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

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:

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:

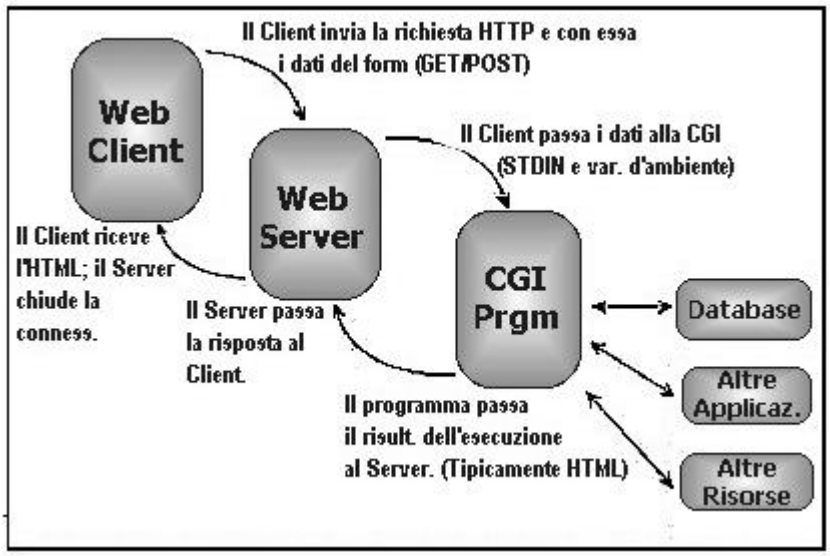
1. quali dati fornire come *input*;
2. come elaborare i dati ottenuti in *output*;
3. i meccanismi da imparare sono pertanto quelli necessari a maneggiare i dati che il *server* passa al programma e a restituire i dati da questo generati.

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:

1. un *client Web* realizza una connessione con un *server Web* (tipicamente tramite un *browser*), all'indirizzo specificato nella *URL*.
2. Il *client Web* invia una richiesta (tramite uno tra i metodi *POST* e *GET* visti nel paragrafo precedente).
3. I dati inviati dal *client* sono passati dal *server* al programma **CGI** referenziato nella *URL*.
4. Il programma **CGI** legge i dati ed esegue il processo per cui è stato creato.
5. 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.
6. Dopo aver passato la risposta fornita dal programma **CGI** al *client*, il *server* chiude la connessione aperta al punto 1.



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:

1. **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
```

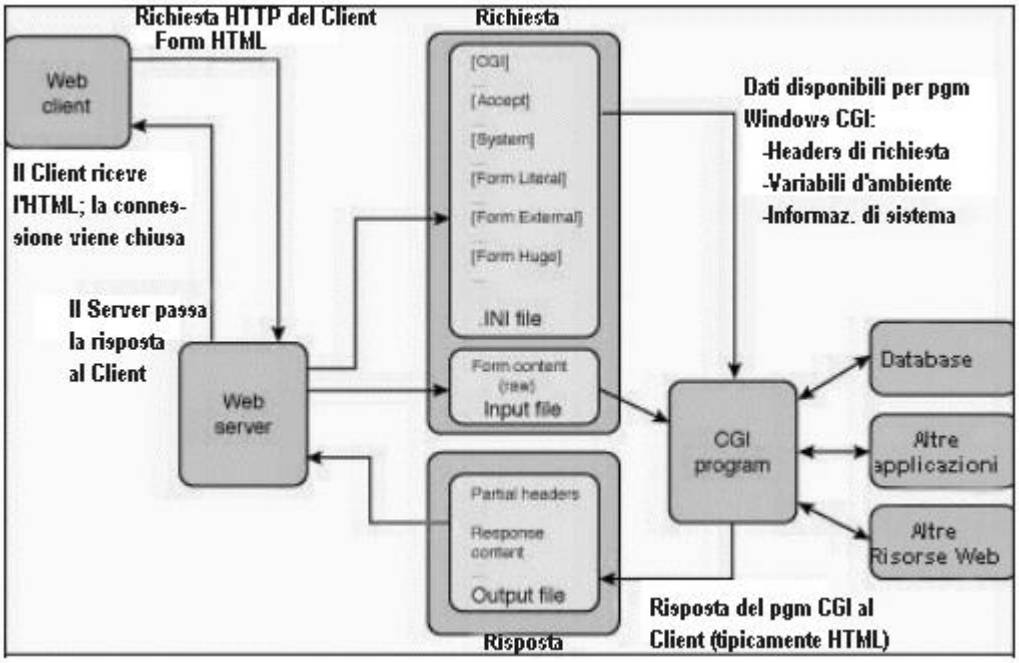
- **AUTH_TYPE**. Identifica il metodo da utilizzare per validare gli utenti nel caso il *server* supporti l'autenticazione ed esegua i programmi **CGI** in modalità protetta.

2. **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.
3. **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.
4. **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*.



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*.

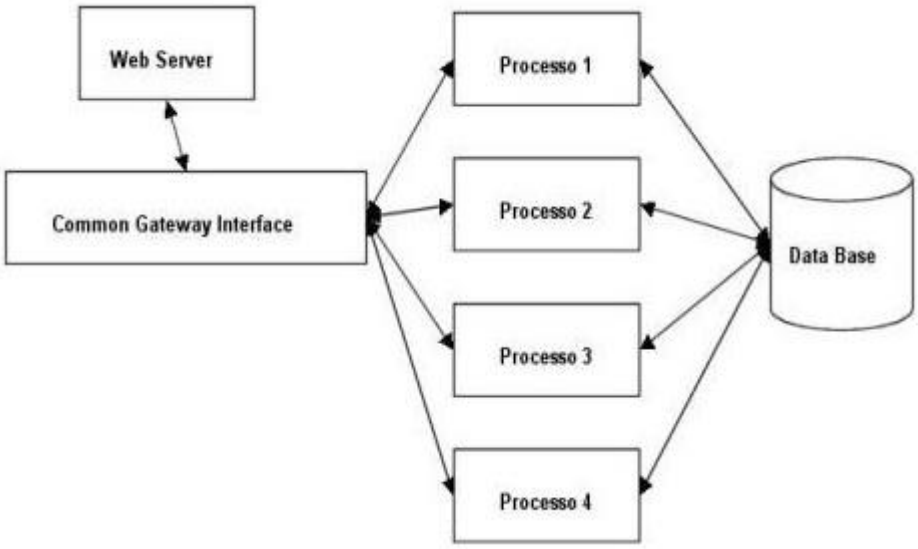
```
<% For i = 3 To 7 %>
<FONT size="<%=i%>">
Hello World!<BR>
</FONT>
<% Next %>
```

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.



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*.

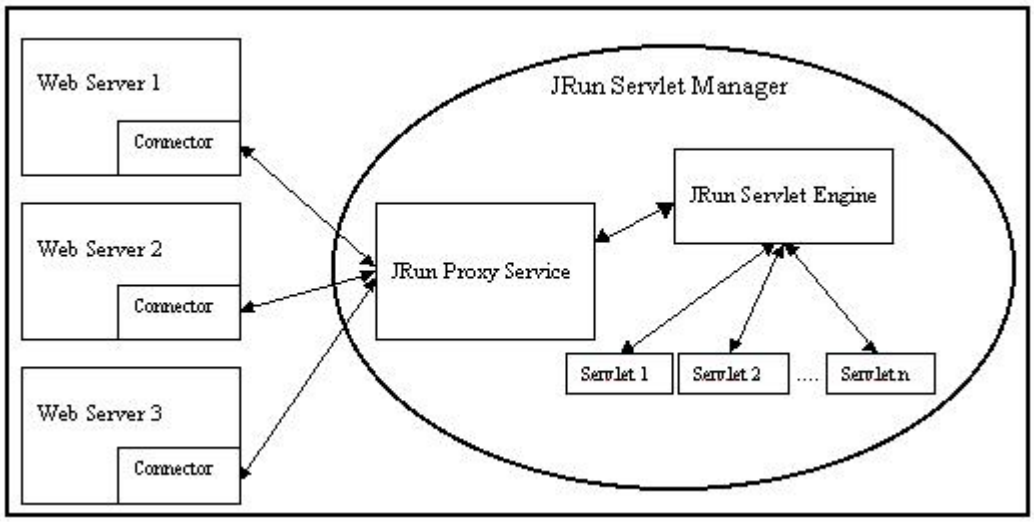
Principio di funzionamento

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.



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 *Http**Servlet**Request, Http**Servlet**Response*. 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 { }
}
```