CORSO DI LAUREA IN INFORMATICA

Tecnologie Software per il Web

AJAX & JSON

a.a. 2020-2021

AJAX: **A**synchronous **J**avaScript **A**nd **X**ml

AJAX is a developer's dream, because you can:

Update a web page without reloading the page
Request data from a server - after the page has loaded
Receive data from a server - after the page has loaded
Send data to a server - in the background

Un nuovo modello

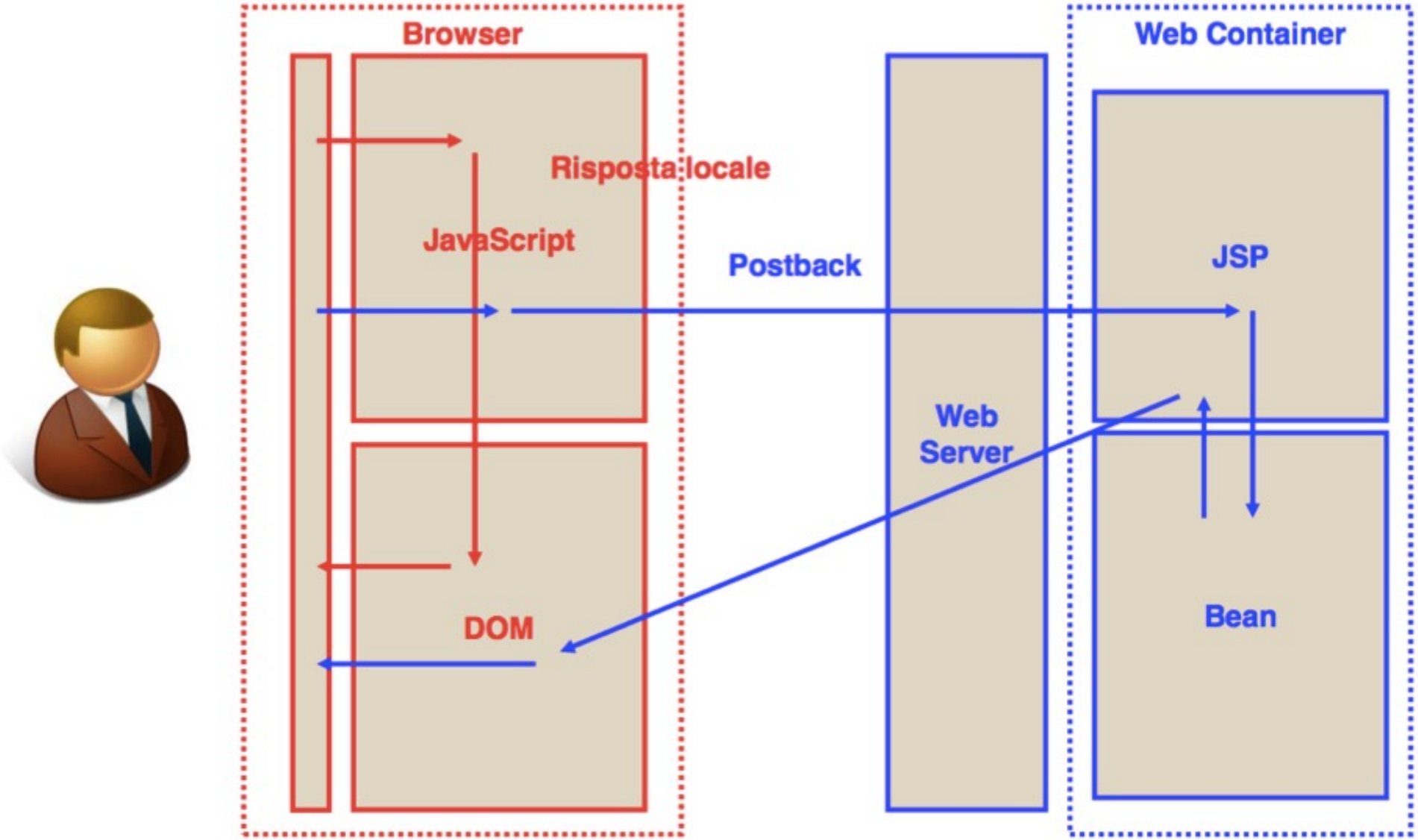
- L'utilizzo di **DHTML** (JavaScript/Eventi + DOM + CSS) delinea un nuovo modello per applicazioni Web

  =>

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

Modello a eventi a due livelli



Esempio di postback

- 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

Esempio di postback (2)

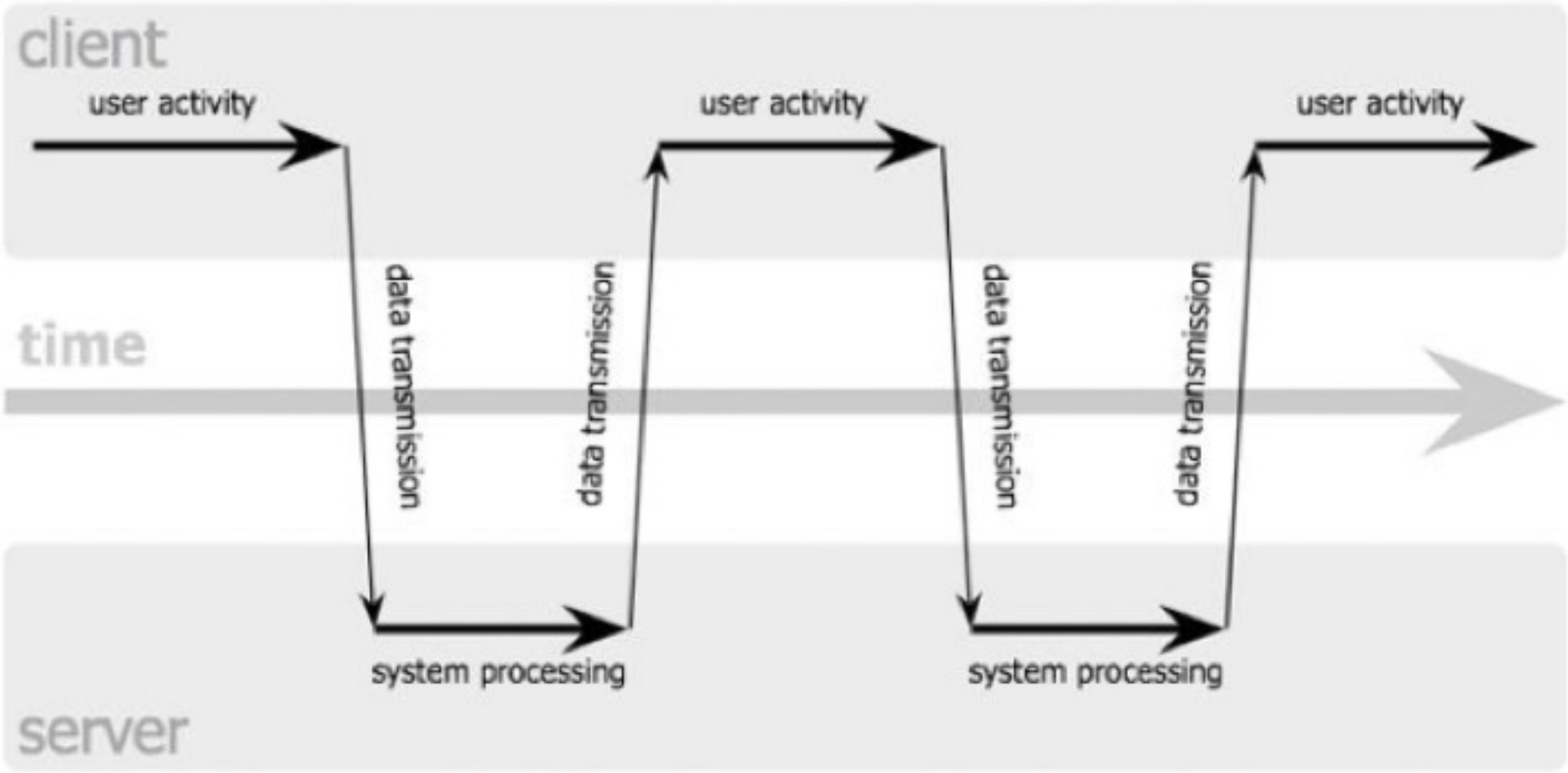
- Per realizzare questa interazione si può procedere in questo modo:
  1. Si crea una JSP che inserisce nella tendina dei comuni l'elenco di quelli che appartengono alla provincia passata come parametro
  2. Si definisce un evento **onChange** collegato all'elemento **select** delle province
  3. Lo 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 ancora un **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

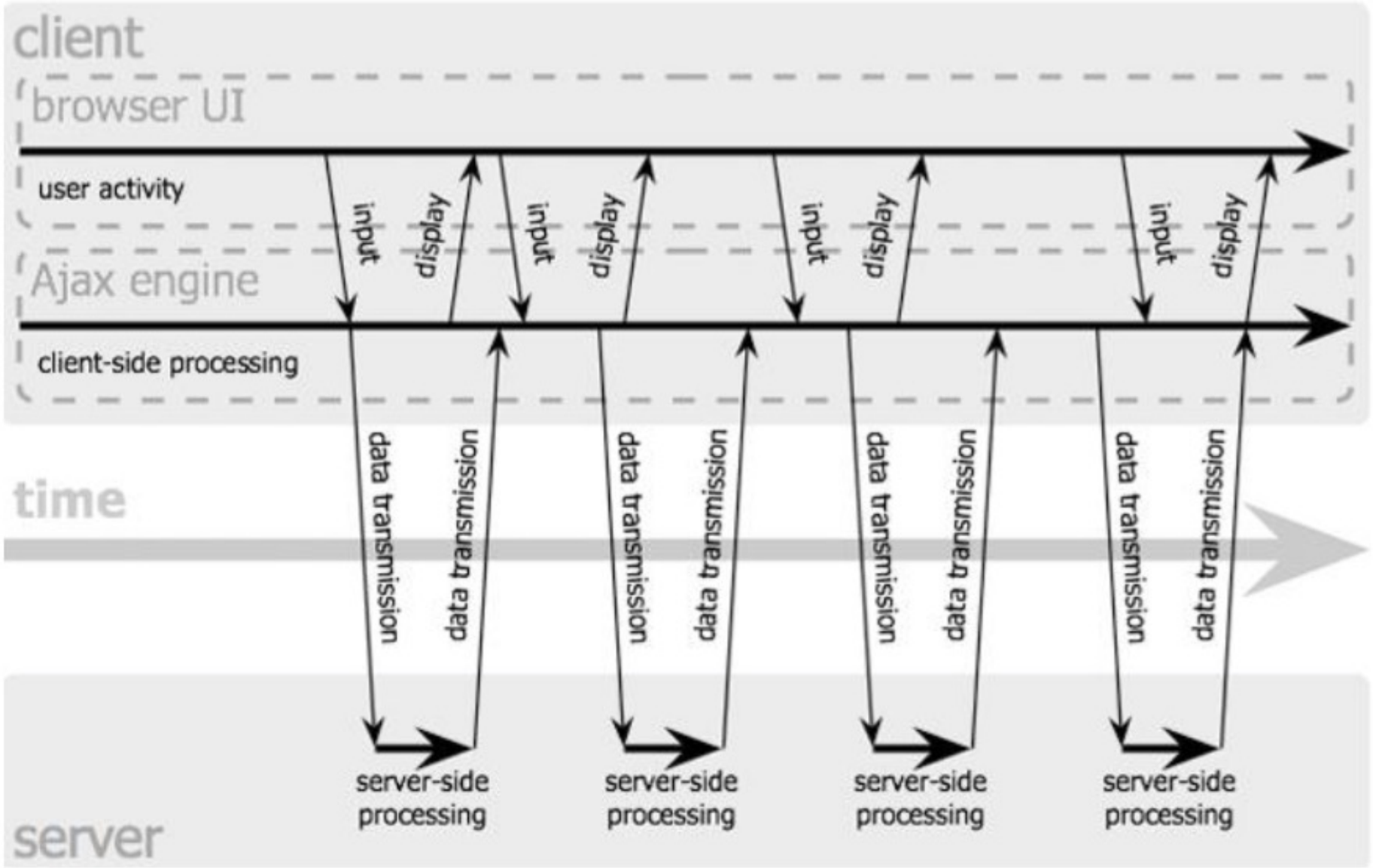
Asynchronous **J**avascript **A**nd **X**ml

- È basato su tecnologie standard 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 una 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



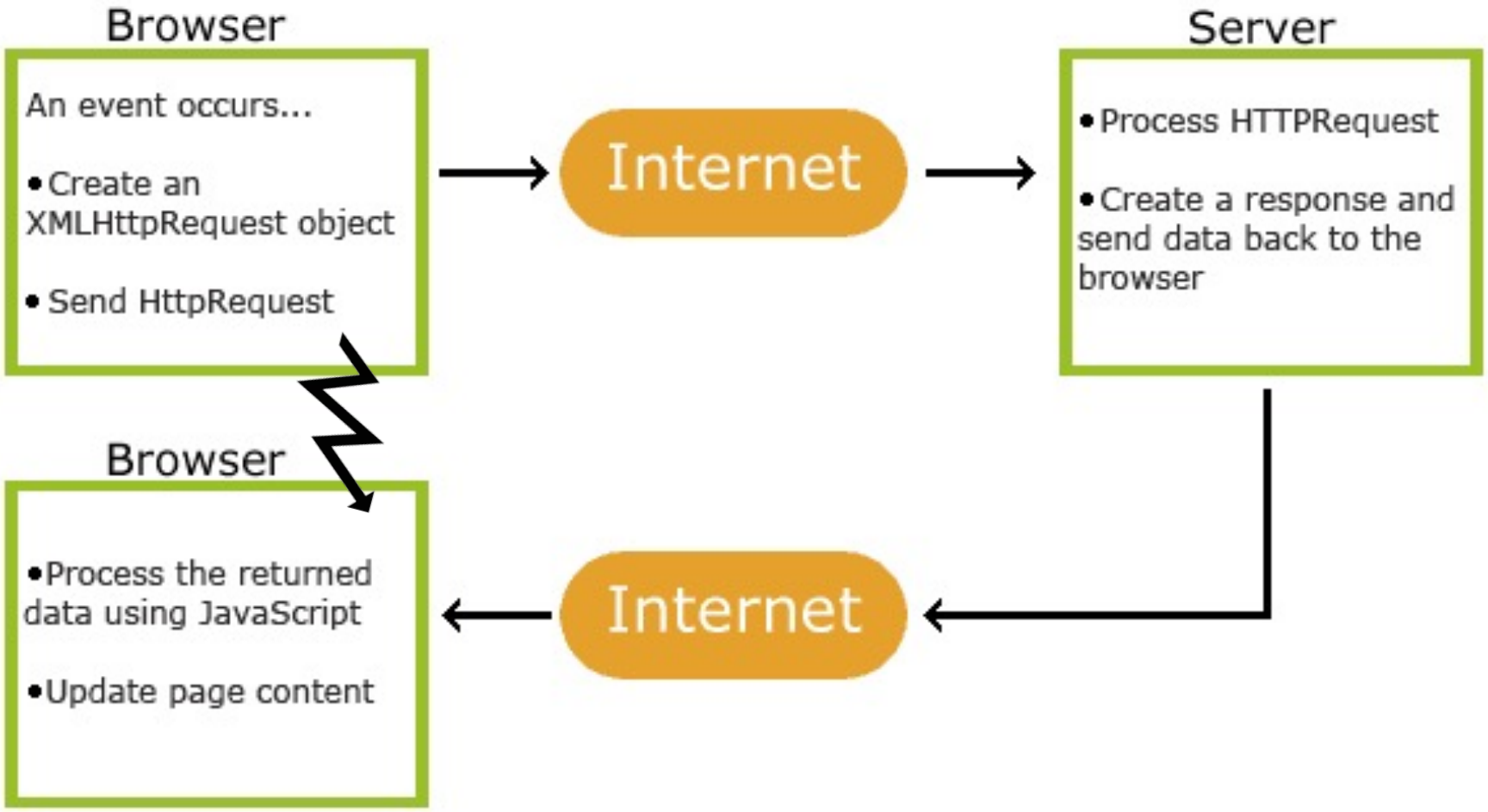
Tipologie di interazioni in AJAX

- **Semplici**
  - **modifica del valore dell'attributo innerHTML** di un elemento della pagina, accesso ai contenuti di uno *span*, *div*, di un *p*, ecc…
  - possibile assegnare non solo testo semplice, ma altro HTML!
  - **uso del DOM** per aggiungere, popolare, o modificare elementi *getElementById()* o *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 *setIntervall() / clearInterval()* o *setTimeout()*

Tipica sequenza AJAX

1. Si verifica un evento determinato dall'interazione fra utente e pagina Web
2. 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 una configurazione, …
   - Si effettua chiamata asincrona al server
3. Il server elabora la richiesta e risponde al client
4. Il browser invoca la funzione di **callback** che:
   - elabora il risultato
   - aggiorna il DOM della pagina per mostrare i risultati dell'elaborazione

Ricapitolando…



Google suggest

- *AJAX was made popular in 2005 by Google, with Google Suggest…*

- Google Suggest is using AJAX to create a very dynamic web interface: When you start typing in Google's search box, a JavaScript sends the letters off to a server and the server returns a list of suggestions

XMLHttpRequest

- È l'oggetto **XMLHttpRequest** che si occupa di effettuare la 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 eventuali informazioni (parametri) sotto forma di variabili (come una form)
  - E più risorse richieste concorrentemente?
- 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 JavasScript ne le operazioni dell'utente sulla pagina (**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 pò le cose (necessità di accesso universale da sistemi legacy long-lived); la esamineremo in seguito per non rendere difficile la comprensione del modello*

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 la codifica utilizzata nelle richieste POST
  - È comunque importante impostare header **connection** al valore **close** (*"close" connection option for the sender to signal that the connection will be closed after completion of the response*)
- **send(body)** consente di inviare la richiesta al server
  - Non è bloccante 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",
      "x-www-form-urlencoded");
xhr.setRequestHeader("connection","close");
xhr.send("p1=v1&p2=v2");
```

Create a new XMLHttpRequest (gestione della compatibilità)

```
function getXmlHttpRequest() {

    var xhr = false;
    var activeXoptions = new Array("Microsoft.XmlHttp", "MSXML4.XmlHttp",
            "MSXML3.XmlHttp", "MSXML2.XmlHttp", "MSXML.XmlHttp");

    try {
        xhr = new XMLHttpRequest();
    } catch (e) {
    }

    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;
}
```

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()***

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, 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() e 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**

Esempio

https://www.w3schools.com/xml/tryit.asp?filename=tryajax_first

```
<!DOCTYPE html>
<html>
<body>

<h2>AJAX</h2>

<button type="button" onclick="loadDoc()">Request data</button>

<p id="demo"></p>

<script>
function loadDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (xhttp.readyState == 4 && xhttp.status == 200) {
      document.getElementById("demo").innerHTML = xhttp.responseText;
    }
  };
  xhttp.open("GET", "demo_get2.asp?fname=Pow&lname=Mal", true);
  xhttp.send();
}
</script>

</body>
</html>
```

Example: handler as a function

```
function wait(state)
{
    if(state == true) {
        //Show wait
    }
    else {
        //Hide wait
    }
}
```

```
function getReadyStateHandler(req, responseXmlHandler) {
    return function() {
        if (req.readyState == 1) {
        } else if (req.readyState == 4) {
            if (req.status == 200 || req.status == 304) {
                responseXmlHandler(req.responseXML);
            } else {
                // window.alert("HTTP error " + req.status + ": " + req.statusText);
            }
        } else {
            wait(false);
        }
    };
}
```

Example: ajax call (ajax.zip)

```
function ajaxCall(id, url, callback, parameter) {
    var req = getXmlHttpRequest();
    try {
        wait(true);

        req.onreadystatechange = getReadyStateHandler(req, callback);
        req.open('POST', url, true);
        req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
        req.send("id=" + id + "&param=" + parameter);
    } catch (e1) {
        wait(false);
    }
}
```

Example: servlet

```
@WebServlet("/ServletResponse")
public class ServletResponse extends HttpServlet {

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        response.setContentType("text/xml");

        StringBuffer packed = new StringBuffer();
        packed.append("<info>");

        String id = (String)request.getParameter("id");
        if(id != null) {
            packed.append("<id>");
            packed.append(id);
            packed.append("</id>");
        }

        String param = (String)request.getParameter("param");
        if(param != null) {
            packed.append("<person>");
            packed.append(param);
            packed.append("</person>");
        }

        packed.append("</info>");

        try {
            Thread.sleep(1000);
        } catch(InterruptedException e) {}

        response.getWriter().write(packed.toString());
    }
```

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

Example: html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>AJAX Call</title>
<script type="text/javascript" src="ajax.js"></script>

<script>
function displayResults(listXML) {
    try {
        var ids = listXML.getElementsByTagName("id")[0].firstChild.nodeValue;
        var obj = document.getElementById(ids);
        if(obj != null) {
            var rdfs = listXML.getElementsByTagName("person")[0].firstChild.nodeValue;
            obj.innerHTML =rdfs;
        }
    } catch(e1) {
    }

}
</script>

</head>
<body>

<h3>Perform an AJAX call</h3>

<div id="result">...</div>
<br>
<input type="button" onclick="ajaxCall('result','/ajax/ServletResponse',displayResults,'Michele Risi');" value="Call">
<br>
<div id="wait"></div>
</body>
</html>
```

Example: simple wait

```
function wait(state) {
    if (state == true) {
        // Show wait
        var obj = document.getElementById("wait");
        if(obj != null) {
            obj.style.background = "red";
            obj.style.display = "inline";
            obj.innerHTML = "loading...";
        }
    } else {
        // Hide wait
        var obj = document.getElementById("wait");
        if(obj != null) {
            obj.innerHTML = "";
        }
    }
}
```

obj.style.display = "none";

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
  - 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("companyname").innerHTML=
      compName;
    ...
  }
}
```

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

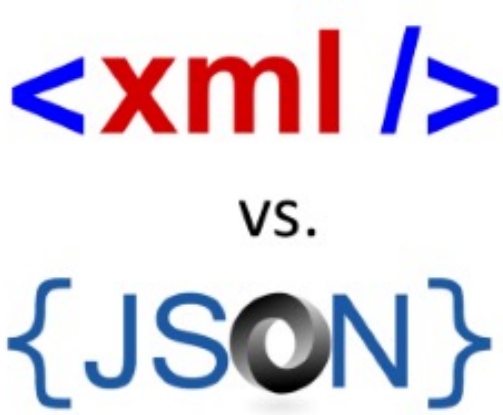
Il metodo abort()

- **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

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 (ad es. Jquery, Mootools, Scriptacolus, Prototype, ...)
- Mancanza di standardizzazione di XMLHttpRequest e assenza di supporto nei vecchi browser



XML VS. JSON

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 generazione e all'utilizzo di quantità di byte piuttosto elevate e non ottimizzate*
  - Difficile da leggere e da manutenere
  - 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 SI; questo formato è nella pratica industriale quello più utilizzato oggi

JSON

- JSON è l'acronimo di JavaScript Object Notation
  - 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 un 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 = "England";
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 (old)

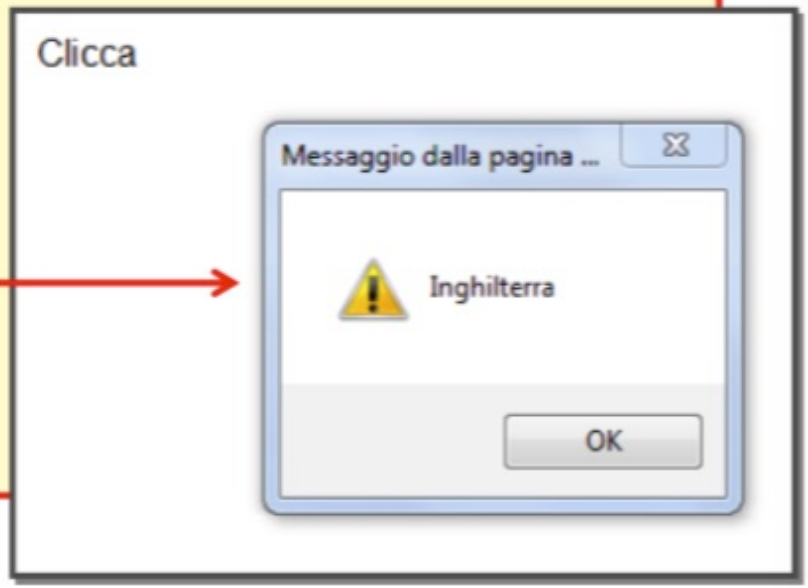
- JavaScript mette a disposizione la funzione **eval()** che invoca l'interprete per la traduzione della stringa passata come parametro
- La sintassi di JSON è un sottoinsieme di JavaScript: *con eval possiamo trasformare una stringa JSON in un oggetto*
- La sintassi della stringa passata a eval deve 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 eval

```
<!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>
```



Da stringa JSON a oggetto (new)

- **eval()** è usata dai vecchi browser
- Più recentemente si usa **JSON.parse(text)** per convertire un oggetto JSON in un oggetto JavaScript

```
var obj = JSON.parse(text);
```

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
- Il più diffuso è quello messo disposizione dal sito **www.json.org** (*il punto di riferimento su JSON*)
- 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

Example: parse (simpleJSON.html)

```
<!DOCTYPE html>
<html>
<body>

<h2>JSON Object Creation in JavaScript</h2>

<p id="demo"></p>

<script>
var text = '{"name":"John Johnson","street":"Oslo West 16",
    "phone":"555 1234567"}';

var obj = JSON.parse(text);

document.getElementById("demo").innerHTML =
obj.name + "<br>" +
obj.street + "<br>" +
obj.phone;
</script>

</body>
</html>
```

converts a JSON text into a JavaScript object

JSON e AJAX

- Ad esempio, in una interazione client-server in cui il cliente vuole trasferire un oggetto JSON
- Sul lato client:
  1. Si crea un oggetto JavaScript e si riempiono le sue proprietà con le informazioni necessarie
  2. Si usa **JSON.stringify()** per convertire l'oggetto in stringa JSON
  3. Si usa la funzione **encodeURIComponent()** per convertire la stringa in un formato utilizzabile in una richiesta HTTP
  4. Si manda la stringa al server mediante **XMLHttpRequest** (*stringa viene passata come variabile con GET o POST*)

JSON e AJAX (2)

- Sul lato server:
  1. Si decodifica la stringa JSON e la si trasforma in oggetto Java utilizzando un apposito **parser** (*si trova su www.json.org*)
  2. Si elabora l'oggetto
  3. Si crea un nuovo oggetto Java che contiene dati della risposta
  4. Si trasforma l'oggetto Java in stringa JSON usando il parser suddetto
  5. Si trasmette la stringa JSON al client nel corpo della risposta HTTP:

response.out.write(strJSON);

content type: **application/json**

JSON e AJAX (3)

- Sul lato client, all'atto della ricezione:
  1. Si converte la stringa JSON in un oggetto JavaScript usando **JSON.parse()**
  2. Si usa liberamente l'oggetto per gli scopi desiderati …

jQuery and AJAX

- ***Without jQuery, AJAX coding can be a bit tricky!***
- Writing regular AJAX code can be a bit tricky, because different browsers have different syntax for AJAX implementation. This means that you will have to write extra code to test for different browsers. However, the jQuery team has taken care of this for us, so that we can write AJAX functionality with only one single line of code

```
$.ajax({
        "type" : "POST",
        "url" : "ajax.php",
        "data" : ”param1=abc&param2=123",
        "success" : function(data) {
                $("#bar").css("background", "yellow").html(data);
        },
        “error” : function (xhr, ajaxOptions, thrownError) {
                alert(xhr.status);
                alert(thrownError);
        }
});
```

Specificare opzioni di default

- $.ajaxSetup() consente di specificare le opzioni comuni a tutte le chiamate:

```
$.ajaxSetup({
        "type": "POST",
        "url": "ajax.php",
        "success": function(data) {
                $("#bar") .css("background", "yellow") .html(data);
        }
});
```

- E poi effettuare la chiamata:

```
$.ajax({
        "data": { "var1":"abc", "var2":"123" }
});
```

jQuery latest version…

```
$.ajax({
    url: "ajax.php",
    method: "POST"
  })
  .done(function( msg ) {
      $( "#log" ).html( msg );
  })
  .fail(function( xhr, textStatus ) {
    alert( "Request failed: " + xhr.status + "  " + textStatus );
  })
  .always(function() {
    alert( "complete" );
  });
```

Load

- The jQuery **load()** method is a simple, but powerful AJAX method
- The load() method loads data from a server and puts the returned data into the selected element

- **Syntax:**

  $(*selector*).load(*URL, data, callback*);

- The required URL parameter specifies the URL you wish to load

Example

```
<!DOCTYPE html>
<html>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.2/jquery.min.
js"></script>
<script>
$(document).ready(function(){
    $("button").click(function(){
        $("#div1").load("demo_test.txt");
    });
});
</script>
</head>
<body>

<div id="div1"><h2>Let jQuery AJAX Change This Text</h2></div>

<button>Get External Content</button>

</body>
</html>
```

Example

- The following example displays an alert box after the load() method completes. If the load() method has succeeded, it displays "External content loaded successfully!", and if it fails it displays an error message:

```
$("button").click(function(){
    $("#div1").load("demo_test.txt", function(responseTxt, statusTxt, xhr){
        if(statusTxt == "success")
            alert("External content loaded successfully!");
        if(statusTxt == "error")
            alert("Error: " + xhr.status + ": " + xhr.statusText);
    });
});
```

$.ajax shortcut: $.get() and $.post()

- Chiamata GET senza parametri:

```
$.get("ajax.php", function(data) {
                $("#bar").css("background", "yellow").html(data);
});
```

- Chiamata POST con parametro:

```
$.post("ajax.php", {"var1":"abc"}, function(data) {
                $("#bar") .css("background", "yellow") .html(data);
});
```

$.getJSON()

- Supponiamo di avere una pagina json.php come segue:

```
<?php echo '{"var1":"abc", "var2":"123"}'; ?>
```

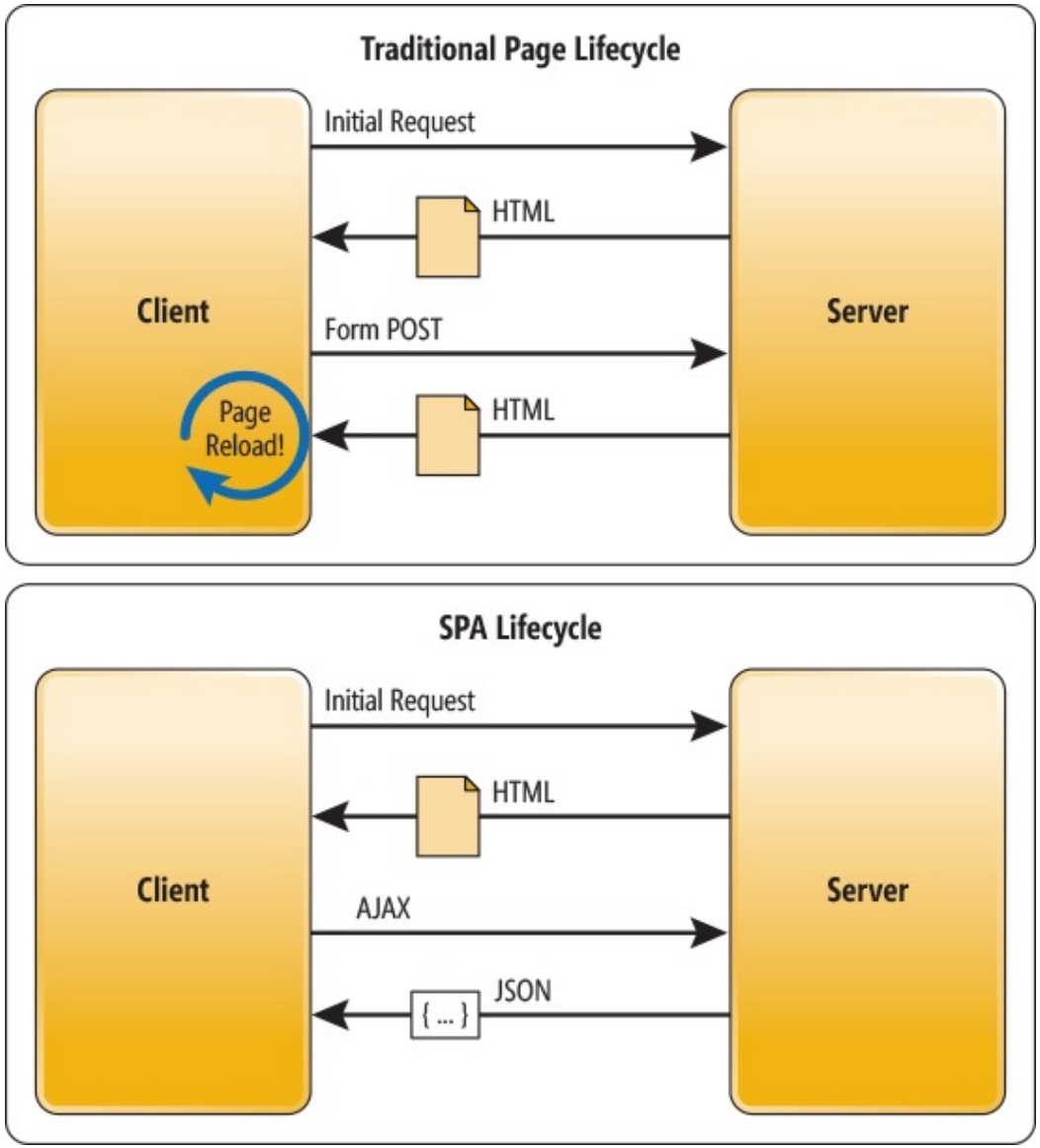
- Possiamo utilizzare il seguente codice per recuperare i dati in formato JSON:

```
$.getJSON("json.php", function(data){
        $("#bar").css("background", "yellow")
        .html(data.var1 + ", " + data.var2);
});
```

Single Page Application

- A single-page application (SPA) is a web application or website that interacts with the web browser by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages
- The goal is faster transitions that make the website feel more like a native app

Single Page Application (2)



Riferimenti

- AJAX introduction
  - **https://www.w3schools.com/xml/ajax_intro.asp**

- jQuery:
  - **http://api.jquery.com/**