Lezione 9: XMLHttpRequest e Fetch

Connessione e Scambio Dati con un Server

- Applicazioni Web necessitano di scambiare dati con il server in background
- È il sogno degli sviluppatori, permette di
  - Aggiornare una pagina senza ricaricarla
  - Richiedere i dati da un server dopo che la pagina è stata caricata
  - Ricevere i dati dal server dopo che la pagina è stata caricata
  - Inviare dati al server in background
- Tutti i browser moderni (Firefox, Chrome, Safari, and Opera) supportano scambio di dati in background
  - XMLHttpRequest
  - Fetch-Promise

XMLHttpRequest

XMLHttpRequest Object

- Nonostante il nome, tramite questo oggetto è possibile ricevere dati anche in formati diversi dall'XML, come il JSON o il formato binario
  - Il nome deriva dal fatto che all'epoca l'XML era il modo principale attraverso il quale si trasmettevano dati

- Supporta diversi protocolli oltre all'*HTTP* e all'*HTTPS* anche *file* e *FTP*

- E' la colonna portante del paradigma **AJAX** (**A**synchronous **J**avascript **A**nd **X**ML)

AJAX – Come funziona

1. Si verifica un evento su di una pagina, ad esempio il click su un bottone
2. Viene creato un oggetto XMLHttpRequest
3. L'oggetto in questione viene utilizzato per effettuare una richiesta al server
4. Il server processa una richiesta
5. Il server ritorna una risposta
6. La risposta viene ricevuta dall'oggetto
7. In base al contenuto della risposta vengono effettuate altre operazioni sulla pagina

AJAX – Status Codes

- Ogni volta che si effettua una richiesta il server risponde al client con un appropriato status code in base al risultato della richiesta
- Appartengono a 5 grandi famiglie
  - **1xx – Informational**: la richiesta è stata ricevuta e il processing continua
  - **2xx – Successful**: la richiesta è stata ricevuta, capita e accettata
  - **3xx – Redirection**: per processare la richiesta si devono intraprendere altre azioni
  - **4xx – Client Error**: la richiesta contiene un errore di sintassi o non può essere completata
  - **5xx – Server Error**: il server non può rispondere nonostante la richiesta sia valida

AJAX – Status Codes

- Tra gli status codes più comuni possiamo annoverare:
  - **100 – Informational**
    - 100 – Continue: generalmente utilizzato quando la richiesta è grande e quindi ritornato appena il server riceve gli headers. Serve a indicare che ora è possibile inviare anche il body
  - **2xx - Successful**
    - *200 – OK*: risposta standard per una richiesta andata a buon fine
    - *201 – Created*: richiesta andata a buon fine, è stata creata la nuova risorsa
  - **3xx - Redirection**
    - *301 – Moved Permanently*: la richiesta deve essere inoltrata al nuovo URI ritornato
    - *302 – Found*: dice al client che la risorsa si può raggiungere tramite un altro URL
    - *304 – Not Modified*: indica che la risorsa non è stata modificata e quindi viene tornata una copia dalla cache

AJAX – Status Codes

- **4xx – Client Error**
  - *400 – Bad Request*: il server non processa la richiesta perché c'è un errore del client (sintassi sbagliata, richiesta con un body troppo grande…)
  - *401 – Unauthorized*: simile alla 403, ma specifica per risorse che richiedono un'autenticazione esplicita
  - *403 – Forbidden*: la richiesta è valida, ma il server si rifiuta di procedere
  - *404 – Not Found*: la risorsa richiesta non è stata trovata

- **5xx – Server Error**
  - *500 – Internal Server Error*: messaggio di errore generico quando il server ha un errore inaspettato
  - *501 – Not Implemented*: il server non riconosce il metodo della richiesta
  - 5*03 – Service Unavailable*: il server è temporaneamente irraggiungibile

XMLHttpRequest

- **Costruttore**
  - XMLHttpRequest()
    - Deve essere chiamato prima di ogni altro metodo per avere un'istanza dell'oggetto da utilizzare

- **Metodi(principali)**
  - *abort()*
    - Cancella la richiesta anche se è già stata fatta la richiesta
  - *getAllResponseHeaders()*
    - Ritorna tutti gli header della risposta
  - *getResponseHeader()*
    - Ritorna uno specifico header della risposta.
      - Es. *client.getResponseHeader('Content-Type')*

XMLHttpRequest

- **Metodi(principali)**
  - *open(method, url, async, user, password)*
    - *Serve per inizializzare la richiesta o reinizializzare una richiesta già esistente*
    - Solo method e url sono necessari. Async se non specificata è messa a *true* come default
    - **Deve essere chiamato prima del metodo send()**
  - *send(body)*
    - *Effettua la richiesta al server. Il body viene utilizzato solo nel caso venga effettuata una richiesta di tipo **POST***
  - *setRequestHeader(header, value)*
    - *Setta il valore dell'header in base al value inserito*

XMLHttpRequest

- **Proprietà**
  - *readyState*
    - Contiene un valore che fornisce lo stato della richiesta che è stata effettuata
      - 0: richiesta non inizializzata
      - 1: connessione con il server stabilita
      - 2: richiesta ricevuta
      - 3: richiesta che è in fase di lavorazione
      - 4: richiesta conclusa e risposta inviata
  - *responseText*
    - Contiene i dati della risposta del server in formato testuale
  - *responseXML*
    - Contiene i dati della risposta del server come un oggetto XML

XMLHttpRequest

- **Proprietà**
  - *response*
    - Contiene i dati della risposta del server in base al ResponseType definito nell'oggetto
  - *status*
    - Contiene lo stato della risposta del server utilizzando la convenzione numerica. Es. 200 – OK, 403 – Forbidden, 500 – Internal Server Error
  - *statusText*
    - Ritorna lo stato della risposta in formato testuale. Es. (OK, Not Found…)

XMLHttpRequest

- **Proprietà**
  - *onreadystatechange*
    - La funzione assegnata a questa proprietà viene chiamata ogni volta che lo stato della richiesta cambia.
    - Deve essere definita **prima** della open()
  - *onload, onabort, onerror, onloadstart, onprogress*
    - Queste proprietà accettano delle funzioni di callback esattamente come la onreadystatechange, ma invece di essere chiamate ad ogni cambio di stato vengono lanciate solo quando la richiesta entra nello stato specifico. Dessere definite **prima** della open()
    - Es. la *onload* chiama la funzione solo quando la richiesta ha terminato ed ha ricevuto una risposta dal server

XMLHttpRequest

- **Proprietà**
  - *timeout(duration)*
    - Specifica in ms quanto tempo può durare una richiesta prima di essere automaticamente terminata
- **EventListener**
  - *addEventListener(eventName, listener)*
    - Questo EventListener permette di associare una funzione di callback allo specifico evento.
    - Deve essere specificato **prima** della open()
    - Questo tipo di listener non funziona su tutti i browser quindi ricordarsi di controllare la compatibilità

XMLHttpRequest – Esempio 1 (XML)

- xmldoc_doc.html
- Definizione struttura pagina HTML

```
<div>
    <b>To:</b> <span id="to"></span><br>
    <b>From:</b> <span id="from"></span><br>
    <b>Message:</b> <span id="message"></span>
</div>
```

XMLHttpRequest – Esempio 1 (XML)

- Creazione oggetto XMLHttpRequest
  - xmlhttp=new XMLHttpRequest();

XMLHttpRequest – Esempio 1 (XML)

- Caricamento file XML

  - ```
    xmlhttp.open("GET","https://homes.di.unimi.it//note.xml",false);
    xmlhttp.send();
    xmlDoc=xmlhttp.responseXML;*
    ```

*xmlhttp.responseText può essere usato quando la risposta non è basata su XML (ad esempio JSON)
    ```

XMLHttpRequest – Esempio 1 (XML)

- Caricamento dati da file XML

  - document.getElementById("to").innerHTML=xmlDoc.getElementsByTagName("to")[0].childNodes[0].nodeValue;

    document.getElementById("from").innerHTML=xmlDoc.getElementsByTagName("from")[0].childNodes[0].nodeValue;

    document.getElementById("message").innerHTML=xmlDoc.getElementsByTagName("message")[0].childNodes[0].nodeValue;

XMLHttpRequest – Esempio 2 (XML)

- Caricamento XML File
- Parsing di una stringa XML contenente alcune note in un oggetto XML DOM
  - http://www.w3schools.com/xml/tryit.asp?filename=tryxml_parsertest

```
<notes>
 <note>
  <to>…</to>
  <from>…</from>
  <message>…</message>
 </note>
 …
</notes>
```

XMLHttpRequest – Esempio 6 (XML)

- XMLHttpRequest asincrona Lancia una callback quando il dato è stato interamente ritornato

```
var xmlhttp = new XMLHttpRequest(),
   method = "GET",
   url = "note.xml";

xmlhttp.open(method, url, true);
xmlhttp.onreadystatechange = function () {
  if(xmlhttp.readyState == 4 && xmlhttp.status === 200) {
    console.log(xmlhttp.responseXML);
  }
};

xmlhttp.send();
```

La proprietà XMLHttpRequest.onreadystatechange contiene l'event handler che deve essere invocato quando l'evento readystatechange si verifica, ovvero ogni volta in cui la proprietà readyState del XMLHttpRequest viene modificata

https://homes.di.unimi.it/xmldoc_async.html

XMLHttpRequest – Esempi JSON

- **Per i nostri esempi utilizzeremo delle API pubbliche**
  - **https://swapi.dev/**
    - **Queste API forniscono informazioni sui film di Star Wars**
  - **Documentazione:**
    - **https://swapi.dev/documentation**

XMLHttpRequest – Esempio 1 (JSON)

- **Esempio 1**
  - Ricevere le informazioni del film con id=1 e stampare il risultato in console al termine utilizzando la onreadystatechange

```
var req = new XMLHttpRequest();
req.onreadystatechange = function() {
      if (this.readyState == XMLHttpRequest.DONE && this.status == 200) {
         console.log(this.response);
      }
   };
req.open("GET", "https://swapi.dev/api/films/1/");
req.send();
```

XMLHttpRequest – Esempio 2 (JSON)

- **Esempio 2**
  - Ricevere le informazioni del personaggio con con id=4 e stampare il risultato in console al termine utilizzando la onload

```
var req = new XMLHttpRequest();
req.onload = function() {
        if(this.status == 200) {
          console.log(this.response);
        }
   };
req.open("GET", "https://swapi.dev/api/people/4/");
req.send();
```

XMLHttpRequest – Esempio 3 (JSON)

- **Esempio 3**
  - Ricevere le informazioni del film con id=1 e utilizzare il listener di eventi

```
function reqListenerLoad (evt) {
  if(evt.status==200) {
    console.log("HO FINITO");
  } else {
    console.log("QUALCOSA NON VA");
  }
};

function reqListenerAbort (evt) {
  console.log("ABORT");
};

function reqListenerError (evt) {
  console.log("HO FINITO CON ERRORE");
};
```

XMLHttpRequest – Esempio 3 (JSON)

```
function reqListenerProgress (evt) {
  if (evt.lengthComputable) {
    var percentComplete = evt.loaded / evt.total * 100;
    console.log(percentComplete);
  } else {
    console.log("Sto avanzando");
  }
};

var req = new XMLHttpRequest();
req.addEventListener("load", reqListenerLoad);
req.addEventListener("abort", reqListenerAbort);
req.addEventListener("error", reqListenerError);
req.addEventListener("progress", reqListenerProgress);
req.open("GET", "https://swapi.dev/api/films/1/");
req.send();
```

Fetch

Introduzione

- Alternativa moderna a XMLHttpRequest per scambiare dati con il server in background (asincrono)

- Fetch
  - Nativa e supportata da tutti i browser moderni
  - Più semplice e meno verbosa per richieste asincrone e gestione delle risposte corrispondenti
  - Supporta promise (no callback)

Promises

- Le promises tentano di risolvere il problema di mettere in comunicazione un codice produttore con un codice consumatore
  - Un produttore svolge un' operazione che richiede tempo (Ad esempio il download di un file da un server)
  - Un consumatore è un'operazione che necessita del risultato del produttore
  - Una promise è un particolare oggetto che connette il produttore con il consumatore, rendendo il risultato disponibile al consumatore quando il produttore ha terminato

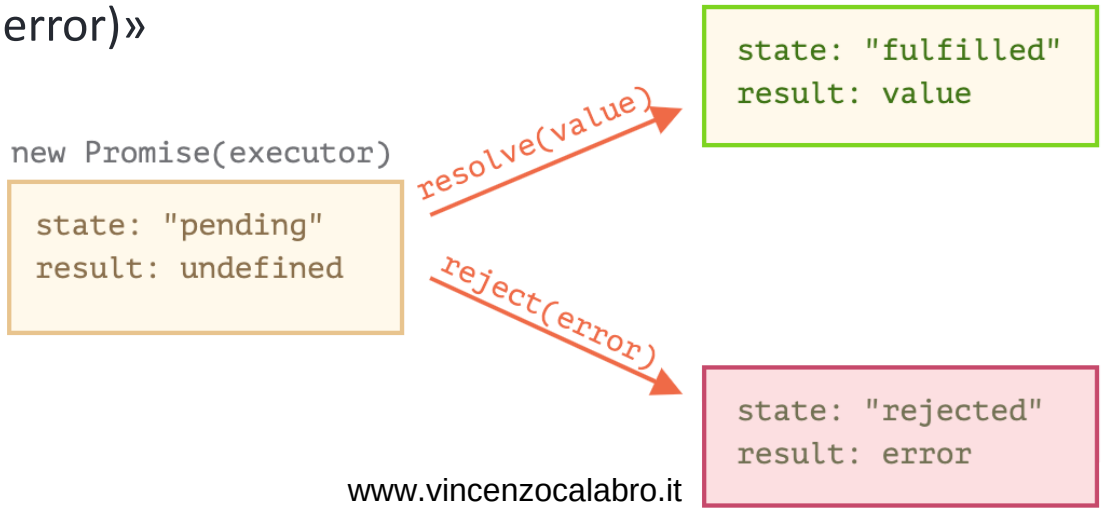
Promises

```
var promise = new Promise(function(resolve, reject) {
  // codice produttore
});
```

- I due argomenti della funzione sono delle callback che vengono eseguite quando la promise conclude l'operazione. Quando l'operazione viene conclusa la promise può essere risolta o rifiutata:
  - risolta - resolve(value) - se il processo termina correttamente con il valore passato come parametro
  - rifiutata - reject(reason) - se il processo termina con un errore

Promises

- Una Promise possiede anche alcune proprietà interne
  - state – inizialmente ha valore «pending», cambia in:
    - «fulfilled» se la promise è risolta
    - «rejected» se la promise è rifiutata
  - result – inizialmente ha valore «undefined», cambia in:
    - «value» se la promise è risolta e cioè a seguito della chiamata di «resolve(value)»
    - «error» se la promise è risolta e cioè a seguito della chiamata di «reject(error)»



Fetch

- La sintassi di base è:

```
var promise = fetch(url, [options])
```

  - *url* – l'URL da raggiungere
  - *options* – parametri **opzionali**: metodi, header, etc.
    - Senza specificarne nessuna , la richiesta è una semplice GET che scarica il contenuto di *url*.
- La funzione restituisce una promise
- Ottenere una risposta è comunemente un processo che si svolge in due fasi:
  1. Valutazione dello stato
  2. Risposta

1 – Valutazione dello stato

- Possiamo valutare lo stato HTTP dalle proprietà:
  - *status* – codice di stato HTTP (es: 200)
  - *ok* – boolean, true se lo stato HTTP è compreso fra 200 e 299
- Esempio:

```
var response = await fetch(url);

if (response.ok) { // se l'HTTP-status è 200-299
  var json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

2 – Risposta

- La risposta vera e propria è un oggetto che fornisce molteplici metodi per accedere al contenuto in diversi formati:
  - response.text() – legge la risposta e la restituisce come stringa
  - response.json() – legge la risposta (che deve essere in formato JSON) e la restituisce come oggetto javascript
  - response.formData() – legge la risposta e la restituisce come oggetto FormData
  - response.blob() – legge la risposta e la restituisce come Blob
- Esempio:

```
fetch('https://swapi.dev/api/films/')
  .then(response => response.json())
  .then(results => alert(results.results[0].title));
```

Richieste POST (1)

- Per eseguire una richiesta POST o di altro tipo (PUT, PATCH, DELETE, …), è possibile utilizzare le opzioni fornite dalla funzione fetch:
  - method – metodo HTTP da utilizzare
    - ES: POST, PUT, …
  - body– il contenuto della richiesta da inviare al server
    - Es: una stringa JSON, i dati di un form
- Per esempio, il codice che segue invia un oggetto rappresentante un utente sotto forma di JSON

Richieste POST (2)

```
var user = {
  nome: 'Valerio',
  cognome: 'Bellandi'
};

var response = await fetch('esempio.com/utenti/', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

var result = await response.json();
alert(result.message);
```

Conclusioni

- XML e JSON come linguaggi di definizione e rappresentazione delle informazioni

- XMLHttpRequest per scambiare dati con il server in background