

# **Applicazioni Web**

## **Introduzione al Server Side**

Danilo Croce

Maggio 2020

# Introduzione

- **Formati di dati per il Web**
  - HTML, XML, DTD
- **Architetture a tre livelli nel Web**
- **Il livello di presentazione**
  - Moduli HTML: GET e POST HTTP, codifica di URL; Javascript;
- **Il livello intermedio**
  - CGI, application server, servlets, JavaServerPages, passaggio di argomenti, Gestione dello stato (cookie)

# FORMATI DI DATI PER IL WEB

# Hypertext Transfer Protocol

- Cos'è un protocollo di comunicazione?
  - Insieme di standard che definisce la struttura dei messaggi
  - Esempi: TCP, IP, HTTP
- Che succede se fate click su <http://www.cs.wisc.edu/~dbbokk/index.html?>
  - Il client (browser web) manda una richiesta HTTP al server
  - Il server riceve la richiesta e risponde
  - Il client riceve la risposta; invia altre richieste

# HTTP (segue)

dal client al server :

```
GET ~/index.html HTTP/1.1
User-agent: Mozilla/4.0
Accept: text/html, image/gif,
        image/jpeg
```

il server risponde :

```
HTTP/1.1 200 OK
Date: Mon, 04 Mar 2002 12:00:00 GMT
Server: Apache/1.3.0 (Linux)
Last-Modified: Mon, 01 Mar 2002
          09:23:24 GMT
Content-Length: 1024
Content-Type: text/html
<HTML> <HEAD></HEAD>
<BODY>
<h1>Libreria Internet di Barns e Nobble
    </h1>
Il nostro catalogo :
<h3>Scienza</h3>
<b>Natura della legge fisica </b>
...
```

# Struttura del protocollo HTTP

- Richieste HTTP
- Linea di richiesta: GET ~/index.html HTTP/1.1
  - GET: campo del metodo HTTP (valori possibili sono GET e POST, più avanti)
  - ~/index.html: campo URI
  - HTTP/1.1: campo della versione HTML
- Tipo di client: User-agent: Mozilla/4.0
- Che tipi di documenti verranno accettati dal client:  
Accept: text/html, image/gif, image/jpeg

# Struttura del protocollo HTTP (segue)

## Risposte HTTP

- Linea di stato: HTTP/1.1 200 OK
  - Versione HTTP: HTTP/1.1
  - Codice di stato: 200
  - Messaggio del server: OK
  - Combinazioni comuni di codice di stato/messaggio del server:
    - 200 OK: la richiesta ha avuto successo
    - 400 Bad Request: il server non ha potuto soddisfare la richiesta
    - 404 Not Found: l'oggetto richiesto non esiste sul server
    - 505 HTTP Version not Supported
- Data di creazione dell'oggetto:
- Last-Modified: Mon, 01 Mar 2002 09:23:24 GMT
- Numero di bytes spediti: Content-Length: 1024
- Tipo di oggetto che viene spedito: Content-Type: text/html
- Altre informazioni quali il tipo di server, l'ora del server, etc.

# Formati di dati per Web

- HTML
  - Il linguaggio di **presentazione** per Internet
- XML
  - Un **modello di dati** gerarchico auto-descrittivo
- DTD
  - Schemi standardizzati per XML
- XSLT (non trattato nel corso)



# HTML: un esempio

```
<HTML>
  <HEAD></HEAD>
  <BODY>
    <h1>Libreria Internet Barns &
      Nobble </h1>
    Il nostro inventario :

    <h3>Scienza</h3>
    <b>Natura della legge fisica </b>
    <UL>
      <LI>Autore: Richard
        Feynman</LI>
      <LI>Pubblicato nel 1980</LI>
      <LI>Copertina dura</LI>
    </UL>
```

```
    <h3>Fiction</h3>
    <b>Aspettando il Mahatma</b>
    <UL>
      <LI>Autore: R.K. Narayan</LI>
      <LI>Pubblicato nel 1981</LI>
    </UL>
    <b>L'insegnante di Inglese</b>
    <UL>
      <LI>Autore: R.K. Narayan</LI>
      <LI>Pubblicato nel 1980</LI>
      <LI>Tascabile</LI>
    </UL>

  </BODY>
</HTML>
```

# HTML: breve introduzione

- L'HTML è un linguaggio di marcatura
- I comandi sono tag
  - Tag di inizio e di fine
  - Esempi
    - `<HTML>...</HTML>`
    - `<UL>...</UL>`
- Molti editor generano automaticamente l'HTML direttamente dal documento (ad esempio Microsoft Word ha una funzione "Salva come HTML")

# HTML: esempio di comandi

- `<HTML>`
- `<UL>`: lista non ordinata
- `<LI>`: elemento di una lista
- `<h1>`: intestazione più grande
- `<h2>`: intestazione di secondo livello, analogamente `<h3>`, `<h4>`
- `<B>Title</B>`: grassetto

# XML: un esempio

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<LISTALIBRI>
  <LIBRO GENERE="Scienza" FORMATO="Copertina dura">
    <AUTORE>
      <NOME>Richard</NOME>
      <COGNOME>Feynman</COGNOME>
    </AUTORE>
    <TITOLO>Natura della legge fisica</TITOLO>
    <PUBBLICATO>1980</PUBBLICATO>
  </LIBRO>
  <LIBRO GENERE="Fiction">
    <AUTORE>
      <NOME>R.K.</NOME>
      <COGNOME>Narayan</COGNOME>
    </AUTORE>
    <TITOLO>Aspettando il Mahatma</TITOLO>
    <PUBBLICATO>1981</PUBBLICATO>
  </LIBRO>
  <LIBRO GENERE="Fiction">
    <AUTORE>
      <NOME>R.K.</NOME>
      <COGNOME>Narayan</COGNOME>
    </AUTORE>
    <TITOLO>L'insegnante di inglese</TITOLO>
    <PUBBLICATO>1980</PUBBLICATO>
  </LIBRO>
</LISTALIBRI>
```

# XML: eXtensible Markup Language

- Language
  - Un modo di comunicare informazione
- Markup
  - Note o meta-dati che descrivono i dati o il linguaggio
- Extensible
  - Capacità illimitata di definire nuovi linguaggi o insiemi di dati

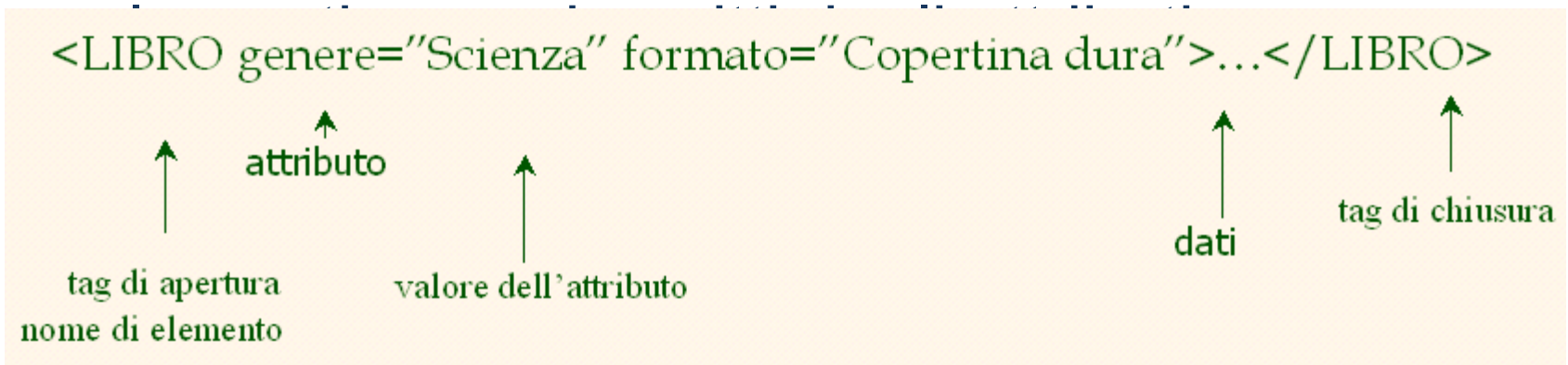
# XML – Qual è il punto?

- Si possono includere i propri dati e una descrizione di ciò che tali dati rappresentano
  - Utile per definire il proprio linguaggio o protocollo personale
- Esempio: Chemical Markup Language

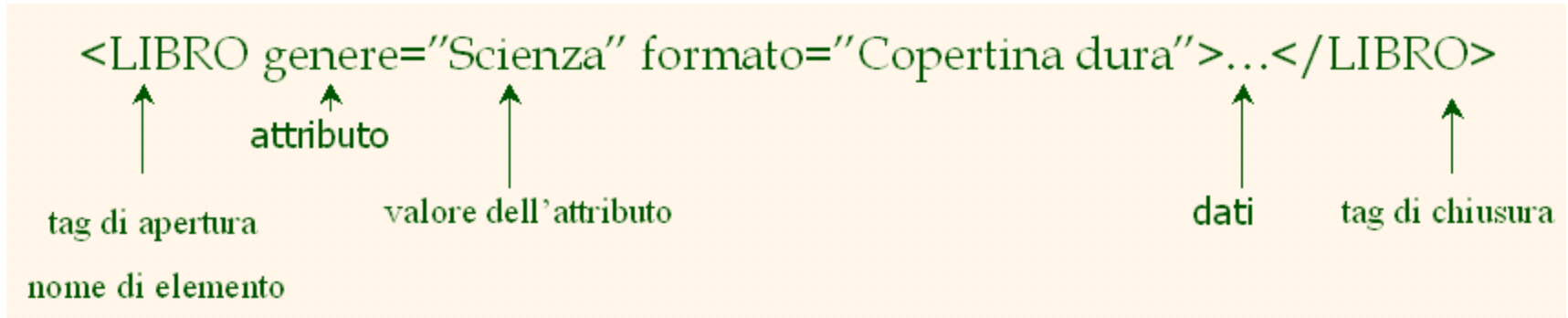
```
<molecola>  
  <peso>234.5</peso>  
  <Spettro>...</Spettro>  
  <Numeri>...</Numeri>  
</molecola>
```
- Obiettivi del progetto XML:
  - L'XML dovrebbe essere compatibile con SGML
  - La scrittura di programmi che elaborano documenti XML dovrebbe essere un compito semplice
  - Il progetto dovrebbe essere formale e preciso

# XML – Struttura

- XML: punto di incontro di SGML e HTML
- L'XML somiglia all'HTML
- L'XML è una gerarchia di tag definiti dall'utente chiamati elementi con attributi e dati
- I dati sono descritti dagli elementi, gli



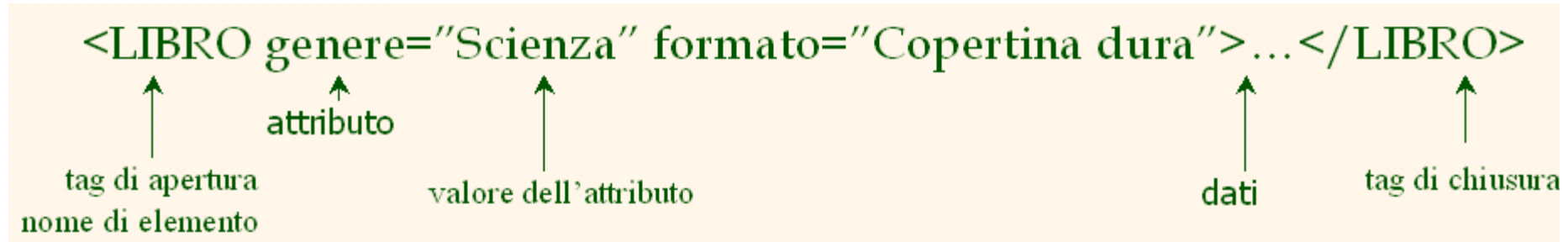
# XML – Attributi



- L'XML è sensibile alle maiuscole e agli spazi
- I nomi dei tag di apertura e chiusura devono essere identici
- Tag di apertura: "`<`" + nome elemento + "`>`"
- Tag di chiusura: "`</`" + nome elemento + "`>`"
- Elementi vuoti non hanno dati e non hanno tag di chiusura:

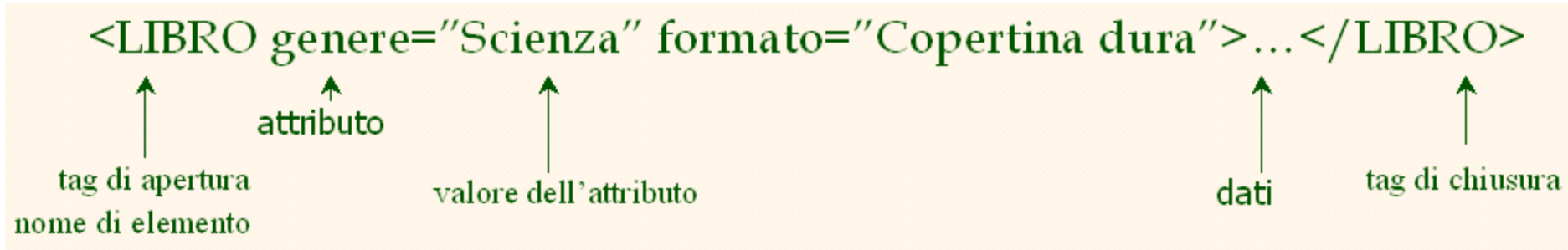


# XML – Attributi



- Gli attributi forniscono informazioni aggiuntive sui tag elementi
- Ci possono essere zero o più attributi in ogni elemento; ciascuno ha la forma
  - Nome\_attributo='valore\_attributo'
    - Non ci sono spazi tra il nome e "="
    - I valori degli attributi devono essere racchiusi dai caratteri ' oppure "
- Attributi multipli sono separati da spazi bianchi (uno o più spazi o tabulazioni)

# Dati e commenti



- I dati XML sono qualunque informazione tra un tag di apertura e un tag di chiusura
- I dati XML non devono contenere i caratteri '`<`' oppure '`>`'
- Commenti:  
`<!-- commento -->`

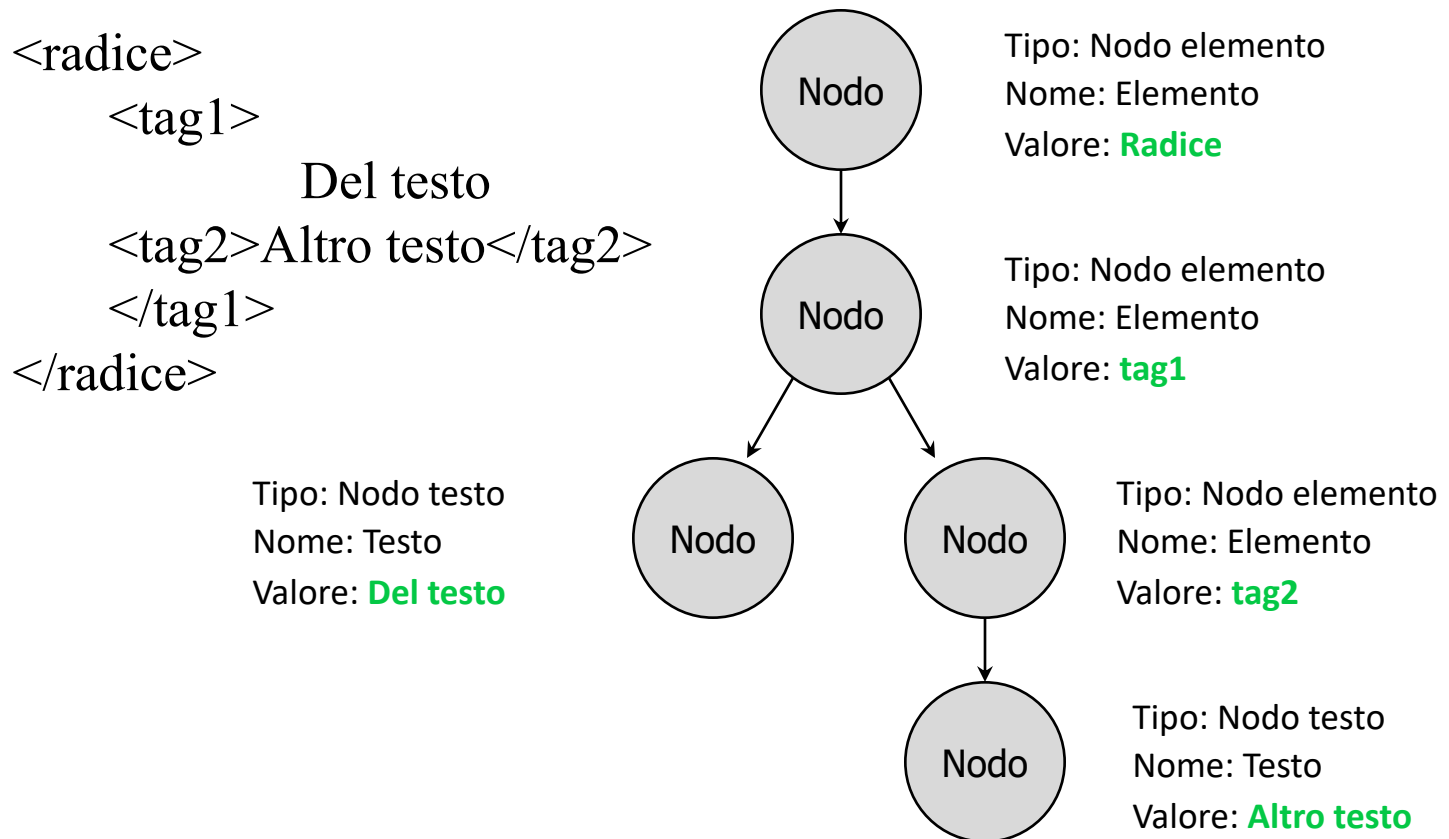
# Annidamento e gerarchia

- I tag XML possono essere annidati in una gerarchia ad albero
- I documenti XML possono avere un solo tag radice
- Tra un tag di apertura e un tag di chiusura si possono inserire:
  1. dati
  2. altri elementi
  3. una combinazione di dati ed elementi

```
<radice>  
  <tag1>  
    Del testo  
    <tag2>Dell'altro</tag2>  
  </tag1>  
</radice>
```

# Memorizzazione

- La memorizzazione viene effettuata proprio come in un albero n-ario (DOM)



# DTD - Document Type Definition

- Un DTD è uno schema per i dati XML
- I protocolli e i linguaggi XML possono essere standardizzati con file DTD
- Un DTD dice quali elementi e attributi sono obbligatori e quali opzionali
  - Definisce la struttura formale del linguaggio

# DTD – Un esempio

```
<?xml version='1.0'?>
<!ELEMENT Cesto (Ciliegia+, (Mela | Arancia)*)>
<!ELEMENT Ciliegia EMPTY>
    <!ATTLIST Ciliegia sapore CDATA #REQUIRED>
<!ELEMENT Mela EMPTY>
    <!ATTLIST Mela colore CDATA #REQUIRED>
<!ELEMENT Arancia EMPTY>
    <!ATTLIST Arancia provenienza 'Florida'>
```

---



<Cesto>

```
<Ciliegia sapore='buono' />
<Mela colore='rosso' />
<Mela colore='verde' />
```

</Cesto>



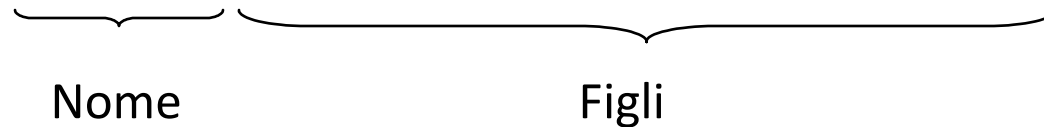
<Cesto>

```
<Mela />
<Ciliegia sapore='buono' />
<Arancia />
```

</Cesto>

# DTD - !ELEMENT

<!ELEMENT Cesto (Ciliegia+, (Mela | Arancia)\*)>



- !ELEMENT dichiara il nome di un elemento, e quali elementi figli dovrebbe avere
- Tipi di contenuto:
  - Altri elementi
  - #PCDATA (parsed character data)
  - EMPTY (nessun contenuto)
  - ANY (nessun controllo all'interno di questa struttura)
- Una espressione regolare

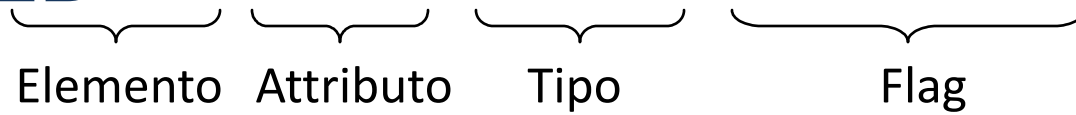
# DTD - !ELEMENT (segue)

- Una espressione regolare ha la seguente struttura:
  - $\text{exp1}, \text{exp2}, \text{exp3}, \dots, \text{expk}$ : una lista di espressioni regolari
  - $\text{exp}^*$ : una espressione opzionale con zero o più occorrenze
  - $\text{exp}^+$ : una espressione opzionale con una o più occorrenze
  - $\text{exp1} \mid \text{exp2} \mid \dots \mid \text{exp3}$ : una disgiunzione di espressioni



# DTD - !ATTLIST

<!ATTLIST Ciliegia sapore CDATA  
#REQUIRED>



<!ATTLIST Arancia provenienza CDATA #REQUIRED colore 'arancione'>

!ATTLIST definisce una lista di attributi per un elemento

- Gli attributi possono essere di tipi diversi, possono essere obbligatori o opzionali, e possono avere valori predefiniti

# DTD – Ben formato e valido

```
<?xml version='1.0'?>  
<!ELEMENT Cesto (Ciliegia+)>  
  <!ELEMENT Ciliegia EMPTY>  
  <!ATTLIST Ciliegia sapore CDATA #REQUIRED>
```

Non ben formato

```
<Cesto>  
  <Ciliegia sapore=buono>  
</Cesto>
```

Ben formato ma non valido

```
<Lavoro>  
  <Luogo>Casa</Luogo>  
</Lavoro>
```

Ben formato e valido

```
<Cesto>  
  <Ciliegia sapore='buono' />  
</Cesto>
```

# XML e DTD

- Un numero sempre maggiore di DTD verrà sviluppato
  - MathML
  - Chemical Markup Language
- Permette rapidi scambi di dati con la stessa semantica
- Sono disponibili sofisticati linguaggi di interrogazione:
  - Xquery
  - Xpath

# ARCHITETTURE A TRE LIVELLI E WEB

# Componenti dei sistemi *“data-intensive”*

Tre tipi separati di funzionalità:

- gestione dei dati
  - logica di applicazione
  - presentazione
- 
- L'architettura del sistema determina se queste tre componenti risiedono su un singolo sistema (tier) oppure se sono distribuite su diversi tier

# Architettura a livello singolo

Tutte le funzionalità sono combinate in un singolo tier, generalmente un mainframe

- Accesso utente tramite terminali non intelligenti

Vantaggi :

- facilità di manutenzione e amministrazione

Svantaggi :

- Oggi gli utenti si aspettano interfacce utente di tipo grafico
- Il calcolo centralizzato di tutte le interfacce grafiche è troppo costoso per un singolo sistema

# Architetture client-server

- **Divisione del lavoro: thin client**
  - Il client implementa solo l'interfaccia utente grafica
  - Il server implementa la logica dell'applicazione e la gestione dei dati
- **Divisione del lavoro: thick client**
  - Il client implementa sia l'interfaccia grafica che la logica dell'applicazione
  - Il server implementa la gestione dei dati

# Architetture client-server

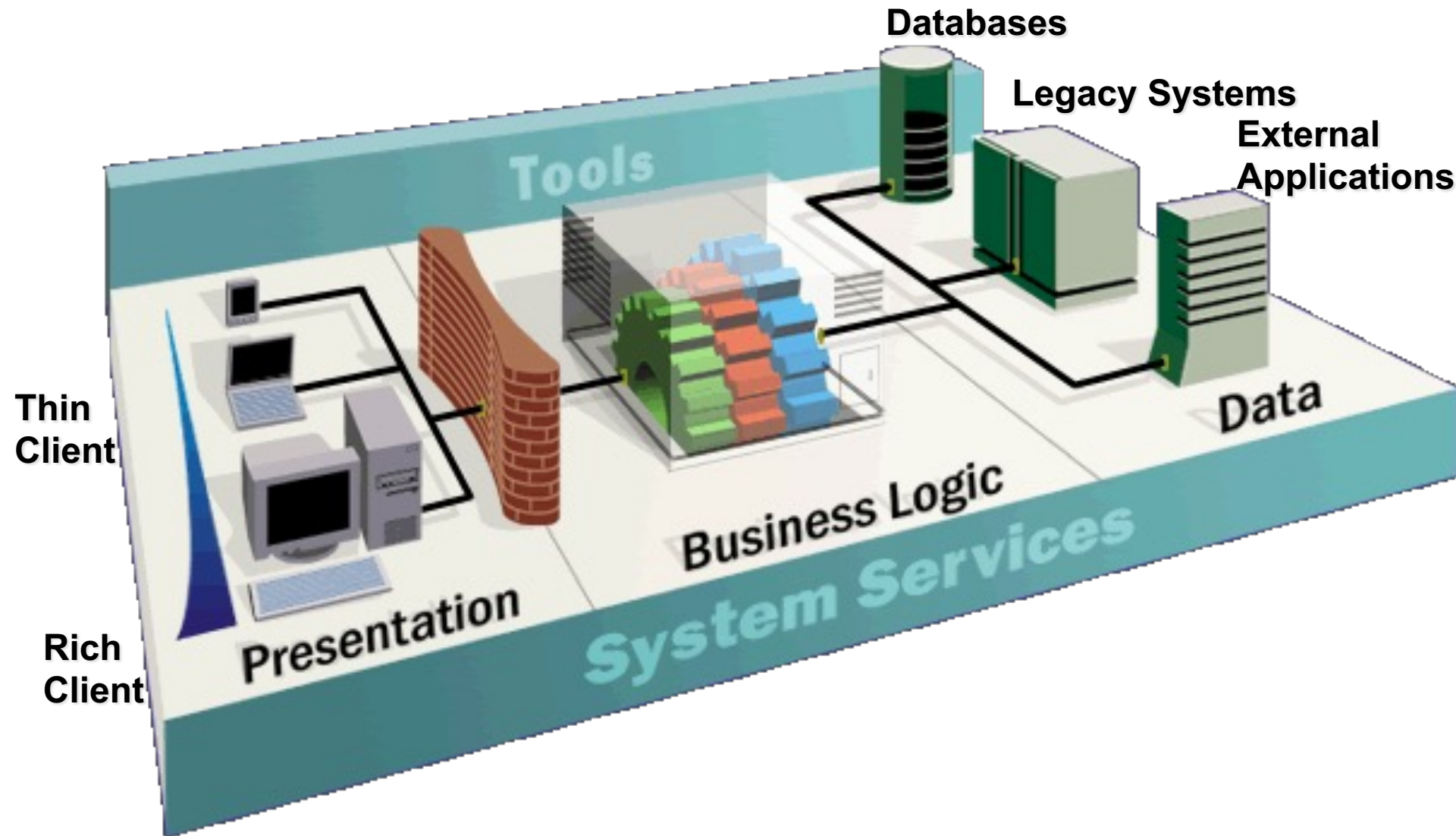
## (segue)

### Svantaggi dei thick client

- Nessun luogo centralizzato per aggiornare la logica dell'applicazione
- Problemi di sicurezza: il server deve fidarsi dei client
  - Il controllo di accesso e l'autenticazione devono essere gestiti dal server
  - I client devono lasciare la base di dati del server in uno stato consistente
  - Una possibilità: incapsulare tutti gli accessi alla base di dati in stored procedure
- Non scalabile a più di un centinaio di client
  - Grossi trasferimenti di dati tra server e client
  - Più di un server crea un problema:  $x$  client,  $y$  server:  $x*y$  connessioni



# Architettura 3-tier



# L'architettura a tre livelli

Livello di presentazione

Programma *client* (*browser web*)

Livello intermedio

Application Server

Livello di gestione dati

Sistema di base di dati

# I tre livelli

## Livello di presentazione

- Interfaccia primaria con l'utente
- Deve adattarsi a diversi dispositivi di visualizzazione (PC, PDA, telefoni cellulari, accesso vocale?)

## Livello intermedio

- Implementa la logica dell'applicazione (implementa azioni complesse, mantiene lo stato tra diversi passi di un flusso di lavoro)
- Accede a diversi sistemi di gestione dei dati

## Livello di gestione dei dati

- Uno o più sistemi standard per la gestione di basi di dati

# Esempio 1: prenotazioni aeree

- Costruire un sistema per prenotazioni aeree
- Cosa viene fatto dai vari livelli?
- Sistema di basi di dati
  - Informazioni sulle aerolinee, posti disponibili, informazioni sui clienti, etc.
- Application server
  - Logica per fare le prenotazioni, cancellare le prenotazioni, aggiungere nuove aerolinee, etc.
- Programma client
  - Log in dei vari utenti, visualizzazione di moduli e output in forma leggibile

# Esempio 2: iscrizione a corsi

- Costruire un sistema usando il quale degli studenti possono iscriversi a dei corsi
- Sistema di base di dati
  - Informazioni sugli studenti, informazioni sui corsi, informazioni sui docenti, disponibilità dei corsi, prerequisiti, etc.
- Application server
  - Logica per modificare un corso, cancellare un corso, creare un nuovo corso, etc.
- Programma client
  - Login dei vari utenti (studenti, personale, professori), visualizzazione di moduli e output in forma leggibile

# Tecnologie

Programma client  
(*Browser web*)

*HTML*  
*Javascript*  
*XSLT*

Application Server  
(*Tomcat, Apache*)

*JSP, Servlets, PHP*  
*CGI, Cookies*

DBMS  
(*MySQL, Oracle, DB2*)

*SQL,*  
*Stored Procedures, XML*

# Vantaggi dell'architettura a tre livelli

- Sistemi eterogenei
- Thin client
- Accesso integrato ai dati
- Scalabilità
- Sviluppo software

# IL LIVELLO DI PRESENTAZIONE



# Introduzione al livello di presentazione

- **Richiamo: funzionalità del livello di presentazione**
  - Interfaccia primaria per l'utente
  - Deve adattarsi ai diversi dispositivi di visualizzazione
  - Funzionalità semplice, come il controllo della validità dei campi
- **Tecnologie:**
  - moduli HTML: come passare dati al livello intermedio
  - JavaScript: funzionalità semplice al livello di presentazione
  - Fogli di stile: separare i dati dalla visualizzazione

# Moduli HTML

- Modo diffuso per comunicare dati **dal client al livello intermedio**
- Formato generale di un modulo :  

```
<FORM ACTION="pagina.jsp" METHOD="GET"  
NAME="ModuloLogin">  
  
...  
</FORM>
```
- Componenti di un tag FORM HTML:
  - ACTION: specifica l'URI che gestisce il contenuto
  - METHOD: specifica il metodo HTML GET o POST
  - NAME: nome del modulo; può essere usato in script sul lato *client* per far riferimento al modulo

# Dentro i moduli HTML

- Tag INPUT:

- Attributi:

- TYPE: text (campo per l'inserimento di testo), password (campo per l'inserimento di testo dove il testo immesso è visualizzato in maniera protetta, reset (ripristina tutti i campi del modulo)
    - NAME: nome simbolico, usato per identificare il valore del campo al livello intermedio
    - VALUE: valore predefinito

- Esempio: `<INPUT TYPE="text" Name="titolo">`

- Modulo di esempio:

```
<form method="POST"  action="Sommaro.jsp">
  <input type="text"      name="userid">
  <input type="password" name="password">
  <input type="submit"   value="Login"  name="Invia">
  <input type="reset"    value="Reimposta">
</form>
```

# Passaggio di argomenti

## Due metodi: GET e POST

- GET

- I contenuti del modulo vanno nell'URI specificato
- Struttura:
  - azione?nome1=valore1&nome2=valore2&nome3=valore3
    - azione: nome dell'URI specificato nel modulo
    - le coppie (nome, valore) provengono dai campi INPUT del modulo; campi vuoti hanno valori vuoti ("nome="))

- esempio dal precedente modulo per l'immissione di una password:

- `Sommario.jps?userid=john&password=johnpw`

- Notate che la pagina chiamata azione deve essere un programma, uno script o una pagina che dovrà elaborare i dati inseriti dall'utente

# Codifica dei campi del modulo

- I campi del modulo possono contenere caratteri ASCII generici che possono non apparire in un URI
- Una speciale convenzione di codifica converte tali valori in caratteri “compatibili con gli URI”:
  - Converte tutti i caratteri “speciali” in %xyz, dove xyz è il codice ASCII del carattere. I caratteri speciali includono &, =, +, %, etc.
  - Converte tutti gli spazi nel carattere “+”
  - Incolla le coppie (nome, valore) dai tag INPUT del modulo tramite “&” per formare l’URI

# Moduli HTML: un esempio completo

```
<form method="POST" action="Sommario.jsp ">
  <table align = "center" border="0" width="300">
    <tr>
      <td>Userid</td>
      <td><input type="text" name="userid" size="20"></td>
    </tr>
    <tr>
      <td>Password</td>
      <td><input type="password" name="password" size="20"></td>
    </tr>
    <tr>
      <td align = "center"><input type="submit" value="Login"
        name="submit"></td>
    </tr>
  </table>
</form>
```

# JavaScript

- Scopo: aggiungere funzionalità al livello di presentazione
- Applicazioni di esempio:
  - rilevare il tipo di browser e caricare una pagina specifica per quel browser
  - validazione di moduli: validare i campi di immissione testo del modulo
  - controllo del browser: aprire nuove finestre, chiudere finestre esistenti (esempio: finestre pop-up di pubblicità)
- Di solito incapsulato direttamente nell'HTML tramite il tag `<SCRIPT>...</SCRIPT>`
- `<SCRIPT>` ha diversi attributi:
  - LANGUAGE: specifica il linguaggio dello script (ad esempio javascript)
  - SRC: file esterno con il codice di script
  - Esempio:
- `<SCRIPT LANGUAGE="JavaScript" SRC="validazione.js">  
</SCRIPT>`

# JavaScript (segue)

- Se il tag `<SCRIPT>` non ha un attributo `SRC`, allora il JavaScript è direttamente nel file HTML
- Esempio:  
`<SCRIPT LANGUAGE="JavaScript">`  
    `<!--alert("Benvenuto nella nostra libreria")`  
    `//-->`  
    `</SCRIPT>`
- Due diversi stili di commento:
  - `<!--` commento per HTML, poiché il codice JavaScript che segue dovrebbe essere ignorato dall'elaboratore HTML
  - `//` commento per JavaScript allo scopo di chiudere il commento HTML



# JavaScript (segue)

- JavaScript è un linguaggio di scripting completo
  - Variabili
  - Assegnazioni (`=`, `+=`, ...)
  - Operatori di confronto (`<`, `>`, ...) operatori booleani (`&&`, `||`, `!`)
  - Comandi
    - `If (condizione) {comandi;} else {comandi;}`
    - Cicli `for`, cicli `do-while` e cicli `while`
  - Funzioni con restituzione di valori
    - Si creano funzioni usando la parola chiave `function`
    - `Function F(arg1, ..., argk) {comandi;}`

# JavaScript: un esempio completo

## Modulo HTML :

```
<form method="POST"
  action="tmp.html"
  id="LoginForm">
  <input type="text"
    name="userid">
  <input type="password"
    name="password">
  <input type="submit"
    value="Login"   name="Invia"
    onClick="controllaLoginVuoto()"
  >
  <input type="reset"
    value="Reimposta">
</form>
```

## JavaScript associato :

```
<script>
function controllaLoginVuoto() {
  loginForm =
    document.getElementById("LoginForm")
  if ((loginForm.name.value == "") ||
    (loginForm.password.value == ""))
  {
    alert("Immettere un valore per userid e
      password") ;
    return false;
  }
  else return true;
}
</script>
```

# Fogli di stile

- Idea: separare la visualizzazione dal contenuto e adattarla a differenti formati di presentazione
- Due aspetti:
  - le trasformazioni del documento decidono quali parti del documento visualizzare, e in quale ordine
  - “Spezzettamento” del documento per decidere come ciascuna parte deve essere visualizzata
- Perché usare i fogli di stile?
  - Riutilizzo dello stesso documento per visualizzazioni differenti
  - Adattamento della visualizzazione alle preferenze dell’utente
  - Riutilizzo dello stesso documento in contesti diversi
- Due linguaggi per i fogli di stile
  - Fogli di stile ad albero (CSS): per documenti HTML
- Extensible stylesheet language (XSL): per documenti XML

# CSS: fogli di stile ad albero

- Definiscono come devono essere visualizzati i documenti HTML
- Molti documenti HTML possono far riferimento allo stesso CSS
  - Si può cambiare il formato di un sito web cambiando un singolo foglio di stile
  - Esempio

```
<LINK rel="foglio di stile" TYPE="text/css" HREF="libri.css"/>
```

- Ogni riga consiste di tre parti:

Selettore {proprietà: valore}

- Selettore: tag di formato definito
- Proprietà: attributo del tag il cui valore viene impostato
- Valore: valore dell'attributo

# CSS: fogli di stile ad albero (segue)

Esempio di foglio di stile:

```
body {background-color: yellow}
```

```
h1 {font-size: 36pt}
```

```
h3 {color: blue}
```

```
p {margin-left: 50px; color: red}
```

La prima riga ha lo stesso effetto di

```
<body background-color="yellow">
```

# IL LIVELLO INTERMEDIO

# Introduzione al livello intermedio

- **Richiamo: funzionalità del livello intermedio**
  - Codifica la logica dell'applicazione
  - Effettua le connessioni al/ai sistema/i di basi di dati
  - Riceve il testo immesso nei moduli al livello di presentazione
  - Genera i risultati per il livello di presentazione
- **Tecnologie**
  - **CGI**: protocollo per il passaggio di argomenti ai programmi in esecuzione al livello intermedio
  - **Application server**: ambienti di esecuzione al livello intermedio
  - **Servlet**: programmi Java al livello intermedio
  - **JavaServerPages**: script Java al livello intermedio
  - Mantenimento dello stato: come mantenere lo stato al livello intermedio. Argomento principale: **cookie**

# CGI: Common Gateway Interface

- Scopo: trasmettere argomenti dai moduli HTML ai programmi applicativi che vengono eseguiti al livello intermedio
- I dettagli del reale protocollo CGI non sono importanti -> le librerie implementano le interfacce ad alto livello
- Svantaggi:
  - Il programma dell'applicazione viene eseguito come un nuovo processo ad ogni invocazione (rimedio: FastCGI)
  - Non c'è condivisione di risorse tra i programmi applicativi (ad esempio connessioni a basi di dati)
  - Rimedio: application server



# CGI: esempio

- Modulo HTML:

```
<FORM ACTION="trovaLibri.cgi" METHOD="POST">
```

Inserisci il nome di un autore:

```
<INPUT TYPE="text" NAME="nomeAutore">
```

```
<INPUT TYPE="submit" VALUE="Invia">
```

```
<INPUT TYPE="reset" VALUE="Cancella">
```

```
</FORM>
```

- Codice Perl:

```
use CGI;
```

```
$dataIn=new CGI;
```

```
$dataIn->header();
```

```
$authorName=$dataIn->param('nomeAutore');
```

```
print("<HTML><TITLE>Prova di passaggio di argomenti</TITLE>");
```

```
print("Il nome dell'autore è " + $authorName);
```

```
print("</HTML>");
```

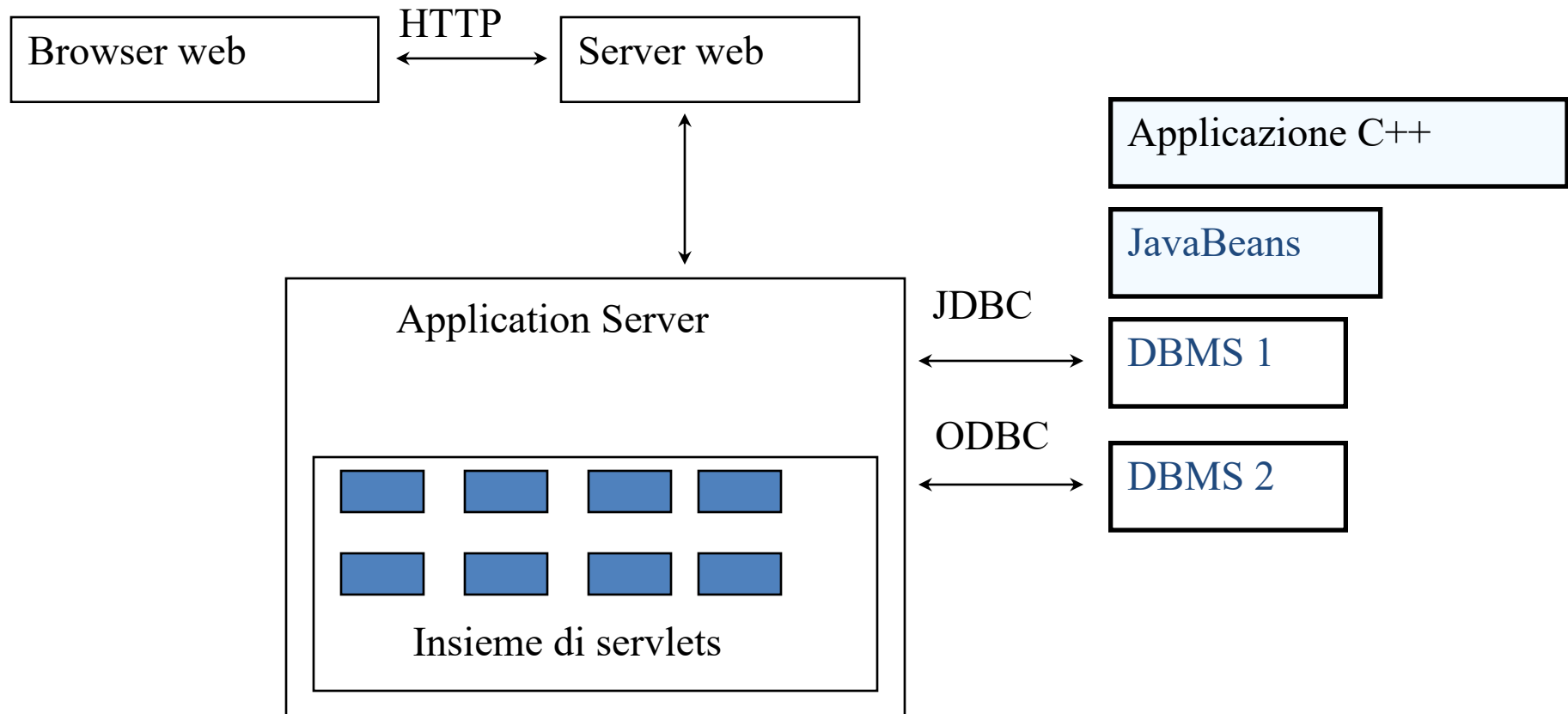
```
exit;
```

# Application Server

- **Idea: evitare il sovraccarico delle CGI**
  - Insieme principale dei thread dei processi
  - Gestisce le connessioni
  - Consente l'accesso a sorgenti di dati eterogenee
  - Altre funzionalità quali API per la gestione delle sessioni

# Application server: struttura dei processi

## Struttura dei processi



# Servlet

- Java Servlets: codice Java che viene eseguito al livello intermedio
  - Indipendente dalla piattaforma
  - API Java completamente disponibile, incluso JDBC

Esempio :

```
import java.io.*;
import java.servlet.*;
import java.servlet.http.*;
public class ScheletroDiServlet extends HttpServlet {
    public void doGet(HttpServletRequest richiesta,
                        HttpServletResponse risposta)
        throws ServletException, IOException {
        PrintWriter out=risposta.getWriter();
        out.println("Ciao mondo");
    }
}
```

# Servlet (segue)

- Vita di un servlet?
  - Il server web inoltra la richiesta al contenitore del servlet
  - Il contenitore crea una istanza del servlet (chiama il metodo `init()`; al momento della deallocazione: chiama il metodo `destroy()`)
  - Il contenitore chiama il metodo `service()`
    - `Service()` chiama `doGet()` per il GET HTTP o il `doPost()` per il POST HTTP
    - Di solito `service()` non viene sovrascritto, ma vengono sovrascritti `doGet()` e `doPost()`

# Servlet: un esempio completo

```
public class LeggiNomeUtente extends HttpServlet {
    public void doGet( HttpServletRequest richiesta,
                      HttpServletResponse risposta)
        throws ServletException, IOException {
        risposta.setContentType("text/html");
        PrintWriter out=risposta.getWriter();
        out.println("<HTML><BODY>\n <UL> \n" +
            "<LI>" + richiesta.getParameter("userid") + "\n" +
            "<LI>" + richiesta.getParameter("password") + "\n" +
            "<UL>\n<BODY></HTML>");
    }
    public void doPost( HttpServletRequest richiesta,
                      HttpServletResponse risposta)
        throws ServletException, IOException {
        doGet(richiesta, risposta);
    }
}
```

# Java Server Pages

- **Servlet**
  - Generano HTML scrivendolo sull'oggetto **`PrintWriter`**
  - Prima il codice, poi la pagina web
- **JavaServerPages**
  - Codice scritto in HTML, simile al codice dei servlet, incapsulato nell'HTML
  - Prima la pagina web, poi il codice
  - Di solito sono compilate in un servlet

# JavaServerPages: esempio

```
<html>
<head><title>Benvenuto alla B&N</title></head>
<body>
  <h1>Bentornato!</h1>
  <% String name="NuovoUtente";
      if (request.getParameter("UserName") != null) {
          nome=request.getParameter("UserName");
      }
  %>
  Sei connesso come <%=nome%>
  <p>
</body>
</html>
```



# Mantenimento dello stato

- L'HTTP è senza memoria
- Vantaggi
  - Facile da usare: non c'è bisogno di nulla
  - Ottimo per applicazioni con informazioni statiche
  - Non richiede spazio extra in memoria
- Svantaggi
  - Niente registrazione delle richieste precedenti significa
    - Niente carrelli per la spesa
    - Niente login degli utenti
    - Nessun contenuto personalizzato o dinamico
- Maggiore difficoltà di implementazione della sicurezza

# Stato delle applicazioni

- Stato sul lato server
  - L'informazione è memorizzata in una base di dati, o nella memoria locale dello strato applicativo
- Stato sul lato client
  - L'informazione è memorizzata sul computer client sotto forma di cookie
- Stato nascosto
  - L'informazione è nascosta in pagine web create dinamicamente

# Stato delle applicazioni

Così tanti tipi di stati...  
quale scegliere ?



# Stato sul lato server

- Molti tipi di stato sul lato server:
  - 1. Mantenimento delle informazioni in una base di dati
    - I dati sono al sicuro nella base di dati
    - MA: richiede un accesso alla base di dati per interrogare o aggiornare le informazioni
  - 2. Uso della memoria locale del livello dell'applicazione
    - Possibile mappare l'indirizzo IP dell'utente in qualche stato
    - MA: questa informazione è volatile e impiega parecchia della memoria principale del server
- 5 milioni di IP = 20 MB

# Stato sul lato server (segue)

- Si dovrebbe usare il mantenimento dello stato sul lato server per le informazioni persistenti
  - Vecchi ordini del cliente
  - Memorizzazione dei movimenti di un utente in un sito
  - Scelte permanenti fatte dall'utente

# Stato sul lato client: cookie

- Memorizzare sul client del testo che verrà passato all'applicazione con ogni richiesta HTTP.
  - Possono essere disabilitati dal client
  - Sono erroneamente percepiti come “pericolosi”, e quindi potenziali visitatori del sito saranno spaventati dalla richiesta di abilitare i cookie!
- Sono una collezione di coppie (Nome, Valore)

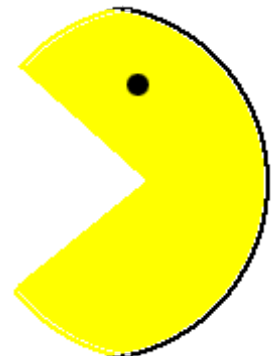
## Stato sul lato client: cookie (segue)

- Vantaggi
  - Facilità d'uso in servlet Java / JSP
  - Forniscono un modo semplice per mantenere dati non essenziali sul client anche quando il browser è chiuso
- Svantaggi
  - Limite di 4 KB di informazione
  - Gli utenti possono (e spesso lo fanno) disabilitarli
- Si dovrebbero usare i cookie per memorizzare lo stato interattivo
- Informazioni sul login dell'utente corrente
- Carrello della spesa corrente
- Qualunque scelta non permanente fatta dall'utente

# Creare un cookie

```
Cookie mioCookie =  
    new Cookie("nomeutente", "jeffd");  
response.addCookie(mioCookie);
```

- Si può creare un cookie in qualunque momento





# Accedere ad un *cookie*

```
Cookie[] cookies = request.getCookies();  
String Utente;  
for(int i=0; i<cookies.length; i++) {  
    Cookie cookie = cookies[i];  
    if(cookie.getName().equals("username"))  
        Utente = cookie.getValue();  
}  
  
// a questo punto Utente == "username"
```

- Si deve accedere ai cookie PRIMA di impostare l'intestazione della risposta:

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

# Caratteristiche dei cookie

- I *cookie* possono avere
  - Una durata (scadono immediatamente oppure persistono anche dopo che il browser è stato chiuso)
  - Filtri per stabilire a quali domini/cartelle il cookie viene spedito
- Maggiori informazioni nei manuali Java Servlet API e Servlet

# Stato nascosto

- Spesso gli utenti disabilitano i cookie
- Si possono “nascondere” i dati in due posti:
  - Campi nascosti all’interno di un modulo
  - Usando le informazioni sul percorso
- Non richiede “memorizzazione” di informazioni perché le informazioni sullo stato sono passate all’interno di ciascuna pagina web

# Stato nascosto: campi nascosti

- Dichiarare campi nascosti all'interno di un modulo:
  - `<input type='hidden' name='utente' value='nomeutente'/>`
- Gli utenti non vedranno queste informazioni (a meno che non guardino il codice HTML)
- Se usati in quantità, sono micidiali per le prestazioni poiché OGNI pagina deve essere contenuta all'interno di un modulo

# Stato nascosto: informazioni sul percorso

- Le informazioni sul percorso sono memorizzate nella richiesta dell'URL:  
<http://server.com/index.htm?utente=jef fd>
- Si possono separare i “campi” con un carattere &:  
`index.htm?utente=jef fd&preferenza=pepsi`
- In Java ci sono meccanismi per analizzare questo campo. Si rimanda al metodo

```
javax.servlet.http.HttpUtils  
    parserQueryString()
```

# Metodi multipli per lo stato

- Tipicamente vengono usati tutti i metodi di mantenimento dello stato:
  - L'utente effettua il login e questa informazione viene memorizzata in un cookie
  - L'utente effettua una interrogazione che viene memorizzata nelle informazioni sul percorso
  - L'utente inserisce un oggetto in un cookie per il carrello della spesa
  - L'utente compra degli oggetti e le informazioni sulla carta di credito vengono memorizzate su/lette da una base di dati
  - L'utente esegue una sequenza di click che viene mantenuta in un registro sul server web (e che può essere analizzata in seguito)