Anno Accademico 2007-2008

Corso di Tecnologie Web

Web Application: Servlet

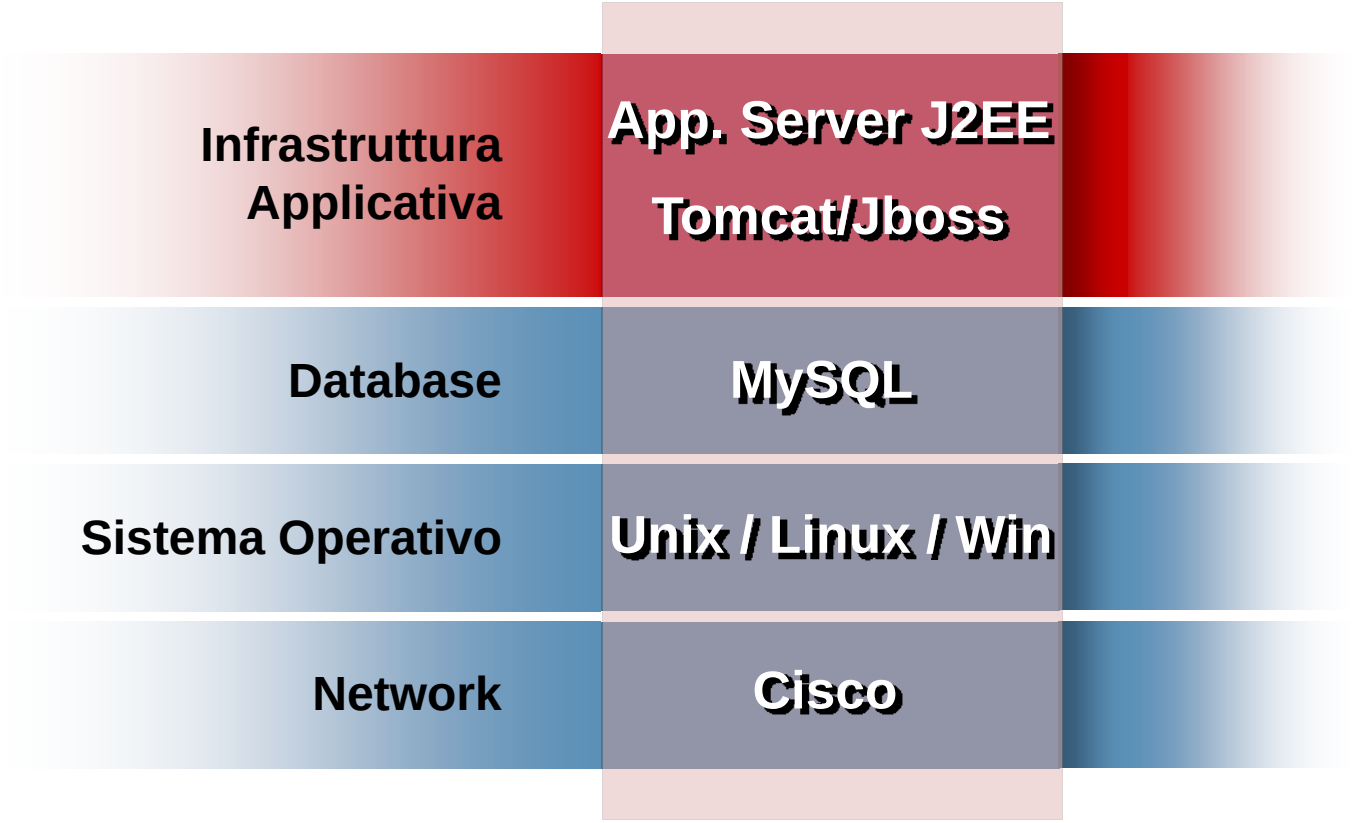
http://www-lia.deis.unibo.it/Courses/TecnologieWeb0708/

Requisiti applicazioni di classi Enterprise

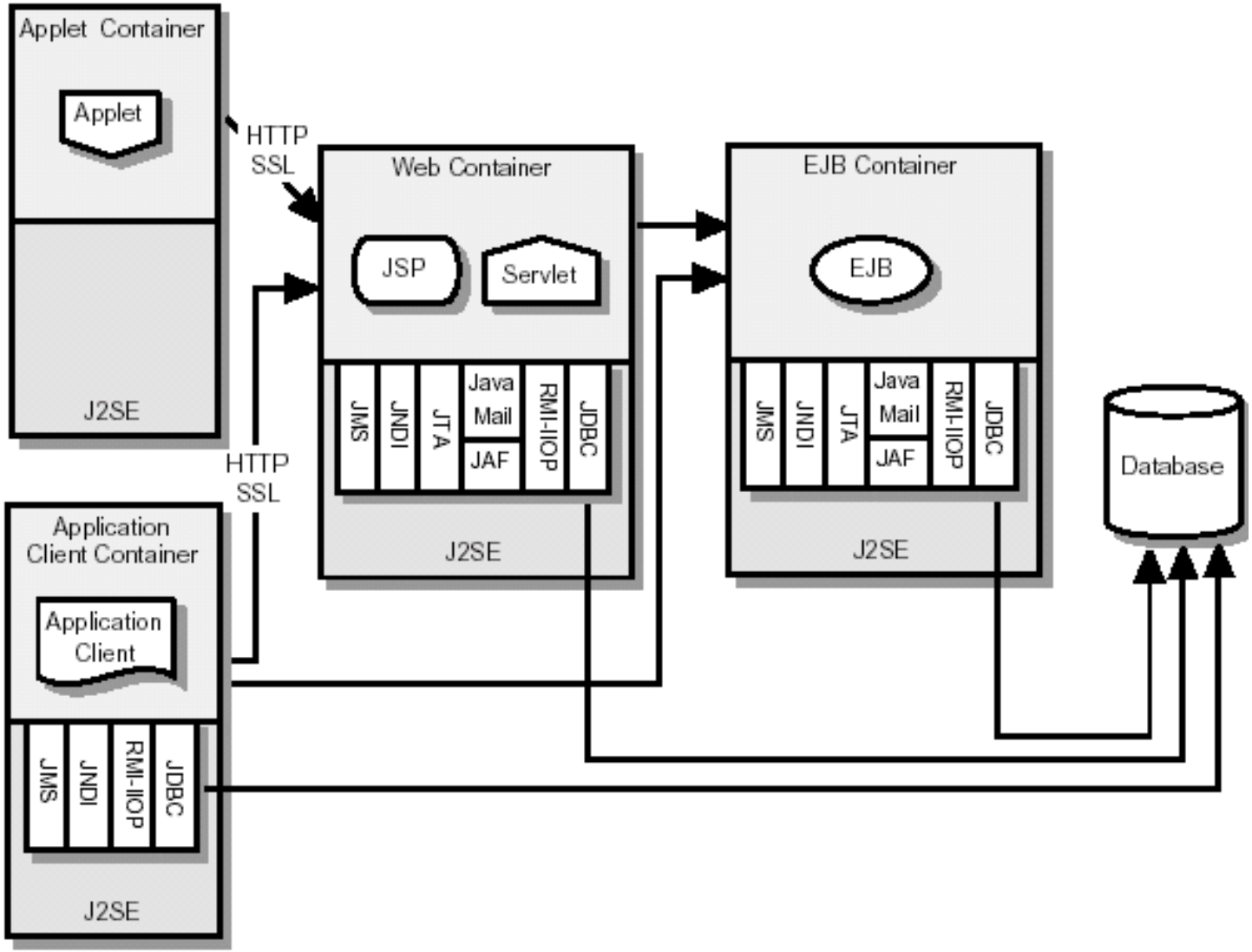
- Indipendenza dalla piattaforma
- Architettura multi-tier
- Architettura orientata ai componenti
- Controllo della concorrenza
- Gestione Autenticazione/Autorizzazione
- Naming
- Transazioni
- Load balancing e fault tolerance
- Sicurezza

Un’infrastruttura distribuita

> Key decision point



L'architettura J2EE



Web Client

> I Web Client hanno sostituito, in molte situazioni, i più tradizionali “fat client”

> I Web Client:

- sono accessibili via browser
- comunicano via HTTP e HTTPS con il server (il browser è, tra le altre cose, un client HTTP)
- effettuano il *rendering* della pagina in HTML (o altre tecnologie *mark-up* come, per esempio, XML e XSL)
- possono essere sviluppati utilizzando varie tecnologie (tra cui J2EE)
- sono spesso implementati come parti di architetture multi-tier

J2EE Web Application e Web Container

> Una **Web Application** è un gruppo di risorse *server-side* che creano una applicazione interattiva *online*.

> Le risorse server-side includono:

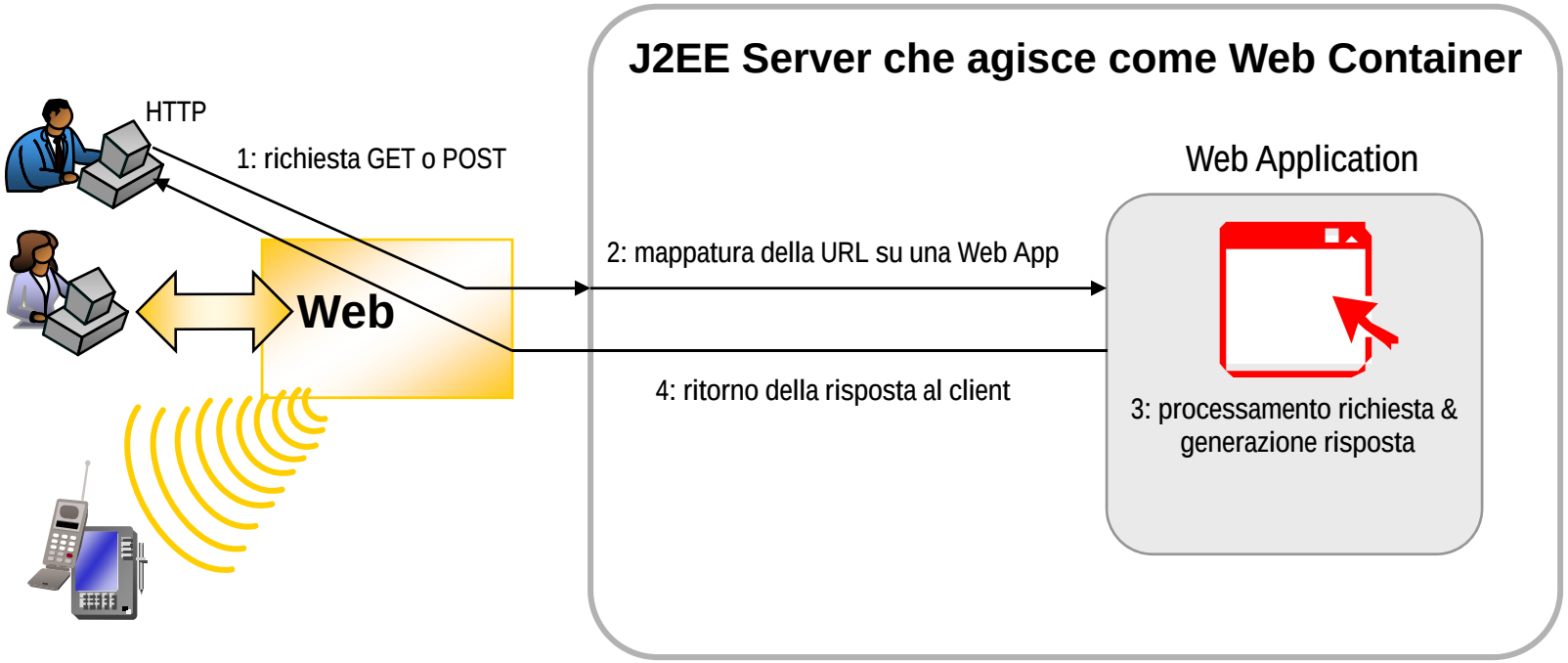
- Java Server Pages
- classi server-side (Servlet e classi std Java)
- risorse statiche (HTML, immagini, css, javascript, ecc.)
- Applet e/o altri componenti attivi *client-side*
- informazioni di configurazione e *deployment*

> I **Web Container** forniscono un ambiente di esecuzione per le **Web Application**.

> I **Container** garantiscono servizi di base alle applicazioni sviluppate secondo un paradigma a *componenti*.

Accesso ad una Web Application

> L'accesso ad una Web Application è un processo multi-step:



Servlet e JSP

> Servlet:

- sono classi Java che forniscono risposte a richieste HTTP (più precisamente sono classi che forniscono un servizio comunicando con i client mediante protocolli *request/response*. Tra questi protocolli il più noto e diffuso è HTTP).
- estendono le funzionalità di un web server generando contenuti dinamici *programmaticamente*
- eseguono direttamente in un Web Container

> Java Server Pages:

- consentono di separare la logica di navigazione (*business flow*) dalla presentazione
- rappresentano *template* per contenuto dinamico
- estendono HTML con codice Java custom
- sono compilate in servlet dal server

Il Deployment e gli Archivi Web

> Il Deployment, per una Web Application, è il processo di:

- definizione del *run time environment* di una Web Application
- mappatura delle URL su servlet e JSP
- definizione delle impostazioni di default di un'applicazione; per esempio: *welcome page* e *error pages*
- configurazione dei vincoli di sicurezza dell'applicazione

> Gli Archivi Web (*Web Archives*) sono file con estenzione "**.war**". Essi rappresentano la modalità con cui avviene la *distribuzione* delle Web Application. Sintassi: (es: **`jar –cvf newArchive.war myWebApp\)`**

```
jar {ctxu} [vf]  [jarFile]  files
```

-ctxu: create, get the table of content, extract, update content

-v: verbose

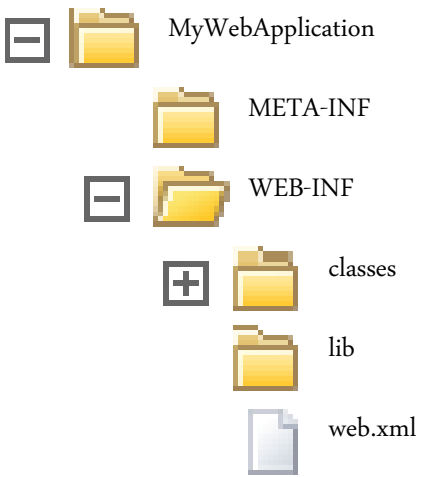
-f: il JAR file sarà specificato con `jarFile` **option**

-jarFile: nome del JAR file

-files: lista separata da spazi dei file da includere nel JAR

Web Application Layout

> La struttura di directory delle Web Application è basata sulle *Servlet 2.4 specification*

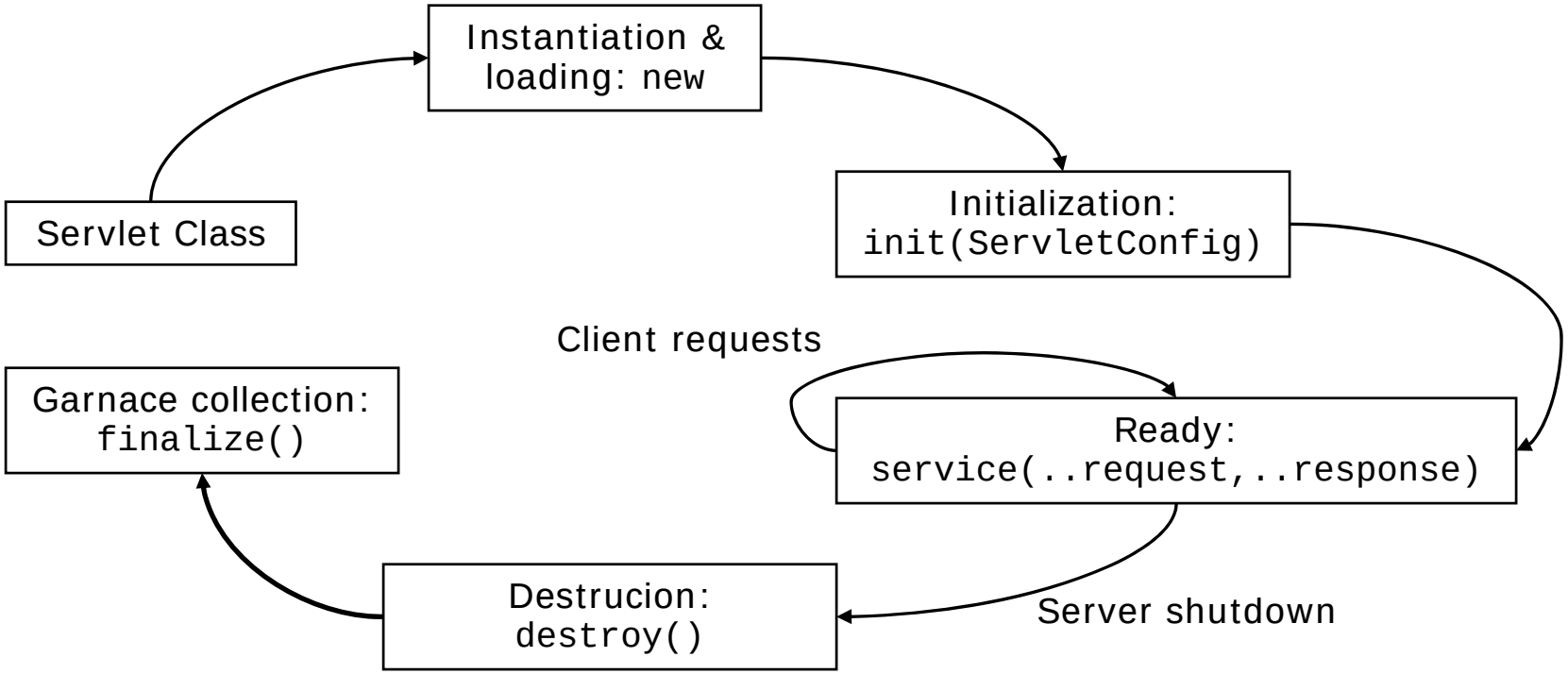


Root della Web Application
Informazioni per i tool che generano archivi (manifest)
File privati (config) che non saranno serviti ai client
Classi server side: servlet e classi Java std
Archivi .jar usati dalla web app
Web Application deployment descriptor

> Molti Application Server prevedono un descriptor file proprietario (oltre a web.xml previsto da specifiche).

Servlet: il ciclo di vita

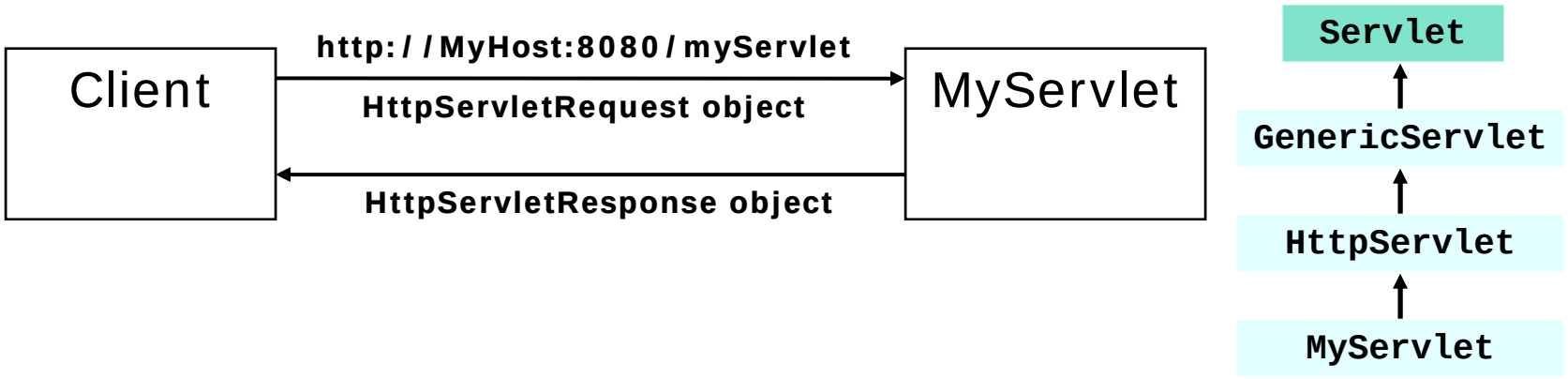
> Il ciclo di vita delle servlet è controllato dal servlet container



HTTP Servlet

> Le *servlet* sono classi Java che processano richieste specifiche *protocol based*

> Le *servlet* HTTP sono il tipo più comune di servlet e possono processare richieste HTTP.




```
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
  public void service(HttpServletRequest req, HttpServletResponse res) {
     …process request, generate response
  }
}
```

L'interfaccia HttpServletRequest

> L'interfaccia **HttpServletRequest** rappresenta la richiesta da parte di un'applicazione client (modella la richiesta del protocollo HTTP)

```
public void service(HttpServletRequest req, HttpServletResponse res) {
     Cookie[] cookies = req.getCookies();
     HttpSession session = req.getSession();
     Enumeration headers = req.getHeaders();
     String qty = req.getParameter("quantity");
     String rHost = req.getRemoteHost();
}
```

L'interfaccia `HttpServletResponse`

> L'interfaccia **`HttpServletResponse`** rappresenta la risposta da parte di una servlet (modella la risposta del protocollo HTTP)

```
public void service(HttpServletRequest req, HttpServletResponse res) {
     PrintWriter out = res.getWriter();
     …
     …
     res.addCookie(new Cookie("name", "value"));
     out.println("Clicca su: " + res.encodeURL("link"));
     res.addHeader("headerName", "value"):
}
```

Processamento delle richieste in GET o POST

> Quando una servlet riceve una richiesta in GET il corrispondente metodo doGet( ) viene invocato

```
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse res) {
     …process GET request, generate response
 }
}
```

> Quando una servlet riceve una richiesta in POST il corrispondente metodo doPost( ) viene invocato

```
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
 public void doPost(HttpServletRequest req, HttpServletResponse res) {
     …process POST request, generate response
 }
}
```

InputStream, OutputStream e PrintWriter

> L'esempio illustra l'utilizzo degli stream

```
protected void doPost(HttpServletRequest request,HttpServletResponse response) {
  try {
     PrintWriter out = response.getWriter();
     Enumeration e = request.getHeaderNames();
     StringBuffer message = new StringBuffer();
     message.append("<html>\n<body>");
     String httpRequestBody = "Questo è il contenuto del body del pacchetto: ";
     message.append(httpRequestBody);
     message.append(getBytes(request.getInputStream())+ "<br />\n");
     while (e.hasMoreElements()) {
       String hName = (String) e.nextElement();
       Enumeration e1 = request.getHeaders(hName);
       while (e1.hasMoreElements()) {
         String headerValue = (String) e1.nextElement();
         message.append("Header: ");
         message.append(hName).append(" = ").append(headerValue).append("<br/>\n");
       }
     }
     message.append("</body>\n</html>");
     out.println(message.toString());
  } catch (IOException e) {e.printStackTrace();}
}
```

Processamento delle richieste con il metodo `service()`

> Se non ne viene fatto l'*override*, il metodo `service` effettua il *dispatch* delle richieste ai metodi `doGet, doPost` e agli altri metodi di gestione delle richieste a seconda del metodo HTTP definito dalla *request*.

> Si può effettuare l'*override* del metodo `service` se si intendono trattare tutti i tipi di richiesta allo stesso modo. In questo caso, il metodo `service` processa direttamente la richiesta.

```
public void service(HttpServletRequest req, HttpServletResponse res) {
     int reqId = getReqIdFromReq(req);
     switch(reqId) {
        case REQ1 : handleReq1(req, res); break; //metodo di gestione 1
        case REQ2 : handleReq2(req, res); break; //metodo di gestione 2
        default   : handleReqUnknown(req, res);  //metodo di default
     }
}
```

Processamento Query String delle URL

> La *query string* consiste in un set di coppie *chiave=valore* dopo il **?** Nella URL (es: `http://MyHost:8080/myServlet`**`?P1=V1&P2=V2`**)

> `HttpServletRequest` fornisce metodi per processare la *query string*

```
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
 public void doGet(HttpServletRequest req, HttpServletResponse res) {
   PrintWriter out = res.getWriter();
   out.println(req.getParameter("P1"));
   java.util.Enumeration pNames = req.getParameterNames();
   while (pNames.hasMoreElements())
     out.println((String)pNames.nextElement());
   String[] pValues = req.getParameterValues("P2");
   for(int j=0; j<pValues.size(); j++)
     out.println(pValues[j]);
 }
}
```

Processamento dei FORM HTML

> I Form dichiarano i campi utilizzando l'attributo *name*

>Quando il FORM viene inviato al server, il nome dei campi e i loro valori sono inclusi nella request:

- agganciati alla URL come query string (GET)
- inseriti nel body del pacchetto HTTP (POST)

```
<form action="myServlet.surf" method="post">
        First name: <input type="text" name="firstname"/><br/>
        Last name: <input type="text" name="lastname"/>
</form>
```

```
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
 public void doPost(HttpServletRequest req, HttpServletResponse res) {
   String firstname = req.getParameter("firstname"));
   String lastname = req.getParameter("lastname"));
 }
}
```

Dichiarazione e configurazione di una servlet

> Il file **web.xml** (*deployment descriptor*) viene utilizzato per registrare e configurare le servlet.

```
<web-app>
        <servlet>
                <servlet-name>myServlet</servlet-name>
                <servlet-class>myPackage.MyServlet</servlet-class>
        </servlet>
        <servlet-mapping>
                <servlet-name>myServlet</servlet-name>
                <url-pattern>*.surf</url-pattern>
        </servlet-mapping>
</ web-app >
```

Esempi di URL che verranno mappati su myServlet

```
http://MyHost:8080/MyWebApplication/insUser.surf
http://MyHost:8080/MyWebApplication/ins.surf
```

Servlet context

> Tutte le web application eseguono in un determinato contesto e vi è una corrispondenza 1-to-1 tra una specifica web-app e il suo contesto.

>*ServletContext* è l'interfaccia che rappresenta la vista della web application (del suo contesto) da parte della servlet

> Un'istanza di *ServletContext* è ottenuta all'interno della servlet utlizzando il metodo `getServletContext()`

> Consente di accedere ai parametri di inizializzazione relativi al *context* e agli attributi di contesto

> Consente di accedere alle risorse statiche della web application mediante il metodo `getResourceAsStream(String path)`

> Il contesto viene condiviso tra i molteplici utenti, richieste, servlet della web abbplication

Parametri di inizializzazione del contesto

> I parametri di inizializzazione del contesto sono accessibili a tutte le servlet e JSP della web application

```
<web-app>
        <context-param>
                <param-name>feedback</param-name>
                <param-value>feedback@deis.unibo.it</param-value>
        </context-param>
</ web-app >
```

Accesso ai parametri di inizializzazione del contesto dalla servlet

```
…
ServletContext ctx = getServletContext();
String feedback = ctx.getInitParameter(“feedback”);
…
```

Attributi di contesto

> Gli attirbuti di contesto sono accessibili a tutte le servlet e JSP come variabili “globali”

>Il *binding* degli attributi di contesto viene fatto a *run time*

```
//setting degli attributi di contesto
ServletContext ctx = getServletContext();
ctx.setAttribute(“utente1”, new User(“Giorgio Bianchi”));
ctx.setAttribute(“utente2”, new User(“Paolo Rossi”));
…
//getting degli attributi di contesto
ServletContext ctx = getServletContext();
Enumeration aNames = ctx.getAttributeNames();
while (aNames.hasMoreElements) {
  String aName = (String)aNames.nextElement();
  User user = (User) ctx.getAttribute(aName);
  ctx.removeAttribute(aName);
}
```

Servlet *configuration*

> L'interfaccia `ServletConfig` rappresenta la configurazione iniziale di una determinata servlet

> Si ottiene un riferimento ad un oggetto di tipo `ServletConfig` utilizzando il metodo `getServletConfig()`

> L'interfaccia `ServletConfig` fornisce metodi per ottenere informazioni quali il nome della servlet, il contesto della servlet, i parametri di inizializzazione della servlet, ecc.

> I parametri di configurazione possono essere gestiti nel metodo `init` specificato nell'interfaccia `Servlet`

Servlet *configuration*

Inizializzazione servlet in web.xml

```
<web-app>
        <servlet>
            <servlet-name>myServlet</servlet-name>
            <servlet-class>myPackage.MyServlet</servlet-class>
            <init-param>
                <param-name>feedback</param-name>
                <param-value>feedback@deis.unibo.it</param-value>
            </init-param>
        </servlet>
</ web-app >
```

Accesso ai parametri di inizializzazione della servlet

```
String feedback = null;
public void init(ServletConfig config) {
        feedback = config.getInitParameter("feedback");
}
```

Servlet e *multi-threading*

> Le servlet possono essere configurate e programmate per gestire richieste concorrenti oppure *single-threaded*

> Thread multipli che eseguono un'unica condivisa istanza di servlet possono condurre a stati inconsistenti. Le richieste concorrenti a una singola servlet possono essere sincronizzate per evitare *corse critiche.*

> Alternativamente, più istanze di servlet possono essere create per gestire richieste multiple. Questa eventualità, definita *Single-Threaded Model,* è tuttavia deprecata nelle specifiche 2.4 delle servlet. `SingleThreadModel` è l'interfaccia da implementare da parte delle servlet per ottenere il comportamento suddetto.

```
…Object synchObiect = new Object();
public void service(…request, …response) {
        synchronize (synchObiect ) {
          //solo un thread esegue questa sezione critica di codice
        }
}
```

Scoped objects

> Gli oggetti di tipo `ServletContext, HttpSession, HttpServletRequest` forniscono metodi per immagazzinare e ritrovare oggetti nei loro rispettivi *scope.*

> Lo *scope* è definito dal *lifespan* e dall'*accessibilità* da parte di JSP e servlet

Scope	**Lifespan**	**Accessibile da**
Request	Finché la risposta non viene inviata al client, lo scope è *vivo*.	Pagina/servlet corrente e ogni altra pagina inclusa o in *forward*.
Session	Lo stesso della sessione utente	Richiesta corrente ed ogni altra richiesta dallo stesso client.
Application	Il *lifespan* della Web Application	Richiesta corrente e ogni altra futura richiesta (alla stessa Web App) anche da client diversi

Storing & Retrieving oggetti dallo scope

> Gli oggetti *scoped* forniscono i seguenti metodi per immagazzinare e ritrovare oggetti (tipicamente bean) nel loro scope:

- `setAttribute(String, Object)`
- `Object getAttribute(String)`
- `Enumeration getAttributeNames()`

Java DataBase Connectivity

> JDBC è un API per accedere ai database in modo uniforme

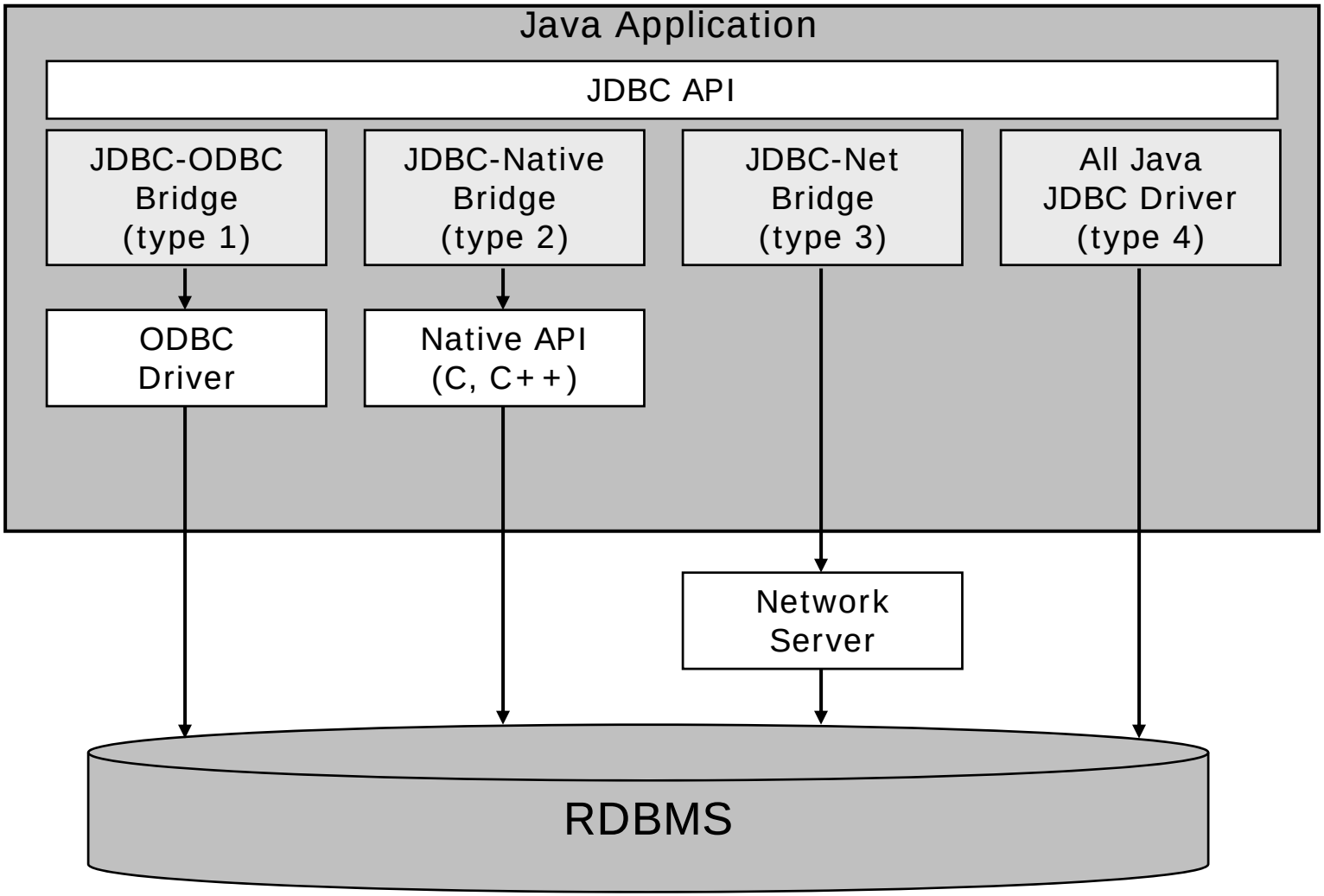
> JDBC garantisce accesso ai database in modo *platform independent*

> I driver JDBC sono collezioni di classi Java che implementano metodi definiti dalle specifiche JDBC

> I driver possono essere suddivisi in due categorie:

- 2-tier, in cui i client colloquiano direttamente con il DB
- 3-tier, in cui i client comunicano con un middle-tier che accede al DB

Architettura JDBC



Accesso diretto al DB con JDBC

> L'accesso diretto al DB consiste di:

- caricamento della classe del driver JDBC
- ottenimento della Connection dal driver e suo utilizzo

```
import java.sql.*;
…
try {
   Class.forName("org.hsqldb.jdbcDriver");
   Connection conn =
        DriverManager.getConnection("jdbc:hsqldb:hsql://localhost:1701");
   Statement stm = conn.createStatement();
   String sql = "Select * from MYTABLE";
   ResultSet res = stm.executeQuery(sql);
   while (res.next()) {
      String col1 = res.getString("MYCOL1");
      int col2 = res.getInt("MYCOL2");
   }
} catch (Exception e) {…}
```

Connection Pool

> I Connection Pool sono oggetti -amministrati dall'application server- preposti a gestire le connessioni verso DB

> Sono configurabili attraverso opportuni *config* file. Negli application server prodotti da *vendor* commerciali (Oracle, IBM, BEA, ecc.), tali file sono gestiti da *console* d'amministrazione.

> Il vantaggio principale nell'utilizzo di Connection Pool risiede nel fatto che le connessioni sono esistenti quando l'applicazione necessita di connettersi a DB. Ciò garantisce all'applicazione di eliminare l'inevitabile *overhead* -che altrimenti ci sarebbe- dovuto alla creazione delle connessioni.

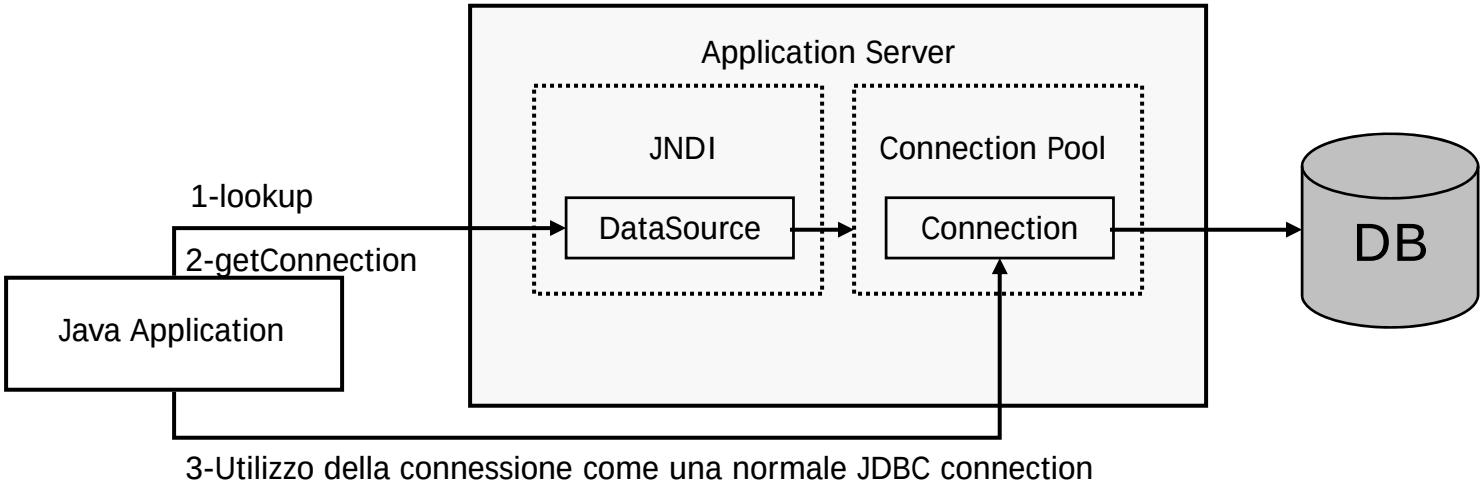
> L'application server può applicare un bilanciamento di carico alle applicazioni che usano un DB, assegnando o rilasciando connessioni alle applicazioni dipendentemente dalle necessità. Il *load balancing* può anche includere un incremento (*growing*) o riduzione (*shrinking*) del numero di connessioni nel pool al fine di adattarlo al cambiamento delle condizioni di carico.

Data Source

> I Data Source sono *factory* di connessioni verso sorgenti dati fisiche che un oggetto di tipo `javax.sql.DataSource` rappresenta.

> Oggetti di tipo `DataSource` vengono pubblicati su JNDI e configurati mediante opportuni file (o via console di amministrazione nel caso di vendor commerciali)

> Il Data Source è un *wrapper* di un connection pool



Accesso al DB usando Data Source

> Per accedere a DB via data source è necessario fare il *lookup* da JNDI ed ottenere dall'istanza di tipo `DataSource` una `Connection`.

```
// Contesto iniziale JNDI
Context initCtx = new InitialContext();
Context envCtx = (Context)initCtx.lookup("java:comp/env");

// Look up del data source
DataSource ds = (DataSource)envCtx.lookup("jdbc/EmployeeDB");

//Si ottiene una connessione da utilizzare come una normale
//connessione JDBC
Connection conn = ds.getConnection();
 ... uso della connessione come visto nell'esempio JDBC ...
conn.close();
```

Esempi di file di configurazione per i Data Source

STANDARD

/WEB-INF/ web.xml

```
<resource-ref>
        <description>
                Riferimento JNDI ad un data source
        </description>
        <res-ref-name>jdbc/EmployeeDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>

</resource-ref>
```

TOMCAT

/META_INF/context.xml

```
<Context ...>
...
        <Resource name="jdbc/EmployeeDB" auth="Container"
          type="javax.sql.DataSource“
          username="dbuser" password="dbpwd“
          driverClassName="org.hsql.jdbcDriver"
          url="jdbc:HypersonicSQL:database“
          maxActive="8" maxIdle="4"/>
...
</Context>
```

Gestione dello stato Client

> Il protocollo HTTP è un protocollo *stateless.* Esso non garantisce nativamente alle applicazioni server il mantenimento dello stato tra le diverse richieste provenienti dallo stesso client.

> Sono definite le seguenti tecniche per mantenere traccia delle informazioni di stato:

- uso dei cookie
- uso della sessione *(session tracking).* Essa può essere identificata:
  - da un cookie
  - utilizzando una tecnica detta *URL rewriting*

Cos'è un cookie

> Il cookie è un'unità di informazione che il web server deposita sul browser, cioè sul client

> Il cookie può contenere valori che sono propri del dominio funzionale dell'applicazione (in genere informazioni associate all'utente)

> I cookie, in quanto specifici header HTTP, sono trasferiti in formato testuale

> I cookie possono essere rifiutati dal browser (tipicamente perché disabilitati)

> I cookie sono spesso considerati un fattore di rischio

> Settando il tag `setSecure(true)` viene forzato il client a utilizzare un protocollo sicuro (HTTPS)

> I diversi browser possono utilizzare tecniche diverse per rendere persistenti le informazioni contenute nei cookie

La classe `Cookie`

> Un cookie contiene un certo numero di informazioni distinte. Tra queste:

- una coppia *name/value*
- il *dominio internet* dell'applicazione che ne fa uso
- Il *path* dell'applicazione
- una *expiration date* espressa in secondi (-1 indica che il cookie non sarà reso persistente)
- un valore booleano per definirne il livello di sicurezza

> La classe **`javax.servlet.http.Cookie`** modella il cookie HTTP.

> Si recuperano i cookie dalla request (HttpServletRequest) utilizzando il metodo **`getCookies()`**

> Si aggiungono cookie alla response (HttpServletResponse) utilizzando il metodo **`addCookie()`**

Esempi di utilizzo della classe Cookie

Retrieving cookies

```
Cookie[] cookies = request.getCookies();
if(cookies != null) {
        for(int j=0; j<cookies.length(); j++) {
           Cookie c = cookies[j];
           System.out.println("Un cookie: " + c.getName());
        }
}
```

Adding cookies

```
Cookie c = new Cookie("MyCookie", "test");
c.setSecure(true);
c.setMaxAge(-1);
c.setPath("/");
response.addCookie(c);
```

La sessione web

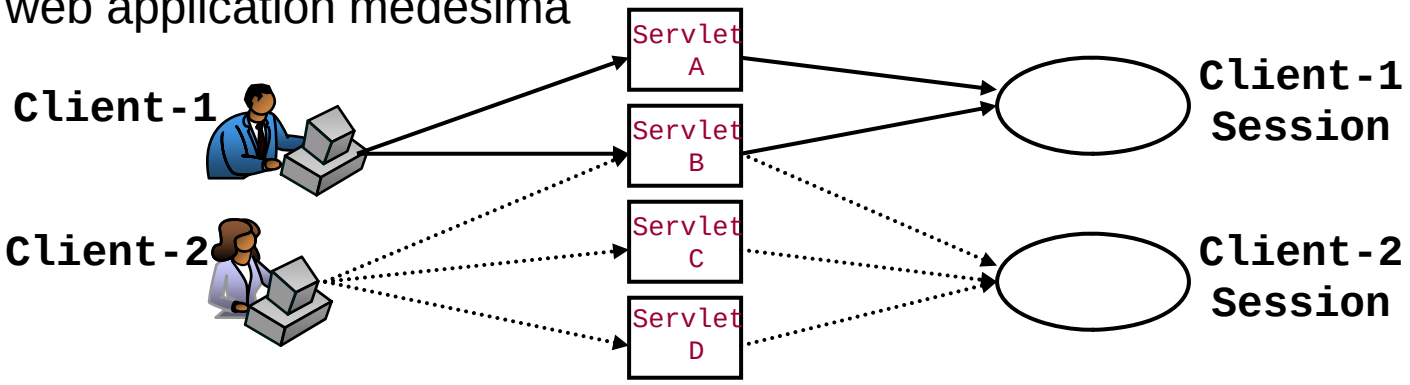
> La sessione web è un'entità gestita dal web container (in generale, in altre tecnologie, dal server web)

> La sessione web è condivisa fra tutte le richieste provenienti dallo stesso client (consente di mantenere, quindi, informazioni di stato)

> Rappresenta un meccanismo di *storage* per dati di varia natura usati dalle servlet e JSP

> Essa è idendificata da un unico *session ID*

> La sessione viene usata dai componenti di una web application per mantenere lo stato del client durante le molteplici interazioni dell'utente con la web application medesima



Recuperare l'oggetto `HttpSession`

> Per ottenere un riferimento all'oggetto `HttpSession` si usa il metodo `getSession()` dell'interfaccia `HttpServletRequest`

```
//definizione del metodo nell'interfaccia HttpServletRequest
public HttpSession getSession (boolean createNew);
//true – ritorna la sessione esistente o, se non esiste, ne
//crea una nuova
//false – ritorna, se possibile, la sessione esistente,
//altrimenti ritorna null


//uso del metodo
HttpSession session = request.getSession(true);
```

Getting/Setting/Removing dati dalla sessione

> Si possono memorizzare dati specifici dell'utente nell'oggetto `HttpSession` in termini di coppie *key/value*

```
Cart sc = (Cart)session.getAttribute("shoppingCart");
sc.addItem(item);
session.setAttribute("shoppingCart", new Cart());
session.removeAttribute("shoppingCart");
Enumeration e = session.getAttributeNames();
while(e.hasMoreElements())
   System.out.println("Key; " + (String)e.nextElements());

//altre operazioni con le sessioni
String sessionID = session.getId();
if(session.isNew())
   System.out.println("La sessione e' nuova");

//distruzione della sessione
session.invalidate();
System.out.println("Millisec:" + session.getCreationTime());
System.out.println(session.getLastAccessedTime());
```

URL *Rewriting*

> L'utilizzo dei cookie, come gestori dello stato client o come identificativi di sessione, fa ovviamente affidamento sull'abilitazione dei cookie sul browser.

> Un'alternativa all'utilizzo del cookie di sessione (il cookie che *trasporta* l'ID di sessione) è rappresentata dall'inclusione del *session ID* nella URL.

> Il ***session ID*** è usato per identificare le richieste provenienti dallo stesso utente (più precisamente dallo stesso client, cioè dallo stesso browser) e mapparle sulla corrispondente sessione.

> E' buona prassi effettuare sempre l'*encoding* delle URL generate dalle servlet e/o JSP. Il metodo `encodeURL()` di `HttpServletResponse` è a ciò preposto (`encodeRedirectURL()` viene usato nel caso di *redirection*)

> Il metodo `encodeURL()` dovrebbe essere usato per:

- hyperlink (<a href="…">)
- form (<form action="…">)
- servlet incluse e *forward* (servletContext.getRequestDispatcher(…))

URL *Rewriting*: la logica del metodo encodeURL()

> Il metodo encodeURL() determina se la URL debba essere riscritta. Se sì, include in essa l'identificativo di sessione *session ID*

> L'implementazione del metodo include la logica suddetta. Per esempio, nel caso in cui siano abilitati i cookie sul browser il metodo encodeURL() ritorna la URL invariata.

> Alla prima interazione di un client con la web application in cui venga richiesta la sessione, il metodo encodeURL(), non sapendo a priori se i cookie siano o meno abilitati, ritorna URL con incluso l'identificativo di sessione. Nelle interazioni successive il metodo valuterà la necessità di riscrivere le URL a seconda che sia o meno presente il cookie.

```
//1) utilizzo di encodeURL in una servlet
out.println("<a href='"+response.encodeURL("/myPrj/product")+
            "'>prodotti</a>");
//2) utilizzo di encodeRedirectURL
response.sendRedirect(response.encodeRedirectURL("…"));
```