Ajax

Rev Digitale 4.5 del 04/03/2021

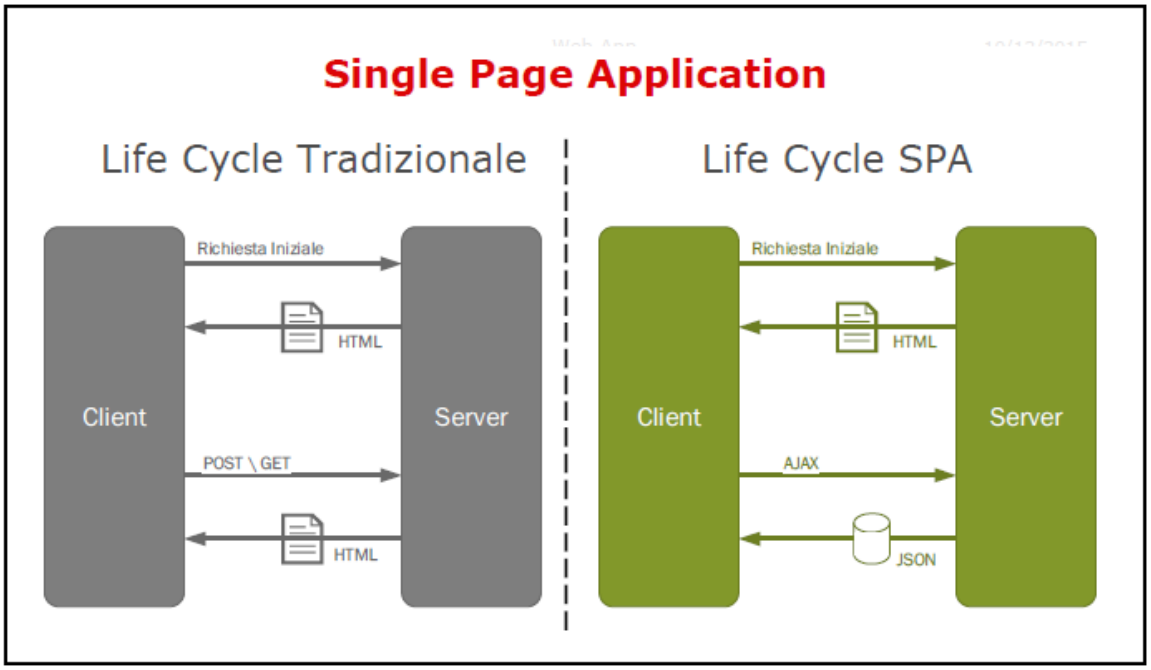
Introduzione ai Web Services

Un **web service** è un servizio web (detto anche **API** = Application Programming Interface) in esecuzione su un web server in grado di restituire un insieme di dati, tipicamente in formato json.

Il concetto di **web service** nasce negli anni 2004-2005 quando ci si rende conto che ricaricare ogni volta una intera pagina risulta troppo oneroso, mentre invece spesso è molto più veloce (e anche molto più semplice) ricaricare soltanto un insieme di dati che nel frattempo potrebbero essere cambiati e che verranno visualizzati in sostituzione dei dati precedenti.

La tecnologia utilizzata dei browser per **accedere ai dati di un Web Service** si chiama **AJAX** che significa Asynchronous JavaScript And XML

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.



In realtà il client di un web service non deve essere necessariamente 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 servizio **non** si utilizza più il pulsante di submit, ma un **normale button** che richiama una una apposita procedura java script che invia la richiesta, attende i dati ed aggiorna la visualizzazione.

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.

Ajax

Asynchronous JavaScript And **XML.**
E' la tecnologia utilizzata dai browser 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 sempre l'intera pagina** con conseguente inutile speco di banda e relativi rallentamenti.
Con Ajax è possibile richiedere soltanto i dati necessari. Le applicazioni risultano così molto più veloci, dato che la quantità di dati scambiati fra browser e server si riduce notevolmente.

L'oggetto base per l'invio di una richiesta Ajax ad un web server è un oggetto **javascript** denominato **`XMLHttpRequest`**, di cui il primo standard ufficiale è stato rilasciato dal World Wide Web Consortium (**W3C**) il 5 aprile 2006. Inizialmente i dati venivano trasmessi in formato **xml** (da cui la X finale di Ajax), oggi quasi completamente sostituito dal formato **json**.

A dispetto del nome, la tecnologia AJAX oggi:
- oltre a javascript, è utilizzabile con moltissimi altri ambienti di programmazione
- può scambiare dati in qualsiasi formato, principalmente XML, JSON ma anche CSV o semplice testo

Una caratteristica fondamentale di AJAX è che si tratta di una tecnologia **asincrona** nel senso che i dati sono richiesti al server e caricati in background senza interferire con il comportamento della pagina.
Cioè 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 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).

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 vengono passati i parametri. Può assumere i seguenti valori:

```
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
contentType:"text/plain; charset=utf-8"
```

Se si imposta **`contentType:`**`"application/x-www-form-urlencoded;"` i parametri sia get sia post possono anche essere passati in formato json (ma come object e non come stringa) e **vengono automaticamente convertiti** in formato urlencoded. E' però ammesso un unico oggetto (eventualmente costituito da più campi); **non sono ammessi vettori di oggetti o oggetti compositi.**

I parametri **POST** possono essere passati anche in un formato **`contentType:"application/json"`** In questo caso però i parametri DEVONO essere passati esclusivamente come **STRINGA** json e non come object. Bisogna in pratica eseguire un JSON.stringify prima di trasmetterli.

Lo scopo di **`content-type`** è quello di avvisare il server sul formato dei dati trasmessi. In realtà, in node.js, il metodo **`url.parse(request.url, true).query`** parsifica allo stesso modo sia i parametri url-encoding, sia i parametri scritti come stringa json, ignorando il parametro content-type.

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. I
- idem per xml.

L'attributo async

Impostando il valore **false** il metodo diventa sincrono, bloccando di fatto le istruzioni successive fino a quando non sono arrivati i dati

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

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 contente di “wrappare” una procedura asincrona all’interno di una nuova procedura all’apparenza sincrona.

Si suppoga 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 una parte di elaborazione e inietta il risultato ad una callback. L’ultima callback riceverà l’immagine finale rielaborata.

```
function elaboraImmagine(img){
      elaborazione annidata tramite “Pyramid of Doom”
      _lastLibrary.onEnd(err, finalImage){
            console.log(finalImage)
      }
}
```

Si supponga che questa `finalImage` debba essere inviata ad un server o comunque gestita dalla nostra applicazione. Il codice di gestione della `finalImage` dovrà pertanto essere scritto nell’apice destro della Pyramid of Doom. Il risultato è che 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. E’ qui che intervengono le Promise che, come detto all’inizio, consentono di “wrappare” una procedura asincorna all’interno di una nuova procedura soltanto **in apparenza** sincrona:

```
function elaboraImmagine(img){
    return new Promise(function(resolve, reject) {
        elaborazione annidata tramite “Pyramid of Doom”
        _lastLibrary.onEnd(err, finalImage){
            if(err)
                reject(err)
            else
                resolve(finalImage);
    })
}
```

L'oggetto **Promise** riceve come parametro una funzione di callback che verrà eseguita **in modo asincrono** all'interno della Promise (nel nostro caso la funzione di elaborazione dell'immagine) alla quale funzione di callback vengono automaticamente iniettati i puntatori a due metodi interni della Promise stessa, i metodi **`resolve e reject`**.

Il codice utente interno alla promise dovrà *soltanto* preoccuparsi di richiamare :

- Il metodo **`resolve()`** in caso di terminazione corretta, passandogli come parametro il risultato dell'elaborazione
- Il metodo **`reject()`** in caso di errore

Il **`return`** iniziale fa sì che venga ritornato al chiamante un puntatore alla Promise.
L'oggetto Promise, oltre ai metodi **`resolve e reject`**, dispone anche di due eventi visibili esternamente che sono :

- **`.then`** richiamato automaticamente in corrispondenza del **`resolve`**
- **`.catch`** richiamato automaticamente in corrispondenza del **`reject`**

Per cui alla fine il codice utente si riduce alle tre semplicissime righe seguenti, in cui tutta la parte di elaborazione dell'immagine è spostata dentro `elaboraImmagine()` e non crea 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)
})
```

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

In jQuery per creare una Deferred/Promise si utilizza Il metodo statico **`$.Deferred()`** che è l'analogo del `new Promise()` di prima. Questo metodo statico:

- istanzia un nuovo oggetto **`Deferred`**
- richiama la funzione di callback ricevuta come parametro
- inietta alla funzione un puntatore `d` all'oggetto deferred medesimo, il quale dispone dei due metodi **`resolve`** e **`reject`** esattamente come prima.
- infine restituisce al chiamante il medesimo puntatore **`d`** **sotto forma di** **`Promise`** (cioè prima di restituire l'oggetto esegue un cast da Deferred a Promise)

Esempio

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

potrebbe essere riscritto nel modo seguente:

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

Esattamente come prima il codice della callback viene "portato fuori" rispetto alla funzione iniziale.

- il metodo **`d.resolve()`** all'interno **`$.Deferred`** deve essere invocato in caso di successo.
- il metodo **`d.reject()`** all'interno **`$.Deferred`** deve essere invocato in caso di errore
- il metodo **`d.notify()`** all'interno **`$.Deferred`** può essere invocato in corso di esecuzione

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()`**

Differenza tra Promise e Deferred

L'oggetto **`Promise`** è sostanzialmente un oggetto **`Deferred`** di tipo read only, cioè che dispone soltanto degli eventi (**`done, fail, progress, then, catch, always`**) ma NON dei metodi di risoluzione o rigetto del Deferred (`resolve, reject, notify`)

Gli eventi **promise.done()** e **promise.fail()** sono eventi specifici disponibili soltanto in jQuery.
Gli eventi **promise.then(), promise.always()** e **promise.catch()** sono eventi generali definiti all'interno dell'oggetto javascript **Promise** .

L'evento `promise.then()`

accetta in realtà tre funzioni di callback:

- la prima viene richiamato in corrispondenza del metodo d.resolve(),
- la seconda viene richiamato in corrispondenza del metodo d.reject()
- la terza viene richiamato in corrispondenza del metodo d.notify()

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

Se usato con una sola callback diventa quasi equivalente a **`promise.done()`.**

L'evento `promise.always()`

accetta due funzioni di callback:

- la prima viene richiamata in corrispondenza del metodo d.resolve(),
- la seconda viene richiamata in corrispondenza del metodo d.reject()

L'evento `promise.catch(fn)`

E' quasi equivalente a **`promise.fail`**`(fn)`, ed è equivalente a `promise.then(null, fn)` 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.
- **`promise.fail(fn)`** "termina" l'esecuzione e resituisce la stessa promise, per cui eventuali `then(fn)` successivi NON verranno più eseguiti.

La stessa differenza esiste tra .done() e .then(). In sostanza la differenza è solo che **`.then()`** e **`.catch()`** possono anche essere usati in forma concatenata, **`.done()`** e **`.fail()`** no.

Note Aggiuntive

1) Le funzioni di callback di done() e fail() possono ricevere uno o più parametri di qualunque tipo che dovranno essere impostati 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).

Il metodo statico $.when()

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.

Nel caso in cui una delle resolve() restituisca più parametri, questi verranno passati da when() alla promessa finale sotto forma di vettore

```
var promise1 = $.Deferred(function(d) {d.resolve("ris promessa 1") })
var promise2 = $.Deferred(function(d) {d.resolve("ris promessa 2", "a") })

$.when( promise1, promise2).done(function(risp1, risp2) {
        console.log(risp1  +  ","  +  risp2[0] )
});
```

Esempio di utilizzo del metodo $.Deferred

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');
});
```

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

Il metodo .promise() applicato agli elementi della pagina

Il metodo `.promise()` può essere applicato ad un qualunque oggetto della pagina html. Nel momento in cui viene applicato restituisce un oggetto `promise` contenente tutte le procedure di callback in quel momento accodate su quell'oggetto. Se la coda è vuota genera immediatamente l'evento `done`.
Se la coda non è vuota genera l'evento `done` nel momento in cui la coda si svuota:

```
$("button").on("click", function() {
    $( "p" ).append( "Started..." );
    $("div").each(function( i, ref ) {
        $(this).fadeIn().fadeOut( 1000 * (i+1) );
    });
    $("div").promise().done(function() {
        $( "p" ).append( " Finished! " );
    });
```

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**

Esempio 1

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);
    });
```

Esempio 2: Riscrittura della funzione inviaRichiesta tramite utilizzo delle Deferred / Promise

```
function inviaRichiesta(method, url, parameters={}) {
    var promise = $.ajax({
        url: url,         //default: currentPage
        type: method.toUpperCase(),
        data: parameters,
        contentType: "application/x-www-form-urlencoded; charset=UTF-8",
        dataType: "json",
        async : true,     // default
        timeout : 5000
    });
    return promise;
}
```

Il client potrà utilizzare il seguente codice:

```
var request = inviaRichiesta("get", "/elencoFiliali.php", {"codBanca":7});
request.done(function(data){
    alert(JSON.stringify(data));
});
request.fail(errore);
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 presenta **due vantaggi** rispetto a quello 'tradizionale':

1. E' possibile associare più funzioni di callback in risposta alla medesima richiesta
   `request.done(function1).done(function2);`
2. 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.

Attenzione che le chiavi di primo livello possono avere come contenuto soltanto vettori enumerativi o associativi (json). Non sono ammessi valori scalari come interi, booleani e nemmeno stringhe.

Servizi esposti

In realtà json-server è un vero e proprio crud server in grado di esporre tutti i tipici metodi crud.
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"
}]
```

json-server è in grado di servire tutte le seguenti richieste, andando automaticamente ad aggiornare i dati sul file:

```
GET     /persons        elenco di tutte le persone.  Restituisce un vettore di record
GET     /persons/2      restituisce il singolo record avente id=2
GET     /persons?nome=alfio&classe=4B.  Restituisce un vettore di record

POST    /persons        aggiunge in coda a persons il record passato come parametro
                        {"id":4, "nome":"pippo", "genere":"m", "classe":"2A"}
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 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 entreranno automaticamente a far parte del record corrente.

- La seconda forma della chiamata **GET** (quella in cui viene passato il solo ID) restituisce un record singolo. La prima e la terza restituiscono invece un vettore di record.
- La terza forma della chiamata **GET** (quella con i parametri dopo il punto interrogativo) può anche essere scritta passando i parametri in formato JSON nel modo seguente:

  ```
  inviaRichiesta("GET", URL + "/persons", {"nome":"alfio", "classe"="4B"})
  ```

- Nel caso delle chiamate **POST** 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**, incrementato di 1. Non è possibile postare più record nell'ambito di una stessa chiamata.

L'URI "data:" e la codifica base64

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` che definisce un risorsa web (`www.google.com`) ed un 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:

```
<img src="
  AAAAHElEQVQI12P4//8/w38GIAXDIBKE0DHxgljNBAAO9TXL0Y4OHwAAAABJRU5ErkJggg…"/>
<iframe src="data:application/pdf;base64,JVBERi0xLjUNCiW1tbW1DQ/oxIDAgb2...">
<object data="data:application/pdf;base64,JVBERi0xLjUNCiW1tbW1DQ/oxIDAgb2...">
```

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

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

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

`data:[<MIME-type>][;charset="<encoding>"][;base64],<data>`

- Il primo parametro rappresenta l'eventuale **media/type** della risorsa terminato da un punto e virgola; ad esempio `image/png` oppure `application/pdf`
- Il secondo parametro rappresenta l'eventuale **formato** di memorizzazione testuale della risorsa, (default "`utf-8`"). Nel caso delle immagini, può essere omesso
- Il terzo parametro rappresenta la codifica utilizzata nella conversione binario-testo che n pratica è SEMPRE base64. Dopo il terzo parametro c'è una virgola che è obbligatoria e che serve da **separatore** tra l'intestazione ed il corpo dati vero e proprio.
- **`<data>`** rappresenta la codifica base64 dell'immagine
- La forma minima di una data URI risulta pertanto essere la seguente **`data:,<data>`**

Altri tipi di URI

```
String uri = "tel:3337684747";
String uri = "sms:3337684747";
String uri = "mailto:nome.cognome@vallauri.edu";
```

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. Lo scopo è quello di poter trasmettere i dati senza perdita o modifica del contenuto. La codifica Base64 opera in questo modo:

- suddivide la sequenza binaria in gruppi di 6 bit. Le combinazioni possibili su 6 bit sono 64
- memorizza ciascuna delle 64 combinazioni tramite un preciso carattere ASCII (ad esempio 000000 -> 'A', 000001 -> 'B', 000010 -> 'C', 000011 -> 'D',
- I caratteri ASCII utilizzati per la codifica sono le 26 lettere minuscole, le 26 lettere maiuscole, i 10 caratteri numerici e i caratteri speciali + e /
- Il carattere corrispondente a ciascun gruppo di 6 bit viene poi salvato con il suo normale codice ascii, cioè 'a' -> 01100001, 'A' -> 01000001, '0' -> 00110000, '+' -> 00101011
- Per cui ogni 24 bit originali (3 bytes) crea 4 gruppi da 6 bit che poi con la codifica ascii diventano 4 bytes. Cioè crea in pratica 4 cifre base64 ogni 3 bytes. **Un dato codificato in base64 è 1.333 volte più grande rispetto al datapiece originale**

Note:

- La codifica Base64 è nata nei sistemi di posta elettronica che inizialmente gestivano solo dati testuali. Base64 è stata introdotta nei sistemi di posta 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

In sintesi base64 è una tecnica per aumentare la sicurezza della trasmissione di una informazione binaria a scapito di un incremento delle dimensioni.

Codifica e invio delle immagini in formato base64

Talvolta, per accentrare il salvataggio di tutti i dati all'interno del database, si decide di salvare nel DB anche il contenuto delle immagini (o di un pdf) che, a tale scopo, vengono convertite in stringhe base64 e vengono quindi trasmesse al server come normali parametri post all'interno della post request.

Per convertire una immagine in formato base64 si può utilizzare il seguente codice:

HTML

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

JavaScript

```
function elabora() {
    var filename = $("#txtFile").prop("files")[0];
    var reader = new FileReader();
    reader.readAsDataURL(filename);
    reader.onloadend = function() {
       console.log('RESULT', reader.result)
    }
}
```

Cioè all'interno di **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**,iVBORw0KGg+oAAAANSUhEUgAABsYAAA/R4C..... "

"data:application/pdf;base64,JVBERi0xLjUNCiW1tbW1DQ/oxIDAgb2..... "

L'attributo<a> download

In HTML5 al tag **<a>** è stato aggiunto un attributo **download** che, anziché visualizzare in una nuova scheda il contenuto della risorsa indicata dell'attributo **href,** consente di aprire una **Finestra di Dialogo** del browser la quale, tramite il pulsante Sfoglia, consente di scegliere dove salvare la risorsa puntata da **href** all'interno del computer client.

```
<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 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="ombrelloni.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", 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 "ombrelloni.json".

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 può 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 che restituisce una URL del canvas corrente come immagine png codificata in formato base64 (**`data:image/png;base64`**)

Come salvare un'immagine binaria su disco

Se si dispone della URL (attributo **`.src`**) si può utilizzare l'esempio iniziale.
Se invece l'immagine è disponibile soltanto in formato binario, queste non dispongono di un metodo che ne restituisca la url, per cui o si passa attraverso un Blob oppure, più frequentemente, si passa attraverso un Canvas.

```
$("<a>").prop({"download":"newImage.jpg", "href":"#"}).text("salva immagine")
 .appendTo(wrapper).on("click", function(){
      const img = $('img')[0];
      const dataUrl = _getDataUrl(img);  // img è un puntatore js
      $(this).prop("href", dataUrl);
})

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

Upload di un file tramite SUBMIT

Si consideri la seguente form html:

```
<form action="upload.php" method="post"  enctype="multipart/form-data">
    Select file to upload:
    <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), 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 vettore enumerativo di oggetti **File** contenenti ognuno tutte le principali informazioni relative al file selezionato.

- 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.

Il suddetto vettore enumerativo è accessibile da js attraverso l'attributo **txtFiles.files**.
Tra i campi più significativi dei singoli **File** ci sono:

"name"	nome del file così come impostato sul client. Alcuni browser restituscono 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 dati 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 il vettore enumerativo precedente viene convertito in un vettore associativo molto particolare (**ciclico**) in cui la chiave corrisponde al nome del controllo e (in caso di multiple) sarà la stessa per tutti i files, mentre il value conterrà i vari oggetti **File** relativi ai files da uploadare.

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

Lo screen shot precedente è stato ottenuto mediante il seguente codice:

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

Se all'interno della form 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 neanche senso che le abbia). Dovrà essere il codice client a definire eventualmente una **variabile vettoriale** da passare come parametro al server ed utilizzare questa variabile per aggiungere tutti i vari object all'interno dell 'oggetto 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 come semplice flusso binario senza eseguire alcun tipo di controllo / serializzazione. Dovrà pertanto contenere al suo interno le seguenti chiavi:

```
(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 si può utilizzare un pulsante esterno).

Concetto di same-domain-policy

Per quanto riguarda le richieste inviate da un browser, la tecnologia normalmente utilizzata per inviare la richiesta di una risorsa ad un server rest è **AJAX**.

Tutti i browser attuali implementano però delle restrizioni dette **`"same-domain-policy"`** introdotte fin all'epoca di Netscape Navigator 2.0 (1995) ed attualizzate fino ai browser di recente sviluppo, restrizioni in base alle quali uno **script** di una pagina web può accedere soltanto alle risorse appartenenti alla stessa origine della pagina nella quale si trova lo script.

Questo significa che un qualsiasi script non può inviare richieste Ajax verso risorse (HTML, XML, JSON, testo) **che si trovino su un server diverso da quello di partenza**.

Cosa si intende per "stessa origine"?

L'origine è definita da tre "parametri": **dominio**, **porta** e **protocollo** e solo se tutte tre queste **URI**(Uniform Resource Identifier) sono comuni, si può dire che le risorse appartengono alla stessa origine

Esempio 1: uno script scaricato dal dominio google.com potrà accedere solamente a file che rispondano a URL che appartengono a google.com.

Esempio 2: **una pagina HTML aperta localmente su un PC non può inviare richieste ad un web server nemmeno se questo si trova in esecuzione sullo stesso PC, in quanto il server è visto come appartenente ad un dominio diverso**.

Esempio 3: una APP per smartphone non può inviare richieste AJAX a nessun web server, a meno che non si definiscano le opportune policy.

Lo scopo è quello di impedire a script scaricati dalla rete di accedere a risorse che si trovano su server diversi rispetto a quello iniziale che ha inviato lo script. Quella che può sembrare una forte limitazione nello sviluppo di applicazioni web rappresenta invece un fattore importante per la sicurezza dei web host. Basta infatti pensare a quante possibilità esisterebbero per far eseguire al nostro browser codice maligno scaricato da chissà quale server semplicemente aprendo un sito che a prima vista può sembrare innocuo.

Soluzioni

Le restrizioni precedenti violano ovviamente il concetto di web service che deve essere pubblicamente fruibile da tutti. Le tecniche che permettono l'invio di richieste AJAX cross-domain sono sostanzialmente due:

- **JSON-P**(JSON with Padding)
- **CORS**(Cross-Origin Resource Sharing)

JSON-P è decisamente più semplice e consente di inserire dinamicamente un tag `<script>` all'interno di una pagina HTML da cui viene richiamata una funzione che conterrà i dati richiesti. Presenta però dei limiti come ad esempio il supporto al solo metodo GET e la facile vulnerabilità.

CORS è più recente, standardizzato dal W3C, supportato ormai da tutti i browser ed è reputato migliore e più sicuro rispetto a JSON-P.
E' basato sull'aggiunta da parte del client di una intestazione HTTP in cui il client dichiara il proprio dominio. Il server verifica l'affidabilità del dominio client e aggiunge a sua volta altre intestazioni HTTP di accettazione della richiesta.

JSON-P

JSON with Padding (con imbottitura)
Scopo: risolvere il problema delle restrizioni dovute alle "**same-domain-policy**"
Il meccanismo di funzionamento di **JSON-P** è basato sul fatto che i browser non attuano le "**same-domain-policy**" sul tag **<script>**, ne senso che il tag script può richiamare senza problemi uno script js eventualmente anche esterno al dominio corrente. La strategia è quella di modificare il **DOM** della pagina aggiungendo un nuovo tag **<script>** nella sezione di `<head>` in modo da poter accedere a documenti JavaScript remoti. Se lo script ricevuto contiene un insieme di istruzioni dirette, queste verranno eseguite automaticamente dall'interprete JavaScript appena ricevute dal server. Nel caso di jSON-P non interessa però tanto l'accesso ad un file js, quanto l'accesso ad una stringa jSON.

Il principale limite di questa tecnica è che, essendo basata sul tag **<script>** può inviare soltanto richieste **GET**.

Si supponga di inserire nella HEAD di una pagina HTML il seguente script

```
<script type="application/javascript" src="http://example.com/Users/1234">
</script>
```

il quale richiama un servizio web passandogli come parametro un codice utente, ad es 1234.

Su supponga che il servizio web risponda con il seguente oggetto JSON serializzato:

```
{
    "id": 1234,
    "name": "Foo",
    "tank": 7          }
```

Il browser, in corrispondenza dello script, scaricherà il file indicato, valuterà il contenuto e, non trovando istruzioni js ma un blocco json, genererà un errore continuando poi nell'interpretazione della pagina. Anche interpretando il blocco come un Literal Object questo risulterebbe comunque inaccessibile al programmatore perché non assegnato a nessuna variabile e quindi non elaborabile.

JSONP utilizza esattamente questo approccio, con l'accorgimento che il server deve wrappare i dati JSON come parametri all'interno della chiamata ad una funzione definita all'interno della pagina client.
In questo modo la funzione verrà richiamata ed eseguita al ricevimento di dati.
Ritornando all'esempio precedente, immaginiamo che il server, invece del semplice oggetto JSON, restituisca la seguente porzione di codice :

```
displayBlock({
    "id": 1234,
    "name": "Foo",
    "rank": 7
});
```

In questo caso appena ricevuto lo script, l'interprete eseguirà la funzione **displayBlock** definita all'interno di un'altra sezione di script della pagina stessa. **displayBlock** rappresenta il **padding** aggiunto al server rispetto ai dai JSON (da cui il nome JSON-P). Questa funzione riceve come parametri i dati JSON. Si immagini di avere all'interno della pagina la funzione `displayBlock` così definita:

```
function displayBlok(book) {
      alert(book.name, book.rank);  }
```

Come è possibile però istruire il server riguardo al nome della funzione da invocare? La soluzione alquanto banale è quello di passare al server questo nome come parametro GET all'interno della chiamata. Il server, una volta ottenuti i dati richiesti, creerà un semplice stringa contenente il nome della funzione ricevuto come parametro, concatenando all'interno i dati serializzati in formato JSON.

Esempio Completo : pagina client

```
<html>
<head>
<script>
   function jsonp_request(url) {
      var head = document.getElementsByTagName("head")[0];
      var script = document.createElement("SCRIPT");
      script.type = "text/javascript";
      script.src = url + "?callback=update_table";
      head.appendChild(script);
   }

   function update_table(data) {
      var container = document.getElementsByTagName("div")[1];
      var html = "";
      for(var i = 0; i<data.length; i++) {
          html += "<span>" + data[i].id   + "</span>";
          html += "<span>" + data[i].name + "</span>";
          html += "<span>" + data[i].rank + "</span>";
      }
      container.innerHTML += html;
   }

   window.onload = function() {
        jsonp_request("http://externSite/pag.php");
   }
</script>


<style>
    div { width: 300px; }
    div span { display:block; float:left; width:100px; }
    div span.th { color: blue; }
</style>
</head>

<body>
<div>
    <span class="th">Id</span>
    <span class="th">Name</span>
    <span class="th">Rank</span>
</div>

<div>
</div>
```

Il markup della pagina è abbastanza semplice: sono presenti due div con larghezza fissa; il primo serve come intestazione per la pseudo-tabella mentre il secondo sarà il container delle informazioni scaricate da remoto. Grazie a poche regole CSS è possibile simulare una tabella utilizzando solamente e . Focalizzando l'attenzione su JavaScript notiamo due funzioni:

- **jsonp_request** permette di aggiungere un tag <script> all'interno della testata del file HTML. Aspetto importante di questa funzione è l'aggiunta all'url ricevuto di un parametro GET di nome **callback** che contiene una stringa che rappresenta la funzione client che dovrà essere eseguita in corrispondenza del ricevimento dei dati. Questo parametro lo ritroveremo nella parte server-side.
- **update_table** è la funzione che viene passata come parametro e che visualizza i dati ricevuti dal server remoto.

All'evento **window.onload** viene assegnata la funzione **jsonp_request** che provvede a creare lo script per l'invio della richiesta al server.

Esempio di Server PHP

Il server PHP potrebbe essere il seguente:

```
<!--?php
    $books = array();
    $books[] = array("id" => "b1", "title" => "title1", "author" => "author1");
    $books[] = array("id" => "b2", "title" => "title2", "author" => "author2");
    echo $_GET['callback'] . "(" . json_encode($books) . ");" ;
?>
```

Una volta riempito il vettore $books, esso viene codificato in JSON tramite la funzione json_encode e la stringa ritornata viene costruita anteponendo il valore ricevuto mediante il parametro callback seguito dalle corrette parentesi in modo da generare una chiamata alla funzione JavaScript.

In corrispondenza della chiamata **pag.php?callback=update_table** l'output generato sarebbe :

```
update_table({ .... } );
```

In questo modo la funzione di callback verrà invocata automaticamente dal client appena scaricato lo script dal server.

Conclusioni:

Nonostante implementare un client JSONP sia abbastanza semplice, è sempre meglio, una volta capita la tecnica, utilizzare una libreria che permetta una migliore suddivisione dei compiti all'interno dell'applicazione e una maggiore sicurezza e configurabilità.

Ultimamente sono molti i framework che hanno aggiunto al loro set di funzioni anche alcuni componenti in grado di effettuare richieste JSONP. Ad esempio nelle ultime versioni di JQuery la funzione $.getJSON è stata estesa ed è ora disponibile anche per richieste JSONP.

D'altro lato molti web services disponibili su internet (es Yahoo, Flickr) possono ritornare dati in formato JSON-P. Occorre però verificare singolarmente l'esatta sintassi utilizzata. Il nome utilizzata per passare il parametro al server è solitamente **callback=** oppure **jsonp=**

Esempio di richiesta AJAX cross domain eseguita tramite jQuery

Di seguito è riportato il codice di una semplice richiesta AJAX cross-domain eseguita tramite JQuery.
In coda ai parametri della **url** viene aggiunto un parametro **callback=?**. Questo parametro contiene un segnaposto che il framework JQuery sostituirà con il **nome** di una funzione interna definita a runtime (e distrutta una volta completata la richiesta). Questo nome viene **generato** automaticamente ed univocamente in base alla data e all'ora in cui viene effettuata la request.

success contiene la funzione che verrà eseguita all'arrivo della risposta mentre **result** rappresenta il contenuto della risposta del server.
Con questo approccio JQuery "nasconde" l'intera implementazione di JSONP all'utilizzatore.

Client-side

```
function richiestaAjax(){
   $.ajax({
         url: INDIRIZZO_SERVER+"/cercaDati?callback=?",
         type:"GET",
         crossDomain:true,
         success: function(result){...}
   });
}
```

Server-side

Di seguito è riportato il codice Node.js di gestione della richiesta alla risorsa **cercaDati** tramite il metodo JSONP. Oltre alla normale gestione dell'invio della risposta tramite il Dispatcher, i dati vengono racchiusi all'interno di una funzione con lo stesso nome di quella passata come parametro.

```
dispatcher.addListener("get","/cercaDati",function(request, response){
   ...
   var nomeFunzione = request.get.callback
   var jsonString = JSON.stringify(datiOttenutiDaOperazioniPrecedenti)
   response.writeHead(200, HTTPheaderApplicationJavascript);
   response.end(nomeFunzione + "("+jsonString+")");
});
```

Utilizzo del metodo statico `$.getJSON()`

Una alternativa ancora più semplice è rappresentata dal metodo statico **`$.getJSON`**

L'esempio di utilizzo proposto nella documentazione ufficiale è simile a questo:

```
var url = http://api.flickr.com/services/feeds/photos_public.gne?
                          tags=cat&tagmode=any&format=json&callback=?"
$.getJSON(url, function(data){
          showMyData(data);
});
```

In assenza del parametro **`callback=?"`** verrà eseguita una normale chiamata AJAX. Con il parametro **`callback=?"`** viene invece eseguita una chiamata JSON-P. In automatico il motore di JQuery sostituirà il punto interrogativo con una funzione interna definita a runtime che invocherà la funzione passata come parametro a **`$.getJSON`** (esattamente come per il metodo precedente).