Ministero dell'Istruzione, dell'Università e della Ricerca Servizio Automazione Informatica e Innovazione Tecnologica

Modulo 15

Interattività sul Web

ForTIC

Piano Nazionale di Formazione degli Insegnanti sulle Tecnologie dell'Informazione e della Comunicazione

Percorso Formativo C

Materiali didattici a supporto delle attività formative

2002-2004

Promosso da:
- Ministero dell'Istruzione, dell'Università e della Ricerca, Servizio Automazione Informatica e Innovazione Tecnologica
- Ministero dell'Istruzione, dell'Università e della Ricerca, Ufficio Scolastico Regionale della Basilicata

Materiale a cura di:
- Università degli Studi di Bologna, Dipartimento di Scienze dell'Informazione
- Università degli Studi di Bologna, Dipartimento di Elettronica Informatica e Sistemistica

Editing:
- CRIAD - Centro di Ricerche e studi per l'Informatica Applicata alla Didattica

Progetto grafico:
- Campagna Pubblicitaria - Comunicazione creativa

In questa sezione verrà data una breve descrizione del modulo.

Gli scopi del modulo consistono nel mettere in grado di:

- Configurare e gestire elementi di interattività in una pagina *Web*: *password*, *cookies*, *chat room*, gruppi di discussione.
- Conoscere le principali tecniche di programmazione sul lato *server*.

Il modulo è strutturato nei seguenti argomenti:

- **Sicurezza**
  - Inserire protezioni basate su *password* in una pagina *Web*.
  - Inserire *Internet cookies* in una pagina *Web*.
- ***Chat rooms* e gruppi di discussione**
  - Configurare ed ospitare una *chat room*.
  - Inserire una *chat room* in una pagina *Web*.
  - Configurare un gruppo di discussione asincrono.
  - Inserire un gruppo di discussione asincrono in una pagina *Web*.
- ***Scripting***
  - Cenni su *CGI*, *Servlet*, ASP e altre principali tecniche di programmazione sul lato *server*.

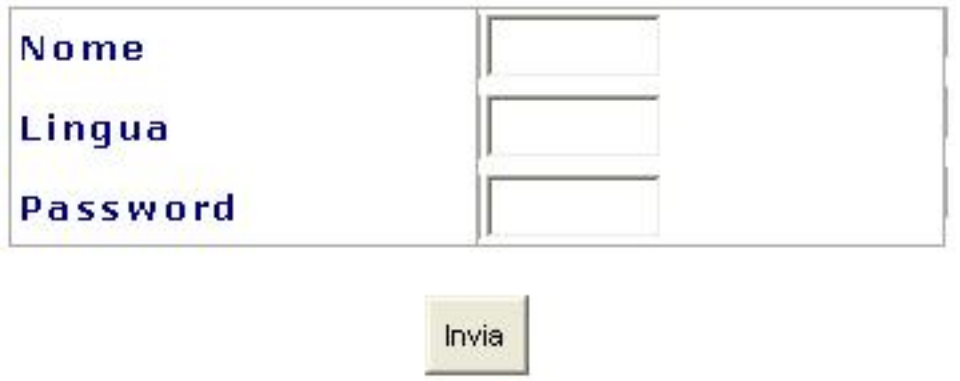
Introduzione

Sicurezza

Cosimo Laneve

Protezioni in una pagina Web

HTML è un linguaggio che consente di presentare informazioni in pagine di formato predefinito e accattivante. Quando non si hanno a disposizione informazioni sulle preferenze dell'utente, tali pagine sono uguali per ogni utente. Alternativamente, è possibile far arrivare informazioni dal *client* al *server* secondo schemi prestabiliti, facendo uso della struttura modulo. I moduli, o *form*, consentono di inserire dei controlli che sono già in uso in altri programmi di uso comune (per esempio per l'*Office Automation*): Pulsanti, caselle di testo, caselle combinate, eccetera I dati che possono essere inseriti dall'utente e poi inviati, potranno essere oggetto di elaborazione presso il *server* con programmi presenti allo scopo (e aventi tecnologia differente, per esempio **CGI**, *Common Gateway Interface*, o **ASP**, *Active Server Pages*, eccetera).



una pagina HTML con una form

Il marcatore fondamentale è senz'altro <FORM> che permette di inserire un modulo. Il tag richiederà l'inserimento del corrispondente marcatore di chiusura. Tra i due elementi possono essere inseriti tutti i controlli cui si è già accennato. Nel caso dell'esempio precedente, il codice è riportato a seguire:

```
<FORM NAME="Prima" METHOD="GET" ACTION="http://www.indir.com/pro">
  <LABEL> Nome </LABEL>
    <INPUT NAME="Nome" TYPE="TEXT">
  <LABEL>Lingua</LABEL>
    <INPUT NAME="lingua" TYPE="TEXT">
  <LABEL>Password</LABEL>
    <INPUT NAME="pw" TYPE="PASSWORD">
    <INPUT ID="visitatorep" TYPE=SUBMIT SIZE=3 VALUE="Invia">
</FORM>
```

Un modulo può contenere i vari controlli disponibili, ma non può essere presente un *form* annidato (è possibile tuttavia averne diversi all'interno di una pagina).

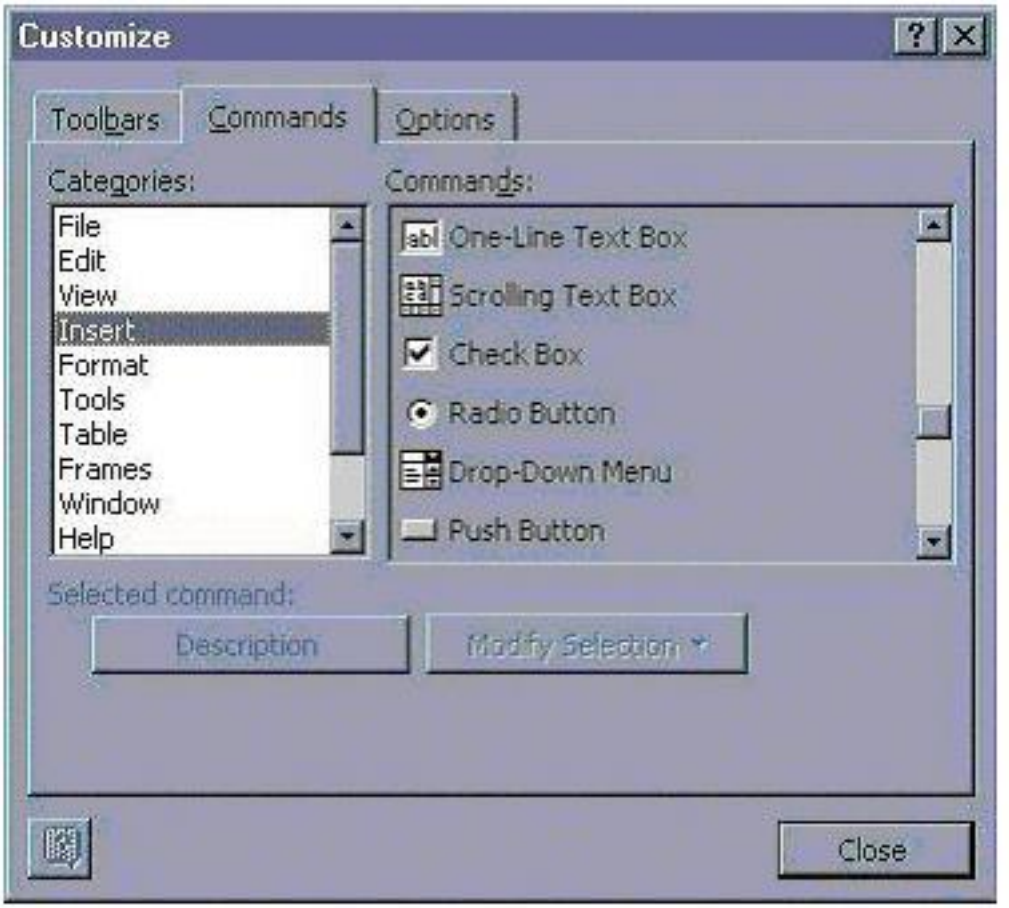
Quando viene lanciato un *form*, tutti i suoi campi vengono inviati al *server*. Il tag <form> indica al *browser* l'inizio e la fine del *form*. Questo significa che un *form* può agevolmente includere una tabella o un'immagine insieme ai *form field* illustrati in seguito. Per esempio:

```
<html>
  <head>
    <title>Esempio di un form</title>
  </head>
  <body>
    <form>
      <!-- Qui vanno form field e HTML -->
    </form>
  </body>
</html>
```

Diversamente da una tabella, i *form* non sono visibili sulla pagina. Il *form* nel nostro esempio è inutile. Innanzitutto non contiene *form field*. In secondo luogo, non contiene un ricevente per il *form*. Per fare sì che il *browser* sappia dove inviare il contenuto, dobbiamo aggiungere questi attributi al tag <form>:

I controlli

Una *form* necessita di controlli che consentano di inviare i dati ad un *server*. *form* e controlli possono essere inseriti agevolmente con un *editor Web* (per esempio *Front Page*), magari rendendo disponibili tali controlli sulle barre.



controlli nell'editore Web FrontPage

Da un punto di vista pratico ognuno dei controlli ha un nome, e contiene un valore. E' importante ricordarsene quando tali valori vengono raccolti dal *server* per l'elaborazione. Una stringa tipica generata da una *form* è la seguente:

www.sito.it/pagina.html?nomecontrollo1=valore1&nomecontrollo2=valore2
dove nomecontrollo è il nome di ognuno dei controlli presenti nella *form*, e valore è il dato presente nel controllo quando viene premuto il tasto di conferma.

Vediamo in dettaglio quali sono i controlli che è possibile inserire in una *form*.

Check box: Le caselle di scelta (check boxes) si adoperano quando si vuole dare al visitatore la possibilità di selezionare una o più opzioni da una serie di alternative. I *check* sono controlli indipendenti (ognuno assume un valore indipendente dagli altri). Se si vuole permettere una sola opzione, bisogna allora usare i bottoni radio. Il controllo ed il codice **HTML** per ciascun *check* è il seguente:

<input TYPE="checkbox" NAME="Checkbox1" VALUE="ON">

l'attributo value assume i due valori on e off.

Radio button: I *radio buttons* (bottoni radio) vengono usati quando si vuole che il visitatore selezioni una - e soltanto una - opzione da una serie di alternative. Se si

vogliono permettere più opzioni contemporaneamente, bisogna invece usare i *check boxes*. Il controllo ed il codice **HTML** per ciascun *check* è il seguente:

<input TYPE="radio" NAME="radio" VALUE="radiobutton3" CHECKED="true">

l'attributo *checked* (modificato di solito a *runtime*) è presente soltanto sul controllo selezionato.

Submit button:Quando un visitatore clicca su un tasto d'invio, il *form* viene mandato all'indirizzo specificato nell'attributo action del tag <form>. Il controllo ed il codice **HTML** è il seguente:

<input TYPE="submit" ACTION="..." METHOD="post" NAME="submit">

Il testo invia è un testo di *default* stabilito dal *browser*. E' possibile modificarlo mediante l'attributo *value*.

Reset button: quando un visitatore clicca su un bottone di *reset*, i controlli sono resettati ai valori di *default*.

<input TYPE="reset" VALUE="Reset" NAME="resetbutton">

Image button: i bottoni di immagine hanno lo stesso effetto dei bottoni di invio, tranne che per la possibilità di essere personalizzati graficamente. Quando un visitatore clicca su un bottone di immagine il *form* viene mandato all'indirizzo specificato nell'attributo *action* del tag <form> .

<input TYPE="image" SRC="immagine.gif" NAME="image">

Text field: I *text fields* sono aree di una riga sola che permettono all'utente di inserire testo. L'opzione *size* definisce la larghezza del *field* e quindi la quantità dei caratteri visibili che il *field* riesce a contenere. *Maxlength* invece la lunghezza massima del *field* e la quantità di caratteri che possono entrare nel *field*.

<input TYPE="TEXT" SIZE="10" NAME="shorttext">

La variante TYPE="password" sostituisce i caratteri digitati con asterischi, permettendo così l'inserimento di *password*. Tale controllo però non cifra il testo, che quindi viene inviato in chiaro al *server*.

Hidden field: I *field* nascosti (hidden fields) sono simili a *field* di testo, con una differenza importantissima: il *field* nascosto non è mostrato sulla pagina. Di conseguenza il visitatore non può scriverci nulla sopra; lo scopo di questo tipo di *field* è dunque di introdurre informazioni non accessibili al visitatore. Il codice è il seguente:

<input TYPE="HIDDEN" NAME="nascosto">
Select: I *drop-down* menu sono probabilmente gli oggetti più flessibili da aggiungere ai *form*. Il menù a discesa ha lo stesso scopo dei bottoni radio (una selezione soltanto) o dei check boxes (dove sono permesse selezioni multiple).

voce1 voce2
<select size="1" name="D1">
<option value="valore1">voce1</option>
<option selected="true" value="valore2">voce2</option>

</select>

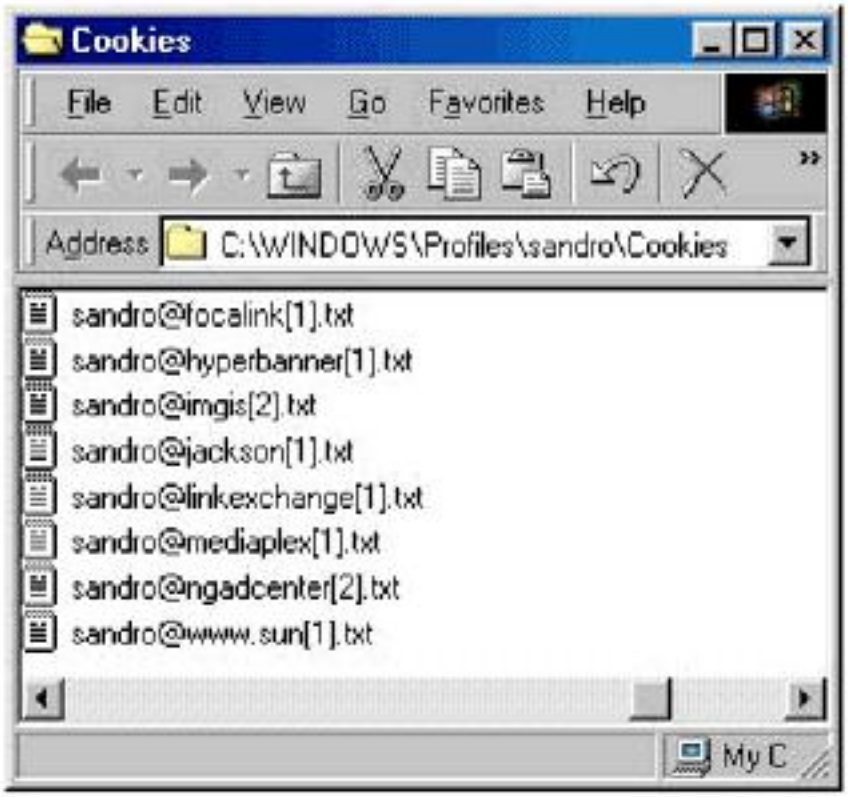
I vari *option* definiscono le voci, e *SELECTED* indica la voce selezionata.

Cookies

Un *cookie* (biscottino) è un piccolo frammento di informazione che un *server Web* può immagazzinare temporaneamente in un *browser Web*. Ciò risulta utile per permettere al *browser* di ricordare alcune informazioni specifiche che il *server Web* può recuperare successivamente.

È questa la definizione di *cookie* data da *Netscape*. Si tratta di un meccanismo di connessione server side attraverso il quale il *server Web* può immagazzinare informazioni sulla connessione con il *client*; in altre parole, il *server* su cui risiede la pagina ***Internet*** visualizzata ha la possibilità di memorizzare alcune informazioni sia sul *server* che sul *client*, per utilizzarle in modi diversi; per esempio è possibile personalizzare le pagine in modo diverso in funzione delle preferenze, oppure fornire in video il numero di accessi di un determinato utente alla pagina corrente. In pratica attraverso l'uso dei *cookie* si può memorizzare lo stato della connessione (altrimenti non previsto dal protocollo HTTP), in modo da consentire al *server* di agire in funzione di esso. Tuttavia, quello dei *cookie* può essere uno strumento utilizzato per carpire indebitamente informazioni all'utente, trasgredendo alle norme sulla *privacy*.

La memorizzazione dei *cookie* avviene in modi e in cartelle diverse in funzione del *browser*: *Netscape* crea un unico *file* denominato cookies.txt memorizzato nella cartella relativa al nome dell'utente e posta all'interno di Programmi\Netscape\Users. *Explorer* invece registra separatamente ogni *cookie*. La cartella di memorizzazione si chiama proprio ***Cookie*** e si trova all'interno della *directory Windows*. Quando vengono impostati diversi utenti, verranno create automaticamente cartelle diverse atte a contenere i *cookie*.



memorizzazione dei cookies in Windows

Di seguito viene visualizzato il contenuto del *cookie* **sandro@www.sun[1].txt**
sun_visitor_uid

313333531383237325e30
www.sun.com/0
3578172800
29305075
2662236192
29296023

Effetti dei cookie

Attraverso i *cookie* vengono memorizzate informazioni sul *browser* in modo che a una successiva connessione allo medesimo sito il *server* possa leggere lo stato della stessa. Viene scritto un numero identificativo che consente di riconoscere l'utente e, insieme a questo, altre informazioni di servizio o utili al *server*: per esempio, si utilizza spesso una data di scadenza del *cookie*, oppure si può memorizzare la data della connessione o il numero di collegamenti effettuati. I *cookie* possono essere quindi utilizzati per diversi scopi; di seguito viene fornito un elenco di possibili appicazioni:

- salvataggio di informazioni generiche sulla connessione (identificativo utente, data di connessione, data di scadenza, numero di connessioni, eccetera);
- salvataggio dello stato della navigazione (esempio pagine visitate) per

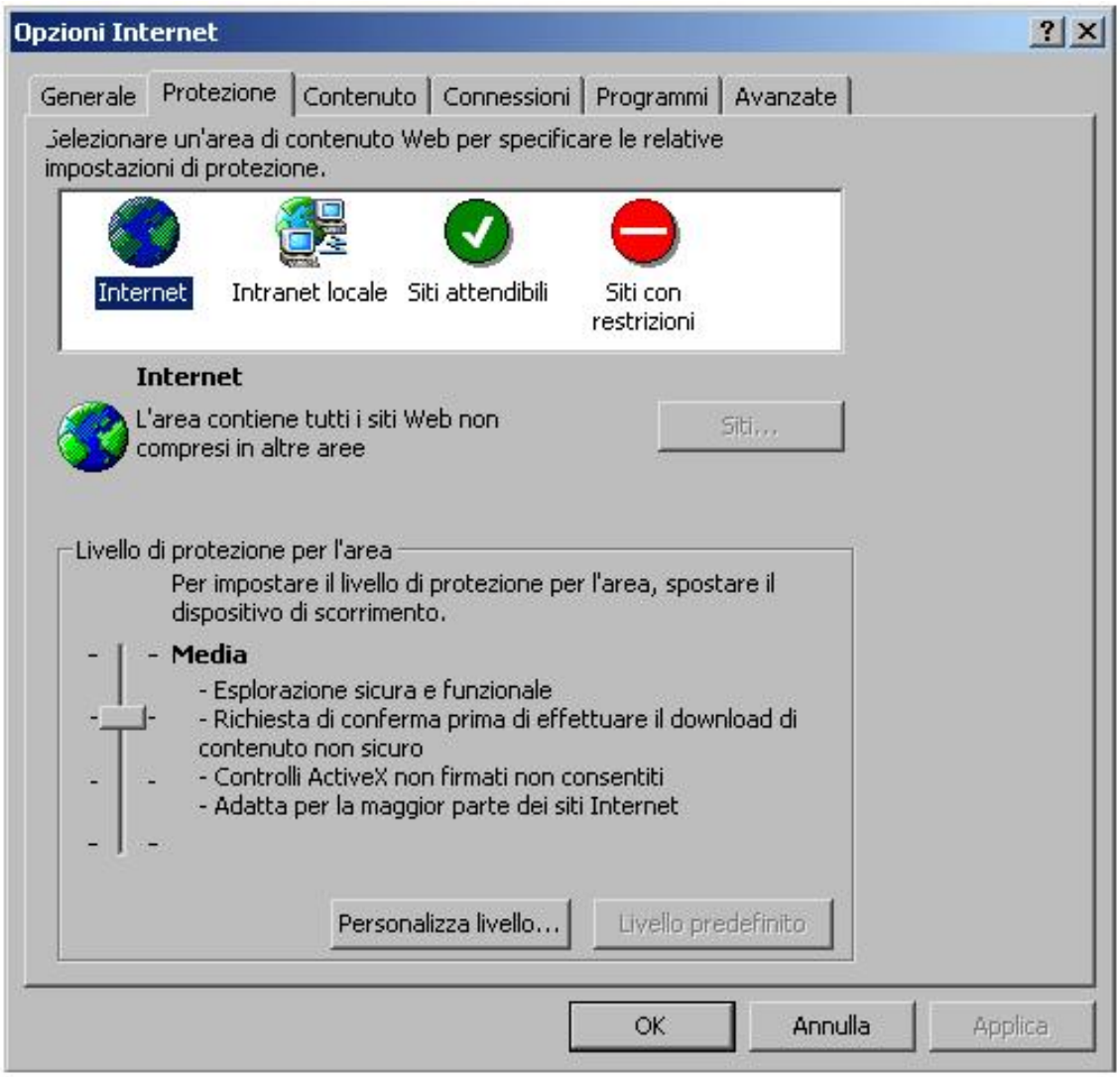
permettere una migliore fruizione del sito;
- memorizzazione di informazioni utili per il commercio elettronico (per esempio quando si vuole riempire un carrello elettronico della spesa le informazioni parziali possono essere memorizzate nei *cookie*);
- salvataggio di informazioni utili a fini statistici;
- salvataggio di informazioni utili per applicazioni ludiche.

Tuttavia la maggior parte dei *cookie* permangono nella macchina *client* solo durante la connessione, al termine della quale vengono distrutti. I *cookie*, in ogni caso, non interagiscono con altri *file* presenti sul *client* o con il sistema operativo. Sopravvalutando le capacità di questi oggetti, li si ritengono erroneamente capaci di carpire *password*, numero di carte di credito o altre informazioni sul *software* installato sul nostro *computer*. Attraverso i *cookie* vengono memorizzati in un *database* i siti visitati durante la navigazione, costruendo un profilo dell'utente riguardante i sui interessi; con un sistema legato ai *cookie* viene selezionata la pubblicità da mostrare durante la navigazione, pubblicità che risponderà selezionata in base ai gusti individuati dell'utente (per esempio il sito *DoubleClick* spesso si occupa della selezione dei *banner* pubblicitari da mostrare).

E' possibile definire le impostazioni del *browser* in modo da impedire l'uso dei *cookie*; tuttavia alcuni siti non funzionano regolarmente qualora vengano attivate tali misure di protezione. Il sistema di comunicazione dei *cookie* tra *client* e *server* prevede che quest'ultimo possa richiedere al *browser* le informazioni eventualmente presenti, creandole se non ci sono o modificandole se è il caso. Il *server* comunque viene a conoscenza delle generalità dell'utente (ad esempio nome o indirizzo di posta elettronica) solo attraverso una comunicazione diretta (e volontaria) degli stessi, non avendo la possibilità di prelevarli autonomamente dal disco rigido.

Impostazioni di protezione

I *browser* sono tipicamente configurati per consentire la creazione di *cookie*; l'utente tuttavia può stabilire che venga visualizzato un messaggio prima che il sito collochi il *cookie* sul disco rigido, in modo che l'utente possa decidere di acconsentire o meno all'operazione. In alternativa è possibile configurare i *browser* in modo da impedire l'accettazione di qualsiasi *cookie*. Di seguito si fa riferimento alle procedure atte all'impostazione delle protezioni su *Internet Explorer* 5.



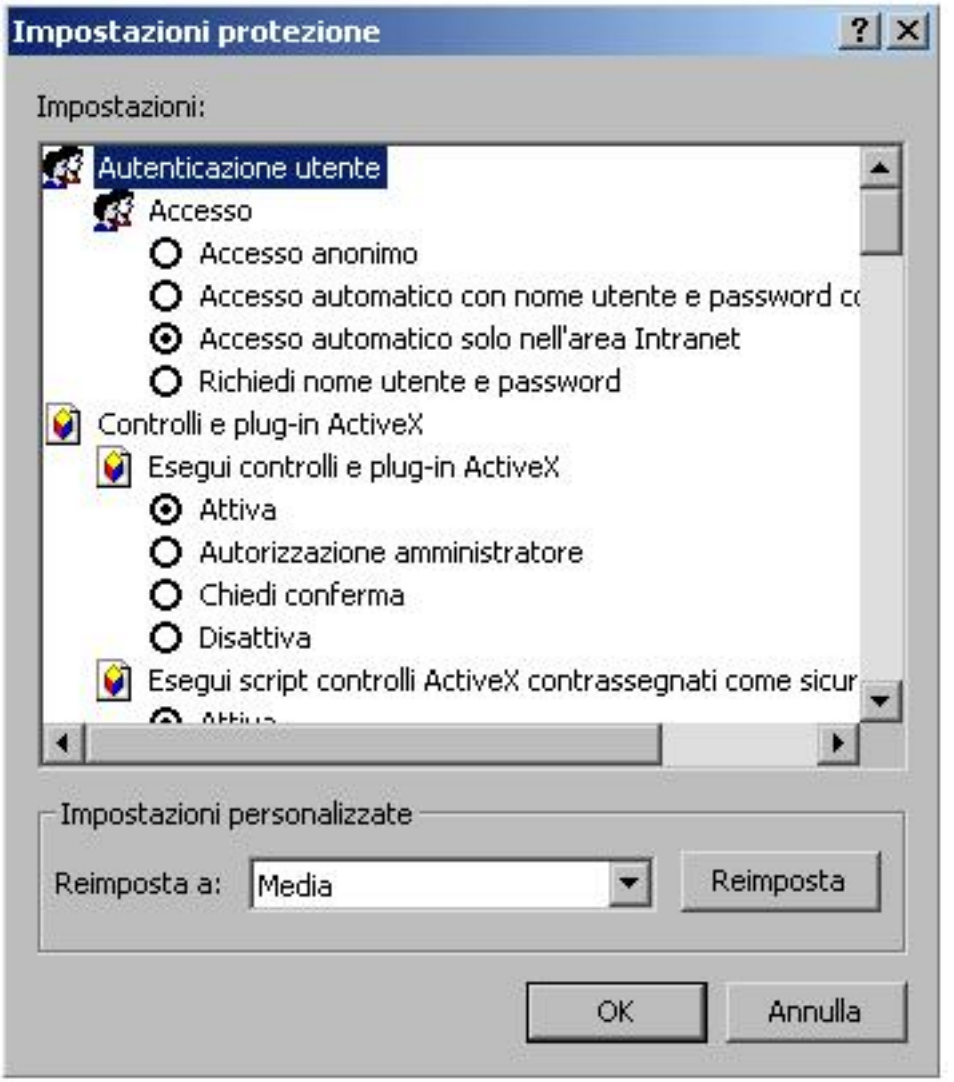
impostazioni di protezione in Explorer

Selezionando la voce Strumenti --> Opzioni Internet --> Personalizza livello è possibile definire diverse modalità per la gestione dei *cookie*. A proposito della gestione dei *cookie*, è possibile prevedere un diverso modo di procedere tra i *cookie* temporanei e quelli memorizzati nel disco rigido; in ogni ipotesi è possibile scegliere tra tre diverse soluzioni:

- si desidera accettare un *cookie* da un sito ***Internet*** senza ricevere alcun messaggio di avviso (Attiva).
- Prima di accettare un *cookie* da un sito ***Internet*** si desidera ricevere un messaggio di avviso, (Conferma).
- Ai siti *Web* non è consentito inviare *cookie* al *computer* in uso, né leggere quelli salvati sul disco rigido(Disattiva). Tuttavia alcuni siti sono visualizzabili in modo ottimale solo se è possibile gestire i *cookie*.

È importante segnalare che ogni volta che che viene inviata una richiesta al *server*, essa include l'indirizzo *IP* del mittente, il *browser* e il sistema operativo utilizzato, a prescindere dall'utilizzo o meno di un *cookie*, che non è quindi da ritenersi responsabile

della diffusione di tali dati di riferimento.



impostazioni di protezione in Explorer

Chat Rooms e Gruppi di discussione

Cosimo Laneve

Introduzione

Internet offre due tipologie di comunicazione:

- **Comunicazione sincrona**, in cui lo scambio di informazioni avviene in tempo reale, senza sensibili pause tra invio e ricezione del messaggio
- **Comunicazione asincrona**, o in differita in cui c'è uno scarto temporale sensibile tra il momento in cui il messaggio viene inviato, quello in cui viene ricevuto, e quello in cui una eventuale risposta viene spedita.

Esempi di comunicazioni sincrone e asincrone si trovano nell'esperienza di tutti, senza scomodare ***Internet***. La comunicazione telefonica è un tipo di comunicazione sincrona; la corrispondenza postale è un tipo di comunicazione asincrona.

In questo modulo analizziamo due noti esempi di comunicazione di ***Internet***, uno per tipologia, le *chat* testuali - sincrona - e i gruppi di discussione - asincrona.

Configurare e ospitare una Chat Room

IRC (*Internet Relay Chat*)

IRC (*Internet Relay Chat*) è il protocollo più diffuso per la conversazione tra due o più persone attraverso la scrittura alla tastiera di un *computer*, invece che mediante mezzi tradizionali come telefono o viva voce. Il funzionamento di **IRC** si basa su un nodo *server* con la funzione di ripetitore dei messaggi scambiati dagli utenti. Il ***server* IRC** è un programma installato sul *computer* che ospita (**host**) il servizio di *chat*: ad esso si accede utilizzando uno dei *software* ***client* IRC** (*freeware*, *shareware* o commerciale) reperibili anche sulla rete. Ci sono diversi tipi di *software client* per *chat line* e l'utente può utilizzare quello che meglio lo soddisfa: dal più essenziale, con **interfaccia a carattere**, al più ricco di funzioni e strumenti e con **interfaccia grafica**: la lavagna, il *browser*, la posta, il trasferimento di *file*, la voce. Le interazioni in tempo reale fra un **server IRC** e i **client IRC** rendono possibile il flusso comunicativo fra persone che si collegano dai più disparati luoghi del pianeta. Nella funzionalità *standard* di base il dialogo è **testuale** e via via digitato da tastiera: tutti i partecipanti vedono le frasi degli interlocutori remoti scorrere sul monitor, via via che vengono inviate.

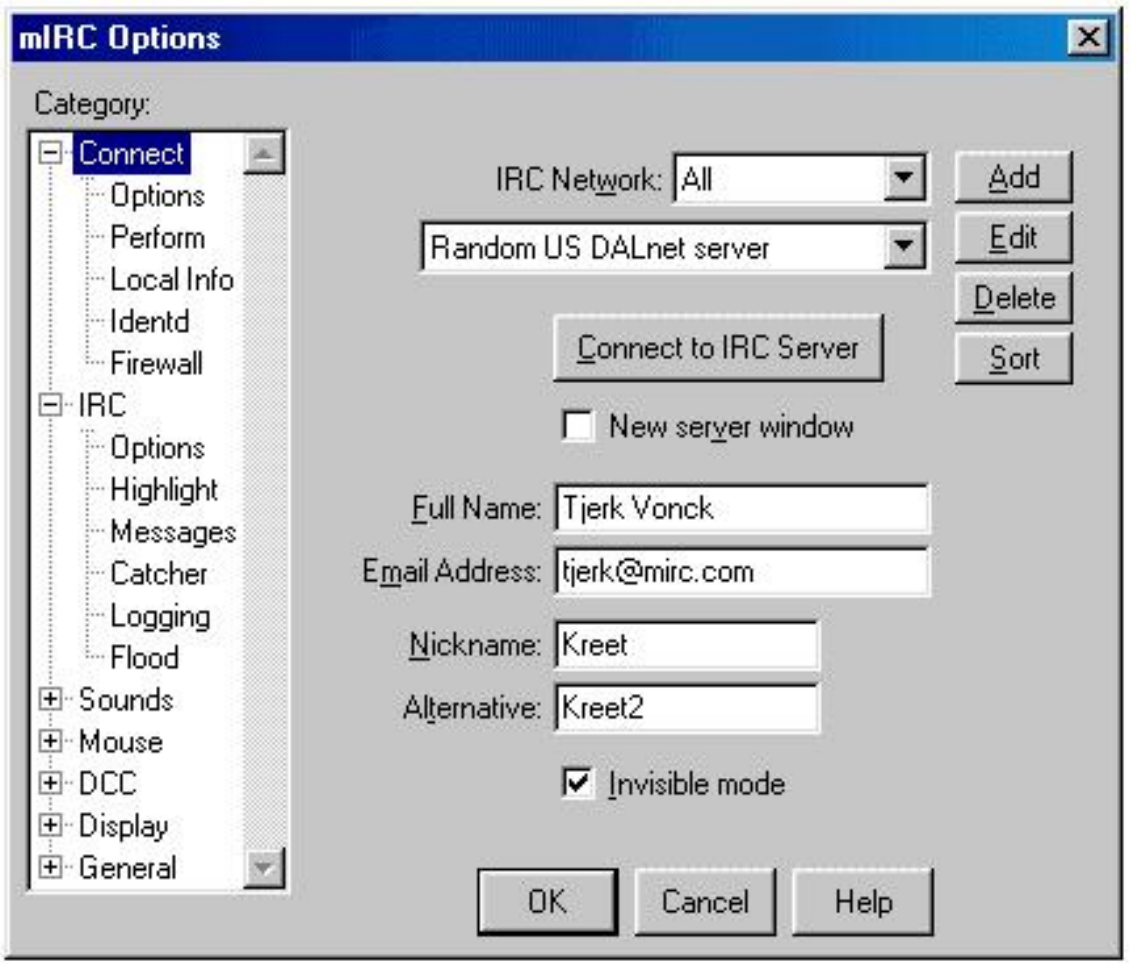
I *server* **IRC** consentono l'accesso a spazi virtuali di conversazione detti canali, stanze, gruppi di conversazione. Questi spazi possono essere di pubblico accesso o richiedere una parola chiave; possono essere **aree tematiche** di dialogo, scambio, confronto, con o senza moderatore. **IRC** consente, durante la sessione di *chat* su canale condiviso da più utenti, di conversare in modo appartato (*chat* privata, *DCC - Direct **Client** to **Client***) con uno dei presenti.

Lo si sceglie nella **lista** dei **nickname** (nome convenzionale di un utente) che appare sullo schermo durante la sessione di *chat* e registra le presenze in **tempo reale**, via via che gli utenti entrano ed escono dal canale o stanza. Si può semplicemente inviare un invito e attendere la risposta; durante la *chat* privata si possono trasferire *file* di qualunque tipo di formato (testo, immagine, suono..).

Collegamanto e creazione di chat room

Per collegarsi a una *chat* room è necessario quanto segue:

Discutiamo l'installazione di *mIRC*, uno dei *software* più diffusi. Una volta scaricato il *file*, lo si lancia e si segue la procedura di installazione automatica. Dopo la finestra con la richiesta di pagamento, compare un'altra finestra che consente di scegliere un *server* **IRC** a cui collegarsi e di immetere i propri dati personali e del *nickname*, l'identificativo con cui ogni utente è riconosciuto dagli altri. Il programma non consente omonimie di *nickname*: se il nome immesso è già usato, mIRC chiede di immettere un nuovo nome. Una volta fatte queste operazioni, è necessario scegliere la voce di menù File, Connect per stabilire la connessione.



Opzioni in mIRC

A tal punto, mIRC visualizza una finestra mIRC Channel Folder con tutti i canali su cui è possibile ciattare. È sufficiente digitare nell'apposito spazio il nome del canale e premere il bottone Join per entrare in *chat*.

Se invece si vuole avviare un nuovo canale, si digita il nuovo canale nella finestra mIRC *Channel Folder* e si preme il bottone Add.

Inserire una Chat Room in una pagina Web

Le modalità per inserire una *chat room* all'interno di pagine *Web* si riducono di solito a scrivere degli *applet* i cui prototipi si trovano sulla rete, ed a inserirli nella propria pagina. È possibile quindi creare una *chat room* e gestirla nella propria pagina, consentendo a chi si collega di poter colloquiare con altri utenti che sono collegati nello stesso momento.

Un esempio di codice è mostrato nella figura sottostante.

```
<applet codebase="http://ircqnet.icq.com/users/IrCQNet-Current/" code="IRCQNet.class" archive="IRCQNet.jar" name="IRCQNet" width=550 height=300><param name="firstJoin" value="#Web"></applet>
<applet CODEBASE="http://wwp.icq.com/Panel/" code="Panel.class" name=Panel width=550 height=60>
<param name=uin value="23456">
<param name=showwelcome value="no">
<param name=ticker1 value="pippolo">
<param name=ticker2 value="">
<param name=ticker3 value="">
<param name=ticker4 value="">
<param name=ticker5 value="">
<param name=ticker6 value="">
<param name=ticker7 value="">
<param name=ticker8 value="">
<param name=ticker9 value="">
<param name=ticker10 value="">
</applet>
```

Il codice riportato viene utilizzato per inserire un collegamento a *chat room* in una pagina. Il codice si trova a www.icq.com.

Configurare un gruppo di discussione asincrono

I gruppi di discussione, o *newsgroup*, sono bacheche elettroniche dedicate ad argomenti specifici. Per consultarli, bisogna collegarsi via ***Internet*** ad un *news server*, e scegliere il *newsgroup* di interesse. Ci sono molti programmi che consentono di far ciò, come **Free Agent**, che è specifico per questo servizio, o i *browser* ***Netscape*** e **Internet Explorer**. Di solito questi programmi sono integrati un un servizio di posta elettronica, visto che i due sistemi sono simili.

Ci sono molti *news server* nel mondo, che contengono dunque più o meno le stesse notizie. La corrisondenza non è esattamente uguale, per diverse ragioni. La prima, l'aggiornamento non avviene mai in tempo reale, ma in momenti della giornata in cui la rete è meno trafficata. Quindi una notizia può trovarsi su un *news server* e non in un altro. La seconda è dovuta al fatto che alcuni *news server* possono impedire l'accesso a certi gruppi di interessi. Questo è il caso di *news server* italiani per quanto riguarda gruppi di discussione in lingua russa, ad esempio. Oppure perchè contengono discussioni censurate.

Alcuni gruppi di discussione sulla rete sono:

- **alt.bonsai** sui bonsai;
- **alt.binaries** su foto/video;
- **comp.sys** sui sistemi operativi e architetture;
- **rec.arts** sulle arti;
- **sci.med** sulle scienze mediche;

Le gerarchie

Il nome di un *newsgroup* è composto da diverse parti separati da un punto. Ad esempio **first.second.third.** La prima parte è quella più generale ed indica la categoria a cui appartiene il *newsgroup*, la seconda la sottocategoria e così via per le altre.

Esempi di categorie generali sono it che identifica *newsgroup* italiani, comp che riguarda l'informatica e i *computer*, sci che riguarda le scienze.

Programmi per accedere ai newsgroup

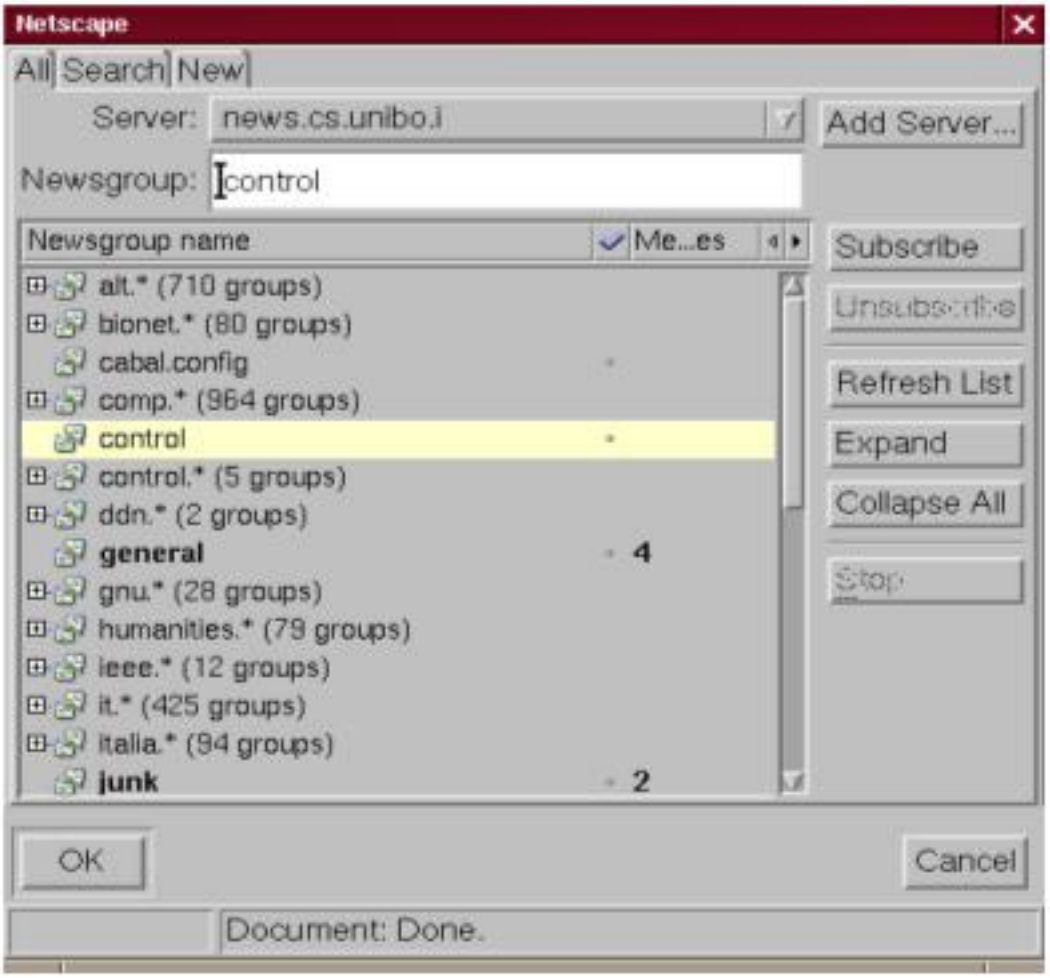
Ci sono diversi programmi per accedere alle *newsgroup*. Questi programmi sono chiamati **newsreader**, ed è possibile reperire informazioni e documenti da un apposito gruppo di discussione: **news.software.readers.**

Ci sono diversi programmi per leggere le *news*. Tra i più vecchi, ma anche più noti, c'è **Free Agent**, un programma creato dalla Fortè che ha rivoluzionato la fruizione delle *news*. A questo programma si sono ispirati i due *newsreader* di cui si discute di seguito: quello di *Netscape* e di *Outlook Express*.

Netscape

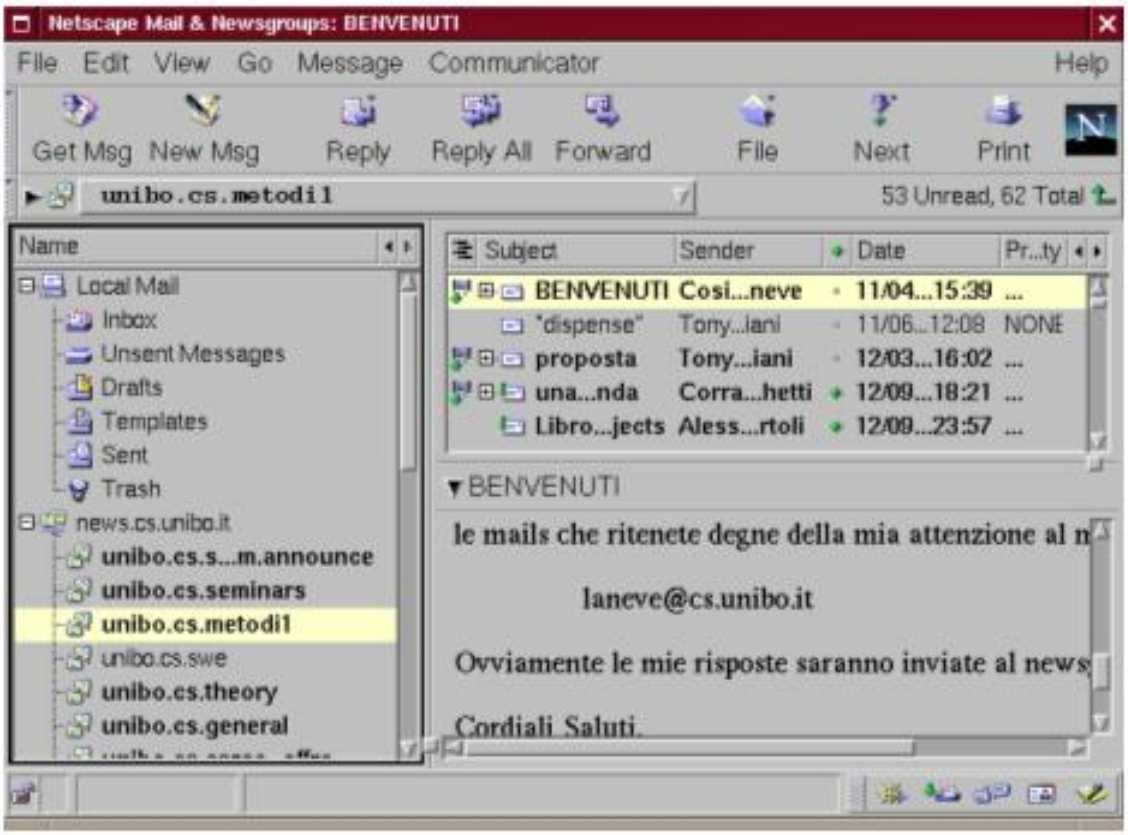
In *Netscape*, i *newsgroup* sono gestiti dallo stesso modulo *software* della posta elettronica. Per accedervi, occorre scegliere in Communicator l'opzione Newsgroups. La consultazione dei *newsgroups* richiede alcuni semplici passi di configurazione. A tal fine occore selezionare Edit e poi Preferenze; quindi la scheda Mail & Newsgroups ed infine Nesgroups Servers. Sarà sufficiente aggiungere la voce news e il *browser* si collegherà automaticamente al *server* di *news*. Se ciò non avviene, occorre richiedere l'indirizzo del *server* di *news* al vostro ***Internet*** *Provider*, ed aggiungerlo al posto di news.

A questo punto si può selezionare i *newsgroup* di interesse a cui sottoscriversi. Per far ciò, occorre selezionare File e poi Subscribe e compare una finestra come la figura sottostante.



elenco dei Newsgroups disponibili su un determinato server

Qui si possono scegliere i gruppi di proprio interesse. Una volta scelti I *newsgroup*, si può iniziare l'esplorazione delle *news*. Aprire quindi Communicator e poi Newsgropus: i *newsgroup* che sono stati selezionati appariranno nella finestra di *Netscape Messanger* assieme alle cartelle di posta elettronica. Per leggere i singoli messaggi è sufficiente un doppio click sul *newsgroup* che interessa. Si arriva quindi a una schermata simile a quella della posta elettronica, in cui però i messaggi sono indirizzati al gruppo piuttosto che al singolo utente.

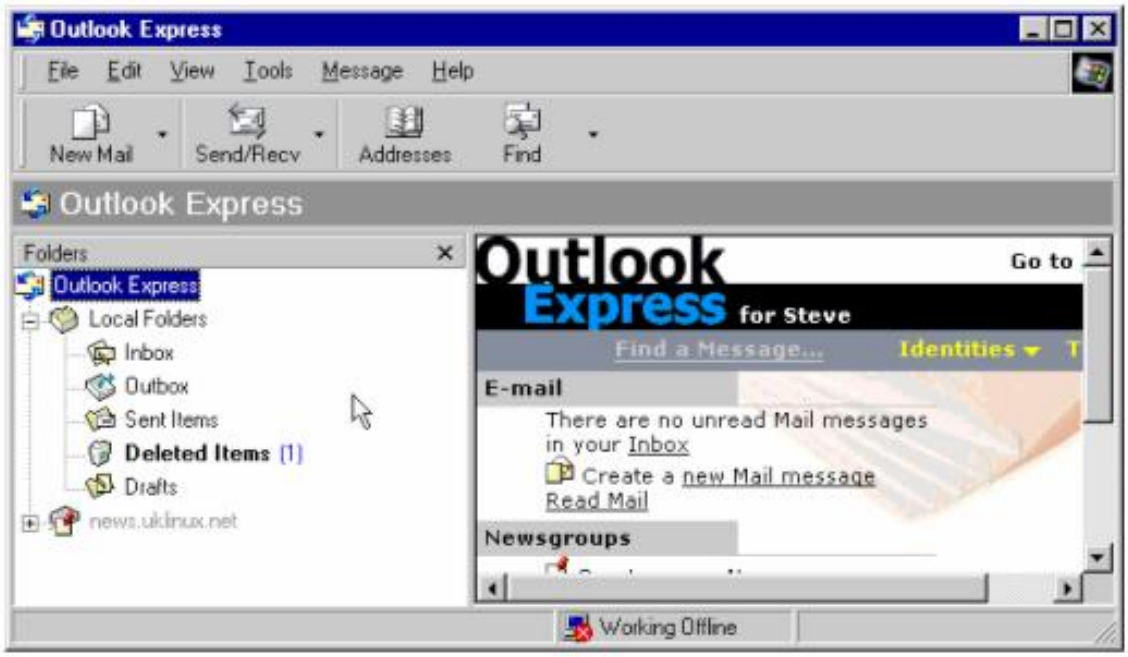


consultazione di Newsgroups in Netscape Messanger

È possibile rimuovere la sottoscrizione a un *newsgroup*. Basta cliccare col tasto di destra sul *newsgroup* indesiderato e scegliere l'opzione Unsubscribe.

Outlook Express

Explorer, mediante *Outlook Express*, consente di gestire *newsgroups* e posta elettronica attraverso lo stesso modulo. Infatti, selezionando Strumenti, poi Posta elettronica e News ed infine Leggi *News* si entra nella finestra di Outlook. In Outlook, I *news server* si possono selezionare atraverso l'opzione "Strumenti", poi "Account" e infine selezionando la voce "News". Qui si può aggiungere un *news server* indicato dal proprio *Provider*.



Gestione delle news in Outlook Express

Selezionando il *news server* (cliccando due volte), si ottiene la lista dei *newsgroups* forniti dal *server*, da cui si può scegliere quelle a cui abbonarsi, cliccando sul *newsgroup* e selezionando Subscribe. Come in *Netscape*, la lettura e scrittura di *news* avviene allo stesso modo della lettura e scrittura di *mail*.

Inserire un gruppo di discussione asincrono in una pagina Web

È molto semplice inserire un *link* ad un particolare *newsgroup* in una pagina *Web*. Infatti è sufficiente inserire un *hyperlink* che inizia con news: (invece che con http://) seguito immediatamente dal nome del *newsgroup*. Se il *browser* è settato correttamente (cioè quel *newsgroup* è uno di quelli a cui si è abbonati) allora tutto ciò che si deve fare è di cliccare sul *link* e iniziare a leggere le *news*.

Scripting

Cosimo Laneve

Applicazioni server side

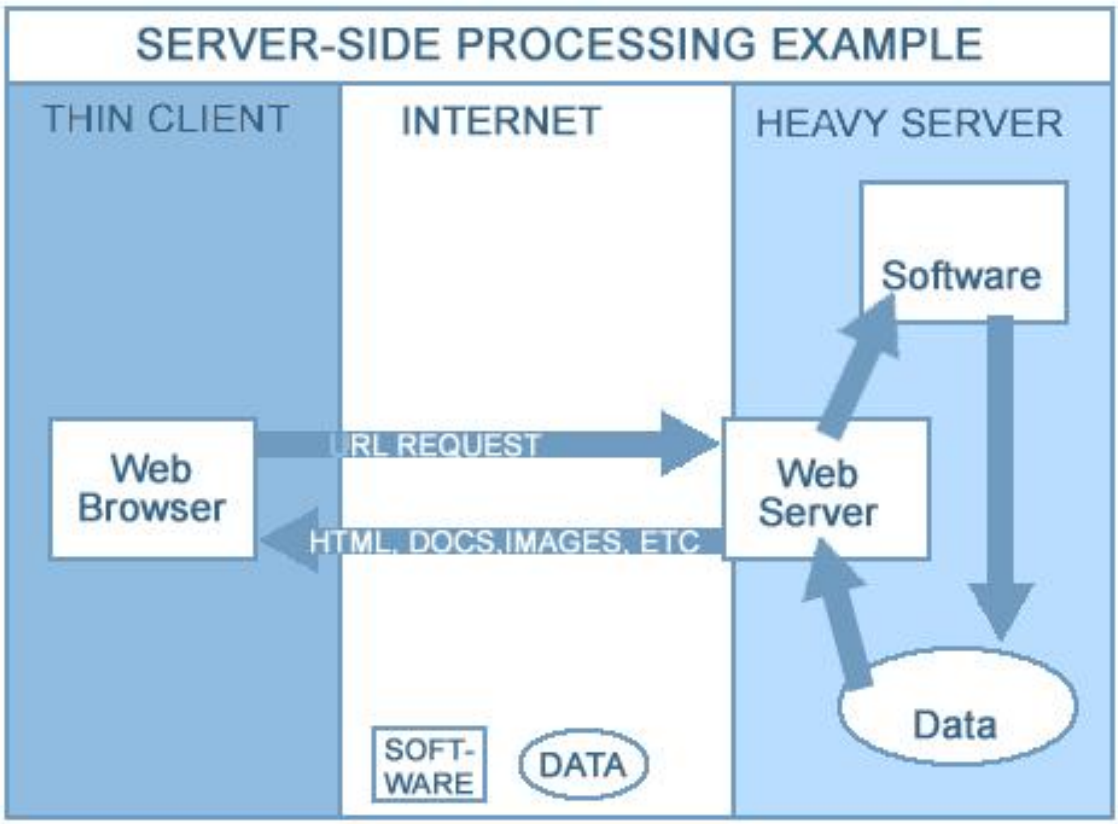
Con tecnologia *Web server-side* si indica, un insieme di meccanismi che permettono al *server Web* di elaborare informazione. Il *server Web* non si limita solo a rispondere a richieste HTTP restituendo documenti **HTML**, ma é in grado di eseguire anche una fase di elaborazione dei dati. L'utente puo' quindi interagire con il *server* ad esempio sottomettendo dati che il *server* elabora e restituisce poi la risposta sotto forma di pagina **HTML**. Un tipico caso é l'accesso da parte dell'utente ad un *database* che risiede sul *server*: in questo caso la pagina *Web* funziona come una interfaccia per accedere ai dati. Si realizza in questo modo un meccanismo di interazione tra l'utente *Web* e le applicazioni/dati che risiedono su un *server* centralizzato.

Ci sono varie tecnologie *server side* utilizzate tra cui **Common Gateway Interface** (**CGI**), **ASP**, **JSP** e **PHP**, **ISAPI**, **NSAPI**, **Servlet**. Queste si distinguono tra loro per sfruttare in modo diverso l'interazione col *Web server*: alcune di esse sfruttano ***Script*** all'interno delle pagine **HTML** (tecnica detta di Inclusione Lato ***Server***) mentre altre utilizzano dei veri e propri programmi, che passano informazioni al *Web server* sfruttando le *API* (*NSAPI*, *ISAPI*) dei *Web server*.

Server Side Include	*API*
ASP (*Microsoft*)	**ISAPI (Microsoft)**
PHP	**NSAPI (Netscape)**
JSP (java server pages)	**Servlet**

Esecuzione di processi lato server

Il passaggio di semplici documenti **HTML** tra il *server* e il *client* non permette lo sviluppo di applicazioni *Web* complesse che coinvolgano una fase di elaborazione oltre che di passaggio di dati. Applicazioni complesse necessitano di un elevato grado di *processing* (*query*, analisi, transazioni...). Dove deve essere eseguita tale computazione? Solo dalla parte server? O è meglio spostare parte del peso della computazione anche sul *client*? E se si, quanto?



esempio di elaborazione lato server

Il modello ***Client-Server*** permette la condivisione di informazioni e il livello di elaborazione è modificabile, in base ad un certo numero di fattori che determinano tale scelta (mercato, esperienza dell'utente, connessione ***Internet***, potenza di calcolo del *computer*...). Per questo motivo sono state sviluppate tecnologie per permettere una maggiore interazione dell'utente con il *server Web* e una capacità di elaborazione sia del *server* che del *client Web*.

I programmi eseguiti dal *server Web* sono detti *server-side*, mentre le tecnologie che aggiungono potere di calcolo al *client* sono dette *client-side*. Globalmente quindi una applicazione *Web* può essere realizzata con tecnologia *server side* quando il peso della computazione risiede tutta sul *server*, oppure con tecnologia *client-side* quando la computazione avviene principalmente sul *browser*. Generalmente le applicazioni *Web* complesse usano entrambe queste strategie.

Tecnologie Web Server-Side e Client-Side

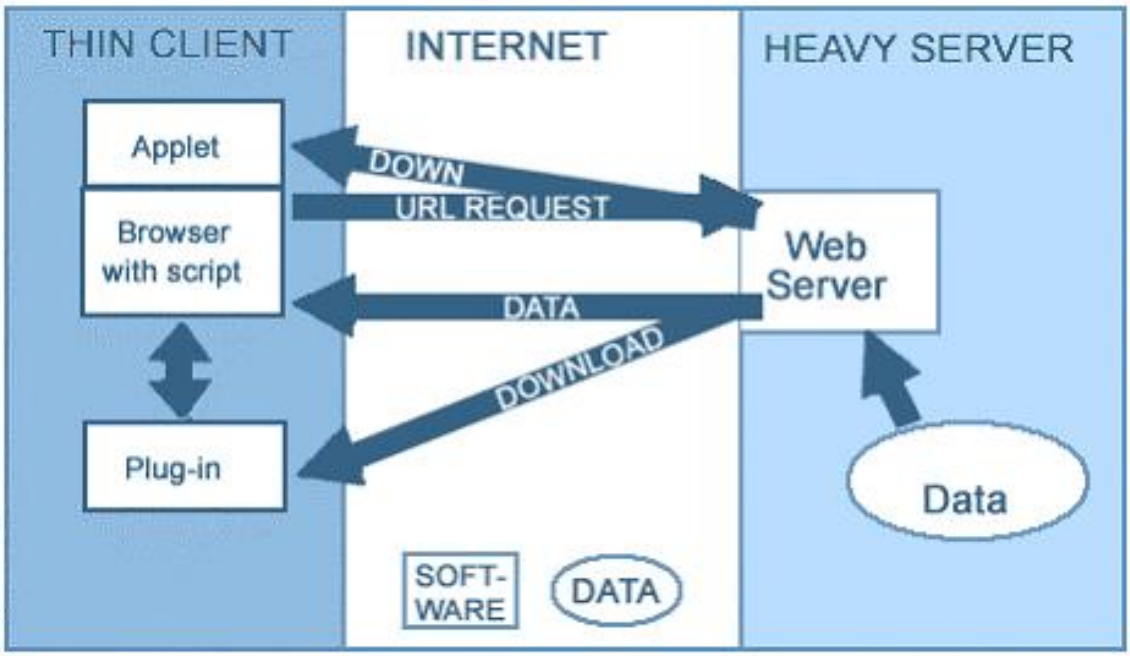
I vantaggi di una tecnologia *server side* sono quelli di avere un sito centralizzato e quindi:

Gli svantaggi del *server side* si hanno quando il processore deve eseguire compiti particolarmente pesanti e quando si devono trasferire sulla rete grossi volumi di dati, che può causare:

- il tempo di risposta può crescere considerevolmente;

- la capacità di calcolo del *client* non viene sfruttata;
- la comunicazione via ***Internet*** e l'elaborazione da parte del *server* è necessaria per ogni richiesta, il che accresce il traffico della rete e il carico del *server*.

Usare una tecnologia *client side* significa spostare parte o tutto il peso della computazione sul *client*. Il *client* di una applicazione *Web* è il *browser*.

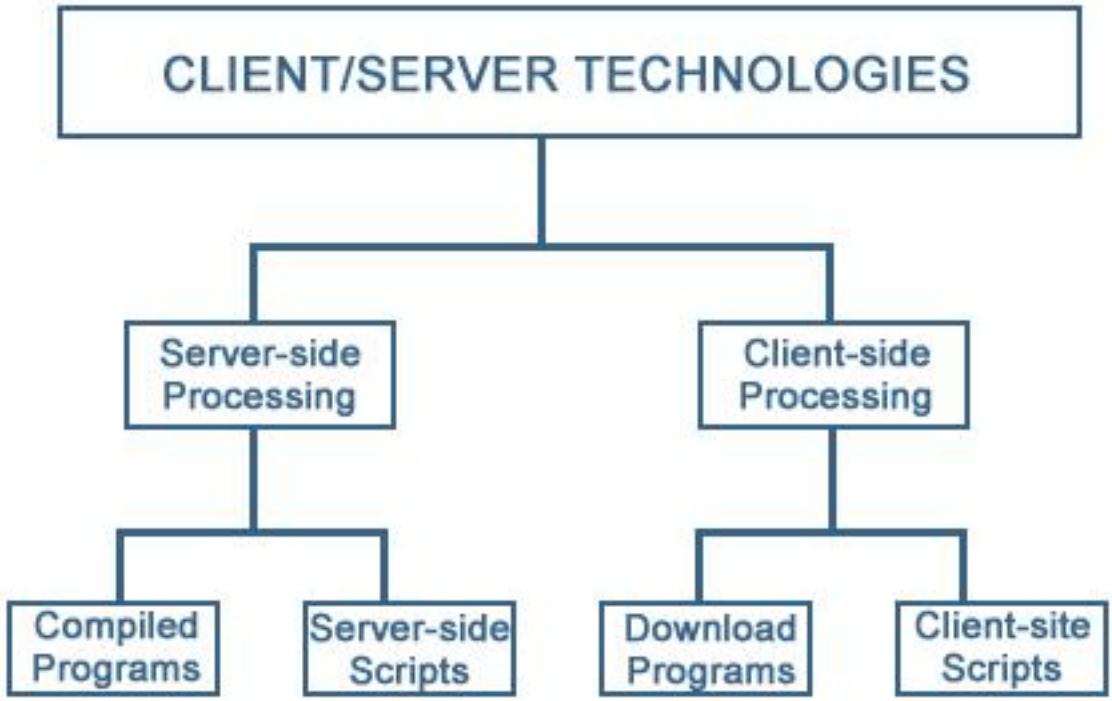


esempio di elaborazione lato client

I *browser Web* sono chiamati thin client cioè *client* che non hanno potere di calcolo, ma solo capacità di visualizzazione. Comunque, i *browser* forniscono meccanismi per includere altre tecnologie come *Java Applets*, *Active X* e *Plug-in*. I vantaggi di usare una tecnologia *client side* stanno nel fatto che i *browser* diventano thick clients, e quindi:

- aiutano a superare gli svantaggi del *server-side* (riducono il carico del *server* e decrementano il traffico in rete)
- danno maggiore autonomia all'utente ad esempio per *map browsing* (pan, zoom), controllo della visualizzazione dei *layers*, *input* di *query* spaziali...
- permettono il trasferimento di dati in forma vettoriale (più piccoli, più veloci, più versatili)

Gli svantaggi del *client-side* sono:



tecnologie client-server

Negli approfondimenti si discutono alcune tecnologie *server-side* molto note: **CGI**, **ASP**, **Servlet** e **PHP**.

Approfondimento

Common Gateway Interface (CGI), Active Server Page (ASP) e Servlet CGI

Cosimo Laneve

15.3.1 (Cenni su CGI, Servlet, ASP e altre principali tecniche di programmazione sul lato server)

CGI

Il protocollo HTTP é cresciuto per includere, oltre a **HTML**, anche altri meccanismi come **CGI** (*Common Gateway Interface*) che permette di costruire pagine dinamiche, cioé pagine che non risiedono staticamente sul *server*, ma che vengono costruite dinamicamente da un programma **CGI** in dipendenza di dati che prevengono da altre fonti, ad esempio inseriti dall'utente o che risiedono su un *database* esterno. Il *browser* puo' richiedere di eseguire un programma **CGI** sul *server*. I **CGI** sono particolari programmi (eseguibili o *script*) che vengono eseguiti sulla macchina *server* e che ritornano l'*output* al *browser*.

Un tipico esempio di uso di programmi **CGI** é il trattamento delle *form* **HTML**. Le *form* in **HTML** sono un meccanismo per permettere all'utente di immettere dati e di attivare in conseguenza dell'invio dei dati, una applicazione sul *server* (la *ACTION* della *form*). Quando un utente compila una *form* su una pagina *Web* e la sottomette, generalmente c'é bisogno di una qualche elaborazione da parte di un programma applicativo che risiede nel *server*. Tale programma riceve le informazioni dal *browser*, ne elabora i dati e poi spedisce indietro al *browser* una risposta. Questo meccanismo fa parte del protocollo HTTP.

Esempio di *form* in **HTML**:

```
<HTML>
.......
<FORM name=prova action="helloworld.pl" METHOD=GET>
<INPUT TYPE=text NAME="Nome">
<INPUT TYPE=submit VALUE="submit" >
</FORM>
....
</HTML>
```

Il passaggio dei dati tra una *form* **HTML** e il **CGI** puo' avvenire secondo due metodi: *GET* e *POST*. Passare i dati di una *form* con metodo *GET* significa codificare i dati nella URL. Esempio di URL con passaggio di dati *GET*:

http://146.48.82.93/lucidiwebgis/helloworld.pl?Nome=Chiara

Con il metodo *POST* invece i dati sono passati con un messaggio di tipo *POST*, come definito nello *standard* HTTP e non appaiono nella URL.

I linguaggi piu' diffusi per scrivere applicazioni **CGI** sono *C, C++, Java* e *Perl*.

Pagina HTML visualizzata sul *browser*:

pagina HTML visualizzata sul browser

Codice HTML

```
<HTML>
<HEAD>
<TITLE>Hello World </TITLE>
</HEAD>
<BODY>
<H2> Hello World </H2>
<p><font face="Arial,Helvetica">
<FORM name=prova action="helloworld.pl" METHOD=GET> Inserire il proprio nome:
<INPUT TYPE=text NAME="Nome">
<P> Premere Submit per attivare il CGI Hello World
<P> <INPUT TYPE=submit VALUE="submit" >
</FORM>
</BODY>
</HTML>
```

Nella *action* della *form* si indica quale programma viene mandato in esecuzione dal *server* quando l'utente sottomette dei dati. Qui, ad esempio, il *server* manda in esecuzione il programma **helloword.pl** che risiede nella *directory* di *default* del *server Web*. Questo è un programma molto semplice che si limita a ricevere in *input* dati da parte dell'utente e restituirli al *browser*. In generale in applicazioni *Web* più complesse i dati inseriti vengono poi elaborati dal *server*, ad esempio inserendoli in una base di dati, oppure inviandoli per *email*, oppure passandoli ad un'altra applicazione che può risiedere sul *server* stesso o in rete. In linea teorica, il **CGI** può comunicare con qualunque applicazione con cui si possa interfacciare il linguaggio con cui è scritto il **CGI**. Supponiamo che il nome inserito sia Chiara, otteniamo:

una pagina HTML generata da uno script CGI

Notiamo che nella casella di *location* appare il parametro passato con il metodo *GET*. Se avessimo usato il metodo *POST* il parametro non sarebbe stato visibile nella URL.

Esempio di sorgente del programma helloword.pl scritto in *Perl*

```
#!usr/local/bin/perl
  local(%in) ;
  local($name, $value) ;
  # Resolve and unencode name/value pairs into %in
  foreach (split('&', $ENV{'QUERY_STRING'})) { s/\+/ /g ;
    ($name, $value)= split('=', $_, 2) ;
    $name=~ s/%(..)/chr(hex($1))/ge ;
    $value=~ s/%(..)/chr(hex($1))/ge ;
    $in{$name}.= "\0" if defined($in{$name}) ;
  # concatenate multiple vars
    $in{$name}.= $value ;
}
print "Content-type: text/html\n\n";
print $in{$name} ;
print ", questa é la tua pagina di Hello World";
```

Limiti di *performance* della CGI

Sebbene la **CGI** costituisca uno strumento sostanzialmente duttile per ottenere interattività sul *Web*, ammettendo l'uso di diversi linguaggi, interpretati o compilati, permettendo, in accordo con la politica di sicurezza adottata, di accedere potenzialmente a tutte le risorse del sistema e venendo così incontro alle esigenze più eterogenee, soffre di un limite importante, insito nel proprio meccanismo di funzionamento. Ogni volta che da un *browser* viene lanciata l'esecuzione di uno *script*, il *server*, ricevuta la richiesta, crea un nuovo processo, e questo, per un sito ad alto traffico può portare ad un superlavoro per il processore, con conseguente drastico decadimento di tutte le prestazioni del sistema.

I programmi CGI

Un primo passo verso un uso più dinamico del *Web*, che lo facesse evolvere da una semplice collezione di ipertesti ed ipermedia è stato quello di permettere ai *server* di comunicare con applicazioni esterne. In questo contesto nasce la *Common Gateway Interface*, che è, essenzialmente, un processo di tipo *server-side* che fa da tramite tra il *server Web* ed altre applicazioni o risorse generiche (*database*, immagini, video, eccetera) residenti anche su macchine diverse, fornendo un'interfaccia per apposite applicazioni esterne, denominate anche gateway programs, *script* **CGI** o, ancora, programmi **CGI**. Tale interfaccia permette di astrarre dai dettagli della comunicazione dei dati, offrendo al programmatore la possibilità di focalizzare l'attenzione esclusivamente sui due fattori per lui più importanti:

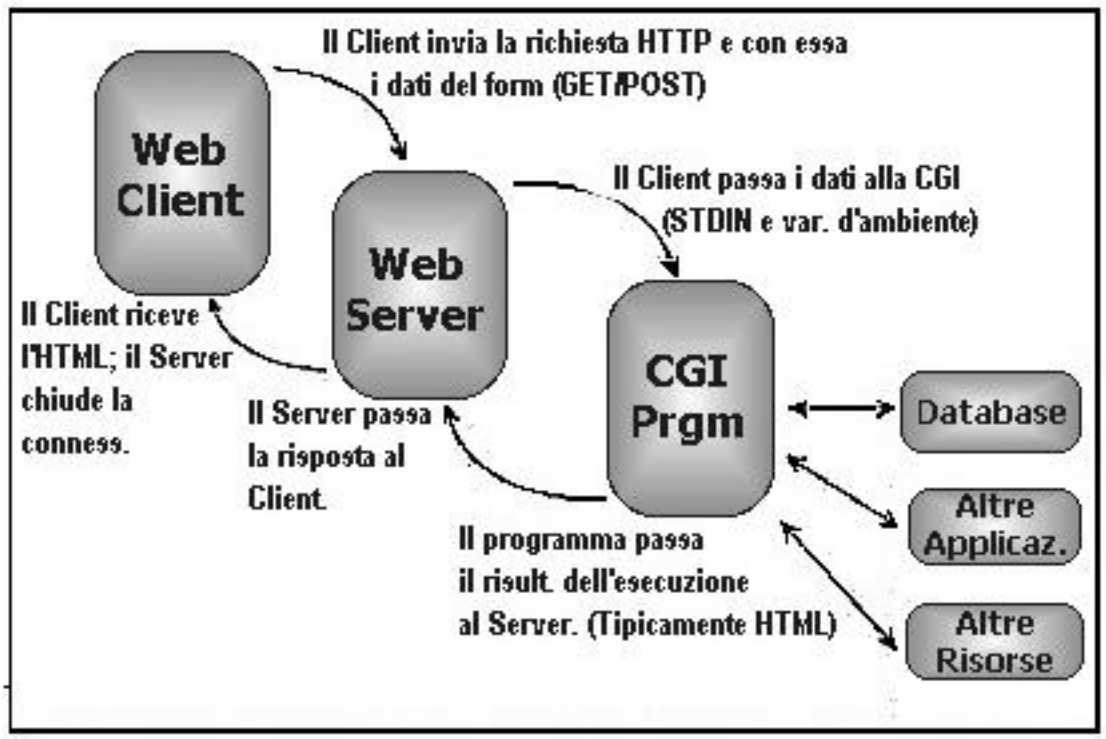
- quali dati fornire come *input*;
- come elaborare i dati ottenuti in *output*;
- i meccanismi da imparare sono pertanto quelli necessari a maneggiare i dati che il *server* passa al programma e a restituire i dati da questo generati.

I programmi **CGI** estendono le funzionalità base del *server Web*, dandogli la capacità di servire una varietà di richieste utente che altrimenti non potrebbe gestire. Contengono il codice che permette di ricevere i dati dal *server* ed elaborarli a seconda delle necessità. Possono essere realizzati in qualsiasi linguaggio supportato dal *server Web*.

Flusso dei dati in un processo CGI

Tipicamente, il flusso dei dati in un processo **CGI** avviene come in figura, seguendo questo iter:

- un *client Web* realizza una connessione con un *server Web* (tipicamente tramite un *browser*), all'indirizzo specificato nella *URL*.
- Il *client Web* invia una richiesta (tramite uno tra i metodi *POST* e *GET* visti nel paragrafo precedente).
- I dati inviati dal *client* sono passati dal *server* al programma **CGI** referenziato nella *URL*.
- Il programma **CGI** legge i dati ed esegue il processo per cui è stato creato.
- Il programma **CGI** genera un risultato da restituire al *client* tramite il *server*, come risposta alla richiesta del punto 2. Tale risultato è tipicamente sotto forma di documento **HTML**, ma può essere anche un altro tipo di documento.
- Dopo aver passato la risposta fornita dal programma **CGI** al *client*, il *server* chiude la connessione aperta al punto 1.



Passi di una elaborazione server-side

CGI e sicurezza

Eseguire un programma **CGI** comporta dei rischi. Per usare le parole di Bob Breedlove, esperto nell'uso di questa tecnica, è un po' come invitare il mondo ad eseguire un programma sul nostro sistema. In effetti, ciò che avviene è proprio che da un generico *browser*, vengano lanciate, sul nostro *server*, delle attivazioni di processi. Attivazioni, per altro, facilmente localizzabili e modificabili, giacché sono innestate nelle pagine **HTML** caricate sul *browser* del *client*. Qualora qualche malintenzionato riuscisse a modificare in maniera congruente il nome del processo da eseguire, attivandone un altro, gli effetti sul sistema potrebbero essere disastrosi. Se il malintenzionato fosse anche in grado di installare del codice malizioso sul ***Server***, una sorta di virus informatico, tramite il meccanismo delle **CGI** sarebbe in grado di attivarlo comodamente seduto davanti al proprio computer! Per evitare tali abusi, sono state introdotte regole e restrizioni. La maggior parte degli HTTP *daemon* pone i seguenti limiti ai programmi **CGI** da eseguire:

- limiti sulle azioni da eseguire. Ai programmi **CGI** si impediscono quelle azioni ritenute particolarmente pericolose per il sistema, come, ad esempio, la cancellazione dei *file* o l'installazione di programmi eseguibili.
- Limiti sul campo di accessibilità alle informazioni. Consistono nel rendere le *directory* o i singoli *file* ritenuti di interesse riservato inaccessibili ai programmi **CGI**.
- Limiti sulla posizione dei programmi **CGI** eseguibili. Raggruppando tutti gli *script* eseguibili dal *server* in una *directory* si può evitare l'attivazione dall'esterno di codice malizioso nascosto chissà dove nel sistema. Il

raggruppamento semplifica, inoltre, il controllo continuo sul codice di *script* installato.
- L'utilizzo ulteriore di *password*, in alcuni casi, può permettere di alleggerire i limiti.

Gestione dell'I/O

Un *server Web* ed un programma **CGI** possono comunicare e passarsi i dati l'un l'altro in quattro modi:

- **Variabili d'ambiente**: contengono valori settati dal *server Web* che deve eseguire lo *script* ed ivi mantenute. Si distinguono due tipi di variabili d'ambiente:
  - quelle il cui valore è settato indipendentemente dal tipo di richiesta del *client*.
  - quelle, denominate request-specific, dipendenti dal tipo di richiesta effettuata dal *client*. Fra queste una serie di variabili contenenti i valori delle variabili di richiesta HTTP trattate nel **paragrafo 2.1.** Queste variabili permettono di accedere ai dati forniti dal *client*, di determinare il tipo di *browser Web* in uso, di mantenere e passare informazioni sullo stato tra diverse richieste indipendenti, sopperendo così allo svantaggio del linguaggio **HTML** di essere *stateless*.

  La lista completa di tutte le variabili d'ambiente è lunga e può variare da *server* a *server*. È comunque utile esaminarne qualcuna per capire, nella pratica, come e dove intervengano. Le prime tre variabili esaminate coincidono con le variabili di richiesta HTTP inviate al ***Server*** dal *browser* del ***Client*** per riconoscere il tipo di trasmissione (*GET/POST*) e gestirla di conseguenza.

  - **REQUEST_METHOD**. Specifica il metodo di richiesta (*GET* o *POST*) adottato dal *client*.
  - **QUERY_STRING**. *QUERY_STRING* contiene tutto ciò che compare nella *URL* al momento della chiamata del programma **CGI**.
    Consideriamo, ad esempio, il seguente *URL*:

    http://www.uniud.it/cgi-bin/test.cgi?test_di_prova

    Questo va ad attivare lo *script* test.cgi nella *directory* cgi-bin presente sull'*host* il cui indirizzo ***Internet*** è **www.uniud.it**. L'*input*, costituito dalla stringa test di prova viene passato al programma ponendolo nella variabile QUERY_STRING come segue: ***QUERY_STRING* = test+di+prova**.

    Osservazione: Il *browser* pone i valori nella *URL*, come nell'esempio appena visto, in due casi: quando al suo interno viene definito un *form* **HTML**, con il metodo *GET* specificato, oppure quando nella testata della pagina **HTML** viene posizionato un particolare elemento, chiamato ISINDEX, che provoca l'inserimento nella pagina di un campo di *input* simile al controllo per l'inserimento di una stringa, già visto esplorando i *form*. In questi due casi, quando l'utente esegue il comando di *SUBMIT* (ovvero clicca sul bottone atto a far eseguire l'operazione di invio dati - *data submitting*), il *browser* legge i valori e li posiziona a completamento dell'*URL*.

- **CONTENT_LENGTH**. Rappresenta la dimensione, in caratteri , del *buffer* dati trasferito dal *client* al *server* durante una richiesta. Qualora venga utilizzato il metodo *POST*, il *server* non invia alcun indicatore di *fine-file*, ecco quindi che *CONTENT_LENGTH* diventa fondamentale per conoscere l'esatta dimensione dell'*input* da leggere.

  Esempio: Considerando il seguente *form* **HTML**

  <FORM ACTION = "...." METHOD=POST>
  <INPUT NAME="A" SIZE=5> {Input="A B C"}
  <INPUT NAME="B" SIZE=4> {Input="1234"}
  </FORM>

  All'atto dell'invio dei dati il programma **CGI** riceve i seguenti valori:

  CONTENT_LENGTH = 14
  STDIN: A=A+B+C&B=1234

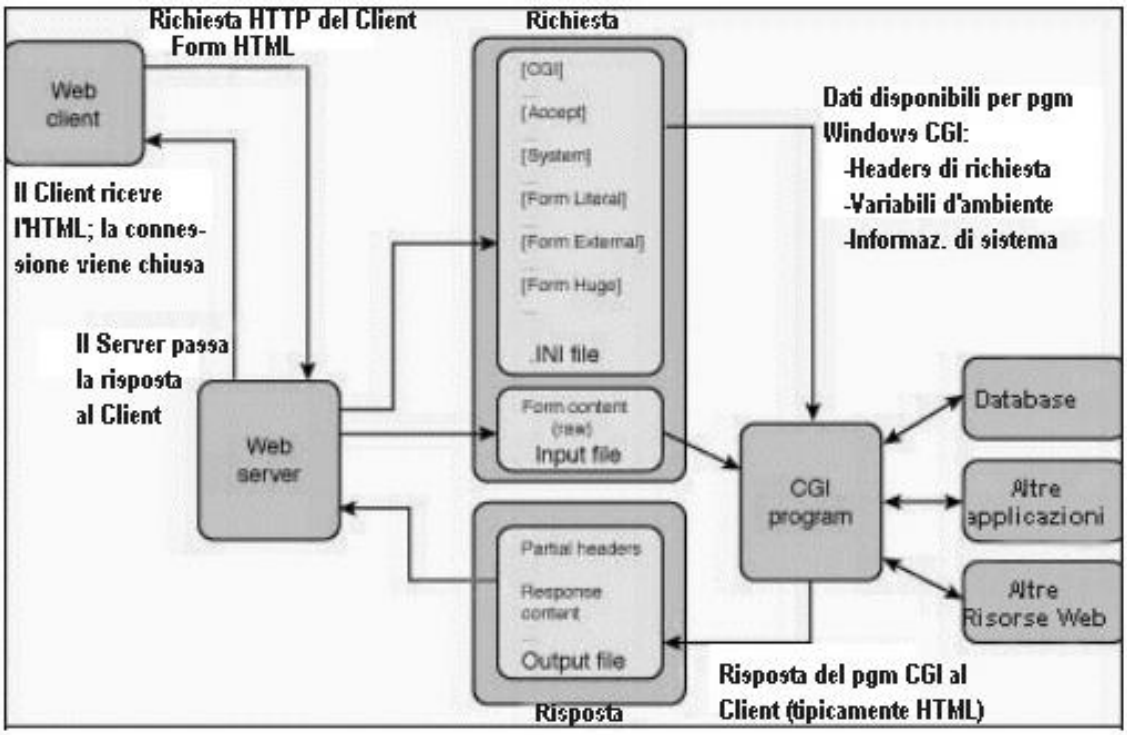
- **AUTH_TYPE**. Identifica il metodo da utilizzare per validare gli utenti nel caso il *server* supporti l'autenticazione ed esegua i programmi **CGI** in modalità protetta.

- **Standard *Input***: è lo *standard input file descriptor* di sistema. Su molti sistemi *Unix*, ad esempio, lo *Standard Input* è il *buffer* dove un comando o un programma leggono il proprio valore di *input*. Tipicamente coincide con un dispositivo terminale di *input* o con l'*output* di un altro programma.
- **Standard *Output***: è lo *standard output file descriptor* di sistema. Tipicamente coincide con un dispositivo terminale di *output* o con l'input di un altro programma.
- **Command *line***: è un metodo attraverso il quale i dati vengono prelevati e passati al programma così come sono stati messi nella riga di comando resa disponibile dall'elemento ISINDEX.

Per passare i valori al programma **CGI** viene utilizzato lo *standard Unix*, inviando un *array* di puntatori a stringhe, che formano l'*input*, *ARGV*, ed una quantità, *ARGC*, indicante il numero di valori significativi nell'*array ARGV*. Un metodo addizionale di comunicazione è stato sviluppato per un'implementazione specializzata della **CGI**, conosciuta come *Windows* **CGI** (o *WinCGI*) ed operante su macchine con sistema operativo *MS Windows*. Tale metodo si basa sull'utilizzo di un *file* temporaneo caratterizzato dall'estensione .INI, che viene utilizzato come segue:

- al momento del ricevimento della richiesta del *client*, il *server* preleva i dati inviati nell'*header* della stessa, li converte in variabili associate ad un nome, combina queste con altre variabili d'ambiente e le salva nel *file* .INI. Ulteriori dati ricevuti dal *client* e non presenti nell'*header*, vengono invece salvati in un altro *file* temporaneo (lo identificheremo come *file* temporaneo di *input*).
- Il *server* decodifica ed analizza il contenuto di ogni *form* presente nella richiesta. Associa una variabile ad ogni campo del *form* ed aggiunge queste al *file* .INI.
- Il *server* crea un nome per il *file* temporaneo di *output*, e lo aggiunge, assieme al nome del *file* temporaneo di *input*, al *file* .INI. A questo punto, tutti i dati necessari all'esecuzione del programma **CGI** sono presenti nel *package* formato dal *file* .INI e dal *file* temporaneo di *input*.

- Il programma **CGI**, una volta lanciato dal *server* tramite la procedura di sistema *CreateProcess*(), localizza il *file* .INI (l'indirizzo viene fornito a cura della CreateProcess).
- Dopo aver seguito i suoi compiti, il programma **CGI** tipicamente genera una risposta da inviare al *browser* del *client*. A tal fine crea un *file* temporaneo di *output*, utilizzando il nome creato precedentemente dal *server*, e vi scrive i dati. Questi sono organizzati in due sezioni: una intestazione atta a definire il tipo dei dati che può variare da semplice testo *ascii* piano, a testo **HTML** a immagini o altro ancora, e l'effettivo contenuto informativo della risposta.
- Non appena il *server* determina la fine dell'esecuzione, legge i dati dal *file* temporaneo di *output*, li riorganizza in modo da creare una risposta aderente al protocollo HTTP, e così li invia al *browser* del *client*.



Flusso dei dati in un processo Windows CGI

ASP

Superare i limiti dell'html per creare dei siti sempre più rispondenti alle esigenze dei visitatori è stato una delle mete a cui i programmatori di linguaggi di *scripting* hanno puntato nel corso della storia del *Web*. Dalle prime pagine statiche, manifesto di un sito, si è progressivamente arrivati non solo all'esplosione del multimediale, ma, soprattutto, al diffondersi di pagine interattive, in grado non solo di affascinare, ma di fornire un utile strumento a chi le volesse usare. Oggigiorno è possibile, grazie ai nuovi linguaggi di *scripting*, superare la staticità delle pagine *Web*, mantenendo al contempo una semplicità di programmazione che consenta a tutti di intervenire senza prima dovere leggere voluminosi manuali.

Fra tutti si distingue sicuramente **ASP** (*Active **Server*** Pages) per la rapidità e flessibilità di utilizzo che lo caratterizzano, che però sono controbilanciate da uno svantaggio non

indifferente: l'utilizzo di questo linguaggio è confinato ai *server Microsoft* come ad esempio IIS, e non funziona quindi con tutti gli altri *server* che popolano il *Web*. La sempre maggiore diffusione dei *server Windows* contribuisce però a rendere meno limitante questo ostacolo e, tutto sommato, non è difficile vedere diversi *provider* abbandonare il mondo *Unix* per le nuove possibilità offerte da *Windows NT*.

Grazie all'utilizzo delle pagine **ASP**, l'utente può quindi creare dei documenti che possono fornire informazioni, rispondendo in modo diverso alle differenti richieste dei navigatori. Ma quali sono, in breve, i vantaggi nell'utilizzo di questo linguaggio di *scripting*?

- Le pagine **ASP** sono completamente integrate con i *file* html, essendo in definitiva loro stesse pagine **HTML**
- Sono facili da creare e non necessitano di compilazione.
- Sono orientate agli oggetti e possono accedere a componenti *server* **ActiveX**.

Visti i vantaggi, e viste anche le limitazioni cui abbiamo accennato in precedenza, riassumiamo le tecnologie coinvolte nello sviluppo e funzionamento delle *active server pages*:

- *Windows NT* (l'utilizzo di altri sistemi sebbene consentito è sconsigliato):
- Un *Web server* che supporti *Active* ***Server***, come *IIS* (oppure *PWS* per un uso personale)
- *ODBC* (*Open DataBase Connectivity*) o tecnologia ADO per l'acceso ai dati.

Le basi di ASP

Esaminando più da vicino l'anatomia di questo genere di pagine possiamo constatare che esse sono costituite da differenti parti:

- Marcatori html
- Comandi *script*

In un documento con estensione .asp è consentito utilizzare variabili, cicli, istruzioni di controllo, eccetera, grazie alla possibilità di richiamare la sintassi un linguaggio di *scripting*, come ad esempio il VB Script e il *JScript*, ma anche *Perl*. Una prima distinzione che possiamo operare a livello di codice sorgente è a livello di comandi nativi, propri di **ASP**, e comandi di *scripting* che appartengono al particolare linguaggio utilizzato. Tra i marcatori fondamentali di **ASP** ci sono sicuramente i delimitatori, che come nell'html delimitano l'inizio e la fine di una sequenza di codice, e sono rappresentati dai simboli "<%" e "%>".

Ad esempio il comando seguente assegna alla variabile x il valore ciao.

```
<% x="ciao" %>
```

Abbiamo già detto che è possibile includere anche *script* nel codice asp e utilizzare così funzioni create, ad esempio, in *JScript* o *VBScript*, richiamandole tramite il comando nativo <% Call _ %>, come nell'esempio 1 che mostra come costruire una pagina che visualizzi la data del giorno corrente:

```
<% Call PrintDate %>
```

```
<SCRIPT LANGUAGE=JScript RUNAT=Server>
function PrintDate() {
var data
// data è un'istanza dell'oggetto Date
data = new Date()
// il comando Response.Write scrive la data sul navigatore
Response.Write(data.getDate())
}
// Questa è la definizione della procedura PrintDate.
// Questa procedura manderà la data corrente al navigatore.
</SCRIPT>
```

La funzione PrintDate definita in JScript è scritta tra i marcatori <SCRIPT> e </SCRIPT> come sempre, però questa volta sono stati inclusi gli elementi LANGUAGE=JScript e RUNAT=server. Un indubbio vantaggio che deriva dall'uso del delimitatore RUNAT di *script* è costituito dal fatto che il codice sorgente non è mai presente nella pagina html che viene spedita al navigatore dal *server*. Infatti, il sorgente viene rielaborato dal *server* che invia come risultato una pagina costruita al volo nella quale sono visibili solo i codici html e quelle funzioni per le quali non sia stato specificato il valore server. E' interessante notare che non è possibile utilizzare i delimitatori <% %> per definire una funzione, dato che non è possibile assegnare nomi a blocchi di codice **ASP**; in seguito vedremo come includere dei *files* utilizzando il comando <!-- #include --> .

L'oggetto response

Passiamo ora all'esame dell'oggetto *Response*, il quale consente di gestire l'interazione fra il *server* e il *client*. Questo oggetto possiede una serie di metodi che consentono di effettuare una serie di operazioni che avremo occasione di osservare più dettagliatamente nelle varie lezioni di cui si compone questo corso. Ecco di seguito un elenco dei metodi sopra citati:

- AddHeader
- AppendToLog
- BinaryWrite
- Clear
- End
- Flush
- Redirect
- Write

Tenendo presente che il metodo write richiede una stringa di testo tra virgolette, o una funzione che restituisca una stringa, esaminiamo l'esempio 2 che illustra come creare e chiamare delle procedure usando due differenti linguaggi di *scripting* (*VBScript* e *JScript*).

```
<html>
<body>
<table>
<% Call Echo %>
</table>
<% Call PrintDate %>
```

```
<SCRIPT LANGUAGE=VBScript RUNAT=Server>
Sub Echo
Response.Write "<tr><td> Name </td><td>Value </td></tr> "
' L'istruzione Set imposta la variabile Params
Set Params = Request.QueryString
For Each p in Params
Response.Write " <tr><td>" & p & "</td><td> " & Params(p) & "</TD></TR> "
Next
End Sub
</SCRIPT>
<SCRIPT LANGUAGE=JScript RUNAT=Server>
function PrintDate() {
var x
x = new Date()
Response.Write(x.toString())
}
</SCRIPT>
```

L'oggetto Request

Nel precedente esempio abbiamo introdotto anche il metodo *Request*, che può essere definito come il primo oggetto che incontriamo ad essere intrinseco al *server*, dato che esso rappresenta l'elemento di connessione tra il programma *client* ed il *Web server*. In pratica, esso si occupa di trasmettere le informazioni provenienti da alcune variabili del *server* (le collections), mentre l'oggetto *Response* si occupa dell'interazione tra *server* e *client* tramite l'utilizzo di metodi quali *write*, che permette la scrittura a *video*. La sintassi propriadi questo oggetto è:

Request[.Collection] ("variabile")

Ora, per cominciare, *date* un'occhiata a questa semplice pagina asp che consente di visualizzare il classico testo *Hello World*! in una pagina html. La particolarità di questo testo consiste nel potere apparire in una dimensione variabile da 3 punti per carattere fino a 7, grazie ad un comando VBScript (*For To*) che controlla un ciclo di assegnazione di valori alla variabile i, la quale a sua volta definisce la grandezza del parametro html *SIZE*; il ciclo viene effettuato su tutto ciò che si trova tra *For To* e *Next*.

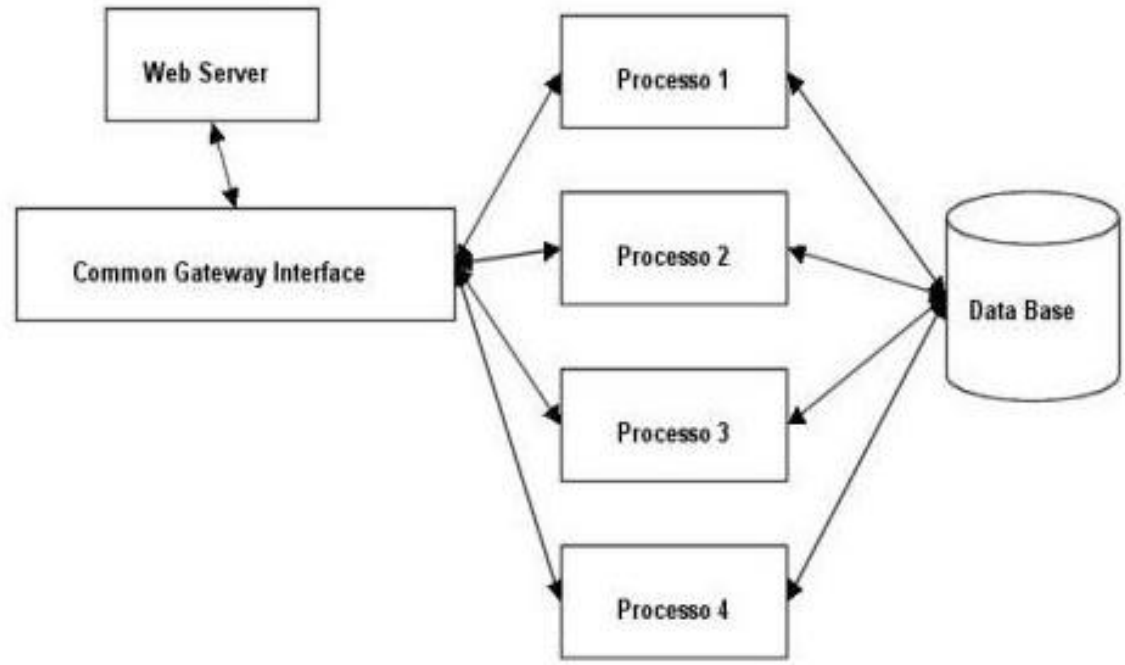
```
<% For i = 3 To 7 %>
<FONT size="<%=i%>">
Hello World!<BR>
</FONT>
<% Next %>
```

Un altro utilizzo dell'oggetto *Request* che utilizzando il metodo ***Server****Variables*, permette di richiedere al *server* una delle variabili di sistema, come ad esempio HTTP_USER_AGENT, che identifica il nome del navigatore che il *client* sta usando per richiedere la pagina.

```
<%a=Request.ServerVariables("HTTP_USER_AGENT")%>
<% Response.write(a)%>
```

I Servlet

Ogni volta che un *server WEB* ha bisogno di elaborare dati inviati da un *client* viene attivato un programma **CGI**, solitamente scritto in un linguaggio *C-like*. Il **CGI** riceve dei parametri in base ai quali esegue una determinata operazione, ottiene un risultato, lo invia al *server* sotto forma di pagina *Web*, e termina la sua esecuzione. Questo vuol dire che per ogni richiesta proveniente da un *client*, viene caricata, eseguita e quindi terminata una nuova istanza dello stesso programma **CGI**. Tale procedura, pur essendo attualmente la più diffusa nel mondo dei *Web server*, costituisce un grosso carico di lavoro per il *server*: tra l'altro per esecuzioni parallele dello stesso *script* **CGI** viene ricaricata in memoria tutta l'applicazione.



server con diversi processi CGI attivi

I **servlet** sono applicazioni scritte in *Java* in grado di assolvere agli stessi compiti fino ad oggi delegati ai programmi **CGI**, e la loro rapida diffusione dimostra che essi ne rappresentano una valida alternativa. I vantaggi offerti dai *servlet* sono da ricercare innanzitutto nelle *Java* ***Servlet*** *API* grazie alle quali è assicurata una totale portabilità. Lo sviluppo di *servlet* in *Java* risulta inoltre meno complesso e quindi più robusto di un programma scritto in C o *Perl*. Sebbene questi vantaggi siano ottenuti a scapito di un tempo di esecuzione generalmente più lungo, al fine di garantire un'efficienza paragonabile a quella dei **CGI**, il meccanismo di esecuzione dei *servlet* prevede un unico caricamento al momento della prima esecuzione. Questo vuol dire che una volta attivato ed inizializzato, il *servlet* rimane in memoria ed è pronto a soddisfare le richieste provenienti dai *client*, anche paralleli. L'esecuzione del *servlet* termina solo quando viene invocato un particolare metodo che provvede a rilasciare la memoria occupata e concludere l'esecuzione. La possibilità di richiamare all'interno di un *servlet* altri *servlet*, i modelli di sicurezza ereditati dal linguaggio *Java*, la semplicità con cui possono essere modificati e aggiornati sono altri dei vantaggi che caratterizzano i *servlet*.

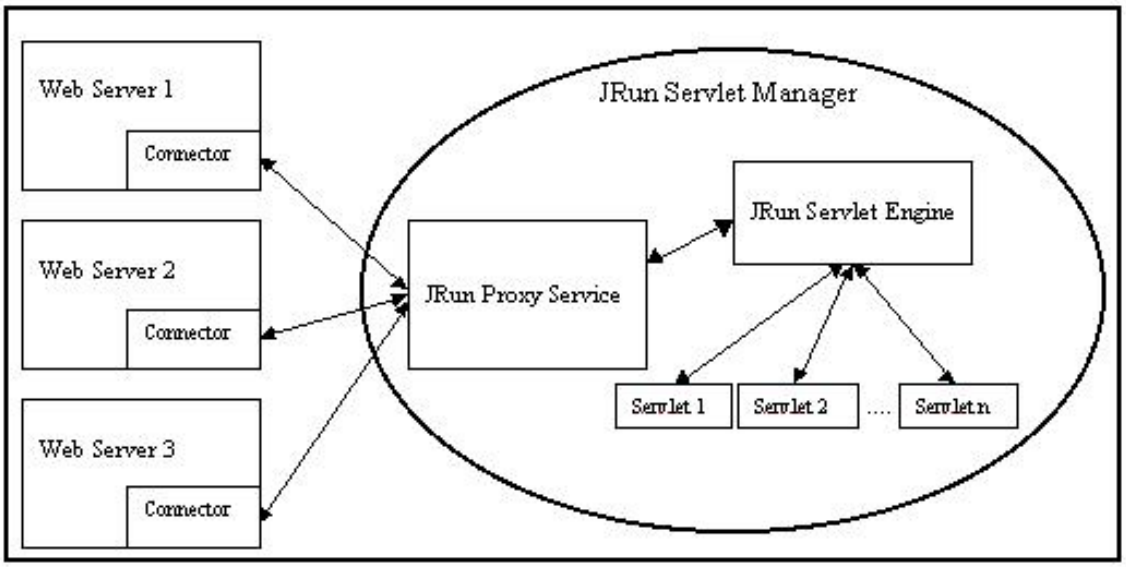
Principio di funzionamento

Per eseguire un *servlet* è necessario collegare al *Web server* un **engine** e configurare il *Web server* in modo che sia in grado di riconoscere le richieste di avvio di *servlet* provenienti dai *client*. Tale *engine* in pratica si comporta come la macchina virtuale di *Java* su un *client*, però è attiva server-side. Un *servlet* generalmente riceve richiesta di un servizio tramite una operazione di *POST* eseguita ad esempio tramite una *form* **HTML**, quindi elabora i dati ricevuti e genera dinamicamente una pagina **HTML** in risposta. Il più diffuso tra i *servlet engine* è sicuramente **JRun** (Java **Servlet** *Runner*), ma alcuni *Web server* ne hanno di proprietari, ed il funzionamento è grosso modo lo stesso. Il componente principale di *JRun* è *JRun Service Manager* (*JSM*) che si occupa di caricare ed inizializzare i servizi. I servizi offerti sono:

- *JRun* ***Servlet*** *Engine Service* (*JSE*): si occupa di caricare i ed eseguire i *servlet*
- *JRun Connector* ***Proxy*** *Service* (*JCP*): è il modulo che consente la comunicazione tra il *Web server* e *JSE*
- *JRun Web* ***Server*** *Service* (*JWS*): un *Web server* completamente scritto in *Java* tramite il quale è possibile testare i *servlet*.

JRun Service Manager viene avviato, ed avvia i servizi sopra elencati, in maniera indipendente dal *Web server* al quale è connesso. Quest'ultimo agisce come un *client* richiedendo a *JSM* l'esecuzione di un *servlet* e ottenendo in risposta un risultato. Questo modo di procedere, indicato col nome di out of process presenta alcuni importanti vantaggi: è possibile eseguire i *servlet* su una *Java Virtual Machine* diversa da quella adottata dal *Web server*; limita l'interferenze tra *JRun* ed il *Web server* aumentandone la stabilità; rende *JRun* indipendente da eventuali cadute del *server*; consente di collegare più di un *Web server* ad un unico *JSM*.

Al momento del collegamento di un *Web server* con JRun quest'ultimo provvede ad installare un "connector" per mezzo del quale il *Web server* comunica, tramite un *socket*, col servizio *proxy* (JCP) di JRun. Il connector cambia a seconda del *Web server* su cui viene installato e provvede a tradurre le richieste che dal *Web server* vengono inviate a JCP e le risposte che si transitano in senso opposto. Il tutto accade secondo lo schema della figura seguente.



interazioni tra le componenti del JRun Servlet Manager (JRun Proxy Service, JRun Servlet Engine e Servlet) e WebServer

La procedura da adottare per eseguire un *servlet* è del tutto analoga a quella usata per l'esecuzione di un qualsiai programma **CGI**. Ad esempio è possibile accedere ad un *servlet* tramite un *Web browser* utilizzando la sintassi:

http://www.dominio:numeroporta/servletdirectory/nomeservlet

Dove *servletdirectory* indica la *directory* in cui sono contenuti i *servlet*. Come accade per i **CGI**, ricordiamo che è necessario configurare il *Web server* in modo che sia in grado di distinguere le richieste di attivazione di un *servlet*. Ad esempio di potrebbe decidere che tutte le richieste contenenti il testo servlet si riferiscono ad applicazioni di tipo *servlet*. E' possibile richiedere l'avvio di una lista di *servlet* tramite la richiesta:

http://www.dominio:numeroporta/servletdirectory/servlet1,servlet2,..., servletn

In questo modo verrà avviato prima il servlet1, quindi completatane l'esecuzione verrà avviato il servlet2, al quale sarà dato in *input* l'*output* del servlet1, e così via fino all'ultimo *servlet* richiesto, il cui *output* verrà restituito al *client*.

Tipicamente il meccanismo di funzionamento di un *servlet* è il seguente:

- il *client* (il *browser*) richiede ad un *Web server* remoto una pagina html al cui interno è presente un riferimento ad un *servlet*.
- Il *Web server* invia la pagina richiesta, e, manda in esecuzione il *servlet*, il quale effettua tutta una serie di operazioni ed eventualmente invia al *client* un risultato che, incapsulato nella pagina html, verrà visualizzato all'interno della finestra del *browser*.
- Il *servlet* quindi viene eseguito da una *JVM* inserita direttamente del *Web server*.
- La caratteristica più importante di un *servlet* è che, contrariamente ad un programma *standard* **CGI**, è che non viene creato un processo ogni volta che il *client* effettua la richiesta, ma solo la prima volta. Questo comporta

grossi vantaggi in termini di prestazioni e di potenzialità (ad esempio più facile gestione del mantenimento dello stato).

Il codice per creare servlet

I due metodi principali di un *servlet* sono:

- **init()**: rappresenta il momento della creazione ed istanziazione del *servlet*: come nelle *applet*, serve per inizializzare tutti i parametri e le variabili da utilizzare per il funzionamento. Nella init spesso si ricavano i parametri passati al *servlet* dal sistema.
- **service()**: rappresenta la richiesta da parte del *client* http sotto forma di una *GET* o *POST* http. I due parametri fondamentali del metodo *service* sono *HttpServletRequest, HttpServletResponse*. Il primo rappresenta la richiesta http (con il quale ottenere informazioni sulla richiesta, come i parametri), mentre il secondo identifica la risposta con la quale restituire informazioni al *client*.

Per creare un *servlet* è sufficiente estendere la classe *Http Servlet* ed implementare questi due metodi. In questo esempio si invia come parametro al *servlet* il proprio nome per mezzo di un *form* ed il *servlet* risponderà con un saluto indirizzato al nome inviato come parametro. Per far questo dobbiamo:

- ricavare il parametro
- inviare il risultato sotto forma di pagina html in modo che il *browser* possa visualizzarla

Ricavare i parametri della richiesta questo compito può essere svolto con la semplice istruzione

String param=req.getParameter(nome_parametro);

dove req è lo *stream* corrispondente alla richiesta. Per inviare un qualsiasi risultato all'utente dobbiamo per prima cosa ricavare lo *stream* che è rediretto nella finestra del *browser*. Questa operazioni viene gestita automaticamente dal sistema (*JVM+ Web Browser*), e per la creazione del *servlet* si traduce nelle semplici istruzioni che seguono

```
ServletOutputStream out;
res.setContentType("text/html");
out = res.getOutputStream();
```

La prima riga definisce un paricolare tipo di *stream* utilizzabile all'interno dei *servlet*, mentre la seconda prepara tale *stream* per l'invio di dati di tipo testuale/html. Infine con getOutputStream() otteniamo il riferimento allo *stream* di *output* vero e proprio. Per invocare un *servlet* è necessario eseguire una chiamata del tipo

http://nome_host/nome_servlet?parametro1=valore1&parametro2=valore2

se si vuole invocare un *servlet* per mezzo di un *form* dovremo scrivere del codice html in maniera opportuna come ad esempio

```
<form action="http://host/serlvet" method="POST">
<table BORDER=0 COLS=2 WIDTH="40%" >
<tr>
```

```
<td >Nome Utente</td> <td><input type="text" name="user_id"></td>
</tr>
<tr>
<td>Password</b></td>
<td><input type="text" name="password"></td>
</tr>
</table>
```

Per quanto riguarda l'installazione di un *servlet*, si seguano le istruzioni del particolare *Web server* utilizzato: in genere l'operazioni si traduce nel copiare il *file* o i *file* .class corrispondenti al *servlet* nella *directory* assegnata dal *Web server* ai *servlet*.

```
import javax.servlet.*;
import javax.servlet.http.*;
public class Finder extends HttpServlet{
  public void init(ServletConfig conf) throws ServletException {
  }
  public void doGet (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    ServletOutputStream out;
    res.setContentType("text/html");
    out = res.getOutputStream();
    out.print("<HTML>");
    out.print("<HEAD><TITLE> ");
    out.print("Pagina di risposta dal servlet");
    out.print("</TITLE> </HEAD>");
    out.print("<BODY>");
    out.print("<CENTER> Hello World </CENTER>");
    out.print("</BODY></HTML>");
  }
  public void doPost (HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    StringBuffer sb=new StringBuffer();
    ServletOutputStream out;
    res.setContentType("text/html");
    out = res.getOutputStream();
    out.print("Servlet v1.16 <BR>"+Message);
  }
public void service(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException { }
}
```

Il linguaggio PHP

Cosimo Laneve

15.3.1 (Cenni su CGI, Servlet, ASP e altre principali tecniche di programmazione sul lato server)

Introduzione

Il nome **PHP**, acronimo per *Professional Home Pages*, già la dice lunga sulla sua vocazione per la Rete: lo scopo del linguaggio è quello di consentire agli sviluppatori *Web* di realizzare in modo veloce pagine dinamiche. La definizione ufficiale di **PHP**, per la quale condigliamo di visitare www.php.net, chiarisce le caratteristiche peculiari di questo linguaggio, e precisamente:

- il **PHP** è un linguaggio di *scripting*
- è un linguaggio **HTML**-embedded
- opera *server-side*, cioè lato *server*

Vediamo, uno per volta, il significato di questi punti. Il **PHP** **è un linguaggio di *scripting***. I programmi scritti in linguaggio **PHP**, denominati brevemente *script*, vengono eseguiti tramite un apposito *software*, l'interprete **PHP**. Quest'ultimo si occupa di leggere il codice **PHP** e, interpretandone le istruzioni, esegue le operazioni corrispondenti (ad esempio la lettura di un *file* o un calcolo aritmetico). Dunque il **PHP** è quello che tecnicamente si definisce un linguaggio interpretato ed in questo esso si differenzia da altri linguaggi di programmazione, come ad esempio *C++* e *Java*, il cui codice sorgente, per poter essere eseguito, deve prima essere compilato (tradotto cioè in codice macchina).

E' **HTML-embedded**. Questa caratteristica si riferisce al fatto che il codice **PHP** è immerso nell'**HTML**; gli *script* sono inseriti, in altre parole, nelle pagine **HTML** in cui devono produrre i loro effetti. Il *Web server* riconosce le pagine **PHP**, distinguendole da quelle statiche, sulla base dell'estensione, che non sarà la solita .htm o .html ma piuttosto .php3, .phtml o simile; quando il *server* riconosce una estensione associata a **PHP** passa il testimone all'interprete, lasciando che sia quest'ultimo ad occuparsene (come descritto al punto precedente).

Opera server-side. Ovvero: il **PHP** opera lato *server*. Di conseguenza, chi accede ad una pagina **PHP** non ha la possibilità di leggere le istruzioni in essa contenute: essendo state processate ciò che il *client* vedrà sarà il risultato dell'elaborazione; per riassumere in uno slogan, insomma, il *client* vedrà cosa fa lo *script* ma non come lo fa.

Il Primo programma

Il primo esempio che vedremo sarà il classico messaggio di saluto Ciao mondo e ci servirà per mostrare la sintassi da utilizzare per includere codice **PHP** in una pagina *Web*. Ecco il nostro sorgente:

```
<html>
<head><title>Esempio 1</title></head>
<body>

<?php
echo "<h1>Ciao mondo!</h1>";
?>
```

```
</body>
</html>
```

Come si vede, si tratta di una normale pagina **HTML** in cui compaiono speciali marcatori che denotano l'inizio e la fine di un blocco di istruzioni **PHP**. Nell'esempio mostrato viene utilizzata la sintassi classica: l'inizio del codice viene contrassegnato con <?php mentre con ?> se ne indica la fine. Il risultato che otterremo, e cioè la pagina che verrà inviata al *browser*, sarà il seguente.

```
<html>
<head><title>Esempio 1</title></head>
<body>

<h1>Ciao mondo!</h1>

</body>
</html>
```

Si nota immediatamente che non vi è nessuna traccia del codice originario! In altri termini, il *client* non ha alcun modo per risalire alle istruzioni **PHP** che hanno generato la pagina richiesta.

Tornando alla sintassi per l'immersione di codice nell'**HTML**, ne esistono altre varianti; lo stesso blocco di istruzioni del primo esempio può essere scritto, in modo del tutto equivalente, così (sintassi abbreviata):

```
<?
echo "<h1>Ciao mondo!</h1>";
?>
```

così (sintassi in stile *Microsoft* **ASP**):

```
<%
echo "<h1>Ciao mondo!</h1>";
%>
```

o, ancora, così:

```
<script language="php">
echo "<h1>Ciao mondo!</h1>";
</script>
```

Quest'ultima forma può essere particolarmente conveniente se si utilizzano degli *editor* **HTML** visuali (come ad esempio *Front Page*) che potrebbero non tollerare gli altri tipi di sintassi.

Funzioni di base

La prima funzione di cui ci occupiamo è **phpinfo()**. Quello che fa è generare dinamicamente una (lunga) pagina contenente moltissime informazioni sulla versione di **PHP** installata e sull'ambiente di esecuzione. Per richiamare **phpinfo()** é sufficiente uno *script* come il seguente:

```
<html>
<head><title>phpinfo()</title></head>
<body>
<?php
phpinfo();
?>
</body>
</html>
```

La pagina *Web* generata da phpinfo() consente di visualizzare la configurazione della nostra installazione di **PHP**, di conoscere quali estensioni sono disponibili e, cosa particolarmente importante per un neofita, di imparare i nomi delle variabili predefinite che **PHP** mette a disposizione del programmatore. La seconda funzione di cui facciamo la conoscenza è certamente la più utilizzata in ogni *script* **PHP**: echo(). La funzione echo() serve per scrivere (o stampare) l'*output* che viene inviato al *browser* del visitatore che accede al nostro *script*. Si consideri il seguente esempio:

```
<html>
<head><title>echo</title></head>
<body>
<?php
echo "<h1>Benvenuto!</h1>";
?>
</body>
</html>
```

La pagina che verrà inviata al *client*, dopo l'elaborazione da parte dell'interprete **PHP**, sarà la seguente:

```
<html>
<head><title>echo</title></head>
<body>
<h1>Benvenuto!</h1>
</body>
</html>
```

Grazie ad **echo()** possiamo visualizzare il contenuto di variabili; nell'esempio seguente viene mostrato, nel messaggio di benvenuto, anche il nome di dominio del sito su cui lo *script* viene eseguito (nome contenuto nella variabile $HTTP_HOST).

```
<html>
<head><title>echo</title></head>
<body>
<?php
echo "<h1>Benvenuto su $HTTP_HOST!</h1>";
?>
</body>
</html>
```

Se, ad esempio, lo *script* viene eseguito sul sito www.latoserver.it, la pagina risultante sarà...

```
<html>
```

```
<head><title>echo</title></head>
<body>
<h1>Benvenuto su www.latoserver.it!</h1>
</body>
</html>
```

A rigore occorre rilevare che *echo* non è propriamente una funzione bensì un costrutto del linguaggio; per questo motivo non è necessario, come si è visto negli esempi, utilizzare le parentesi tonde per racchiudere gli argomenti.

exit() e die()

entrambe producono il risultato di arrestare l'esecuzione dello *script*, con la differenza che die() consente anche di stampare un messaggio. Ad esempio il seguente *script*...

```
<html>
<head><title>exit</title></head>
<body>
<? exit(); ?>
<p>Questa frase non si vedrà</p>
</body>
</html>
```

produce questo *output*:

```
<html>
<head><title>exit</title></head>
<body>
```

Le funzioni exit() e die() possono essere utilizzate, quindi, per gestire eventuali situazioni di errore che non consentono di proseguire l'esecuzione del nostro *script* **PHP** (i cosiddetti errori fatali). In queste circostanze può essere conveniente usare die() per mostrare un messaggio di errore appropriato.

Vediamo un semplice esempio. Nello *script* seguente controlliamo il valore della variabile globale $n e mostriamo un messaggio di errore, bloccando l'esecuzione del programma, se questo è maggiore di 1.

```
<html>
<head><title>die</title></head>
<body>

<?
$n = 5;
if ($n > 1) die("<h1>\$n è maggiore di uno!!!</h1>");
?>

<h1>$n è minore o ugale ad uno!</h1>

</body>
</html>
```

Il risultato sarà il seguente:

```
<html>
<head><title>die</title></head>
<body>

<h1>$n è maggiore di uno!!!</h1>
```

Se, invece, sostituiamo l'istruzione $n=5 con $n=1 il risultato diventa:

```
<html>
<head><title>die</title></head>
<body>

<h1>$n è minore o ugale ad uno!</h1>

</body>
</html>
```

Variabili

Per creare una variabile è sufficiente assegnarle un valore; in altre parole non vi è alcuna necessità di dichiararla esplicitamente, come avviene, invece, in altri linguaggi. Si parla, dunque, di dichiarazione implicita. Se vogliamo visualizzare nella nostra pagina **PHP** il valore di una variabile, in modo che venga mostrata dal *browser* del visitatore, possiamo usare la funzione *echo*. Ad esempio:

```
// Questa istruzione visualizza il valore della
// variabile $a
echo $a;
```

A disposizione del programmatore c'è anche un certo numero di variabili predefinite, cioè di variabili il cui valore è già impostato. Solitamente queste variabili contengono informazioni circa l'ambiente di esecuzione dello *script* **PHP**. Esaminiamone alcune di uso frequente.

- **$PHP_SELF**. La variabile $PHP_SELF contiene il percorso dello *script* in esecuzione; ad esempio, all'interno dello *script* **PHP** raggiungibile all'indirizzo
  http://www.latoserver.it/index.php3

  il valore della variabile $**PHP**_SELF sarà /index.php3.

- **$HTTP_HOST**. La variabile $HTTP_HOST contiene il nome del *server* su cui lo *script* viene eseguito; nel caso dell'esempio precedente il valore di $HTTP_HOST sarebbe www.latoserver.it.
- **$HTTP_REMOTE_HOST e $HTTP_REMOTE_ADDR**. Le variabili $HTTP_REMOTE_HOST e $HTTP_REMOTE_ADDR, invece, forniscono rispettivamente il nome di dominio e l'indirizzo *IP* del visitatore.

In uno *script* **PHP** i nomi di variabili sono prefissati dal simbolo $ (dollaro); ad esempio, la seguente istruzione:

```
$a = $b + $c
```

assegna ad una variabile di nome a la somma dei valori di altre due variabili, rispettivamente b e c. Il simbolo = è l'operatore di assegnamento.

I Tipi di Dato

Vediamo adesso cosa può contenere una variabile. Abbiamo parlato di valori, ma a che tipo di valori facciamo riferimento? Introduciamo dunque i tipi di dato che **PHP** mette a disposizione del programmatore.

I tipi di dato supportati da **PHP** si distinguono in tipi scalari e tipi compositi: sono tipi scalari i numeri (sia interi che in virgola mobile) e le stringhe; sono tipi compositi gli *array* e gli oggetti. A partire dalla versione 4 di **PHP**, inoltre, è stato introdotto un nuovo tipo scalare: quello booleano. Una variabile booleana può assumere solo due valori, vero (costante TRUE) o falso (costante FALSE).

```
// $b è una variabile di tipo bool
$b = TRUE;

// $num è una variabile di tipo intero
$num = 25;

// $pi_greco è un numero in virgola mobile;
// si noti il punto (virgola decimale)
$pi_greco = 3.14;

// $messaggio è una stringa
$messaggio = "Ciao a tutti!";
```

Un *array* in **PHP** può corrispondere sia ad un vettore, cioè ad una struttura dati in cui ogni elemento è individuato da un indice numerico, sia ad una tabella di *hash*, cioè (in modo molto semplice) ad una collezione di coppie nome/valore. In **PHP**, infatti, tali strutture sono sostanzialmente la stessa cosa. Un *array* può essere creato esplicitamente utilizzando il costrutto array() oppure implicitamente. Vediamo alcuni esempi.

```
// Questo è un array di numeri interi
// Lo creo esplicitamente usando array()
$primi = array( 2, 3, 5, 7, 11 );

// Questo array lo creo implicitamente
$pari[0] = 2;
$pari[1] = 4;
$pari[2] = 6;

// Questa è una tabella di hash
$bookmark["puntoinf"] = "punto-informatico.it"
$bookmark["latoserver"] = "www.latoserver.it"
```

Per accedere ad un elemento di un *array* si può utilizzare sia il corrispondente indice numerico, sia la chiave (se si tratta di una tabella di *hash*). Ad esempio:

```
// Questa istruzione stampa "7"
// cioé l'elemento di indice 3 dell'array $primi
```

```
echo $primi[3];

// Questa istruzione stampa "www.latoserver.it"
echo $bookmark["latoserver"];
```

A differenza di quanto avviene in altri linguaggi di programmazione, un *array* in **PHP** può contenere elementi di tipo diverso. Ad esempio, è perfettamente legittimo costruire un *array* che contenga sia numeri interi che stringhe.

```
// Questo è un array valido!
// Contiene: un numero intero, una stringa,
// un numero in virgola mobile ed un altro array!
$mix = array( 1, "ciao", 3.14, array( 1, 2, 3 ) );
```

Il tipo di dato *object* verrà trattato in una lezione successiva dedicata alla programmazione orientata agli oggetti con **PHP**

I metodi GET e POST

Come è stato discusso all'inizio del modulo 15, una *form* **HTML** utilizza i metodi *GET* o *POST* per trasferire informazioni a/dal *server*. In alternativa, si può accedere alle informazioni inviate con i metodi *GET* e *POST* utilizzando gli *array* associativi **$HTTP_GET_VARS** e **$HTTP_POST_VARS**, rispettivamente. Ad esempio, se tramite il metodo *GET* inviamo ad uno *script* **PHP** un parametro di nome pagina e di valore pari a 1, all'interno dello *script* stesso potremo accedere a tale informazione sia usando la variabile globale $p, sia accedendo a $HTTP_GET_VARS["p"], cioè all'elemento dell'array associativo $HTTP_GET_VARS individuato dalla chiave p.

Supponiamo di scrivere nella casella di testo la parola Schumacher e di premere il pulsante Invia i dati. Il *browser* si sposterà all'indirizzo

http://www.pippo.it/scripts/elabora.php?campione=Schumacher

Cosa è successo? Poiché la *form* dell'esempio usa il metodo *GET* per trasferire i propri dati, questi vengono codificati nell'indirizzo dello *script* a cui devono essere inviati. Nello *script* elabora.php adesso troveremo definita una variabile globale di nome $campione il cui valore è la stringa Schumacher.

```
// Nel file `elabora.php' ...
// Questo stampa "Schumacher"
echo $campione;
```

A questo punto è utile accennare al modo in cui le informazioni provenienti da una *form* **HTML** vengono codificate per essere trasmesse con il metodo *GET*. Partiamo dall'indirizzo URL dello *script* a cui vogliamo inviare tali dati; ad esso aggiungiamo (a destra) un punto interrogativo che separerà la URL vera e propria (a sinistra) dalla query string che andiamo a costruire.

La struttura della *query string* consiste di una serie di coppie nome/valore separate da una e commerciale (&); i caratteri non ammissibili in un indirizzo URL (ad esempio gli spazi) vengono sostituiti dal simbolo di percentuale seguito dal corrispondente codice *ASCII* (in esadecimale).

Adesso che sappiamo come funziona il metodo *GET* possiamo sfruttarlo per passare parametri ad uno *script* **PHP**. Supponiamo di avere uno *script* di nome news.php che estrae notizie ed articoli da un *database*; ad esso vogliamo passare un parametro, chiamiamolo $argomento, che determinerà il tipo di notizie che ci verranno mostrate. Ad esempio, per ottenere notizie sportive, invocheremo:

http://www.pippo.it/news.php?argomento=Sport

In questo caso la codifica manuale della URL era semplice, ma cosa fare se ci interessano le notizie di Attualità e Cultura? Niente paura, esiste una funzione **PHP**, urlencode(), che ci permette di codificare una stringa in una forma ammissibile per una URL. Il frammento di *script* per creare il *link* appropriato sarà:

```
<?
echo '<a href="http://www.pippo.it/news.php?argomento=';
echo urlencode("Attualità e Cultura");
echo '">Clicca qui</a>';
?>
```

I Cookies

Vediamo adesso come si manipolano i *cookies*, introdotti nel modulo 15, con il linguaggio **PHP**. Tutte le operazioni di scrittura, modifica o cancellazione di *cookies* in **PHP** avvengono mediante una stessa funzione, setcookie(). Tale funzione deve obbligatoriamente essere invocata prima che qualsiasi contenuto venga inviato al *browser*; i *cookies*, infatti, vengono trasmessi tra *client* e *server* sotto forma di intestazioni (*headers*) HTTP.

La funzione prevede solo due argomenti obbligatori: il nome da assegnare al *cookie* ed il suo valore. Ad esempio, se vogliamo memorizzare nel *browser* di un visitatore un *cookie*, che chiameremo $nomeutente, contenente la stringa "latoserver", l'istruzione da utilizzare sarà la seguente

```
// Imposto un cookie: $nomeutente = "pippo.it";
setcookie( "nomeutente", "pippo.it" );
```

E' possibile specificare anche altri argomenti (facoltativi); nell'ordine abbiamo: scadenza del *cookie*, percorso, dominio e *secure*. Per una discussione dettagliata si rinvia alla sitografia in fondo alla pagina.

Quando un *cookie* è stato impostato, il suo valore può essere modificato richiamando nuovamente la funzione setcookie() ed associando allo stesso nome il nuovo valore. La cancellazione di un *cookie*, infine, può avvenire in due modi: assegnandogli un valore nullo o impostando la scadenza ad una data passata.

Resta da vedere in quale modo si possano leggere da uno *script* **PHP** le informazioni memorizzate nei *cookies*; in questo compito siamo molto facilitati, in quanto l'interprete **PHP** analizza automaticamente i *cookies* inviati dal *browser* e li rende disponibili in altrettante variabili globali e nell'*array* associativo $HTTP_COOKIE_VARS.

Passiamo ad analizzare un esempio pratico di utilizzo dei *cookies*, realizzando una semplice pagina **PHP** che ricorda la *data* e l'ora dell'ultimo accesso del visitatore. A tale

scopo impostiamo sul *browser* un *cookie* $ultimavisita, la cui scadenza determina quanto a lungo viene conservata tale memoria. Nel nostro esempio non è stata specificata alcuna scadenza, in modo da far scomparire il *cookie* alla chiusura del *browser*. Il valore assegnato di volta in volta al *cookie* $ultimavisita è il risultato della funzione time() e consiste nel numero di secondi trascorsi dalla cosiddetta Unix epoch (primo gennaio 1970).

```
<?php
// file `saluto.php'
// Il saluto predefinito
$saluto = "Benvenuto!";

// Controllo se esiste il cookie...
if (isset($HTTP_COOKIE_VARS["ultimavisita"])) {
// Cambio il saluto con uno piu' appropriato
$saluto = "Bentornato!";
}

// Imposto il cookie relativo a questa visita
setcookie( "ultimavisita", time() );
?>
<html>
<head>
<title><? echo $saluto ?></title>
</head>
<body>
<h1><? echo $saluto ?></h1>

<?php if (isset($HTTP_COOKIE_VARS["ultimavisita"])) {
// Stampo la data dell'ultima visita
echo "L'ultima volta sei stato qui il " . date( "d/m/Y");
echo " alle ore " . date( "H:i:s.", $ultimavisita );
// Link per cancellare il cookie
echo "<p><a href=\"cancella.php\">Cancella il cookie</a>";
} else {
echo "Non sei mai stato qui prima?";
}
?>
</body>
</html>
```

Lo *script* cancella.php, richiamabile cliccando sul *link* Cancella il cookie, non fa altro che cancellare il *cookie* $ultimavisita, assegnandogli un valore nullo, e dirottare il *browser* di nuovo alla pagina precedente.

```
<?
// file `cancella.php'
setcookie( "ultimavisita", "" );
header( "Location: saluto.php" );
?>
```

L'esempio proposto, sebbene di improbabile utilità, dovrebbe aver chiarito le modalità d'impiego dei cookies; pur nella sua semplicità, inoltre, può essere il punto di partenza

per lo sviluppo di funzionalità personalizzate, lasciate alla vostra fantasia.

L'accesso ai database

La coppia **PHP** e *MySQL* è sicuramente una delle più diffuse nella realizzazione di applicazioni *Web* basate su *software OpenSource*. *MySQL* è un *DBMS*, *Data Base Management System*, cioè un *software* per la gestione di basi di dati; la sua popolarità è indiscussa, grazie alle prestazioni di tutto rispetto e nonostante la mancanza di diverse caratteristiche avanzate (transazioni, *stored procedures*, eccetera).

Nel seguito vedremo in che modo è possibile, da uno *script* **PHP**, collegarsi ad un *database MySQL* ed eseguire operazioni su di esso tramite il linguaggio *SQL*. Per una completa comprensione della lezione è necessaria una conoscenza di base di *SQL*; assumeremo, inoltre, che *MySQL* sia correttamente installato.

L'accesso ad un *database MySQL* avviene mediante autenticazione; questo vuol dire che, prima di poter effettuare qualsiasi operazione su di esso, dobbiamo disporre dei privilegi necessari. In particolare, le informazioni di cui abbiamo bisogno sono: il nome dell'*host* su cui è in esecuzione il *server MySQL*, il nome del nostro *database*, il nome utente che ci è stato assegnato e la relativa *password* (per averle occorre rivolgersi all'amministratore di sistema). Supponiamo che i parametri nel nostro caso siano i seguenti:

```
// Il nome dell'host (hostname) su cui si trova MySQL
$dbhost = "localhost";

// Il nome del nostro database
$dbname = "dbprova";

// Il nostro nome utente (username)
$dbuser = "luca";

// La nostra password
$dbpass = "secret";
```

E' opportuno salvare tali parametri in apposite variabili piuttosto che utilizzarli direttamente nelle chiamate di funzione; in tal modo, infatti, potremo inserirli in un *file* separato che includeremo in tutti gli *script* che accedono al *database*.

La prima funzione che utilizziamo è *mysql_connect()*, che ci servirà per instaurare la connessione con il *server MySQL*. I parametri da fornire a tale funzione sono il nome dell'*host*, il nome utente e la *password*. Vediamo un esempio:

```
// Funzione mysql_connect()
$conn = mysql_connect($dbhost,$dbuser,$dbpass)
or die("Impossibile collegarsi al server MySQL.");
```

Si osservi la sintassi utilizzata nell'esempio precedente; il significato è il seguente: se la funzione restituisce un valore nullo, situazione che denota l'impossibilità a collegarsi al *server MySQL*, viene invocata die() per arrestare l'esecuzione e visualizzare un messaggio di errore. Inoltre, in una variabile che abbiamo chiamato $conn salviamo il valore restituito da mysql_connect(), che servirà da identificativo della connessione stabilita.

Il passo successivo è la selezione del *database* su cui operare; la funzione da utilizzare è mysql_select_db(), alla quale occorre fornire il nome del *database* e, opzionalmente, l'identificativo della connessione (se non viene indicata verrà utilizzata l'ultima aperta).

```
// Funzione mysql_select_db()
mysql_select_db($dbname,$conn)
or die("Impossibile selezionare il database $dbname");
```

Opinioni e commenti

Scrivi nuovo

Anche questa volta in caso di insuccesso viene arrestata l'esecuzione del programma **PHP** e viene visualizzato un messaggio di errore appropriato.

Arriviamo così alla parte più importante, quella dell'interazione con la base di dati, interazione che avverrà mediante il linguaggio *SQL*. Le funzioni **PHP** che utilizzeremo in questa fase sono mysql_query(), per l'esecuzione di comandi *SQL*, e mysql_fetch_row(), per prelevare il risultato restituito da *MySQL* quando il comando eseguito è un'istruzione *SELECT* (quindi un'interrogazione, o *query*).

Supponiamo di voler creare una tabella del *database* con cui gestire una rudimentale rubrica telefonica. Il comando *SQL* da utilizzare sarà del tipo

```
CREATE TABLE rubrica(
Progressivo int PRIMARY KEY AUTO INCREMENT,
Nome varchar(40),
Cognome varchar(40),
Telefono varchar(20))
```

Per eseguire tale comando dal nostro *script* **PHP**, lo inseriamo dapprima in una stringa, ad esempio nel modo seguente

```
$sql = "CREATE TABLE rubrica( "
. "Progressivo int PRIMARY KEY AUTO INCREMENT, "
. " Nome varchar(40), Cognome varchar(40), Telefono varchar(20))";
```

Passiamo poi all'esecuzione vera e propria, invocando la funzione mysql_query().

```
// Esegue il comando SQL o stampa un messaggio di errore
$res = mysql_query($sql,$conn)
or die( "Errore: " . mysql_error() );
```

Voilà, il gioco è fatto! In caso di errori, comunque, l'istruzione mysql_error() ci fornisce la descrizione del problema che si è verificato. Esaminiamo adesso il caso di una interrogazione del *database*. L'esecuzione del relativo comando *SQL* (sarà, naturalmente, una *SELECT*) è del tutto analoga al caso già visto. L'unica differenza risiede nel fatto che, in questo caso, abbiamo un risultato da prelevare. Una delle funzioni che possiamo utilizzare a tale scopo è mysql_fetch_row().

```
// Interroghiamo la nostra rubrica

// Comando SQL da eseguire
$sql = "SELECT Telefono FROM rubrica "
```

```
. "WHERE Nome='Luca' AND Cognome='Balzerani'";

// Esecuzione comando SQL o messaggio di errore
$res = mysql_query($sql,$conn)
or die( "Errore: " . mysql_error() );

// Estrazione del risultato
$info = mysql_fetch_row($res);
echo "Il mio numero di telefono è " . $info[0];
```

Al termine della sessione di lavoro si può invocare la funzione mysql_close() per chiudere la connessione con il *server MySQL*. Si tratta, in ogni caso, di un passo opzionale in quanto tutte le connessioni lasciate aperte verranno chiuse automaticamente alla fine dello *script*.

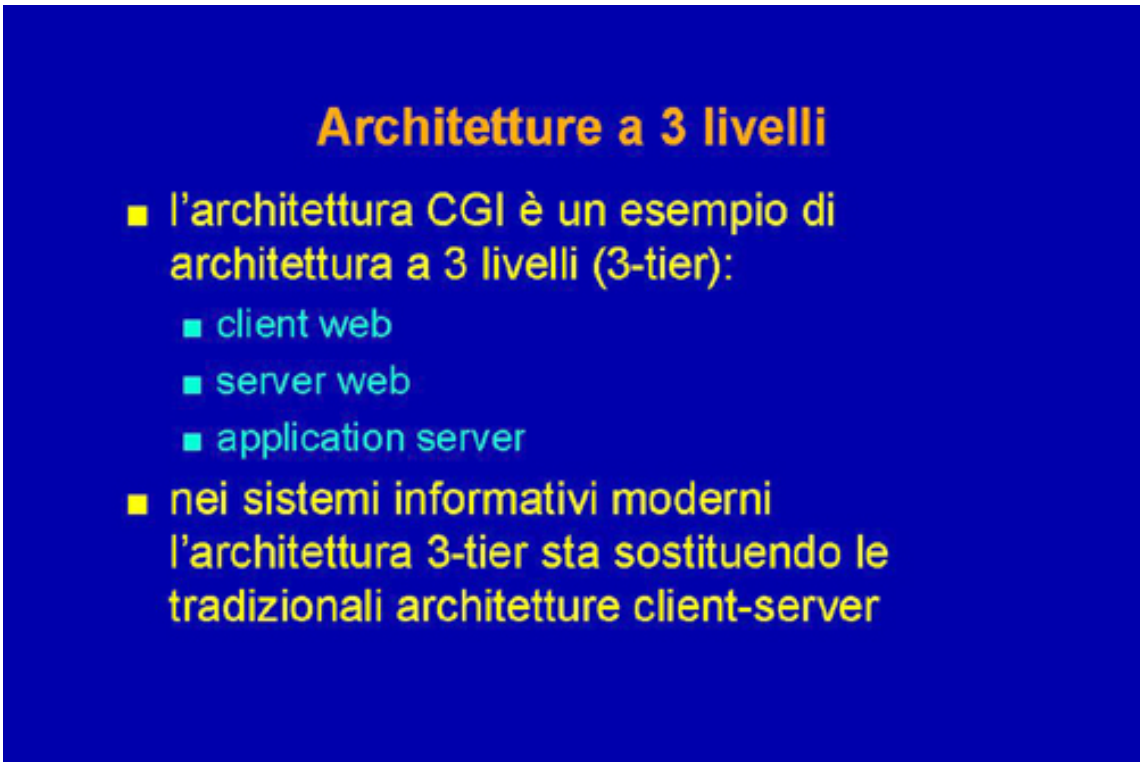
```
// Funzione mysql_close()
mysql_close($conn);
```

Architetture Web a tre livelli: CGI, SSI, ISAPI e codice mobile

Cosimo Laneve

15.3.1 (Cenni su CGI, Servlet, ASP e altre principali tecniche di programmazione sul lato server.)

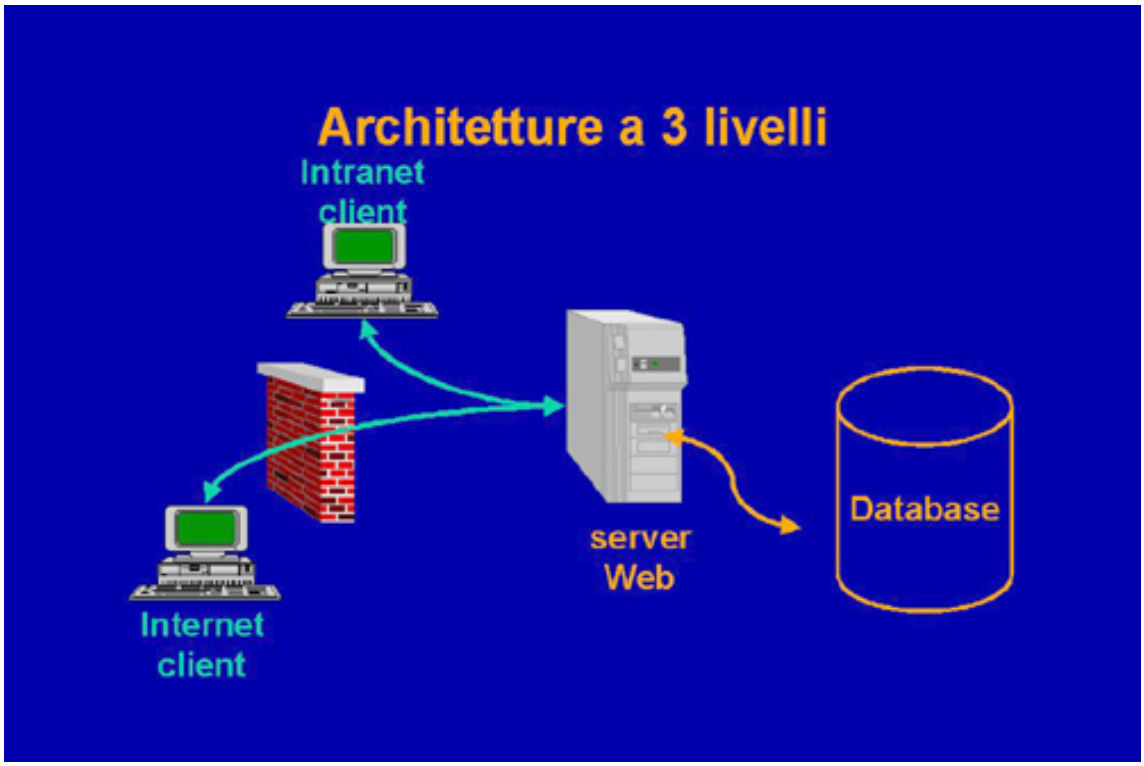
Architetture a 3 livelli (1)



Architetture a 3 livelli (1)

Nel corso della lezione precedente abbiamo analizzato le caratteristiche dell'architettura *CGI*. L'architettura *CGI* è un esempio di architettura *Web* a tre livelli, ma non è l'unico. Nel corso di questa lezione andremo a vedere quali sono le altre architetture a tre livelli e vedremo quali sono le caratteristiche, i vantaggi e gli svantaggi rispetto all'architettura *CGI*. Si parla di architetture a tre livelli quando abbiamo a che fare con architetture distribuite dove viene utilizzato un *client Web*, un *server Web* ed un *application server*. Questo tipo di architettura sta sostituendo, in quelli che sono i sistemi informativi moderni, la tradizionale architettura *client-server*, dove l'accesso ad un *application server* era fatto attraverso un *client* dedicato, costruito e programmato appositamente per accedere a quel *application server*.

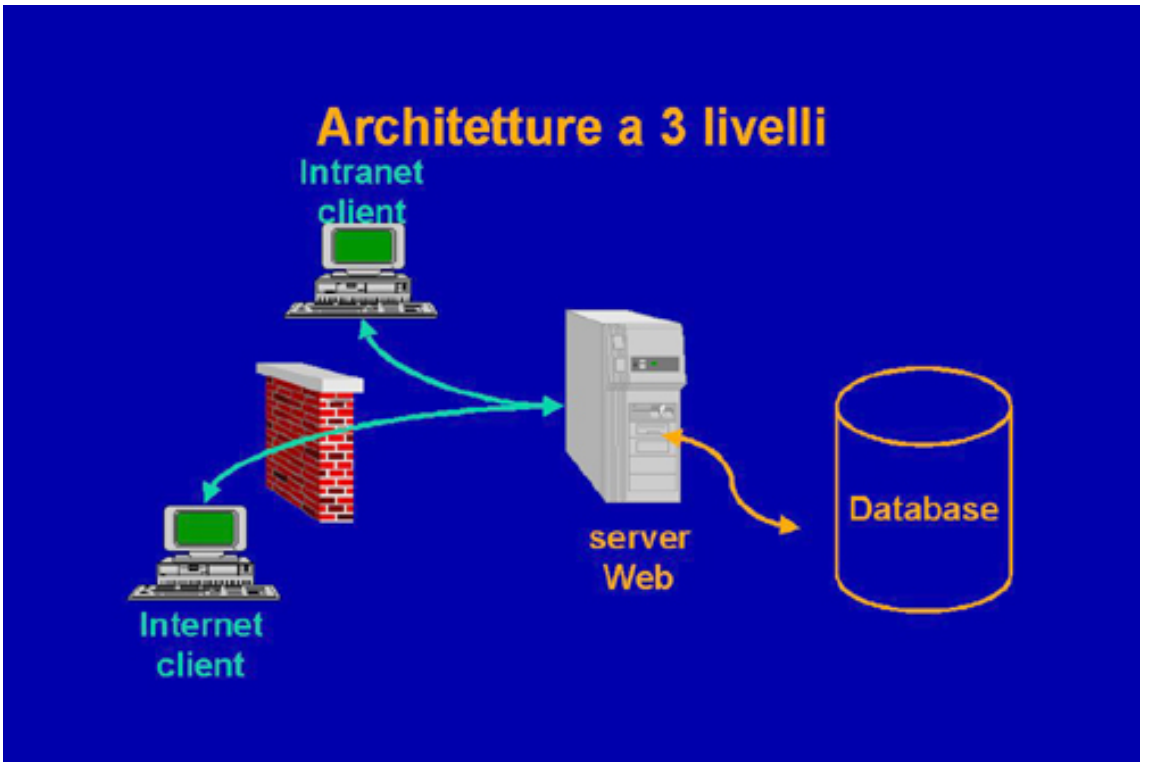
Architetture a 3 livelli (2)



Architetture a 3 livelli (2)

Un architettura di questo tipo, un architettura di tipo *client-server*, richiede quindi la costruzione di uno specifico *client* per ciascuna specifica applicazione. Questo può essere modificato andando ad utilizzare una architettura a tre livelli nella quale, appunto, viene utilizzato, non più un *client* specifico per accedere all'*application server*, ma un tradizionale *client Web*. L'architettura generale è quella indicata in questa figura. Come vedete il *client* è un normale *client Web* che accede al *data base* applicativo non direttamente, ma attraverso un *server Web* intermedio. Questo fa sì che possano essere utilizzati *client Web* standard e che non debba essere sviluppato un *client* specifico per ogni applicazione che intende accedere a questo *data base*. Questa architettura, quindi, prevede tre livelli: il primo è il *client Web*, il secondo livello è il *server Web* e il terzo livello è il *data base* applicativo.

Architetture a 3 livelli (3)



Architetture a 3 livelli (3)

Uno dei vantaggi di questa architettura è che è un'architettura di tipo aperto: ovvero non ci si lega in particolare ad un tipo di *client*, ad un tipo di *server* e ad un tipo di *data base* applicativo, ma abbiamo la flessibilità necessaria per andare a sostituire un *client Web* con un diverso *client Web*, o per andare a modificare il *server Web* che stiamo utilizzando. Un altro aspetto fondamentale di questo tipo di architettura è la sua scalabilità: la sua apertura verso ambienti *Internet*. Se sviluppiamo, infatti, un sistema informativo all'interno di una *intranet*, il cui limite è rappresentato da questo muro che divide la *intranet* dalla rete *Internet* esterna, possiamo facilmente estendere i servizi che distribuiamo all'interno della nostra *intranet*, semplicemente evolvendo quelli che sono i servizi distribuiti dal nostro *server Web*. Per arrivare su *Internet* non avremo bisogno di nulla in più di quanto non utilizziamo per arrivare ai nostri utenti *intranet*, quindi, anche chi utilizzerà da *Internet* il proprio navigatore *Web*, potrà accedere a quelli che sono i servizi del nostro sistema informativo. È fondamentale in questo caso avere un meccanismo di sicurezza che consenta di distinguere, da un punto di vista logico, quello che è il traffico *Internet* da quello che è il traffico *intranet* ed era rappresentato, nella *slide* precedente, da quel muro che rappresenta un sistema *firewall* che protegge la rete *intranet* dalle richieste provenienti dalla rete *Internet*.

Architetture a 3 livelli (4)

Architetture a 3 livelli (4)

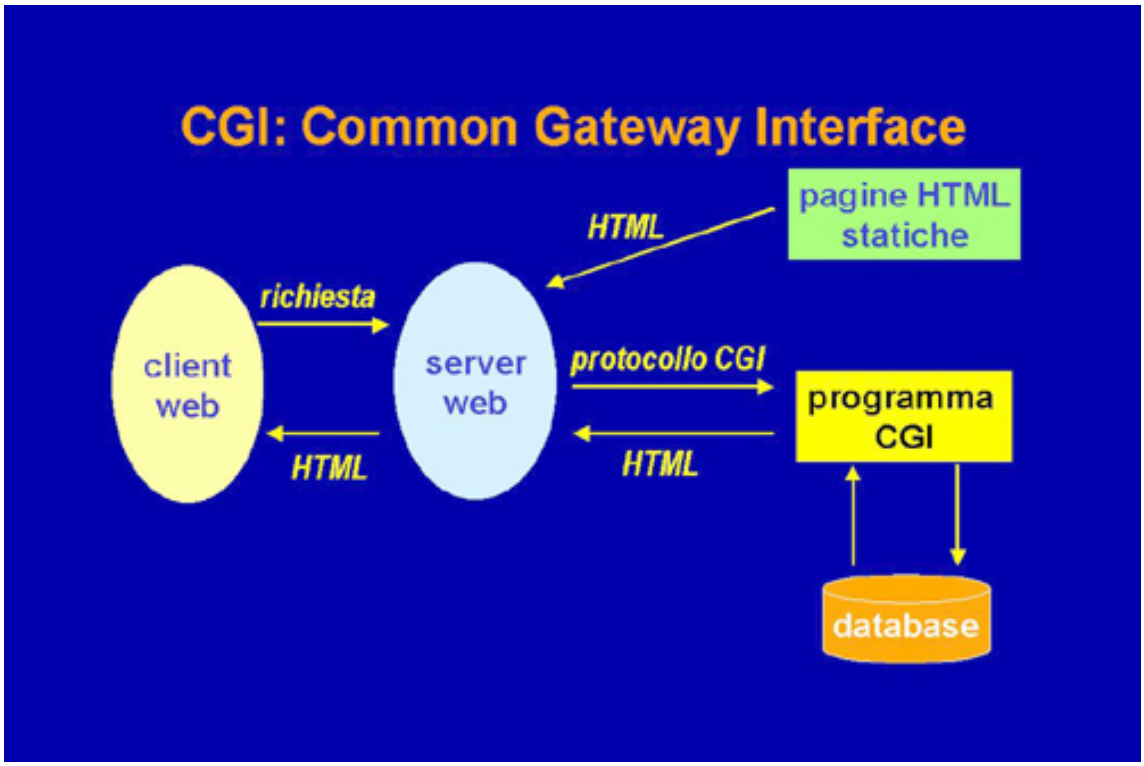
Quali sono i vantaggi principali di una architettura a tre livelli? Li vediamo indicati in questa *slide*. Abbiamo, innanzitutto, indipendenza dal *client*, nel senso che il nostro utente potrà utilizzare il *client Web* di sua preferenza, quindi il suo *Netscape* o il suo *Internet Explorer*, a seconda dei suoi gusti e delle sue preferenze personali. Abbiamo indipendenza dal *server* applicativo, in quanto andiamo ad utilizzare un'interfaccia *Web* che passa attraverso un *server Web*. E abbiamo un servizio che è scalabile, in quanto possiamo far crescere il numero di utenti che accedono al nostro servizio, semplicemente, scalando in maniera orizzontale, o verticale, quelle che sono le caratteristiche del *server Web* che dà accesso al nostro servizio. Infine, come abbiamo detto poco fa, possiamo espandere facilmente i servizi di *intranet* verso utenti *Internet*.

CGI: Limitazioni

CGI: Limitazioni

Abbiamo visto, quindi, che una tipica architettura a tre livelli è la cosiddetta architettura *CGI*. Questa è stata la prima architettura sviluppatasi per distribuire applicazioni a tre livelli *Web-oriented*, ma non è l'unica. In particolare, l'architettura *CGI* ha una serie di applicazioni in quanto la sua semplicità consente un rapido sviluppo delle applicazioni, ma ci sono due grosse limitazioni: la prima è che il processo di attivazione di un processo *CGI* è molto inefficiente e introduce una certa latenza. Infatti, per ogni richiesta *CGI* il *server Web* dovrà lanciare un nuovo processo, separato da quello nel quale sta eseguendo il *server Web* stesso. Questo è un procedimento che consuma molte risorse e che richiede tempo. L'efficienza, quindi, in condizioni di carico molto elevate, di un programma *CGI*, non potrà andare oltre una certa soglia. L'altro problema legato all'uso di architetture *CGI* è dovuto al tipo di linguaggio specifico che si utilizza. Molto spesso per sviluppare applicazioni *CGI* si utilizzano linguaggi interpretati, questo fa sì che le prestazioni non siano eccezionali. Soprattutto nel caso di applicazioni *computation intensive*, le prestazioni di un'architettura di tipo *CGI* possono non raggiungere quelli che sono i requisiti del servizio che intendo offrire. Ovviamente, posso decidere di utilizzare un linguaggio non interpretato, un linguaggio compilato, per sviluppare le mie applicazioni *CGI*, ma rimane comunque la prima delle due limitazioni indicate, legate al procedimento con cui deve essere lanciato un processo del tutto indipendente dal *server Web*. Vedremo, quindi, che sono state sviluppate una serie di architetture, anch'esse a tre livelli, che forniscono un'alternativa all'architettura *CGI* stessa. Parleremo in particolare dell'architettura *Server Side Include*, parleremo dell'architettura *Web* API e dell'architettura a codice mobile.

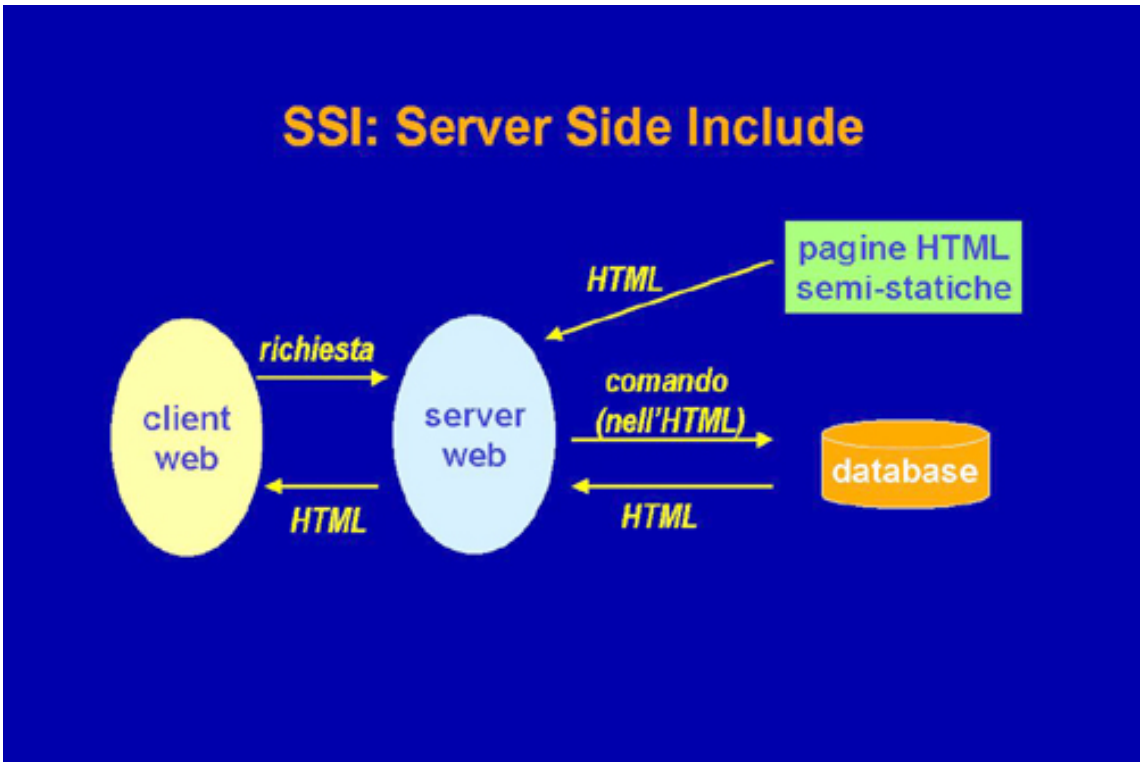
CGI: Common Gateway Interface



CGI: Common Gateway Interface

Riprendendo quelli che sono gli elementi di una applicazione *CGI*, ritroviamo in questa *slide* che un'architettura *CGI* fa uso di un *client* che invia una richiesta HTTP. Questa richiesta HTTP, rivolta al *server Web*, anziché essere diretta ad ottenere una pagina HTML statica, è diretta ad invocare un programma *CGI* che, attraverso il protocollo *CGI*, viene attivato e consente di eseguire delle elaborazioni al volo. Negli esempi che abbiamo visto nella lezione precedente abbiamo visto delle semplici elaborazioni che coinvolgevano soltanto il programma *CGI*, ma questa architettura, generalmente, può essere estesa per andare ad interrogare dal nostro programma *CGI* un sistema informativo più complesso, come ad esempio un *data base*. Il programma *CGI* deve occuparsi, quindi, di formattare l'*output* in formato HTML in modo da poterlo passare al *server Web*, che si occuperà semplicemente di girarlo al *client Web*. In questo modo riusciamo ad invocare un processo, il programma *CGI*, che sta sul *server Web* e che esegue delle elaborazioni in tempo reale a partire da un normale *client Web*.

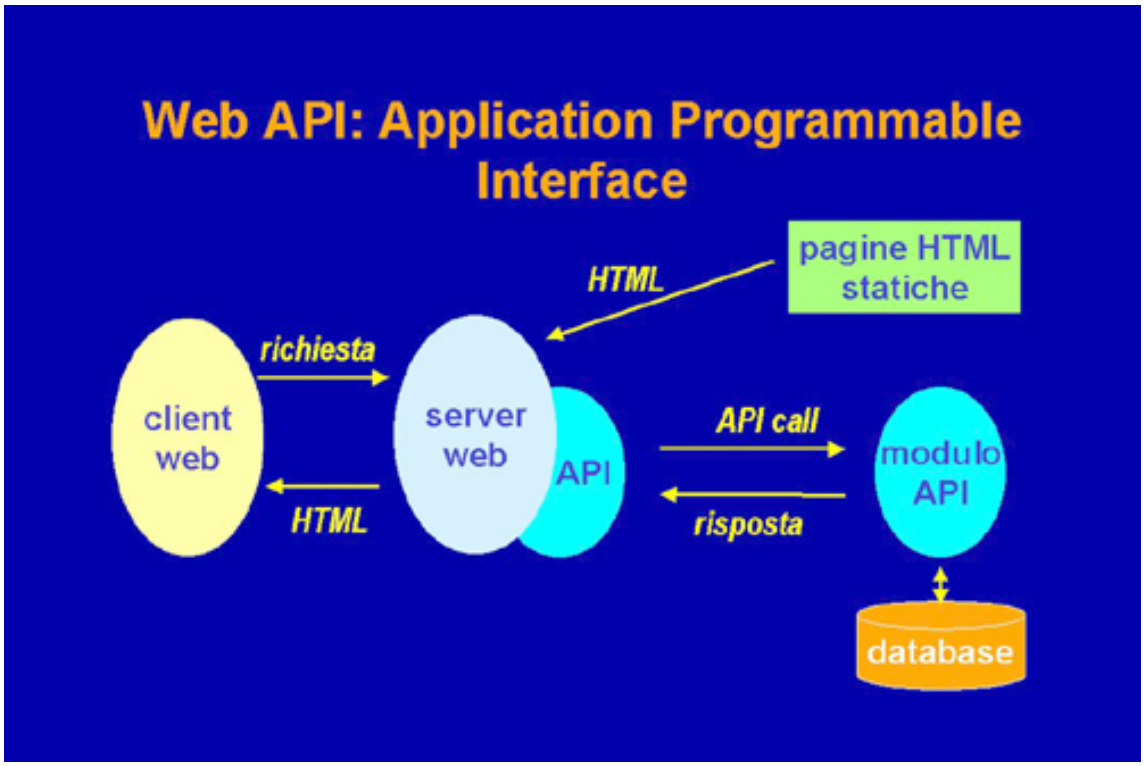
SSI: Server Side Include



SSI: Server Side Include

Questa, abbiamo detto, non è l'unica architettura che abbiamo a disposizione, ma ne esistono altre con i loro vantaggi e i loro svantaggi. Cominciamo dall'architettura cosiddetta *Server Side Include*. In questo caso si cerca di ovviare alla complessità, nel caso *CGI*, di avere separate le pagine dinamiche e le pagine statiche. In questo caso, in particolare, le pagine HTML contengono delle direttive, dei comandi, che vengono eseguiti dal *server* prima di distribuire la pagina richiesta. Per questo motivo questa architettura viene detta *Server Side Include*, perché le pagine HTML sul lato *server* contengono delle direttive che vengono eseguite sul lato *server* prima che la pagina venga distribuita. Il procedimento, in questo caso, prevede ancora che il *client Web* invii la sua richiesta HTTP, questa richiesta va ad invocare una pagina HTML semi-statica la quale, prima di esser distribuita dal *server Web*, verrà eseguita dal *server Web* stesso e gli eventuali comandi contenuti nell'HTML consentiranno, ad esempio come indicato in questa *slide*, di accedere al nostro sistema informativo. L'*output* di questi comandi, che sarà ancora in formato HTML, verrà sostituito nell'esatto punto in cui quella direttiva semi-statica era contenuta. Il *server*, a questo punto, avrà confezionato una pagina completamente HTML da spedire al *client*. Notate che per il *client* non c'è nessuna differenza rispetto a quanto avveniva nel meccanismo *CGI*, in quanto ad una richiesta HTTP fa eco una risposta HTTP che contiene una pagina HTML. Il *client* non percepisce in alcun modo il fatto che quella pagina HTML sia stata creata al volo con un meccanismo di *Server Side Include* piuttosto che con un meccanismo di tipo *CGI*.

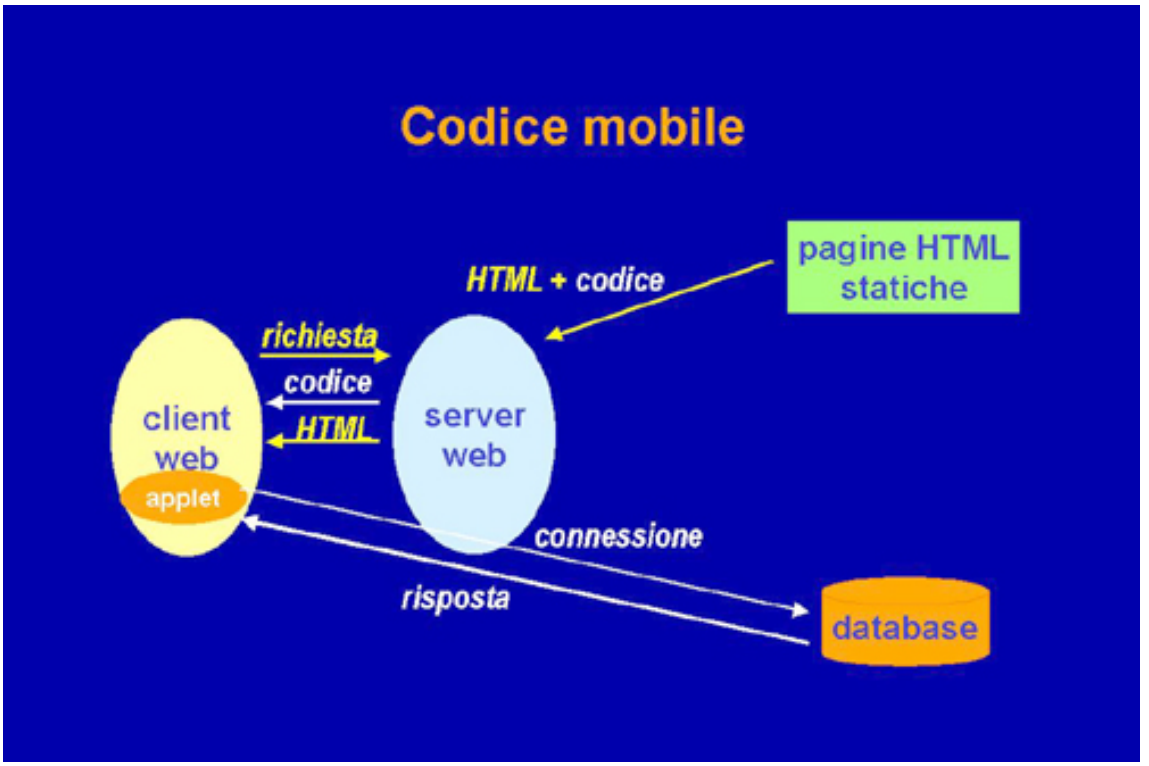
Web API: Application Programmable Interface



Web API: Application Programmable Interface

Un terzo tipo di approccio è quello che va sotto il nome di *Web* API (*Application Programmable Interface*). Che cosa significa? Significa che, come mostrato in questa *slide*, i *server Web* vengono forniti con un'interfaccia con la quale possono essere estesi. Ovvero possono essere definiti dei moduli, i cosiddetti moduli API, che estendono quelle che sono le funzionalità di base del *server Web*. Se vogliamo sviluppare una particolare applicazione che andrà ad elaborare delle richieste eseguite sul nostro *server Web*, possiamo andare a costruire un modulo API che risponde alle chiamate previste dall'interfaccia API di quel *server*. Questo modulo dovrà essere, ovviamente, una libreria dinamica collegata al *server Web*. Questo ne consentirà delle caratteristiche molto interessanti in termini di prestazioni e di *performance*, in quanto il modulo API diventa un estensione strettamente collegata del *server Web* stesso. In particolare vedremo, nella prossima lezione, un esempio di questo tipo di architettura e in particolare vedremo l'esempio *Microsoft* delle cosiddette ISAPI (*Information Server API*) che sono il meccanismo API previsto dalla architettura *Microsoft*. Le caratteristiche di questo tipo di architettura sono innanzi tutto le elevate prestazioni. Si tratta in effetti di un estensione del *server Web* stesso, strettamente collegata al *server*, che non richiede, come nel caso del programma *CGI*, un processo separato da attivare ad ogni invocazione. Le *performance* di una soluzione di questo tipo, quindi, saranno molto più elevate delle *performance* offerte da una soluzione basata su architetture *CGI* o su architetture di tipo *Server Side Include*.

Codice mobile



Codice mobile

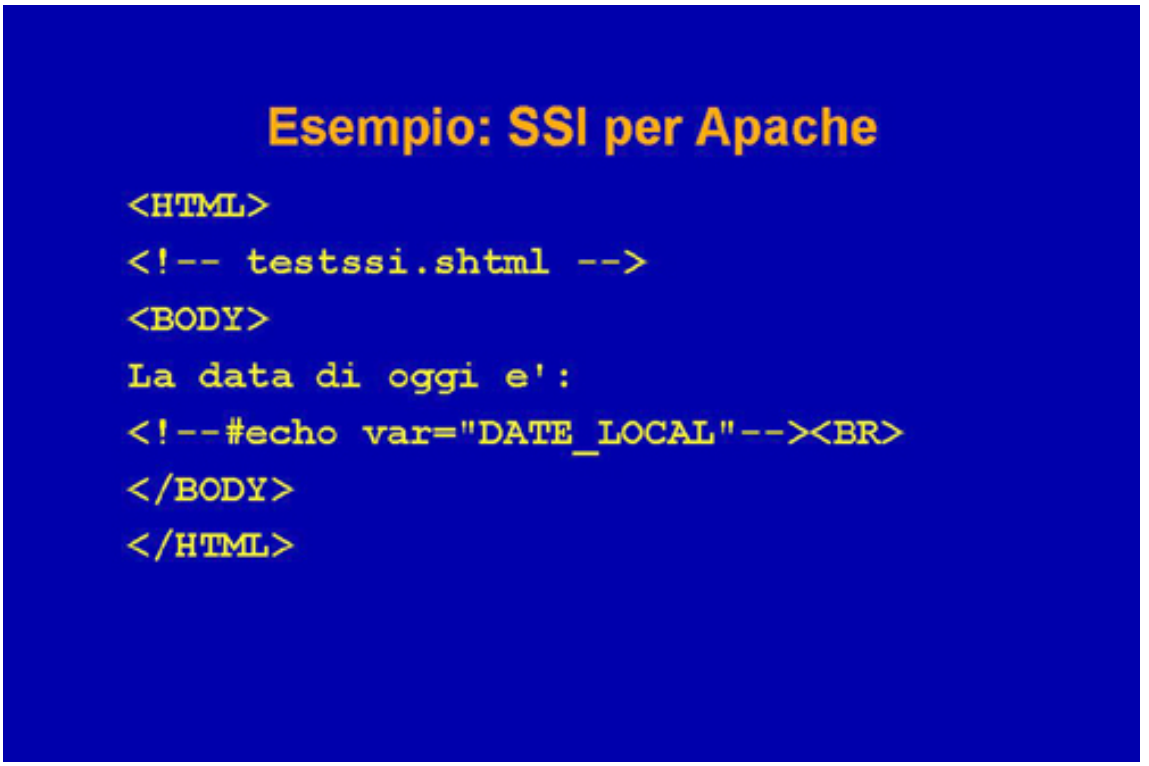
Esiste, infine, un quarto modello di architettura a tre livelli ed è il cosiddetto modello a codice mobile. In questo caso, vedremo, oltre ad inviare sul *client* del codice HTML, si provvede ad inviare del codice che verrà eseguito sul *client* stesso: dei piccoli programmi che sfrutteranno le capacità di elaborazione del *client*. Questo fa sì che per interazioni con l'utente non sia necessario accedere nuovamente alla rete, ma queste possano essere fatte in locale dal pezzo di codice inviato dal *server* sul *client* stesso. L'architettura è quella indicata in questa *slide* e, come vedete, in questo caso ad una richiesta HTTP proveniente dal *client* corrisponde una risposta del *server* contenete non solo HTML ma anche del codice, che verrà inviato dal *server* verso il *client*. Questo codice verrà, a questo punto, eseguito sul *client* stesso. In questo caso abbiamo l'esempio di una *applet*, una applicazione proveniente dalla rete, che verrà eseguito sulla macchina virtuale del *client* stesso. Questo codice potrà, a sua volta, effettuare ulteriori connessioni verso la rete ed accedere a risorse che si trovano, ad esempio, su un sistema informativo remoto. Il sistema informativo potrà, a questo punto, rispondere a questa richiesta di connessione, inviando i dati richiesti. In questo caso, notate, che l'architettura prevede che il *client* sia in grado di eseguire delle applicazioni provenienti dalla rete. Questo tipo di architettura richiederà quindi dei *client* particolari, in grado di eseguire questo tipo di codice. Vedremo che le architetture principali che prevedono questo meccanismo sono le architetture basate su *Java* e quelle basate su *ActiveX*, in ambiente *Microsoft*. Vedremo anche che ci sono una serie di linguaggi di *script Web* che possono essere utilizzati per spedire applicazioni dal *server* verso il *client*.

SSI per Apache Web Server

SSI per Apache Web Server

Proseguiamo in questa analisi dell'architettura a tre livelli e cominciamo ad analizzare alcune specifiche soluzioni che prevedono l'utilizzo di una di queste architetture. Nel corso di questa lezione vedremo, ancora, come è organizzata la soluzione *Server Side Include* per il *Web server Apache*. Nella lezione successiva ci occuperemo invece di soluzioni tipicamente *Microsoft*; vedremo in particolare l'architettura ASP che è anch'essa una soluzione di tipo *Server Side Include* e vedremo la soluzione ISAPI, che invece è una soluzione di tipo *Web* API. Quasi tutti i principali *server Web* consentono di utilizzare *Server Side Include* nelle pagine HTML. Andiamo a considerare un *server* di esempio, il *server Apache*, che è uno dei *server* più diffusi e tra i più utilizzati in *Internet*, e andiamo a vedere in che modo è possibile inserire delle *Server Side Include* in questo tipo di *server*. Questo ci consentirà di comprendere meglio quali sono gli aspetti architetturali di questo tipo di soluzione. In particolare, il *server Apache* ha una direttiva *include* che consente di inserire all'interno di una pagina HTML un altro *file*, oppure di eseguire una serie di comandi built-in (comandi supportati dal *server* stesso), oppure che consente di eseguire dei processi che si trovano sul *server Web* stesso.

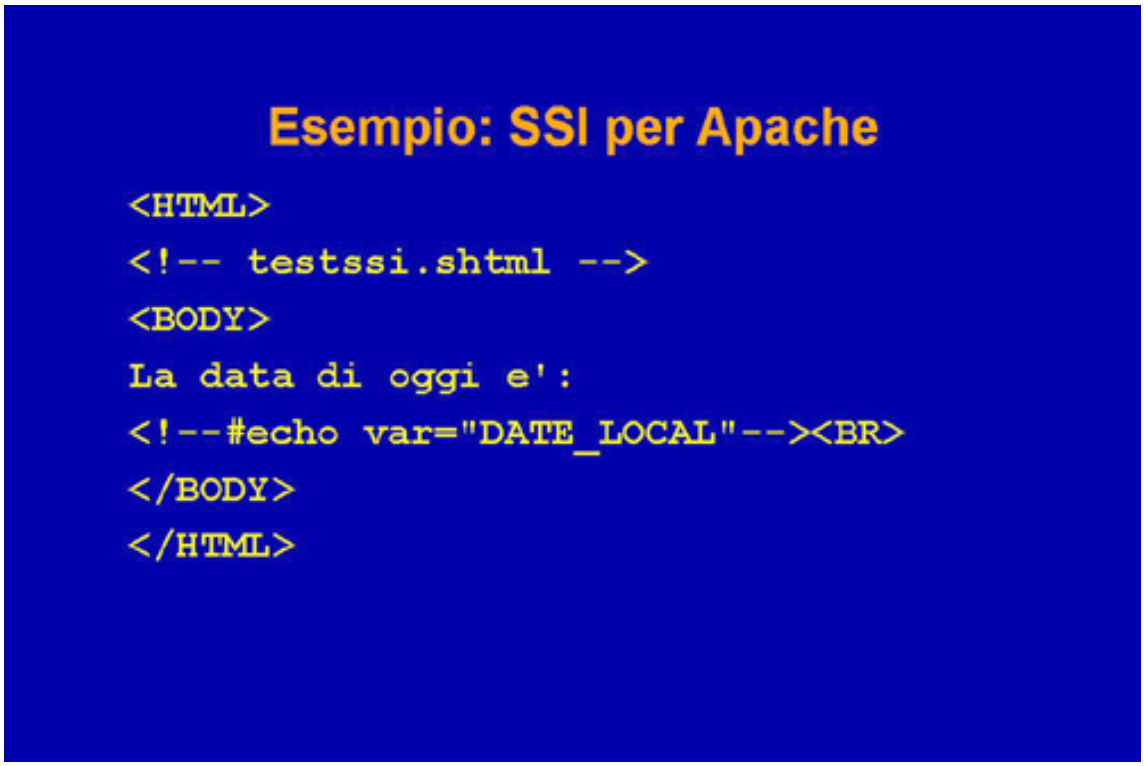
Esempio: SSI per Apache (1)



Esempio: SSI per Apache (1)

Andiamo ad analizzare come questa soluzione, di tipo *Server Side Include*, sia supportata nel *server Web Apache* e cerchiamo di comprendere bene quali sono i concetti strutturali e procedurali. Nell'esempio, molto semplice, che andiamo a vedere, abbiamo un documento HTML che contiene una direttiva *Server Side Include*. In questo caso la sintassi mostrata è quella prevista dai *server Web Apache*. Altri *server Web* avranno sintassi simili che consentono di realizzare funzionalità equivalenti. Notate che la pagina HTML è quasi statica, nel senso che contiene informazioni HTML tradizionali, come ad esempio questa linea dove viene scritto: la data di oggi è, ma contiene anche dei comandi che devono essere eseguiti ed interpretati dal *server* prima di distribuire la pagina. In particolare, notate il comando echo, che esegue la visualizzazione della variabile specificata nell'attributo var, che in questo caso si chiama DATE_LOCAL. Questa variabile conterrà, quindi, la data di sistema del *server* al quale ci colleghiamo. Notate che queste direttive di *Server Side Include* sono in mezzo all'HTML, occorrerà quindi utilizzare dei meccanismi per farle riconoscere al *server Web* nel momento in cui questa pagina viene distribuita. In questo caso non è un vero e proprio commento, ma è un commento HTML che inizia con questa sequenza di 4 caratteri a cui segue immediatamente un carattere #. Questo è il segnale che il *server* utilizza per capire che si tratta di una direttiva di *Server Side Include*, e quindi, per eseguire il comando built_in echo che visualizza il valore di questa variabile.

Esempio: SSI per Apache (2)



Esempio: SSI per Apache (2)

Spostiamoci quindi sul PC e andiamo a vedere come funziona questa pagina HTML. Dobbiamo andare a specificare, ovviamente, l'URL di questo documento e in particolare, in questo caso, utilizziamo l'URL testssi.shtml che individua il *file*. Dicendo al nostro *browser* di caricare questa pagina, la pagina verrà caricata e il *server*, prima di spedire questa pagina al *client*, andrà a sostituire la direttiva *Server Side Include* con l'*output* di quel comando. In questo caso si trattava del contenuto della variabile DATE_LOCAL, che specifica qual è la data locale e qual è il fuso orario corrente sul nostro *server Web*. Notate, e lo possiamo vedere andando ad analizzare il sorgente di questa pagina HTML ricevuta, che il contenuto di questa pagina è un contenuto puramente HTML. Il *server*, prima di spedire questa pagina, ha sostituito l'*output* del comando echo e ha consentito quindi di passare un parametro generato in tempo reale dal *server* stesso. Questo esempio chiude questa prima lezione sulle architetture a tre livelli e in particolare ci mostra un esempio della architettura *Server Side Include* per il *server Web Apache*. Nella prossima lezione andremo a vedere quali sono le soluzioni a tre livelli per i *server Microsoft*, in particolare andremo a vedere l'architettura ASP e l'architettura ISAPI.

Bibliografia

Introduzione

15.3 Scripting

Castro E.*HTML 4 per il World Wide Web;* 2000*Addison Wesley*

R. Boschin*HTML 4;* 2ed. 2000Apogeo

S. Isaacs*Inside Dynamic HTML;* 1999Mondadori informatica

Marco Calvo, Gino Roncaglia, Fabio Ciotti, Marco A. Zela*Internet 2000;* 2000Editori Laterza

Consorzio W3C; http://www.w3.org/

Guida HTML; http://html.it/

Sito di riferimento per le chat; http://www.irc.net

Approfondimenti

15.4 Common Gateway Interface (CGI), Active Server Page (ASP) e Servlet CGI

J. Reilly*ASP.NET progettare applicazioni;* 2002Mondadori Informatica

Malacarne*Java Servlet;* 2001Apogeo

Notizie sui CGI; http://www.cgidir.com/

Servlet, sito ufficiale SUN; http://java.sun.com/products/servlet/

ASP piattaforma .NET; http://www.microsoft.com/net/

15.5 Il linguaggio PHP

Hughes, Zmievski*PHP Soluzioni professionali per lo sviluppatore;* 2001Apogeo

Sito ufficiale PHP; http://www.php.net

Glossario

ActiveX : Una tecnologia che si prefigge gli stessi scopi di *Java* ma non ad architettura aperta (è di proprietà della *Microsoft* e permette la realizzazione di moduli di codice incorporabili tramite OLE).

Anchor : Sinonimo di rimando (o link o *hyperlink*).

Applet : Un piccolo programma che può essere prelevato velocemente dalla rete e usato da qualsiasi *computer* dotato di un *browser* capace di eseguire codice *Java*.

Applicazione : Un programma (*software*) che svolge determinate funzioni per l'utente finale. Esempi di applicazioni sono i *client* FTP, Telnet, *E-mail* e i *browser*.

Asincrona : Tipo di trasmissione dati, a volte chiamata trasmissione *start-stop*, in cui la sincronizzazione tra trasmettitore e ricevitore viene ripristinata tramite uno o più bit di *start* all'inizio di ogni carattere.

ASP : Un ambiente di *scripting* per *Microsoft Internet Information Server*, in cui è possibile combinare HTML, *script* e componenti riusabili di *ActiveX*, per creare pagine *Web* dinamiche.

CGI : (*Common Gateway Interface*). È un metodo con cui un *server* HTTP interagisce con *database*, documenti, e altri programmi inviando o ricevendo dati in formato HTML sul *browser client*. L'adozione di questo schema di elaborazione distribuita comporta l'esecuzione *server-side* di un programma (*CGI script*) che effettua l'elaborazione e restituisce la risposta in HTML al *client*. Molto spesso si può capire che viene usato un programma *CGI* se compare il termine cgi-bin nell'URL della pagina.

Chat : Qualunque sistema che consente a un qualunque numero di utenti collegati di avere una conversazione in tempo reale, sia consentendo che gli utenti siano collegati sullo stesso *computer* o, più comunemente, su una rete.

Client : Un programma usato per ottenere dati da un programma *server* residente su un altro *computer* situato da qualche parte nel mondo. Ogni programma *client* è progettato per colloquiare solo con uno o più particolari tipi di programmi *server*, ed ogni *server* richiede un determinato tipo di *client*.

Cookie : Un meccanismo che consente a programmi attivi sul *server* (per esempio, i programmi *CGI*) di immagazzinare e recuperare dati sul lato *client* della connessione. Ciò significa che quando si accede di nuovo ad un certo URL, il *server* può riutilizzare i dati presenti direttamente sulla macchina *client*. Qualche esempio dati relativi al *login*, a registrazioni, ad acquisti in linea, eccetera. Poiché i *cookie* possono conservare informazioni utente sul *computer* di quest'ultimo, sono spesso usati per riconoscerlo e personalizzargli l'ambiente. Tipicamente, le informazioni memorizzate dai *cookie* scadono dopo un certo periodo di tempo.

E-mail : Abbreviazione di *electronic mail* (posta elettronica). È un sistema che consente la trasmissione e la ricezione su una rete informatica, di messaggi indirizzati secondo indirizzi a struttura standard. È una comunicazione di tipo asincrono perché anche in caso di indisponibilità dell'utente ricevitore, i

messaggi vengono memorizzati nel *server* a cui questo utente riferisce.

Frame (html) : Un'area dello schermo, all'interno della finestra del *browser*, che visualizza una pagina *Web*. Sullo schermo possono così esistere più pagine contemporaneamente, ognuna in un *frame* diverso. Tale tecnica è consentita dalle ultime versioni di HTML.

Get : Istruzione data ad un *server* HTTP per richiedere una risorsa. Al contrario del metodo *post*, restituisce sempre una risorsa al *client*.

HTML : (*HyperText Markup Language*). Linguaggio di realizzazione di ipertesti (interpretato), utilizzato per la realizzazione di pagine *Web* trasmesse mediante protocollo applicativo HTTP. Una pagina HTML può contenere testo, immagini, brani audio e sequenze video con vari formati e trasmessi come *file* dal *server* al *client*.

Internet : *internet* (con la i minuscola) indica una qualsiasi connessione tra due reti. Con la I maiuscola: la più grande rete di calcolatori al mondo, basata sull'architettura di rete *TCP/IP*.

IRC : (*Internet Relay Chat*). Servizio applicativo, basato sul servizio di trasporto fornito da UDP, che consente ad un utente *Internet* di comunicare in tempo reale con altri utenti di *Internet*.

Java : Un linguaggio di programmazione *Object Oriented*, sviluppato da *Sun Microsystems* e disponibile già da diversi anni, specificatamente progettato per la scrittura di programmi che possono essere scaricati sul proprio *computer* dalla rete ed immediatamente eseguiti localmente. Utilizzando piccoli programmi *Java* (chiamati *Applet*), le pagine *Web* possono includere animazioni, effettuare calcoli e quant'altro.

Javascript : Mentre un programma scritto in *Java* va sottoposto ad un processo di meta-compilazione per poter essere eseguito, *Javascript* è un linguaggio interpretato che può essere inserito direttamente nel codice HTML dei documenti *Web*.

Linguaggio di *Scripting* : Un termine generico per qualunque linguaggio che è debolmente tipato o senza tipi, e non consente di utilizzare strutture dati complesse. Un programma in questo linguaggio è di solito interpretato.

Mailing list : Un sistema (solitamente automatizzato) che, ricevuto un messaggio *E-mail* da un utente, lo invia a tutti i componenti registrati di una lista. Così facendo è possibile partecipare a discussioni su vari argomenti.

PERL : (*Practical Estraction and Report Language*). Si tratta di un linguaggio finalizzato principalmente alla trattazione di stringhe e *file* di testo. Per questi motivi il Perl è usato nella scrittura di procedure *CGI* installate su un *server Web*, o per lo sviluppo di procedure di manutenzione delle attività di un *server*.

PHP : (*Professional Home Pages*). Rappresenta un linguaggio standard per la

realizzazione di pagine HTML dinamiche ottenute mediante elaborazione lato *server*. È un linguaggio interpretato dal motore PHP del *server* che ospita le pagine elaborate, ed è HTML *embedded*, ossia le istruzioni sono ospitate all'interno di pagine HTML.

Proxy : Componente *software* e/o *hardware* utilizzato per varie funzioni per esempio per controllare da un punto unico l'accesso ad ***Internet*** e per analizzare i pacchetti *IP* che ***Internet*** accedono all'interno di una rete privata. Un *proxy* può avere anche le funzioni di traduzione di indirizzi *IP* privati in indirizzi *IP* pubblici e viceversa.

Script : Un programma scritto in un linguaggio di *scripting*.

Sincrona : Tipo di trasmissione dati in cui la comunicazione tra trasmettitore e ricevitore avviene allo stesso tempo, o alla stessa velocità, o in modo regolare e predicibile.

Server : Un *computer* o un programma che fornisce un determinato tipo di servizio ad un programma *client* in esecuzione su un *computer* remoto. Una stessa macchina può eseguire contemporaneamente più di un programma fungendo quindi da più *server* per molti *client* sulla rete.

Servlet : Un programma per *server* che garantisce funzionalità aggiuntive ai *server* abilitati *Java*.

Tag : Marcatore, racchiuso tra parentesi angolari, che costituisce l'elemento caratterizzante l'HTML (per esempio: <h1>, </H1>, <IMG SRC="stella.gif">).

Autori

Hanno realizzato il materiale di questo modulo:

Prof. Cosimo Laneve

Professore Straordinario di Informatica presso l'Università di Bologna, dove insegna Linguaggi di Programmazione e Qualità del *Software*. Ha ricevuto il Dottorato di Ricerca in Informatica dall'Università di Pisa ed è stato *Research Associate* presso l'INRIA di *Sophia Antipolis* in Francia. Attualmente è coordinatore di progetti nazionali e internazionali che riguardano i fondamenti teorici e l'implementazione di linguaggi di programmazione distribuiti, di verifica statica di programmi e di teoria dei tipi. Relativamente a queste tematiche, ha pubblicato su numerose riviste e conferenze internazionali.