Fetch API, Callbacks en Promises

Inleiding

Webbrowsers zijn zo ontworpen dat er na het laden van een website geen internetverbinding meer nodig is. Zolang je alleen maar leest en de website niet interactief gebruikt, is er geen network nodig want alle HTML, CSS en JavaScript zitten in het geheugen van de browser. Alles staat lokaal op het apparaat waarmee je werkt. Traditionele websites werken op deze manier. Als je verse informatie wilt zien, moet je de hele website verversen, waarmee alle onderdelen opnieuw worden opgehaald.

Moderne websites reageren continu op de activiteiten van de gebruiker. Iets eenvoudigs als scrollen kan al gekoppeld worden aan een actie die extra informatie ophaalt en daarmee de website aanvult of verandert.

Nu is het wel belangrijk dat de code die wordt uitgevoerd niet de hele website laat bevriezen omdat de code langer duurt dan verwacht. Daarom zijn er technieken in JavaScript waarmee dit voorkomen kan worden. Bij elkaar vallen deze technieken onder de naam **AJAX**: **A**synchronous **J**avaScript **A**nd **X**ML. Asynchroon voor technieken die zelfstandig hun werk doen en XML voor de standaardvorm van de uitgewisselde gegevens.

De oudste techniek is XMLHttpRequest. Dit standaardobject bevat alle methoden en variabelen om met ander bestanden of websites te kunnen communiceren. Het vergt alleen een flinke studie van dit object om het te kunnen gebruiken. Daarom is de Fetch API toegevoegd om het afhandelen van dergelijke verzoeken direct met JavaScript-instructies te kunnen doen.

Fetch API vs. XMLHttpRequest

De fetch()API is vrij nieuw en is de verbeterde manier om asynchroon data op te halen of te verzenden. Het is daarom beter om deze manier te gebruiken en *niet* meer het oude XMLHttpRequest. Toch zijn beide technieken nog beschikbaar en is er met het complexere XMLHttpRequest meer controle over de uitwisseling van gegevens uit te oefenen.

Voorbeeld XMLHttpRequest API:

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    // Alleen uitvoeren als het document succesvol geladen is
    if (this.readyState == 4 && this.status == 200) {
       document.getElementById("demo").innerHTML = xhttp.responseText;
     }
};
xhttp.open("GET", "readme.txt", true);
xhttp.send();
```

Voorbeeld Fetch API:

```
async function haalTekstUitBestand() {
    let response = await fetch('readme.txt');

    if (response.status === 200) {
        let data = await response.text();
        return (data);
    }
}
document.getElementById("demo").innerHTML = haalTekstUitBestand();
```

Referenties:

XMLHttpRequest - Web APIs | MDN (mozilla.org)
Fetch API - Web APIs | MDN (mozilla.org)

Callback functions

In JavaScript kunnen functies, net als variabelen, worden doorgegeven aan andere functies. Dit principe heet 'callback'. De functie die wordt doorgegeven, wordt hierdoor een lokale functie binnen een andere functie en kan worden aangeroepen met de naam die als argument is opgegeven.

De voornaamste reden voor het bestaan van callback-functies is de herbruikbaarheid van code. Functies kunnen vooraf worden klaargezet, en waar nodig worden meegegeven aan andere functies. Dat voorkomt dat vergelijkbare code binnen iedere functie herhaald moet worden.

Callback-functies worden ook veel gebruikt met asynchrone functies zoals setTimeout().

```
function mijnFunctie(value) {
  document.getElementById("demo").innerHTML = value;
}

setTimeout(function() {
     mijnFunctie("Deze tekst verschijnt na 3 seconden");
}, 3000);
// overige code gaat direct verder
```

Een nadeel van het gebruik van callback-functies is dat als er gewacht moet worden op het resultaat van de callback dit 'genest in de callback moet gebeuren (een functie als argument van een tweede functie etc). Dit is onder programmeurs bekend geworden als 'callback hell', eenvoudig omdat de code lastiger leesbaar wordt met meer functies-in-functies-in-functies.

Bekijk deze code, waarbij iedere regel moet wachten op de voorgaande:

```
Uitkomst1 = haalData();
Uitkomst2 = haalMeerData(Uitkomst1);
Uitkomst3 = haalMeerData(Uitkomst2);
```

Als we dit bijvoorbeeld willen aanroepen via setTimeOut om het asynchroon te doen, ziet het er zo uit:

```
haalData(function(Uitkomst1){
    haalMeerData(Uitkomst1 , function(Uitkomst2){
        haalMeerData(Uitkomst2, function(Uitkomst3){
            ...
        });
    });
});
```

Asynchrone verwerking zal de code sneller maken, maar maakt het ook veel lastiger om te debuggen.

Referenties:
[Callback function - MDN Web Docs Glossary: Definitions of Web-related terms | MDN (mozilla.org)]

Promises

Vanaf JavaScript versie ES5 kan er gebruikgemaakt worden van de Promise. Promises zijn een manier om asynchrone code te schrijven die nog steeds lijkt alsof deze top-down wordt uitgevoerd en die meer soorten fouten afhandelt als gevolg van gebruik van foutafhandeling in try/catch-stijl: Probeer (try) code, en als er iets misgaat, zeg je vooraf alvast wat er dan moet gebeuren (catch).

```
try {

      alert('Probeer deze code');

      teller; // error, variable is niet gedefinieerd

      alert('Hier komt de code nooit');

} catch (err) {
  alert('ERROR: De variabele was niet gedefinieerd');
}
```

In het woord promise (belofte) schuilt de essentie van het concept, wat je belooft, moet je ook doen. Een promise wordt dus op enig moment óf ingelost, óf afgewezen, wat te zien is in de twee argumenten van Promise:

```
const mijnPromise = new Promise((bijSucces, bijError) => {

      bijSucces("gelukt");

      bijError("mislukt");

});

mijnPromise.then(

  function(value) { /* code bij succes */ },

  function(error) { /* code bij een error */ }

);
```

Een groot verschil met callback-functies is dat ze niet genest worden, maar via 'chaining' gekoppeld worden en daarmee een overzichtelijke en goed te onderhouden asynchrone keten kunnen vormen. Dit 'chainen' gebeurt met `.then()`.

Promise chaining wordt onmisbaar wanneer we een reeks asynchrone taken achter elkaar moeten uitvoeren. Elke taak in de keten kan pas beginnen zodra de vorige taak is voltooid en gecontroleerd door `.then`. Belangrijk hierbij is dat het resultaat van elke taak in de vorm van het 'promise'-object aan de volgende stap wordt doorgegeven. Zo creëert u als het ware een lopende band waarover de desbetreffende data getransporteerd wordt en steeds bewerkt. Doe *dit* en dan (.then) *dat* (.then) *dat* (.then) dat etc.

Die `.then`-blokken zijn zo geprogrammeerd dat ze altijd weer een promise retourneren, die vervolgens op elke `.then` in de keten wordt toegepast.

Alles wat asynchroon geretourneerd wordt, wordt uiteindelijk een 'resolved promise' die aan een andere functie doorgeven kan worden, óf een 'rejected promise' (afhankelijk van een error, en afkomstig van `.catch` blokken) die een error routine kan aanspreken.

```
function toonOpScherm(tekst) {
	document.getElementById("demo").innerHTML = tekst;
}

let myPromise = new Promise(function(myResolve, myReject) {
	let x = 0;

// << Hier code die langer of korter kan duren, daarom liever asynchroon

	if (x == 0) {
		myResolve("OK");
	} else {
		myReject("Error");
	}
});

myPromise.then(
	function(value) {toonOpScherm(value);},
	function(error) {toonOpScherm(error);}
);
```

Bronnen:

Promise - JavaScript | MDN (mozilla.org)

Fetch API en promise

De `fetch()` API retourneert overigens ook een promise. Hieronder een zeer beknopt, maar erg bruikbaar voorbeeld, ook voor je eigen memory game:

```
fetch("http://example.com/kaartjes.json")
 .then(response => response.json())
 .then(data => myLocalDataObject = data)
 .catch((error) => {
 console.error("Error:", error);
 });
```

AJAX

Als er gevraagd wordt of je AJAX beheerst, wordt er eigenlijk bedoeld of er ervaring is met bovenstaande technieken. AJAX is slechts een verzamelterm voor deze technieken.

De keuze voor de juiste techniek hangt af van je eigen kennis en ervaring. Voor iedere uitdaging zullen oplossingen zijn met alle technieken, en zolang de website er niet door vastloopt, kun je ervan uitgaan dat jouw keuze voor de oplossing de juiste is. Vergeet alleen niet goed te documenteren, want voor een ander zal jouw AJAX-oplossing misschien lastig te volgen zijn door het asynchrone aspect van deze techniek.

Referenties:

Ajax - Developer guides | MDN (mozilla.org)