

dott. Longari Michele
Matricola: 180852
Ingegneria informatica spec.
E-mail: longari@ce.unipr.it

Integrazione della piattaforma Ajax in un sistema web multi-agente realizzato col framework applicativo Spring™

*Progetto realizzato sotto la guida dell'ing. M. Mari per i corsi di:
Ingegneria del Software B (prof. A. Poggi)
Laboratorio di Ingegneria Informatica (prof.ssa M. Mordonini)*



1. Ajax e Spring: perché cercare di integrarli?

Prima di addentrarci nella spiegazione del progetto, forse è meglio chiarirsi le idee su cosa si è cercato di fare, e soprattutto perché.

Partiamo descrivendo brevemente cosa sono Ajax e Spring.

Parlare di Ajax corrisponde a parlare di un insieme di tecnologie sfruttate in sinergia per la costruzione di pagine web dinamiche, secondo la filosofia della trasmissione asincrona tra applicazioni web.

Non si tratta di qualcosa di sostanzialmente nuovo, ma di un naturale progresso di un insieme di tecniche preesistenti e già consolidate, integrate insieme attraverso l'introduzione di una serie di specifiche necessarie per rendere questa tecnologia quanto più universale e standardizzata possibile.

L'acronimo del nome (Asynchronous JavaScript and XML) mette già in primo piano le due principali tecnologie utilizzate da Ajax, ossia JavaScript e XML, e il fatto che questa tecnica sia basata sulla trasmissione asincrona per lo sviluppo di applicazioni interattive.

Spring, invece, è un framework leggero per lo sviluppo di applicazioni J2EE. In particolare, Spring è uno strumento di sviluppo molto potente per quanto riguarda lo sviluppo di applicazioni web efficienti utilizzando i mezzi forniti dalle Servlet/Portlet.

Ecco quindi chiarito, prima di tutto, il legame primitivo tra Spring e Ajax: il Web.

Integrare Ajax con Spring, vuol dire dare il via alla possibilità potenziale di creare applicazioni web sofisticate e con una portabilità praticamente infinita.

Si possono creare applicazioni web sempre più vicine ad un comportamento “*desktop-like*”, che riescono a far interagire *client* e *server* in maniera completamente asincrona, alleggerendo così in maniera importante il lavoro da fare lato *client*.

Combinando i due framework, inoltre, è possibile fare in modo che da semplici pagine .jsp, si possa accedere a metodi e variabili di classi remote Java, in maniera bidirezionale, dando spazio ad infinite possibilità di progettazione.

E' chiaro quindi, come questo tipo di ricerca, si collochi in un ambito di primaria importanza per la creazione di applicazioni web sempre più potenti e solide.

2. Le prime vie intraprese per l'integrazione

Prima di iniziare a parlare in maniera dettagliata del progetto e della sua fase d'implementazione, c'è da dire che la fase che in assoluto ha richiesto più tempo è stata quella introduttiva di ricerca, che ha avuto il fine di trovare il framework Ajax specifico, che ottimizzasse la risoluzione del problema nelle parti di fondamentale importanza del progetto.

Questa parte ha richiesto particolare tempo e lavoro poiché, attualmente, i framework utilizzabili per sviluppare applicazioni web con Ajax sono davvero molti (è meglio evitare un inutile e ridondante elenco), e tutti fondamentalmente validi e funzionali.

Così, prima di trovare il framework che facesse al caso nostro sono state dovute provare numerose vie, che poi si sono rivelate non molto indicate alla nostra situazione.

La prima idea che è stata presa in considerazione, è stata quella ovvia di provare a vedere se Spring non offrisse già di per sé una sorta di primitiva possibilità di integrazione con Ajax. In effetti questa possibilità è presente, infatti con il rilascio di Spring Web Flow (SWF) 1.0 RC3 (Release Candidate 3), è stata prevista una possibile integrazione con Ajax corredata anche da un esempio applicativo.



Purtroppo questo non è sufficiente per il nostro *use-case*, essendo quest'ultimo un sistema web multi-agente che utilizza Spring Framework 1.2.8. Quindi, il passaggio a SWF, avrebbe voluto dire cambiare l'impostazione globale del sistema, e quindi *bypassare* i requisiti del progetto.

Dopo questo primo tentativo, la successiva idea affrontata, è stata quella di trovare, dopo ricerche puntuali, quale fosse il framework Ajax più usato e "famoso" in rete, e successivamente provare ad usarlo per i nostri fini: eccoci quindi approdati a Dojo. Dojo mette a disposizione una serie di componenti che rendono i siti più utilizzabili, interattivi e funzionali. Dojo è in assoluto uno dei framework più utilizzati per Ajax, e velocizza il tempo di sviluppo di pagine dinamiche lato client. Anche in questo caso però, dopo un'attenta fase di testing, si è osservato che se pur ottimo a livello di funzionalità, Dojo non possedeva espressamente quelle caratteristiche che si stavano cercando. In particolare, si sono trovate difficoltà nell'implementare l'invio asincrono di oggetti remoti, come possono essere i *Beans* di Spring. Non è escluso che una soluzione plausibile al problema potesse essere trovata, però sicuramente avrebbe richiesto maggiore quantità di lavoro e di codice da scrivere.

Allora, dopo ancora svariate ricerche sulla rete, sono stati presi in considerazione gli strumenti di TIBCO, in particolare TIBCO General Interface™ (GI). TIBCO General Interface™ è un framework di sviluppo avanzato basato su Ajax e RIA (Rich Internet Application), ed integra il supporto per componenti Ajax e di terze parti commerciali. TIBCO General Interface, in pratica, è un'affidabile soluzione di sviluppo grafico Ajax che offre oltre 90 controlli a interfaccia grafica (GUI) pre-configurati, nonché funzionalità di comunicazione, dati, servizi eventi "*publish and*

subscribe”, registrazione, debug e altro basati su Ajax. Sulla guida ufficiale TIBCO, è stato trovata la spiegazione *passo-passo*, di come effettuare una possibile integrazione tra Spring e Ajax usando GI. Di per sé i passaggi da effettuare non erano di particolare difficoltà, però comunque vi erano da apportare alcune piccole modifiche al sistema Spring. Questa motivazione, aggiunta al fatto che GI non è uno strumento Open Source, hanno portato ad abbandonare anche questa strada.

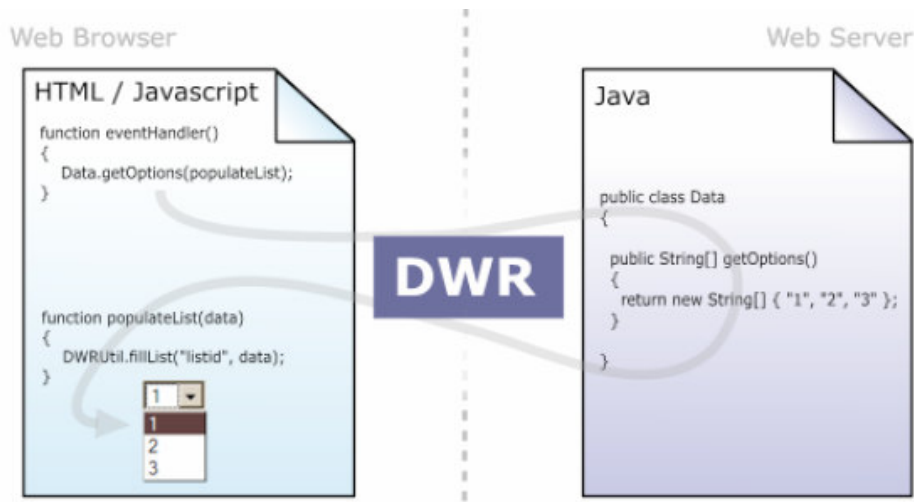
Ma dopo questi vari tentativi infruttuosi, eccoci finalmente arrivati a trovare, quello che sicuramente si è rivelato lo strumento migliore che si potesse scegliere per lo scopo di questo progetto: **DWR**, *Direct Web Remoting*.

3. DWR: Ajax e Spring “made easy”

DWR è un framework che serve per utilizzare Ajax e l’oggetto *XMLHttpRequest*, in maniera facile e *diretta*.



Esso semplifica le chiamate JavaScript, a codice residente lato *server*, permettendo inoltre, tramite un’appropriata configurazione XML, di trasformare classi Java remote lato *server*, in oggetti JavaScript accessibili liberamente lato *client*. Chiaramente questa funzionalità si è rivelata di grandissimo interesse per il tipo di applicazione che si stava sviluppando, in quanto permette un’integrazione praticamente perfetta con Spring e i suoi *Beans*.



Nella figura sopra, è spiegato graficamente, il modo in cui opera, a livello web, DWR, in particolare nella sua interazione *web-server/browser*.

4. Come effettuare l'integrazione tramite DWR

Una volta scelto definitivamente di usare DWR come strumento di integrazione Ajax-Spring, i passi che è stato necessario fare per rendere compatibile il sistema per questo tipo di operazione, sono stati relativamente pochi e semplici.

Eccoli elencati passo-passo, così da renderli maggiormente chiari e conseguenti.

Per prima cosa occorre dichiarare la servlet DWR nel file `web.xml` del sistema Spring:

```
<servlet>
  <servlet-name>dwr</servlet-name>
  <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>dwr</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

Da notare che è stato impostato come valore di default a *true* il parametro di *debug*. Questo fa sì che una volta avviato il nostro server locale (nel nostro caso *Apache Tomcat*) per fare il testing applicativo, è sufficiente andare nella cartella <http://localhost:8080/NomeApplicazione/dwr/>, per vedere le classi che al momento sono state impostate dall'utente per essere usate col *Direct Web Remoting*. Questo sarà più chiaro più avanti quanto verranno date le indicazioni su come aggiungere nuove classi al dominio delle soluzioni di DWR.

Sempre in `web.xml`, è stato necessario settare un *Listener*, che specificasse il file di configurazione XML, in cui fosse possibile trovare la definizione delle le classi remote di Spring (*Beans*). Nel nostro *use-case* di progetto (*AgentTrip: TRavel Information Portal*), il file specifico era quello di sotto riportato:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/AgentTrip-servlet.xml</param-value>
</context-param>

<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

Seconda operazione importante da fare, è stato creare nella cartella /WEB-INF, il file `dwr.xml`.

Esso contiene sia i “*creatori*” per le classi che si vuole far gestire a DWR, sia eventuali “*convertitori*”, per queste o altre eventuali classi di sostegno. Nel progetto usato come esempio applicativo, il file `dwr.xml` è stato così costruito:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE dwr PUBLIC
"-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
"http://www.getahead.ltd.uk/dwr/dwr20.dtd">

<dwr>
<allow>
  <create creator="new" javascript="JavaDate">
    <param name="class" value="java.util.Date"/>
  </create>

  <create creator="new" javascript="UserProfileManager">
    <param name="class" value="... .UserProfileManager"/>
  </create>

  <create creator="spring" javascript="PlanningTrip">
    <param name="beanName" value="planningTripFormController"/>
  </create>

  <convert converter="bean" match="... .PlanningTripFormController">
  <convert converter="bean" match="... .User"/>
  <convert converter="bean" match="jade.core.AID"/>

</allow>
</dwr>
```

Come si può vedere dal codice evidenziato, per il nostro progetto è stata usata la nuova release 2.0 di DWR, che presenta notevoli funzionalità in più rispetto alla 1.0. Sono inoltre messi in evidenza, due tipi di *creator*, di fondamentale importanza; il primo che crea in automatico un codice JavaScript (`javascript="NomeFile"`) eseguibile lato *client*, a partire da una normalissima classe Java; il secondo che esegue la stessa operazione, a partire però da una classe remota di Spring.

E' inutile spiegare le potenzialità di quest'ultima operazione, a prima vista così semplice e banale.

Il *converter* invece è usato per quelle classi remote, che, nella fase di *debug*, vengono contrassegnate come necessarie di conversione per un corretto uso con DWR e i file JavaScript da esso automatizzati.

Una volta effettuate queste operazioni, per usare metodi di classi remote precedentemente definiti nei file di configurazione, non si dovranno fare altre operazioni complicate, bensì basterà includere i file .js creati da DWR nei template .jsp, e accedervi normalmente come a qualsiasi metodo usato in classi e file .java.

Per esempio, nel nostro caso progettuale, nel file `login.jsp`, sono state aggiunte nell'intestazione queste semplici righe di codice:

```
<script src='dwr/interface/UserProfileManager.js'></script>
<script src='dwr/interface/JavaDate.js'></script>
<script src='dwr/engine.js'></script>
<script src='dwr/util.js'></script>
```

I primi due script inclusi sono quelli creati in automatico da DWR; gli altri due sono delle utility che servono al corretto funzionamento del sistema.

Per accedere ai metodi, sempre nel file .jsp sopra indicato, è stata scelta questa semplice via:

```
<input class='ibutton'
      type='button'
      onclick='UserProfileManager.getUserames(reply2);'
      value='Who?'
      title='Who is an user of AgentTrip?'/>

<script type='text/javascript'>
  var reply2 = function(data)
  {
    if (data != null && typeof data == 'object')
      alert(DWRUtil.toDescriptiveString(data, 2));

    else
      DWRUtil.setValue('d2', DWRUtil.toDescriptiveString(data, 1));
  }
</script>

<span id='d2' class='reply'></span>
```

In questo pezzo di codice di esempio, viene eseguito l'accesso al metodo evidenziato, ed il risultato ritornato dalla funzione, viene visualizzato con un messaggio di *alert*. Naturalmente è anche possibile visualizzare i risultati come messaggi di testo, nonché eseguire esclusivamente un metodo, evitando di scrivere gran parte delle righe di codice qui sopra, e lasciando solo quelle riguardanti il tipo di script usato, e l'accesso al metodo remoto.

5. Conclusioni

Nella relazione, non è stato precisato che gran parte dei test effettuati su DWR, è stato fatto con l'uso di oggetti Java, in taluni casi anche piuttosto complessi, che venivano semplificati notevolmente nel loro utilizzo grazie ai finissimi automatismi di DWR. Inoltre molti oggetti Java "tradizionali", come le Liste e gli Array per esempio, presentano delle visualizzazioni di *default* ottimamente già implementate, che tra l'altro sono state usate anche in alcune parti di questo progetto.

Un'altra funzionalità molto importante di DWR, che è stata osservata attraverso lo studio effettuato, è che questo framework è in grado di passare in maniera del tutto nascosta ed automatica, la sessione *Http* in corso, ai metodi remoti delle classi Java. Nella pagina .jsp non sarà necessario fare altro che invocare il metodo Java che

necessita tra i suoi argomenti della *HttpSession*, e DWR penserà da solo a far sì che a tale metodo arrivi la giusta sessione e che possa essere utilizzata senza problemi. Concludendo, dopo quello che è stato possibile fare usando gli strumenti forniti in abbondanza da DWR, di sicuro si può dire che questo framework presenta potenzialità molto importanti ed utili per sviluppare applicazioni web sempre più sicure, affidabili e veloci.

6. Bibliografia e riferimenti

Ecco, per eventuali chiarimenti o desideri di maggiori informazioni specifiche, i riferimenti al materiale che è stato utilizzato per la realizzazione intera di questo progetto:

- <http://it.wikipedia.org/wiki/AJAX;>
- http://www.javaworld.com/javaworld/jw-06-2005/jw-0620-dwr_p.html;
- <http://ajaxian.com/archives/spring-web-flow-10-released-with-ajax-integration;>
- <http://java.sun.com/developer/technicalArticles/J2EE/AJAX/;>
- <http://bram.jteam.nl/?p=2;>
- <http://www.gridshore.nl/blog/index.php?/archives/29-Doing-ajax-with-ajaxtags-and-springframework.html;>
- <http://www-128.ibm.com/developerworks/java/library/j-ajaxportlet/?ca=dgr-jw22DWR4Ajax;>
- <http://www.theserverside.com/tt/articles/article.tss?l=AjaxandSpring;>
- <http://getahead.ltd.uk;>
- <http://getahead.ltd.uk/dwr/getstarted;>
- [http://getahead.ltd.uk/dwr/server/spring.](http://getahead.ltd.uk/dwr/server/spring;)