1 Inleiding JSON

JSON is 'tekst' en staat voor JavaScript Object Notation. JSON is een eenvoudige, zelfbeschrijvende syntax voor het sorteren en uitwisselen van data. JSON maakt het mogelijk om eenvoudig met data als Javascript objecten te werken.

Wanneer we data gaan uitwisselen tussen een webbrowser en een server, moet deze data in tekstvorm staan. We kunnen ieder Javascript object in JSON converteren en verzenden naar de server. We kunnen ook omgekeerd alle ontvangen JSON van de server converteren naar Javascript objecten.

1.1 JSON syntax

JSON wordt op volgende manier geschreven:

- Data wordt altijd geschreven in naam-waarde-paren
- Data wordt gescheiden door komma's
- Accolades staan rond de objecten
- Vierkante haakjes staan rond de arrays

1.1.1 JSON data

JSON data wordt geschreven als 'naam-waarde paren'. Een naam-waarde-paar bestaat uit een veldnaam tussen *dubbele aanhalingstekens*, gevolgd door een dubbelepunt en een waarde.

```
"voornaam": "Rik"
```

1.1.2 JSON Objecten

JSON objecten worden geschreven binnen accolades. JSON objecten kunnen meerdere naam-waarde-paren bevatten. De naam-waarde-paren worden gescheiden door een komma.

```
{"voornaam":"Rik", "leeftijd":21, "stad":"Genk"}
```

1.1.3 JSON Arrays

JSON arrays worden geschreven binnen vierkante haken. Net zoals in Javascript, kunnen arrays meerdere objecten bevatten.

```
"studenten": [
      {"voornaam":"Rik", "leeftijd":21, "stad":"Genk"},
      {"voornaam":"Anne", "leeftijd":19, "stad":"Hasselt"},
      {"voornaam":"Dorien", "leeftijd":18, "stad":"Hasselt"}
]
```

1.2 Data verzenden

Wanneer je data hebt opgeslagen in een Javascript object, kan je dit object converteren naar JSON en naar de server verzenden *(gaan we in deze cursus niet doen!)*.

```
let persoon = {naam: "Rik", leeftijd: 21, stad: "Genk"};
let persoonJSON = JSON.stringify(persoon);
window.location = "demo_json.php?x=" + persoonJSON;
```

1.3 Data ontvangen

Wanneer je data ontvangt in een JSON-formaat, kan je deze converteren naar een Javascript object.

```
let persoonJSON = '{"naam":"Rik", "leeftijd":21, "stad":"Genk"}';
let persoon = JSON.parse(persoonJSON);
document.getElementById("demo").innerHTML = persoon.naam;
```

2 Inleiding AJAX

AJAX staat voor Asynchronous Javascript And XML. AJAX is geen programmeertaal, maar een manier om interactieve webapplicaties te ontwikkelen door een combinatie van verschillende web technologieën te gebruiken:

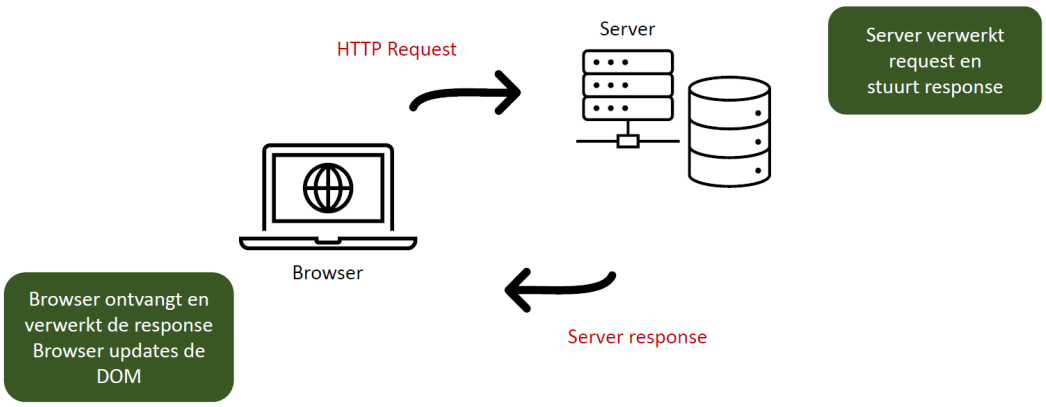
- HTML en CSS
- Javascript
- Document Object Model
- XML en JSON
- Fetch en XMLHttpRequest
- …

Met AJAX kunnen we, zonder dat er nieuwe pagina-ladingen nodig zijn:

- data van een server opvragen
- data van een server ontvangen
- data naar een server sturen

Opgelet! Ondanks dat de 'X' in AJAX verwijst naar XML, kunnen we informatie in meerdere formaten verzenden en ontvangen. Zo wordt in de praktijk JSON meer gebruikt omdat het lichter is én een 'onderdeel' van Javascript.

2.1 Werking AJAX



1. Een event vindt plaats op de webpagina.
2. Er wordt een *XMLHttpRequest object* aangemaakt door Javascript.
3. Het XMLHttpRequest object stuurt een *HTTP request* naar de server.
4. De server verwerkt de HTTP request en stuurt een antwoord terug.
5. Het antwoord wordt gelezen door Javascript en een *actie* wordt uitgevoerd.

2.2 HTTP Request

Met een *HTTP request* kunnen we de gewenste *actie* aangeven.

GET	Ontvang, lees
POST	Verstuur, creëer
PUT	Upload, wijzig
DELETE	Verwijder

2.3 Het XMLHttpRequest object

Het *XMLHttpRequest object* is een Javascript object. Met het XMLHttpRequest object kunnen we op een gemakkelijke manier informatie uit een URL ophalen zonder de gehele pagina te herladen. De webpagina kan een gedeelte van de pagina bijwerken zonder dat de gebruiker daar last van heeft.

Het XMLHttpRequest object wordt veel gebruikt in AJAX programmering. Via een XMLHttpRequest kunnen we verschillende soorten data ophalen.

De syntax voor het creëren van een XMLHttpRequest ziet er als volgt uit:

```
let xhttp = new XMLHttpRequest();
```

In deze cursus gaan we echter geen gebruik meer maken van het XMLHttpRequest, maar van Fetch!

XMLHttpRequest	**Fetch**
Ondersteuning door moderne browsers Bestaat reeds geruime tijd	Makkelijker in gebruik en krachtiger Maakt gebruik van 'Promises' Toegang overheen domeinen mogelijk. De webpagina en de bestanden kunnen op verschillende servers staan,
Slordiger en ingewikkelder Geen toegang meer overheen domeinen door diverse moderne browsers, de webpagina en de bestanden moeten op dezelfde server staan	Relatief nieuw Comptabiliteit in sommige browsers nog problematisch

3 Fetch API

3.1 GET - Data ophalen

Via de Fetch API en het HTTP Request 'GET' kunnen we data ophalen van een server. Onderstaande script geeft weer hoe we stap voor stap data ophalen en weergeven in de console. Daarnaast voorzien we ook een mogelijke error-weergave in de console.

JS-code

```
// URL/pad van de bron ingeven
fetch('URL')

// Response (data) in JSON-formaat ophalen.
.then(function(response) {
      return response.json()
})

// Response (data) in de console weergeven.
.then(function(data) {
      console.log(data)
})

// Eventuele error bij problemen in de console weergeven.
.catch(function(error) {
      console.log(error)
})
```

De Fetch API werkt met *promises*. Een promise is een object dat de uiteindelijke voltooiing of mislukking van een asynchroon event vertegenwoordigt. Het is een manier om met een asynchroon event om te gaan. Het is een soort van 'placeholder' of 'proxy' voor de response dat we asynchroon (in de toekomst) gaan krijgen.

Een promise heeft altijd één van deze drie statussen:

- *Pending,* de initiële status, de actie is niet fulfilled én niet rejected.
- *Fulfilled,* de actie is succesvol voltooid.
- *Rejected*, de actie is mislukt.

Een 'pending' promise kan zowel de status 'fulfilled' als 'rejected' krijgen. Een promise roept, wanneer 'fulfilled', de then-methode met een response-object op. Het response-object kent verschillende manieren om de response te verwerken (bvb. JSON).

Bovenstaande script kunnen we ook vereenvoudigen door te werken met de *'arrow function expression'*.

JS-code

```
fetch('URL')
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.log(error))
```

Voorbeeld 1 – Random User API

In dit voorbeeld halen we één random user op via de open-source Random User API *(meer info: https://randomuser.me/)*. De random user wordt na het uitvoeren van het script weergegeven in JSON-formaat in de console.

HTML-code

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Random User Ophalen</title>
    </head>
    <body>
        <script src="script.js"></script>
    </body>
</html>
```

JS-code

```
fetch('https://randomuser.me/api')
.then(function(response) {
    return response.json()
})
.then(function(data) {
    console.log(data)
})
.catch(function(error) {
    console.log(error)
})
```

Voorbeeld 2 – Random User API

In dit voorbeeld halen we per keer één random user op. De user wordt weergegeven op de webpagina nadat de bezoeker op de knop 'Random User Ophalen' heeft geklikt en het bijhorende event uitvoert. De nodige HTML-elementen zijn reeds voorzien op de HTML-pagina om te vullen met de data.

HTML-code

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>Random User Ophalen</title>
    </head>
    <body>
        <button id="ophalen">Random User Ophalen</button>
        <h1 id="naam"></h1>
        <p id="email"></p>
        <img src="" id="foto" />

        <script src="script.js"></script>
    </body>
</html>
```

JS-code

```
document.getElementById('ophalen').addEventListener('click',
getRandomUser);

function getRandomUser(){
    fetch('https://randomuser.me/api')
    .then(function(response) {
        return response.json()
    })
    .then(function(data) {
        document.getElementById('naam').innerHTML =
        data.results[0].name.first + ' ' + data.results[0].name.last
        document.getElementById('email').innerHTML =
        data.results[0].email
        document.getElementById('foto').src =
        data.results[0].picture.large
     })
     .catch(function(error) {
        console.log(error)
     })
}
```

3.2 POST – Data versturen

Via de Fetch API en het HTTP Request 'POST' kunnen we data versturen. Onderstaande script geeft stap voor stap weer hoe we de data versturen en bijkomend weergeven in de console. Daarnaast voorzien we ook een mogelijke error-weergave in de console.

JS-code

```
fetch('URL', {
     // Geef de methode op die we gebruiken.
     method: 'POST',
     // Voeg de HTTP Headers toe.
     headers: {
     "Content-type": "application/json; charset=UTF-8"
     },
     // Voeg de body/inhoud toe in JSON-formaat.
     body: JSON.stringify({
          inhoud: "Dit is de inhoud"
     })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.log(error))
```

Voorbeeld 1 – JSON Place Holder

In dit voorbeeld versturen we een post met een vaste titel en inhoud via de open-source *REST API* 'JSON Place Holder' (*meer info: http://jsonplaceholder.typicode.com/*). De bezoeker van de webpagina heeft hierbij geen invloed op de post en de inhoud ervan. In de console kunnen we de post bekijken met de opgegeven userID.

HTML-code

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <title>Een post plaatsen</title>
</head>
<body>
    <script src="script.js"></script>
</body>
</html>
```

JS-code

```
fetch('http://jsonplaceholder.typicode.com/posts', {
    method: 'POST',
    headers: {
    "Content-type": "application/json; charset=UTF-8"
    },
    body: JSON.stringify({
        title: "Dit is de titel van mijn post",
        body: "Dit is de inhoud van mijn post",
        userId: 1
    })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.log(error))
```

REST API oftewel 'RESTful API' staat voor **Re**presentational **S**tate **T**ransfer **A**pplication **P**rogram **I**nterface en is een architectuurstijl waarmee software kan communiceren met andere software via een netwerk of op eenzelfde apparaat.

Doorgaans gebruiken developers REST API's om webservices te bouwen. Deze webservices gebruiken HTTP-methoden om data tussen een client en een server op te halen, te posten, te verwijderen, …

REST omvat een reeks principes die de developer moet volgen vooraleer hij zijn API als 'RESTful' kan noemen.

Voorbeeld 2 – JSON Place Holder

In dit voorbeeld versturen we een post met een titel en stukje inhoud ingevuld door de bezoeker in een webformulier. De post wordt verzonden na het klikken op de knop 'Verzenden'. In de console kunnen we de post bekijken met de opgegeven userID.

Opgelet: de post wordt niet écht toegevoegd. Via de preventDefault-methode stoppen we de actie die bij het event hoort, namelijk het submitten van de post. We krijgen wel het nodige te zien in de console, waardoor we de Fetch API met de HTTP Reauest 'POST' kunnen testen.

HTML-code

```
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8'>
    <title>Reactie posten</title>
</head>
<body>
    <h1>Plaats hier een reactie!</h1>
    <form id="toevoegenPost">
        <div>
```

```
            <input type="text" id="titel">
        </div>
        <div>
            <textarea id="inhoud"></textarea>
        </div>
            <input type="submit" value="Verzenden">
    </form>

    <script src="script.js"></script>
</body>
</html>
```

JS-code

```
document.getElementById('toevoegenPost').addEventListener
('submit', toevoegenPost);

function toevoegenPost(e) {
    e.preventDefault();

    fetch('http://jsonplaceholder.typicode.com/posts', {
        method: 'POST',
        headers: {
        "Content-type": "application/json; charset=UTF-8"
        },
        body: JSON.stringify({
            title: document.getElementById('titel').value,
            body: document.getElementById('inhoud').value,
            userId: 5
        })
    })
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.log(error))
}
```