CORSO DI LAUREA IN INFORMATICA

Tecnologie Software per il Web

AJAX & JSON

a.a. 2021-2022

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

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

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

https://www.w3schools.com/whatis/whatis_ajax.asp

Un nuovo modello

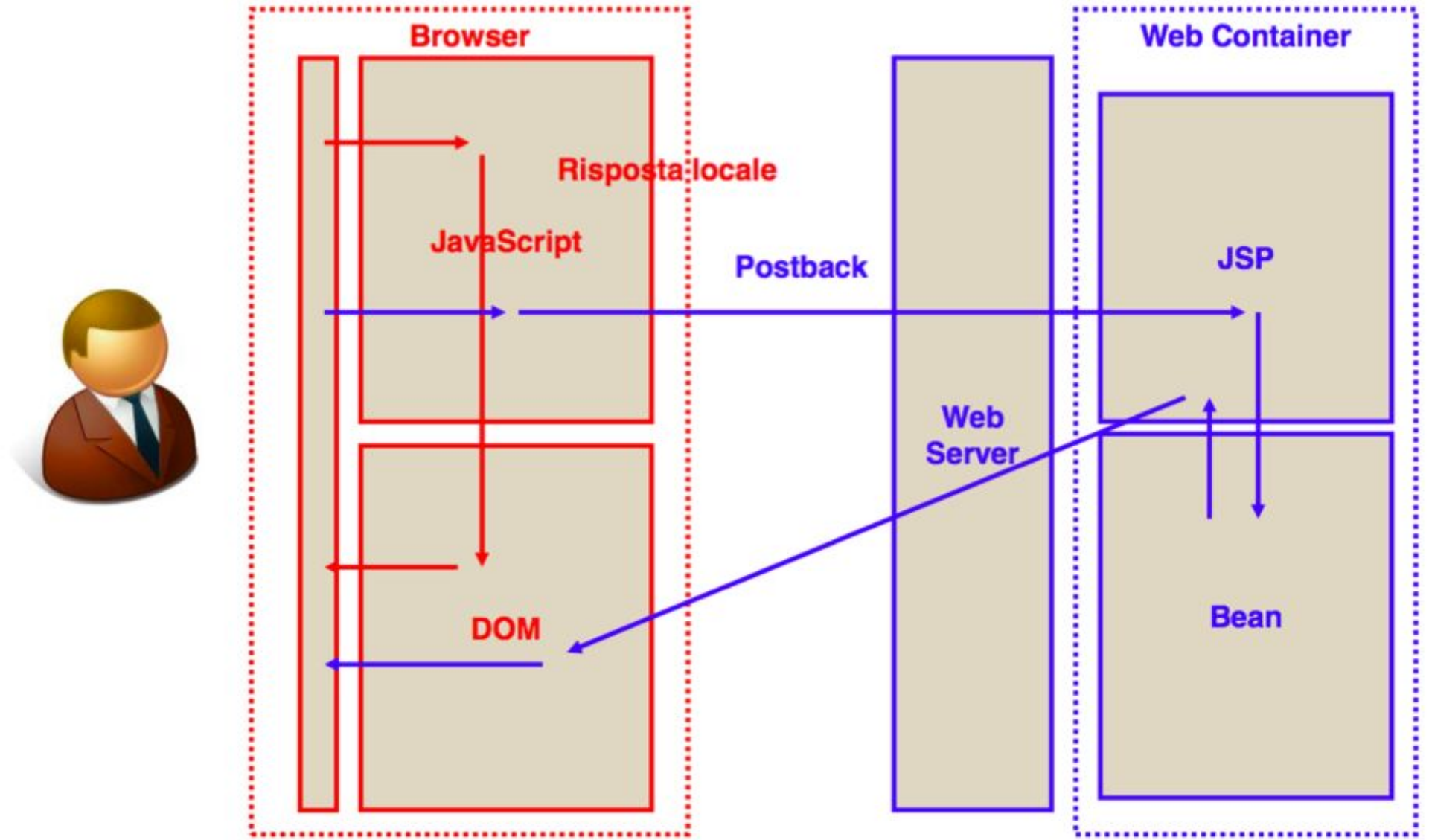
- L'utilizzo di **DHTML** (JavaScript/Eventi + DOM + CSS) delinea un nuovo modello per applicazioni Web

  =>

  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 cambiamenti locali 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)

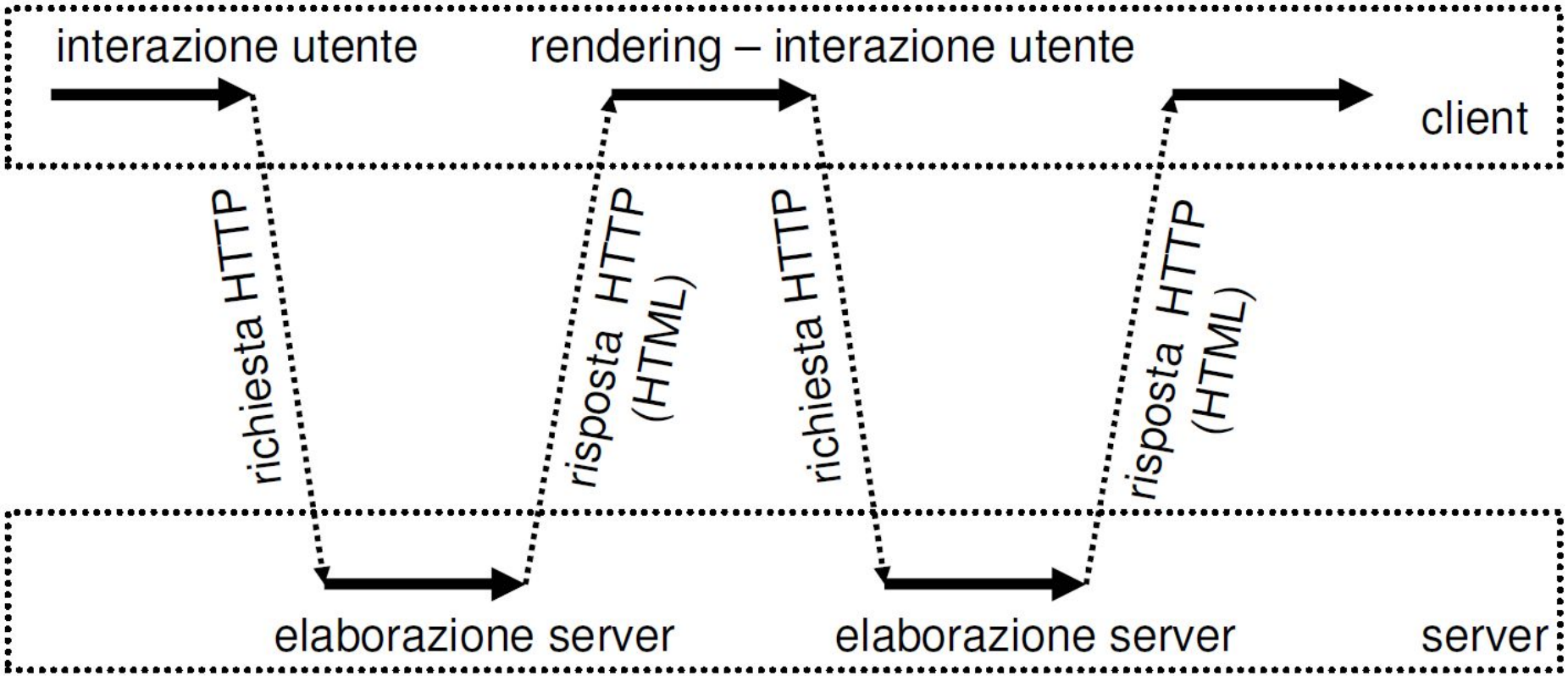
- Per realizzare questa interazione si può procedere in questo modo:
  1. Si crea una JSP che inserisce nella tendina dei comuni l'elenco di quelli che appartengono alla provincia passata come parametro
  2. Si definisce un evento **onChange** collegato all'elemento **select** delle province
  3. Lo script collegato ad onChange forza il ricaricamento della pagina con una richiesta get/post (**postback**)

- Quindi:
  - L'utente sceglie una provincia
  - Viene invocata la 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

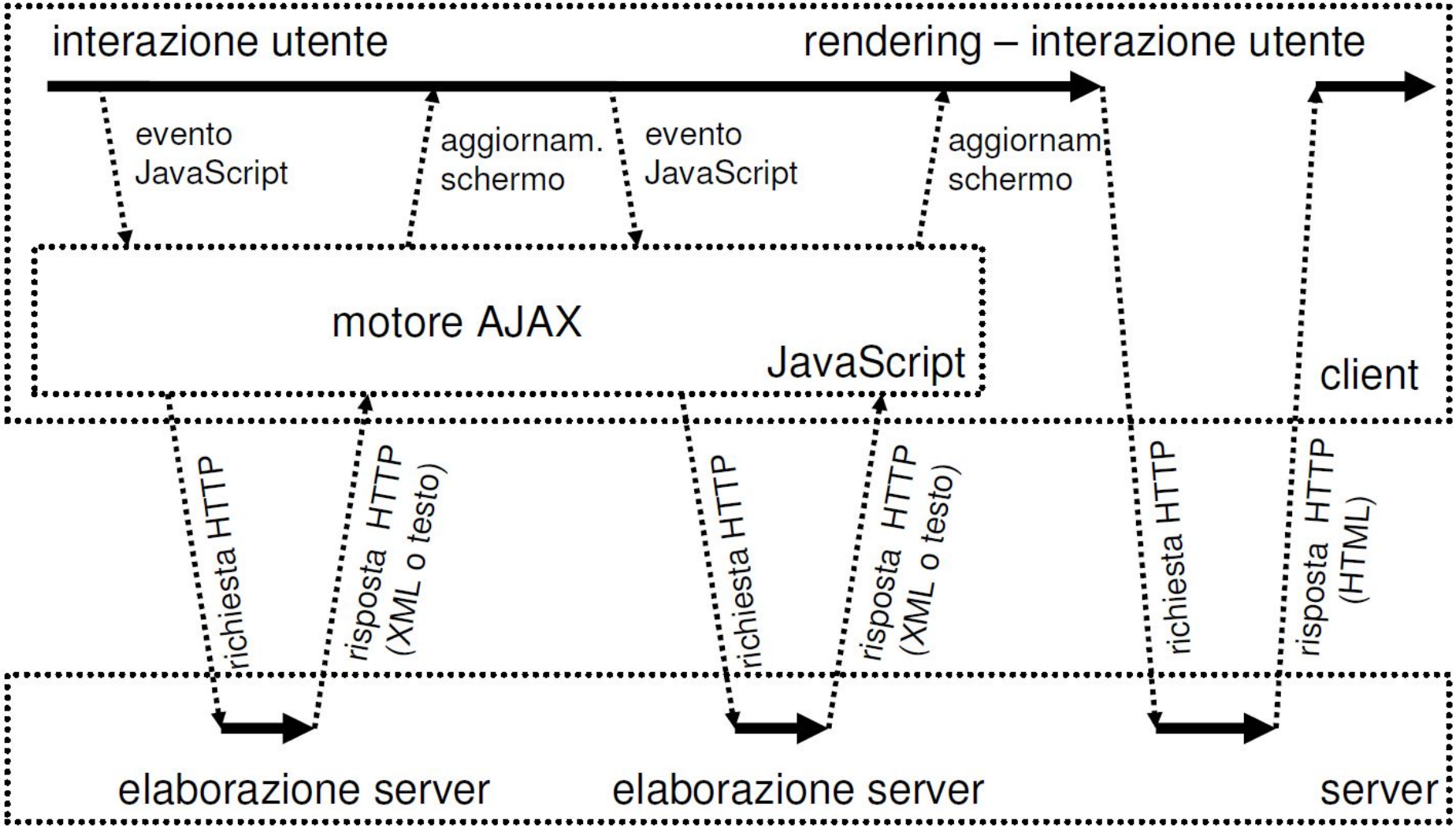
Asynchronous **J**avascript **A**nd **X**ml

- È basato su tecnologie standard e combinate insieme per realizzare un modello di interazione più ricco:
  - JavaScript/Eventi
  - 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** tra client e server: *il client non interrompe interazione con utente anche quando è in attesa di risposte dal server*

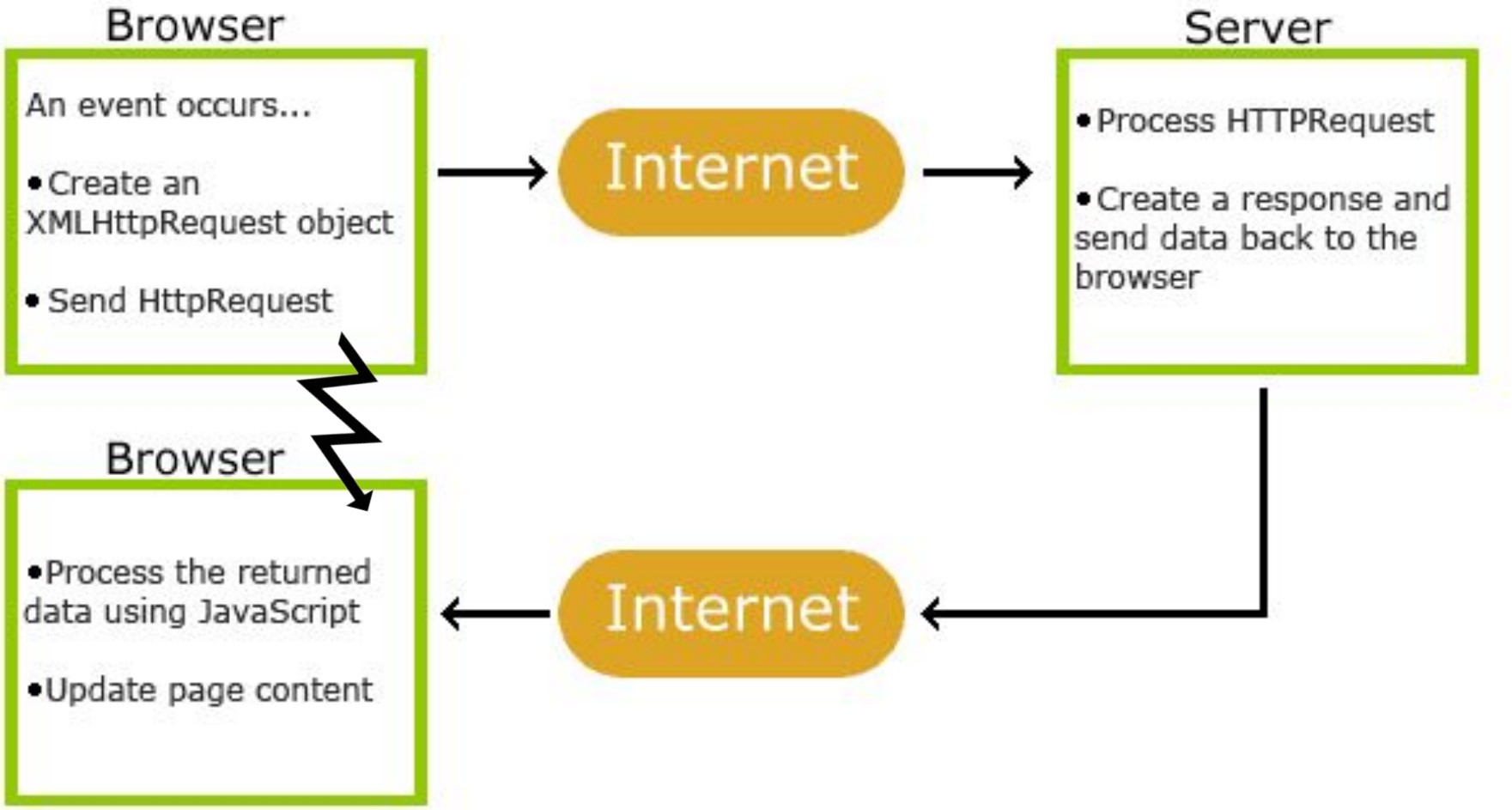
Modello di interazione con AJAX



Tipica sequenza AJAX

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 tipo **XMLHttpRequest**
   - Si configura l'oggetto 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** provoca cambio di pagina o ricaricamento della pagina
  - Può inviare eventuali informazioni sotto forma di parametri (come avviene con una form)
- Può effettuare sia richieste GET che POST
- Le richieste possono essere di tipo
  - **Sincrono**: blocca flusso di esecuzione del codice JavaScript (*NON ci interessa*)
  - **Asincrono**: **NON** interrompe il flusso di esecuzione del codice JavasScript, **NÉ** 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, la esamineremo in seguito per non rendere difficile la comprensione del modello di interazione AJAX*

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 createXMLHttpRequest() {
    var request;
    try {
        // Firefox 1+, Chrome 1+, Opera 8+, Safari 1.2+, Edge 12+, Internet Explorer 7+
        request = new XMLHttpRequest();
    } catch (e) {
        // past versions of Internet Explorer
        try {
            request = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e) {
            try {
                request = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (e) {
                alert("Il browser non supporta AJAX");
                return null;
            }
        }
    }
    return request;
}
```

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** (non disponibile in versioni di IE precedenti alla 7)

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, cioè:

      (**readyState==4**)

  - **responseXML** body della risposta convertito in documento XML (*se possibile*)
    - anch'esso disponibile solo a interazione ultimata
    - consente la navigazione del documento XML 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 associata ad onreadystatechange

- 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>

<div id="demo">
<h1>The XMLHttpRequest Object</h1>
<button type="button" onclick="loadDoc()">Change Content</button>
</div>

<script>
function loadDoc() {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      document.getElementById("demo").innerHTML =
      this.responseText;
    }
  };
  xhttp.open("GET", "ajax_info.txt", true);
  xhttp.send();
}
</script>

</body>
</html>
```

Gestire la risposta XML

- AJAX è in grado di ricevere **documenti XML** accessibili tramite la proprietà **responseXML** dell'oggetto **XMLHttpRequest**
- É possibile elaborare i documenti XML ricevuti utilizzando l'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 del DOM XML di nostro interesse
  - L'oggetto **XMLHttpRequest** ha un in-built XML parser

Ottengo il nodo testuale figlio del primo nodo "result"

```
var response = request.responseXML.documentElement;
alert("Risposta: \n" + request.responseText);
var result = response.getElementsByTagName("result")[0].firstChild.nodeValue;
document.getElementById("datiCapoluogo").innerHTML = result;
```

- 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
  - 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;
    ...
  }
}
```

Funzioni di Callback Multiple

- Il codice mostrato in precedenza non è in grado di svolgere più di un AJAX task (l'oggetto XMLHttpRequest è referenziato tramite una variabile globale)
- Se si devono eseguire più AJAX task in una web application, occorre creare una funzione per l'esecuzione dell'oggetto XMLHttpRequest e una funzione di callback per ciascun AJAX task
  - La chiamata alla funzione dovrebbe contenere (almeno) l'URL e la funzione di callback da chiamare quando la risposta è pronta

```
loadDoc("url-1", myFunction1);
loadDoc("url-2", myFunction2);

function loadDoc(url, cFunction) {
  const request = new XMLHttpRequest();
  request.onreadystatechange = function() {cFunction(this);}
  request.open("GET", url, true);
  request.send();
}

function myFunction1(request) {
  // action goes here
}
function myFunction2(request) {
  // action goes here
}
```

Example: ajax call (index.jsp in ajax.zip)

```
<div>
    <!-- "javascript:void(0)" is used here to avoid submitting the form -->
    <form id="form" action="javascript:void(0)" method="get">
        <p>
            CAP: <input type="text" id="CAP" name="CAP" onchange="cercaCapoluogo()"/>
            <input type="submit" value="Cerca Capoluogo" onclick="cercaCapoluogo()" />
        </p>
    </form>
</div>

<p>
    Capoluogo: <strong><span id="datiCapoluogo"></span></strong>
</p>
```

cercaCapoluogo (in questo caso) gestisce sia l'evento change che l'evento cerca

Example: ajax call (2)

```
function cercaCapoluogo() {
    var input = document.getElementById('CAP').value;
    var params = 'CAP=' + input;
    loadAjaxDoc('cercaCapoluogo', "GET", params, handleCAP);
}

function createXMLHttpRequest() {
    var request;
    try {
        // Firefox 1+, Chrome 1+, Opera 8+, Safari 1.2+, Edge 12+, Internet Explorer 7+
        request = new XMLHttpRequest();
    } catch (e) {
        // past versions of Internet Explorer
        try {
            request = new ActiveXObject("Msxml2.XMLHTTP");
        } catch (e) {
            try {
                request = new ActiveXObject("Microsoft.XMLHTTP");
            } catch (e) {
                alert("Il browser non supporta AJAX");
                return null;
            }
        }
    }
    return request;
}
```

createXMLHttpRequest lo abbiamo già visto qualche slide fa…

Example: ajax call (3)

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

    /**
    private static final long serialVersionUID = 1L;

    private IDAOCapoluogo daoCapoluogo = new DAOCapoluogoMock();

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

        response.setContentType("text/xml");
        PrintWriter out = response.getWriter();
        String CAP = request.getParameter("CAP");
        Capoluogo capoluogo = null;
        if (CAP != null && !CAP.equals("")) {
            capoluogo = daoCapoluogo.findCapolougoByCAP(CAP);
        }
        String risultato = null;
        if (capoluogo != null) {
            risultato = capoluogo.getNome() + " (" + capoluogo.getRegione() + ")";
        } else {
            risultato = "non trovato";
        }
        out.println("<?xml version=\"1.0\" encoding=\"iso-8859-1\" ?>");
        out.println("<response>");
        out.println("<functionName>aggiornaDatiCapoluogo</functionName>");
        out.println("<result>" + risultato + "</result>");
        out.println("</response>");
    }

    /** Handles the HTTP <code>GET</code> method.
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
```

Example: ajax call (4)

```
function loadAjaxDoc(url, method, params, cFuction) {
    var request = createXMLHttpRequest();
    if(request){

        request.onreadystatechange = function() {
            if (this.readyState == 4) {
                if (this.status == 200) {
                    cFuction(this);
                } else {
                    if(this.status == 0){ // When aborting the request
                        alert("Problemi nell'esecuzione della richiesta: nessuna risposta ricevuta nel tempo limite");
                    } else { // Any other situation
                        alert("Problemi nell'esecuzione della richiesta:\n" + this.statusText);
                    }
                    return null;
                }
            }
        };

        setTimeout(function () {      // to abort after 15 sec
            if (request.readyState < 4) {
                request.abort();
            }
        }, 15000);
        ...
```

di **setTimeout** ne parliamo tra qualche slide..ignoriamolo per ora…

loadAjaxDoc continua nella prossima slide

Example: ajax call (5)

```
        ...
        if(method.toLowerCase() == "get"){
            if(params){
                request.open("GET", url + "?" + params, true);
            } else {
                request.open("GET", url, true);
            }
            request.setRequestHeader("Connection", "close");
            request.send(null);
        } else {
            if(params){
                request.open("POST", url, true);
                request.setRequestHeader("Connection", "close");
                request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
                request.send(params);
            } else {
                console.log("Usa GET se non ci sono parametri!");
                return null;
            }
        }
    }
}
```

il codice riportato può gestire sia chiamate GET che POST

Example: ajax call (6)

```
function handleCAP(request){
    var response = request.responseXML.documentElement;
    alert("Risposta: \n" + request.responseText);
    var result = response.getElementsByTagName("result")[0].firstChild.nodeValue;
    document.getElementById("datiCapoluogo").innerHTML = result;
}
```

HandleCAP è la funzione di callback usata quando **readystate=4** e **status=200**

Vantaggi e svantaggi di Ajax

- Si guadagna in **interattività,** 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

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()

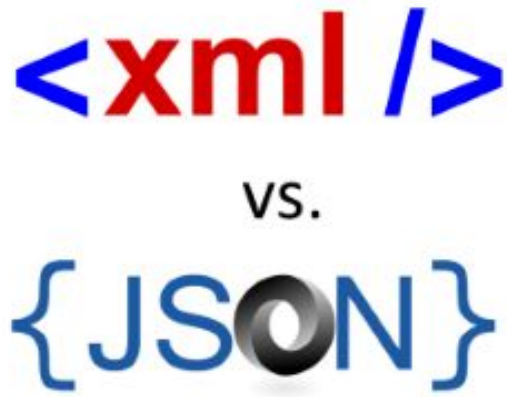
- **abort()** consente l'interruzione delle operazioni di invio o ricezione
  - non ha bisogno di parametri
  - termina immediatamente la trasmissione dati
- *Attenzione: non ha senso invocarlo dentro la funzione di* ***callback***
  - Se readyState non cambia, il metodo non viene richiamato; readyState 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*, *timeOutInMillis*)
- All'interno della ***funzioneAsincronaPerAbortire*** si valuta se continuare l'attesa o abortire l'operazione

```
setTimeout(function () {      // to abort after 15 sec
    if (request.readyState < 4) {
        request.abort();
    }
}, 15000);
```

Aspetti critici per il programmatore

- *È accresciuta la complessità delle Web Application*
- Le applicazioni AJAX pongono problemi di debug, test e mantenimento
  - Il test di codice JavaScript è complesso
- 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 maggiore di byte rispetto al testo piano*
  - 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 gli object literal e gli array literal in JavaScript

Object e object literal

- In JavaScript è possibile creare un oggetto in base mediante un object literal:

```
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 array literal

- In modo analogo è possibile creare un array utilizzando un array literal

  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 degli object e array literial di JavaScript
- Un **"oggetto JSON"** altro non è che una stringa equivalente ad un object literial JavaScript

Object literal JavaScipt

```
{
 "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"] }'
```

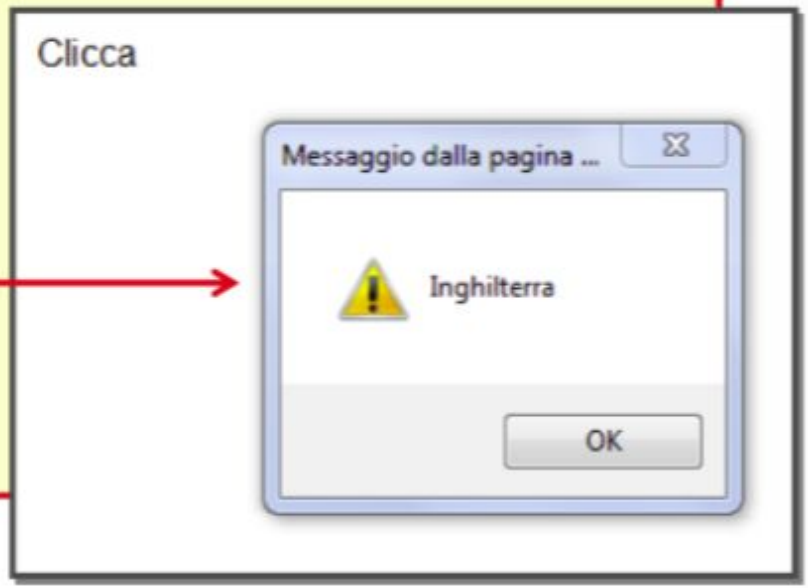
Da stringa JSON a oggetto (old)

- 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>
```



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
- Per convertire una stringa JSON in un oggetto JavaScript, si usa quindi la funzione built-in di JavaScript **JSON.parse(JSONstring)**
- Se si vuole convertire un oggetto JavaScript in una stringa JSON, si usa **JSON.stringify(JSONobject)**

Esempio di parsing JSON

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

<h2>Create Object from JSON String</h2>

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

<script>
let text = '{"employees":[' +
'{"firstName":"John","lastName":"Doe" },' +
'{"firstName":"Anna","lastName":"Smith" },' +
'{"firstName":"Peter","lastName":"Jones" }]}';

const obj = JSON.parse(text);
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " + obj.employees[1].lastName;
</script>

</body>
</html>
```

Create Object from JSON String

Anna Smith

https://www.w3schools.com/js/tryit.asp?filename=tryjs_json_parse

AJAX e ricezione file JSON

- Simile a quanto avviene con la ricezione di file XML, con due differenze, una lato server e una lato client
- Lato Server:
  - La servlet deve trasmettere la stringa JSON (non XML) nel body della risposta HTTP al client
    1. Occorre impostare il content type "application/json"
    2. Occorre un parser JSON (scarica da https://github.com/stleary/JSON-java la libreria **JSON-Java** e importala nel progetto Eclipse) per creare la stringa JSON
- Lato client:
  - Si converte la stringa JSON in un oggetto JavaScript usando **JSON.parse()**

* Un'altra libreria JSON molto usata in Java è GSON: https://mvnrepository.com/artifact/com.google.code.gson/gson

Esempio (index-json.jsp in ajax.zip)

```
<body>
    <h1>Esempio AJAX con JSON</h1>

    <p>
        <a href="esci.jsp">Esci</a>
    </p>

    <div>
        <!-- "javascript:void(0)" is used here to avoid submitting the form -->
        <form id="form" action="javascript:void(0)" method="get">
            <p>
                CAP: <input type="text" id="CAP" name="CAP" onchange="cercaCapoluogo()"/>
                <input type="submit" value="Cerca Capoluogo" onclick="cercaCapoluogo()" />
            </p>
        </form>
    </div>

    <p>
        Capoluogo: <strong><span id="datiCapoluogo"></span></strong>
    </p>

</body>
```

Esempio (2)

- Lato Server:

```
@WebServlet("/cercaCapoluogoJson")
public class ServletCercaCapoluogoJson extends HttpServlet {
    /**
    private static final long serialVersionUID = 1L;

    private IDAOCapoluogo daoCapoluogo = new DAOCapoluogoMock();

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

        response.setContentType("application/json");
        PrintWriter out = response.getWriter();
        String CAP = request.getParameter("CAP");
        Capoluogo capoluogo = null;
        if (CAP != null && !CAP.equals("")) {
            capoluogo = daoCapoluogo.findCapolougoByCAP(CAP);
        }
        String risultato = null;
        if (capoluogo != null) {
            risultato = capoluogo.getNome() + " (" + capoluogo.getRegione() + ")";
        } else {
            risultato = "non trovato";
        }
        JSONObject json = new JSONObject();
        json.put("functionName", "aggiornaDatiCapoluogoJSON");
        json.put("result", risultato);
        out.print(json.toString());
    }

    /** Handles the HTTP <code>GET</code> method.
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
```

Esempio (3)

- Lato Client:

```
function handleCAP(request){
    var response = JSON.parse(request.responseText);
    alert("Risposta: \n" + request.responseText);
    document.getElementById("datiCapoluogo").innerHTML = response.result;
}
```

AJAX e invio JSON

- Sul lato client:
  1. Si crea l'oggetto JavaScript da convertire in stringa riempiendo le sue proprietà con le informazioni necessarie
  2. Si usa **JSON.stringify()** per convertire l'oggetto in stringa JSON
  3. Si manda la stringa al server usando l'oggetto **XMLHttpRequest** via POST o GET, in ogni caso va impostato l'header come segue **setRequestHeader("Content-Type", "application/json")**
     - Per richieste GET: la stringa json va codificata usando il metodo **encodeURIComponent(JSONString)**

Esempio di invio di stringa JSON via POST

```
var xhr = new XMLHttpRequest();
var url = "url";
xhr.open("POST", url, true);
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
        var json = JSON.parse(xhr.responseText);
        console.log(json.email + ", " + json.password);
    }
};
var data = JSON.stringify({"email": "hey@mail.com", "password": "101010"});
xhr.send(data);
```

Esempio di invio di stringa JSON via GET

```
var xhr = new XMLHttpRequest();
var url = "url?data=" + encodeURIComponent(JSON.stringify({"email": "hey@mail.com",
    "password": "101010"}));
xhr.open("GET", url, true);
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function () {
    if (xhr.readyState === 4 && xhr.status === 200) {
        var json = JSON.parse(xhr.responseText);
        console.log(json.email + ", " + json.password);
    }
};
xhr.send();
```

jQuery and AJAX

- ***Without jQuery, AJAX coding can be a bit tricky!***
  - Different browsers have different syntax for AJAX implementation, thus we have to write extra code to test for different browsers
  - The jQuery team has taken care of this for us, thus we can write AJAX functionality with only one single line of code

AJAX call in JQuery

- **$.ajaxSetup()** to set default options before any AJAX call
- There are different ways to make an AJAX call in JQuery:
  - **$.ajax()**
  - **load()**
  - **$.get()**
  - **$.post()**
  - …

Specify default options for next AJAX calls

- The $.ajaxSetup() method sets default values for future AJAX calls
- Syntax: **$.ajaxSetup({name:value, name:value, ... })**
- See all the setup options at:
  - https://www.w3schools.com/jquery/ajax_ajaxsetup.asp
- A usage example:

```
// Set the GET method and a timeout of 10 sec (after which aborting the request) as default values

$.ajaxSetup({ type: "GET", timeout : 10000 })

// Then make the ajax call

$.ajax({url: "demo_test.txt", success: function(result){$("#div1").html(result);}

});
```

See a running example at:
https://www.w3schools.com/jquery/ajax_ajaxsetup.asp

AJAX calls with $.ajax()

- The **$.ajax()** method is used to perform an AJAX calls
- The other jQuery AJAX methods leverage the **$.ajax()** method
  - This method is mostly used for requests where the other methods -- e.g., **load()**, **$.get()** and **$.post()** -- cannot be used
- Syntax: **$.ajax({name:value, name:value, ... })**
- See all the options at:
  - https://www.w3schools.com/jquery/ajax_ajax.asp
- A usage example:

```
// Make an AJAX call (with GET method) to "demo_test.txt" and then execute the specified function when the call succeeds

$.ajax({url: "demo_test.txt", type: "GET", success: function(result){

  $("#div1").html(result); }});
```

See a running example at:

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_ajax

AJAX calls with load()

- The jQuery **load()** method is a simple, but powerful AJAX method
- The load() method loads data from a server (i.e., by making an AJAX call) and puts the returned data into the selected element
- Syntax: **$(*selector*).load(*URL, data, callback*);**
- Parameters:
  - **URL:** (mandatory) It specifies the URL of the resource
  - **data** (optional): It specifies a set of key/value pairs (by using an object literal) to send along with the request
  - **callback** (optional): It specifies a callback function to run when the load() method is completed. The callback function can have optional parameters:
    - **responseTxt**: the resulting textual content if the call succeeds
    - **statusTxt:** the status of the call
    - **xhr:** the jqXHR object (it is a superset of the XMLHTTPRequest object)

Example

```
<!DOCTYPE html>
<html>
<head>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/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>
```

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_load

Another 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:

```
$(document).ready(function(){
  $("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);
    });
  });
});
```

See the full example at:
https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_load_callback

$.get()

- The $.get() requests data from the server using an HTTP GET request
- Syntax: **$.get(URL,data,function(data,status,xhr),dataType)**
- Parameters:
  - **URL:** The required URL parameter specifies the URL you wish to load
  - **data**: (optional) It specifies the data to send to the server along with the request
  - **function**: (optional) It specifies a function to run if the request succeeds. Additional function parameters:
    - **data**: the resulting data from the request. The data type expected of the server response is, by default, automatically guessed by JQuery (based on the content type of the response). For example, if the content type is JSON, data is a JavaScript object
    - **status:** the status of the call (e.g., "success")
    - **xhr:** the jqXHR object (it is a superset of the XMLHTTPRequest object)
  - **dataType**: Specifies the data type expected of the server response (“xml”, “html”, “text”, “json”, …).

Examples

Request "test.php", but ignore return results:

```
$.get("test.php");
```

Request "test.php" and send some additional data along with the request (ignore return results):

```
$.get("test.php", { name:"Donald", town:"Ducktown" });
```

Request "test.php" and pass arrays of data to the server (ignore return results):

```
$.get("test.php", { 'colors[]' : ["Red","Green","Blue"] });
```

Request "test.php" and alert the result of the request:

```
$.get("test.php", function(data){
  alert("Data: " + data);
});
```

Example

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $.get("demo_test.asp", function(data, status){
      alert("Data: " + data + "\nStatus: " + status);
    });
  });
});
</script>
</head>
<body>

<button>Send an HTTP GET request to a page and get the result back</button>

</body>
</html>
```

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_get

$.post()

- Similar to $.get(), but it requests data from the server using a HTTP POST request
- Syntax: **$.post(URL,data,function(data,status,xhr),dataType)**
- Parameters:
    - the same as **$.get()**

Example

```
<!DOCTYPE html>
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
<script>
$(document).ready(function(){
  $("button").click(function(){
    $.post("demo_test_post.asp",
    {
      name: "Donald Duck",
      city: "Duckburg"
    },
    function(data,status){
      alert("Data: " + data + "\nStatus: " + status);
    });
  });
});
</script>
</head>
<body>

<button>Send an HTTP POST request to a page and get the result back</button>

</body>
</html>
```

https://www.w3schools.com/jquery/tryit.asp?filename=tryjquery_ajax_post

The jqXHR object

- Both the **$.get()** and **$.post()** methods return a jqXHR object
- The jqXHR object exposes the following methods:
  - **done(function(jqXHR, textStatus, errorThrown))** execute the function in case of success (the function parameters are optional)
  - **fail(function(jqXHR, textStatus, errorThrown))** execute the function in case of failure
  - **always(function(jqXHR, textStatus, errorThrown))** execute the function in any case

```
var jqxhr = $.post( "example.php", function() {
  alert( "success" );
})
  .done(function() {
    alert( "second success" );
  })
  .fail(function() {
    alert( "error" );
  })
  .always(function() {
    alert( "finished" );
  });
```

Esempio (index.jsp in ajax-jquery.zip)

```
<body>
    <h1>Esempio AJAX JQuery con JSON</h1>

    <p>
        <a href="esci.jsp">Esci</a>
    </p>

    <div>
        <!-- "javascript:void(0)" is used here to avoid submitting the form -->
        <form id="form" action="javascript:void(0)" method="get">
            <p>
                CAP: <input type="text" id="CAP" name="CAP"/>
                <input type="submit" id="cerca" value="Cerca Capoluogo"/>
            </p>
        </form>
    </div>

    <p>
        Capoluogo: <strong><span id="datiCapoluogo"></span></strong>
    </p>

</body>
```

Esempio (2)

```
$(document).ready(function(){

    $.ajaxSetup({timeout : 10000}); // Imposta (per tutte le richieste) un timeout di 10sec per ricevere la risposta HTTP

    $("#cerca").click(handleCAP);
    $("#CAP").change(handleCAP);

    function handleCAP(){
        var jqxhr = $.post("cercaCapoluogoJson", { "CAP": $("#CAP").val()}, function(data){
            //alert(JSON.stringify(data)); de-commenta questa linea per stampare la stringa JSON tramite un alert
            $("#datiCapoluogo").html(data.result);
        });

        jqxhr.fail(function(_jqXHR, textStatus, errorThrown){
            if(textStatus == "timeout"){
                alert( "Problemi nell'esecuzione della richiesta: nessuna risposta ricevuta nel tempo limite");
            } else {
                alert("Problemi nell'esecuzione della richiesta:" + errorThrown);
            }
        });
    }

});
```

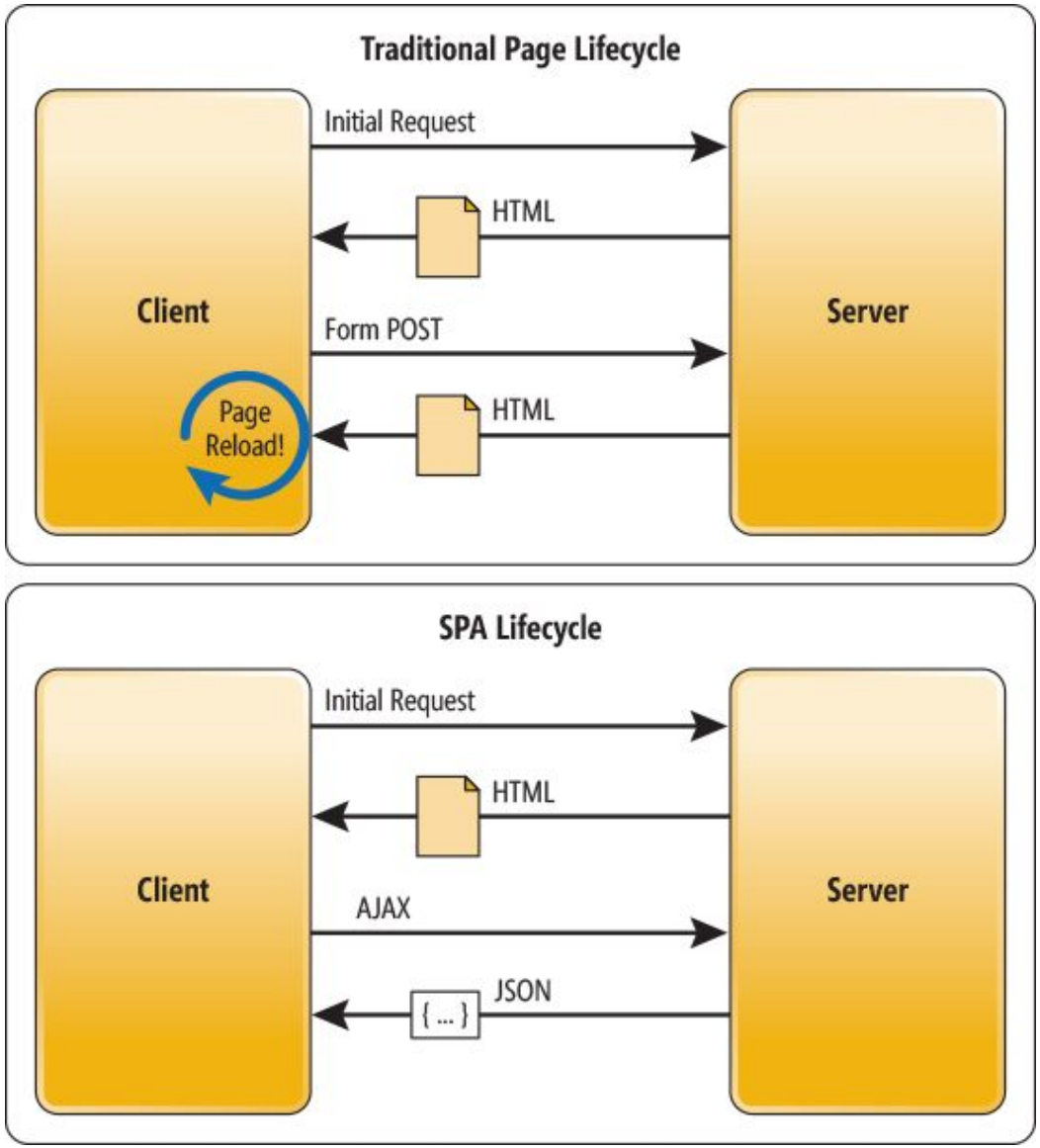
Esempio (3)

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

    /**
    private static final long serialVersionUID = 1L;

    private IDAOCapoluogo daoCapoluogo = new DAOCapoluogoMock();

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

        response.setContentType("application/json");
        PrintWriter out = response.getWriter();
        String CAP = request.getParameter("CAP");
        Capoluogo capoluogo = null;
        if (CAP != null && !CAP.equals("")) {
            capoluogo = daoCapoluogo.findCapolougoByCAP(CAP);
        }
        String risultato = null;
        if (capoluogo != null) {
            risultato = capoluogo.getNome() + " (" + capoluogo.getRegione() + ")";
        } else {
            risultato = "non trovato";
        }
        JSONObject json = new JSONObject();
        json.put("functionName", "aggiornaDatiCapoluogoJSON");
        json.put("result", risultato);
        out.print(json.toString());
    }

    /** Handles the HTTP <code>GET</code> method.
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        processRequest(request, response);
    }
```

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 desktop app

Single Page Application (2)



Example (AjaxServletIntegration.zip)

- The purpose of the application is simple:
  - Add a band name with a list of albums (separated by commas) and press the **"Submit"** button to add them to the database.
  - Press the **"Show bands!"** button to get a list of the bands, or the **"Show bands and albums!"** button to get a list of bands with their albums

Example (2)

Ajax - Servlets Integration Example

This is an example of how to use Ajax with a servlet backend.

Select a button to get the relevant information.

Show bands! Show bands and albums!

Add the band information and press submit!

Band name:

Albums:

(Separated by commas)

Submit

Example (3)

index.jsp

```
17
18      <body>
19          <h1>Ajax - Servlets Integration Example</h1>
20          <p>This is an example of how to use Ajax with a servlet
    backend.</p></br>
21
22          <h3>Select a button to get the relevant information.</h3>
23
24          <!-- Buttons that will call the servlet to retrieve the
    information. -->
25          <button id="bands" type="button">Show bands!</button>
26          <button id="bands-albums" type="button">Show bands and
    albums!</button>
27
28          <!-- We need to have some empty divs in order to add the
    retrieved information to them. -->
29          <div id="band-results"></div></br></br>
30          <div id="bands-albums-results"></div></br></br>
31
32
33          <h3>Add the band information and press submit!</h3>
34          <h4>Band name: </h4><input type="text" id="band-name-input"
    value=""><br>
35          <h4>Albums: </h4><input type="text" id="album-input"
    value="">(Separated by commas)<br>
36          <input type="submit" id="submit-band-info" value="Submit">
37      </body>
38  </html>
```

Example (4)

buttonEventsInit.js

```
01 // When the page is fully loaded...
02 $(document).ready(function() {
03
04     // Add an event that triggers when ANY button
05     // on the page is clicked...
06     $("button").click(function(event) {
07
08         // Get the button id, as we will pass it to the servlet
09         // using a GET request and it will be used to get different
10         // results (bands OR bands and albums).
11         var buttonID = event.target.id;
12
13         // Basic JQuery Ajax GET request. We need to pass 3
   arguments:
14         //      1. The servlet url that we will make the request
   to.
15         //      2. The GET data (in our case just the button ID).
16         //      3. A function that will be triggered as soon as the
   request is successful.
17         // Optionally, you can also chain a method that will handle
   the possibility
18         // of a failed request.
19         $.get('DBRetrievalServlet', {"button-id": buttonID},
20             function(resp) { // on sucess
21                 // We need 2 methods here due to the different ways
   of
22                 // handling a JSON object.
23                 if (buttonID === "bands")
24                     printBands(resp);
25                 else if (buttonID === "bands-albums")
26                     printBandsAndAlbums(resp);
27             })
28             .fail(function() { // on failure
29                 alert("Request failed.");
30             });
31     });
32 });
```

Example (5)

DBRetrievalServlet.java

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

    @Override
    public void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

        // We set a specific return type and encoding
        // in order to take advantage of the browser capabilities.
        response.setContentType("application/json");
        response.setCharacterEncoding("UTF-8");
        String buttonID = request.getParameter("button-id");
        switch (buttonID) {
            case "bands":
                response.getWriter().write(MusicDatabase
                        .getInstance()
                        .getBands());
                break;

            case "bands-albums":
                response.getWriter().write(MusicDatabase
                        .getInstance()
                        .getBandsAndAlbums());
                break;
        }
    }
}
```

Example (6)

MusicDatabase.java

```
    public void setBandAndAlbums(String bandName, ArrayList
bandAlbums) {
        bandNames.add(bandName);
        bandsAndAlbums.add(new BandWithAlbums(bandName,
bandAlbums));
    }

    public String getBands() {
        return new Gson().toJson(bandNames);
    }

    public String getBandsAndAlbums() {
        return new Gson().toJson(bandsAndAlbums);
    }
}
```

Example (7)

resultsPrinter.js

```
function printBands(json) {

    // First empty the <div> completely and add a title.
    $("#band-results").empty()
        .append("<h3>Band Names</h3>");

    // Then add every band name contained in the list.
    $.each(json, function(i, name) {
        $("#band-results").append(i + 1, ". " + name + " </br>");
    });
};
```

```
function printBandsAndAlbums(json) {

    // First empty the <div> completely and add a title.
    $("#bands-albums-results").empty()
        .append("<h3>Band Names and Albums</h3>");

    // Get each band object...
    $.each(json, function(i, bandObject) {

        // Add to the <div> every band name...
        $("#bands-albums-results").append(i + 1, ". " +
bandObject.bandName + " </br>");
        // And then for every band add a list of their albums.
        $.each(bandObject.bandAlbums, function(i, album) {
            $("#bands-albums-results").append("--" + album + "
</br>");
        });
    });
};
```

Example (8)

insertBandInfo.js

```
$(document).ready(function() {

    // Add an event that triggers when the submit
    // button is pressed.
    $("#submit-band-info").click(function() {

        // Get the text from the two inputs.
        var bandName = $("#band-name-input").val();
        var albumName = $("#album-input").val();

        // Fail if one of the two inputs is empty, as we need
        // both a band name and albums to make an insertion.
        if (bandName === "" || albumName === "") {
            alert("Not enough information for an insertion!");
            return;
        }

        // Ajax POST request, similar to the GET request.
        $.post('DBInsertionServlet',{"bandName": bandName,
"albumName": albumName},
            function() { // on success
                alert("Insertion successful!");
            })
            .fail(function() { //on failure
                alert("Insertion failed.");
            });
    });
});
```

Example (9)

DBInsertionServlet.java

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

        Map<String, String[]> bandInfo = request.getParameterMap();

        // In this case here we are not using the data sent to just
do different things.
        // Instead we are using them as information to make changes
to the server,
        // in this case, adding more bands and albums.
        String bandName =
Arrays.asList(bandInfo.get("bandName")).get(0);
        String albums =
Arrays.asList(bandInfo.get("albumName")).get(0);

        MusicDatabase.getInstance()
            .setBandAndAlbums(bandName,
getAlbumNamesFromString(albums));

        // return success
        response.setStatus(200);
    }

    // Split the album String in order to get a list of albums.
    private ArrayList getAlbumNamesFromString(String albums) {
        return new ArrayList(Arrays.asList(albums.split(",")));
    }
}
```

Riferimenti

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

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