Appunti di Tecnologie Web

Anna Corazza

aa 2021/22

First printing, July 2017

Indice

Prefazione

Questi appunti sono stati raccolti come aiuto per gli studenti del mio corso di Tecnologie Web ed esclusivamente a loro uso. Vanno considerati solo un supporto alle lezioni e non una loro sostituzione. Infatti, i testi in essa contenuti vengono spiegati e completati durante la lezione. Per alcuni capitoli, vengono segnalate delle fonti esterne per lo studio: esse sono parte integrante del programma, e quindi il loro studio è **obbligario**, a meno che non sia esplicitamente detto il contrario. Si noti che ho scelto sempre fonti disponibili gratuitamente su web. In particolare, essi vengono inseriti con la finalità di aiutare gli studenti ad acquisire la capacità di mantenersi aggiornati autonomamente.

Questi appunti possono inoltre contenere errori: eventuali segnalazioni mi permetteranno di correggerli e quindi migliorare il supporto che posso offrire agli studenti.

Terminologia e notazione

Avvertenze Il carattere '\' viene usato nel codice per spezzare righe troppo lunghe per essere riportate nel testo. Le sigle tra parentesi quadre indicano invece i riferimenti bibliografici riportati alla fine del volume.

Si ricorda qui di seguito la traduzione dall'inglese di alcuni comuni termini tecnici usati all'interno di questi appunti.

Header : intestazione.
Mark-up language : linguaggio di annotazione.
Scope : ambito di definizione (di una variabile).
Signature : prototipo (di un metodo).
Statement : istruzione.
Tag : etichetta.

1. Introduzione

In questa lezione cercheremo di capire cos'è il web, quali sono le sue componenti fondamentali e quali le funzionalità di ciascun componente. Per eventuali approfondimenti (facoltativi), si può far riferimento al sito del W3C[1] e a [Eck02]. Per statistiche sembre aggiornate sulle dimensioni del web (oltre che di internet) si veda `http://www.internetlivestats.com/`.

1.1 World Wide Web: introduzione

Il World Wide Web (WWW) è molto recente: nasce nel 1994. Padre del WWW è universalmente considerato **Tim Berners-Lee**,[2] nel periodo in cui lavorava al CERN, tra il 1989 e il 1991. Nel 1989, infatti, scrive un primo proposal in cui già si riconosce la prima idea di WWW (che però lui chiama "mesh"). Tuttavia, l'idea in qualche modo fondante su cui si basa il WWW è quella di **ipertesto**, che risale agli anni '40-'60.

Le componenti chiave di questa prima versione del web:

1. la versione iniziale di HTML (HyperText Mark-up Language);
2. la versione iniziale di HTTP (HyperText Transfer Protocol);
3. server web;
4. browser.

L'applicazione **browser** è essenzialmente un **client HTTP**, e quindi prepara le richieste HTTP da inoltrare al server web considerato e interpreta le relative risposte. In questa prima versione molto schematica di WWW, la risposta HTTP conteneva un documento HTML da visualizzare. Quindi, tra le funzionalità essenziali di un browser sta la **visualizzazione di oggetti web**, che possono essere documenti HTML, ma non solo.

1.1.1 Ipertesti

Normalmente un documento testuale o **testo** ha una struttura sequenziale, nel senso che si compone di una stringa di caratteri che viene letta dall'inizio alla fine. Nelle lingue a noi più familiari, infatti

[1] `https://www.w3.org/`

[2] Una simulazione del primo sito web è messo a disposizione dal CERN in `http://info.cern.ch/`

un testo si legge da sinistra verso destra e dall'alto verso il basso.

Un **ipertesto** è un testo a cui sono stati aggiunti degli archi con direzione o **link** che rappresentano associazioni. Un ipertesto quindi non si presta ad una semplice lettura sequenziale, ma ad essere **navigato**, nel senso che in diversi punti del testo si presenta la scelta se proseguire la lettura o seguire il link: la direzione indica appunto il verso di lettura, per cui il link può venir seguito in quel verso ma non nel verso opposto. Si noti che i link possono essere sia **interni**, se diretti verso una parte del documento da cui hanno origine, che **esterni**, se sono diretti ad un diverso documento.

La struttura associata ad un ipertesto è quindi quella di un grafo con direzione, in cui i nodi corrispondono a documenti testuali e gli archi a link. Tale struttura a grafo con direzione viene ereditata anche dal web, che può essere visto come un grande ipertesto in cui i link esterni vengono seguiti attraverso richieste HTTP. Nell'evoluzione del web verso una sempre maggiore multimedialità, i nodi non si riferiscono necessariamente solo a documenti testuali, ma più in generale a **oggetti web**, che possono essere immagini, suoni, filmati, etc..

L'introduzione del concetto di ipertesto e i primi studi su come rappresentarlo precedono l'avvento del web, e hanno portato alla definizione dell'SGML (Standard Generalized Markup Language) come di un linguaggio generale che permette di definire una grammatica per testi, in particolare per il tipo di etichettatura (mark-up) che contengono. L'HTML è un'istanza dell'SGML.

Possiamo quindi considerare i seguenti quattro linguaggi utili per rappresentare gli ipertesti:

1. SGML
2. HTML: HyperText Markup Language
3. XML: eXtensible Markup Language
4. XHTML: eXtensible HyperText Markup Language

L'**HTML** (HyperText Markup Language) è un **linguaggio di etichettatura** (in inglese, markup language) per gli ipertesti. Un browser deve quindi essere in grado di rendere fruibile l'informazione contenuta nel documento HTML e "seguire" i link. Seguire un link significa preparare una richiesta HTTP da spedire al server web identificato dal link per ottenere la risorsa desiderata.

I primi browser non avevano un'interfaccia grafica, ma con modalità a linea. Alla fine del '92 erano funzionanti circa 200 server web. Ne esistono ancora: *lynx*, *links*, ...

Esercizio

Installate uno di questi browser senza interfaccia grafica e provatelo: vi darà un'idea più chiara di quali sono le funzionalità principali che deve implementare un browser.

Nello stesso periodo, molti ricercatori stavano sviluppando browser web con interfacce grafiche: tra gli altri, Marc Andreesen sviluppò Mosaic (versione preliminare nel 1993, Mosaic Communications nel 1994, che diventò poi la Netscape Communications Corporation).

1.1.2 Il Consorzio del W3C

Nell'ottobre 1994, Tim-Berners-Lee ha fondato il World Wide Web Consortium (W3C) presso il Laboratorio di Computer Science del Massachusetts Institute of Technology in collaborazione col CERN e con il supporto del DARPA e della Commissione Europea. Il W3C [3] rimane un'istituzione fondamentale nello sviluppo delle tecnologie web a cui noi faremo spesso riferimento.

Non si tratta di un produttore di strumenti, ma di protocolli e linee guida atte ad assicurare lo sviluppo a lungo termine del web. Lo sviluppo di nuovi **standard** vede la partecipazione non solo delle organizzazioni membri e dello staff permanente del consorzio, ma anche dell'opinione pubblica attraverso processi di condivisione e sviluppo.

Il concetto di standard sta alla base del principio di interoperabilità, visto che non è possibile pensare alla coordinazione tra strumenti diversi se non ci si accorda sui formati e protocolli di scambio. In generale, quindi, per i vari strumenti noi cercheremo di studiare le specifiche prodotte

[3] `www.w3.org`

dal W3C, invece che le singole implementazioni, che devono rispettare tali specifiche, ma possono differenziarsi in altri punti.

La definizione degli standard non può che basarsi su una discussione il più ampia possibile tra tutti gli enti coinvolti. Il Consorzio ha messo a punto un codice etico a cui i diversi enti devono adeguarsi e che consiste nelle seguenti regole[4]:

1. Treat each other with respect, professionalism, fairness, and sensitivity to our many differences and strengths, including in situations of high pressure and urgency.
2. Never harass or bully anyone verbally, physically or sexually.
3. Never discriminate on the basis of personal characteristics or group membership.
4. Communicate constructively and avoid demeaning or insulting behavior or language.
5. Seek, accept, and offer objective work criticism, and acknowledge properly the contributions of others.
6. Be honest about your own qualifications, and about any circumstances that might lead to conflicts of interest.
7. Respect the privacy of others and the confidentiality of data you access.
8. With respect to cultural differences, be conservative in what you do and liberal in what you accept from others, but not to the point of accepting disrespectful, unprofessional or unfair or unwelcome behavior or advances.
9. Promote the rules of this Code and take action (especially if you are in a leadership position) to bring the discussion back to a more civil level whenever inappropriate behaviors are observed.

1.1.3 I client HTTP: i browser

Nel **1995** si diffondono i browser Mosaic e Netscape. Società commerciali pubblicano e mantengono server web che mettono in grado l'utente (non necessariamente l'utente generico) non solo di accedere a informazione, ma anche di eseguire operazioni commerciali sul web. L'evento importante del **1996** è l'entrata della Microsoft nella lotta dei browser: qualche anno dopo risultò vincitrice con Internet Explorer.

Si apre cosí la strada ad una moltitudine di nuove applicazioni multimediali, tra cui la trasmissione di video ad altà qualità on-demand e la videoconferenza interattiva ad alta qualità.

1.1.4 Cosa si intende per World Wide Web?

Il World Wide Web è un'applicazione **client/server** operante su Internet e reti intranet TCP/IP (Transmission Control Protocol/Internet Protocol) attraverso il protocollo **HTTP** (HyperText Transfer Protocol). Le applicazioni web vedono la rete come fornitore di servizi.

1.2 Servizi forniti da Internet

1.2.1 Terminali, client e server

I computer collegati a Internet vengono chiamati in modo equivalente *host* o *terminali* o *end system*. Un'applicazione **client** è un'applicazione che usa un servizio da un programma **server**. Client e server possono venir eseguiti su macchine diverse oppure sulla stessa macchina. Questo modello **client/server** è la struttura prevalente per le applicazioni Internet (Web, e-mail, trasferimento di file, login remoti, newsgroups). Tipicamente, client e server vengono eseguiti su macchine diverse: sono quindi **applicazioni distribuite** che interagiscono scambiandosi messaggi in Internet.

Non in tutte le applicazioni un programma puramente client interagisce con un programma puramente server: nelle applicazioni per la condivisione di file **da pari a pari** (in inglese, peer-

[4] `https://www.w3.org/Consortium/cepc/`

to-peer), l'applicazione agisce in alcuni casi da client (quando richiede un file) e in alcuni casi da server (quando li fornisce).

1.2.2 Internet come *infrastruttura* per applicazioni

Distribuzione delle applicazioni su Internet: login remoti, posta elettronica, navigazione web, messaggistica in tempo reale, streaming audio e video, telefonia in Internet, giochi fra più componenti, condivisione di file peer2peer (P2P), ...

Web non è altro che **una delle molte applicazioni distribuite** che usano il servizio di comunicazione fornito da Internet.

1.2.3 Protocolli

Tutte le attività in Internet che coinvolgono almeno due entità remote atte alla comunicazione sono gestite da un protocollo che definisce:

1. formato e ordine dei messaggi scambiati tra due o più entità comunicanti;
2. le azioni che hanno luogo a seguito della trasmissione e/o ricezione di un messaggio o di altri eventi

Tra i più importanti protocolli di Internet abbiamo:

TCP (Transmission Control Protocol, protocollo di controllo della trasmissione);

IP (Internet Protocol, protocollo Internet): specifica il formato dei pacchetti che sono scambiati tra router e tra terminali;

HTTP (HyperText Transmission Protocol);

I browser implementano il lato client dell'**HTTP**: per questo spesso vengono anche indicati semplicemente come **client**. Esempi di browser sono Netscape Communicator, Microsoft Internet Explorer, Mozilla, Safari, e molto altri. Tuttavia è riduttivo pensare ai browser come unico possibile client HTTP, o, in altre parole, come unico possibile client di un'applicazione web. Infatti, spesso a fungere da client sono programmi, quali ad esempio bot o crawler. Per questo motivo, quando ci si riferisce al client di un'applicazione web, molto spesso si usa il termine più generale di **user agent**.

Un **server web** funge da deposito di oggetti web, ognuno dei quali indirizzabile da un URI; inoltre implementa il lato server di **HTTP**. Tra i server web abbiamo Apache, Microsoft Internet Information Server, Netscape Enterprise Server.

1.2.4 Pagine web

La crescita del web è stata immediatamente molto veloce, sia se si considera come parametro di stima il numero di pagine accessibili, che rispetto al numero di query inviate dai motori di ricerca. Nel 1994, uno dei primi motori di ricerca, il World Wide Web Worm (WWWW) usava un indice riferito a circa 110 000 pagine web e documenti accessibili dal web. Nel novembre 1997, i principali motori di ricerca dichiaravano di indicizzare da 2 milioni (WebCrawler) a 100 milioni di documenti web (Search Engine Watch). Oggi Google dichiara di indicizzare miliardi di pagine. Per quel che riguarda i motori di ricerca, in marzo/aprile 1994 il World Wide Web Worm riceveva in media circa 1500 queries al giorno. Nel Novembre 1997, Altavista dichiarava di gestire attorno ai 20 milioni di query al giorno. Nell'aprile 2006, solo all'interno degli Stati Uniti, Google dichiarava circa 91 milioni di query al giorno, Yahoo 60 milioni, per un totale di circa 213 milioni di query tra tutti i vari motori di ricerca.

Pagina Web è un termine piuttosto generico che indica un insieme di oggetti web il cui rendering viene fatto contemporaneamente perché strettamente collegato: ad esempio, una pagina di testo in cui sono incluse immagini e video. Un **oggetto web** è una risorsa, in genere un file, indirizzabile da uno specifico Uniform Resource Locator o URL; ad esempio:

- un file HTML;
- un'immagine JPEG o GIF;

- un applet Java;
- un audio clip;
- etc.

Molte pagine web consistono di un documento HTML principale e da vari oggetti indirizzati dai corrispondenti Uniform Resource Identifier o URI. Un **URI** ha la forma `schema : dettagli`. Il termine URI è più generale del termine URL perché comprende anche gli Uniform Resource Names o URN. La differenza tra URN e URL assomiglia a quella tra il nome di una persona e il suo indirizzo: un URN serve solo a identificare univocamente un oggetto (ad esempio un libro col suo codice ISBN 0-486-27557-4 (urn:isbn:0-486-27557-4) (si tratta di un'edizione in lingua originale del "Romeo e Giulietta" di Shakespeare. Non ci dice però come accedervi. Al contrario, un URL potrebbe avere la seguente forma: file:///home/username/books/RomeoAndJuliet.pdf che ci dice come arrivare a una copia di quella risorsa salvata in locale. Ne consegue che URN e URL danno informazioni complementari.

Un **URL** quindi è un URI che dà le seguenti informazioni:

1. il nome del protocollo (nel nostro caso, http)
2. il nome dell'host su cui si trovano gli oggetti;
3. il path degli oggetti sull'host.

Esempi di URL:

- http://www.unina.it/...
- http://wpage.unina.it/anna.corazza/
- https://www.docenti.unina.it/anna.corazza (si noti che in questo caso il protocollo utilizzato è la versione sicura di HTTP)
- http://www.w3c.org/TR/2004/REC-xmlschema-2-20041028/
- etc

Esempio di pagina web: si consideri una pagina web che contiene cinque immagini JPEG: in totale la pagina è composta da 6 oggetti, ovvero 1 file HTML e 5 file JPEG. La pagina principale fa riferimento alle immagini tramite i loro URI.

Abbiamo già visto che un **browser** è prima di tutto un client HTTP; quindi prepara richieste HTTP e interpreta le risposte HTTP. Tale interpretazione comporta ad esempio la visualizzazione di una pagina HTML, ma non solo. Le informazioni possono arrivare all'interno della risposta HTTP in vari formati, testuali, grafici, video, audio, multimediali. Oltre a ciò il browser offre altri tipi di funzionalità, tra cui supporto alla configurazione.

Come dicevamo sopra, il browser è un tipo di user agent. Il concetto di **user agent** (agente dell'utente) è del tutto generale e indica un'interfaccia tra l'utente e l'applicazione; per fare un altro esempio, si pensi alla posta elettronica, per la quale lo user agent è rappresentato dal lettore di posta (Microsoft Outlook, Eudora, pine, etc.).

Nel caso del web, altri tipi di user-agent sono **sistemi di information retrieval**, che si muovono nel web alla ricerca di informazioni (crawler, che vedremo nel capitolo 15) e gli **interpreti VXML (VoiceXML)** che hanno le stesse funzionalità del browser ma interagiscono con l'utente usando solo la modalità vocale

Ricapitolando, il web può essere visto un enorme sistema client/server, in cui tutti i client e i server coesistono contemporaneamente sulla medesima rete. Tuttavia, in ogni istante la connessione riguarda **un solo** server, anche se due interazioni successive possono riguardare due server anche lontanissimi tra loro.

1.3 Dal web statico al web dinamico

Inizialmente il web era caratterizzato da un tipo di scambio informativo **monodirezionale**: il client inviava una richiesta ad un server, che in cambio inviava un file, visualizzato dal browser sul client.

Tale schema, però, ha subito evidenziato i suoi limiti: sarebbe stato utile che il client potesse associare alla richiesta alcuni parametri.

Rapidamente la comunicazione è diventata quindi **bidirezionale**, permettendo al client di passare alcune informazioni al server. Queste informazioni potevano servire a "parametrizzare" la richiesta (per esempio, se si vogliono ottenere i prezzi di alcuni prodotti) oppure a passare al server in modo definitivo alcuni dati (ad esempio, dati da inserire in una base di dati). Ovviamente, il protocollo HTTP deve dare supporto a questo scambio di informazione bidirezionale.

A questo punto, però il server deve essere in grado di leggere ed elaborare i parametri, adattando la risposta alla specifica richiesta, specificata insieme ai parametri. Viene quindi introdotta la **programmazione lato server**.

Alla base di tutta questa architettura sta il vincolo che la stessa informazione deve venir mostrata nello **stesso** modo da tutti i client, quindi da tutti i browser. Anche se è impossibile soddisfare questo vincolo alla lettera, la necessità di assicurare che l'aspetto dell'applicazione sia lo stesso su tutti i browser è uno dei grossi problemi dello sviluppo di applicazioni web. Si parla a questo proposito di **portabilità** inter-browser.

Al momento **non esiste uno standard** a cui i browser debbano conformarsi neppure per il primo tra i linguaggi usati sul web, ovvero l'HTML. Il W3C dà delle direttive, ma benché abbiano un certo numero di caratteristiche in comune, ogni browser può implementare una resa diversa per gli stessi file di ingresso. Esempi di questi standard sono l'XHTML e, più di recente, l'HTML5.

I primi browser erano molto primitivi: pensati per semplici ipertesti, non erano in grado di supportare nessuna interattività: l'esecuzione di qualsiasi compito era demandata al server, compreso il controllo ortografico di ogni richiesta, e quindi **inefficiente**. In compenso ne giovava la **sicurezza**, perché i browser non potevano eseguire alcunché, neppure codice malevolo o pericoloso per eventuali bachi in esso contenuti.

Ora sono stati introdotti nuovi strumenti che permettono di aggiungere dinamicità al browser, quindi lato client:

- **fogli di stile** in grado di definire opportunamente la rappresentazione del documento, adattandola ai vari media disponibili lato client, ma con un'interattività limitata; tra i fogli di stile noi vedremo CSS nel Capitolo 11 e XSL nel Capitolo **??**;
- **programmazione lato client**, incluse le applet Java, ma soprattutto JavaScript (Capitolo 12) e AJAX nel Capitolo 14.

L'**interattività** fornita dall'architettura server-browser iniziale era tutta **a carico del server**, il server non faceva che produrre **pagine statiche** che il browser interpretava e visualizzava.

HTML di per sé comprende solo semplici meccanismi per la raccolta di dati: diversi tipi di form che permettono di raccogliere testi o di scegliere tra diverse opzioni, liste, pulsanti per rincominciare da capo o per sottomettere la richiesta.

Il primo meccanismo introdotto per raccogliere ed elaborare questa informazione lato server è il **Common Gateway Interface** (CGI) messo a disposizione da tutti i server web. L'indicazione di cosa fare con la sottomissione si trova codificata nella richiesta HTTP stessa: una delle azioni più comuni è di eseguire un programma che sta sul server nella directory che tipicamente si chiama *cgi-bin*. Questi programmi possono essere scritti in qualsiasi linguaggio, ma tipicamente si usava molto il Perl, per le sue ottime caratteristiche nella manipolazione di testi. In linea di principio praticamente qualsiasi operazione può essere implementata col meccanismo del CGI. Vedremo il CGI nel Capitolo 4. I problemi legati a questo meccanismo sono:

- manutenibilità
- tempo di risposta, che dipende da:
  1. dimensione dei dati trasferiti;
  2. carico del server;
  3. carico della rete.

Ad esempio, l'implementazione di un'applicazione che faccia uso della **grafica dinamica** richiederebbe di spostare dal client al server e indietro dal server al client diversi file GIF (Graphics Interchange Format) corrispondenti ad ogni versione del grafico. Come altro esempio, si pensi a cosa succede per la validazione delle informazioni inserite dall'utente attraverso una form.

La **programmazione lato client** rappresenta una soluzione a questo problema di efficienza. Nelle prime applicazioni web, la macchina su cui veniva eseguito il browser veniva sfruttata per una percentuale infinitesima, anche perché molte delle macchine da cui si accedeva al web non avevano una potenza di calcolo sufficiente ad elaborazioni computazionalmente pesanti. Con la programmazione lato client, il browser esegue parte del lavoro, rendendo possibile una maggiore interattività ed efficienza. Ovviamente questo richiede maggiore sforzo computazionale al client.

In generale, la programmazione lato client è sostanzialmente programmazione, ma all'interno del browser, che costituisce l'ambiente ristretto all'interno del quale vengono eseguite tutte le elaborazioni. Uno dei modi di realizzare la programmazione lato client è attraverso **plug-in**. Vengono aggiunte nuove funzionalità al browser scaricando un pezzo di codice che si incastra nel browser, permettendogli, da quel momento in poi, di compiere una determinata attività. Basta scaricare il plug-in **una sola** volta.

I plug-in sono utili per fornire al browser funzionalità veloci e potenti; tuttavia, la loro implementazione non è un compito banale, e non è normalmente annoverata tra il lavoro da fare per costruire un nuovo sito. Da un punto di vista della programmazione lato client, il valore dei plug-in sta nella possibilità per il programmatore esperto di sviluppare un nuovo linguaggio e aggiungerlo al browser senza dover chiedere il permesso al produttore del browser. Tuttavia, anche se il plug-in offre un modo di implementare nuovi linguaggi di programmazione lato client, non tutti tali linguaggi sono implementati via plug-in.

L'effetto dell'introduzione dei plug-in fu un'esplosione dei **linguaggi di scripting** per la programmazione lato client. Un linguaggio di scripting permette di inserire il codice direttamente nella pagina HTML: il corrispondente plug-in viene attivato dal browser durante la visualizzazione della pagina.

Vantaggi dei linguaggi di scripting:

- di facile uso e comprensione;
- veloci da caricare (testo nella pagina HTML).

Svantaggio:

- codice esposto e facile da copiare,

ma di solito non si costruiscono programmi particolarmente sofisticati con un linguaggio di scripting.

I linguaggi di scripting usati nei browser web sono usualmente progettati per risolvere problemi specifici, in particolare per la creazione di interfacce grafiche (GUI = Graphical User Interface) più ricche e interattive. Un linguaggio di scripting risulta adeguato alla soluzione di un 80% dei problemi per cui si usa la programmazione lato client: in questi casi il loro uso permette maggiore **facilità e velocità di sviluppo**, e li fa quindi preferire a linguaggi di programmazione veri e propri, quali Java o ActiveX.

Tra i linguaggi di scripting più diffusi: **JavaScript**, **VBScript**, e **Tcl/Tk**, derivato dal popolare linguaggio inter-piattaforma per lo sviluppo di GUI. Noi studieremo solo il primo.

Per il restante 20% dei problemi per cui un linguaggio di scripting non risulta adeguato, una scelta molto diffusa è quella di usare **Java**, a cui dedicheremo più di una lezione. Java si applica alla programmazione lato client attraverso le *applet* e *Java Web Start*, un modo relativamente recente di distribuire programmi standalone che non hanno bisogno di un browser in cui venire eseguiti.

Una **applet** è un programma che viene eseguito nel browser: viene scaricata come un qualsiasi oggetto web all'interno della pagina e, quando attivata, manda in esecuzione il programma. In questo modo il software viene distribuito dal server al client esattamente quando ce n'è bisogno, e non prima, permettendo di fornire sempre la versione più aggiornata, senza bisogno di reinstallazioni.

Data la caratteristica di Java di essere compilato in un bytecode, non occorre creare diverse versioni del programma adatte alle diverse piattaforme, ma lo stesso bytecode può venir eseguito da tutti i browser provvisti di un interprete Java. Dal momento che Java è un linguaggio di programmazione completo, è possibile eseguire tutta l'elaborazione richiesta al client prima di mandare la richiesta al server.

Ne risulta una reazione più veloce ed efficace, oltre ad un alleggerimento del carico sia della rete che dei server, con vantaggi su tutta Internet. Al contrario dei linguaggi di scripting, una applet Java viene scaricata nella sua forma compilata (bytecode): anche se non è immediatamente manipolabile, esso può essere decompilato abbastanza facilmente.

Tuttavia i punti da tener presenti nella scelta di quale linguaggio utilizzare sono altri:

- tempo per scaricare una applet;
- curva di apprendimento perché il programmatore diventi efficace col linguaggio considerato.

1.3.1 Programmazione lato server

Le richieste possono semplicemente riguardare degli oggetti web, quali pagine HTML, immagini grafiche, applet Java, script, etc..

In molti casi, però, le richieste sono più complicate e spesso coinvolgono l'accesso ad una **base di dati**:

- è possibile che la richiesta sia già stata elaborata dal lato client,
- comunque viene inviata al server che la elabora e inoltra alla base di dati
- sempre il server prepara la risposta della base di dati in una pagina web (ad esempio, un documento HTML) che rimanda al client

Altri casi sono la registrazione dei propri dati in una base di dati quando ci si registra in un gruppo o si inoltra un ordine: è un caso importante perché richiede di modificare una base di dati.

Le elaborazioni dal lato server sono state implementate tradizionalmente con linguaggi come Perl, Python, C e C++, o con programmi CGI.

Approcci più recenti includono **servlets** e **portlets**, su server basati su Java. Notiamo che un vantaggio è quello di non dover dipendere dalle funzionalità del browser.

1.4 Internet versus intranet

Man mano che le applicazioni sviluppate su web divenivano sempre più numerose e mature, cresceva l'interesse a farne il porting su reti intranet aziendali, che usano la stessa tecnologia, ma sono più piccole e con minori problemi di sicurezza. Le applicazioni web sono più facili da imparare per l'impiegato qualunque: ormai chiunque sa usare un browser, o comunque lo impara con facilità.

Nello sviluppo di applicazioni web occorre prestare molta attenzione al fatto che il codice sviluppato per il lato client possa venir eseguito su piattaforme molto diverse senza bachi. Quando invece si sviluppano applicazioni per una intranet, si può contare su un ambiente molto più controllato, in cui il tipo di piattaforme coinvolte sia (in generale) noto a priori. Inoltre, in un'intranet, spesso esiste del (prezioso) codice legacy sviluppato per applicazioni tradizionali da portare su web.

Gli aggiornamenti del codice possono venir fatti via browser, e questo è un altro vantaggio delle applicazioni web anche su intranet.

1.5 La sicurezza sul WWW

Scaricare ed eseguire automaticamente programmi su Internet sono operazioni che presentano il rischio di diffusione di virus. Gli oggetti che si scaricano da web possono essere di vari tipi, tra i quali:

- file GIF, che non sono pericolosi;
- codice di script, che di solito hanno grossi limiti su quello che possono fare;
- codice Java compilato, la cui sicurezza è assicurata dal fatto che le applet vengono eseguite in una specie di "scatola di sabbia" (*sandbox*), al di fuori della quale non possono né accedere alla memoria né scrivere su disco;
- componenti ActiveX, che sono estremamente pericolosi, perché non ci sono limiti a quello che possono fare.

Le **firme digitali** possono rappresentare una soluzione, nel senso che certificano chi è l'autore. Non rappresentano però una soluzione per i bachi non intenzionali e comunque, il tempo intercorso tra quando il codice viene scaricato e quando ci si accorge del danno, può rendere impossibile rintracciare l'autore.

In generale si tratta di un problema importantissimo per chi si occupa di applicazioni web, che tuttavia non affronteremo, visto che servirebbe un corso a sé stante. Cercheremo invece via via di porre attenzione almeno ad alcuni degli aspetti più immediati.

A differenza di altri casi, il web è vulnerabile anche agli attacchi **contro i server**. In caso di attacco ad un server web usato dalle aziende sia per promuovere marchio e prodotti, che per transazioni commerciali, si corrono gravi rischi sia **economici** che **in termini di reputazione**. I recenti sviluppi nelle implementazioni di browser e server che hanno reso i primi sempre più facili da utilizzare e i secondi sempre più facili da configurare e gestire, con contenuti sempre più facili da sviluppare, richiedono un software estremamente complesso, in cui possono facilmente annidarsi vulnerabilità. Colpendo un server web può essere possibile ottenere l'accesso a dati e sistemi privati (attacco alla privatezza dell'informazione).

Gli utenti delle applicazioni web sono spesso non esperti degli aspetti informatici, non in grado quindi di prendere le contromisure necessarie. Gli attacchi possono essere:

- **passivi**: tra cui, intercettazione del traffico di rete tra un browser e un server, accesso a informazioni riservate;
- **attivi**: tra cui, la simulazione di altri utenti, la modifica dei messaggi in transito fra client e server e l'alterazione delle informazioni contenute in un sito web.

e possono riguardare:

- il server web;
- il browser web;
- il traffico di rete in transito tra server e browser.

1.6 Web e dispositivi mobili

La diffusione pervasiva dei dispositivi mobili è forse l'aspetto più evidente della tecnologia attuale. Il W3C ha creato un gruppo di interesse [5] per cercare di rendere le tecnologie web più adatte allo sviluppo di applicazioni per dispositivi mobili.

[5] `http://www.w3.org/Mobile/IG/`

2. Il protocollo HTTP

HTTP: come funziona e quali servizi offre? [KR03] Per la parte su cookies e privacy, maggiori chiarimenti possono essere trovati in `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Referer` and `https://gdpr.eu/cookies/`.

2.1 Il protocollo HTTP (HyperText Transfer Protocol)

Il protocollo HTTP rappresenta un po' il cuore del web, che di fatto è un'applicazione client-server. Una caratteristica importantissima del protocollo HTTP è di essere **senza stato**: questo significa che il protocollo non prevede che né il server né il client mantengano informazioni sullo stato della comunicazione. Ovviamente questo significa che se un'applicazione web ha bisogno, e di solito è così, di mantenere delle informazioni sullo stato della comunicazione, deve implementare dei meccanismi che lo facciano, non essendo sufficiente appoggiarsi al protocollo.

HTTP, come protocollo di strato di applicazione, deve definire:

- tipo dei messaggi scambiati (richiesta e risposta)
- sintassi dei vari tipi di messaggio
- semantica dei diversi campi (= significato dell'informazione in essi contenuta)
- regole che determinano quando e come un processo invia messaggi o risponde a messaggi

HTTP prevede esclusivamente due tipi di messaggio, entrambi in formato **testuale**:

1. **richiesta** dal client verso il server;
2. **risposta** dal server verso il client;

Entrambi i tipi di messaggio sono composti da:

1. una prima riga, che chiameremo **request line** nella richiesta e **status line** nella risposta;
2. un certo numero di **righe di intestazione** (anche nessuna)
3. una riga vuota;
4. il **body** del messaggio.

Due tipi di **messaggi** HTTP: di richiesta e di risposta.

Ecco un esempio di messaggio di **richiesta**:

```
GET/somedir/page.html HTTP/1.1
```

```
Host: www.someschool.edu
Connection: close
User-agent: Mozilla/4.0
Accept-language:fr
```

La prima riga viene chiamata **request line** e ha tre campi:

1. **metodo**: in HTTP/1.0:
   - GET: richiede un oggetto;
   - POST: invia informazione;
   - HEAD: come GET, ma restituisce solo intestazione; usato per debugging;

   In HTTP/1.1, abbiamo anche:
   - PUT: carica un oggetto;
   - DELETE: cancella un oggetto.
   - altri (TRACE, CONNECT)
2. **URL**: oggetto richiesto;
3. **versione HTTP**.

Seguono le linee di **intestazione**, di cui abbiamo dato solo qualche esempio:

1. Host
2. *Connection: close* significa che non usa connessione persistente
3. User-agent: permette di condizionare la risposta sul tipo di user-agent
4. molti altri, e HTTP/1.1 ne ha introdotti ancora di nuovi

Dopo una riga vuota, segue il **body**, che nell'esempio è vuoto.

Ricapitolando: nello scambio di informazione bidirezionale, l'informazione passa dal client al server all'interno della richiesta HTTP in un modo che dipende dal metodo HTTP adottato. Se si usa il GET, l'informazione viene passata come parametri alla fine dell'URL. Se invece si usa un POST, l'informazione viene passata nel body del messaggio.

Passiamo ora a considerare un esempio di messaggio HTTP di **risposta**, che viene preparata dal server e passata al client: contiene dunque l'informazione che il browser visualizzerà. :

```
HTTP/1.1 200 OK
Connection: close
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 09:23:24 GMT
Content-Length: 6821
Content-Type: text/html

dati ... dati ...
```

La prima riga viene detta **status line** e ha tre campi:

- versione del protocollo
- codice di stato:
  - `200 OK`
  - `301 Moved Permanently`: il nuovo URL sta nell'intestazione in `Location:`, in modo che il client possa accedervi automaticamente;
  - `400 Bad Request`
  - `404 Not Found`
  - `505 HTTP Version Not Supported`
- corrispondente messaggio di stato

Per quel che riguarda i codici, ovviamente ce ne sono molti di più rispetto a 5 riportati. Seguono la seguente convenzione:

1xx. (Informativo): la richiesta è stata ricevuta, l'elaborazione continua;
2xx. (Successo): la richiesta è stata non solo ricevuta, ma anche compresa e accettata;
3xx. (Redirezione): sono necessarie ulteriori operazioni per portare a buon fine la richiesta;
4xx. (Errore lato client): la richiesta contiene errori sintattici e non può quindi venire accolta;
5xx. (Errore lato server): il server non è in grado di soddisfare una richiesta che appare corretta.

Si noti che il file corrispondente all'oggetto web viene trasferito cosí com'è e non ha nulla a che fare con HTTP: sarà poi responsabilità del browser (o del generico user agent) sapere come visualizzarlo (o, nel caso più generale, come elaborarlo), e non è detto che la visualizzazione (meglio: il rendering) implementata da due client diversi sia esattamente la stessa. HTTP si occupa solo ed esclusivamente della comunicazione tra client e server Web. Occorre quindi che l'informazione sul tipo di file sia passata dal server al client: questo è il ruolo della linea di intestazione `Content-Type:`. Se assente, si intende `text/html`.

2.1.1 Esempio: richiesta HTTP via telnet

Si provi a fare una richiesta HTTP via telnet. Ad esempio, da una macchina UNIX:

`telnet wpage.unina.it 80`

apre una connessione TCP alla porta 80 dell'host `telnet wpage.it 80`

`GET /anna.corazza HTTP/1.0`

Occorre inserire due a capo per segnalare la fine dell'header.

```
corazza@desktop:~$ telnet wpage.unina.it 80
Trying 192.132.34.19...
Connected to wpage.unina.it.
Escape character is '^]'.
GET /anna.corazza/index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 04 Sep 2018 07:12:51 GMT
Server: Apache
Last-Modified: Sat, 04 Aug 2018 17:02:01 GMT
ETag: "cf5399-81c-b85440"
Accept-Ranges: bytes
Content-Length: 2076
Connection: close
Content-Type: text/html; charset=ISO-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 \
                    Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>

<head>
  <title>Anna Corazza: home page</title>
...
</body>
</html>
```

Se sostituiamo il metodo GET con il metodo HEAD

```
corazza@desktop:~$ telnet wpage.unina.it 80
```

```
Trying 192.132.34.19...
Connected to wpage.unina.it.
Escape character is '^]'.

HEAD /anna.corazza/index.html HTTP/1.0

HTTP/1.1 200 OK
Date: Tue, 04 Sep 2018 07:18:18 GMT
Server: Apache
Last-Modified: Sat, 04 Aug 2018 17:02:01 GMT
ETag: "cf5399-81c-b85440"
Accept-Ranges: bytes
Content-Length: 2076
Connection: close
Content-Type: text/html; charset=ISO-8859-1
```

Altri metodi HTTP/1.0: POST, HEAD... e HTTP/1.1: PUT, DELETE, ma attenzione, con HTTP/1.1 occorre mettere una riga di header con Host, anche vuoto:

```
corazza@desktop:~$ telnet wpage.unina.it 80
Trying 192.132.34.19...
Connected to wpage.unina.it.
Escape character is '^]'.
HEAD /anna.corazza/index.html HTTP/1.1
Host:

HTTP/1.1 200 OK
Date: Tue, 04 Sep 2018 07:23:01 GMT
Server: Apache
Last-Modified: Sat, 04 Aug 2018 17:02:01 GMT
ETag: "cf5399-81c-b85440"
Accept-Ranges: bytes
Content-Length: 2076
Content-Type: text/html; charset=ISO-8859-1
```

Exercise 2.1 Ripetere l'operazione richiedendo la risorsa home.html per lo stesso dominio. ■

Exercise 2.2 Ripetere l'operazione usando server e URL diversi e provando a cambiare i diversi parametri della richiesta HTTP, quali il metodo e le righe dell'intestazione. ■

2.2 Esempi con uno strumento per l'esame dell'HTTP

Per costruire gli esempi riportati di seguito abbiamo utilizzato l'applicazione SERVISTATE reperibile in https://www.servistate.com/.

Exercise 2.3 Installare l'applicazione suggerita o altra equivalente. Esercitarsi per essere in grado di commentare i risultati ottenuti nei casi di seguito e in loro eventuali varianti. ■

Richiesta

```
GET /wpage.unina.it/anna.corazza/index.html HTTP/1.1
```

```
Host:
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) \
                                                Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,it;q=0.8
Cookie: __utma=64676715.1882009769.1515421363.1515421363.1515421363.1
```

Risposta

```
HTTP/1.1 200 OK
date: Tue, 04 Sep 2018 06:51:56 GMT
last-modified: Sat, 04 Aug 2018 17:02:01 GMT
server: Apache
accept-ranges: bytes
etag: "cf5399-81c-b85440"
content-length: 2076
content-type: text/html; charset=ISO-8859-1

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" \
                                    "http://www.w3.org/TR/html4/loose.dtd">
<html>

<head>
  <title>Anna Corazza: home page</title>
    ...
</body>
</html>
```

Exercise 2.4 Analogie e differenze tra il risultato ottenuto con telnet e con SERVISTATE. ■

Mandiamo ora una richiesta all'URL http://www.w3c.org/, che corrisponde ad una vecchia versione del sito del consorzio.

Richiesta

```
GET /www.w3c.org HTTP/1.1
Host:
User-Agent: Mozilla/5.0 (X11; Linux x86_64) \
     AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,it;q=0.8
```

Risposta

```
HTTP/1.1 301 Moved Permanently
Content-length: 0
Location: http://www.w3.org/
```

Come già sapevamo, il sito è stato spostato. In genere, quando il browser riceve una risposta di questo tipo, prepara autonomamente una nuova richiesta da inviare all'URL indicato nel campo `Location` dell'header. Possiamo simulare questo comportamento con `SERVISTATE` impostando `REDIRECTS` a `ON`.

Proviamo ora ad inviare una richiesta al nuovo URL.

Richiesta

```
GET /www.w3.org HTTP/1.1
Host:
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) \
                                Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,it;q=0.8
```

Risposta

```
HTTP/1.1 200 OK
date: Tue, 04 Sep 2018 08:51:12 GMT
content-encoding: gzip
last-modified: Mon, 03 Sep 2018 15:00:10 GMT
etag: "ad51-574f8cc309280;89-3f26bd17a2f00-gzip"
vary: negotiate,accept,Accept-Encoding,upgrade-insecure-requests
content-type: text/html; charset=utf-8
cache-control: max-age=600
tcn: choice
accept-ranges: bytes
content-location: Home.html
content-length: 10392
expires: Tue, 04 Sep 2018 09:01:12 GMT

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" \
                  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<!-- Generated from data/head-home.php, ../../smarty/{head.tpl} -->
<head>
...
```

2.2.1 Meccanismo di cache

In quest'ultima risposta possiamo notare che il sito ci permette di adottare un meccanismo di cache: invece di scaricare tutta la pagina ogni volta che accediamo al sito, ne conserveremo sempre l'ultima copia e la scaricheremo solo quando è necessario. Naturalmente se fosse il client a dover decidere quando la pagina è stata modificata, il meccanismo non funzionerebbe. Perché un meccanismo di cache possa venir adottato proficuamente, occorre che il client sia in grado di controllare se la pagina richiesta dall'utente è stata modificata oppure se quella che è stata salvata in cache è ancora valida senza riscaricare la pagina. Per farlo, HTTP prevede il campo **If-Modified-Since** da inserire nell'intestazione della richiesta e il codice **304 Not Modified** da inserire nella status line della risposta. Potremo quindi riaccedere al sito chiedendo di scaricare la pagina solo se è stata modificata.

Richiesta

```
GET /www.w3.org HTTP/1.1
Host:
If-Modified-Since: Tue, 04 Sep 2018 09:40:12 GMT
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
                    (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
```

```
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,it;q=0.8
```

Risposta

```
HTTP/1.1 304
date: Tue, 04 Sep 2018 09:41:56 GMT
etag: "ad64-574f8cc309280;89-3f26bd17a2f00"
vary: negotiate,accept
status: 304
cache-control: max-age=600
content-security-policy: upgrade-insecure-requests
strict-transport-security: max-age=15552000; includeSubdomains; preload
content-location: Home.html
expires: Tue, 04 Sep 2018 09:51:56 GMT
```

dove il codice 304 sta per `NOT MODIFIED`

Come ulteriore esempio, consideriamo la richiesta seguente è rivolta ad ottenere un oggetto web a cui si fa riferimento all'interno di questa home page, e più precisamente un foglio di stile.

Richiesta

```
GET /www.w3.org/StyleSheets/home.css HTTP/1.1
Host:
If-Modified-Since: Tue, 04 Sep 2018 09:40:12 GMT
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
              (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,it;q=0.8
```

Risposta

```
HTTP/1.1 304
date: Mon, 03 Sep 2018 14:31:26 GMT
content-encoding: gzip
last-modified: Mon, 16 May 2005 15:59:15 GMT
etag: "609-3f73b56693ac0-gzip"
vary: Accept-Encoding,upgrade-insecure-requests
content-type: text/css
status: 304
cache-control: max-age=604800
content-security-policy: upgrade-insecure-requests
strict-transport-security: max-age=15552000; includeSubdomains; preload
expires: Mon, 10 Sep 2018 14:31:26 GMT
```

2.2.2 Session tracking: cookies e autenticazione

I seguenti esempi riguardano vari aspetti dell'HTTP e in particolare il meccanismo di session tracking via cookies e via autenticazione e il meccanismo di cache. L'adozione di strategie di **session tracking** è resa necessaria dal fatto che il protocollo HTTP è **senza stato**, nel senso che i messaggi HTTP non contengono informazioni sugli scambi precedenti tra client e server. Tali informazioni risultano però necessarie almeno in alcuni casi per lo svolgimento delle funzionalità richieste dall'utente. In questi casi occorre provvedere con programmazione lato server e/o lato client.

In questo capitolo considereremo le due strategie di session tracking (cookies e autenticazione) che si appoggiano su delle caratteristiche previste dal protocollo HTTP, e in particolare su specifici campi dell'intestazione. Nel capitolo 10 tratteremo in modo più esaustivo le tecniche di session tracking.

Session tracking via cookies

Un **cookie** è un pezzo di testo che viene scambiato tra client e server secondo opportune modalità. Esso contiene alcune informazioni essenziali:

1. host e path dell'applicazione;
2. expiration date;
3. nome e valore del cookie;
4. altro.

Ogni cookie viene sempre creato dal server, che lo passa al client nel campo **set-cookie** dell'intestazione della risposta HTTP. Il client **può** decidere di salvarlo. Questo ad esempio avviene se in un browser i cookies sono abilitati per quella particolare applicazione, univocamente determinata dall'host e dal path contenuti all'interno del cookie.

Richiesta

```
GET /www.dieti.unina.it HTTP/1.1
Host:
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
                      (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,it;q=0.8
```

Risposta

```
HTTP/1.1 303 See other
Date: Tue, 04 Sep 2018 10:26:33 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.26
Set-Cookie: 500542cb87c7389c01175a86ab33264d=5aq2m540k12v3b10pj9s0nno01; path=/
Location: http://www.dieti.unina.it/index.php?lang=en
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 20
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

NB

Non conosco il codice `303 See other`, a parte sapere che 3xx indica una redirezione. Lo cerco quindi sul sito del W3C `https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html`

> **10.3.4 303 See Other**
>
> The response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource. This method exists primarily to allow the output of a POST-activated script to redirect the user agent to a selected resource. The new URI is not a substitute reference for the originally requested resource. The 303 response MUST NOT be cached, but the response to the second (redirected) request might be cacheable.

> The different URI SHOULD be given by the Location field in the response. Unless the request method was HEAD, the entity of the response SHOULD contain a short hypertext note with a hyperlink to the new URI(s).

Ne concludiamo quindi che il sito ci redirige su una risorsa a cui viene passato come parametro la lingua di default del browser.

Ogni volta che lo user-agent prepara una richiesta per una certa applicazione identificata all'interno dell'host da un particolare path, inserisce all'interno dell'intestazione della richiesta tutti i cookie che ha memorizzato e che corrispondono a quella combinazione di host e path. Quindi se ripeto la stessa richiesta di prima, il browser aggiungerà all'header una riga col cookie.

Richiesta

```
GET /www.dieti.unina.it HTTP/1.1
Host:
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) \
                                        Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,it;q=0.8
Cookie: 500542cb87c7389c01175a86ab33264d=5aq2m540k12v3b10pj9s0nno01
```

A questo punto il server riceve il cookie e quindi evita di inserire nella risposta il campo `Set-Cookie`. Si noti che la scelta se inserire questo campo nell'intestazione della risposta non può venir eseguita automaticamente dal server, ma richiede **programmazione lato server**.

Risposta

```
HTTP/1.1 303 See other
Date: Tue, 04 Sep 2018 11:58:12 GMT
Server: Apache/2.2.22 (Ubuntu)
X-Powered-By: PHP/5.3.10-1ubuntu3.26
Location: http://www.dieti.unina.it/index.php?lang=en
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 20
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
```

L'expiration date serve a consigliare allo user-agent quando cancellare il cookie. Se l'expiration date è negativo, allora lo user agent dovrebbe cancellare il cookie immediatamente. Questo serve perché ogni cookie è univocamente determinato da host, path e nome. Se il server manda un cookie con la stessa combinazione di queste tre informazioni, allora il cookie viene sovrascritto. Se il server vuole che un cookie venga cancellato, ne invia quindi uno con lo stesso nome di quello da cancellare, in modo che quest'ultimo venga prima sovrascritto e immediatamente cancellato.

Session tracking via autenticazione

Consideriamo il sito `https://auth-demo.aerobaticapp.com/protected-standard/` che ci fornisce una demo per l'autenticazione basata su HTTP. Se noi proviamo ad accedere al sito otteniamo un messaggio 401 Authorization Required.

Richiesta

```
GET /protected-standard/ HTTP/1.1
Host: auth-demo.aerobaticapp.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
            (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,it;q=0.8
```

Risposta

```
HTTP/1.1 401
x-aero-deploy-stage: production
date: Tue, 04 Sep 2018 13:20:43 GMT
content-encoding: gzip
x-aero-app-id: e25a180f-afa1-447a-bbc8-a7c8addc54df
x-powered-by: Aerobatic
x-cache: Error from cloudfront
status: 401
x-amzn-trace-id: Root=1-5b8e86aa-44559658e19cb1fc6e556414;Sampled=0
server-timing: load-app=11.56974, get-version=4.422639, total=50.562554; \
                                                    "Total Response Time"
x-amz-apigw-id: Msn6tEcEliAFoAw=
content-length: 1089
x-amz-cf-id: bYJkyCaws4QzdRU2Qyj-6dlhKz1g1EMYmy5_DpCkRrxyYcrAs_93SQ==
x-aero-version-name: v5
x-amzn-requestid: 53311e1d-b045-11e8-83ab-f9625975a889
etag: "d-9H2B/dgx9Gre3+DoMBMCAdyVH5o"
vary: Accept-Encoding
content-type: text/html; charset=utf-8
via: 1.1 5f98af95e7d2e10f47bd5c14f60ab7de.cloudfront.net (CloudFront)
cache-control: public, max-age=31536000, no-cache
x-aero-app-last-modified: Sat, 09 Dec 2017 18:26:33 GMT
x-amzn-remapped-date: Tue, 04 Sep 2018 13:20:43 GMT
x-amzn-remapped-connection: close
x-aero-version-id: d488de00-617e-473f-a1da-e7a393a7116d

<!DOCTYPE html><html><head><meta charset="utf-8"><meta http-equiv="X-UA-Compatible" \
                                                    content="IE=edge">
...
```

A questo punto lo user agent deve procurarsi le credenziali per accedere al sito. I browser di solito la prima volta producono una finestrella di pop-up in cui l'utente inserisce username e password. Al posto della finestrella di pop-up il browser può usare la finestra principale. L'utente inserisce le credenziali che in questo modo vengono messe a disposizione del browser, che non solo le inserisce nella richiesta che sottopone al server, ma in aggiunta a ciò, le salva, in modo da poterle introdurre in ogni successiva richiesta.

La nuova richiesta:

Richiesta

```
GET /protected-standard/index.html HTTP/1.1
Host: auth-demo.aerobaticapp.com
Authorization: Basic YWVyb2JhdGljOmFlcm9iYXRpYw==
```

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 \
                            (KHTML, like Gecko) Chrome/68.0.3440.106 Safari/537.36
Accept: */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9,it;q=0.8
```

Sfortunatamente, essendo una demo e non un vero e proprio sistema, in questo particolare caso la richiesta non va a buon fine. Nel caso generale, comunque a questo punto dovremmo ottenere la pagina richiesta.

Nei prossimi passaggi, il browser inserirà le credenziali automaticamente, senza doverle chiedere ogni volta all'utente.

2.2.3 Conclusioni

L'HTTP fornisce due meccanismi a sostegno del session tracking:

- autenticazione: il codice 401 della status line e i campi per la gestione dell'autorizzazione
- cookies: i campi `Set-Cookie` e `Cookie` per lo scambio dei cookies

L'HTTP fornisce anche un meccanismo che permette al client di impiegare il caching pur assicurando che gli oggetti passati al browser siano aggiornati:

- GET condizionato (conditional GET)

3. HTML

In questa lezione considereremo una breve introduzione sulle caratteristiche più interessanti di HTML, XHTML (che riprenderemo quando tratteremo XML) e di HTML5. Testi di riferimento:

1. HTML Living Standard del WHATWG (Web Hypertext Application Technology Working Group).
2. Tutorial da W3schools.
3. Il concetto di `origin` come definito dal W3C.
4. Per l'HTML5, si faccia riferimento alle specifiche del W3C (`http://www.w3.org/html/wg/drafts/html/CR/`); nel dicembre 2014 è stata anche pubblicata una nota riguardante le differenze tra HTML4 e HTML5: `http://www.w3.org/TR/2014/NOTE-html5-diff-20141209/`.

3.1 Cos'è HTML?

Si tratta di un **linguaggio di etichettatura** (in inglese, markup) per ipertesti: HTML sta infatti per HyperText Markup Language. Ci sono diversi motivi per cui si aggiunge dell'annotazione ad un testo. Nel caso di HTML, la funzione dell'annotazione è quella di dare alcune indicazioni su come il testo andrà formattato. Ad esempio, HTML dà la possibilità di indicare i diversi paragrafi, diversi livelli di titolo, liste numerate e non, e cosí via.

Ecco un semplice esempio di documento HTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Qui va il titolo del documento</title>
</head>
<body>
qui va il contenuto del documento
</body>
</html>
```

La prima riga del documento dichiara quale versione di HTML si adotta. In questo caso si tratta di HTML4: come vedremo, HTML5 introduce dei cambiamenti piuttosto importanti. Il documento HTML vero e proprio è poi composto da un insieme di **elementi**: un elemento è una parte di documento compresa tra l'apertura di un'**etichetta** (ad esempio, `<head>`) e la relativa chiusura (`</head>`), etichette comprese. L'etichetta (o tag) rappresenta il **nome dell'elemento**.

ATTENZIONE

Non bisogna confondere il concetto di **elemento** con quello di **etichetta**. Infatti, l'elemento è una parte di documento, mentre l'etichetta è semplicemente un nome, che in un certo senso caratterizza il tipo dell'elemento (ad esempio, se si tratta di un titolo o di un paragrafo). In un documento possono esserci più elementi con la stessa etichetta. La stessa considerazione ovviamente vale anche per l'XML.

Exercise 3.1 Si consideri una pagina HTML presa dal web e si contino il numero di elementi e il numero di etichette presenti in essa. Naturalmente diverse etichette uguali contano per una. ■

Va tenuto presente che HTML non dà indicazioni precise sull'aspetto grafico, ma solo sulla funzione delle diverse parti di testo. Per definire in dettaglio gli aspetti della rappresentazione grafica, ed eventualmente adattarli ai diversi media su cui si ha il rendering del documento, occorre introdurre dei **fogli di stile** e in particolare un **Cascading Style Sheet** o CSS.

In altre parole,

- HTML descrive la **struttura** del documento;
- CSS descrive invece il **layout** grafico o sonoro (*aural* in inglese) del documento stesso.

Sottolineiamo che il termine *rendering* include sia la resa grafica che quella sonora.

Ovviamente HTML deve prevedere la possibilità di introdurre dei **link**

```
<p>This a link to <a href="peter.html">Peter's page</a>.</p>
<p><a href="../mary.html">Mary's page</a></p>
<a href="friends/sue.html">Sue's page</a>
<a href="../college/friends/john.html">John's page</a>
This is a link to <a href="http://www.w3.org/">W3C</a>.
<a href="/"><img src="logo.gif" alt="home page"></a>
<h2 id="night-spots">Local Night Spots</h2>
<ul>
  ...
  <li><a href="#night-spots">Local Night Spots</a></li>
  ...
</ul>
```

dove "/" indica la radice dell'albero delle directory visibili da web, quindi la home page. Quest'ultimo esempio mostra come HTML preveda delle etichette anche per l'inclusione di immagini, suoni e video.

HTML, però oltre alle indicazioni sulla struttura del documento, prevede anche delle etichette per **interagire** con l'utente. Infatti, l'utente può dare origine a degli **eventi** a cui verranno associate diverse azioni. Esempi di questo tipo di etichette sono quelle per la costruzione di moduli o form per la raccolta dati e di bottoni per la creazione di una richiesta HTTP da inoltrare.

Le form rappresentano uno, **ma non l'unico**, dei modi per passare parametri al server.

```
<FORM action="http://somesite.com/prog/adduser" method="post">
    <P>
    <LABEL for="firstname">First name: </LABEL>
```

```
            <INPUT type="text" id="firstname"><BR>
    <LABEL for="lastname">Last name: </LABEL>
            <INPUT type="text" id="lastname"><BR>
    <LABEL for="email">email: </LABEL>
            <INPUT type="text" id="email"><BR>
    <INPUT type="radio" name="sex" value="Male"> Male<BR>
    <INPUT type="radio" name="sex" value="Female"> Female<BR>
    <INPUT type="submit" value="Send"> <INPUT type="reset">
    </P>
 </FORM>
```

Normalmente la presenza di una form in un documento HTML presentato lato client implica programmazione lato server per implementare un programma che prenda in ingresso i parametri raccolti dalla form e li elabori in modo opportuno. Per impratichirsi con le form senza aver ancora alcuna competenza di programmazione lato server, tuttavia, si consideri il seguente esercizio, che si basa sul motore di information retrieval di google. Si noti che seguendo l'URL `https://www.google.com/search?q=keyword` si ottengono i risultati della ricerca per la parola chiave *keyword*.

Exercise 3.2 Si implementi un documento HTML contenente una form che raccoglie una parola chiave e ci restituisce, attraverso google, i risultati della ricerca di questa parola sul web. ■

3.1.1 Differenza tra gli attributi `name` e `id`

Nell'esempio precedente, l'attributo `name` determina il nome del parametro che viene passato al server. Ci potrebbero essere più parametri con lo stesso `name`. Al contrario il valore di `id` deve essere univoco: ne consegue che al di fuori delle form, conviene sempre usare `id` al posto di `name`, che infatti è deprecato. Come vedremo, in XML esiste un attributo particolare `id`, ma non `name`. In HTML5 `name` si trova solo associato a `form`.

3.2 XML e XHTML

XML sta per e**X**tensible **M**arkup **L**anguage: si tratta dunque di un **linguaggio di markup** come HTML, ma con la caratteristica di essere estendibile, nel senso che le etichette non sono prefissate, ma possono essere scelte dall'utente. Questo perché è stato progettato per **descrivere dati**.

Ecco un semplice esempio di documento XML che riprenderemo nel Capitolo **??**, dedicato appunto all'XML.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<corso>
<crediti>6</crediti>
<anno>4</anno>
<docente>Anna Corazza</docente>
<nome>Tecnologie Web</nome>
</corso>
```

Nello stesso capitolo, introdurremo anche XHTML: per il momento basti dire che XHTML è una variante dell'HTML che segue la sintassi dell'eXtensible Markup Language o XML. Le etichette sono le stesse dell'HTML, ma la sintassi è leggermente diversa. Il fatto che un documento XHTML sia un documento XML, ha il vantaggio di permettere l'applicazione di tutti gli strumenti sviluppati per XML.

3.3 HTML5

1990-1995: siamo agli albori di HTML, che comunque viene elaborato prima al CERN e poi da IETF producendo una serie di versioni e estensioni. In seguito viene creato il W3C che dapprima tenta di estendere HTML nella versione 3.0 e poi lo stabilizza con la 3.2 e la 4.0 nel 1997.

Una caratteristica della storia dell'HTML è sempre stata che specifiche e implementazioni hanno avuto evoluzioni diverse, a volte convergenti, a volte divergenti. La specifica di HTML4 risale al 1997 ed ha introdotto una serie di importanti novità:

- definisce un unico linguaggio HTML che può essere scritto utilizzando sia la sintassi HTML che la sintassi XML;
- definisce dettagliati modelli di processo per favorire la creazione di applicazioni interoperabili;
- migliora il mark-up per i documenti;
- introduce nuove etichette e API per varianti dell'HTML che stanno emergendo, tra cui le applicazioni web.

Nel periodo successivo il W3C decide di impegnarsi nello sviluppo di un'alternativa basata su XML, dando origine a XHTML. Nel frattempo i produttori di browser cercano di mettersi d'accordo su una DOM comune.

Nel 2004, Apple, Mozilla e Opera hanno annunciato l'intenzione di lavorare insieme su un'evoluzione dell'HTML fondando il Web Hypertext Application Technology Working Group (WHATWG), dopo che il W3C aveva deciso di non proseguire su quella linea preferendole l'XHTML.

Nel 2007, il W3C ha costituito un working group per lavorare al progetto assieme al WHATWG e da allora il progetto viene portato avanti insieme.

16 settembre 2014: l'HTML Working Group ha pubblicato una proposta di specifiche per l'HTML5 ed accetta commenti fino al 14 ottobre.

9 dicembre 2014: viene pubblicato un documento che discute le differenze tra le specifiche di HTML4 e HTML5.

3.3.1 Compatibilità con le precedenti versioni

Come vedremo, HTML5 introduce delle modifiche piuttosto importanti, che pongono problemi di compatibilità con le precedenti versioni di HTML. Ovviamente gli user agents devono essere in grado di gestire sia HTML5 che le precedenti versioni. Ad esempio, nel ripensamento che è stato fatto del linguaggio, alcuni elementi sono stati esclusi, ad esempio perché trattati meglio coi fogli di stile CSS che vedremo nel Capitolo 11. Questo significa che mentre gli sviluppatori di applicazioni web, chiamati **autori** nelle specifiche, devono evitare di usare questi elementi, gli **user agents** devono tuttavia essere in grado di gestirli in modo corretto, ovvero in linea col significato che avevano nelle recenti versioni.

In altre parole, i requisiti vengono divisi in due parti:
requisiti per gli autori
requisiti per gli user agent
rendendo così inutile la definizione di alcune features come deprecate.

3.3.2 Motivazioni

Nella progettazione di questa nuova evoluzine di HTML sono stati seguiti i seguenti principi:

1. Lo sviluppo di pagine di contenuti da pubblicare in web ormai non può prescindere da un gruppo di strumenti che si integrano strettamente l'uno con l'altro:
   (a) **HTML** per l'annotazione degli ipertesti; va notato che mentre all'inizio lo scopo dell'HTML era quello di dare una definizione abbastanza precisa della formattazione del testo, nel corso degli anni cresce sempre di più la necessità di separare le indicazioni

di formattazione, da quelle più legate ai contenuti del testo. I vantaggi che si ottengono con questa separazione sono sia di portabilità che di manutenibilità.

(b) **CSS** (Cascading Style Sheets) si occupa più precisamente della formattazione nel rendering dei contenuti.

(c) **DOM** (Document Object Model) dà indicazioni su come il documento HTML viene rappresentato in memoria mediante delle API. Semplificando, si tratta di un albero in cui ogni nodo corrisponde ad un elemento del documento.

(d) **JavaScript**, un linguaggio di scripting che permette di elaborare la DOM per eseguire operazioni lato client.

Occorre quindi introdurre in HTML5 nuove features che siano basate non solo su HTML, ma anche su CSS, DOM e JavaScript.

2. Uno degli scopi importanti che ci si prefigge è quello di ridurre la necessità di plugins esterni, come Flash.
3. Migliorare la gestione degli errori.
4. Laddove possibile, sostituire l'uso di un linguaggio di scripting con un'adeguata annotazione (markup).
5. Indipendenza dal device.
6. Il processo di sviluppo delle specifiche deve avvenire in modo trasparente verso il pubblico.

Si noti che nei documenti di specifiche dell'HTML5, col termine **semantico** ci si riferisce alla descrizione della struttura e non dei contenuti del documento. Viene quindi contrapposto alla presentazione del documento.

Vengono quindi completamente rimosse tutta una serie di etichette tese a descrivere la formattazione del documento, oltre a elementi ormai sostituiti da altri, come ad esempio applet. Per completezza, vengono rimossi:

1. acronym
2. applet
3. basefont
4. big
5. center
6. dir
7. font
8. frame
9. frameset
10. noframes
11. strike
12. tt

Altre etichette, tra cui *i* e *b* che potrebbero essere considerati di formattazione, sono invece stati lasciati, sulla base del fatto che indicano non come il testo va formattato, ma che il testo compreso tra queste etichette va stilisticamente evidenziato dal resto senza tuttavia conferirgli una caratteristica di maggiore importanza.

Sono quindi state inserite delle nuove etichette che, ripetiamo, si riferiscono alla struttura del documeno, tra cui *section*, *article*, *footer*, *progress*, *nav*.

3.3.3 Doppia sintassi

A partire dalla versione 5, un documento HTML può seguire una tra due possibili sintassi:

`text/html` Si tratta della "sintassi HTML", totalmente compatibile con HTML4 e XHTML1, a parte un paio di particolarità piuttosto esoteriche.[1] Ad esempio:

[1] Si tratta di processing instructions (`http://www.w3.org/TR/1999/REC-html401-19991224/appendix/notes.html#h-B.3.6`) e shorthand markup (`http://www.w3.org/TR/1999/REC-html401-19991224/`

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Non appare nel documento</title>
  </head>
  <body>
    <p>Testo inserito nel documento</p>
  </body>
</html>
```

`application/xhtml+xml` **o** `application/xml` La "sintassi XML" è invece compatibile con XHTML1: il documento va definito come un documento XML in cui tutti i documenti vanno messi in un apposito namespace. Studieremo in dettaglio i documenti XML e i namespaces nel Capitolo **??**, ma intanto il documento visto qui sopra si presenta così:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Non appare nel documento</title>
  </head>
  <body>
    <p>Testo inserito nel documento</p>
  </body>
</html>
```

3.3.4 Memorizzazione (storage)

Un'importante caratteristica di HTML5 riguarda la possibilità di memorizzare dati localmente, all'interno del browser, in alternativa ai cookies, sempre come coppia di stringhe nome=valore. Questi dati non vengono inseriti in ogni richiesta HTTP inoltrata al server, ma solo quando risulta necessario farlo. La dimensione dei dati memorizzati può anche essere notevole, senza tuttavia pesare sulle prestazioni del sito. Per una trattazione più completa di questo argomento occorre conoscere un po' di JavaScript: la rimandiamo quindi alla Sezione **??**.

3.4 Precauzioni per salvaguardare la sicurezza nello sviluppo di documenti HTML

L'argomento è complesso e questa non è la sede per una trattazione esauriente. Tuttavia alcuni accenni saranno utili a rendersi conto del tipo di problemi che si possono creare.

3.4.1 Same-origin policy

Gli user agent eseguono azioni suggerite loro dai contenuti che scaricano dai diversi siti, la maggior parte dei quali saranno stati creati da sviluppatori in buona fede. Tuttavia, visto che alcuni siti possono provvedere dei contenuti malevoli, occorre che lo user agent applichi delle politiche per salvaguardare le informazioni di cui è responsabile, sia proprie che altrui.

Come vedremo nelle prossime lezioni, inoltre, ad uno user agent quasi sempre viene richiesto di mandare in esecuzione script prodotti dal sito che sta visitando. Ovviamente occorre cautelarsi

`appendix/notes.html#h-B.3.7`), peralatro utilizzate poco o nulla per lo scarso supporto offerto dalla maggior parte degli user agents.

contro eventuali azioni malevole che questi script possono prevedere. In particolare si potrebbe correre il rischio di dare accesso ad informazioni riservate fornite da altri siti.

Tipicamente, per affrontare questo problema, gli user agent usano il concetto di **origine** per implementare la cosiddetta **same-origin policy**. Il principio seguito è che i contenuti possono interagire senza restrizione con tutti i contenuti provenienti dalla stessa origine, mentre questo non può avvenire tra contenuti di origine diversa.

La same-origin policy basa la fiducia accordata al sito sul suo URI. Per esempio, in un documento HTML possiamo trovare il seguente elemento vuoto:

```
<script src="https://example.com/library.js"></script>
```

con cui una libreria JavaScript viene scaricata da quell'URI e le vengono assegnati gli stessi privilegi di esecuzione del documento: in altre parole, è come se il documento dichiarasse che si fida delle informazioni scaricate da quell'URI.

Oltre a scaricare librerie da URI, gli user agent mandano anche informazioni a partners remoti designati attraverso un URI, ad esempio attraverso le form:

```
<form method="POST" action="https://example.com/login">
 ... <input type="password"> ...
</form>
```

In questo modo il documento chiede all'utente di mandare le proprie informazioni all'URI dato e l'utente potrebbe voler proteggere tali informazioni. Anche in questo caso è come se il documento dichiarasse la propria fiducia nelle garanzie di confidenzialità garantite da quell'URI.

Due URI sono considerate appartenere alla stessa origine se hanno uguali **schema, host e porta**. Ad esempio, le seguenti risorse hanno tutte la stessa origine:

```
http://example.com/
http://example.com:80/
http://example.com/path/file
```

visto che la porta 80 è quella di default per il protocollo HTTP. Le seguenti, invece, hanno origini diverse:

```
http://example.com/
http://example.com:8080/
http://www.example.com/
https://example.com:80/
https://example.com/
http://example.org/
http://ietf.org/
```

3.4.2 Mancata validazione dell'input dell'utente; cross-site scripting (XSS); SQL injection

Quando si introducono delle form nel documento HTML che viene servito al client, occorre fare molta attenzione al testo che viene effettivamente inserito e all'uso che se ne fa. Le conseguenze di trascuratezza in questa fase possono comportare non solo comportamenti inattesi da parte del sito, quali ad esempio età negative, ma anche danni al sito stesso o al sito con cui si sta interagendo. I danni possono anche essere molto seri, quali la cancellazione di tutti i dati sul server.

Occorre quindi prevedere filtri per la validazione dei dati in ingresso, ma è importantissimo che questi filtri siano basati sull'enumerazione dei casi favorevoli (in inglese, **whitelist-based**), bloccando tutto ciò che non sia esplicitamente previsto. Se al contrario ci si limita a bloccare solo gli ingressi che si ritengono pericolosi, con un approccio di tipo blacklist-based, si rischia non

tanto di dimenticarsi dei casi, magari rari, ma non impossibili, ma soprattutto che in futuro tali casi sfavorevoli e imprevedibili al momento dell'implementazione del sistema si possano presentare.

A titolo di esempio, si supponga di avere una risorsa che mostra all'utente la **query string** nel suo URL, ovvero la parte dell'URL che segue il punto di domanda e contiene i parametri, e ridirige l'utente ad uno script CGI che mostra quel messaggio:

```
Say Hello
Say Welcome
Say Kittens
```

Anche un caso tutto sommato semplice come questo può risultare pericoloso nel caso in cui il messaggio venga presentato all'utente cosí come viene ricevuto nella query string, senza introdurre caratteri di escape. Si supponga infatti di creare una query string come questa:
`http://example.com/message.cgi?say=%3Cscript%3Ealert%28%27Oh%20 ...`
`... no%21%27%29%3C/script%3E`
e di riuscire a convincere l'utente a cliccare sul link ottenuto: in questo modo si manda in esecuzione uno script scelto dall'attaccante che può portare a termine azioni ostili, nei limiti delle azioni offerte dal sito. Ma in un sito di e-commerce, si potrebbe far compiere all'utente un numero di acquisti non voluti. Un attacco di questo tipo viene chiamato **cross-site scripting attack**.

Questo ovviamente non è l'unico modo in cui si può indurre l'utente a mandare involontariamente in esecuzione del codice. Ecco alcune cose da tener presenti quando si costruiscono dei filtri di validazione basati su whitelist per HTML:

- Anche con elementi che possono sembrare innocui, quali quelli di tipo `img`, occorre ricordarsi di validare, sempre sulla base di una whitelist, anche gli attributi, per evitare che un eventuale attaccante possa usare l'attributo `onload` per mandare in esecuzione uno script di sua scelta.
- Se si decide di permettere che la costruzione di URL, da usarsi per esempio in link, sia basata su input, anche lo schema dell'URL deve essere basato su whitelist, perché tali schemi possono costituire un'importante sorgente di abusi. L'esempio più evidente riguarda `javascript:`, ma non è l'unico.
- Se si permette l'inserimento di un elemento di tipo `base`, tutti gli elementi `script` di quella pagina possono venir piratati, e in particolare gli URL per la sottomissione dei form possono venir rediretti a siti ostili.

3.4.3 Cross-site request forgery (CSRF)

In un sito in cui si permette all'utente di usare la sottomissione di form per portare a termine delle azioni a nome dell'utente, come inviare dei messaggi, compiere degli acquisti, richiedere documenti, è della massima importanza verificare che l'utente abbia fatto la richiesta intenzionalmente e non perché tratto in inganno. L'origine di questo problema risiede nel fatto che HTML permette all'utente di sottomettere le form anche a origini diverse da quelle dell'applicazione.

La prevenzione di questo tipo di attacchi richiede di popolare le form con token nascosti caratterizzanti l'utente oppure di controllare il campo `<Origin` nell'intestazione di tutte le richieste HTTP.

3.4.4 Clickjacking

Supponiamo di avere una pagina web in cui siano previste delle azioni che l'utente può non voler intraprendere: in questo caso bisogna garantire che non sia possibile che l'utente le scelga senza averne coscienza. Questo potrebbe per esempio succedere se il sito della vittima viene messo in un minuscolo `iframe` e poi si convince l'utente a cliccarvi, per esempio traendolo in inganno con un gioco interattivo di velocità. Mentre l'utente gioca, infatti, il sito ostile può spostare velocemente l'iframe sotto il mouse, in modo da indurre l'utente a cliccare sul sito e quindi a richiedere, senza

volerlo, l'azione considerata. Si possono evitare problemi di questo tipo permettendo l'attivazione dell'interfaccia critica solo nel caso in cui si sia certi che la pagina non sia stata inclusa in un frame. Il controllo può ad esempio basarsi sul confronto tra l'oggetto `window` e il valore dell'attributo `top`.

4. Common Gateway Interface

Programmazione lato server: Common Gateway Interface (CGI) schema di funzionamento; vantaggi e svantaggi; passaggio dei parametri; esempi.

4.1 CGI – Common Gateway Interface

4.1.1 Riferimenti disponibili in rete

Una fonte importante di approfondimento (facoltativo) per gli argomenti trattati in questa lezione è il tutorial CGI sul sito del server web Apache[1].

Concetti base di informatica che ovviamente già sapete e che sono utili alla comprensione della lezione:

- differenza tra programma e processo;
- variabili d'ambiente

4.2 Il meccanismo del CGI

Si tratta di un meccanismo per far eseguire al server Web dei programmi (o degli script). Tra le funzionalità del server web ci sarà quella di permettere di impostare la configurazione in modo che i programmi CGI si possano eseguire. Poiché il meccanismo del CGI permette all'utente di richiedere l'esecuzione di programmi sul server, per tutelare la sicurezza del server stesso occorre assicurarsi che tali programmi non abbiano possibilità di nuocere. Occorre quindi che chi imposta il server permettendo l'esecuzione di programmi CGI sappia cosa sta facendo. È questo il motivo per cui tipicamente le impostazioni di default dei server web non permettono l'esecuzione dei programmi CGI.

Il meccanismo del CGI viene usato molto per riutilizzare sistemi legacy: essi vengono "avvolti" in un wrapper e tramite CGI resi accessibili su Web. Un'altra applicazione tipica ha lo scopo di costruire sistemi di accesso a basi di dati, anche se ormai altri sistemi hanno un tale supporto da essere molto più competitivi. In tali sistemi con un form HTML vengono raccolti i parametri

[1] `http://httpd.apache.org/docs/1.3/howto/cgi.html`

con cui l'applicazione CGI accede alla base di dati (ad esempio con una SELECT) e prepara la visualizzazione dei risultati (ad esempio, in una tabella).

L'accesso ad una base di dati è un esempio tipico di applicazione web dinamica e il suo funzionamento si può scomporre nelle seguenti fasi:

1. il **browser** raccoglie i parametri, ad esempio usando un form HTML, ma non solo e li introduce nella **richiesta HTTP** che passa al server
2. il **server** usa questi parametri per costruire la query di accesso al **database**
3. il **server** organizza i risultati della query in una tabella che passa nella **risposta HTTP** al **browser**

Pur essendo questi due casi tipici, va sottolineato che il CGI, pur essendo un meccanismo poco sofisticato, è molto flessibile e viene quindi utilizzato in molti contesti diversi.

Esempio:

Se, in una pagina HTML il form

```
<form action="http://example.com/cgi-bin/search" method="get">
    Find <input type="text" name="q">
    <input type="submit" value="Go">
</form>
```

introduco la chiave di ricerca `pippo`, il browser prepara una richiesta HTTP simile alla seguente[2]

```
http://example.com/cgi-bin/search?q=pippo

GET http://example.com/cgi-bin/search?q=pippo HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; /
        rv:1.7.13) Gecko/20060717 Debian/1.7.13-0.2ubuntu1
Accept: text/xml,application/xml,application/xhtml+xml,/
        text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Proxy-Authorization: Basic YW5uYS5jb3JhenphOlN2MmltZHZoaQ==
```

e il server potrebbe quindi inoltrare la seguente risposta:

```
HTTP/1.x 404 Not Found
Date: Wed, 03 Oct 2007 07:14:01 GMT
Server: Apache/2.2.3 (CentOS)
Content-Length: 288
Content-Type: text/html; charset=iso-8859-1
X-Cache: MISS from www-cache.unina.it
Proxy-Connection: keep-alive
```

Si consideri infatti il caso in cui se non c'è un programma CGI in grado di gestire la richiesta: una risposta deve necessariamente essere prodotta e questo non può che farlo il server web. Se invece le cose vanno bene e il programma CGI viene messo in esecuzione, **il server web produce solo la status line, mentre il programma CGI produce il resto della risposta**.

Ripeto lo stesso esercizio lasciando tutto uguale eccetto che il metodo HTTP sarà il post invece che il get:

[2] Si noti che il carattere / è stato introdotto per spezzare le righe troppo lunghe.

```
<form action="http://example.com/cgi-bin/search" method="post">
    Find <input type="text" name="q">
    <input type="submit" value="Go">
</form>
```

Se provo ad introdurre la chiave di ricerca `pluto`, il browser prepara una richiesta HTTP a `http://example.com/cgi-bin/search`

```
POST http://example.com/cgi-bin/search HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.7.13) /
                    Gecko/20060717 Debian/1.7.13-0.2ubuntu1
Accept: text/xml,application/xml,application/xhtml+xml,text/html;/
                    q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Proxy-Authorization: Basic YW5uYS5jb3JhenphOlN2MmltZHZoaQ==
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

q=pluto
```

Il CGI è il risultato di un accordo tra i produttori di server HTTP su come integrare tali scripts e programmi gateway. Il W3C offre un'introduzione a questa metodologia. Esempi con Apache. Un sistema per riciclare vecchi sistemi informativi è quello di metterli sotto un server HTTP attraverso cui si può ad esempio dare accesso ad un corpus di documenti già esistente o ad un'applicazione basata su una base di dati. L'idea generale è che quando lo user agent manda al server una richiesta HTTP per una risorsa relativa ad un programma sul CGI, il corrispondente programma viene mandato in esecuzione sul server e i risultati prodotti vengono ripassati allo user agent mediante la risposta HTTP.

In questo semplice **esempio**, lo script *date.cgi* è scritto in un linguaggio di shell, con l'opzione -f per cui la shell ignora il file /.tcshrc, per ottenere la massima efficienza:

```
#!/bin/csh -f

echo "Content-type: text/html"
echo ""

echo "<HTML>"
echo "la data di oggi: "
date
echo "</HTML>"
```

Lo script produce sullo standard output la dichiarazione

```
Content-type: text/html
```

seguita da **una riga vuota** e dalla seguente pagina HTML:

```
<HTML>
la data di oggi: <data>
</HTML>
```

con data e ora al posto della parola <data>.

Occorre impostare il server in modo che possa riconoscere quali richeste vanno gestite con CGI. Gli script e programmi vengono posti in una specifica directory (spesso, *cgi-bin*) dando indicazione al server che tali programmi sono da eseguirsi sotto CGI, ad esempio, sotto Apache, nel file .htaccess:

```
Options +ExecCGIAddHandler cgi-script cgi pl
```

4.2.1 Input al programma CGI

La modalità con cui i parametri vengono passati al programma dipende dal tipo di metodo utilizzato. In generale, vengono utilizzati lo **stdin** o le **variabili d'ambiente** (a seconda del tipo di richiesta:

1. richiesta HTTP di tipo `GET`: per il passaggio dei parametri viene utilizzata la variabile d'ambiente `QUERY_STRING` in cui viene inserito tutto ciò che segue il ? nell'URL; ovviamente la lunghezza è limitata;
2. richiesta HTTP di tipo `POST`: tutto quello che si trova nel `body` della richiesta HTTP viene passato al processo CGI sullo `stdin`; non viene terminato con un EOF, e quindi occorre conoscerne la lunghezza: a tale scopo viene usata la variabile d'ambiente `CONTENT_LENGTH`.

4.2.2 Output del programma CGI

La risposta HTTP da inoltrare allo user agent viene preparata utilizzando lo `stdout` del processo CGI, a cui il server premette **esclusivamente** la riga di stato. Ricapitolando:

- lo user agent invia una richiesta HTTP al server;
- il server web la riconosce come programma Common Gateway Interface, e quindi
- il server web manda in esecuzione il programma (passando i parametri come spiegato sopra);
- se l'esecuzione del programma produce dei risultati (ma il caso contrario è solo teorico!), questi vanno organizzati sullo **stdout**;
- il server Web prepara la risposta HTTP da inviare al client includendo:
  - status line, ad esempio, `HTTP/1.1 200 OK`
  - output del programma (che deve quindi contenere anche il resto dell'intestazione – header – HTTP)

È istruttivo il seguente script che stampa e quindi restituisce al client le variabili d'ambiente:

```
#!/bin/sh
# cgi-test.sh

echo "Content-type: text/html"
echo
echo "<HTML>"
echo "<HEAD>"
echo "<TITLE>Test CGI</TITLE>"
echo "</HEAD>"
echo "<BODY>\n";
echo "<H1>Test CGI</H1>"
echo "<PRE>"
echo "N. argomenti = $#"
echo "Argomenti = $*"
echo
echo "SERVER_SOFTWARE = $SERVER_SOFTWARE"
echo "SERVER_NAME = $SERVER_NAME"
echo "GATEWAY_INTERFACE = $GATEWAY_INTERFACE"
```

```
echo "SERVER_PROTOCOL = $SERVER_PROTOCOL"
echo "SERVER_PORT = $SERVER_PORT"
echo "SERVER_ADMIN = $SERVER_ADMIN"
echo "REQUEST_METHOD = $REQUEST_METHOD"
echo "HTTP_ACCEPT = $HTTP_ACCEPT"
echo "HTTP_USER_AGENT = $HTTP_USER_AGENT"
echo "HTTP_CONNECTION = $HTTP_CONNECTION"
echo "PATH_INFO = $PATH_INFO"
echo "PATH_TRANSLATED = $PATH_TRANSLATED"
echo "SCRIPT_NAME = $SCRIPT_NAME"
echo "QUERY_STRING = $QUERY_STRING"
echo "REMOTE_HOST = $REMOTE_HOST"
echo "REMOTE_ADDR = $REMOTE_ADDR"
echo "REMOTE_USER = $REMOTE_USER"
echo "AUTH_TYPE = $AUTH_TYPE"
echo "CONTENT_TYPE = $CONTENT_TYPE"
echo "CONTENT_LENGTH = $CONTENT_LENGTH"
echo
echo "Standard input:"
cat -
echo "</PRE>"
echo "</BODY>"
echo "</HTML>"
```

Richiamando senza parametri, si ottengono i corrispondenti valori del server.

Se viene prodotto un file (documento, audio, video, ...) da passare al browser, occorre un'intestazione che dà il tipo MIME del file:`Content-type`, che può essere `text/html` o tt text/plain o altri formati.

Di seguito vengono riportate le variabili di ambiente più importanti nel sistema di comunicazione tra server Web e processo CGI.

4.3 Informazioni sul server

- `SERVER_SOFTWARE` Il nome e la versione del software utilizzato come server.
- `SERVER_NAME` Il nome del server.
- `SERVER_PROTOCOL` Il nome e la versione del protocollo utilizzato dal server.
- `SERVER_PORT` Il numero della porta di comunicazione utilizzata dal server.
- `GATEWAY_INTERFACE` Letteralmente, è l'interfaccia gateway, ovvero la versione del protocollo CGI utilizzato dal server.
- `PATH_INFO` Quando l'URI contiene l'indicazione di un percorso aggiuntivo, questa variabile riceve quel percorso.
- `PATH_TRANSLATED` assieme a <samp>PATH_INFO</samp> indica il percorso reale nel file system che ospita il server.
- `SCRIPT_NAME` La parte dell'URI che identifica il percorso del programma utilizzato come gateway.

4.3.1 Informazioni sulla connessione client-server

- `REQUEST_METHOD` Il metodo della richiesta (`GET`, `POST`).
- `REMOTE_HOST` Il nome del client. Se il nome non è disponibile, si deve fare uso della variabile `REMOTE_ADDR` che contiene l'indirizzo IP.

- `REMOTE_ADDR` Indirizzo IP del cliente.
- `AUTH_TYPE` Contiene l'eventuale metodo di autenticazione.
- `REMOTE_USER` Il nome dell'utente se si utilizza l'autenticazione.

4.3.2 Informazioni passate dal client al server

- `QUERY_STRING` Contiene la stringa di richiesta se si utilizza il metodo `GET`.
- `CONTENT_LENGTH` Contiene la dimensione in byte dei dati ricevuti dal client. Questa informazione è disponibile solo se si utilizza il metodo `POST`.
- `CONTENT_TYPE` Contiene la definizione del tipo di codifica dei dati ricevuti dal cliente e riguarda solo il metodo `POST`. La codifica più comune è `application/x-www-form-urlencoded` e significa che i dati sono stati codificati secondo lo standard utilizzato per il metodo `GET`: gli spazi sono convertiti in + e tutti i simboli speciali in un formato esadecimale.

4.4 Informazioni addizionali dal client

Il contenuto di questi campi viene tradotto in altrettante variabili di ambiente il cui nome inizia per `HTTP_` seguito dal nome del campo stesso. In particolare, i caratteri minuscoli sono convertiti in maiuscoli e i trattini sono sostituiti dal simbolo di sottolineatura. Seguono alcuni esempi.

- `HTTP_ACCEPT` equivale al campo `Accept` nell'intestazione della richiestaHTTP.
- `HTTP_USER_AGENT` equivale al campo `User-Agent` nell'intestazione della richiesta HTTP.

Un programma CGI è un eseguibile che chiunque può mandare in esecuzione: ci sono ovvi problemi di sicurezza di cui discuteremo alla fine della lezione. È però ovvio fin da ora che un processo CGI deve sottostare a delle restrizioni.

La prima restrizione è che i programmi CGI devono risiedere in una directory particolare, in modo che il server Web sappia che sono programmi da mandare in esecuzione e non semplici file da visualizzare. Questa directory si trova usualmente sotto il diretto controllo del webmaster, che può dunque decidere se un utente è abbastanza affidabile da creare programmi CGI. Inoltre è possibile anche limitare il tipo di estensione dell'eseguibile.

Un programma CGI può venir scritto in qualsiasi linguaggio di programmazione, **tra cui**:

- C/C++
- Fortran
- PERL
- TCL
- una qualsiasi shell Unix
- Visual Basic

La scelta dipende da quali linguaggi sono disponibili sul server: non ci sono vincoli imposti dal client. Ovviamente, i programmi scritti in linguaggi compilati, quali ad esempio il C e il Fortran, dovranno venire compilati prima di poter essere eseguiti.

Ogni volta che un client richiede, attraverso una richesta HTTP, la risorsa corrispondente al programma CGI, il server lo manda in esecuzione. Se arrivano più richieste per la stessa risorsa, il programma viene mandato in esecuzione più volte, ottenendo cosí più processi che vengono eseguiti contemporaneamente, mantenendo tuttavia l'informazione che permette di accoppiare il processo con la relativa richiesta HTTP, in modo da poter preparare la risposta.

Come discusso sopra, non essendo possibile utilizzare la sintassi usuale per passare opzioni o argomenti da linea di comando, quali

```
command\% mioprogramma -opz argomento1 argomento2
```

il meccanismo del CGI usa variabili di ambiente per passare i parametri al programma, e lo `stdin` per passare il contenuto del body della richiesta. La più importante è tra le variabili d'ambiente utilizzate dal CGI è `QUERY_STRING`, che viene definita come qualsiasi cosa segua il

primo ? nell'URL. Un'ulteriore possibilità è quella di introdurre manualmente gli argomenti in un link del documento HTML:

```
<a href="http://example.com/cgi-bin/search?q=pippo">Cerca</a>
```

Naturalmente dire che tutto quello che sta nell'URI dopo il primo punto di domanda viene interpretato come parametri da passare al server è un po' semplicistico. In realtà la stringa deve subire delle trasformazioni, ad esempio, gli spazi vengono sostituiti da dei segni '+', e i caratteri speciali vengono codificati in esadecimale.

Di conseguenza, il CGI prima di ripassare la stringa al programma, deve ritrasformarla: si parla al proposito di parsing. Ad esempio, se viene passata la stringa "nome+cognome" il programma la riceverà come due argomenti distinti.

Si ricordi che l'output prodotto dal programma viene inserito nella **risposta HTTP** da rispedire al client: occorre quindi che sia opportunamente formattato.

I programmi CGI possono produrre documenti di tipi diversi, anche multimediali: documenti HTML, documenti di puro testo, immagini, audio clip, riferimenti ad altri documenti. Perché lo user agent sappia come trattare il documento che gli viene passato, occorre dichiarare nell'intestazione della risposta HTTP il tipo del documento e se si tratta di un riferimento. Il meccanismo CGI assegna questa responsabilità allo sviluppatore del programma CGI.

L'intestazione della risposta HTTP è in puro ASCII e consiste di linee separate da caratteri di fine riga (linefeeds and/or carriage returns) seguite da una linea vuota. Il documento invece segue la sintassi prevista per il formato corrispondente al documento stesso.

Un documento col suo tipo MIME:

```
Content-type: text/html

<HTML><HEAD>
<TITLE>output of HTML from CGI script</TITLE>
</HEAD><BODY>
<H1>Sample output</H1>
What do you think of <STRONG>this?</STRONG>
</BODY></HTML>
```

È anche possibile che l'output prodotto dal programma CGI sia un riferimento ad un altro documento; in questo caso, è possibile dare indicazioni allo user agent perché richieda direttamente l'oggetto indicato, usando l'intestazione (tipica dell'**HTTP**):

```
Location: {\em nuovo URL}
```

```
#!/bin/csh -f

echo "Content-type: text/html"
echo "Location: http://www.unina.it"
echo ""
```

(ricordarsi il secondo a capo ...)

Dal sito w3c:

The Location response-header field is used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource. For 201 (Created) responses, the Location is that of the new resource which was created by the request. For 3xx responses, the location SHOULD indicate the server's preferred URI for automatic redirection to the resource. The field value consists of a single absolute URI.

4.4.1 Considerazioni riguardo alla sicurezza

Ogni volta che un programma interagisce con un client attraverso la rete, occorre valutare il rischio che il cliente possa ottenere controlli per cui non è autorizzato attraverso un attacco a quel programma.

Consideriamo alcuni accorgimenti proposti dall'NCSA Software Development Group e suggeriti dal W3C:[3]

- Evitare l'istruzione *eval*:
  tale istruzione è fornita ad esempio da Perl, o dalla Bourne shell e permette di costruire una stringa e poi eseguirla come fosse un comando. Esempio:

```
eval ‘echo $QUERY\_STRING | awk ’BEGIN{RS="""} {printf "QS\_\%s\n",$1}’ ‘

 {\em $QUERY\_STRING = ls"env}, esegue {\em eval
’QS\_ls\nQS\_env’}, che provoca solo errori, visto che i due comandi
non hanno significato. Se invece {\em $QUERY\_STRING = ;ls";env},
esegue {\em eval ’QS\_;ls\nQS\_;env’}, che provoca due errori, ma
esegue anche i comandi {\em ls}, {\em env}, o qualsiasi altro sia
stato passato con questo meccanismo, con le permission del
proprietario dello script.
```

RS: record separator variable: normally, AWK reads one line at a time, and breaks up the line into fields. You can set the "RS" variable to change AWK's definition of a "line." If you set it to an empty string, then AWK will read the entire file into memory.

- Non fidarsi a lasciar far nulla al client: Un client che sia in grado e voglia comportarsi correttamente provvederà a rendere inoffensivi con opportuni escape tutti i caratteri della query string che possono avere significati speciali per la Bourne shell, evitando cosí di creare problemi. Un client malizioso, invece, userà tutti questi mezzi per cercare di mandare in confusione lo script e ottenere accesso a informazioni e permission senza autorizzazione.
- Prestare attenzione a *popen* e *system*:
  - La *popen()* prende come parametri in ingresso una stringa corrispondente ad un comando da eseguire in pipe, e una corrispondente al tipo di pipe, se in lettura o in scrittura. La *popen()* quindi apre un processo attraverso la creazione di una pipe, un'operazione di fork, e l'invocazione della shell.
  - La *system()*, invece, prende in ingresso una stringa corrispondente ad un comando, che esegue chiamando */bin/sh -c*; una volta che il comando è stato completato, la system ritorna.

  Con entrambe queste funzioni, nel caso vengano chiamate con una linea di comando che viene costruita a partire dalla query string inviata dal client, occorre assicurarsi con un prefiltraggio che tutti i caratteri che possono avere significati particolari per la Bourne shell.
- Chiudere tutti gli include sul server. Gli include lato server possono essere usati da client alla ricerca di script in grado di riprodurre direttamente quello che viene loro passato.

Exercise 4.1 Si cerchi un numero telefonico da una form dato un file con la lista di nomi e numeri. Con post e con get ■

[3] `http://hoohoo.ncsa.uiuc.edu/cgi/security.html`

5. Java

Il capitolo prevede un'introduzione a Java che si focalizza soprattutto sulle caratteristiche che lo rendono adatto alle applicazioni Web: portabilità e sicurezza. Di seguito verranno introdotte le applet, esempio di programmazione lato client. Per lo studio si può utilizzare [Eck02].

5.1 Java per applicazioni Web

Java è stato progettato fin dall'inizio come un linguaggio con caratteristiche di sicurezza e buona programmazione che lo rendono particolarmente adatto alle applicazioni Web. Si tratta di un linguaggio **orientato agli oggetti**: questo lo rende di uso relativamente semplice, anche se non è un linguaggio di scripting. Inoltre ha una gestione automatica della memoria (garbage collector) che semplifica le corrispondenti operazioni evitando l'introduzione di errori, in particolare quelli legati alla deallocazione della memoria.

Si consideri un semplice programma di HelloWorld:

```
class HelloWorld {
    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
```

Si tratta di un linguaggio fortemente tipato, con tutti i controlli corrispondenti. Non ammette costrutti insicuri, quali un accesso a vettore senza controllo sugli indici, perché questo tipo di costrutti possono dare origine a comportamenti del programma non specificati e non predicibili.

Si dice che è indipendente dalla piattaforma, nel senso che viene compilato in un bytecode eseguibile da una **Java Virtual Machine**. Java risulta adeguato alle applicazioni Web perché ha determinate caratteristiche di **sicurezza**: la maggior parte delle azioni di attacco ad un sistema non possono essere descritte come un programma Java.

I programmi Java non possono richiamare funzioni globali, né ottenere l'accesso a risorse del sistema in modo arbitrario: la macchina virtuale Java è in grado di esercitare un grado di controllo impossibile ad altri sistemi.

Java è più robusto, nel senso che offre un miglior controllo degli errori. La maggior parte degli errori sono legati a due aspetti che vanno affrontati nello sviluppo di ogni sistema non banale:

- **gestione della memoria**: nei linguaggi tradizionali occorre tener traccia di tutta la memoria allocata in modo da poterla liberare; non è infrequente dimenticare di liberare della memoria o, peggio, liberare della memoria in uso da altre parti del programma;
- **condizioni eccezionali**: quali divisioni per 0 o errori di lettura dovuti alla mancanza del file di ingresso, che richiedono controlli noiosi che appesantiscono il codice.

La sintassi di Java è molto rigida per quel che riguarda la gestione dei tipi e le dichiarazioni: questo dà la possibilità di rilevare molti errori già durante la compilazione. Si noti che durante la compilazione gli unici **warning** riguardano metodi deprecati (o superati): tutto il resto è considerato errore.

Un buon supporto al **multi-thread** può essere utile nell'implementazione di interfacce sofisticate (anche sul client senza bisogno di fare conto sul sistema operativo) e nella gestione di richieste concorrenti.

Il programma Java viene compilato in un **bytecode** che viene eseguito dalla macchina virtuale (che funge da interprete). In questo modo si ottiene un buon compromesso tra linguaggi compilati e linguaggi interpretati:

- è più **portabile** di un linguaggio compilato;
- è più **efficiente** di un linguaggio interpretato;
- ha caratteristiche che un linguaggio interpretato di solito non ha (ad esempio è fortemente tipato).

Ne deriva un linguaggio che pur di uso meno diretto di un linguaggio di scripting, risulta molto più facile di altri linguaggi di programmazione, quali C e C++.

Va ricordata anche l'ampia disponibilità di librerie.

Java, a partire dalla versione 2, offre un buon supporto alla programmazione di applicazioni di rete: in particolare con le classi e le interfacce del pacchetto *java.net* per i socket, dei 5 pacchetti *java.rmi* per la Remote Method Invocation.

La Java Virtual Machine è una macchina astratta e non vengono fatte ipotesi sulla particolare tecnologia in cui viene implementata né sulla piattaforma ospite. Non sa nulla del linguaggio Java, ma solo del formato *class* (bytecode). Un file *class* contiene istruzioni per la Java Virtual Machine, una tabella di simboli e altre informazioni ausiliarie. Oltre che interpretati dalla Java Virtual Machine, i bytecode possono anche venir compilati in file eseguibili dalle diverse macchine.

Una *piattaforma* Java è composta da:

- linguaggio Java;
- Java Virtual Machine;
- Application Programming Interfaces (API libraries).

Comunque l'aspetto più importante di Java per quel che riguarda il suo uso in applicazioni Web è la **sicurezza**. Essa dipende da:

- il progetto del linguaggio come sicuro e di facile uso;
- verifica del bytecode prima dell'esecuzione;
- durante l'esecuzione:
  - le classi vengono caricate e ne viene fatto il linking;
  - opzionale: generazione del codice macchina e ottimizzazione dinamica;
  - esecuzione del programma.

  durante questo processo, il class loader definisce un namespace locale, in modo che codice non sicuro non possa interferire con l'esecuzione di altri programmi Java.
- la mediazione da parte della Java Virtual Machine mediante una classe **security manager** per l'accesso alle risorse di sistema.

Alcune delle caratteristiche rendono inerentemente sicuro il linguaggio, rendendo impossibili certi tipi di azioni: ad esempio, l'assenza di aritmetica dei puntatori. Altre possono essere sfruttate per rendere sicure le applicazioni.

È sempre importante trovare il giusto compromesso tra:

- sicurezza;
- usabilità;
- prestazioni.

5.1.1 Caratteristiche di sicurezza a livello del linguaggio

Nel progetto del linguaggio sono state tolte le caratteristiche che risultavano pericolose sia perché sorgente di errori che perché abusabili. In Java non è possibile accedere alla memoria in modo non controllato.

Una delle caratteristiche di altri linguaggi (ad esempio, C e C++) che rende impossibili questi controlli è l'**aritmetica dei puntatori**, che in Java è invece esclusa.

In C è possibile uscire dai limiti sui valori degli indici degli array (provocando effetti indefiniti), in Java no. Errori nell'accesso alla memoria provocano errori molto difficili da identificare, come ben sanno i programmatori C/C++.

Infatti gli indirizzi di memoria sono per loro natura dipendenti dalla macchina, e quindi questi accessi danno dei risultati non definiti. Accessi non protetti alla memoria rendono possibili alcuni degli attacchi più frequenti alla sicurezza delle applicazioni. In un linguaggio che vuol essere sicuro, tutte le caratteristiche che rendono possibili accessi non protetti alla memoria devono essere eliminate, e tra queste l'aritmetica dei puntatori.

Infatti, se ai programmi è permessa ogni manipolazione degli indirizzi attraverso puntatori senza limitazioni specifiche imposte dal tipo, non può essere applicato nessun concetto di **dato privato**. Risulta quindi impossibile imporre un qualsiasi modello di integrità o sicurezza dei dati. Inoltre, permettere l'accesso non controllato allo stack può addirittura permettere manipolazioni del flusso di controllo del programma.

Si tenga presente che praticamente tutti i virus mai scritti si avvantaggiano in un modo o nell'altro della capacità del programma di fare accesso e modificare la RAM usando i puntatori. I controlli sull'accesso alla memoria vanno implementati sia a livello di linguaggio, non permettendo quelle caratteristiche che li renderebbero possibili, che durante l'esecuzione, perché non tutti i controlli necessari possono essere implementati staticamente.

Ovviamente vietare l'aritmentica dei puntatori non equivale a non supportare una qualche forma di puntatori, che in Java sono chiamati **references** e pienamente supportati (e usati per implementare liste concatenate, gli alberi binari e non, etc.).

Oltre ad abolire l'aritmetica dei puntatori, le specifiche di Java a livello di linguaggio definiscono chiaramente il comportamento di **variabili non inizializzate**.

Le variabili:

- sullo **heap** vengono automaticamente inizializzate; ne consegue che le variabili (gli attributi) di classi e istanze non possono avere valori non definiti;
- Stack
- sullo **stack** non vengono automaticamente inizializzate e quindi tale operazione deve avvenire in modo esplicito prima dell'uso pena un errore di compilazione (questo vale anche per le variabili di classi e istanze dichiarate *final*).

Un altro aspetto di Java che contribuisce alla robustezza e quindi alla *safety* del codice è la **garbage collection** automatica. Per **garbage collection** si intende la capacità della macchina virtuale di liberare automaticamente la memoria che non viene più indirizzata. Ovviamente la garbage collection riguarda solo la heap, dove viene allocata tutta la memoria richiesta dinamicamente.

Infine, tra gli aspetti che rendono più difficili gli errori annidati nel codice e quindi più affidabile il codice Java, va riportato il controllo dei tipi forte fatto durante la compilazione, che esclude operazioni di *cast* non legali.

Questo controllo evita che un blocco di memoria venga interpretato in modo diverso da quello che è, col rischio di dare accesso palesemente illegale a zone di memoria.

5.1.2 Caratteristiche di sicurezza a livello di macchina virtuale

La macchina virtuale Java usa una **sandbox** per mandare in esecuzione programmi Java:

- per verificare tutte le classi che entrano;
- per operare numerosi controlli a runtime che evitano che i programmi Java compiano azioni non valide;
- per alzare una barriera attorno all'ambiente di runtime e controllare tutti gli accessi all'esterno della sandbox.

Non tutti i controlli possono essere fatti staticamente (durante la compilazione). Si pensi ad esempio al controllo degli indici in un vettore. Inoltre, in alcuni casi il compilatore può non seguire le specifiche, il codice può non essere stato testato a sufficienza oppure le classi possono essere semplicemente scaricate da Internet: in tutti questi casi la macchina virtuale offre un luogo sicuro in cui mandare in esecuzione i programmi senza rischiare di danneggiare il sistema. In questo caso, il peggio che può accadere è una condizione di *denial of service*, in cui il programma consuma tutte le risorse della CPU.

Il "linguaggio macchina" della macchina virtuale Java è il *bytecode*, definito dalle specifiche della macchina virtuale. Un compilatore converte in bytecode (un file *.class*) il file sorgente, che però può anche non essere codice Java: esistono compilatori in bytecode per altri linguaggi, come Ada, COBOL o Delphi. Se ne evince che **non è sufficiente implementare costrutti sicuri nel linguaggio**, perché il sistema potrebbe comunque essere attaccato da del bytecode ostile generato da compilatori Java modificati, da compilatori che partono da linguaggi meno sicuri, o addirittura a mano: è necessario implementare anche dei controlli sull'ambiente di **esecuzione**.

L'esecuzione di una classe prevede:

1. il caricamento del file *.class*
2. l'esecuzione del metodo *main*
3. il caricamento delle classi accessorie

Ogni volta che una classe viene caricata, il bytecode viene verificato. Non tutte le classi vengono verificate: tipicamente le classi caricate localmente vengono considerate affidabili e non verificate, mentre le classi caricate attraverso la rete verranno verificate. Controlli diversi (quindi a costi diversi) vengono applicati a seconda delle caratteristiche della classe da caricare. La verifica del bytecode caricato è compito del *ClassLoader*, che solo al termine della verifica crea le classi vere e proprie. Il *ClassLoader* non cerca mai di caricare le classi di *java.** da remoto: in questo modo si evita che esse possano venir sostituite con classi modificate in cui i meccanismi di sicurezza siano stati in qualche modo manomessi. Inoltre, prepara uno spazio dei nomi (*name space*) diverso per ogni locazione da cui vengono scaricate le classi, in modo da evitare collisioni tra classi diverse che si chiamano con lo stesso nome.

A classi provenienti da host diversi non è permesso di comunicare all'interno della macchina virtuale Java, in modo da evitare che programmi non necessariamente affidabili possano ottenere informazioni da programmi considerati affidabili.

Una volta che le classi sono state caricate, la macchina virtuale compie i controlli necessari a tempo di esecuzione. Sono necessari controlli durante l'esecuzione sia per quel che riguarda i tipi nelle istruzioni di assegnamento che per quel che riguarda i limiti dei vettori. Infatti vi sono casi in cui il tipo specifico di un oggetto non può essere determinato durante la compilazione, ma solo in

esecuzione (ad esempio, se ad una variabile di tipo sovraclasse è stato assegnato un oggetto di tipo sottoclasse, e si vogliono usare metodi e attributi tipici della sottoclasse).

In Java, invece, i riferimenti agli oggetti includono informazioni complete sulla classe di cui l'oggetto è un'istanza: questo rende possibile il controllo della compatibilità tra tipi a runtime. Se il controllo fallisce, viene sollevata un'eccezione. Anche il controllo se l'indice di un vettore sta dentro ai limiti deve necessariamente essere compiuto durante l'esecuzione, ed eventualmente viene lanciata un'eccezione *ArrayIndexOutOfBoundsException* e l'accesso alla corrispondente zona di memoria negato.

Ogni macchina virtuale Java in esecuzione ha al più un *SecurityManager* installato, responsabile per la gestione delle politiche di sicurezza. *SecurityManager* è una classe del pacchetto *java.lang*: se ne può quindi definire una sottoclasse e usare il metodo *System.setSecurityManager()* per stabilire un security manager personalizzato. Una volta che un manager è stato installato, ogni tentativo di sostituirlo genera un errore (viene lanciata l'eccezione *Security Exception*): quindi nessuno può variare questa funzione in modo ostile, rimpiazzandola. Attraverso le politiche di sicurezza definite dal *Security Manager*, è possibile allentare o restringere i vincoli sulle operazioni che possono essere eseguite da un programma Java: ogni tentativo di eseguire un'operazione vietata solleva una *Security Exception*.

5.2 Applets

Si tratta di un esempio di strumento per la programmazione **lato client** basato su Java. Cruciali le caratteristiche di:

- **portabilità**, visto che lo sviluppatore non sa praticamente nulla della piattaforma su cui l'applet verrà messa in esecuzione;
- **sicurezza**, visto che l'host su cui viene eseguito lo user agent, e che manderà quindi in esecuzione l'applet, non sa nulla di chi l'applet ha sviluppato e deve aver fiducia per accettare di mandarlo in esecuzione.

Come abbiamo visto, Java ci offre buone caratteristiche per soddisfare entrambi i requisiti. Non possiamo però mai dare per scontato che lo user agent accetti le applet. Gli strumenti lato client devono offrire supporto per l'interazione con l'utente: devono quindi gestire gli **eventi**, quali i click del mouse, bottoni, campi in cui l'utente può introdurre testo, etc.. Anche i controlli che vengono eseguiti sul testo introdotto dall'utente sono finalizzati esclusivamente al supporto dell'interazione: non si può in nessun caso darli per scontati, in particolare se sono necessari alla sicurezza del sistema.

Anche le azioni delle applet sono ristrette all'interno di una **sandbox**, ovvero un'area del browser allocata specificatamente per l'applet. I **browser** hanno in genere un *SecurityManager* definito dal produttore. Le politiche di sicurezza previste nei browser prevedono che applet scaricate dalla rete:

- non possono leggere o scrivere **file** sulla macchina ospite;
- non possono creare **connessioni di rete** con host diversi dal server da cui sono stati scaricati;
- non possano caricare **librerie** o essere composte da metodi nativi (possono chiamare metodi nativi, ma non averli al proprio interno);
- non possono avviare nuovi **processi** sulla macchina ospite;
- non possono avere accesso a **informazioni** sulla macchina locale, tranne la versione di Java in uso, il nome e la versione del sistema operativo e alcune informazioni sui caratteri utilizzati; in particolare, non possono accedere al nome utente, al suo indirizzo di posta elettronica, etc.;
- tutte le **finestre** aperte da un applet visualizzano un messaggio di avvertimento.

Esistono situazioni in cui queste restrizioni sono troppo rigide: ad esempio in una **intranet aziendale**. Sono quindi stati introdotti strumenti per definire politiche a granularità più fine (si

veda il programma *policytool*). Per concedere il permesso di compiere determinate azioni ci si può basare su chi ha firmato il codice, sulla provenienza del codice oppure concederlo a chiunque.

Un altro metodo per distribuire applets attraverso la rete è di usare **Java Web Start**. Programmazione lato client: tuttavia scaricare le applet è relativamente costoso, anche perché occorre una richiesta diversa per ogni classe. Per questo motivo risulta conveniente impacchettare tutti i file *.class* ed eventuali immagini e/o suoni in un unico file JAR (Java ARchive) compresso che può essere scaricato con una sola richiesta.

La firma digitale può essere applicata ad ogni singolo file di un archivio JAR. Si noti che, se i browser sono abilitati per Java, le applets non danno problemi di installazione e che non c'è il rischio di provocare danni nella macchina del client a causa di qualche errore nel codice.

Essenziali per l'implementazione di applet sofisticati sono la padronanza delle librerie di supporto alla rete e al multi-threading.

- Originariamente le applet venivano richiamate nel file HTML attraverso la etichetta `<APPLET>` con opportuni parametri.
- Il W3C ne ha sconsigliato l'uso, preferendo l'etichetta `<OBJECT>` (HTML 4.0).
- I browser adesso supportano entrambe le etichette, ma le versioni più vecchie supportano solo `<APPLET>`.

Esiste un plug-in prodotto da Sun per avere una visualizzazione uniforme sia su Netscape che su Internet Explorer e che si adatta anche alle versioni più recenti di Java. Per utilizzare questo plug-in occorre una sintassi HTML piuttosto complessa, che può essere ottenuta con un convertitore automatico.

Se ci si basa solo sulle macchine virtuali Java incorporate nei browser e ci si rivolge ad un pubblico vasto, occorre fare qualche considerazione su se sia meglio usare solo le funzioni delle versioni di Java più diffuse. Si noti che applet molto semplici probabilmente non giustificano l'uso di Java rispetto a linguaggi di scripting (ad esempio JavaScript).

In generale, applicazioni Web basate su applet Java sono molto usate nelle intranet aziendali (uso uniforme del browser per l'utente e facilità di gestione e aggiornamento per l'amministratore). Sulla intranet, l'uso del plug-in è perfettamente accettabile e permette di mantenere il controllo sulla piattaforma Java senza problemi di portabilità. È anche possibile adottare Java 1.1, il cui supporto è garantito da quasi tutti i browser, e scaricare la libreria Swing sotto forma di file JAR come un'applet. Si tenga però presente che la dimensione del file da scaricare non è trascurabile (circa 1 Mbyte).

Un'applet è un'istanza della classe *java.applet.Applet*. Ecco un esempio di applet implementato cosí:

```
// very simple applet
// </applet>
import java.awt.*;
import javax.applet.*;

public class HelloWorldApplet extends Applet {
        public void paint(Graphics g) {
                g.drawString("Hello World!", 20, 20);
        }
}
```

Se invece si usa Swing, l'applet si ottiene come istanza della classe JApplet, che è la superclasse delle applet Swing e una sottoclasse immediata della classe Applet.

```
// very simple applet, con Java 2 e le librerie Swing
// </applet>
```

```
import javax.swing.*;
import java.awt.*;

public class Applet1 extends JApplet {
        public void init() {
                getContentPane().add(new JLabel("Buongiorno!"));
        }
}
```

5.2.1 Ciclo di vita di una applet

Quattro metodi della classe Applet:

1. *init()*: viene chiamato automaticamente dal sistema quando l'applet comincia l'esecuzione; comprende quindi tutte le inizializzazioni;
2. *start()*: viene chiamato dopo *init()* e ogni volta che l'utente torna alla pagina contenente l'applet; quindi, mentre *init()* viene chiamato una sola volta, *start()* può venir chiamato più volte; di solito è da qui che viene riavviato il thread, ad esempio per riprendere l'animazione; non è necessario implementarlo (in coppia con *stop()*), se non c'è alcuna operazione che debba essere sospesa quando l'utente esce dalla pagina corrente;
3. *stop()*: non è necessario richiamarlo esplicitamente, in quanto viene richiamato ogni volta che l'utente esce dalla pagina in cui si trova l'applet; serve a sospendere tutte quelle attività che consumerebbero risorse inutilmente;
4. *destroy()*: viene richiamato quando il browser viene chiuso normalmente, dopo aver chiamato *stop()*; non occorre distruggere il pannello.

Supponiamo che l'applet si trovi inserita in una **pagina HTML**: va inserito un elemento che indica allo user agent che deve scaricare l'applet. Gli attributi di tale elemento specificano la **posizione** dell'applet, e i **parametri** per la chiamata. Un'applet viene incorporata nella pagina visualizzata dal browser, in un frame le cui dimensioni sono fissate dai valori degli attributi WIDTH e HEIGHT specificate nell'etichetta HTML che richiama la applet.

Tra gli attributi dell'etichetta <APPLET> ricordiamo:

CODE : file *.class* corrispondente all'applet, a partire dalla pagina corrente; è obbligatorio indicare o CODE o OBJECT, che contiene l'oggetto applet serializzato;

CODEBASE : (facoltativo) indica la directory in cui si trovano i file relativi alle classi;

ARCHIVE : (facoltativo) il file o i file (separati da virgola) JAR, contenenti classi e altre risorse dell'applet; questi file vengono scaricati prima del caricamento dell'applet;

NAME : è possibile associare un nome all'applet in modo da poter chiamare alcuni dei suoi metodi da altri punti della pagina tramite del codice JavaScript (scripting language che vedremo e che non c'entra nulla con Java) o per far comunicare due applet.

Vediamo un esempio di **JavaScript** che richiama un metodo di un applet Java. Supponiamo di aver implementato un applet *CalculatorClass* che corrisponde ad una semplice calcolatrice. L'applet viene richiamata da una pagina HTML tramite l'etichetta:

```
<APPLET CODE="CalculatorApplet.class"
          WIDTH=150
          NAME="calc">
<APPLET/>
```

A questo punto ci si può riferire all'oggetto come *document.applets.nomeapplet*; nell'esempio:

```
var calcApplet = document.applets.{\bf calc};
```

e, supponendo che l'applet preveda un metodo *clear()*:

```
calcApplet.clear();
```

Un'altra applicazione dell'attributo NAME è di mettere in **comunicazione** due applet che stanno sulla stessa pagina, purché provengano dallo stesso CODEBASE (e quindi provengono dallo stesso sito). Il metodo *getApplet(String)* dell'interfaccia *AppletContext* permette di ottenere il riferimento all'applet. Per esempio, nel file HTML, supponendo la classe *Chart* sia una sottoclasse di *Applet*:

```
<APPLET CODE="Chart.class" WIDTH=100 HEIGHT=100 NAME="Chart1">
```

In un'altra applet, la chiamata

```
Applet chart1 = getAppletContext().getApplet("Chart1");
```

assegna a chart1 un riferimento all'applet.

Supponiamo ora che *Chart* contenga un metodo per ridisegnare in modo opportuno il grafico. Questo metodo potrà essere applicato previo un opportuno cast:

```
((Chart)chart1).replot()
```

Un'applet **non** può comunicare con un'applet su una pagina diversa.

A prescindere dalla presenza dell'attributo NAME, è possibile anche elencare tutte le applet presenti in una pagina Web: il metodo *getApplets()* restituisce un **oggetto enumerazione**.

```
Enumeration e = getAppletContext().getApplets();
while(e.hasMoreElements()) {
    Object a = e.nextElement();
    System.out.println(a.getClass().getName());
}
```

Si noti che i browser che non riconoscono l'etichetta APPLET visualizzano il testo compreso tra l'etichetta di apertura e quello di chiusura: questo può essere usato per implementare un comportamento **robusto**, che in questi casi visualizzi del testo alternativo.

Come abbiamo detto, al posto di APPLET, lo standard HTML 4.0 consiglia l'uso di **OBJECT** (con 35 diversi attributi). Questa etichetta serve per caricare oggetti di tipo diverso, tra i quali applet Java o componenti ActiveX (come il plug-in): l'attributo CODETYPE specifica la natura dell'oggetto (*application/java* per le applet Java).

Naturalmente è anche possibile passare dei **parametri di ingresso** all'applet, usando l'etichetta HTML PARAM, con attributi definiti dal programmatore. I parametri sono sempre restituiti come stringhe, che devono poi, ove necessario, essere convertite nel tipo richiesto. Il nome dell'attributo deve corrispondere esattamente al parametro passato al metodo *getParameter*, maiuscole e minuscole comprese. Esempio:

```
<APPLET CODE="FontParamApplet.class" WIDTH=200 HEIGHT=200>
    <PARAM NAME="font" VALUE="Helvetica">
    <PARAM NAME="size" VALUE="24">
<APPLET/>

public class FontParamApplet extends JApplet {
    public void init() {
        String fontName = getParameter("font");
        int fontSize = Integer.parseInt(getParameter("size"));
    }
}
```

Se un parametro è stato omesso, *getParameter* restituisce *null*.

Può essere utile ricordare che se un'applet apre una finestra di pop-up, essa comprenderà sempre un **messaggio** che avverte che si tratta di una finestra aperta da un applet. Tale messaggio non può essere tolto, per motivi di **sicurezza**: infatti la sua presenza rende impossibile simulare con una applet una finestra di un'altra applicazione, inducendo l'utente a rilasciare informazioni riservate (password, numero di carta di credito). Il messaggio può essere evitato solo se richiesto da una applet firmata.

Ovviamente le applet possono elaborare sia **immagini** che **audio**, attraverso file che spesso sono identificati dai relativi URL (si ricordi però che un'applet può collegarsi solo col server da cui è stata scaricata.

5.2.2 JNLP e Java Web Start

Come abbiamo visto, le applet firmate possono essere abilitate a essere eseguite con molte meno restrizioni delle applet qualsiasi e quindi possono in qualche modo prendere il posto delle normali applicazioni.

Tuttavia, esse hanno sempre il pesante vincolo di dover essere eseguite all'interno di un browser:

- con un maggior carico computazionale per il client che deve mandare in esecuzione anche il browser;
- con un impatto visivo non trascurabile, visto che vengono visualizzati anche tutti i pannelli di comandi del browser.

JNLP = Java Network Launch Protocol, permette di scaricare ed eseguire una applet anche fuori dal browser (da linea di comando, da un'icona o dal gestore di applicazioni). Un'applicazione JNLP può scaricare dinamicamente risorse da Internet durante l'esecuzione. La sua versione può venir controllata automaticamente (naturalmente sempre nell'ipotesi che l'utente sia collegato a Internet).

Naturalmente, cosí come con le applet, occorre cautela da parte del client nel trattare queste applicazioni, a cui in generale vengono applicate le stesse restrizioni della sandbox. Come per le applet, tuttavia, esse possono venir scaricate in file JAR firmati, in modo da dar modo all'utente di fidarsi di chi firma. Inoltre, anche quando vengono scaricate senza firma elettronica, possono chiedere il permesso di accedere ad alcune risorse del sistema del cliente attraverso le API JNLP. L'utente può concedere il permesso durante l'esecuzione del programma.

JNLP è un protocollo: ne serve un'implementazione, ad esempio **Java Web Start (JAWS)** Un'applicazione basata su JNLP prevede un JAR file contenente un'applicazione standard, e un file XML per indicare al client come scaricare e installare l'applicazione.

6. Servlet

Questo capitolo è dedicato alle *servlets*, un primo esempio di programmazione in Java lato server. Come vedremo, esse sono strettamente legate anche ad un altro importante strumento della programmazione Java lato server, le Java Server Pages o JSP.

Va inoltre ricordato come, trattandosi di programmazione Java, l'accesso alla base di dati, tipica delle applicazioni web standard, può venir implementata tramite *Java DataBase Connectivity o JDBC*. Per approfondire lo studio di questo argomento si può utilizzare [Eck03].

6.1 Introduzione

Le servlet sono un esempio dell'uso di Java per la programmazione Web lato server. Come vedremo, rispetto al CGI presentano vantaggi in termini di efficienza, di facilità di sviluppo, di accesso all'informazione di contesto.

Le servlet offrono tutti i vantaggi che abbiamo discusso nella scorsa lezione a proposito della programmazione Java. Occorre però andare oltre a ciò ed avere chiari quali sono i vantaggi dell'uso delle servlet rispetto ad un'applicazione CGI sviluppata in Java. Ovviamente anche quest'ultima presenta i vantaggi insiti nella programmazione Java, ma le servlet hanno diversi vantaggi in più quali maggiore efficienza, facilità di sviluppo e quindi sicurezza, maggiore condivisione dell'informazione sia all'interno dell'applicazione che col server.

D'altra parte, però, occorre che il server sia predisposto all'esecuzione delle servlet, e preveda quello che si chiama *servlet engine* o *servlet container*, quale ad esempio Tomcat.

6.1.1 Confronto tra servlet e CGI

Come dicevamo, e come torneremo a discutere nel seguito del corso, possono essere viste come un secondo passo nell'adattamento della programmazione lato server alle caratteristiche proprie delle applicazioni Web, in cui il server riceve tipicamente molte richieste in contemporanea e deve gestirle senza problemi di sincronizzazione e con un occhio alle prestazioni, sia in termini di tempo che di spazio.

Il primo passo infatti può venir rappresentato dal meccanismo del CGI, in cui lo sviluppo del programma è standard, senza particolari caratteristiche orientate al web se non il modo di

gestire l'input e l'output. In seguito, vedremo invece strumenti che ci permettono di sviluppare l'applicazione partendo direttamente dal concetto di *pagina* e di poter avere un rapporto più diretto col suo rendering.

Inoltre, le servlet permettono di *condividere delle informazioni* sia all'interno di un'applicazione che col server, siano esse le connessioni ad una o più basi di dati, un contatore per il conteggio delle richieste ad una data servlet, o altro. Nelle CGI, l'unico ambito di definizione (o scope) previsto è quello di una richiesta: in corrispondenza di ogni richiesta, viene avviato un processo, che termina una volta che la richiesta viene soddisfatta. Per condividere informazione tra due richieste diverse, anche se appartengono alla stessa applicazione o sono addirittura richieste per la stessa risorsa, occorre che tale informazione venga salvata in modo persistente (su disco o addirittura in una base di dati). In questo caso, la sincronizzazione tra i diversi accessi a questa informazione persistente o viene gestita dal database management system, o va esplicitamente considerata.

Un altro problema del CGI è che non dà modo ai programmi CGI di interagire direttamente col server o comunque di usare le funzionalità del server dopo che il programma è entrato in esecuzione, perchè viene eseguito come processo separato. Tanto per fare un esempio, uno script CGI non può scrivere sui log del server.

Nelle servlet, ad ogni richiesta corrisponde un nuovo *thread* all'interno del servlet engine: thread diversi possono accedere a variabili comuni, e questo permette di condividere informazioni tra richieste diverse alla stessa servlet. Ovviamente ci possono essere, e vanno di conseguenza gestiti, problemi di sincronizzazione tra richieste (e quindi thread) diverse.

Quello che ancora manca alle servlet è la separazione tra aspetti grafici e aspetti di programmazione: vedremo come questa verrà ottenuta in altri strumenti, tra cui le JSP, un modo di sviluppare servlet che consente di separare questi due aspetti.

Ricapitolando, come per il CGI, una **servlet** permette di implementare delle funzionalità eseguite lato server. Quando un Web server riceve una richiesta per un programma CGI, esso:

- dà origine ad un nuovo processo per eseguire il programma;
- passa a questo processo le informazioni necessarie per costruire la risposta via standard input e variabili d'ambiente.

La creazione di un nuovo processo per ogni richiesta ricevuta richiede risorse sia in termini di tempo che di memoria da parte del server, e limita quindi le risposte che possono essere elaborate contemporaneamente.

Per completezza, citiamo alcune altre alternative:

- **FastCGI**: sono una variante del CGI che però crea un singolo processo persistente per ogni programma CGI. Tuttavia, in caso di richieste concorrenti occorre comunque duplicare i processi, e comunque non risolve il problema dell'interazione col server.
- **perl**: poiché il *perl* è senz'altro il linguaggio più diffuso per implementare programmi CGI, diversi server hanno cercato di velocizzarne l'esecuzione. Apache ha integrato l'interprete *perl* (mod_perl) nel server, permettendo cosí di eseguire gli script senza duplicare il processo, con miglioramento di prestazioni sia in tempo che in efficienza. ActiveState ha creato PerlEx, per i Web server di Windows NT, per ottimizzare le prestazioni.
- API di estensione del server (NSAPI di Netscape, ISAPI di Microsoft), non rappresentano un miglioramento, ma un'alternativa al CGI, perchè forniscono delle API per estendere le funzionalità del server.
- **Active Server Pages** (ASP), prodotte da Microsoft, in cui del codice immerso nelle pagine HTML (VBScript, JScript, altro), viene letto ed eseguito dal Web server prima di spedire la pagina al client.
- **JavaScript lato server**, prodotto da Netscape, è simile, ma usa JavaScript come codice; inoltre, le pagine vengono precompilate per migliorare l'efficienza.
- **PHP**, ancora immerso nell'HTML ed eseguito dal server.

6.1.2 Come si presenta una servlet

Le **servlet** sono delle classi Java che possono venir caricate *dinamicamente* per estendere le funzionalità del server. A differenza delle estensioni proprietarie, le servlet vengono eseguite all'interno della Java Virtual Machine sul server, e sono quindi sicure e portabili. Operano solo sul server, e non richiedono quindi alcun supporto da parte del browser. Al contrario del CGI e FastCGI, che usano processi multipli per gestire programmi diversi e/o richieste diverse, le servlet vengono gestite all'interno del processo del server da thread separate. Questo presenta due vantaggi:

- efficienza e scalabilità;
- ottima interazione col server, visto che vengono eseguite all'interno dello stesso processo.

Inoltre le servlet offrono un'ottima portabilità sia su sistemi operativi diversi che su server e servlet engines diversi. Oltre alle servlets vere e proprie, le **Java Server Pages (JSP)** vengono tradotte in servlets e forniscono quindi un sistema per implementare il meccanismo di richiesta e risposta del server senza doversi occupare di tutti i dettagli implementativi delle servlets.

Normalmente, si usano:

- le **JSP** quando la maggior parte del contenuto passato al client è testo statico e markup, mentre solo una piccola porzione del contenuto viene prodotta dinamicamente mediante codice Java;
- le **servlets** quando la porzione di testo da produrre staticamente è poca; infatti, la maggior parte delle servlets non produce testo, ma porta a termine dei compiti richiesti dal client, e alla fine richiama altre servlets o JSPs per produrre una risposta.

Le servlet normalmente passano il loro risultato al client sotto forma di file HTML, XHTML, XML da visualizzare nel browser, ma anche in altri formati, quali immagini e dati binari.

Anche se vengono tipicamente usate per server Web, le servlet possono estendere ogni tipo di server: di posta, FTP, etc.. Infatti, tutte le servlet devono implementare l'interfaccia `Servlet`, che è la classica API di un fornitore di servizi.

Le *interfacce* in Java presentano delle analogie con le classi, ma non hanno variabili di istanza e, pur avento le dichiarazioni dei metodi, ma tali dichiarazioni non hanno alcun body associato. Ogni classe può implementare un numero qualsiasi di interfacce. Per implementare un'interfaccia, occorre che la classe definisca un'implementazione per tutti i metodi dell'interfaccia, in modo che i prototipi coincidano. La gerarchia delle interfacce è indipendente da quella delle classi, quindi classi diverse e in posizioni completamente scorrelate della gerarchia delle classi possono implementare la stessa interfaccia.

I metodi dell'interfaccia `Servlet` vengono invocati automaticamente dal servlet engine.

```
public void init(ServletConfig config)
                            throws ServletException;
public ServletConfig getServletConfig();
public void service(ServletRequest req, ServletResponse res)
                           throws ServletException, IOException;
public String getServletInfo();
public void destroy();
}
```

L'interfaccia dichiara quindi cinque metodi, di cui però non dà il body e quindi l'implementazione:

1. *void init(ServletConfig config)*: inizializza la servlet una volta sola durante il ciclo di esecuzione della servlet;
2. *ServletConfig getServletConfig()*: ritorna un oggetto che implementa l'interfaccia *ServletConfig* e fornisce l'accesso alle informazioni sulla configurazione della servlet, tra cui i parametri di inizializzazione e il **ServletContext**, ovvero il suo ambiente (il servlet engine in cui vengono eseguite);

3. *String getServletInfo()*: viene definito in modo da restituire una stringa con informazioni quali autore e versione;
4. *void service(ServletRequest request, ServletResponse response)*: viene mandato in esecuzione in risposta alla richiesta mandata da un client alla servlet;
5. *void destroy()*: invocato quando la servlet viene terminata; usato per rilasciare le risorse (connessioni a basi di dati, file aperti, etc.).

Il metodo *init()* viene invocato solo quando la servlet viene caricata in memoria, di solito a seguito della *prima* richiesta per quella servlet. Solo dopo che il metodo è stato eseguito, il server può soddisfare la richiesta del client invocando il metodo *service*, che riceve la richiesta, la elabora e prepara la risposta per il client. Quindi il metodo *service* viene richiamato ogni volta che si riceve una richiesta. Tipicamente, per ogni nuova richiesta il servlet engine crea una nuova thread di esecuzione in cui viene mandato in esecuzione *service*. Solo quando il servlet engine termina la servlet, viene invocato il metodo *destroy* per rilasciare le risorse della servlet. Ecco il guadagno in efficienza rispetto al CGI: una thread invece di un nuovo processo per ogni richiesta.

I pacchetti delle servlet definiscono due classi astratte che implementano l'interfaccia *Servlet*:

- **GenericServlet** (in javax.servlet);
- **HttpServlet** (in javax.servlet.http).

Queste classi forniscono un'implementazione di default per tutti i metodi dell'interfaccia.

A noi interessa

```
public abstract class {\bf HttpServlet}
          extends GenericServlet implements java.io.Serializable
```

che implementa una servlet rispetto al protocollo HTTP, e quindi per il web. Essa fornisce una classe astratta che verrà estesa dall'implementazione delle classi che lo sviluppatore implementa per realizzare la propria applicazione. Ogni sottoclasse di `HttpServlet` deve fare overriding di almeno un metodo, di solito uno dei seguenti:

- `doGet`, se la servlet viene richiamata col metodo HTTP GET;
- `doPost`, per richieste HTTP con metodo POST;
- `doPut`, per richieste HTTP con metodo PUT;
- `doDelete`, per richieste HTTP con metodo DELETE;
- `init` e `destroy`, per gestire eventuali risorse necessarie al funzionamento della servlet (ad esempio, connessioni con la base di dati);
- `getServletInfo`, usata dalla servlet per fornire informazioni riguardanti se stessa.

Una strategia tipicamente utilizzata è di non riscrivere il metodo *service*, ma piuttosto i metodi "doXXX". La maggior parte delle servlet estende una di queste due classi e sovrascrive (override) uno o più dei suoi metodi. In applicazioni Web di solito si usano sottoclassi di `HttpServlet`. Il metodo più significativo in una servlet è ovviamente il metodo *service*, o in alternativa di metodi *doXXX*, che prendono in ingresso:

- un oggetto **ServletRequest**, corrispondente ad uno stream di ingresso, da cui leggere la richiesta del client;
- un oggetto **ServletResponse**, corrispondente ad uno stream di uscita, su cui scrivere la risposta.

Questi stream possono essere basati su byte o su caratteri. In caso di problemi, viene lanciata una `ServletException` o una `IOException`. In caso di richieste multiple, diverse chiamate al metodo `service` verranno eseguite in *parallelo*: può quindi rendersi necessario che i programmatori serializzino l'accesso alle risorse mediante tecniche di sincronizzazione.

Le servlet possono anche implementare l'interfaccia `javax.servlet.SingleThreadModel`: in ogni istante, per ogni servlet si può avere solo una thread di esecuzione per il metodo `service()`. In questo caso, il servlet engine può creare più istanze della stessa servlet per gestire richieste multiple che giungono alla servlet in parallelo e

può rendersi necessario provvedere a sincronizzare l'accesso a risorse condivise usate dal metodo `service()`, come verrà mostrato in un esempio più avanti.

Il funzionamento tipico delle servlet è basato su multithreading: ne consegue la necessità di gestire richieste concorrenti e la sincronizzazione nell'accesso alle risorse.

6.1.3 La classe HttpServlet

Le servlet per il Web tipicamente estendono la classe **HttpServlet**.

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
   int i = 0; // Servlet "persistence"
   public void service(HttpServletRequest req, HttpServletResponse res) \
                                                          throws IOException {
      res.setContentType("text/html");
                    // {\bf prima} di inizializzare un Writer o un OutputStream
      PrintWriter out = res.getWriter();
             // in alternativa, getOutputStream( ) produce un OutputStream, usato
                                      // per risposte binarie
      out.print("<HEAD><TITLE>");
      out.print("A server-side strategy");
      out.print("</TITLE></HEAD><BODY>");
      out.print("\section{Servlets Rule! " + i++);
      out.print("}</BODY>");
      out.close();
   }
}
```

Il valore di `i` viene mantenuto tra due diverse richieste alla servlet: poiché una sola servlet di quella particolare classe viene caricata nel *servlet engine*, e non viene mai scaricata se non quando viene terminato il *servlet engine* stesso, cosa che avviene di norma al reboot della macchina, gli attributi della classe divengono di fatto oggetti persistenti. Con CGI, invece, occorreva scrivere i valori sul disco, a scapito di semplicità e portabilità.

Per non perdere l'informazione quando il servlet engine viene, per qualsiasi motivo, terminato, si utilizzano i metodi *init()* e *destroy()*, che vengono chiamati automaticamente quando la servlet viene caricata e scaricata, e in ogni caso in cui il servlet engine viene terminato.

Il metodo `service()` viene sovrascritto per tener conto delle diverse richieste che arrivano ad un client Web. Il metodo `service()` prima di tutto decide quale richiesta è stata inoltrata, e poi chiama il metodo opportuno: ad esempio, `doGet()` o `doPost()` a seconda del tipo di richiesta. Altre chiamate di uso molto meno frequente: `doDelete`, `doOptions`, `doPut`, `doTrace`.

Tutti questi metodi ricevono in ingresso due parametri:

- `HttpServletRequest request`: che rende accessibili i dati relativi alla richiesta inoltrata dal client;
- `HttpServletResponse response`: per la risposta da passare al client.

e hanno tipo di ritorno void.

6.1.4 L'interfaccia HttpServletRequest

I metodi dichiarati da questa interfaccia provengono in parte dall'interfaccia `ServletRequest`. Un'implementazione dell'interfaccia è data dalla classe HttpServletRequestWrapper.

- `String getParameter( String name )`: torna il valore del parametro *name* passato nella get/post;
- `Enumeration getParameterNames()`: returns an enumeration of string objects containing the names of the parameters contained in this request. "
- `String[] getParameterValues( String name )`: se name ha valori multipli
- `Cookie[] getCookies()`: memorizzati dal server nel client;
- `HttpSession getSession( boolean create )`: se l'oggetto non esiste e `create = true`, allora lo crea.

6.1.5 L'interfaccia HttpServletResponse

- `void addCookie( Cookie cookie )`: nell'header della risposta passata al client; essa viene memorizzata a seconda dell'età massima e dell'abilitazione o meno ai cookie disposta nel client;
- `ServletOutputStream getOutputStream()`: stream basato su byte per dati binari;
- `PrintWriter getWriter()`: stream basato su caratteri per dati testuali;
- `void setContentType( String type )`: specifica il formato MIME.

6.1.6 Esempi di servlet

La classe *HttpServlet* fornisce un metodo *doGet()* di default che semplicemente presenta un messaggio di errore. Ecco un esempio con `doGet()` che stampa semplicemente una pagina di benvenuto in risposta ad una richiesta get:

```
// Una semplice servlet per elaborare una richiesta di tipo get

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class SempliceServlet extends HttpServlet {

   // elabora le richieste get dal client
   protected void doGet( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // send XHTML page to client
      // start XHTML document
      out.println( "<?xml version = \"1.0\"?>" );

      out.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
         "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
         "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
```

```
      out.println(
         "" );

      // head section of document
      out.println( "" );
      out.println( "<h4>Un semplice esempio di servlet</h4>" );
      out.println( "" );

      // body section of document
      out.println( "" );
      out.println( "<b>Welcome to Servlets!</b>}" );
      out.println( "" );

      // end XHTML document
      out.println( "" );
      out.close(); // lo stream va chiuso
   }
}
```

La servlet viene invocata attraverso una form XHTML:

```
<?xml version = "1.0"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">


<H1>Gestione di una richiesta HTTP di tipo GET<H1>



<form action = "SempliceServlet" method = "get">
<label>Click the button to invoke the servlet
 <input type = "submit" value = "Ottieni la risposta" />
</label>
</form>
```

Alla servlet possiamo anche passare dei parametri: ecco un esempio in cui viene passato il nome proprio della persona:

```
public class SecondaSempliceServlet extends HttpServlet {
   // ancora la richiesta di tipo get
   protected void doGet( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      String firstName = request.getParameter( "firstname" );

      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // send XHTML document to client
```

```
      // start XHTML document
      out.println( "<?xml version = \"1.0\"?>" );
      out.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
         "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
         "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
      out.println(
         "" );

      // head section of document
      out.println( "" );
      out.println(
         "</h4>Elaborazione di una richiesta get che include dati<h4>" );
      out.println( "" );

      // body section of document
      out.println( "" );
      out.println( "<h4>Hello " + firstName + "," );
      out.println( "Welcome to Servlets!</h4>" );
      out.println( "" );

      // end XHTML document
      out.println( "" );
      out.close(); // chiusura dell stream
   }
}
```

I parametri vengono passati come coppie nome/valore nelle richieste get (e, essendo un get, compaiono nell'URL, e potrebbero anche essere direttamente scritti lí).

```
<?xml version = "1.0"?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
           "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">


<h1>Elaborazione di una richiesta get che include dati</h1>


<form action = "SecondaSempliceServlet" method = "get">
<label>Qual \‘e il tuo nome? ricordati di premere Submit!
<input type = "text" name = "firstname" />
<input type = "submit" value = "Submit" />
</label>
</form>
```

Essendo un get, nell'URL avremo `SecondaSempliceServlet?firstname=Renato`.

```
<form action="SecondaSempliceServlet" method="get">
   <label>Qual \‘e il tuo nome? ricordati di premere Submit!
      <input type="text" name="firstname">
      <input type="submit" value="Submit">
   </label>
```

```
</form>
```

Lo stesso, ma con una richiesta di tipo post: utilizziamo la stessa Servlet, ma aggiungiamo il seguente metodo:

```
// elabora una richiesta di tipo post
   protected void doPost( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      {\bf String firstName = request.getParameter( "firstname" );}
      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // send XHTML page to client
      // start XHTML document
      out.println( "<?xml version = \"1.0\"?>" );
      out.println( "<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
         "XHTML 1.0 Strict//EN\" \"http://www.w3.org" +
         "/TR/xhtml1/DTD/xhtml1-strict.dtd\">" );
      out.println(
         "" );

      // head section of document
      out.println( "" );
      out.println(
         "<h4>Elaborazione di una richiesta di tipo post con dati</h4>" );
      out.println( "" );

      // body section of document
      out.println( "" );
      out.println( "<h4>Hello " + firstName + "," );
      out.println( "Welcome to Servlets!</h4>" );
      out.println( "" );

      // end XHTML document
      out.println( "" );
      out.close(); // chiudi lo stream
   }
}
```

Richiamata con:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

   <h1>Elaborazione di una richiesta di tipo post con dati</h1>



   <form action = "SecondaSempliceServlet" method = "post">
```

```
      <label>
         Qual \‘e il tuo nome? ricordati di premere Submit!
         <input type = "text" name = "firstname" />
         <input type = "submit" value = "Submit" />
      </label>
   </form>
```

Al posto di `doGet()` o `doPost()` si può comunque utilizzare anche semplicemente `service()`, come nel seguente esempio:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
   public void service(HttpServletRequest req, HttpServletResponse res) \
                                                        throws IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      Enumeration flds = req.getParameterNames();
      if(!flds.hasMoreElements()) { // la richiesta non contiene alcun parametro:
                                    // costruisco una form per raccoglierli
         out.print("");
         out.print("<form method=\"POST\"" +
                  " action=\"EchoForm\">");
         for(int i = 0; i < 10; i++)
            out.print("{\bf Field" + i + "} " +
                     "<input type=\"text\""+" size=\"20\" name=\"Field" + i +
                     "\" value=\"Value" + i + "\">");
         out.print("<INPUT TYPE=submit name=submit"+
         " Value=\"Submit\"></form>");
      } else {
         out.print("\section{Ecco i valori che hai inserito nella form:}");
         {\bf while(flds.hasMoreElements()) {
            String field= (String)flds.nextElement();
            String value= req.getParameter(field);}
            out.print(field + " = " + value + "");
         }
      }
      out.close();
   }
}
```

6.1.7 Servlets e multithreading

Le richieste dei *client* vengono soddisfatte dal servlet engine facendo uso di un pool di thread. Per evitare collisioni, occorre che il metodo `service()` sia `thread-safe`: ogni accesso a risorse comuni quali files e basi di dati va protetto col costrutto `synchronized`.

```
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.io.*;

public class ThreadServlet extends HttpServlet {
   int i;
   public void service(HttpServletRequest req, \
                       HttpServletResponse res) throws IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      synchronized(this) {
         try {
            Thread.sleep(5000); // 5 secondi
         } catch(InterruptedException e) {
            System.err.println("Interruzione");
         }
      }
      out.print("<h4>Finished " + i++ + "</h4>");
      out.close();
   }
}
```

Se la parte da sincronizzare viene comunque eseguita in ogni caso, si può mettere la parola chiave `synchronized` prima del metodo `service`.

7. CMS e Web Frameworks

Le moderne applicazioni per il web hanno molte caratteristiche in comune con le classiche applicazioni desktop, sia in termini tecnologici, che in termini di esperienza e interazione utente, tanto da guadagnarsi la denominazione di *Rich Internet Applications* (RIA). Le RIA, infatti, utilizzano tecnologie lato client in maniera intensiva, quali Javascript e CSS3 in combinazione con AJAX per l'invio asincrono di richieste HTTP, per la realizzazione di interfacce utente sempre più simili a quelle desktop. A titolo di esempio, si consideri l'applicazione web progettata per la realizzazione di un tour virtuale per le principali funzionalità dalla distribuzione Linux Ubuntu[1]. Si noti, infatti, come l'esperienza dell'utente durante l'interazione sia progettata in modo da avere una corrispondenza quasi totale con quella "reale" (desktop) del sistema operativo, dando l'impressione che l'interazione stia avvenendo più con una distribuzione su macchina virtuale[2], piuttosto che con un'applicazione web all'interno del proprio browser.

Per tali ragioni, lo sviluppo e la diffusione delle RIA hanno sancito la nascita del cosiddetto Web 2.0, acquisendo la denominazione di soluzioni *batterie incluse*, vale a dire soluzioni integrate (tutto compreso).

In questo contesto vanno considerati i *Content Management Systems* (CMS) e i *Web Application Frameworks* o, più brevemente, *Web Frameworks*, entrambi discussi in questo capitolo. I CMS hanno lo scopo di gestire workflows e lo sviluppo di contenuti in ambienti di lavoro collaborativo e vengono quindi progettati per semplificare la pubblicazione di documenti su web. Alcuni dei più noti e più utilizzati CMS sono `Joomla`, `Drupal`, `WordPress`.

I Web Frameworks, invece, si prefiggono lo scopo di alleggerire il lavoro legato allo sviluppo di applicazioni web, fornendo supporto per le funzionalità tipicamente richieste per la loro realizzazione quali, ad esempio, la gestione e la visualizzazione dei dati. Attualmente esiste un notevole numero di web frameworks, i quali si differenziano principalmente per le funzionalità di supporto messe a disposizione dello sviluppatore e, soprattutto, per il linguaggio di programmazione in cui essi sono implementati. Tra i web frameworks più noti si possono citare: `Apache Struts`

[1] `http://www.ubuntu.com/tour/en/`
[2] `http://it.wikipedia.org/wiki/Macchina_virtuale`

2 e Play, basati su Java; Ruby on Rails basato su Ruby; CakePHP e Symfony basati su PHP; Django e Google App Engine basati su Python.

Con riferimento alle RIA, infine, è possibile enfatizzare la differenza che sussiste tra CMS e web frameworks: i CMS **sono** RIA, mentre i web frameworks mettono a disposizione gli **strumenti** per la sviluppo delle RIA.

7.1 Content Management Systems

Come dice la parola, un CMS è un sistema per la *gestione dei contenuti*. Questo tipo di sistemi si propone di organizzare il lavoro sui contenuti di un gran numero di utenti. Gli utenti ricoprono ruoli diversi sia in generale che rispetto al singolo documento.

Noi ci focalizziamo soprattutto sui Web CMS, ma ce ne sono anche di altri tipi. Un Web CMS è un CMS che funziona su web e che prevede la possibilità di accesso principalmente attraverso un browser. A volte, soprattutto per la parte di amministrazione, può venir previsto anche un diverso client. Una wiki è un esempio di Web CMS.

Ovviamente agli utenti (a parte eventualmente alcuni) non viene richiesta nessuna competenza informatica o web, al di là di saper utilizzare un generico browser. La maggior parte dei CMS usa una *base di dati relazionale* per immagazzinare sia i contenuti che i metadati che qualsiasi altro contenuto necessario al sistema. I contenuti sono spesso immagazzinati come *XML*, per facilitarne il riutilizzo e per permettere una maggior flessibilità a livello di presentazione. Oltre a documenti testuali, può gestire anche documenti *multimediali*: immagini, file audio, video, etc. e anche files di dati.

Prima di tutto, un CMS offre supporto alla *creazione*, alla *modifica* e alla *pubblicazione* dei documenti. In particolare, un CMS può mettere a disposizione di chi crea il documento uno o più template approvati dall'organizzazione. Possono anche esserci strumenti più dinamici, quali wizards, che aiutano a soddisfare i requisiti di formattazione richiesti dal documento. E poi i soliti strumenti quali controllo ortografico e cosí via. È generalmente possibile anche sviluppare un documento in *collaborazione* tra più persone, con ovvi problemi di sincronizzazione e di controllo delle versioni.

In genere, si dà la possibilità anche di *importare* documenti legacy o eventualmente esterni nel sistema, eventualmente anche mediante l'uso di OCR. Anche in questo caso, i documenti verranno riportati nei formati previsti dal sistema, quali HTML o PDF. È poi previsto anche un sistema di *controllo delle versioni*.

Un'altra importante funzionalità offerta dai CMS è il supporto al *retrieval* dei documenti, con metodi di ricerca che possono variare da sistema a sistema: ricerca per parole chiave, query booleane, eventualmente specializzate sui diversi campi del documento. I documenti immessi nel CMS devono quindi venir organizzati in una repository con indicizzazioni opportune, in modo da facilitare l'accesso.

Alcuni CMS insistono anche sulla possibilità di separare i contenuti dalla formattazione, permettendo la configurazione degli aspetti grafici.

Controllo degli accessi : un aspetto importante che un CMS deve essere in grado di gestire, definendo diverse categorie di utenti con diversi profili e quindi permessi di accesso.

Workflow : supporto al lavoro in collaborazione che segue il percorso di un documento elettronico prevedendo una serie di azioni ad esso associate da parte di diverse persone. Fa uso di strumenti anche diversi dai semplici strumenti web, come posta elettronica e indirizzamento automatico. Gli eventi associati al workflow devono poter venir segnalati agli interessati in modo asincrono.

Se il workflow permette la gestione di procedure molto strutturate, altri strumenti, quali ad esempio i forum, offrono supporto a procedure completamente non strutturate o poco strutturate.

one-to-one marketing incluso da molti dei CMS, prevede strumenti per personalizzare la pagina per lo specifico utente (ad esempio, un cliente), programmi di fidelizzazione, e cosí via.

7.2 Cosa si intende per web framework

Il termine *framework* è uno dei termini più usati e abusati in informatica, a cui si attribuiscono spesso differenti significati e accezioni. Inoltre, esistono diverse tipologie di framework: framework di *testing*, framework *di sviluppo*, *web* framework, ecc. Pertanto è bene sottolineare per prima cosa che cosa si intende per framework, in generale, e per web framework, in particolare.

Un *framework* non è né un'API, che specifica come le componenti software devono interagire l'uno con l'altra, né una libreria, vale a dire una raccolta di funzioni o oggetti destinati ad un particolare scopo. Inoltre un framework non è un'applicazione, ma uno strumento che offre supporto allo sviluppo di applicazioni, mettendo a disposizione una serie di strumenti e soluzioni integrate già pronte all'uso.

Un *web framework*, dunque, non è altro che un framework per lo sviluppo di applicazioni specifiche per il web, mettendo a disposizione sia strumenti per la programmazione lato server, che per la programmazione lato client.

I web framework nascono dalla consapevolezza che le operazioni svolte da uno sviluppatore con la programmazione lato server sono tipicamente sempre le stesse: connessione alla base di dati, scelta di dettagli quali il `content-type` della pagina restituita, gestione delle variabili di sessione e altre operazioni simili. Un web framework si fa quindi carico di sistematizzare e di risolvere questi problemi in modo standard, in modo che lo sviluppatore possa concentrarsi esclusivamente sulle caratteristiche significative dell'applicazione che sta sviluppando.

7.2.1 Architettura delle applicazioni web

Un'**architettura** (software) descrive la **struttura** di un'applicazione, la decomposizione in **componenti**, le loro **interfacce** e **interazioni**. L'architettura rappresenta il passaggio tra analisi e implementazione. Le applicazioni Web sono un esempio di architettura **multi-tier** (multistrato) o **n-tier**. Per **tier** si intende un raggruppamento logico di funzionalità. I diversi *tier* possono stare sulla stessa macchina o su macchine diverse.

Le applicazioni Web più semplici sono a tre *tier*, quindi **three-tier**, come quella illustrata in Figura 7.1.

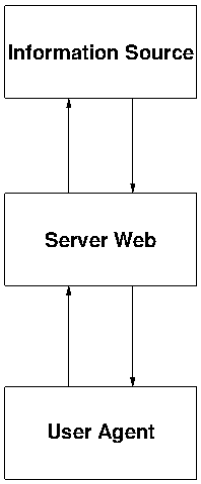


Figura 7.1: Architettura a tre *tier*

Il *tier* intermedio controlla l'interazione tra client e sorgente di informazione; implementa:

- il meccanismo di controllo (**controller logic**): elabora la richiesta del client e si procura i dati da presentargli;
- il meccanismo di presentazione (**presentation logic**) elabora questi dati per presentarli all'utente (in HTML, XHTML, WML, ...)
- le regole del dominio applicativo (**business logic**) e si assicura che i dati siano affidabili prima di usarli per aggiornare la sorgente di informazione o per rispondere all'utente; le **business rules** regolano l'accesso dei client ai dati e l'elaborazione dei dati da parte dell'applicazione.

7.2.2 Modello MVC e Web Frameworks

Uno degli aspetti maggiormente rilevanti relativi all'impiego di un framework per lo sviluppo delle applicazioni (web) riguarda il dover adattare il proprio modello di design a quello "imposto" dal framework utilizzato[3]. Per ciò che riguarda nello specifico i web frameworks, una prima idea potrebbe essere quella di adottare la solita architettura software a tre strati (3-tier) introdotta in Sezione 7.2.1 che contraddistingue le applicazioni web. Tuttavia questa struttura è caratterizzata da una forte dipendenza e da un alto grado di accoppiamento tra i vari strati e risulta quindi non sufficientemente modulare da poter essere sistematizzata in un framework.

Si ricordi come l'obiettivo principale di un framework sia quello di fornire componenti che possano poi essere combinate in base alle particolari esigenze di sviluppo. Per cui, il requisito di modularità risulta essere di fondamentale importanza per la realizzazione di un (web) framework. Il modello architetturale 3-tier, infatti, presenta un'integrazione troppo stretta tra livello di presentazione (*View o Vista*), di elaborazione (*Controller o Controllore*) e l'acquisizione dei dati dalla sorgente di informazione, che tipicamente è una base di dati (*Model o Modello.*) Inoltre, dal punto di vista dello sviluppo, ciascuno dei tre strati richiede competenze diverse e come abbiamo spesso sottolineato questo rende più difficile la spartizioni delle responsabilità all'interno di un gruppo di lavoro.

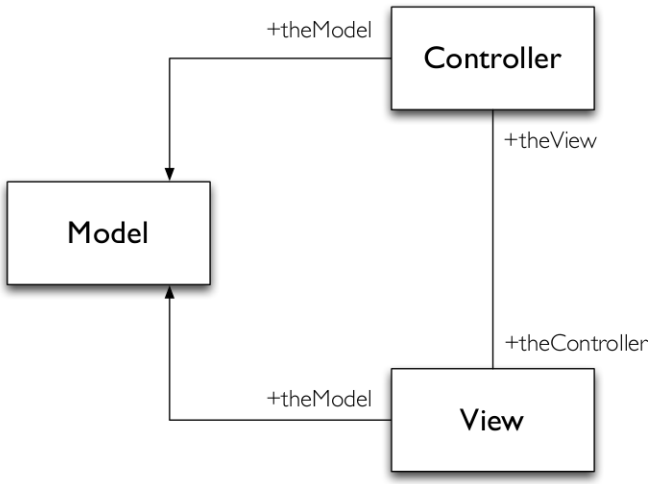


Figura 7.2: Modello UML del Pattern MVC.

Il modello **MVC**, ModelViewController, è stato adattato dal pattern architetturale precedente ed è rappresentato in Figura 7.2 [4].

Nella versione Web, tuttavia, non vi è alcuna relazione diretta tra modello e vista, a causa del fatto che la comunicazione tra essi è veicolata da richieste HTTP che è notoriamente un protocollo

[3] Si vedrà nel seguito quali siano i vantaggi derivanti da tale "imposizione"

[4] Tuttavia si ricorda che non sussiste necessariamente una corrispondenza tra i tre livelli (tier) e le tre entità coinvolte nel pattern MVC.

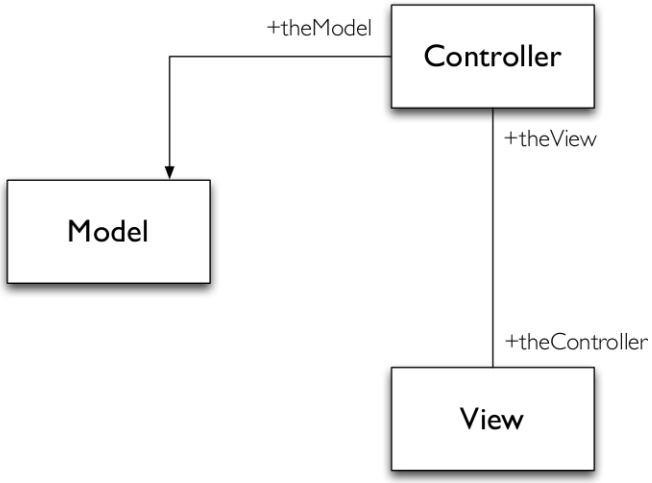


Figura 7.3: Modello UML del Pattern WebMVC

stateless (senza stato). Per tale ragione, il pattern MVC per il web è a volte indicato con il termine di *WebMVC* (Figura 7.3); tuttavia, quando è chiaro che si sta trattando applicazioni web, spesso lo si nomina semplicemente MVC.

In questa versione del pattern, il modello ha la responsabilità della gestione dell'acquisizione dei dati e del dominio, il controllore dell'elaborazione dei dati e della comunicazione tra le diverse componenti dell'applicazione, mentre la vista è la sola responsabile della visualizzazione e presentazione dei dati.

I tre diversi livelli di astrazione del pattern (Web-)MVC sono delegati ad assolvere specifiche funzionalità, cui corrispondono specifiche componenti software messe a disposizione dai web frameworks:

Model : centralizza il controllo e l'accesso ai dati, rendendo fruibile al livello di controllo la gestione dei dati. Usualmente lo scambio dei messaggi tra il livello "modello" e "controllo" è veicolata da un *ORM (Object Relational Mapping)* responsabile del mapping tra modello dei dati e corrispondente modello ad oggetti dell'applicazione. In qualche modo, quindi, si fa carico dell'interfacciamento con la base di dati.

View : è responsabile della composizione e visualizzazione delle risorse, spesso rappresentate da pagine HTML, ma anche JSON, XML, e altri ancora. A tale scopo, questo livello integra tipicamente librerie JavaScript per la programmazione lato-client e meccanismi di composizione e aggregazione delle pagine web, attraverso sistemi di *template* per la compilazione dinamica delle risorse web[5].

Controller : ha la responsabilità di acquisire le richieste HTTP, gestire la comunicazione con il modello dei dati e di trasferire i dati alla "vista" per la compilazione delle risorse (grazie al motore di template precedentemente citato). Caratteristiche tecniche afferenti a tale livello comprendono:

1. URL Routing RESTful: meccanismo di indirizzamento degli URL supportati dalla applicazione web, che rispetti le linee guida della specifica REST (REpresentational State Transfer).[6]

[5] `http://en.wikipedia.org/wiki/Web_template_system`

[6] REST(Representational State Transfer) si riferisce ad un insieme di vincoli per il progetto di architetture software che punta ad ottenere sistemi distribuiti che siano efficienti, affidabili e scalabili. Un sistema è detto RESTful se aderisce a tali vincoli. L'idea di base è che una risorsa, ad esempio, un documento, viene trasferita insieme al suo stato e alle sue relazioni attraverso operazioni standard ben definite. I formati o i servizi vengono detti RESTful se modificano direttamente il tipo della risorsa (documento) anziché applicare delle azioni. Dal momento che anche HTTP trasferisce

2. meccanismi di sicurezza contro attacchi quali SQL-Injections, Clickjacking e CSRF discussi nel Capitolo 3.

In generale, quindi, i Web frameworks offriranno supporto per tutte quelle che sono caratteristiche tipiche di un'applicazione web, inclusi per esempio il supporto al session-tracking e all'autenticazione dell'utente.

I web frameworks, dunque, forniscono supporto per tutti (o alcuni de) i livelli del pattern Web-MVC, mettendo a disposizione dello sviluppatore uno *stack* di componenti software. Difatti, si parla di *full-stack* frameworks per riferirsi a quei frameworks che offrono supporto, attraverso componenti propriamente implementate o integrate da strumenti di terze parti, per ciascun livello dello stack software.

7.2.3 Micro-stack framework

Di più recente introduzione, sono i *micro-stack* frameworks, che, a differenza dei primi, non offrono alcun supporto *out-of-the-box* per i livelli di modello e vista, ma offrono solo l'infrastruttura in grado di gestire e smistare le richieste HTTP per l'applicazione web (controller)

Un esempio di micro-stack frameworks è Flask, basato, come Django, su python.

Tuttavia, entrambe le tipologie di web framework, aggiungono allo stack software un web server di sviluppo integrato, da impiegare esclusivamente e tassativamente *solo* per lo sviluppo (manca completamente, ad esempio, di meccanismi per il controllo degli accessi o gestione del carico), ma è progettato per sollevare lo sviluppatore dal dover necessariamente configurare l'infrastruttura in grado di elaborare le richieste HTTP (es. Tomcat o Server LAMP).

7.3 Principi di progettazione e scopi

Tutti i web framework, a prescindere dalla tipologia (full o micro stack), condividono alcuni principi di design e finalità, riportate brevemente di seguito:

Convention over Configuration : principio che favorisce l'adozione di un insieme di "convenzioni" (di progettazione e implementazione) piuttosto che la configurazione nel dettaglio dell'applicazione. Un esempio di applicazione di tale principio è quello che impone, in framework quali Ruby on Rails, che ci sia una corrispondenza nei nomi tra classi di modello e relativi controller: ad una classe di modello denominata `ExampleModel` dovrà necessariamente corrispondere un relativo `ExampleModelController`. Tale associazione, dunque, non è configurabile, ma è imposta dal framework. Se questo aspetto, da un lato, limita la "flessibilità" nella scelta dei nomi, dall'altro solleva lo sviluppatore dal dover configurare l'associazione tra modello e controller, riducendo drasticamente i tempi e l'overhead di implementazione.

DRY (Don't Repeat Yourself) : si tratta di un principio generale che sconsiglia la duplicazione dell'informazione e insiste nel mantenere ogni parte dell'informazione in un unico punto del sistema.

Accoppiamento lasco I diversi livelli del framework non dovrebbero avere conoscenza approfondita l'uno dell'altro se non nei casi in cui questo sia assolutamente necessario.

Minor quantità di codice In particolare, non dovrebbe essere necessario sviluppare codice per le parti scontate e ripetitive. Lo sviluppo delle applicazioni (web) deve (o dovrebbe) richiedere meno codice possibile, dato che la maggior parte del codice è (o dovrebbe essere) a carico del framework.

---

documenti e ipertesti (risorse) nella letteratura meno precisa può capitare di veder etichettate delle semplici funzionalità o servizi web come RESTful, anche se non sempre soddisfano i vincoli di cui sopra. Naturalmente voi siete invitati, come sempre, ad assumere un linguaggio tecnico preciso. Maggiori dettagli a partire dalla voce REST dell'MDN Web Docs Glossary: Definitions of ldots `https://developer.mozilla.org/en-US/docs/Glossa`

Sviluppo veloce In un certo senso questo è il punto principale che spiega l'esistenza dei web frameworks, ovvero di velocizzare lo sviluppo su web, in particolare evitando o riducendo il più possibile gli aspetti più noiosi e ripetitivi, assicurando meccanismi di gestione già sistematizzati e testati.

7.4 Un esempio: Django

Django[7] è un Web framework basato sul linguaggio Python, un linguaggio di programmazione ad un alto livello di astrazione che deve molto del suo successo ad una curva di apprendimento notevolmente bassa, soprattutto considerando le caratteristiche del linguaggio. È un linguaggio multi-paradigma, che si adatta sia alla programmazione procedurale che alla programmazione ad oggetti, ed è multi-piattaforma, essendo supportato indistintamente Windows, Linux, MacOSX o Unix in generale. Caratteristica del linguaggio che lo distingue da altri linguaggi quali Java o C++, riguarda l'uso della cosiddetta *off-side rule* per la definizione dei blocchi e degli *scope* nel codice: i blocchi di codice sono identificati attraverso l'indentazione senza uso delle usuali parentesi graffe, né della punteggiatura (punto e virgola) per segnalare la fine di un'istruzione o di un blocco di istruzioni.

Python è utilizzato in progetti di notevole importanza da diverese aziende, tra cui Google, Amazon, IBM, NASA, Sun Microsystems, Microsoft. Pur meno efficiente di Java o C, dispone di un gran numero di librerie per il calcolo scientifico.

Una delle caratteristiche più interessanti di Python è che pur essendo *tipato dinamicamente*, nel senso che non occorre dichiare un tipo da associare ad ogni variabile, visto che il tipo viene inferito automaticamente dalle operazioni in cui la variabile si trova coinvolta, esso è tuttavia *fortemente tipato*, nel senso che una volta che il tipo di una variabile è stato inferito, esso non può più venir variato se non attraverso un'operazione di cast esplicito.

Django è quindi basato su Python. Le applicazioni vengono costruite in modo altamente modulare cosí da poter riutilizzare moduli già sviluppati in altri progetti. Offre inoltre supporto per una serie di necessità tipiche dello sviluppo di applicazioni web: ORM, interfaccia utente per l'amministrazione, gestione utenti, controllo degli accessi, viste, forms, configurazione degli URL, templates, supporto all'internazionalizzazione, geo-referenziazione e altro ancora. Per maggiori dettagli e approfondimenti sul web framework, si rimanda alle slide `http://goo.gl/ckWoHV` e alla documentazione ufficiale disponibile sul sito del progetto (`http://www.djangoproject.com`).

Django è basato su una variante del pattern MVC, chiamata *Model-view-template* (MVT). In questo modello, i controllori sono chiamati viste e le viste templates: ovviamente questo rischia di creare confusione e bisogna quindi farvi attenzione.

Un altro aspetto interessante è come Django realizza il *routing* ovvero come mette in corrispondenza l'URL della richiesta HTTP con il codice che va messo in esecuzione per produrre la risposta HTTP. In generale, una possibilità potrebbe essere di associare con un mapping 1:1 ad ogni URL dell'applicazione una funzione che esegua le necessarie operazioni. Questo tuttavia è un approccio troppo semplicistico, perché non prevede che la richiesta possa contenere parametri.

La gestione del routing di Django prevede quindi l'uso di espressioni regolari per associare una funzione, ed eventualmente dei parametri, ad ogni URL. Ad esempio, tutti gli URL che corrispondono all'espressione regolare

```
^/users/(?P<id>\d+)/$
```

mettono in esecuzione la funzione `display_user(id)`, dove `id` è esattamente il parametro previsto dall'espressione regolare. In questo modo, possiamo fare in modo che un URL `/users/<some`

[7] `http://www.djangoproject.com`

number>/ attivi una funzione display_user. Le espressioni regolari possono essere complicate a piacere e prevedono parole-chiave e parametri di posizione.

Una volta che ha risolto il problema di associare all'URL della richiesta la funzione adatta a generare la risposta, Django ci deve dare la possibilità di progettare la risposta in modo che sia facile e non troppo vincolato. Django usa i template, ovvero dei pezzi di testo con la possibilità di inserire dei placeholder nelle posizioni in cui andranno inseriti i valori delle variabili. Per chi conosce python, è molto simile al funzionamento di str.format(). In più, Django offre anche un minimo di supporto per la programmazione all'interno dei template, mediante l'uso di espressioni, che permettono di eseguire semplici operazioni, quali scorrere i valori di una lista è stata passata al template.

Per quel che riguarda la parte di modello, Django, come molti altri Web Frameworks, propone l'uso di ORM in grado prima di tutto di specificare i tipi dei campi da memorizzare. In questo modo è possibile applicare una validazione dell'informazione introdotta a livello di campo. Ad esempio, potrebbero essere permessi, per un particolare campo, esclusivamente indirizzi di posta elettronica. Si possono inoltre specificare la dimensione massima, dei valori di default, una lista di opzioni, del testo di documentazione, etichette per le form, etc.. A livello di programmazione, non vi è alcuna dipendenza dalla base di dati utilizzata, perché queste dipendenze sono raggruppate a livello di impostazioni, in modo da poter essere modificate senza toccare il codice.

8. WebUML

Analisi e progettazione di applicazioni Web; estensione di UML per il Web di Conallen.

8.1 Web engineering

La progettazione di sistemi basati su web ha una serie di peculiarità da un punto di vista dell'ingegneria del software.

- i requisiti sono altamente instabili;
- è richiesta molta più interazione con i committenti;
- Lo sviluppo è caratterizzato da forti pressioni sui tempi, con conseguente compressione dello scheduling;
- le tecnologie cambiano con frequenza molto maggiore che in ambito desktop;
- non si riesce a sfruttare appieno il paradigma orientato agli oggetti;
- sviluppati da team altamente eterogenei, con differenti competenze;
- non esistono strumenti di sviluppo solidi come quelli in ambito desktop.

8.2 Web Engineering: estensione di UML per il Web di Conallen

Jim Conallen, "Modeling business logic in Web-specific components can be done in a coherent and consistent way. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.36.7083` Modeling Web Application Architectures.

Molto ad alto livello, i sistemi basati su web non sono altro che un tipo particolare di sistemi client-server, in cui quindi gli elementi principali sono client, server e rete di connessione.

La maggior parter dei sistemi basati su web considerano utenti anonimi, che non hanno alcuna abilità o competenza informatiche. Risulta quindi essenziale un'interfaccia intuitiva e dal comportamento predicibile. Nella progettazione sono quindi molto importanti i *casi d'uso* (use cases).

Il linguaggio di modellazione UML è stato ideato per l'analisi e la progettazione di sistemi orientati agli oggetti e permette di rappresentare graficamente i sistemi o mediante *diagrammi*

strutturali, basati su classi, componenti, oggetti, o mediante *diagrammi comportamentali* per i casi d'uso, sequenze, collaborazioni, statechart e attività.

8.2.1 Peculiarità dei sistemi basati su web

Un aspetto importante di UML è che prevede dei meccanismi per estenderlo, nel caso si presenti la necessità di modellare al corretto livello di astrazione e di dettaglio nuovi sistemi, come appunto nel caso dei sistemi basati su web, per i quali la strutturazione in classi o comunque di tipo orientato agli oggetti non risulta la più corretta per una efficace rappresentazione della struttura del sistema.

Ovviamente questa struttura va usata con attenzione: estensioni introdotte con superficialità possono facilmente condurre a problemi di inconsistenza dei modelli. Inoltre, sembra opportuno mantenere la consistenza anche con la semantica del cuore del linguaggio e di possibili altre estensioni.

È importante sottolineare come un fattore chiave per ottenere una buona modellizzazione sia proprio la scelta del corretto livello di astrazione e dettaglio. Infatti, scegliere un livello di astrazione troppo alto o troppo basso conduce a modelli che risultano fuorvianti o perché omettono importanti dettagli o perché ne riportano troppi, facendo perdere di vista la struttura fondamentale del progetto.

Gli schemi architetturali di base per le applicazioni web usano le *pagine web* come elemento di base. Poiché uno degli obiettivi nella fase di design è catturare tutte le componenti del sistema, risulta cruciale catturare le pagine web come elementi principali e utilizzarle insieme alle classi e alle altre componenti del sistema.

Le *pagine client* sono artefatti che possono essere considerati come ogni altra interfaccia utente in un sistema. Tuttavia, dobbiamo porci il seguente problema di modellizzazione: come modellare le *pagine server*, ovvero le pagine web che hanno script che devono essere elaborati lato server? Questo tipo di pagina interagisce con le risorse lato server prima di essere inviata al client come interfaccia utente completa. Non essendo un approccio specificatamente orientato agli oggetti, UML non riesce a modellare bene questo importante aspetto dei sistemi web.

La notazione di UML così come vi è stata presentata non è sufficiente a descrivere le pagine server come oggetti nel class diagram, e a farle coesistere con le altre classi e oggetti del sistema. L'unica soluzione è quindi quella di usare gli strumenti previsti da UML e costruire un'esensione che permetta di includere questi aspetti dei sistemi web.

Tecnicamente, le estensioni sono rese possibili attraverso un meccanismo che mette assieme:

- stereotipi;
- tagged values;
- vincoli o contraints impliciti che modificano la semantica.

Quello degli *stereotipi* è un meccanismo di estensione che consente di classificare (marcare) i model element di UML. Sia S uno stereotipo del modello M. S conserva tutte le caratteristiche di M, ma può inoltre soddisfare dei vincoli addizionali e avere dei nuovi tagged values. S può quindi semplicemente essere usato per definire una differenza nella semantica o nel modo d'uso del modello M. La notazione standard prevede che si aggiunga $<< S >>$ alla rappresentazione visuale di M viene fornita una nuova rappresentazione utilizzando una nuova icona o una differente disposizione degli elementi di M.

I *tagged values*, invece, rappresentano un'estensione delle proprietà associate ad un elemento del modello. Per esempio le classi hanno un nome, una visibilità, e altri attributi assiociati. Un tagged value corrisponde alla definizione di una nuova proprietà che può essere associata ad un elemento del modello. È costituita da una coppia che permette di aggiungere informazioni arbitrarie ad ogni elemento del modello (Tag, Value).

I vincoli, infine, rappresentano un'estensione della semantica del linguaggio. Si tratta di un ruolo attribuito ad un elemento del modello che restringe la sua semantica. Consentono di rappresentare fenomeni che altrimenti non potrebbero essere espressi con UML.

Un vincolo è un modo per definire come i modelli possono essere messi insieme. Specifica le condizioni sotto le quali il modello può essere considerato ben formato. Possono essere espresse con una stringa informale tra una coppia di parentesi graffe, come mostrato in Figure 8.1, oppure usando un linguaggio formale come OCL.

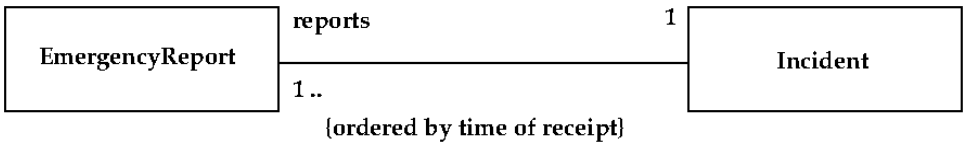


Figura 8.1: Esempio di vincolo

8.2.2 L'estensione di Conallen

Come abbiamo visto, usare UML per modellare un sistema web richiede di introdurre il concetto di pagina. Per risolvere questo punto, l'estensione di UML proposta da Conallen propone di introdurre due stereotipi: la pagina server e la pagina client. La prima contiene metodi e variabili relativi allo scripting server-side, e quindi CGI, JSP, ecc; la seconda, invece, contiene elementi relativi alla formattazione, allo scripting client-side, Applet, ActiveX, JavaScript, ecc.

Ai modelli usualmente usati dagli altri tipi di sistemi software (tra cui, ad esempio, il modello degli use case, modello di implementazione, etc.), introduciamo anche la **site map**, un'astrazione dell'insieme delle pagine Web e dei collegamenti di navigazione all'interno del sistema.

Occorre modellare le **pagine**, i collegamenti (**link**) tra di esse, il **contenuto dinamico** necessario alla loro costruzione e il contenuto dinamico che arriva fino al client.

Si noti che ci proponiamo di modellare la **business logic** e non la **presentation logic**: non rappresenteremo quindi tutti i singoli elementi visuali di interazione con l'utente, quali ad esempio i bottoni.

UML si basa sulla modellizzazione in classi, e quindi non può essere utilizzato cosí com'è: una pagina Web assomiglia più ad un componente che ad una vera e propria classe.

Occorre ora far corrispondere ciascuno di questi quattro elementi dell'applicazione ad un diverso elemento del modello.

Hyperlinks: associazioni tra elementi.

Pagine: classi (con tre aree: nome, attributi e operazioni).

Scripts: operazioni nella classe.

Variabili definite con scope di pagina: attributi della classe.

Ma cosa facciamo se, come succede di solito, una pagina contiene sia scripts lato server (ad esempio, JSP) che lato client (ad esempio, applet o JavaScript)?

UML può venir esteso tramite l'uso di **stereotipi** che ci permettono di associare un nuovo significato ad un elemento del modello, **tagged values**, ovvero coppie chiave-valore da associare ad elementi del modello e **constraints** che permettono di definire delle regole necessarie a che il modello sia ben formato.

Nell'estensione di UML di Conallen, ogni pagina Web, sia essa statica o dinamica, viene modellata come un componente. La Implementation View o Component View descrive i componenti del sistema e le relazioni che intercorrosno tra loro. Corrispondono quindi alle pagine Web e ai relativi collegamenti, dando origine a una mappa del sito (**site map**).

Il comportamento di una pagina Web sul server è completamente diverso che sul client: può accedere alle risorse del server (basi di dati, file system, ...) nel primo caso; al browser e alle sue funzionalità nel secondo.

Ciascuno di questi due aspetti di ogni pagina verrà quindi modellato separatamente, e la relazione tra le due viene stereotipata come **build**, nel senso che la pagina lato server "costruisce" quella lato client.

Ovviamente mentre ogni pagina lato client può venir costruita da al più una pagina lato server, una pagina lato server può costruire anche più pagine lato client.

In un'applicazione Web, un link permette di navigare da una pagina all'altra. Nel modello questa relazione viene modellata da una associazione stereotipata di tipo **link**. Un'associazione di questo tipo ha sempre origine da una pagina lato client e punta ad un'altra pagina che può essere sia lato client (ad esempio, pagina statica) che lato server (ad esempio, una JSP, una servlet o una CGI).

I tagged values permettono di associare dei parametri ai link.

Nelle prossime sezioni i principali elementi di una site map vengono considerati nel dettaglio.

Server page

Si tratta di una pagina web contenente script eseguiti dal server, che interagiscono con risorse lato server come database, business logic, sistemi esterni. Le operazioni della classe rappresentano le funzioni dello script, gli attributi rappresentano le variabili visibili nello scope della pagina. È soggetta all'unico vincolo di avere relazioni soltanto con oggetti sul server. I tagged values di una pagina server descrivono lo scripting engine, ovvero il linguaggio o il motore che dovrebbe essere utilizzato per eseguire o interpretare questa pagina (J2EE, .NET, PHP, etc...).

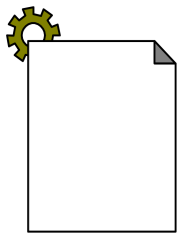


Figura 8.2: Simbolo grafico di una server page.

Client page

Si tratta di una pagina direttamente rappresentabile dallo user agent, quale ad esempio il browser. Può contenere script che sono interpretati dallo user agent. Le operazioni corrispondono alle funzioni degli script, mentre gli attributi corrispondono alle variabili. Una client page può avere associazioni con altre pagine, client o server. Non è soggetta a nessun vincolo e ammette i seguenti' tagged values:

Titolo-tag : il titolo della pagina è mostrato dal browser

Base-tag : URL di base per dereferenziare relativi URLs

Body-tag : l'insieme degli attributi per l'elemento <body> che impostano il background e gli attributi di testo di default.

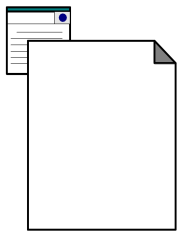


Figura 8.3: Simbolo grafico di una client page.

Partizionamento lato client e lato server

Questa attività richiede di individuare le server page e le client page e di stabilire le relazioni tra loro e con gli altri oggetti del sistema. Dipende dal tipo di architettura, ad esempio Thin Web Client, Fat Web Client o Web Delivery. Le due attività principali nel design di applicazioni Web sono significativamente differenti dal design di altri sistemi software: Partizionamento degli oggetti lato client e lato server e definizione delle pagine web come interfacce utenti. Quando si utilizzano le architetture di tipo Thin Client e Fat Client, nei primi passi possono essere partizionati un gran numero di oggetti individuati durante la fase di analisi.

Per le applicazioni che usano un'architettura Thin web client tutti gli oggetti sono lato server (running on the web server or another tier associated with the server) Per le applicazioni che usano un'architettura Fat web client, in gran parte, oggetti persistenti, oggetti contenitori, oggetti condivisi, e oggetti complessi appartengono al lato server. Inoltre, oggetti con associazioni a tali risorse lato server come database e sistemi legacy, appartengono al lato server. Oggetti candidati per il lato client sono gli oggetti che non hanno associazioni con risorse o oggetti lato server, ovvero che hanno dipendenze solo con risorse lato client Inoltre, oggetti che contengono campi da validare, controlli per le interfacce utenti, controlli per la navigazione, etc,

Link

Tale stereotipo è definito per un'associazione tra pagine client e altre pagine (server o client): si noti però che anche se il link collega due pagine client, il server viene sempre coinvolto perché il link presuppone una richiesta HTTP al server: semplicemente non abbiamo una pagina server perché la pagina viene restituita senza elaborazioni. Un'associazione di tipo link può essere sia unidirezionale che bidirezionale, nel caso ci siano due link che collegano le due pagine nelle due direzioni. Corrisponde ad un'associazione UML e non viene rappresentato da alcuna icona. Rappresenta un puntatore tra una client page e un'altra pagina. Ha un unico tagged value per i parametri: una lista di nomi di parametri che dovrebbero essere passati insieme al momento della richiesta per la pagina destinazione del link.

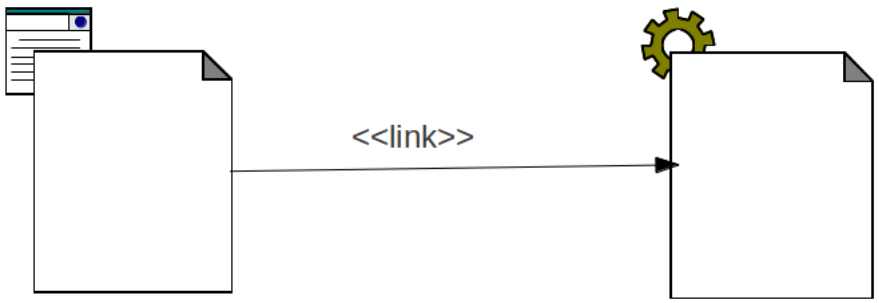


Figura 8.4: Esempio di link.

Un'altra importante associazione è quella che collega una pagina server con la pagina client che essa produce in uscita (build). In questo caso. La relazione, indicata dallo stereotipo builds è unidirezionale è esemplificata in Figura 8.5. Si tratta di una relazione direzionale, visto che la cliente page non contiene informazioni relative al perché è stata creata; inoltre, una server page può creare più client page, che invece sono create da una sola server page.

Redirect

Un'altra importante associazione è rappresentata dalla redirect, che indica il trasferimento del controllo da una pagina ad un'altra. In questo caso sia la sorgente che la destinazione possono

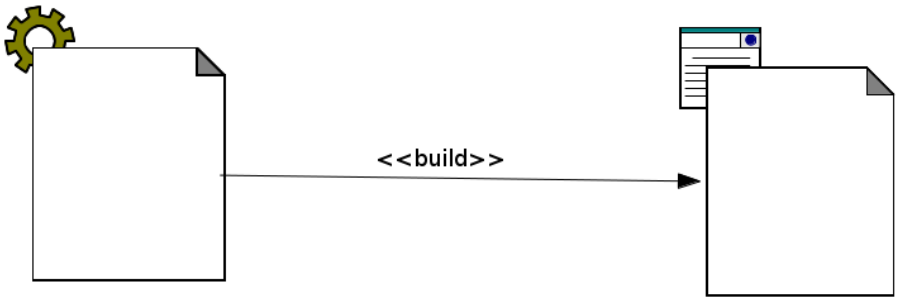


Figura 8.5: Esempio di associazione di tipo build.

essere client pages o server pages. Se l'origine è una client page, la pagina destinazione verrà automaticamente richiesta dal browser, allo scadere di un determinato intervallo di tempo. Non ha nessun vincolo. Ha come tagged value il delay che dà l'ammontare di tempo che la client page dovrebbe aspettare prima del redirect alla prossima pagina.

Esempi

Progettazione di una singola pagina web

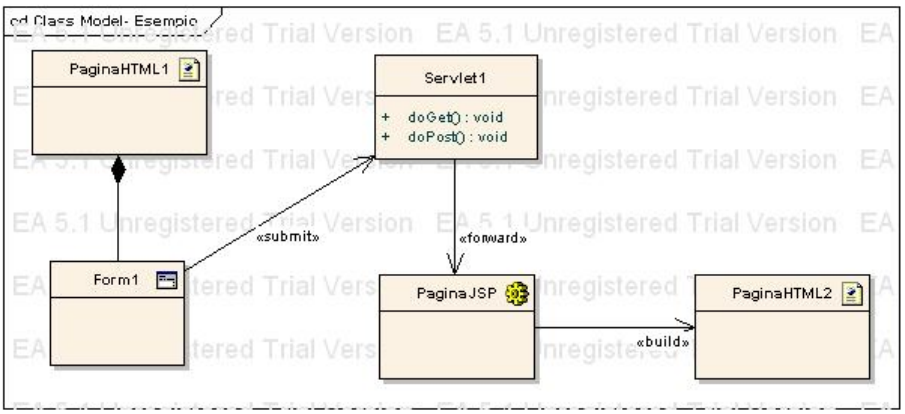


Figura 8.6: Esempio di site map.

Progetto delle singole pagine web

Una volta che sono state identificate le Web page, le principali interazioni tra oggetti e web page, e le responsabilità di ogni componente, ha inizio la definizione di ogni singola pagina. Per le architetture Thin web client, questa operazione si limita alle operazioni e gli attributi delle pagine server.

Le *form* rappresentano una componente caratteristiche di moltissime pagine client e quindi bisogna chiedersi come rappresentarle. Contengono attributi addizionali rispetto alle normali pagine. È possibile che una pagina contenga più form. Un form può essere rappresentato come

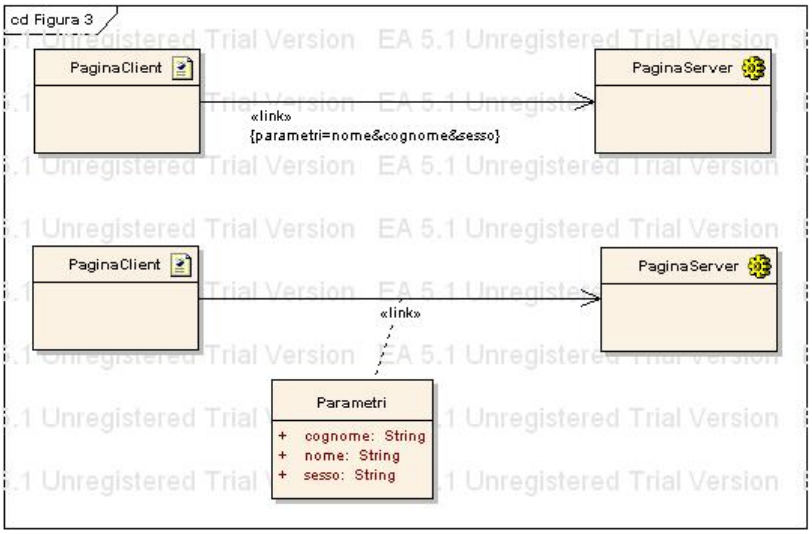


Figura 8.7: Esempio di site map.

una classe i cui attributi sono i campi del form. La relazione tra pagina e form è un'aggregazione. Lo stereotipo della relazione tra il form e la pagina server che ne utilizza i dati è un submit. In Figura 8.9 si può vedere un esempio di utilizzo di form. Si tratta di una serie di campi di input che sono parte di una pagina client. Corrisponde all'etichetta HTML <form>. Gli attributi di questa classe sono rappresentati dai campi di input (input box, text area, radio buttons, check boxes, campi hidden) dell'elemento. Un form non possiede operazioni. Nessun vincolo. Il metodo (ad esempio, GET o POST) usato per sottomettere i dati nella action URL viene rappresentato come tagged value.

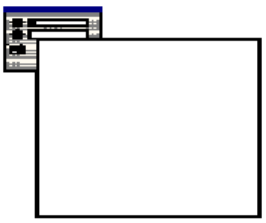


Figura 8.8: Icona di una pagina client contenente form.

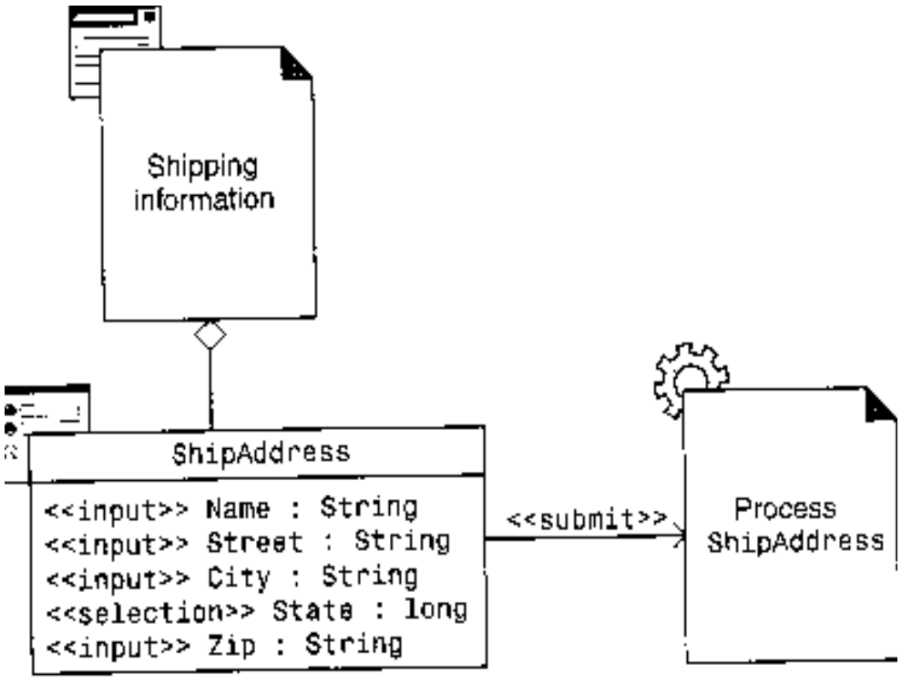


Figura 8.9: Esempio di site map con uso di form.

Submit

Si tratta di un'associazione che associa un form ad una server page, alla quale il form invia i dati. Poi sono elaborate le pagine server che utilizzano le informazioni nel form sottoposto. Non ha nessun vincolo, ma un tagged value per i parametri: una lista di nomi di parametri che dovrebbero essere passati con la richiesta della pagina destinazione dell'associazione.

Frameset

Viene rappresentata come una classe contenitore di pagine web multiple. L'area viene divisa in frame, ed ogni frame viene associato ad al più un <<target>> (anche nessuno). Il contenuto di ciascun frame può essere una web page o un altro frameset. Essendo una client page, può avere attributi e operazioni, che sono definiti in maniera analoga Nessun vincolo. Tagged value:
Rows : il valore dell'attributo della riga del tag <<frameset>>.
Cols : il valore dell'attributo della colonna del tag <<frameset>>. Icona:

Logiche lato client

Progettare applicazioni web che hanno pagine client dinamiche richiede attenzione nel partizionamento degli oggetti. Abilità di fornire funzionalità (come: vista di una shopping cart, cambiare i valori visualizzati, ricalcolare il totale dovuto) senza dover rimandare al server l'elaborazione. Questi tipi di funzionalità possono essere realizzati tramite: ClientScript (e.g., JavaScript), o ClientObject (e.g., ActiveX control, Java applet) . Utilizziamo gli stereotipi per modellare tali componenti.

I ClientScript Object (ad esempio, file .js) vengono rappresentati da una classe. La semantica di questo stereotipo è una collezione di scripts client-side che esistono in file e sono incluse in richieste da parte di un client browser. Questi oggetti sono spesso accomunati da funzioni usate per le applicazioni. Non hanno né vincoli né tagged values.

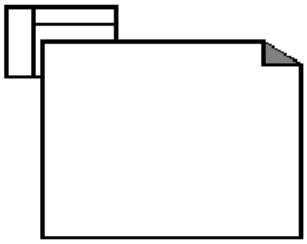


Figura 8.10: Icona di una pagina client contenente un frameset.

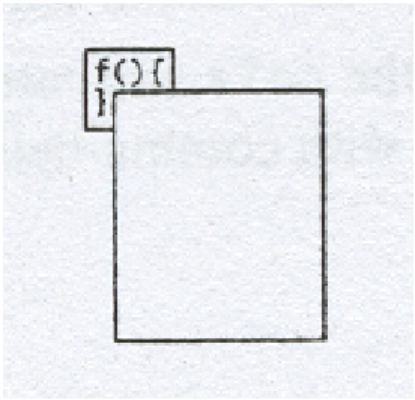


Figura 8.11: Icona di una pagina client contenente un oggetto di scripting lato client.

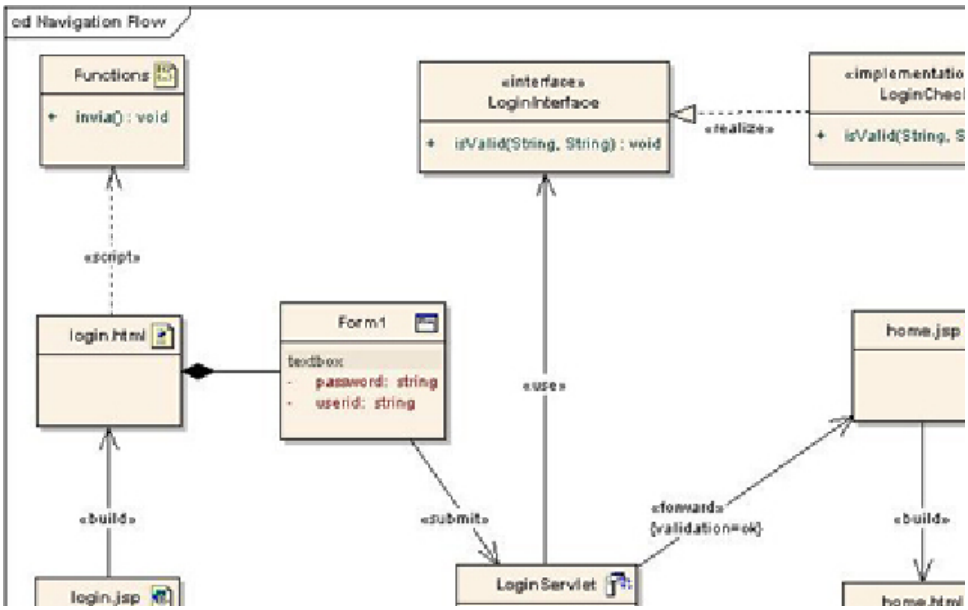


Figura 8.12: Esempi.

9. JSP e Struts 2

Java Server pages: ancora programmazione in Java lato server; etichette personalizzate e JavaBeans; il Web Framework Struts. **Riferimenti per i javabeans**: dal sito SUN.

9.1 JavaServer Pages (JSP)

Insieme alle servlet, le JSP rappresentano il principale strumento per sviluppare applicazioni web in Java. Il principale problema delle servlet è che la preparazione di una pagina da mandare al client richiede istruzioni intricate in cui il formato utilizzato per la presentazione della pagina (ad esempio HTML, ma non solo) risulta immerso in istruzioni di stampa. Ecco un esempio:

```
PrintWriter out = response.getWriter();
out.println(\"<html><head><title>Testing</title></head>\");
out.println(\"<body style=\"background:#ffdddd\">");
```

Questo non solo aumenta il rischio di introdurre errori, ma richiede di ricompilare la servlet per ogni modifica anche banale. Da questa constatazione sono nate le JSP, che ci permettono di implementare pagine in cui le istruzioni Java e il template statico (ad esempio in HTML), ancora convivono, ma sono organizzati in un modo più chiaro e quindi più leggibile e più facile da manutenere. La compilazione avviene in modo automatico la prima volta che arriva una richiesta per la pagina e ogni volta che la pagina viene modificata, ma non occorre ricompilare se le modifiche riguardano solo il template statico.

Una pagina JSP è una pagina di testo formata da:

- un template statico, espresso in un qualsiasi formato testuale (HTML, SVG, WML, XML, ...)
- istruzioni Java

```
<p>The time in seconds is:
<\%= System.currentTimeMillis()/1000 =\%> </p>
```

Estensione *.jsp* per indicare al server il tipo di elaborazione richiesta. La prima volta che una pagina JSP viene richiamata, essa viene *tradotta* in una servlet. A questo punto, come una qualsiasi

servlet, essa viene messa in esecuzione dal *servlet engine*: questo carica la classe con un *class loader*, e la esegue.

Viene prodotta una pagina Web da mandare al browser per essere visualizzata. La servlet può creare oggetti necessari alla computazione e scrivere le corrispondenti stringhe su uno stream di uscita che passa alla risposta HTTP.

Si possono avere due tipi di errore:

- errori durante la **traduzione** (translation time)
- errori durante l'elaborazione della **richiesta** (request time)

Negli esempi di questa lezione considereremo JSP con un documento XHTML come template statico. Una pagina JSP ha in più alcuni elementi:

- **direttive**: `<%@ directive %>`
- **dichiarazioni**: `<%! declaration %>`
- **scriptlet**: `<% scriptlet %>`
- **espressioni**: `<%= expression %>`
- **librerie di etichette**

Il resto della pagina è detto di solito *fixed-template data* o *fixed-template text*: si usano le servlet se questa parte fissa è poca, le JSP in caso contrario.

Ad esempio, all'interno di una pagina Web, posso introdurre la linea:

```
<\%= new java.util.Date() \%>
```

per inserire informazioni su data e ora.

Quando il client richiede la pagina .jsp contenente questa espressione, il server manda in esecuzione una servlet che crea un oggetto di tipo *Date* dal pacchetto *java.util* e stampa la stringa corrispondente nella risposta da passare al client. Il meccanismo delle JSP converte in stringa il risultato di ogni espressione JSP, in modo da poterlo includere nella risposta che viene passata al client.

In questo caso, sarà anche opportuno inserire nella *head* della pagina Web un elemento *meta* che la rinfreschi spesso (ogni 60 secondi):

```
<meta http-equiv = "refresh" content = "60" />
```

9.1.1 Direttive

```
<\%@ directive {attr="value"}* \%>
```

Messaggi al meccanismo JSP per:

- specificare le impostazioni della pagina (ad esempio, se è una pagina di errore): **page**; ve ne possono essere diverse occorrenze, ma con una sola occorrenza per attributo, eccetto l'import (attributi: language, extends, import, session, buffer, autoFlush, isThreadSafe, info, errorPage, isErrorPage, contentType); Ad esempio:

  ```
  <\%@ page language="java" \%>
  <\%@ page session="true" import="java.util.*" \%>
  ```

- includere contenuti da altre risorse: **include**; tali contenuti verranno poi tradotti come fossero originariamente nelle JSP; unico attributo: file;
- specificare librerie di etichette personalizzate (*custom tag libraries*): **taglib**; attributi: uri (del file che descrittore della libreria) e tagPrefix (esclusi: jsp, jspx, java, javax, servlet, sun, sunw)

Le direttive, `<%@ ... %>`, vengono elaborate durante la traduzione in servlet.

Ad esempio:

```
<\%@ include file = "banner.html" \%>
<\%@ include file = "clock.jsp" \%>
<\%@ taglib uri = "advjhtp1-taglib.tld" prefix = "advjhtp1" \%>
```

Quindi, le direttive non producono nessun output immediato, dal momento che vengono elaborate a prescindere dalla singola richiesta

9.1.2 Elementi di scripting

Gli **elementi di scripting** possono essere usati non solo per produrre una pagina dinamica, ma anche per produrre pagine statiche da restituire solo se sono soddisfatti determinati requisiti. Sono componenti di scripting:

scriptlets : <% ... %>; le istruzioni contenute in uno scriptlet vengono messe in esecuzione durante l'elaborazione della richiesta HTTP;

commenti :

- commenti **JSP**: <%- ...-%>, ovunque, ma non all'interno di uno scriptlet;
- commenti **XHTML**: <!- ... ->, idem
- commenti del **linguaggio di scripting**: in questo momento solo Java.

I commenti XHTML sono gli unici che vengono riportati nella risposta che giunge al client.

espressioni : <%= ... %>, contenente un'espressione Java, che il contenitore converte sempre in un oggetto String;

dichiarazioni : <%! ... %>, per la definizione di variabili e metodi, che diventano membri della classe ottenuta dalla traduzione della JSP;

sequenze di escape : <%, %>, ', '', \

9.1.3 Oggetti impliciti

- scope di **applicazione**
  - application: il contenitore all'interno del quale viene eseguita la servlet;
- scope di **pagina**
  - config
  - exception: solo in una pagina di errore
  - out
  - page
  - pageContext
  - response
- scope di **richiesta**
  - request
- scope di **sessione**
  - session

```
<\%-- questo e' un commento che non apparira' nella pagina che
viene inoltrata al client --\%>

<\%-- La seguente e' una direttiva --\%>

<\%@ page import="java.util.*" \%>

<\%-- Alcune dichiarazioni: --\%>

<\%!
  long loadTime= System.currentTimeMillis();
```

```
   Date loadDate = new Date();
   int hitCount = 0; ## variabili d'istanza, non locali al metodo!
\%>



<\%-- ecco un esempio di espressioni --\%>

<H1>This page was loaded at <\%= loadDate \%> </H1>

<H1>Hello, world! It's <\%= new Date() \%></H1>

<H2>Here's an object: <\%= new Object() \%></H2>

<H2>This page has been up
<\%= (System.currentTimeMillis()-loadTime)/1000 \%> seconds</H2>

<H3>Page has been accessed <\%= ++hitCount \%> times since <\%= loadDate \%></H3>
<\%-- A "scriptlet" that writes to the server console and to the client page.
     Note that the ';' is required: --\%>

<\%
System.out.println("Goodbye"); \## sullo stdout del server
out.println("Cheerio"); ## sulla risposta!
\%>
```

Cosí come per le servlet, le informazioni riguardo lo stato del client non possono essere salvate nelle variabili di istanza, ma nell'oggetto implicito **session**.

I metodi e le variabili dichiarate nelle dichiarazioni JSP vengono inizializzate nel momento in cui la JSP viene inizializzata e possono essere usate da tutti gli scriptlet e le espressioni all'interno di quella JSP. Le variabili dichiarate in questo modo divengono variabili di istanza della classe servlet ottenuta dalla traduzione.

Ricordarsi il punto e virgola alla fine delle dichiarazioni.

Mettiamo a confronto una servlet e una JSP che fanno la stessa cosa

```
//:! c15:jsp:DisplayFormData.jsp
<\%-- Fetching the data from an HTML form. --\%>
<\%-- This JSP also generates the form. --\%>
<\%@ page import="java.util.*" \%>


    <H1>DisplayFormData</H1><H3>
<\%
   Enumeration flds = request.getParameterNames();
   if(!flds.hasMoreElements()) { // No fields \%>
<form method="POST"
      action="DisplayFormData.jsp">
<\% for(int i = 0; i < 10; i++) { \%>
   Field<\%=i\%>: <input type="text" size="20"
      name="Field<\%=i\%>" value="Value<\%=i\%>">
<\% } \%>
```

```
   <INPUT TYPE=submit name=submit
         value="Submit"></form>
<\%} else {
   while(flds.hasMoreElements()) {
      String field = (String)flds.nextElement();
      String value = request.getParameter(field);
\%>
   \item <\%= field \%> = <\%= value \%>
<\% }
} \%>
    </H3>
///:~
```

e

```
//: c15:servlets:EchoForm.java
// Dumps the name-value pairs of any HTML form
// {Depends: j2ee.jar}
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
   public void service(HttpServletRequest req,
                       HttpServletResponse res) throws IOException {
      res.setContentType("text/html");
      PrintWriter out = res.getWriter();
      Enumeration flds = req.getParameterNames();
      if(!flds.hasMoreElements()) {
         // No form submitted -- create one:
         out.print("");
         out.print("<form method=\"POST\"" + " action=\"EchoForm\">");
         for(int i = 0; i < 10; i++)
             out.print("{\bf Field" + i + "} " + "<input type=\"text\""+
                 " size=\"20\" name=\"Field" + i +
                 "\" value=\"Value" + i + "\">");
             out.print("<INPUT TYPE=submit name=submit"+
                 " Value=\"Submit\"></form>");
         } else {
            out.print("\section{Your form contained:}");
            while(flds.hasMoreElements()) {
               String field= (String)flds.nextElement();
               String value= req.getParameter(field);
               out.print(field + " = " + value+ "");
         }
    }
    out.close(); }
}
```

9.2 Discussione

Con questa organizzazione le JSP presentano indubbi vantaggli rispetto alle intricate istruzioni di stampa che abbiamo discusso all'inizio di questa lezione, ma continuano a presentare dei problemi:

1. avere il codice Java e il template statico frammisto nello stesso file presenta problemi sia per la leggibilità che per la manutenzione;
2. per come sono organizzate non favoriscono il riuso del codice;
3. naturalmente è possibile inserire tutti i metodi Java in una JSP che verrà inclusa da tutte le JSP che devono usare uno o più di quei metodi: questo modo di programmare, tuttavia, si allontana dalla programmazione orientata agli oggetti; basti pensare, tra l'altro, che non si può usare l'ereditarietà;
4. gli ambienti di sviluppo (IDE) per Java sono pensati per analizzare il codice all'interno di una classe, non in una JSP: queste rende più difficile sia lo sviluppo che il debugging;
5. in fase di testing, è più facile verificare la business logic the è stata incapsulata in una classe;
6. il codice organizzato in classi è più facile da rimodulare.

Per cercare di ovviare, almeno in parte, a questi problemi, nelle versioni successive delle JSP sono stati studiati dei meccanismi basati sulle **azioni standard**: `<jsp:action>`:

`<jsp:include>` , dinamico, rispetto alla direttiva **include**; attributi
- page: parte della stessa applicazione Web
- flush: deve sempre essere true in JSP 1.1

`<jsp:forward>` : inoltra la richiesta a un'altra JSP, servlet o pagina statica; fa terminare l'esecuzione della JSP corrente. Unico attributo *page*.

`<jsp:plugin>` : permette di aggiungere ad una pagina un componente plug-in, sotto forma di un elemento HTML **object** o **embed** specifico di quel browser. Nel caso si tratti di un applet Java, l'azione provoca lo scaricamento del plug-in Java, se non è già installato sul client.

`<jsp:param>` : specifica i parametri per **include**, **forward** e **plugin**; ha due attributi:
- name (se il nome esiste già, il valore viene aggiornato;
- value

Tutti i valori di ogni parametro vengono ottenuti applicando il metodo **getParameterValues** sull'oggetto implicito **request**, e ottenendo come valore di ritorno, un array di stringhe;

`<jsp:useBean>` : specifica anche lo scope del bean e vi assegna un identificativo con cui potercisi riferire;

`<jsp:setProperty>` , in una istanza specificata di un JavaBean.

`<jsp:getProperty>` , come stringa.

del client.

9.3 JavaBeans

Per permettere una maggiore separazione tra la business logic rappresentata dal codice Java e il template statico, rappresentato di solito dal codice HTML, la versione JSP 1.0 permette di incapsulare il codice all'interno dei JavaBeans. In altre parole, dalla versione 1.0 le JSP offrono immediato supporto alla programmazione per componenti che si appoggia ai JavaBeans.

I JavaBeans sono **componenti**: classi Java pensate per essere facilmente riutilizzate e organizzate insieme per la costruzione di applicazioni. Ogni classe Java può essere un JavaBean, purché soddisfi alcuni requisiti (le convenzioni discusse di seguito). Le

Nel seguito vengono riportate (adattandole leggermente) parti di un tutorial disponibile da `java.sun.com/j2ee/tutorial/1_3-fcs/doc/JSPBeans.html`, adattato: tra le competenze di cui dovete impadronirvi è la capacità di documentarvi autonomamente, quindi cercate di studiare la documentazione originale.

JavaBeans Component Design Conventions

JavaBeans component design conventions govern the **properties** of the class, and govern the **public methods that give access to the properties**.

A JavaBeans component property can be:

1. Read/write, read-only, or write-only
2. Simple, which means it contains a single value, or indexed, which means it represents an array of values

There is no requirement that a property be implemented by an instance variable; the property must simply be accessible using public methods that conform to certain conventions:

For each readable property, the bean must have a method of the form `PropertyClass getProperty() ...` For each writable property, the bean must have a method of the form `setProperty(PropertyClass pc) ...` In addition to the property methods, a JavaBeans component must define a constructor that takes no parameters.

The JSP pages catalog.jsp, *showcart.jsp*, and *cashier.jsp* use the *util.Currency* JavaBeans component to format currency in a locale-sensitive manner. The bean has two writable properties, *locale* and *amount*, and one readable property, format. The format property does not correspond to any instance variable, but returns a function of the locale and amount properties.

```
public class Currency {
   private Locale locale;
   private double amount;

   public Currency() {
      locale = null;
      amount = 0.0;
   }

   public void setLocale(Locale l) {
      locale = l;
   }

   public void setAmount(double a) {
      amount = a;
   }

   public String getFormat() {
      NumberFormat nf =
         NumberFormat.getCurrencyInstance(locale);
      return nf.format(amount);
   }
}
```

Why Use a JavaBeans Component?

A JSP page can create and use any type of Java programming language object within a declaration or scriptlet. The following scriptlet creates the bookstore shopping cart and stores it as a session attribute:

```
<\%
   ShoppingCart cart = (ShoppingCart)session.getAttribute("cart");
   // If the user has no cart, create a new one
   if (cart == null) {
```

```
      cart = new ShoppingCart();
      session.setAttribute("cart", cart);
   }
\%>
```

If the shopping cart object conforms to JavaBeans conventions, JSP pages can use JSP elements to create and access the object. For example, the Duke's Bookstore pages bookdetails.jsp, catalog.jsp, and showcart.jsp replace the scriptlet with the much more concise JSP useBean element:

```
<jsp:useBean id="cart" class="cart.ShoppingCart" scope="session" />
```

Creating and Using a JavaBeans Component

You declare that your JSP page will use a JavaBeans component using either one of the following formats:

```
<jsp:useBean id="beanName" class="fully\_qualified\_classname" scope="scope" />
```

or

```
<jsp:useBean id="beanName" class="fully\_qualified\_classname" scope="scope">
   <jsp:setProperty .../>
</jsp:useBean>
```

The second format is used when you want to include jsp:setProperty statements, described in the next section, for initializing bean properties.

The jsp:useBean element declares that the page will use a bean that is stored within and accessible from the specified scope, which can be application, session, request, or page. If no such bean exists, the statement creates the bean and stores it as an attribute of the scope object (see below). The value of the id attribute determines the name of the bean in the scope and the identifier used to reference the bean in other JSP elements and scriptlets.

I componenti Web possono condividere l'informazione utilizzando 4 oggetti che corrispondono a 4 scope (ambiti):

Scope	**Class**	**Accessible From**
Web context	javax.servlet.ServletContext	Contesto in cui vengono eseguiti i componenti Web: comprende i parametri di inizializzazione, le risorse associate al contesto Web, gli attributi, le utilità per il log.
session	javax.servlet.http.HttpSession	variabili di sessione
request	Subtype of javax.servlet.ServletRequest	l'oggetto corrispondente alla richiesta HTTP
page	javax.servlet.jsp.PageContext	Nelle JSP, la pagina in cui è stato creato l'oggetto.

The following element creates an instance of Currency if none exists, stores it as an attribute of the session object, and makes the bean available throughout the session by the identifier currency:

```
<jsp:useBean id="currency" class="util.Currency" scope="session"/>
```

There are two ways to set JavaBeans component properties in a JSP page: with the jsp:setProperty element or with a scriptlet

```
<\% beanName.setProp{\em Name}(value); \%>
```

The syntax of the jsp:setProperty element depends on the source of the property value. The following table summarizes the various ways to set a property of a JavaBeans component using the `jsp:setProperty` element.

Value Source	Element Syntax
String constant	`<jsp:setProperty name="beanName" property="propName" value="string constant"/>`
Request parameter	`<jsp:setProperty name="beanName" property="propName" param="paramName"/>`
Request parameter name matches bean property	`<jsp:setProperty name="beanName" property="propName"/>` `!<jsp:setProperty name="beanName" property="*"/>`
Expression	`<jsp:setProperty name="beanName" property="propName" value=<%= expression %>/>`

1. beanName must be the same as that specified for the id attribute in a useBean element.
2. There must be a setPropName method in the JavaBeans component.
3. paramName must be a request parameter name.

Retrieving JavaBeans Component Properties

There are several ways to retrieve JavaBeans component properties. Two of the methods (the jsp:getProperty element and an expression) convert the value of the property into a String and insert the value into the current implicit out object:

```
<jsp:getProperty name="beanName" property="propName"/>
<\%= beanName.getPropName() \%>
```

For both methods, beanName must be the same as that specified for the id attribute in a useBean element, and there must be a getPropName method in the JavaBeans component.

For both methods, beanName must be the same as that specified for the id attribute in a useBean element, and there must be a getPropName method in the JavaBeans component.

If you need to retrieve the value of a property without converting it and inserting it into the out object, you must use a scriptlet:

```
<\% Object o = beanName.getPropName(); \%>
```

Note the differences between the expression and the scriptlet: the expression has an = after the opening % and does not terminate with a semicolon, as does the scriptlet.

The Duke's Bookstore application demonstrates how to use both forms to retrieve the formatted currency from the currency bean and insert it into the page. For example, bookstore3/showcart.jsp uses the form

```
<jsp:getProperty name="currency" property="format"/>
```

whereas bookstore2/showcart.jsp uses the form

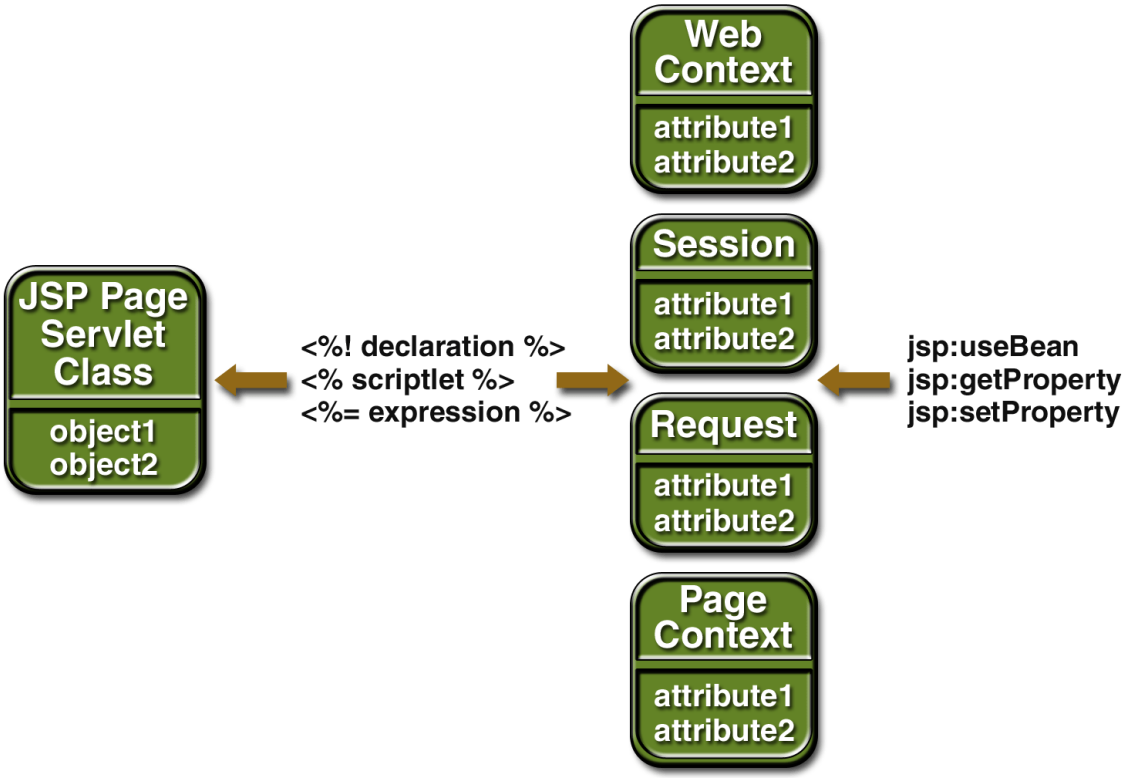
```
<\%= currency.getFormat() \%>
```

The Duke's Bookstore application page bookstore2/showcart.jsp uses the following scriptlet to retrieve the number of books from the shopping cart bean and open a conditional insertion of text into the output stream:

```
<\%
   // Print a summary of the shopping cart
   int num = cart.getNumberOfItems();
   if (num > 0) {
\%>
```

The following Figure summarizes where various types of objects are stored and how those objects can be accessed from a JSP page. Objects created by the jsp:useBean tag are stored as

attributes of the scope objects and can be accessed by jsp:[get|set]Property tags and in scriptlets and expressions. Objects created in declarations and scriptlets are stored as variables of the JSP page's servlet class and can be accessed in scriptlets and expressions.



9.4 Etichette personalizzate

Anche i JavaBeans, però, presentano dei problemi. Ad esempio, i nomi dei metodi devono seguire convenzioni piuttosto rigide, dando luogo a volte a nomi lunghi, complicati e alla fin fine poco chiari. Inoltre, per passare degli argomenti ai metodi occorre comunque ricorrere agli scriptlet. Un'ulteriore proposta di soluzione più flessibile dei JavaBeans è rappresentata dalle etichette personalizzate, introdotte nella versione 1.1.

Tag libraries: classi che implementano l'interfaccia **Tag**, e, di solito, estendono le classi **TagSupport** o **BodyTagSupport**.

In molti casi, le etichette personalizzate rappresentano un'alternativa all'uso dei *JavaBeans*, ma questi ultimi non possono manipolare il contenuto della pagina, e comunque il loro uso richiede una qualche conoscenza di Java.

Esempio: welcome senza parametri di ingresso

- Pagina JSP;
- Implementazione Java;
- Descrittore

Esempio: welcome con parametri di ingresso

- Pagina JSP;
- Implementazione Java;
- Il descrittore è lo stesso di prima.

Esempio: guestBook

- Pagina JSP;

- Implementazione Java;
- informazioni accessorie;
- descrittore come prima.

9.4.1 Discussione

Pur ovviando ad alcuni dei problemi che caratterizzando JSP e JavaBeans, le etichette personalizzate sono piuttosto complicate da implementare ed hanno un ciclo di vita abbastanza complesso. Versioni successive che hanno introdotto delle funzionalità comuni a tutte le etichette, le JavaServer Pages Standard Tag Libraries (JSTL), che permettono di gestire oggetti in ambiti di definizione diversi, di iterare su collection, di eseguire test condizionali, di fare parsing e formattazione dei dati, etc. Tuttavia nessuno di questi strumenti ha evitato che gli sviluppatori continuassero ad utilizzare scriptlets, che spesso rappresentano, almeno per applicazioni di dimensioni modeste, una soluzione più immediata da implementare, anche se questo espone a rischi se o quando l'applicazione cresce.

Data la difficoltà, nel lavoro in team, di controllare che nessuno dei componenti del team introducesse istruzioni Java all'interno degli scriptlets, la versione 2.0 delle JSP permette di disabilitarli. Inoltre questa versione ha introdotto un ciclo di vita più semplice e la possibilità di costruire le etichette in un file ad esse dedicato, rendendone più agevole la costruzione.

9.5 Architetture multi-tier per il Web

Un'**architettura** (software) descrive la **struttura** di un'applicazione, la decomposizione in **componenti**, le loro **interfacce** e **interazioni**. L'architettura rappresenta il passaggio tra analisi e implementazione. Le applicazioni Web sono un esempio di architettura **multi-tier** (multistrato) o **n-tier** Per **tier** si intende un raggruppamento logico di funzionalità. I diversi *tier* possono stare sulla stessa macchina o su macchine diverse.

Le applicazioni Web più semplici sono a tre *tier*, quindi **three-tier**, come quella illustrata in Figura 9.1.

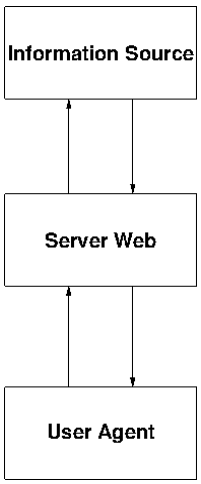


Figura 9.1: Architettura a tre *tier*

Il *tier* intermedio controlla l'interazione tra client e sorgente di informazione; implementa:

- il meccanismo di controllo (**controller logic**): elabora la richiesta del client e si procura i dati da presentargli;
- il meccanismo di presentazione (**presentation logic**) elabora questi dati per presentarli all'utente (in HTML, XHTML, WML, ...)

- le regole del dominio applicativo (**business logic**) e si assicura che i dati siano affidabili prima di usarli per aggiornare la sorgente di informazione o per rispondere all'utente; le **business rules** regolano l'accesso dei client ai dati e l'elaborazione dei dati da parte dell'applicazione.

Problema di **partizionare** un'applicazione tra i diversi tier: in particolare, posso scaricare una parte della computazione sul client? I pattern propongono soluzioni per problemi di progettazione ricorrenti. Tra i pattern a livello architetturale, che descrivono la struttura dei sottosistemi, le loro responsabilità e interazioni, troviamo il pattern Model-View-Controller (MVC) [1]. MVC si adatta particolarmente bene a linguaggi orientati agli oggetti.

Model : informazioni e operazioni su di esse specifiche del dominio applicativo.

View : produce una rappresentazione (di parti) del modello a supporto dell'interazione con l'utente.

Controller : gestisce le variazioni del modello e di eventuali viste correlate; in un'applicazione web non può che partire da eventi scatenati dall'utente.

9.5.1 Esempio: un negozio virtuale

Per dare concretezza a quanto detto consideriamo un classico esempio di applicazione web: un sistema di acquisto on-line (con carrello gestito tramite variabili di sessione). Includiamo nell'esempio anche pagine per l'amministrazione del sistema.

- **Model**: carrelli virtuali, ordini, prodotti e utenti dell'area riservata, che rappresentano le entità fondamentali dell'applicazione.
- **View**: interfaccia grafica con cui l'utente può interagire.
- **Controller**: viene implementato da una servlet che reagisce all'input dell'utente aggiornando opportunamente il modello e selezionando la nuova vista da proporre all'utente.

Per **piattaforma (framework)** si intende un sistema software riutilizzabile con funzionalità generali già implementate.

Distinguiamo diversi tipi di architetture a seconda di:

- **Organizzazione in livelli**: separazione delle responsabilità.
- **Presentazione dei dati**: strutturati (basi di dati relazionali, XML) versus non strutturati (dati multimediali).

La Sun propone un'architettura che ingloba l'uso delle JSP in un pattern MVC.

Tale architettura viene ulteriormente potenziata da **Struts**, un progetto open-source della Apache Software Foundation [2], che tra le altre cose permette di controllare il flusso di controllo, configurando il sistema mediante un documento XML.

9.6 Apache Struts 2

Quando le JSP sono state introdotte, pareva che dovessero soppiantare completamente le servlet. Tuttavia questo non è successo. Infatti in questa visione, a cui ci si riferisce spesso come "Modello 1", l'applicazione veniva costruita mediante una serie di JSP, tra le quali ci si muoveva attraverso dei link: più nel dettaglio, questo significa che ogni JSP costruiva una pagina da passare al client. Il passaggio ad un'ulteriore JSP, quindi, avveniva attraverso una successiva richiesta HTTP normalmente causata da un'azione dell'utente, quali il click su un link, o la sottomissione di una form. In questo senso, possiamo affermare che il Modello 1 è *centrato sulla pagina*. Questo modello è facile da progettare ed implementare, ma difficile da manutenere e poco flessibile. Inoltre non incentiva la separazione delle responsabilità tra chi si occupa del rendering della pagina e lo sviluppatore, visto che quest'ultimo viene ad essere coinvolto sia nella preparazione della pagina che nell'implementazione della business logic. Questo, insieme alle difficoltà legate all'uso dei JavaBeans e delle etichette personalizzate che abbiamo discusso nelle lezioni precedenti,

[1] `http://java.sun.com/blueprints/patterns/MVC.html`

[2] `"http://struts.apache.org/`

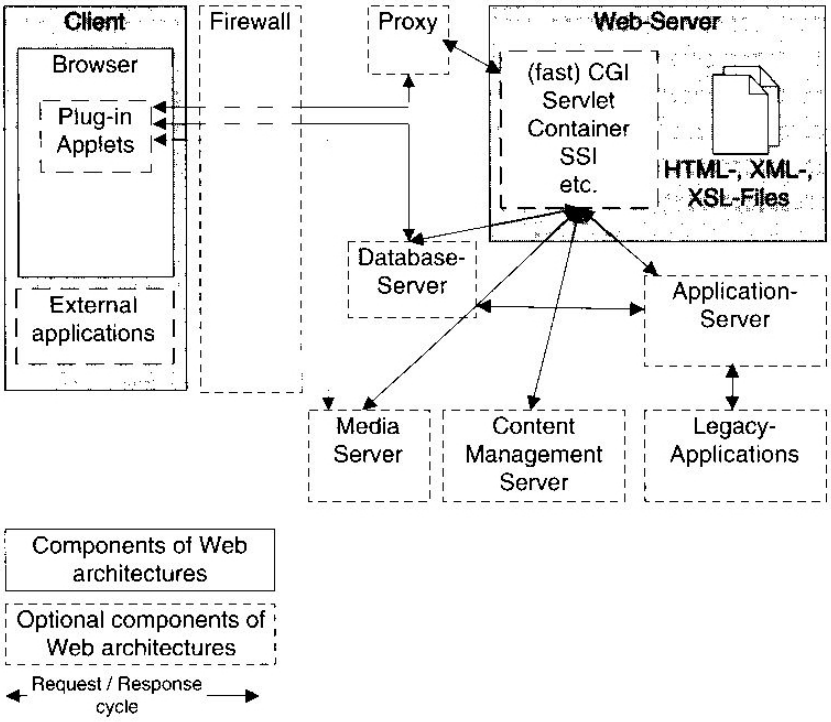


Figure 4-2 Basic components of Web application architectures.

Figura 9.2: Architettura web completa

porta a mescolare la parte Java col template statico, con tutti gli svantaggi che conosciamo. L'alternativa all'introduzione degli scriptlet comporta la necessità di dedicare più tempo allo sviluppo dell'applicazione, dovendo sviluppare etichette personalizzate per la maggior parte della business logic: ovviamente sviluppare direttamente delle classi Java risulta più agevole e rapido.

L'alternativa al Modello 1, chiamata nella documentazione "Modello 2" è invece basata sul pattern MVC per il Web che abbiamo visto nel Capitolo 7. Quando questo modello è stato introdotto, nella versione 0.92 delle specifiche delle JSP, una servlet (o un filtro) è usata come controller, mentre le JSP sono usate come presentazione. Per la sua modularità e in generale per la sua organizzazione, questo modello è più facile da applicare in fase di sviluppo, di testing, di manutenzione e per eventuali estensioni, tant'è vero che, come abbiamo visto nel Capitolo 7 è stato adottato dalla maggior parte dei Web Frameworks, e in particolare dal principale tra quelli basati su Java, ovvero Apache Struts, di cui ci occupiamo in questo capitolo. In particolare faremo riferimento ad Struts 2, anche se tipicamente ci dimenticheremo di esplicitare il 2. Ovviamente fa uso di JSP, ma senza codice Java all'interno delle JSP: la business logic viene implementata da classi Java dette *action classes*. Viene fornito un Expression Language (OGNL) con cui fare accesso agli oggetti del model dalle JSP. L'uso di etichette personalizzate è ridotto al minimo.

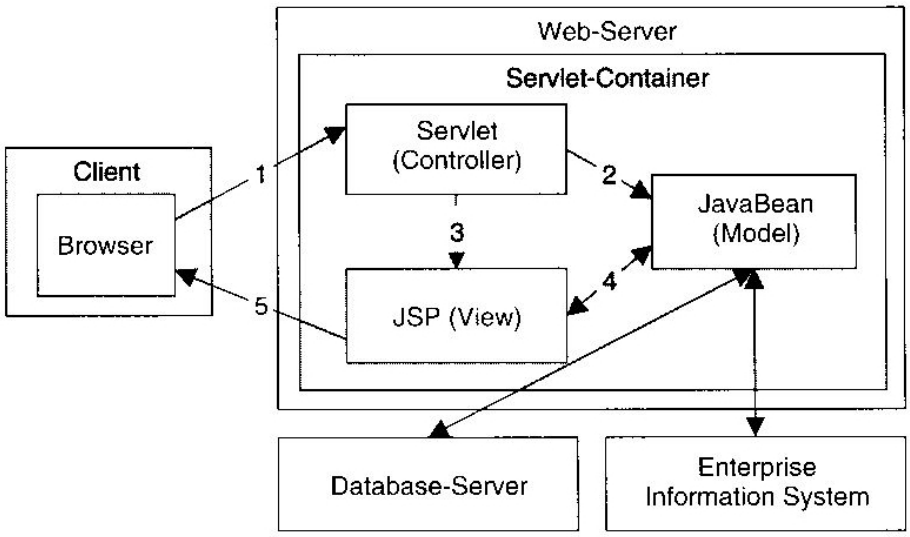


Figure 4-5 The JSP-Model-2 architecture.

Figura 9.3: Architettura MVC basata su JSP

10. Session tracking

Questo capitolo è dedicato alle soluzioni standard adottabili per la soluzione di un problema tipico delle applicazioni web, riconducibile al fatto che HTTP è un protocollo senza stato: il session-tracking, che abbiamo già introdotto nel Capitolo 2. Gli esempi sono dati tutti in Java, per ovvi motivi didattici, ma ovviamente gli stessi meccanismi possono essere implementati in qualsiasi linguaggio di programmazione. Si noti infine come la possibilità di memorizzare informazione in **HTML5** provveda un nuovo sistema di session tracking. Per approfondire esempi di session tracking con Java, risulta utile il cap. 7 di [HC01].

Ci dobbiamo porre le seguenti domande:

1. Cosa si intende per *sessione*?
2. Quali sono i criteri per preferire l'una o l'altra delle soluzioni?

Consideriamo le seguenti sei tecniche di session tracking:

1. autorizzazione richiesta all'utente;
2. campi nascosti nelle form;
3. riscrittura dell'URL;
4. cookies;
5. variabili di sessione;
6. web storage di HTML5, che vedremo nel Capitolo 12.

Exercise 10.1 Cosa si intende per session tracking e perché è necessario affrontare il problema in un'applicazione web? ■

Exercise 10.2 Spiegare perché i meccanismi di caching non sono compresi tra le tecniche di session tracking. ■

10.0.1 Autenticazione richiesta all'utente

Implementata a livello di HTTP, come abbiamo già evidenziato nel capitolo dedicato al protocollo, in Sezione 2.2.2. Tramite le variabili di sessione è anche possibile implementare un meccanismo di autenticazione che però risulta molto meno sicuro e quindi sconsigliabile. Ricordiamo che finché il

browser rimane aperto, username e password vengono salvati sul client, in modo che l'utente non debba digitarli tutte le volte che viene inoltrata una richiesta.

Vediamo ora come tramite servlet si possa accedere all'informazione della sessione quando si usa un meccanismo di autenticazione via HTTP, Il metodo *getRemoteUser()* fornisce alla servlet lo username dell'utente dopo che questo ha fatto il login fornendolo insieme ad una password.

Aggiunge un oggetto al carrello della spesa:

```
String name = request.getRemoteUser();
if(name == null) {
  // message: page is protected
} else {
  String[] items = request.getParameterValues("item");
  if( items != null) {
    for( int i=0; i < items.length; i++ ) {
      addItemToCart( name, items[i] );
    }
  }
}
```

E per elencare gli oggetti nel carrello:

```
String name = request.getRemoteUser();
if(name == null) {
  // message: page is protected
} else {
  String[] items = getItemsFromCart(name);
}
```

Vantaggi

- Facile da implementare: il server protegge determinate pagine e può poi identificare ogni client (usando ad esempio il metodo *getRemoteUser()* per le servlet).
- Il meccanismo delle autenticazioni funziona anche se l'utente accede al sito da macchine diverse.
- Continua a funzionare anche se l'utente esce dal sito o addirittura esce dal browser prima di tornare a concludere la transazione.

Svantaggi

- La richiesta di registrazione si adatta solo a determinate situazioni (transizioni commerciali, informazioni sensibili); in tutti gli altri casi occorre un session tracking anonimo.
- (Meno importante) Un utente può, in ogni momento, mantenere attiva una sola sessione dato un sito.

10.0.2 Campi nascosti nelle form

```
<form action="/servlet/MovieFinder" method="post">
  ...
  <input type="hidden" name="zip" value="94040">
  <input type="hidden" name="level" value="expert">
  ...
</form>
```

Servlet per la gestione del carrello della spesa coi campi nascosti:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartViewerHidden extends HttpServlet {
  public void doPost(HttpServletRequest request,
                  HttpServletResponse response)
    throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      out.println("<h4>Contenuto del carrello "
                + "della spesa</h4>");
      out.println("");

      // Gli oggetti del carrello vengono passati come parametri
      String[] items = request.getParameterValues("item");

      // Stampa del contenuto del carrello
      out.println("Il tuo carrello al momento contiene:");
      if(items == null) {
        out.println(" nulla}");
      } else {
        out.println("<ol>");
        for(int i=0; i<items.length; i++) {
          out.println("<li> " + items[i]) + "</li>";
        }
        out.println("</ol>");
      }

      // L'utente puo' aggiungere oggetti al carrello o uscire
      // Gli oggetti correnti vengono aggiunti come campi nascosti, in
      // modo da conservarli per la prossima richiesta
      out.println("<form action=\"/servlet/ShoppingCart\"
                   method=\"post\">");
      if(items != null) {
        for(int i=0; i < items.length; i++) {
          out.println("<input type=\"hidden\" name=\"item\" value\""
                      + items[i] + "\">");
        }
      }
      out.println("Preferisci:");
      out.println("<input type=\"submit\" value=\" Aggiungere oggetti \">");
      out.println("<input type=\"submit\" value=\" Uscire \">");
      out.println("</form>");
      out.println(" ");
    }
  }
```

```
Il tuo carrello al momento contiene:
```

```
<ul>
  <li>Primo oggetto</li>
  <li>Secondo oggetto</li>
</ul>

<form action="/servlet/ShoppingCart" method="post">
Preferisci:
  <input type="submit" value=" Aggiungere oggetti ">
  <input type="submit" value=" Uscire ">
</form>
```

Nei casi in cui l'informazione da passare avanti e indietro ad ogni richiesta è di dimensione non trascurabile, può divenire conveniente salvare l'informazione direttamente nel server e passare solo un identificativo che permetta di identificare la sessione.

Vantaggi

- sono supportati da tutti i browser più diffusi;
- non richiedono nessuna particolare funzionalità da parte del server;
- sono anonimi, e non richiedono quindi all'utente di registrarsi.

Svantaggi

- funziona solo quando lo scambio prevede una sequenza di schede da riempire generate dinamicamente;
- fallisce nel caso di errore che provoca la chiusura del browser.

10.0.3 Riscrittura dell'URL

Normalmente c'è solo per un unico identificativo di sessione.

http://server:porta/servlet/Rewritten	originale
http://server:porta/servlet/Rewritten/123	path aggiuntivo
http://server:porta/servlet/Rewritten?sessionid=123	parametri aggiunti
http://server:porta/servlet/Rewritten;*sessionid*123	cambiamento personalizzato

Discussione:

- *path aggiuntivo*:
  - funziona su tutti i server;
  - può essere usato come target per forms che usano sia il *get* che il *post*;
  - dà però problemi se una servlet deve usare il path aggiunto come path vero e proprio;
- *parametri aggiunti*:
  - funziona su tutti i server;
  - non funziona per form che usano il *post*;
  - può causare collisioni nei nomi dei parametri;
- *cambiamento personalizzato*:
  - funziona su tutti i server che supportano tale cambiamento;
  - non funziona per se il server non lo supporta.

Servlet per la gestione del carrello della spesa con la riscrittura dell'URL aggiungendo l'identificativo di sessione come path aggiuntivo:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartViewerRewrite extends HttpServlet {
```

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse response)
  throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    out.println("<p>Contenuto del carrello "
              + "della spesa</p>");
    out.println("");

    // L'identificativo di sessione viene estratto dall'URL o,
    // se necessario, creato
    String sessionid = request.getPathInfo();
    if(sessionid == null) {
      sessionid = generateSessionId();
    }

    // Gli oggetti del carrello sono associati all'identificativo di
    // sessione e vengono estratti con un metodo da implementare
    String[] items = getItemsFromCart(sessionid)};

    // Stampa del contenuto del carrello
    out.println("Il tuo carrello al momento contiene:");
    if(items == null) {
      out.println(" nulla}");
    } else {
      out.println("<ul>");
      for(int i=0; i<items.length; i++) {
        out.println("<li> " + items[i]) + "";
      }
      out.println("</ul>");
    }

    // L'utente puo‘ aggiungere oggetti al carrello o uscire
    out.println("<form action=\"/servlet/ShoppingCart/" + sessionid + \
                                                    "\" method=POST>");
    out.println("Preferisci:");
    out.println("<input type=\"submit\" value=\" Aggiungere oggetti \">");
    out.println("<input type=\"submit\" value=\" Uscire \">");
    out.println("</form>");

    // Pagina di aiuto, includendo l'identificativo di sessione
    // nell'URL NB: impossibile coi campi nascosti, perch\'e
    // la servlet Help non e‘ il target di una sottomissione
    // di un modulo da riempire.}
     out.println("Per ottenere <a href=\"/servlet/Help/" + sessionid
                + "?topic=ShoppingCartViewerRewrite\">aiuto" );}

    out.println(" ");
```

```
    }

    private static String generateSessionId() {
      String uid = new java.rmi.server.UID().toString(); // garantisce unicita'
      return java.net.URLEncoder.encode(uid); // codifica caratteri speciali
    }

    private static String() getItemsFromCart(String sessionid) {
      // implementazione
    }
  }
```

Vantaggi e svantaggi simili a quelli dei campi nascosti, ma funziona per tutti i documenti dinamici; addirittura, con la sintassi personalizzata può essere usato anche con documenti statici. Piuttosto noioso da implementare.

10.0.4 Cookies

Come già visto nel capitolo sull'HTTP, in Sezione 2.2.2, un **cookie** è un pezzo di informazione testuale mandata dal server Web al browser, che può in seguito venire ripassata dal browser al server.

I browser che offrono supporto ai cookies devono essere in grado di immagazzinare almeno 20 cookies per ogni sito Web e almeno 300 per ogni utente; tuttavia possono imporre un limite superiore alla dimensione di ogni cookie di 4K (4096 bytes).

Java, e quindi le servlet, offrono una classe <tt>Cookie</tt> con:

- Costruttore: *public Cookie(String name, String value)*, dove il nome identifica il cookie, mentre il valore rappresenta l'informazione associata col cookie.
- Per mandare un cookie da una servlet ad un client: *public void HttpServletResponse.addCookie(Cookie cookie)*: altri cookies possono essere aggiunti con lo stesso metodo, ma prima di ogni altro contenuto.
- Per estrarre i cookies da una richiesta di un client una servlets usa *public Cookie[] HttpServletRequest.getCookies()*

```
// per spedire un cookie
Cookie cookie = new Cookie("ID","123");
response.addCookie(cookie);

// per estrarre i cookies
Cookie[] cookies = request.getCookies();
if(cookies != null) {
  for(int i=0; i<cookies.length; i++) {
    String name=cookies[i].getName();
    String value=cookies[i].getValue();
  }
}
```

- *public void setVersion(int v)*: 0 (Netscape, default), 1 RFC 2109
- *public int getVersion()*
- *public void setDomain(String pattern)*, (es, .foo.com per *.foo.com): restrizioni sul dominio; per default, dominio da cui sono state spedite
- *public String getDomain()*

- *public void setMaxAge(int expiry)*, in secondi; default, expiry < 0, fine della sessione; 0 immediatamente;
- *public int getMaxAge()*
- *public void setPath(String url)*: specifica ulteriormente il dominio, e deve comprendere la servlet che ha creato il cookie;
- *public String getPath()*
- *public void setSecure(boolean flag)*, default *false*
- *public boolean getSecure()*
- *public void setComment(String comment)*, con lo scopo del cookie; il browser può visualizzarlo o no;
- *public String getComment()*
- *public void setValue(String newValue)*
- *public String getValue()*
- *public String getName()*

Servlet per la gestione del carrello della spesa coi cookies:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartViewerCookie extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();

      // L'identificativo di sessione viene ottenuto cercando il
      // il cookie corrispondente
       String sessionid = null;
      Cookie[] cookies = request.getCookies();
      if(cookies != null) {
        for(int i=0; i<cookies.length; i++) {
          if(cookies[i].getName().equals("sessionid")) {
            sessionid = cookies[i].getValue();
            break;
          }
        }
      }}

      // Se l'identificativo di sessione non e' stato mandato,
      // generarne uno; includerlo nella risposta per il client
      if(sessionid == null) {
        sessionid = generateSessionId();
        Cookie c = new Cookie("sessionid",sessionid);
        response.addCookie(c);
      }

      // Solo ora posso cominciare a scrivere la risposta
      out.println("<h4>Contenuto del carrello " + "della spesa</h4>");
```

```
    out.println("");

    // Gli oggetti del carrello sono associati all'identificativo di
    // sessione e vengono estratti con un metodo da implementare
    String[] items = getItemsFromCart(sessionid);

    // Stampa del contenuto del carrello
    out.println("Il tuo carrello al momento contiene:");
    if(items == null) {
      out.println(" nulla}");
    } else {
      out.println("<ul>");
      for(int i=0; i<items.length; i++) {
        out.println("\item " + items[i]) + "";
      }
      out.println("</ul>");
    }

    // L'utente puo' aggiungere oggetti al carrello o uscire
    out.println("Preferisci:");
    out.println("<input type=\"submit\" value=\" Aggiungere oggetti \">");
    out.println("<input type=\"submit\" value=\" Uscire \">");
    out.println("</form>");

    // Pagina di aiuto
     out.println("Per ottenere <a href=\"/servlet/Help/"
                + "?topic=ShoppingCartViewerCookie\">aiuto" );}

    out.println(" ");
  }

  private static String generateSessionId() {
    String uid = new java.rmi.server.UID().toString(); // garantisce
                                                       // unicita'
    return java.net.URLEncoder.encode(uid);
  }

  private static String() getItemsFromCart(String sessionid) {
    // implementazione
  }
}
```

Il problema più importante coi cookies è che non sempre il browser li accetta.

Per default, i cookie vengono eliminati alla fine della sessione di browsing. Per impostare una durata maggiore, si usa il metodo *setMaxAge*, che dà il numero di secondi dopo i quali il cookie verrà eliminato.

Importantissimo: i browser inviano i cookies esclusivamente allo stesso dominio memorizzato nel cookie.

10.0.5 Le variabili di sessione

Variabili sul server: solo l'identificativo di sessione viene scambiato tra client e server tramite

1. cookies
2. riscrittura dell'URL

Nelle servlet si usa l'interfaccia HttpSession

```
// Esempio con le variabili di sessione
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class SessionServlet extends HttpServlet {
   private final Map books = new HashMap();

   // initialize Map books
   public void init()
   {
      books.put( "C", "001" );
      books.put( "C++", "002" );
      books.put( "Java", "003" );
      books.put( "Python", "004" );
   }

   protected void doPost( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      String language = request.getParameter( "language" );
      HttpSession session = request.getSession( true );
      session.setAttribute( language, books.get( language ) );

      response.setContentType( "text/html" );
      PrintWriter out = response.getWriter();

      // send XHTML page to client

      // start XHTML document
      out.println( "<?xml version = \"1.0\"?>" );

         ....

      out.println( "Hai scelto " + language + "." );

      // display information about the session
      out.println( "Your unique session ID is: " + session.getId() + "" );

      out.println(
         "This " + ( session.isNew() ? "is" : "is not" ) +
         " a new session" );
```

```
      out.println( "The session was created at: " +
         new Date( session.getCreationTime() ) + "" );

      out.println( "You last accessed the session at: " +
         new Date( session.getLastAccessedTime() ) + "" );

      out.println( "The maximum inactive interval is: " +
         session.getMaxInactiveInterval() + " seconds" );

      out.println( "<a href = " +
         "\"servlets/SessionSelectLanguage.html\">" +
         "Click here to choose another language" );

      out.println( "" +
         "Click here to get book recommendations" );
      out.println( "" );

      ...

      out.close(); // close stream
   }

   protected void doGet( HttpServletRequest request,
      HttpServletResponse response )
         throws ServletException, IOException
   {
      HttpSession session = request.getSession( false );
      Enumeration valueNames;

      if ( session != null )
          valueNames = session.getAttributeNames();
      else
         valueNames = null;

      PrintWriter out = response.getWriter();
      response.setContentType( "text/html" );

      ...

      if ( valueNames != null ""
           valueNames.hasMoreElements() ) {
         out.println( "<p>Ti raccomandiamo i seguenti libri</p>" );

         String name, value;

         // get value for each name in valueNames
         while ( valueNames.hasMoreElements() ) {
            name = valueNames.nextElement().toString();
```

```
            value = session.getAttribute( name ).toString();

            out.println( name + " Introduzione alla programmazione. " +
               "codice: " + value + "" );
         }

         out.println( "" );
      }
      else {
         out.println( "<p>Non sappiamo che libri segnalarti perche' non ");
         out.println( "hai scelto un linguaggio di programmazione." );
      }

      ...

      out.close(); // close stream
   }
}
```

10.1 Cookies, privacy and GDPR

The General Data Protection Regulation (GDPR) is Europe's new data privacy and security law. Though it was drafted and passed by the European Union (EU), it imposes obligations onto organizations anywhere, so long as they target or collect data related to people in the EU. The regulation was put into effect on May 25, 2018. The GDPR will levy harsh fines against those who violate its privacy and security standards, with penalties reaching into the tens of millions of euros. See `https://gdpr.eu/what-is-gdpr/` for an introduction.

On the other hand, we yet know cookies: let us recapitulate the way in which they work with an example. Recall that cookies are included in the header of HTTP requests and responses. The User Agent (UA) accesses the web site http://firstParty.com, sends an HTTP request to http://firstParty.com; http://firstParty.com returns an http response to the UA containing, in the header: `Set-Cookie: foo=bar`. The UA stores the cookie; it puts in the header of every subsequent request to http://firstParty.com the cookie `foo=bar` and eventually deletes the cookie because it is expired or for some other reason.

When a web page contains some other object (an image, or an inline frame, for example), an HTTP request is sent for every object. Objects can come from different servers.

```
<html>
 <p>Consider this example:
 <img
  src="http://www.dieti.unina.it/images/logo/sigillo.png"
  title="Sigillo Universit\'a Federico II">
</html>
```

The browser sends the `foo=bar` cookie with any request to `http://firstParty.com`, regardless of who initiated the request or what the context is.

10.1.1 The field referer

The Referer request header contains the **address of the page making the request** (English: referrer). When following a link, this would be the url of **the page containing the link**. Possible uses: analytics, logging, or optimized caching, ...

Important: the use of this fields can originate undesirable consequences for user security and privacy. A Referer header is not sent by browsers in two cases: (a) The referring resource is a local "file" or "data" URI. (b) An unsecured HTTP request is used and the referring page was received with a secure protocol (HTTPS).

Let us consider the case of two websites (WebsiteA and WebsiteB) containing the same ad. More specifically, WebsiteA contains an ad that is served by websitePromo: this implies that WebsitePromo can the send a cookie to the UA, while WebsiteA contains <img src="http://websitePromo.com/ad.html"> to serve the ad from websitePromo. When the UA fetches http://websitePromo.com/ad.html, the response will come back with a `Set-Cookie` header that sets a cookie. If websiteB also includes an ad from websitePromo, then that same cookie will be sent when the ad on websiteB is fetched from websitePromo. No information leakage without the `Referer` field in the heading.

When the UA sends the HTTP request to websitePromo, it adds the `Referer` field with the URL of Web site A or B. If the cookie contains a unique code identifiying the user, it can be used to collect statistics about which of the sites containing ads from websitePromo UA visits. If websiteA is cooperating with websitePromo, websiteA can just directly tell websitePromo that you're coming from websiteA. For example, websiteA could include the ad from websitePromo by using `<img src="http://websiteB.com/ad.html?referer=websiteA.com">`

10.1.2 Types of cookies

The cookies can have different origin:

First-party cookies are sent directly by the website you are visiting.

Third-party cookies are sent, not by the website you are visiting, but by a third party like an advertiser or an analytic system (like in the `Referer` example above).

Different types of cookies can have different durations, but, according to the ePrivacy Directive, cookies should not last longer than 12 months. In practice, their scope can be much longer if no action is taken.

By considering their goal, and in an imprecise way, we can distinguish between session cookies and persistent cookies. The former are related to a session and end when the session ends, while the later are thought, at least from the application designer, to last forever. Of course we need to remember that from another point of view, cookies are a session tracking tool, and therefore they last for definition as long as the session last. However, in some cases designers aims at considering a session with a limited duration, while in others they would prefer the session not to end.

10.1.3 Cookie's purpose

Now we want to distinguish between the cookies by considering their purpose. First of all, some cookies are strictly necessary while others are preferences cookies. The former are essential for site functionality, like accessing secure parts of the site, holding objects in the cart, ... They are usually first-party session cookies. It is not required to obtain the user's consent for them: it is sufficient to explain to the user what they do and why they are necessary.

On the other hand, preferences (or functionalities) cookies have the purpose of storing choices made in the past, like the preferred language, the region of interest for weather reports, or user name and password to automatically log in (more in future lessons).

Another important class of cookies are statistics (or performance) cookies. They collect information about a website use, like which pages are visited and which links are clicked on by a user. Their main characteristics include that none of the information they contain can be used to identify the user. This is also due to the fact that such information is aggregated and, therefore, anonymized. Their sole purpose is to improve website functions. This class includes cookies from

third-party analytics services as long as the cookies are for the exclusive use of the owner of the website visited.

Last, but not least, we can consider marketing cookies: they track your online activity to help advertisers deliver more relevant advertising or to limit how many times you see an ad. As a consequence, they can share that information with other organizations or advertisers. From the point of view of their duration, they are persistent cookies, and almost always they have third-party provenance.

10.1.4 Conclusions

There are cookies that will not fit neatly into these categories or may qualify for multiple categories. Privacy risks are usually related to third-party, persistent, marketing cookies. The **chain of responsibility** (who can access a cookies' data) for a third-party cookie can get complicated. This heighten their potential for abuse. Perhaps because of this, the use of third-party cookies has been in decline since the passage of the GDPR and new tools are being proposed as an alternative, including Google Topics API (not included in this course).

11. Cascading StyleSheets o CSS

In questo capitolo introdurremo un importante strumento per curare la formattazione del documento HTML o XML: il CSS. Per lo studio di questo argomento si possono utilizzare:

1. Capitolo 6 di [DND02];
2. CSS dal W3C[1];
3. Media Queries dal W3C[2].

11.1 Cos'è CSS?

Si tratta di un linguaggio dichiarativo per associare ad un documento HTML o XML uno stile di presentazione. Lo fa attraverso dei **fogli di stile** con lo scopo di adattare i contenuti alla varietà dei media disponibili. Applica un principio fondamentale di buona preparazione dei contenuti per il web, ovvero quello della **separazione tra contenuto e presentazione**: in questo modo, per portare i contenuti su media diversi, non occorre riporgettare tutto, ma solo applicare stili diversi. In particolare, il css supporta diversi media tra cui la stampante, video, palmare, speech, braille, TV, ...

Un'importante alternativa al CSS è rappresentata dall'eXtensible Stylesheet Language o XSL. Essi però hanno delle caratteristiche profondamente diverse, in quanto possono fare delle cose diverse. Va detto subito, anche se la giustificazione completa verrà data nel Capitolo **??** quando si introdurrà l'XSL, che quest'ultimo dà la possibilità di applicare al documento delle trasformazioni che col CSS risultano impossibili. Tuttavia, mentre il CSS può venir usato anche con l'HTML, l'XSL funziona solo se applicato ad un documento XML.

Il CSS non ha il tradizionale sistema di versioni, ma un'organizzazione a **livelli**. Ogni livello del CSS costruisce sul livello precedente, introducendo rifiniture nelle definizioni e aggiungendo nuove features. Ne deriva che l'insieme delle feature di ogni livello è un soprainsieme di quello del livello immediatamente inferiore, visto che nuove feature possono venir aggiunte ad ogni livello. D'altra parte per ogni feature, l'insieme dei comportamenti ammessi è un sottoinsieme di quelli ammessi nei livelli inferiori, visto che ogni comportamento può venir ulteriormente raffinato da un

[1] `http://www.w3.org/TR/CSS/`
[2] `http://www.w3.org/TR/css3-mediaqueries/`

livello superiore. In questo modo, uno user agent che rispetti un livello superiore di CSS è garantito rispettare anche quelli inferiori. Attualmente il W3C sta lavorando al livello 3.

CSS può essere usato per definire uno stile di presentazione sia per HTML che per XML, mentre XSL trasforma documenti (ad esempio, da XML a HTML/CSS sul server Web). Da questo punto di vista, i due linguaggi si completano quindi a vicenda e possono venir usati insieme.

Entrambi i linguaggi possono venir usati per definire uno stile di presentazione per i documenti XML.

11.1.1 Le regole

CSS permette di formulare delle regole per associare ad ogni elemento del documento in ingresso, HTML o XML, delle proprietà che ne determinano il trattamento (colore, dimensioni, ...). Sia HTML che XML presentano una struttura logica ad albero, nel senso che tutti gli elementi eccetto la radice hanno uno ed un solo sovraelemento. Questa struttura risulta importante perché certi attributi dei nodi sono ereditati dai discendenti.

```
H1 { color : blue}
```

Una regola CSS è formata da due parti: un **selettore** (`H1`) e una **dichiarazione** (`color:blue`). La dichiarazione a sua volta ha due parti: la **proprietà** (`color`) e il **valore** (`blue`). Quello riportato sopra, pur molto semplice, è di per sè uno stylesheet. Gli stylesheet possono venir combinati per ottenere la presentazione finale del documento.

Esistono tre modi per inserire le regole CSS in un documento:

- con l'attributo STYLE;
- style sheet incluso nel file
- style sheet esterno

Per metterle all'interno, si usi l'elemento STYLE:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>
<TITLE>Home page di Dante</TITLE>
<STYLE type="text/css">
H1 { color: blue }
</STYLE>
</HEAD>
<BODY>
<H1>Home page di Dante</H1>
<P>Dante ha scritto la Divina Commedia.
</BODY>
</HTML>
```

che dà come risultato:

Un altro esempio:

```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Esempio di CSS</title>

      <!-- le regole CSS sono state introdotte all'interno del documento
        tramite l'elemento STYLE-->
      <style type = "text/css">

         em { background-color: #8000ff;
                   color: wheat }

         h1 { font-family: arial, sans-serif }

         p { font-size: 14pt }

         .special { color: green }

      </style>
   </head>

   <body>

      <!-- definiamo una classe che applica la regola special -->
      <h1 class = "special">Corso di Tecnologie Web: obiettivi formativi</h1>

<p>Scopo del corso &egrave; di fornire concetti e tecniche per la
progettazione di siti web sofisticati. Alla fine del corso, lo
studente dovrebbe quindi essere in grado di progettare un'applicazione
web scegliendo gli strumenti pi&ugrave; adatti e di seguire
l'evoluzione delle tecnologie legate a questo campo di applicazione.
</p>
```

```
     <h1>Corso di Tecnologie Web: contenuti</h1>

<p class="special">Introduzione al web: protocollo HTTP,
architettura a tre e pi&ugrave; strati, web statico e web dinamico,
linguaggi di mark-up (XML/HTML/XHTML). Programmazione lato server:
strumenti basati su Java, quali servlet, JSP, JavaBeans, JDBC; cenni
di PHP. Programmazione lato client: applet, JavaScript, fogli di
stile (CSS e XSL), DOM, AJAX. Introduzione a terminali mobili,
browsing vocale e programmazione su web (bots, spiders e crawlers).
</p>

   </body>
</html>
```

che risulta in:

Può tuttavia essere consigliabile metterli all'esterno:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<HTML>
<HEAD>
<TITLE>Home page di Dante</TITLE>
  <LINK rel="stylesheet" href="dante.css" type="text/css">
</HEAD>
<BODY>
<H1>Home page di Dante</H1>
<P>Dante ha scritto la Divina Commedia.
</BODY>
</HTML>
```

L'elemento LINK specifica:

- tipo di link: ad uno `stylesheet`;
- la locazione dello style sheet attraverso l'attributo `ref`;
- il tipo di style sheet: `text/css`;

Altri esempi di regole CSS, sempre di forma "Selettore { proprietà : valore }":

```
HEAD{ display: none }
A[href] { text-decoration: underline }
H1 + H2 { margin-top: -5mm }
```

I selettori stabiliscono a quali elementi si applica una regola. Le proprietà (e il loro valore) stabiliscono come l'elemento deve essere rappresentato. Tuttavia, le regole possono anche estendere il contenuto dell'elemento. La sintassi dei selettori è molto importante perché determina quali insiemi di selettori possiamo descrivere per associarli nella stessa regola:

*	qualunque elemento
E	qualunque elemento di nome E
E, F	qualunque elemento E e qualunque elemento F
E F	qualunque elemento F dentro/sotto un E <E> ... <X> ... <F> </F> .... </X> </E>
E > F	qualunque elemento F figlio di un E
E + F	qualunque elemento F preceduto da un E <E> ... </E> <F> ... </F>
E[attr]	qualunque elemento E con l'attributo attr impostato
E[attr="x"]	qualunque elemento E con attr uguale a x
E[attr ="x"]	qualunque elemento E con attr uguale ad una lista di valori separati da spazi, che include x

Esistono poi gli **pseudo-elementi** che corrispondono a delle pseudo-etichette che ci permettono di identificare quando un elemento sta in un certo stato:

E:first-child	ovvio <X> <E> ... </E></X>
E:link	per sorgenti di hyperlink non visitati
E:visited	per sorgenti di hyperlink già visitati
E:active	elementi selezionati col mouse
E:hover	elementi su cui si trova il mouse
E:focus	elementi che hanno il focus ad es. per finestre che accettano input

Classi: per aumentare la granularità dell'applicazione delle regole, si possono definire delle classi.

```
H1.pastoral { color: red }
...
$<$H1 CLASS="pastoral">Titolo</H1>
```

NB: le regole di ereditarietà funzionano normalmente anche per gli elementi con classe.

In ogni selettore si può specificare una sola classe: 'P.classeA.classeB' non è valido (una classe per selettore nei selettori composti).

Proprietà degli elementi: CSS permette di definire le proprietà dei nodi

11.1.2 Conflitti e loro risoluzione

CSS prescrive 3 tipi di stilesheets:

1. quello definito dall'autore del documento
2. quello definito dall'utente
3. quello definito come default dallo user agent

Nel caso di conflitti, la regola di soluzione prevede che autore > utente > stile di default dello user agent.

Overriding di regole ereditate:

- regole esplicite > proprietà ereditate

```
<STYLE TYPE="text/css">
  BODY { color: green }
  H1 { color: navy }
</STYLE>
```

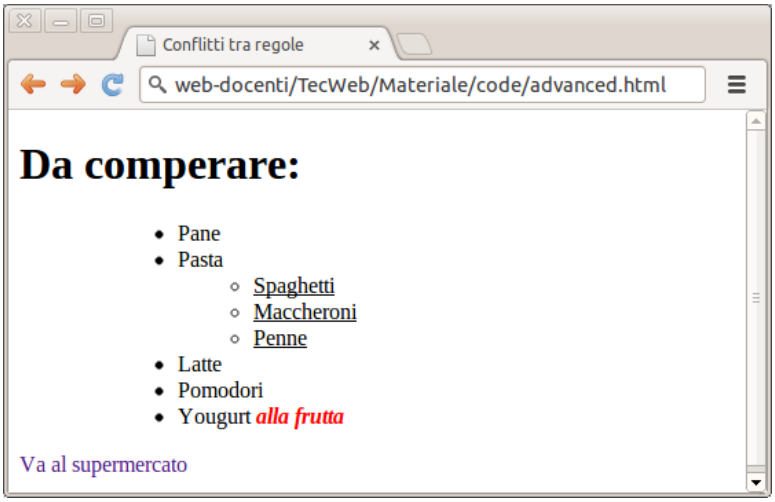
L'ordine in cui sono poste queste due regole è irrilevante: l'effetto è di avere una pagina scritta in verde coi titoli H1 in blu.

Esempio

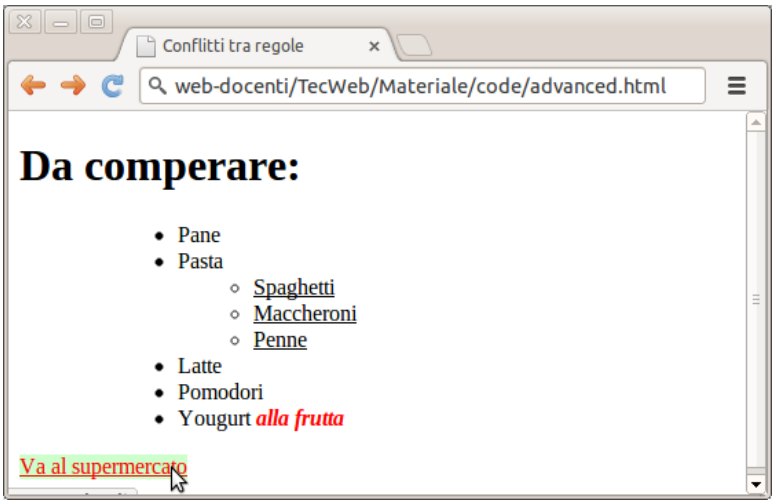
```
<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Conflitti tra regole</title>

      <style type = "text/css">

         a.nodec { text-decoration: none }

         a:hover { text-decoration: underline;
                   color: red;
                   background-color: #ccffcc }

         li em { color: red;
                   font-weight: bold }

         ul { margin-left: 75px }

         ul ul { text-decoration: underline;
                   margin-left: 15px }

      </style>
   </head>

   <body>

      <h1>Da comperare:</h1>

      <ul>
         <li>Pane</li>
         <li>Pasta
            <ul>
               <li>Spaghetti</li>
               <li>Maccheroni</li>
               <li>Penne</li>
            </ul>
         </li>
         <li>Latte</li>
         <li>Pomodori</li>
         <li>Yougurt <em>alla frutta</em></li>
      </ul>
```

```
      <p><a class = "nodec" href = "http://www.unina.it">
      Va al supermercato</a></p>

   </body>
</html>
```

che dà:



e



Maggiore specificità (comprende anche l'ereditarietà il discendente è più specifico dell'antenato e quindi il suo stile, se specificato, ha la precedenza):

- H2 margin-top: +5mm (*sovrascritto da*)
- H1 + H2 margin-top: -5mm (*più specifico*)

Ma ci sono anche delle proprietà che non vengono ereditate: ad esempio **background**.

```
<HTML>
  <TITLE>Home page di Pluto</TITLE>
  <STYLE TYPE="text/css">
    BODY {
      background: url(texture.gif) white;
      color: black;
    }
  </STYLE>
```

```
  <BODY>
    <H1><EM>home</EM> page di Pluto</H1>
    <P>Pluto &egrave; il cane di Topolino.
  </BODY>
</HTML>
```

L'immagine di background copre tutta la superficie della pagina, compresi gli elementi H1 e P: questo perché di solito lo sfondo è trasparente. L'ulteriore specifica white viene usata se non si trova l'immagine.

11.1.3 Fogli di stile dipendenti dai tipi di media

Con HTML4 e CSS2 è possibile condizionare il foglio di stile da applicare ai diversi **tipi di media**. Ad esempio, un documento potrebbe adottare un font sans-serif quando mostrato a video e serif quando stampato. "screen" e "print" sono due tra i tipi di media previsti, probabilmente quelli usati più di frequente.

Ci sono due sistemi per condizionare il foglio di stile al tipo di media:

1. dal foglio di stile, con le regole `@media`: `@media print, handheld background: white; color: black`

   ```
   @media print {
       BODY { font-size: 10pt }
   }
   @media screen {
       BODY { font-size: 12pt }
   }
   @media screen, print {
       BODY { line-height: 1.2 }
   }
   ```

2. dal documento, ad esempio, specificando un attributo `media` nell'elemento `LINK` dell'HTML 4.0:

   ```
   <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
   <HTML>
     <HEAD>
        <TITLE>Link to a target medium</TITLE>
        <LINK rel="stylesheet" type="text/css"
        {\bf media="print, handheld"} href="foo.css">
      </HEAD>
      <BODY>
          <P>Il contenuto della pagina ...
      </BODY>
   </HTML>
   ```

Le **media queries** estendono le funzionalità dei tipi di media permettendo una classificazione dei fogli di stile ancora più precisa, visto che oltre al tipo di media si possono specificare anche delle espressioni che controllano se sono verificate delle **media features**. Tanto per fare un esempio, si possono controllare i valori di `width`, `height` e `color`. In questo modo è possibile ottenere un adattamento molto preciso al particolare device senza dover minimamente toccare i contenuti del documento.

Ovviamente non tutte le implementazioni offrono supporto per tutti i media previsti dal CSS: usualmente vengono implementate solo le funzionalità richieste da uno specifico profilo CSS, dove

un profilo è definito proprio come un sottoinsieme del CSS considerato fondamentale per una specifica classe di implementazioni CSS. Il gruppo del W3C che lavora sui CSS ne ha definiti tre:

- CSS Mobile Profile 2.0;
- CSS Print Profile 1.0;
- CSS TV Profile 1.0.

Relazione tra gruppi e tipi di media:

continuous/paged	visual/aural/tactile	grid/bitmap	interactive/static	
aural	continuous	aural	N/A	both
braille	continuous	tactile	grid	both
emboss	paged	tactile	grid	both
handheld	both	visual	both	both
print	paged	visual	bitmap	static
projection	paged	visual	bitmap	static
screen	continuous	visual	bitmap	both
tty	continuous	visual	grid	both
tv	both	visual, aural	bitmap	both

Posizionamento degli elementi: figure testo

11.1.4 Media di tipo aural

Esempio di foglio di stile per un media orale (sintetizzatori):

```
H1, H2, H3, H4, H5, H6 {
voice-family: paul; # tipo di voce
    stress: 20;   # piatta
    richness: 90; # ricca
    cue-before: url("ping.au") # suono che viene attivato
                               # prima di ciascun header
}
P.heidi { azimuth: center-left } # come se venisse da davanti
                               # a sinistra}
P.peter { azimuth: right } # come se venisse da destra}
P.goat { volume: x-soft } # molto leggera}
```

11.2 CSS3 Speech Module: il modulo CSS3 per il parlato

Le parti seguenti sono prese dal sito del W3C[3].

Integrazione tra parlato e visuale; importanza della struttura.

The speech rendering of a document, already commonly used by the blind and print-impaired communities, combines speech synthesis and "auditory icons". Often such aural presentation occurs by converting the document to plain text and feeding this to a screen reader - software or hardware that simply reads all the characters on the screen. This results in less effective presentation than would be the case if the document structure were retained. Style sheet properties for text to speech may be used together with visual properties (mixed media) or as an aural alternative to visual presentation.

Non solo per i disabili

Besides the obvious accessibility advantages, there are other large markets for listening to information, including in-car use, industrial and medical documentation systems (intranets), home entertainment, and to help users learning to read or who have difficulty reading.

[3] `http://www.w3.org/TR/2004/WD-css3-speech-20041216/`

Si lavora su due canali stereo e la dimensione temporale; si possono variare le caratteristiche del parlato sintetico.

When using voice properties, the canvas consists of a two channel stereo space and a temporal space (you can specify audio cues before and after synthetic speech). The CSS properties also allow authors to vary the characteristics of synthetic speech (voice type, frequency, inflection, etc.).

Examples:

```
h1, h2, h3, h4, h5, h6 {
    voice-family: paul;
    voice-stress: moderate;
    cue-before: url(ping.au)
}
p.heidi { voice-balance: left; voice-family: female }
p.peter { voice-balance: right; voice-family: male }
p.goat { voice-volume: soft }
```

This will direct the speech synthesizer to speak headers in a voice (a kind of "audio font") called "paul". Before speaking the headers, a sound sample will be played from the given URL. Paragraphs with class "heidi" will appear to come from the left (if the sound system is capable of stereo), and paragraphs of class “peter” from the right. Paragraphs with class "goat" will be played softly.

11.3 E con XML

Si può usare CSS con qualsiasi formato di documento strutturato, e quindi anche con **XML**: anzi, con XML è senz’altro più importante dare indicazioni per la visualizzazione che con HTML, visto che senza di esse lo user-agent non sa come rappresentare l’informazione.

11.3.1 Esempio

Nella regola che segue il valore `inline` associato a `display` impedisce di spezzare gli elementi su più righe.

```
GOAL { display: inline }
ARTICLE, HEADLINE, AUTHOR, PARA { display: block }
```

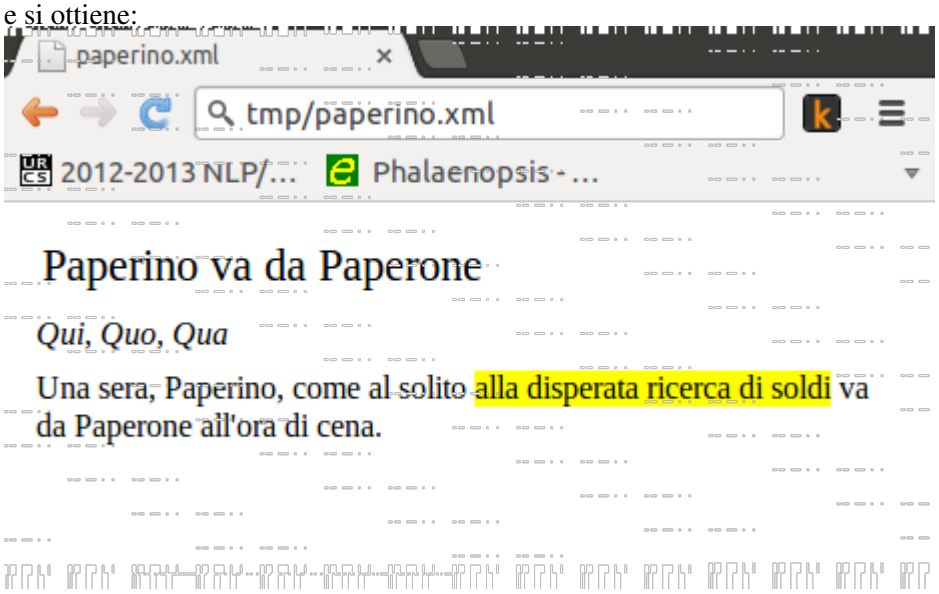
Per collegare il foglio di stile al documento XML si può usare una **processing instruction**:

```
<?xml-stylesheet type="text/css" href="paperino.css"?>
<ARTICLE>
  <HEADLINE>Paperino va da Paperone</HEADLINE>
  <AUTHOR>Qui, Quo, Qua</AUTHOR>
  <PARA>Una sera, Paperino, come al solito
      <GOAL>alla disperata ricerca di soldi</GOAL> va da Paperone
       all’ora di cena.
   </PARA>
</ARTICLE>
```

Un possibile foglio CSS potrebbe contenere le seguenti regole

```
GOAL { display: inline }
ARTICLE, HEADLINE, AUTHOR, PARA { display: block }
HEADLINE { font-size: 1.4em }
AUTHOR { font-style: italic }
ARTICLE, HEADLINE, AUTHOR, PARA { margin: 0.5em }
```

e si ottiene:


paperino.xml
tmp/paperino.xml
2012-2013 NLP/...
Phalaenopsis - ...
Paperino va da Paperone
Qui, Quo, Qua
Una sera, Paperino, come al solito alla disperata ricerca di soldi va da Paperone all'ora di cena.

12. Programmazione lato client: JavaScript

Lo studio di JavaScript seguirà la Guida messa a disposizione dal sito per gli sviluppatori di Mozilla: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide`.

13. Document Object Model

Document Object Model (DOM): come manipolare un documento HTML o XML.

13.1 Motivazioni

Scopo della programmazione lato client è, come visto, manipolare la pagina servita all'utente senza interagire col server. Come visto, si possono scegliere diversi strumenti, tra i quali il più diffuso è JavaScript. Abbiamo inoltre visto come JavaScript non sia stato ideato esclusivamente per l'uso lato client, ma possa adattarsi a ambienti host diversi a cui accede attraverso gli oggetti dell'host.

Quando JavaScript viene usato lato client, gli oggetti dell'host devono permettere al linguaggio di compiere tutte le elaborazioni richieste lato client, e quindi di elaborare il documento da presentare all'utente e tutti gli eventi ad esso collegati. Occorre quindi una struttura dati che rappresenti tale documento.

La DOM rappresenta la risposta a questa esigenza: l'acronimo sta per Document Object Model e si tratta di un'API (ovvero Application Programming Interface) che permette al linguaggio di accedere alle diverse parti del documento. Sottolineiamo che si tratta di un'API, quindi un'interfaccia, per la quale esistono diverse implementazioni.

Tale interfaccia vuole essere indipendente sia dal linguaggio di programmazione che dalla piattaforma. Quindi, anche se noi faremo tipicamente esempi in JavaScript, molti altri linguaggi hanno implementazioni per la DOM. Inoltre, la DOM permette di trattare non solo documenti HTML, ma anche documenti XML e può essere usata non solamente lato client, ma anche lato server o in applicazioni diverse.

13.2 Di cosa si tratta

Come abbiamo visto, la struttura logica di un documento XML o HTML è ad albero: ogni nodo corrisponde ad un elemento e i figli di ogni nodo corrispondono ai sottoelementi dell'elemento corrispondente. Va subito messo in evidenza che tale albero rappresenta una **gerarchia di contenimento**, non di ereditarietà, visto che:

- nessun oggetto eredita proprietà o metodi da un oggetto superiore nella catena;

- non esiste un passaggio automatico di messaggi da un oggetto all'altro in nessuna direzione: ad esempio, se vogliamo chiamare un metodo di una finestra, non lo possiamo fare attraverso un oggetto document o modulo, visto che tutti i riferimenti agli oggetti devono essere espliciti.

Originariamente ogni produttore di browser aveva definito una sua DOM a scapito dell'interoperabilità che è cosí importante nelle applicazioni web. Il problema è particolarmente importante quando si tratta di programmazione lato client: si pensi infatti alla verbosità di un programma che debba considerare uno switch su tutti i possibili browser in circolazione. Oltretutto, la comparsa sul mercato di un nuovo browser rischia di mettere in crisi l'applicazione.

La soluzione di questi problemi si basa sulla condivisione di uno standard. Ovviamente, l'ente ideale per lavorare su uno standard del genere era il W3C, che ha subito cominciato a lavorarci. Noi considereremo proprio la DOM del W3C, che è quella che dà maggiori garanzie di interoperabilità. Al momento, il consorzio sta mettendo a punto la DOM4[1] con lo scopo di adattarla all'evoluzione delle tecnologie, tra le quali HTML5, e di semplificarne l'uso.

L'ereditarietà ha un ruolo nel modello di oggetto definito dal W3C: la nuova gerarchia è più generale, perché pensata per soddisfare i requisiti di XML oltre che di HTML. Tuttavia, la gerarchia di contenimento qui introdotta è ancora valida nei browser compatibili con la DOM W3C.

Gli oggetti documento predefiniti sono generati solo quando viene caricato nel browser il codice HTML contenente le loro definizioni. I browser compatibili con JavaScript creano in memoria gli oggetti a partire dal codice HTML man mano che questo viene caricato, indipendentemente dal fatto che gli script li usino o meno. L'unica differenza vista dal codice HTML per la definizione di tali oggetti sono gli attributi dedicati specificatamente a JavaScript.

Gli oggetti vengono creati nell'ordine in cui vengono caricati: dopo aver creato un ambiente multiframe, uno script in un frame non può comunicare con gli oggetti di un altro frame fino a quando entrambi i frame non sono caricati.

Proprietà: numeri, stringhe, ma anche array e, naturalmente, metodi. Sensibili alle maiuscole in JavaScript, ma non in HTML. Alcune proprietà sono accessibili solo in lettura, mentre per altre è possibile cambiarne il valore

```
<form>
<input type="text" name="telefono">
</form>

<form>
<input type="text" name="telefono">
</form>
```

Si noti che a questo punto il form è stato caricato e quindi l'oggetto corrispondente creato in memoria:

```
<script type="text/javascript">
  document.forms[0].telefono.value = "555-1212"
</script>

<script type="text/javascript">
  document.forms[1].telefono.value = "555-1212"
</script>
```

Una volta che l'oggetto esiste in memoria, ovviamente è anche possibile creare nuove proprietà, come visto nella trattazione riguardo agli oggetti.

[1] Working Draft di giugno 2015, `http://www.w3.org/TR/domcore/`

Queste nuove proprietà esistono fino a quando il documento resta caricato nella finestra e script non sovrascrivono l'oggetto. Attenzione però che se il documento viene ricaricato, le proprietà aggiunte vengono perse (a meno che non vengano ricreate da script della pagina).

```
<form>
<input type="text" name="valore">
<input type="button" value="aggiungi proprieta"
       onClick="document.forms[0].novella = document.forms[1].valore.value">
<input type="button" value="mostra proprieta"
       onClick="alert(document.forms[0].novella)">
</form>
```

Nello stesso modo si possono aggiungere metodi a oggetti predefiniti.

```
<script type="text/javascript">
function fullScreen() {
  this.moveTo(0,0);
  this.resizeBy(100,100);
}

window.maximize = fullScreen;
</script>
<input type="button" value="allarga la finestra"
       onClick="window.maximize()">

<script type="text/javascript">
function fullScreen() {
  this.moveTo(0,0);
  this.resizeBy(100,100);
}

window.maximize = fullScreen;
</script>

<input type="button" value="allarga la finestra"
       onClick="window.maximize()">
```

Un **gestore di eventi** specifica la reazione di un oggetto a un evento attivato da un'azione dell'utente (click, ...) o del browser (caricamento di un documento, ...)

Focus indotto

```
<input type="text" name="testo"
       onFocus="var h=new Date; value=h.toString()">
```

è come definire la funzione (si noti che è tutta minuscola!):

```
function onfocus() {
  var h = new Date;
  value = h.toString;
}
```

D'altra parte posso anche richiamare direttamente la funzione:

```
<input type="button" name="bottone" value="focus indotto"
       onClick="testo.onfocus()">
```

13.3 Il DOM W3C

Il DOM (Document Object Model) del W3C si propone come un'interfaccia indipendente sia dalla piattaforma che dal linguaggio che permette a programmi e script l'accesso e l'aggiornamento dinamico di **contenuti, struttura e stile** dei documenti: si tratta quindi di un API = Application Programming Interface, un insieme di funzioni o metodi usate per accedere a delle funzionalità.

Permette quindi di elaborare il documento incorporando i risultati di tale elaborazione nel documento stesso.

Il DOM permette ai programmatori di costruire documenti, navigare all'interno della loro struttura e aggiungere, modificare o cancellare elementi e contenuti.

Essendo una specifica del W3C, l'accento viene messo sulla necessità che l'interfaccia di programmazione sia standard e possa venire usata in una varietà di ambienti e applicazioni: in particolare, vengono considerati sia Java che ECMAScript.

Il suo scopo è di rendere possibile per i programmatori scrivere applicazioni che lavorino adeguatamente su tutti i browser e server e su tutte le piattaforme.

L'architettura della DOM è divisa in **moduli**, ciascuno dei quali si riferisce ad un particolare **dominio**; i domini sono:

- XML
- HTML
- CSS (Cascading Style Sheet)
- tree events

Modello di oggetto documento applicabile sia ai documenti HTML che ai documenti XML:

- **Core DOM**: specifiche per la struttura fondamentale del documento condivisa dai documenti HTML e XML.
- **XML DOM**: estende la piattaforma Core per le necessità di XML 1.0 (istruzioni di elaborazione, CDATA, entità).
- **HTML DOM**: la seconda parte della specifica DOM è rivolta agli elementi e alle altre caratteristiche che si applicano solo ad HTML. La parte HTML eredita tutte le caratteristiche del Core DOM, garantendo in una certa misura la compatibilità con i modelli di oggetti già implementati in browser precedenti e fornendo una struttura per le nuove caratteristiche.
- **Eventi DOM**: definisce gli eventi orientati alla manipolazione dell'albero XML attraverso variazioni nell'albero e eventi orientati all'utente, quali quelli legati al mouse, alla tastiera o specifici dell'HTML.
- **DOM Cascading Style Sheets (CSS)**
- **DOM Load and Save**: l'esigenza di caricare un documento XML in un albero DOM e viceversa di salvare un albero DOM in un documento XML è fondamentale.
- **DOM Validation**: modifica l'albero DOM mantenendone la validità.
- **DOM XPath**

Qualsiasi elemento si intenda usare nell'elaborazione (ad esempio per cambiarne contenuto o stile) deve essere contraddistinto da un identificatore assegnato all'attributo ID.

Anche se ci si può riferire ad un elemento attraverso l'indice numerico nel contesto di un elemento che lo contiene (ad esempio BODY), questa è una pratica rischiosa, soprattutto quando il contenuto è in fase di costruzione. Gli identificatori univoci, al contrario, rendono molto più facili i riferimenti agli oggetti e non sono influenzati da eventuali modifiche all'ordine degli elementi.

I nuovi concetti introdotti dal DOM W3C che hanno maggiore impatto sugli script sono nuovi modi per fare riferimento ad elementi e nodi.

Ogni elemento di testo libero in un documento HTML (o XML) è un oggetto contenuto nel suo contenitore immediatamente superiore

```
<html>
  <head>
```

```
    <title>
      Una semplice pagina
    </title>
  </head>
  <body>
    <p id="paragrafo1">
      Questo e‘
      <em id="enfasi">
        l’unico
      </em>
      paragrafo della pagina.
    </p>
  </body>
</html>
```

L’oggetto document esiste automaticamente quando questa pagina viene caricata: racchiude tutto ciò che si vede nel codice HTML.

Nel DOM, i documenti hanno una struttura logica che assomiglia molto ad un **albero** o, più precisamente, una "foresta" o "boschetto", che può contenere uno o più alberi. Si tratta di un **modello logico**, che può venir implementato in qualsiasi modo risulti più conveniente.

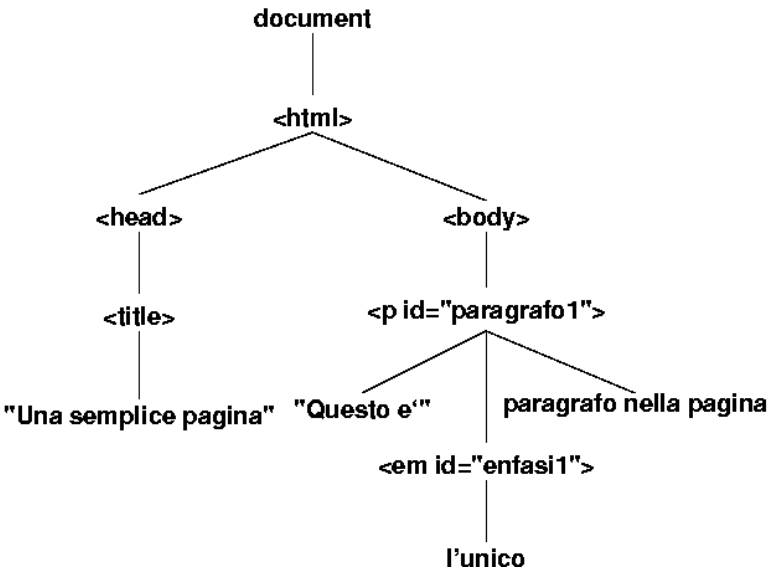


Figura 13.1: Nell’esempio, l’oggetto document ha un solo elemento annidato (è un figlio nella gerarchia di contenimento): l’elemento HTML. Questo, a sua volta, ha due elementi annidati: HEAD e BODY; head contiene solo TITLE che a sua volta contiene testo; BODY contiene un solo elemento P che a sua volta contiene tre elementi: testo, EM e testo.

Ciascun contenitore, elemento solo (ad esempio, un elemento BR) o parte di testo rappresenta un **nodo**. Se il codice sorgente del documento contiene una definizione di tipo di documento (DTD = Document Type Definition) prima dell’elemento HTML, essa diventa un fratello di HTML e un figlio di document.

Il DOM W3C fino al secondo livello prevede 12 tipi di nodi, di cui 7 si applicano ai documenti HTML:

Elemento , con indice 1, in cui il nome del nodo corrisponde all’etichetta associata all’elemento; questo tipo di nodo si applica quindi a tutti gli elementi HTML.

Attributo , con indice 2, avente per nome e per valore i corrispondenti nome e valore dell'attributo; questo tipo di nodo si applica ad ogni coppia nome-valore associata come attributo ad un elemento HTML.

Testo , con indice 3, ha come nome la stringa "#text" e come valore il contenuto del testo; si applica ad ogni frammento di testo contenuto in un elemento, compresi eventuali caratteri o sequenze di caratteri di formattazione.

Commento , con indice 8, ha come nome la stringa "#comment" e come valore il testo del commento; viene associato ai commenti HTML.

Document , con incide 9, ha come nome la stringa "#document" e valore `null`; viene associato alla radice del documento.

TipoDocumento , con indice 10, nome `DOCTYPE` e valore `null`, si associa a eventuali specifiche DTD.

Frammento , con indice 11. nome "#document-fragment" e valore `null` serve ad indicare serie di uno o più nodi fuori dal documento.

Nel primo esempio visto sopra, si avevano 1 nodo documento (tipo 9), 6 nodi elemento (tipo 1) e 4 nodi di testo (tipo 3).

Per quanto riguarda gli attributi associati ad ogni nodo, essi servono per indicare caratteristiche della parte di documento corrispondente al nodo. In particolare:

Proprietà	**Valore**	**Descrizione**
nodeName	string	Varia col tipo di nodo
nodeValue	string	Varia col tipo di nodo
nodeType	number	Costante corrispondente a ciascun tipo
parentNode	object	Riferimento al contenitore immediatamente più esterno
childNodes	array	Tutti i nodi figli nell'ordine in cui si trovano nel sorgente
firstChild	object	Riferimento al primo nodo figlio
lastChild	object	Riferimento all'ultimo nodo figlio
previousSibling	object	Riferimento al precedente fratello
nextSibling	object	Riferimento al successivo fratello
attributes	MappaNodo	Array di nodi attributi
ownerDocument	object	Riferimento al documento contenitore che funge da radice
namespaceURI	string	URI alla definizione dello spazio dei nomi (solo nodi elemento e attributo
prefix	string	prefisso nella definizione dello spazio dei nomi (solo nodi elemento e attributo
localName	string	Applicabile ai nodi interessati alla spazio dei nomi

Mentre per quel che riguarda i metodi:

METODO	DESCRIZIONE
appendChild(nuovoFiglio)	Aggiunge un nodo figlio alla fine del nodo corrente
cloneNode(profondità)	Prende una copia del nodo corrente (opzionalmente con figli)
hasChildNodes()	Determina se il nodo corrente ha figli (booleano)
insertBefore(nuovo,rif)	Inserisce un nuovo figlio davanti ad un altro figlio
removeChild(vecchio)	Cancella un figlio
replaceChild(nuovo, vecchio)	Sostituisce un vecchio figlio con uno nuovo
supports(funzione,versione)	Determina se il nodo supporta una determinata funzione

Un esempio di elaborazione del DOM con JavaScript (ECMAScript):

```
// access the tbody element from the table element
var myTbodyElement = myTableElement.firstChild;

// access its second tr element
// The list of children starts at 0 (and not 1).
```

```
var mySecondTrElement = myTbodyElement.childNodes[1];

// remove its first td element
mySecondTrElement.removeChild(mySecondTrElement.firstChild);

// change the text content of the remaining td element
mySecondTrElement.firstChild.firstChild.data = "Peter";
```

L'approccio utilizzato è decisamente orientato agli oggetti (OO): i documenti vengono modellati utilizzando oggetti, e il modello comprende non solo la struttura di un documento, ma anche il comportamento del documento e gli oggetti di cui è composto.

13.3.1 Esempi

Coi metodi visti si possono creare, distruggere o modificare nodi. Si tenga comunque presente che un nodo creato all'interno di un documento può venire inserito solo nel documento stesso.

Esplorazione della struttura a albero

```
<html>
	<head><title>Esempio: stampa dei nodi dell'albero</title>
		<script type="text/javascript">
			function cellAdd(item,riga){
				var colonna = document.createElement("TD");
				var contenuto = document.createTextNode(item);
				colonna.appendChild(contenuto);
				riga.appendChild(colonna);
			}

			function esplora(elem,tab){
				if(elem != null){
					var riga = document.createElement("TR");
					cellAdd(elem.nodeName,riga);
					cellAdd(elem.nodeValue,riga);
					cellAdd(elem.nodeType,riga);
					cellAdd(elem.childNodes.length,riga);

					tab.appendChild(riga);
					for(var i=0; i< elem.childNodes.length; i++){
						esplora(elem.childNodes[i],tab);
					}
				}
			}

			function stampa(tag){
				var elemList = document.getElementsByTagName(tag);
				var colonna = document.createElement("TH");
				var contenuto = document.createTextNode("nodeName");
				colonna.appendChild(contenuto);
				var riga = document.createElement("TR");
				riga.appendChild(colonna);
```

```
                        colonna = document.createElement("TH");
                        contenuto = document.createTextNode("nodeValue");
                        colonna.appendChild(contenuto);
                        riga.appendChild(colonna);

                        colonna = document.createElement("TH");
                        contenuto = document.createTextNode("nodeType");
                        colonna.appendChild(contenuto);
                        riga.appendChild(colonna);

                        colonna = document.createElement("TH");
                        contenuto = document.createTextNode("Numero figli");
                        colonna.appendChild(contenuto);
                        riga.appendChild(colonna);
                        var nuovoHead = document.createElement("THEAD");
                        nuovoHead.appendChild(riga);

                        var nuovoBody = document.createElement("TBODY");
                        for(var i=0; i<elemList.length; i++){
                                esplora(elemList[i],nuovoBody);
                        }
                        var nuovo = document.createElement("TABLE");
                        nuovo.appendChild(nuovoHead);
                        nuovo.appendChild(nuovoBody);
                document.getElementsByTagName("body")[0].appendChild(nuovo);
        }
                </script>
        </head>
        <body id="body">
                <input type="text" id="tag">
                <input type="button"
                onClick='stampa(document.getElementById("tag").value)'>

                <p id="pippo">Questa &egrave; una prova</p>
        </body>
</html>
```

Ordinamento di una tabella

```
<html>
        <head>
                <title>Esempio: ordinamento</title>

                <script type="text/javascript">
                        function compare(s1,s2) {
                                var out = 0;
                                if(s1 > s2) {
                                        out=1;
                                } else if (s1 < s2) {
                                        out=-1;
                                }
```

```
                        return out;
                    }

                    function sort(tab) {
                      var tbody = tab.childNodes[0];
                      if(tbody.nodeType == 3) {
                         tbody = tab.childNodes[1];
                      }
                      var rowList = tbody.childNodes;
                      for(var i=1; i<rowList.length;i++) {
                        if(rowList[i].nodeType == 1){ //elemento, non testo
                          j=i-1;
                          while(j >= 0 &&
                          (rowList[j].nodeType != 1 || \
                                          cmpRow(rowList[i],rowList[j]) > 0)) {
                            j--;
                          }

                          if(j>=0 && (rowList[j].nodeType == 1)) {
                                  var tmp = rowList[i].cloneNode(5);
                           rowList[0].parentNode.removeChild(rowList[i]);
                           rowList[0].parentNode.insertBefore(tmp,rowList[j+1]);
                                }
                      }

                          }
                        }

                    function cmpRow(r1,r2){
                      return compare(r1.firstChild.firstChild.nodeValue,
                                    r2.firstChild.firstChild.nodeValue);
                    }

                    function doCompare(){
                      var form = document.getElementById("compForm");
                      form.output.value = compare(form.stringa1.value,
                                                  form.stringa2.value);
                    }

                    function doSort(){
                           sort(document.getElementById("tab"));
                    }

                </script>
            </head>

<body>
<p>Proviamo la funzione compare:</p>
```

```
<form id="compForm">
<input type="text" id="stringa1" value="primo">
<input type="text" id="stringa2" value="secondo">
<input type="button" id="compare" onClick="doCompare()"><br/>
<input type="text" id="output" value="risultato">
</form>

<p>E ora vediamo come ordinare la seguente tabella:
<table id="tab">
<tbody>
<tr><td>zebra</td>
    <td>4</td>
    <td>mammifero</td>
    <td>a strisce</td>
    <td>erbivoro</td>
</tr>
<tr><td>ippopotamo</td>
    <td>4</td>
    <td>mammifero</td>
    <td>grigio</td>
    <td>erbivoro</td>
</tr>
<tr><td>rinoceronte</td>
    <td>4</td>
    <td>mammifero</td>
    <td>grigio</td>
    <td>erbivoro</td>
</tr>
<tr><td>scimpanz&egrave;</td>
    <td>4</td>
    <td>mammifero</td>
    <td>marrone</td>
    <td>onnivoro</td>
</tr>
<tr><td>elefante</td>
    <td>4</td>
    <td>mammifero</td>
    <td>grigio</td>
    <td>erbivoro</td>
</tr>
<tr><td>leone</td>
    <td>4</td>
    <td>mammifero</td>
    <td>giallino</td>
    <td>carnivoro</td>
</tr>
<tr><td>tigre</td>
    <td>4</td>
    <td>mammifero</td>
```

```
    <td>a strisce</td>
    <td>carnivoro</td>
</tr>
</tbody>
</table>
</p>
</body>

<input type="button" id="ord" onClick="doSort()">

</html>
```

Apertura di una finestra

```
<html>
  <head>
    <title>Esempio Javascript per la gestione delle finestre</title>
    <script language="JavaScript">
       function apri_finestra(url, width, height) {
         finestra = window.open(url,"", 'left=200, top=100, width=' + width + ',
                                                              height=' + height);
         finestra.focus();
       }
    </script>
  </head>

  <body>
    <div> click <a href="javascript:
            apri_finestra('http://people.na.infn.it/~corazza/',1000,700)">here</a>
  </body>
</html>
```

13.3.2 Attributi

Gli attributi associati ai tag di elemento vengono convertiti dal browser in proprietà dell'oggetto.

- *getAttribute(string name)*: torna il valore dell'attributo *name*
- *setAttribute(string name, string value)*: se c'è già un attributo con quel nome, ne cambia il valore, altrimenti aggiunge l'attributo.

```
<script type="text/javascript">
  function changeSize(px){
       var table = document.getElementById("tabDimensioni");
       table.setAttribute("width",px);
  }
</script>

<table id="tabDimensioni"
       width="300" cellspacing="2" cellpadding="2" border="1">
\begin{tabular}{ccc}
    one&
    two&
    three \\
    four&
```

```
    five&
    six \\
    seven&
    eight&
    nine
\end{tabular}

<button onclick="changeSize(100);">100px</button>
<button onclick="changeSize(200);">200px</button>
<button onclick="changeSize(300);">300px</button>

<script type="text/javascript">
  function changeSize(px){
        var table = document.getElementById("tabDimensioni");
        table.setAttribute("width",px);
  }
</script>
```

Le due istruzioni seguenti fanno la stessa cosa:

```
table.setAttribute("border","2");

table.border = 2;
```

È poi possibile controllare se l'attributo c'è già:

```
var table = document.getElementById("tableMain");
if (table.hasAttribute("border")){
  table.setAttribute("border","2");
} else {
  alert("Table has no border");
}
```

13.3.3 Elementi Style

Ogni regola CSS per un elemento è rappresentata come una proprietà dell'oggetto corrispondente all'elemento *style*.

```
<script type="text/javascript">
function inspectStyle(elm){
  if (elm.style){
    var str = "";
      for (var i in elm.style){
        str += i + ": " + elm.style[i] + "\n";
      }
      alert(str);
    }
}

var header = document.getElementById("h2");
inspectStyle(header);
</script>
```

```
<h2 id="headerSample"
       style="font-family:arial;font-size:16px;font-color:blue;background-color:yellow;">
Headline Here</h2>

<button onclick="inspectStyle(document.getElementById('headerSample'))">Inspect
       Header CSS</button>
```

Ovviamente, le proprietà che non sono state inizializzate sono vuote.

Se si vuole leggere il colore dello sfondo di una pagina HTML (o più precisamente del BODY) ed eventualmente modificarlo, si può usare uno script del tipo:

```
var doc = document.getElementsByTagName("body").item(0);
var color = doc.style.backgroundColor;
alert ("Background color is: " + color);

doc.style.backgroundColor = "\#0000ff";
```

Come esempio, si possono modificare colore di sfondo e posizione di una tabella andando a modificare le proprietà LEFT e BACKGROUND-COLOR del CSS.

```
<script language="JavaScript" type="text/javascript">
  function tableRight(){
    var table = document.getElementById("tableMain3");
    table.style.left = "100px";
  }

  function tableLeft(){
    var table = document.getElementById("tableMain3");
    table.style.left = "0px";
  }

  function changeTColor(col){
    var table = document.getElementById("tableMain3");
    table.style.backgroundColor = col;
  }
</script>

<table>
    one&
    two&
    three&
\\

    four&
    five&
    six&
\\

    seven&
    eight&
    nine&
\\
```

```
</table>

Change Position
  <button onclick="tableRight();">right 100px</button>
  <button onclick="tableLeft();">left 100px</button>


Change Color:
  <button onclick="changeTColor('\#ff0000');" style="background-color:\#ff0000;
                                                     width:30px; height:10;"></button>
  <button onclick="changeTColor('\#00ff00');" style="background-color:\#00ff00;
                                                     width:30px; height:10;"></button>
  <button onclick="changeTColor('\#0000ff');" style="background-color:\#0000ff;
                                                     width:30px; height:10;"></button>
  <button onclick="changeTColor('\#cccccc');" style="background-color:\#cccccc;
                                           width:30px; height:10;"></button>


<script language="JavaScript" type="text/javascript">
  function tableRight(){
    var table = document.getElementById("tableMain3");
    table.style.left = "100px";
  }

  function tableLeft(){
    var table = document.getElementById("tableMain3");
    table.style.left = "0px";
  }

  function changeTColor(col){
    var table = document.getElementById("tableMain3");
    table.style.backgroundColor = col;
  }
</script>

Change Position
  <button onclick="tableRight();">right 100px</button>
  <button onclick="tableLeft();">left 100px</button>


Change Color:
  <button onclick="changeTColor('\#ff0000');" style="background-color:\#ff0000;
                                     width:30px; height:10;"></button>
  <button onclick="changeTColor('\#00ff00');" style="background-color:\#00ff00;
                                     width:30px; height:10;"></button>
  <button onclick="changeTColor('\#0000ff');" style="background-color:\#0000ff;
                                     width:30px; height:10;"></button>
  <button onclick="changeTColor('\#cccccc');" style="background-color:\#cccccc;
                                     width:30px; height:10;"></button>
```

Poiché i nomi delle proprietà non possono contenere il trattino, occorre convertire le proprietà delle regole CSS in proprietà del DOM da elaborare con JavaScript.

```
<table>

  <td valign="top">CSS property&
  <td valign="top">JavaScript equivalent&
\\

  <td valign="top"><code>background-color</code>&
  <td valign="top"><code>style.backgroundColor</code>&
\\

  <td valign="top"><code>font-size</code>&
  <td valign="top"><code>style.fontSize</code>&
\\

  <td valign="top"><code>left</code>&
  <td valign="top"><code>style.left</code>&
\\

  <td valign="top"><code>border-top-width</code>&
  <td valign="top"><code>style.borderTopWidth</code>&
\\
</table>
```

13.3.4 Gestione degli eventi

Il modulo della DOM dedicato agli eventi si pone le seguenti finalità:

- permettere la registrazione dei rilevatori di eventi (**event listners**);
- descrivere il **flusso degli eventi** attraverso una struttura ad albero;
- mantenere fin dove possibile la compatibilità con il sistema di eventi dei browser che implementano la DOM di livello 0.

Con il termine **evento** si indica la rappresentazione di qualcosa che avviene in modo **asincrono**:

- click del mouse sulla presentazione di un elemento;
- cancellazione di un nodo figlio di un elemento;
- moltissime altre possibilità.

Possiamo dividere gli eventi in:

- **eventi UI**: eventi riguardanti l'interfaccia utente: questi eventi sono generati dall'interazione dell'utente attraverso un dispositivo esterno (mouse, tastiera, ecc.)
- **eventi logici UI**: eventi UI indipendenti dal dispositivo, come il cambio di focus.
- **eventi di mutazione**: eventi causati da un'azione che modifica la struttura del documento.

Ogni evento è associato ad un oggetto che è il suo **event target**.

Rilevatore di eventi: un meccanismo che dice a un oggetto di rispondere a un particolare tipo di evento:

AddEventListener() è un metodo associato ad ogni nodo per decidere se un evento deve essere forzato ad andare verso l'alto nella gerarchia o se deve essere catturato a un livello più alto.

Le funzioni chiamate dal rilevatore di eventi ricevono un singolo parametro costituito dall'oggetto evento le cui proprietà contengono dettagli contestuali sull'evento (quali la posizione di un clic del mouse, il codice di carattere di un tasto o un riferimento a un oggetto destinazione).

Rappresentazione grafica di un evento inviato nell'albero DOM usando il flusso degli eventi DOM:

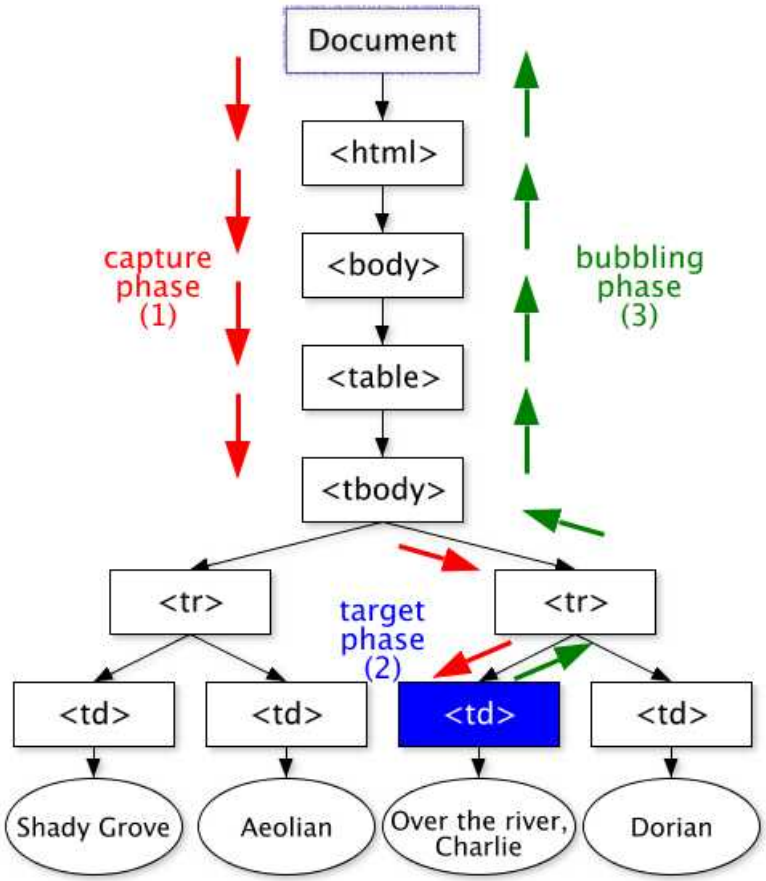


Figura 13.2: Flusso degli eventi (dal sito del W3C).

Per esempio, supponiamo di avere un pulsante per richiamare una funzione di calcolo:

```
document.getElementById("calcButton").addEventListener("click",doCalc,false)
```

con tre argomenti:

1. stringa che indica l'evento da rilevare
2. riferimento alla funzione da chiamare quando l'evento viene rilevato
3. valore booleano: se *true* rilevamento ogni volta che l'evento è diretto verso la sua destinazione (*cattura*)

```
function doCalc(event) {
  // dall'evento ricavo un riferimento al form
  var form = event.target.form;
  var results = 0;
  // conti vari
  form.result.value = results;
}
```

- **Flusso di propagazione degli eventi**: processo secondo il quale un evento ha origine nell'implementazione del DOM e viene passato all'interno del DOM.
- **Target dell'evento**: nodo verso il quale l'evento è diretto (proprietà *target* del nodo *Event*)

- Tutti gli EventListener sul target vengono attivati, ma l'ordine non è specificato.
- Un EventListener può bloccare l'ulteriore propagazione dell'evento chiamando il metodo *StopPropagation* dell'interfaccia *Event* oltre il nodo corrente, sia in discesa che in salita (ma gli EventListner registrati su quel nodo ricevono l'evento).
- Durante la fase di bubbling, vengono attivati tutti gli EventListner trovati fino al document compreso, ma escludendo gli EventListener registrati per la cattura.
- Anche se l'albero viene modificato durante l'elaborazione dell'evento, il flusso dello stesso procede seguendo lo stato **iniziale** dell'albero.
- **Cancellabile**: questo termine indica che gli EventListener registrati per l'evento considerato possono cancellare le azioni associate all'evento di default (ad esempio, l'attivazione di un hyperlink).

Nella gestione degli eventi bisogna fare attenzione ai tempi in cui vengono messi in esecuzione gli script rispetto al momento in cui si scatena l'evento. Gli script introdotti in un documento HTML o XML hanno una semantica di tipo **"run-to-completion"**: vengono completati prima che venga intrapresa qualsiasi altra azione, come ad esempio quella di scatenare un evento o di continuare al analizzarlo.

Il processo di analisi del documento, invece, avviene in modo **asincrono** ed **incrementale** e può quindi subire interruzioni per permettere l'esecuzione di script.

Occorre quindi fare attenzione a non perdere eventi. Ci sono due modi sicuri in questo senso:

1. aggiungere un gestore di eventi e impostare i suoi attributi in quell'istruzione;
2. creare sia l'elemento che il relativo gestore di eventi **nello stesso script** in modo che le due azioni vengano eseguite in un blocco unico.

13.3.5 Esempio

Il problema si può ad esempio presentare con elementi di tipo *img* e l'evento *load*: l'evento infatti potrebbe essere lanciato non appena l'elemento è stato lanciato, soprattutto se l'immagine era in cache, come spesso succede.

```
<img src="games.png" alt="Games" onload="gamesLogoHasLoaded(event)">
```

I seguenti sono modi **sicuri**:

```
<script>
 var img = new Image();
 img.src = 'games.png';
 img.alt = 'Games';
 img.onload = gamesLogoHasLoaded;
 // img.addEventListener('load', gamesLogoHasLoaded, false); // would work also
</script>
```

Se invece la creazione dell'evento viene fatta fuori dallo script si rischia di perdere degli eventi che vengono lanciati tra la creazione dell'elemento e l'esecuzione dello script:

```
<!-- Do not use this style, it has a race condition! -->
 <img id="games" src="games.png" alt="Games">
 <!-- the 'load' event might fire here while the parser is taking a
      break, in which case you will not see it! -->
 <script>
  var img = document.getElementById('games');
  img.onload = gamesLogoHasLoaded; // might never fire!
 </script>
```

14. AJAX e Single Page Applications

Asynchronous JavaScript and XML (AJAX) non è di per sè una tecnologia, ma è un termine che descrive un "nuovo" approccio all'utilizzo di diverse tecnologie esistenti, compresi: HTML o XHTML, CSS, JavaScript, DOM, XML, XSLT e l'oggetto XMLHttpRequest. Grazie all'utilizzo di queste tecnologie in combinazione con il modello AJAX, le applicazioni web possono eseguire aggiornamenti rapidi e **incrementali** dell'interfaccia utente senza ricaricare nel browser l'intera pagina. Questo rende l'applicazione più reattiva alle azioni dell'utente.

Su questo principio sono basate le SPA (Single-page application). Tali applicazioni caricano un singolo documento web e poi lo modificano secondo la necessità.

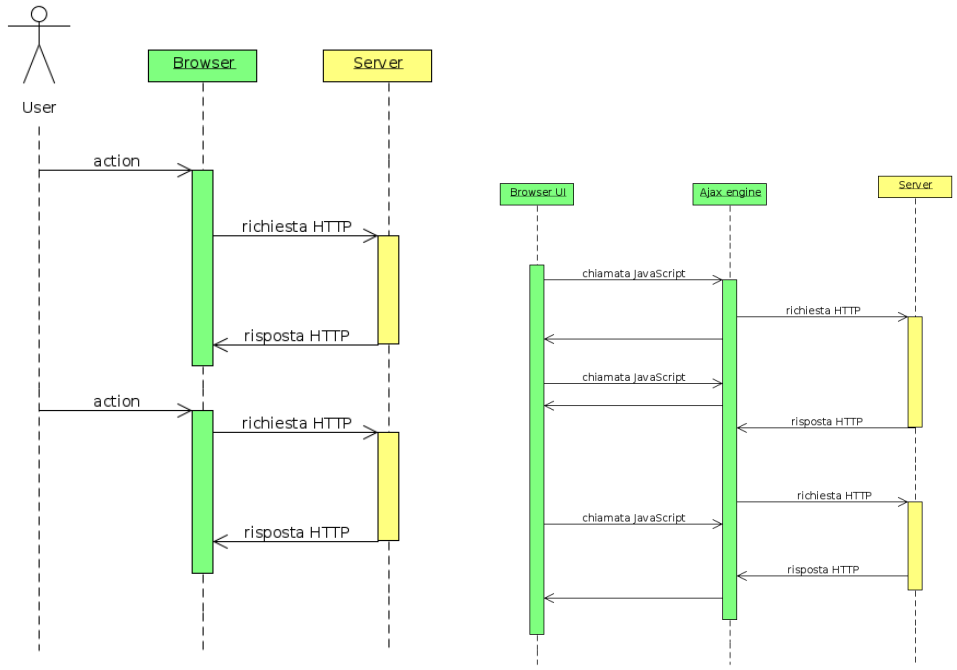
14.1 Motivazioni

Nelle classiche applicazioni web una richiesta HTTP viene preparata e inviata a seguito di una qualche azione dell'utente. Il server web reagisce alla richiesta con un'opportuna risposta, sempre HTTP, a cui il browser reagisce preparando il materiale da presentare all'utente. Questo funzionamento è **sincrono** rispetto alle azioni dell'utente.

Tuttavia, molto spesso non tutte le azioni dell'utente devono necessariamente richiedere un contributo da parte del server. A volte basta un'azione lato client, che può ad esempio essere risolta con uno strumento di programmazione lato client, quale JavaScript o una applet. A volte, inoltre, la maggior parte della pagina rimane invariata, e quindi basta aggiornarne solo una piccola parte. AJAX permette di programmare un'interazione **asincrona** rispetto alle azioni dell'utente, dove una richiesta HTTP viene preparata e inoltrata solo quando risulta necessario per l'applicazione nel suo complesso.

14.2 Precursori

Prima di AJAX si usava un trucco per ottenere un comportamento asincrono da parte del browser: un **frame fittizio** di dimensioni nulle conteneva un form che veniva riempito e sottomesso mediante JavaScript, senza che l'utente se ne avvedesse direttamente. Quando la risposta HTTP tornava al frame nascosto, essa conteneva un altro script JavaScript che comunicava allo script di partenza



(a) (b)

Figura 14.1: (a) Scambio sincrono di una tradizionale applicazione web; (b) scambio asincrono reso possibile dal modello AJAX; in verde il client, in giallo il server.

l'avvenuta ricezione dei risultati. Netscape Navigator 2.0 è stato il primo browser a supportare contemporaneamente frame e JavaScript, rendendo cosí possibile il trucco.

Con l'introduzione dell'elemento `<iframe/>` in HTML 4.0, che permetteva l'inserimento, in un documento HTML di frame invisibili, eventualmente con CSS oppure dinamicamente con la DOM, il meccanismo risultava ancora più facilitato.

Nel 2001, Microsoft ha introdotto un **oggetto ActiveX** chiamato XMLHttp per facilitare l'interazione tra client e server, in coppia con un'estensione di JavaScript che permetteva la creazione di controlli ActiveX. Tale oggetto è stato poi inserito nella libreria MSXML, per il supporto a XML. In realtà, l'oggetto permetteva di fare ben di più di una semplice manipolazione XML, in quanto assomigliava più ad una richiesta HTTP ad hoc, in cui il programmatore aveva il controllo sulla request line e sull'intestazione e poteva poi elaborare la risposta fornita dal server, che poteva contenere XML, HTML, oggetti JavaScript serializzati o dati in qualsiasi altro formato.

Da parte sua, Mozilla creò l'oggetto host del browser XMLHttpRequest, che sostanzialmente copiava tutte le funzionalità del controllo ActiveX, spingendo poi sia Opera che Safari ad adottarlo.

Oggi come oggi, Google sta favorendo molto lo sviluppo dell'approccio Ajax, che sta risultando vincente ed è ormai molto diffuso. Da parte sua, anche la SUN si sta muovendo per produrre strumenti che facilitino l'uso di AJAX.

14.3 Ruolo dei diversi componenti

In generale Ajax permette di ottimizzare lo scambio di dati tra client e server andando a richiedere solo i dati necessari a costruire la parte della pagina che va modificata anziché ricaricare ogni volta tutta la pagina. Come accennato sopra, il modello Ajax si basa sugli strumenti seguenti:

JavaScript per permettere la comunicazione col browser e per rispondere ai diversi eventi;

DOM per poter accedere e modificare la struttura della pagina HTML o XHTML;

XML per rappresentare i dati che vengono scambiati tra server e client;
L'oggetto XMLHttpRequest per permettere uno scambio asincrono dei dati tra client e server.

Il diagramma di Figura 14.2 mostra il funzionamento di Ajax. In particolare, vengono

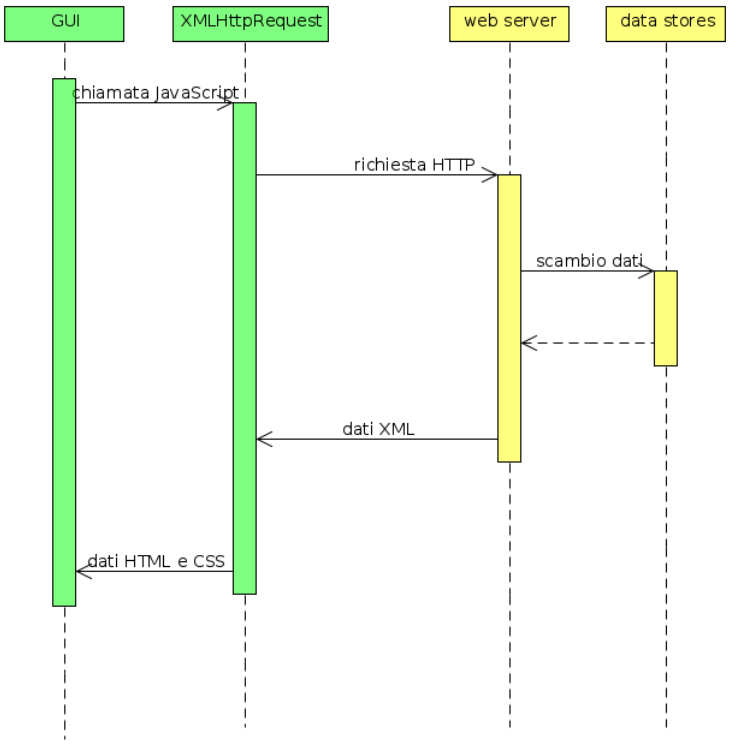


Figura 14.2: Sequenza del modello Ajax: in verde il client, in giallo il server.

considerati i seguenti passi:

1. a seguito di un evento generato dall'utente, che ad esempio preme un bottone, viene eseguita una chiamata JavaScript;
2. viene creato e configurato un oggetto XMLHttpRequest; in particolare viene impostato un parametri di richiesta che include l'identificativo del componente che ha generato l'evento e qualsiasi valore immesso dall'utente;
3. l'oggetto XMLHttpRequest manda al server una richiesta HTTP in modo asincrono rispetto all'azione dell'utente. Il server web elabora la richiesta come farebbe normalmente (ad esempio, attraverso un CGI, o una servlet) e, se necessario, salva i dati passati nella richiesta in modo persistente;
4. il server web prepara quindi una risposta HTTP nel cui body viene inserito un documento XML con tutti gli aggiornamenti che sono stati richiesti dal client;
5. l'oggetto XMLHttpRequest riceve i dati XML, li elabora e aggiorna opportunamente la DOM della pagina (X)HTML visualizzata dal browser.

In realtà, anche altre tecnologie sono usalmente presenti in uno strumento per AJAX:

- **HTML/XHTML**
- CSS, per definire lo stile di XHTML
- **DOM**: Dynamic updating of a loaded page
- XML
- XSLT per trasformare XML in XHTML
- XMLHttp o XMLHttpRequest

- **JavaScript**

e poi ovviamente servono strumenti per la programmazione lato server.

Per il trasporto degli aggiornamenti, si può utilizzare anchde JSON al posto di XML. In questo caso la conversione in HTML è basata su di un template.

14.3.1 L'oggetto XMLHttpRequest

L'oggetto XMLHttpRequest risulta centrale in tutto il meccanismo, perché rappresenta un'interfaccia che permette ai programmi (o agli scripts) di implementare funzionalità proprie di un client HTTP, quali ad esempio quella di sottomettere i dati di un form o di caricare dati da un sito web remoto.

Benché l'oggetto XMLHttpRequest venga ormai implementato da un certo numero di browser, queste implementazioni spesso non sono completamente interoperabili. Il W3C ha quindi specificato un minimo insieme di caratteristiche interoperabili basato sulle implementazioni esistenti, in modo che gli sviluppatori possano utilizzarle senza dover sviluppare del codice che sia dipendente dalla piattaforma.

In ECMAScript, si crea un'istanza di `XMLHttpRequest` usando il costruttore:

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

14.4 Single Page Applications (SPA)

Quindi si parla di Single Page Applications (SPA) quando un'applicazione o sito web parte da una singola pagina web che poi aggiorna dinamicamente con dati ottenuti dal server. Si basa su HTML, JavaScript (e DOM) e CSS. AJAX, con l'oggetto XMLHttpRequest, è il principale approccio utilizzato in questo tipo di applicazioni. Altre possibilità sono strumenti che permettono di aprire una comunicazione col server in cui anche quest'ultimo può prendere l'iniziativa, come i WebSockets (previsti da HTML5), gli eventi spediti dal server. Sarebbe anche possibile usare dei plugin del browser, come Silverlight, Flash o le Java applets, ma si tratta di tecnologie ormai superate.

Vantaggi:

- non occorre ricaricare sempre la pagina completa, con un miglioramento delle prestazioni e della dinamicità dell'applicazione

Svantaggi:

- rende più difficile l'indicizzazione del sito da parte dei motori di ricerca
- problemi nella navigazione (il back va gestito dal browser)
- problemi nell'analisi del funzionamento del sito
- in generale, è più difficile da gestire

Alcuni web frameworks, oggi molto popolari, si basano su questo approccio:

- React
- Angular
- Vue.JS

14.4.1 Esecuzione in locale

Data un'architettura di questo tipo, è anche possibile pensare di costruirla in modo che si possa scaricare ed eseguire in locale, senza dover contare sulla connettività verso il server. In questo caso si usa il Web Storage previsto da HTML5 per salvare l'informazione che altrimenti andrei a cercare sul server.

15. Programmazione su web

Per approfondire la programmazione su web, si faccia riferimento al Capitolo 19, “Web search basics” e per la parte sui crawlers, alle sezioni 20.1 2 20.2 (con le relative sottosezioni) del Capitolo 20, di [MRS08], disponibile sia in formato pdf e html. .

15.1 Programmazione su web: bots, spiders e crawlers.

In questa lezione ci occuperemo di come accedere alle informazioni rese disponibili sul web (quindi via HTTP) non attraverso un browser, ma attraverso lo sviluppo di altri tipi user-agent. Considereremo il linguaggio Java non perché sia l’unico linguaggio possibile per risolvere il problema, ma perché risulta comunque una buona scelta visto il supporto built-in che fornisce per il protocollo HTTP e per il parsing di documenti HTML. Per approfondimenti si considerino [Hea06; MRS08].

Come primo esempio di programmazione per il web, consideriamo un semplicissimo Server web che, a fronte di qualsiasi richiesta HTTP, risponde con una pagina HTML fissa.

Definizioni

Bot: un programma che lavora in internet e raccoglie informazioni da specifiche locazioni.

Agent: un programma a cui l’utente può chiedere di raccogliere informazioni di suo interesse.

Aggregator: un bot che combina l’informazione raccolta da un numero anche grande di siti contenenti informazioni analoghe e le organizza in modo aggregato per presentarle all’utente.

Spider: un bot ottimizzato per visitare un numero molto grande di pagine web, scelte in modo dinamico.

Spiders

Nell’articolo Jeff Heaton[1] riporta e spiega un semplice spider che controlla se il sito considerato contiene dei link non più attivi.

La parte più significativa dello spider risiede nel metodo `spiderFoundURL`, che viene richiamato quando lo spider trova un nuovo URL. Per prima cosa, controlla la validità del nuovo URL e

[1] Disponibile a `http://www.developer.com/java/other/article.php/1573761`

riporta un eventuale errore, restituendo <tt>false</tt>. Se invece il link è valido e se si riferisce allo stesso server, allora il metodo torna <tt>true</tt> e l'URL verrà seguito e espanso.

Il passo ulteriore consiste nello scaricare il contenuto dell'URL e analizzarlo alla ricerca di ulteriori link da considerare:

Si noti come ho la possibilità di controllare il `ContentType` del documento (in particolare, in questo caso, controllo che cominci con "text/" e sia quindi testuale.

Per leggere il contenuto del file basta aprire uno stream:

```
InputStream is = connection.getInputStream();
Reader r = new InputStreamReader(is);
```

Per analizzare il documento HTML:

```
HTMLEditorKit.Parser parse = new HTMLParse().getParser();
parse.parse(r,new Parser(url),true);
```

Per estrarre i link:

```
String href = (String)a.getAttribute(HTML.Attribute.HREF);
```

Parsing dell'HTML

Problema: sintassi non precisa e consistente.

La libreria Swing di Java comprende un parser HTML: può venire usata per estrarre

1. le informazioni che servono allo specifico bot e
2. i link necessari a continuare l'esplorazione della rete.

Consideriamo il problema di estrarre i link da una pagina. Dovremo considerare tutti gli elementi che ricadono nelle seguenti categorie:

link : è il caso più ovvio, per cui vanno estratti gli attributi `href` di tutti gli elementi `a`;

scripts : all'interno degli scripts possono essere contenuti degli URL a cui collegarsi;

image, maps : le immagini e le mappe possono fare riferimento ad un URL da cui ricavare l'immagine; inoltre, sia mappe che immagini possono comprendere una o più zone in cui un evento di tipo click rimanda ad un altro URL;

forms : il tasto submit delle forms rimanda all'URL dato dal valore dell'attributo `action` dell'elemento `form`;

commenti : in linea di massima, possono essere saltati; tuttavia, come abbiamo visto a proposito di JavaScript, essi possono anche contenere uno script.

Programmare su Web

Crawler

Un *crawler* è uno spider specializzato nell raccolta di informazioni sulla rete, allo scopo di indicizzarla per un motore di ricerca. Il suo scopo è quindi quello di raccogliere in modo rapido ed efficiente il più grande numero possibile di pagine, insieme con la struttura che le interconnette. In particolare deve essere:

1. Robusto rispetto a eventuali *spider traps*, pagine web preparate allo scopo di depistarli e bloccarli in cicli infiniti.
2. Rispettoso delle politiche implicite ed esplicite imposte dagli amministratori dei siti.

e inoltre:

1. Distribuito.
2. Scalabile.
3. Ottimizzato in termini di prestazioni e di efficienza.
4. In grado di focalizzarsi sulle pagine "utili".

5. Lavorare in modo continuo, mantenendo l'indice il più possibile aggiornato.
6. Facili da estendere verso nuovi formati, protocolli, ecc (quindi, modulari).

Passi:

1. Insieme dei semi (seed set): uno o più URL di partenza va a costituire la *URL frontier* iniziale;
2. Sceglie un URL dalla frontiera e importa la pagina corrispondente.
3. La pagina viene analizzata (parsing) e ne viene estratto il testo e i link (ognuno dei quali corrisponde ad un nuovo URL.
4. Il testo estratto viene usato per costruire l'indice inverso.
5. I link, invece, vengono aggiunti alla frontier, che consiste quindi in un insieme di link pronti per essere analizzati.

Il processo, nel suo insieme, non è altro che una visita del web graph.

Architettura:

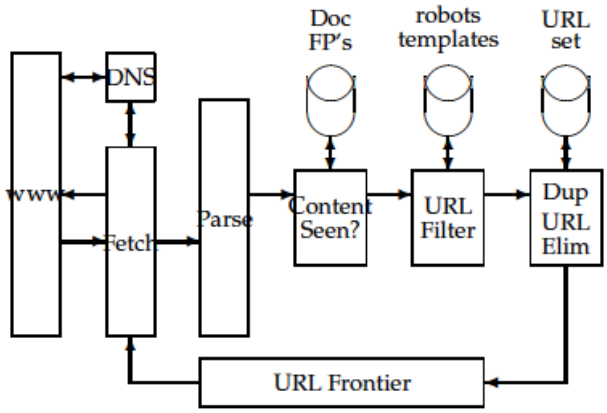


Figura 15.1: Architettura di un crawler

1. La *URL frontier*, contenente gli URL da esaminare (nel caso di crawling continuato, possono appartenere alla frontier anche URL già esaminati).
2. Un modulo di risoluzione DNS, che restituisce l'IP del server a cui richiedere la pagina da esaminare (rischio di rallentamenti, caching; sincrono blocca tutto, multi-thread: un DNS server locale che organizza e inoltra le richieste).
3. Un modulo che prepara la richiesta HTTP, la inoltra e analizza la risposta (tra l'altro, alla ricerca di possibili duplicazioni).
4. Un modulo di parsing che estrae dalla pagina i link da esaminare al prossimo passo.
5. Un modulo che aggiorna la frontier con gli URL estratti (aggiungendoli se già non ci sono, passandoli al filtro – robots.txt in cache, de-relativizzandoli, assegnando una priorità).

Robots Exclusion Protocol previsto da molti host sul web (*robots.txt* alla root del sito).

```
User-agent: *
Disallow: /yoursite/temp/
User-agent: searchengine
Disallow:
```

Hammering e contromisure che un webmaster può adottare

Bot exclusion file

Come identificare un bot assillante: user agent name, frequenza di accesso da un certo IP, modalità di accesso (ad esempio, solo file di testo)

Possibili contromisure: segnalazioni pubbliche, vie legali, bot exclusion file, filtri basati sull'IP, filtri basati sull'agent name.

16. Servizi web

Per approfondimenti in generale per quello che si intende per Web Science e più in particolare sul ruolo dei servizi web all'interno del web stesso, si consiglia la lettura di [Ber+06]. In generale, dettagli più tecnici sui servizi web sono disponibili al solito a partire dal sito del W3C.

16.1 Web Semantico (dati) e Servizi Web (programmi)

Da [Ber+06]:

> *Il Web Semantico può essere visto come il tentativo di collegare i* **dati** *presenti nelle diverse basi di dati, mettendoli in relazione con modelli del mondo, le* **ontologie**. *In questo modo diventa possibile aggregare e analizzare i dati producendo cosí interpretazioni consistenti attraverso sorgenti di dati eterogenee.*
>
> In qualche modo, l'idea è quella di raccogliere tutti i **dati** disponibili sul web in una enorme base di dati che trascenda i singoli componenti. In questo modo, le applicazioni possono inferire usando dati eterogenei.
>
> Ruolo centrale del **Resource Description Framework** (RDF) che integra tutta una serie di applicazioni usando l'XML come sintassi di interscambio.
>
> Stretta relazione tra RDF e le basi di dati relazionali: i records delle tabelle corrispondono ai nodi RDF, i campi a proprietà RDF e i valori nei campi di ogni record ai valori in RDF.
>
> I **servizi web** rappresentano un mezzo standard per interoperare tra diverse applicazioni software, eseguiti su piattaforme e/o contesti diversi.
>
> Scambio di dati e servizi non solo tra applicazioni diverse, ma addirittura tra imprese diverse.
>
> Diffusione sempre più ampia dell'uso di XML per operazioni remote tra compagnie diverse.
>
> Pensiamo al seguente **caso d'uso**: un'agenzia viaggi vuole automatizzare il servizio ai clienti, interfacciandosi sia a chi offre i servizi necessari per il viaggio (compagnie aeree, ferroviarie e di trasporto) che agli alberghi e inoltre ai servizi di carte di credito.

Diversi dagli strumenti usati in precedenza per operazione remote (CORBA, DCOM, RMI) perchè le transazioni sono meno frequenti, più lente e coinvolgono controparti la cui affidabilità non è necessariamente comprovata.

Assumono quindi particolare rilievo aspetti quali la conferma del ricevimento e la conservazione dei documenti anche per anni.

I servizi web poggiano sui seguenti strumenti:

- **XML**: per serializzare l'informazione;
- **SOAP**: protocollo su HTTP per lo scambio dei messaggi;
- **WSDL**: per descrivere l'interfaccia in un formato che sia elaborabile automaticamente.

Un **servizio Web** è una nozione astratta (insieme di funzionalità) che deve essere implementato da un **agente**, ovvero un pezzo di software o hardware che spedisce e riceve messaggi.

In due momenti diversi, lo stesso servizio Web può essere implementato da due agenti diversi, magari realizzati con due diversi linguaggi di programmazione.

- **Provider**: fornisce i servizi
- **Requester**: (quasi sempre) richiede i servizi; in ogni caso ne fruisce

- **Entity**: persona o organizzazione
- **Agent**: programma

Occorre prima di tutto un accordo tra provider entity e requester entity su **semantica** e **meccanismo** dello scambio di messaggi.

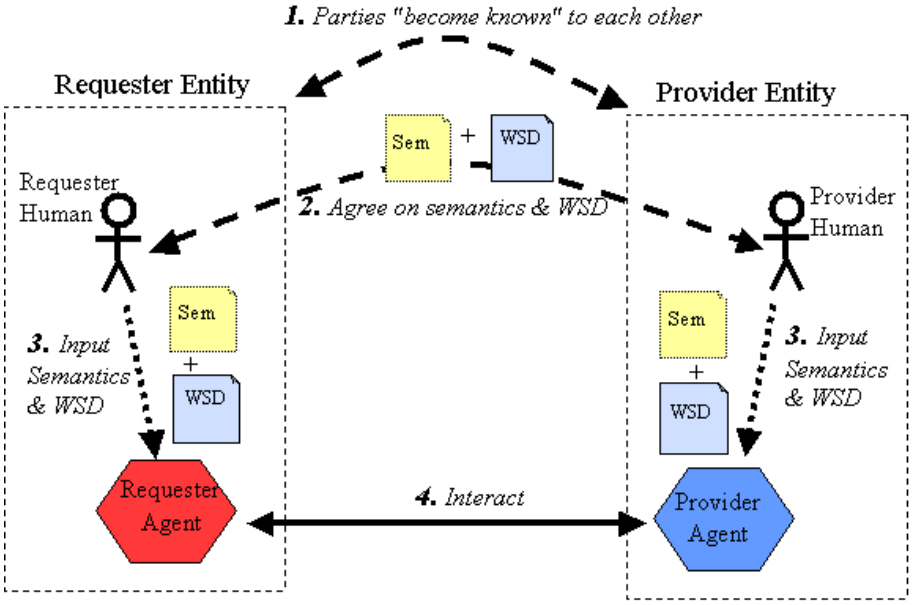
Per la descrizione del meccanismo dello scambio dei messaggi si usa un linguaggio per la descrizione dei servizi Web: il **Web Service Description Language (WSDL)**.

WSDL permette di descrivere il servizio, e rappresenta un accordo che sta alla base dei meccanismi di interazione con quel servizio:

- formato dei messaggi, tipo dei dati, protocolli di trasporto, formati di serializzazione usati nel trasporto tra agente requester e agente provider;
- una o più locazioni in cui un agent provider può venir invocato, dando indicazioni sullo scambio di messaggi che si aspetta.

La **semantica** di un servizio Web è invece data dalle aspettative comuni sul comportamento del servizio, in particolare in risposta ai messaggi inviati. Coinvolge le **entità** più che gli agenti, e non è necessariamente esplicito.

Le tre figure seguenti sono prese dal sito del W3C. La prima schematizza l'*interazione requester/provider* nel caso statico.



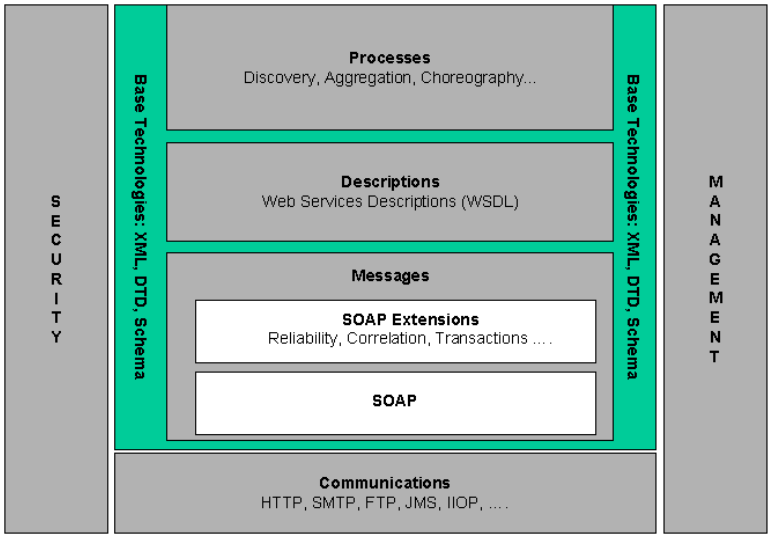
I quattro passi di figura sono necessari, ma non sufficienti: vari scenari possono richiedere ulteriori passi, o raffinamenti dei passi presentati in figura; inoltre l'ordine in cui vengono eseguiti i vari passi può variare di situazione in situazione.

1. Requester e provider diventano noti l'uno all'altro; due casi:
2. inizia l'agente del requester, e deve ottenere l'indirizzo dell'agente del provider, o direttamente dall'entità del provider, o attraverso un *discovery service* che fornisce l'indirizzo attraverso una descrizione funzionale che può essere esaminata manualmente o automaticamente;
3. quando invece (più di rado) a cominciare l'interazione è il provider, esso è venuto a conoscenza dell'indirizzo del requester in un modo che dipende dall'applicazione.
4. accordo sulla descrizione del servizio (un documento WSDL), non necessariamente esplicito: questo passo può anche precedere il precedente, del tutto o in parte;
5. l'agente del provider e quello del requester devono implementare le funzionalità loro richieste;
6. L'agente del requester e quello del provider si scambiano messaggi SOAP da parte dei rispettivi proprietari.

I servizi Web (SOAP/WSDL) non sono necessariamente la miglior soluzione per implementare un'architettura orientata ai servizi: COM e CORBA sono alternative.

I servizi Web risultano vantaggiosi quando:

- devono operare in Internet dove affidabilità e velocità non sono garantite;
- non è possibile garantire che requester e provider siano aggiornati contemporaneamente;
- si ha una disomogeneità nelle piattaforme su cui vengono eseguite le diversi componenti del sistema distribuito;
- un'applicazione esistente deve essere esposta sulla rete e può venir "impacchettata" come un servizio Web.



SOAP

SOAP impacchetta e scambia messaggi XML.

I messaggi SOAP possono venir trasportati da una varietà di protocolli, tra cui HTTP, SMTP, FTP, RMI/IIOP e anche protocolli proprietari.

SOAP è diventato un nome, e non più un acronimo; tuttavia:

- **Service Oriented Architecture Protocol**: un messaggio SOAP rappresenta l'informazione necessaria per invocare un servizio o riflette il risultato dell'invocazione di un servizio e contiene l'informaizone specificata nella definizione dell'interfaccia del servizio.
- **Simple Object Access Protocol**: meccanismo per invocare oggetti remoti, serializzando la lista degli argomenti che deve essere trasportata dall'ambiente locale a quello remoto.

WSDL

Linguaggio per **descrivere** servizi Web.

La descrizione comincia coi messaggi scambiati tra gli agenti del requester e del provider.

Le definizioni dei servizi Web possono poi venir mappate verso un qualsiasi linguaggio di programmazione, piattaforma, modello di oggetti o sistemi di messaggi.

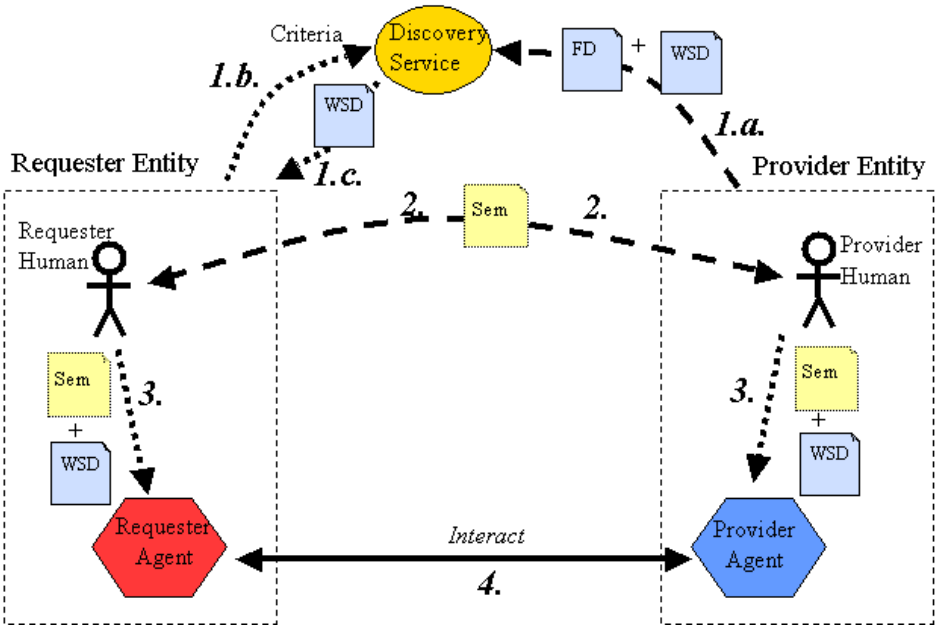
Possibili implementazioni usano COM, JMS, CORBA, COBOL, o una qualsiasi soluzione proprietaria: a patto che mittente e ricevente si accordino sulla descrizione del servizio (ad esempio, in un file WSDL), le implementazioni dietro i servizi Web possono essere qualunque.

Scoperta di servizi Web

Per **scoperta** (discovery) si intende la localizzazione di una descrizione elaborabile in modo automatico di un **servizio Web** che, pur sconosciuto fino a quel momento, risponde a criteri funzionali dati.

Per **servizio di scoperta** (discovery service) si intende un servizio che aiuta nel processo di scoperta; può essere implementato sia dall'agente provider che dall'agente requester o da un altro agente.

La seguente figura, anch'essa presa dal sito del W3C, schematizza il processo di scoperta:



1. requester e provider divengono noti l'uno all'altro
2. Il servizio di discovery ottiene sia la descrizione del servizio (WSD = Web Service Description) che la descrizione funzionale associata (FD = Functional Description).
   La complessità della descrizione funzionale è molto variabile: può andare da poche parole di metadati o un URI all'uso di TModel (in UDDI) o una collezione di istruzioni RDF, DAML-S o OWL-S.
   La descrizione del servizio può essere ottenuta in diversi modi: ad esempio con un search engine, che naviga nel Web raccogliendo tutte le descrizioni di servizi che trova, anche senza l'intervento del provider; oppure con un registro (ad esempio, UDDI), dove si richiede al provider stesso di pubblicare la descrizione e la descrizione funzionale del servizio per il servizio di discovery.
3. I criteri di ricerca specificati dal requester al servizio di discovery per selezionare un servizio Web basato sulla descrizione funzionale ad esso associata, o altre caratteristiche quali il nome del provider o altre caratteristiche del provider.
4. Il servizio di discovery produce una o più descrizioni di servizi che soddisfano i requisiti richiesti. Nel caso in cui siano prodotte più proposte, il requester ne seleziona una.
5. Requester e provider si accordano sulla semantica dell'interazione desiderata: di solito il provider la definisce lasciando al requester l'opzione prendere o lasciare. Si rende opportuno la definizione di standard.
   Devono essere d'accordo anche sulla descrizione del servizio, ma questo di fatto è già avvenuto nel passo 1.c.
6. La descrizione del servizio e la sua semantica sono passate agli agenti del requester e del provider.
7. Avviene lo scambio di messaggi SOAP tra i due agenti, da parte dei rispettivi proprietari.

Scoperta manuale o autonoma?

- Scoperta **manuale**: l'umano, tipicamente in fase di sviluppo, cerca la descrizione di un servizio che soddisfi i suoi desiderata.
- Scoperta **autonoma**: è l'agente del requester che esegue questa ricerca, sia essa eseguita durante lo sviluppo o l'esecuzione.

Ovviamente i due casi richiedono condizioni molto diverse:

- **Requisiti di interfaccia**: in un caso orientati all'umano, nell'altro ad una elaborazione automatica.
- Necessità di **standardizzazione**: più importante per scoperta autonoma
- **Fiducia**: non è detto che gli umani si fidino di decisioni prese dalle macchine.

Per poter arrivare ad una discovery autonoma, è necessaria una semantica che possa venir elaborata automaticamente.

Si pensi anche a situazioni in cui la discovery non parte da zero, ma da un servizio resosi ad un certo punto per qualche ragione inaccessibile.

Discovery: registro, indice o peer-to-peer?

Un **registro** (registry) rappresenta un deposito di descrizioni di servizi controllato centralmente e autorevole:

- Viene richiesto al provider un passo preciso: deve collocare l'informazione riguardante il servizio nel registro perchè questa possa essere disponibile per gli altri.
- Di conseguenza è il provider a decidere **chi** ha l'autorità per rendere disponibile l'informazione, che non può quindi essere gestita da terze parti.
- Inoltre, è sempre il provider che decide **quale** informazione va messa nel registro: terze parti non possono integrare questa informazione in alcun modo.

UDDI può essere usato sia come registro che come indice.

L'**indice** si differenzia dal registro perchè pur essendo una raccolta di informazioni esterna non ha nè controllo centrale nè, di conseguenza, autorevolezza.

In questo caso la pubblicazione è passiva: il provider espone sul Web il servizio e la sua descrizione funzionale, e sarà chi prepara gli indici a raccogliere queste informazioni senza che il provider ne venga necessariamente informato.

- Chiunque può preparare un indice, raccogliendo le informazioni usando degli spider e organizzandola in un indice.
- Organizzazioni diverse possono quindi avere indici di questo tipo.
- L'informazione contenuta in un indice può risultare superata: occorre controllarla prima dell'uso. Per questo motivo conterrà riferimenti a informazioni autorevoli che rendono possibile il controllo.
- un indice può comprendere anche informazioni di terze parti
- indici diversi posso fornire informazioni di tipo diverso, alcuni più completa, altri meno;
- saranno le leggi di mercato a decidere quale sarà l'indice effettivamente usato

Un esempio di approccio a indice può essere rappresentato da motori come Google.

La differenza più importante tra registro e indice riguarda chi **controlla** l'informazione che vi è contenuta: in un caso il proprietario del registro, nell'altro le forze di mercato, che fanno aumentare o diminuire la popolarità di un servizio.

L'approccio **peer-to-peer** non ha repositories centralizzate: sono i servizi che si scoprono vicendevolmente in modo dinamico passandosi le richieste sulla rete.

È più robusto, perchè non dipende da un solo punto centrale (o pochi punti centrali).

Ogni nodo potrebbe contenere un proprio indice dei servizi Web esistenti.

Il tempo di latenza è minore, perchè i nodi vengono contattati direttamente.

Svantaggi tipici delle applicazioni peer-to-peer: inefficienze e overhead, perchè ogni nodo si fa carico di ricevere e propagare richieste che il più delle volte non lo interessano e nessuna garanzia di trovare il servizio voluto.

In conclusione:

- i sistemi **P2P** saranno da preferire in ambienti dinamici, in cui vincoli di vicinanza limitano comunque la propagazione delle inondazioni di richieste (ubiquitous computing);
- i **registri** centralizzati si adattano meglio ad ambienti statici e ben controllati, in cui l'informazione non cambia spesso;
- gli **indici**, infine, sono in grado di adattarsi meglio al crescere delle dimensioni del problema e sostengono la competizione e la differenziazione delle strategie di ricerca.

Quando l'informazione viene raccolta da registri o indici multipli, si parla di **Federated Discovery Services**.

Discovery e descrizioni funzionali

Condizione necessaria per poter implementare agenti che siano in grado di scoprire servizi è una **descrizione funzionale** del servizio che possa essere proficuamente elaborata in moco automatico.

Occorre quindi che essa sia:

- **Web friendly**: basata su URI e scalabile ad una rete globale;
- non ambigua;
- in grado di esprimere una qualsiasi funzionalità, presente o futura;
- in grado di esprimere vocabolari esistenti e nuovi, oltre a relazioni tra le funzionalità;.

Occorre standardizzazione, quale **OWL-S**: OWL per i Servizi (OWL = Ontology Web Language: linguaggio basato su logiche descrittive per il Web semantico).

Uso diffuso dei **metadati** come parte della descrizione della semantica del servizio Web leggibile in automatico.

In particolare, l'uso di WSDL permette di specificare la forma dei messaggi attesi, dei tipi dei diversi elementi dei messaggi e, attraverso l'uso di un linguaggio di descrizione della **coreografia**, il flusso di messaggi attesi tra i diversi agenti del servizio Web.

16.2 Esempio di servizi Web con PHP + XML o SOAP

In questo paragrafo viene commentato un esempio preso da [WT03]. Scopo: costruire un sito per la vendita di libri (Tahuayo) che usa il sito di Amazon per concludere la transazione. Vedremo due approcci alternativi:

- SOAP
- XML/HTTP

Mediante i **servizi Web** il sito Amazon viene usato come un motore: è possibile implementare una ricerca in Amazon e mostrare i risultati nel proprio sito oppure riempire il carrello con oggetti nel proprio sito. Quando è il momento di concludere l'acquisto, il visitatore viene trasferito al sito di Amazon.

Ad esempio, per implementare una ricerca su Amnazon e importare i risultati su Tahuyo:

- **XML/HTTP**:
  1. Tahuyo invia la richiesta HTTP con i parametri per la ricerca;

2. Amazon restituisce un documento XML, che viene analizzato con un parser della libreria XML di PHP;
3. i risultati vengono quindi mostrati sul sito Tahuyo.

- **SOAP**:
  1. viene costruito un client SOAP in grado di spedire richieste e ricevere le risposte dal server SOAP di Amazon;
  2. le risposte ottenute conterranno esattamente le stesse informazioni del caso precedente, che però verranno estratte mediante la libreria SOAP di PHP.

Documento XML che contiene la descrizione della prima edizione del libro di riferimento. La sua DTD e Schema sono forniti da Amazon. I **servizi Web** sono interfacce di applicazioni rese disponibili via Web: si può pensare ad una classe che espone i propri metodi pubblici in rete. Anche Google e altri. Diversi protocolli, tra cui SOAP e WSDL.

Esempio di messaggio SOAP che potremmo spedire ad Amazon per cercare nella loro base di dati il libro "Way of Weasel". In risposta, si ottiene un file XML di descrizione del libro simile a quello visto prima, ma racchiuso in un envelope SOAP.

L'uso di SOAP presuppone di solito di una libreria apposita, a prescindere dal linguaggio di programmazione usato, sia per generare le richieste che per interpretare le risposte. **WSDL = Web Services Description Language** viene usato per descrivere l'interfaccia dei servizi disponibili ad un determinato sito Web.

La soluzione è composta da parti diverse:

- carrello della spesa
- uso delle interfacce ai servizi Web di Amazon: download del kit con accettazione di una convenzione che pone delle condizioni
- **parser per l'XML**: Amazon offre un'interfaccia ai suoi servizi Web con XML su HTTP;
- **uso di SOAP con PHP**: tre librerie PHP per SOAP disponibili
  1. PHP-SOAP: estensione C per PHP
  2. PEAR SOAP: file PHP di classi PEAR per creare client e server PHP;
  3. **NuSOAP**: classi PHP per creare client e server SOAP;
- implementazione di una **cache**: tra le condizioni imposte da Amazon c'è l'obbligo di conservare in una cache i dati scaricati da Amazon attraverso i servizi Web.

Nella terminologia usata da Amazon, **browse node** indica le categorie in cui vengono divisi gli oggetti (libri) da vendere. L'applicazione ruota quindi attorno ad un ciclo centrale che prevede le seguenti possibili azioni:

Richieste ad Amazon	
browsenode	default: mostra i libri in una certa categoria
detail	dettagli di un particolare libro
image	copertina del libro
search	risultati di una ricerca dell'utente
Senza richieste ad Amazon	
addtocart	
deletefromcart	
emptycart	
showcart	

16.2.1 Parser XML

Per accedere da un programma ad un documento XML si usa un parser che, a partire da un documento, rende disponibili delle API. Le **API** = Application Programming Interfaces disponibili per un documento XML sono di due tipi:

1. basate su eventi;
2. basate su oggetti, a loro volta distinte in:
   (a) basate su oggetti generici;
   (b) basate su oggetti dell'applicazione.

Una prima possibilità è quella di usare la **DOM**, che abbiamo già visto: si tratta di una generica API orientata agli oggetti, standardizzata dal WWW. Oltre che nei browser, essa può essere utilizzata più in generale per leggere, scrivere e trasmettere documenti XML. Un parser basato su DOM restituisce un oggetto di tipo albero corrispondente all'intero documento fornito in ingresso. L'approccio basato su DOM diventa eccessivamente gravoso e quindi ingestibile al crescere delle dimensioni del documento.

Un'altra possibile modalità di elaborazione è quella **basata su eventi**: ad ogni passo, il parser rende disponibile all'applicazione il frammento di dati che corrisponde ad un singolo "evento" nel documento di ingresso.

Tali eventi riflettono la marcatura e i dati nell'ordine in cui si trovano nel documento di ingresso. Esempi di eventi sono le etichette di apertura dei vari elementi, le corrispondenti etichette di chiusura, i contenuti testuali.

Si tratta di una visione frammentaria del documento, che perde la visione di insieme data dalla DOM, e quindi non è adatta a tutte le applicazioni; d'altronde, è molto più efficiente anche su documenti di piccole dimensioni. Si noti che di solito anche una DOM viene costruita usando un elaboratore basato su eventi.

L'API basata su eventi più diffusa si chiama **SAX** = Simple API for XML. L'esempio di Amazon che stiamo studiando usa la libreria XML di PHP, che è basata sul parser expat di James Clark: si tratta di un parser SAX.

Si dice che la DOM fornisce **oggetti XML generici**, intendendo che la struttura dei suoi dati e i suoi metodi riflettono i costrutti presenti in ogni documento XML: elementi, attributi, dati testuali, etc.. Tuttavia a volte può essere più utile avere a disposizione delle strutture dati che siano specifiche dell'applicazione considerata, e in particolare degli oggetti (**oggetti dell'applicazione**) con metodi creati apposta per quella particolare applicazione.

Esistono programmi che offrono supporto nell'operazione di **data binding**, che lega i dati agli oggetti dell'applicazione, e in particolare al **data mapping** da essa richiesto, che crea la corrispondenza tra le componenti del documento XML e le strutture dati specifiche dell'applicazione.

Come visto i risultati vengono sempre salvati in una **cache**: prima di inoltrare una nuova query ad Amazon la cache viene controllata per vedere se i risultati sono già presenti in quella cache.

16.3 Tecnologie per i servizi Web

Ricapitolando, tre sono le tecnologie fondamentali per l'implementazione di servizi Web:

1. **SOAP** = Simple Object Access Protocol;
2. **WSDL** = Web Services Description Language, per descrivere i servizi disponibili e le modalità per accedervi;

3. **UDDI** = Universal Description, Discovery and Integration.

Vediamo ora **come si rende disponibile un servizio Web**

Tre azioni:

1. **pubblicare**: un servizio Web si fa conoscere;
2. **trovare**: un'applicazione trova un servizio Web;
3. **legare** (*bind*): l'atto di attaccarsi ad un servizio Web e usarne i metodi.

Tre attori:

1. **fornitore di servizi**: un'applicazione che pubblica e fornisce un servizio su una rete;
2. **richiedente**: un'applicazione che cerca un servizio per usarlo;
3. **intermediario**: per far incontrare fornitori e richiedenti attraverso un registro di servizi.

- Per pubblicare un servizio Web, il fornitore aggiunge la corrispondente descrizione al registro.
- Per trovare un servizio, il richiedente descrive il servizio e l'intermediario restituisce informazioni su tutti i servizi Web che corrispondono alla richiesta.
- Dopo che il richiedente ha selezionato un particolare servizio, il richiedente contatta il fornitore e dà inizio alla transazione.
-
-

```
public class HelloWorld {
String message = "Hello, World";

public String getMessage() {
return message;
}

public void setMessage(String s) {
message = s;
}
}
```

Apache SOAP toolkit, a `xml.apache.org` mette a disposizione una form per sottomettere l'informazione necessaria alla completa descrizione del servizio.

Occorre fornire un nome (`urn:Services:HelloWorld`) al servizio, i nomi dei metodi che si vogliono esporre (`getMessage, setMessage`), ...

Per richiedere il servizio, il client manda una richiesta come questa:

```
URL url = new URL("http://localhost:8080/soap/servlet/rpcrouter");
String urn = "urn:Services:HelloWorld";
Call call = new Call();
call.setTargetObjectURI("urn:Services:HelloWorld");
call.setMetchodName("getMessage");
...
Response response = call.invoke(url,"");
```

Anche queste linee di codice dipendono dal toolkit SOAP di Apache, che definisce gli oggetti **Call** e **Response** necessari per invocare il servizio.

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

```
<SOAP-ENV:Envelope
xmlns:SOAD-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
{\bf <hw1:getMessage xmlns:hw1="urn:Services:HelloWorld"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
</hw1:getMessage>}
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

E la risposta:

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:SOAD-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
{\bf <hw1:getMessageResponse xmlns:hw1="urn:Services:HelloWorld"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<return xsi:type="xsd:string">Hello, World!</return>
</hw1:getMessageResponse>}
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Con questo meccanismo ogni applicazione che abbia accesso al router SOAP può invocare i metodi del nostro servizio Web: questo significa che le applicazioni legacy possono essere messe a disposizione come servizio Web con nessuna o pochissime modifiche. Da notare, inoltre, come SOAP sia completamente neutrale rispetto a sistema operativo e linguaggio di programmazione.

Prossimo passo: come possiamo fare in modo che le informazioni necessarie ad accedere al servizio siano rese disponibili a tempo di esecuzione, anziché cablate nella richiesta? Per rendere possibile la scoperta dei servizi Web in esecuzione occorre uno standard per descriverli: WSDL.

Un'applicazione può analizzare l'informazione fornita da WSDL (sempre in formato XML), collegarsi ad un router SOAP e fornire tutti i dettagli richiesti quali i nomi dei metodi, gli argomenti, ecc.. Serve infine sapere **dove** va messa la descrizione del servizio: in un registro UDDI.

Tutte le strutture dati e API sono rappresentabili come XML. Un registro UDDI contiene quattro tipi di informazione, ciascuna in un elemento XML a sè:

1. `businessEntity`
2. `businessService`
3. `bindingTemplate`
4. `tmodel`

```
<businessEntity businessKey="">     {\em vuota => assign unique id}
<name>DougCo Manifacturing</name>
<description xml:lang="en">
Sample business created to exercise a UDDI registry
```

```
</description>
<contacts>
Doug Tidwell</personName>
1-919-555-5583</phone>
1-919-555-2389</phone>
<email>dtidwell@us.ibm.com</email>

1234 Main Street</addressLine>
Anytown, TX 73958</addressLine>
</address>
</contacts>
</businessEntity>
```

Quando un richiedente trova un fornitore che implementa il servizio `HelloWorld`, recive la `businessKey` del fornitore e può usarla per richiedere il servizio col seguente documento XML:

```
<find\_service generic='1.0' xmlns='urn:uddi-org:api'
businessKey='00038F2D-...'>
<name>Hello World service</name>
</find\_service>
```

In risposta, il registro restituirà un documento XML con un elemento `serviceList` che descrive i servizi in maggior dettaglio. Le informazioni cosí ottenute potranno essere utilizzate per focalizzare le richieste verso il servizio giusto. Per tale servizio occorrerà l'informazione sul `bindingTemplate`, che dà il punto di accesso al servizio. Per i dettagli riguardo alle API, invece, occorre fare riferimento al `TModel`.

17. Architettura ai micro-servizi

Le architetture basate su microservizi andranno studiate su [LF14].

Il concetto di servizio che abbiamo visto nel precedente capitolo viene portato alle estreme conseguenze nelle *architetture basate su micro-servizi* (microservice architecture), che oggi vengono seguite da un considerevole numero di sistemi. L'idea è di costruire il sistema combinando un numero di servizi autonomi che svolgono funzioni elementari. Ovviamente tali micro servizi devono essere accoppiati in modo lasco (loosely coupled) e devono comunicare l'uno con l'altro scambiandosi messaggi basati sulle loro API. Molto spesso questa communicazione avviene via HTTP.

Anche una Service-Oriented Architecture (SOA) è caratterizzata da grande modularità e da comunicazioni basate su messaggi tra i diversi moduli (si pensi a SOAP). Nel caso dei micro-servizi, però, ogni servizio è associato ad una funzione elementare, anche se non esiste un criterio universalmente riconosciuto per decidere quando la funzione è sufficientemente elementare e il servizio può di conseguenza essere definito un *micro*-servizio. In generale, dipende dal tipo di sistema implementato, ma anche dal tipo di azienda.

Non solo ogni micro-servizio può venir implementato in un diverso linguaggio, ma al fine di mantenere un accoppiamento lasco tra i microservizi, anche il data storage non è in comune, come vedremo meglio nel seguito.

Come abbiamo visto nel Capitolo 9.5, i sistemi web tradizionali (cosiddetti monolitici o monolithic in inglese), erano caratterizzati da una architettura a tre strati (three-tier architecture): user-agent, server web e information source. Coi servizi web siamo passati ad una struttura multi-strato (multi-tier architecture), in cui al posto dello strato di information source posso avere un altro server web e via così.

Il maggior problema con questo tipo di sistemi è che ogni cambiamento, anche se piccolo, richiede di ricostruire (rebuilding) e rilasciare (deployment) l'intero sistema. In più, man mano che il numero di variazioni aumenta, diventa sempre più difficile mantenere la modularità dei sistema iniziale. Anche quando devo far scalare il mio sistema, occorre ricostruirlo e rilasciarlo da capo.

17.1 Vantaggi di una microservice architecture

Anche in questo caso, organizzazioni diverse possono raggiungere vantaggi diversi. Eccone comunque alcuni che sono stati evidenziati da organizzazioni che hanno adottato un'architettura a micro-servizi.

1. Ridurre le dipendenze tra le diverse squadre di lavoro, col risultato di velocizzare la produzione di codice.
2. Permettere lo svolgimento in parallelo di iniziative diverse.
3. Offrire supporto a una pluralità di tecnologie, di linguaggi, di piattaforme.
4. Permettere una degradazione dolce del servizio.
5. Offrire aiuto all'innovazione visto che è facile fare a meno di una parte del codice nel caso in cui l'innovazione proposta fallisca.

Quindi da una parte viene potenziata la *velocità* di produzione del software, ma dall'altra anche la sua *sicurezza*, legata alla facilità di fare il testing e il deploy individualmente di ogni singolo servizio.

Si noti che poiché ogni squadra che ha lavorato al singolo servizio ne mantiene anche la manutenzione, le competenze sul funzionamento dell'applicazione vengono migliorate, e anche questo è un grosso vantaggio.

17.2 Cosa serve per costruire un'architettura ai micro-servizi

Ovviamente per costruire un'architettura distribuita che davvero riesca a rendere ogni micro-servizio completamente indipendente da ogni altro occorre intervenire a molti livelli. Noi accenneremo alle due tematiche più importanti, ovvero la progettazione delle interfacce e la persistenza dei dati, lasciando allo studente interessato ulteriori approfondimenti.

17.2.1 Le API

Dalla trattazione precedente risulta chiaro che ogni micro-servizio deve essere il più possibile autonomo e slegato dagli altri o da un contesto. A questo punto, però, risulta chiaro che per poter essere utile all'applicazione è essenziale come comunica con gli altri micro-servizi. Ogni micro-servizio è dotato di un'interfaccia o *API*: se vogliamo mantenere un accoppiamento lasco tra i diversi microservizi, sarà necessario disaccoppiare il più possibile anche le loro API. Solo in questo modo ogni servizio potrà venir rilasciato (deployed) indipendentemente da tutti gli altri, e questa è una proprietà cruciale per ottenere vantaggi di una architettura ai microservizi.

Le API dei micro-servizi sono tipicamente o orientate ai messaggi o in stile hypermedia.

API orientate ai messaggi

I messaggi scambiati permettono sia di esporre degli ingressi (ad esempio, un indirizzo IP e relativa porta) ad un componente, sia di scambiarsi delle informazioni che dipendono dal singolo task. Questo permette di poter fattorizzare i cambiamenti legati all'evoluzione dell'applicazione attraverso variazioni dei messaggi scambiati.

Ovviamente ogni azienda fa scelte diverse su quali formati e quali protocolli utilizzare per scambiare messaggi tra micro-servizi. Per esempio, Netflix distingue tra comunicazioni interne o verso consumers esterni, quali ad esempio dispositivi mobili o browsers. Nel primo caso usa diversi formati (Avro, Protobuf, Thrift) su TCP/IP; nel secondo tipicamente JSON su HTTP.

Si noti come quando parlavamo di servizi web nel Capitolo 16 l'attenzione era soprattutto posta sul fatto che il servizio ci forniva determinate informazioni, quindi determinati oggetti, via web. Ora invece vediamo l'applicazione nel suo complesso come un insieme di servizi che scambiano messaggi su una rete.

API in stile hypermedia

Questo secondo tipo di comunicazione è in realtà un arricchimento del primo. Quello che viene scambiato non sono più semplicemente informazioni, ma un mix di informazioni e possibili azioni simile a quello che possiamo fare con HTML, dove l'ipertesto contiene anche form e link che chi riceve può usare per compiere delle azioni (sottomettere delle informazioni o seguire un link verso un nuovo URL). Bisogna infatti pensare che chi progetta le API dei micro-servizi sono in genere esperti di sistemi web, che sono quindi abituati a ragionare in questo modo.

17.2.2 La persistenza dei dati

Un punto centrale nelle applicazioni è la progettazione dello schema delle basi di dati, attorno a cui poi si muovono le operazioni necessarie per portare a termine le operazioni necessarie all'applicazioni. Ma questo risulta essere un grave problema in un sistema distribuito come quello formato da micro-servizi. Infatti se due servizi devono fare accesso (in lettura e/o in scrittura) ad una stessa base di dati, se non addirittura alla stessa tabella nella base di dati, essi risultano ovviamente fortemente accoppiati, ed è impossibile farne il deploying indipendentemente.

In qualche modo la progettazione non deve più partire dai dati (data-centric), ma dalle azioni (capabilities-oriented). In generale, l'ottimo sarebbe che ogni micro-servizio potesse avere un proprio micro-database a cui accedere. Quando questo è possibile, rappresenta una soluzione. Purtroppo però questa soluzione non è sempre percorribile.

Si pensi infatti alla necessità di ottenere un report delle operazioni eseguite: la necessità di una reportistica è diffusa in quasi tutte le applicazioni, e ovviamente richiede che ogni micro-servizio acceda ad un solo report. Ci sono sostanzialmente due soluzioni per affrontare questo tipo di situazioni. Il primo viene chiamato *event sourcing* e richiede di salvare i singoli passi anziché il risultato finale, ovvero lo stato dell'oggetto che consideriamo. Ovviamente dalla lista dei singoli passi io posso ricavare il risultato finale, e quindi non perdo informazione, anzi, semmai ne conservo di ridondante. Infatti, salvare la lista dei singoli passi normalmente richiede più memoria rispetto a salvare il solo risultato, ma ha l'importante vantaggio che ogni micro-servizio può salvare i propri passi e in questo modo non ho la necessità di condividere informazione persistente (potete pensarla come la tabella di una base di dati) tra diversi micro-servizi. Ho quindi evitato di accoppiare tra loro i micro-servizi attraverso informazione persistente condivisa.

Una strategia alternativa per affrontare questo problema si basa su Command Query Responsibility Segregation (CQRS) e in particolare su una completa divisione tra azioni che cambiano un'entità rispetto a quelle che semplicemente la leggono. Noi tuttavia non approfondiremo questa metodologia.

Notiamo infine che laddove sia possibile conviene evitare event sourcing e CQRS che finiscono sempre per complicare l'applicazione e vanno quindi implementate solo quando strettamente necessario.

17.3 Esempi

- Amazon https://queue.acm.org/detail.cfm?id=1142065
- Walt Disney Company https://www.computerworld.com/article/2999969/modular-software-creates-agility-and-complexity.html
- Spotify https://www.infoq.com/news/2015/12/microservices-spotify/

18. Semantic web e Linked Open Data

Per lo studio, è obbligatorio [RMD11], mentre per approfondimenti e dettagli tecnici, e quindi facoltativo, si consiglia *Introducing Linked Data And The Semantic Web*[1].

18.1 Internet of Things: una nuova frontiera in rapida espansione

adattato da un testo a cura di Daniele Bucello, Action Institute[2]

Secondo i dati dell'Osservatorio Internet of Things del Politecnico di Milano[3], il mercato italiano è cresciuto del 30% ogni anno e nel 2015 ha attraversato un boom: oggi vale 2 miliardi di euro. A trainare questo settore sono stati i contatori del gas e le auto connesse, mentre per il 2016 si prevede che i trend più interessanti saranno Smart Home, Smart City e Industrial IoT, oltre alle possibilità di valorizzazione dei dati raccolti dai sensori.

Entro il 2020 vi saranno 34 miliardi di dispositivi connessi a internet, rispetto ai 10 miliardi del 2015. Tra essi, i dispositivi tradizionali di calcolo come smartphone, tablet e smartwatches saranno circa 10 miliardi mentre quelli in ambito Internet of Things (IoT) raggiungeranno i 24 miliardi. Ciò è quanto viene stimato da un nuovo report di BI Intelligence, nel quale si prevede anche che quasi 6 trilioni di dollari saranno spesi per soluzioni IoT nel corso dei prossimi 5 anni.

Una possibile definizione descrive l'Internet of Things come quell'insieme di tecnologie web-based che permettono agli oggetti di comunicare con le persone o con altre macchine, offrendo quindi un nuovo livello di interazione e di informazione rispetto all'ambiente in cui gli oggetti stessi si trovano. Esempi: un pneumatico che avverte l'autista quando è sul punto di rompersi, una pianta che comunica all'annaffiatoio quando è il momento di essere innaffiata, scarpe da ginnastica che trasmettono

[1] `http://www.linkeddatatools.com/semantic-web-basics`

[2] `http://www.huffingtonpost.it/action-institute/internet-of-things-una-nu_b_9885490.html?utm_hp_ref=italy`

[3] `http://www.internet4things.it/iot-library/\politecnico-di-milano-un-convegno-il-14-aprile-per-l-internet`

la velocità di corsa dell'atleta e il suo stato di affaticamento, piuttosto che flaconi di medicine che segnalano quando ci si è dimenticati di assumere un farmaco.

Il fenomeno IoT è stato anche chiamato "la prossima Rivoluzione Industriale", la quale cambierà radicalmente il modo in cui le tre entità fondamentali, ossia imprese, consumatori e Governi, interagiscono con il mondo fisico. Per oltre due anni, BI Intelligence ha attentamente monitorato la sua crescita, analizzando le interazioni tra tali entità e gli ecosistemi di oggetti, con i relativi dispositivi di connessione e di controllo, in 16 contesti, tra cui la produzione manifatturiera, la casa, il trasporto e l'agricoltura. Grazie a queste analisi è stato elaborato il report prima citato, il quale -oltre alle già menzionate proiezioni fino al 2020- prevede che saranno le imprese, tra le tre entità, le principali adottanti di soluzioni IoT e ne ricaveranno tre tipologie di benefici: la riduzione dei costi di gestione, l'aumento della produttività e la possibilità di espansione in nuovi mercati o di sviluppo di nuovi prodotti. I Governi, secondo il report, saranno invece la seconda entità maggiormente utilizzatrice di ecosistemi di oggetti IoT e si focalizzeranno anch'essi su aumento della produttività e riduzione dei costi, con il fine ultimo di migliorare la qualità della vita dei propri cittadini. I consumatori, infine, pur adottando meno soluzioni IoT di imprese e Governi, garantiranno comunque l'acquisto di un alto numero di dispositivi ed ecosistemi di oggetti investendovi una notevole quantità di denaro.

Un altro report a riguardo è stato formulato da McKinsey Global Institute[4], al fine di determinare come la tecnologia IoT possa determinare valore economico reale. Sono stati analizzati più di 150 contesti di utilizzo, quali ad esempio quello healthcare, con dispositivi che consentono di monitorare la salute e il benessere delle persone, e quello manifatturiero, con imprenditori che utilizzano sensori per ottimizzare la manutenzione di attrezzature e per tutelare la sicurezza dei lavoratori. Secondo il risultato di tale analisi si stima che la tecnologia IoT avrà un potenziale impatto economico complessivo di un valore compreso tra i 3,9 e gli 11,1 trilioni di dollari all'anno entro il 2025.

Il livello di valore aggiunto, comprendente anche il surplus del consumatore, sarebbe in altri termini equivalente a circa l'11% dell'economia mondiale. In particolare, l'impatto sarebbe predominante nelle fabbriche, ad esempio in contesti di gestione di processi operativi e di manutenzione predittiva, con una stima tra gli 1,2 ed i 3,7 trilioni di dollari nel 2025, nelle città, ad esempio in ambiti di sicurezza pubblica e di controllo del traffico, con una stima tra gli 0,9 e gli 1,7 trilioni di dollari sempre nel 2025, e nel contesto healtchare, riguardo al quale è previsto che il monitoraggio e la gestione delle malattie e il miglioramento del benessere individuale tramite soluzioni IoT presenteranno un valore tra gli 0,2 e gli 1,6 trilioni di dollari nel medesimo anno. Proiezioni un po' inferiori, ma comunque non affatto trascurabili e anzi alcune di esse vicine, secondo le stime più ottimistiche, al trilione di dollari, concerneranno l'impatto di sistemi IoT nel retail, nei cantieri, nei veicoli, nella navigazione, nella domotica e negli uffici.

Il raggiungimento di questi livelli di impatto richiederà comunque alcune condizioni per essere effettivo nel futuro prossimo, in particolare superando ostacoli tecnici, organizzativi e normativi; le imprese che utilizzano tecnologia IoT avranno un ruolo fondamentale nello sviluppo di sistemi e processi adatti a ottimizzare il suo valore. L'interoperabilità tra differenti sistemi IoT è ad esempio fondamentale, così come l'utilizzo esaustivo di tutti i dati prodotti da essi. La digitalizzazione di macchinari,

[4] http://www.mckinsey.com/business-functions/business-technology/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world

veicoli e altri elementi del mondo fisico è sicuramente necessaria alla cattura del pieno potenziale delle applicazioni IoT, la quale richiederà quindi innovazioni aziendali in tecnologie e in modelli di business, ma anche investimenti in nuove conoscenze, capacità e talenti. Grazie infine ad azioni di policy mirate per incoraggiare l'interoperabilità, garantire la sicurezza e tutelare la privacy e i diritti di proprietà, l'universo IoT potrà raggiungere il suo massimo valore, soprattutto se anche i leader mondiali opteranno finalmente per l'adozione di processi decisionali basati sui dati.

18.2 Semantic Web

Nella prima parte del corso abbiamo visto come il web possa essere visto come un gigantesco ipertesto distribuito sui diversi server HTTP. Tale ipertesto può venir rappresentato come un grafo in cui i nodi corrispondono a documenti, tipicamente a documenti HTML o XML, ma anche in altri formati, e sono collegati l'uno all'altro tramite link che vanno da un determinato punto di un documento ad un punto o dello stesso documento o di un documento diverso. In quest'ottica, i browser sono delle applicazioni che ci permettono di navigare questo ipertesto, visualizzando i documenti per l'utente e percorrendo i link per passare da un documento all'altro.

Quando dal web si passa al **semantic web**, questo ipergrafo diventa una struttura che può venir navigata automaticamente per ricercare e combinare informazioni annotate semanticamente secondo uno standard che può venir elaborato automaticamente. Trattando di XML, abbiamo discusso di come XML venga utilizzato per annotare non solo dei documenti, ma anche dei dati. In questa prospettiva, diviene quindi interessante rendere raggiungibili via web una serie di dati che possono tornare utili agli utenti, annotandoli in modo appropriato e collegandoli tra loro per rappresentare le relazioni che li legano. La domanda è: quali sono i formati e gli standard che permettono tecnicamente di realizzare tutto ciò?

Si tratta di pensare al web come ad una gigantesca base di dati che ha però una particolare struttura: non più relazionale come le classiche basi di dati, né ad albero, come nelle basi di dati basate su XML, ma **a grafo**. Infatti il web contiene una grande quantità di dati che però così come sono sono difficilmente utilizzabili dalle applicazioni, perché in formati come l'HTML. Anche se contengono dei dati, ad esempio perché sono stati costruiti a partire da una base di dati, questi dati sono inseriti in tabelle HTML.

Le azioni che si prefigge il semantic web in questo campo sono quindi diverse:

- rendere i dati sul web fruibili da applicazioni di intelligenza artificiale che possano elaborarli in maniera intelligente e fornire quindi delle informazioni più vicine a quello che serve all'utente finale;
- incoraggiare aziende, organizzazioni e singoli a pubblicare i loro dati in modo gratuito, seguendo standard aperti;
- incoraggiare le aziende ad usare i dati già disponibili sul web.

Come vedremo in questo capitolo, l'introduzione dei **linked data** permette di vedere il web come una sorta di enorme base di dati, in forma di grafo, in cui ogni nodo corrisponde ad una **risorsa**, ogni arco una **proprietà** di questa risorsa che la lega ad un nodo che rappresenta il **valore della proprietà**.

In particolare ci concentreremo sui **Linked Open Data** o LOD che hanno l'ulteriore caratteristica di essere aperti, ovvero accessibili senza restrizioni. Spesso si tratta di iniziative di trasparenza intraprese da istituzioni pubbliche o organizzazioni che si prefiggono di rendere accessibili ai cittadini tutta una serie di informazioni.

Partiremo quindi dal considerare i dati in forma di grafo, introdurremo RDF, cercheremo di capire in che senso la semantica, ovvero il significato entra in tutto questo, e poi accenneremo brevemente a strumenti quali RDFS e OWL per codificare il significato all'interno dei dati e a come interrogare questa base di dati semantici. Questi temi verranno sviluppati nel corso di Semantic Web della laurea specialistica.

18.3 Linked data, open data e linked open data

In generale, dei dati possono venir messi a disposizione in modo che possano venir liberamente utilizzati, riciclati e redistribuiti senza alcuna restrizione, se non, in alcuni casi, che i risultati di tali operazioni vengano resi anch'essi fruibili alle medesime condizioni. In questo caso si parla di dati aperti o **open**.

Il concetto di **Linked Data** è invece più tecnico e si riferisce all'interoperabilità tra dati provenienti da sorgenti diverse. Nel Capitolo 16 abbiamo sottolineato come mentre nel web tradizionale venissero collegati tra loro documenti (ipertesti), nel Web Semantico vengono messi in relazione dei dati. La differenza fondamentale tra documenti e dati è che questi ultimi hanno più struttura rispetto ad un documento, ed in particolare ad un ipertesto. In questo contesto, si parla di Linked Data per riferirsi a come dati strutturati possono venir pubblicati e messi in relazione tra loro sul web.

Ovviamente un'importante differenza è che mentre i documenti sono indirizzati all'utente umano, i dati strutturati sono tipicamente pensati per un accesso automatico all'informazione. Questo diventa tanto più importante quando le repository di tali dati diventano grandi, come succede spesso in questo periodo. Dare quindi a questo tipo di dati una forma che ne faciliti l'interoperabilità rappresenta la chiave per favorirne la lettura e l'utilizzo, arrivando non solo alla cercata trasparenza, ma anche alla possibilità di trovare collegamenti tra dati che provengono da sorgenti diverse.

Ricapitolando quindi, mentre il concetto di open si riferisce ad una politica di diffusione dei dati, e quindi di licenza d'uso, la possibilità di avere dati tra loro connessi, soprattutto in presenza di grandi quantità di dati, si poggia su considerazioni tecniche e su metodologie tipiche del mondo web. Esistono diversi modelli possibili per rappresentare ed implementare la relazione tra due entità, ma quella che è risultata vincente in questo campo, trovando un buon compromesso tra possibilità di rappresentare l'informazione e efficienza nell'elaborazione, si basa sul **Resource Description Framework** o **RDF**. Per capire meglio come funziona, partiamo dal considerare la rappresentazione dei dati in forma grafica.

18.4 Rappresentazione dei dati sotto forma di grafo

Per capire come funziona il Semantic Web è essenziale capire come organizza i dati sotto forma di grafo.

Una base di dati **relazionale** è composta da un certo numero di tabelle legate tra loro attraverso le chiavi primarie. In una base di dati **gerarchica**, quali ad esempio quelle basate su XML, abbiamo una serie di nodi, che per l'XML corrispondono agli elementi, che sono legati tra loro da relazioni (padre, figlio). Tali relazioni presuppongono una diversa importanza tra i due nodi, per cui il padre è più importante del figlio. Ad esempio, nel caso di XML il figlio è un sottoelemento rispetto al padre. In una base di dati **grafica** i nodi sono legati tra loro da relazioni orientate che però non presuppongono alcuna relazione di maggiore o minore importanza tra i due nodi coinvolti. Ogni nodo corrisponde ad una **risorsa**, che può essere legata ad altre risorse, ma senza che questo implichi che una risorsa è più importante delle altre.

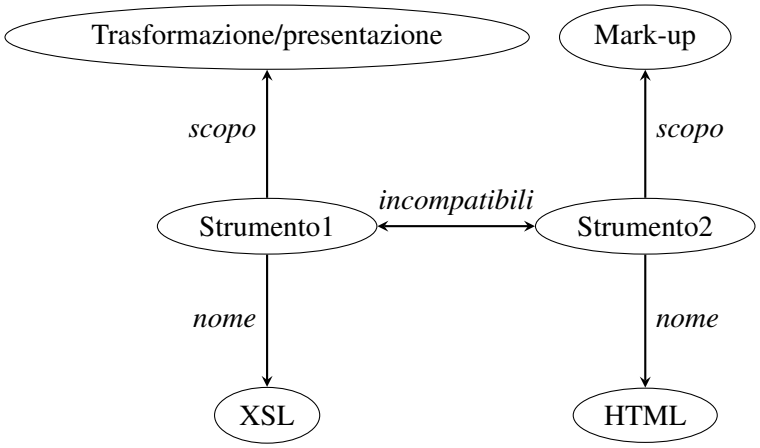


Figura 18.1: Esempio di rappresentazione a grafo.

In Figura 18.1 sono rappresentate due "cose": *Strumento1* e *Strumento2*, che hanno diverse proprietà, a volte chiamate anche *predicati*: *nome*, *scopo*, *incompatibili*.

18.5 RDF – Resource Description Framework

L'acronimo RDF sta per Resource Description Framework, ovvero framework per la descrizione di risorse, ci fa capire come un concetto fondamentale al suo interno sia proprio quello di **risorsa**. Benché RDF rappresenti uno strumento essenziale per rappresentare i dati per il web semantico, esso non ha in sé alcuno strumento per associare una semantica ai dati. Tali strumenti comprendono **dizionari** e **ontologie**, con i propri strumenti di rappresentazione.

Lo scopo di RDF è quello di rappresentare l'informazione in modo che sia facile da leggere ed elaborare via calcolatore, non per rappresentarla all'utente umano. Tra gli strumenti più diffusi e che favoriscono l'interoperabilità è ovvio pensare a XML, ed infatti RDF può essere rappresentato tramite XML, ottenendo RDF/XML. È facile capire RDF se si parte dal modello grafico che abbiamo introdotto nella sezione precedente.

In Figura 18.2 viene riportato un semplice esempio che riporta in formato RDF l'informazione contenuta nella Tabella 18.1.

Titolo	**Artista**	**Nazione**	**Casa**	**Prezzo**	**Anno**
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988

Tabella 18.1: Informazioni contenute nell'RDF di Figura 18.2.

La prima linea contiene la dichiarazione di documento XML, seguita dall'elemento radice che ha etichetta <rdf:RDF>. Vengono definiti due namespaces: rdf, che riguarda caratteristiche appunto dell'rdf, e cd, che si adatta al fatto che ci si sta occupando di CD. Il namespace rdf è ovviamente essenziale alla definizione di un documento RDF.

L'elemento <rdf:Description> contiene la descrizione della risorsa identificata dall'attributo rdf:about ed è alla base della struttura dell'RDF. Si noti che nel primo caso la risorsa è individuata univocamente dall'URI `http://www.recshop.fake/cd/`

```
<?xml version="1.0"?>
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:cd="http://www.recshop.fake/cd#">

<rdf:Description
rdf:about="http://www.recshop.fake/cd/Empire Burlesque">
  <cd:artist rdf:resource="http://www.somesite.com/Bob Dylan"/>
  <cd:country>USA</cd:country>
  <cd:company>Columbia</cd:company>
  <cd:price>10.90</cd:price>
  <cd:year>1985</cd:year>
</rdf:Description>

<rdf:Description
rdf:about="http://www.recshop.fake/cd/Hide your heart">
  <cd:artist rdf:resource="http://www.someothersite.com/Bonnie Tyler"/>
  <cd:country>UK</cd:country>
  <cd:company>CBS Records</cd:company>
  <cd:price>9.90</cd:price>
  <cd:year>1988</cd:year>
</rdf:Description>
</rdf:RDF>
```

Figura 18.2: Esempio di un documento RDF/XML (adattato da `http://www.w3schools.com/webservices/ws_rdf_example.asp`).

`EmpireBurlesque`, nel secondo da `http://www.recshop.fake/cd/Hideyourheart`. Come abbiamo visto nel Capitolo 1, l'URI identifica univocamente una risorsa: l'attributo rdf:about contiene quindi un **identificativo** della risorsa di cui stiamo trattando. Gli elementi con etichetta <rdf:Description>, comprese le etichette aperta e chiusa, vengono quindi detti **Statements RDF**.

I sottoelementi di <rdf:Description> corrispondono a **predicati** aventi per soggetto la risorsa: <cd:artista>, <cd:nazione>, <cd:casa>, ... Si noti come gli **oggetti** dei predicati possono essere dati dal contenuto dell'elemento o da un'altra risorsa, indicata dall'attributo rdf:resource. In altre parole, il valore della proprietà può essere dato da una risorsa che verrà descritta da proprietà in un altro statement RDF. Questo ovviamente corrisponde ad un grafo in cui la proprietà, corrispondente ad un arco, può puntare ad una foglia (valore) o ad un nodo interno (risorsa).

Un elemento rdf:Description ha quindi la seguente struttura:

```
<rdf:Description rdf:about="subject">
    <predicate rdf:resource="object"/>
    <predicate>literal value</predicate>
<rdf:Description>
```

e corrisponde a **due** statements RDF che si riferiscono alla stessa risorsa, il subject. Nel primo caso l'oggetto è dato da una risorsa, nel secondo da un letterale.

Esercizio

Si individuino soggetti, predicati ed oggetti nel seguente documento XML:

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

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/"
         xmlns:region="http://www.country-regions.fake/">

<rdf:Description rdf:about="http://en.wikipedia.org/wiki/Oxford">
    <dc:title>Oxford</dc:title>
    <dc:coverage>Oxfordshire</dc:coverage>
    <dc:publisher>Wikipedia</dc:publisher>
    <region:population>10000</region:population>
    <region:principaltown rdf:resource="http://www.country-regions.fake/oxford"/>
</rdf:Description>

</rdf:RDF>
```

Riferimenti bibliografici

[Ber+06] Tim Berners-Lee et al. «A Framework for Web Science». In: *Found. Trends Web Sci.* 1.1 (gen. 2006), pagine 1–130. ISSN: 1555-077X. DOI: 10.1561/1800000001. URL: http://dx.doi.org/10.1561/1800000001 (citato a pagina 161).

[DND02] Harvey M. Deitel, T.R. Nieto e Paul J. Deitel. *Internet & World Wide Web, How to program*. Prentice Hall, 2002 (citato a pagina 121).

[Eck02] Bruce Eckel. *Thinking in Java*. Prentice Hall, 2002 (citato alle pagine 13, 53).

[Eck03] Bruce Eckel. *Thinking in Enterprise Java*. Mindview Inc., 2003 (citato a pagina 63).

[Hea06] J. Heaton. *Programming Spiders, Bots, and Aggregators in Java*. Wiley, 2006. URL: http://books.google.it/books?id=VA%5C_KWgvEV9cC (citato a pagina 157).

[HC01] Jason Hunter e William Crawford. *Java Servlet Programming*. 2nd. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001. ISBN: 0596000405 (citato a pagina 107).

[KR03] Kurose e Ross. *Internet e Reti di Calcolatori*. McGraw-Hill, 2003 (citato a pagina 23).

[LF14] James Lewis e Martin Fowler. *Microservices*. https://martinfowler.com/articles/microservices.html. Accessed: 2019-26-08. 2014 (citato a pagina 173).

[MRS08] C.D. Manning, P. Raghavan e H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN: 0521865719. URL: http://nlp.stanford.edu/IR-book/ (citato a pagina 157).

[RMD11] G. Rizzo, F. Morando e J.C. De Martin. «Open Data: la piattaforma di dati aperti per il Linked Data». In: *Informatica e Diritto* 20 (ott. 2011), pagine 493–511. ISSN: 0390-0975. DOI: 10.1007/978-3-642-22233-7_20. URL: http://nexa.polito.it/nexacenterfiles/RizzoEtAl2011-IeD.pdf (citato a pagina 177).

[WT03] L. Welling e L. Thomson. *Php and Mysql Web Development*. Developer's Library. Sams Publishing, 2003. ISBN: 9780672325250. URL: http://books.google.it/books?id=h69QAAAAMAAJ (citato a pagina 167).

Indice analitico

X