Ajax

Rev Digitale 5.1 del 19/01/2022

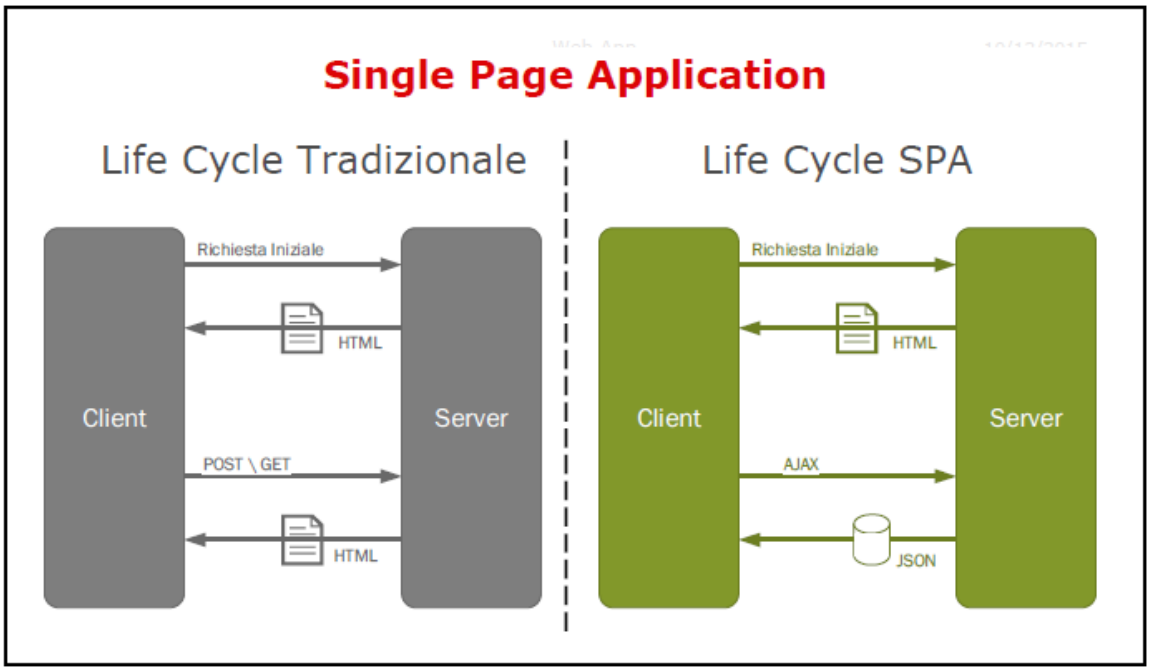
Introduzione ai Web Services

Un web service (detto anche API = Application Programming Interface) 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 conveniente (e anche molto più semplice) ricaricare di volta in volta soltanto i dati necessari a seconda dei contesti.

La tecnologia utilizzata dei browser per accedere ai dati esposti da un Web Service si chiama AJAX (Asynchronous JavaScript And XML)

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 aggiornarla tramite richieste dati successive.



Anche se Ajax nasce in ambito web, allo stato attuale il client di un web service non deve necessariamente essere un browser, ma può essere anche :

- **una normale applicazione desktop (ad esempio una applicazione C#)**
- **una app per smartphone (Android o IOS).**

Per accedere ad un servizio **non** si utilizza il pulsante di submit, ma un **normale button** che richiama una procedura java script, la quale invia la richiesta, attende i dati e li visualizza all'interno della pagina.

Tipologie di web services

Esistono diversi protocolli per la realizzazione di un web service, i principali dei quali sono:

- **Soap** Simple Object Access Protocol
- **Rest** REpresentational State Transfer

SOAP utilizza principalmente il concetto di **`servizio`**.
REST utilizza invece una visione del Web incentrata sul concetto di **`risorsa`**

REST, a differenza di SOAP, non è un'architettura nè uno standard, ma semplicemente un **insieme di linee guida** per la realizzazione di un'architettura di elaborazione distribuita (esattamente come il mondo web) **basata essenzialmente sul concetto di "collegamento fra risorse".**
Un **Web Service REST** è custode di un insieme di **`risorse`** sulle quali un client può chiedere le classiche operazioni del protocollo HTTP. L'approccio REST è molto più semplice rispetto a SOAP e tende ad esaltare le caratteristiche intrinseche del Web che è di per se una piattaforma per l'elaborazione distribuita. Non è necessario aggiungere nulla a quanto è già esistente sul Web per consentire ad applicazioni remote di interagire con i servizi REST.
Il formato standard utilizzato per lo scambio dei dati è il formato **JSON.**
Tecnicamente è comunque possibile utilizzare qualunque altro formato (XML, stringhe, Atom).

Un **Web Service SOAP** espone un insieme di **`metodi`** richiamabili da remoto da parte di un client. Utilizza HTTP come protocollo di trasporto, ma non è limitato nè vincolato ad esso, dal momento che può benissimo usare altri protocolli di trasporto (anche se in realtà http è l''unico ad essere stato standardizzato dal W3C,).

L'approccio SOAP è derivato dalle tecnologie di interoperabilità esistenti al di fuori del Web e basato essenzialmente su chiamate di procedura remota, come **DCOM**, **CORBA** e RMI. In sostanza questo approccio può essere visto come una sorta di adattamento di queste tecnologie al Web. I Web Service basati su SOAP utilizzano anche uno standard **WSDL**, *`Web Service Description Language`*, per definire l'interfaccia di un servizio. Il WSDL realizza una interfaccia del web service (in pratica un semplice schema XML) che segnala ai client ciò che il servizio è in grado di fare. Da un lato l'esistenza di WSDL favorisce l'uso di tool per creare automaticamente client in un determinato linguaggio di programmazione, ma allo stesso tempo induce a creare una forte dipendenza tra client e server. Il formato standard utilizzato per lo scambio dei dati è il formato **XML**

In conclusione, i Web service basati su SOAP costruiscono un'infrastruttura prolissa e complessa al di sopra del Web per fare cose che il Web è già in grado di fare. Il vantaggio di questo tipo di servizi è che in realtà definisce uno standard indipendente dal Web e l'infrastruttura può essere basata anche su protocolli diversi. REST invece intende riutilizzare il Web quale architettura per la programmazione distribuita, senza aggiungere sovrastrutture non necessarie.

Elementi Base di Ajax

Asynchronous JavaScript And **XML.**
E' la tecnologia utilizzata dai browser (e non solo) per accedere ai dati esposti da un web server.
Lo scopo principale è 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.
Con Ajax è possibile richiedere soltanto i dati necessari. Le applicazioni risultano così 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
- può scambiare dati in qualsiasi formato (XML, JSON, CSV o anche semplice testo)

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

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.

Utilizzo dell'oggetto XMLHttpRequest

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 valifo oppure no.

A tale scopo può essere utilizzato l'evento onChange (o 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);
    }
```

Invio di una richiesta Ajax tramite jQuery

Il metodo statico **`$.ajax()`** (non presente nella libreria jquery **slim**) rappresenta un semplice ed ottimo wrapper dell'oggetto java script **`XMLHttpRequest`** per l'invio di una richiesta ajax ad un server.

Si aspetta come parametro un **json** costituito dai seguenti campi:

```
$.ajax({
   url: "/url?nome=pippo",
   data: { "nome": "pippo" },
   type: "GET",        // default
   contentType: "application/x-www-form-urlencoded; charset=UTF-8", // default
   dataType: "json",  // default
   async : true,       // default
   timeout : 5000,
   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",
})
```

Eventuali parametri possono essere indifferentemente

- accodati alla url *oppure*
- inseriti all'interno del campo **data**

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
```

Impostando **`contentType`**`:"application/x-www-form-urlencoded"`, se i parametri vengono passati a $.ajax() come oggetto JSON (*non serializzato*), **$.ajax() provvede automaticamente a convertirli in formato urlencoded**. E' però ammesso un unico oggetto e non vettori di oggetti o oggetti annidati. Nel caso di oggetti annidati occorre eseguire una conversione manuale:

```
let json = {
   nome:"pippo",
   location: {city:"London", street:"Carnaby Street"}
}
let result = $.param(json).replaceAll("%5B", ".").replaceAll("%5D", "")
// nome=pippo&location[city]=London&location[street]=Carnaby+Street
%5B e %5D sono le parentesi quadri che json-server non riconosce. Quindi occorre un replaceAll
// nome=pippo&location.city=London&location.street=Carnaby+Street
```

I parametri **POST** possono essere passati anche in un formato **`contentType:"application/json"`**
In questo caso, prima di inviare la richiesta, occorre SERIALIZZARE i parametri 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"
dataType:"jsonp"
```

- 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 è sicuramente comoda ma costringe il server a restituire sempre una risposta in formato corretto. Cioè se il client si aspetta ad esempio un elenco json di nominativi ed il server non trova nessun nominativo, deve restituire un oggetto vuoto e non una stringa che provocherebbe errore sul parser.
`textStatus` indica lo stato in cui si è conclusa la XMLHttpRequest
`jqXHR` è un wrapper dell'oggetto XMLHttpRequest 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 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 dipendente dal codice di errore restituito dal server

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

Nota

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.

Deferred and Promise

Si tratta di oggetti simili mirati a facilitare la gestione dei processi asincroni.

La programmazione asincrona interviene nel momento in cui si eseguono operazioni “lente” che altrimenti bloccherebbero l’interfaccia grafica; ad esempio l’elaborazione grafica di una immagine oppure l’attesa di ricezione dati da un server esterno. In questi casi il sistema si prende **l'incarico** di invocare la funzione di callback al momento opportuno, e ritorna immediatamente il controllo all’applicazione.

Sono frequenti anche i casi in cui un’operazione asincrona deve essere eseguita in coda ad un’altra operazione ascinrona, dove ogni operazione dipende dal risultato dell’operazione precedente. In questi casi occorre annidare le callback una dentro l’altra con estensione del codice verso destra creando quella che è conosciuta come "**Piramide della sventura**" (Pyramid of Doom) orientata da sinistra verso destra.

Le Deferred / Promise consentono di ritornare ad una scrittura verticale riportando il codice ad una maggiore separazione e leggibilità

L’oggetto Promise in javascript

In javascript esiste un comodissimo oggetto Promise che consente di “wrappare” una procedura asincrona all’interno di una nuova procedura all’apparenza sincrona.

Si supponga di avere una procedura asincrona **elaboraImmagine**() molto pesante in termini di esecuzione e di righe di codice che esegue l’elaborazione di una immagine richiamando in cascata diverse funzioni di libreria ciascuna delle quali esegue delle operazioni asincrone annidate e inietta il risultato ad una propria callback. L’ultima callback riceverà l’immagine finale rielaborata.

```
function elaboraImmagine(img){
      ---------- elaborazione ----------------
      _lastLibrary.onEnd(err, finalImage){
            console.log(finalImage)
      }
}
```

Se invece di eseguire un semplice `console.log()` questa **`finalImage`** dovesse essere inviata ad un server o comunque gestita dalla nostra applicazione, il codice di gestione della **`finalImage`** dovrà essere scritto nell’apice destro della Pyramid of Doom.
Cioè tutto il codice di elaborazione dell’immagine viene a ‘mescolarsi’ con il flusso principale del programma diminuendo fortemente la leggibilità.

A livello di programmazione sarebbe molto più comodo se **`elaboraImmagine`** fosse sincrona, cioè bloccante, bloccando il programma fino al termine dell’elaborazione:

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

- Nel momento in cui viene richiamato il metodo **resolve**, automaticamente la promise provvede a generare l'evento **then()**, iniettandogli il risultato passato a resolve, cioè **finalImage**.
- Nel momento in cui viene richiamato il metodo **reject**, automaticamente la promise provvede a generare l'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 **return** finale ritorna la Promise al chiamante il quale si limiterà a gestire le seguenti semplicissime tre righe di codice, in cui tutta la parte di elaborazione dell'immagine è spostata dentro `elaboraImmagine()` che non crea più nessun tipo di interferenza con il flusso principale:

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

Esempio 2

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

let promise = foo()
promise.then(function(msg){console.log(msg)} )
```

Le Deferred/Promise in jQuery ed il metodo statico $.Deferred()

In jQuery per gestire le Promise occorre utilizzare due oggetti che sono **Promise** e **Deferred**.

- L'oggetto **Deferred** è quello che va istanziato per gestire il codice asincrono (a cui vengono iniettati i puntatori ai metodi `resolve, reject, notify`. (`notify` consente di inviare delle notifiche al chiamante sullo stato di avanzamento dell'elaborazione).
- In corrispondenza dell'istanza, l'oggetto **Deferred** restituisce una **Promise** jQuery la quale rende disponibile gli eventi **done e fail** che sono gli equivalenti del **then** e **catch** di javascript

Di seguito il codice precedente riscritto con la sintassi jQuery:

```
function elaboraImmagine(img){
    let promise $.Deferred(function(deferred) {
       ---------- elaborazione ----------------
        _lastLibrary.onEnd(err, finalImage){
            if(err)
               deferred.reject(err)
            else
               deferred.resolve(finalImage);
    })
    return promise;
}

let request = elaboraImmagine(img)
request.fail(function(err) {
     console.log(err.message)
})
request.done(function(finalImage) {
     console.log(finalImage)
})
```

Il metodo statico **`$.Deferred()`** :

- istanzia un nuovo oggetto **`Deferred`**
- inietta alla funzione di callback un puntatore **`deferred`** all'oggetto Deferred medesimo, il quale dispone dei due metodi **`resolve`** e **`reject`** esattamente come in java script.
- Ritorna una **`Promise`** che rende disponibili gli eventi **`.done`** e **`.fail`**

L'oggetto **`Promise`** jQuery dispone in realtà di 6 eventi che sono:
`done, fail, progress, then, catch, always`

L'evento **`promise.done()`** viene richiamato in corrispondenza del metodo **`d.resolve()`**
L'evento **`promise.fail()`** viene richiamato in corrispondenza del metodo **`d.reject()`**
L'evento **`promise.progress()`** viene richiamato in corrispondenza del metodo **`d.notify()`**
L'evento **`promise.then()`** accetta tre funzioni di callback:

- la prima richiamata in corrispondenza del metodo deferred.resolve()
- la seconda (facoltativa) richiamata in corrispondenza del metodo deferred.reject()
- la terza (facoltativa) richiamata in corrispondenza del metodo deferred.notify()

```
promise.then(
    function(){ alert("ok")    },
    function(){ alert("error")},
    function(){ alert("in progress ....")
});
```

Se usato con una sola callback il metodo di evento **`.then()`** diventa esattamente equivalente al metodo di evento **`.done()`.**

L'evento **`promise.always()`** accetta due funzioni di callback:

- la prima richiamata in corrispondenza del metodo deferred.resolve()
- la seconda (facoltativa) richiamata in corrispondenza del metodo deferred.reject()

L'evento **`promise.catch()`** (supportato fino alla versione 3.6.0) è simile a **`promise.fail()`**, però:

- **`promise.catch(fn)`** "**gestisce l'errore**" e restituisce una **nuova** promise alla quale potrà eventualmente essere accodato un `then(fn)` successivo che verrà quindi eseguito (fino a 3.6.0)
- **`promise.fail(fn)`** "termina" l'esecuzione e restituisce la stessa `promise` ormai risolta, per cui eventuali `then(fn)` successivi NON verranno più eseguiti.

Note

1) Le funzioni di callback di done() e fail() possono ricevere uno o più parametri di qualunque tipo che dovranno essere settati adeguatamente in fase di chiamata nei metodi resolve() e reject() e che, di conseguenza, verranno poi iniettati nelle callback dei metodi done() e fail()

2) Ad ogni promise è possibile associare più funzioni di callback (eventualmente anche in cascata tra di loro) che verranno 'lanciate' sequenzialmente una dopo l'altra

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

3) Gli eventi .done() e .fail() vengono eseguiti **anche** se la loro assegnazione viene eseguita *dopo* che le operazioni di $.Deferred() sono terminate (nel qual caso verranno eseguiti immediatamente).

Esempio 2

```
_div1.show(3000, function(){
     console.log("ok");
});
```

potrebbe essere riscritto nel modo seguente:

```
var promise = $.Deferred(function (deferred){
     _div1.show(3000, function(){
          deferred.resolve();
      });
});
promise.done( function(){
     console.log ("ok");
});
```

Esempio 3

Anziché scrivere il codice dentro **`$.Deferred()`** si può utilizzare il puntatore restituito da **`$.Deferred()`**

```
function visualizza (selector, speed) {
     var d = $.Deferred();
     var s  = speed || "slow";
     $(selector).show(s, d.resolve);
     return d;
}
$.when(
      visualizza('#div1', 4000),
      visualizza('#div2', 3000)
 ).done(function () {
      alert('Animazioni terminate');
});
```

Il metodo statico `$.when(primise1, promise2, promise3)` è un metodo che viene eseguito solo dopo che ***tutte*** le promesse ricevute come parametro sono state terminate con esito positivo.
Restituisce un oggetto promise al quale possono essere applicati i vari eventi precedenti.
Eventuali parametri passati a **`d.resolve(par)`** vengono automaticamente propagati alla funzione di callback dell'evento done: **`.done(function(par){})`**

Le istanze **d** ritornate da visualizza sono utilizzate all'interno del metodo statico `$.when` per attendere che ogni promise sia terminata con successo.

Utilizzo delle Deferred / Promise con $.ajax()

Il metodo $.ajax(), così come tutti i vari metodi

```
$.get({})
$.post({})
```

Implementano tutti l'interfaccia **Promise** per cui, in fase di chiamata, viene creato un oggetto Deferred all'interno del quale viene incapsulata la funzione $.ajax() che restituisce una Promise.
In corrispondenza della corretta terminazione della funzione di callback viene generato l'evento .**done**
In caso di errore viene generato l'evento **.fail**
Allo stesso modo vengono generati gli eventi .**always** e **.then**

Utilizzo delle promise con $.get()

Si consideri il seguente esempio in cui :

- il primo servizio restituisce una certa URL,
- il secondo servizio richiede dei dati alla URL ricevuta
- il terzo servizio elabora i dati restituendo un risultato finale

Soluzione tradizionale

```
$.get("service1/geturl", function(url) {
   $.get(url, function(data) {
       $.get("service3/" + data, function(ris) {
            $("#output").append("Il risultato è: " + ris);
       });
   });
});
```

Soluzione Deferred

```
var promise1 = $.get("service1/geturl");
var promise2 = promise1.done( function(url) {
     return $.get(url);
});
var promise3 = promise2.done( function(data) {
     return $.get("service3/" + data);
});
promise3.done( function(data2) {
     $("#output").append("Il risultato è: " + data2);
);
```

// oppure, in forma ancora più compatta

```
$.get("service1/geturl")
   .done( function(url) {
       return $.get(url);
    })
   .done( function(data) {
       return $.get("service3/" + data);
    })
   .done( function(data2) {
        $("#output").append("Il risultato è: " + data2);
    });
```

Riscrittura della funzione inviaRichiesta tramite utilizzo delle Promise

```
const URL = "https://randomuser.me"

function inviaRichiesta(method, url, parameters={}) {
    let contentType;
    if(method.toUpperCase()=="GET")
        // Nelle chiamate GET i parametri vengono passati in urlencoded
        contentType="application/x-www-form-urlencoded;charset=utf-8";
    else{
        // Nelle chiamate POST i parametr vengono passati in json serializzato
        contentType = "application/json; charset=utf-8"
        parameters = JSON.stringify(parameters);
    }
    return $.ajax({
        "url": URL + url,
        "type": method,
        "data": parameters,
        "contentType": contentType,
        "dataType": "json",   // default
        "timeout": 5000,      // default
    });
}
```

Per inviare uan richiesta al server Il client potrà utilizzare il seguente codice:

```
var request = inviaRichiesta("get",  "/api?results=100");          // oppure
var request = inviaRichiesta("get",  "/api", {"results":100});
request.fail(errore);
request.done(function(data, test_status, jqXHR){
    console.log(data);
});

function errore (jqXHR, test_status, 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);
}
```

L'approccio deferred / promise, oltre che molto più leggibile, presenta due ulteriori vantaggi:

- E' possibile associare più funzioni di callback in risposta alla medesima richiesta
  `request.done(function1).done(function2);`
- E' possibile associare una funzione di callback ad una promise anche dopo che la funzione asincrona è terminata, nel qual caso la nuova callback viene eseguita immediatamente.

La funzione $.getJSON()

La funzione **`$.getJSON()`** è molto simile a **`$.ajax()`**. Presenta però un unico parametro obbligatorio che è **`url`** e, al suo interno, provvede a richiamare **`$.ajax()`**.

- `$.getJSON()` può eseguire SOLTANTO chiamate **GET;** il parametro **method** viene automaticamente impostato a GET
- Nel caso di chiamate GET i parametri possono essere accodati alla URL per cui, anziché passarli in un campo separato DATA, `$.getJSON()` accetta soltanto parametri accodati alla url

- `$.getJSON()` dopo la url presenta due funzioni di callback facoltative, la prima è quella corrispondente a `success`, la seconda è quella corrispondente a `error`.
- Anziché passare le due funzioni di callback come 2° e 3° parametro è possibile utilizzare la **promise** restituita da `$.getJSON()` esattamente come nel caso di `$.ajax()`

```
let request = $.getJSON(" https://www.alphavantage.co/query?........");
request.done(function(data){  });
request.fail(function (jqXHR, test_status, str_error){  });
```

Esempi di Scambio di dati tra client e server

Parsing e Serializzazione di uno stream JSON in javascript

Il metodo statico **`JSON.stringify(obj)`** serializza un qualunque oggetto JSON in una stringa.
Il metodo statico **`JSON.parse(str)`** parsifica una stringa JSON e restituisce l'oggetto corrispondente

invio Richiesta ed Elaborazione della risposta

```
var request = inviaRichiesta("get", "elencoFiliai.php", "codBanca=7");
request.done(function(data){
      lstFiliali=$("#lstFiliali");
      for (var i=0; i<data.length; i++){
            var filiale = data[i];
            var option = $("<option></option>");
            option.val(filiale["cBanca"]);
            option.html(filiale["Nome"]);
            lstFiliali.append(option);
      }
      lstFiliali.prop("selectedIndex", -1);
)};
request.fail(errore);
```

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 pregiudicherebbe la conversione in json dello stream ricevuto.
  Esempi di risposte 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`** .
Cioè 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.

Note sulla scrittura del codice lato client

- Fare molta attenzione a NON eseguire l'associazione di eventi all'interno di un evento .**done**() oppure all'interno di cicli, altrimenti una nuova associazione viene eseguita in corrispondenza di ogni done, e poi l'evento si verifica più volte. In alternativa, in corrispondenza di ogni evento, prima del .on("click") si potrebbe richiamare SEMPRE .**off()** che rilascia tutti gli handler di evento associati all'oggetto.

- 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. In alternativa si potrebbero utilizzare i **delegated events**

- 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).

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);
        }
    }
  }
```

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, esattamente come se i dati provenissero da un vero web server.

Passi necessari all'installazione ed utilizzo di json-server

1. Installare NodeJS (https://nodejs.org)
2. Installare globalmente Json-Server:
   `npm install -g json-server`
3. Creare una cartella di lavoro contenente i file del progetto ed entrarci
4. Scrivere nella cartella di progetto un qualunque file.json
5. Aprire un terminale nella cartella di lavoro
6. Se si è opportunamente configurata la variabile d'ambiente PATH, json-server potrà essere eseguito da qualsiasi cartella
7. 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
8. Il server risponde con un messaggio che conferma l'attivazione in localhost sulla porta **3000**

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 chiave di primo livello, il server risponde inviando l'intero contenuto associato a quella chiave: **http://localhost:3000/mainKey.** Come parametro concatenato alla url è possibile passare **qualsiasi chiave di primo livello** presente all'interno del file db.json
- Le chiavi di primo livello possono avere come contenuto soltanto vettori enumerativi o associativi. Non sono ammessi valori scalari come interi, booleani e nemmeno stringhe.
- **Ogni record DEVE avere un id numerico, altrimenti insert, update e delete vanno in errore**
- 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 --watch db.json
  ```

Servizi esposti

In realtà json-server è un vero e proprio ***crud server*** in grado di servire tutte le seguenti richieste ed aggiornando automaticamente i dati sul file. Si consideri ad esempio la seguente chiave:

```
persons: [
{     "id": 1,
      "nome": "Alfio",
      "genere": "m",
      "classe": "4B"
},
{     "id": 2,
      "nome": "Beatrice",
      "genere": "f",
      "classe": "4B"
},
{     "id": 3,
      "nome": "Carlo",
      "genere": "m",
      "classe": "1C"
}]
```

I vari tipi di chiamate GET

Sia `/persons` la **risorsa** richiesta

GET `/persons` senza parametri restituisce un *vettore di record* contenente tutte le persone
GET `/persons/2` restituisce un *singolo record* relativo alla persona avente **id=2**
GET `/persons?genere=m&classe=4B.` Esegue una **AND** e restituisce un vettore di record
GET `/persons?classe=1B&classe=2B.` Esegue una **OR** e restituisce un vettore di record

I parametri normalmente devono essere accodati alla risorsa in formato **url-encoded** cioè suddivisi dalla risorsa stessa tramite un **?** come negli esempi precedenti: `/persons?genere=m&classe=4B`

Fa eccezione l'ID che può essere passato al server in due modi:

- Come normale parametro **url-encoded** `/persons?id=2`
  nel qual caso la chiamata restituisce un normale *vettore di record*
- **Accodato direttamente alla risorsa tramite slash:** `/persons/2`
  In questo caso la chiamata restituisce non un *vettore di record* ma un *singolo record*

Inoltre, per come è stata scritta invia Richiesta(), i parametri, anziché accodati in formato **url-encoded**, possono essere passati a inviaRichiesta() come terzo parametro in formato JSON (molto più comodo). invia RIchiesta() provvederà lei a **convertire** il json in formato url-encoded accodandolo alla risorsa.

```
inviaRichiesta("GET", "/persons", {"genere":"m", "classe"="4B"})
```

AND fra i parametri

I parametri url-endoded possono essere più di uno, nel qual caso il server esegue una AND fra i parametri
GET `/persons?genere=m&classe=4B` // Studenti della classe 4B di genere maschile

OR fra i parametri

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
GET `/persons?classe=1B&classe=2B.` // Restituisce *sia* gli studenti della 1B *sia* gli studenti della 2B

Notare che in questo caso i parametri possono SOLTANTO essere accodati al nome della risorsa in formato url-encoded e non passati come terzo parametro in formato JSON .
Infatti non è consentito creare all'interno di un json più chiavi aventi lo stesso nome.

Note:

- Se in coda alla url viene aggiunto un ? senza successivi parametri NON è un problema
- Se in coda alla lista dei parametri permane una &, NON è un problema
- Le ricerche possono anche accedere agli oggetti interni: **`/persons?location.country=italy`**
- Il parametro ?**`q=testo`** ricerca su tutti i campi il testo indicato (**`full search`**)

Altri metodi di chiamata

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

il campo ID può anche non essere assegnato, nel qual caso json-server provvede automaticamente ad assegnare al nuovo record un ID pari al **maggiore fra tutti gli ID presenti nel DB**, incrementato di 1.
Non sembra possibile postare più record in una stessa chiamata.

DELETE `/persons/2` `elimina il record avente id=2. Non ha parametri.`

PUT `/persons/2` `sostituisce il record id=2 con quello passato come` **`parametro`**
PATCH `/persons/2` `simile al precedente, ma aggiorna soltanto i singoli campi.`

`PATCH`, a differenza di `PUT` (che sostituisce completamente l'intero record), aggiorna soltanto i campi passati come terzo parametro di tipo json, per cui non è necessario passare tutti i campi.
E' anche possibile passare uno o più campi che non sono esistenti all'interno del database, nel qual caso verranno automaticamente aggiunti al record corrente.

La codifica base64

La codifica Base64 è una tecnica per codificare i dati binari in forma testuale tramite un insieme di **caratteri ASCII** noti a tutti i sistemi informatici. In pratica **è una tecnica di serializzazione dei file binari in modo da aumentare la sicurezza di trasmissione a scapito di un incremento delle dimensioni**.

Note:

- La codifica Base64 nasce nei sistemi di posta elettronica che inizialmente gestivano solo dati testuali. Base64 è stata introdotta per poter allegare ad una mail immagini o file binari che diversamente avrebbero potuto essere trasmessi non correttamente o 'modificati'.
- In internet le richieste ajax e le relative risposte vengono trasmessi SEMPRE SOLO come stringhe. Si pensi ad esempio al caso in cui si vuole trasmettere una immagine come parametro url-encoded: alcuni codici binari (quasi tutti quelli < 32, cioè il backspace, tab, CR, etc.) sarebbero difficilmente scrivibili nella URL con rischi di compromettere gli altri dati (il backspace cancellerebbe il dato precedente). Con base64 questi caratteri speciali vengono trasformati in normali caratteri ascii

Algoritmo di codifica:

La codifica Base64 opera nel modo seguente:

- suddivide la sequenza binaria in gruppi di 6 bit. Le combinazioni possibili su 6 bit sono 64
- I 64 caratteri ASCII utilizzati per la codifica base64 sono le 26 lettere minuscole, le 26 lettere maiuscole, i 10 caratteri numerici e i caratteri speciali + e /
- memorizza ciascuna delle 64 combinazioni a 6 bit tramite un preciso carattere ASCII (ad esempio 000000 -> 'A', 000001 -> 'B', 000010 -> 'C', 000011 -> 'D', etc.)
- Il carattere corrispondente a ciascuna combinazione di 6 bit viene salvato con il suo normale codice ascii, cioè 'a' -> 01100001, 'A' -> 01000001, '0' -> 00110000, '+' -> 00101011
- Per cui i 6 bit dello stream originale vengono salvati su 8 bit base64.
  Un dato codificato in base64 è 1.333 volte più grande (4:3) rispetto al file originale

Conversione di una immagine da binaria a base64

Esistono diversi online converter che restituisco il base64 di una immagine selezionata tramite file system.

Una volta scelta l'immagine tramite il tag **<input type='file'>**
il codice da utilizzare per la conversione è il seguente, basato sull'utilizzo dell'oggetto **FileReader:**

HTML:
```
<input type="file" id="txtFile" onchange="elabora()" />
```

JavaScript:
```
function elabora() {
   let file = $("#txtFile").prop("files")[0];
   let reader = new FileReader();
   reader.readAsDataURL(file);
   reader.onloadend = function() {
      console.log(reader.result)
   }
}
```

Cioè all'interno dell'evento **reader.onloadend** il campo **reader.result** conterrà la codifica base64 del file selezionato, codifica che potrà essere trasmessa al server come normale parametro url-encoded. Il contenuto di **reader.result** sarà il seguente:

" **data:image/jpeg;base64**,codificaBase64iVBORw0KGg+oAAAANSUhEUgAABsYAAA/R4C....."

Tecnologie - Classe Quarta robertomana.it
Ajax

Concetto di URI - L'URI data:

URI sta per **Uniform Resource Identifier** e rappresenta uno **schema** di identificazione delle risorse.
Si parla infatti normalmente di **URI Scheme**.

- Un comune esempio di URI è costituito dalle web URL (uniform resource locator).
  Ad esempio ***http://www.google.com*** definisce un risorsa web (`www.google.com`) con il corrispondente protocollo di accesso (**`http`**)
- Le URI sono una estensione del concetto di URL applicabili a qualsiasi tipo di risorsa

Particolarmente importante nella programmazione web è l'URI **`data:`** che consente di inserire all'interno di una pagina html dei dati in forma testuale esattamente come se si trattasse di una risorsa esterna.
Ad esempio una immagine in formato base64:

```



```

Questa tecnica di incorporare oggetti testuali all'interno della pagina consente di caricare questi oggetti nell'ambito di una unica richiesta HTTP velocizzando quindi il caricamento della pagina.

Talvolta, per velocizzare l'accesso alle immagini, queste vengono salvate direttamente nel DB come stringhe base64. Può andare bene quando le immagini da salvare non sono troppe e non troppo pesanti.

Nota: Una immagine base64 può ovviamente essere utilizzata come immagine di sfondo di un tag DIV.
Anche in questo caso però (come per i path) è sempre obbligatorio anteporre la funzione **`url()`**

```
if (immagine.startsWith("data:image/"))
    _div.css('background-image', "url(" + immagine + ")");
```

Struttura completa di una URI schema

La sintassi dell'URI **`data:`** utilizza il cosiddetto **DATA URI SCHEME** che opera nel modo seguente:

`data:[][;charset=""][;base64],`

- Il primo parametro rappresenta il **media/type** della risorsa terminato da un punto e virgola; ad esempio `image/png` oppure `application/pdf`
- Il secondo parametro rappresenta la codifica di carattere (default "`utf-8`").
  Nel caso delle immagini di solito viene omesso
- Il terzo parametro rappresenta la codifica utilizzata nella conversione binario-testo che, nel caso delle immagini, è SEMPRE base64.
- Dopo il terzo parametro c'è una **virgola** che è obbligatoria e che serve da **separatore** tra l'intestazione e la codifica vera e propria base64 dell'immagine.
- La forma minima di una data URI risulta pertanto essere la seguente **`data:,`**

Altri esempi di URI :

```
"tel:3337684747"
"sms:3337684747"
"mailto:nome.cognome@vallauri.edu"
```

pag 19

L'attributo "download" del tag <a>

In HTML5 al tag **<a>** è stato aggiunto un attributo **download** che, anziché caricare il contenuto della risorsa indicata dall'attributo **href,** consente di aprire una **Finestra di Dialogo** del browser la quale, tramite il pulsante Sfoglia, consente di scegliere dove salvare sul pc client la risorsa puntata da **href**.

```
<a href="/img/myImage.jpg" download > salva immagine </a>
```

All'attributo **download** si può assegnare un valore che rappresenta il nome di file proposto come default nella Finestra di Dialogo, nome che l'utente evidentemente potrà modificare a suo piacimento.

- Se questo valore viene omesso, la finestra propone il nome originale del file oppure un nome random nel caso di data: o blob:
- E' ammesso come **href** qualunque tipo di file (html, pdf, txt, img, etc).
- Sono anche ammesse URI del tipo **blob:** e **data:** che sono molto comodi per poter accedere e scaricare un contenuto generato dinamicamente tramite javascript.
- data: è riferito tipicamente a immagini / pdf memorizzate in formato base64.

C'è però una particolarità: l'attributo **download** **funziona SOLO nel caso di URL intra-domain**, cioè la pagina ed il file devono condividere lo stesso dominio, sottodominio e protocollo di accesso (**CORS**=Cross-origin resource sharing). Fanno eccezione le risorse in formato **blob:** e **data:** per le quali il controllo CORS è disabilitato ed il download è SEMPRE consentito.
Per le immagini binarie no, per cui l'esempio iniziale funziona solo utilizzando un server.

Come salvare un json su disco

Si supponga di avere il seguente tag HTML

```
<a href="#" download="db.json"> salva json su disco </a>
```

All'interno del javascript, in corrispondenza del click su un certo pulsante "SALVA" si può eseguire il seguente codice:

```
let json = {"utente":"pippo", "eta":16}
json = JSON.stringify(json, null, 3)
let blob = new Blob([json], {'type':'application/json'});
$("a").prop("href", window.URL.createObjectURL(blob));
```

In pratica costruisco dinamicamente un json, lo serializzo, lo trasformo in **Blob**, creo la URI e la memorizzo all'interno di **href**. Se l'utente clicca sul tag <a> prima che vengano eseguire queste righe sostanzialmente non succede nulla perché **href** è vuoto. Dopo che queste righe sono state eseguite e la proprietà **href** impostata, quando l'utente cliccherà sul tag <a> si aprirà automaticamente la finestra di dialogo che chiederà all'utente dove salvare il file "db.json".
Nota: In firefox è obbligatorio anteporre window.URL davanti a URL !!

L'oggetto Blob

Un Blob (letteralmente ***grumo***) è un oggetto simile ad un file temporaneo in memoria. La sua peculiarità consiste nella possibilità di utilizzare il metodo **URL.createObjectURL()** che restituisce una URI del Blob esattamente come se si trattasse di un file fisico residente sul server.

Questa URL può poi essere normalmente passata alla proprietà **src** del tag <img>, alla proprietà **href** del tag <a>, alla funzione **url()** di una CSS property
Nell'esempio precedente, prima si trasforma la variabile json in un file temporaneo denominato **blob**, poi tramite il metodo **URL.createObjectURL()** si assegna una URL a questo file temporaneo come se si trattasse di un file fisico sul server e si salva questa url all'interno della proprietà **href** del tag **<a>**

Il **costruttore** richiamato in corrispondenza del **`new Blob()`** restituisce un nuovo Blob contenente il concatenamento di tutti i dati passati nel vettore relativo al 1° parametro.
Il 2° parametro indica il MIME Type relativo ai dati contenuti all'interno del 1° parametro.

Miglioramento della soluzione precedente

Un miglioramento alla soluzione precedente consiste nel creare dinamicamente il tag <a> soltanto nel momento in cui il json è pronto per essere salvato. A questo scopo si sfrutta di solito l'evento **`click`** del tag <a> **che viene eseguito prima del link indicato da href**. Per cui diventa possibile creare il Blob e settare l'attributo **`href`** nel momento stesso in cui si crea il tag <a> da utilizzare per eseguire il download.

```
$("<a>").prop({"download":"db.json", "href":"#"}).text("salva json su disco ")
 .appendTo(wrapper).on("click",
    function(){
            let json = {"nome":"pippo", "eta":16}
            json = JSON.stringify(json, null, 2)
            let blob = new Blob([json], {type: 'text/plain'});
            $(this).prop("href", URL.createObjectURL(blob));
    }
)
```

Il valore iniziale `"href":"#"` è messo affinché il link assuma graficamente il solito aspetto di collegamento ipertestuale. Omettendo href sul tag <a>, esso non funge più da collegamento ipertestuale.

Come salvare un canvas su disco

```
$("<a>").prop({"download":"newImage.png", "href":"#"}).text("salva immagine")
 .appendTo(wrapper).on("click", function(){
     $(this).prop("href", canvas.toDataURL("image/png"))
 })
```

Il codice è molto simile a quello dell'esempio precedente. L'oggetto **`canvas`** dispone anch'esso di un metodo **`toDataURL`** simile al **`URL.createObjectURL()`** dell'esempio precedente. Restituisce il contenuto del canvas corrente come URI **`data`** codificata in formato base64 (**`data:image/png;base64`**)

Come salvare un'immagine binaria su disco

Se si dispone della URL e la richiesta è intra-domain si può utilizzare l'esempio iniziale.
In alternativa si può passare attraverso un Blob oppure, più frequentemente, attraverso un Canvas.

```
$("<a>").prop({"download":"newImage.jpg", "href":"#"}).text("salva immagine")
 .appendTo(wrapper).on("click", function(){
     const img = $('#img1');
     const dataUrl = _getDataUrl(img);
     $(this).prop("href", dataUrl);
})

function _getDataUrl(img) {
     const canvas = $('<canvas>')[0];
     const ctx = canvas.getContext('2d');
     canvas.width = img.css("width");
     canvas.height = img.css("height");
     ctx.drawImage(img, 0, 0);
     // in caso di risorsa extra-domain, il metodo toDataURL va in errore.
     return canvas.toDataURL('image/jpeg')   }
```

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 molto cambiata ed è ***incompatibile*** rispetto alla sweetAlert 1 iniziale.
Viene distribuita come JS + CSS oppure **ALL** che contiene sia js che css
L'ultima versione della sweetAlert2 (gennaio 2022) è la versione**11** che può essere linkata 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 presenta molti overload differenti. Si tratta di un metodo asincrono che restituisce una promise javascript la quale al solito potrà essere risolta tramite il metodo **`.then().`**

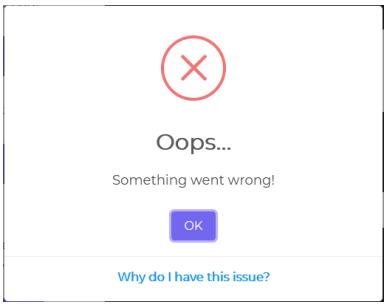
I parametri del metodo fire()

Passando una semplice stringa la sweetAlert la visualizza esattamente come una normale alert()

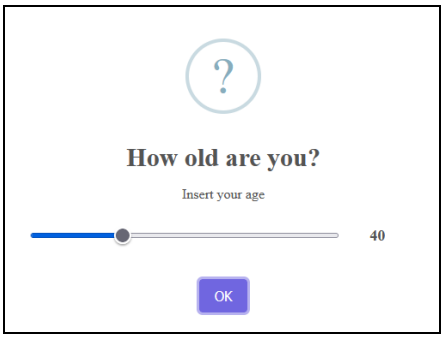
```
Swal.fire('Any fool can use a computer')
```

Come parametro è però possibile passare un json molto strutturato i cui campo principali sono i seguenti:

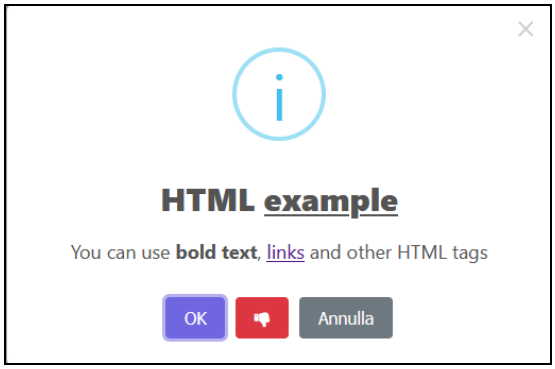
```
icon: 'success' 'error' 'warning' 'info' 'question'
iconColor:                          // colore dell'icona
title: 'Oops...'                    //  titolo (accetta tag html di formattazione del testo)
text: 'Something went wrong!'       //  testo del messaggio
footer: '<a href="#">Why do I have this issue?</a>'   //  sotto il pulsante di chiusura
```




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




```
color: '#716add'                    // colore del testo
width: 600                          // larghezza della finestra including paddings (box-sizing: border-box)
padding: '3em'
background: '#fff url(/images/trees.png)',
html: 'html tags'                   // html tags come in un normale tag div.  Ai vari elementi è
                                       anche possibile applicare classi definite all'interno del file CSS
position: 'top-end'                 // dove posizionare la finestra

showCloseButton: true               // pulsante di chiusura in alto a destra

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

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

showCancelButton: false             // viene visualizzato il Cancel Button (CANCEL)
cancelButtonColor: '#aaa'           // colore di sfondo del Cancel Button
cancelButtonText: 'Annulla'

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
reverseButtons: false               // se true inverte la posizione dei tre Default Buttons precedenti
```




```
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

Come detto sweetAlert2 restituisce una Promise alla cui funzione di callback viene iniettato un oggetto **result** contenente i seguenti campi:

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

```
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 del singolo html input type presente.
                   Se non ci sono tag di input sarà true nel caso del Confirm Button e false negli altri due casi
```

await

Come per tutte le Promise invece del .then() è possibile utilizzare **await** all'interno di una funzione **async**

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

Pagina HTML completa

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** o **preDeny**

Cenni sulla libreria chart.js (versione 3)

chartjs.org è una libreria JavaScript, **responsive**, open-source, molto flessibile, che permette di creare rapidamente dei grafici su un canvas HTML5 (area di disegno), con dati e opzioni in formato JSON. 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**
(Prestare attenzione al fatto che sia la **versione 3** in cui sono cambiate diverse cose rispetto alla versione precedente, come ad esempio la possibilità di impostare le dimensioni direttamente sul canvas senza dover far ricorso ad un tag DIV esterno). Oppure può anche essere scaricata dal sito **chartjs.org**

E' costituita sostanzialmente da 3 file

```
chart.css
chart.js
chart.bundle.js
```

La versione `bundle.js` congloba anche `moment.js` per la gestione delle date

Esempio

Di seguito è riportato un codice esplicativo dal sito **chartjs.org / getStarted**
Le impostazioni sono sostanzialmente le stesse per qualsiasi tipo di chart.
Tra un tipo e l'altro cambia sostanzialmente solo il **type**

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

var canvas = document.getElementById('canvas');

var data = {
      type: 'bar',
      data: {
            labels: ['pippo', 'pluto', 'minnie'],  // keys
            datasets: [{
                  label: 'Titolo del grafico',    // solo per diagramma a barre
                  data: [12, 19, 32],             // values
                  backgroundColor: [
                        'rgba(255, 99, 132, 0.2)',
                        'rgba(54, 162, 235, 0.2)',
                        'rgba(255, 206, 86, 0.2)'
                  ],
                  borderColor: [
                        'rgba(255, 99, 132, 1)',
                        'rgba(54, 162, 235, 1)',
                        'rgba(255, 206, 86, 1)'
                        // Nel caso si voglia utilizzare sempre lo stesso colore,
                        // al posto del vettore si può assegnare una semplice stringa
                  ],
                  borderWidth: 1    /* default 2 */
            }]
      },
```

```
      options: {
            scales: {                        (*1)
                  y: {
                        suggestedMax:  40,
                        suggestedMin: -40,      // oppure
                        beginAtZero: true
                  }
            },
            /* responsive:true,              (*2)
               maintainAspectRatio:false,  */

      }

let chart = new Chart(canvas, data);    // bloccante;  canvas è un è JS
```

(*1) ***scales*** fa si che venga disegnato l'asse verticale fra i valori MIN e MAX indicati.
Utile soprattutto 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.

(*2) ***responsive:****true* abbinato a **maintainAspectRatio:**`false` dovrebbe fare si che il canvas diventi responsive (dopo averlo avvolto all'interno di un tag DIV dedicato avente width e height).
Non sembra però funzionare.

Tra una chiamata e l'altra, anche se Chart viene reistaziato, non è possibile riutilizzare il canvas precedente per un nuovo grafico per cui, prima di istanziare il Chart, occorre **necessariamente** 'ripulirlo' con un destroy

```
if(chart)   // != undefined
    chart.destroy();
```

Affinchè la variabile **chart** mantenga il proprio valore tra una chiamata e l'altra, deve **necessariamente** essere dichiarata globale.

I principali grafici supportati

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

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

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

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();
```

Upload di un file tramite SUBMIT

Si consideri la seguente form html:

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

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. Il method da utilizzare per l'upload può essere soltanto **POST** (o PUT o PATCH), in quanto il contenuto del file deve essere inviato all'interno del body della HTTP request e non può certo essere concatenato alla URL

In tutti i casi il controllo **<input type="file">** restituisce sempre un semplice vettore enumerativo di oggetti **File** accessibile da js attraverso l'attributo **txtFiles.files**

- In presenza dell'attributo **multiple,** il controllo **<input type="file">** restituisce all'interno dell'attributo **files** un **vettore enumerativo di oggetti File**.
- In assenza dell'attributo **multiple,** il controllo **<input type="file">** restituisce comunque in ogni caso un **vettore enumerativo di files**, contenente però un solo file.

Ogni **oggetto File** contiene tutte le informazioni relative al file selezionato. Tra i campi più significativi:

"name"	nome del file così come impostato sul client. Alcuni browser restituiscono soltanto il nome del file, altri (chrome) il path completo. Lato server occorre pertanto estrarre il nome 'pulito' dopo l'ultimo slash. Ad esempio in PHP si può utilizzare la funzione **basename**.
"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
"tmp_name"	è un campo aggiunto dal server PHP che rappresenta un nome random assegnato temporaneamente al file. Rappresenta in pratica il puntatore al file binario vero e proprio.

Nota:

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

L'oggetto FormData

L'attributo **enctype="multipart/form-data"** indica al browser di trasmettere i dati **NON** nel solito formato url-encoded, ma in un formato particolare denominato **FormData** che è un formato di trasmissione di tipo **key/value** in cui i values vengono trasferiti così come sono cioè come flussi binari senza subire nessun controllo né conversione.

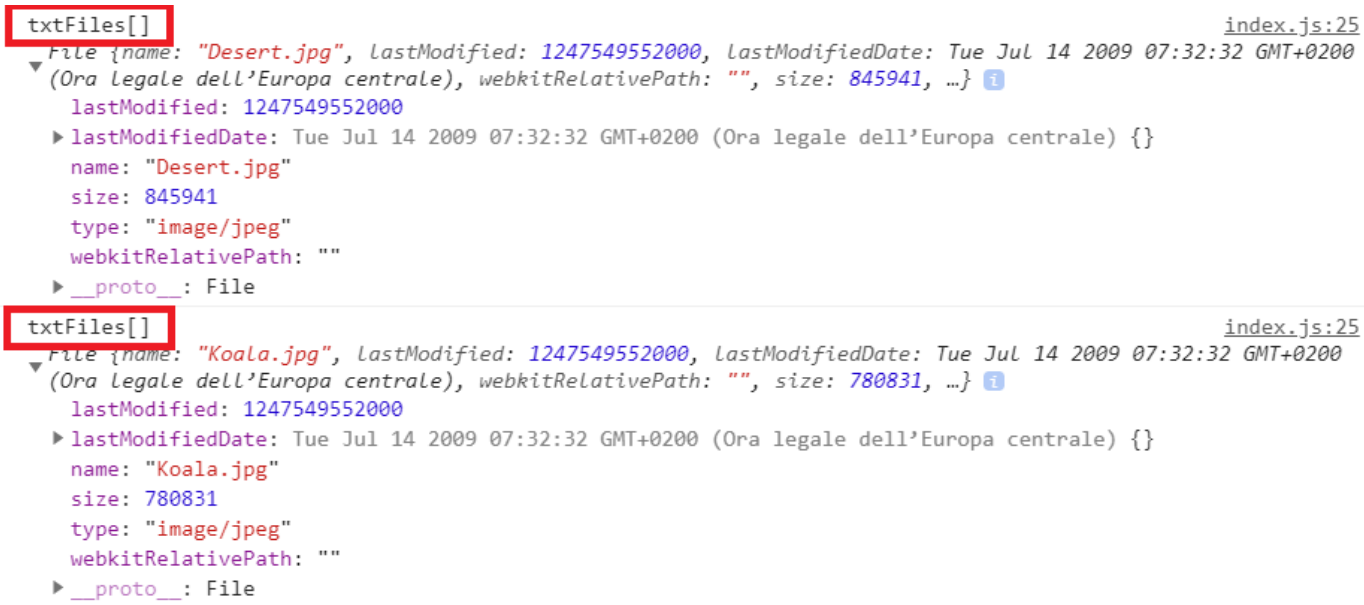
In pratica **FormData** è un **vettore ciclico,** cioè un vettore enumerativo di vettore enumerativi.
I vettori enumerativi interni sono tutti costituiti da due elementi:

- L'elemento in posizione 0 rappresenta la chiave dell'elemento
- L'elemento in posizione 1 rappresenta il contenuto dell'elemento

In caso di files multipli, ogni file sarà memorizzato in un vettore enumerativo interno e tutti i file presenteranno la stessa chiave (coincidente con il nome del controllo). Il contenuto sarà invece il file reale

Esempio di **formData** relativo all'upload di 2 files:

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



Note

- **`console.log(formData)`** ha poco senso e non produce nessun effetto. Per vedere il contenuto di una formData occorre dar ricorso al ciclo precedente
- Se all'interno della form, oltre al type=file, sono presenti altri controlli (textbox, radiobutton, etc) anch'essi verranno automaticamente aggiunti al **FormData** sempre in formato chiave-valore.
- La parola **`multipart`** sta ad indicare che alcuni campi sono di un tipo (stringa) mentre altri campi sono di altro tipo (object). I campi di tipo object sono sostanzialmente riservati all'upload dei files.

Upload di un file tramite Ajax

Innanzitutto occorre ricordare che java script non può accedere al file system locale, quindi non può leggere alcun file in modo diretto ma occorre necessariamente passare attraverso il tag html **`<input type='file'>`**.

Dal momento che si utilizza Ajax e non più il pulsante di submit, non è più necessario utilizzare una form. Anche il name dei controlli non serve più. Al suo posto si usa un ID. Per cui l'html si limiterà alle seguenti righe:

```
Scegli un file : <input type="file" id="txtFiles" multiple>
<input type="button" value="Upload" id="btnInvia">
```

In corrispondenza della scelta di uno o più files, il controllo `<input type="file">` restituisce all'interno dell'attributo **`files`**, **un vettore enumerativo di oggetti File** strutturati come indicato nella pagina precedente.

Nota : A differenza del **name**, l'**ID** non accetta sintatticamente l'utilizzo delle parentesi quadre (e non ha senso che le abbia). Dovrà essere il codice client a definire una **variabile vettoriale** da passare come parametro al server ed utilizzare questa variabile per aggiungere i vari object all'interno di FormData.

Codice javascript per il caricamento del FormData (files multipli)

Il codice client dovrà scorrere il vettore enumerativo restituito dal controllo `input[type=file]` e caricare ogni singolo File all'interno di un apposito oggetto FormData da passare poi come parametro alla funzione invia Richiesta()

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

let rq = inviaRichiestaMultipart("POST", "server/upload.php", formData);
```

Il metodo **`formData.append()`** consente di accodare più oggetti ad una stessa chiave denominata **`txtFiles[]`**. Se la variabile non esiste ancora, viene automaticamente creata in corrispondenza del primo append(). Il risultato finale sarà quello illustrato nello screen shot precedente.

Caricamento di un file singolo

Se il controllo **`input[type=file]`** non dispone dell'attributo multiple, il vettore enumerativo restituito da **`txtFiles.files`** conterrà un unico record, per cui diventa inutile l'esecuzione del ciclo ma sarà sufficiente scrivere:

```
_formData.append("txtFiles[]",  $('#txtFiles').prop('files') [0]);
```

- Se sul primo `txtFiles` **NON** si mettono le parentesi quadre dopo il name, la struttura inviata sarà restituita da PHP come un un singolo json. In tal caso i valori associati alle varie chiavi di **`txtFiles`** non saranno vettori enumerativi di valori ma semplici valori scalari
- Se sul primo `txtFiles` **SI** mettono le parentesi quadre dopo il name, la struttura inviata sarà restituita da PHP come un vettore enumerativo contenente un singolo json

In tutti i casi è sempre possibile accodare all'interno di formData altri parametri da uploadare insieme al file

```
formData.append('nome', $("#txtNome").val());
```

La funzione `inviaRichiestaMultipart`

La funzione **`inviaRichiestaMultipart`** provvede a trasmettere i parametri **`formData`** come semplice flusso binario senza eseguire alcun tipo di controllo / serializzazione. Dovrà pertanto contenere al suo interno le seguenti chiavi:

```
            data:formData
(a)         contentType:false,
(b)         processData:false,
```

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

(a) non vada ad aggiungere una Content-Type header , per cui il server interpreterà i dati direttamente in formato binario

(b) non effettui la serializzazione dei dati da inviare

Nota

Se nella 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

Evento onChange

Il controllo `<input type="'file">` dispone di un evento **`onChange()`** richiamato ogni volta che l'utente seleziona un nuovo file tramite la finestra sfoglia. Questo evento può essere utilizzato per avviare l'upload al server (in alternativa al pulsante esterno).