Universita' degli Studi di Bologna
Facolta' di Ingegneria

Anno Accademico 2008-2009

Laboratorio di Tecnologie Web
Highlights della soluzione del progetto DoItYourSelf

http://www-lia.deis.unibo.it/Courses/TecnologieWeb0809

RISCALDAMENTO

Prima di cominciare…

- Un nuovo target ANT: *update…ecc…ecc…*
  - le risorse **NON** caricate in memoria una tantum* (es: il bytecode delle servlet) e **NON** interpretate dal server all'avvio** (es: il descrittore *web.xml*) possono essere aggiornate direttamente sostituendone le versioni nel direttorio di deploy
    - script javascript, fogli di stile CSS, pagine HTML, pagine JSP, risorse, …
    - eventualmente pulire la cache del browser se non si notano cambiamenti
- L'altra volta abbiamo visto l'editing "a cuore aperto" di tali file…
  - piu' rapido: aperti e salvati direttamente nel direttorio di deploy con un text editor
  - possibile trovare la soluzione, salvo poi scordarsi di copiarla in Eclipse!!
- …oggi aggiungiamo/modifichiamo file dentro al progetto
  - salvataggio, deploy delle sole modifiche* **, (pulizia cache), verifica sul browser
  - iterativamente

Index_solved.html

- linee 8, 9
  - tag *meta* usato per comandare lo scaricamento della pagina dal server, senza fare cache e impostazione di una data di scadenza già passata, per sicurezza
- linea 16
  - import del file di utility javascript già visto durante le altre esercitazioni
    - myGetElementById, myGetXmlHttpRequest, …
- linea 19
  - attivazione di firebug lite (versione modificata: prima controlla che non sia già attivo il plugin firebug originale)
- linea 26
  - evento *onload* associato al body = primo momento utile per operare sul DOM
  - funzione javascript generica, con argomenti: non tarata su questa pagina

scripts/solution1,js

- linea 4:
  - a costo di scrivere codice verboso e poco efficiente, ricondursi dapprima alle funzionalità già note e collaudate per risolvere i problemi!
  - eventuali ottimizzazione solo in un secondo momento!
- linea 7:
  - esempio di utilizzo della console di firebug (o firebug lite)
  - utile se non si capisce dove è il problema, dove lo script “sbrocca”, ecc…
- linea 11:
  - sfrutiamo l’ *innerHTML* fino in fondo:
    - aggiunta dell’immagine come richiesto mediante un tag *img*
    - circondiamo l’immagine con un anchor *a* che punti alla prossima pagina

Caricamento di contenuto testuale mediante AJAX

Validazione di una form via Javascript

more_solved.html (1)

- linee 8, 9:
  - stesse considerazioni di prima sui problemi legati alla cache durante lo sviluppo
- linee 16, 19:
  - inclusione delle “commodity” javascript
- linea 27:
  - sfruttamento dell’evento *onload* del *body* per fare quanto richiesto
    - l’effetto è che la pagina contenga la form completa già all’avvio
    - se il contenuto del file non fosse statico, ma generato dinamicamente lato server si potrebbe ottenere form sensibili al contesto (es: identità dell’utente, parametri in sessione, altro, …)
  - aggiunta di un parametro casuale per simulare richieste a URI diverse e evitare la cache del browser
    - i tag *meta* valgono per la pagina corrente, non per le richieste AJAX
- linea 39:
  - aprite il file per vedere come è fatto dentro, ce l’avete no?

scripts/solution2.js

- Obiettivo:
  - l'unico modo di modificare la *form* a pagina caricata è via DHTML e script (js)
  - l'unico modo di farlo con contenuto proveniente dal server è AJAX
  - in particolare, in questo caso, mediante l'accesso alla responseText di una XHR

    → *TemplateAjax: callback-external.js*
- linea 95:
  - solita biforcazione: supporto AJAX sì/no
  - qui in realtà, anziché usare un *iframe*, si potrebbe scrivere un messaggio di errore nell' *innerHTML* dell'elemento obiettivo delle modifiche
- linee 62, 68:
  - solito modo di lanciare una XHR, guardiamo subito la funzione di callback
- linee 12, 31:
  - tutto si riduce a riuscire ad eseguire questa istruzione

more_solved.html (2)

- linee 49:
  - uso di un anchor privo dell'attributo *href*: serve solo per raccogliere l'evento *onClick* e lanciare la validazione (lo vestiamo da pulsante grazie ai CSS)
  - la funzione Javascript qui è invece cablata sulla struttura della form
    - niente argomenti (al limite potevamo passargli la *form* per intero)
    - deve "ben-conoscere" la struttura del DOM su cui opera perché deve applicare controlli diversi a campi diversi
    - la ricerca "estetica" della modularità ci complicava la vita, qui
  - in alternativa si potevano richiamare più funzioni nell'evento
    - passare a ciascuna i corrispondenti elementi obiettivo per il controllo e per il messaggio di errore
    - ottenere un unico risultato booleano mettendo in AND i risultati di tutte

scripts/solution3.js (1)

- Obiettivo:
  - valutare il contenuto dei campi *input* di una *form,* così come riempiti dall'utente
  - scrivere un messaggio di errore, per ciascun componente

    → *TemplateHTML: validate.html, validator.js*
- linea 2:
  - ottengo gli elementi obiettivo su cui operare come figli dell'unica *form* nel DOM
- linee 4,5, 7,8, 10,11:
  - seconda del *name* di ciascun figlio scateno un test diverso
  - l'elemento per il messaggio di errore lo posso conoscere solo per *id* invece
- linee 6, 9, 12, 14:
  - risultato *false* al primo errore, *true* se tutto fila liscio
  - si poteva eseguire comunque tutti i test e avere tutti i messaggi di errore, restituendo un booleano "accumulatore" del risultato solo alla fine

scripts/solution3.js (2)

- linea 17, 29:
  - a ogni funzione di test passo sia l'elemento sotto test che quello dove scrivere il messaggio di errore
- linea 36:
  - per il controllo dell'anno uso una espressione regolare (riferimenti sul sito del corso, nella sezione *Risorse*)
  - si poteva fare in 1000 altri modi, anche senza REGEX:
    - accettando solo stringhe di lunghezza pari a 4 caratteri
    - controllando, carattere per carattere, che si tratti di un valore in {0,1,…,8,9}
    - ecc

Invio di contenuto raccolto dal DOM mediante AJAX

Scrittura di una Servlet minimale

even_more_solved_partial.html

- Sorvolate, per ora, sul fatto che si tratti di una pagina JSP
  - inclusione di frammenti di codice Java (che verrà eseguito lato server a fronte della richiesta per la URI associata a questa pagina) al fine di avere campi della *form* consistenti durante la navigazione tra le pagine del progetto
- linea 39:
  - predisporre dapprima una qualche risorsa associata alla URI della XHR
    - anche solo un file di testo
      - i parametri della richiesta saranno ignorati
      - sarà comunque scaricato il contenuto del file
      - se si tratta di un file XML corrispondente alle specifiche si può continuare l'esercizio anche senza essere riusciti a fare la servlet!
    - in questo caso predisponiamo una servlet che non fa nulla di utile tranne esistere e verificare che *web.xml* e altre strutture siano tutte a posto
- linea 41:
  - funzione cablata sul DOM della pagina anche in questo caso
  - può servire per fare prima; eventuali rifiniture solo in seguito, se tutto gira

ServletPartialSolution.java

- Obiettivo:
  - rispondere a una richiesta HTTP con una servlet che svolga logica di business

    → *TemplateServlet: YourServer, web.xml*
- linea 29:
  - unifico la gestione di richieste GET e POST in questo modo
- linea 40, 41:
  - un problema alla volta; ora la servlet è su e risponde.
  - per ora tanto basta a lanciare una XHR
  - in seguito sistemeremo la servlet (o eventualmente cambieremo la URI della richiesta in quella di un file XML adatto al resto dell'esercizio, se non ci riusciamo)
  - poi cureremo il comportamento della funzione di callback della XHR

scripts/solution4_partial.js (1)

- Obiettivo:
  - estrarre del contenuto dal DOM della pagina e allegarlo come parametro a una richiesta AJAX
- linee 116, 117:
  - analogamente al controllo Javascript che validava la *form*, anche qui siamo invocati senza argomenti
  - per prima cosa estraiamo dal DOM i nodi che ci interessano
- linee 119 e ss.:
  - cicliamo su tali nodi e salviamo in variabili di comodo i dati che servono
- linea 128, 129 e ss.:
  - aggreghiamo il tutto in un XML valido secondo lo schema dato
  - **dichiaramo il namespace della radice a scopo futura validazione XSD !!!**
- linea 134:
  - passiamo la palla al solito insieme di funzioni relative ad AJAX

scripts/solution4_partial.js (2)

- Obiettivo:
  - lanciare una richiesta AJAX che invii contenuti al server

    → *TemplateAJAX*
- linea 99, 66:
  - rispetto al solito codice, questa volta dobbiamo portarci dietro anche gli argomenti da inviare
  - aggiungiamo semplicemente un parametro alla signature delle funzioni
- linea 74, 85:
  - richiesta **POST** (l'XML è "grosso", può superare il limite per la URI in GET)
  - necessario descrivere il contenuto del body della XHR con un **apposito header**
- linea 88:
  - escaping dei dati inviati per evitare caratteri speciali come "=", ecc…
- linea 27:
  - per ora non ci concentriamo sulla funzione di callback

Gestione di contenuti XML e della sessione attraverso una Servlet

Parsing di una risposta AJAX e utilizzo dei risultati

even_more_solved.html (1)

- linea 45:
  - invochiamo la servlet senza controllare i dati, anche se potremmo, per fare in modo che eventuali errori arrivino fino a lei e da lei siano riconosciuti
- linea 43:
  - un modo di combinare i due controlli javascript, per completezza
- linee 51, 54, 57:
  - le cose di cui si occuperà la servlet

ServletSolution.java

- linee 43,44,45:
  - ottenimento di un oggetto dalla sessione
  - sua inizializzazione se nullo (la prima volta)
- linea 48:
  - lettura di un parametro dalla richiesta
  - notate che non c'è bisogno di fare una *unescape( )*
- linea 49:
  - lettura di un parametro dal contesto dell'applicazione web
  - inizializzato nel descrittore *web.xml*
- linea 52:
  - procediamo scomponendo il problema top-down, una cosa alla volta
  - possiamo così lasciare in sospeso il validatore e completare la servlet
- linea 58:
  - risposta di tipo XML, non HTML
  - alcuni browser, tuttavia, non associano la risposta al campo *responseXML* della XHR se non è presente la riga di intestazione

MyValidator.java

- Obiettivo:
  - validare una stringa XML contro uno schema XSD disponibile a una URI nota

    → *TemplateXMLValidationPlus: XSD_SAX_Validator.java*
- linea 37:
  - unica modifica al codice originale: non leggo da un *FileReader*, ma da uno *StringReader*

even_more_solved.html (2)

- linea 64:
  - "…al posto di queste istruzioni…."
  - casualmente contrassegnate da un *id* esplicito, tramite il quale ottenere l'elemento da passare alla funzione di callback

scripts/solution4.js

- Obiettivo:
  - parsare un contenuto XML via Javascript e utilizzarlo per valorizzare dell'HTML

    → *TemplateAjax: rssparser.js*
- linea 65:
  - finalmente invochiamo la funzione di callback passandogli l'XML
- linea 72:
  - un modo per vedere cosa abbiamo scaricato se qualcosa va storto
- linee 15, 39:
  - dove è possibile sfruttiamo il fatto di poter scrivere direttamente l'HTML finale
- linea 20:
  - estrazione dei nodi di interesse dall'XML ricevuto
- linee 26-32:
  - estrazione dei dati di interesse e loro salvataggio in variabili di comodo
- linee 35-37:
  - scrittura della tabella HTML prelevando i dati dalle variabili di comodo (comode!)

Terminazione della sessione

even_more_solved.html (3)

- linee 76, 81, 83:
  - "…realizzarlo come si preferisce…." → io l'ho fatto in due modi:
    - *grezzo*: rimozione del cookie che mantiene l'id di sessione
      - via servlet o JSP (non implementato):
        - accesso ai cookie e cancellazione del cookie *JSESSIONID*
        - in alternativa, impostazione di una data di scadenza già passata
      - via javascript (implementato, ma…):
        - sono accessibili i cookie associati al dominio della URI visualizzata
    - *elegante*: rimozione delle strutture dati in sessione mantenute dal server
      - via servlet o JSP (l'oggetto session è predefinito)
- linea 78:
  - il server comunque associa un cookie di sessione al vostro browser
  - in caso di successo nelle operazioni di cui sopra cambia il suo valore, a seguito di nuove richiesta
- linea 85:
  - questo link con *href* vuoto serve soltanto a scatenare tali richieste
  - viene tuttavia ricaricata la pagina iniziale, non quella con il DOM già modificato!

scripts/solution5.js

- Non l'avevo mai fatto prima, ho cercato in rete…
  - però non gira ☹
  - non lancia errori, ma la modifica dura solo finché esegue lo script
  - libri, forum, tutorial, … tutti dicono di fare così…

↓

se qualcuno scopre cosa non va

sono ben contento di imparare una cosa nuova

- Ho lasciato questo file come **riferimento per avere routine di accesso (perlomeno in lettura) ai cookie via javascript**

ServletToInvalidateSession.java

- linee 39:
  - tutto qui

- linea 43:
  - nella risposta codifico una pagina HTML che resta visualizzata per 2 secondi
  - dopo 2 secondi dall'evento *onload* del *body* viene lanciata una funzione che ordina al browser di navigare indietro nella propria cronologia, accessibile via javascript
  - di fatto l'utente è riportato alla pagina da cui proveniva la richiesta