

JavaScript Advanced

Classes en overerving

Classes

Leerdoelen

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

Met de komst van ES6 kunnen we klassen definiëren zoals dat ook in veel object georiënteerde programmeertalen kan. Echter, het is een schil om bestaande ‘mogelijkheden’ heen. Het is dus niet ‘echt’. Binnen JavaScript gebruikten we voorheen constructor functions en prototype overerving. Dit kan nog steeds, maar de classes in JavaScript zijn een goede stap in de toekomst wat betreft leesbaarheid, consistentie over programmeertalen (emuleren van klassen) en gebruik van this, maar heeft ook wat nadelen. Deze nadelen hebben vooral te maken met wanneer we gebruik maken van het zogeheten ‘Factory design pattern’ waar we verderop in dit hoofdstuk meer over gaan zien en met het feit dat wij properties niet ‘private’ kunnen maken.

Praktijk toepassing

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

➔ Lees eerst pagina 102; ‘Class Notation’ uit het boek.

```
<iframe height="271" style="width: 100%;" scrolling="no" title="JavaScript - klassen - basic"
src="//codepen.io/5hart/embed/LaKoYN/?height=271&theme-id=29061&default-
tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - klassen - basic by Vijfhart-IT

(@5hart) on CodePen.

```
</iframe>
```

We zien in het vorige voorbeeld dat een klasse wordt gedeclareerd m.b.v. het sleutelwoord ‘class’ gevolgd door een set accolades {} om de body (beschrijving) van de klasse in te plaatsen.

Verder zien we dat we een constructor gebruiken. Dit is het stukje code dat het daadwerkelijke object maakt van de klasse (=blauwdruk!) dat wordt gedraaid wanneer we ‘new Tafel()’ draaien.

Probeer dit zelf uit door het bovenstaande code voorbeeld te ‘[forken](#)’ naar je eigen CodePen account. Je krijgt dan een kopie, die je vrij kunt aanpassen én opslaan.

Is het gelukt?

Goed gebruik is om de class notatie te gebruiken, ook voor het declareren van gedrag (functies) binnen een klasse. De volgende oudere notatie is uiteraard ook nog steeds te gebruiken:

```
[code]
let t = new Tafel();
t.verf = function(){ .. }
```

of met functie hergebruik:

```
t.prototype.verf = function() { .. }
```

```
[/code]
```

Ook het maken van een tafel als 'object literal' kan natuurlijk, maar dan gaan we voorbij aan het concept van hergebruik:

```
[code]
let t = {
  lengte: 10,
  breedte: 10,
  kleur: 'groen',
  prijs: 500
}
[/code]
```

Classes in JavaScript bieden ons helaas *niet* de mogelijkheid om gebruik te maken van bijvoorbeeld access modifiers zoals in bijvoorbeeld Java. We kunnen een variabele bijvoorbeeld nog steeds niet 'private' maken in JavaScript. De enige omweg om dat te doen, is door een variabele af te screenen, door hem in een zogeheten 'closure' te plaatsen.

Wil je meer weten over wat classes precies zijn, bekijk dan de volgende video van [FunFunFunction](#):

```
<iframe width="560" height="315"
src="https://www.youtube.com/embed/Tllw4EPHliQ?start=1122" frameborder="0"
allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>
```

In JavaScript kunnen we middels getters en setters waarden instellen en uitlezen van een object. Hier was één manier en zijn inmiddels twee manieren voor. Wanneer je wilt dat objecten die al bestaan óók aangepast worden, kun je het woord 'get' of 'set' plaatsen voor de naam van een methode. Deze methoden kunnen dan gebruikt worden als zijnde properties!

-> Bekijk het voorbeeld op pagina 110 van het boek.

De andere manier, waarbij bestaande objecten *niet* aangepast worden is door gebruik te maken van Object.defineProperty() zoals in het volgende voorbeeld te zien is:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - klassen (old) -
```

```
defineProperty" src="//codepen.io/5hart/embed/KEjEbW/?height=300&theme-id=29061&default-  
tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - klassen (old) -
defineProperty by Vijfhart-IT

(@5hart) on CodePen.
</iframe>

Dit voorbeeld maakte nog niet gebruik van classes. Uiteraard kan dat ook, op deze manier:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="drBENV"  
src="//codepen.io/5hart/embed/drBENV/?height=300&theme-id=29061&default-  
tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen drBENV by Vijfhart-IT

(@5hart) on CodePen.
</iframe>

Houdt dus wel rekening mee, dat de defineProperty() methode vooral handig is als er al objecten
zijn, waarvan je wilt dat die niet worden aangetast.

-> Lees voor meer informatie en voorbeelden pagina 109 en 110; 'Getters, Setters and Statics' uit het
boek.

Overerving

Binnen de nieuwe schrijfwijze, gebruikmakend van classes kunnen we uiteraard ook werken met
overerving.

Bekijk de volgende twee voorbeelden voor het maken van een klasse, gevolgd door een voorbeeld
over overerving.

```
<iframe height="485" style="width: 100%;" scrolling="no" title="JavaScript - klassen (1)"  
src="//codepen.io/5hart/embed/wObYVw/?height=485&theme-id=29061&default-  
tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - klassen (1) by
Vijfhart-IT

(@5hart) on CodePen.
</iframe>

```
<iframe height="742" style="width: 100%;" scrolling="no" title="JavaScript - klassen (2)"  
src="//codepen.io/5hart/embed/qvGQdK/?height=742&theme-id=29061&default-  
tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - klassen (2) by Vijfhart-IT

(@5hart) on CodePen.

</iframe>

We zien bij het laatste voorbeeld dus dat er een 'Bureau' klasse is gemaakt, wat een subklasse is van 'Tafel', te zien aan het sleutelwoord extends.

Static eigenschappen en methoden

We kunnen in JavaScript inmiddels ook properties en functies van een klasse als 'static' definiëren. Dit betekent dat deze horen bij de blauwdruk (klasse), in plaats van bij het object. Of in JavaScript termen: horen bij de Constructor functie en niet bij het Prototype.

Praktijk toepassing:

- Bijhouden van statistieken over objecten van een klasse, binnen de klasse zelf

We beginnen met een aantal voorbeelden om de property 'static' te verduidelijken.

Bekijk het volgende voorbeeld en probeer te achterhalen wat hier de zogeheten 'modifier' genaamd static doet:

<iframe height="744" style="width: 100%;" scrolling="no" title="JavaScript - klassen - static (1)" src="//codepen.io/5hart/embed/WXQpmw/?height=744&theme-id=29061&default-tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">

See the Pen JavaScript - klassen - static (1) by Vijfhart-IT

(@5hart) on CodePen.

</iframe>

Bekijk nu het volgende voorbeeld. Hier hebben we ook de subklasse 'Bureau' weer toegevoegd:

<iframe height="779" style="width: 100%;" scrolling="no" title="JavaScript - klassen - static en overerving (2)" src="//codepen.io/5hart/embed/XzmgKv/?height=779&theme-id=29061&default-tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">

See the Pen JavaScript - klassen - static en overerving (2) by Vijfhart-IT

(@5hart) on CodePen.

</iframe>

We zien dat we static eigenschap 'aantal' hebben gedeclareerd. Deze is gelijk voor *alle* objecten die gemaakt zijn van deze klasse (ofwel; geldt voor alle 'instanties'). Daarnaast nog een voordeel: er

hoeft daadwerkelijk nog *geen* object te zijn gemaakt van de klasse (m.b.t. 'new Tafel()'), voordat de static eigenschap of methode gebruikt kan worden.

Voor de volledigheid, een voorbeeld met een static methode:

```
<iframe height="780" style="width: 100%;" scrolling="no" title="JavaScript - klassen - static methoden" src="//codepen.io/5hart/embed/YgbmyX/?height=780&theme-id=29061&default-tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - klassen - static methoden by Vijfhart-IT

(@5hart) on CodePen.

```
</iframe>
```

Let op: het sleutelwoord *static* heeft dus niets te maken met een 'vaste' waarde. Hiervoor gebruiken we *const* in variabele declaraties. In classes zou je daarvoor [Object.defineProperty\(\)](#) weer kunnen gebruiken, óf werken met modules (wat in een later hoofdstuk aan bod komt).

Interfaces en abstracte klassen

Deze bestaan in JavaScript niet. Deze zijn bedoelt om regels in te stellen; welke eigenschappen moet een object hebben bijvoorbeeld. Maar omdat JavaScript erg los geschreven is, zou je dan alsnog alle eigenschappen van het object kunnen aanpassen (toevoegen, wijzigen en verwijderen), bewust of onbewust! Een interface zal dus in JavaScript ook geen zin hebben.

Abstracte klassen worden in programmeertalen vaak gebruikt om overeenkomende code vast te leggen in een superklasse die zelf *niet* geïnstantieerd (object van gemaakt) mag worden. In JavaScript (of ES6) bestaan abstracte klassen (voor alsnog) niet. Een mogelijke workaround is in de constructor van een klasse afvangen of de constructor van dit huidige object deze klasse is. Dat kan op deze manier:

```
<iframe height="465" style="width: 100%;" scrolling="no" title="JavaScript - klassen - abstract" src="//codepen.io/5hart/embed/rRgXNq/?height=465&theme-id=29061&default-tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - klassen - abstract by Vijfhart-IT

(@5hart) on CodePen.

```
</iframe>
```

Factory design pattern

Het gebruik van klassen in JavaScript biedt zeker niet alleen maar voordelen, maar is een stap in de goede richting. We kunnen tegen problemen aanlopen bij bestaande code die we willen re-factoren en gebruik maakt van 'factories' middels het 'factory design pattern'. Dit is een patroon wat een nieuwe instantie (object) van een klasse (of ouderwets: constructor functie) oplevert middels de aanroep van een methode, zonder het woord 'new' dus.

Praktijk toepassing

- Afsluiten van een nieuw verzekeringsobject waarbij er aan bepaalde condities moet worden voldaan, voordat er een instantie kan worden uitgegeven.
- Maken van een database connectie als instantie, waarbij achter de schermen wordt geregeld hoeveel connecties er zijn en/of er eerst andere connecties moeten worden gesloten.

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - Factory Pattern (latest)"
src="//codepen.io/5hart/embed/aMggMj/?height=300&theme-id=29061&default-
tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - Factory Pattern (latest) by Vijfhart-IT

(@5hart) on CodePen.

```
</iframe>
```

We zien hierin dat we binnen de Object Literal notatie, de object notatie met accolades, nu overal het keyword 'function' hebben weggelaten waar het om een functie gaat. Dit is optioneel. Gebruik is om bij instantiëren een hoofdletter voor de klassenaam te gebruiken, daarom hebben wij ook gekozen voor een hoofdletter bij de Game(genre) functie.

Oude manier ter referentie: hier [hier](#).

JSON

Leerdoelen

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

JSON is sinds enige jaren *sterk* in opkomst als schrijfwijze voor het uitwisselen van gegevens. Omdat we werken met veel verschillende systemen, verschillende programmeertalen, verschillende inzichten en verwachtingen, is er al sinds jaar en dag behoefte aan consistentie. Hiervoor is in midden jaren 90 de taal XML bedacht. Inmiddels zijn we veel jaren verder en is er vooral veel gebeurd in de web hoek. Vooral JavaScript is 'gelanceerd' door o.a. de libraries [jQuery](#), [NodeJS](#) en vele andere.

Het bleek het ook erg handig als er een standaard uitwisselingsformaat kon zijn, die nog minder code kostte, veel overeenkomsten had met de taal JavaScript en wel ten koste ging van validatiemogelijkheden zoals deze er zijn in XML-land ([XML schema's](#)). JSON is begin van dit decennium opgezet op een basis van ES3 ('99).

Praktijk toepassing

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

Een JSON tekst wordt opgeslagen in een bestand met de extensie .json en een vaak ingesteld **MIME-type** van 'application/json'.

Inmiddels wordt JSON (of afgeleiden ervan) ook als formaat toegepast binnen NoSQL databases (zoals [MongoDB](#)), waarin gegevens niet relationeel middels tabellen, maar als objecten en collecties worden opgeslagen.

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

In JavaScript en veel andere talen zitten mogelijkheden om JSON (wat essentieel platte tekst is), om te zetten naar JavaScript objecten en andersom. Hierna volgt een klein voorbeeld:

```
<iframe height="265" style="width: 100%;" scrolling="no" title="JSON - Hoe werkt het"
src="//codepen.io/stijnjanssen/embed/wOKgOW/?height=265&theme-id=0&default-tab=js,result"
frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JSON - Hoe werkt het by Stijn Janssen

(@stijnjanssen) on CodePen.

</iframe>

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

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

Met modules kunnen we voornamelijk functionaliteiten van onze applicatie groeperen, onder één naam onderbrengen en daarnaast de globale [scope](#) schoon houden. Eigenlijk wordt ook de werkbaarheid met JavaScript wat vergroot. Daarnaast kun je ook middels nieuwe EcmaScript modules, je afhankelijkheden duidelijk regelen, dat kan middels onderstaande weg nog niet.

Praktijk toepassing

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

Closures

Om binnen JavaScript de globale scope schoon te houden, kun je variabelen in functies plaatsen. Nadeel is natuurlijk dat zodra je deze functie hebt gedraaid, deze 'klaar' is, en de lokale variabele daarbinnen 'weg' is. Een oplossing hiervoor is het gebruik van een *closure*. Hiermee kun je een functie, met daarbinnen een lokale variabele, vaker dan eens aanroepen én behoudt de variabele zijn waarde (wordt apart gezet in het geheugen).

Belangrijk is om de functie iets te laten retourneren wat an sich weer een *functie* óf *object* is. Deze functie of object behoudt dan namelijk toegang tot de variabele(n) die in de buitenste functie gedeclareerd zijn.

```
<iframe height="470" style="width: 100%;" scrolling="no" title="Closure (2)"
src="//codepen.io/5hart/embed/XepxbV/?height=470&theme-id=29061&default-tab=js"
frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen Closure (2) by Vijfhart-IT

(@5hart) on CodePen.

```
</iframe>
```

We zien hier een BankRekening 'constructor' functie aangemaakt die een object teruggeeft (return { .. }). De eigenschappen nummer en saldo binnen de buitenste functie blijven in het geheugen staan, maar zijn niet van buitenaf te benaderen. De methoden getSaldo en getNummer zijn de toegangswegen. Één nadeel: indien je maar één BankRekening zou willen maken, en deze functie dus ook maar éénmalig zou willen uitvoeren, moeten we nog iets om de closure heen bouwen.

Als dit gesneden koek voor je is, kun je voor een nog compactere schrijfvorm kijken:

➔ In het boek op pagina 48 en 49

IIFE's

Een IIFE is een functiebeschrijving die direct uitgevoerd wordt. Om de functie zelf staan een setje haken en na de functie ook. Door deze laatste set haken wordt de beschreven functie direct uitgevoerd. Dit is handig om allerlei functies binnen de binnenste functie of object naar buiten kenbaar te maken, zónder dat de variabelen in de buitenste functie bekend worden naar buiten toe.

Een voorbeeld:

```
<iframe height="618" style="width: 100%;" scrolling="no" title="JavaScript - IIFY - Basic"
src="//codepen.io/5hart/embed/VGLBYN/?height=618&theme-id=29061&default-
tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - IIFY - Basic by Vijfhart-IT

(@5hart) on CodePen.

</iframe>

Hier zien we een persoon beheer app. Hiermee kunnen wij gegevens van een persoon wijzigen via setNaam, getNaam, setLeeftijd en getLeeftijd kunnen uitlezen en wijzigen. De eigenschappen 'naam' en 'leeftijd' blijven behouden, maar zijn *niet* te benaderen van buitenaf. Alle toegang verloopt via de hiervoor genoemde methoden.

JavaScript zelfbouw modules

In essentie is een zelf gebouwde JavaScript module een closure in combinatie met een IIFE; we willen één functie hebben die al onze functionaliteit bevat, met daarin o.a. lokale variabelen en andere functies. Ook dient deze buitenste functie direct aangeroepen te worden om direct beschikbaar te zijn.

Een voorbeeld:

```
<iframe height="265" style="width: 100%;" scrolling="no" title="Module_example"
src="//codepen.io/stijnjanssen/embed/JzYOOZ/?height=265&theme-id=0&default-tab=js"
frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen Module_example by Stijn Janssen

(@stijnjanssen) on CodePen.

</iframe>

Een ander voorbeeld van een module die gegevens ophaalt uit een Harry Potter API volgt hierna. Let op: het ophalen van de gegevens middels een HTTP verzoek hoeven wij nog niet te kunnen:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="DOM manipulatie met Harry Potter"
src="//codepen.io/5hart/embed/jJgVG/?height=300&theme-id=29061&default-
tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen DOM manipulatie met Harry Potter by Vijfhart-IT

([\) on \[.\]\(https://codepen.io\)](https://codepen.io/5hart)

</iframe>

Je hebt mogelijk al nagedacht over het feit dat afhankelijkheden naar andere stukken code, functies, niet in een module zoals bovenstaand verwerkt worden. Alle benodigdheden dienen binnen dezelfde globale scope bekend te zijn om te kunnen gebruiken. Dit maakt onze code eigenlijk weer een stuk minder *modulair*. We gaan hier in een volgende paragraaf naar kijken hoe dit op te lossen middels EcmaScript modules.

Tot slot, wil je weten hoe modules in de praktijk (zoals bij jQuery) eruit kunnen zien, bekijk dan eens de inhoud van een jquery JavaScript file via: <https://code.jquery.com/>

Meer over closures en IIFE's leert u in onze: '[Front-end development met JavaScript](#)' cursus.

EcmaScript modules

Wil je ook controle hebben over afhankelijkheden? Dan zijn EcmaScript modules mogelijk dé oplossing die je zoekt. We gebruiken binnen dit systeem:

- de sleutelwoorden 'import' en 'export' om functies, klassen of declaraties respectievelijk als verwijzing in te laden of naar buiten kenbaar te maken.
- type="module" bij de <script> tag om aan te geven dat we met een module te maken hebben.
- Normale (named) exports en default exports
- De code dient te draaien op een (al dan niet lokale) webserver(!)

Bekijk het volgende voorbeeld, opgesplitst in drie fictieve bestanden (HTML, JS en JS):

[code]

```
<head>
  <title>EcmaScript import en export voorbeeld</title>
  <!-- import character als module niet nodig, ivm import vanuit game.js -->
  <script type="module" src="game.js"></script>
</head>
```

[/code]

[code]

```
import { Character } from "../character.js";

window.onload = function() {
  let c = new Character("Unicorn");
  console.log(c.naam);
  document.getElementsByTagName("main")[0].innerHTML = c.naam;
}
```

[/code]

[code]

```
export class Character {
  constructor(naam){
```

```
        // Door set en get methode wordt nu set naam() aangeroepen!  
        this.naam = naam;  
    }  
  
    // Aanroep van karakter.naam, wordt nu vervangen door getter!  
    get naam(){  
        return this._naam;  
    }  
  
    // Aanroep van karakter.naam=waarde wordt nu vervangen door setter!  
    set naam(n){  
        this._naam = n;  
    }  
}  
[/code]
```

- ➔ Bekijk [deze pagina](#) als *zeer degelijk* stuk naslagwerk over JavaScript modules. Ook lees je in het boek meer op pagina's 173 t/m 175.

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

Het werken met asynchrone data is sinds jaren erg belangrijk in JavaScript, bijvoorbeeld alleen al het sturen en ontvangen van gegevens van een server wat asynchroon gebeurt: het is onbekend wanneer de gegevens daar aankomen én het is onbekend wanneer wij data mogen terug verwachten. We zullen alvast code moeten schrijven om rekening te houden met die situaties. Asynchrone data gaat hand-in-hand met events. Ook van events weten wij niet wanneer ze optreden. Er zullen event handlers geschreven moeten worden om het gedrag na optreden af te vangen.

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 (uiteindelijk) geheel asynchroon dient te gebeuren

Asynchrone data zorgt voor een andere manier van denken én programmeren. Eigenlijk zelfs voor veranderingen in programmeertalen in algemene zin.

Het gebruik van Promises maakt het mogelijk om code te schrijven voor de ‘goed’ en de ‘niet gelukt/fout’ situaties middels een `.then()` en een `.catch()` methode.

Stel je voor dat we het boek [Eloquent JavaScript](#) graag op papier willen bestellen. Het bestelproces kan in verschillende fases zitten (in willekeurig volgorde):

1. Boek is in bestelling of in verzending (pending)
2. Boek is gearriveerd (fulfilled; ‘goed’)
3. Boek wordt niet geleverd (rejected; ‘fout’)

Middels Promises kunnen we concreet stap 2 en 3 afhandelen:

```
<iframe height="430" style="width: 100%;" scrolling="no" title="JavaScript - Promise -
Boekbestelling" src="//codepen.io/5hart/embed/VRowME/?height=430&theme-id=29061&default-
tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
  See the Pen <a href='https://codepen.io/5hart/pen/VRowME/'>JavaScript - Promise -
Boekbestelling</a> by Vijfhart-IT
  (<a href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

Om te beginnen hebben we in de `window.onload` een timer (`setTimeout`) geplaatst, die de variabele ‘bestellingAccepted’ wijzigt. Dit is om het geheel iets dynamischer te maken. In de praktijk zal deze variabele aan de hand van andere invloeden worden gewijzigd.

We maken vervolgens een Promise waarin wij de ‘goed’ en ‘fout’ situatie *beschrijven* (dus *niet* afhandelen!). Middels de `goed(...)` en `fout(...)` functie aanroepen wordt dus aangegeven dat dát wordt gedaan zodra (in dit geval) de variabele `true` of `false` is. Deze goed en fout situatie haken we in de code buiten de Promise op in door de `.then()` en `.catch()` methoden waarin we wél beschrijven wat te doen zodra dit momenten optreden.

Let tot slot op de \$ notatie van de boekLevering\$ variabele. Dit wordt regelmatig toegepast om aan te geven dat deze variabele een Promise of een zogeheten Observable bevat. Niets meer dan goed gebruik.

We kunnen Promises ook 'chainen'. Ofwel; doorgaan op het resultaat van de vorige Promise, met de volgende Promise. Dit is erg handig wanneer je code onoverzichtelijk gaat worden met het gebruik van callback functies die je in volgorde wilt uitvoeren (denk aan: http call in reactie op http call in reactie op http call, enzovoorts). Je code wordt dan een soort grote boom van geneste blokken.

Hoe lossen wij dit op? Kijk maar eens naar deze code:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - Promise Chaining - Boekbestelling" src="//codepen.io/5hart/embed/KEOpKG/?height=300&theme-id=29061&default-tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
  See the Pen <a href='https://codepen.io/5hart/pen/KEOpKG/'>JavaScript - Promise Chaining - Boekbestelling</a> by Vijfhart-IT
  (<a href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

We kunnen dus in de afhandeling van onze Promise(s), meermaals .then() gebruiken. We moeten daar wél opletten dat in ieder geval de 'goed' situatie van de eerste Promises een nieuwe Promise opleveren, andere kunnen wij er natuurlijk niet op voort boorduren.

Als je een video over chaining wilt zien, bekijk dan de volgende opname van The Coding Train:

```
<iframe width="560" height="315"
src="https://www.youtube.com/embed/QO4NXhWo_NM?start=53&end=1122" frameborder="0"
allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen></iframe>
```

Tot slot, een belangrijke functie die we ongetwijfeld gaan tegenkomen bij het werken met Promises: waitUntil(). Omdat een Promise asynchroon is en wij, noch de browser weet wanneer deze klaar is, werken we met de then() en catch() functies.

Lees [hier](#) alvast meer over de waitUntil() functie. We gaan in het hoofdstuk over Service Workers, er zelf mee oefenen.

- ➔ Voor meer documentatie over Promises, Promise chaining en het verschil met async/await en Observables, bekijk [deze](#) uiterst informatieve pagina van Google Expert [Jecelyn Yeen](#).
- ➔ Een mooie vergelijking tussen Promises en async/await lees je [hier](#).
- ➔ Lees voor meer informatie over Asynchroon programmeren in het algemeen en het gebruik van Promises pagina 181 t/m 188 van het boek.

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

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.

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.

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

Een paar verschillen m.b.t. 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 operators
Gaat hoe dan ook een keer 'af'	Stuurt pas data bij subscribe() (abonnement)

RXJS

De installatie van RXJS is vrij eenvoudig. We kunnen dit het meest eenvoudig doen via NPM:

```
[code]
npm install rxjs
[/code]
```

(of kijk op: [rxjs](#))

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

```
[code]
npm list
[/code]
```

Let op dat je een omgeving nodig hebt die ECMAScript 5 ondersteunt. Een goede oefenomgeving is: <https://stackblitz.com/>. Kies daar voor: 'Start a new app', gevolgd door: 'RXJS TypeScript'.

Operators

Er is een boel mogelijk met Observables. Je kunt na abonneren ook nog extra acties, zogeheten 'operators' toepassen op de datastroom. Denk aan dingen als:

- 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 kft geven

En dit is nog maar het tipje van de ijsberg.

Let op: sinds enkele versies van RXJS dienen alle operators die je op een Observable toepast, omringd te zijn door een 'pipe()' operator. Houdt dus in de gaten met welke versie je werkt; , wanneer je online gaat zoeken naar voorbeelden (en er mist een pipe()).

Een voorbeeld van het gebruik van een observable:

[code]

```
import { of } from 'rxjs';
```

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

```
of(1,2,3).pipe(
```

```
  // 'x' aangepast naar tekst, met x, vermenigvuldigd erin:
```

```
  map(x => `Prachtige vermenigvuldigde waarde: ${x*2}`),    // let hier op de komma op het eind
```

```
  // 'tap' doet niets meer dan mogelijkheid bieden tot uitvoeren van een actie die niets met 'x' doet.
```

```
  tap(x => console.log('Ik log alleen even!' + x));
```

```
);
```

```
.subscribe(
```

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

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

```
);
```

```
[/code]
```

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

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

Tot slot, een uitgebreide toelichting op *reactive programming* met Observables: [The Introduction to Reactive Programming you've been missing.](#)

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

In dit hoofdstuk gaan wij kijken naar de eigenschappen van het HTTP protocol en benutten we de mogelijkheden ervan door server verzoeken te sturen middels het XMLHttpRequest object en via de 'fetch' methode.

HTTP protocol

HTTP, wat staat voor HyperText Transfer Protocol, is een set met afspraken (protocol) over het versturen en ontvangen van gegevens tussen browsers en servers over het Internet. We zijn nu aangekomen bij versie 2.0, ofwel: HTTP/2. In deze versie zitten veel performance verbeteringen t.o.v. de vorige versie (1.1), onder andere het multiplexen (combineren) van verzoeken, wat zorgt voor een verlaagde kans op '[head-of-line blocking](#)' op het TCP protocol. Wat betreft merkbare verschillen als het gaat om HTTP requests, waar je al web programmeur mee te maken krijgt, zijn er praktisch geen.

Achtergrond

Sinds de opkomst van Google er veel veranderd in de ontwikkeling van web applicaties. Met name het doen van verzoeken naar de server is iets wat mede door die opkomst uitgegroeid tot één van de meest krachtige en nuttige features van een moderne webpagina. Deze verzoeken gebeuren bijna altijd asynchroon. Dit houdt in dat wanneer het verzoek wordt gedaan (zoekveld Google invullen, e-mail in webclient opslaan, formulier verzenden, afbeeldingen inladen na klik op knop, enzovoorts) de pagina *niet* hoeft te wachten op het antwoord van de server om verder te kunnen functioneren.

In de jaren 90 waren we nog genooddaakt de hele pagina te laten wachten totdat het verzoek was ontvangen bij de server en ook was verwerkt. Deze hele webpagina aan de client-side bleef dan blanco; heel ongebruiksvriendelijk. Inmiddels zijn we vele stappen verder.

HTTP requests en responses

Een HTTP bericht bestaat uit een tweetal componenten, namelijk:

- header
- body

De header bestaan uit een 'startregel' waarin het protocol, de statuscode en het soort verzoek staat. Daarnaast bevat de header 0 of meer header opties die het verzoek of de body beschrijven.

Ook hoort er nog een lege regel tussen de header en de body te staan om aan te geven dat er geen header onderdelen meer zijn.

Lees eerst [deze documentatie](#) van MDN als je meer details wilt weten over hoe een HTTP bericht er precies uit ziet voordat we zelf berichten gaan maken.

Praktijk toepassing

- Googelen
- Informatie ophalen van en sturen naar een server

HTTP berichten opstellen

Voordat we een verzoek kunnen versturen, zullen we een aantal zaken in overweging moeten nemen:

- Versturen en ontvangen we synchroon of asynchroon?
- Gaan we:
 - Gegevens opsturen om bijv. aan een database toe te voegen? POST
 - Gegevens opvragen uit bijv. een API? GET
 - Gegevens in de database willen laten bijwerken? PUT/PATCH
 - Gegevens uit de database willen laten verwijderen? DELETE
- 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?

Om prettig te werken met HTTP berichten en goed te kunnen kijken wat er gebeurt, raden wij aan een eenvoudige (Chrome) extensie te gebruiken zoals 'HTTP Trace', of een zeer uitgebreide extensie tool zoals 'Postman'.

➔ Meer over HTTP in het algemeen lees je in het boek op pagina's 311 t/m 314.

HTTP berichten met het XMLHttpRequest object

In meest basale vorm, kunnen we een HTTP bericht opstellen middels het zogeheten XMLHttpRequest object. Dit object wordt ook achter de schermen gebruikt door libraries en HTTP wrappers die het makkelijker maken om verzoeken te doen. Daarom is het goed de functionaliteiten, en quirks hiervan te kennen.

1. Allereerst is er een XMLHttpRequest object nodig. Deze maken wij met de hand en daarin plaatsen wij straks het verzoek naar de server.
2. Wij dienen vervolgens de verwerking van het verzoek af te vangen; kijken of het gelukt is en er dus geen fouten zijn opgetreden. Dit is een gebeurtenis (event) genaamd 'readystatechange'. Wanneer deze gebeurtenis 'af gaat', dienen wij te kijken of het bij de server ontvangen van het verzoek en ook terugsturen van de response goed is gegaan. Hiervoor kijken we naar een statusCode en naar de readyState.

Let op! Een HTTP bericht versturen en/of ontvangen gebeurt vrijwel altijd asynchroon. Dat

houdt in dat we niet direct code onder onze te gebruiken `send()` methode kunnen plaatsen die uitgevoerd moet worden zodra we bericht terug hebben!

Dus: de code die uitgevoerd moet worden zodra we bericht terug hebben, plaatsen wij dus in de functie van het `'readystatechange'`.

3. We krijgen tot slot een `responseText` eigenschap terug waar wij mee kunnen doen wat wij willen; denk aan tonen op het scherm in een tabel.

Bekijk het volgende voorbeeld:

```
<iframe height='318' scrolling='no' title='JavaScript - AJAX'
src='//codepen.io/5hart/embed/bLPPBg/?height=318&theme-id=29061&default-
tab=js,result&embed-version=2&editable=true' frameborder='no' allowtransparency='true'
allowfullscreen='true' style='width: 100%;'>See the Pen <a
href='https://codepen.io/5hart/pen/bLPPBg/'>JavaScript - AJAX</a> by Vijfhart-IT (<a
href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

De volgende belangrijke stappen zien we bovenstaand terug:

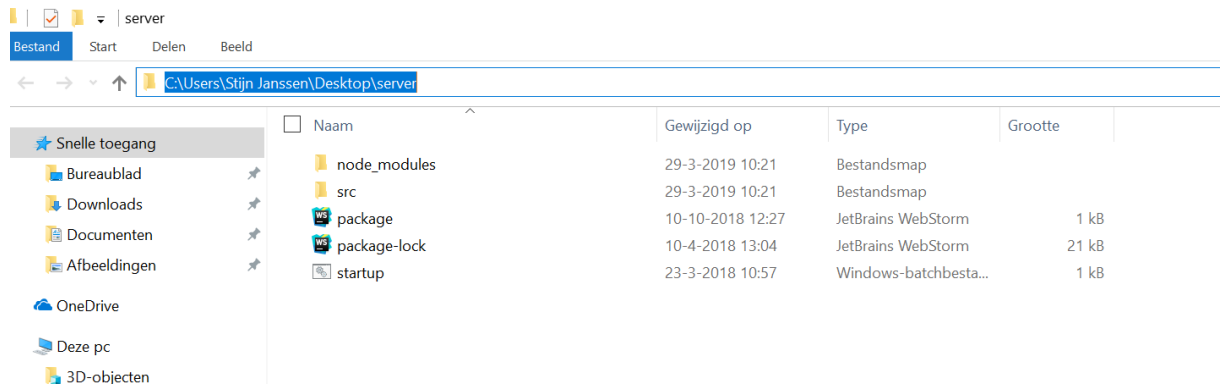
- Het aanmaken van een `XMLHttpRequest` object
- Het afvangen van het `readystatechange` event
- We kijken in de eventhandler of de `'readyState'` gelijk is aan 4 én of de `'status'` gelijk is aan 200.
- Tot slot openen we de connectie middels de `open()` methode. Deze verwacht van ons:
 - o Het soort verzoek (`get`, `post`, `put/patch`, `delete`)
 - o De URL waar wij een bericht naar willen sturen
 - o Moet het verzoek asynchroon (`true`) of synchroon (`false`) verstuurd worden (default = `true`).

Een overzicht van mogelijke ready states zijn [hier](#) te vinden. En een overzicht van mogelijke status codes vind je [hier](#). Al deze states zijn dus te gebruiken in je event handler; erg handig!

In het volgende voorbeeld gaan we zelf een aantal headers toevoegen en kijken of we de server middels de juiste methode (`POST`, `GET`, enz) kunnen benaderen.

Doorloop de volgende stappen voor dit voorbeeld:

- Download eerst **dit bestand (HTTP_server.zip)**. 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).
- Pak nu het gedownloade .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, direct in de juiste directory.
- Geef nu het commando: `npm install`
- Wacht even ... ;)
- Geef nu het commando: `npm start`
- Je zou nu dit venster moeten zien:

```

(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!
  
```

De server op onze computer is nu klaar om mee te communiceren om onze verzoeken te testen!

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

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

Omdat het afhandelen van berichten middels het XMLHttpRequest object vrij omslachtig / langdradig / onoverzichtelijk kan zijn, 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.

Bekijk het volgende voorbeeld ter illustratie; we hebben de random GET call uit het vorige voorbeeld (testpagina.html), herschreven naar de veel kortere `fetch()` variant:

```

[code]
fetch('http://localhost:1234/random')
  
```

```
.then(data=>{
    document.getElementById("merkWijzig").value = JSON.parse(data).merk;
    document.getElementById("typeWijzig").value = JSON.parse(data).type;
});
[/code]
```

Wát is de code ingekort! We zien dus dat de `fetch()` methode wordt aangeroepen en een Promise terug geeft. Daar kunnen wij dus op aanhaken middels `'then()'`.

Bij het versturen van een bericht middels POST, dienen we een extra parameter mee te geven. We kunnen hier naar wens ook headers instellen óf ervoor kiezen een [Headers](#) object te maken.

```
[code]
fetch('http://localhost:1234/versturen, { method: "POST" })
.then(..);
[/code]
```

Heb je tijd over? Probeer dan de volgende opdracht: maak een kopie van `'testpagina.html'` en pas de gehele code ervan aan zodat deze enkel gebruik maakt van `fetch()`. Zorg ervoor dat je ook het codeblok `'STATIC FILES HOSTEN'` in `'server.js'` aanpast (zoals daarin aangegeven), met je nieuwe file.

Gebruik [deze](#) pagina als naslag voor de `fetch()` API.

Progressive Web Apps

Leerdoelen

- Drie of meer voordelen kunnen benoemen van het maken van een PWA
- Twee redenen kunnen benoemen waarom het gebruik van HTTPS goed is
- Het kunnen maken van een Web Manifest volgens de minimale eisen
- Het kunnen maken van een Service Worker die zich installeert en activeert
- Het kunnen maken van een Notificatie

Progressive web apps, ofwel PWA's, veranderen het web in die zin dat het een *mobile first* platform wordt. Applicaties draaien sneller, het is mogelijk offline te werken en de applicaties kunnen aan het mobiele homescreen worden toegevoegd. De user experience, misschien wel het meest belangrijke onderdeel van iedere applicatie (in ieder geval voor klanten), kan significant verbeterd worden door je web applicatie om te bouwen naar een PWA. Met PWA's komen we héél dicht bij de vervanging van locale (niet-web) applicaties door geheel responsive, snelle, veelzijdige en goed te onderhouden web applicaties.

Praktijk toepassing

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

Volgens Google zijn er vier ervaringen die een PWA dient te bieden, namelijk: snelheid, betrouwbaarheid, betrokkenheid en integratie.

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

1. Een Web Manifest file moet aanwezig zijn
2. De web applicatie dient te draaien via HTTPS
3. Service Worker met fetch event handler dient geregistreerd te worden

We gaan deze voorwaarden nu uitgebreid behandelen.

Web Manifest

Laten we beginnen met het voornaamste profijt wat je als bedrijf achter de web applicatie hebt van een Web Manifest bestand. Dit bestand geeft je de mogelijkheid om jouw web applicatie als daadwerkelijke applicatie *met icoon* toe te voegen op de desktop van zowel Android als iOS mobiele devices alsook op Windows systemen. Dit noemt men de 'add to homescreen experience'.

Opbouw web manifest bestand

Het web manifest bestand dient een JSON bestand te zijn. Het is dus opgebouwd uit 'key' en 'value' paren volgens de JavaScript Object Notatie met gebruik van accolades {} en dubbele quotes "" om de key's en values heen.

De volgende onderdelen dienen in ieder geval aanwezig te zijn in het JSON bestand:

- **name**
Deze naam wordt gebruikt om de app naam te tonen bij het app installatie scherm. Deze property óf 'short_name' dient aanwezig te zijn.
- **short_name**
De invulling van deze waarde wordt gebruikt op het home-screen waar de app is geïnstalleerd én bij het draaien van de app. Deze property óf 'name' dient aanwezig te zijn.
- **icons**
Bevat de verschillende formaten van de iconen die deze app gebruikt. Deze iconen worden gebruikt op het home-screen, bij het switchen tussen apps, enzovoorts.
- **theme_color**
Stelt de kleur in van de toolbar (balk bovenin) van deze app. Er wordt aangeraden ook een meta-tag in de HTML file toe te voegen met dezelfde kleur instelling. De volgende opmaak is een optie:
`<meta name="theme-color" content="#hexkleurcode" />`

Let erop: zodra een gebruiker je app heeft toegevoegd aan zijn of haar home-screen als PWA, dan zal énkél de optie

- **background_color**
Achtergrond kleur van het splash-screen als hexadecimale code.
- **start_url**
Stuurt de user direct door naar het relatieve pad binnen je app, zonder de 'oude' pagina vast te houden waar de gebruiker was voordat hij de pagina toevoegde aan het home-screen.
- *orientation (optioneel)*
Stelt een orientatie in van de app waarin hij enkel werkt ('landscape' of 'portrait'). Dit is dus wel redelijk beperkend voor de eindgebruiker.
- *display (optioneel)*
Deze eigenschap bepaalt *hoe* onze web applicatie opgestart dient te worden en kan de volgende waarden bevatten: fullscreen, standalone (eigen window los van browser), minimal-ui (fullscreen, maar minimalistiser qua user interface), browser (standaard browser pagina).

Er mogen meer onderdelen aanwezig zijn, zelf in te vullen, maar zullen door de browser genegeerd worden en kunnen enkel t.b.v. je eigen implementatie extra details toevoegen.

Bekijk ons volgende voorbeeld van een uitgewerkt Manifest bestand:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - Manifest.json"
src="//codepen.io/5hart/embed/gyBveq/?height=300&theme-id=29061&default-
tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
```

See the Pen JavaScript - Manifest.json by Vijfhart-IT

(@5hart) on CodePen.

</iframe>

Dit Manifest bestand kun je zelf opbouwen, of laten genereren via bijvoorbeeld <https://app-manifest.firebaseio.com/>.

Validatie van het web manifest bestand

Om te kijken of je manifest bestand aan alle eisen voldoet en op de juiste wijze is opgebouwd, heeft Google een online validator opgesteld, te vinden op: <https://manifest-validator.appspot.com/>. Hiermee kun je dus jouw manifest bestand valideren. Wil je in een automatisch test traject ook jouw validatie doen, dan kun je hiervoor de (basale) validator van San650's [Github](#) gebruiken.

Web manifest koppelen aan je webpagina

Via een <link> element in de <head> van je pagina koppel je een manifest bestand aan je webpagina. De syntax hiervoor is doorgaans:

```
<link rel="manifest" href="manifest.json" />
```

Site HTTPS laten gebruiken

Een PWA zit heel dicht tegen native applicaties aan. En bij het werken met native applicaties is het eigenlijk zo goed als vanzelfsprekend dat veiligheid van de data die over de connectie gaat, gegarandeerd moet zijn. Dit is evenzo bij PWA's. Middels HTTPS, is het in ieder geval mogelijk dat de communicatie beveiligd is en er niet zomaar iemand de 'lijn' kan afsluisteren en daadwerkelijk iets met die data kan.

Naast deze uit security oogpunt bijna noodzakelijke reden voor het gebruiken van HTTPS, zijn er nog meer voordelen:

- Identiteit
- Veiligheid
- Integriteit
- SEO optimalisatie
- Web API ondersteuning
- Snelheid i.c.m. HTTP/2

Daarnaast; wil je gebruik gaan maken van HTTPS, dan zul je van een certificaat gebruik moeten maken. Dit is bij voorkeur **niet** een zogeheten [self-signed](#) certificaat, omdat een Service Worker zich daar vaak **niet** goed op registreert ([quick fix die op oudere Chrome versies werkt](#)) en is ook niet bedoeld voor productie omgevingen. Vaak zullen we moeten betalen voor het laten opzetten van

een goed certificaat wat we kunnen gebruiken om onze HTTPS omgeving mee op te zetten.

Onderstaand zetten wij de voordelen van het gebruik van HTTPS nog even op een rijtje:

Identiteit

Je wilt met je klanten een wederzijdse connectie, waarbij voor beide partijen duidelijk is met wie ze te maken hebben.

Veiligheid

Een HTTPS verbinding is beveiligd met een encryptie. Dat betekent dat kwaadwillende personen of systemen zoals niet zomaar informatie van de lijn kunnen uitlezen. Welbekende aanvallen zoals een '[man-in-the-middle-attack](#)' zijn hierdoor bijvoorbeeld een heel stuk lastiger uit te voeren; er is geen duidelijke data te lezen over wat er gecommuniceerd wordt tussen de betrokken partijen. De hacker kan zich niet voordoen als iets of iemand anders als hij of zij geen idee heeft waar de digitale conversatie of transactie over gaat.

Daarnaast zijn browsers heel actief bezig om HTTPS te promoten middels meldingen tijdens het browsen, wat een goede ontwikkeling is.

Integriteit

Omdat beide partijen weten wie er aan de andere kant zit én er geen externe partijen ook in de veilige HTTPS verbinding betrokken zijn, weten we altijd dat het over de juiste data gaat die gecommuniceerd wordt.

SEO Optimalisatie

Google heeft een betere score wanneer je site gebruik maakt van HTTPS. Het bedrijf staat er echt achter dat gebruik maken van HTTPS zorgt voor een beter en veiliger Internet. En daar hebben ze een punt.

Web API ondersteuning

Alsmar meer web API's en externe partijen willen zekerheid dat ze met de juiste partij communiceren. Digitale veiligheid staat bij veel partijen hoog op de agenda. Op het vlak van API's is het dus logisch dat ook daar alsmar meer HTTPS gebruikt gaat worden.

Snelheid i.c.m. HTTP/2

Het relatief nieuwe HTTP/2 protocol maakt gebruik van SSL (secured socket layer), waar HTTPS ook gebruik van maakt. Je webpagina kan vlotter worden indien hij geschikt is voor HTTP/2. Wél is voor beheerders het HTTP/2 protocol een stuk complexer dan HTTP.

Overwegingen bij migratie van HTTP naar HTTPS

Indien je overweegt over te stappen van HTTP naar HTTPS hebben wij alvast een aantal tips (wat dient er te gebeuren?) om mee te nemen in je overweging:

- Links bijwerken (van HTTP naar HTTPS), ook social media links.
- Gebruik protocolloze links (dus gebruik de dubbele slash // in plaats van http:// of https://).
- Auto re-direct instellen op de webserver bij navigatie naar HTTP variant van de webpagina.
- Eventueel Google Analytics bijwerken naar default gebruik van HTTPS.

- Eventueel een [sitemap](#) XML file bijwerken.
- Eventueel [robots.txt](#) bijwerken als deze links bevat

Service Worker met fetch API

Wat is een Service Worker?

Een Service Worker is een stuk JavaScript code, dat onder andere bedoelt is om een beter gebruiker interactie te leveren. Dit kan hij doen door als soort proxyserver tussen de grafische interface van de bezoeker en de server te fungeren. Hij biedt uitgebreide mogelijkheden voor caching van bestanden en in essentie offline gebruik van de webpagina. Daarnaast kun je via een Service Worker zogeheten [push notificaties](#) naar een gebruiker sturen, achtergrond synchronisatie uitvoeren, pre-caching mechanismen toepassen en tot slot zelfs HTTP verzoeken onderscheppen om de data aan te passen of als reactie een ander verzoek uitvoeren.

Het werken met een service worker doen wij middels het afvangen van events die op kunnen treden. Dit zijn events zoals 'install', 'active' en 'fetch'.

Let op: we zullen in de code vaak gebruik gaan maken van het keyword 'self' dit verwijst naar de Service Worker zelf in specifieke zin. Een soort globale scope zoals je dat ook hebt bij het window object.

Service worker maken

Een service worker is afhankelijk van browser ondersteuning voor zichzelf, als voor wat samenhangende technologie. Het geheel is afhankelijk van [ondersteuning](#) van het navigator.serviceWorker object, van het [gebruik](#) van promises, de mogelijkheid tot [gebruik](#) van de fetch API én dat HTTPS actief is. Deze vier functionaliteiten worden al door de meeste browsers ondersteund, maar kijk voor de zekerheid op de voorgaande links.

We gaan als demo pagina de URL '<https://teacherstijn.github.io/>' gebruiken. Je mag ook zelf een GitHub pages pagina maken om tot PWA om te bouwen.

De service worker thread

Een Service Worker draait in een eigen thread op de computer. Hij werkt onafhankelijk van overige pagina's en is alleen te gebruiken door pagina's binnen zijn 'scope'. Een Service Worker heeft **geen** toegang tot het Document Object Model en blokkeert het laden van een pagina niet (door de aparte thread).

Service worker life cycle

Belangrijk is om te weten dat het bestaan van een Service Worker uit vier fasen bestaat: de register, download, install en activate fasen. We gaan deze fasen nu bespreken.

Register fase

Het registreren van een Service Worker zorgt ervoor dat er een entry in het Service Worker register wordt toegevoegd wat door de User Agent wordt beheerd.

Een Service Worker op zichzelf beheerd één of meerder clients. Deze clients kunnen zijn; tabbladen in de browser (van eenzelfde domein), push notificaties of background synchronisatie events.

De daadwerkelijke Service Worker registratie gebeurt vanuit je webpagina door het aanroepen van de navigator.serviceWorker.register() functie. Deze functie verwacht minimaal één en maximaal twee parameters: allereerst het pad naar je Service Worker bestand (vaak /sw.js), en als tweede eventueel een object met daarin een 'scope' property.

Aanbevolen wordt om de Service Worker in de root directory van je webpagina te plaatsen. Vervolgens kun je middels de bovenstaande scope parameter als volgt aangeven over wélke directory (ten opzichte van je root!) in jouw applicatie je Service Worker beheer heeft:

```
[code]
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js', { scope: '/personen/' });
}
[/code]
```

Je ziet dat we eerst begonnen met een controle of de property 'serviceWorker' wel aanwezig is in 'navigator', dit om te checken of de browser wel Service Workers ondersteund.

Download fase

Zodra de registratie is voltooid (gelukt), wordt de Service Worker naar de machine van de client gedownload.

Install fase

De installatie van een Service Worker begint zodra er een eerste Service Worker wordt gevonden voor deze webpagina óf wanneer de gedownloade worker nieuwer is dan de huidige.

Activate fase

Activatie gebeurt zodra de installatie succesvol is verlopen. Dit is te volgen via het **install** event. Indien er al een Service Worker aanwezig was, zal de nieuwe pas geactiveerd worden zodra de oude is opgeschoond en er geen openstaande clients (vaak tabbladen) zijn die nog gebruik maken van de oude worker.

Let op: er kan altijd maar één Service Worker tegelijk actief zijn.

Caching principes

Eén van de belangrijkste features van een PWA is het optimaal inrichten van een offline omgeving en het snel laden. Hiervoor gebruiken wij HTTP interception en caching. In dit hoofdstuk gaan wij kijken naar hoe een HTTP interception eruit ziet. In een volgend hoofdstuk gaan we bezig met caching principes.

HTTP interception

Belangrijk om te beseffen is dat we middels een service worker HTTP verzoeken kunnen onderscheppen. Dit wordt ook wel HTTP interception genoemd. Zie het als een tussenkomst op de lijn, waarmee wij de ingediende verzoek kunnen aanpassen, managen of bijvoorbeeld combineren met andere verzoeken.

De service worker zit tussen de webpagina en de webserver in. Dus alle HTTP verzoeken lopen via de service worker.

Als er vanuit de webpagina een HTTP verzoek (GET, POST, PUT, DELETE) gedaan wordt, kunnen wij bijvoorbeeld:

- Het verzoek doorgeven naar het netwerk (de server), zoals altijd
- Een respons uit de cache terugsturen en dus het netwerk (de server) overslaan

- Een op maat gemaakte reactie geven

Dit alles kunnen wij relatief eenvoudig doen door het 'fetch' event af te vangen;

```
[code]
self.addEventListener('fetch', event => {
  /* afhandeling van event middels bijv. if-else structuur en opzetten van eigen 'request' en
  'response' objecten volgens de fetch API.
  */
});
[/code]
```

Een voorbeeld voor het afvangen van een event vind je op de <https://teacherstijn.github.io> pagina. Ook hebben de interception code op CodePen geplaatst (deze code kun je *niet* los draaien);

Ergens anders op de pagina wordt dit verzoek gedaan:

```
let url = 'https://bgg-json.azurewebsites.net/collection/edwalter';
fetch(url).then(response => {
    return response.json();
})
[..]
```

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JS - Service Worker - HTTP
Interception" src="//codepen.io/5hart/embed/RmraRP/?height=300&theme-id=29061&default-
tab=js" frameborder="no" allowtransparency="true" allowfullscreen="true">
  See the Pen <a href='https://codepen.io/5hart/pen/RmraRP/'>JS - Service Worker - HTTP
Interception</a> by Vijfhart-IT
  (<a href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

Notificaties

We kunnen middels een pushManager object, berichten sturen naar de eindgebruiker zijn bureaublad op een desktop computer, mobiele telefoon of ander device. Deze berichten kunnen vanaf de server (push notification) of vanuit de app zelf (local notification) geïnitieerd en zijn iets anders dan bijvoorbeeld SMS berichten, want ze worden altijd door (via) een app ontvangen en verzonden.

Allereerst zullen we moeten testen of push notificaties überhaupt worden ondersteund door de browser. Dit kan op de volgende manier:

```
[code]
if ('pushManager' in window){
  console.log("Push berichten worden ondersteund!");
}
[/code]
```

Binnen het bovenstaande if-blok kunnen we dan onze registratie verder afhandelen.

Registratie (subscribe en unsubscribe)

Om een gebruiker de melding te laten krijgen zich te registreren op push notificaties, dienen we hem of haar eerst om toestemming te vragen. Dit is een afgesproken regel in de MDN specificatie en dient dus echt te gebeuren.

Let op; de **oude** manier is om een callback functie mee te geven aan de `requestPermission()` functie:

```
[code]
Notification.requestPermission(callback);
[/code]
```

Hoe we tegenwoordig **wel** het verzoek schrijven voor toestemming:

```
[code]
Notification.requestPermission().then(function(result) {
  if (result === 'denied') {
    console.log('Geen toestemming gekregen, later nog eens proberen?');
    return;
  }
  if (result === 'default') {
    console.log('De toestemming is afgewezen.');
```

```
    return;
  }

  // Anders result === 'granted'
  // Doe iets moois
});
[/code]
```

Vervolgens kunnen wij de gebruiker registreren.

De gebruiker registratie vereist een unieke code per applicatie. Deze code dient door de server gegenereerd te worden en bestaat uit een public en een private key. In JavaScript is de code uit te lezen middels een eigenschap genaamd `'applicationServerKey'`. Deze code wordt ook wel een [VAPID](#) key genoemd. Om VAPID sleutels te kunnen generen op je server, zou je vanuit een MS-DOS venster (cmd) de tool `'web-push'` kunnen downloaden:

```
[code]
npm install -g web-push
web-push generate-vapid-keys
[/code]
```

Er worden nu codes gegenereerd. De private key dien je voor jezelf te houden op de server. De public key kun je gaan gebruiken in je front-end web applicatie.

Voordat we iemand aanmelden, dienen we uiteraard eerst te kijken of er niet al een aanmelding is voor deze eindgebruiker. Dit doen we door de huidige `'subscription'` op te halen:

```
[code]
registratie.pushManager.getSubscription().then(
  sub => {
    if (sub === null) {
```

```

        console.log("Er is nog geen geregistreeerde Push-gebruiker");
        maakPushUser(registratie);
    }
}
)
[/code]

```

We kijken dus of het object, de Promise die wij terug krijgen van de `getSubscription()` leeg is.

Wanneer we een geldige back-end hebben om een unieke code per gebruiker aan te maken, zou de 'maakPushUser()' methode er als volgt uit kunnen zien:

<iframe height="300" style="width: 100%;" scrolling="no" title="JS - ServiceWorker - PushRegistratie" src="//codepen.io/5hart/embed/wbMWox/?height=300&theme-id=29061&default-tab=js" frameborder="no" allowtransparency="true" allowfullscreen="true">
 See the Pen JS - ServiceWorker - PushRegistratie by Vijfhart-IT
 (@5hart) on CodePen.</iframe>

De `urlBase64ToUint8Array()` functie is een functie beschreven door [Google](#).

Ontvangst en tonen van notificatie

Hoe gaan we om met een binnenkomende push melding? Zodra er een push melding wordt ontvangen, zal door onze Service Worker het 'push' event afgaan. Uiteraard kunnen wij deze weer afvangen.

Het object wat wij terugkrijgen heeft een eigenschap 'data'. Deze heeft weer een aantal methoden, die bepalen in welk formaat wij onze data willen zien. Dit kan zijn: `text()`, `json()`, `blob()` of `arrayBuffer()`.

Voor de hand liggende code zou zijn:

<iframe height="300" style="width: 100%;" scrolling="no" title="JS - ServiceWorker - Push-OLD" src="//codepen.io/5hart/embed/MdKerW/?height=300&theme-id=29061&default-tab=js" frameborder="no" allowtransparency="true" allowfullscreen="true">
 See the Pen JS - ServiceWorker - Push-OLD by Vijfhart-IT
 (@5hart) on CodePen.</iframe>

Probleem: binnen de W3C Service Worker beschrijving, staat dat indien er een push melding optreedt, er altijd eerst een melding aan de gebruiker gegeven dient te worden alvorens deze event handler klaar is.

We kunnen dus onze code uitbreiden naar:

<iframe height="300" style="width: 100%;" scrolling="no" title="JS - ServiceWorker - Push - New" src="//codepen.io/5hart/embed/LoGZdy/?height=300&theme-id=29061&default-tab=js" frameborder="no" allowtransparency="true" allowfullscreen="true">
 See the Pen JS - ServiceWorker - Push - New by Vijfhart-IT

([\) on \[.\]\(https://codepen.io\)](https://codepen.io/5hart)

Probleem 2: de browser weet niet precies wanneer de popup is getoond. Er kan een vertraging in zitten; het gebeurt namelijk asynchroon. De `showNotification` methode is namelijk een Promise.

Een veel voorkomende functie die hiervoor de oplossing biedt, is `waitUntil()`.

`waituntil()`

De browser zal vaak op ogenschijnlijk niet gebruikelijke momenten een Service Worker stoppen. Omdat het voor de browser niet op voorhand duidelijk is wanneer een Promise 'resolved' is, zal hij soms het beste even kunnen wachten met afsluiten van een Service Worker. Dit kun je realiseren door de `waitUntil()` methode uit te werken. Deze methode verwacht als parameter een Promise. Hij zal deze in ieder geval in zijn geheel laten resoluten (goed of fout), alvorens hij (mogelijk!) de Service Worker stopt.

De bovenstaande code wordt met gebruik van de `waitUntil()` functie:

`<iframe height="300" style="width: 100%;" scrolling="no" title="JS - ServiceWorker - Push - New - Complete" src="//codepen.io/5hart/embed/mYVEjL/?height=300&theme-id=29061&default-tab=js&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">`

See the Pen [by Vijfhart-IT](https://codepen.io/5hart/pen/mYVEjL/)

([\) on \[.\]\(https://codepen.io\)](https://codepen.io/5hart)

Reactie van de gebruiker

Hoe gaan we om met een click op een notificatie? We zijn in de mogelijkheid het klikken op een opgekomen notificatie af te vangen. Ook hiervoor is een event (binnen de Service Worker), genaamd 'notificationclick'.

Een veel toegepast idee is om na klikken bijvoorbeeld naar een andere pagina te navigeren. Gezien een Service Worker echter geen toegang heeft tot het window object, kunnen we deze code gebruiken:

```
[code]
self.addEventListener('notificationclick', ev => {
  ev.notification.close();
  clients.openWindow("https://www.vijfhart.nl");
});
[/code]
```


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

Een trend die je eindgebruiker hopelijk niet doorhebt, is dat steeds meer web applicaties ogenschijnlijk geen 'last' hebben van verlies van verbinding; geen wifi, bijvoorbeeld.

In de huidige tijd moeten we web applicaties en web pagina's in algemene zin zo weergeven dat de eindgebruiker niets, tot zeer weinig merkt van het offline zijn van een pagina. We kunnen events schrijven die dat in de gaten houden, we schrijven gegevens een lokale storage en kunnen tot slot bestanden cachen om de ervaring voor de eindgebruiker toevallig als mogelijk te laten zijn, ongeacht het al dan niet wegvallen van een netwerk verbinding.

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 wege gevallen 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

Online en property

We kunnen in JavaScript kijken of de browser waar we in draaien een Internet connectie heeft. Dit kunnen we doen middels de *navigator.onLine* eigenschap. Zoals de eigenschap leest, levert deze 'true' op indien we connectie hebben en 'false' indien dat niet het geval is.

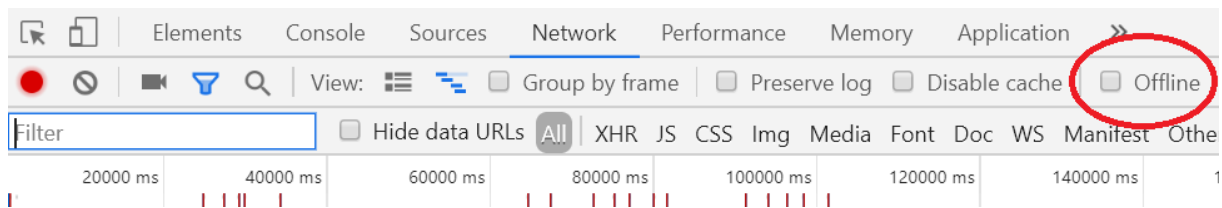
Volgens de specificatie van MDN heeft deze property de waarde false indien:

"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)..."

Hoe ziet het gebruik van deze property eruit? Vaak vangen wij de 'online' en 'offline' events af op het window object:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - Online en Offline"
src="//codepen.io/5hart/embed/VNJeez/?height=300&theme-id=29061&default-
tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
See the Pen <a href='https://codepen.io/5hart/pen/VNJeez/'>JavaScript - Online en Offline</a> by
Vijfhart-IT (<a href='https://codepen.io/5hart'>@5hart</a>) on <a
href='https://codepen.io'>CodePen</a>.</iframe>
```

Het simuleren van bovenstaand voorbeeld kan relatief eenvoudig door in de Chrome Developer Tools onder het tabblad 'network' het vinkje aan te zetten bij 'offline':



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 en kunnen ze direct van de harde schijf van de client worden ingeladen.

We beginnen met het noemen van een aantal belangrijke overwegingen, gevolgd door caching met behulp van het cache manifest en tot slot caching middels de cache API in een PWA.

Caching overwegingen

Hoeveel bestanden er gecached dienen te worden is geheel afhankelijk van de situatie:

- 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; denk aan beperkte opslag van deze devices en eventueel aan beperkte mobiele data bundels.
- 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? Misschien eerst enkel de miniatures, óf alleen degenen die bij het openen van de pagina al direct zichtbaar zijn op het huidige device?
- Zijn er afbeeldingen, stukken JavaScript of CSS code die pas hoeven ingeladen te worden na het optreden van een event (klik op een knop bijvoorbeeld)?
- 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 zogeheten appCache manifest bestand (naampje.appcache) gebruikt volgens een opbouw zoals deze:

```
[code]
CACHE MANIFEST
# Gemaakt door Vijfhart-IT
# Commentaar regels beginnen met een hash-teken
CACHE:
index.html
```

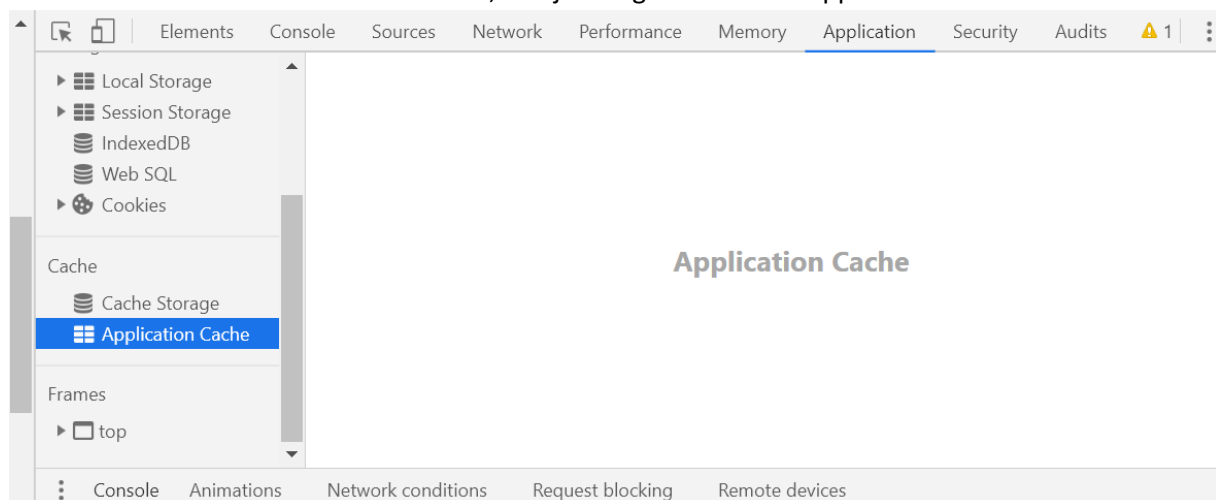
producten.html
images/groot_plaatje.jpg
scripts/jquery.js

NETWORK:
database_verzoeken.php
login_check.php

FALLBACK:
index.html
belangrijk.html
[/code]

Hierbij zien we een opsplitsing tussen CACHE, NETWORK en FALLBACK. Lees [hier](#) meer over deze categorisering.

De files die al dan niet in de cache zitten, kun je terug vinden in de Application Cache:



Het koppelen van het appcache bestand aan je applicatie doe je door:

```
<html manifest="example.appcache">
```

De browser cached standaard geen pagina's als er geen manifest attribuut is toegevoegd. Hij wordt wél gecached indien het bestand al *in* de cachemanifest file zelf benoemd is. Daarnaast, zodra je een manifest attribuut toevoegt aan het HTML element van een pagina, zal de browser deze file automatisch toevoegen aan je lokale cache manifest.

Een file die in eerste instantie niet in de cache manifest staat, maar door toevoeging van het manifest attribuut in het HTML element, wordt ook wel een master entry genoemd.

Een aangepast bestand, waarvan je wilt dat deze opnieuw wordt gedownload, wordt niet automatisch opgemerkt door de browser. In dit geval is het aan te raden je cache manifest bestand te wijzigen, alsdoor er een regel commentaar `#ikbencommentaar` in te zetten, of bijvoorbeeld enkel een versie nummer achter het hekje `#`. Hierdoor zal de browser wél een gewijzigd manifest bestand herkennen en deze geheel opnieuw doorlopen.

Het gebruik van de cache zorgt ervoor dat de normale flow van het laden van een document wordt aangepast:

- Indien er een applicatie cache aanwezig is, zal de browser het document en bijbehorende bronnen direct uit deze cache uitlezen, zonder het netwerk (Internet) te gebruiken. Dit versnelt de laadtijd van een pagina aanzienlijk.
- De browser kijkt nu of de cache manifest file is aangepast op de server
- Als het cache manifest is aangepast, zal de browser een nieuwe versie van het manifest downloaden en van de bronnen die in het manifest vermeld staan. Dit wordt in de achtergrond gedaan en zal *geen* aanzienlijke impact hebben op de performance.

Wanneer onze pagina een appCache manifest bestand gebruikt, maar ook een geregistreerde service worker heeft, zal de service worker het cachen regelen en wordt de appCache manifest gepasseerd. Het managen van de cache biedt veel controle aan de programmeur.

We kunnen vanuit JavaScript bekijken of de Manifest file bijgewerkt is middels het 'updateready' event:

```
[code]
function updFunctie;() {
  console.log('Er is een bijgewerkt Manifest bestand!');
}
window.applicationCache.addEventListener('updateready', updFunctie);
[/code]
```

Omdat we geen zekerheid hebben dat het Manifest bestand wijzigt ná het koppelen van deze event listener, wordt aangeraden om onder de eventlistener ook deze code uit te voeren, die met de hand de status bekijkt en e.v.t. de functie uitvoert:

```
[code]
if(window.applicationCache.status === window.applicationCache.UPDATEREADY) {
  updFunctie();
}
[/code]
```

Cache in een PWA met behulp van de cache API

Deze API is zowel te gebruiken binnen een gewoon browser tabblad (en het window object) én in Service Workers. Hoewel het meest wordt gekozen voor gebruik in de laatste. We zullen binnen de scope van de Cache API focussen op het gebruik ervan binnen een PWA.

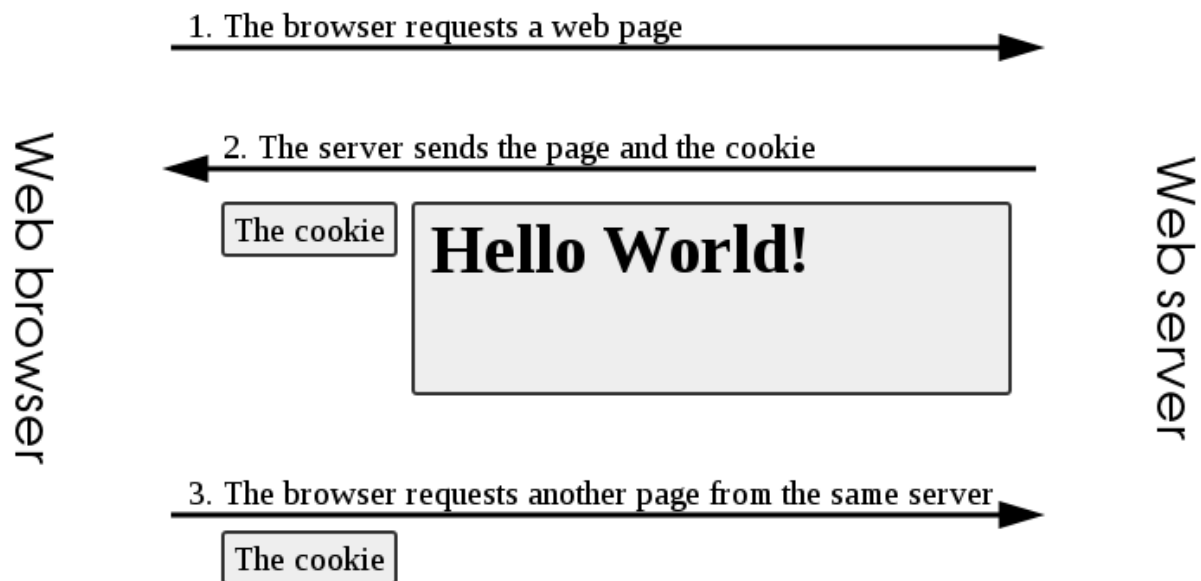
Bekijk de volgende video van Google over het gebruik van de cache API binnen een PWA, hierin wordt uitgebreid toegelicht hoe de cache API werkt:

[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 zogeheten 'cookie' kunnen webserver 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](#)):



Praktijktoeepassing

- Opslaan gebruikersvoorkeuren
- Koppelen gebruiker (browser) aan specifieke sessie
- Volgen van gebruikers voor advertentie doeleinden

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.

Zo ziet een cookie er bijvoorbeeld uit:

```
[code]
SPORT=Tennis;
Path=/sports
HttpOnly
[/code]
```

Aan een cookie zijn verschillende eigenschappen (attributen) mee te geven, waarvan de voornaamste 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](#).

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

[code]

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

[/code]

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

Session Storage en local storage

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.

Wanneer wij local storage willen gaan gebruiken, moeten wij kiezen uit een tweetal implementaties: sessionStorage, dat is lokale opslag, die verloren gaat zodra de eindgebruiker de browser afsluit. Een 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, dat is eveneens lokale opslag, blijft behouden wanneer de browser wordt afgesloten en/of heropend. Een voordeel van het localStorage mechanisme, is dat we bijvoorbeeld gebruikersprofielen (instellingen, winkelwagentjes et cetera) kunnen behouden op de computer, waardoor deze zelfs nog beschikbaar zijn als de webbrowser afgesloten is geweest en de gebruiker bijvoorbeeld na 2 weken weer verder wilt shoppen met hetzelfde winkelwagentje.

Via de gelijknamige eigenschappen (zijnde objecten) window.sessionStorage en window.localStorage kunnen wij gebruik maken van deze opslag methoden.

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

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.

In onderstaand voorbeeld gaan wij deze twee functies (setItem en getItem) toepassen. 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.

```
<iframe height='1044' scrolling='no' title='HTML 5 - LocalStorage API'
src='//codepen.io/5hart/embed/zpdgEj/?height=1044&theme-id=29061&default-
tab=html,result&embed-version=2&editable=true' frameborder='no' allowtransparency='true'
allowfullscreen='true' style='width: 100%;'>See the Pen <a
href='https://codepen.io/5hart/pen/zpdgEj/'>HTML 5 - LocalStorage API</a> by Vijfhart-IT (<a
href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

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:

```
<iframe height='502' scrolling='no' title='HTML 5 - Localstorage'
src='//codepen.io/5hart/embed/yKgGxx/?height=502&theme-id=29061&default-
tab=js,result&embed-version=2&editable=true' frameborder='no' allowtransparency='true'
```

```
allowfullscreen='true' style='width: 100%;'>See the Pen <a
href='https://codepen.io/5hart/pen/yKgGxx/'>HTML 5 - Localstorage</a> by Vijfhart-IT (<a
href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

Een aantal verschillen tussen cookies en het localStorage concept worden onderstaand uitgelegd. Deze opname is niet essentieel voor het doorlopen van de cursus en kan als bonus worden gezien: <https://youtu.be/5ttpghXjG0g> (door: [DrapsTV](#)).

IndexedDB

Opbouw van de database

In iedere moderne browser zit een database ingebouwd. Dit is een NoSQL database, waarbij dus geen taal als SQL wordt gebruikt, maar waarbij we middels een JavaScript API met de database kunnen praten. Deze specifieke database heet een IndexedDB.

Het voornaamste verschil met veel andere databases, is dat we hierin objecten kunnen opslaan. JavaScript objecten kunnen an sich weer eigenschappen bevatten (data).

We kunnen zelf een database maken, kijken of de huidig gebruikte versie in de browser van de specifieke eindgebruiker nog actueel is, deze eventueel bijwerken en vervolgens ermee aan de slag gaan.

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.

FILMPJE OVER OPSLAGVORMEN (IVM CAPACITEIT)

Voor het werken met IndexedDB databases zouden we ook een library kunnen gebruiken die het werken met Promises mogelijk maakt (ipv de standaard callback functies. Een goed voorbeeld hiervan is:

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

Transacties

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`):

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

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.

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.

Object store

De object store is onze collectie waarin we gegevens kunnen opslaan. Deze collectie kan allerlei soorten gegevens bevatten, maar voornamelijk zullen dit JavaScript objecten zijn, die zelf uiteraard weer allerlei eigenschappen (zoals objecten) kunnen bevatten.

CRUD applicatie

Create Read Update Delete: een veel toegepaste term om de werking met gegevens aan te kunnen tonen. Je zult zodadelijk een voorbeeld zien waarin we bovenstaande concepten (connectie, transactie en object store) toepassen en manipulaties gaan uitvoeren op onze opgeslagen collectie:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - IndexedDB - Extended"
src="//codepen.io/5hart/embed/zLWvZa/?height=300&theme-id=29061&default-
tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
  See the Pen <a href='https://codepen.io/5hart/pen/zLWvZa/'>JavaScript - IndexedDB -
  Extended</a> by Vijfhart-IT
  (<a href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

Cursoren

Een cursor kunnen we gebruiken als alternatieve navigatiewijze door onze collectie(s) heen. Ook is het mogelijk door een index heen te navigeren. We kunnen in beide richtingen door een verzameling heen, als enkel gebaseerd op een conditie voor de velden waar we doorheen itereren. 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.

Extra FILMPJE OVER 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

Wanneer we een pagina hebben gemaakt waar veel afbeeldingen, video's of andere externe bronnen worden ingeladen, maar niet direct in volledige glorie zichtbaar hoeven te zijn op de pagina, 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.

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.

Zoals je ziet; bijna op alle momenten dat je een afbeelding, audio of video wilt tonen *zou* je lazy loading kunnen toepassen.

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).

Een voorbeeld:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - Lazy Loading (1)"
src="//codepen.io/5hart/embed/xENGqQ/?height=300&theme-id=29061&default-
tab=html,result&editable=true" frameborder="no" allowtransparency="true"
allowfullscreen="true">
```

See the Pen [JavaScript - Lazy Loading \(1\)](https://codepen.io/5hart/pen/xENGqQ/) by Vijfhart-IT

([@5hart](https://codepen.io/5hart)) on [CodePen](https://codepen.io).

```
</iframe>
```

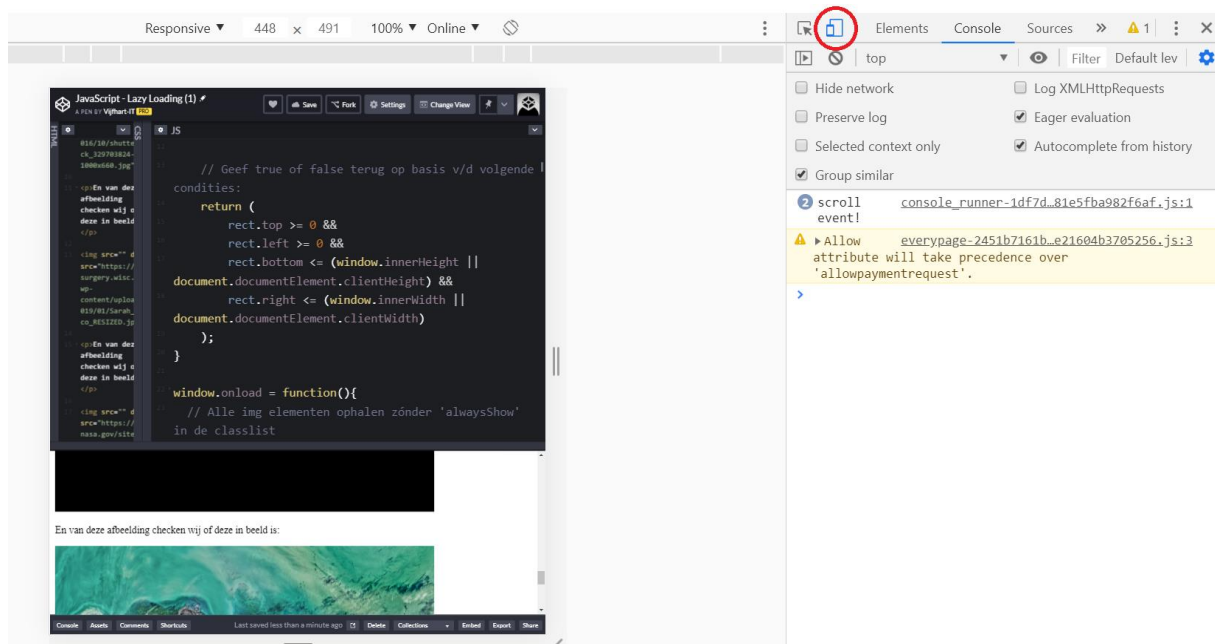
Als we het hebben over nog niet zichtbare objecten, is het vanuit performance én esthetisch oogpunt (responsiveness en aanzicht), aan te raden de afbeeldingen (en andere grote objecten) die nog niet zichtbaar zijn tijdelijk te vervangen of voor alsnog geheel weg te laten.

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 miniatures inladen
- Niet zichtbare afbeeldingen alternatieve tekst inladen

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.

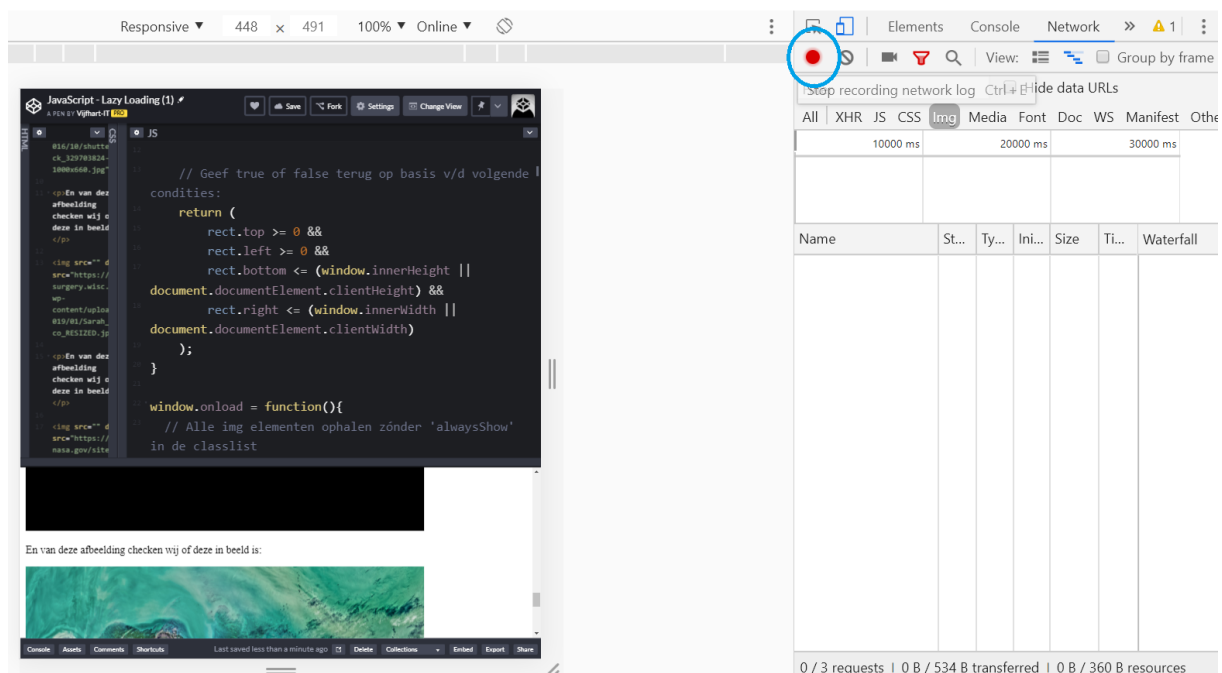
Onderstaand een voorbeeld hoe we dit toepassen. Let op! Schakel over naar 'mobiele modus' door op de knop 'Toggle Device Bar' te klikken.



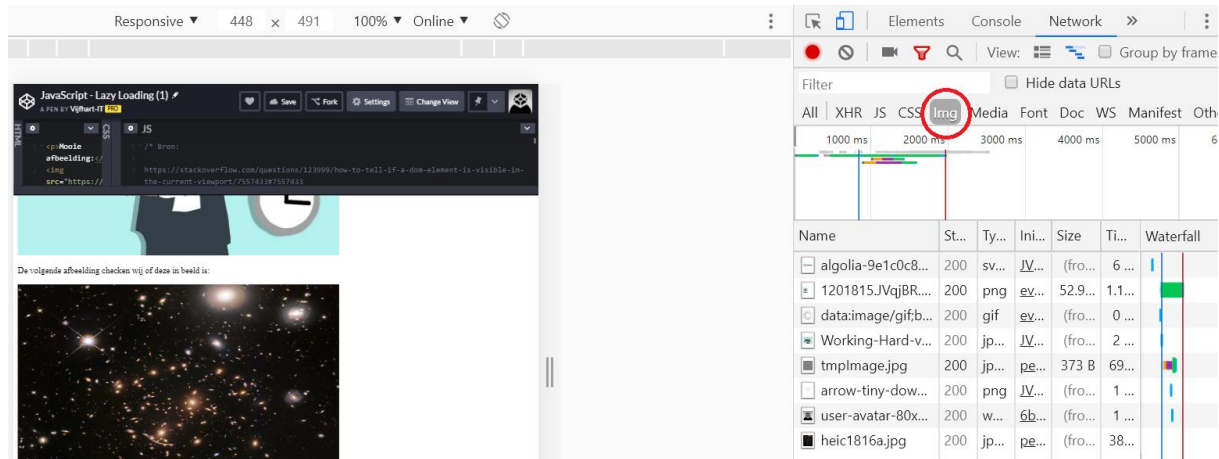
Hierdoor zie je goed wanneer een afbeelding *geheel* in beeld is.

<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - Lazy Loading (1)" src="//codepen.io/5hart/embed/JVqjBR/?height=300&theme-id=29061&default-tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true"> See the Pen JavaScript - Lazy Loading (1) by Vijfhart-IT (@5hart) on CodePen.</iframe>

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:



- 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:



We hebben in de code een array gemaakt met daarin alle afbeeldingen van de pagina die géén class genaamd 'alwaysShow' hebben, want die specifieke afbeeldingen willen wij altijd direct geladen hebben in ons voorbeeld.

Vervolgens hebben we een functie beschreven die bekijkt of een opgegeven element geheel binnen de huidige viewport valt (o.a. door gebruik te maken van `getBoundingClientRect()` voor de kaders van het element, `window.innerHeight` of `document.documentElement.clientHeight` en de gelijknamige Width varianten voor het bepalen van de viewport).

We vangen nu 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.

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

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

Hier zullen wij in deze cursus uit tijdsoverweging niet op in gaan.

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

Met WebSockets is het mogelijk een communicatielijn open te zetten van een client browser naar een server. Via deze lijn kunnen we data heen en weer sturen. De afhandeling van het versturen, ontvangen en de tussentijdse status daarvan is geregeld middels JavaScript events. Het aanmaken van een WebSocket object gaan wij zo onderstaand tonen. 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.

Let dus op: we kunnen niet zondemeer een WebSocket object maken en data proberen te versturen. Er zijn de nodige installatiestappen nodig op een server.

Praktijk toepasbaarheid

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

Het idee achter Web Sockets is dat het een geheel ander protocol is waar gegevens overheen gaan, dan bijvoorbeeld HTTP(s). Het voornaamste voordeel van een Web Socket is dat er een continue lijn open gezet kan worden naar de server. Er is een full-duplex verbinding beschikbaar. 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.

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

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

We kunnen ervoor kiezen het ws of het wss protocol te gebruiken, waarbij het wss protocol staat voor de secure variant. Omdat de echo server in bovenstaand voorbeeld gebruik maakt van een wss protocol, dienen wij dat als client / eindgebruiker ook te doen.

Zodra de connectie gemaakt is, dienen we de 'onopen', 'onerror' en 'onmessage' events van dit socket object af te vangen.

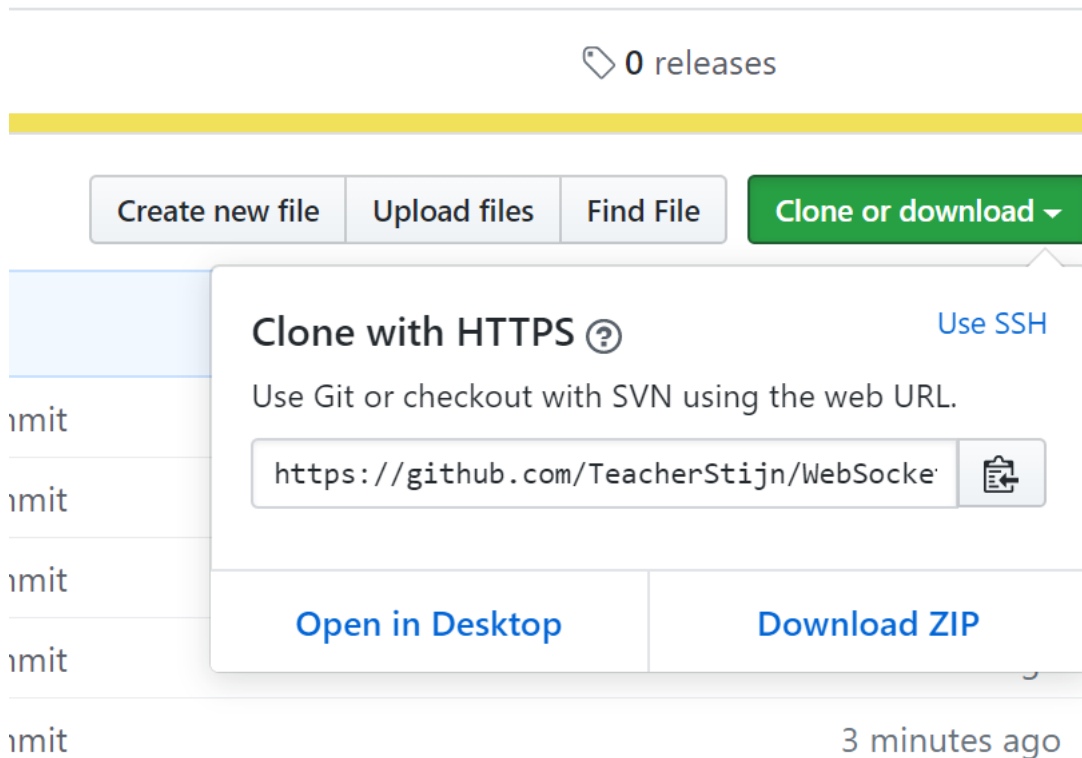
Onderstaand zien we de gehele code uitgewerkt:

```
<iframe height="300" style="width: 100%;" scrolling="no" title="JavaScript - WebSocket "
src="//codepen.io/5hart/embed/pBXmQw/?height=300&theme-id=29061&default-
tab=js,result&editable=true" frameborder="no" allowtransparency="true" allowfullscreen="true">
  See the Pen <a href='https://codepen.io/5hart/pen/pBXmQw/'>JavaScript - WebSocket </a> by
  Vijfhart-IT
  (<a href='https://codepen.io/5hart'>@5hart</a>) on <a href='https://codepen.io'>CodePen</a>.
</iframe>
```

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 dus hier op de 'Clone or Download' knop:



Een voorbeeld als deze is perfect uit te breiden met afvangen van specifieke berichten uit bijvoorbeeld een zelf bedachte game.

Vrij nagedacht:

- 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](#)).

Bronnen

Opbouw cursus generiek:

<https://www.taniarascia.com/how-to-connect-to-an-api-with-javascript/>

Classes en hasOwnProperty:

<http://thecodebarbarian.com/static-properties-in-javascript-with-inheritance.html>

<https://stackoverflow.com/questions/1291942/what-does-javascriptvoid0-mean>

Offline applicaties:

<https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook/>

Boeken algemeen:

<https://www.imkrish.com/best-javascript-book-to-learn-in-2019/>

<https://frontendmasters.com/books/front-end-handbook/2018/learning/web-api.html>

<https://developers.google.com/books/docs/overview>

Boek: [progressive web application development](#)

HTTP:

<https://www.taniarascia.com/how-to-connect-to-an-api-with-javascript/>

Design patterns (wat oud sommige):

Factory: <https://medium.com/javascript-scene/javascript-factory-functions-with-es6-4d224591a8b1>

Overig: <https://jstherightway.org/#patterns>

DIE HARD JS Deel 3(?) boek:

<https://leanpub.com/composingsoftware>

Usefull libraries (bonus?):

Lodash, Moment, jQuery