Alma Mater Studiorum
Università di Bologna

Scuola di Ingegneria

Tecnologie Web T
A.A. 2023 – 2024

Esercitazione 05 - AJAX

Su Virtuale:
Versione 1 pagina per foglio = L.05.AJAX.pdf
Versione 2 pagine per foglio = L.05.AJAX-2p.pdf

AJAX - Asynchronous Javascript and XML

Richieste al server effettuate via Javascript in modo **asincrono**

- l'utente non ottiene una nuova pagina, ma solo modifiche a quella corrente
- non è necessario aspettare la response per continuare a interagire
- non è necessario aspettare la response per eseguire altro codice AJAX

Interfacciamento client-server mediante lo scambio di:

- testo semplice (frammenti di pagina e/o singole informazioni)
- documenti XML (informazioni strutturate e complesse)

Non una nuova tecnologia, ma un **nuovo modo di utilizzare tecnologie esistenti** per sviluppare **rich Internet application**

- Gmail, Google Maps, Google Suggest, FaceBook, ...

Alternative tecnologiche

- Adobe-Macromedia Flash
- Java applet
- uso del tag *iframe* (banalmente si modifica l'attributo *src* del frame per simulare un aggiornamento parziale dei contenuti)

Oggetto *XMLHttpRequest*

L'oggetto Javascript usato per realizzare le richieste AJAX

Effettua la **richiesta di una risorsa via HTTP** ad un server Web

- in modo indipendente dal browser
  - **non sostituisce l'URI** della propria richiesta all'URI corrente
  - non provoca un cambio di pagina
- inviando eventuali informazioni sotto forma di variabili (come una *form*)
  - di tipo GET
  - di tipo POST
- in modo
  - sincrono (blocca il flusso di esecuzione del codice Javascript; non ci interessa)
  - **asincrono** (non interrompe il flusso di esecuzione del codice Javascript né le operazioni dell'utente sulla pagina)

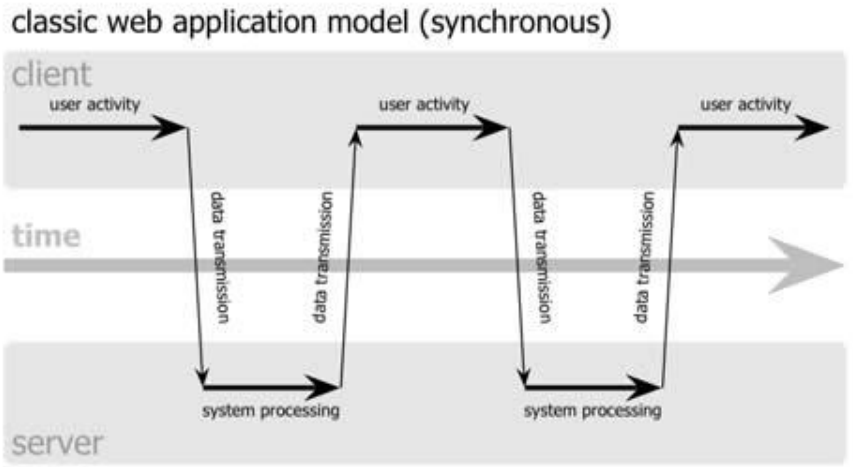
Alterazione del paradigma classico di interazione C/S

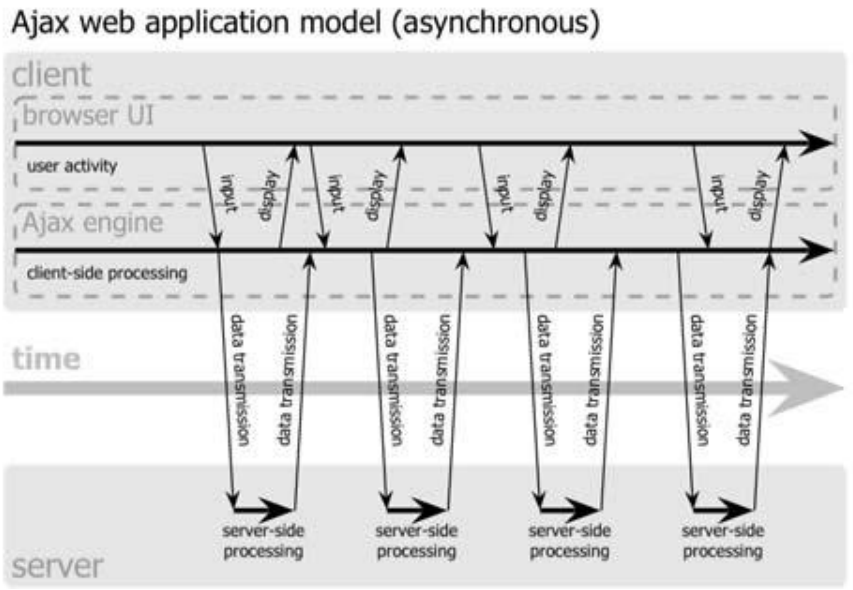
Si guadagna in espressività, ma **si perde la linearità dell'interazione**

- mentre l'utente è all'interno della stessa pagina, le richieste sul server possono essere numerose e indipendenti
- il tempo di attesa passa in secondo piano o non è avvertito affatto

Possibili criticità sia per l'utente che per lo sviluppatore

- percezione che non stia accadendo nulla (sito che non risponde)
- problemi nel programmare logica che ha bisogno di aspettare i risultati delle richieste precedenti





Tipologie di interazioni AJAX

Semplici

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

Avanzate

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

Una cosa che capita spesso…

Astrarre i diversi comportamenti di diversi browser dietro a…

- …librerie di terze parti (es: *jquery*)
- …funzioni *ad hoc*

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

   // elemento da restituire
   var elemento;

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

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

   // restituzione elemento
   return elemento;

}
```

Primi passi...

Evento *onload* e disponibilità del DOM

http://localhost:8080/05_TecWeb/1_beforeonload.html
http://localhost:8080/05_TecWeb/2_onload.html

Chrome Developer Tools: dalla scheda Sources inserire breakpoint su...
- *scripts/almostajax.js:6*

...lettura degli eventuali messaggi di errore

```
<html>

  <head>
    <title>AJAX, le basi prima dell'utilizzo</title>
    <script type="text/javascript" src="myutils.js"></script>
    <script type="text/javascript" src="almostajax.js"></script>
  </head>

  <body onload="almostAjax('paragrafo')">

    <p id="paragrafo">
      testo del paragrafo che verr&agrave; cambiato al caricamento del documento
    </p>
  </body>

</html>
```

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

invocare la procedura direttamente nel tag *<head>* della pagina avrebbe generato errore!
Prima del verificarsi dell'evento *onload* non è nota al browser nemmeno l'esistenza di un elemento con *id="paragrafo"*!

INVIO
DELLE RICHIESTE

Ottenere l'oggetto *XMLHttpRequest*

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

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

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

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

```
var xhr = new ActiveXObject("Microsoft.XmlHttp")
var xhr = new ActiveXObject("MSXML4.XmlHttp")
var xhr = new ActiveXObject("MSXML3.XmlHttp")
var xhr = new ActiveXObject("MSXML2.XmlHttp")
var xhr = new ActiveXObject("MSXML.XmlHttp")
```

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

dalla più recente → alla più obsoleta

Controllo del supporto

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

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

LETTURA
DELLE RISPOSTE

Proprietà di *XMLHttpRequest*

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

Lista dei parametri comunemente supportati dai vari browser:

readyState

onreadystatechange

status

responseText

responseXML

Proprietà *readyState*

Variabile di tipo intero, con valori che vanno da 0 a 4

- ***0*** **:** ***uninitialized*** - l'oggetto *XMLHttpRequest* esiste, ma non è stato richiamato alcun metodo per inizializzare una comunicazione
- ***1*** **:** ***open*** - è stato precedentemente invocato il metodo *open()*, ma il metodo *send()* non ha ancora effettuato l'invio dati
- ***2*** **:** ***sent*** - il metodo *send()* è stato eseguito ed ha effettuato la richiesta
- ***3*** **:** ***receiving*** - i dati in risposta cominciano ad essere letti
- ***4*** **:** ***loaded*** **- l'operazione è stata completata**

Accessibile in **sola lettura**: rappresenta in ogni istante lo stato della richiesta

Attenzione:

- nello stato 3 (che può essere assunto più volte…) si possono già leggere alcuni header restituiti dal server o parte della risposta
- questo ordine non è sempre identico e non è sfruttabile allo stesso modo su tutti i browser
- se la richiesta fallisce *readyState* potrebbe non assumere mai il valore 3
- **l'unico stato supportato da tutti i browser è il 4**: a prescindere dalla riuscita dell'operazione, le operazioni sono terminate e lo stato non cambierà più

Proprietà *onreadystatechange*

L'esecuzione del codice **non si blocca sulla *send()*** in attesa dei risultati

Occorre registrare una funzione che sia richiamata dal sistema

- **come?** in maniera asincrona rispetto al resto del programma
- **perché?** al momento della disponibilità di risultati (anche parziali!)
- **quando?** a sua volta segnalata dal cambio di stato della richiesta

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

Occorre evitare alee!

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

*bisogna fare tale assegnamento **prima** della send()*

Proprietà *status*

Valore intero corrispondente al codice HTTP dell'esito della richiesta

- ***200***: caso di **successo** (l'unico in base al quale i dati ricevuti in risposta devono essere ritenuti corretti e significativi)
- possibili altri valori (in particolare 403, 404, 500, …)

Una descrizione testuale del codice HTTP restituito dal server…

- è contenuta nel parametro *statusText* (supportato in quasi tutti i browser tranne alcune versioni di Opera)

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

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

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

Proprietà *responseText* e *responseXML*

Contengono i dati restituiti dal server

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

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

Lettura degli header HTTP che descrivono la risposta del server

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

```
const xhr = myGetXMLHttpRequest();
xhr.onreadystatechange = () => {
  if (client.readyState === client.HEADERS_RECEIVED) {
    const contentType = client.getResponseHeader("Content-Type");
    if (contentType !== my_expected_type) {
      client.abort();
    }
  }
}; ……
```

La funzione di “callback” (1)

Legge lo stato di avanzamento della richiesta

- *readystate*

Verificare il successo o fallimento della richiesta

- *status*

Ha accesso agli header di risposta rilasciati dal server (parziali se *readystate* == *3*, completi se *readystate* == *4*)

- *getAllResponseHeaders()*
- *getResponseHeader(header_name)*

Può leggere il contenuto della risposta (se e solo se *readystate* == *4*)

- *responseText*
- *responseXML*

La funzione di “callback” (2)

Assegnata all'attributo *onreadystatechange* di *XMLHttpRequest*

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

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

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

Richiamata **AD OGNI VARIAZIONE** del parametro *readyState*

- su alcuni browser lo stato 3 può essere assunto più volte in caso di ricezione di una risposta molto lunga in successivi trunk

Chiusura della funzione di “callback”

È costituita dalla funzione stessa e dall’insieme di tutte le variabili a cui essa può accedere (scope)

- se definita **in linea**, può riferire le variabili dello scope in cui si trova

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

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

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

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

/* NON È INVECE POSSIBILE QUESTO TIPO DI ASSEGNAMENTO !!! */
xhr.onreadystatechange = myPopup(xhr,textHolder);          ---> perche'?
```

ESEMPI

Ovviamente…

OVVIO, MA IMPORTANTE

Per poter testare questo esempio, come qualunque altra applicazione basata su *XMLHttpRequest*, **è necessario richiamare la pagina HTML attraverso un Web server**

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

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

Chi dovrebbe ricevere le HTTP Request e inviare HTTP Response?

~~*C:\pagina.html*~~

Scaricamento di dati in formato testo (1)

Pagine

http://localhost:8080/05_TecWeb/3_plaintext.html

http://localhost:8080/05_TecWeb/4_plaintext-external.html

```
/* VEDERE IL CODICE (E RELATIVI COMMENTI) NEL PROGETTO DI ESEMPIO */
```

Verifica della possibilità di usare tecniche AJAX

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

Chrome Developer Tools: dalla scheda Sources inserire breakpoint su…

- *scripts/callback.js:6*

…eseguire una prima volta passo passo

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

Scaricamento di dati in formato testo (2)

Pagine

http://localhost:8080/05_TecWeb/3_plaintext.html

http://localhost:8080/05_TecWeb/4_plaintext-external.html

```
/* VEDERE IL CODICE (E RELATIVI COMMENTI) NEL PROGETTO DI ESEMPIO */
```

Esecuzione di operazioni differenti a seconda

- dello stato della richiesta
- della sua condizione di successo o fallimento

Chrome Developer Tools: breakpoint su...
- *scripts/callback.js:38, 42, 50, 55*

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

Scaricamento di dati in formato testo (3)

Uso di funzioni di callback…

- …interne (semplice accesso agli oggetti presenti nello scope della funzione, ma difficile manutenibilità e riuso del codice)

  http://localhost:8080/05_TecWeb/3_plaintext.html

- …esterne (**evitando tassativamente di** dichiarare oggetti XmlHttpRequest globali – perché?)

  http://localhost:8080/05_TecWeb/4_plaintext-external.html

```
/* VEDERE IL CODICE (E RELATIVI COMMENTI) NEL PROGETTO DI ESEMPIO */
```

Chrome Developer Tools: inspect Script…

- *scripts/callback-external.js:66*
- *scripts/callback-external.js:12*

…leggere bene i commenti!

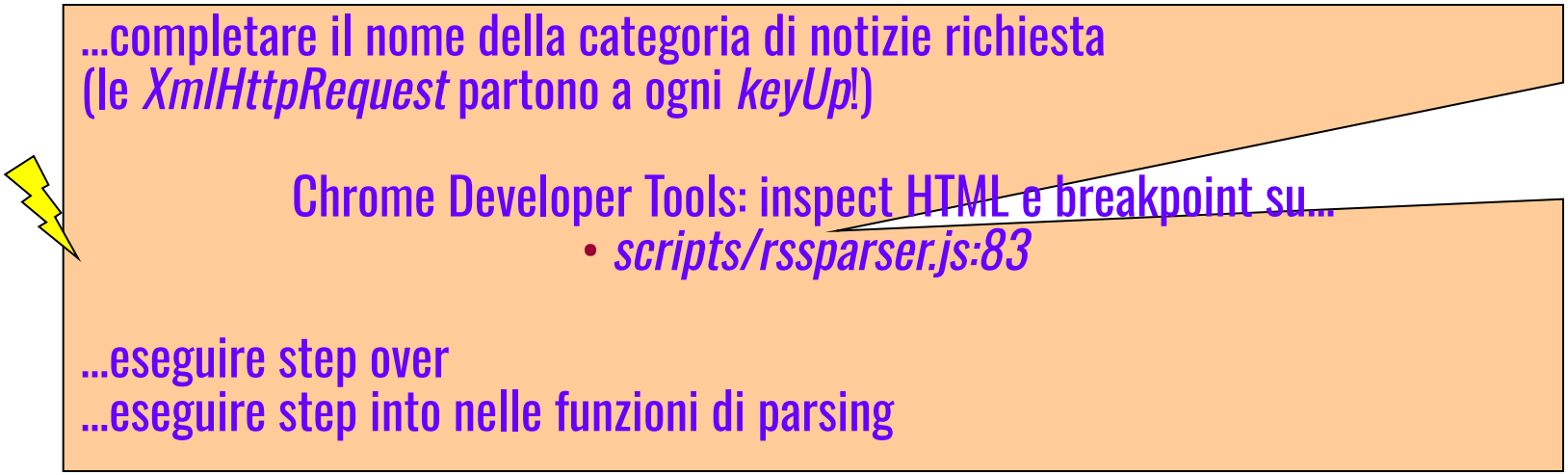
Scaricamento di dati in formato XML (1)

Utilizzo di un feed RSS, prodotto da una pagina JSP che legge da database implementato come Java Bean

- utilizzo di *responseXML* in caso di successo
- utilizzo di *responseText* in caso di errori o fallimenti

http://localhost:8080/05_TecWeb/5_rssfeed.html

```
/* VEDERE IL CODICE (E RELATIVI COMMENTI) NEL PROGETTO DI ESEMPIO */
```



Richieste AJAX cross-domain

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

- provate ad esempio a tentare l'accesso alla risorsa *testo.txt,* usata per il primo esempio, rendendola disponibile anche su un altro sito Web diverso dal dominio che AJAX sta utilizzando…

E per "leggere" gli RSS di un sito esterno (es: *Repubblica.it*) ?

- occorre recuperarli tramite logica server-side (Servlet, JSP, ma anche JSF, PHP, .NET, CGI in genere, …) che faccia da "proxy"

sito esterno → nostro Web server

→ risorse scaricate da URI del nostro sito

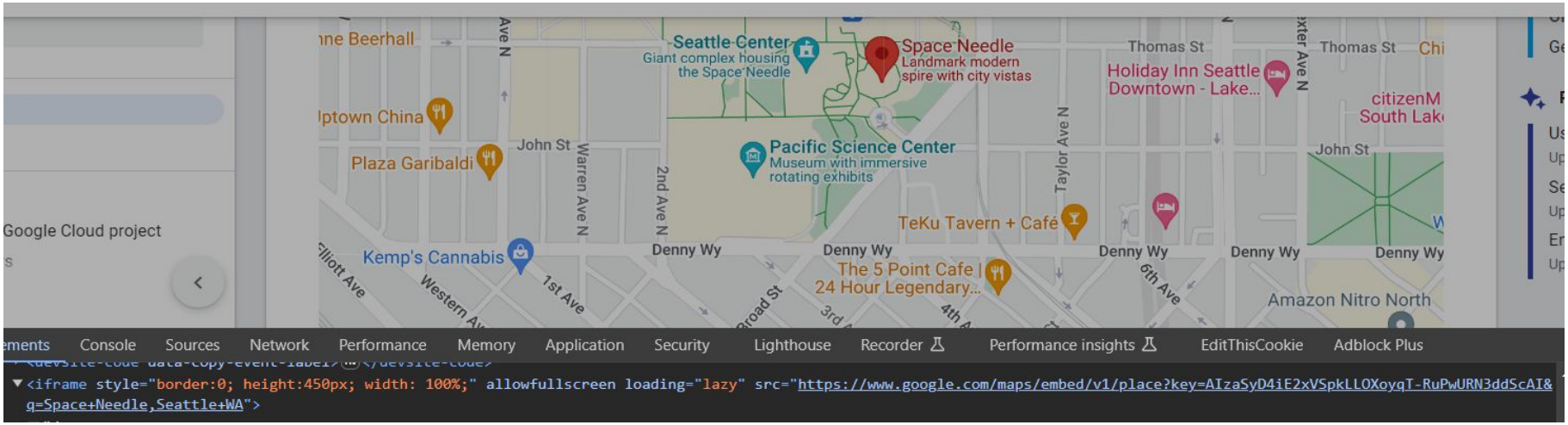
E le mappe di Google, allora?

Come è possibile che alcuni siti abbiano embedded le mappe di google?

- le mappe producono richieste AJAX verso i server di google…
- …ma sono su pagine scaricate da altri domini

Semplicemente **le mappe girano all'interno di *iframe***!

- le *XmlHttpRequest* dell'*iframe* sono dirette a Google…
- **...così come l'attributo *src* dell'*iframe* stesso**
- per approfondire**:** **documentazione qui**



Enhanced User Interaction: Terminazioni Forzate, …

L’utente abbandonato

Le richieste AJAX

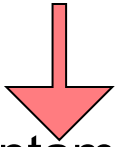
- permettono all’utente di continuare a interagire con la pagina
- ma non necessariamente lo informano di cosa sta succedendo
- e possono durare troppo!

L’utente non sa cosa fanno i nostri script

- …o gli insegniamo a usare Chrome Developer Tools
- …o facciamo in modo di informarlo noi!

È giusto interrompere le richieste che non terminano in tempo utile

- server momentaneamente sovraccarico
- problemi di rete



- disorientamento dell’utente

Richieste fantasma

È molto difficile generare una richiesta fantasma *ad hoc* scaricando solo contenuti statici dal Web server (sebbene in modo asincrono, dinamico…)

- caduta della connessione tra *readyState* 3 e 4… (bisogna essere svelti a staccare i cavi di rete ☺)
- scaricamento di un file di grosse dimensioni (difficile da distribuire in lab a causa dei problemi di spazio, quota, ecc… ☺ e non si può neanche farlo dal sito del corso… le richieste AJAX partono dai vostri *localhost*, ricordate?)

Capita anche troppo spesso, invece, quando si invoca logica server-side.

Uso smodato di AJAX genera un numero elevato di richieste verso il server!

ecco perché sostituiamo l'header ***connection=keep-alive*** con ***connection=close (quando reso possibile dall'implementaz browser)*** → dobbiamo riaprire la connessione ogni volta e andiamo più piano, ma il server non collassa a causa dell'esaurimento di connessioni disponibili in caso di accesso di molteplici utenti contemporaneamente.

Metodo *abort( )*

Interruzione delle operazioni di invio o ricezione

- non ha bisogno di parametri
- termina immediatamente la trasmissione dati

Per poterlo utilizzare in modo sensato, tuttavia, non si può richiamarlo in modo sincrono dentro la funzione di callback

- se *readyState* non cambia non viene richiamato!
- e tipicamente *readyState* non cambia quando la risposta si fa attendere

Si crea un'altra funzione da far richiamare in modo **asincrono** mediante il metodo ***setTimeOut****(funzioneAsincronaPerAbortire, timeout)*

- e al suo interno si valuta se continuare l'attesa o abortire l'operazione

Per provare… (1)

Pagine

http://localhost:8080/05_TecWeb/6_loadwait.html?wait=[secondi]

Richiamano una risorsa server-side (una servlet) che attende per il numero di secondi indicato prima di restituire il controllo e il risultato

Gli script nella pagina attendono comunque solo per 5 secondi

- viene mostrata un'immagine animata nell'attesa
- ed eventualmente un messaggio di fallimento se la risposta necessita di più tempo

```
/* VEDERE IL CODICE (E RELATIVI COMMENTI) NEL PROGETTO DI ESEMPIO */
```

Chrome Developer Tools: dalla scheda Script inserire breakpoint su...

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

...esecuzione step by step

Per provare… (2)

Pagine

http://localhost:8080/05_TecWeb/6_loadwait.html?wait=[secondi]

Notate l'**uso della console di Chrome Developer Tools per effettuare il log informazioni** relative alle attività in corso quando la richiesta viene abortita!

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

```
/* VEDERE IL CODICE (E RELATIVI COMMENTI) NEL PROGETTO DI ESEMPIO */
```

Chrome Developer Tools: dalla scheda Script inserire breakpoint su…

- *scripts/loadabort.js:124*

…controllare la console!

Nuovo Esercizio Proposto

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

Nell'ordine che preferite…

- estendete le funzionalità del JavaBean **FeedDb** per ottenere le categorie esistenti a partire dalla parte iniziale del loro nome
  public List<String> ***getCategories****(String categoryStartingWith);*

- spostate la funzione di **scaricamento dei feed** in XML dall'evento onKeyUp del campo di input a quello di **onClick** su un opportuno (prendete esempio dalle altre pagine dell'esercitazione di oggi)

- associate a **onKeyUp** (e scrivetela) una funzione AJAX che **scarichi il nome della prima categoria che inizia con le lettere immesse dall'utente** e sostituisca tale risultato (se presente) all'attributo value del campo di input (prendete esempio dalle altre pagine dell'esercitazione di oggi)

- scrivete infine la pagina JSP (o Servlet, se preferite) associata all'URL chiamato dalla vostra funzione AJAX che legga dai parametri in GET le iniziali della categoria cercata, interroghi il Java Bean e **restituisca la prima stringa ottenuta**

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

E ancora…

- aggiungete a WEB-INF/lib le librerie che trovate nell'archivio zip dell'esercitazione: parser/serializzatore **JSON** per Java (**Gson** o la più datata Jabsorb) e sue dipendenze (slf4j)

- scrivete una Servlet che, come la pagina feed.jsp, recuperi la categoria di notizie desiderata dai parametri della richiesta, interroghi il JavaBean e **restituisca il risultato mediante serializzazione JSON** («sbirciate» nella soluzione soltanto al momento di usare JSON)

- aggiungete la libreria Javascript **json.js** alle risorse Web del progetto (scegliete pure la posizione che preferite, ad esempio web/scripts/)

- modificate la pagina 5_rssfeed.html per a) riferire ANCHE la libreria json.js e b) **rivolgere la propria richiesta asincrona alla Servlet che avete appena realizzato** (ricordarsi di mappare la Servlet nel descrittore web.xml)

- modificate la funzione Javascript che analizza l'XML delle notizie per **analizzare**, al suo posto, **la stringa di testo JSON** restituita dalla nuova Servlet

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 immodificabili pre-esistenti (non occorre modificare il sorgente)
- Supporta oggetti di complessità arbitraria

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

https://github.com/google/gson

Gson in una slide

L'ultima versione di Gson è la 2.8.4

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