Ajax e la programmazione asincrona

Rev 6.3 del 15/03/2025

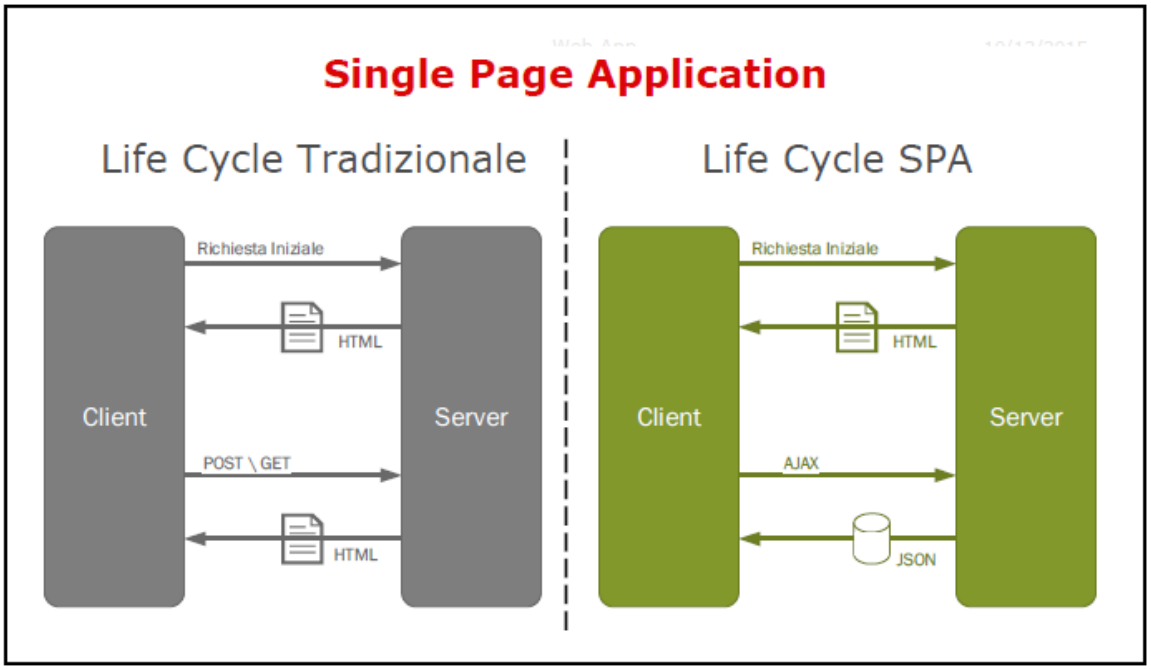
Introduzione ai Web Services

Un web service (detto anche API = Application Programming Interface), *da non confondere con web server*, è una applicazione in esecuzione su un web server che, a fronte di una richiesta http, restituisce un insieme di dati (in formato XML o JSON).

Il concetto di **web service** nasce negli anni 2004-2005 quando ci si rese conto che ricaricare ogni volta una intera pagina HTML risultava molto oneroso, mentre sarebbe stato molto più semplice e conveniente ricaricare di volta in volta soltanto i dati necessari a seconda dei contesti.

AJAX è la tecnologia utilizzata da un client per richiedere i dati esposti da un Web Service.

Insieme ad Ajax nasce anche il concetto di SPA **Single Page Application**. Il client effettua una unica richiesta iniziale di pagina e poi provvede ad aggiornare SOLO i dati tramite richieste ajax successive.



Elementi Base di Ajax

Asynchronous JavaScript And **XML**

Ajax nasce in ambito web avente come scopo principale quello di consentire l'aggiornamento dinamico dei dati contenuti in una pagina html senza dover ricaricare l'intera pagina con conseguente inutile speco di risorse. In questo modo le applicazioni risultano molto più veloci e la quantità di dati scambiati fra client e server si riduce notevolmente.

A dispetto del nome, la tecnologia AJAX oggi:

- oltre a javascript, è disponibile in quasi tutti gli ambienti di programmazione (ad esempio nelle **applicazioni desktop** scritte in C# e nelle **app per smartphone** scritte in Android o IOS)
- può scambiare dati in qualsiasi formato (XML, JSON, CSV o anche semplice testo)

Per inviare le richieste ajax **non** si utilizza il pulsante di submit, ma un **normale button html** che richiama una procedura java script, la quale :

1. **invia la richiesta**
2. **attende i dati**
3. **li visualizza all'interno della pagina.**

Una caratteristica fondamentale di AJAX è che si tratta di una tecnologia **asincrona** nel senso che la richiesta viene inviata in background all'interno di un 'thread' separato senza interferire con il comportamento della pagina. L'utente, una volta inviata la richiesta, può continuare ad interagire con l'interfaccia grafica anche mentre l'applicazione è in attesa dei dati.

Concetto contrapposto al modello tradizionale di **comunicazione sincrona** tipica delle vecchie applicazioni web (le cosiddette applicazioni Web Form) in cui, in corrispondenza del submit, viene inviata una richiesta al server rimanendo poi in attesa del caricamento della nuova pagina.

Tipi di chiamate http

Per inviare una richiesta ajax ad un web service sono disponibili principalmente due tipi di chiamate :

- **GET**
- **POST**

In entrambi i casi occorre 'passare' al web service alcune informazioni riguardo ai dati che si vogliono richiedere. Ad esempio 'mandami tutte le informazioni dello studente Pippo'. Pippo dovrà essere passato come **parametro** al web service. La differenza sostanziale tra i metodi GET e POST consiste nel come vengono inviati i parametri al web service:

- nel caso delle chiamate GET i parametri vengono trasmessi al server in modo visibile **in formato URL-encoded** in coda alla URL Es: http://miosito.it/studenti**?**nome=MarioRossi**&**classe=1C
- nel caso delle chiamate POST i parametri vengono trasmessi al server in forma nascosta all'interno del body della http request (che nel caso delle chiamate GET rimane vuoto)
  Nelle chiamate POST i parametri possono essere trasmessi in qualsiasi formato.
  Il più usato è il formato **JSON** trasmesso in forma serializzata.

Tipologie di web services

I web service basati direttamente sui metodi http GET e http POST e sul concetto di **risorsa** sono detti

- **REST** *REpresentational State Transfer* basati sul concetto di collegamento fra **risorse**

Esisteva in passato un'altra tipologia di web service oggi sempre meno diffusi detti

- **SOAP** *Simple Object Access Protocol* basati sul concetto di **servizio**

L'oggetto XMLHttpRequest

In javascript l'oggetto base per l'invio di una richiesta Ajax ad un web server è l'oggetto **XMLHttpRequest** il cui primo standard ufficiale è stato rilasciato dal World Wide Web Consortium (**W3C**) il 5 aprile 2006.

Un semplice esempio di utilizzo di Ajax potrebbe essere la scelta di un nuovo nickname in fase di creazione di un nuovo account su un sito web. In corrispondenza di ogni carattere digitato si invia una richiesta al server chiedendo se il nickname fino a quel momento inserito è valido oppure no.

A tale scopo può essere utilizzato l'evento onKeyUp della Text Box.

- in corrispondenza di ogni carattere, java Script invia in tempo reale una richiesta al server
- Il server valuta se il nome fin'ora inserito è valido o no elaborando una risposta molto 'leggera'.
- In base alla risposta Java Script aggiorna opportunamente l'aspetto del textbox

Invio della richiesta

```
var richiesta = new XMLHttpRequest();
var url=”controlla.php?parametro=” + encodeURIComponent(txtUsername.value);

// apro la connessione TCP con il server
richiesta.open("GET", url, true);

// le requestHeader possono essere assegnate solo DOPO l’apertura della connessione
richiesta.setRequestHeader(
          “Content-type”, “application/x-www-form-urlencoded; charset=utf-8”);

// funzione di callback da eseguire in corrispondenza della risposta
richiesta.onreadystatechange = aggiorna;
richiesta.send(null);
```

Parametri del metodo open

1. Il metodo con cui inviare i parametri al server (GET / POST)
2. La url della risorsa richiesta. La url può essere espressa in due modi:
   - path relativo a partire dalla cartella corrente. Es **`url=”controlla.php”`**
   - path assoluto a partire dalla cartella htdocs. Es **`url=”/5B/ese12/controlla.php”`**

   La funzione **`encodeURIComponent`** consente di codificare eventuali caratteri speciali.
3. La modalità di esecuzione della send (true=asincrona, false=sincrona). Se si intende gestire una funzione di callback per la lettura della risposta, l’invio dovrà necessariamente essere asincrono.

Attributi da associare alla richiesta

- L’attributo **`setRequestHeader`** definisce il formato dei parametri da inviare al server.
- L’attributo **`onreadystatechange`** consente di definire un riferimento alla funzione JavaScript di callback che dovrà essere eseguita in corrispondenza del ricevimento della risposta.
- La funzione di callback **NON è obbligatoria**. **Il server (nel caso ad es di comandi DML) può anche NON inviare una risposta, nel qual caso la funzione di callback deve essere omessa**.

Il metodo send( )

- Il metodo **`send`** consente di inviare la richiesta al server. Il parametro del metodo send vale:
  - **null** nel caso di richieste di tipo GET (come quella attuale)
  - nel caso delle richieste POST contiene l’elenco dei parametri in formato nome=valore scritti all’interno di una unica stringa e separati da & (oppure scritti in formato json o formData).

Poiché il client potrebbe inviare una nuova richiesta prima che sia giunta la risposta alla richiesta precedente, è buona regola istanziare un apposito oggetto **`XMLHttpRequest`** per ogni comunicazione, oppure inviare la nuova richiesta soltanto in corrispondenza del ricevimento della risposta precedente.

Gestione della risposta

La risposta testuale elaborata dal server viene restituita all'interno della proprietà **.responseText** dell'oggetto richiesta.

```
function aggiorna(){
     if (richiesta.readyState==4 && richiesta.status==200)
                  alert(richiesta.responseText);
```

La funzione aggiorna() può essere richiamata più volte nel corso della comunicazione.
In corrispondenza delle varie chiamate il parametro readyState può assumere valori differenti :

```
0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready
```

Se la riposta è pronta (readyState=4) e lo stato è corretto, allora si può leggere il contenuto della risposta.

Aggiornamento della pagina

Supponendo di usare un servizio lato server per il controllo di uno username e supponendo che tale servizio risponda :

- "OK" in caso di username valido
- "NOK" in caso di username non valido

lato client si può scrivere la seguente procedura di visualizzazione:

```
function aggiorna () {
    if (richiesta.readyState==4 && richiesta.status==200) {
        var msg = $("#msg");
        var btn = $("#btnInvia");
        var risposta = richiesta.responseText;

        if (risposta.toUpperCase() == "OK") {
            msg.text("Nome valido");
            msg.css("color", "green");
            btn.prop("disabled", false);
        }
        else if (risposta.toUpperCase() == "NOK"){
            msg.text("Nome gi&agrave; esistente");
            msg.css("color", "red");
            btn.prop("disabled", true);
        }
        else
            alert("Risposta non valida \n"+risposta);
    }
```

La programmazone asincrona : l'oggetto Promise

Si tratta di un oggetto mirato a facilitare la gestione dei processi asincroni. Tutti gli oggetti moderni utilizzati per l'invio di richieste ajax restituiscono al chiamante una **Promise**.

Con il termine **programmazione sincrona** si intende quella programmazione tradizionale in cui ogni operazione, prima di essere eseguita, attende la terminazione dell'operazione precedente.

La **programmazione asincrona** interviene nel momento in cui si eseguono operazioni particolarmente pesanti che altrimenti bloccherebbero l'interfaccia grafica; ad esempio l'elaborazione grafica di una grande immagine oppure l'attesa di ricezione dati da un server esterno. In questi casi l'applicazione lancia il comando di esecuzione e prosegue nell'esecuzione delle istruzioni successiva. Il sistema si prende **in carico** di invocare una apposita funzione di callback al momento opportuno, cioè al temine dell'elaborazione dell'immagine oppure al momento del ricevimento dei dati dal server.

L' oggetto **Promise** consente di "wrappare" una procedura asincrona all'interno di una nuova procedura all'apparenza sincrona, consentendo così una maggiore leggibilità del codice.

Si supponga di avere una procedura asincrona **elaboraImmagine**() molto pesante in termini di esecuzione e n. di righe di codice. La riga `imgLibrary.elaboraImmagine(img)` lancia l'elaborazione dell'immagine e poi termina immediatamente Quando l'immagine sarà stata generata verrà generato un evento **onEnd** a cui viene iniettata l'immagine elaborata.

```
function elabora(img){
      imgLibrary.elaboraImmagine(img)
      imgLibrary.onEnd(finalImage){   // elaborazione terminata
            console.log(finalImage)
            return finalImage !!!
      }
}
```

Se questa **finalImage** dovesse poi essere gestita dal main, il codice di gestione della **finalImage** **dovrebbe** essere scritto all'interno della callback finale. **Una procedura di evento non può eseguire un return !** Un eventuale return può essere inserito soltanto ALLA FINE della procedura **elabora()**, dove però la variabile **finalImage** non è disponibile.

A livello di leggibilità (ma anche di programmazione) sarebbe molto più comodo se **elaboraImmagine** fosse sincrona, cioè bloccante, bloccando il programma fino al termine dell'elaborazione e restituendo il risultato al main:

```
finalImage = elaboraImmagine(img)
```

Nella programmazione web questo però non è possibile perché bloccherebbe l'interfaccia grafica per tutto il tempo di elaborazione. A questo scopo intervengono le Promise che, come detto all'inizio, consentono di "wrappare" una procedura asincorna all'interno di una nuova procedura soltanto **in apparenza** sincrona.

Sintassi sull'utilizzo delle Promise in java script

Per poter rendere "*sincrona*" la procedura precedente **elabora()** occorre avvolgere tutto il suo contenuto all'interno di un oggetto **Promise**, che ha come unico parametro una function alla quale vengono automaticamente iniettati due puntatori a funzione: **resolve e reject.**

Terminata l'elaborazione, il codice interno della **Promise** dovrà richiamare il metodo **resolve** in caso di terminazione corretta oppure il metodo **reject** in caso di errore.

```
function elabora(img){
    let promise = new Promise(function(resolve, reject) {
        imgLibrary.elaboraImmagine(img)
        imgLibrary.onEnd(err, finalImage){  // elaborazione terminata
            if(err)
                reject(err)
            else
                resolve(finalImage);
        })
    })
    return promise;
}
```

- Cioè la funzione **elabora()** istanzia una Promise e termina immediatamente restituendo al main la Promise istanziata. Il main dovrà salvarsi questa Promise all'interno di una apposita variabile.
- Nel momento in cui la funzione **elabora()** richiama il metodo **resolve**, automaticamente la promise provvede a generare un evento **then()**, iniettandogli il risultato passato a resolve, cioè **finalImage**.
- Nel momento in cui la funzione **elabora()** richiama il metodo **reject**, automaticamente la promise provvede a generare un evento **catch()**, iniettandogli un oggetto err relativo all'errore verificatosi
- L'evento **finally()** viene generato sia in seguito del then(), sia in seguito del .catch()
- Se il codice interno non richiama né il metodo **resolve** né il metodo **reject**, la Promise rimane "appesa", cioè non vengono generati gli eventi **then()**, **catch()** e tantomeno **finally()**

Il main dovrà semplicemente gestire gli eventi **.then()** e **.catch()**, in cui tutta la parte di elaborazione dell'immagine è spostata dentro `elaboraImmagine()` completamente distaccata dal flusso principale:

```
let promise = elabora(img)
promise.then(function(finalImage) {
    console.log(finalImage)
})
promise.catch(function(err) {
    console.log(err.message)
})
promise.finally(function(){
    // close connection
})
```

Passaggio dei parametri ai gestori di evento

Come si può evincere dall'esempio precedente, le funzioni di callback di **.then()** e **.catch()** possono ricevere uno o più parametri di qualunque tipo che dovranno essere passati adeguatamente in fase di chiamata ai metodi resolve() e reject() e che verranno poi iniettati nelle callback di **.then()** e **.catch()**

Note

1) Gli eventi **.then** e **.catch** vengono comunque eseguiti **anche** se la loro assegnazione viene fatta *dopo* che la **promise** è terminate (cioè *dopo* che sono stati invocati i metodi resolve o reject).

2) Non è possibile assegnare due **.then** o due **.catch** alla stessa promise, nel qual caso se dovesse verificarsi la seconda assegnazione viene rieseguita la prima.

3) Se al posto di una variabile intermedia si utilizza il concatenamento, diventa è possibile ad ogni promise associare più funzioni di callback in cascata in quanto ogni promise restituisce a sua volta un'altra promise

```
promise.then(function(){ console.log(1); })
       .then(function(){ console.log(2); })
```

Concatenamento degli eventi

Il codice precedente può anche essere scritto nel modo seguente, senza l'utilizzo della variabile intermedia *let promise* = , concatenando il **`.then()`** e **`.catch()`** direttamente alla funzione principale.

```
elaboraImmagine(img)
.then(function(finalImage) {
    console.log(finalImage)
})
.catch(function(err) {
    console.log(err.message)
})
```

Questa sintassi però non è del tutto equivalente alla precedente.

Sia **`.then`** che **`.catch`** restituiscono a loro volta una **nuova** promise che '*maschera*' la promise principale alla quale possono essere associati un nuovo **`.then()`** ed un nuovo **`.catch()`** Per cui:

- **Se si utilizza prima `.then` e dopo il `.catch`, il catch diventa gestore dell'errore sia rispetto alla promise principale, sia rispetto al codice interno del then** e pertanto può visualizzare errori fuorvianti che 'mascherano' il vero motivo dell'errore.
- **Se si mette prima il `.catch` e dopo il `.then`,** se il catch non introduce altri errori, il then viene eseguito in cascata

Mentre il primo caso è ancora tollerabile, il secondo assolutamente NO.

La soluzione migliore è quella di usare SEMPRE una variabile intermedia, che non genera mai casi indesiderati.

Altro Esempio

Inclusione di un setTimeout all'interno di una promise :

```
function timer() {
    let promise = new Promise(function (resolve, reject) {
        setTimeout(function () { resolve("sono trascorsi 1000 ms") }, 1000);
    })
    return promise
}

let attendiUnSecondo = timer()  // attendiUnSecondo sarà una Promise
attendiUnSecondo.then(function(msg){
     console.log(msg)  // sono trascorsi 1000 ms
})
```

Oggetti per l'invio di richieste ajax

Per l'invio delle richieste ajax esistono diversi oggetti **ciascuno dei quali** restituisce al chiamante **SEMPRE** una **promise** che il chiamante potrà gestire all'interno del proprio codice.

$.ajax()

E' un metodo statico della libreria jQuery. (non presente nella libreria jquery **slim**) che rappresenta un wrapper dell'oggetto java script **XMLHttpRequest** per l'invio di una richiesta ajax ad un server.
Molto più semplice e flessibile rispetto ad XMLHttpRequest.

A differenza di **XMLHttpRequest,** $.ajax() è in grado di gestire **richieste multiple** verso una stessa risorsa con una stessa funzione di callback. In pratica passa un indice al server che lo rimanderà indietro in modo che la callback possa capire a quale elemento è riferita la risposta.

jQuery però gestisce le promise in modo leggermente diverso rispetto a javascript. Senza entrare nel dettaglio del funzionamento delle promise jQuery si può semplificare dicendo che le promise jQuery :

- Non generano gli eventi **then** e **catch**
- Al loro posto generano gli eventi **done** e **fail** simili ma non identici rispetto a quelli javascript

Sintassi di $.ajax()

$.ajax() si aspetta come parametro un **json** di opzioni costituito dai seguenti campi:

```
let options={
   url: "/url?nome=pippo",
   type: "GET",        // default
   data: { "nome": "pippo" },
   contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default
   dataType: "json",  // default
   timeout : 5000,
   async : true,       // default
   success: function(data, [textStatus], [jqXHR]) {
         console.log(data)
   },
   error : function(jqXHR, textStatus, str_error){
    if(jqXHR.status==0)
       alert("connection refused or server timeout");
    else if (jqXHR.status == 200)
       alert("Errore Formattazione dati\n" + jqXHR.responseText);
    else
       alert("Server Error: "+jqXHR.status+ " - " +jqXHR.responseText);
   },
   username: "nome utente se richiesto dal server",
   password: "password se richiesta dal server",
}
```

Richiamo

```
$.ajax(options)
```

In caso di successo della richiesta i dati ricevuti dal server vengono iniettati alla funzione **success** che potrà provvedere alla loro elaborazioni.

Elaborare i dati all'interno delle **options** a livello operativo è una soluzione molto poco modulare (e abbastanza scomoda). Come detto però $.ajax restituisce una promise che potrà essere gestita direttamente all'interno del codice del chiamante.

L'attributo contentType

Indica il formato con cui passare i parametri al server. Può assumere i seguenti valori:

```
contentType:"text/plain; charset=utf-8"
contentType:"application/x-www-form-urlencoded; charset=utf-8"
contentType:"application/json; charset=utf-8"    //stringa json
contentType:"application/xml; charset=utf-8"     //stringa xml
```

Nel caso di chiamate GET eventuali parametri possono essere indifferentemente

- accodati alla url *oppure*
- inseriti come json all'interno del campo **data** delle Options

Se **contentType:"application/x-www-form-urlencoded"**, i parametri passati a $.ajax() in formato JSON, **vengono automaticamente a convertirli in formato urlencoded**. Esempio:

```
let json = {"user1":{"nome":"pippo", "hobbies":["calcio","pesca"]}}
?user1[nome]=pippo&user1[hobbies][0]=calcio&user1[hobbies][1]=pesca
```

I parametri **POST** possono invece essere passati direttamente in un formato **"application/json"**
In questo caso, prima di inviare la richiesta, occorre SERIALIZZARE l'intero json.

L'attributo dataType

Indica il formato con cui `$_ajax()` deve restituire al chiamante i dati ricevuti dal server.
Può assumere i seguenti valori esprimibili **solamente** in modo diretto senza application/ davanti.

```
dataType:"text"
dataType:"json"
dataType:"xml"
dataType:"html"
dataType:"script"
```

- Scegliendo "text" la risposta viene restituita a onSuccess() cos'ì com'è, indipendentemente dal fatto che sia testo oppure json oppure xml.
- Scegliendo **json**, $_ajax provvede automaticamente ad eseguire il **parsing** dello stream json ricevuto, restituendo a onSuccess() un oggetto json. Idem per xml

Il metodo success(data, textStatus , jqXHR)

Viene richiamato in corrispondenza della ricezione della risposta. In caso di content-type non testuale (ad esempio json o xml) **provvede automaticamente a parsificare la risposta ricevuta restituendo al chiamante l'object corrispondente**. Questa conversione automatica viene fatta in un "thread" separato e quindi alleggerisce la nostra applicazione..
textStatus indica lo stato in cui si è conclusa la XMLHttpRequest
jqXHR è un riferimento all'oggetto XMLHttpRequest sottostante utilizzato per inviare la richiesta

Il metodo error(jqXHR, textStatus, str_error)

In caso di errore, invece di richiamare la funzione **success**, viene richiamata la funzione **error.**
Questa funzione viene richiamata :

- **in caso di timeout**
- in corrispondenza della ricezione di un codice di **errore diverso da 200**
- in caso di ricezione di status==200 ma con un **oggetto json non valido** (se **dataType="json"**) (ad esempio se il server va in syntax error e restituisce un messaggio di errore)

jqXHR è un riferimento all'oggetto XMLHttpRequest sottostante utilizzato per inviare la richiesta
textStatus indica lo stato in cui si è conclusa la XMLHttpRequest
Il terzo parametro rappresenta un msg di errore fisso legato al codice di errore

L'attributo async

Impostando il valore **false** il metodo diventa sincrono, bloccando di fatto l'interfaccia grafica fino a quando non sono arrivati i dati

Scrittura di una funzione compatta per l'invio delle richieste ajax

```
const URL = "https://randomuser.me"
function inviaRichiesta(method, url, parameters={}) {
    let contentType;
    if(method.toUpperCase()=="GET")
          contentType="application/x-www-form-urlencoded;charset=utf-8";
    else{
          contentType = "application/json; charset=utf-8"
          parameters = JSON.stringify(parameters);
    }
    let options={
       url: URL + url,
       type: method
       data: parameters,
       contentType: contentType
       dataType: "json",  // default
       timeout : 5000,
    }
    return $.ajax(options)   // ritorna una promise
}
function errore (jqXHR, test_status, str_error) {stessa di prima}
```

Chiamante

```
let promise = inviaRichiesta("get", "/api?student=pippo")    // oppure
let promise = inviaRichiesta("get", "/api", {"student":"pippo"})
promise.fail(errore);
promise.done(function(data, test_status, jqXHR){
    console.log(data);
});
```

axios.js

Visto l'attuale declino di jQuery, **axios.js (**dal greco "***degno***") rappresenta una degna alternativa a $.ajax in direzione "vanilla" javascript. Restituisce una promise js con gli eventi .then e .catch. Dal sito ufficiale:

Axios is a promise-based HTTP Client for **node.js** and the **browser**. It is **isomorphic** (= it can run in the browser and node.js with the same codebase). On the server-side it uses the native node.js http module, while on the client (browser) it uses **XMLHttpRequest**.

Analogamente a $.ajax() i parametri GET possono essere :

- accodati direttamente alla url in formato urlencoded *oppure*
- inseriti in formato ***json*** all'interno del campo **params** delle Options. **Axios provvederà automaticamente e convertire questi parametri in url-encoded e ad accodarli alla URL**.

I parametri POST possono essere passati in formato **json** all'interno del campo **data** delle options.
La serializzazione viene fatta automaticamente da axios.

```
function inviaRichiesta(method, url, parameters={}) {
     let options = {
          "baseURL":"http://webserver:3000",  // indirizzo del web server
          "url":  url,                        // risorsa da richiedere
          "method": method.toUpperCase()      // GET or POST
          "headers": {"Accept": "application/json"},
          "responseType": "json",             // tipo di risposta atteso
          "timeout": 5000                     // tempo di attesa della risposta
     }
     if(method.toUpperCase() == "GET"){
       options.headers["Content-Type"]='application/x-www-form-urlencoded;charset=utf-8'
       options["params"]=parameters   // plain object or URLSearchParams object
     }
     else{
       options.headers["Content-Type"] = 'application/json; charset=utf-8'
       options["data"]=parameters     // Accept FormData, File, Blob
     }
     return axios(options)               // return a promise
}

function errore(err) {
     if(!err.response)
          alert("Connection Refused or Server timeout");
     else if (err.response.status == 200)
        alert("Formato dei dati non corretto : " + err.response.data);
    else
        alert("ServerError: " + err.response.status + "-" + err.response.data)
```

Chiamante

Al **then** viene iniettata l'intera **http-response**. I dati ricevuti si trovano dentro **response.data**

```
let promise = inviaRichiesta("get", "/api?student=pippo")    // oppure
let promise = inviaRichiesta("get", "/api", {"student":"pippo"})
promise.catch(errore);
promise.then(function(httpResponse){
     console.log(data);
});
```

Nota sui parametri GET in formato url-encoded

Nel caso delle chiamate **GET** `config["params"]` accetta al suo interno anche dei ***plain object***, cioè dei json semplici, cioè NON annidati. Tenere presente che tra i ***plain object*** rientrano anche i vettori enumerativi Quindi è possibile passare come parametro sia un vettore enumerativo sia un json di 1° livello

```
{ "data" : [1,2,3,4] }
      viene tradotto in URL-encoded nel modo seguente:
      ?data[]=1&data[]=2& data[]=3& data[]=4

{ "data" : {"a":1, "b":2 }
      viene tradotto in URL-encoded nel modo seguente:
      ?data[a]=1&data[b]=2
```

fetch

Metodo nativo javascript (2015) sostitutivo del vecchio xmlHttpRequest.

Fetch is the modern replacement for XMLHttpRequest: unlike XMLHttpRequest, which uses callbacks, Fetch is promise-based and is integrated with features of the modern web such as service workers and Cross-Origin Resource Sharing (CORS).

fetch() presenta un unico argomento obbligatorio, la URL della risorsa a cui accedere. E' possibile passare secondo parametro di **`options`**, che sono più o meno le stesse di $.ajax() e axios().

La chiamata va sempre a buon fine, cioè fetch restituisce sempre un oggetto **response** indipendentemente dall'http status code della risposta (sia 200 oppure un qualunque codice di errore). fetch genera errore soltanto in caso di errore di rete a seguito di timeout.

L' oggetto **Response** restituito da fetch espone:

- una proprietà **ok** impostata su **true** in caso di ok o su **false** per risposte diverse da 2xx

Espone inoltre

- una proprietà **status** contenente l'http response code
- una proprietà **statusText** contenente un http response message fisso

L'oggetto **Response** non contiene però direttamente l'effettivo corpo della risposta JSON (come nel caso di axios), ma per ottenere i dati veri e propri occorre richiamare un ulteriore metodo asincrono **`response.json()`** il quale restituisce una seconda **`promise`** che si risolve iniettando i dati veri e propri.

Esempio completo di invio di una richiesta

```
const _URL =  ""     // "http://localhost:1337"

async function inviaRichiesta(method, url="", params={}) {
  method = method.toUpperCase()
  let options = {
    "method": method,
    "headers":{},
    "mode": "cors",      // default
    "cache": "no-cache", // default
    "credentials": "same-origin",    // default
    "redirect": "follow", // default
    "referrerPolicy": "no-referrer", // default no-referrer-when-downgrade
    // riduce il timeout rispetto al default (6s) Non sembra possibile increm
    //"signal": AbortSignal.timeout(500)
  }

  if(method=="GET") {
    const queryParams = new URLSearchParams();
    for (let key in params) {
      let value = params[key];
      if (value && typeof value === "object")
        queryParams.append(key, JSON.stringify(value));
      else
        queryParams.append(key, value);
    }
    url += "?" + queryParams.toString()
    options.headers["Content-Type"]="application/x-www-form-urlencoded"
  }
```

```
  else if(params instanceof FormData)
       options["body"]=params     // Accept FormData, File, Blob
  else{
       options.headers["Content-Type"]="application/json";
       options["body"] = JSON.stringify(params) }
  }

  try{
    const response = await fetch(request)
    if (!response.ok) {
        let err = await response.text()
        return {"status":response.status, err}
    }
    else{
        let data = await response.json().catch(function(err){
            console.log(err)
            return {"status":422, "err":"Response contains an invalid json"}
        })
        return {"status":200, data}
    }
  }
  catch{
     return {"status":408, "err":"Connection Refused or Server timeout"}
  }
}
```

Chiamante

```
let response = inviaRichiesta("POST", "https://example.com/answer", {n:42} )
```

Note:

1) Notare il **`.toUpperCase()`** sul method. fetch() esige che i metodi siano scritti in MAIUSCOLO !
2) fetch non espone i classici eventi per controllare lo stato di avanzamento dell'upload
3) CORS requests may only use the HTTP or HTTPS ***URL scheme***.
   fetch non invia richieste con ***URL schema*** non http, per cui non è utilizzabile con il protocollo file://
4) Eventuali **parametri GET** devono essere manualmente accodati alla URL. Non esiste il `config["params"]` di axios che esegue la conversione URL-encoded in automatico.
5) Occhio alle **headers**, cha vanno sempre scritte fra parentesi quadre. Non esiste la versione camelCase
6) Notare che **statusText** non contiene il messaggio di errore impostato dal server, ma un testo fisso di errore http associato al codice di errore ricevuto. Per accedere al messaggio ricevuto dal server occorre utilizzare il metodo asincrono **response.text()** come nell'esempio precedente.
7) Confronto Axios – Fetch : **`https://blog.logrocket.com/axios-vs-fetch-best-http-requests/`**

Metodi dell'oggetto response

Esiste un apposito metodo **asincrono** per ogni possibile oggetto ammesso all'interno del campo **body**:

- `.text()`: Returns the response as string data
- `.json()`: Returns the response as JSON
- `.formData()`: Return the response as `FormData` object (which stores key-value pairs of string data)
- `.blob()`: Returns the response as blob object (binary data along with its encoding)
- `.arrayBuffer()`: Return the response as `ArrayBuffer` (low-level representation of binary data)

Upload di un file binario

Come detto fetch accetta un `Content-Type= multipart/form-data` senza alcun ulteriore vincolo. Per l'upload dei file binari, la tendenza attuale è quella di utilizzare il metodo PUT

```
const formData = new FormData();
formData.append("username", "abc123");
formData.append("avatar", txtFile.files[0]);
fetch("https://example.com/profile/avatar", {
   method: "PUT",
   body: formData,
})
```

Interceptors

```
const requestInterceptor = function(request)  {
  if (!request.headers)
     request.headers = new Headers();
  if ("authToken" in localStorage) {
    let authToken = localStorage.getItem("authToken");
    request.headers.set('authorization', authToken);
  }
  return request;
}

const responseInterceptor = function(response) {
  if (response.headers) {
    let authToken = response.headers.get("authorization")
    if(authToken)
      localStorage.setItem("authToken", authToken)
  }
}
```

Approfondimenti sulle fetchOptions

mode : **`cors`** = normale funzionamento CORS
no-**`cors`** = cioè senza resource sharing. Consente solo SAFE request
same-origin =. Consente solo chiamate sulla stessa origine

credentials: **`omit`** = non invia credenziali (cookies e HTTP Credential)
`same-origin` = invia le credenziali solo nelle same-origin request
`include` = invia le credenziali sia nelle same-origin request che nelle cross-origin
(il valore *include* è incompatibile con `Access-Control-Allow-Origin = "*"`)

Content-Type: Può assumere i valori `application/x-www-form-urlencoded, application/json, text/plain, multipart/form-data,`

body : può contenere i seguenti oggetti: `String (or String Literal), URLSearchParams, FormData, File, Blob, ArrayBuffer, TypedArray, DataView`

referrerPolicy: L'intestazione http **`Referer`** è simile a **`Origin`** ma, mentre Origin contiene SOLO l'origine della richiesta (protocollo, dominio e porta), Referer contiene l'intero path della richiesta compresi i parametri. Al fine di preservare la privacy, le policy del Referer della richiesta possono definire i dati che possono essere inclusi / omessi. E' possibile anche omettere l'intera l'intestazione del Referer. Viceversa Origin non può essere esclusa
`"no-referrer"` – non invia mai il `Referer`.
`"unsafe-url"` – invia sempre l'url completo nel `Referer`, anche per richieste HTTPS→HTTP.

Purtroppo tra le **`fetchOption`**s no c'è un **timeout** che deve eventualmente essere gestito a mano.

Il costrutto Async / Await

Rappresenta una modalità alternativa rispetto al **.then** e **.catch** per gestire una promise restituita da una certa funzione. Consente di rendere 'bloccante' una funzione asincrona facendola '*sembrare'* sincrona.

"await" makes javaScript wait until the promise settles, and then go on with the result. That doesn't cost any CPU resources, because the engine can do other jobs meanwhile: execute other scripts, handle events etc.
It's just a more elegant syntax of getting the promise result than promise.then(), easier to read and write.

Si riconsideri l'esempio iniziale di una funzione asincrona che restituisce una promise contenente il risultato di una elaborazione grafica di una immagine:

```
function elaboraImmagine(img){
    let promise = new Promise(function(resolve, reject) {
        _lastLibrary.onEnd(err, finalImage){
            if(err)
                reject(err)
            else
                resolve(finalImage);
    })
    return promise;
}
```

Sintassi **then** / **catch**

```
let promise = elaboraImmagine(img)
promise.then(function (finalImage){ })
promise.catch( )
```

Sintassi **async** / **await**

```
async function visualizzaImmagineElaborata(){
      let finaleImage = await elaboraImmagine(img)
```

- **await** attende la risoluzione della promise e, **anziché restituire la promise, restituisce direttamente i *dati* che nella versione 'tradizionale' verrebbero iniettati al then**.
- la parola chiave **await** può essere utilizzata soltanto all'interno di una funzione dichiarata **async**.

Gestione dell'errore

Se si intende gestire anche l'errore si può concatenare il **.catch** direttamente in coda alla funzione.

```
let finaleImage = await elaboraImmagine(img).catch(function(err){ …… })
```

In alternativa si può utilizzare un TRY – CATCH che produce lo stesso identico risultato

```
async function visualizzaImmagineElaborata(){
      try { let finaleImage = await elaboraImmagine(img) }
      catch (err) { errore(err) }
```

Return del risultato

La grande comodità dell'await sta nel fatto che i dati restituiti da await possono essere direttamente **ritornati** al chiamante (tramite semplice return a fine procedura). *Notare che all'interno del catch il return è inutile in quanto, in sua assenza, viene automaticamente eseguito un return undefined.*

Attenzione però che, se una funzione dichiarata async ritorna un qualunque risultato, in realtà **ciò che viene ritornato NON sono i dati ma una nuova Promise contenente i dati indicati dopo il return. Per cui Il chiamante a sua volta dovrà gestire questa Promise con il then e catch oppure con un nuovo await.**

`$(document).ready`

La funzione jQuery **`$(document).ready`** non consente di dichiarare **async** la funzione di callback.
Nel caso dovesse servire, occorre utilizzare l'analoga javascript, cioè
`window.onload=async function()`

Approfondimenti su await

await può essere utilizzata in due modi:

1) davanti al richiamo di una funzione asincrona che restituisce una promise (come nell'es precedente):

```
let finaleImage = await elaboraImmagine(file)
```

2) direttamente davanti ad una promise dichiarata all'interno della funzione stessa:

```
async function foo() {
   let promise = new Promise(function(resolve, reject) {
       setTimeout(() => resolve("done!"), 1000)
   });
   let result = await promise;
   alert(result);   // "done!"
}
```

In entrambi i casi l'esecuzione si bloccherà sull'**`await`**, fino alla risoluzione della Promise.

Approfondimenti su async

La parola chiave **async,** oltre che nel caso tipico degli esempi precedenti, può anche essere utilizzata davanti a qualsiasi funzione **per fare in modo che la funzione ritorni una `promise`**.

```
async function foo() {
   return 1;
}
```

Nel caso in cui il codice interno alla funzione ritorni una <non-promise>, automaticamente il risultato verrà "wrappato" all'interno di una promise. Nell'esempio la funzione restituirà una promise con dentro il valore 1

```
let promise = foo();
promise.then(function(val){
  alert(val);       // 1
// oppure, più semplicemente
primise.then(alert);  // 1
```

In alternativa all'uso di **`async`**, la funzione foo() potrebbe anche essere scritta nel modo seguente, con un Promise.resolve() esplicito:

```
function foo() {return Promise.resolve(1)}
```

Il metodo Promise.all( )

Capita talvolta di dover eseguire parallelamente più operazioni asincrone.

Si supponga di voler prenotare più **ombrelloni** di un certo stabilimento balneare, in cui occorre eseguire una richiesta di prenotazione per ogni ID. Se tutte le prenotazioni sono andate a buon fine si vuole visualizzare un unico messaggio riassuntivo di **OK**. Gli approcci possibili sono due:

1. Eseguire le richieste in modo **annidato** ognuna all'interno del then della precedente.
   In corrispondenza della terminazione dell'ultima richiesta si visualizza l'**OK**.
   Pesante sia dal punto di vista della scrittura sia dal punto di vista dell'esecuzione.
2. Lanciare tutte le query in sequenza una dopo l'altra (con conseguente elaborazione parallela).
   Come faccio a dare il messaggio di OK? L'ordine di esecuzione delle richieste non è detto che coincida con l'ordine di lancio. Occorre pertanto gestire un contatore incrementato dopo ogni query. Quando il contatore raggiunge `vet.length` si da l'OK. Soluzione efficace ma un po' spartana.

Un terzo approccio più strutturato, è quello di lanciare tutte le richieste in parallelo accodando **le singole Promise all'interno di un vettore**. Poi si passa il vettore al metodo **`Promise.all()`** il quale **attende** che tutte le promise ricevute come parametro vengano risolte e, al termine, genera una nuova Promise in cui:

- il **`.then()`** viene generato se tutte le Promise sono state risolte con esisto positivo.
- Se anche **una sola delle singole Promise** viene risolta con esito negativo, verrò generato il **`.catch()`** a cui viene iniettato l'errore generato dalla prima Promise che fallisce.

Al **`.then()`** viene iniettato un **vettore enumerativo** di json contenente tutte le httpResponse iniettate a tutte le singole Promise, **nello stesso ordine con cui state accodate le Promises**, **indipendentemente dall'ordine di terminazione delle stesse**. In questo modo si risolve perfettamente il problema dell'asicronicità delle chiamate. Va bene quando presumibilmente tutte le Promises dovrebbero andare a buon fine e comunque abbiamo bisogno di tutti risultati positivi per proseguire. ESEMPIO:

```
let ombrelloniDaPrenotare = [7, 12, 24, 52]
let promises = []

for (let id of obrelloniDaPrenotare)
    let request = inviaRichiesta("PATCH", `/ombrelloni/${id}` {stato:true})
    promises.push(request)
}

Promise.all(promises).then(function(httpResponses) {
    alert("Prenotazioni eseguite correttamente")
    for (httpResponse of httpResponses)
        console.log(httpResponse.data)
});
.catch(function(err){
    console.log(err)
})
```

Recentemente è stata aggiunta anche **`Promise.allSettled()`** che aspetta che tutte le promise siano terminate (con successo o con errore) e inietta al then un vettore con tutti i risultati (positivi e negativi). L'array iniettato al then sarà un array di json così strutturato :

```
{status:"fulfilled", value:result} // per le risposte con successo
{status:"rejected", reason:error}  // per gli errori.
```

```
httpResponses.forEach((httpResponse) => {
  if (httpResponse.status == "fulfilled")
     console.log(httpResponse.value.data)+
  else if (httpResponse.status == "rejected")
    console.log(httpResponse.reason);
```

Promise.all() abbinato al metodo map()

Si consideri il seguente caso di studio.
Si parte da un vettore di risposte selezionate da uno studente durante un quiz a risposte chiuse in cui lo studente deve scegliere la risposta esatta mediante una sequenza di radio buttons.

```
let risposte = [
     {"idDomanda" : "1",  "rispostaSelezionata" : 3},
     {"idDomanda" : "2",  "rispostaSelezionata" : 5},
     {"idDomanda" : "3",  "rispostaSelezionata" : 0}
]
```

Si vogliono confrontare le risposte selezionate con le risposte corrette contenute all'interno di un database.

```
let voto = 0;
const risultati = await Promise.all(
   risposte.map(async function(risposta) {
      try {
         const domanda = await tabellaDomande.findOne({id:risposta.idDomanda})
         if (domanda) {
            const correct = domanda.correct == risposta.rispostaSelezionata;
            if (correct)
              voto += 1;
             else
              voto -= 0.25;
            return { ...risposta, correct };
         }
      }
      catch (err) {
         console.log(err)
      }
      return risposta;
   })
)
```

Al metodo **`risposte.map()`** viene passata una **funzione asincrona** alla quale, ad ogni ciclo, viene iniettato un singolo json del vettore *`risposte`*.
La funzione va nel database a cercare la domanda avente come id = **`risposta.idDomanda`**

Se la domanda viene trovata si confronta `domanda.correct` con `risposta.rispostaSelezionata`.
Se la risposta data dallo studente è corretta si aggiunge 1 voto altrimenti si sottrae 0.25
Alla fine, per ogni risposta, si crea un nuovo json che contiene le stesse due chiavi del json originale ***`risposta`***, con l'aggiunta di una chiave ***`correct`*** che potrò essere true/false.

In caso di errori (catch) oppure se la domanda non è stata trovata con ***`if(domanda)`***, la callback del metodo .map() ritorna semplicemente la risposta originale senza la proprietà `correct`.
Questo return finale è abbastanza inutile.
Se non ci fosse verrebbe automaticamente eseguito un ***`return undefined`***

Ricordando che quando una funzione asincrona esegue un return, ciò che in realtà viene ritornato non sono i **dati** ma una **promise**, possiamo concludere che il metodo .**map()** ritorna un vettore enumerativo di promise (una per ogni domanda analizzata) in cui ciascuna promise contiene il singolo json **risposta** contenente a sua volta anche il campo correct.

Questo vettore enumerativo di promise viene passato a Promise.all() il quale, quando tutte le promise saranno state risolte, restituirà un vettore **risultati** contenente per ogni cella
il singolo json ritornato dal metodo .map().

JSON-Server

JSON-Server è una piccola utility disponibile in ambiente nodejs in grado di rispondere a chiamate Ajax e restituire al chiamante il contenuto di un **`file.json`** memorizzato all'interno della cartella di lavoro.

Passi necessari all'installazione ed utilizzo di json-server

1. Installare NodeJS (https://nodejs.org)
2. Dal pulante WINDOWS selezionare **Node / Nodejs Command Prompt**
   Il prompt normale dal 2024 ha iniziato a segnalare problemi di sicurezza
3. Installare globalmente `json-server`
   `npm install -g json-server`
4. Scrivere nella cartella di progetto un file.json contenente il database dell'esercizio
5. Aprire un **Nodejs Command Prompt** nella cartella di lavoro
6. Lanciare json-server in modo che utilizzi ad esempio il file db.json
   `json-server --watch db.json`
   L'opzione **--watch** indica che il file deve essere monitorato con continuità. Dalla versione 2024 questa opzione viene settata in automatico e dunque non è più necessaria
7. Il server risponde con un messaggio che conferma l'attivazione in localhost sulla porta **3000** e visualizza l'elenco delle risorse contenute all'interno del file db.json. Se la porta non viene impostata automaticamente la si può specificare nel comando di avvio
   `json-server --port 3000 db.json`

All'interno del file db.json si possono inserire più gruppi di dati (detti **risorse**) che possono essere soltanto **vettori associativi** (json)**.** Dalla versione 2024 i vettori enumerativi non sono più ammessi.

Per default json-server viene avviato soltanto sull'interfaccia **`localhost`** e dunque non sarà visibile sulla rete. Per avviarlo sull'interfaccia di rete occorre utilizzare l'opzione **`--host`**

```
json-server --host 10.0.1.2 --port 3000 --watch db.json
```

Esempio di file db.json contenente una unica risorsa denominata people

```
{ people: [
  {   "id":"abc1",
      "nome": "Alfio",
      "genere": "m",
      "classe": "4B"
  },
  {   "id":"abc2",
      "nome": "Beatrice",
      "genere": "f",
      "classe": "4B"
  },
  {   "id":"abc3",
      "nome": "Carlo",
      "genere": "m",
      "classe": "1C"
  }]
}
```

Gestione degli ID

Dalla versione 2024, **`id`** deve tassativamente essere dichiarato come **`stringa`** di qualunque lunghezza.

Non sono più ammessi id numerici.

Se non è necessario ai fini del progetto, ID **può anche essere omesso**.

Per i record creati **dinamicamente** da codice (*e solo in questo caso*), ID viene creato automaticamente in modo univoco come stringa su 4 caratteri

A questo punto si può aprire un browser e :

- richiedendo la url **http://localhost:3000**, il server risponde con una semplice pagina di benvenuto.
- Se invece si concatena alla url il nome di una risorsa di primo livello, il server risponde inviando l'intero contenuto associato a quella risorsa: **http://localhost:3000/people.** E' possibile richiedere **qualsiasi risorsa di primo livello** presente nel file db.json

Richieste GET

Sono quelle più usate e consentono di richiedere dei dati al server

GET /people senza parametri restituisce un *vettore enumerativo* con tutte le persone
GET /people/2 restituisce un *singolo record* relativo alla persona avente **id=2**
GET /people**?**id=2 restituisce un *vettore enumerativo* lungo 1 contenente la persona indicata
GET /people**?genere**=m&**classe**=4B. Esegue una **AND** e restituisce un vettore di record

Richiesta di un singolo record sulla base dell'ID

L'ID può essere direttamente accodato alla risorsa dopo lo slash nel qual caso il server restituisce un *singolo record* relativo alla persona avente **id=2** oppure può essere passato in formato url-encoded dopo il punto interrogativo, nel qual caso il server restituisce un vettore enumerativo lungo 1

Richiesta di più record sulla base di altri campi

I parametri diversi dall'ID non possono essere accodati alla risorsa ma vanno concatenati alla url dopo il **?** e suddivisi tra loro mediante una **&**, utilizzando cioè quello che si chiama formato **url-encoded**.
Ad esempio Cerca tutti gli studenti di genere maschile della classe 4B
GET /people**?genere**=m**&classe**=4B. // Tutti i maschi della classe 4B

inviaRichiesta(), per come è stata scritta, accetta i campi di filtro come terzo parametro in formato JSON (molto più comodo) e provvede automaticamente a **convertirli** in url-encoded concatenandoli alla url. Esempo: `inviaRichiesta("GET", "/people", {"genere":"m", "classe"="4B"})`

Altre modalità di richiesta

In tutte le chiamate diverse dalla GET, **ID può essere passato SOLO in coda alla risorsa** !

DELETE /people/2 elimina il record con id=2. Non ha parametri.

PATCH /people/2 {"residenza":"genola", "classe":"2A"}
modifica lo studente #2 a cui assegna una nuova residenza ed una nuova classe.

PUT /people/2 sostituisce l'intero record #2 con quello passato come **parametro**
PATCH, a differenza di PUT, aggiorna soltanto i campi passati come terzo parametro, per cui non è necessario passare tutti i campi. A differenza di mongo consente di aggiornare anche l'ID, **il che è estremamente pericoloso**. Se un campo non esiste viene automaticamente creato,

POST /people aggiunge in coda a **people** il record passato come terzo parametro
{"nome":"pippo", "genere":"m", "classe":"2A"}

Nelle chiamate POST è buona norma non assegnare l'ID, nel qual caso json-server provvede automaticamente ad assegnare al nuovo record un ID casuale **univoco,** di tipo stringa lunga 4 caratteri. Restituisce dentro **response.data** l'intero record inserito, **comprensivo dell'ID** assegnato automaticamente. Non sembra possibile postare più record in una stessa chiamata.

Nota

- Se in coda alla url permane un **?** oppure anche **?&** NON è un problema

Full Search

- Il parametro ?**q=testo** restituisce tutti i record che contengono quel testo, in un qualsiasi campo
- **?titolo_like=testo** restituisce tutti i record in cui un certo campo contiene un certo testo. Funzionalità supportata fino alla versione 0.17 e non più supportata nella versione 1.0 (gennaio 2024)

Campi annidati

Le ricerche possono anche accedere ad eventuali oggetti interni al json del database.
In caso di chiamate GET, la sintassi da utilizzare per passare a json-server un parametro annidato in formato url-encoded**, deve necessariamente essere la sintassi con il puntino :**

`/people?location.country=italy`
che in formato json diventa: **`{"location.country":"italy"}`**

Quindi la chiamata ajax dovrà *espressamente* avere il seguente formato:
`inviaRichiesta( "GET", "/risorsa", {"location.country":"italy"} )`

Viceversa, una chiamata del tipo
`inviaRichiesta( "GET", "/risorsa", {location:{country:"italy"}} )`

di default viene convertita in url-encoded nel seguente modo, cioè con le parentesi quadre.
`location[country]=italy`

cioè sia $.ajax che axios traducono i json annidati utilizzando le parentesi quadre e non il puntino.

json-server però **non riconosce le quadre** come strumento di indicizzazione ma **richiede espressamente il puntino.**

Volendo si potrebbe eseguire una conversione manuale delle quadre con il puntino, utilizzando ad esempio il metodo statico **`$.param`**

```
let result = $.param(json).replaceAll("%5B",".").replaceAll("%5D","")
```

`%5B` e `%5D` sono le parentesi quadre. In questo modo si ottiene il classico formato url-encoded con il puntino

```
nome=pippo&location.city=London&location.street=Carnaby+Street
```

Visualizzazione

Notare infine che la precedente sintassi per il passaggio dei parametri annidati in formato url-encoded funziona SOLO ed ESCLUSIVAMENTE in fase di passaggio dei parametri **URLencoded**
Non funziona invece in fase di visualizzazione / assegnazione. Sia **`data`** il record da visualizzare:

`data["location"]["country"]`	**OK**
`data.location.country`	**OK**
`data["location.country"]`	**NOK** cerca una chiave che si chiama “location.country”

Campi vettoriali

json-server **NON** consente di ricercare singoli valori all’interno di campi vettoriali.
Però riconosce singoli valori scalari posizionati all’interno di vettori lunghi 1.

OR fra i parametri – Funzionalità **non** più supportata a partire da gennaio 2024

Tra i parametri url-encoded si possono inserire anche più parametri con lo stesso nome.
In questo caso il server esegue una OR tra i parametri.
Ad esempio tutti gli studenti della classe 1B e della classe 2B
`GET /people?classe[]=1B&classe[]=2B.` // Restituisce *sia* gli studenti della 1B *sia* della 2B

Nel caso della OR i parametri non possono essere passati come terzo parametro in formato JSON .
Infatti non è consentito creare all’interno di un json più chiavi aventi lo stesso nome.
E’ però possibile creare un json del tipo `{"classe" : ["1B", "2B"]}`
cioè la chiave classe viene dichiarata come vettore con all’interno tutti i valori da utilizzare per la OR

Note sul passaggio dei parametri GET e POST

Richieste HTTP GET

Quando si invia una richiesta al server digitando la URL nella barra degli indirizzi, il browser invia al server una richiesta **HTTP GET**, di solito priva di parametri che possono eventualmente essere accodati alla url in formato *URL_ENCODED*, cioè un elenco di coppie **Chiave=Valore** introdotte da un **?** e separate tra loro da una &

```
http://www.miosito.it/pagina2.html?nome=mario&cognome=rossi&eta=16
```

Usando la funzione inviaRichiesta(), i parametri GET possono essere passati a inviaRichiesta() come terzo parametro in formato JSON. inviaRichiesta() provvederà lei a leggere questi parametri, a trasformarli in URL_ENCODED ed accodarli alla url della chiamata

Richieste HTTP POST

In caso di richiesta **HTTP POST** i parametri vengono passati al server **all'interno del body della HTTP REQUEST**, in modo non visibile. Maggiore riservatezza. Il formato utilizzato è di solito il formato JSON.
La funzione inviaRichiesta provvede semplicemente a prendere il JSON ricevuto come terzo parametro, serializzarlo, e salvarlo all'interno del body della HTTP REQUEST

Modalità mista

Quando si esegue una chiamata HTTP POST, è anche possibile **aggiungere manualmente alcuni parametri GET in coda alla URL**.
In questo modo le informazioni passate come terzo parametro ad inviaRichiesta() verranno passate in modalità POST all'interno del body della HTTP REQUEST, mentre quelli direttamente concatenati alla URL verranno passati in modalità GET.
In caso invece di chiamata GET, eventuali parametri passati come terzo parametro a inviaRichiesta() sovrascrivono i parametri concatenati manualmente alla URL, per cui una modalità mista non è consentita.

Parametri passati come parte della risora

Molto spesso l'ID viene passato al server NON come 'normale' parametro GET o POST, ma come parametro intero alla risorsa:

```
PATCH http://mioServer/people/2?residenza=genola&classe=2A
```

Il **2** rappresenta l'ID e viene passato all'interno della risorsa, cioè in modo ancora diverso rispetto ai normali parametri GET concatenati DOPO il punto interrogativo

Questa modalità in realtà non è riservata all'ID ma potrebbe essere utilizzata anche per altri paramaetri

Esempi di Scambio dati tra client e server

invio Richiesta ed Elaborazione della risposta

```
var request = inviaRichiesta("get", "/elencoFiliai", {codBanca:7})
request.done(function(filiali){
      let lstFiliali=$("#lstFiliali");
      for (let filiale of filiali){
            $("<option>").val(filiale["cBanca"])
            .html(filiale["Nome"])
            .appendTo(lstFiliali);
      }
      lstFiliali.prop("selectedIndex", -1);
)};
request.fail(errore);
```

Note

- Ricordare sempre che l'associazione statica del tipo

  ```
  $("button").on("click", function(){ })
  ```

  agisce SOLTANTO sugli elementi già appesi al DOM, e non su eventuali elementi creati successivamente in modo dinamico.

- Attenzione che una istruzione del tipo

  ```
  _label.html(_label.html() + "Voce 1");
  ```

  sovrascrive completamente (eliminandoli) tutti gli eventuali gestori di evento eventualmente associati agli elementi posizionati all'interno della _label (es option buttons).

Risposta di tipo ok / nok

Ad esempio utente valido SI / NO. In questo caso ci si comporta normalmente nel modo seguente:

- In caso di nok si invia una risposta con codice diverso da 200 che sul client forzerà il .fail
- In caso di ok si restituisce un codice **200** che sul client porterà all'esecuzione del done. Oltre al codice 200 occorre però restituire anche un json valido che il client potrà anche non leggere ma la cui assenza pregiudica la conversione in json dello stream ricevuto. Esempi di risposte PHP valide :

  ```
  echo  '{"ris" : "OK"}';
  echo  '"OK" ';
  $json["ris"] = "OK";   echo json_encode($json);
  ```

Assenza della risposta

Il server non deve obbligatoriamente inviare una risposta al client (come ad esempio nel caso del logout). Se il comando inviato al server non richiede il ritorno di una risposta, sul client occorre **omettere** **sia** la proprietà **`success`** (*oppure assegnargli il valore* ***`null`***) **sia** la proprietà **`error`** .
Nella versione con l'utilizzo delle Promise in entrambi i casi occorre omettere sia il done che il fail.

Attenzione che impostando l'attributo dataType="json", l'assenza di risposta viene interpretata da $.ajax() come stringa vuota, per cui la conversione json fallisce e viene richiamato il metodo **`error.`**
Se però il metodo error non è stato gestito non succede praticamente nulla.

Scambio dati in formato XML

Ricezione di uno stream xml relativo ad un elenco di studenti:

```
function aggiorna(){
if (richiesta.readyState==4 && richiesta.status==200){
    var tab = document.getElementById("gridStudenti");
    tab.innerHTML="";
    // Se si riceve un text/plain occorre parsificarlo
    var parser=new DOMParser();
    var xmlDoc=parser.parseFromString(richiesta.responseText,"text/xml");
    var root = xmlDoc.documentElement;

    // Se si riceve un albero già parsificato (content-type=application/xml)
    var root = richiesta.responseXML.documentElement;
    var table = document.getElementById("gridStudenti");
    table.innerHTML = "";
    var riga = document.createElement("tr");
    riga.innerHTML="<th>ID</th> <th>Nome</th> <th>Età</th> <th>Città</th>";
    table.appendChild(riga);

    for(var i=0; i<root.childNodes.length; i++){
        var record = root.childNodes[i]; // singolo studente
        var riga = document.createElement("tr");
        table.appendChild(riga);

        for (var j=0; j<record.childNodes.length; j++){
            var td;
            td = document.createElement("td");
            td.innerHTML = record.childNodes[j].textContent;
            riga.appendChild(td);
        }
    }
  }
```

Cenni sulla libreria sweetAlert2

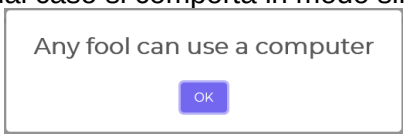
Consente di visualizzare delle finestre modali simili ad 'alert' e 'prompt' ma molto più friendly, con la possibilità di inserire all'interno molteplici contenuti html.

sweetAlert2 è *incompatibile* rispetto a sweetAlert 1. Viene distribuita come JS + CSS oppure **ALL** che contiene sia js che css. L'ultima versione a gennaio 2024 è la **11.10** disponibile al seguente cdn

```
https://cdnjs.cloudflare.com/ajax/libs/limonte-
sweetalert2/11.3.7/sweetalert2.all.min.js
```

Per aprire una finestra sweetAlert2 si utilizza il metodo **`Swal.fire()`** che accetta come parametro:

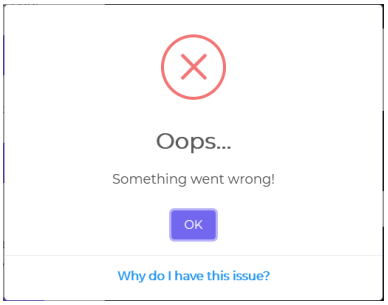
- Un json di opzioni
- Una semplice stringa, nel qual caso si comporta in modo simile ad una semplice alert()



`Swal.fire()` restituisce una **promise** javascript che **verrà risolta soltanto in corrispondenza della chiusura delle finestra**. Al solito può essere risolta tramite `.then/catch` oppure `async/await`

Le opzioni di Swal.fire({ })

Di seguito un elenco delle principali opzioni utilizzabili all'interno di **`Swal.fire()`**



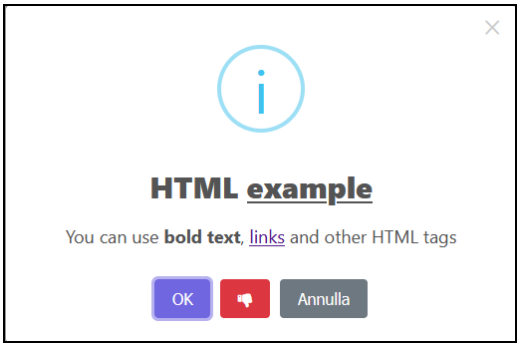

```
icon: 'success' 'error' 'warning' 'info' 'question'
iconColor:                          // colore dell'icona
title: 'Oops...'                    // titolo (accetta tag html di formattazione del testo)
text: 'Something went wrong!'       // testo del messaggio
html: 'Something <b>wrong!</b>'     // testo del messaggio formattato in html. Accetta anche un
                                       puntatore js ad un oggetto creato dinamicamente.
                                       L'intero html della finestra può essere scritto qui dentro.
color: '#716add'                                  // colore del testo
background: '#fff url(/images/trees.png)'   // colore dello sfondo ed eventuale img
footer: '<a href="#">Why do I have this issue?</a>'   // sotto il pulsante di chiusura

position: 'top-end'                 // posizione di apertura della finestra
width: 600                          // larghezza della finestra including paddings (box-sizing: border-box)
padding: '3em'                      // 3 volte rispetto al valore del font-size

showConfirmButton: true             // visualizzazione del Confirm Button (OK)
confirmButtonColor: #aaa            // colore di sfondo Confirm Button
confirmButtonText:'<i class="fa fa-thumbs-up"></i> Great!'   // dito su
```

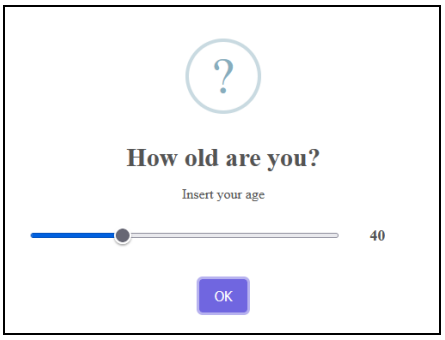
```
showDenyButton: false              // visualizzazione del Deny Button (NO)
denyButtonColor: '#aaa'            // colore di sfondo del Deny Button
denyButtonText: '<i class="fa fa-thumbs-down"></i>'          // dito giù

showCancelButton: false            // visualizzazione del Cancel Button (CANCEL)
cancelButtonColor: '#aaa'          // colore di sfondo del Cancel Button
cancelButtonText: 'Annulla'
```




```
reverseButtons: false              // se true inverte la posizione dei tre Default Buttons precedenti
showCloseButton: true              // pulsante di chiusura in alto a destra
timer: 1500                        // timer di autochiusura della finestra (con tutti i pulsanti a false)
focusConfirm: true,                // se true il focus sarà sul pulsante di conferma,
                                   altrimenti sul primo elemento della finestra
```

```
input: 'range'                     // un input type per l'inserimento di un valore
inputLabel: 'Insert your age'      // label associata al precedente tag input
inputPlaceholder
inputAttributes: {min: 20, max: 90, step: 1} // attributi del tag input
inputValue: 40                     // value dal tag input che verrà iniettato al metodo .then()
```




```
imageUrl: 'https://unsplash.it/400/200',    // visualizza una immagine
imageWidth: 400,
imageHeight: 200,
showClass: { popup:'animate__animated  animate__fadeInDown'}  // effetto in apertura
hideClass: { popup:'animate__animated  animate__fadeOutUp'}   // effetto in chiusura
```

Lettura del risultato

In tutti i casi il metodo fire() restituisce SEMPRE una Promise alla cui funzione di callback viene iniettato un oggetto **result** contenente campi indicati di seguito:

```
.then(function(result){
    console.log(result)
})
```

```
result.isConfirmed          // true se è stato premuto il Confirm Button
result.isDenied             // true se è stato premuto il Deny Button.
result.isDismissed          // true se è stato premuto il Cancel Button.
result.dismiss  // consente di approfondire la causa del cancel. I valori possibili sono:
                   Swal.DismissReason.cancel
                   Swal.DismissReason.close
                   Swal.DismissReason.esc
                   Swal.DismissReason.timer
                   Swal.DismissReason.backdrop

value:          // valore dell'eventuale input type presente.
                   Se non ci sono tag di input sarà true nel caso del Confirm Button e false negli altri due casi
```

await

L'await, come sempre, restituisce direttamente l'oggetto **result** iniettato al then

```
let result = await Swal.fire({  })
console.log(result)
```

Elementi di input multipli

Nel caso di pagine html contenenti più elementi di input, tutti i vari risultati possono essere ritornati alla funzione di callback nel modo seguente:

```
Swal.fire({
  title: 'Multiple inputs',
  html:
     '<p> nome </p>' +
     '<input id="txtNome" class="swal2-input">' +
     '<p> età </p>' +
     '<input id="txtEta" class="swal2-input">',
  preConfirm: () => {
     return [
       document.getElementById('txtNome').value,
       document.getElementById('txtEta').value
     ]
  }
}).then(function(result){
     console.log(result.value)
})
```

In questo caso **result.value** sarà un vettore enumerativo contenente tutte le voci inserite secondo l'ordine utilizzato all'interno del **preConfirm.**

In alternativa, all'interno del ***then***, è anche possibile accedere in modo diretto a qualsiasi elemento della sweetAlert mediante il puntatore js all'**ID**:

```
let nome = txtNome.value
```

Cenni sulla libreria chart.js (versione 3)

chartjs.org è una libreria JavaScript, intrinsecamente **responsive**, open-source, molto flessibile, che permette di creare rapidamente dei grafici su un canvas HTML5 (area di disegno).
Si aspetta come parametro il canvas su cui tracciare il grafico ed un json contenente sia il tipo di grafico sia i dati da visualizzare, memorizzati sotto forma di *vettori enumerativi paralleli.*

Per l'utilizzo in una applicazione javascript sono disponibili diversi **CDN**
`https://cdn.jsdelivr.net/npm/chart.js` // Ultima versione
`https://cdnjs.cloudflare.com/ajax/libs/Chart.js/4.2.0/chart.min.js`

Notare che la versione 4.2.0 è esposta come modulo **ESM** (ES6 Modules)
E' però disponibile anche la classica versione **UMD** (Universal Module Definition)
La versione `bundle.js` congloba anche `moment.js` per la gestione delle date

I tipi di grafici supportati

```
'doughnut' ->  ciambella
'pie'    ->  torta
'bar'    ->  diagramma a barre verticali
'line'   ->  diagramma per punti
'radar'  ->  diagramma a ragnatela
```

Esistono poi altri grafici a base tridimensionale (es bubble, scatter, etc.) che si aspettano come values un vettore enumerativo di JSON costituiti da tre campi:

```
{   x: number,   // coordinata X
    y: number,   // coordinata Y
    r: number    // Bubble radius in pixels.    }
```

Le impostazioni principali sono più o meno le stesse per tutti i tipi di grafico. Per ogni tipo ci sono comunque delle impostazioni specifiche come ad esempio **`fill:true`** per il `type:line` che ricolora l'area sottostante con il colore definito all'interno di `backgroundColor`

Impostazione delle chartOptions

```
let chartOptions = {
      type: 'bar',
      data: {
            labels: ['pippo', 'pluto', 'minnie'],   // x axis keys
            datasets: [{
                  label: 'risultati 2020',            // label of dataset (*)
                  data: [12, 19, 32],                 // values
                  fill: true,                         // solo per type=line
                  backgroundColor: [                 // colore delle singole barre
                        'rgba(255, 99, 132, 0.2)',
                        'rgba(54, 162, 235, 0.2)',
                        'rgba(255, 206, 86, 0.2)'
                  ],
                  borderColor: [                  // colore dei bordi delle barre
                        'rgba(255, 99, 132, 1)',
                        'rgba(54, 162, 235, 1)',
                        'rgba(255, 206, 86, 1)'
                  ],
                  borderWidth: 1    /* default 2 */
            }]
      },
      options: { }
```

Note

Nel caso si voglia utilizzare lo stesso colore per tutti i record, al posto di un vettore si può assegnare una semplice stringa con il colore desiderato

(*) Per ogni grafico è possibile rappresentare contemporaneamente **più datasets** (es risultati 2020, risultati 2021, risultati 2022, etc) ognuno con i propri dati ed i propri colori. Il campo label interno ad un singolo dataset, rappresenta il titolo del singolo dataset. .

Il **titolo principale** del grafico va messo invece all'interno della chiave **options** alla voce **plugins**.

```
options: {
      scales: {
            y: {
                  suggestedMax: 40,
                  suggestedMin: -10,    // oppure
                  beginAtZero: true
            }
      },
      plugins: {
            // titolo principale del grafico
            title: {
                  display: false,  // default
                  text: 'Main Title'
            }
            // per visualizzare il legend dei diagrammi multi-bar o pie
            legend: {
                  display: true,    // default
                  position: 'top',  // Il testo è dentro ogni singolo dataset
            },
      },
      animation: {
            onComplete: function() {alert("ok")}
      }
}
```

scales fa si che venga disegnato l'asse verticale fra i valori MIN e MAX indicati.
Utile nel caso del diagramma a barre in cui, per default, come valore minimo di partenza delle barre viene automaticamente impostato il dato con valore minimo.
Per cui l'oggetto con valore minimo avrebbe una barra con altezza zero.

animation Per default il grafico viene visualizzato con una certa animazione in ingresso (con tempo fisso di 1000 msec) che può essere disabilitata impostando
`chart.options.animation = false;`

Istanza del grafico

```
<div id="canvasContainer">
     <canvas> </canvas>
</div>
```

```
let canvas = document.getElementsByTagName('canvas')[0]  // puntatore js !!
let chart = new Chart(canvas, chartOptions);
```

Il `canvas` deve necessariamente essere inserito all'interno di un proprio tag `DIV` avente WIDTH e HEIGHT ben definiti.

Notare che il disegno del grafico lanciato tramite new Chart() è **asincrono**.

Update del grafico

L'istanza del grafico può essere fatta una sola volta all'inizio.
Per aggiornare il grafico con i nuovi dati ricevuti dal server, occorre aggiornare i due vettore **`labels[]`** e **`data[]`** e poi richiamare il metodo

```
chart.update()
```

Se invece occorre re-istanziare l'intera **chart** da associare al medesimo canvas, prima di istanziare il Chart, occorre **necessariamente** rimuovere il chart precedente con un destroy

```
if(chart)
    chart.destroy()     // oppure chart.remove()
chart = new Chart(canvas, chartOptions)
```

A questo scopo la variabile **`chart`** dovrà necessariamente essere dichiarata globale.

Responsive

Per rendere il grafico responsive occorre aggiungere le seguenti **options**:

```
responsive:true,
aspectRatio:true
maintainAspectRatio: false,
```

Dopo di che occorre assegnare al contenitore una larghezza **relativa** in funzione della larghezza della pagina. L'altezza invece può essere assegnata a piacimento.

Impostazione dinamica delle chartOptions

Tutte le chartOptions, oltre che essere definite staticamente in fase di strutturazione del grafico, possono poi anche essere impostate / modificate dinamicamente sulle singole istanze:

```
chart.options.animation.duration = 0;
```

Cenni sulla libreria crypto-js

```
<script src = "https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.0.0/crypto-
js.min.js"> </script>
```

Consente il calcolo di impronte md5, sha256, aes, base64.

Esempio :

```
let pass = CryptoJS.MD5(_txtPassword.val()).toString();
```

Lettura e Upload di un file binario

LETTURA : il controllo <input type="file">

```
Select file: <input type="file"  id="txtFile"   multiple>
```

Il controllo **<input type="file">** consente di selezionare uno o più files sul computer client tramite la tipica finestra "sfoglia". L'attributo **multiple** consente di selezionare contemporaneamente più files.

In corrispondenza dell'evento **onchange**, cioè dopo aver selezionato un file, il controllo **<input type="file">** restituisce all'interno dell'attributo dinamico **files** (accessibile solo da js) un **vettore enumerativo di Blob** (Binary Large Object) che è un contenitore generico in grado di contenere diverse tipologie di oggetti, con in aggiunta alcuni campi descrittivi.

Anche in assenza dell'attributo **multiple,** il controllo **<input type="file">** restituisce comunque **sempre** un vettore enumerativo di blob, contenente un solo file :

```
let file = $('#txtFile').prop("files")[0]
```

La proprietà **value** di input[type=file] restituisce una semplice stringa con il nome del/dei files scelti

Nel caso di un File Object i campi aggiunti all'interno del Blob oltre all'immagine binaria sono i seguenti:

"name"	nome del file scelto dall'utente (***senza il path***) Siccome l'utente può scegliere il file ovunque all'interno del file system, diventa impossibile assegnare l'immagine ad un tag **<img src=`path+name`>** perché non si dispone del path completo. Per fare una preview occorre necessariamente eseguire una conversione in base64.
"size"	dimensioni in bytes del file
"type"	contiene il MIME TYPE del file (es image/jpg)
"lastModified"	timestamp unix contenente la data dell'ultimo salvataggio del file
"lastModifiedDate"	serializzazione dell'informazione precedente

Il **contenuto binario** vero e proprio è nascosto all'interno del Blob ed è accessibile solo tramite **FileReader**

Preview dell'immagine

L'immagine scelta mediante <input **type="file">** può essere visualizzata in anteprima all'interno di un tag <img> convertendo il contenuto del Blob in base64:

```
txtFile.addEventListener("change", async function(){
  let blob = $("#txtFile").prop("files")[0];
  let base64Img = await base64Convert(blob).catch(function(err){alert(err)})
  imgPreview.src = base64Img
})

function base64Convert(blob) {
  return new Promise(function(resolve, reject){
    let reader = new FileReader();
    reader.readAsDataURL(blob);
    reader.onload = function (event) {
        resolve(event.target.result);   // event.target sarebbe reader
    };
    reader.onerror = function (error) {
        reject(error);
    };
  })
}
```

UPLOAD

L'immagine scelta mediante `<input type="file">` può essere uploadata al server in due modi:

- Come **immagine binaria** trasmessa tramite utilizzo dell'oggetto **FormData**
- Come semplice **stringa base64** trasmessa all'interno di un normale parametro **POST** di tipo string

Form Submit (approccio SSR)

```
<form action="upload.php" method="post" enctype="multipart/form-data">
    Select file: <input type="file" name="txtFiles" id="txtFile" >
    <input type="submit" value="Upload">
</form>
```

In corrispondenza del submit, il file scelto viene trasmesso al server nel formato **name=value** come avviene normalmente per tutti i controlli della form (file binari e semplici controlli), per cui sostanzialmente non c'è nulla da scrivere. Il file viene inviato in forma binaria tramite utilizzo dell'oggetto FormData.

Il contenuto del file binario deve essere inviato all'interno del body della HTTP request e ovviamente non può essere concatenato alla URL. Per cui il metodo di trasmissione può essere soltanto **POST** e non GET

La parola **`multipart`** sta ad indicare che vengono trasmessi flussi binari puri con significato logico differente (immagini, video, object, semplici stringhe)

Nota: Se all'interno della **`form`** dovessero esserci più controlli di tipo `<input type file>` l'oggetto FormData trasmesso al server presenterà più chiavi, una per ogni widget presente sulla form

L'oggetto FormData

Per trasmette una immagine binaria al server, occorre caricarla all'interno di un oggetto speciale denominato **FormData** che è un vettore ciclico, cioè un oggetto di trasmissione di tipo **key/value** in cui i names rappresentano le **chiavi** mentre i contenuti rappresentano i **values** che vengono trasmessi così come sono, cioè come flussi binari senza controlli né conversioni.

In presenza dell'attributo **`multiple`** è bene assegnare al controllo **`txtFile`** un **nome vettoriale** del tipo **`txtFiles[]`**, in modo da consentire la lettura dei dati lato server, esattamente come avviene nel caso di checkbox multipli aventi tutti lo stesso name.

Console.log() di un formData

`console.log(formData)` ha poco senso e non produce nessun effetto.
Per vedere il contenuto di una formData occorre utilizzare un ciclo del tipo :

```
for (let item of formData) {
        let name = item[0];     // chiave
        let value = item[1];    // contenuto
        console.log(name, value)      }
```

```
txtFile                                                                    index.js:25
▼File {name: "Desert.jpg", lastModified: 1247549552000, lastModifiedDate: Tue Jul 14 2009 07:32:32 GMT+0200
(Ora legale dell'Europa centrale), webkitRelativePath: "", size: 845941, …}
  lastModified: 1247549552000
  ▶lastModifiedDate: Tue Jul 14 2009 07:32:32 GMT+0200 (Ora legale dell'Europa centrale) {}
  name: "Desert.jpg"
  size: 845941
  type: "image/jpeg"
  webkitRelativePath: ""
  ▶__proto__: File
```

Caricamento di una singola immagine

In corrispondenza dell'INVIO occorre creare manualmente un oggetto **FormData** ed appendere al suo interno il contenuto di tutti i campi che si vogliono inviare al server.

```
let formData = new FormData()
formData.append("txtFile1",  txtFiles.files[0]);
formData.append('nome', txtNome.value);
formData.append('maggiorenne', chkMaggiorenne.checked
```

- **Tutti i parametri 'normali' (cioè diversi dai Blob) vengono automaticamente convertiti in stringa**
- All'intero di FormData è anche possibile passare un intero json, che però **deve essere serializzato** manualmente, altrimenti viene applicato il metodo .toString() che restituisce Obiect Obiect

```
formData.append("myJSON", JSON.stringify(myJSON))
```

Caricamento di immagini multiple

Il client dovrà scorrere il vettore enumerativo restituito da `input[type=file]` e caricare ogni singolo File all'interno del FormData

```
let formData = new FormData()
for (let file of txtFiles.files)
      formData.append('txtFiles[]', file);

let rq = inviaRichiesta('POST', 'server/upload.php', formData);
```

Il metodo **`formData.append()`** consente di accodare più oggetti ad una stessa chiave che però dovrà avere un nome **vettoriale**. La chiave viene automaticamente creata in corrispondenza del primo append(). In alternativa si potrebbero anche impostare chiavi differenti per ogni file, ma è sicuramente più scomodo.

Invio della richiesta tramite il metodo $.ajax()

Per trasmettere un flusso binario tramite **`formData`** con $.ajax()occorre aggiungere nella chiamata le seguenti intestazioni HTTP:

```
data:formData
contentType:false,
processData:false,
```

Queste impostazioni fanno sì che la funzione `$.ajax()`, prima di trasmettere il file:

- non vada ad aggiungere una Content-Type header,
- non effettui una ulteriore serializzazione dei dati da inviare

Trasmissione dell'immagine come stringa base64

L'immagine binaria restituita da `input[type=file]` può essere convertita in stringa base64 tramite il metodo **`readAsDataURL`** dell'oggetto FileReader come indicato all'inizio e poi inserita come value di un qualsiasi parametro POST. Il server, prima di salvarla su disco, dovrà riconvertirla in binario.
Ricordare però che il file base64 pesa 4:3 in più rispetto al file binario.

Come modificare l'aspetto grafico di un input type=file

Molto semplicemente si può nascondere l'input type=file ed associare una label che, quando cliccata, forza il click sull'input type=file aprendo automaticamente la finestra di scelta del file:

```
<label for="newImg" class="btn btn-info">Add an image ...</label>
<input type="file" id="newImg" style="visibility:hidden;"  >
```