



# Ajax

**Su Virtuale:**

**Versione 1 pagina per foglio = 2.05.Ajax.pdf**

**Versione 2 pagine per foglio = 2.05.Ajax-2p.pdf**

# Un nuovo modello

---

L'utilizzo di **DHTML** (JavaScript/Eventi + DOM + CSS) delinea un nuovo modello per applicazioni Web

=> *Modello a eventi simile a quello delle applicazioni tradizionali*

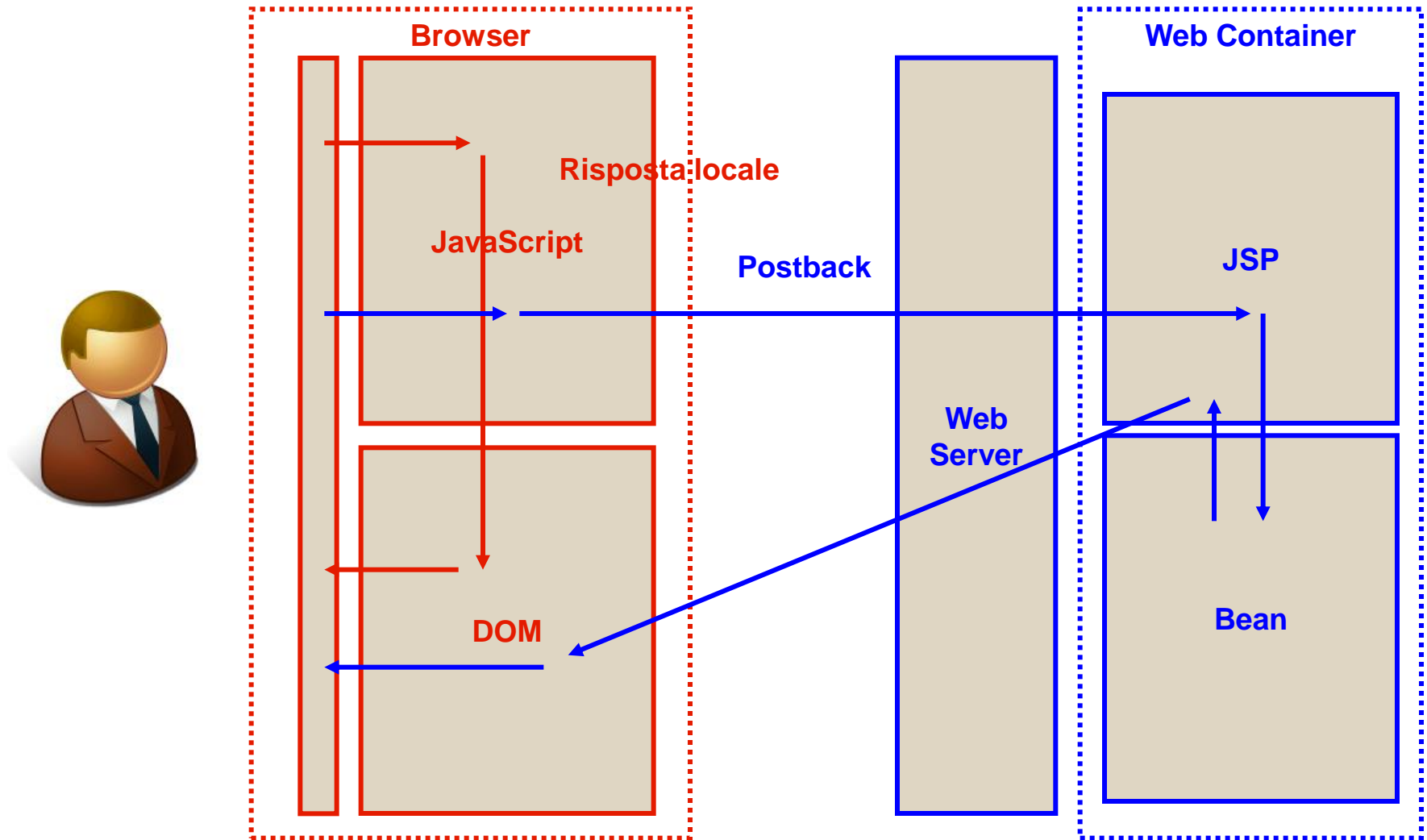
A livello concettuale abbiamo però due livelli di eventi:

- **Eventi locali** che portano ad una modifica diretta DOM da parte di Javascript e quindi a cambiamento locale della pagina
- **Eventi remoti** ottenuti tramite ricaricamento della pagina che viene modificata lato server in base ai parametri passati in GET o POST

Si tratta veramente di eventi remoti?

Il ricaricamento di pagina per rispondere a interazione con l'utente prende il nome di **postback**

# Modello a eventi a due livelli



## Esempio di postback - 1

---

Consideriamo un form in cui compaiono due tendine che servono a selezionare il comune di nascita di una persona

- Una con province
- Una con 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 ▼
-----------------------------------	-----------	---------------------

## Esempio di postback - 2

---

Per realizzare questa interazione si può procedere in questo modo:

- Si crea 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 JSP con parametro provincia impostato al valore scelto dall'utente
- La pagina restituita contiene nella tendina dei comuni l'elenco di quelli che appartengono alla provincia scelta

## Limiti del modello a ricaricamento di pagina

---

Quando lavoriamo con applicazioni desktop *siamo abituati a un elevato livello di interattività:*

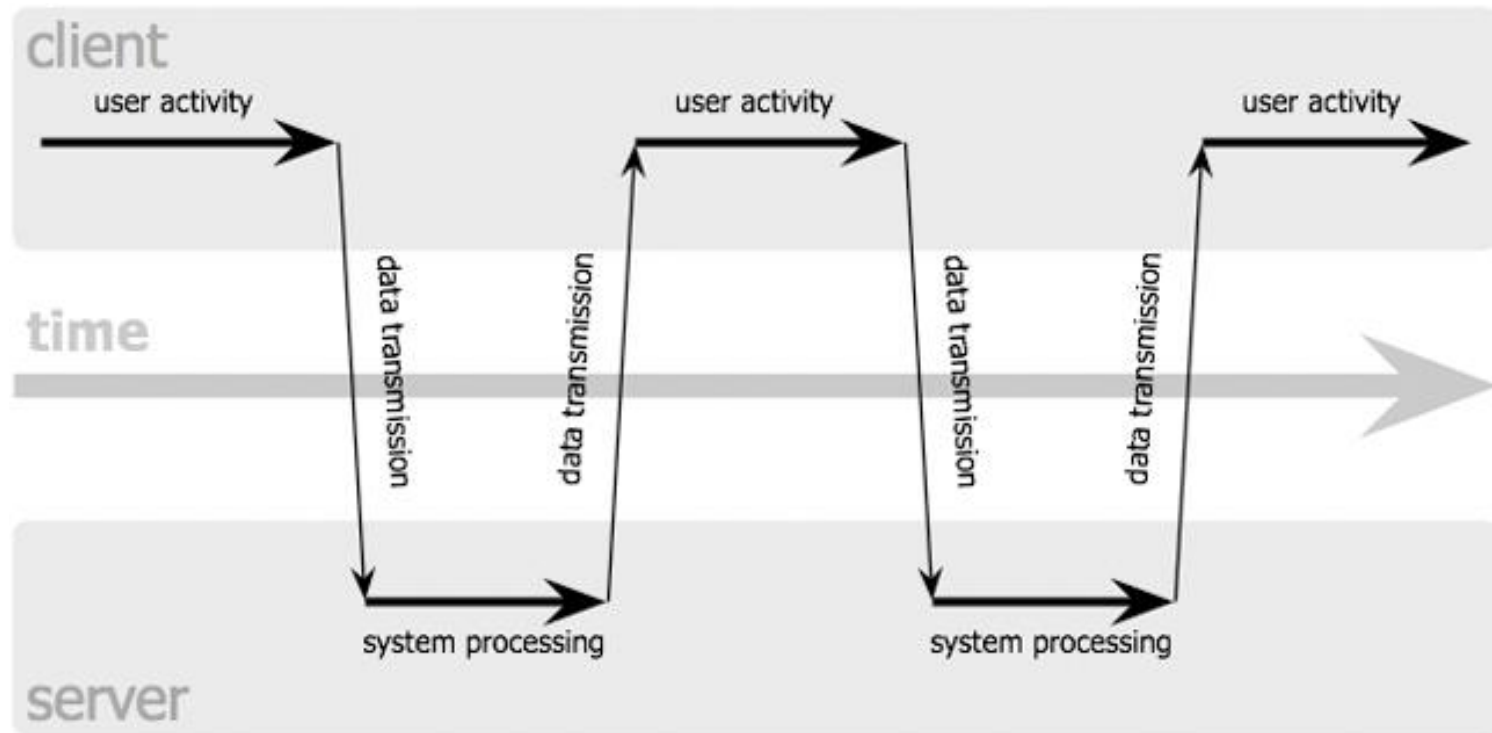
- applicazioni reagiscono in modo rapido e intuitivo ai comandi

*Applicazioni Web tradizionali espongono invece un modello di interazione rigido*

- Modello “**Click, wait, and refresh**”
- È necessario *refresh della pagina da parte del server* per la gestione di qualunque evento (sottomissione di dati tramite form, visita di link per ottenere informazioni di interesse, ...)

➡ È comunque ancora **modello sincrono**: l'utente effettua una richiesta e deve attendere la risposta da parte del server

# Modello di interazione classico



# AJAX e asincronicità

---

Il modello di interazione (tecnologia?) **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
- AJAX non è una nuova tecnologia per se ma è basato su tecnologie standard (già viste all'interno del corso) e combinate insieme per realizzare un *modello di interazione più ricco*:
  - JavaScript
  - DOM
  - XML
  - HTML
  - CSS



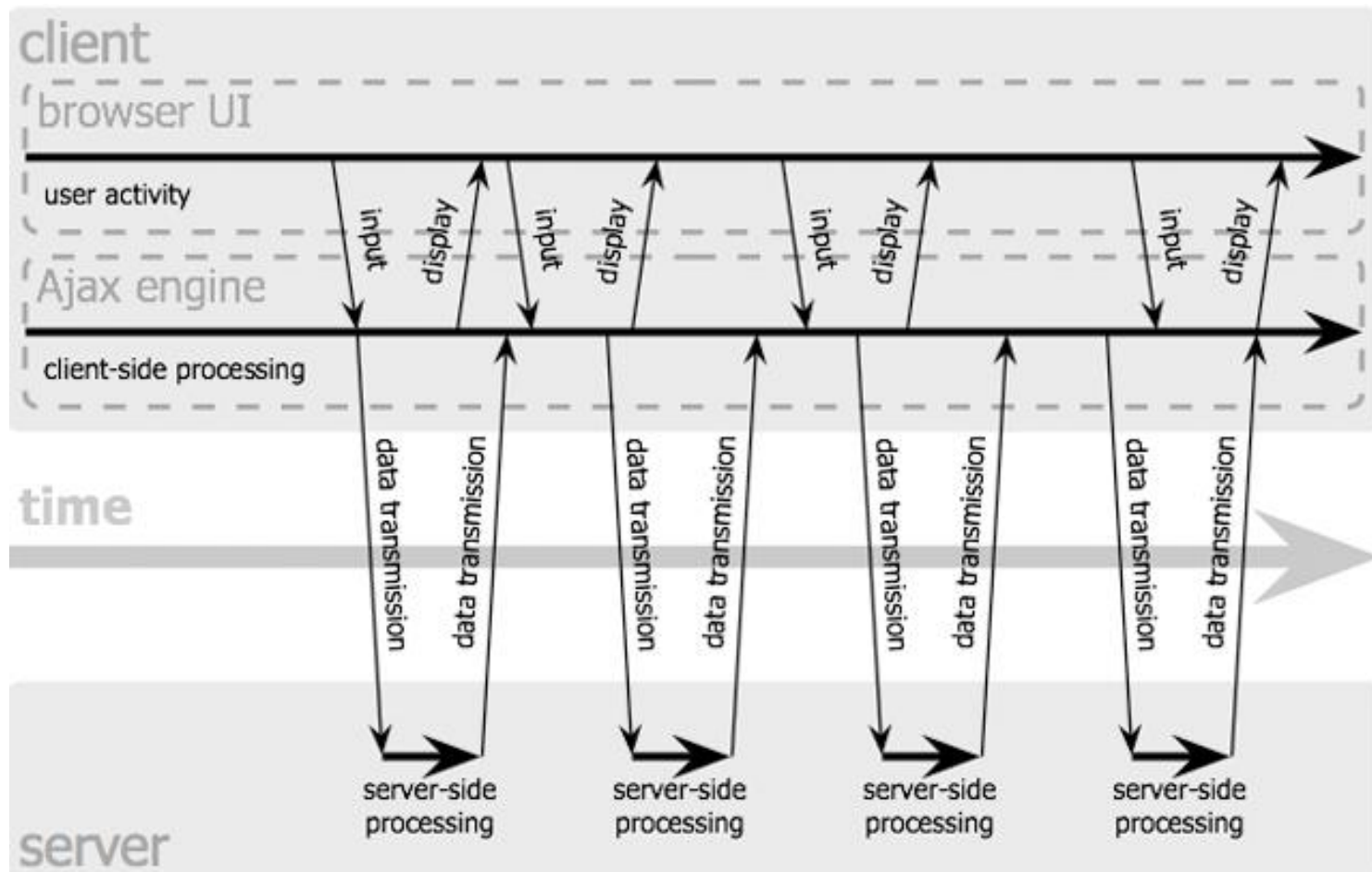
## AJAX e asincronicità

---

AJAX punta a supportare applicazioni user friendly con elevata interattività (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*
- *L'elemento centrale è l'utilizzo dell'oggetto JavaScript **XMLHttpRequest***
  - Consente di ottenere dati dal server senza necessità di ricaricare l'intera pagina
  - Realizza *comunicazione* **asincrona** fra client e server: il client non interrompe interazione con utente anche quando è in attesa di risposte dal server

# Modello di interazione con AJAX



## Tipica sequenza AJAX


---

- Si verifica un evento determinato dall'interazione fra utente e pagina Web
- L'evento comporta l'esecuzione di una funzione JavaScript in cui:
  - *Si istanzia un oggetto* di classe **XMLHttpRequest**
  - Si configura XMLHttpRequest: si associa una funzione di callback, si effettua configurazione, ...
  - *Si effettua 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

# XMLHttpRequest

---

È l'oggetto **XMLHttpRequest** che effettua richiesta di una risorsa via HTTP a server Web

- NON sostituisce URI della propria richiesta all'URI corrente; NON provoca cambio di pagina
- Può inviare info (parametri) sotto forma di variabili (come form)
- *E più risorse richieste concorrentemente? È sufficiente un unico oggetto XMLHttpRequest?*
- Può effettuare sia richieste GET che POST
- Le richieste possono essere di tipo
  - **Sincrono**: blocca 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  *quindi thread dedicato*

## Creazione di un'istanza: dettagli browser-specific

---

I browser recenti 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 «molto» vecchi complica un po' le cose (necessità di accesso *universale* da sistemi legacy long-lived); la esamineremo poi (in fondo ai lucidi) per non rendere difficile la comprensione iniziale del modello

- **Attenzione:** per motivi di sicurezza `XMLHttpRequest` può essere utilizzata solo verso dominio da cui proviene la risorsa che la utilizza

# 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()`

## 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*

## 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 codifica utilizzata nelle richieste POST
- È comunque importante impostare header `connection` di solito al valore `close` (perché secondo voi?)  
VEDRETE POI NEL PROX LAB...

`send(body)` : consente di inviare la richiesta al server

- Non è bloccante in attesa di risposta se il parametro `async` di `open` è stato impostato a `true`. Che cosa succederebbe altrimenti?
- Prende come parametro una stringa che costituisce il body della richiesta HTTP



# Esempi

## GET

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

## POST

```
var xhr = new XMLHttpRequest();  
xhr.open("POST", "pagina.html", true );  
xhr.setRequestHeader("Content-type",  
    "application/x-www-form-urlencoded");  
xhr.setRequestHeader("connection", "close"); //  
xhr.send("p1=v1&p2=v2");
```

Eventualmente  
da eliminare  
perché...

# 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**

## 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 ancora richiamato open()
- **1: open** - è stato invocato il metodo open(), ma send() non ha ancora effettuato l'invio dati
- **2: sent** - metodo send() è stato eseguito e ha effettuato la richiesta
- **3: receiving** – la risposta ha cominciato 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**

## 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 a eventi*
- Occorre registrare una *funzione di callback* che viene *richiamata in modo asincrono* ad ogni cambio di stato della proprietà ReadyState

La sintassi è

```
xhr.onreadystatechange = nomefunzione
```

```
xhr.onreadystatechange = function() {istruzioni}
```

**Attenzione:** per evitare comportamenti imprevedibili l'assegnamento va fatto prima del **send()**

Ovviamente, capite il perché...

## 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 restituito dal server...

**Esempio:**

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

## Proprietà `responseText` e `responseXML`

---

Contengono i dati restituiti dal server

**`responseText`**: stringa che contiene il body della risposta HTTP

- disponibile solo a 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

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

Consentono di leggere gli header HTTP che descrivono la risposta del server

- Sono utilizzabili *solo nella funzione di callback*
- Possono essere *invocati sicuramente in modo safe solo a richiesta conclusa* (`readystate==4`)
- In alcuni browser possono essere invocati 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
- 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`



# 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

## 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 che cosa stia succedendo e possono durare troppo!  
=> L'effetto è un possibile disorientamento dell'utente

Di conseguenza, di solito si agisce su due fronti per limitare i comportamenti impropri a livello utente:

- Rendere visibile in qualche modo l'andamento della chiamata (barre di scorrimento, info utente, ...)
- *Interrompere le richieste che non terminano in tempo utile* per sovraccarichi del server o momentanei problemi di rete (timeout)

## 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, il metodo non viene richiamato; `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

## PIÙ RILEVANTE: aspetti critici per il programmatore

---

- *È accresciuta la complessità delle Web Application*
- *La logica di presentazione è ripartita fra client-side e server-side*
- ***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
- Mancanza di standardizzazione di XMLHttpRequest e assenza di supporto nei vecchi browser

## 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 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 e accediamo agli elementi di nostro interesse
- Per visualizzare i contenuti ricevuti modifichiamo il DOM della pagina HTML

## 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

## Esempio - 2

---

Ipotizziamo che i dati sui contatti siano contenuti in un database. Il server:

- riceve una request con l'identificativo della persona
- legge da file system o 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>
```

## 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("companymname").innerHTML=
            compName;
        ...
    }
}
```



# XML è la scelta giusta?

---

(secondo un'interpretazione molto comune la X di AJAX sta per XML)

**Abbiamo però visto nell'esempio precedente che utilizzo di XML come formato di scambio fra client e server porta a generazione e utilizzo di quantità di byte piuttosto elevate e non ottimizzate**

- Non semplicissimo 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; questo formato è nella pratica industriale quello più utilizzato oggi – lo vedremo nel dettaglio fra poco

## **Richieste Ajax concorrenti dello stesso cliente?**

---

Quali vantaggi e svantaggi, per richiesta onerosa lato servitore, fra:

- **Gestione manuale thread lato servitore**
- **Invio concorrente di richieste Ajax, distinte e indipendenti**

**Aperta la discussione 😊...**

**Attenzione a pause attive lato servitore e a «rompere» il classico modello request-response del Web...**

**JSON** è l'acronimo di **J**ava**S**cript **O**bject **N**otation

- Formato per lo scambio di dati, considerato molto più comodo di XML
  - Leggero in termini di quantità di dati scambiati
  - Molto semplice ed efficiente da elaborare da parte del supporto runtime al linguaggio di programmazione (in particolare per JavaScript)
  - Ragionevolmente semplice da leggere per operatore umano
- È 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

## Oggetti e costanti oggetto

---

In Javascript è possibile creare un oggetto in base a 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 = "Inghilterra";  
Beatles.AnnoFormazione = 1959;  
Beatles.TipoMusica = "Rock";
```

## 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"]  
}
```

## Array di oggetti

---

È infine possibile definire array di oggetti:

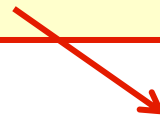
```
var Rockbands = [  
  {  
    "Nome" : "Beatles",  
    "Paese" : "Inghilterra",  
    "AnnoFormazione" : 1959,  
    "TipoMusica" : "Rock",  
    "Membri" : ["Paul", "John", "George", "Ringo"]  
  },  
  {  
    "Nome" : "Rolling Stones",  
    "Paese" : "Inghilterra",  
    "AnnoFormazione" : 1962,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

# La sintassi JSON

- La sintassi JSON si basa su quella delle costanti oggetto e array di JavaScript
- Un “oggetto JSON” altro non è che una stringa equivalente a una costante oggetto di JavaScript

Costante oggetto Javascript

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



Oggetto  
JSON

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

## Da stringa JSON a oggetto

---

JavaScript mette a disposizione la funzione `eval()` che invoca l'interprete per la traduzione della 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

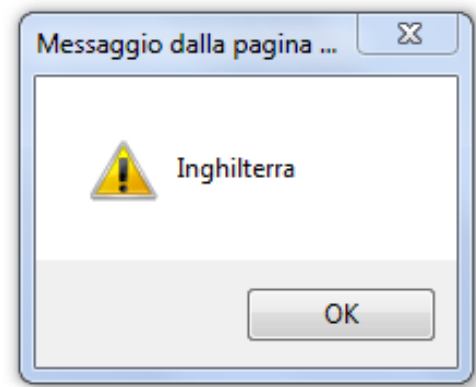
```
var s = '{ "Paese" : "Inghilterra",  
"AnnoFormazione" : 1959, "TipoMusica" : "Rock",  
"Membri" : ["Paul", "John", "George", "Ringo"] }';  
  
var o = eval('(' + s + ')');
```



# 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>
```

Clicca



# Parser JSON

---

Uso di `eval()` presenta rischi: 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

Alcuni parser molto diffusi:

- **Google GSON** - <https://github.com/google/gson>
- «più old-fashioned» **jabsorb** - <https://github.com/Servoy/jabsorb>

Ad esempio il parser **jabsorb** 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

Ad esempio in una interazione client-server in cui il cliente vuole trasferire un oggetto JSON,

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 stringa JSON
- Si usa la funzione `encodeURIComponent()` per convertire la stringa in un formato utilizzabile in una richiesta HTTP (vedi esercitazione su AJAX)
- Si manda la stringa al server mediante `XMLHttpRequest` (stringa viene passata come variabile con GET o POST)

### 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); ne parleremo anche nella esercitazione di lab dedicata)
- Si elabora l'oggetto
- Si crea un nuovo oggetto Java che contiene dati della risposta
- Si trasforma l'oggetto Java in stringa JSON usando il parser suddetto
- Si trasmette la stringa JSON al client nel corpo della risposta HTTP:  
`response.out.write(strJSON) ;`

Sul lato client, all'atto della ricezione:

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

# Parsing JSON in Java: Gson (al posto di jabsorb)

---

**Gson è una libreria java per il parsing/deparsing di oggetti JSON**

È stata realizzata da Google per l'esclusivo sviluppo di prodotti interni, ora Gson è libreria open source

Libreria molto potente e largamente utilizzata sia in ambito accademico che industriale. **Alcuni punti di forza:**

- Fornisce dei metodi semplici e facili da usare per «conversione» Java-JSON e viceversa
- Genera output JSON compatti e leggibili
- Consente rappresentazioni custom per gli oggetti
- Consente la conversione da/a JSON di oggetti Java immutabili pre-esistenti (non occorre modificare sorgente)
- Supporta oggetti di complessità arbitraria

Altre info utili/tutorial/guide e link per il download sono disponibili qui:

<https://github.com/google/gson>

---

# Gson in una slide

---

In versioni recenti di Gson (a partire da 2.8):

Inizializzazione dell'oggetto Gson:

```
Gson g = new Gson();
```

Serializzazione di un oggetto:

```
Person santa = new Person("Santa", "Claus", 1000);  
g.toJson(santa);
```

Deserializzazione di un oggetto:

```
Person peterPan = g.fromJson(json, Person.class);
```

Tutto qui... o quasi...

Trovate molti altri esempi, API complete, JavaDoc e tutorial al link nella slide precedente

## Riassumendo

---

- AJAX consente gestione asincrona
- AJAX aggiunge un nuovo elemento al modello a eventi
- L'uso di **XmlHttpRequest** rappresenta una modalità alternativa per gestire gli eventi remoti

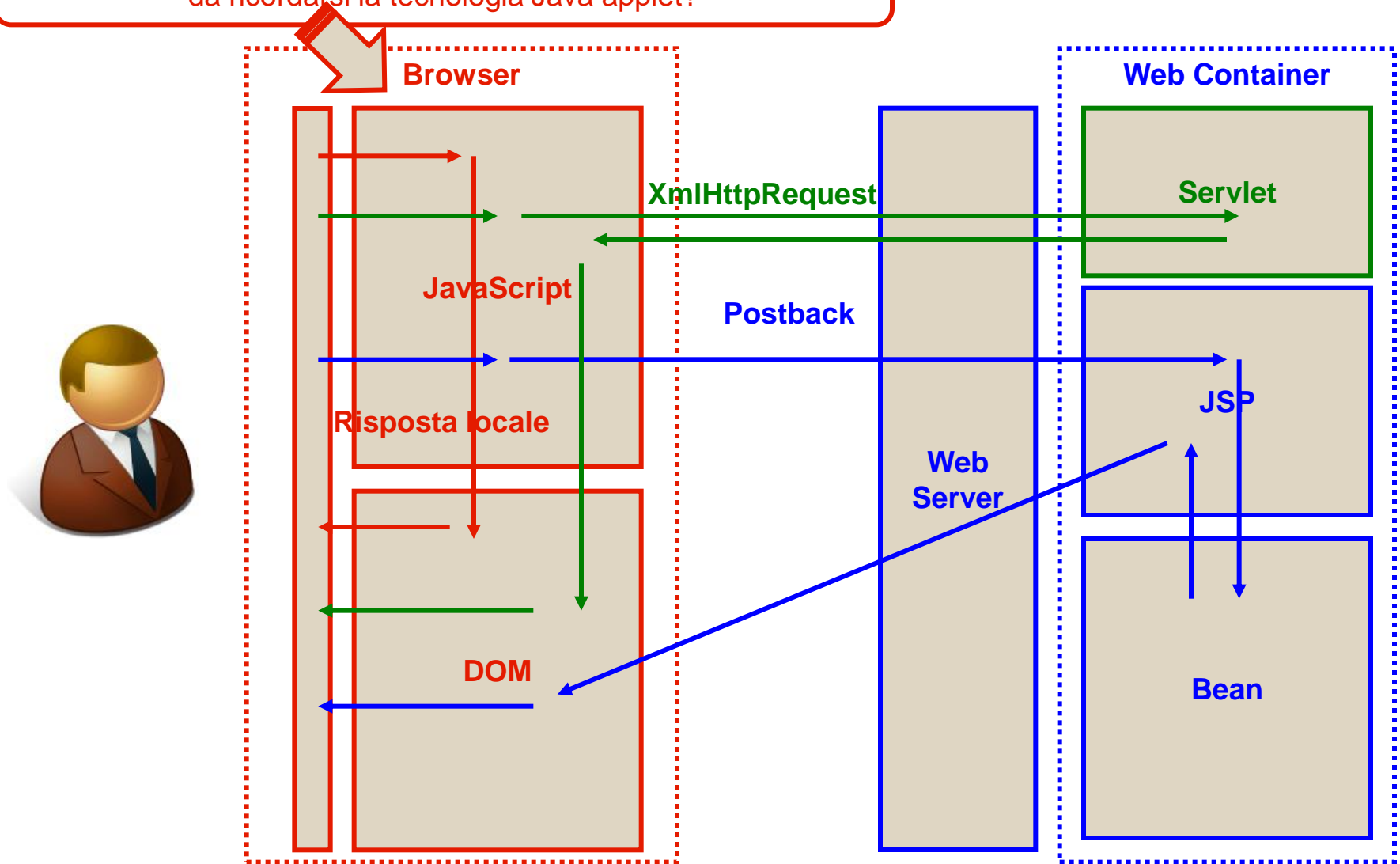
Abbiamo quindi:

- Una modalità per gestire gli **eventi a livello locale**
  - Due modalità per gestire gli eventi remoti (**postback** e **XmlHttpRequest**)
- Si può adottare la tecnologia in modo più “radicale” 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 eventi a due livelli con AJAX

i più anziani fra voi sono sufficientem. anziani ☺  
da ricordarsi la tecnologia Java applet?



## Appendice: gestione della compatibilità

---

- Tutti i browser recenti supportano XMLHttpRequest come oggetto nativo (Firefox, Opera 7+, Safari, Internet Explorer da versione 8):

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

- Versioni precedenti di IE 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")
```

- Se si vuole proprio essere compatibili con ogni versione di browser ancora installata...

# Gestione della compatibilità

Ad esempio, nella pratica industriale 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 più 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;
}
```