Servlet

Tendenzialmente, l’esecuzione di una Servlet si basa su 6 punti fondamentali:

1. Il cliente invia una request al web server
2. Il web server riceve la request
3. Il web server passa la request alla corrispondente servlet
4. La servlet processa la request e genera una risposta
5. La servlet manda la risposta indietro al web servet
6. Il web server manda la risposta indietro al cliente e il client browser la mostra sullo schermo

Tutte le HTTP servlet sono costruite a partire dal package javax.servlet.http:

- HttpServlet (classe)
- HttpServletResponse (interface)
- HttpServletRequest (interface)

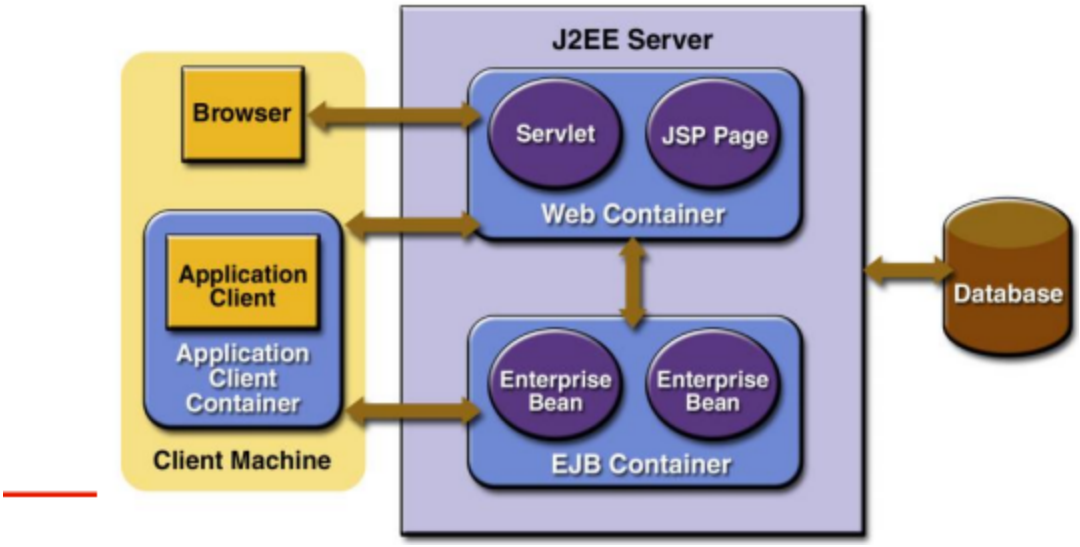
Gli oggetti di tipo Request rappresentano la chiamata al server effettuata dal client, e sono caratterizzati da varie informazioni quali:

- Chi ha effettuato la Request
- Quali parametri sono stati passati nella Request
- Quali header sono stati passati

Gli oggetti Respond rappresentano le informazioni restituite al client in risposta appunto ad una Request:

- Dati in forma testuale
- HTTP header cookie, ecc

Architettura JEE (*Java Enterprise Edition*)



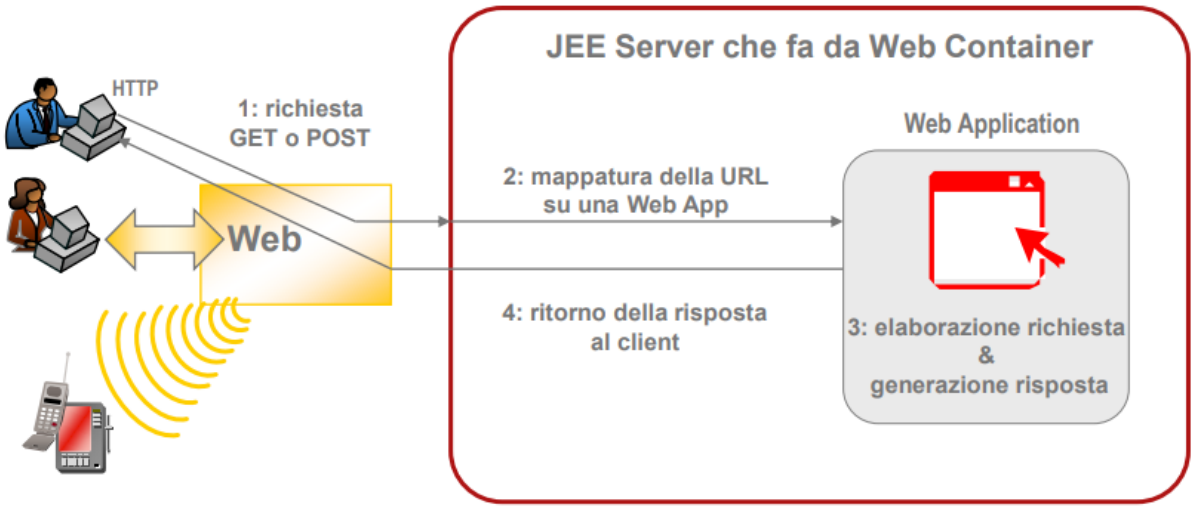
JEE = JSE + 2 container (Web e EJB) . Il Web Container include Servlet e JSP Page. EJB Container include Enterprise Bean che erano i componenti fondamentali per fare applicazioni industriali java (*oramai un po' datati*).

Una **Web Application** è un gruppo di risorse server-side che nel loro insieme creano una applicazione interattiva fruibile via Web. Le risorse Server-side includono:

- **Classi server-side**
- **Java server pages**: simili ad HTML
- **Risorse statiche**
- **Applet, Javascript**
- **Informazioni di configurazione**

I **Web Container** forniscono un ambiente di esecuzione per Web Application

Accesso a una Web Application



Una **Servlet** è una classe Java che fornisce risposte a richieste HTTP. *E' una classe che fornisce un servizio comunicando con il client mediante protocolli di tipo request/ response, tipo HTTP che è il più diffuso.*

Le Servlet estendono le funzionalità di un Web Server generando contenuti dinamici e superando i classici limiti delle applicazione CGI.

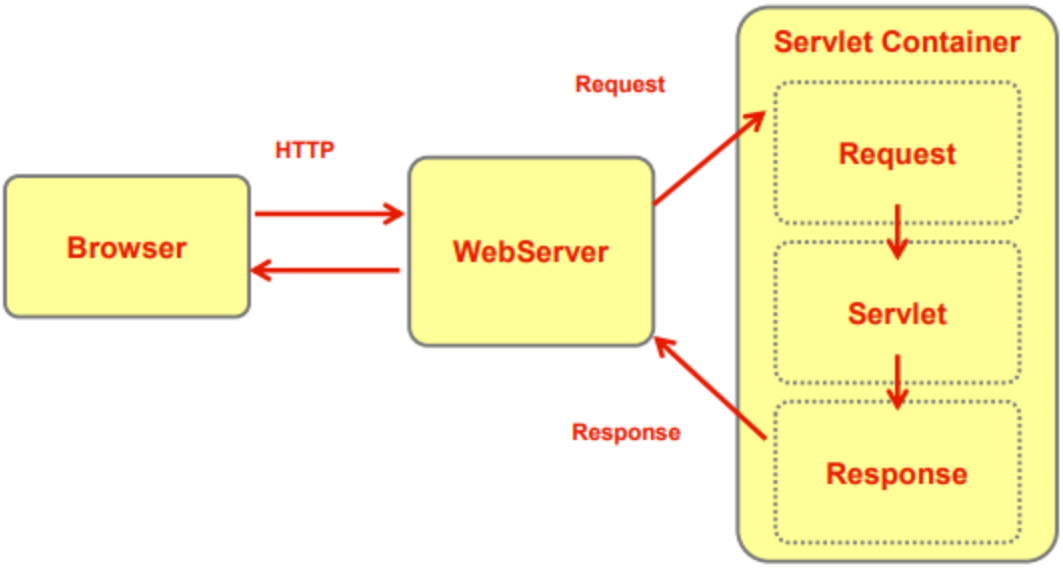
Dentro le classi che estendono HttpServlet possiamo fare override di alcuni metodi, ad esempio doGet:

```
public class HelloServlet extends HttpServlet{
 public void doGet(HttpServletRequest request, HttpServletRespon
  response.setContentType("text\html");
  PrintWriter out = response.getWriter();
  out.println("<title>Hello World!</title">;
  }
...
}
```

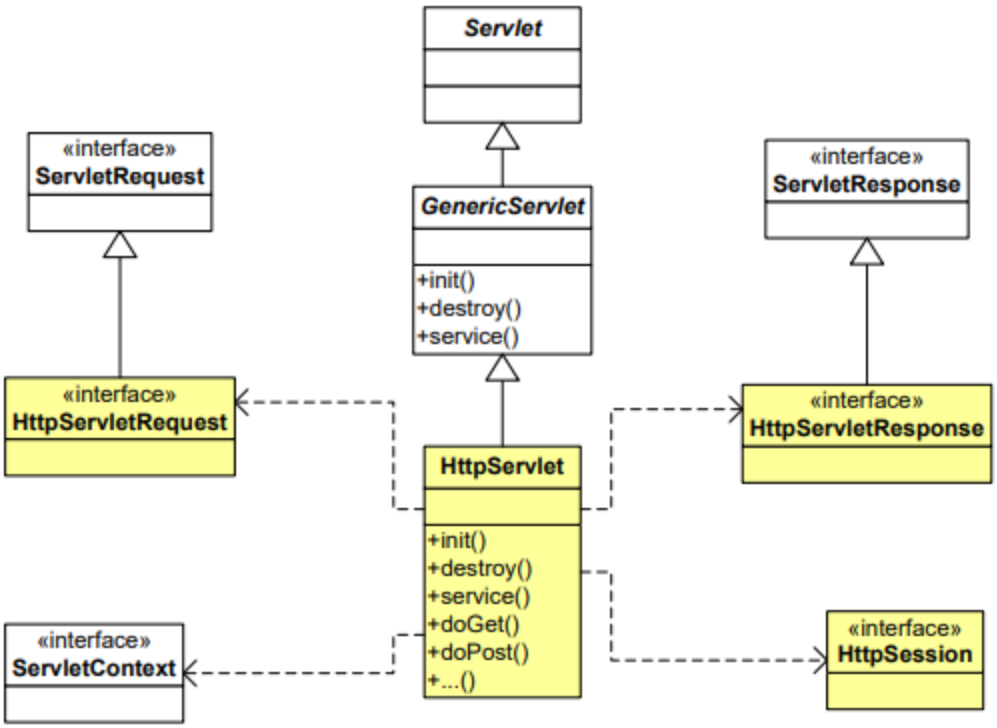
Una **Servlet** è dunque una semplice classe java, che viene realizzata facendo extends di HttpServlet. Questa classe è solo un ‘pezzo’ di ciò che esegue a runtime, e funziona solo perché dietro c’è un Web Container (*che è Tomcat nel nostro caso*).

Tutte le classi che ci interessano sono contenute nel package *javax.servlet.http.**

All’arrivo di ogni richiesta HTTP il servlet Container crea un oggetto request e un oggetto response e li passa alla servlet:



Gli oggetti di tipo Request rappresentano la chiamata al server effettuata dal client, gli oggetti di tipo Response rappresentano le informazioni restituite al client in risposta ad una request.



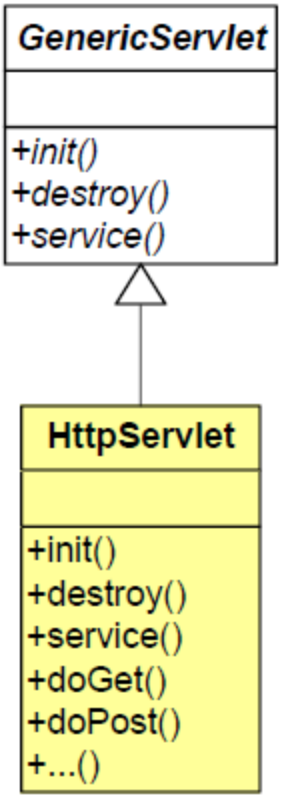
Il “modello normale” prevede una sola istanza di servlet e un thread assegnato ad ogni richiesta http per servlet, anche se richieste per questa servlet sono già in esecuzione.

Più thread condividono quindi la stessa istanza di una servlet e quindi si crea una situazione di concorrenza.

- Il metodo `init()` della servlet viene chiamato una sola volta quando la servlet è caricata dal Web container.
- I metodi `service()` e `destroy()` possono essere chiamati solo dopo il completamento dell’esecuzione di `init()`.
- Il metodo `service()` (*e quindi* `doGet()` *e* `doPost()`) può essere invocato da numerosi client in modo concorrente ed è quindi NECESSARIO gestire le sezioni critiche (*a carico di noi programmatori tramite blocchi synchronized, semafori e mutex*)

In alternativa si può indicare al container di creare un’istanza della servlet per ogni richiesta concorrente, cioè il modello diventa Single-Threaded; funziona nel seguente modo: il server container genera un’istanza della classe servlet per ogni richiesta che arriva.
cioè i thread sono multipli, ma ognuno ha la propria istanza

Metodi per il controllo del ciclo di vita



• **init()**: si inizializza l'istanza. Questo metodo viene chiamato una sola volta al caricamento della servlet.
•
service(): metodo di smistamento che invoca doGet() o doPost() a seconda del tipo di HTTP Request ricevuta. Viene chiamata ad ogni HTTP Request.
•
destroy(): serve per rilasciare le risorse acquisite, inizializzate in init(). Viene chiamato una sola volta quando la servlet deve essere disattivata.

Gestione Header e body HTTP

- `public void setHeader(String headerName, String headerValue)` imposta un header arbitrario
- `public void setDateHeader(String name, long millisecs)` imposta la data
- `public void setIntHeader(String name, int headerValue)` imposta un header con un valore intero (evita la conversione intero-stringa)
- `addHeader`, `addDateHeader`, `addIntHeader` aggiungono una nuova occorrenza di un dato header
- `setContentType` configura il content-type (**si usa sempre**)
- `setContentLength` utile per la gestione di connessioni persistenti
- `addCookie` consente di gestire i cookie nella risposta
- `sendRedirect` imposta location header e cambia lo status code in modo da forzare una ridirezione

Per definire il response body possiamo operare in due modi utilizzando due metodi di `response`

- `public PrintWriter getWriter`: mette a disposizione uno stream di caratteri (un'istanza di `PrintWriter`)
  - utile per restituire un testo nella risposta (tipicamente HTML)
- `public ServletOutputStream getOuputStream()`: mette a disposizione uno stream di byte (un'istanza di `ServletOutputStream`)
  - più utile per una risposta con contenuto binario (per esempio un'immagine)

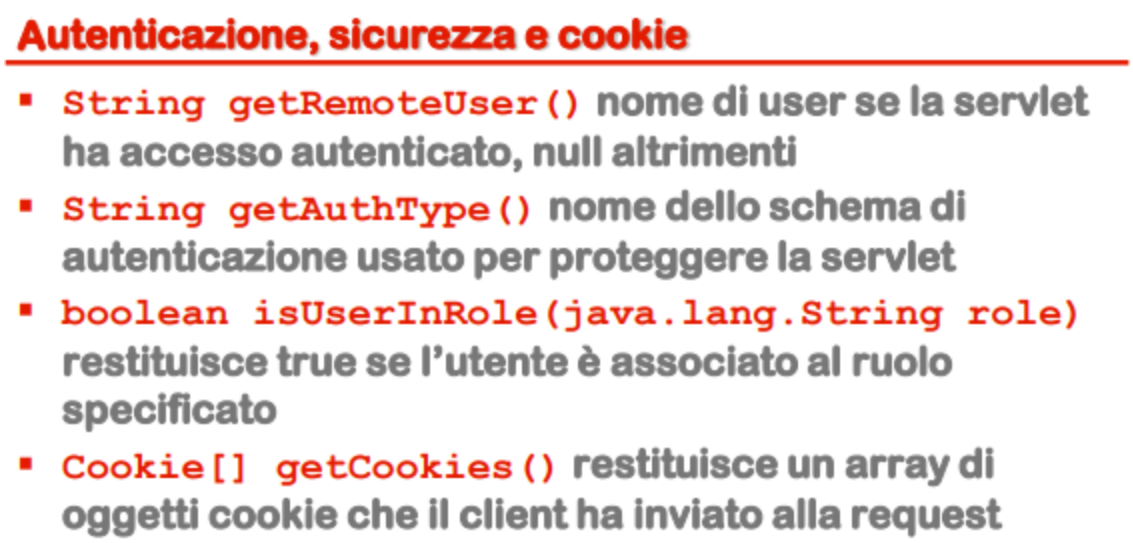
Accedere a Header, URL e sicurezza/ autenticazione

Metodi per accedere all'URL

- `String getParameter(String parName)` restituisce il valore di un parametro individuato per nome
- `String getContextPath()` restituisce informazioni sulla parte dell'URL che indica il contesto della Web application
- `String getQueryString()` restituisce la stringa di query
- `String getPathInfo()` per ottenere il path
- `String getPathTranslated()` per ottenere informazioni sul path nella forma risolta

Metodi per accedere agli header

- `String getHeader(String name)` restituisce il valore di un header individuato per nome sotto forma di stringa
- `Enumeration getHeaders(String name)` restituisce tutti i valori dell'header individuato da name sotto forma di enumerazione di stringhe (utile ad esempio per Accept che ammette n valori)
- `Enumeration getHeaderNames()` elenco dei nomi di tutti gli header presenti nella richiesta
- `int getIntHeader(name)` valore di un header convertito in intero
- `long getDateHeader(name)` valore della parte Date di header, convertito in long



Gestione dei form

Nelle JSP o nelle pagine HTML i form sono utilizzati per comunicare con le servlet. In particolare indichiamo con l'attributo **action** la servlet da lanciare all'invocazione del form e con l'attributo **method** il metodo (*doGet o doPost*) da utilizzare *(se non lo specifichiamo viene lanciata sempre in automatico la doGet)*.

Nella request ci sono tutti i dati "interessanti" (*agganciati all'URL come query string se GET, inseriti nel body del pacchetto HTTP se POST*). Con la funzione `getParameter("name")` recuperiamo il valore associato al rispettivo tag **name**.

Per **scrivere** su una nuova pagina web, a partire dalla servlet, utilizziamo l'oggetto **PrintWriter**:

```
PrintWriter out = response.getWriter();
out.println("something");
```

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

```
public class MyServlet extends HttpServlet
{
  public void doPost(HttpServletRequest rq, HttpServletResponse rs)
   {
     String firstname = rq.getParameter("firstname"));
     String lastname = rq.getParameter("lastname"));
   }
}
```

Redirection

Nel caso in cui vogliamo reinderizzare una richiesta, “collegando” più servlet l’una con l’altra, abbiamo due possibilità:

- **RequestDispatcher**
- **Redirection**

Qual è la differenza? **RequestDispatcher** non è “*trasparente*”: l’utente non si accorge che la servlet1 reindirizza una qualche operazione alle servlet2. L’URL non viene modificato. **Redirection** è invece “*trasparente*”: avviene un vero e proprio reindirizzamento ad un altra pagina, e l’URL cambia.

sostanzialmente però fanno la stessa cosa: inoltrano la request ad un altra servlet e la delegano a svolgere una qualche operazione. *RequestDispatcher deve essere utilizzato quando la seconda servlet è parte dello stesso sito web e appartiene allo stesso dominio.*

```
RequestDispatcher rd = req.getRequestDispatcher("nomeServlet");
rd.forward(req, res);



res.sendRedirect("nomeServlet");
```

Notiamo un’altra cosa: sendRidirect gestisce solo doGet, quindi la servlet che viene chiamata deve necessariamente implementare una doGet pena qualche errore. Ragion per cui non possiamo passare parametri aggiuntivi (*come vedremo tra poco*) se non utilizzando l’escamotage del “?”:

```
//ricollecandoci al codice di prima, supponendo di voler passare
//se RequestDispatcher
req.setAttribute("k", k);
//se Redirection
res.sendRedirect("nomeServlet?k="+k);            //più parametr


//nella servlet2, in entrambi i casi:
int k = (int) req.getAttribute("k");
```

Se ci pensiamo ha anche senso il perché la redirection non permette di passare nuovi parametri in chiaro: essendo appunto una redirezione a una pagina esterna dalla nostra applicazione, facente parte di un altro dominio, è giusto che questa non riceva alcun dato “*interno*”.

Http Session e Cookie

Come abbiamo detto HTTP è stateless ma le Application Web hanno spesso bisogno di stato, soprattutto lo stato di ciascun utente. Sono state definite quindi due tecniche per mantenere traccia delle informazioni di stato:

1. **cookie**: *basso livello*
2. **session tracking**: *alto livello*

L’accesso alla **sessione** avviene mediante l’interfaccia HttpSession.

Per ottenere un riferimento ad un oggetto di tipo HttpSession si usa il metodo `public HttpSession getSession (boolean createNew)` . `createNew` può essere

- true: ritorna la sessione esistente o, se non esiste, ne crea una nuova.
- false: ritorna se possibile la sessione esistente, altrimenti null.

con `setAttribute` impostiamo un attributo per la sessione

```
HttpSession session = req.getSession(true);
session.setAttribute("valore", valore);


//nella servlet2:
HttpSession session = req.getSession(true);
session.getAttribute("valore");
```

Possiamo anche utilizzare i **Cookie**. Cos'è un cookie? Un cookie è un piccolo blocco di dati che viene creato dal web server mentre un utente sta navigando sul sito. E' un altro modo per dare uno "stato" al browsing.

```
Cookie cookie = new Cookie("k", k +"");              //accetta
res.addCookie(cookie);

//nella servlet2
int k = 0;
Cookie cookies[] = req.getCookies();                 // disponi
for(Cookie c : cookies){                             // con il
    if(c.getName().equals("k")) k = Integer.parseInt(c.getValue
}
```

Deployment

Una applicazione Web deve essere uno zip, un file di archivio con suffisso .war (*Web Archive*).

```
jar {ctxu} [vf] [jarFile] files
-ctxu: create, get the table of content, extract, update content
-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
```

Esempio di struttura di directory di Web Application:

Dove MyWebApplication è il nome del .war

Nota: nella cartella `lib` *vanno inserite solo le librerie particolari, non presenti in Tomcat.*

Spesso si mettono le lib solo su Eclipse e non su Tomcat, o anche viceversa: in questi casi avremo problemi di compilazione o a runtime. Occhio anche alle versioni delle librerie (deve essere la stessa).

Il file .xml è un file di configurazione che contiene una serie di elementi descrittivi: l'elenco delle servlet attive sul server, il loro mapping verso URL e per ognuna di loro definisce una serie di parametri presentati come coppie *nome-valore*.

Nel .xml viene fatto il mapping tra URL logico e nome della classe che lo implementa:

Nota: dobbiamo mappare TUTTE le servlet

ServletConfig & ServletContext

ServletContext rappresenta il contesto dell'applicazione web. Ogni applicazione web in un server ha un oggetto di questo tipo associato ad esso. Può essere utilizzato per condividere informazioni tra le diverse servlet all'interno della stessa applicazione: fornisce un modo per ottenere risposte comuni, come directory di caricamento, parametri di inizializzazione ed oggetti condivisi.

L'idea è quella di recuperare il ServletContext dell'applicazione web e settare un attributo: una volta fatto ciò, in un'altra servlet lo recuperiamo.

```
this.getServletContext().setAttribute("gruppo", gruppo);


//servlet2
String str = this.getServletContext().getAttribute("gruppo");
```

ServletConfig rappresenta la configurazione specifica di una singola servlet. Ogni servlet ha un oggetto di questo tipo associata ad essa, che viene fornito dal container servlet al momento della creazione della servlet. Contiene informazioni di configurazioni specifiche per la servlet, come parametri di inizializzazione definiti nel file di web.xml.

Nel file xml:

```
//per i parametri ServletContext
<context-param>
    <param-name> nome </param-name>
    <param-value> "riccardo" </param-name>
</context-param>

//per i parametri ServletConfig (all'interno del tag <servlet>)
<init-param>
    <param-name> nome </param-name>
    <param-value> "riccardo" </param-name>
</init-param>
```