache API: [Caching files with the Service Worker](#)
- Ben je op zoek naar goede strategieën voor de implementatie voor het cachen van bestanden, dan is [PWA Development by Example](#) een aanrader!

Cache API

- Bij gebruik appCache Manifest én geregistreeerde service worker heeft, zal de service worker het cachen regelen middels de Cache API.
- Tijd over? Video van Google over het werken met de cache API: [Caching files with the Service Worker](#)
- Ben je op zoek naar goede strategieën voor de implementatie voor het cachen van bestanden, dan is [PWA Development by Example](#) een aanrader!

Cookies

- Middels een 'cookie' kunnen webserverns een stukje data opslaan op de pc van de client.
- Deze cookie kan bij een volgend bezoek weer uitgelezen worden.
- Schematisch is dat als volgt weer te geven (bron: [WikiCommons](#)):



Cookies

- Inhoudelijk (nog zonder enige opties) kan een cookie er als volgt uit zien:

```
SPORT=Tennis;  
Path=/sports  
HttpOnly
```

- Vaak zullen er attributen worden meegegeven die de cookie verder specificeren

Cookies

De voornaamste attributen zijn:

- **Expires:** zorgt ervoor dat de cookie blijvend (persistent) is of een session cookie is (tijdelijk, enkel in geheugen ingeladen).
- **Path:** cookie alleen gebruiken als de betreffende HTML pagina('s) in dit pad zitten.
- **HttpOnly:** deze cookie mag enkel vanuit de server uitgelezen worden en niet via JavaScript. Dit maakt hem wat veiliger tegen [cross-site scripting](#).
- **Secure:** deze cookie kan alleen via een HTTPS verbinding verstuurd worden, wat hem dus ook veiliger maakt tegen [netwerk sniffers](#).

Cookies

Zodra een gebruiker gegevens van een web service heeft gedownload, kan de server bijvoorbeeld een HTTP header sturen met deze inhoud:

```
Set-Cookie: FAVMOVIE=Matrix; Path=/movies;  
Expires=Sun, 30 Aug 2020 22:22:22 GMT;  
Secure; HttpOnly
```

Deze code beschrijft een cookie met als naam: 'FAVMOVIE' en als waarde: 'Matrix'.

Meer over Path:

<https://flaviocopes.com/cookies/#set-a-cookie-path>

Cookies

- Een cookie kan 4096 bytes / 4 kb aan gegevens opslaan.
- Daarnaast mogen er maximaal 20 cookies per domein worden opgeslagen.
- Alleen een server vanuit de oorspronkelijk zelfde afkomst kan weer dezelfde cookie uitlezen bij een vervolgbezoek.
- Hier maken reclamebedrijven gretig gebruik van door cookies van hun afkomst op verschillende websites te plaatsen, waardoor ze precies zien op welke sites een gebruiker is geweest.

SessionStorage en LocalStorage

- We kunnen gegevens op de lokale computer van de clients persisteren (wegschrijven en behouden).
- Door gebruik te maken van sessionStorage of localStorage.
- Dit zijn twee manieren van opslaan van gegevens die we sinds de komst van HTML 5 kunnen gebruiken.

SessionStorage en LocalStorage

Praktijk toepasbaarheid:

- Opslaan van de inhoud van een winkelwagen voor later gebruik
- Bewaren van gebruiker specifieke wensen zoals kleur en taal instellingen
- Opslaan van meest recent opgehaalde zoekopdrachten ten behoeve van performance

SessionStorage

- Session storage is lokale opslag, die verloren gaat zodra de eindgebruiker de browser afsluit.
- Voordeel van deze manier van opslag is dat er geen (eventueel) gevoelige data op de computer van de eindgebruiker blijft opgeslagen nadat de webbrowser is afgesloten.

LocalStorage

- LocalStorage lokale opslag, blijft behouden wanneer de browser wordt afgesloten en/of heropend.
- Voordeel is dat we bijvoorbeeld gebruikersprofielen (instellingen, winkelwagentjes et cetera) kunnen behouden op de computer
- Zelfs nog beschikbaar als de webbrowser afgesloten is geweest en de gebruiker (bijvoorbeeld na 2 weken) weer verder wilt shoppen met hetzelfde winkelwagentje.

SessionStorage en LocalStorage

- Via de de respectievelijke eigenschappen (zijnde objecten) *window.sessionStorage* en *window.localStorage* kunnen wij gebruik maken van beide opslag methoden.
- Bekijk een toelichting op deze Session Storage en Local Storage concepten:

<https://youtu.be/kLLMeL7I4O0>

- Voor beide storage vormen zijn er een tweetal functies geschreven die wij op het sessionStorage en localStorage object kunnen aanroepen.

SessionStorage en LocalStorage

In onderstaand voorbeeld gaan wij deze twee functies, `setItem()` en `getItem()` toepassen:

<https://codepen.io/5hart/embed/zpdgEj>

- Beide functies accepteren parameters.
- Dit zijn in het geval van de `setItem()` functie twee parameters: de key en de value.
- In het geval van de `getItem()` functie (dus om een waarde uit de storage te halen), hebben wij alleen een key nodig. Daarmee wordt voor ons de bijbehorende value opgehaald.

SessionStorage en LocalStorage

- Om af te vangen of er (door een bepaalde taak) een item is aangepast in één van beide storage mechanismen, kunnen wij gebruik maken van het `window.onstorage` event.
- Daar zullen wij dan een [event listener](#) voor moeten schrijven. bekijk het volgende voorbeeld:

<https://codepen.io/5hart/embed/yKgGxx>

SessionStorage en LocalStorage

Een aantal verschillen tussen cookies en het localStorage concept worden onderstaand uitgelegd:

<https://youtu.be/5ttpghXjG0g> (door: [DrapsTV](#)).

(bonus)

IndexedDB

- In iedere moderne browser zit NoSQL database
- Gebruiken dan geen taal als SQL
- Middels een JavaScript API praten we met de database
- Deze specifieke database heet een IndexedDB.
- Voornaamste eigenschap is dat we hier objecten in opslaan.
- We kunnen zelf een database maken of verder gaan met een bestaande (die hoort bij dit domein)

IndexedDB

- Het uitlezen, schrijven en verwijderen van regegevens respectievelijk van, naar en uit de database gebeurt middels `get()`, `add()` en `delete()` methoden die we op een collectie aanroepen.
- Voor het werken met IndexedDB databases zouden we ook een library kunnen gebruiken (op basis van Promises ipv callback functies):

<https://github.com/jakearchibald/indexeddb-promised>

IndexedDB

- Alle lees en schrijf acties die wij op de database willen uitvoeren verlopen via transacties.
- Dus wanneer we bijvoorbeeld gegevens willen uitlezen, zullen we eerst een transactie moeten aanmaken.
- Bij het aanmaken van een transactie, dienen we te zowel de collectie op te geven waarmee wij willen gaan werken, en het type transactie (readonly / readwrite).

IndexedDB

```
let transactieSpace = db.transaction(["series"],  
  "readwrite");  
let collectie =  
  transactieSpace.objectStore("series");
```

- We zien hier een variabele *transactieSpace* die wordt gevuld met het resultaat van het aanmaken van een transactie.
- Daarna vullen we de *collectie* variabele met een verwijzing naar onze Object Store (de naam zegt wat het doet). Deze laatst genoemde is ook de collectie waarop we straks mutaties kunnen gaan uitvoeren.
- Een transactie wordt overigens automatisch in een aparte 'thread' gedraaid op de processor.

IndexedDB

Onderstaand een CRUD:

<https://codepen.io/5hart/embed/zLWvZa>

- Navigatie en manipulatie door collecties is ook mogelijk middels zogeheten *Cursoren*.
- Het resultaat wordt asynchroon opgehaald en er is geen limiet aan het aantal cursoren dat je wilt gebruiken in je applicatie.
- Lees [hier](#) meer over het gebruik van cursoren.

IndexedDB

Onderstaand een CRUD demo:

<https://codepen.io/5hart/embed/zLWvZa>

- Navigatie en manipulatie door collecties is ook mogelijk middels zogeheten *Cursoren*.
- Het resultaat wordt asynchroon opgehaald en er is geen limiet aan het aantal cursoren dat je wilt gebruiken in je applicatie.
- Lees [hier](#) meer over het gebruik van cursoren.

- Bonus: **FILMPJE OVER IndexedDB**

Opdrachten

Gebruik de opgave van het hoofdstuk 'HTTP' om de opgehaalde data, na druk op een zelf gemaakte knop, toe te voegen aan de 'localStorage'.

Optioneel: maak ook een knop die de data toevoegt aan een indexedDB.

Lazy Loading

Leerdoelen

- Kunnen benoemen van de voordelen van lazy loading
- Kunnen benoemen van de verschillende manieren van lazy loading
- Een afbeelding pas inladen zodra deze volledig in beeld is

Praktijk toepassing

- Je een verslag van een event hebt gedaan en de foto's ervan tussen de tekst door moeten verschijnen, maar bij laden van de pagina nog niet direct zichtbaar hoeven te zijn.
- Detail foto's van een vakantie bestemming wilt tonen in een zogeheten carousel
- De uploads- of profielpagina van een bekende Youtube vlogger wilt bekijken, waarbij je nog niet wilt dat alle video's al ingeladen worden.
- Een foto album hebt gemaakt, waarop dus opsommingen van afbeeldingen of andere bronnen staan, waarvan er telkens maar enkelen in beeld zijn en de rest pas zichtbaar wordt na scrollen.

Inhoud hoofdstuk

- Lazy Loading (algemeen)
- Visibility bepalen
- Performance

Lazy Loading

Toepasbaar?

- Wanneer we een pagina hebben gemaakt waar veel afbeeldingen, video's of andere externe bronnen worden ingeladen
- Deze bronnen niet direct in volledige glorie zichtbaar hoeven te zijn op de pagina
- Dan kunnen we gebruik maken van een concept genaamd lazy loading.
- We kunnen een placeholder voor een afbeelding of video tonen totdat hij is ingeladen. Het uiteindelijke object kan dan later ingeladen worden.

Lazy Loading

Er zijn verschillende mogelijkheden om je content later dan volgens de gewone flow in te laden.

- Dit kan redelijk statisch door CSS `<style>` en JavaScript `<script>` codeblokken verder naar beneden te plaatsen op je HTML pagina
- Of dynamisch, door vanuit JavaScript code je afbeeldingen e.d. te voorzien van inhoud (o.a. middels `src` en `data-` attribuut).

Bekijk voor de eerste optie het volgende voorbeeld met dummy data:

<https://codepen.io/5hart/pen/xeNGqQ>

Lazy Loading

Overweging

Als we het hebben over nog niet zichtbare objecten, is het vanuit performance én esthetisch oogpunt (responsiveness en aanzicht), aan te raden de afbeeldingen (en andere grote objecten) die nog niet zichtbaar zijn tijdelijk te vervangen of voor alsnog geheel weg te laten.

Lazy Loading

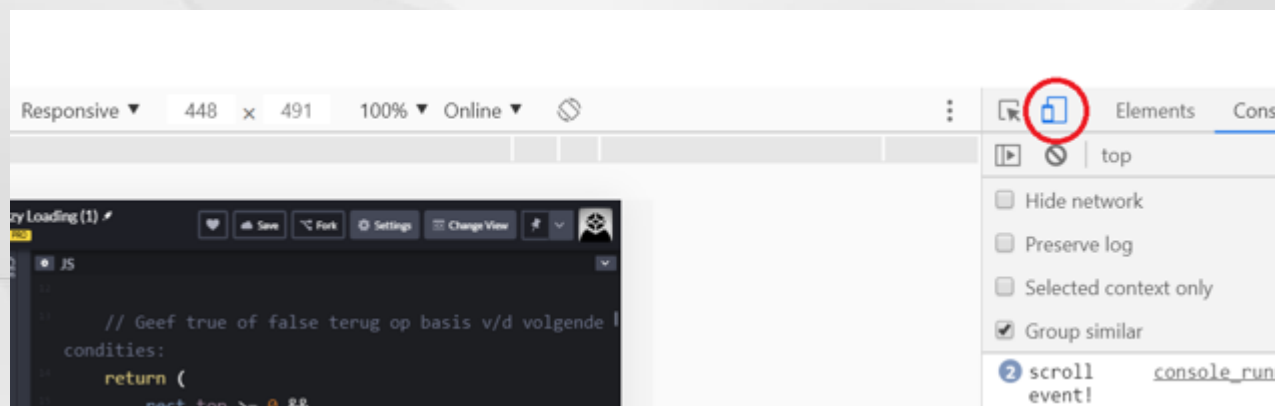
In volgorde van minst naar meest positieve effect op de performance van je pagina:

- Niet zichtbare afbeeldingen zwart-wit placeholders inladen
- Niet zichtbare afbeeldingen 'blurred' placeholders inladen
- Niet zichtbare afbeeldingen miniaturen inladen
- Niet zichtbare afbeeldingen alternatieve tekst inladen

Visibility bepalen

Één van de voornaamste technieken die je zult toepassen om lazy loading te realiseren, is het bepalen van welke elementen (afbeeldingen e.d.) er momenteel in de viewport (het zichtbare deel van de pagina) zitten van de gebruiker.

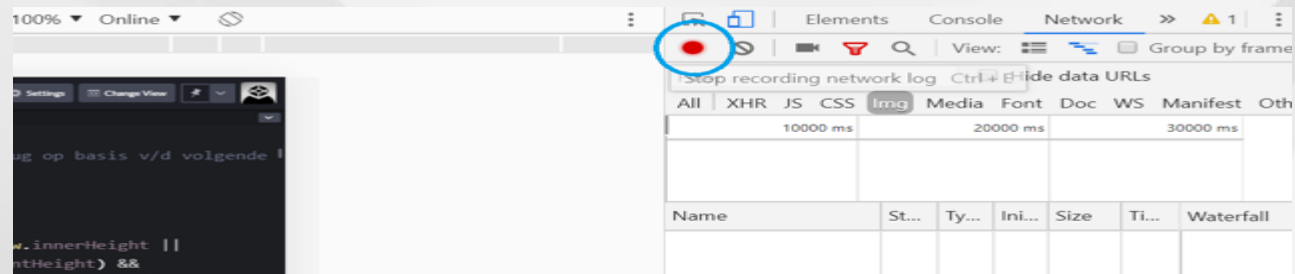
Schakel over naar 'mobiele modus' om te zien hoe ver een afbeelding in beeld (in de viewport) is:



Visibility bepalen

Voorbeeld: <https://codepen.io/5hart/pen/JVqjBR/>

- Hoe kun je nu testen of er inderdaad pas zodra een afbeelding in beeld is, deze gedownload wordt?
- Bekijk het tabblad 'Network' in je Chrome Developer Tools:

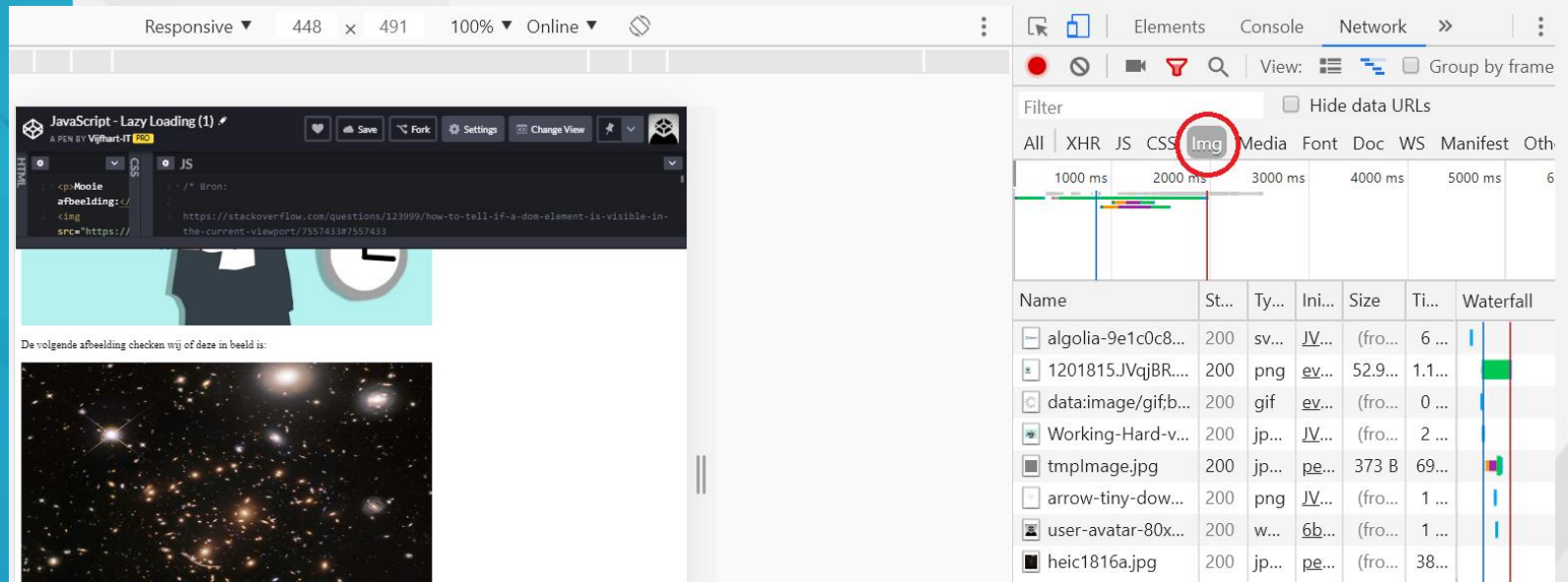


Verder...

Visibility bepalen

- Zorg dat de 'Record' knop aan staat en dus rood gekleurd is.
- Zorg ervoor dat de CodePen pagina minder ruimte in beslag neemt.
- Scroll naar beneden in de webpagina, zodat een afbeelding (behalve de eerste) in beeld is.
- Bekijk de uitgaande verzoeken voor afbeeldingen aan de rechterzijde, eventueel gefilterd op afbeeldingen

Visibility bepalen



The screenshot shows a web browser window with a JavaScript Lazy Loading demo. The demo includes a code editor with the following JavaScript code:

```

  <script>
    (function() {
      // Bron:
      // https://stackoverflow.com/questions/123999/how-to-tell-if-a-dom-element-is-visible-in-the-current-viewport/755743387557433
    })();
  </script>
  
  </script>
  </body>
  </html>
  
```

The demo text says: "De volgende afbeelding checken wij of deze in beeld is:" followed by a large image of a galaxy cluster.

The Network tab is open, showing a list of resources. The 'img' filter is selected, and the following resources are listed:

Name	St...	Ty...	Ini...	Size	Ti...	Waterfall
algolia-9e1c0c8...	200	sv...	<u>IV</u> ...	(fro...	6 ...	
1201815.JVqjBR...	200	png	<u>ev</u> ...	52.9...	1.1...	
data:image/gif;b...	200	gif	<u>ev</u> ...	(fro...	0 ...	
Working-Hard-v...	200	jp...	<u>IV</u> ...	(fro...	2 ...	
tmplImage.jpg	200	jp...	<u>pe</u> ...	373 B	69...	
arrow-tiny-dow...	200	png	<u>IV</u> ...	(fro...	1 ...	
user-avatar-80x...	200	w...	<u>6b</u> ...	(fro...	1 ...	
heic1816a.jpg	200	jp...	<u>pe</u> ...	(fro...	38...	

Visibility bepalen

- We hebben in de code een array gemaakt met daarin alle afbeeldingen van de pagina die géén class genaamd 'alwaysShow' hebben
- *Die* specifieke afbeeldingen willen wij altijd direct geladen hebben in ons voorbeeld.
- We hebben een functie die bekijkt of een opgegeven element geheel binnen de huidige viewport valt.
- Deze functie gebruikt o.a. `getBoundingClientRect()` voor de kaders van het element, `window.innerHeight` of `document.documentElement.clientHeight` en de gelijknamige *Width* varianten voor het bepalen van de viewport.

Visibility bepalen

- We vangen in onze code een 'scroll' event af en doorlopen dan telkens alle afbeeldingen.
- Dit werkt prima voor demonstratie doeleinden en kleinere webpagina's, maar niet voor webpagina's met vele grote afbeeldingen.
- Een dergelijke situatie zou bijvoorbeeld beter te behandelen zijn met een Promise.

Visibility bepalen

- Zo is een alternatief voor het bepalen of een item op je webpagina 'in beeld' / in je viewport zit, het gebruik van een IntersectionObserver:

<https://developers.google.com/web/updates/2016/04/intersectionobserver>

- Voorbeeld van bovenstaande IntersectionObserver:

<https://codepen.io/stijnjanssen/pen/XWWawXb>

Opdrachten

- Maak een pagina met daarop bovenin beeld een div met tekst.
- Deze div mag behoorlijk groot zijn (lees: het hele verticale scherm vullen).
- Plaats onder deze div een afbeelding, die pas ingeladen wordt zodra de afbeelding 50% of geheel in de viewport is.
- Dit test je dus door het 'Network' tabblad in de Chrome Developer Tools open te zetten en te kijken wanneer de juiste afbeelding gedownload wordt.

Web Sockets

Leerdoelen

- Het kunnen benoemen van één of meer toepassingen van Web Sockets
- Het kunnen opzetten van een eenvoudige Web Socket aan de client kant

Praktijk toepassing

- Het maken van een online chat applicatie
- Webbased gamen over het Internet
- Omgevingen waar heel veel (te veel?) HTTP verzoeken gedaan moeten worden

Inhoud hoofdstuk

- Web Sockets algemeen

Web Sockets

- Met WebSockets is het mogelijk een communicatielijn open te zetten van een client browser naar een server.
- Via deze full-duplex lijn kunnen we data heen en weer sturen.
- We zitten niet met het probleem dat er veel HTTP verzoeken tussen client en server verstuurd moeten worden, die zwaar zijn voor de pagina en dus de ervaring van de eindgebruiker verslechteren.
- De afhandeling van het versturen, ontvangen en de tussentijdse status daarvan is geregeld middels JavaScript events.

Web Sockets

- Het aanmaken van een WebSocket object gaan wij verderop in het hoofdstuk doen.
- Om dit te kunnen testen dient er een server beschikbaar te zijn die onze connectie accepteert.
- Dat kan een Node.js server zijn of bijvoorbeeld een PHP server.
- Ook dient de server een WebSocket omgeving te hebben draaien.

Web Sockets

- Het maken van een verbinding met een WebSocket server doen we vanaf de client middels de volgende JavaScript code:

```
let socket = new WebSocket('wss://echo.websocket.org');
```

- We kunnen ervoor kiezen het ws of het wss protocol te gebruiken, afhankelijk van welke de server gebruikt.
WSS = secure
- Zodra de connectie gemaakt is, dienen we de 'open', 'error' en 'message' events van dit socket object af te vangen.

Web Sockets

- Een praktisch voorbeeld, waarmee we met een 'echo server' communiceren:

<https://codepen.io/5hart/embed/pBXmQw>

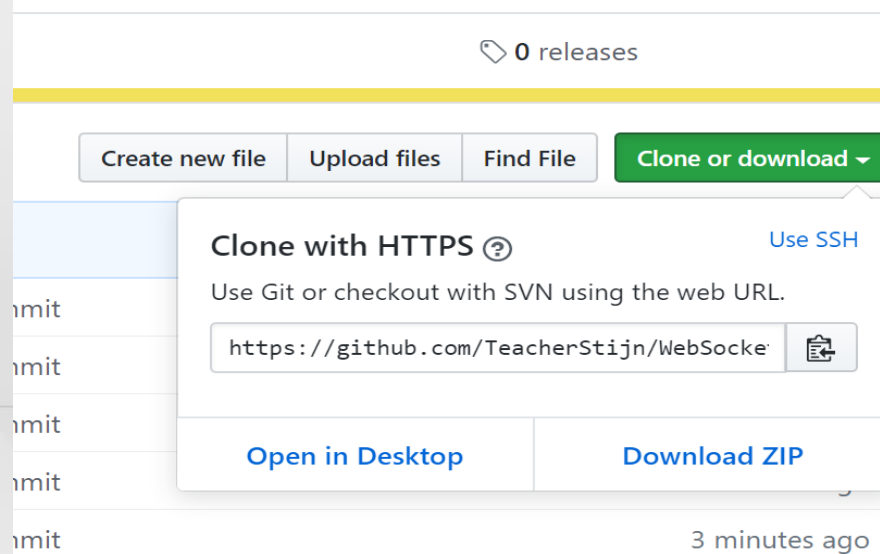
- Hier worden dus enkel onze berichten direct terug gegeven

Web Sockets

- Wil je niet gebruik maken van een bestaande 'echo-server' maar een eigen nodejs WebSocket server? Download of clone dan dit project(je!) met een lokale WebSocket server:

<https://github.com/TeacherStijn/WebSocket-test>

- Klik op de 'Clone or download' knop:



Web Sockets

- Er zijn nog legio van voorbeelden met Web Sockets te bedenken:

Game gerelateerd:

- Gegevens van speler die andere speler aanvalt (in [Street Fighter](#)).
- Chatbericht wat speler stuurt die op zoek is naar een team om mee te spelen in (in [WoW](#)).
- Gegevens van een piratenschip dat op zoek is naar vrachtschepen (in [Star Citizen](#)).

Opdrachten

- Maak een webpagina die kan communiceren met de <wss://demos.kaazing.com/echo> server, óf middels een eigen NodeJS WebSocket server (zie voorbeelden voor een link).
- Maak een invoerveld waarin een gebruiker een bericht kan typen.
- Dit bericht dient via een Web Socket verstuurd te worden en de reactie (hetzelfde?) op het scherm getoond te worden.

PWA introductie

Leerdoelen

- Drie of meer voordelen kunnen benoemen van het maken van een PWA
- Twee redenen kunnen benoemen waarom het gebruik van HTTPS goed is

Praktijk toepassing

- Webwinkel pagina als app op een mobiel device beschikbaar maken
- HTML 5 game als app beschikbaar maken
- Op afstand aansturen van informatieborden

Progressive Web Apps

Progressive web apps, ofwel PWA's, veranderen het web:

- Het wordt een *mobile first* platform
- Applicaties draaien sneller
- Het is mogelijk offline te werken
- Applicaties kunnen aan het mobiele homescreen worden toegevoegd.

Google [pleit](#) ervoor dat een PWA vier ervaringen moet aanbieden: *snelheid, betrouwbaarheid, betrokkenheid en integratie*.

Progressive Web Apps

Om een PWA te maken, of webpagina om te zetten naar een PWA, zijn er drie voornaamste vereisten:

- Er moet een Web Manifest file aanwezig zijn
- De web applicatie dient te draaien via HTTPS
- Service Worker met fetch event handler dient geregistreerd te worden

Web Manifest

- Een web manifest bestand zorgt er voornamelijk voor dat je website toegevoegd kan worden aan het homescreen van een mobiel device
- Is een JSON bestand en dus opgebouw uit *key-value* paren, beiden met dubbele quotes "" omringd en voor objecten gebruik gemaakt van accolades {}.

HTTPS gebruik

- Een PWA heeft zekerheid nodig dat er niet met de connectie geknoeid kan worden; hij moet kunnen garanderen dat de data die verstuurd wordt correct is.
- Nog een aantal voordelen voor het gebruik van HTTPS zijn: identiteit, veiligheid, integriteit, SEO optimalisatie, Wep API ondersteuning en snelheid i.c.m. HTTP/2.

HTTPS gebruik

- Zorgen voor een HTTPS hosting vereist een SSL certificaat.
- Via [Google](#) kom je al redelijk snel 'self-signed' certificaten tegen, maar dat werkt niet goed genoeg als test omgeving voor een PWA; de zogeheten Service Workers kunnen zich daar niet goed op registreren.
- In de praktijk zullen we een echt SSL certificaat actief moeten hebben op de server.
- Gebruik voor testdoeleinden gratis hosting met SSL, zoals bij: [Github Pages](#).

Service Workers

- JavaScript code bedoeld om betere gebruikerservaring te leveren.
- Is een tussenlaag (soort proxy) tussen de grafische interface van de bezoeker en de server
- Biedt mogelijkheden voor:
 - Offline gebruik, middels caching en HTTP interception
 - Push notificaties
- Vele andere mooie mogelijkheden vind je [hier](#)

Service Workers

Het kunnen gebruiken van een SW hangt samen met:

- navigator.serviceWorker [ondersteuning](#)
- het [gebruik](#) van promises
- de mogelijkheid tot [gebruik](#) van de fetch API
- Actief zijn van HTTPS

Progressive Web Apps

- Meer over weten?
- Kom naar onze PWA Workshop dag en maak **zelf** een progressive web app!

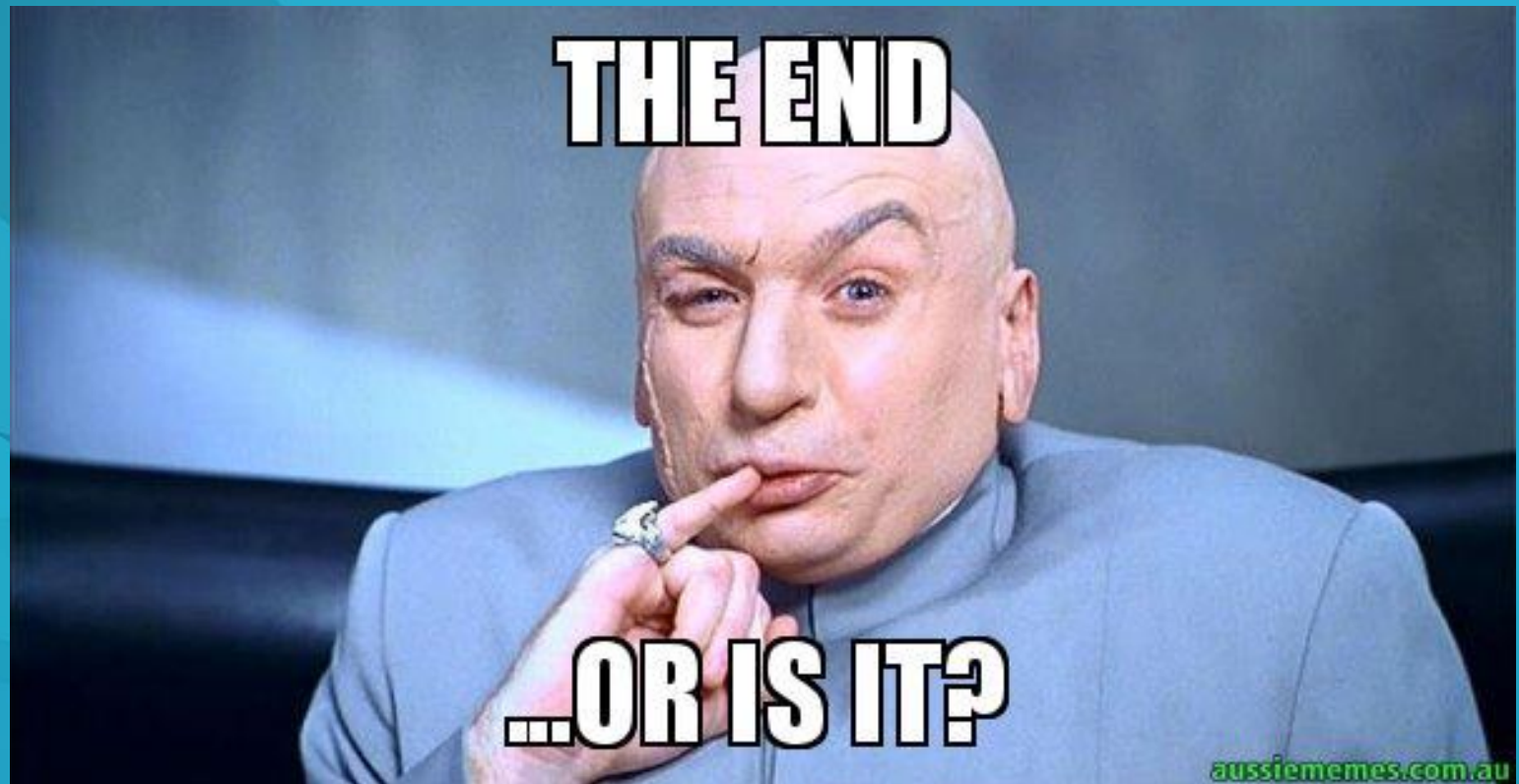
Progressive Web Apps

Tijd over?

1) Maak een eigen [Github pagina](#) aan en maak daar een '[GitHub Pages](#)' pagina van zodat je er middels <https://jouwnaam.github.io>

2) Maak voor zover mogelijk een eigen PWA van deze webpagina. Dus:

- Zorg voor een index.html pagina
- Zorg voor een Web Manifest bestand en koppel deze
- Zorg voor een Service Worker die zich kan registreren



Dank voor jullie aandacht!
(en succes met het toepassen!)