UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea

Il Paradigma REST per lo sviluppo di applicazioni Web 2.0

Anno Accademico 2007/2008

relatore:
Ch.mo prof. Domenico Cotroneo

correlatore:
Ch.mo prof. Marcello Cinque

candidato:
Antonio Alonzi
matr. 831/000205

Indice

Introduzione

1 Introduzione al Web
1.1 La nascita del web
1.2 Tecnologie lato server
1.2.1 P
HP e ASP
1.2.2 ASP.NET
1.2.3 J
SP e caratteristiche fondamentali
1.2.4 JSP: Pattern Model-View-Controller
1.2.5 J
SP: I Filter
1.2.6 J
SP: I Listener
1.3 Tecnologie lato client
1.3.1 JavaScript
1.3.2 DOM (Document Object Model)
1.3.3 JavaScript per migliorare un sito
1.3.4 JavaScript per facilitare form

2 Il Web 2.0
2.1 Introduzione al Web 2.0
2.2 AJAX
2.2.1 A
jax: oggetto XMLHttpRequest
2.2.2 G
estire un sito con AJAX
2.2.3 I
Framework: Dojo
2.3 I Web Services
2.3.1 Introduzione ai Web Services
2.3.2 S
OAP e WSDL
2.3.3 A
xis2 e Pojo

3 REST

3.1 Introduzione a REST
3.2 Il concetto di Risorsa
3.3 L'architettura orientata alle Risorse
3.4 HTTP e similitudini con ROA
3.5 Progettare secondo REST
3.6 REST e RPC a confronto
3.7 Dettagli implementativi
3.8 Vantaggi e Svantaggi
3.9 Soluzioni per l'accessibilità tramite browser

4 Esempi

4.1 Controllo Login PHP-Ajax
4.2 ParliamoDArte: Elenco Dinamico JSP-Ajax
4.3 ParliamoDArte: WebServices, ServerMeteo e ClientMeteo
4.4 MeteoMondo: REST

Conclusioni

Bibliografia

Introduzione

Il Web è cambiato.

L'evoluzione ha portato in questi ultimi anni a sviluppare molte applicazioni web come dei *web service*. Ma in realtà i web service non hanno nulla a che fare il Web. Non fanno altro che aggiungere complessità alle applicazioni senza ricavarne reali vantaggi.

Il Web è una *rete di risorse di informazioni*, quindi vogliamo una tecnologia che utilizzi il Web per ciò che è. L'ottimale sarebbe avere servizi enormemente scalabili come **Google search engine** che, a partire da una stringa, effettua una ricerca in database e restituisce come risposta un elenco di risorse seguendo una precisa formattazione. Il servizio offerto dai motori di ricerca di Google non è visto dagli utenti come una Procedura Remota da invocare (visione RPC) ma è visto ancora come una risorsa, come un web site.

Un web site è semplice se le sue API sono facilmente interpretabili sia da utenti che da programmi. Per ottenere applicazioni semplici bisogna utilizzare strumenti elementari come HTTP, XML, le URI, che, se utilizzati rispettando dei particolari principi, portano allo sviluppo di applicazioni secondo il paradigma REST.

REST è un'insieme di regole che designano un nuovo stile di progettazione. Programmare secondo questo nuovo paradigma significa seguire uno stile architetturale orientato alle risorse (ROA). Alla base di REST c'è il protocollo HTTP scelto proprio per la sua semplicità. HTTP anche se non è il protocollo più utilizzato in rete (il traffico maggiore è quello legato all'invio di e-mail o all'uso di applicazioni Peer2Peer come BitTorrent), è il protocollo che ha più rilievo dal punto di vista commerciale proprio perché è il protocollo del Web.

Possiamo distinguere due tipi di servizi sulla rete: i *web services* e i *Web Services* (a volte, identificati anche con l'aggettivo Big per esaltarne le differenze). I Big Web Services sono quelli per cui è indispensabile definire delle funzioni remote, poiché è molto più comodo utilizzarli richiamando funzioni RPC. Ma spesso si abusa di queste funzionalità e si tende ad implementare ogni tipo di servizio come web service in maniera inappropriata.

In generale i web services si portano dietro alcune complessità: è difficile eseguire un debug del servizio, aumentano il carico del protocollo aggiungendo ulteriori informazioni ai dati effettivi, diminuiscono la compatibilità con altre applicazioni.

Ma spesso vengono preferiti perché hanno un enorme vantaggio: con IDE avanzati è possibile costruire un web services con un semplice click del mouse. Gli IDE nascondono la complessità dei web services, mostrando una falsa somiglianza con le applicazioni REST; ma ciò non vuol dire che la complessità non esiste.

I primi componenti che si accorgono della complessità dei web services sono le CPU e le bande di trasmissione. Ma questi sono molto economici, quindi oggi non sono il primo problema.

Quindi il punto cruciale dei web services rimane la *scalabilità*. Più i servizi sono semplici più sono scalabili.

Spesso accade che questi servizi vengono utilizzati da nuovi client, i quali non erano stati per nulla presi in considerazione al momento della scrittura del servizio. L'astrazione dei web services non è mai perfetta, infatti essi aggiungono dei vincoli che un client deve rispettare per costruire richieste compatibili con i server (protocolli SOAP e WSDL). Da ciò si deduce che l'unico modo per ottenere una completa compatibilità tra client e server è utilizzare semplici protocolli standard.

REST si oppone ai web service e tenta di risolvere questi problemi. Nella sua prima definizione (di Roy Fielding) si nota un'estrema semplicità, ma allo stesso

tempo sono lasciate molte domande senza risposta dando ampia scelta a chi dovrà implementare l'applicazione. Infatti Roy Fielding non dice come fare per creare un'applicazione REST né quali linguaggi utilizzare, ma enuncia delle proprietà che essa deve avere per essere catalogata tale. REST gode di un'interfaccia standard proprio perché si basa su un unico protocollo, HTTP, e questo gli da una compatibilità assoluta nel mondo del web. E' giusto affermare che REST non è uno standard per produrre applicazioni web, ma è basato su standard, quindi qualunque applicazione sviluppata secondo REST sarà facilmente esportata su qualunque tipo di hardware e sarà semplice interoperare con nuovi client.

Ecco una breve descrizione dei capitoli:

- CAPITOLO PRIMO

In questo capitolo si tratterà la storia del Web, dalle origini (nascita dell'HTML) alla nascita di linguaggi lato-server per la creazione di pagine dinamiche e la comunicazione con i database. Infine tratterà linguaggi lato-client per modificare il contenuto di una pagina in base a scelte di utenti o per eseguire piccole funzionalità sui client.

- CAPITOLO SECONDO

Il secondo capitolo mostra come i Web Services hanno modificato il modo di programmare web (Web 2.0) e di come essi possono aggiungere funzionalità ad un'applicazione web, mostrando i vantaggi rispetto alle normali tecniche di programmazione. Inoltre viene mostrato come incorporare i web services nelle preesistenti applicazioni web con tecnologie di programmazione AJAX.

- CAPITOLO TERZO

Questo capitolo tratta il paradigma REST: quali sono le proprietà che si devono rispettare per creare un'applicazione REST, cos'è l'architettura orientata alle

risorse, quali sono gli standard che si utilizzano e quali sono i vantaggi e gli svantaggi di questo nuovo modello di progettazione.

- CAPITOLO QUARTO

In quest'ultimo capitolo sono presentati degli esempi che evidenziano i vantaggi di AJAX, dei Web Services, e di REST in sostituzione ai Web Services.

Capitolo 1
Introduzione Al Web

1.1 La Nascita del Web

[2] Com'è nato il Web?

Rispondere a questa domanda non è banale in quanto il web come lo intendiamo noi oggi è molto diverso da come era stato ideato.

Oggi, per noi, è normalissimo accedere ad un nostra casella di posta elettronica tramite il web; quindi collegarci ad un sito internet, digitare il nostro nome ed una password, ed accedere ai tanti servizi offerti dal server.

Ma in realtà il web è nato in maniera molto diversa.

Quando Tim Berners-Lee (fisico del CERN di Ginevra) nel 1989 ha inventato l'**HTML** (*HyperText Mark-up Language* **[1]**), il suo unico scopo era quello di creare un linguaggio semplice che rendesse agevole la visualizzazione dei dati.

L'HTML non è altro che un linguaggio di markup, ossia descrive il contenuto, testuale e non, di una pagina web. La descrizione della pagina è fatta con dei **TAG** che individuano degli elementi nel documento.

Una volta scritto, un documento viene inserito su un Web Server e, tramite il protocollo **HTTP** (*HyperText Transfert Protocol*), passato alla macchina Client che ne ha fatto richiesta. Questo ha portato non solo alla nascita dei Server ma anche dei browser, programmini installati sui Client per agevolare l'invio di richieste HTTP per pagine web e costruire un'illustrazione grafica dei documenti in risposta.

Il primo browser a interfaccia grafica fu MOSAIC, che poi divenne un prodotto commerciale acquistato dalla *Netscape Communication*.

Il web divenne presto un prodotto amato dagli utenti, quindi anche la Microsoft cercò di conquistare questo nuovo mercato lanciando un proprio browser: *Internet Explorer*.

Nacquero tanti siti internet e presto ci si rese conto che mantenere un sito scritto

completamente in HTML è una cosa abbastanza costosa.

Infatti l'HTML “è un'unica pagina”, nel senso che al suo interno contiene informazioni fini solo a mostrare se stessa. Se abbiamo bisogno di creare una serie di pagine “omogenee”, con uno stesso stile, il puro HTML non basta più. E' così che nascono i **CSS** (*Cascading Style Sheet* **[3]**) che separano completamente la presentazione di una pagina dai dati della stessa.

Nei CSS si prendono degli elementi (per id) o degli insieme di elementi (per classi) a cui applicare dei particolari attributi, come colori, dimensioni, margini etc... . Poi, includendo il foglio di stile .css in tutte le pagine in cui si vuole mantenere quel design, si riesce a creare un vero e proprio stile ricorrente in tutto il sito internet che può essere modificato cambiando solo una volta (nel .css) alcuni valori, e non tanti valori uguali riportati su tutte le pagine.

Infine dai semplici siti, che permettevano solamente di visualizzare pagine sul proprio computer, sono iniziati a nascere dei siti che permettevano di memorizzare informazioni su database e di accedere a queste informazioni tramite web.

Questo all'inizio venne implementato con dei programmini CGI *(Common Gateway Interface)* che affiancavano il server, ma poi vennero presto sostituiti a dei veri e propri linguaggi di *scripting* che nacquero proprio per questo: **PHP** e **ASP**.

1.2 Tecnologie lato server

1.2.1 PHP e ASP

PHP (*Personal Home Pages* **[5]**) e **ASP** (*Active Server Pages* **[7]**) sono due linguaggi di programmazione web lato-server. Il Server è un programma software che gira su una macchina remota rimanendo in ascolto su determinate porte e raggiungibile da un Client (paradigma client-server). Alcune operazioni devono essere compiute lato-server perché richiedono l'accesso a informazioni o funzionalità non disponibili sul client, oppure per misure di sicurezza che sarebbero inaffidabili se eseguite lato client.

I due linguaggi consistono nell'inserimento, all'interno delle semplici pagine HTML, di istruzioni che permettono di personalizzarle e modificarle dinamicamente a seconda delle "azioni" di un utente. Le pagine adesso contengono, oltre a tutti i TAG definiti dal linguaggio HTML, anche dei nuovi TAG speciali, i quali identificano l'inizio del codice che deve essere interpretato ed eseguito.
Il server non invierà quindi il documento HTML così com'è al client che ne ha fatto richiesta, ma passerà il documento ad un preprocessore, che si occuperà di interpretare il codice, creando una pagina ad hoc per ogni richiesta.
I due linguaggi nominati in questo capitolo sono molto simili tra loro.
Entrambi permettono di gestire alcuni aspetti molto importanti per poter iniziare a scrivere un'applicazione web.

PHP e ASP dispongono di tutti i costrutti dei consueti linguaggi di programmazione: dichiarazione di variabili; costrutti condizionali (*if* e *switch*); costrutti ciclici (*for*, *while*, *do while*); possibilità di scrivere funzioni (*function* e *subroutine*); possibilità di importare codice scritto in altre pagine (*include*).
Inoltre entrambi i linguaggi supportano la programmazione ad oggetti, quindi è possibile scrivere classi personalizzate e utilizzare una progettazione ad oggetti.
PHP è un linguaggio nato su server Apache su sistemi Unix. E' un software completamente open source ed è nato come una raccolta di script CGI Perl. Per questo la

sua sintassi deriva sia dal Perl che dal C++.
Il codice è inserito direttamente nel contenuto HTML della pagina distinguendo i *tag* php da quelli HTML con questa sintassi:

```
<□□□□ #□□□□□□□□□□ □□□ □>
```

ASP è un linguaggio messo a punto dalla Microsoft e che gira su server IIS. Il server è anch'esso distribuito in maniera gratuita ma è molto difficile programmare senza l'ausilio di IDE come Microsoft Visual Studio. La sua sintassi è derivata dal Visual Basic e, come per il PHP, il codice è iniettato direttamente nelle pagine HTML con appositi *tag*.

```
<% '□□□□□□□□□□ □□□ %>
```

Per inviare informazioni da client a server in maniera semplice e pulita si fa l'utilizzo dei **form** definiti nell'HTML. Un *form* non è altro che una collezione di dati creata lato-client poi spedita lato-server ad uno script che ne interpreta il significato e utilizza i dati ricevuti.
I dati possono essere inviati utilizzando due differenti metodi HTTP il metodo **GET** e il metodo **POST**.
Nel caso di GET si accodano delle informazioni aggiuntive all'URL della risorsa richiesta (avendo dei limiti di lunghezza sulla URL) mentre nel caso di POST le informazioni vengono inviate nel payload della richiesta HTTP (quindi in maniera non visibile all'utente e non avendo limiti nella lunghezza dei dati).
In entrambi i casi ciò che viene inviato è una stringa, un'associazione del tipo *nome=valore* con opportune codifiche per i caratteri che non possono essere inseriti nelle URL così come sono.
Entrambi i linguaggi forniscono metodi semplici per recuperare i dati inviati e utilizzarli a loro piacimento.

In genere il metodo POST si utilizza quando si riempono dei form e si vogliono spedire al server.

Il metodo GET invece può risultare molto utile quando ci si vuole riferire ad una risorsa parametrizzata. Per esempio, si può pensare ad una pagina "*/elenco.php*" che restituisce un elenco di risorse memorizzate su un database a cui ci si può accedere mediante un'altra pagina "/risorsa.php" ma solo indicando esplicitamente l'id della risorsa da visualizzare. Quindi con le query string e il metodo GET si può pensare ad un modo semplicissimo: per ogni oggetto da visualizzare nella pagina *"/elenco.php"* inserire un link del tipo:

<□ □□□□="/□□□□□□□·□□□□□□=1">□□□□□□□□1</□>

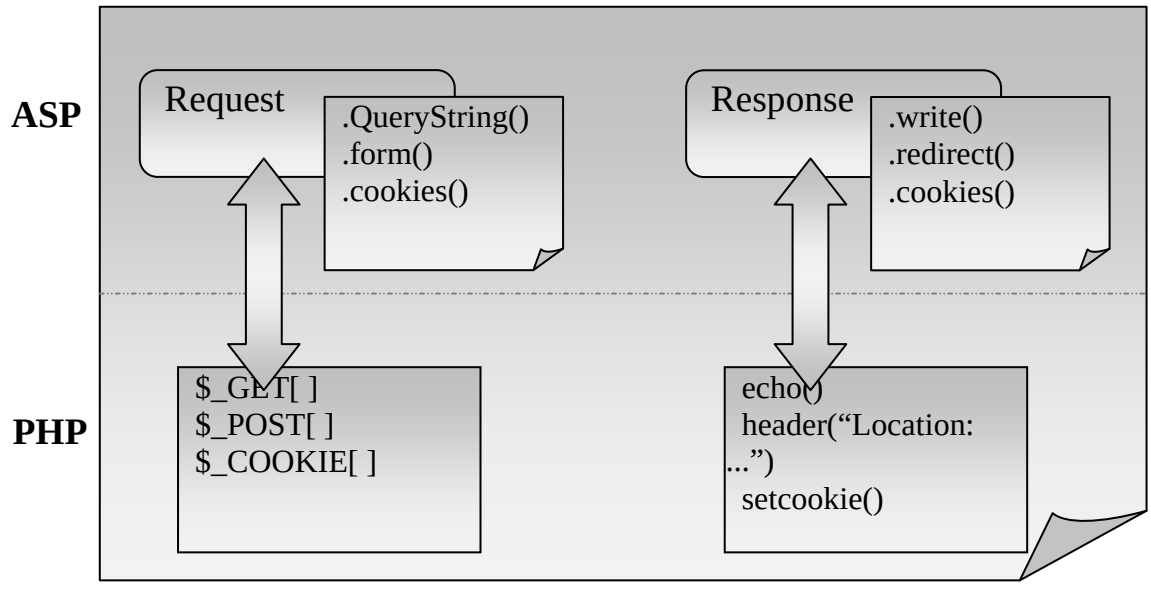
Fare la stessa cosa utilizzando il metodo POST sarebbe molto complicato, infatti si dovrebbe creare per ogni oggetto non un link ma un form con un input nascosto, e legare l'invio della richiesta ad un bottone di tipo *submit* che invii la richiesta.

Come già detto, il web si basa sul protocollo HTTP che, essendo "stateless", non permette di mantenere uno stato di connessione tra Client e Server necessario per gestire un'applicazione.
Questo problema viene egregiamente risolto da PHP e ASP creando degli oggetti **Session**.
All'arrivo di una richiesta HTTP al server, quest'ultimo controlla se nella request c'è un Header del tipo "*Session: value*". Se non è presente (oppure è presente ma il valore non è valido), il server genera un nuovo codice di sessione (in genere un codice di 32 caratteri che possono essere lettere o numeri), lo memorizza in un database associando ad esso un nuovo oggetto sessione, infine nella risposta HTTP indica al Client la necessità di creare una sessione tramite un altro Header del tipo "*Set Session: value*".
In questo modo il Client memorizza questo valore in un cookie e allega questo valore ad ogni ulteriore richiesta che farà al server.
D'altra parte il server quando, controllando gli header delle request, si ritroverà un valore noto dell'ID di una sessione, capirà che si tratta di un utente che ha già la sua sessione, quindi potrà immagazzinare delle informazioni associandole a lui o recuperare informazioni precedentemente memorizzate.

Esiste una differenza sostanziale tra ciò che vengono definiti l'oggetto **Session** e l'oggetto **Application**. Entrambi permettono di memorizzare semplicemente dei dati nel contesto della nostra applicazione, ma nel primo caso abbiamo un'istanza per ogni utente. Ciò significa che un utente può visualizzare e modificare solamente i dati collegati al proprio oggetto *Session*.

L'oggetto *Application* invece è messo a disposizione dal server per permettere agli utenti di condividere i dati tra di loro. Questo oggetto viene creato allo start-up del server e il suo ciclo di vita termina contemporaneamente al server. Tutti gli utenti possono utilizzarlo e (sempre con politiche decise dal programmatore) visualizzare e modificare il suo contenuto.

Per gestire l'invio di dati tra client e server (in ASP) si fa l'uso degli oggetti **Request** e **Response**. Tramite l'oggetto *Request* è possibile accedere ai parametri inviati nella richiesta (sia tramite il metodo GET che il metodo POST) oppure ai cookie della richiesta. Tramite l'oggetto *Response* è possibile inserire dei cookie da inviare al client, ridirigere la richiesta o modificare il contenuto della pagina. In PHP non abbiamo dei reali oggetti Request e Response ma ci vengono offerti dei metodi e delle variabili, visibili in tutti i nostri script, che si comportano in maniera molto simile agli oggetti ASP.



Un'altra funzionalità enormemente utile di questi linguaggi di programmazione è la connessione a **database**. I database sono un elemento essenziale per la memorizzazione in modo sicuro di dati persistenti. In genere la connessione viene inizializzata una sola volta, con un utente fittizio, e poi viene richiesta ed utilizzata da tutti gli utenti che necessitano reperire o inserire dei dati nelle tabelle. La sintassi per effettuare *query* è molto semplice; questo rende molto veloce la stesura di script che estraggono informazioni da tabelle e ne visualizzano il risultato in pagine HTML.

1.2.2 ASP.NET

Il framework .NET è una tecnologia composta da un insieme di classi che possono essere usate mediante molti linguaggi di programmazione a oggetti (C#, VB.NET, Visual C++, J#) e che possono essere compilati in bytecode.
[8] ASP.NET fa parte del framework ed è stato ideato perché lo sviluppo di un'intera applicazione in ASP portava notevoli svantaggi nella manutenzione del codice e nella gestione di applicazioni molto grandi.

Per evitare ciò, ASP.NET propone una:

- gestione delle applicazioni inserendo i file già compilati in una cartella *App_Code* e i file dei database in una cartella *App_Data*. Questo porta ad una netta separazione tra codice HTML e codice ASP.NET (classi che implementano la business logic).
- una progettazione e sviluppo delle pagine mediante le *Master Pages* che costruiscono un aspetto comune a più pagine ottenendo una netta separazione tra codice ASP.NET e codice XHTML (*eXtensible HTML* nato per uniformare l'HTML con l'XML);
- uso di particolari controlli come *Gridview* che permettono di editare tabelle e

TreeView che visualizzano dati da file XML e realizzano semplici mappe del sito;

- una semplice gestione dei dati;
- incremento della sicurezza grazie all'aggiunta di supporti per l'autenticazione e l'autorizzazione degli utenti: *Login, Password Recovery, LoginName, LoginStatus, LoginView*.

1.2.3 JSP e caratteristiche fondamentali

[9] JSP è un linguaggio nato nel 1998, molto dopo l'introduzione dei vecchi linguaggi PHP e ASP. E' l'acronimo di *Java Server Pages*, ciò ci fa intuire che rispetta una sintassi derivata dal Java. Come i vecchi linguaggi le pagine sono principalmente scritte in HTML con all'interno dei TAG che identificano del codice JSP che deve essere interpretato da un preprocessore.

```
<% // □□□□□□ □□□ %>
```

I maggiori vantaggi del JSP sono:

- la portabilità su qualunque tipo di piattaforma, vista sia lato server che lato client, in quanto il java può essere portato facilmente su sistemi windows, unix, e anche su piccoli PDA;
- la possibilità di riutilizzare il codice con l'uso dei *Bean* che rendono molto più facile lo sviluppo software e la manutenzione del codice;
- un facile mantenimento di una sessione HTML;
- velocità nelle risposte e richiesta (la JVM su cui è in esecuzione in server è sempre caricata in RAM).

Un punto di forza del linguaggio JSP sono le Servlet.

Le servlet sono delle “pagine” che non si occupano più di mostrare dei dati, e quindi di

restituire delle pagine HTML, ma il cui compito è quello di eseguire funzionalità sul server. Quindi non c'è più necessità di scrivere delle pagine con codice misto, formato da template HTML interrotto da piccole righe di codice JSP che eseguono quelle funzioni, ma le servlet sono proprio le funzioni. Sono scritte interamente in Java e staccano completamente la logica dall'applicazione, consentendo di dividere i lavori: i programmatori programmano le Servlet Java, i Web Designer programmano le pagine JSP.

Un server JSP è essenzialmente una servlet. Ossia un'applicazione Java in esecuzione su una JVM residente su un server capace di servire delle richieste.
Il ciclo di vita di una servlet è formato da tre passi fondamentali:

- *Init()*: segna la nascita di una servlet, è un metodo richiamato una sola volta e che si occupa dell'inizializzazione delle risorse;
- *Service()*: si occupa di servire tutte le richieste che arrivano;
- *Destroy()*: segna la fine del server, si occupa di memorizzare tutte le informazioni utili ad un prossimo caricamento, e di deallocare tutte le risorse.

Una volta caricato il server (come accadeva nei linguaggi precedenti), quando si fa una richiesta di una pagina .jsp questa viene passata al preprocessore che interpreta il codice JSP e lo esegue.
Anche in JSP è possibile utilizzare i costrutti *if*, *while*, etc... che sono sintatticamente identici al Java.

Esistono oggetti molto simili a quelli enunciati per i precedenti linguaggi.
L'oggetto **Request** permettere di accedere alle informazioni di intestazione del protocollo HTTP oppure ai parametri passati nei form sia tramite GET che tramite POST. Il server può facilmente leggere al momento della richiesta (oltre agli *input*) dati come *cookies*, *metodo HTTP* utilizzato, *porta* o *ip* del client o del server e altro.
L'oggetto **Response** permette di inviare i risultati dell'esecuzione della pagina JSP al client.

L'oggetto **Session** permette di recuperare valori memorizzati in una sessione o memorizzare nuovi valori in essa.
L'oggetto **Application** permette di recuperare valori globali o memorizzarne nuovi. Cosa molto comune è memorizzare i *Bean* nell'Application in modo da dare l'opportunità a tutti di invocarne i metodi in maniera molto semplice.
Altri oggetti che negli altri linguaggi non erano dei veri e propri oggetti ma solo delle funzioni (java invece essendo un linguaggio ad oggetti deve inglobare ogni funzione in oggetto) sono: l'oggetto **Out** che con i suoi metodi di *print* e *println* permette di scrivere direttamente sulla pagina HTML, l'oggetto **PageContext** rappresentante il contesto di esecuzione della servlet e l'oggetto **Exception** che serve per gestire le pagine di errore.
Questi oggetti possono essere direttamente utilizzati senza istanziarli nelle pagine JSP.
Se invece invece ci troviamo in una *Servlet* questi oggetti possono essere ricavati dal *ServletContext* che è un dato già noto appena creiamo la *Servlet*, oppure dalla *request* o *response* che sono passati alla servlet.

1.2.4 JSP: Pattern Model-View-Controller

[10] Il pattern *Model-View-Controller* è sicuramente il più utilizzato nell'ambito della programmazione Web. Sebbene il codice JSP potrebbe essere iniettato direttamente nelle pagine HTML, non è questo un modo consono di lavorare. Infatti la nostra pagina HTML (come dice stesso il nome) è scritta in un linguaggio di markup che dovrebbe contenere solo codice per la visualizzazione dei dati, mantenendo quanto più possibile la logica dell'applicazione al di fuori di essa.

Il pattern consiste nel suddividere la nostra applicazione in tre parti principali:

- il **Model**: che rappresenta il “modello di funzionamento” della nostra applicazione, ossia la nostra “business logic”.
- la **View**: che rappresenta la “vista” della nostra applicazione, designa tutto ciò che deve essere visualizzato all'utente e di come questo debba essere fatto.

- Il **Controller**: che, appunto, “controlla” le nostre interazioni tra il modello dei dati e l'interfaccia mostrata all'utente.

Applicando un certo stile di programmazione è possibile progettare la nostra applicazione seguendo questo pattern anche quando usiamo linguaggi di programmazione come PHP o ASP, ma ciò risulta molto più semplice grazie alla definizione di tre strutture completamente diverse per programmare le tre parti: i **Bean**, le **JSTL**, e le **Servlet**.

- I **Bean** per implementare la parte **Model**

Quando si progetta un'applicazione è bene porre le prime attenzioni sui dati che si voglio gestire. I *Bean* possono essere un buon punto di partenza per un lavoro di progettazione.
Essi sono degli oggetti java che si occupano di memorizzare i dati utili alla nostra applicazione e di fornire i metodi necessari per implementare le funzionalità volute.

Come prima cosa quando si progetta un'applicazione web in JSP si devono ricercare tutte le entità che l'applicazione deve gestire. Un esempio di entità da gestire è un “Utente”. Quindi si scriverà un Bean contenente informazioni sull'utente: il suo nick, la password ed altro. Poi si devono pensare a tutte le funzioni che si vogliono svolgere sull'entità “Utente” ad esempio l'inserimento dell'utente in un database, la modifica delle informazioni dell'utente, la cancellazione dell'utente dal database, etc...
Una volta completata la stesura della classe Utente che rappresenterà il nostro Bean, la cosa più comune è creare un'istanza del Bean allo start-up del server e memorizzarlo nell'oggetto *Application* (in genere creando un pool di connessioni ad un database e fornendo con il bean dei metodi per accedere al database).
Adesso quando vogliamo eseguire una delle azioni presenti nel bean non dobbiamo far altro che recuperare il nostro Bean dall'oggetto Application e invocare su di esso i nostri metodi.

In questo modo abbiamo creato una netta separazione tra la *business logic* e tutto il resto dell'applicazione. Infatti la nostra applicazione (il nostro Model) non è nient'altro che l'insieme di tutti i nostri Bean. Ora non ci resta che fornire all'utente un'interfaccia per

utilizzarli.

- Scrivere le **View** usando **JSTL**

Come abbiamo già detto le View non sono altro che una descrizione di un'interfaccia da mostrare all'utente finale per utilizzare le nostre funzionalità descritte nel Model.

Le viste devono essere scritte in JSP puro. Includere del codice Java in una vista potrebbe non essere un'idea ottimale perché questo riduce la leggibilità del codice.
Chi scrive codice HTML e JSP potrebbe essere benissimo un Web Designer che non conosce (ma che soprattutto non ha bisogno di conoscere) un linguaggio di programmazione come il Java.

Inoltre inserire codice Java nella view potrebbe indurre anche all'errore di aggiungere un po' di logica dell'applicazione nella vista, il che sarebbe scorretto secondo il pattern Model-View-Controller.

Nonostante questo, il JSP ha bisogno delle istruzioni e costrutti dei comuni linguaggi di programmazione come il costrutto **if** oppure i cicli **for**. L'attenzione che bisogna fare è nell'usarli semplicemente per visualizzare dati e non per eseguire operazioni su di esse.
A questo scopo sono nati i **JSTL**, (JavaServerPages Standard Tag Library). Questi sono dei tag personalizzati da inserire nelle pagine per compiere alcune azioni.

I tag più utilizzati sono sicuramente questi:

> `<□:□□□ □□□□□="□□□□□□" />`
>
> Utilizzato nella nostra pagina per stampare un valore che non conosciamo a priori ma che si trova memorizzato in una variabile (ad esempio un parametro inviato dal server oppure una variabile memorizzata nell'oggetto Session etc..). La prima cosa importante che questa funzione fa (e che quindi viene preferita all'utilizzo della funzione *print* invocata sull'oggetto *out*), è di codificare correttamente il contenuto della variabile nella pagina HTML. Infatti le pagine HTML contengono dei caratteri speciali (ad es. “<”, “/”) per definire i tag che se fraintesi porterebbero ad un'interpretazione errata di tutto il documento.

```
<□:□□ □□□□="□□□□□□□□□□□□" />
  // □□□□□□ □□□□ □/□ □□□
</□:□□>
```

Utilizzato quando si deve visualizzare qualcosa di diverso a seconda del dato che viene inserito nella pagina JSP. Funziona come un comune if di un linguaggio di programmazione. Se ci sono più scelte si può utilizzare un costrutto simile ad uno *switch case* che in JSTL è *<c:choose>* etc...

```
<□:□□□□□□□ □□□□□="□□□□" □□□="□□" />
  // □□□□□□ □□□□ □/□ □□□
</□:□□□□□□□>
```

Utilizzato quando si devono eseguire delle operazioni su elementi di un vettore. Come i forEach dei comuni linguaggi di programmazione *vett* è il vettore che si vuole scansionare *el* rappresenta per ogni ciclo un'istanza del vettore.
Questo è molto utile quando si devono visualizzare elenchi di dati e quindi per ogni istanza dell'elenco eseguire determinate operazioni (per esempio inserire una riga in una tabella).

Ci si può riferire alle variabili che devono essere manipolate (nelle pagine JSP) con una sintassi molto semplice: le "**Expression Language**".
Un *Expression Language* è un'espressione che permette facilmente di accedere a qualunque variabile presente nel pageContext.
Essa è racchiusa in una sequenza di caratteri: *${···}* e si utilizza ad esempio nel campo value di c:out, nel campo test di c:if o altrove.
Si può accedere ai parametri della pagina con *pageScope*, a parametri della request con *requestScope*, a parametri dell'oggetto Session o Application con *sessionScope* o *applicationScope*, ai cookie, agli header o ai parametri della richiesta HTTP con *cookie, header, headerValues, param, paramValues.*

Ecco alcuni esempi di Expression Language:

```
${□□□□□·□□□□□□□□□□}
```

```
// □□□□□□□ □□ □□□□□□ □□□ □□□□□□□□□□
□□□□□□□□□□

${□□□□□□·□□□□□□·□□□□□□} // □□□□□□□ □□
□□□□□□ □□□ □□□□□□
```

Per utilizziare le Expression Language nel campo test di c:if bisogna scrivere un'operazione che ritorna un booleano. Ecco le operazioni che si possono utilizzare: +, -, *, / (o *div*), *%* (o *mod*), *&&* (o *and*), || (o *or*), *!* (o *not*), *empty* (per vedere se una stringa è vuota o uguale a null), == (o *eq*), *!*= (o *ne*), < (o *lt*), > (o *gt*), <= (o *le*), >= (o *ge*).

C'è da notare che JSTL non si può usare così com'è da subito, per utilizzarlo è necessario reperire su internet il file jstl.jar (che contiene le librerie per i nostri tag di default) e inserirlo nella cartella /WEB-INF/lib che si trova sul nostro server.
Adesso dobbiamo inserire in testa alle pagine JSP in cui usare JSTL questa riga:

```
<%□ □□□□□□ □□□□□□=”□”
□□□=”□□□□://□□□□:□□□:□□□/□□□/□□□□/□□□□”%>
```

Questa non è l'unica libreria esistente, ne esistono tante altre ad esempio per costruire siti in diverse lingue, per formattare date, numeri o valute secondo la convenzione del paese dell'utente, per recuperare dati direttamente da database senza passare per i Bean (ovviamente da utilizzare solo in lettura per rispettare il pattern) e altro ancora.

- Le Servlet per implementare la parte Controller

Una **servlet** non è altro che una classe che estende la classe *javax.servlet.http.HttpServlet* ed impementa l'interfaccia *javax.servlet.Servlet*.

```
□□□□□□ □□□□□ □□□□□□□□□ □□□□□□□
□□□□□·□□□□□□□·□□□□·□□□□□□□□□□□
□□□□□□□□□□ □□□□□·□□□□□□□·□□□□□□□
```

Questo fa si che possiamo, e dobbiamo, ridefinire due funzioni:

```
□□□□□□□□□ □□□□
□□□□□(□□□□□□□□□□□□□□□□□□ □□□□□□□,
```

```
□□□□□□□□□□□□□□□□□□□ □□□□□□□□)
□□□□□□□□□ □□□□
□□□□□□(□□□□□□□□□□□□□□□□□□ □□□□□□□,
□□□□□□□□□□□□□□□□□□□ □□□□□□□□)
```

doGet viene invocata quando viene richiesta la servlet con metodo HTTP GET, dualmente funziona *doPost*. Entrambe le funzioni ricevono come parametri la request (ossia la richiesta che è stata fatta dal client) e la response (che rappresenta la risposta che dovrà essere inviata al client).
La prima cosa che si fa è recuperare il ServletContext

```
□□□□□□□□□□□□□□□ □□ = □□□□□□□□□□□□□□□□□□();
```

indispensabile per recuperare le variabili memorizzate nell'oggetto Application oppure per fare il dispatching della richiesta.

Come abbiamo già detto una Servlet deve gestire il Controller, questo può essere visto come il “cuore” dell'intelligenza dell'applicazione. Lo scopo di una Servlet è recuperare tutti i dati inviati dall'utente, validarli, e se tutto è corretto eseguire una sequenza di operazioni (invocando uno o più Bean dipendentemente dalla complessità dell'operazione). L'ultima cosa da fare è dare un risultato all'utente.

Per restituire un risultato ad un utente (essendo un risultato ancora una volta una vista) ciò che si fa è un dispatching della richiesta HTTP ad una pagina JSP (molto probabilmente avendo inizializzato dei determinati parametri). Quindi la vista mostrerà il risultato, sfruttando i parametri per personalizzarlo, ma senza conoscere ciò che la Servlet fa e come lo fa.

Le Servlet sono molto simili alle normali pagine JSP, nel senso che vengono invocate direttamente dal browser dell'utente senza che quest'ultimo sappia di stare invocando una Servlet, e non come i Bean che non sono mai invocati direttamente dall'utente ma sono sempre controllati dalle Servlet.

Per essere invocata da un browser la Servlet deve essere mappata su un URL, quindi per l'utente non è altro che una risorsa da invocare.

1.2.5 I Filter

I **Filter** (come dice stesso il termine) sono dei meccanismi per filtrare le richieste fatte da un utente. Il luogo più comune di utilizzarle è quello di voler limitare la visualizzazione di pagine (o comunque l'utilizzo di funzionalità) o risorse web ad alcuni utenti.

L'idea è di dividere gli *utenti* in *classi di utenti*, e assegnare ad ogni classe funzionalità limitate. La prima classificazione da fare è quella di *utente abilitato* e di *utente non abilitato*, quindi la prima funzionalità da offrire ad un utente è *l'abilitazione* (in genere, registrazione login e logout). Poi si possono creare classi diverse (pensando a forum la classe amministratore e la classe utente, oppure sviluppando siti di vendita on-line le classi acquirenti e venditori, etc...).

Ogni classe può quindi accedere ad un gruppo di risorse, e l'appartenenza ad una classe significa avere dei requisiti ben precisi.
La prima cosa che verrebbe in mente di fare è ciò che si faceva con i vecchi linguaggi di programmazione PHP e ASP. Creare, quindi, una Servlet che, in base a dei parametri inviati mediante form (login e password), autenticano un utente e settano una variabile nell'oggetto Session del tipo *autenthicated = true*.

Ora per limitare l'accesso a delle risorse si può pensare di usare un <*c:if*> (o qualche struttura JSTL simile) e vedere se l'utente è autenticato oppure no, quindi agire di conseguenza.

```
<□:□□□□□□>
  <□:□□□□
□□□□="□□□□□□□□□□□□□·□□□□□□□□□□□□□□==□□□□">
    <!-- □□□□□□ □□□ □□□□ -->
  </□:□□□□>
  <□:□□□□□□□□□>
    <!-- □□□□□□ □□ □□□□□□ -->
  </□:□□□□□□□□□>
</□:□□□□□□>
```

Risolvere il problema in questo modo non è l'ideale, infatti ancora una volta non abbiamo rispettato il pattern Model-View-Controller. Stiamo facendo un copia-incolla del codice su tutte le pagine del sito, il che comporta sempre errori e difficoltà nella manutenzione del codice.
Inoltre in questo modo non possiamo controllare l'accesso ad ogni risorsa, ma soltanto alle risorse JSP e al massimo alle Servlet. Se per esempio volessimo limitare l'accesso ad un'immagine, ci dovremmo limitare ad includere le immagini in dei file JSP e inserire le righe di codice prima di mostrare l'immagine, con il problema che l'utente, venendo a conoscenza in qualche modo dell'URL dell'immagine, può semplicemente digitarlo nella barra degli indirizzi e richiederla senza essere bloccato.

La maniera corretta ed efficace di risolvere questo problema è creare un filtro. Un filtro è altro che un oggetto Java che implementa la classe Filter. Implementare la classe Filter significa anche ridefinire un metodo: il metodo *doFilter()*.

```
□□□□□□ □□□□□ □□□□□□ □□□□□□□□□□ □□□□□□
 {□□□□□□ □□□□ □□□□□□□□(□□□□□□□□□□□□□□□
```

```
□□□□□□□□,
        □□□□□□□□□□□□□□□□ □□□□□□□□□,
□□□□□□□□□□□□ □□□□□□)
        □□□□□□ □□□□□□□□□□□□,
□□□□□□□□□□□□□□□□□□
    {···
     }
  }
```

Una volta creato il filtro deve essere eseguito un *mapping* che indica come le servlet nel file *web.xml*. Eseguire un *mapping* significa associare ad un inizio di un path (una cartella sul nostro filesystem) il nostro filtro (quindi la nostra classe contente il codice).
Ora basta semplicemente mettere tutte le risorse, cui vogliamo limitare l'accesso, nella cartella e gestire le politiche di accesso esattamente come facevamo prima.
Ad esempio creiamo una cartella “private” e all'interno mettiamo tutte le nostre pagine jsp (ma possiamo mappare anche le servlet in modo tale che l'URL inizi con la parola chiava “private/”) o qualunque altra risorsa che non vogliamo rendere visibile a tutti.
Ora il server ogni volta che un utente tenta di accedere ad una qualunque risorsa che si trova nella cartella private (quindi non solo pagine jsp ma più in generale risorse, che prima non potevamo bloccare perché non potevamo iniettare del codice in esse), sa che ci sono dei requisiti da rispettare per poter accedere alla risorsa, quindi esegue il codice nella funzione doFilter() della classe filter che (previa autenticazione) provvederà a passare la richiesta alla risorsa giusta. Quindi, una volta progettato bene il filtro, possiamo completamente dimenticarci di esso quando continuiamo la nostra programmazione.

Il codice di *doFilter()* può essere banalmente, come prima, un controllo sulla variabile authenticate dell'oggetto Session che abbiamo inizializzato al momento del login, oppure si possono pensare a scenari molto più complessi ma che hanno un'evoluzione molto più elegante.

1.2.6 I Listener

Un **listener** è un componente software istanziato (come i filter) all'avvio del server e che si mette in ascolto di particolari eventi reagendo con determinate azioni.

Esistono diversi tipi di listener anche se due sono i più utilizzati: *ServletContextListener* e *HttpSessionListener*.

- ServletContextListener:

Un listener di questo tipo ascolta gli eventi di creazione e distruzione dell'applicazione, quindi è utilizzato per gestirne il ciclo di vita. Per implementare questo filtro bisogna creare una classe che implementi l'interfaccia *ServletContextListener*:

```
□□□□□□ □□□□□ □□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□ □□□□□□□□□□□□□□□□□□□□□
```

e che quindi implementi i suoi due metodi astratti:

```
□□□□
□□□□□□□□□□□□□□□□□(□□□□□□□□□□□□□□□□□□
□ □□□)

□□□□
□□□□□□□□□□□□□□□□□(□□□□□□□□□□□□□□□□□□
□□□)
```

Implementare il primo metodo significa scrivere del codice che indichi al server cosa si vuole che esso faccia appena viene creato, dualmente il secondo metodo indica cosa si deve fare alla chiusura del server.

In genere, ciò che si fa allo start-up del server è istanziare tutti i Bean necessari alla nostra applicazione (quindi creando anche le nostre connessioni a database se necessario) e poi memorizzarli nell'oggetto Application rendendoli visibili a tutti.
Ugualmente il codice che implementerà l'evento distruzione del server, dovrà verificare che tutte le risorse aperte vengano deallocate correttamente, e se è stato aperto qualche file

in scrittura chiuderlo correttamente con la sua funzione in modo che tutti i dati vengano realmente scritti in esso e non si perda informazione.

L'oggetto Application si recupera dal *ServletContextEvent*, che è passato ad entrambi i metodi, in questo modo:

```
□□□□□□□□□□□□□□ □□□□□□□□□□□ =
□□□·□□□□□□□□□□□□□□□□□□□();
```

Creare un Bean e inserirlo nell'oggetto application è un'operazione molto semplice:

```
□□□□□□ □□ = □□□ □□□□□□();
□□□□□□□□□□□□·□□□□□□□□□□□□□□("□□□□□□□", □□);
```

Se la nella nostra applicazione ci sono dei parametri che dovrebbero essere passati dall'esterno al nostro server (come ad esempio l'url dei driver di database, utente e password o altro) possono essere scritti nel file di configurazione web.xml e letti alla creazione del server proprio nel listener.

Per memorizzare un valore nel file web.xml si usa la seguente sintassi:

```
<□□□□□□□-□□□□□>
   <□□□□□-□□□□>□□□□□□□□□□□□□□□</□□□□□-
□□□□>
   <□□□□□-□□□□□>□□□□□□□□□□□□□□□□□</□□□□□-
□□□□□>
</□□□□□□□-□□□□□>
```

Per recuperare tale valore nel listener si usa invece:

```
□□□□□□ □□□□□□1 =
□□□□□□□□□□□·□□□□□□□□□□□□□□□□□("□□□□□□□□
□□□□□□");
```

- HttpSessionListener:

Un listener di questo tipo serve per gestire correttamente le sessioni.

Per creare il suddetto listener bisogna costruire una classe che implementi l'interfaccia HttpSessionListener:

```
□□□□□□ □□□□□ □□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□ □□□□□□□□□□□□□□□□□□
```

e che quindi implementi i suoi metodi astratti:

```
□□□□ □□□□□□□□□□□□□□(□□□□□□□□□□□□□□□□
□□□)

□□□□ □□□□□□□□□□□□□□□□(□□□□□□□□□□□□□□□□
□□□)
```

Il primo metodo è invocato ogni volta che viene creata una nuova sessione (quindi ogni volta che un utente richiede una pagina jsp o una servlet non inviando un Entity Header valido nella sua request), mentre il secondo metodo viene invocato ogni volta che viene distrutta una sessione (ogni volta che si richiama in una pagina jsp o in una servlet il metodo *invalidate()* sull'oggetto Session, in genere in una pagina di *logout*).

Entrambe le funzioni passano come argomento un oggetto del tipo *HttpSessionEvent* da cui è possibile ricavare l'oggetto Session invocando su di esso la funzione *getSession().*

Un uso molto comune che si fa di questo listener è un contatore di quanti utenti stanno visitando il sito. Fare ciò risulta molto semplice.

Alla creazione del server (quindi sfruttando l'altro listener), si aggiunge una variabile nell'oggetto Application *int contatore = 0*, indicante che nessun utente è connesso al server (volendo migliorarci si può aggiungere anche un vettore di stringhe che memorizza tutti i nomi degli utenti).

Adesso non resta che metterci in ascolto sull'evento *SessionCreated* e, ogni nuova sessione

creata, recuperare la variabile contatore dall'oggetto Application incrementandola (eventualmente al momento del login inserire il nome dell'utente nel vettore di nomi).
L'ultima cosa che ci rimane da fare è metterci in ascolto sull'evento *SessionDestroyed* e ogni volta che ciò accade decrementare la variabile contatore (e casomai eliminare dal vettore dei nomi il nome dell'utente che era connesso).

C'è da notare che l'evento SessionDestroyed non è richiamato solamente esplicitamente dal programmatore (ossia quando implementa la funzione *logout*) ma viene richiamato anche quando una sessione risulta "scaduta". In genere la durata della sessione è di 30 minuti (ma questo valore può essere cambiato nella configurazione a nostro piacimento) e indica che se per 30 minuti non arrivano più richieste da parte di quell'utente la sessione viene considerata scaduta... altrimenti ad ogni richiesta viene avanzata nel tempo la data di scadenza.

E' bene fare attenzione alla durata della nostra variabile Session e al numero di dati che inseriamo in essa. Il server crea un oggetto Session per ogni utente quindi se abbiamo sessioni molto lunghe (ad es. sessioni che durano 12 ore), al suo interno memorizziamo tanti dati (1Mb di dati) e la nostra applicazione ha un'affluenza di 1.000 utenti al giorno, ci troveremo con:

*1·000 * 1□□ = 1□□*

1 Gb di dati occupati solamente per memorizzare tutti gli oggetti Session, il che potrebbe non essere adeguato.

E' comunque buona norma realizzare funzioni di logout che permettono ad un utente di rilasciare le risorse prima dello scadere della sua sessione, questo porta vantaggi sia al server, che si trova a gestire un numero minore di sessioni, ma anche al client, che deallocando le risorse non rischia di subire attacchi di *Session hijacking*.

Anche l'utilizzo dei listener deve essere esplicitamente indicato nel nostro file di configurazione web.xml con questa sintassi:

```
<□□□□□□□□>
   <□□□□□□□□-□□□□□>
      □□□□□□□□·□□□□□□□□□□□□□□□□
   </□□□□□□□□-□□□□□>
</□□□□□□□□>
```

ma utilizzando strumenti come eclipse ciò viene fatto automaticamente.

1.3 Tecnologie lato client

1.3.1 JavaScript

[11] Il ***JavaScript*** è un linguaggio di scripting lato-client introdotto per la prima volta dai programmatori del browser Netscape. All'inizio questo codice si chiamava *LiveScript* proprio perché cercava di rendere la pagina HTML più "viva". Poi è stato rinominato JavaScript acquisendo molto dalla sintassi Java.

Il JavaScript è un codice che può essere scritto sia direttamente nelle pagine HTML oppure può essere semplicemente incluso in esse.
Quando si digita una URI nella barra degli indirizzi il browser fa una richiesta di GET per una pagina su un server. Se la pagina è HTML allora inizia ad interpretarla riga per riga. Quando trova tag che fanno riferimento a file esterni (immagini, script, fogli di stile e altro) compie delle richieste asincrone per quelle risorse e una volta caricate le include nella pagina.
Quando una di queste risorse è un file .js (javascript), il browser inizia a leggerne il contenuto ed interpretare il codice. Spesso per non avere problemi di eseguire del codice che fa riferimento ad elementi non ancora caricati (a causa dell'asincronicità con cui il browser carica gli elementi), si fa partire l'esecuzione all'evento *onLoad().*

Il JavaScript deve comunque essere eseguito con delle politiche di gestione molto rigide. Infatti essendo un codice che viene eseguito sulla macchina dell'utente potrebbe (se le intenzioni del programmatore non sono delle migliori) provocare danni all'utente.
E' per questo motivo che vengono effettuate molte restrizioni.
In primis gli viene negata la possibilità di accedere al *filesystem* in lettura o in scrittura, così da evitare di scrivere virus sull'hard-disk oppure di leggere informazioni personali dell'utente.
Poi viene negata la possibilità di interagire con qualunque altro script che è eseguito nel browser su un'altra *scheda*, in modo da non poter leggere informazioni su connessioni dell'utente verso altri siti.

Con JavaScript è possibile creare *connessioni* solamente verso il dominio da cui è stato caricato lo script, non è invece possibile inviare richieste HTTP su domini diversi (in maniera tale da non poter fare attacchi distribuiti sulla rete all'insaputa dell'utente).
Non è possibile accedere ai *cookie* che appartengono a domini diversi, ma solo i cookie memorizzati dal proprio dominio.
Infine gli vengono dati dei limiti alle *risorse* (nel senso di memoria) a cui può accedere in modo da evitare che uno script blocchi l'intero sistema (oppure anche solo gli script contenuti in altre schede dello stesso browser) prendendo tutta la memoria RAM.

Ma allora che può fare il JavaScript?
Come tutti i linguaggi di programmazione può istanziare variabili (o, essendo anche orientato ad oggetti, può creare oggetti), creare ed eseguire funzioni, utilizzare i soliti costrutti condizionali e ciclici, e reagire ad eventi.

Sostanzialmente sono due le cose che si fanno con il JavaScript:

- Modificare il documento accedendo al DOM;
- Effettuare richieste asincrone al Server.

1.3.2 DOM (Document Object Model)

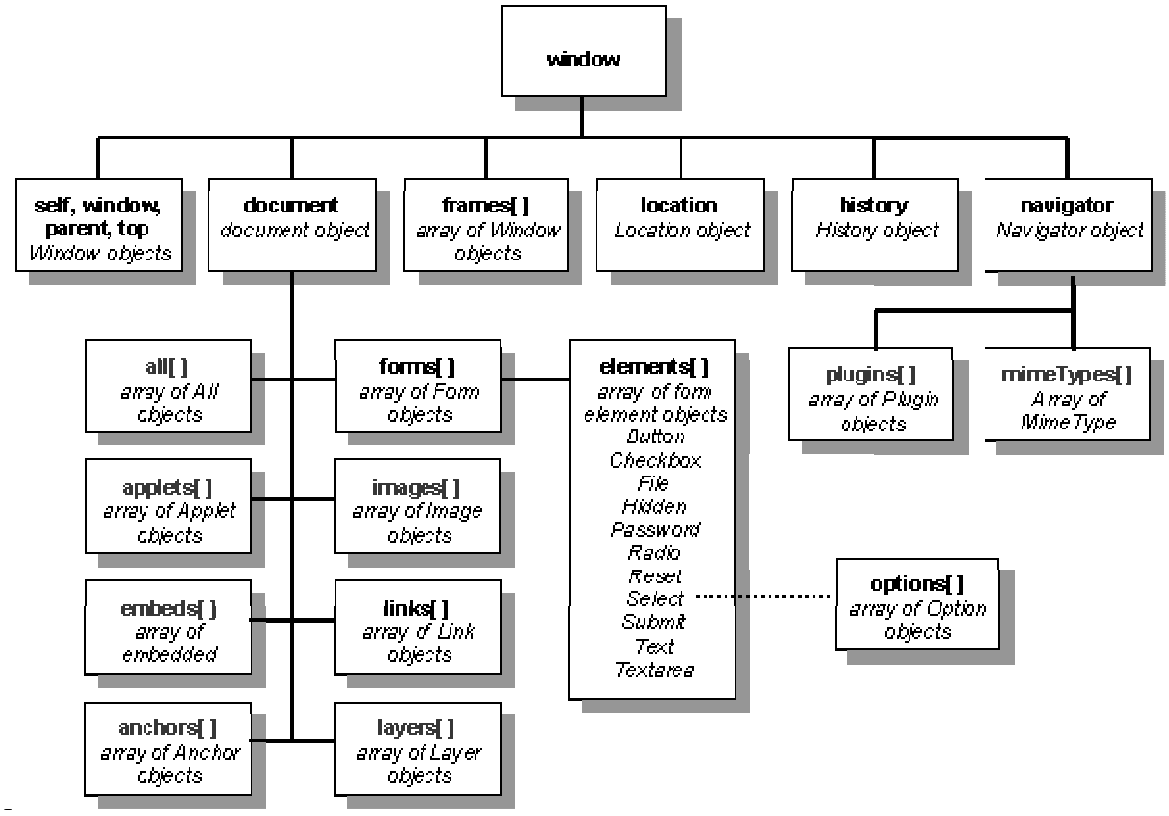
[14] Il DOM non è altro che un modello che descrive un documento HTML come un insieme di oggetti collegati tra di loro. Quindi c'è a disposizione un set di API per leggere il contenuto della pagina e per modificarlo. Il JavaScript non è l'unico modo per accedere a queste API, ma sicuramente il modo più comune per farlo.
Tutto il documento è visto come un oggetto, l'oggetto *document* e su di esso è possibile invocare dei metodi per recuperare elementi *getElementById()* o *getElementsByTagName()* che restituiscono elementi già esistenti nel documento oppure metodi per creare degli elementi ex novo *createElement()* e *createTextNode()*.
Su questi elementi possiamo applicare metodi per modificarne il contenuto, lo stile, la

posizione nel documento, per rimuovergli o appendergli altri elementi in coda. Alcune volte si fanno anche strappi alle regole utilizzando dei metodi non inseriti nel W3C come il famoso *innerHTML* ma che godono della proprietà di essere estremamente semplici e soprattutto crossbrowser.

In pratica tramite il DOM, con JavaScript possiamo modificare completamente la nostra pagina dinamicamente in corrispondenza di azioni eseguite dell'utente. Non abbiamo più pagine HTML con elementi fissi ma gli elementi possono essere spostati, e creati allo scattare di eventi.

Vengono definiti tanti tipi di eventi, legati agli oggetti (*onmouseover*, *onclick*, ...), legati alla tastiera (*onkeypress*, *onkeydown*, ...), legati al caricamento degli oggetti (*onload*, *onabort*, ...) a cui è possibile legare handler di funzioni.

Ecco un'immagine grafica di tutti i componenti del DOM disposti in maniera gerarchica tra loro.



browser rispettano la sintassi del JavaScript messa a punto dal W3C. Ma in passato (in piena guerra dei browser), non tutte le funzioni erano supportate allo stesso modo dai

browser. Quindi scrivere il codice era un'operazione non molto agevole. Spesso si dovevano trovare delle soluzioni alternative per far funzionare il codice almeno su *Netscape* e su *Explorer*. Per fare ciò, si doveva scrivere del codice che controllasse prima che tipo di browser si stava utilizzando, e poi eseguisse delle precise funzioni per il rispettivo browser. Ancora oggi se si vuole mantenere una certa compatibilità con i browser più remoti bisogna seguire queste regole.

Un esempio può essere la funzione *document.getElementById()* che inizialmente era supportata solo da Netscape. Per ottenere la stessa funzione su Explorer si doveva utilizzare il metodo document.all scrivendo un codice molto simile a questo:

```
□□□□□□□□ □□□□□□□□□□□□□□□□□□□□(□□)
  {□□ (□□□□□□□□·□□□)
     □□□□□□ □□□□□□□□·□□□[□□];
   □□□□
     □□□□□□ □□□□□□□□·□□□□□□□□□□□□□□□(□□);
   }
```

Come si può notare da questo frammento di codice, ciò che spesso si fa in JavaScript non è vedere qual è il browser in questione, ma semplicemente un test per capire se il browser conosce la funzione:

```
□□ (□□□□□□□□·□□□)
```

se la conosce la utilizza, altrimenti prova ad utilizzarne un'altra.

1.3.3 JavaScript per migliorare un sito

Possiamo distinguere tre modi diversi di usare il JavaScript: per creare effetti grafici che decorano le nostre pagine, per agevolare un utente nella sua navigazione, per programmare AJAX (descritto in seguito **[12]**).

Il linguaggio è nato molto presto nella storia del Web, anche se inizialmente le sue potenzialità erano molto ridotte rispetto ad oggi.

Il motivo per cui inizialmente nacque era quello di decorare le pagine web.

Le pagine internet poiché scritte in HTML non contenevano altro che dati, quindi le uniche animazioni erano create con l'inclusione di immagini GIF.

Inserendo invece un linguaggio di programmazione lato-client, si sono potute creare animazioni più complesse.

E' possibile ad esempio cambiare un'immagine della pagina quando il mouse ci passa sopra creando quello che viene chiamato effetto “RollOver”, oppure creare menù a tendina che si aprono con click del mouse dando una certa dinamicità alla pagina.

1.3.4 JavaScript per facilitare i form

Spesso un utente si trova a disagio quando deve compilare dei form per inviare dati al Server. Non sa mai se ciò che ha scritto è corretto oppure se deve scriverlo in maniera diversa. D'altronde non è sempre possibile aiutare la sua scelta utilizzando le *ComboBox*, già definite nell'HTML, ma spesso si utilizzano delle semplici caselle di testo in cui inserire i dati. Infine l'utente cliccando su un bottone invia i dati al server. Il server controlla tutti i campi inviatogli e, nel caso in cui ci sono stati errori di compilazione del form, avvisa il client dell'errore chiedendogli di riscrivere tutti i dati.

Per evitare una notevole perdita di tempo al cliente e un affaticamento del server nel controllare tutte le richieste arrivategli, si è pensato di utilizzare il JavaScript per validare i dati digitati dall'utente.

Questo è stato il primo meccanismo per spostare un po' di intelligenza dal server al client

(comunque non eliminando la validazione lato server poiché il JavaScript essendo un linguaggio lato client può essere facilmente riscritto e modificato a proprio piacimento).

Recuperare il valore di un campo in un form è molto semplice:

```
□□□□□□□□·□□□□□[‘□□□□□□□□’]·□□□□□□□□·□□□□
□
```

In questo modo possiamo ricavare il valore inserito all'interno di un campo *input* di un *form* e valutarne la correttezza. Per fare ciò ci sono offerti dei mezzi molto potenti da applicare alle stringhe come le espressioni regolari.

Il controllo dei campi può essere effettuato sia all'evento *onChange* o *onBlur* sull'elemento desiderato, oppure sull'evento *onClick* del *Button sumbit*.
L'errore riscontrato può essere sia visualizzato in un *MessageBox* con il comando *alert* oppure può essere inserito nel documento con DOM.

Capitolo 2
Il Web 2.0

2.1 Introduzione al Web 2.0

[16] Il termine **Web 2.0** delinea una metodologia di sviluppo e di design di seconda generazione, orientata verso la *creatività*, la *collaborazione* e la *condivisione di idee* tra utenti. Infatti è proprio con l'avvento di questa nuova tecnologia che si sono diffusi i *blog*, i *forum*, i *wiki* e tutti i siti per la *condivisione* di immagini e file.

Web 2.0 perché si vuole alludere ad un miglioramento del World Wide Web, infatti il nome richiama il “numero di versione” di un software usato quando un programma viene aggiornato.

Il miglioramento è inteso dal punto di vista della percezione dell'utente che assume un ruolo non più solo passivo, ma anche attivo. Infatti l'utente non si limita solamente a recuperare risorse dalla rete ma le crea e le distribuisce.

[17] E' difficile catalogare un sito **Web 2.0**. Spesso questo nuovo termine viene attribuito impropriamente a numerosi siti internet che poi non introducono nulla di realmente innovativo al Web 1.0.

Infatti il Web 2.0 non ha confini rigidi ma un'anima gravitazionale. Può essere visualizzato come un insieme di principi e di procedure che collegano un autentico sistema solare di siti che dimostrano in toto o in parte di questi principi, a una distanza variabile dal centro stesso.

L'immagine che segue mostra un insieme di applicazioni Web che sono state riconosciute come Esempi di Web 2.0 nella prima conferenza sul tema Web 2.0, nell’ottobre 2004.

Tutte queste applicazioni hanno aspetti innovativi come:

- vedere un'applicazione come una vera e propria piattaforma (Google);
- vedere il web non più formato solo da client e da server, ma in cui ogni utente può diventare un nuovo server (BitTorrent);
- l'*hyperlinking* ossia la capacità che hanno gli utenti di aggiungere nuovi concetti o informazioni ai siti ad esempio condividendo i propri gusti, (Yahoo, eBay, Amazing);
- vedere l'utente come costruttore dell'applicazione, è lui a condividere idee, informazioni, immagini (Wikipedia, Flikr e Blogs);
- capacità di creare RSS *Really Simple Syndication*, che permettono ad utenti di "abbonarsi" ad applicazioni, ricevere quindi le ultime informazioni e leggerle anche con strumenti diversi dai comuni browser;
- il tentativo di accaparrarsi il pieno controllo sui dati, che per alcune applicazioni come le mappe on-line è critico: (GoogleMap, YahooMap, MsnMap);

Tutti queste caratteristiche migliorano una delle qualità peculiari che un software non

dovrebbe mancare: la **scalabilità**.

Applicazioni come Google sono considerate piatteformi in quanto forniscono un set di API utilizzabili da chiunque nella propria applicazione. Esistono interfacce come **Google AJAX Search API** da includere in un'applicazione AJAX per effettuare ricerche del web, le **Google Maps API** per inserire mappe e altre.

La scalabilità di BitTorrent è ottenuta trasformando ogni client in un potenziale server, aumentando in questo modo notevolmente la potenza di calcolo e la velocità di trasmissione di informazioni.
Wiki, Flikr, e Blog sono scalabili grazie al loro contenuto che evolve a rapidissima velocità. Si sta aumentando il numero di sviluppatori trasformando ogni client in collaboratore a pieni diritti.
Applicazioni come queste sono scalabili soprattutto perché: quanti più utenti utilizzano l'applicazione tanto più l'applicazione funziona meglio.

Gli RSS, invece, contribuiscono alla scalabilità grazie alla capacità di essere *computer-readeble* e di poter funzionare da *feed*. Infatti con l'avvento del Web 2.0 si è passati da tecniche di *stickiness* a tecniche di *syndacation*. Il Web 1.0 doveva essere sticky (“appiccicoso”), ossia doveva invogliare l'utente a rimanere quanto più a lungo possibile sulle pagine web. Oggi, grazie agli RSS, il Web 2.0 utilizza tecniche di *syndacation* che modifica il contatto tra utente e fruitore. L'utente si stacca dal sito ma può richiedere di abbonarsi ad esso e ricevere delle informazioni, in genere in linguaggi formattati come XML, e visualizzarli con appositi programmi anziché con browser.

2.2 AJAX

2.2.1 Ajax: Oggetto XMLHttpRequest

Ajax è l'acronimo per (*Asynchronous JavaScript And XML*). Asincrono perché il client può fare richieste per pagine HTML o altre risorse non più delegando il browser, ma a suo piacimento.
XML perché le risposte in genere sono costituite da file .xml. Questo è un altro linguaggio di Markup simile all'HTML con la differenza che può contenere qualunque tipo di dati e non sono pagine che devono essere lette da un browser.

Ajax fornisce un approccio completamente nuovo alle applicazioni web. Grazie a questo nuova tecnica di programmazione è possibile distribuire un po' di intelligenza lato client. Quest'ultimo non si limiterà a chiedere semplici richieste di GET per pagine HTML ma invierà richieste asincrone per richiedere solo pezzi di informazione che servono a completare la vista. Questo comporta tre grossi vantaggi.

Il primo vantaggio lo si può riscontrare nella banda di utilizzo. Le richieste asincrone richiederanno solo i dati veri e propri da inserire nelle pagine eludendo tutte quelle meta-informazioni contenute nelle pagine HTML che servono a rendere una visualizzazione tabellare.

Un altro vantaggio è quello di riuscire a semplificare il server. E' possibile far fare al client alcune operazioni, come quelle di ricerca, ordinamento, costruzione di una vista... snellendole dal server. Questo comporta dei server più leggeri quindi più snelli e veloci, che hanno bisogno di minori requisiti e che quindi possono essere installati su dispositivi mobili senza problemi di risorse.

Un terzo vantaggio, anche se non tecnico ma ai fini dell'utente, è quello di riuscire a aggiungere informazioni in maniera graduale ad una vista corrispondentemente a ciò che

richiede un utente. Il fatto di non dover ricaricare una pagina per recuperare delle informazioni aggiuntive, rende la navigazione dell'utente più agevole e veloce.

Per inviare richieste asincrone al Server il JavaScript ci mette a disposizione un oggetto capace di creare nuove connessioni TCP e inviare request Http: l'oggetto ***XMLHttpRequest***.
Ancora una volta abbiamo un enorme problema, la compatibilità con i diversi browser. Tutti quelli che derivano dalla famiglia Netscape mettono a disposizione nativamente questo oggetto. La famiglia Explorer mette a disposizione un oggetto simile (che permette di fare più o meno le stesse operazioni), ma che è identificato con un altro nome: ***ActiveXObject***.

Per far funzionare il codice sui diversi browser, si controlla l'esistenza di uno dei due oggetti sopra citati e se ne crea un'istanza; questa sarà la nostra funzione *assegnaXMLHttpRequest()*.
Una volta scritta la funzione multibrowser possiamo utilizzarla senza problemi per recuperare il nostro oggetto XMLHttpRequest.

Fortunatamente ci sono una lista di metodi su quest'oggetto che sono supportati da tutti i browser che supportano l'AJAX; i più importanti sono:

- *open(method, uri [,async]):*

  Dice al browser che l'utente vuole fare una HTTP Request con metodo “method” (ad es. 'GET') per l'oggetto indicato nella “uri” (ad es 'risorsa.xml'), e “async” è una variabile booleana che indica se la richiesta deve essere effettuata in modo asincrono o no.

  C'è da notare che il parametro “uri” deve essere assolutamente una risorsa sul browser e non può cominciare per 'http://...' altrimenti per motivi di sicurezza la richiesta verrebbe bloccata dal browser.

- *setRequestHeader(nome, valore):*

  Questa funzione non fa altro che aggiungere un Header alla richiesta HTTP dove la coppia di parametri nome valore sono due stringhe (per esempio nome='connection' valore='close').

- *send(data):*

  Il metodo send è quello che si occupa di inviare la richiesta vera e propria al Server. La richiesta può sia avere un payload inviato nel parametro data (una stringa) oppure non avere nessun corpo. Quando viene eseguito questo metodo il flusso di esecuzione del programma si divide in due. Un thread continua ad eseguire il codice che segue alla riga che contiene l'istruzione *send()*, un altro thread invece invia una richiesta HTTP al server e solo quando ha ricevuto la risposta (oppure allo scadere di un timeout) richiama il metodo *onreadystatechange()* che il programmatore deve aver avuto cura di ridefinire.

Con soli questi tre metodi possiamo inviare una richiesta asincrona al server per ottenere una risorsa. Ci sono comunque altri metodi che ci consentono di fare cose più complesse.
L'ultima cosa che ci resta da fare è modificare il contenuto della nostra pagina in conseguenza alla risposta che otteniamo dal server.
Per fare ciò (come già accennato) dobbiamo ridefinire il metodo *onreadystatechange();* quindi scriviamo:

```
□□□□·□□□□□□□□□□□□□□□□□□ = □□□□□□□□()
   {// □□□□□□ □□ □□□‘□□□ □□□□□□□□ □□□□ …
    }
```

dove *ajax* è il nostro oggetto che abbiamo ottenuto invocando la funzione *assegnaXMLHttpRequest().*

Questa funzione sarà invocata dal browser quando la risorsa è stata ritornata oppure quando c'è stato un errore. Per vedere cosa è successo esattamente possiamo sfruttare il contenuto di due variabili:

- *readyState:*

  Rappresenta lo stato della richiesta e contiene dei valori interi da 0 a 4, anche se l'unico supportato da tutti i browser è il valore 4 che indica lo stato “loaded”. La nostra funzione *onreadystatechange()* controllerà quindi lo stato della richiesta con un semplice if e se uguale a 4 andrà avanti.

- *status:*

  E' una variabile che ci da informazioni sulla risposta ricevuta. Il valore '*200*' secondo il protocollo HTTP è l'unico che significa '*OK – risorsa ricevuta correttamente*', quindi non dobbiamo fare altro che un controllo se questa variabile vale 200, se sì eseguire le nostra azione,

altrimenti stampare un messaggio di errore.

Per utilizzare il contenuto della risposta possiamo utilizzare queste due variabili dell'oggetto ajax:

- *responseXML:*
  E' un oggetto JavaScript XML che contiene la risorsa XML richiesta. Possiamo utilizzare dei metodi JavaScript per fare il parsing del documento XML e sfruttare le informazioni contenute in esso.

- *responseText*:
  Nel caso in cui la risorsa richiesta non è un oggetto XML ma qualcos'altro, la variabile *responseXML* sarà uguale a null, ciò non vuol dire che la risposta non contine nulla. Il contenuto della risposta adesso non contiene informazioni scritte in un linguaggio standard ma delle informazioni, comunque testuali, che possiamo recuperare dalla variabile responseText.

Abbiamo tutto ciò che ci occorre per effettuare richieste asincrone al server e (con DOM) per modificare il contenuto della pagina con il risultato ottenuto dalla richiesta.

2.2.2 Gestire un sito con AJAX

[18] Scrivere un'applicazione in AJAX significa fare un grande passo in avanti. Bisogna iniziare a vedere il server come un semplice *fornitore di servizi*, e non anche come *gestore*. Quindi spostare gran parte della logica dell'applicazione sul Client che dovrà conoscere tutti i servizi elementari offerti dal server e agglomerarli insieme per fornire funzionalità complete.
Questo è un modo completamente nuovo di vedere le applicazioni web che ci porta alla definizione di **WEB 2.0**.

Usare richieste asincrone per visualizzare i nuovi contenuti comporta un enorme vantaggio. La cosa che si nota subito quando ci si trova in un sito in AJAX è appunto il riuscire a visualizzare informazioni nuove, nuovi contesti, senza dover rileggere l'intera pagina. Quando si ricarica una pagina (a parte il fastidio di vedere l'icona trasformarsi e la barra di stato caricare) si hanno dei veri svantaggi.

Per esempio il browser fa comunque una richiesta di tutti gli oggetti che sono presenti nella pagina portando un sovraccarico inutile nell'applicazione.
O ancora abbiamo un'impossibilità di mantenere uno “stato” lato client. Infatti per ogni caricamento il browser “uccide” lo script che stavamo eseguendo precedentemente deallocando tutto lo spazio che esso utilizzava compreso quello per memorizzare le variabili. Anche se esistono tecniche per creare uno stato lato client appendendo delle variabili sulla barra degli indirizzi (proprio come se stessimo inviando delle variabili al server con il metodo GET) e poi recuperandole all'esecuzione del nuovo script, il modo migliore per creare uno stato lato-client è avere uno script che rimanga “vivo” dalla prima richiesta della pagina all'ultima.

Il nostro *sito internet* adesso non è più un banale documento in cui visualizziamo una serie di dati ma è un vera *applicazione web 2.0* che carichiamo digitando l'URL iniziale. A quell'indirizzo vedremo anche quella che molto probabilmente sarà la nostra prima e unica “pagina”, che si occuperà di inizializzare tutti gli script che ci occorrono per svolgere le operazioni utente.

Esempi di “ottime” applicazioni web 2.0 sono: *Google Maps* oppure *Gmail*, in cui utenti possono girare il mondo intero da una pagina internet oppure gestire la propria casella di posta elettronica senza vedersi aggiornare pagine per ogni azione.

I problemi più gravi da affrontare quando si scrivono applicazioni Ajax (ma non solo quando il codice lato client è il JavaScript) sono due:
Cosa succede se un utente non può utilizzare l'oggetto XMLHttpRequest?
Cosa succede se un utente preme il pulsante del browser “indietro”?
A questi problemi non c'è una risposta precisa. Ognuno quando progetta la propria applicazione dovrebbe pensare bene al perché sta scegliendo di usare l'Ajax.
Per quanto riguarda il primo problema si potrebbe cercare (se possibile) di dare la possibilità ad un utente di effettuare le proprie richieste anche cliccando su dei semplici link.

Per il secondo problema nasce l'esigenza di ridefinire la funzionalità del browser e di memorizzare lato-client delle informazioni che possano riportare l'applicazione in uno stato precedente.
Un ultimo punto in cui bisogna porgere attenzione quando si programma Ajax è nel come si usano le risorse e nello stare attenti a deallocarle (o almeno a non lasciarle in giro negli script). Infatti adesso gli script non sono più interpretati una volta e poi fermati. Molti script rimarranno in esecuzione anche per molte ore, e come nella consueta programmazione bisogna porre attenzione alla deallocazione delle risorse poiché, finché rimane vivo il programma, rimane vivo anche lo script.

2.2.3 I Framework: Dojo

[19] Oggi molte applicazioni web tentano di essere costruite completamente o parzialmente in questa nuova tecnologia. Ma, come abbiamo visto, scrivere Ajax non solo è complicato perché si tratta di un'applicazione distribuita su internet (una rete insicura per eccellenza), ma a ciò si aggiunge il problema delle incompatibilità dei browser. Infatti, solamente per recuperare un elemento dobbiamo controllare che browser stiamo utilizzando.
Fare ciò su un'intera applicazione risulta impensabile. E' principalmente per risolvere questi problemi che sono nati i *Framework Ajax*.

Un Framework Ajax non è altro di una collezione di funzioni e di oggetti JavaScript offerti da case produttrici di software che garantiscono la compatibilità sui diversi browser. Quindi, una volta scelto il nostro Framework, non ci resta che utilizzare le funzioni ridefinite dal framework anziché utilizzare quelle offerte dai browser.
Inoltre questi *framework* sono spesso dei veri e propri *toolkit* che oltre a ridefinire funzioni, cercano di costruire un gruppo di oggetti di facile uso che possono essere inclusi nell'applicazione sostituendo i tag HTML.

Ad esempio un tag *div* che in HTML non è altro che un contenitore di elementi può essere

sostituito da *widget* come *AccordionContainer, ContentPane* e tanti altri che a differenza dei vecchi *div* possono essere, spostati, chiusi, ridimensionati con dei movimenti del mouse su di essi.
Il Framework utilizzato da google per le sue applicazioni è il ***GoogleWebToolkit*** (GWT) creato appunto da Google e messo a disposizione gratuitamente. Il framework consiste di un insieme di classi Java da poter importare nei propri progetti e scrivere le proprie funzioni. Questa è la particolarità di questo framework; si programma in Java e poi un componente si preoccuperà di tradurre tutto il codice in JavaScript e di portarlo all'interno dell'applicazione. Gli utenti vedranno solo il prodotto finale JavaScript che eseguiranno, ignari di tutto, nel proprio browser.

Nel mio percorso ho preferito centrare l'attenzione su un altro toolkit molto utilizzato: ***Dojo***.

Anche questo toolkit è distribuito gratuitamente ma scritto completamente in JavaScript.
Nel framework vengono ridefinite tutte le funzioni principali del browser come *getElementById(),* vengono ridefiniti tutti gli oggetti frequentemente utilizzati come l'oggetto *Math* o l'oggetto *XMLHttpRequest*, e vengono messe a disposizione dell'utente accedendogli con la parola chiave *dojo*. Viene anche data la possibilità all'utente di ridefinire delle proprie librerie o dei propri oggetti in maniera molto semplice e di importare il tutto con delle istruzioni particolare.

D'altro lato il framework fornisce un gruppo sostanziale di elementi grafici che sono utilizzati in genere nelle applicazioni e che possono essere utilizzati nelle pagine HTML al posto di alcuni tag, oppure possono essere creati dinamicamente.
Tutti questi elementi vengono inseriti in una libreria *dijit* e possono essere recuperati proprio con questa parola chiave.

Vengono ridefiniti oggetti contenitori (*AccordionPane, ContentPane, SplitContainer,*

TabContenier...), oggetti di input (*CheckBox, ComboBox, DateTextBox, NumberTextBox*...), finestre (*Dialog, TooltipDialog*...), bottoni (*ComboButton, DropDownButton*...), barre (*Toolbar, Menu*...), altri oggetti (*Grid, Editor, ColorPalette*...).

Per tutti questi elementi è possibile associare uno degli stili predefiniti di dojo, (*tundra.css, noir.css*) oppure è molto semplice sulla base di questi fogli di stile costruirsene alcuni nuovi per ottenere degli oggetti personalizzati al 100%.

Fare richieste asincrone con il nuovo framework è molto semplice, basta richiamare le funzioni *dojo.xhrGet()* o *dojo.xhrPost()*, e per consultare la risposta ci sono oggetti che consentono di eseguire facilmente un parsing XML.

Ecco un esempio di richiesta GET che richiede una risorsa testuale:

```
□□□□.□□□□□□(
  {□□□: "/□□□□□□.□□□",
   □□□□□□□□: "□□□□",
   □□□□□□□: 5000,
   □□□□: □□□□□□□□(□□□□□□□□, □□□□□□)

  {□□□□□.□□□□("□□□□□□□□□□□□").□□□□□□□□□□(□□□
  □□□□□);
      □□□□□□ □□□□□□□□;
     },
   □□□□□: □□□□□□□□(□□□□□□□□, □□□□□□)
     {□□□□□("□□□□□□ □□□□□ □□□□□□□□□!");
      □□□□□□ □□□□□□□□;
     }
  });
```

La funzione xhrGet accetta come parametro di ingresso un vettore associativo. In questo vettore dovranno essere definiti alcuni campi: *url* che indica la risorsa richiesta; *handleAs*

che indica il tipo di risorsa ritornata (in genere testuale o XML); *timeout* che indica quanto tempo (in millisecondi) bisogna attendere la risposta, prima di considerare la sua mancanza come un errore; *load* che rappresenta la funzione da eseguire quando viene ricevuta la risorsa, *error* che rappresenta la funzione da eseguire in caso di errori.

Se volessimo appendere dei parametri alla GET in stile querystring, dovremo farlo aggiungendoli alla variabile url.
Una *xhrPost* è molto simile alla GET con l'aggiunta di un campo *form* nel vettore associativo che indica l'id del form che si sta inviando. E' ovvio che il *button* del form non sarà più legato all'azione *submit* ma all'evento *onclick* eseguirà la funzione *xhrPost*.

2.3 I Web Services

1.3.1 Introduzione ai Web Services

Il web 2.0 ha modificato il modo di progettare applicazioni web: il Server offre dei semplici servizi e il Client li mette insieme per formare delle funzionalità complete.
Allora, perché adattare linguaggi come PHP (nati per costruire pagine HTML dinamiche) per fornire servizi e non utilizzare qualcos'altro il cui compito è proprio fornire servizi?
Questo ha portato alla nascita dei *Web Services*.

[20] Un Web Service è definito dal W3C come "*un sistema software disegnato per supportare l'interoperabilità tra più macchine che interagiscono tra di loro su una rete*".
E' molto simile ad una procedura remota, la URL rappresenta il nome della funzione da invocare e i suoi parametri vengono passati tramite query string appendendo all'url le nostre solite variabili come nel metodo GET.

Adesso possiamo pensare la nostra applicazione come un'insieme di funzioni da invocare tramite browser con richieste asincrone, oppure che possono essere invocate anche da client esterni ai comuni browser scritti in un qualunque linguaggio di programmazione su qualsiasi tipo di piattaforma.

La ragione principale per la creazione e l'utilizzo di Web Service è il "disaccoppiamento" che l'interfaccia standard esposta dal Web Service rende possibile fra il sistema utente ed il Web Service stesso: modifiche ad una o all'altra delle applicazioni possono essere attuate in maniera "trasparente" all'interfaccia tra i due sistemi; tale flessibilità consente la creazione di sistemi software complessi costituiti da componenti svincolati l'uno dall'altro e consente una forte riusabilità di codice ed applicazioni già sviluppate.

Per scambiarsi informazioni su come debbano essere invocati questi metodi, si utilizzano nuovi protocolli che si pongono sopra HTTP: **SOAP** e **WSDL**.

2.3.2 SOAP e WSDL

[21] SOAP (*Simple Object Access Protocol*) è un protocollo che controlla lo scambio di messaggi tra componenti software. La parola Object ci fa capire che ipotizza un componente software come un oggetto, e a partire da esso cerca di costruire un messaggio scritto in un linguaggio esteso dell'XML che descrive l'oggetto, le sue variabili membro e i suoi metodi.

L'**OASIS** (Organizzazione per lo sviluppo di standard sull'informazione strutturata) definisce la SOA così:
"*Un paradigma per l'organizzazione e l'utilizzazione delle risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti. Fornisce un mezzo uniforme per offrire, scoprire, interagire ed usare le capacità di produrre gli effetti voluti consistentemente con presupposti e aspettative misurabili*".

SOAP può essere utilizzato su qualsiasi protocollo a livello applicazione, anche se è comunemente usato su HTTP, e può essere visto come una "busta" in cui inserire degli appositi "messaggi". Tutto il messaggio è chiamato *Envelope*, ed è formato da due parti: l'Header (che contiene meta-informazioni come quelle riguardanti il *routing*, la *sicurezza* e le *transazioni*) e il Body (che trasporta il contenuto informativo o "carico pagante").
Non è legato ad alcuni linguaggi di programmazione in particolare fornendo l'interoperabilità tra diversi software (ad es. Java e Python) e tra diverse piattaforme hardware (come Windows e Linux) grazie soprattutto a standard "aperti" (*open source*).

WSDL (*Web Services Description Language*) è un linguaggio che serve a descrivere un Web Service o meglio la sua interfaccia pubblica. Descrive il *cosa*, il *come* e il *dove* di un Web Services; ossia cosa poter fare con quel servizio, come poterlo invocare (descrizione delle variabili da passare) e dover poterlo trovare.
Anch'esso è scritto in sintassi XML e in genere è usato in coppia con SOAP.

2.3.3 Axis2 e Pojo

[22] Axis2 è un progetto open source di *Apache* per creare dei Web Services. E' esattamente un container per Apache Tomcat, in cui è possibile fare il *deploy* dei servizi (creati in Java).

Per scrivere un servizio occorrono quattro passi:

a) Implementare la classe che svolge il servizio;
b) Scrivere il file *services.xml* che descrive il servizio;
c) Creare un archivio *.aar* (Axis Archive);
d) Fare il deploy del servizio sul Server.

Esistono diversi modi per implementare il servizio: ***Pojo***, *Axiom*, *Adb*, *XMLBeans*, e *JiBX*.
Nel mio studio ho posto l'attenzione sul più semplice e utilizzato: Pojo.

Pojo è l'acronimo per *Plain Old Java Object*. Utilizzare questa tecnologia significa scegliere un modo veloce e facile da implementare, semplice da mantenere.
Per implementare il server, si incomincia scrivendo la classe come una comune classe Java, i suoi metodi dichiarati pubblici saranno quelli invocabili.
Le funzioni pubbliche possono essere di due diversi tipi:
IN-ONLY o **IN-OUT**.
Questo cambia su come dobbiamo poi scrivere il file *services.xml* che dovrà appunto descrivere i servizi offerti dalla classe.
Un metodo che non ritorna nessun valore (*void*) è un metodo di IN-ONLY e la sua descrizione le file xml risulterà così:

```
<□□□□□□□□□□□□□□□□
□□□="□□□□://□□□·□3·□□□/3004/08/□□□□/
      □□-□□□□"
□□□□□="□□□·□□□□□□·□□□□2·□□□·□□□□□□□□□·
      □□□□□□□□□□□□□□□□□□□□□□□□□" />
```

Un metodo può anche ritornare un valore di qualunque tipo. Questo però porta a delle

limitazioni. Se ritorna un valore di tipo semplice esso può essere utilizzato anche da un qualunque browser. Se invece ritorna un oggetto non è più utilizzabile lato client semplicemente da un browser ma può essere utilizzato solo da client Java (o scritti in un qualunque altro linguaggio di programmazione). Per questi metodi di IN-OUT si scrive nel file .xml:

```
<□□□□□□□□□□□□□□□□
□□□="□□□□://□□□·□3·□□□/2004/08/□□□□/
      □□-□□□"
□□□□□="□□□·□□□□□□·□□□□2·□□□·□□□□□□□□□·
      □□□□□□□□□□□□□□□□□□□" />
```

Scritta la classe e il file services.xml non ci resta che esportare tutto come un archivio .jar, rinominarlo .aar e incollarlo nella cartella WEB-INF/Services all'interno di axis2.

Invocare un servizio è molto semplice. Infatti adesso, puntando il nostro browser all'indirizzo del nostro server alla url *axis2,* abbiamo una pagina a interfaccia grafica che ci elenca tutti i servizi mostrandoci a cosa servono e come invocarli (quale url identifica quel servizio).

Quindi per invocare un servizio non ci resta che digitare l'url che lo identifica e per passare i parametri (se sono tutti tipi semplici) possiamo farlo con le *query string* aggiungendo all'url una stringa simile a questa: *?var1=val1&var2=val2...* .

Infine possiamo utilizzare i servizi con JavaScript nella nostra applicazione web come servizi di supporto oppure possiamo creare un client interamente in Ajax.

Se non ci si vuole limitare a scrivere Web Services che possano utilizzare solo valori semplici, si può decidere di utilizzare anche il Java lato-client e poi includerlo in una pagina (applet) oppure costruire un'applicazione stand-alone.

In entrambi i casi c'è bisogno di importare alcune librerie, quelle che si trovano nella cartella *lib* di Axis (tutte le librerie sono fornite in formato .jar).

Ciò che dobbiamo fare è importare quattro classi che si trovano nei jar:

```
□□□·□□□□□□·□□□□2·□□□□□□□□□
□□□·□□□□□□·□□□□2·□□□□□□□□□□·□□□□□□□□□□
```

□□□□□□□

□□□·□□□□□□·□□□□2·□□□□□□·□□□□□□

□□□·□□□□□□·□□□·□□□□□□·□□□□□□□□□□□□□□□□

e utilizzarle.

Si inizializza un *RPCServiceClient*, ed un *EndpointReference* (passandogli l'indirizzo su cui si trova l'oggetto da invocare), si settano delle opzioni grazie all'oggetto *Options* e poi si può istanziare un oggetto della stessa classe creata sul server e invocarne i metodi.

Per invocare i metodi remoti si utilizzano due funzioni dell'oggetto *RPCServiceClient*:

- *invokeRobust()*: se la funzione è di tipo IN-ONLY, questa funzione si limiterà ad inviare i dati necessari al Server per invocare il servizio. Poiché non si aspetta nessuna risposta può, appena finito, passare all'istruzione successiva.
- *invokeBlocking()*: se la funzione è di tipo IN-OUT. Adesso una volta inviati tutti i dati al server non si potrà eseguire le successive istruzioni finché il server non abbia finito con l'esecuzione della funzione remota invocata e non abbia inviato la risposta al client. Il tempo in cui il client rimane bloccato è indeterminato.

Capitolo 3
REST

3.1 Introduzione a REST

[23] Le applicazioni basate sui web service appartengono al paradigma architetturale SOA (*Services Oriented Architecture*). L'interazione tra client e server è effettuata, come abbiamo visto nel capitolo precedente, tramite delle invocazioni a procedure remote.

[24] Il World Wide Web, però, non è altro che un insieme di risorse che gli utenti consultano e utilizzano. Allora, perché utilizzare un modello SOA per gestire Risorse quando il mondo del Web è già fatto di Risorse?
Roy Fielding ha risposto a questa domanda disegnando un nuovo paradigma architetturale che pone al centro dell'attenzione non più i “Servizi” ma le “Risorse”: REST (*REpresentional State Trasfer*).
Questa paradigma ci spinge a pensare la nostra applicazione non più come un'insieme di metodi da invocare e stati da mantenere, ma come un'insieme di risorse da gestire.
Pensare REST significa abbandonare il vecchio modello SOA in favore di un nuovo modello: ROA (*Resource Oriented Architecture*).

Il Web si basa su HTTP, un protocollo che offre già dei metodi di base che permettono di lavorare sulle risorse. Usare il modello SOA, significa definire dei servizi per riferirsi alle reali risorse, ed accedere ad esse come se fossero risorse fittizie con i metodi definiti da HTTP. Abbiamo creato una risorsa “Servizio” gestore della risorsa “dati”.

Quindi richiedere il contenuto di una risorsa significa identificare un servizio (ad es. *Restituisci)* ed identificare la risorsa desiderata (ad es. *Dati)*.

Usare il modello ROA significa utilizzare il protocollo HTTP per come è stato ideato, ossia come mezzo per accedere alle risorse. HTTP ha già tutto ciò che occorre per

identificare una risorsa e indicare una modalità di utilizzo, quindi per richiedere una risorsa basta riferirsi ad essa con l'apposito metodo HTTP GET.

3.2 Il concetto di risorsa

Una risorsa è una qualunque cosa che risulti abbastanza importante nel nostro contesto tanto da poterci riferire ad essa come un'entità a sé stante. E' un qualcosa che può essere memorizzato in un computer come un documento, un'immagine, una riga di un database o comunque una stringa di bit.

[26] Una risorsa, per essere considerata tale, deve avere un modo per riferirci ad essa. Noi ci riferiamo a risorse mediante URI (Uniform Resource Identifier). *Una risorsa deve avere almeno un URI* per essere definita risorsa altrimenti non potremo identificarla e quindi non potrebbe essere catalogata come risorsa.

Gli URI devono essere descrittivi. Dobbiamo capire bene ed in maniera intuitiva leggendo un URI cosa effettivamente stiamo richiedendo. Potrebbe anche capitare che due URI puntino alla stessa risorsa ma ciò non significa che questi URI sono la stessa risorsa. Ad esempio

possiamo avere delle risorse “versione” che indichino tutte le versioni esistenti di un software. Inoltre possiamo avere una risorsa “ultima_versione” che punti all'ultima versione rilasciata del software. Se abbiamo come oggetti versione dei valori 1.0, 2.0 e 3.0 è evidente che l'oggetto ultima_versione risulti pari a 3.0 ma ciò non significa che sarà sempre così perché nel tempo l'oggetto ultima_versione cambierà assumendo diversi valori.

Questo piccolo esempio serve a far capire che ogni risorsa significa qualcosa e che ci possono essere *relazioni tra le risorse*.

3.3 L'architettura orientata alle risorse

[24] L'architettura ROA è l'insieme di quattro concetti:

- Cos'è una risorsa;
- Cos'è un URI;
- Cosa rappresenta una risorsa;
- Quali sono le relazioni tra le risorse;

(Quattro domande a cui già abbiamo dato una risposta)

e di quattro proprietà:

- Addressability;
- Statelessness;
- Connectedness;
- L'esistenza di un'interfaccia uniforme;

Addressability:

Un'applicazione è considerata *addressable* quando espone aspetti interessanti dei dati attraverso Risorse. Esse sono identificate tramite URL, possiamo richiedere informazioni sempre più specifiche modificando la parte finale del path di una URL. Ciò si può fare quando le risorse sono correlate tra loro (in genere in forma gerarchica), e specializzare un URL significa contemporaneamente scendere ad un livello più specifico nella nostra gerarchia di risorse. Anche le query string possono essere viste come un modo per specializzare le richieste, ma è un modo innaturale e quindi più complicato nel mondo del web.

Statelessness:

REST si muove sul protocollo HTTP che è un protocollo *stateless*. L'esigenza di

avere un protocollo *stateless* ci da un enorme guadagno in scalabilità, poiché il server non deve associare più richieste una all'altra per capirle ma può utilizzarle una alla volta e poi deallocare le risorse. Questo porta però ad uno svantaggio, ogni richiesta fatta deve contenere tutte le informazioni necessarie al server per comprenderle. E' compito del server mantenere delle informazioni sullo stato del client.

Connectedness:

Per *connectedness* intendiamo la qualità di avere collegamenti esterni. Molti utenti non hanno la possibilità di digitare gli URL per accedere alle risorse, quindi un'applicazione ben formata dovrebbe, a partire da un qualunque punto, poter arrivare ad accedere a qualunque risorsa sul web, quindi in questo senso il web è *connected*.

Uniform Interface:

Un'interfaccia uniforme è data proprio dal protocollo HTTP. Uniforme perché HTTP definisce dei metodi standard da utilizzare e un modo omogeneo per scambiarsi le informazioni. Il successivo capitolo tratterà in dettaglio quest'argomento.

3.4 HTTP e similitudini con ROA

REST nella sua implementazione cerca di essere il più semplice possibile. Il Web è nato sul protocollo HTTP, che ha già tutto ciò che occorre per fare web. Si tratta solo di ridefinire qualcosa e di utilizzarlo per ciò che è nato.

Iniziamo a vedere che cos'è questo protocollo che è alla base del Web.

[27] HTTP è l'acronimo di *HyperText Trasfer Protocol*, e come dice il suo nome si occupa del trasferimento di ipertesti da un server ad un client o più genericamente da un host ad un altro. La prima versione di questo protocollo, la 1.0, risale alla fine degli anni

'80 e sta alla base del World Wide Web. HTTP/1.0 fu implementata da Tim Berners-Lee e migliorato nel 1997 in una nuova versione: la HTTP/1.1 usata oggi.
Il protocollo HTTP (considerando lo stack protocollare) si trova subito sopra il protocollo **TCP**, ciò garantisce ad HTTP una connessione sicura tra client e server. E' un protocollo *"stateless"* (come vuole che sia REST). Ogni richiesta ha tutto, e solo, ciò che occorre per essere servita. Dopo che la richiesta è stata servita la connessione viene chiusa, e le risorse usate deallocate.
Il server è un programma in attesa di richieste di connessione, quindi in ascolto su una socket (sulla porta 80).

La comunicazione consiste di quattro operazioni fondamentali:

- **Connessione** con TCP/IP;
- **Richiesta** della risorsa desiderata;
- **Risposta** ossia ricezione della risorsa o di un messaggio di errore;
- **Sconnessione** da parte del server;

Le risorse vengono identificate tramite un URL che le definisce univocamente sul server.
Inoltre quando il server invia una risorsa al client deve comunicare ad esso anche il tipo MIME (*Multipurpose Interact Mail Extension*), così quest'ultimo può capire che tipo di dato è contenuto nella risposta e interpretarlo correttamente (ad es. *text/html* se la risposta è scritta in html).
Una volta instaurata la connessione TCP il client invia la richiesta HTTP al server la cui prima riga, *Request-Line*, ha questa sintassi:

□□□□□□ □□ □□□□□□□-□□□ □□ □□□□□□□-□□□□ □□ □□

SP, CR, LF sono dei caratteri speciali (Space, Carriage Return, Line Feed);

Method è uno dei metodi HTTP;

Request-Uri è la stringa identificativa della risorsa;

Version-HTTP indica la versione del protocollo utilizzata (HTTP/1.0 o HTTP/1.1).

I possibili metodi sono:

- **OPTION**: richiesta di informazioni inerenti alle opzioni di comunicazione disponibili sul canale.
- **GET**: richiede l'indirizzo URI definito; può essere condizionato.
- **HEAD**: è identico a GET ma il server non ritorna il corpo del messaggio.
- **POST**: il client invia delle coppie nome=valore che corrispondono all'input del programma indicato nella request dalla URI.
- **PUT**: alloca una nuova risorsa sul server o ne aggiorna una preesistente.
- **DELETE**: viene richiesto al server di cancellare la risorsa.
- **TRACE**: richiede dati dal canale per testare e diagnosticare informazioni.
- **CONNECT**: per connettersi con un proxy.

Ad una richiesta pervenuta il server risponde con una risposta HTTP la cui prima riga, *Status-Line*, è fatta così:

□□□□□□□□-□□□□ □□ □□□□□□-□□□□ □□ □□□□□□-□□□□□□ □□ □□

Lo *Status-Code* è diviso in 5 classi:

- **1xx**: Informazione – richiesta ricevuta e continuo processo;
- **2xx**: Successo – azione ricevuta, capita e accettata;
- **3xx**: Ridirezione – occorrono altre informazioni;
- **4xx**: Client-Error – errori nella richiesta;
- **5xx**: Server-Error – il server fallisce;

La *Reason-Phrase* è invece una stringa contenente un'informazione testuale cercando di spiegare il risultato dello Status-Code.

Ecco come si presentano le intere richieste:

Request-Line	**Response-Line**
[General-Header CR LF]	[General-Header CR LF]
[Request-Header CR LF]	[Response-Header CR LF]
[Entity-Header CR LF]	[Entity-Header CR LF]
CR LF	CR LF
[corpo del messaggio]	[corpo del messaggio]

I *General-Header* sono delle intestazioni applicabili sia alla request che alla response contenenti delle informazioni aggiuntive sul messaggio.
I *Response-Header* e i *Request-Header* invece sono delle intestazioni specifiche di uno dei due messaggi.
Gli *Entity-Header* sono delle intestazioni che non contengono informazioni sul messaggio HTTP ma sulla risorsa contenuta nel payload ossia nel corpo del messaggio.

Il corpo del messaggio è il contenuto informativo che deve essere trasferito. Può essere sia in un linguaggio testuale che del codice HTML, XML o qualsiasi altro formato (anche codifiche di codici binari come immagini o file eseguibili).

Ecco i *Request-Header* più importanti:

- *Accept*: indica quali tipi MIME sono accettati come risposta alla request.
- *Accept-Charset*, *Accept-Encoding*, *Accept Language*: indicano rispettivamente quali caratteri, codifiche e lingue sono accettate.
- *Host*: indica l'host che si sta contattando.
- *If-Modified-Since*: contiene una data. Indica al server di inviare il file se e solo se è stato modificato dopo quella data (stiamo facendo web caching).
- *User-Agent*: indica il tipo di browser che effettua la richiesta.

Ecco i *Response-Header* più importanti:

- *Accept-Range*: indica i ranges che sono stati accettati e inviati dal server.
- *Age*: intero positivo indicante in tempo trascorso dall'invio della response.
- *Server*: contiene parametri che specificano il software utilizzato dal server.

Ecco i *General-Header* più importanti:

- *Connection*: può essere *Close* o *Keep-Alive* che indica se la connessione deve essere chiusa dopo la fine del messaggio oppure no.
- *Date*: contiene la data di generazione del messaggio.
- *Pragma*: contiene delle direttive di comportamento (es. non cacheabilità del messaggio con *no-cache*).

Ecco gli *Entity-Header* più importanti:

- *Content-Length*: lunghezza in byte del corpo del messaggio.
- *Content-Encoding*, *Content-Language*, *Content Location*, *Content-Type*: sono rispettivamente la codifica, la lingua, la locazione sul server e il tipo MIME della risorsa associata.
- *Expires*: data di scadenza della pagina.
- *Last-Modified*: data in cui è avvenuta l'ultima modifica.
- *extension-header*: infine è possibile definire e aggiungere degli header specifici per la propria applicazione, ma se non ci si mette d'accordo tra client e server si rischia di non comprenderli.

Il protocollo HTTP riveste l'applicazione ROA di un'interfaccia uniforme. Per richiedere una risorsa si utilizzerà sempre lo stesso metodo GET, qualunque sia il tipo di risorsa da recuperare. Con una visione RPC, l'azione da eseguire era indicata nel nome del metodo invocato (ad es. *getUtente* per restituire la risorsa utente), portando a molto più complessi.

3.5 Progettare secondo REST.

Il Web lo possiamo concepire come un'insieme di risorse.
Il protocollo HTTP è il mezzo per recuperare le risorse presenti sulla rete. Quindi perché complicare il questo protocollo di trasferimento quando il nostro scopo è smistare risorse ai richiedenti?

Le applicazioni effettuano delle operazioni “classiche” nel mondo della programmazione. Queste sono le operazioni per la *creazione* di una nuova risorsa, per l'*aggiornamento* della risorsa, per il *recupero* della risorsa o per la sua *cancellazione*.

Tutte queste operazioni sono già presenti nello standard HTTP ma in genere vengono bloccate dai server che utilizzano solo i metodi di GET e di POST.

Ciò che propone REST è quindi creare server capaci di accettare in input tutti i metodi HTTP definiti dal protocollo (e casomai dare la possibilità al programmatore di definirne anche altri aggiuntivi) ridefinendoli ed utilizzandoli a proprio favore.

I quattro metodi che per primo vengono ridefiniti nelle applicazioni RESTful standard sono: GET, PUT, POST e DELETE; essenziali per la gestione delle risorse.
Questi quattro metodi sono anche quelli in genere chiamati *CRUD* (create, read, update e delete), metodi basilari nel mondo del software, e quindi associare ad ogni metodo HTTP una di queste quattro funzionalità.

PUT	Corrisponde all'operazione di inserimento di una risorsa nel nostro server.
GET	Corrisponde all'operazione di richiesta di una risorsa.
POST	Corrisponde all'aggiornamento di una risorsa.
DELETE	Corrisponde alla cancellazione di una risorsa.

Non necessariamente le risorse devono essere file presenti sul nostro filesystem. Anzi spesso queste funzioni inseriscono delle informazioni in veri e propri database, o semplicemente in vettori associativi, mantenendole per tutta la durata del server.

Infatti i metodi CRUD vengono anche visti come i metodi INSERT, SELECT, UPDATE e DELETE dei database. Le risorse in questo caso saranno tuple del database (oppure oggetti memorizzati nei vettori associativi).

Altra cosa comune è ampliare il protocollo HTTP aggiungendo nuovi metodi molto usati nel mondo del software. In genere si usano i WebDAV (*Web-based Distributed Authoring and Versioning*), un set di estensioni per il protocollo HTTP che definisce nuovi metodi come MANAGE, MOVE, COPY, RENAME, SAVE e VIEW. Utilizzare questi metodi non significa uscire al di fuori di REST ma porta comunque problemi di compatibilità se l'applicazione deve essere esportata su piatteforme che non supportano questi metodi.

Infine possiamo vedere anche le applicazioni che supportano solo POST e GET come applicazioni RESTful molto limitate. In quel caso si ottiene massima compatibilità perché tutte le piatteforme gestiscono quei due metodi del protocollo HTTP, anche se non è un'applicazione ROA ben formata in quanto il metodo POST deve essere ridefinito per creare, modificare e cancellare risorse.

I metodi come GET e HEAD sono detti *safety* poiché non modificando la risorsa possono essere eseguiti quante volte si vuole. Tutti gli altri metodi invece sono invece *unsafe* e hanno bisogno di controlli più rigidi per non rischiare di intaccare la consistenza dell'applicazione.

3.6 REST e RPC a confronto

Quando progettiamo un'applicazione RPC utilizziamo solamente due metodi definiti dal protocollo HTTP creando tutte le funzionalità basate soltanto su di essi.

Se per esempio vogliamo creare un'applicazione che gestisce delle risorse utenti, possiamo pensare di creare le seguenti operazioni: *inserimento* di un nuovo utente, *elenco* degli utenti presenti sul server, *richiesta* delle informazioni di un utente, *modifica* di un utente, *cancellazione* di un utente e *ricerca* di utenti aventi certe caratteristiche.

Ecco come si propone questo scenario:

- *inserisciUtente:*

  E' una funzione che permette di creare un utente e inserirlo nel nostro server. Utilizziamo il metodo POST per inviare le informazioni relative all'utente, e identifichiamo l'utente con un ID.

- *Utente:*

  E' una funzione che permette di recuperare un utente dal server. In genere è implementato utilizzando il metodo GET, quindi per identificare un utente preciso si fa una richiesta simile: */restituisciUtente?ID=pippo.*

  Possiamo anche pensare di restituire un elenco di utenti che hanno certe caratteristiche (quindi pensare anche ad una funzione di ricerca), ridefinendo il metodo POST e quindi passando i parametri di ricerca nel payload della richiesta Http. (Similmente a REST stiamo effettuando cose diverse utilizzando metodi diversi sulla stessa risorsa... ma con SOA. Ci limitiamo a due metodi e quindi due funzioni diverse: doGet() e doPost().)

- *modificaUtente:*

  E' una funzione che permette di modificare un utente già presente sul server. Utilizziamo il metodo POST per implementare questa funzionalità inviando quindi i parametri nel payload della richiesta.

- *eliminaUtente:*

  E' una funzione che permette di eliminare un utente dal server. In genere viene pensata come GET (ma nulla vieta a farla come POST) in cui per identificare l'utente usiamo la query string: */eliminaUtente?ID=pippo*.

La stessa applicazione pensata utilizzando REST non prevede l'utilizzo di funzioni (utilizzando l'architettura SOA) ma solamente risorse (architettura ROA). In questo esempio quindi ci può bastare un'unica risorsa utente e ridefinire i metodi a nostro favore.

L'unica nostra risorsa è */Utente*, URL di base a cui faremo riferimento per ogni richiesta.

- ***PUT** /Utente/pippo:*

  Il metodo PUT lo ridefiniamo in modo da inserire una risorsa sul nostro server. Le informazioni le inviamo al server inserendole nel corpo della nostra richiesta HTTP. Non utilizziamo più una sintassi come prima (anche nelle POST la sintassi era simile alle query string *nome=pippo&cognome=disney*) ma utilizziamo un linguaggio adatto per mantenere informazioni (in genere l'XML è il più versatile, semplice e comune ma ciò non è una regola). Come si vede il non utilizzare query string rende più facile riferirsi ad utenti particolari; sembra (e possiamo dire che è così) che ogni utente è una nuova risorsa.

- ***GET** /Utente:*

  ***GET** /Utente/pippo:*

  Possiamo utilizzare la GET per implementare due funzionalità diverse: il recupero di una lista di tutti gli utenti presenti sul server (nel primo caso); il recupero di uno specifico utente identificato nel proseguo dell'URL (nel secondo caso).

- ***POST** /Utente/pippo*

  E' una funzione che permette di modificare un utente già presente sul server. Le informazioni necessarie per la modifica dell'utente sono passate nel payload della richiesta HTTP sempre con XML (o il linguaggio usato in precedenza al posto di esso).

- ***DELETE** /Utente/pippo:*

  E' una funzione che permette di eliminare un utente dal server. Non ha bisogno di informazioni aggiuntive poiché l'utente è identificato unicamente dalla URL.

Abbiamo eliminato completamente l'uso delle query string che in genere rendono più complicata la vita. Tutti i parametri che vengono scambiati tra client e server li passiamo con dei linguaggi standard (XML).

Utilizzando un framework si possono pensare a codici molto semplici. Ad esempio per cancellare una qualunque risorsa si può scrivere:

```
□□□□□□□□□□□□□□□ = □□□

□□□□□□□□('□□□□://□□□□□□□·□□□/□□□□□□/001');
□□□□□□□□□□□□□□□·□□□□□□();
```

Non c'è bisogno che il programmatore conosca nel dettaglio ogni tipo di risorsa che gestisce l'applicazione (ad es. *Utente*, *Messaggio*, *Topic* etc..) ma il programmatore usa un'unica classe *Resource*, poiché tutto è una risorsa. Sarà poi a discrezione del framework leggere l'URL della risorsa, capire di che tipo di risorsa si tratta e chiamare il suo costruttore.

Con gli attuali browser ci sono molti problemi di compatibilità. Non tutti permettono di utilizzare metodi diversi da GET e POST (ad esempio il browser *Opera*).
Mentre tutti browser (attualmente) non permettono di utilizzare metodi diversi da GET, PUT, POST, DELETE e HEAD. Ma ancora con il metodo HEAD e PUT non si aspettano (e quindi non leggeranno mai) ciò che viene risposto dal server, o meglio, il payload contenuto nella response del server, ma per questi due metodi il browser si limiterà a leggerne le intestazioni.

Per la PUT non ci sono problemi (a parte il non poter mai sapere se l'inserimento è andato a buon fine, qualora il server lo indicasse), ma per la HEAD non possiamo leggere il payload della response (quindi non potremo effettuare la nostra ricerca in AJAX).

Per aggirare questo problema ciò che si fa (e ciò che ho fatto anche io nella mia applicazione di esempio), è inserire una seconda risorsa fittizia che non sarebbe necessaria

se questa tecnologia (ancora molto giovane) fosse supportata al 100%.

Si può indicare una risorsa */elencoUtenti* e accedervi, per esempio con il metodo POST, inviando nel payload i criteri di ricerca degli utenti per cui siamo interessati.

- ***POST*** */elencoUtenti*:

  E' una ridefinizione del metodo HEAD precedente.

3.7 Dettagli implementativi

[28] Se vogliamo scrivere un'applicazione REST in Java, possiamo sia usare dei framework, oppure scriverla a partire da zero. In questo secondo caso si deve progettare la classe Server, che si occupa di processare le richieste, istanziarla (pubblicandola su un indirizzo URL) e definire le funzioni che si preoccupano di gestire le richieste.

Per fare ciò possiamo sfruttare un'interfaccia Java che si preoccupa proprio di questo: l'interfaccia *Provider*. Scrivere il nostro server significa, dunque, implementare l'interfaccia *Provider* ridefinendo il suo metodo astratto *invoke()*. Questo metodo viene invocato ogni qual volta arriva una richiesta Http al server ed ha, come parametro di ingresso e di uscita, un oggetto che implementa l'interfaccia *Source*. Un oggetto di questo tipo è la rappresentazione di una request o response Http, sia header che payload.

Ecco un esempio:

```
□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□(□□□□□□=□□□□□□·□□□·□□·□□□□□□□□·
□□□□·□□□□□□□)


□□□□□□ □□□□□ □□□□□□□□ □□□□□□□□□□
□□□□□□□□□<□□□□□□□>
 {□□□□□□□□□□
  □□□□□□□□□□□□□□□□□□ □□□□□□□□□□;
```

```
□□□□□□ □□□□□□ □□□□□□(□□□□□□ □□□□□□)
  {□□□□□□□□□□□□□□ □□ =
□□□□□□□□□□·□□□□□□□□□□□□□□□□□□□□();
    □□□□□□ □□□□□□ = (□□□□□□)

□□·□□□(□□□□□□□□□□□□□□·□□□□□□□□□□□□□□□□
□□□);
    □□□□□□ □□□□ = (□□□□□□)
        □□·□□□(□□□□□□□□□□□□□□□·□□□□□□□□□□);
    □□□□□□ □□□□□□□□□ = □□□□;

    □□ (□□□□□□□·□□□□□□□□□□□□□□□□□□□("□□□"))
      {···
       □□□□□□□□□ = ···;
      }
    □□□□ □□
(□□□□□□□·□□□□□□□□□□□□□□□□□□□("□□□□"))
      {···
      }
    □□□□ □□
(□□□□□□□·□□□□□□□□□□□□□□□□□□□("□□□"))
      {···
      }
    ···
    □□□□□□ □□□□□□□□□;
    }

  □□□□□□ □□□□□□ □□□□ □□□□(□□□□□□ □□□□[])
    {□□□□□□□□ □ = □□□□□□□□·□□□□□□(
          □□□□□□□□□□□·□□□□□□□□□□□□□, □□□
□□□□□□□□□());
    □·□□□□□□□□(□□□□□□□□□□□□□□□□);
```

```
        }
    }
```

Creiamo una classe *MyServer* che implementa la nostra interfaccia *Provider* e che quindi implementa il nostro metodo astratto *invoke().*

Le istruzioni che iniziano con la chiocciola sono delle *Annotation*, e permettono di utilizzare alcuni metodi sull'interfaccia *WebServiceContext* anche senza istanziarla (sarebbe impossibile istanziare un'interfaccia). Uno dei metodi che ci interessa è *getMessageContext()* che ci permette di recuperare il *MessageContext* contenente informazioni sulla richiesta. Da esso possiamo recuperare informazioni come il Metodo (*String method*) invocato oppure la risorsa a cui la request fa riferimento (*String path*).
Da queste informazioni possiamo fare un dispatching delle richieste ed eseguire azioni personalizzate per ogni richiesta.
Infine creiamo un *public static void main()* in cui istanziamo una classe *Endpoint* che ci serve per pubblicare il server su un indirizzo.

3.8 Vantaggi e Svantaggi

Il principale vantaggio che si ottiene utilizzando REST è l'estrema semplicità della nostra applicazione. Prima cosa, essa utilizza solo protocolli leggeri, in pratica l'unico protocollo di livello applicazione che utilizza è HTTP. Questo comporta sia vantaggi di tipo tecnologico (il non dover essere legati a tecnologie particolari a volte anche commerciali), sia vantaggi in termini di peso delle request le quali sono molto più brevi.
Inoltre, come abbiamo visto nel capitolo precedente, l'implementazione in Java non significa altro che la scrittura e l'esecuzione di una semplice classe. Questo significa che il server è un'applicazione molto leggera e che può essere eseguita su un qualunque hardware su cui si può installare una JVM compatibile con le classi utilizzate.

Anche i client sono molto versatili. Infatti un client può essere scritto in un qualunque linguaggio di programmazione, può essere ad interfaccia grafica o a linea di comando e soprattutto se scritto anch'esso in Java può essere eseguito su qualunque sistema. Il client deve solo conoscere l'XML ed avere la possibilità di accedere a risorse disponibili sul

web; poi come viene implementato, cosa deve fare e come gestire le risorse sono decisioni del progettista.
Un ultimo vantaggio che salta subito all'occhio è la semplicità di come sono strutturate le risorse. Accedere ad una risorsa è molto semplice avendo eliminato le query string e può essere fatto in maniera meccanica sia da utenti “intelligenti” che da macchine.

A questi pregi, però, si vanno ad aggiungere alcuni svantaggi legati soprattutto al fatto che la tecnologia è molto giovane. Gli svantaggi si riscontrano maggiormente quando vogliamo accedere alle nostre risorse tramite browser e quindi tramite applicazioni AJAX. Infatti le applicazioni AJAX si trovano su server diversi (o almeno su porte diverse) rispetto a dove si trovano le applicazioni REST. Questo comporta problemi nell'effettuare delle richieste asincrone su server diversi da quello in cui si trova l'applicazione AJAX legato soprattuto alla sicurezza per il client.
Per non avere questi svantaggi si possono utilizzare dei Framework che gestiscono l'applicazione Web come un'applicazione REST. Ma essendo la tecnologia non ancora molto diffusa sono pochi i Framework per implementarla.
Altro svantaggio è la difficoltà di reperire informazioni su REST in rete, i Framework sono poco documentati e seppure stanno iniziando a nascere parecchi blog su questo argomento pochi entrano realmente nel dettaglio.

Un Framework interamente Java per programmare REST è *Gomba:* una collezione di Serlvet che permette di implementare velocemente WebServices. Gomba è un progetto open-source che può essere eseguito in un container come Apache Jakarta e facilita la ridefinizione dei metodi HTTP, la connessione ai database, e la creazione di risposte XML.

3.9 Soluzioni per l'accessibilità tramite browser

Quando si implementa un'applicazione REST, si sta scrivendo un nuovo “provider” che viene pubblicato su un server (in ascolto su una particolare porta).

Non può essere pubblicato sulla porta 80 di un server (porta classica per il web) se su quell'host c'è già un server attivo su quella porta, ma deve essere pubblicato o su un altro host o almeno su un'altra porta. Cosa che non avverrebbe se si utilizzasse un framework che molto probabilmente pubblicherebbe le risorse sullo stesso sul web server.

Questo problema comporta che il browser non può aprire una XMLHttpRequest (se viene pensata ad un'applicazione Ajax) sul nuovo indirizzo perché andrebbe incontro alla sicurezza dell'utente.
Un qualunque altro tipo di applicazioni AJAX (per esempio basate su applet java incluse in pagine HTML) comunque non funzionerebbero (il browser blocca ogni tipo di connessione a domini diversi qualunque sia il linguaggio di programmazione client-side).

Per risolvere questo problema abbiamo principalmente tre metodi diversi, ognuno che comporta i suoi problemi.
Il primo metodo più semplice, e che sembrerebbe risolvere completamente il problema, è quello di trasformare un server “proxy” dell'altro. Etichettiamo il server in cui vogliamo mettere la nostra applicazione Ajax con il nome *applicazione* mentre quello REST, in cui abbiamo le nostre risorse, con il nome *dati*. Possiamo pensare di trasformare il server *applicazione* “proxy” del server *dati*, quindi poter accedere ai dati collegandoci ad un particolare path del server *applicazione* senza perdere di astrazione. Quindi scrivere sulla nostra barra degli indirizzi: www.applicazione.it/dati oppure www.dati.it risultano identici. In questo modo possiamo negli script dell'applicazione Ajax inviare richieste asincrone al primo indirizzo che verranno redirette dal server *applicazione* al server *dati* senza intaccare la sicurezza dell'utente. Server come Apache, permettono di fare ciò semplicemente cambiando una riga di codice nel file di configurazione.
Questa soluzione però, porta dei problemi se i nostri dati viaggiano su protocolli protetti con crittazione come HTTPS. Infatti SSL consente una connessione sicura solo tra client e server, ma nel nostro caso il server *dati* non viene invocato direttamente dal client ma dal server *applicazione* che, nei suoi confronti, si comporta come un client. Quindi avremo

una connessione sicura tra *dati* e *applicazione*, ma non una reale connessione sicura tra client e server *dati*.

Un altro metodo potrebbe essere quello di non utilizzare le richieste asincrone utilizzando l'oggetto XMLHttpRequest ma sfruttare il fatto che un browser, quando scansiona la pagina, effettua già delle richieste asincrone per tutti i tag che fanno riferimento ad altri oggetti. Nel fare ciò, inoltre, il browser non considera insicuro scaricare oggetti che si trovano su server diversi da quello in cui si trova la pagina HTML che sta scaricando. Quindi si può risolvere il problema memorizzando i dati sul server (quello etichettato *dati*) non più in formato XML (che dovrebbe essere interpretato da uno script) ma in un nuovo formato JSON (*JavaScript Object Notation*) che come dice il suo nome è già scritto in JavaScript e quindi non deve essere interpretato. Le richieste asincrone vengono effettuate, creando un nuovo tag <*script*> nella pagina HTML, alla risorsa identificata nell'attributo *src* stesso dal browser. Il browser può effettuare una richiesta asincrona ad un qualunque server (senza essere interrotto) e poi, inserendo una *callback*, catturare l'evento *onLoad()* dello script, ed eseguire le azioni che prima mettevamo nella funzione *onreadystatechange()* al richiamo della callback (azioni che non fanno altro che modificare il contenuto della pagina dipendentemente dai dati JSON letti dal server). Vincolo introdotto per utilizzare questo metodo è scrivere i dati in JSON anziché XML.

L'ultima soluzione al problema delle applicazioni Ajax è quella di scrivere del codice specifico per i diversi browser (in genere famiglia Mozilla e famiglia Explorer). Infatti, esistono istruzioni che chiedono dei permessi all'utente (che se accordati) consentono di effettuare richieste asincrone anche su domini diversi da quello che ospita l'applicazione.
Per browser della famiglia Mozilla basta il JavaScript e, dopo aver attivato sulla pagina *about:config* la voce *signed.applets.codebase_principal_support*, è possibile richiedere i provilegi per effettuare le richieste:

□□□

{□□□□□□□□·□□□□□□□□·□□□□□□□□□□□□□□□□·□□□

```
□□□□□□□□□□□□(
                        "□□□□□□□□□□□□□□□□□□□□□□");
  …
 }
□□□□□()
  {□□□□□('□□□□□□□□ □□□□□□!');
   }
```

Se invece stiamo utilizzando il browser Explorer dobbiamo scrivere del codice in *VBScript* (linguaggio della Microsoft interpretato dal browser Explorer).

Ognuno di questi tre metodi, con delle limitazioni, ci consentono di scrivere un'applicazione Web progettata con REST, accessibile tramite browser.

Capitolo 4
Esempi

4.1 Controllo Login PHP-Ajax

L'Ajax può essere utilizzato in maniera complementare ad ogni tipo di tecnologia lato server. In una prima applicazione di esempio, ho utilizzato l'Ajax per validare alcuni campi di un form HTML prima di inviare tutti i dati con l'apposito pulsante di submit. Il form che ho validato è il consueto form di registrazione.

La registrazione prevede l'inserimento di alcune informazioni: *NickName*, *E-Mail* e *Password* che vengono memorizzate in un database MySQL. Come vincolo non possono registrarsi due utenti con stesso *NickName* o stessa *E-Mail*.

Questo controllo può essere fatto quando si invia la richiesta di registrazione una volta inseriti tutti i dati. Ma ciò comporta che se si è scelti un *NickName* già in uso, la registrazione non avrà successo e si chiederà di ripeterla.

Per evitare quest'inconveniente, si può gestire tutto tramite una richiesta asincrona Ajax che, dopo aver inserito i dati nelle caselle di testo, chiede al server se il NickName (o la E-Mail) è già utilizzate. In tal caso informa l'utente di cambiare quel campo prima di cliccare il *button submit*.

Per gestire ciò, ho creato due funzioni PHP che controllano se nel campo del database c'è già presente un certo dato, restituendo il valore *true* o *false*.

```
□□□□□□□□ □□□□□□□□□□□□□□□($□□,$□□□□□)

□□□□□□□□
□□□□□□□□□□□□□□□□□□□($□□,$□□□□□□□□)
```

Queste funzioni devono essere richiamate in script php, e non possono essere richiamate direttamente dal JavaScript.

Le richiamo in due punti diversi.

In un primo punto (usuale funzionamento) lo script *registrazione.php*, recupera i dati

inviati con il submit dell'intero form, li controlla con le due funzioni e se tutto è corretto, inserisce i valori nel database altrimenti restituisce un errore.
In un secondo punto, mi sono servito di due script php (che possono essere intese come viste in XML) *controlloEmail.php* e *controlloNickName.php*, invocabili dal client (ancora con metodo POST) e che accettano in input rispettivamente una variabile *email* e una variabile *nickname*. In output producono un semplice file in XML in cui c'è scritto l'esito dell'operazione.

Adesso non ci resta che collegare, tramite JavaScript, delle funzioni che eseguono richieste asincrone su queste due pagine all'avvenirsi di alcuni eventi (evento *onBlur* sulle caselle di testo *E-Mail* e *NickName*).
Quindi leggere il contenuto della risposta XML (che arriverà dopo un po' di tempo, ce ne accorgeremo grazie all'evento *onreadystatechange*) e, se corretto inserire un'immagine che faccia capire all'utente la correttezza del campo (un visto); se errato inserire un'immagine indicante un errore (un divieto) e un messaggio che spiega il tipo di errore. Nel tempo che intercorre tra l'invio della richiesta asincrona e l'attesa della risposta si può inserire un'immagine indicante un'attesa indefinita.

Da un alto livello di astrazione, possiamo immaginare i due script PHP *controlloEmail.php* e *controlloNickName.php* come dei WebService da invocare che restituiscono una risposta al client in formato XML.

4.2 ParliamoDArte: Elenco Dinamico JSP-Ajax

Un secondo esempio Ajax è contenuto nell'applicazione *ParliamoDArte*. E' un'applicazione scritta in JSP e Ajax utilizzando il framework Dojo e che memorizza i dati in un database MySQL. L'applicazione prevede l'iscrizione di utenti, aggiunta e modifica di informazioni personali (tra le quali anche un'immagine). Poi gli utenti registrati possono eseguire il login ed inserire delle immagini. Per ogni immagine è possibile aggiungere una descrizione, e utenti registrati possono aggiungere dei commenti all'immagine.

Ho utilizzato un filtro per impedire ad utenti non registrati di visualizzare informazioni riguardo utenti registrati o Servlet che gestiscono l'inserimento di dati nuovi. Inoltre ho utilizzato listener per contare quanti utenti sono connessi al sito in ogni istante e mostrarlo in una status line in testa ad ogni pagina.

E' possibile visualizzare tutte le immagini (quadri) immesse nel sito tramite un link “Visita” selezionandole per categoria o in base ad altri criteri di ricerca. Vediamo più nel dettaglio quest'operazione.

Le immagini sono memorizzate in una tabella *Quadro*:

□□□□□□□□: è un numero intero che identifica il quadro;

□□□□: è il nome del quadro;

□□□□: è il nickname dell'utente che ha inserito il quadro nel database;

□□□□□□: è il l'autore del quadro;

□□□□□□□□□□□□: è una descrizione del quadro;

□□□□□□□□□: è la categoria a cui appartiene il quadro;

Una Servlet (*CercaQuadroAction),* mappata su un indirizzo omonimo, effettua una ricerca nel database delle immagini e restituisce un vettore di tuple “Quadri” che risultano rilevanti con i parametri di ricerca. La Servlet fa il dispatching della request ad una vista

che si occupa di formattare il tutto in XML e quindi in un formato standard in modo da poter effettuare una richiesta Ajax.

Ecco il codice della servlet:

```
□□□□□□ □□□□□ □□□□□□□□□□□□□□□□ □□□□□□□□
        □□□□□·□□□□□□□·□□□□·□□□□□□□□□□□
□□□□□□□□□□
        □□□□□·□□□□□□□·□□□□□□□
 {□□□□□□ □□□□□□□□□□□□□□□□□□()
   {□□□□□();
    }
  □□□□□□□□□ □□□□
□□□□□(□□□□□□□□□□□□□□□□□□ □□□□□□□,
        □□□□□□□□□□□□□□□□□□□ □□□□□□□□)
□□□□□□
        □□□□□□□□□□□□□□□□, □□□□□□□□□□
   {□□□□□□□□□□□□□□ □□ =
□□□□□□□□□□□□□□□□□□□();
    □□□□□□□□□□ □□ = (□□□□□□□□□□)
        □□·□□□□□□□□□□□□(“□□□□□□□□□□”);
    // □□□□□□□□ □ □□□□□□□□□ □□□ □□
□□□□□□□:
    □□□□□□ □□□□ =
□□□□□□□·□□□□□□□□□□□□□(“□□”);
    □□□□ □□;
    □□ (□□□□==□□□□)
      □□ = 0;
    □□□□
      □□ =
□□□□·□□□□□□□□□(□□□□□□□·□□□□□□□□□□□□(“□□“
));
    □□□□□□ □□□□ =
□□□□□□□·□□□□□□□□□□□□(“□□□□“);
    □□□□□□ □□□□ =
```

```
□□□□□□□·□□□□□□□□□□□□□(“□□□□”);
    □□□□□□ □□□□□□ =
□□□□□□□·□□□□□□□□□□□□□(“□□□□□□”);
    □□□□□□ □□□□□□□□□ =

□□□□□□□·□□□□□□□□□□□□□(“□□□□□□□□□”);
    // □□□□□□ □□ □□□□□□□ □□ □□□□□□ □□□
□□□□□□□□
    // □□□□□□□ □□□□
    □□□□□□<□□□□□□□> □□□□ =
□□·□□□□□□□□□□□□(□□, □□□□,
                □□□□, □□□□□□, □□□□□□□□□);
    // □□□□□ □□ □□□□□□□□□ □□□□□ □□□□□□□
    □□□□□□□·□□□□□□□□□□□□□(“□□□□□□□□□□□□□”,
□□□□);
    // □□□□□□□□□ □’□□□ □□□□ □□ □□□□□ □□ □□□
□□□□□ □□□
    □□□□□□ □□□ =“/□□□□□□□□□□□□·□□□“;
    // □□ □□□□□□ □□□’□□□
    □□□□□□□□□□□□□□□□□ □□ =

□□·□□□□□□□□□□□□□□□□□□□□(□□□);
    □□·□□□□□□□(□□□□□□□, □□□□□□□□);
   }
 }
```

Il codice (molto semplice e commentato) recupera un Bean (*QuadroBean*) dall'oggetto *Application* (il *ServletContext*) che ci eravamo preoccupati di memorizzare all'avvio del server. Questo oggetto è connesso al database e ci permette di estrarre un vettore di *Quadri* (altra classe che abbiamo implementato per rappresentare una tupla del database in Java) tramite il metodo *cercaQuadri().*

Quindi una volta recuperati dalla request tutti i parametri passati dal client, li inseriamo nel metodo *cercaQuadri* del Bean e abbiamo un vettore di Quadri che memorizziamo

nuovamente nella request. Ora possiamo ridirigere la request alla vista JSP che si occuperà di visualizzare il vettore di Quadri (metodo *forward* sull'oggetto *RequestDispatcher*).

La vista (*ElencoQuadri.jsp*) è completamente scritta in JSP e JSTL, quindi molto semplice da comprendere:

```
<□□□□□□□□□□□□□□>
  <□:□□□□□□□
□□□□□="${□□□□□□□□□□□□□·□□□□□□□□□□□□□}"
                              □□□="□□□□□□□" >
    <□□□□□□>
      <□□><□:□□□ □□□□□="${□□□□□□□·□□}" /></□□>
      <□□□□><□:□□□ □□□□□="${□□□□□□□·□□□□}"
/></□□□□>
      <□□□□><□:□□□ □□□□□="${□□□□□□□·□□□□}"
/></□□□□>
      <□□□□□□><□:□□□ □□□□□="${□□□□□□□·□□□□□□}"
                                /></□□□□□□>
      <□□□□□□□□□□□><□:□□□
□□□□□="${□□□□□□□·□□□□□□□□□□□}"
                               /></□□□□□□□□□□□>
      <□□□□□□□□□□><□:□□□
□□□□□="${□□□□□□□·□□□□□□□□□□}"
                                /></□□□□□□□□□□>
    </□□□□□□>
  </□:□□□□□□□>
</□□□□□□□□□□□□□□>
```

Con il <*c:forEach*> stiamo iterando su tutti gli oggetti del nostro vettore *elencoQuadri* memorizzato nella request dalla Servlet. Per ogni quadro stiamo stampando utilizzando il metodo <*c:out*>. Il risultato finale è un file XML da interpretare con JavaScript.

Una volta invocata la ricerca (mediante un click sul link visita, il riempimento di un form, o un qualunque altro modo) il client viene rediretto su una pagina *RisultatoRicerca.jsp* che si occupa di inizializzare gli script JavaScript per effetture la richiesta asincrona. Utilizzando il framework Dojo, si possono inserire gli script tramite la funzione *dojo.addOnLoad()* da eseguire al caricamento della pagina:

```
<□□□□□□ □□□□="□□□□/□□□□□□□□□□">
  □□□□·□□□□□□□□("□□□□·□□□□□□□");
  □□□□·□□□□□□□□("□□□□□·□□□□·□□□□□□□");
  …

□□□□·□□□□□□□□□□(□□□□□□□□□□□□□□□□□□□□□□□
□□□□);
 </□□□□□□>
```

In questo modo carichiamo tutti gli elementi Widget di Dojo che vogliamo utilizzare e appena si verifica l'evento OnLoad eseguiamo la funziona *InizializzaRisultatoRicerca*.
La pagina contiene molti elementi utili all'applicazione come bottoni per scorrere i risultati per pagine, div per informazioni sul numero di pagine o sul tipo di ordinamento.

InizializzaRisultatoRicerca si occupa di effettuare la richiesta asincrona, richiamando la funzione *leggiXML*, e di mettersi in ascolto a quattro eventi possibili azionati dall'utente:

```
□□□□·□□□□□□□□(□□□□□□□□□□□□□□□□□, '□□□□□□□□',
                   '□□□□□□□□□□□□□□□□□' );
□□□□·□□□□□□□□(□□□□□□□□□□□□□□□□□, '□□□□□□□□',
                   '□□□□□□□□□□□□□□□□□' );
□□□□·□□□□□□□□(□□□□□·□□□□('□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□')

,'□□□□□□□□□','□□□□□□□□□□□□□□□□□□□□□□□□□□□');
□□□□·□□□□□□□□(□□□□□·□□□□('□□□□□□□□□□□□□□□
□□□□□□□□□□□□'),
```

```
'□□□□□□□□','□□□□□□□□□□□□□□□□□□□□');
```

Questi quattro eventi sono: click sul bottone per vedere i risultati precedenti (funzione *paginaPrecedente*), click sul bottone per vedere i risultati successivi (funzione *paginaSuccessiva*), cambio del numero di risultati per pagina (funzione *cambiaRisultatiPerPagina*), cambio dell'ordinamento dei risultati (funzione *cambiaOrdinamento*).

Tutte queste funzionalità, come si può vedere, sono JavaScript, quindi effettuate dal client. Il server non sa nulla di ciò che fa l'utente sui risultati, quindi una qualunque funzione non richiederà al server di recuperare nuovamente i dati dal database (evitando inutili sprechi di risorse).

La funzione *leggiXML* (che si trova nel file *inizializzaRisultatoRicerca.js*) si occupa di leggere i parametri di ricerca che si trovano nella pagina HTML come input hidden, effettuare una richiesta asincrona sull'indirizzo *CercaQuadroAction* e di richiamare la funzione *impostaContenuto,* con parametro il file XML di risposta, quando accade l'evento *onreadystatechange*.

```
□□□□□□□□ □□□□□□□□()
  {// □□□□□ □ □□□□□ □□□□□□□ □□□□ □□□□□□□□□
□□□□ □□□□□ □□□□□□
   □□□ □□□□□□□□□□□□□□□ = '';
   □□ (□□□□·□□□□('□□□□□□□□□□□□□□□□'))
     {□□□□□□□□□□□□□□□ = '□□□□□□□□□=' +

□□□□·□□□□('□□□□□□□□□□□□□□□□')·□□□□□·□□□□□□
□(' ','+');
      }
   □□□ □□□□□□□□ = '';
   □□ (□□□□·□□□□('□□□□□□□□□'))
     {□□□□□□□□ = '□□='+
```

```
□□□□.□□□□('□□□□□□□□□').□□□□□.□□□□□□□(' ','+');
    }

  □□□□.□□□□□□(
   {□□□:
"/□□□□□□□□□□□□□□/□□□□□□□□□□□□□□□□□□□□"+

□□□□□□□□+□□□□□□□□□□□□□□□,
    □□□□□□□□: "□□□", // □□□□□ □□□□ □□□□□□□□
□□□
    □□□□□□□: 10000, // □□□□□□ □□□□ 10 □□□.
    □□□□: □□□□□□□□□(□□□□□□□□, □□□□□□□)
     {// □□□□□□□□ □□ □□□□□□□□ □□□□ □□□. □□□
□□□□□□□□□□□
      □□□ □□□□□□□□□□□□□□□□□□ =
□□□□□.□□□□('□□□□□□□□□□□□');
      □□□□□□□□□□□□□□□□□□.□□□□□□□□ =
□□□□□□□□;
      // □□□□□□□□ □□ □□□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□(□□□□□□□□□□□□□□□□□□);
      □□□□□□ □□□□□□□□;
      },
    □□□□□: □□□□□□□□□(□□□□□□□□, □□□□□□□)
     {□□□□□('□□□□□□ □□ □□□□□□□□.
□□□□□□□□□□');
      □□□□□□ □□□□□□□□;
      }
    });
  }
```

La funzione *leggiXML* deve costruire la URL a cui inviare la richiesta asincrona (quindi la

query string) per effettuare una ricerca di tutte le immagini nel database o vincolata da alcuni criteri di ricerca. Le prime istruzioni servono proprio a recuperare gli oggetti *input hidden* presenti nella pagina (i criteri di ricerca) e creare la query string. Il resto del codice serve per creare una risposta asincrona all'indirizzo desiderato e la modifica della pagina è demandata alla funzione *impostaContenuto* che agisce sull'oggetto *contenitore* e ha a disposizione la risposta del server (oggetto xml) per mezzo di un riferimento nella variabile membro response propria dell'oggetto *contenitore*.
Infine la funzione *impostaContenuto* si occupa di modificare il DOM della pagina HTML inserendo al suo interno alcuni dei dati ricavati dal file XML nella risposta asincrona. In effetti questa funzione non è richiamata solo dopo l'esecuzione della funzione *leggiXML* (che avviene un'unica volta) ma viene richiamata anche nelle funzioni per cambiare pagina (*paginaPrecedente* e *paginaSuccessiva*), e nelle funzioni per cambiare ordinamento o numero di elementi (*cambiaRisultatiPerPagina* e *CambiaOrdinamento*).

L'effetto che un utente ha, è quello di scorrere una lista di elementi senza dover aggiornare ogni volta la pagina HTML.

4.3 ParliamoDArte: WebServices, ServerMeteo e ClientMeteo

Un Web Services può aggiungere ulteriori funzionalità alle applicazioni web. Nell'applicazione precedente ho aggiunto un WebServices che permette di gestire delle informazioni meteo relative ad alcune città e di mostrarle all'interno dell'applicazione. Il WebServices è implementato in Axis2 con la metodologia degli oggetti *Pojo*.

ServerMeteo, l'applicazione server, consta di due classi: *DatiMeteo* che definisce i dati da gestire e *ServiziMeteo* che si occupa di definire le funzioni che i client possono invocare.

La classe *DatiMeteo* contiene due variabili membro: *Float temperatura* e *String tempo*, con i relativi metodi di *set* e di *get*.

Questa è la classe ServiziMeteo che gestisce i servizi:

```
□□□□□□ □□□□□ □□□□□□□□□□□□
 {□□□□□□□ □□□□□□□ □□□□□ = □□□ □□□□□□□□();

  □□□□□□ □□□□□ □□□□□□□□□□□□□□□(□□□□□□
□□□□□)
   {□□□□□□□□□ □□ = (□□□□□□□□□)
□□□□□·□□□(□□□□□);
    □□ (□□==□□□□)
      □□□□□□ □□□□;
    □□□□□□ □□·□□□□□□□□□□□□□□□();
    }

  □□□□□□ □□□□□□ □□□□□□□□(□□□□□□ □□□□□)
   {□□□□□□□□□ □□ = (□□□□□□□□□)
□□□□□·□□□(□□□□□);
    □□ (□□==□□□□)
      □□□□□□ □□□□;
```

```
        □□□□□□ □□·□□□□□□□□□();
        }
    …
```

Queste sono le due funzioni di *get* che possono essere richiamate dal client: *getTemperatura()* e *getTempo().*

I valori vengono recuperati da una HashMap creata allo start-up del server, in cui sono stati inseriti con chiave *String citta* nei seguenti metodi di set:

```
    …
    □□□□□□ □□□□ □□□□□□□□□□□□□□□(□□□□□□
□□□□□, □□□□□
                                    □□□□□□□□□□□)
    {□□□□□□□□□ □□ = (□□□□□□□□□)
□□□□□·□□□(□□□□□);
      □□ (□□!=□□□□)
        {□□·□□□□□□□□□□□□□□□(□□□□□□□□□□□);
          □□□□□·□□□(□□□□□, □□);
        }
      □□□□
        {□□ = □□□ □□□□□□□□□();
          □□·□□□□□□□□□□□□□□□(□□□□□□□□□□□);
          □□□□□·□□□(□□□□□, □□);
        }
      }

    □□□□□□ □□□□ □□□□□□□□(□□□□□□ □□□□□,
□□□□□□ □□□□□)
    {□□□□□□□□□ □□ = (□□□□□□□□□)
□□□□□·□□□(□□□□□);
      □□ (□□!=□□□□)
        {□□·□□□□□□□□(□□□□□);
          □□□□□·□□□(□□□□□, □□);
```

```
        }
    □□□□
      {□□ = □□□ □□□□□□□□□□();
       □□·□□□□□□□□(□□□□□□);
       □□□□□·□□□(□□□□□□, □□);
       }
    }
}
```

Queste due funzioni permettono di inserire dei valori nella HashMap. I valori vengono inseriti con chiave *String citta.*

La HashMap, per ogni chiave, contiene un oggetto di tipo *DatiMeteo* che ci preoccupiamo di inizializzare con i valori che gli sono stati passati dal client.

Ora è possibile effettuare il deploy dell'applicazione dopo aver scritto il file *services.xml* che spiega quali sono i servizi invocabili dal client.

Per richiamare un metodo tramite browser, bisogna collegarsi al server e raggiungere l'indirizzo:

axis2/services/nomeServizio/nomeFunzione?var1=val1&var2=val2

Per esempio per recuperare la temperatura della città di Napoli dovremo scrivere:

axis2/services/ServiziMeteo/getTemperatura?citta=napoli

Se non si vogliono invocare i servizi tramite browser è possibile scrivere un client (ad esempio Java) che li utilizza: applicazione *ClientMeteo*.

Quest'applicazione conterrà le classi definite nel *ServerMeteo* (*ServiziMeteo* e *DatiMeteo*) e importerà le librerie JAR presenti in *webapps/axis2/WEB-INF/lib*.

In aggiunta a queste classi ho definito tre nuove classi *RichiesteMeteo*, *Funzioni* e *Client*. *RichiesteMeteo* è quella con il *main* e si occupa di inizializzare la connessione verso il

server e di istanziare due oggetti, uno *Funzioni* e uno *Client*.

Di seguito riporto la parte saliente dell'applicazione client:

```
□□□□□□ □□□□□ □□□□□□□□□□□□□□□
  {…
   □□□□□□ □□□□□□□ □□□□ □□□□(□□□□□□□[] □□□□1)
    {□□□□□□□□□□□□();
     □□□□□[] □□□□□□□□□□ = □□□ □□□□□□[4];
     □□□□□□□□□□□[0] = □□□□□□□□□□□□□□□□□;
     □□□□□□□□□□□[1] = □□□□□□□□□□□;
     □□□□□□□□□□□[2] = □□□□□□□□□□□□□□□□□;
     □□□□□□□□□□□[3] = □□□□□□□□□□□;
     □□□□□□□□ □ = □□□
□□□□□□□□□(□□□□□□□□□□□□□□□,
                                □□□□□□□□□□□);
     □□□□□□ □ = □□□ □□□□□□□(□);
     □·□□□□□□□□□□□(□□□□□);
     }
   □□□□□□□□□ □□□□□□ □□□□ □□□□□□□□□□□□()
    {□□□ {□□□□□□□□□□□□□□□ = □□□
□□□□□□□□□□□□□□□□□□□□();  }
     □□□□□ (□□□□□□□□□□ □1)
       {□1·□□□□□□□□□□□□□□□□□□();  □□□□□□;}
     □□□□□□□ □□□□□□□□ =
□□□□□□□□□□□□□□□·□□□□□□□□□□□();
     □□□□□□□□□□□□□□□□□□ □□□□□□□□□□ = □□□
□□□□□□□□□□□□□□□□□□□(

"□□□□://□□□□□·□□□/□□□□2/□□□□□□□□□/□□□□□□□□□
□□□□");
     □□□□□□□□·□□□□□(□□□□□□□□□□);
     □□□□□□□□□□□□□□□□□ = □□□
□□□□□□("□□□□://□□□□□·□□□",
```

```
                                        "□□□□□□□□□□□□□□□□");
        □□□□□□□□□□□ = □□□
□□□□□□("□□□□://□□□□□□·□□□",
                                        "□□□□□□□□□");
        □□□□□□□□□□□□□□□□□ = □□□
□□□□□□("□□□□://□□□□□□·□□□",
                                        "□□□□□□□□□□□□□□□□□");
        □□□□□□□□□□□ = □□□
□□□□□□("□□□□://□□□□□□·□□□",
                                        "□□□□□□□□□");
      }
    }
```

La funzione *inizializza* crea un nuovo *RPCServiceClient*, setta l'*EndpointReference* indicandogli la classe remota a cui chiedere i servizi, e inizializza tutte le operazioni che è possibile invocare.

Il resto del *main* non fa altro che costruire un oggetto *Funzioni* passandogli l'*RPCServiceClient*, un vettore con tutte le operazioni e un oggetto *Client* passandogli l'oggetto *Funzioni* creato. L'oggetto *Client* è una classe visuale e si occupa di fornire un'interfaccia grafica con la quale l'utente interagisce per eseguire delle funzionalità definite nella classe *Funzioni*. Quest'ultima ha due metodi: *aggiorna* e *invia*.

Il client è stato pensato per gestire unicamente le 5 province campane, quindi la funzionalità *aggiorna* si occupa di invocare 5 get per le temperature e 5 get per il tempo delle province campane, dualmente *invia* prevede (dopo aver modificato i valori) di allineare i valori sul server con quelli modificati dal client.

```
□□□□□□ □□□□□□□□□[] □□□□□□□□(□□□□□□
□□□□□[])
```

```
  {□□□□□□□□□□[] □□ = □□□
□□□□□□□□□□[□□□□□·□□□□□□];
   □□□ (□□□ □=0; □<□□□□□·□□□□□□; □++)
    {□□□□□□□[] □□□□□□□□□□□□□□□□□□□□□ = □□□
□□□□□□□[]
                                { □□□□□[□] };
     □□□□□□□[] □□□□□□□□□□□□□□□ = □□□ □□□□□□□[]
                                { □□□□□[□] };
     □□□□□□[] □□□□□□□□□□□□□□□□□□□□□ = □□□
□□□□□□[]
                                {□□□□□·□□□□□};
     □□□□□□[] □□□□□□□□□□□□□□□□ = □□□ □□□□□□[]
                                {□□□□□□·□□□□□};
     □□□
      {□□□□□□□ [] □□□□□□□□□□□□□□□□□□□□ =

□□□□□□□□□□□□□□·□□□□□□□□□□□□□□□(□□□□□□□□□
□[0],

□□□□□□□□□□□□□□□□□□□,□□□□□□□□□□□□□□□□
□□□□□);
       □□□□□□□ [] □□□□□□□□□□□□□ =

□□□□□□□□□□□□□□·□□□□□□□□□□□□□□□(□□□□□□□□□
□[1],
          □□□□□□□□□□□□□□□,□□□□□□□□□□□□□□□□);
       □□[□] = □□□ □□□□□□□□□□();
       □□[□]·□□□□□□□□□□□□□□□((□□□□□)

□□□□□□□□□□□□□□□□□□□□[0]);
       □□[□]·□□□□□□□□□((□□□□□□□)
□□□□□□□□□□□□□□□□[0]);
       }
     □□□□□ (□□□□□□□□□ □)
```

```
        {□·□□□□□□□□□□□□□□□□();□□□□□□□□□□□□;}
      □□□□□□□ □□;
      }
  }
```

La funzione *aggiorna* per ogni città recupera il tipo dei parametri di uscita delle funzioni *getTemperatura* e *getTempo*, le invoca (in maniera bloccante perché deve attendere il risultato) passandogli in ingresso degli oggetti (classe *Object*) e aggiorna l'interfaccia grafica.

```
□□□□□□□ □□□□ □□□□□□(□□□□□□□□□□ □□[])
  {□□□ (□□□ □=0; □<□□·□□□□□□□; □++)
    {□□□□□□□[] □□□□□□□□□□□□□□□□□□□□□□□□ = □□□
□□□□□□□[]

{□□[□]·□□□□□□□□□(),□□[□]·□□□□□□□□□□□□□□□□()};
      □□□□□□□[] □□□□□□□□□□□□□□□□ = □□□ □□□□□□□[]
            {□□[□]·□□□□□□□□□(),□□[□]·□□□□□□□□□()};
      □□□

{□□□□□□□□□□□□□□□·□□□□□□□□□□□□□□□(□□□□□□□□□□□[
3],

□□□□□□□□□□□□□□□□□□□□□□□□);

□□□□□□□□□□□□□□□·□□□□□□□□□□□□□□□(□□□□□□□□□□□[4
],
                              □□□□□□□□□□□□□□□□);
        }
      □□□□□ (□□□□□□□□□□ □)
        {□·□□□□□□□□□□□□□□□□□();}
      }
  }
```

La funzione *invia* recupera i dati immessi nell'interfaccia grafica e invoca le funzioni di set passando in ingresso i parametri come *Object*. L'invocazione del servizio può essere fatta con *invokeRobust* che non blocca il client (non dovendo ritornare alcun valore).

Si può anche creare un client Meteo che accede alle informazioni contenute nella HashMap tramite browser e quindi Ajax.
Per fare ciò ho creato una funzione *inizializzaMeteo* che viene richiamata all'evento *OnLoad* sulla pagina.

```
□□□□□□□□ □□□□□□□□□□□□□□□□□()

{□□□□□□□□□□□□□□□□□□□□(□□□□□,'□□□□□□□□□□□□'
);
   □□□□□□□□□□□□□□□(□□□□□,'□□□□□');
   □□□□·□□□□('□□□□□')·□□□□□□□□□ = □□□□□;
   □□□□□□□□□□('□□□□□□□□□□□□□□□□□□□()',10*60*1000);

□□□□·□□□□('□□□□□□')·□□□□□□□□□□□□□('□□□□□□□',
                              "□□□□□□□□□□□();");
   }
```

Questa funzione richiama le funzioni *leggiTemperaturaXML* e *leggiTempoXML* che si occupano di effettuare le richieste asincrone al server. Inoltre viene inserito un timeout che indica di ripetere il tutto dopo dieci minuti e viene collegato un evento *onclick* su di un bottone per aprire una finestra che mostra il meteo di tutta la regione.

Le richieste asincrone vengono fatte utilizzando questi due URL

```
□□□: "/□□□□2/□□□□□□□□/□□□□□□□□□□□/

□□□□□□□□□□□□□□□□□□□□="+□□□□□,
```

```
□□□: "/□□□□2/□□□□□□□□/□□□□□□□□□□□□/
                    □□□□□□□□□□□□□□□="+□□□□□,
```

nelle *dojo.xhrGet* (richieste asincrone con dojo) e il risultato viene preso dall'oggetto response di dojo con quest'istruzione:

```
□□□ □□ =
□□□□□□□□·□□□□□□□□□□□□□□□□□□□□("□□□□□□");
```

Una volta recuperato il risultato è facile modificare la pagina con DOM e inserire i valori in contenitori già presenti nella pagina, oppure cambiare l'attributo src delle immagini per visualizzare cose diverse a seconda del valore tempo letto dal server.

L'applicazione Ajax tramite browser avrebbe potuto modificare i valori esattamente come l'applicazione ClientMeteo Java, anche se non ho sviluppato questa funzionalità immaginando che gli utenti tramite browser vogliano solo visualizzare i dati.

4.4 MeteoMondo: REST

L'applicazione MeteoMondo è un'intera applicazione progettata con paradigma REST. Il codice è scritto interamente in Java costruendo il Provider a partire da zero.

Il *public static void main* si trova all'interno di una classe *MeteoMondo* nel package *meteomondo.server* .

```
…
□□□□□□ □□□□□ □□□□□□□□□□ □□□□□□□□□□
□□□□□□□□□<□□□□□□>
 {…
  □□□□□□ □□□□□□ □□□□□□(□□□□□□
□□□□□□)
   {□□□□□□□□□□□□□□□ □□ =
□□□□□□□□□·□□□□□□□□□□□□□□□□□□();
    □□□□□□ □□□□□□ = (□□□□□□)

□□·□□□(□□□□□□□□□□□□□□·□□□□□□□□□□□□□
□□□□□□);
    □□□□□□ □□□□ = (□□□□□□)

□□·□□□(□□□□□□□□□□□□□□·□□□□□□□□□);
    □□□□□□ □□□□□□□□ = □□□□;


    □□ (□□□□□□·□□□□□□□□□□□□□□□□□□("□□□")
     {□□□□ □□ = □□□□□□□□□□□□□□□□□□(□□□□);
      □□□
       {□□□□ □ = □□·□□□□□□□□□□□□(□□);
        □□□□□□□□ =
□□□□□□□□□□·□□□□2□□□(□, □□);
        }
```

```
        □□□□□
(□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ …
      }
    □□□□ □□
(□□□□□□·□□□□□□□□□□□□□□□□□□(“□□□□“))
      {…

    □□□□□□ □□□□□□□□;
    }

  □□□□□□ □□□□□□ □□□□ □□□□(□□□□□□
□□□□[])
   {□□ = □□□ □□□□□□□□□();
    □□□□□□□□ □ = □□□□□□□□·□□□□□□(
        □□□□□□□□□□□·□□□□□□□□□□□, □□□
□□□□□□□□□□□());
    □□□□□□ □□□□□□□□□□□□□□ = …
    □·□□□□□□□(□□□□□□□□□□□□□□□);
    }
  …
  }
```

Il *main* crea un oggetto *Database* e un provider *MeteoMondo* che pubblica su di un server ad un indirizzo. Il provider MeteoMondo si preoccupa di ricavare il metodo HTTP invocato, recuperare la risorsa sul quale si sta effettuando quel metodo, e reagire alle richieste.

Ad ogni richiesta effettuata viene richiamato il metodo *invoke* che estrae dalla richiesta le

variabili, *metodo* e *path.* La variabile *path* viene inserita in un metodo che ricava l'*id* della risorsa. Tutte le richieste sono del tipo:

□□□ /□□□□□□□□□□/124

Dove GET è uno dei quattro metodi HTTP e 124 è l'id della risorsa (un numero di tipo *Long*). La funzione *recuperaIdDaPath* è scritta all'interno della classe *MeteoMondo* e ritorna il *Long id*.

L'unico metodo che ho mostrato nel codice è il GET. Quando il client effettua una richiesta di GET, il server, dopo aver recuperato l'*id*, recupera la risorsa memorizzata nel database tramite il metodo *restituisci()*, invocabile sull'oggetto *Database*. Infine tramite la classe statica *Trasformer* e il metodo *Dati2XML*, costruisce un oggetto appartenente alla classe *Source* (rappresentante un file XML) e lo ritorna al client.

La classe *Database* appartiene al package *meteomondo.database* e consiste in una *HashMap* che memorizza degli oggetti *Dati* con chiave un valore *Long* (id della risorsa). La classe Database fornisce quattro metodi: cancella(Long), inserisci(Long, Dati), modifica(Long, Dati), restituisci(Long, Dati).
Questi quattro metodi sono utilizzati rispettivamente nelle quattro tipi di request che può effettuare il client: DELETE, PUT, POST, GET.

L'oggetto *Dati* appartiene invece ad un altro package*: meteomondo.dati*; rappresenta la nostra risorsa *MeteoMondo* da gestire ed ha queste variabili membro: *continente, nazione, regione, citta, dm*. Le prime quattro sono di tipo *String* e indicano di che risorsa si tratta. Il quinto parametro è un vettore di tipo *DatiMeteo*, una classe che si trova nello stesso package, che contiene quattro istanze di questa classe. La classe *DatiMeteo* contiene dei dati meteorologici relativi alla città indicata: *condizione, pressione, temperatura, umidita e vento.* Ho scritto un vettore poiché il primo elemento del vettore indica i valori attuali, il secondo elemento indica previsioni a 12 ore, il terzo elemento previsioni a 24 ore e il

quarto elemento previsioni a 48 ore.

I metodi PUT, POST e DELETE, sono implementati in maniera simile al metodo GET richiamando la funzione complementare sull'oggetto *Database*. Per quanto riguarda PUT e POST, esse hanno bisogno di tradurre un file XML che gli è inviato come corpo della request, in un oggetto Dati, questo viene fatto dalla funzione *XML2Dati* sempre della classe statica *Trasformer*.
Questa classe *Trasformer* si trova all'interno di un package *meteomondo.trasformXML* e, per leggere o scrivere file in sintassi XML, usa la libreria JDOM.

L'applicazione potrebbe già risultare completa. Sono stati creati tutti i metodi per creare, modificare, recuperare ed eliminare le risorse. Ma è complicato fare riferimento ad esse per Id (un valore numerico) quando le risorse sono disposte in maniera gerarchica. Infatti ogni risorsa *MeteoMondo* ha come attributi un Continente, una Nazione, una Regione ed una Città. E' evidente che una Nazione è contenuta in un Continente e che un Continente può contenere più Nazioni.

Quindi la proprietà di *addressability* di REST vuole che sia fornito un modo semplice per accedere ad esse. In un primo momento ho modellato l'applicazione inserendo una funzione di ricerca rielaborando il metodo HTTP HEAD in questo modo:

□□□□ /□□□□□□□□□□/

Per avere l'elenco di tutti i continenti presenti nel nostro database;

□□□□ /□□□□□□□□□□/□□□□□□/

Per avere l'elenco di tutte le nazioni presenti nel continente Europa e memorizzati nel database;

□□□□ /□□□□□□□□□□/□□□□□□/□□□□□□

Per avere l'elenco di tutte le regioni presenti nella nazione Italia e memorizzate nel database;

```
□□□□ /□□□□□□□□□□/□□□□□□/□□□□□□/□□□□□□□□
```

Per avere l'elenco di tutte le città presenti nella regione Campania e memorizzate nel database;

Per implementare questa funzionalità ho aggiunto un nuovo metodo nella classe database per effettuare la ricerca:

```
□□□□□□ □□□□□□<□□□□□□□□□□□□□□>
□□□□□□□(□□□□□□ □□□□□□□□□□,
                □□□□□□ □□□□□□□, □□□□□□
□□□□□□□)
```

Questa funzione effettua una ricerca nel database e restituisce un vettore di elementi *RisultatoHEAD*, il quale contiene informazioni riguardo le risorse *MeteoMondo* presenti nel database con quei parametri di ricerca. *RisultatoHEAD* è una nuova classe inserita nel package *meteomondo.dati*; contiene due variabili membro: *String elemento* (una stringa che rappresenta il nome dell'elemento più specializzato nella scala gerarchica della richiesta) e *Long id* (l'eventuale identificativo se si tratta di una città).

Inoltre nella classe *Trasformer* ho aggiunto un nuovo metodo statico *RisutlatoHEAD2XML* che trasforma un Vettore di elementi *RisultatoHEAD* in un oggetto *Source* rappresentante un documento XML adatto a contenere il risultato:

```
<□□□□ □□□□□□□="1.0" □□□□□□□□="□□□-8"□>
<□□□□□□□□□>
  <□□□□□□□□>□□□□□□</□□□□□□□□>
  <□□□□□□□□>□□□□□□□</□□□□□□□□>
</□□□□□□□□□>
```

Se i risultati sono città, quindi rappresentano risorse *MeteoMondo* ogni elemento del documento XML è scritto così:

```
<□□□□□□□□ □□="1">□□□□□□</□□□□□□□□>
```

Infine dobbiamo aggiungere nella classe *MeteoMondo* (quella che implementa il metodo *invoke* dell'interfaccia *Provider*) una modalità di funzionamento per la HEAD.

In questa funzione richiamiamo un metodo (*scomponiPathHEAD*) che recupera tutto il path che è stato inviato e lo scompone utilizzando come carattere separatore "/" e interpretando la prima stringa come continente, la seconda come nazione, la terza come regione e la quarta come città. Se una di queste stringhe dovesse mancare, la variabile che dovrebbe contenerla è posta uguale a null.

```
□□□□ □□ (□□□□□□·□□□□□□□□□□□□□□□□□("□□□□"))
 {□□□□□□ □□□□[] = □□□□□□□□□□□□□□□□□(□□□□);
  □□□□□□ □□□□□□□□□□ = □□□□[0];
  □□□□□□ □□□□□□□□ = □□□□[1];
  □□□□□□ □□□□□□□□ = □□□□[2];
  □□□□□□□<□□□□□□□□□□□□□□□> □ =
□□·□□□□□□□□(□□□□□□□□□□,
                              □□□□□□□□, □□□□□□□□);
  □□□□□□□□□ =
□□□□□□□□□□□·□□□□□□□□□□□□□□□2□□□□(□);
  }
```

L'applicazione Server è completa. Non ci resta che implementare un client che la utilizzi. Io ho implementato un'applicazione client Ajax completamente in HTML e JavaScript che ho chiamato *MeteoMondoClient*. E' formata da un'unica pagina HTML che all'evento *onLoad* effettua una richiesta HEAD asincrona recuperando tutti i continenti presenti nel database e inserendoli in una lista. Cliccando su uno di questi continenti viene effettuata ancora una HEAD che richiede tutte le nazioni presenti in quel continente e li mostra a video e così via. Infine per le città viene effettuata una GET che richiede le relative informazioni meteorologiche e le mostra in una *tooltipDialog*, ovviamente dopo aver effettuato un parsing della risposta che era in formato XML tramite una funzione *xml2html()* presente in uno script *trasformer.js*.

Infine ho implementato dei bottoni per eseguire le richieste di PUT, POST, e DELETE.

PUT e POST sono molto simili; appena si clicca sul bottone si apre una finestra dove

compilare un form. Una volta compilato premendo sul bottone “invia” viene costruito un documento XML, a partire dai valori del form tramite la funzione *form2xml()* presente nel file *trasformer.js*, che viene inviato al server. Si deve indicare l'id della risorsa a cui inviare il documento XML stesso nel form.
L'implementazione della funzione DELETE è più semplice (non dovendo inviare payload XML nella richiesta asincrona), si deve solo indicare che risorsa cancellare dal database.

Il problema di questa applicazione è che i browser non si aspettano una risposta quando vengono eseguiti i metodi di HEAD e di PUT. Per la PUT non ci sono problemi, ma per la HEAD l'applicazione creata non leggerà mai il risultato XML.
Per aggirare questo problema ho fatto in modo che il server potesse eseguire una HEAD anche con metodo GET, indicando esplicitamente oltre alla risorsa da invocare anche la funzione di ricerca.
Quindi queste due richieste per il server saranno equivalenti:

□□□□ /□□□□□□□□□□/□□□□□□/□□□□□□

□□□ /□□□□□□□□□□/□□□□□/□□□□□□/□□□□□□

Fare ciò nell'applicazione è molto semplice, basta cambiare i due **if** che smistano le richieste in questo modo:

Istruzione **if** del metodo GET:

```
□□ (□□□□□□·□□□□□□□□□□□□□□□□□("□□□") && !

□□□□·□□□□□□□□□□□□()·□□□□□□□□("□□□□□"))
```

Istruzione **if** del metodo HEAD:

```
□□ (□□□□□□·□□□□□□□□□□□□□□□□□("□□□□") ||
        (□□□□□□·□□□□□□□□□□□□□□□□□("□□□")

&&□□□□·□□□□□□□□□□□()·□□□□□□□□("□□□□□")) )
```

Mentre il browser deve trasformare tutte le richieste di HEAD in richieste che contengono come prima campo la parola chiave CERCA.

Conclusioni

Le nuove applicazioni Web 2.0 possono essere viste come l'insieme di due parti separate. Da un lato ci sono i servizi da invocare (nel caso in cui si tratta di applicazioni SOA) o risorse da gestire (applicazioni ROA) che si occupano di elaborare i dati e restituirli agli utenti in un linguaggio standard.
Dall'altro lato abbiamo l'applicazione vera e propria che si preoccupa di gestire i dati in maniera corretta per effettuare transazioni complete.

E' come se avessimo scisso il pattern *Model-View-Controller* e assegnato dei compiti al client e altri al server, i quali implementano indipendentemente il pattern.
Il *Model*, come nelle vecchie applicazioni web 1.0, è un'esclusiva del server e ha il solito compito di gestore di informazioni.
La *View* è un'esclusiva del client, il server occupandosi solo di restituire i dati non ha bisogno di sapere come il client vuole visualizzarli. E' prerogativa del client tradurli in un linguaggio a lui noto (HTML per i browser, WML per i palmari, etc...).
Infine il *Controller* potrebbe essere sviluppato da entrambe le parti. Implementare un Controller lato client, significa che è il programmatore a definire una sequenza di azioni da fare in catena per ottenere una funzionalità. Questo da una flessibilità notevole, il programmatore può modificare la sequenza di azioni a suo piacimento per ottenere risultati leggermente diversi rendendo l'intera funzionalità più idonea al suo utilizzo. D'altro canto, sviluppare un Controller lato server permette di aggregare un'insieme di funzionalità base per crearne alcune più complesse, che possono rendere più facile al client il loro utilizzo. Infatti al client basterà richiamare la funzionalità completa una sola volta, anziché fare più richieste per funzioni più semplici, ma con il vincolo di non poter modificare la sequenza e di dover scendere ad un livello di programmazione più basso se si vogliono fare cose più particolare.

Questo ci porta a preferire il Web 2.0 al vecchio Web. Infatti stiamo diminuendo il traffico

tra client e server in quanto il cliente ad ogni richiesta non chiederà informazioni su come visualizzare i dati ma richiederà solo i dati.
Stiamo offrendo ad ognuno la possibilità di creare un proprio programma client che gestirà le risorse come meglio crede.
Inoltre, stiamo creando una netta separazione tra Model, View e Controller rendendo più veloce lo sviluppo dell'applicazione e minimizzando la complessità di ogni parte (chi scriverà le View si occuperà solo di grafica senza necessitare conoscenze di linguaggi di programmazione).

Scegliere il Web 2.0 come modello per un'applicazione significa scegliere uno tra i due paradigmi: SOA e ROA.
Scegliere ROA significa a sua volta avere dei vantaggi:

- Richieste più snelle eliminando SOAP e WSDL dal payload di HTTP;
- Usufruire di Interfacce Uniformi date proprio dal metodo HTTP. L'interfaccia è costituita da uno dei *metodi* HTTP che indica l'operazione da eseguire sulla risorsa, la *url* che identifica la risorsa e il *payload* che specifica eventuali parametri della request secondo un formato semplice e standard;
- Compatibilità assoluta sul web. Tutti i linguaggi di programmazione sanno usare correttamente il protocollo HTTP;
- Un migliore supporto per le reti mobili. Le informazioni da trasmettere richiedono meno banda e sia i server che i client, essendo più semplici, richiedono meno CPU, l'ideale per dispositivi piccoli e wireless;
- Eliminazione delle query string che porta ad un miglior indirizzamento alle risorse;
- Una semplice gestione dello stato. HTTP è un protocollo stateless mentre un'architettura SOA utilizza tecniche complesse per mantenere uno stato tra client e server che richiamano procedure remote.

Anche ROA comporta alcuni problemi di compatibilità, soprattutto quando viene sfruttata

pienamente la *method information* trasportata del protocollo HTTP. Quindi la soluzione migliore (almeno finché tutti i client e i server non si adeguino allo standard) è usare una metodologia ibrida.

Infatti utilizzare solo i metodi HTTP GET e POST significa poter definire solo due operazioni da inviare con la request. Quindi, in genere, si usa lasciare il metodo GET solamente per recuperare i dati, e ridefinire il metodo POST per tutte le altre operazioni. Per indicare quale operazione, tra tutte quelle possibili, si vuole eseguire quando si effettua una request si può aggiungere una *side information* o all'interno del path a cui si sta facendo riferimento oppure nel payload della request.

Bibliografia

[1] Guida HTML di *Wolfgang Cecchin* al sito: http://xhtml.html.it/guide/leggi/51/guida-html/

[2] Storia del web su wikipedia agli indirizzi: http://it.wikipedia.org/wiki/Web_browser e http://it.wikipedia.org/wiki/Html

[3] Guida CSS di base di *Cesare Lamanna* al sito: http://css.html.it/guide/leggi/2/guida-css-di-base/

[4] Complete CSS2 Reference al sito: http://www.w3schools.com/css/default.asp

[5] Guida PHP di base, teorica e pratica rispettivamente di *Gianluca Gillini* e *Gabriele Farina* agli indirizzi: http://php.html.it/guide/leggi/99/guida-php-di-base/ http://php.html.it/guide/leggi/97/guida-php-teorica/ http://php.html.it/guide/leggi/101/guida-php-pratica/

[6] Manuale PHP (function references) al sito: http://asp.html.it/guide/leggi/65/guida-asp/

[7] Guida ASP di base e teorica di *Giorgio Gobbo* e *Andrea Carratta* ai siti: http://asp.html.it/guide/leggi/62/guida-asp-di-base/ http://asp.html.it/guide/leggi/65/guida-asp/

[8] Guida ASP.NET di *Angelo Ranucci* al sito: http://aspnet.html.it/guide/leggi/98/guida-aspnet-20/

[9] JavaServer Pages™, 2nd Edition di Hans Bergsten; editore: *O'Reilly*

[10] Introduzione ad Apache Struts (pattern Model-View-Controller) di Simone Pascuzzi al sito: http://java.html.it/articoli/leggi/2176/introduzione-ad-apache-struts/

[11] Guida JavaScript di base di *Ilario Valdelli* al sito: http://javascript.html.it/guide/leggi/25/guida-javascript-di-base/

[12] Ajax in Action di Dave Crane e Eric Pascarello; edizione *Manning*

[13] Manuale Javascript (Function References) al sito: http://www.w3schools.com/js/default.asp

[14] Guida DOM al sito: http://javascript.html.it/guide/leggi/24/guida-dom/

[15] Guida XML di base di *Andrea Chiarelli* al sito:
http://xml.html.it/guide/leggi/58/guida-xml-di-base/

[16] Introduzione al Web 2.0 al sito: http://en.wikipedia.org/wiki/Web_2.0

[17] Articolo: What Is Web 2.0 di Tim O'Reilly del 09/30/2005 al sito:
http://oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html

[18] Articolo: Building Rich Web Application with Ajax di *Linda Dailay Paulson* Ottobre 2005

[19] The Book of Dojo 1.0 (documentazione del framework) al sito:
http://dojotoolkit.org/book/dojo-book-1-0

[20] I Web Service su wikipedia all'indirizzo:
http://en.wikipedia.org/wiki/Web_service

[21] SOAP e WSDL su wikipedia agli indirizzi:
http://en.wikipedia.org/wiki/Web_Services_Description_Language
http://en.wikipedia.org/wiki/SOAP

[22] Apache Axis2 Documentation al sito: http://ws.apache.org/axis2/

[23] Representional State Tranfer su wikipedia all'indirizzo:
http://en.wikipedia.org/wiki/Representational_State_Transfer

[24] RESTful Web Services di Leonard Richardsone e Sam Ruby; editore *O'Reilly*

[25] Architectural Styles and the Design of Network-based Software Architectures di Roy Thomas Fielding, 2000
http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

[26] Uniform Resource Identifier su wikipedia all'indirizzo:
http://it.wikipedia.org/wiki/URI

[27] HTTP - Hypertext Transfer Protocol all'indirizzo:
http://www.w3.org/Protocols/

[28] RESTful Web Services al sito:
http://java.sun.com/developer/technicalArticles/WebServices/restful/