



Università degli Studi di Bologna
Facoltà di Ingegneria

Tecnologie Web L-A
A.A. 2009 – 2010

Esercitazione 8
AJAX

Tutor:

Ing. Pasini Samuele

samuele.pasini@unibo.it

INTRODUZIONE

AJAX - Asynchronous Javascript and XML

- Richieste al server effettuate via Javascript in modo asincrono:
 - l'utente non ottiene una nuova pagina, ma solo modifiche a quella corrente
 - non è necessario aspettare la response per continuare a interagire
 - non è necessario aspettare la response per eseguire altro codice AJAX
- Interfacciamento client-server mediante lo scambio di:
 - testo semplice (frammenti di pagina e/o singole informazioni)
 - documenti XML (informazioni strutturate e complesse)
- Non una nuova tecnologia, ma un **nuovo modo di utilizzare tecnologie esistenti** per sviluppare **rich internet application**
 - Gmail, Google Maps, Google Suggest, FaceBook, ...
- Alternative tecnologiche
 - Adobe-Macromedia Flash
 - Java applet
 - uso del tag *iframe* (tristezza → si modifica l'attributo *src* del frame per simulare un aggiornamento parziale dei contenuti)

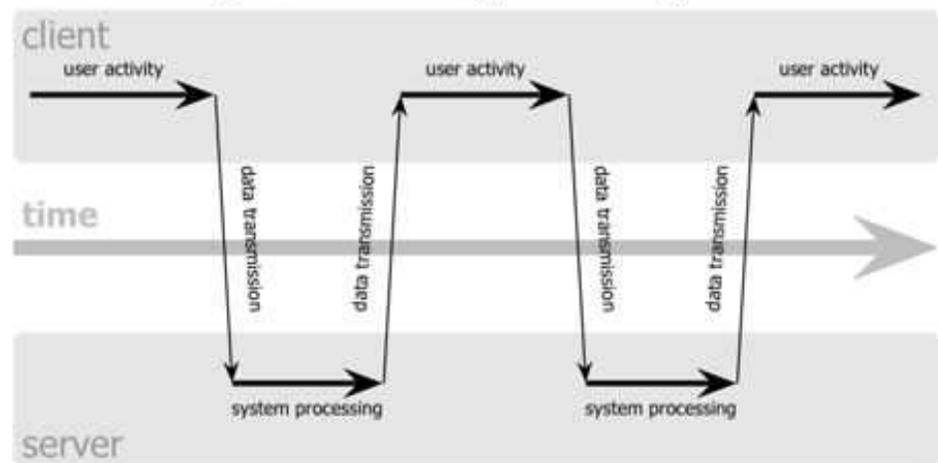
Oggetto XMLHttpRequest

- L'oggetto Javascript usato per realizzare le richieste AJAX
- Effettua la richiesta di una risorsa via HTTP ad un server web
 - in modo indipendente dal browser
 - non sostituisce l'URI della propria richiesta all'URI corrente
 - non provoca un cambio di pagina
 - inviando eventuali informazioni sotto forma di variabili (come una *form*)
 - di tipo GET
 - di tipo POST
 - in modo
 - sincrono (blocca il flusso di esecuzione del codice Javascript.. non ci interessa)
 - asincrono (non interrompe il flusso di esecuzione del codice Javascript né le operazioni dell'utente sulla pagina)

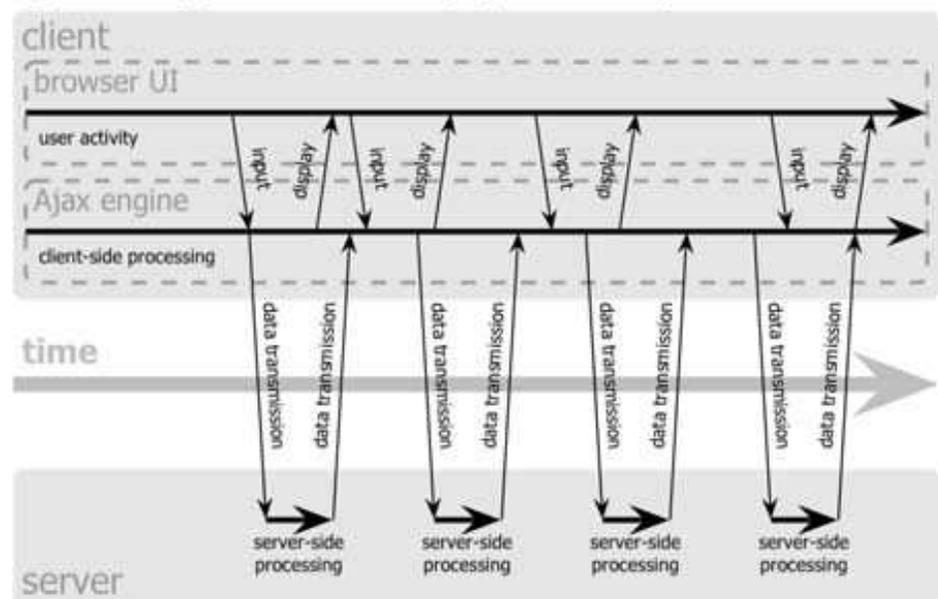
Alterazione del paradigma di interazione client/server

- Si guadagna in espressività, ma si perde la linearità dell'interazione
 - mentre l'utente è all'interno della stessa pagina le richieste sul server possono essere numerose e indipendenti
 - il tempo di attesa passa in secondo piano o non è avvertito affatto
- Possibili criticità sia per l'utente che per lo sviluppatore
 - percezione che non stia accadendo nulla (sito che non risponde)
 - problemi nel programmare logica che HA BISOGNO di aspettare i risultati delle richieste precedenti

classic web application model (synchronous)



Ajax web application model (asynchronous)



Tipologie di interazioni AJAX

- Semplici

- modifica del valore dell'attributo innerHTML di un elemento della pagina
 - accesso ai contenuti di uno *span*, di un *p*, ecc...
 - possibile assegnare non solo testo semplice, ma altro HTML!
- uso del DOM per aggiungere, popolare o modificare elementi
 - *getElementById()* (non compatibile con le vecchie versioni di Explorer)
 - *getElementsByTagName()*

- Avanzate

- invocazione di logica per l'elaborazione e la restituzione di contenuti server-side (necessaria programmazione lato server!)
- metodi del DOM per la creazione avanzata di contenuti strutturati, innestati, dinamici
- metodi del DOM per la creazione, gestione e manipolazione di dati XML
- gestioni di intervalli di tempo multipli o incrociati attraverso l'uso dei metodi Javascript *setInterval()* o *setTimeout()*

Una cosa che capita spesso...

- Astrarre i diversi comportamenti di diversi browser dietro a...
 - ...librerie di terze parti (es: *jquery*)
 - ...funzioni *ad hoc*

```
// from http://javascript.html.it/guide/leggi/95/guida-ajax/
function myGetElementById(idElemento) {

    // elemento da restituire
    var elemento;

    // se esiste il metodo getElementById questo if sarà
    // diverso da false, null o undefined
    // e sarà quindi considerato valido, come un true
    if ( document.getElementById )
        elemento = document.getElementById(idElemento);

    // altrimenti è necessario usare un vecchio sistema
    else
        elemento = document.all[idElemento];

    // restituzione elemento
    return elemento;

}
```

Primi passi...

- Evento *onload* e disponibilità del DOM

http://localhost:8080/TemplateAJAX/1_beforeonload.html ⚡

http://localhost:8080/TemplateAJAX/2_onload.html

Firebug: inspect HTML e breakpoint su...

- *scripts/almostajax.js:6*

...lettura degli eventuali messaggi di errore

```
<html>
  <head>
    <title>AJAX, le basi prima dell' utilizzo /title>
    <script type="text/javascript" src="myutils.js"></script>
    <script type="text/javascript" src="almostajax.js"></script>
  </head>
  <body onload="almostAjax( 'paragrafo' )">
    <p id="paragrafo">
      testo del paragrafo
    </p>
  </body>
</html>
```

l'evento *onload* non ha molto a che fare con AJAX, ma è essenziale per ogni interazione asincrona: **finchè non si è certi di avere gli elementi del DOM caricati è inutile tentare di effettuare modifiche!**

invocare la procedura direttamente nel tag *<head>* della pagina avrebbe generato errore!

Prima del verificarsi dell'evento *onload* non è nota al browser nemmeno l'esistenza di un elemento con *id="paragrafo"*!!

**INVIO
DELLE RICHIESTE**

Ottenere l'oggetto XMLHttpRequest

Funzionalità da astrarre con funzioni *ad hoc* per ottenere cross-browser compatibility

- Alcuni browser lo supportano come oggetto nativo (*Firefox 1+*, *Opera 7+*, *Safari*, *Internet Explorer 7*):

```
var xhr = new XMLHttpRequest();
```

- Versioni precedenti di *Internet Explorer* lo supportano come oggetto ActiveX, solo dalla versione 4 e in modi differenti a seconda della versioni:

```
var xhr = new ActiveXObject("Microsoft.XmlHttp")
```

```
var xhr = new ActiveXObject("MSXML4.XmlHttp")
```

```
var xhr = new ActiveXObject("MSXML3.XmlHttp")
```

```
var xhr = new ActiveXObject("MSXML2.XmlHttp")
```

```
var xhr = new ActiveXObject("MSXML.XmlHttp")
```

- Esistono poi browser che non lo supportano affatto:
 - E' buona norma controllare e prevedere comportamenti non AJAX in loro presenza!

dalla piu' recente → alla piu' obsoleta

Controllo del supporto

```
// ad esempio invocata in corrispondenza dell'evento onload
myAjaxApp = function() {
    var xhr = myGetXMLHttpRequest();
    if ( xhr ) { /* applicazione in versione AJAX */ }
    else { /* versione non AJAX o avviso all'utente */ }
}
```

```
// from http://www.e-time.it/topics/34-ajax/8-Richiamare%20%5C'oggetto%20XmlHttpRequest
function myGetXmlHttpRequest() {
    var xhr = false;
    var activeXoptions = new Array( "Microsoft.XmlHttp", "MSXML4.XmlHttp",
        "MSXML3.XmlHttp", "MSXML2.XmlHttp", "MSXML.XmlHttp" );
    // prima come oggetto nativo
    try { xhr = new XMLHttpRequest(); }
    catch (e) { }
    // poi come oggetto activeX dal piu' al meno recente
    if ( ! xhr ) {
        var created = false;
        for ( var i = 0 ; i < activeXoptions.length && !created ; i++ ) {
            try {
                xhr = new ActiveXObject( activeXoptions[i] );
                created = true;
            }
            catch (e) { }
        }
    }
    return xhr;
}
```

Metodi di *XMLHttpRequest*

- La lista dei metodi disponibili è diversa da browser a browser
- In genere si usano solo quelli presenti in Safari (sottoinsieme piu' limitato, ma comune a tutti i browser che supportano AJAX):

`open ()`

`setRequestHeader ()`

`send ()`

`getResponseHeader ()`

`getAllResponseHeaders ()`

`abort ()`

Metodo *open()*

- Inizializzazione della richiesta da formulare al server

- definito dal W3C per accettare fino a 5 parametri.

```
open (method, uri [,async][,user][,password])
```

- l'uso piu' comune per AJAX ne prevede 3, di cui uno necessariamente fissato

```
open (method, uri , true)
```

Parametri di *open()*

- **method**: stringa che indica il metodo di invio dati ("*get*" o "*post*")
 - scegliendo GET le variabili sono appese alla URI della risorsa richiesta
pagina.html?variabile=valore&altraVariabile=altroValore
 - utilizzando POST sono all'interno del body della richiesta stessa
 - il numero di caratteri dell'URI che si possono inviare con una GET è pari a 256 (in realtà varia a seconda del browser 😊): pochi per scambiare dati XML!
- **uri**: stringa che identifica la risorsa da ottenere
 - mediante un indirizzo assoluto
http://localhost:8080//AJAXapp/contesto/pagina2.html
 - o relativo alla locazione della pagina da cui parte la richiesta
sottocontesto/pagina3.html
- **async**: valore booleano che deve essere impostato come *true* per indicare al metodo che la richiesta da effettuare è di tipo asincrono



Metodi `setRequestHeader()` e `send()`

- **`setRequestHeader()`**: impostazione degli header della richiesta
 - coppie chiave/valore che descrivono lo scambio dati
 - opzionali per le richieste GET
 - necessari per descrivere il body delle richieste POST
- **`send()`**: inoltro della richiesta al server
 - non bloccante se il parametro `async` di `open` è stato impostato a `true`

```
// es: richiesta GET
var xhr = myGetXMLHttpRequest();
xhr.open( "get", "sottocontesto/pagina.html?p1=v1&p2=v2", true );
xhr.setRequestHeader( "connection", "close" );
xhr.send( null );
```

```
// es: richiesta POST
var xhr = myGetXMLHttpRequest();
xhr.open( "post", "sottocontesto/pagina.html", true );
xhr.setRequestHeader( "content-type", "x-www-form-urlencoded" );
xhr.setRequestHeader( "connection", "close" );
xhr.send( "p1=v1&p2=v2" );
```

si possono passare anche valori con caratteri non validi ('&', ecc...) filtrandoli attraverso la funzione Javascript:
`escape(value_with_invalid_characters)`

**LETTURA
DELLE RISPOSTE**

Proprietà di *XMLHttpRequest*

- Stato e risultati della richiesta vengono memorizzati dall'interprete Javascript all'interno dell'oggetto *XmlHttpRequest* durante la sua esecuzione

- Lista dei parametri comunemente supportati dai vari browser:

`readyState`

`onreadystatechange`

`status`

`responseText`

`responseXML`

Proprietà *readyState*

- Variabile di tipo intero, con valori che vanno da 0 a 4.
 - **0** : *uninitialized* - l'oggetto *XMLHttpRequest* esiste, ma non è stato richiamato alcun metodo per inizializzare una comunicazione
 - **1** : *open* - è stato precedentemente invocato il metodo *open()*, ma il metodo *send()* non ha ancora effettuato l'invio dati
 - **2** : *sent* - il metodo *send()* è stato eseguito ed ha effettuato la richiesta
 - **3** : *receiving* - i dati in risposta cominciano ad essere letti
 - **4** : *loaded* - l'operazione è stata completata
- Accessibile in **sola lettura**: rappresenta in ogni istante lo stato della richiesta
- Note:
 - nello stato 3 (che può essere assunto più volte...) si possono già leggere alcuni header restituiti dal server o parte della risposta
 - questo ordine (dalla bozza di lavoro del W3C) non è sempre identico e non è sfruttabile allo stesso modo su tutti i browser
 - se la richiesta fallisce *readyState* potrebbe non assumere mai il valore 3
 - **l'unico stato supportato da tutti i browser è il 4: a prescindere dalla riuscita dell'operazione**, le operazioni sono terminate e lo stato non cambierà più.

Proprietà *onreadystatechange*

- L'esecuzione del codice non si blocca sulla *send()* in attesa dei risultati
- Occorre registrare una funzione che sia richiamata dal sistema
 - **come?** ...in maniera asincrona rispetto al resto del programma
 - **perché?** ...al momento della disponibilità di risultati (anche parziali!)
 - **quando?** ...a sua volta segnalata dal cambio di stato della richiesta

```
xhr.onreadystatechange = function() { /* callback */ }
```

- Occorre evitare alee!

```
var xhr = myGetXMLHttpRequest();  
xhr.open( "post", "sottocontesto/pagina.html?", "post", true );  
xhr.setRequestHeader( "content-type", "application/x-www-form-urlencoded" );  
xhr.setRequestHeader( "connection", "close" );  
xhr.send( "p1=v1&p2=v2" );  
  
← risposta del server  
xhr.onreadystatechange = function() { ... // troppo tardi ☹️
```

è meglio **fare tale assegnamento prima della send() !!!!**

Proprietà *status*

- Valore intero corrispondente al codice HTTP dell'esito della richiesta
 - **200** : caso di successo (l'unico in base al quale è possibile i dati ricevuti in risposta devono essere ritenuti corretti e significativi)
 - possibili altri valori (in particolare d'errore: 403, 404, 500, ...)
- Una descrizione testuale del codice HTTP ritornato dal server...
 - è contenuta nel parametro *statusText* (supportato in quasi tutti i browsers tranne alcune versioni di Opera)

```
if ( xhr.status != 200 ) alert( xhr.statusText );
```

- può essere ottenuta creando e utilizzando un'apposito oggetto di mappe codici-descrizione

```
if ( xhr.status != 200 ) alert( httpCodes[xhr.status] );
```

Proprietà *responseText* e *responseXML*

- Contengono i dati restituiti dal server
- ***responseText*** : dato di tipo stringa, disponibile solo ad interazione ultimata (*readyState* == 4)
 - permette di ricevere qualsiasi informazione dal server
 - la rappresentazione testuale del body della risposta gli viene comunque assegnata se la comunicazione termina con successo
- ***responseXML*** : lo stesso dato, convertito in documento XML (se possibile) ai fini della navigazione via Javascript
 - potrebbe essere *null* qualora i dati restituiti non siano un documento XML ben formato (es: trasmissione di dati non XML, dati XML corrotti durante la trasmissione, ecc)

Metodi *getResponseHeader()* e *getAllResponseHeaders()*

- Lettura degli header HTTP che descrivono la risposta del server
 - utilizzabili solo **nella funzione di callback**
 - e comunque, anche al suo interno
 - da **non invocare immediatamente dopo l'invio dei dati** in maniera asincrona (*send, readystate == 2*)
 - utilizzabili per leggere parte degli header fin dall'inizio della ricezione della risposta (*readystate == 3*)
 - utilità limitata (es: ottimizzazione, ecc..)
 - in grado di accedere con certezza all'elenco completo degli header solo a richiesta conclusa (*readystate == 4*)

La funzione di “callback” (1)

- Legge lo stato di avanzamento della richiesta
 - *readystate*
- Verificare il successo o fallimento della richiesta
 - *status*
- Ha accesso agli header di risposta rilasciati dal server (parziali se *readystate* == 3, completi se *readystate* == 4)
 - *getAllResponseHeaders()*
 - *getResponseHeader(header_name)*
- Può leggere il contenuto della risposta (se e solo se *readystate* == 4)
 - *responseText*
 - *responseXML*

La funzione di “callback” (2)

- Assegnata all'attributo *onreadystatechange* di *XMLHttpRequest*

```
var xhr = // .. etc etc

var textHolder = new Object();
textHolder.testo = "La risposta del server è: ";

xhr.onreadystatechange = function() {
    if ( xhr.readyState == 4 && xhr.status == 200 ) {
        /*
         * anche se la funzione è assegnata a una proprietà di xhr,
         * dal suo interno non è possibile riferirsi a xhr con this
         * perche' la funzione sarà richiamata in modo asincrono dall'interprete
         */
        // alert ( textHolder.testo + this.responseText );
        alert ( textHolder.testo + xhr.responseText );
    }
};
```

- Richiamata AD OGNI VARIAZIONE del parametro *readyState*
 - su alcuni browser lo stato 3 può essere assunto più volte in caso di ricezione di una risposta molto lunga in successivi trunk

Chiusura della funzione di “callback”

- E' costituita dalla funzione stessa e dall'insieme di tutte le variabili a cui essa può accedere (scope)
 - se definita **in linea**, può riferire le variabili dello scope in cui si trova

```
var xhr = // .. etc etc
var textHolder = new Object();
...
xhr.onreadystatechange = function() {
    if ( xhr.readyState == 4 /* ...omissis */
        alert( textHolder.testo ); /* ...omissis */
}
```

- se definita come **funzione esterna** può accettare parametri formali e riferirne i valori attuali al proprio interno mediante i loro nomi

```
function myPopup( oggettoAjax, contenitoreDiTesto ) {
    if ( oggettoAjax.readyState == 4 /* ...omissis */
        alert( contenitoreDiTesto.testo ); /* ...omissis */
}
...
var xhr = // .. etc etc
var textHolder = new Object();
...

/* TUTTAVIA VA UTILIZZATA NECESSARIAMENTE COSI' !!! */
xhr.onreadystatechange = function() { myPopup(xhr, textHolder); }

/* NON E' INVECE POSSIBILE QUESTO TIPO DI ASSEGNAZIONE !!! */
xhr.onreadystatechange = myPopup(xhr, textHolder);
```

ESEMPI

Ovviamente...

IMPORTANTISSIMO!!!

Per poter testare questo esempio, come qualunque altro applicativo basato su *XMLHttpRequest*, è **necessario richiamare la pagina HTML attraverso un webserver**, (il termine *Http* nel nome non è casuale!)

http://localhost:8080/AJAXapp/pagina.html

Non è possibile specificare come URI della richiesta l'ubicazione di risorse sul file system locale della macchina!!!

~~C:\pagina.html~~

Scaricamento di dati in formato testo (1)

- Pagine

http://localhost:8080/TemplateAJAX/3_plaintext.html ⚡

http://localhost:8080/TemplateAJAX/4_plaintext-external.html

```
/* VEDERE IL CODICE, COMMENTATO, NEL PROGETTO DI ESEMPIO */
```

- Verifica della possibilità di usare tecniche AJAX
 - creazione e invio della richiesta asincrona
 - gestione di un'alternativa in caso di mancato supporto ad AJAX

⚡ Firebug: inspect HTML e breakpoint su...

- *scripts/callback.js:6*

...eseguire una prima volta passo passo

...provare a cambiare il valore di xhr a *false*

Scaricamento di dati in formato testo (2)

- Pagine

http://localhost:8080/TemplateAJAX/3_plaintext.html ⚡

http://localhost:8080/TemplateAJAX/4_plaintext-external.html

```
/* VEDERE IL CODICE, COMMENTATO, NEL PROGETTO DI ESEMPIO */
```

- Esecuzione di operazioni differenti a seconda
 - dello stato della richiesta
 - della sua condizione di successo o fallimento

Firebug: breakpoint su...

- *scripts/callback.js:38, 42, 50, 55*

...eseguire una prova facendo resume subito dopo ogni breakpoint
...provare una richiesta a una URI diversa (possibilmente non valida)

Scaricamento di dati in formato testo (3)

- Uso di funzioni di callback...
 - ...interne (semplice accesso agli oggetti presenti nello scope della funzione, ma difficile manutenibilità e riuso del codice)
http://localhost:8080/TemplateAJAX/3_plaintext.html
 - ...esterne (ma senza dichiarare oggetti XMLHttpRequest globali)
http://localhost:8080/TemplateAJAX/4_plaintext-external.html



```
/* VEDERE IL CODICE, COMMENTATO, NEL PROGETTO DI ESEMPIO */
```

Firebug: inspect Script...

- [scripts/callback-external.js:66](#)
- [scripts/callback-external.js:12](#)

...leggere bene i commenti !!

Scaricamento di dati in formato XML (1)

- Utilizzo di un feed RSS (prodotto da una pagina JSP che legge da DB tramite DAO)
 - utilizzo di *responseXML* in caso di successo
 - utilizzo di *responseText* in caso di errori o fallimenti
- http://localhost:8080/TemplateAJAX/5_rssread.html 

```
/* VEDERE IL CODICE, COMMENTATO, NEL PROGETTO DI ESEMPIO */
```

...completare il nome della categoria di notizie richiesta (le *XmlHttpRequest* partono a ogni *keyUp!*)

Firebug: inspect HTML e breakpoint su...

- *scripts/rssparses.js:85*

...modificare la URI specificando una categoria tra "POLITICA", "ECONOMIA", "SPETTACOLI", ecc...

...eseguire step over

...eseguire step into nelle funzioni di parsing

Richieste AJAX cross-domain

- Per motivi di sicurezza la XMLHttpRequest può essere rivolta solo verso il dominio da cui proviene la risorsa che la utilizza

- la risorsa *testo.txt* usata per il primo esempio è disponibile anche su

<http://www-lia.deis.unibo.it/Courses/TecnologieWeb0809/materiale/laboratorio/prove/testo.txt>

- ma il dominio www-lia.deis.unibo.it è irraggiungibile per le richieste AJAX delle pagine del progetto, che il vostro browser riconosce come provenienti localhost
 - potete provare...
 - E per “leggere” gli RSS di un sito esterno (es: *Repubblica.it*) ???
 - occorre recuperarli tramite logica server-side (Servlet, JSP, ma anche JSF, PHP, .NET, CGI in genere, ...) che faccia da “proxy”

sito esterno -> nostro web server -> risorse scaricate da URI del nostro sito

E le mappe di Google, allora?

- Come è possibile che alcuni siti “embeddino” le mappe di google?
 - Le mappe producono richieste AJAX verso i server di google...
 - ...ma sono su pagine scaricate da altri domini
- Semplicemente... le mappe girano all'interno di *iframe*!
 - Le *XmlHttpRequest* dell'*iframe* sono dirette a Google...
 - ...così come l'attributo *src* dell'*iframe* stesso
- Allora posso fare richieste all'esterno da dentro un *iframe*...?
 - ...ad esempio per leggere questi benedetti RSS da *Repubblica.it* ??

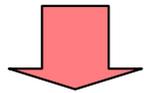


 - **no**; dovrete scaricare da *Repubblica.it* non solo i feed RSS ma anche il documento HTML che richiama gli script che creano le richieste!

**RIFINIRE
IL MODELLO**

L'utente abbandonato

- Le richieste AJAX
 - permettono all'utente di continuare a interagire con la pagina
 - ma non necessariamente lo informano di cosa sta succedendo
 - e possono durare troppo!
- L'utente non sa cosa stanno facendo i nostri script
 - ...o gli insegniamo a usare firebug
 - ...o facciamo in modo di informarlo noi!
- Infine, è giusto interrompere le richieste che non terminano in tempo utile
 - server momentaneamente sovraccarico
 - problemi di rete



- disorientamento dell'utente

Richieste fantasma

- E' molto difficile generare una richiesta fantasma *ad hoc* scaricando solo contenuti statici dal web server (sebbene in modo asincrono, dinamico, ecc...)
 - caduta della connessione tra *readyState* 3 e 4... (bisogna essere svelti a staccare i cavi di rete ☺)
 - scaricamento di un file di grosse dimensioni (difficile da distribuire in lab a causa dei problemi di spazio, quota, ecc... ☺ e non si può neanche farlo dal sito del corso... le richieste AJAX partono dai vostri *localhost*, ricordate?)
- Capita anche troppo spesso, invece, quando si invoca logica server-side
 - l'uso smodato di AJAX genera un numero esagerato di richieste verso il server
 - ecco perché sostituiamo l'header ***connection=keep-alive*** con ***connection=close*** → noi dobbiamo riaprire la connessione ogni volta e andiamo più piano, ma il server non collassa perché esaurisce le connessioni disponibili in caso di accesso di molti utenti contemporaneamente!
 - oppure, semplicemente, la logica server-side ci mette molto!

Metodo *abort()*

- Interruzione delle operazioni di invio o ricezione
 - non ha bisogno di parametri
 - termina immediatamente la trasmissione dati
- Per poterlo utilizzare in modo sensato, tuttavia, non si può richiamarlo in modo sincrono dentro la funzione di callback
 - se *readyState* non cambia non viene richiamato!
 - e tipicamente *readyState* non cambia quando la risposta si fa attendere.
- Si crea un'altra funzione da far richiamare in modo asincrono al sistema mediante il metodo *setTimeout(funzioneAsincronaPerAbortire, timeOut)*
 - e al suo interno si valuta se continuare l'attesa o abortire l'operazione

Per provare... (1)

- Pagine

[http://localhost:8080/TemplateAJAX/6_loadwait.html?wait=\[secondi\]](http://localhost:8080/TemplateAJAX/6_loadwait.html?wait=[secondi]) 

- Richiamano una risorsa server-side (la mitica servlet che attende!!!) che attende per il numero di secondi indicato prima di restituire il controllo e il risultato
- Gli script nella pagina attendono comunque solo per 5 secondi
 - viene mostrata una immagine animata nell'attesa
 - ed eventualmente un messaggio di fallimento se la risposta impiega di più

```
/* VEDERE IL CODICE, COMMENTATO, NEL PROGETTO DI ESEMPIO */
```

Firebug: inspect Script e breakpoint su...

- *scripts/loadabort.js:168, 86*

...esecuzione step by step

Per provare... (2)

- Pagine

[http://localhost:8080/TemplateAJAX/6_loadwait.html?wait=\[secondi\]](http://localhost:8080/TemplateAJAX/6_loadwait.html?wait=[secondi]) ⚡

- Notate l'uso della console di firebug per loggare informazioni relative alle attività in corso quando la richiesta viene abortita!

- l'alternativa è scrivere tante *alert()* mentre si sviluppa e poi commentarle!!

```
/* VEDERE IL CODICE, COMMENTATO, NEL PROGETTO DI ESEMPIO */
```

⚡ Firebug: inspect Script e breakpoint su...

- *scripts/loadabort.js:128*

...controllare la console!

TOCCA A VOI...

Restyling della pagina per la lettura di feed RSS (1)

- Nell'ordine che preferite...
 - estendete le funzionalità del DAO per leggere le categorie esistenti a partire dalla parte iniziale del loro nome

```
public List<String> getCategories(String likeStartingWith);
```

(query *like* nell'esercitazione su DAO - n.b.: qui *like* solo della parte finale)
 - spostate la funzione di scaricamento dei feed in XML dall'evento *onKeyUp* del campo di input a quello di *onClick* su un opportuno `` (prendete esempio dalle altre pagine dell'esercitazione di oggi)
 - associate a *onKeyUp* (e scrivetela) una funzione AJAX che scarichi il nome della prima categoria che inizia con le lettere immesse dall'utente e sostituisce tale risultato (se presente) all'attributo *value* del campo di input (prendete esempio dalle altre pagine dell'esercitazione di oggi)
 - scrivete infine la pagina JSP (o la Servlet, se preferite) associata all'URL chiamato dalla vostra funzione AJAX, che legge dai parametri in GET le iniziali della categoria cercata, interroga il database tramite DAO e restituisce la prima stringa ottenuta

Restyling della pagina per la lettura di feed RSS (2)

- E ancora...
- aggiungete a WEB-INF/lib le librerie che trovate nell'archivio zip dell'esercitazione parser/serializzatore JSON per Java (*jabsorb*) e sue dipendenze (*slf4j*)
- scrivete una Servlet che, come la pagina *feed.jsp*, recuperi la categorie di notizie desiderata dai parametri della richiesta, interroghi il db e restituisca il risultato mediante il serializzatore JSON
(sbirciate nella soluzione soltanto al momento di usare il serializzatore)
 - aggiungete la libreria Javascript *json.js* alle risorse web del progetto (scegliete pure la posizione che preferite... es: *web/scripts/json*)
 - modificate la pagina *5_rssfeed.html* per **a)** riferire ANCHE a libreria *json.js* e **b)** rivolgere la propria richiesta asincrona alla Servlet che avete appena realizzata (vi siete ricordati di mappare la Servlet nel descrittore *web.xml* ?)
- modificate la funzione javascript che analizza l'XML delle notizie per analizzare, al suo posto, la stringa di testo JSON restituita dalla nuova Servlet