Ajax

Rev Digitale 3.3 del 03/12/2020

Introduzione ai Web Services 2
Elementi base di **Ajax** 4
L'oggetto XMLHttpRequest 4
L'oggetto C# HttpWebRequest 6

Invio di una richiesta Ajax tramite jQuery 7
Utilizzo degli oggetti Deferred / Promise 9
Scambio dati tramite **JSON** 11
Scambio dati tramite **XML** 12

Upload di un file tramite submit 13
Upload di un file tramite ajax 14

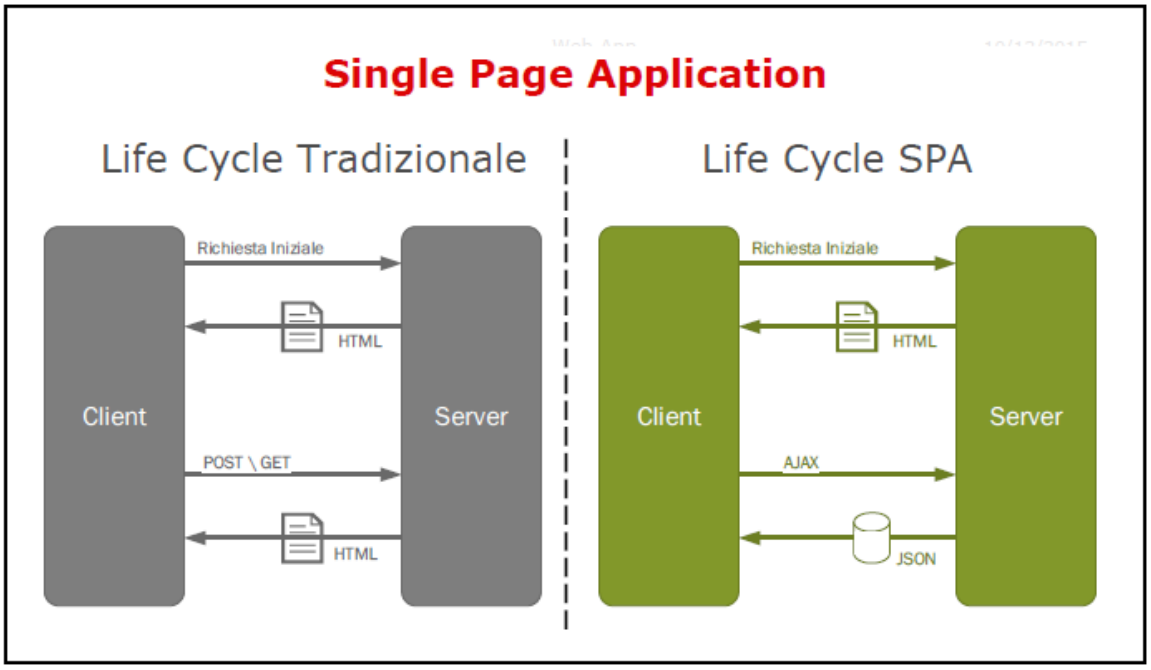
Concetto di same-domain-policy 17
jSON-P 18
Esempio di richiesta AJAX cross domain eseguita tramite jQuery 21

Introduzione ai Web Services

Nella programmazione web tradizionale (la cosiddetta programmazione **Web Form**) un client, attraverso un pulsante di submit, richiede al web server una nuova pagina.html che sostituirà completamente la pagina attuale all'interno della finestra del browser. Questo comporta due tipi di problemi:

- E' inutile ricaricare l'intera pagina quando probabilmente sarebbe sufficiente l'aggiornamento di alcuni dati
- Se la pagina da ricaricare è la pagina stessa, la pagina deve necessariamente mantenere lo stato. Cioè se nella pagina ci sono alcuni listbox o radio button o check box con delle voci selezionati, queste selezioni devono essere mantenute anche dopo il nuovo caricamento della pagina, e questa è una operazione abbastanza complicata da gestire, in quanto alcune cose devono essere fatte lato server altre lato client

Negli anni 2004-2005,nasce il concetto di **Web Service:** cioè un web server, invece di restituire soltanto pagine html può restituire, tramite web service, anche uno **stream di dati** che dovranno essere elaborati dal client e visualizzati in sostituzione dei dati precedenti. Questo consente una notevole velocizzazione nell'aggiornamento dinamico delle pagine. 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. La tecnologia normalmente utilizzata dei browser per **accedere ai dati di un Web Services** si chiama **AJAX** che significa Asynchronous JavaScript And XML



In realtà il client di un web service non deve essere necessariamente un browser, ma può essere anche

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

Il browser per accedere ad un **servizio** utilizza l'oggetto **`XMLHttpRequest.`**
Applicazioni desktop e smartphone, per accedere ai Web Services, utilizzano oggetti analoghi disponibili ormai in tutti gli ambienti di sviluppo (**`HttpWebRequest`** nel caso di **C#**, **`DefaultHttpClient`** nel caso di **Android**).

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 mette in risalto il concetto di **`servizio`**.
REST propone 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.**
Tecnologia nata in ambito web per richiedere dati ad un **servizio web** in esecuzione su 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.`** Il primo standard ufficiale dell'oggetto **XMLHttpRequest** è stato rilasciato dal World Wide Web Consortium (**W3C**) il 5 aprile 2006.
Inizialmente i dati venivano trasmessi dal server al client 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** o testuale puro

AJAX è **`asincrono`** nel senso che i dati sono richiesti al server e caricati in background senza interferire con il comportamento della pagina esistente. Cioè l'utente può tranquillamente continuare ad interagire con l'interfaccia grafica anche mentre l'applicazione è in attesa di nuovi dati dal server.
Concetto contrapposto al modello di **comunicazione sincrona** tipica delle applicazioni web tradizionali (le cosiddette applicazioni Web Form) in cui, in corrispondenza del submit, viene inviata una richiesta al web-server rimanendo poi in attesa della nuova pagina.

Esempio di 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. Nel caso classico, se il nome che abbiamo scelto fosse già esistente, dovremmo compilare prima tutto il modulo ed accorgerci solo dopo aver atteso il caricamento della pagina di conferma che il nome è già esistente. Viceversa con AJAX :

- può essere introdotto un controllo sull'evento onChange o OnKeyUp della Text Box
- in corrispondenza dell'evento, 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 della pagina

```
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 di una risposta testuale

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.

Esempio

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("Errore nella risposta \n"+risposta);
    }
}
```

L’oggetto C# HttpWebRequest

```
HttpWebRequest request =
 (HttpWebRequest)WebRequest.Create("http://www.blogger.com/feeds/12345/posts/default")
request.Method = "GET";
HttpWebResponse response = (HttpWebResponse)request.GetResponse();

if (response.StatusCode == HttpStatusCode.OK) {
    Stream responseStream = response.GetResponseStream();
    visualizza(responseStream);
}
response.Close()
```

Invio di una richiesta Ajax tramite jQuery

Il metodo statico **$.ajax()** (non presente nella libreria **slim**) rappresenta un 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",        // default: currentPage
   type: "GET",
   data: { "nome": "pippo" },
   contentType: "application/x-www-form-urlencoded; charset=UTF-8",
   dataType: "json",
   async : true,      // default
   timeout : 5000,
   success: function(data, [textStatus], [jqXHR]) {
         alert(JSON.stringify(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",
});
```

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.

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 riferimento all'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

Utilizzo delle Deferred / Promise

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 request = $.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 request;
}
```

Il client potrà utilizzare il seguente codice:

```
var request = inviaRichiesta("get", "elencoFiliai.php", "codBanca=7");
request.done(function(data){
    alert(JSON.stringify(data));
});
request.fail(error);
function error (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){  });
```

Scambio dati con un web 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 (jQuery)

```
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 tramite 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);
        }
    }
  }
```

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.

Nel caso di file singolo il controllo **<input type="file">** restituisce un oggetto **File** contenente tutte le principali informazioni relative al file selezionato (**senza però il contenuto del file**).
Questo oggetto **File** è accessibile da js attraverso l'attributo **txtFiles.files**.
Tra i campi più significativi dell'oggetto **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.

Note:

In presenza dell'attributo **multiple**, il controllo **<input type="file">** restituisce all'interno dell'attributo **files** un **vettore enumerativo di oggetti File** ognuno contenente i campi precedenti.

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.

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

Eventi

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).

Codifica e invio dell'immagine 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" onchange="encodeImageFileAsURL(this)" />
```

JavaScript

```
function encodeImageFileAsURL(txtFile) {
    var file = txtFile.files[0];
    var reader = new FileReader();
    reader.onloadend = function() {
       console.log('RESULT', reader.result)
    }
    reader.readAsDataURL(file);
}
```

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..... ”

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).