



## Ajax

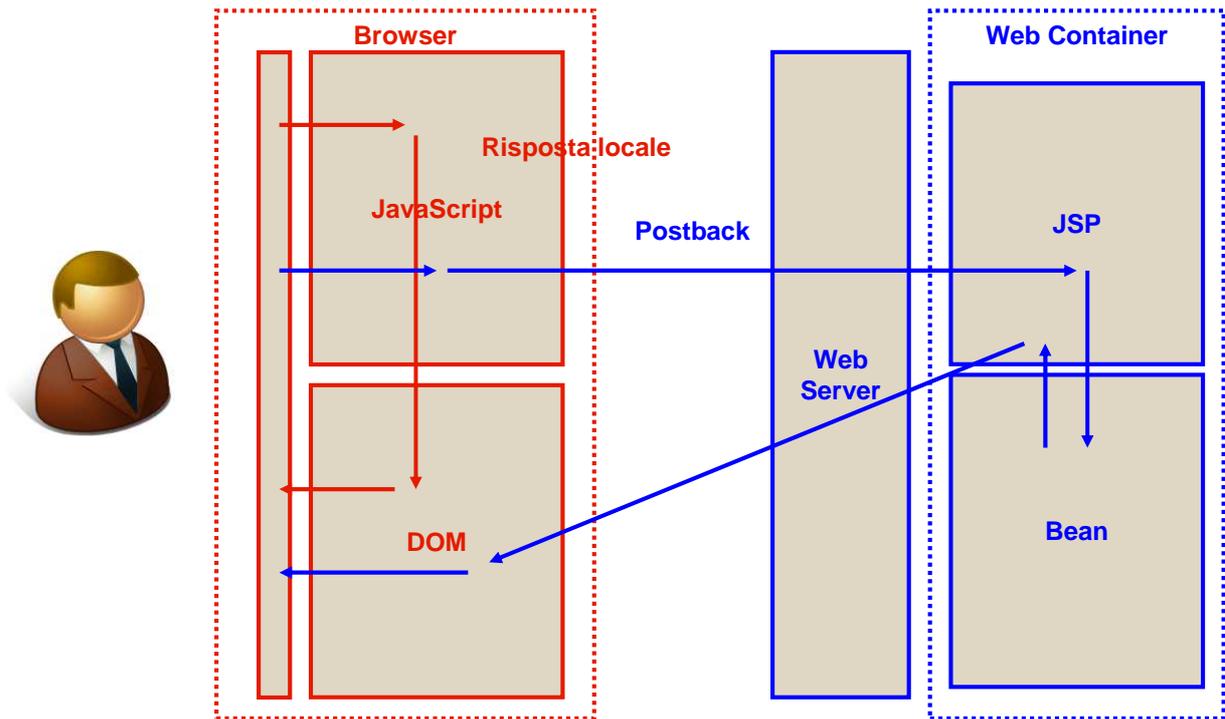
1

### Un nuovo modello

- L'utilizzo di **DHTML** (JavaScript/Eventi + DOM + CSS) delinea un nuovo modello per le applicazioni Web
- In pratica ci troviamo di fronte ad un modello ad eventi simile a quello delle applicazioni tradizionali
- Abbiamo però due livelli di eventi:
  - **Eventi locali** che portano ad una modifica diretta del DOM da parte di Javascript e quindi ad un cambiamento locale della pagina
  - **Eventi remoti** ottenuti tramite un ricaricamento della pagina che viene modificata sul lato server in base ai parametri passati nel GET o nel POST
- Il ricaricamento di una pagina per rispondere ad un'interazione con l'utente prende il nome di **postback**

2

## Modello ad eventi a due livelli



3

## Esempio di evento remoto - 1

- Consideriamo un form in cui compaiono due tendine che servono a selezionare il comune di nascita di una persona
  - Una con le province
  - Una con i comuni
- Si vuole fare in modo che scegliendo la provincia nella prima tendina nella seconda appaiano solo i comuni di quella provincia

Provincia / Comune di nascita*	BOLOGNA ▼	CALDERARA DI RENO ▼
-----------------------------------	-----------	---------------------

4

## Esempio di evento remoto - 2

---

- Per realizzare questa interazione si procede così
  - Si crea una JSP che inserisce nella tendina dei comuni l'elenco di quelli che appartengono alla provincia passata come parametro
  - Si definisce un evento **onchange** collegato all'elemento **select** delle province
  - Lo script collegato ad onchange forza il ricaricamento della pagina con un POST (**postback**)
- Quindi:
  - L'utente sceglie una provincia
  - Viene invocata la JSP con il parametro della provincia impostato al valore scelto dall'utente
  - La pagina restituita contiene nella tendina dei comuni l'elenco di quelli che appartengono alla provincia scelta

---

5

## Limiti del modello a ricaricamento di pagina

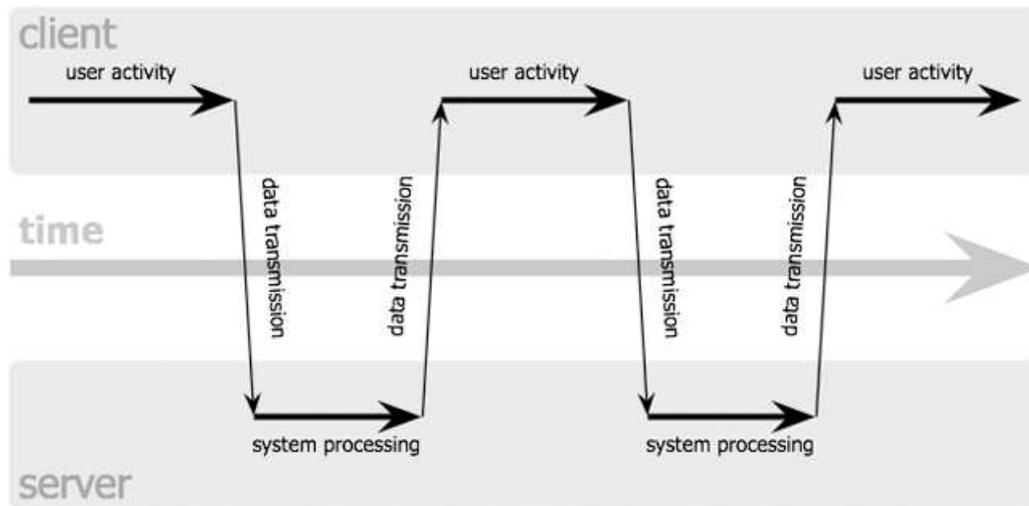
---

- Quando lavoriamo con applicazioni desktop siamo abituati ad un elevato livello di interattività:
  - le applicazioni reagiscono in modo rapido ed intuitivo ai comandi
- Le applicazioni web tradizionali espongono invece un modello di interazione rigido
  - Modello "**Click, wait and refresh**"
  - E' necessario il refresh della pagina da parte del server per la gestione di qualunque evento (sottomissione di dati tramite form, visita di un link per ottenere informazioni di interesse, ...)
- E' ancora un **modello sincrono**: l'utente effettua una richiesta e deve attendere la risposta da parte del server

---

6

## Modello di interazione classico



7

## AJAX

- Il modello **AJAX** è nato per superare queste limitazioni
- AJAX non è un acronimo ma spesso viene interpretato come **A**synchronous **J**avascript **A**nd **X**ml
- E' basato su tecnologie standard:
  - JavaScript
  - DOM
  - XML
  - HTML
  - CSS

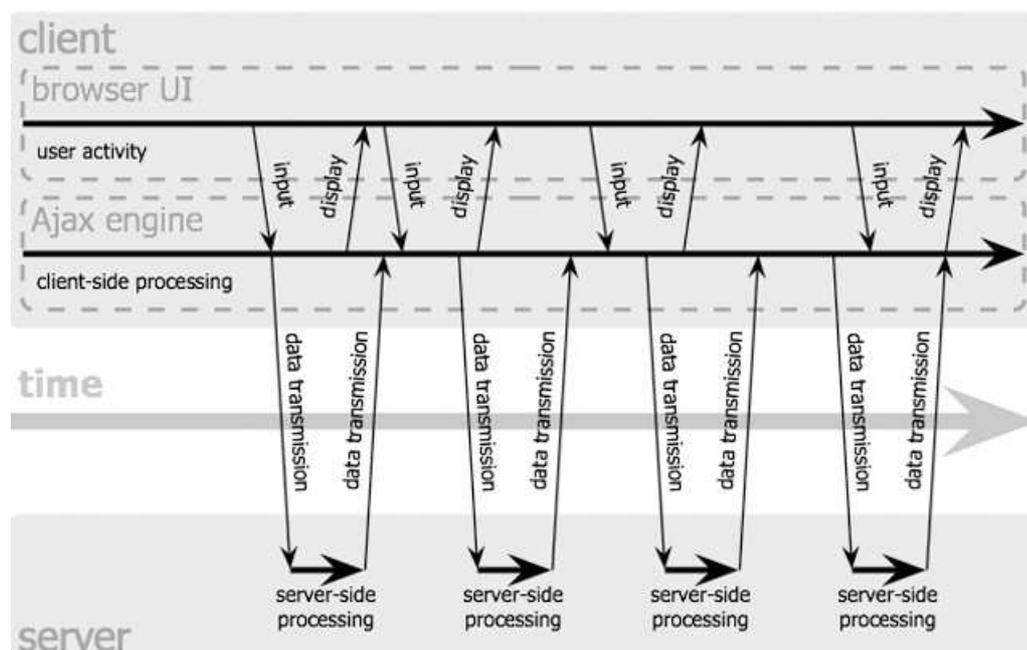
8

## AJAXC

- AJAX punta a supportare applicazioni user friendly con un'interattività elevata
- Si usa spesso il termine RIA (**Rich Interface Application**)
- L'idea alla base di AJAX è quella di consentire agli script JavaScript di interagire direttamente con il server
- Si usa l'oggetto JavaScript **XMLHttpRequest**
  - Consente di ottenere dati dal server senza la necessità di ricaricare l'intera pagina
  - Realizza una comunicazione **asincrona** fra client e server: il client non interrompe l'interazione con l'utente anche quando è in attesa di risposte dal server

9

## Modello di interazione con Ajax



10

## Tipica sequenza Ajax

---

- Si verifica un evento determinato dall'interazione fra l'utente e la pagina web
- L'evento comporta l'esecuzione di una funzione JavaScript in cui:
  - Si istanzia un oggetto di tipo **XMLHttpRequest**
  - Si configura XMLHttpRequest: si associa una funzione di callback, si effettua il setup...
  - Si effettua una chiamata asincrona al server
- Il server elabora la richiesta e risponde al client
- Il browser invoca la funzione di callback che:
  - elabora il risultato
  - aggiorna il DOM della pagina per mostrare i risultati dell'elaborazione

---

11

## XMLHttpRequest

---

- L'oggetto **XMLHttpRequest** effettua la richiesta di una risorsa via HTTP ad un server web
  - Non sostituisce l'URI della propria richiesta all'URI corrente
  - Non provoca un cambio di pagina
  - Può inviare eventuali informazioni (parametri) sotto forma di variabili (come una form)
- Può effettuare sia richieste GET che POST
- Le richieste possono essere di tipo
  - **Sincrono**: blocca il flusso di esecuzione del codice Javascript (ci interessa poco)
  - **Asincrono**: non interrompe il flusso di esecuzione del codice Javascript né le operazioni dell'utente sulla pagina

---

12

## Creazione di un'istanza

---

- I browser più recenti (Firefox 1+, Opera 7+, Safari, Internet Explorer 7 e 8) supportano `XMLHttpRequest` come oggetto nativo
- In questo caso (oggi il più comune) le cose sono molto semplici:  

```
var xhr = new XMLHttpRequest();
```
- La gestione della compatibilità con browser più vecchi complica un po' le cose
- La esamineremo in seguito per non rendere difficile la comprensione del modello
- **Attenzione:** per motivi di sicurezza `XMLHttpRequest` può essere utilizzata solo verso il dominio da cui proviene la risorsa che la utilizza

---

13

## Metodi di XMLHttpRequest

---

- La lista dei metodi disponibili è diversa da browser a browser
- In genere si usano solo quelli presenti in Safari (sottoinsieme più limitato, ma comune a tutti i browser che supportano AJAX):
  - `open()`
  - `setRequestHeader()`
  - `send()`
  - `getResponseHeader()`
  - `getAllResponseHeaders()`
  - `abort()`

---

14

## Metodo open()

---

- `open ( )` ha lo scopo di inizializzare la richiesta da formulare al server
- Lo standard W3C prevede 5 parametri, di cui 3 opzionali:  
`open (method, uri [,async][,user][,password])`
- L'uso più comune per AJAX ne prevede 3, di cui uno comunemente fissato:  
`open (method, uri, true)`
- Dove:
  - **method**: stringa e assume il valore "get" o "post"
  - **uri**: stringa che identifica la risorsa da ottenere (URL assoluto o relativo)
  - **async**: valore booleano che deve essere impostato come true per indicare al metodo che la richiesta da effettuare è di tipo asincrono

---

15

## Metodi setRequestHeader() e send()

---

- `setRequestHeader (nomeheader, valore)` consente di impostare gli header HTTP della richiesta da inviare
  - Viene invocata più volte, una per ogni header da impostare
  - Per una richiesta GET gli header sono opzionali
  - Sono invece necessari per impostare la codifica utilizzata nelle richieste POST
  - E' comunque importante impostare l'header **connection** al valore **close** per evitare di aprire troppe connessioni
- `send (body)`: consente di inviare la richiesta al server
  - Non è bloccante se il parametro `async` di `open` è stato impostato a `true`
  - Prende come parametro una stringa che costituisce il body della richiesta HTTP

---

16

## Esempi

---

### GET

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

### POST

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

---

17

## Proprietà di XMLHttpRequest

---

- Stato e risultati della richiesta vengono memorizzati dall'interprete Javascript all'interno dell'oggetto **XMLHttpRequest** durante la sua esecuzione
- Le proprietà comunemente supportate dai vari browser sono:
  - **readyState**
  - **onreadystatechange**
  - **status**
  - **statusText**
  - **responseText**
  - **responseXML**

---

18

## Proprietà ReadyState

---

- Proprietà in sola lettura di tipo intero che consente di leggere in ogni momento lo stato della richiesta ammette 5 valori:
  - **0: uninitialized** - l'oggetto esiste, ma non è stato richiamato `open()`
  - **1: open** - è stato invocato il metodo `open()`, ma `send()` non ha ancora effettuato l'invio dati
  - **2: sent** - il metodo `send()` è stato eseguito ed ha effettuato la richiesta
  - **3: receiving** - la risposta ha iniziato ad arrivare
  - **4: loaded** - l'operazione è stata completata
- **Attenzione:**
  - Questo ordine non è sempre identico e non è sfruttabile allo stesso modo su tutti i browser
  - **L'unico stato supportato da tutti i browser è il 4**

19

## Proprietà onreadystatechange

---

- Come si è detto l'esecuzione del codice non si blocca sulla `send()` in attesa dei risultati
- Per gestire la risposta si deve quindi adottare un approccio ad eventi
- Occorre registrare una funzione di callback che viene richiamata in modo asincrono dal sistema ad ogni cambio di stato della richiesta
- La sintassi è

```
xhr.onreadystatechange = nomefunzione
xhr.onreadystatechange = function(){istruzioni}
```
- **Attenzione:** per evitare comportamenti imprevedibili l'assegnamento va fatto prima del `send()`

20

## Proprietà `status` e `statusText`

---

- `status` contiene un valore intero corrispondente al codice HTTP dell'esito della richiesta:
  - 200 in caso di successo (l'unico in base al quale i dati ricevuti in risposta possono essere ritenuti corretti e significativi)
  - Possibili altri valori (in particolare d'errore: 403, 404, 500, ...)
- `statusText` contiene invece una descrizione testuale del codice HTTP ritornato dal server...
- Esempio:

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

---

21

## Proprietà `responseText` e `responseXML`

---

- Contengono i dati restituiti dal server
- `responseText`: stringa che contiene il body della risposta HTTP
  - disponibile solo ad interazione ultimata (`readystate==4`)
- `responseXML`: body della risposta convertito in documento XML (se possibile)
  - consente la navigazione via Javascript
  - può essere `null` se i dati restituiti non sono un documento XML ben formato

---

22

## Metodi `getResponseHeader()`, `getAllResponseHeaders()`

- Consentono di leggere gli header HTTP che descrivono la risposta del server
  - Sono utilizzabili solo nella funzione di callback
  - Possono essere sicuramente invocati solo a richiesta conclusa (`readystate==4`)
  - In alcuni browser possono essere invocata anche in fase di ricezione della risposta (`readystate==3`)

- **Sintassi:**

`getAllResponseHeaders()`

`getResponseHeader(header_name)`

## Ruolo della funzione di callback

- Viene invocata ad ogni variazione di `readystate`
- Usa `readystate` per leggere lo stato di avanzamento della richiesta
- Usa `status` per verificare l'esito della richiesta `status`
- Ha accesso agli header di risposta rilasciati dal server con `getAllResponseHeaders()` e `getResponseHeader()`
- Se `readystate==4` può leggere il contenuto della risposta con `responseText` e `responseXML`

## Esempio

---

- **Attenzione:** anche se la funzione è assegnata a una proprietà di xhr, dal suo interno non è possibile riferirsi a xhr con this
- Questo perché la funzione viene richiamata in modo asincrono dall'interprete
- Può usare le variabili dello scope in cui si trova:

```
xhr = new XMLHttpRequest();
...
xhr.onreadystatechange = function()
{
    if ( xhr.readyState == 4 && xhr.status == 200 )
        alert ( "Risposta: "+ xhr.responseText );
};
```

---

25

## Vantaggi e svantaggi di Ajax

---

- 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 gestire un modello di elaborazione che ha bisogno di aspettare i risultati delle richieste precedenti

---

26

## Criticità nell'interazione con l'utente

---

- 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'effetto è un disorientamento dell'utente
- Dobbiamo quindi agire su due fronti:
  - Rendere visibile in qualche modo l'andamento della chiamata (barre di scorrimento ecc.)
  - Interrompere le richieste che non terminano in tempo utile per sovraccarichi del server o momentanei problemi di rete (timeout)

---

27

## Il metodo abort()

---

- **abort()** consente l'interruzione delle operazioni di invio o ricezione
  - non ha bisogno di parametri
  - termina immediatamente la trasmissione dati
- **Attenzione:** non ha senso invocarlo dentro la funzione di callback:
  - Se readyState non cambia non viene richiamato e 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)`
- Al suo interno si valuta se continuare l'attesa o abortire l'operazione

---

28

## Gestione della compatibilità

- I browser più recenti supportano XMLHttpRequest come oggetto nativo (Firefox, Opera 7+, Safari, Internet Explorer 7 e 8):

```
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 browser che non lo supportano affatto ed è quindi bene prevedere una gestione alternativa

29

## Gestione della compatibilità

- Si usano tecniche come quella riportata qui sotto:

```
function myGetXmlHttpRequest()
{
  var xhr = false;
  var activeXopt = 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 < activeXopt.length && !created; i++)
    {
      try {
        xhr = new ActiveXObject( activeXopt[i] );
        created = true; }
      catch (e) { }
    }
  }
  return xhr;
}
```

30

## Aspetti critici per il programmatore

---

- È accresciuta la complessità delle Web Application
- La logica di presentazione è ripartita fra client-side e server-side
- Le applicazioni AJAX pongono problemi di debug, test e mantenimento
- Il test di codice JavaScript è complesso
- Il codice JavaScript ha problemi di modularità
- I toolkit AJAX sono molteplici e solo recentemente hanno raggiunto una discreta maturità (es. Mootools, Scriptaculous, Prototype,...)
- Mancanza di standardizzazione di XMLHttpRequest e sua mancanza nei vecchi browser

---

31

## Gestire la risposta

---

- Spesso i dati scambiati fra client e server sono codificati in XML
- AJAX come abbiamo visto è in grado di ricevere documenti XML
- In particolare è possibile elaborare i documenti XML ricevuti utilizzando le API W3C DOM
- Il modo con cui operiamo su dati in formato XML è analogo a quello che abbiamo visto per ambienti Java
- Usiamo un parser ed accediamo agli elementi di nostro interesse
- Per visualizzare i contenuti ricevuti modifichiamo il DOM della pagina HTML

---

32

## Esempio:

- Scegliamo un nome da una lista e mostriamo i suoi dati tramite Ajax

```
<html>
  <head>
    <script src="selectmanager_xml.js"></script>
  </head>
  <body>
    <form action=""> Scegli un contatto:
    <select name="manager"
      onchange="showManager(this.value)"
      <option value="Carlo11">Carlo Rossi</option>
      <option value="Anna23">Anna Bianchi</option>
      <option value="Giovanni75">Giovanni Verdi</option>
    </select></form>
    <b><span id="companyname"></span></b><br/>
    <span id="contactname"></span><br/>
    <span id="address"></span>
    <span id="city"></span><br/>
    <span id="country"></span>
  </body>
</html>
```

Lista di  
selezione

Area in cui  
mostrare i  
risultati

33

## Esempio - 2

- Ipotizziamo che i dati sui contatti siano contenuti in un database.
- Il server:
  - riceve una request con l'identificativo della persona
  - interroga il database
  - restituisce un file XML con i dati richiesti

```
<?xml version='1.0' encoding='UTF-16'?>
<company>
  <compname>Microsoft</compname>
  <contname>Anna Bianchi</contname>
  <address>Viale Risorgimento 2</address>
  <city>Bologna</city>
  <country>Italy</country>
</company>
```

34

## Esempio: selectmanager\_xml.js

```
var xmlHttp;
function showManager(str)
{
  xmlHttp=new XMLHttpRequest();
  var url="getmanager_xml.jsp?q="+str;
  xmlHttp.onreadystatechange=stateChanged;
  xmlHttp.open("GET",url,true);
  xmlHttp.send(null);
}
function stateChanged()
{
  if (xmlHttp.readyState==4)
  {
    var xmlDoc=xmlHttp.responseXML.documentElement;
    var compEl=xmlDoc.getElementsByTagName("compname")[0];
    var comName = compEl.childNodes[0].nodeValue;
    document.getElementById("companyname").innerHTML=
      comName;
    ...
  }
}
```

35

## XML è la scelta giusta?

- Secondo un'interpretazione molto comune la X di Ajax sta per XML
- Abbiamo però visto nell'esempio precedente che l'utilizzo di XML come formato di scambio fra client e server porta alla scrittura di un codice molto prolisso
  - Difficile da leggere e da mantenere
  - Oneroso in termini di risorse di elaborazione (non dimentichiamo che JavaScript è interpretato)
- Esiste un formato più efficiente e semplice da manipolare per scambiare informazioni tramite Ajax?
- La risposta è sì e questo formato è in pratica quello più utilizzato

36

## JSON

---

- **JSON** è l'acronimo di **J**ava**S**cript **O**bject **N**otation
  - E' un formato per lo scambio di dati molto più comodo di XML
  - **JSON** è :
    - Leggero in termini di quantità di dati scambiati
    - Molto semplice da elaborare per un linguaggio di programmazione (in particolare per JavaScript)
    - Ragionevolmente semplice da leggere per un essere umano
  - E' largamente supportato dai maggiori linguaggi di programmazione
  - Si basa sulla notazione usata per le costanti oggetto (object literal) e le costanti array (array literal) in JavaScript
- 

37

## Oggetti e costanti oggetto

---

- In Javascript è possibile creare un oggetto in base ad una costante oggetto

```
var Beatles =  
{  
    "Paese" : "Inghilterra",  
    "AnnoFormazione" : 1959,  
    "TipoMusica" : "Rock"  
}
```

- Che equivale in tutto e per tutto a:

```
var Beatles = new Object();  
Beatles.Paese = "England";  
Beatles.AnnoFormazione = 1959;  
Beatles.TipoMusica = "Rock";
```

---

38

## Array e costanti array

- In modo analogo è possibile creare un array utilizzando una costante di tipo array:

```
var Membri =  
    ["Paul", "John", "George", "Ringo"];
```

- Che equivale in tutto e per tutto a

```
var Membri =  
    new Array("Paul", "John", "George", "Ringo");
```

- Possiamo anche avere oggetti che contengono array:

```
var Beatles =  
{  
    "Paese" : "Inghilterra",  
    "AnnoFormazione" : 1959,  
    "TipoMusica" : "Rock",  
    "Membri" : ["Paul", "John", "George", "Ringo"]  
}
```

39

## Array di oggetti

- E' infine possibile definire array di oggetti:

```
var Rockbands = [  
    {  
        "Nome" : "Beatles",  
        "Paese" : "Inghilterra",  
        "AnnoFormazione" : 1959,  
        "TipoMusica" : "Rock",  
        "Membri" : ["Paul", "John", "George", "Ringo"]  
    },  
    {  
        "Nome" : "Rolling Stones",  
        "Paese" : "Inghilterra",  
        "AnnoFormazione" : 1962,  
        "TipoMusica" : "Rock",  
        "Membri" : ["Mick", "Keith", "Charlie", "Bill"]  
    }  
]
```

40

## La sintassi JSON

- La sintassi JSON si basa su quella delle costanti oggetto e array di Javascript
- Un “oggetto JSON” è una stringa che equivale ad una costante oggetto di JavaScript:

Costante oggetto

```
{  
  "Paese" : "Inghilterra",  
  "AnnoFormazione" : 1959,  
  "TipoMusica" : "Rock'n'Roll",  
  "Membri" : ["Paul","John","George","Ringo"]  
}
```

JSON

```
'{"Paese" : "Inghilterra", "AnnoFormazione" :  
1959, "TipoMusica" : "Rock'n'Roll", "Membri" :  
["Paul","John","George","Ringo"]}'
```

41

## Da stringa JSON a oggetto

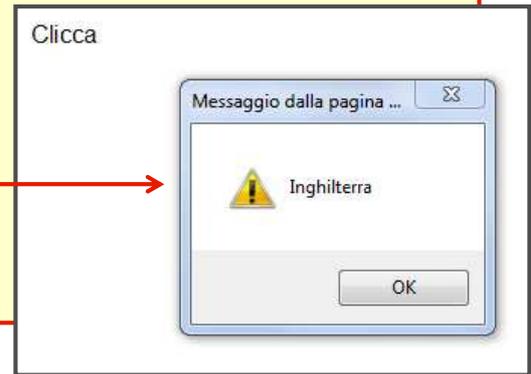
- JavaScript mette a disposizione la funzione `eval()` che invoca il compilatore e traduce la stringa passata come parametro
- La sintassi di JSON è un sottoinsieme di JavaScript: con `eval` possiamo trasformare una stringa JSON in un oggetto
- La sintassi della stringa passata a `eval` deve essere **'(espressione)'**: dobbiamo quindi racchiudere la stringa JSON fra parentesi tonde

```
var s = '{ "Paese" : "Inghilterra",  
"AnnoFormazione" : 1959, "TipoMusica" : "Rock",  
"Membri" : ["Paul","John","George","Ringo"]}';  
  
var o = eval('(' + s + ')');
```

42

## Esempio completo

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">
<html>
  <head>
    <title> Esempio JSON </title>
    <script>
      var s = '{ "Paese" : "Inghilterra", "AnnoFormazione"
: 1959, "TipoMusica" : "Rock", "Membri" :
["Paul","John","George","Ringo"]}' ;
      var o = eval('(' + s + ')');
    </script>
  </head>
  <body>
    <p onclick='alert(o.Paese) '>
      Clicca
    </p>
  </body>
</html>
```



43

## Parser JSON

- `eval()` presenta rischi di sicurezza: la stringa passata come parametro potrebbe contenere codice malevolo
- Di solito si preferisce utilizzare parser appositi che traducono solo oggetti JSON e non espressioni JavaScript di qualunque tipo
- Il più diffuso è quello messo disposizione dal sito [www.json.org](http://www.json.org) (il punto di riferimento su JSON)
- L'indirizzo è <http://www.json.org/json.js>.
- Il parser espone l'oggetto JSON con due metodi:
  - `JSON.parse(strJSON)`: converte una stringa JSON in un oggetto JavaScript
  - `JSON.stringify(objJSON)`: converte un oggetto JavaScript in una stringa JSON.

44

## JSON e Ajax - 1

---

- Sul lato client:
  - Si crea un oggetto JavaScript e si riempiono le sue proprietà con le informazioni necessarie
  - Si usa `JSON.stringify()` per convertire l'oggetto in una stringa JSON
  - Si usa la funzione `encodeURIComponent()` per convertire la stringa in un formato utilizzabile in una richiesta HTTP
  - Si manda la stringa al server mediante `XMLHttpRequest` (la stringa viene passata come variabile con GET o POST)

---

45

## JSON e Ajax - 2

---

- Sul lato server:
  - Si decodifica la stringa JSON e la si trasforma in oggetto Java utilizzando un apposito parser (si trova sempre su [www.json.org](http://www.json.org))
  - Si elabora l'oggetto
  - Si crea un nuovo oggetto Java che contiene i dati della risposta
  - Si trasforma l'oggetto Java in stringa JSON usando il parser
  - Si trasmette la stringa JSON al client nel corpo della risposta HTTP:  
`response.out.write(strJSON);`

---

46

## JSON e Ajax - 3

---

- Sul lato client:
  - Si converte la stringa JSON in un oggetto Javascript usando `JSON.parse()`
  - Si usa liberamente l'oggetto per gli scopi desiderati

## Riassumendo

---

- **Ajax** aggiunge un nuovo elemento al modello ad eventi
- L'uso di **XmlHttpRequest** rappresenta una modalità alternativa per gestire gli eventi remoti
- Abbiamo quindi:
  - Una modalità per gestire gli eventi a **livello locale**
  - Due modalità per gestire gli eventi remoti (**postback** e **XmlHttpRequest**)
- Ci si può spingere molto avanti e utilizzare solo Ajax eliminando i caricamenti di pagina (**Single Page Applications**)
- Nei casi più comuni però Ajax e la modalità di navigazione tradizionale convivono

## Modello ad eventi a due livelli con Ajax

