

# Java EE

**Java Platform Enterprise Edition**



“

## Cos'è la Java Platform

**La piattaforma Java è composta da due componenti:**

- la **Macchina Virtuale Java** (Java Virtual Machine o JVM)
- le **API** (Application Programming Interface) Java, cioè un insieme di librerie (componenti software) a disposizione degli sviluppatori

**La piattaforma Java è disponibile in tre configurazioni:**

- **Standard Edition (Java SE)**: mette a disposizione le API per le esigenze più comuni e permette di scrivere applicazioni stand-alone, client e server, per accesso a database, per il calcolo scientifico etc...
- **Enterprise Edition (Java EE)**: mette a disposizione le API per scrivere applicazioni distribuite.
- **Micro Edition (Java ME)**: mette a disposizione le API per realizzare applicazioni per i dispositivi dotati di poche risorse computazionali (telefoni cellulari, palmari, smart cards ed altri).

“

## Java Development Kit: cos'è e a cosa serve

### Che cos'è il JDK?

Il JDK è un insieme di software che possono essere utilizzati per sviluppare applicazioni basate su Java.

Il JDK è un ambiente di sviluppo «a console», ovvero non ha un'interfaccia grafica ma le istruzioni si eseguono tramite il prompt dei comandi.

### A chi serve?

Il Java Development Kit è fondamentale per chi deve sviluppare applicazioni Java. Il JDK contiene al suo interno il **JRE** (Java Runtime Environment) con le sue **API** e la **JVM**.

“

## Cosa serve per cominciare

Istallazione JDK



Istallazione IDE Eclipse



Istallazione di Apache TomEE





## **JEE** è l'acronimo di **Java Platform Enterprise Edition**

Ad oggi **JEE** è una delle piattaforme tecnologiche più importanti per lo sviluppo di **applicazioni di livello enterprise**, specie in contesti dove è imprescindibile avere **sicurezza e robustezza** del software.



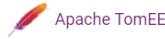
# Istallazione Apache Tomcat (TomEE)

Application Server

<http://tomee.apache.org/download-ng.html>



# Apache Tomcat (TomEE)



Apache TomEE

Documentation

Examples

Community

Security

Downloads

## Downloads

[Download as PDF](#)

Note: Only the TomEE 1.x WebProfile and JAX-RS distributions are certified.



Name	Version	Date	Size	Type	Links
TomEE plume	7.1.0	07 Sep 2018	65 MB	TAR.GZ	<a href="#">TAR.GZ</a> <a href="#">SHA256</a> <a href="#">SHA512</a>
TomEE plume	7.1.0	07 Sep 2018	65 MB	ZIP	<a href="#">ZIP</a> <a href="#">SHA256</a> <a href="#">SHA512</a>
TomEE plus	7.1.0	07 Sep 2018	58 MB	TAR.GZ	<a href="#">TAR.GZ</a> <a href="#">SHA256</a> <a href="#">SHA512</a>
TomEE plus	7.1.0	07 Sep 2018	58 MB	ZIP	<a href="#">ZIP</a> <a href="#">SHA256</a> <a href="#">SHA512</a>
TomEE webprofile	7.1.0	07 Sep 2018	41 MB	TAR.GZ	<a href="#">TAR.GZ</a> <a href="#">SHA256</a> <a href="#">SHA512</a>
TomEE webprofile	7.1.0	07 Sep 2018	41 MB	ZIP	<a href="#">ZIP</a> <a href="#">SHA256</a> <a href="#">SHA512</a>



## Apache Tomcat (TomEE)

**Dopo aver scaricato  
ed estratto i file  
siamo pronti per  
utilizzare il nostro  
Application Server**

to PC > Windows (C:) > java > apache-tomee-plume-7.1.0			
Nome	Ultima modifica	Tipo	Dimensione
bin	02/09/2018 22:00	Cartella di file	
conf	23/09/2018 18:39	Cartella di file	
lib	02/09/2018 22:00	Cartella di file	
logs	29/09/2018 10:36	Cartella di file	
temp	29/09/2018 10:36	Cartella di file	
webapps	02/09/2018 22:00	Cartella di file	
work	23/09/2018 18:39	Cartella di file	
LICENSE	02/09/2018 22:00	File	52 KB
NOTICE	02/09/2018 21:47	File	8 KB
RELEASE-NOTES	20/06/2018 20:51	File	8 KB
RUNNING.txt	20/06/2018 20:51	Documento di testo	17 KB



## Apache Tomcat (TomEE)

All'interno della cartella Bin avremo 2 file che ci permettono di far partire (**startup.bat**) o spegnere (**shutdown.bat**) il nostro Application Server.

 shutdown.bat	20/06/2018 20:51	File batch Windows	2 KB
 shutdown.sh	20/06/2018 20:51	File SH	2 KB
 startup.bat	20/06/2018 20:51	File batch Windows	2 KB
 startup.sh	20/06/2018 20:51	File SH	2 KB



# Apache Tomcat (TomEE)

← → ⌛ ⌂ ⓘ localhost:8080 ⚙️ 🎨 📁 🖼 🖼 🖼 🖼 🖼 🖼

Home Documentation Configuration Examples Wiki Mailing Lists Find Help

**Apache Tomcat (TomEE)/8.5.32 (7.1.0)**

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:  
[Security Considerations HOW-TO](#)  
[Manager Application HOW-TO](#)  
[Clustering/Session Replication HOW-TO](#)

Server Status  
Manager App  
Host Manager

**Developer Quick Start**

Tomcat Setup  
First Web Application      Realms & AAA  
JDBC DataSources      Examples      Servlet Specifications  
Tomcat Versions

**Managing Tomcat**  
For security, access to the [manager\\_webapp](#) is restricted. Users are defined in:  
`§CATALINA_HOME/conf/tomcat-users.xml`  
In Tomcat 8.5 access to the manager application is split between different users.  
[Read more...](#)

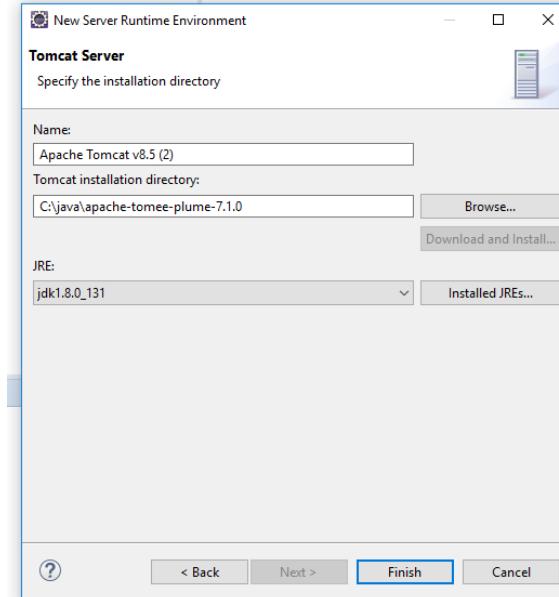
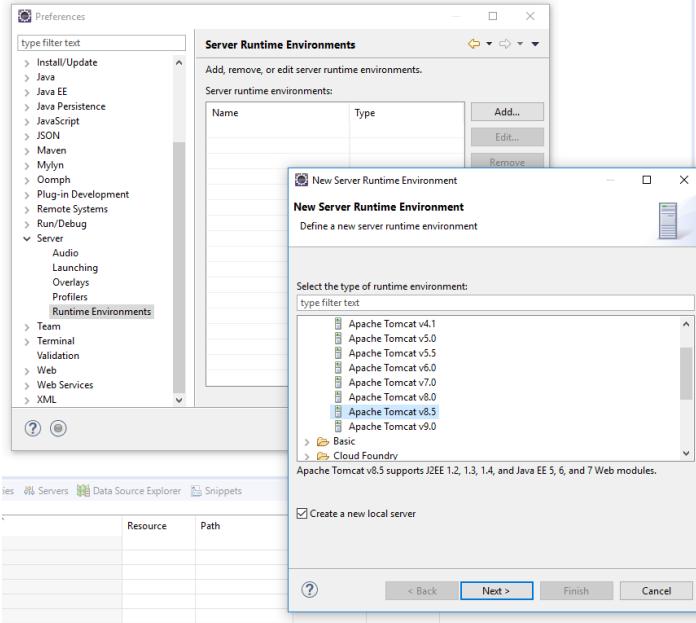
**Documentation**  
[Tomcat 8.5 Documentation](#)  
[Tomcat 8.5 Configuration](#)  
[Tomcat Wiki](#)  
Find additional important configuration information in:  
`§CATALINA_HOME/RUNNING.txt`

**Getting Help**  
[FAQ and Mailing Lists](#)  
The following mailing lists are available:  
`tomcat-announce`  
Important announcements, releases, security vulnerability notifications. (Low volume).  
`tomcat-users`  
User support and discussion

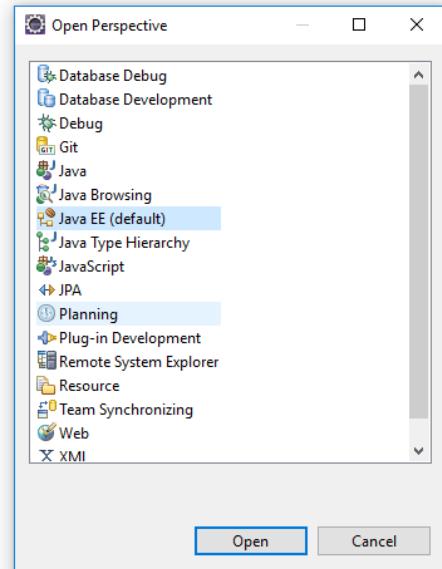
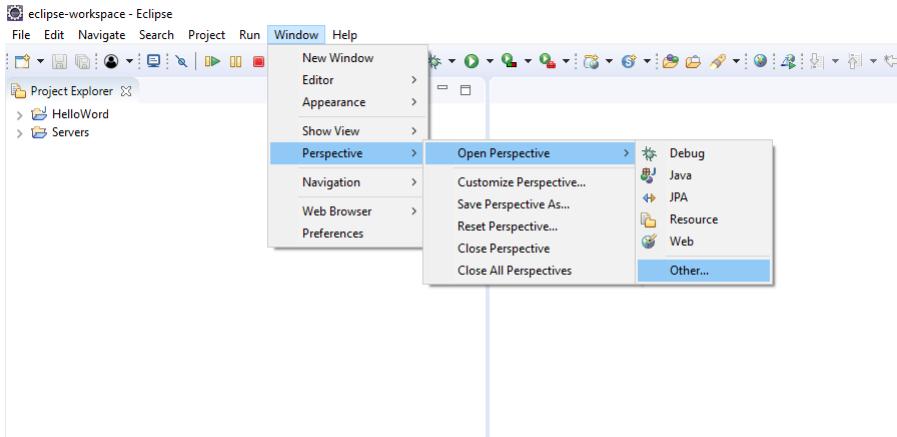


# Configurare Tomcat su Eclipse

# Configurare Tomcat su Eclipse



# Configurare Tomcat su Eclipse





## Configurare Tomcat su Eclipse

# Impostazioni e gestione Tomcat su Eclipse

The screenshot shows the 'Tomcat v8.5 Server at localhost' configuration dialog in the Eclipse IDE. The 'Overview' tab is selected, displaying settings for the server name (Tomcat v8.5 Server at localhost), host name (localhost), runtime environment (Apache Tomcat v8.5), and configuration path (Servers/Tomcat v8.5 Server at localhost). The 'Ports' section lists the port names and numbers: Tomcat admin port (8005), HTTP/1.1 (8080), and AJP/1.3 (8009). The 'MIME Mappings' section is also visible. At the bottom, there are tabs for Overview, Modules, and a large blue arrow pointing right towards the 'Servers' view.

Tomcat v8.5 Server at localhost

Overview

General Information

Specify the host name and other common settings.

Server name: Tomcat v8.5 Server at localhost

Host name: localhost

Runtime Environment: Apache Tomcat v8.5

Configuration path: /Servers/Tomcat v8.5 Server at localhost

Open launch configuration

Server Locations

Specify the server path (i.e., catalina.base) and deploy path. Server must be published with no modules present to make changes.

( Use workspace metadata (does not modify Tomcat installation))

( Use Tomcat installation (takes control of Tomcat installation))

( Use custom location (does not modify Tomcat installation))

Server path: .metadata/.plugins/org.eclipse.wst.server.core/t

Set deploy path to the default value (currently set)

Deploy path: wtpwebapps

Server Options

Enter settings for the server.

Serve modules without publishing

Publish module contexts to separate XML files

Module auto reload by default

Enable security

Enable Tomcat debug logging (not supported by this Tomcat version)

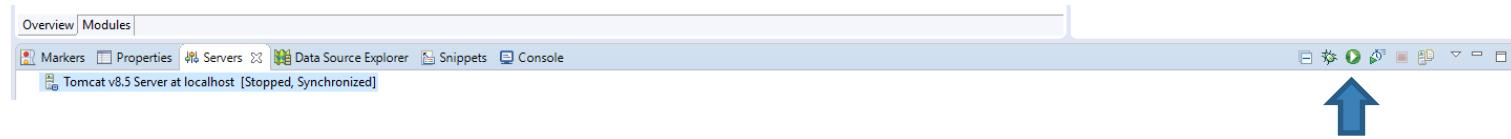
Overview | Modules

Markers Properties Servers Data Source Explorer Snippets Console

Tomcat v8.5 Server at localhost [Stopped]



## Configurare Tomcat su Eclipse



```
Tomcat v8.5 Server at localhost [Apache Tomcat] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (29 set 2018, 11:39:33)
INFORMAZIONI: Configuring Service(id=My Stateful Container, type=Container, provider-id=Default Stateful Container)
set 29, 2018 11:39:36 AM org.apache.openejb.config.ConfigurationFactory configureService
INFORMAZIONI: Configuring Service(id=My Stateless Container, type=Container, provider-id=Default Stateless Container)
set 29, 2018 11:39:36 AM org.apache.openejb.config.DeploymentsResolver loadFrom
AVVERTENZA: File error: <Deployments dir="apps/"> - Does not exist: C:\Users\umber\eclipse-workspace\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\apps
set 29, 2018 11:39:36 AM org.apache.openejb.util.OptionsLog info
INFORMAZIONI: Using 'openejb.deployments.classpath=false'
set 29, 2018 11:39:36 AM org.apache.openejb assembler.classicAssembler createRecipe
INFORMAZIONI: Creating TransactionManager(id=Default Transaction Manager)
set 29, 2018 11:39:36 AM org.apache.openejb assembler.classicAssembler createRecipe
INFORMAZIONI: Creating SecurityService(id=Tomcat Security Service)
set 29, 2018 11:39:36 AM org.apache.openejb assembler.classicAssembler createRecipe
INFORMAZIONI: Creating Resource(id=My DataSource)
```

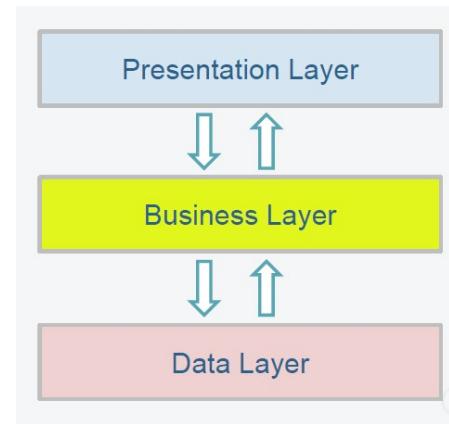


# Web Application



Un software (compresa la web application) è tipicamente composta da 3 livelli applicativi:

Questa suddivisione in livelli è un **design pattern** (cioè uno schema architettonico applicabile ad un problema ricorrente) che consente di separare nettamente il codice sorgente della parte di presentazione(l'interfaccia utente) dal codice di accesso ai dati e dal codice per la loro gestione.





## Il Pattern MVC (Model – View – Controller)

Questo pattern consente di separare nettamente la logica di presentazione dei dati (ovvero le pagine viste dall'utente) dalla logica di business (ovvero le funzionalità per l'accesso a tali dati).

**Separare la vista dalla logica di business ci consente di riutilizzare parti di software evitando ridondanza di codice e funzionalità!**



Il pattern si basa sulla netta separazione delle attività tra model, view e controller:

**Model** implementa i metodi per l'accesso ai dati dell'applicazione e si interfaccia con il database o con eventuali livelli di astrazione dei dati;

**Controller** gestisce le richieste di accesso alle view inviate dall'utente, interfacciandosi, se necessario con il model per l'accesso ai dati;

**View** gestisce le componenti visualizzate dall'utente (pagine testuali, form, liste, dati recuperati dal model, etc...);

## Web Application



Un'applicazione sviluppata senza il pattern **MVC** avrà una pagina web che ogni volta che viene chiamata si occuperà di accedere ai dati presenti sul database, elaborarli e generare l'HTML.

**La pagina sarà un MIX tra HTML e logica!**

E se ho un'altra pagina che deve accedere a parte dei dati visualizzati in un'altra pagina, devo implementare un'altra volta le funzioni di accesso al database!



# Come funziona l'MVC per le Web application?

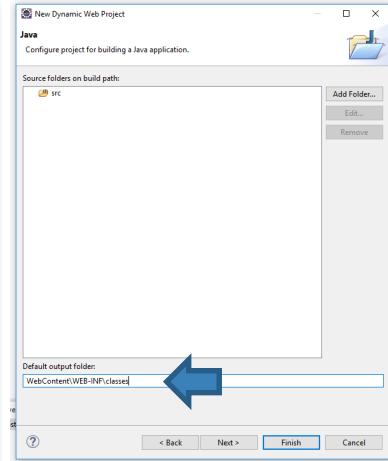
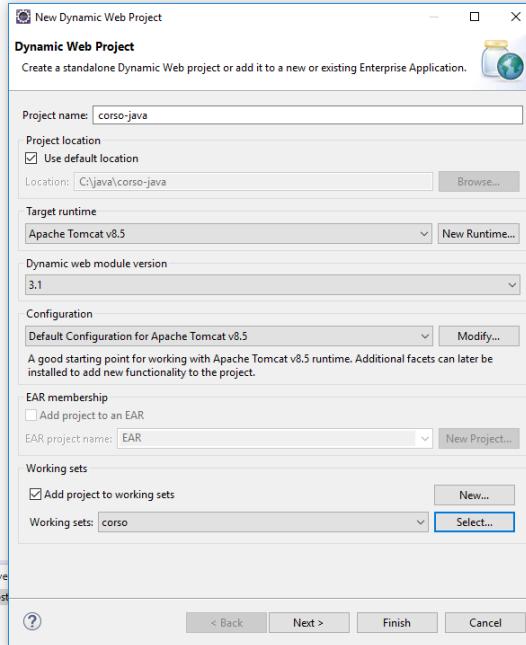
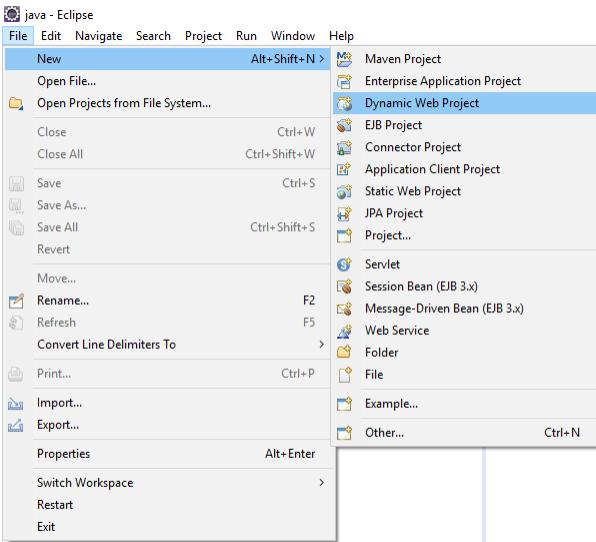
Un'applicazione sviluppata con il pattern MVC avrà, invece, tre componenti:

1. il **model** che si occuperà dei dati e fornirà i metodi per accedervi.
2. la **view**(cioè la pagina web) che si occuperà di creare il codice HTML.
3. il **controller** che fa da intermediario tra model e view e che si occuperà di intercettare e gestire la URL richiesta dall'utente



# Creiamo la nostra Web Application

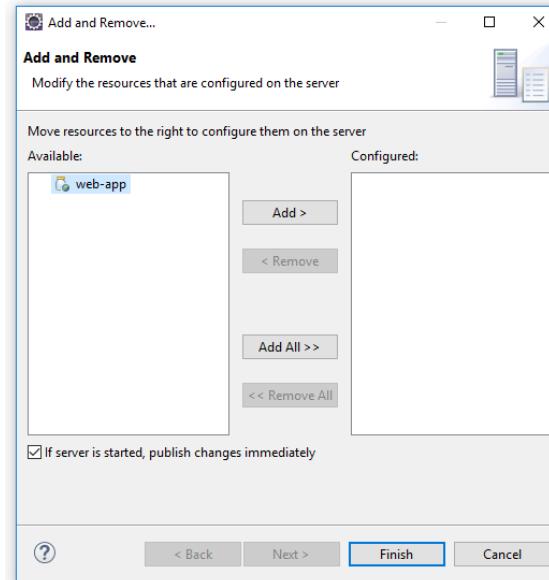
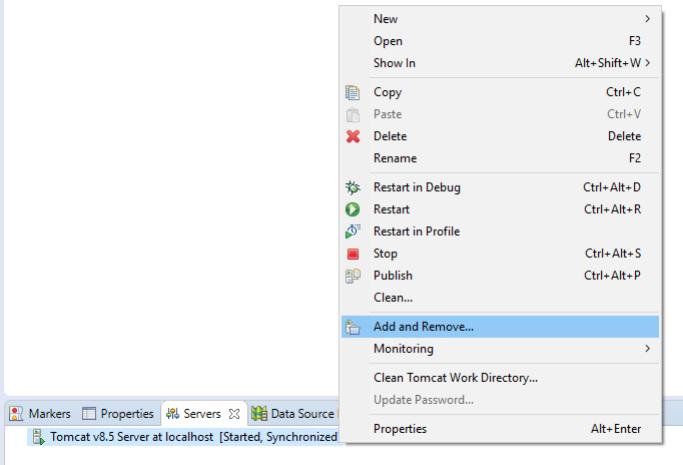
# Web Application



## Web Application



### Caricamento sul server tramite Eclipse





Esportare il nostro progetto tramite WAR per la distribuzione.

Il WAR è un file compresso dove sono presenti tutti i file del nostro progetto.



# Web Application

java - web-app/WebContent/index.html - Eclipse

File Edit Navigate Search Project Run Window Help

Project Explorer Servers web-app

- New Go into Alt+Shift+W
- Copy Ctrl+C
- Copy Qualified Name
- Paste Ctrl+V
- Delete Ctrl+D
- Remove from Context Ctrl+Alt+Shift+Down
- Build Path
- Refactor Alt+Shift+T
- Import
- Export** F5
- Refresh Close Project Close Unrelated Projects
- Show in Remote Systems view Validate Coverage As Run As Debug As Profile As Restore from Local History... Java EE Tools Team Compare With Configure Source Properties Alt+Enter

WAR file Export...

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="UTF-8">
5 <title>Insertion</title>
6 </head>
7 <body>
8 <h1>Hello</h1>
9 </body>
10 </html>
```

Tomcat v8.5 web-app

Export WAR Export Export Web project to the local file system.

Web project: **web-app** Destination: Browse...

Target runtime:  Optimize for a specific Apache Tomcat v8.5

Export source files  Overwrite existing file

Salva con nome

Questo PC > Windows (C) > java > apache-tomee-plume-7.1.0

Nome	Ultima modifica
bin	02/09/2018 22:00
conf	23/09/2018 18:39
lib	02/09/2018 22:00
logs	29/09/2018 10:36
temp	29/09/2018 12:26
Documenti	
Immagini	
Accesso rapido	
Desktop	
Download	
Dropbox	
Dropbox	
webapps	02/09/2018 22:00
work	23/09/2018 18:39

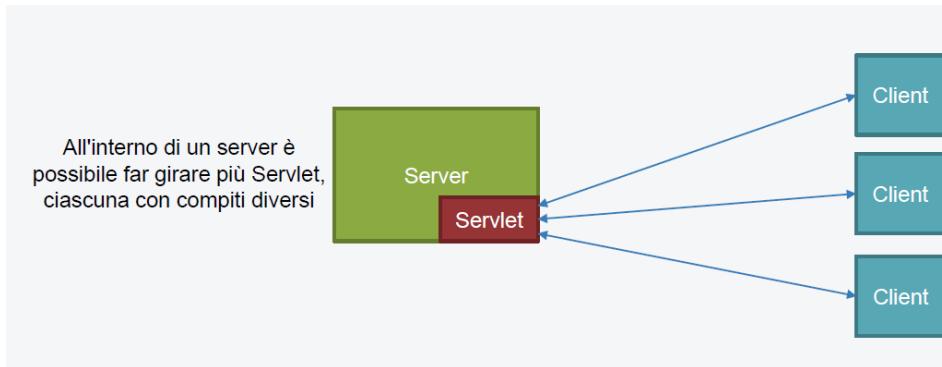


# **Servlet ed HTTPServlet**



# Cos'è una Servlet?

Una Servlet è un software scritto in Java che è in grado di ricevere ed elaborare richieste da uno o più client.





## Servlet ed HTTPServlet

Le **Servlet** generalmente vengono installate (teoricamente si dice fare il «**deploy**») all'interno di **Servlet Container** (ad es. Tomcat).

Il **Servlet Container** è una piattaforma software in grado di eseguire applicazioni Web sviluppate in Java.



## Servlet ed HTTPServlet

Una **Servlet** non ha delle GUI associate, quindi le librerie **SWT**, **AWT** e **Swing** non vengono utilizzate quando si crea una Servlet.

Le Servlet possono essere utilizzate per:

- creare pagine web dinamiche
- effettuare connessioni ad un db con JDBC e restituire dati in diversi formati
- effettuare l'autenticazione di un utente
- ...



## Servlet ed HTTPServlet

In ambito lavorativo, per la creazione di web application generalmente non si sviluppano direttamente Servlet, ma si utilizzano framework che implementano la specifica servlet: **Spring – JSF – Struts**.

Una servlet può essere utilizzata per sviluppare un singolo servizio (ad es. il download di un file PDF...)



Il package che contiene classi ed interfacce di riferimento è  
**javax.servlet.**

Esistono diverse tipologie di Servlet, ognuna delle quali estende  
la classe principale  
**javax.servlet.GenericServlet.**

Quando si lavora in ambito web, la classe da utilizzare per la  
realizzazione di Servlet è  
**javax.servlet.http.HttpServlet.**



## Servlet ed **HTTPServlet**

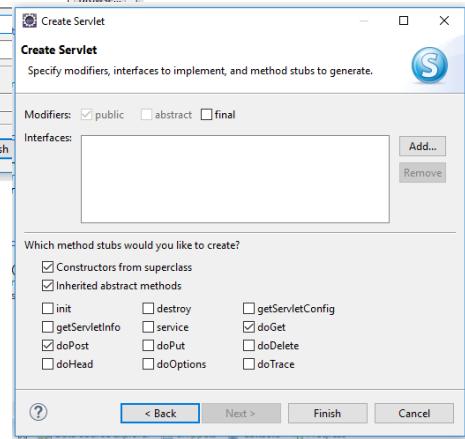
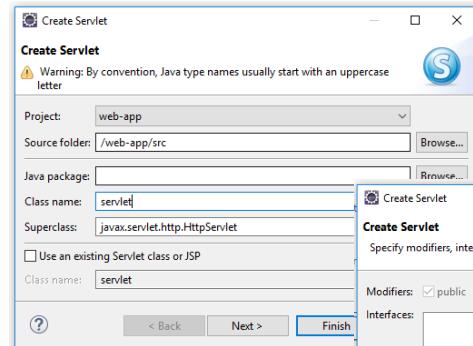
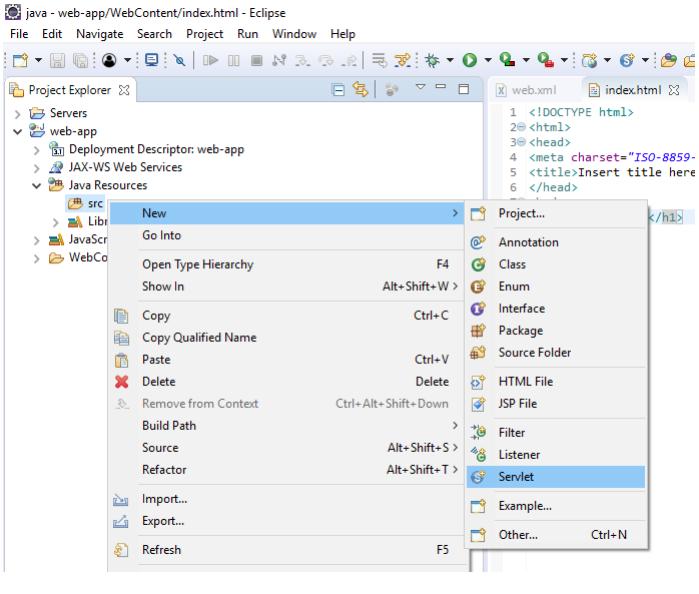
Creare una **Servlet** vuol dire creare una classe che estende la classe **HttpServlet**!

```
@WebServlet("/path-per-invocare-la-servlet")
public class TestServlet extends HttpServlet {
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        ...
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        ...
    }
}
```



# Servlet ed HttpServlet

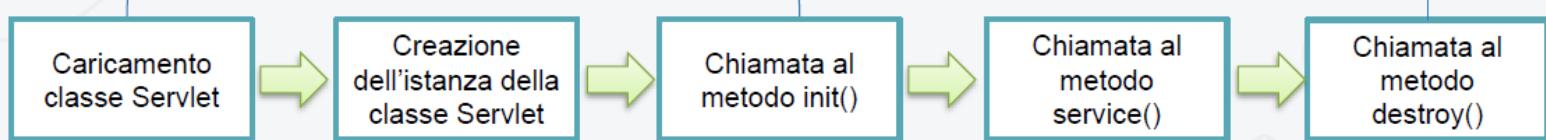




Una **Servlet** ha un ciclo di vita ben definito che definisce:  
**Come caricarla, Come istanziarla, Come inizializzarla, Come vengono gestite le richieste da parte dei client, Come scaricarla.**

Vengono eseguite solo una volta,  
quando la Servlet deve essere avviata

Invocato solo quando  
la Servlet viene rimossa  
(non sarà più accessibile)





Il ciclo di vita di una **Servlet** è gestito da i seguenti tre metodi dell’interfaccia **javax.servlet.Servlet**

- init(ServletConfig config)**
- service(ServletRequest req, ServletResponse res)**
- destroy()**



## Servlet ed HTTPServlet

### Fase 1 e Fase 2 - Caricamento e di creazione dell'Istanza

Il Servlet container è responsabile del caricamento e della creazione dell'istanza della Servlet. Il caricamento può avvenire:

- all'avvio del Servlet engine
- all'arrivo della prima richiesta da parte di un client



## Servlet ed HTTPServlet

### Fase 3 - Inizializzazione

La fase di inizializzazione si ha solo dopo aver creato l'istanza della Servlet, viene effettuata dal container e consente di inizializzare le risorse di cui ha bisogno la Servlet per gestire le richieste (ad es. aprire le connessioni JDBC, collegarsi ad altri servizi, etc...).

Queste operazioni vengono eseguite una sola volta.



## Servlet ed HTTPServlet

### Fase 4 - Gestione delle richieste

La Servlet gestisce le richieste dei client.

La richiesta è rappresentata da un oggetto di tipo **javax.servlet.ServletRequest**.

La risposta è rappresentata da un oggetto di tipo **javax.servlet.ServletResponse**.

Questi due oggetti sono passati in ingresso al metodo service dell'interfaccia Servlet  
*(service(ServletRequest req, ServletResponse res))*.

Nelle richieste che avvengono su protocollo HTTP, gli oggetti sono di tipo

**HttpServletRequest e HttpServletResponse** (package javax.servlet.http).



## Servlet ed HTTPServlet

### Fase 5 - Distruzione

Il container si occupa di invocare il metodo `destroy()` per liberare le risorse utilizzate dalla Servlet e completare l'elaborazione in corso. A questo punto la Servlet viene rimossa dal container.



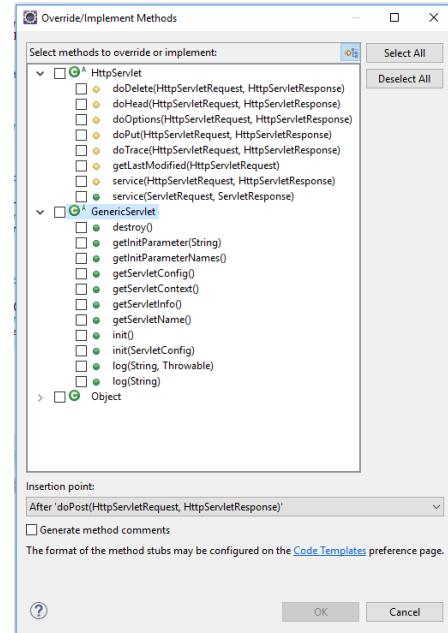
## Servlet ed HttpServlet

La classe astratta `HttpServlet`, oltre ad implementare i metodi dell'interfaccia `Servlet`, implementa dei metodi aggiuntivi per gestire le richieste basate su HTTP.

Questi metodi sono:

- doGet** per gestire richieste HTTP GET
- doPost** per gestire richieste HTTP POST
- doPut** per gestire richieste HTTP PUT
- doDelete** per gestire richieste HTTP DELETE
- doHead** per gestire richieste HTTP HEAD
- doOptions** per gestire richieste HTTP OPTIONS
- doTrace** per gestire richieste HTTP TRACE

Tutti i metodi prendono in ingresso due oggetti di tipo **HttpServletRequest** e **HttpServletResponse**. I metodi più usati nelle web application sono **doGet** e **doPost**. In altri contesti (ad es. le API RESTfull) vengono utilizzati anche gli altri.





**HttpServletRequest** ed **HttpServletResponse** sono interfacce contenute nel package **javax.servlet.http**.

Come funzionano?



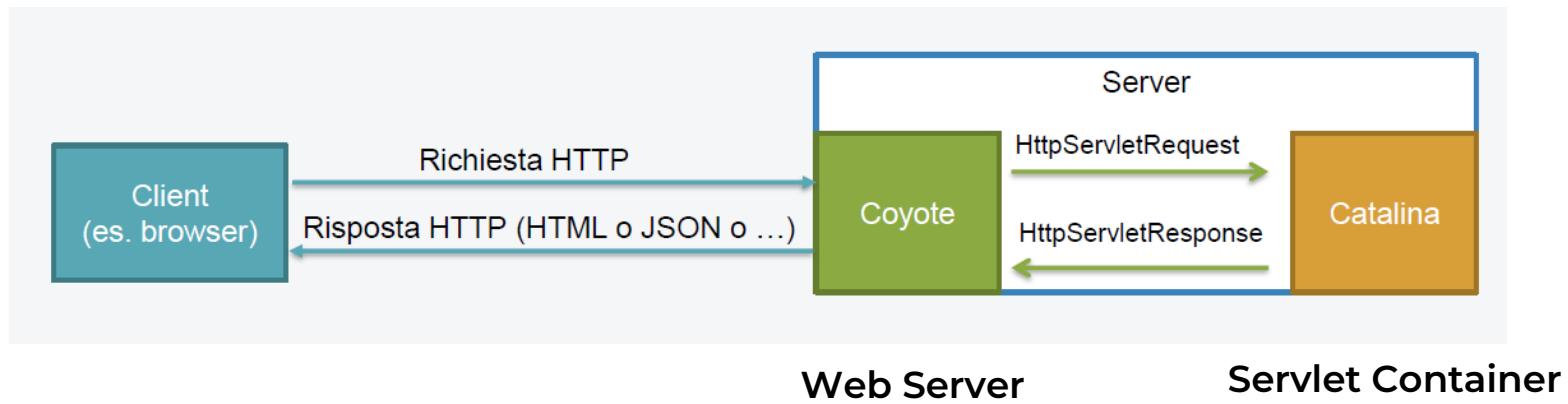
## Servlet ed **HTTPServlet**

Quando richiediamo l'accesso ad una risorsa web (ad es. quando scriviamo l'URL di una pagina web nella barra degli indirizzi), abbiamo il seguente scenario:

- 1.** Il connettore **HTTP** (Coyote per Tomcat) riceve la richiesta HTTP che può essere di tipo **GET** o **POST** e la reindirizza la richiesta HTTP al **servlet container**.
- 2.** Se la **servlet** non è ancora in memoria, il **servlet container** (Catalina per Tomcat) la carica e la inizializza mediante l'invocazione del metodo **init()**.
- 3.** Il **servlet container** incapsula la richiesta HTTP in un oggetto di tipo **HttpServletRequest** e la passa al metodo **doPost** (se la richiesta è di tipo POST) o **doGet** (se la richiesta è di tipo GET) della servlet.
- 4.** La servlet elabora la richiesta e scrive la risposta (ad esempio un codice HTML) nella **HttpServletResponse**
- 5.** L'**HttpServletResponse** viene reindirizzata al web server che si occuperà di inviarla al client via HTTP.



## Servlet ed HTTPServlet





## Servlet ed HttpServlet

Name x Headers Preview Response Timing

server

inject.js

General

Request URL: http://localhost:8080/web-app/servlet  
Request Method: GET  
Status Code: 200  
Remote Address: [::1]:8080  
Referrer Policy: no-referrer-when-downgrade

Response Headers view source

Content-Length: 19  
Date: Sun, 30 Sep 2018 07:15:36 GMT  
Server: Apache TomEE

Request Headers view source

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8  
Accept-Encoding: gzip, deflate, br  
Accept-Language: it-IT,it;q=0.9,en-US;q=0.8,en;q=0.7  
Cache-Control: max-age=0  
Connection: keep-alive  
Host: localhost:8080  
Upgrade-Insecure-Requests: 1  
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36



# I principali metodi dell'interfaccia **HttpServletRequest**

**request.getMethod()**: restituisce il nome del metodo HTTP utilizzato per effettuare la richiesta al server (GET, POST, PUT);

**request.getLocale()**: restituisce il Locale (cioè un oggetto che rappresenta una specifica area, ad es. "en" (English), "it" (Italia)) predefinito del client;

**request.getCharacterEncoding()**: restituisce il nome del character encoding (la codifica dei caratteri) utilizzato nel body della request. Alcuni nomi di codifiche sono: ISO 8859-1, UTF-8. Nelle pagine HTML5 questo attributo è inserito in un meta (<meta charset="UTF-8">);



## Servlet ed **HTTPServlet**

**request.getContentType()**: restituiste il MIME type del body della request o null se il MIME type è sconosciuto. Il MIME type (Multipurpose Internet Mail Extensions) è un formato utilizzato nel protocollo HTTP per codificare i messaggi scambiati tra client e server web. (text/html - multipart/form-data - application/json);

**request.getContextPath()**: restituisce la parte della request URI che corrisponde al path associato alla servlet o in generale alla web application invocata. Il path inizia con il simbolo "/" e non termina mai con il simbolo "/" ;

**request.getCookies()**: restituisce un array contenente tutti i cookies inviati con la request. I cookies vengono inseriti all'interno di oggetti di tipo

**javax.servlet.http.Cookie**.



## Servlet ed HTTPServlet

**request.getHeader(String name)**: ritorna il valore del parametro specificato in ingresso che si trova nell'header della request (ad es. l'user-agent).

**request.getHeaderNames()**: ritorna un enumeration contenente tutti nomi dei parametri presenti nell'header della request.

**request.getQueryString()**: restituisce la query string contenuta nell'URL. La query string è la stringa che si trova dopo il simbolo «?». Il metodo ritorna null se la URL non contiene query string. Nella query string i parametri sono composti dalla coppia **nome=valore**. In caso di più parametri in query string, si usa il simbolo «&» per separarli.



## Servlet ed HTTPServlet

**request.getParameter(String name):** restituisce il valore associato ad un parametro passato nella **request** (compresi quelli passati nella query string). Se il parametro non esiste il metodo ritorna null.

**request.getParameterMap():** restituisce un oggetto di tipo **java.util.Map** che contiene l'elenco dei parametri (con relative valori) presenti nella request.

**request.getParameterNames():** restituisce un enumeration contenente tutti i nomi dei parametri passati nella request.

**request.getParameterValues(String name):** restituisce un array di stringhe associate ad un parametro. Il tipico esempio è un form che contiene degli input che hanno lo stesso nome (generalmente dei checkbox che consentono la selezione di più opzioni).



## Servlet ed HTTPServlet

**request.getAttribute(String name):** restituisce un oggetto (può essere una stringa ma anche un oggetto complesso) associato al nome passato in ingresso o null se non esiste un attributo con quel nome.

**request.getAttributeNames():** restituisce un enumeration contenente i nomi degli attributi presenti nella request.

**request.getRemoteAddr():** restituisce l'indirizzo IP del client che ha effettuato la request.

**request.getRemoteUser():** restituisce una stringa contenente il parametro utilizzato per il login dell'utente. (generalmente l'username o l'email): Il metodo ritorna null se non esiste un utente autenticato.



## Servlet ed HTTPServlet

**request.getUserPrincipal()**: restituisce un oggetto di tipo **java.security.Principal** che contiene il nome dell'utente autenticato o null se non esiste un utente autenticato. L'interfaccia **Principal** definisce un metodo **getName()** per recuperare il nome associato all'oggetto.

**request.getRequestDispatcher(String path)**: restituisce un oggetto di tipo **RequestDispatcher** che agisce come wrapper per la risorsa che è associata al path specificato.

**request.getRequestURL()**: restituisce un oggetto di tipo **StringBuffer** contenente l'URL della request e contiene il protocollo, il server name, il numero della porta ed il server path. La stringa ritornata non contiene la query string.

**request.getRequestURI()**: restituisce la porzione di URL che si trova dopo il server name



## Servlet ed HTTPServlet

**request.getServerName()**: Restituisce il nome del server su cui è in esecuzione la web application.

**request.getServerPort()**: Restituisce il numero della porta del server su cui è in esecuzione la web application.

**request.getServletPath()**: restituisce la parte di URL a cui è associata la servlet.

**request.getSession()**: restituisce un oggetto di tipo **javax.servlet.http.HttpSession** contenente la sessione associata alla request ricevuta. Se non esiste una session, questo metodo la crea.

**request.setAttribute(String name, Object value)**: memorizza un attributo nella request. Tutti gli attributi vengono cancellati tra una request e l'altra.



# I principali metodi dell'interfaccia **HttpServletResponse**

**response.addCookie(Cookie arg0)**: aggiunge un cookie alla response

**response.addHeader(String name, String value)**: aggiunge un parametro nell'header della response.

**response.encodeURL(String url)**: effettua l'encode dell'URL includendo il session ID. L'implementazione del metodo contiene la logica che determina se il session ID debba essere incluso nell'URL (ad es. se il browser supporta i cookies l'URL encoding non è necessario).

**response.flushBuffer()**: forza il l'invio del contenuto presente nel buffer al client.



## Servlet ed HTTPServlet

**response.getOutputStream()**: restituisce un oggetto di tipo `javax.servlet.ServletOutputStream` utilizzabile per scrivere dati binary nella response (ad esempio per inviare un PDF al client...)

L'invocazione del metodo `flush()` sull'oggetto effettua l'invio della response al client.

**response.getWriter()**: restituisce un oggetto di tipo `java.io.PrintWriter` che può essere utilizzato per inviare testo (ad esempio codice HTML) al client.

**response.sendError(int statusCode, String message)**: invia un error con lo status code indicato al client.



## Servlet ed HTTPServlet

**response.sendRedirect(String URL):** è utilizzata per reindirizzare la risposta ad un'altra risorsa (ad es. una servlet o una jsp). Il parametro da passare in input è una URL relativa o assoluta. Il metodo lavora a livello client perché utilizza la barra degli indirizzi del browser per effettuare una nuova request (quando si riceve la risposta, nella barra degli indirizzi comparirà la URL dove si verrà reindirizzati).

**response.setStatus(int sc):** imposta lo status code della response

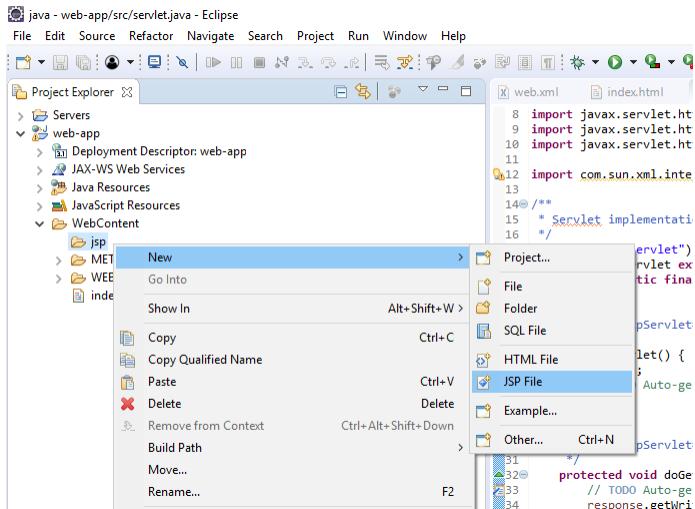


# Passaggio di parametri GET e POST



## Passaggio di parametri GET e POST

Creiamo il nostro file `home.jsp` all'interno di una cartella `jsp` della nostra applicazione.





## Passaggio di parametri GET e POST

```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2     pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4<html>
5<head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>Insert title here</title>
8</head>
9<body>
10<form action=<%=request.getContextPath() + "/servlet" %> method="get">
11     <label for="name">Name:</label>
12     <input type="text" name="name" value="">
13     <label for="lastname">Last Name:</label>
14     <input type="text" name="lastname" value="">
15     <button type="submit" name="submit">Send</button>
16 </form>
17 </body>
18 </html>
```

home.jsp

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) {
    String name = request.getParameter("name");
    String lastName = request.getParameter("lastname");
    PrintWriter out = response.getWriter();
    out.println("<h1>Ciao " + name + " " + lastName + "</h1>");
    out.println("<h2>Method: " + request.getMethod() + "</h2>");
}
```

servlet.java

← → C ⌂ ⓘ localhost:8080/web-app/servlet?name=Umberto&lastname=Emanuele&submit=

# Ciao Umberto Emanuele



## Passaggio di parametri GET e POST

```
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2     pageEncoding="ISO-8859-1"%>
3 <!DOCTYPE html>
4<html>
5<head>
6 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
7 <title>Insert title here</title>
8 </head>
9<body>
10<form action=<%=request.getContextPath()%>/servlet" method="post">
11     <label for="name">Name:</label>
12     <input type="text" name="name" value="">
13     <label for="lastname">Last Name:</label>
14     <input type="text" name="lastname" value="">
15     <button type="submit" name="submit">Send</button>
16 </form>
17 </body>
18 </html>
```

home.jsp

```
protected void doPost(HttpServletRequest request, HttpServletResponse response
    // TODO Auto-generated method stub
    //doGet(request, response);

    String name = request.getParameter("name");
    String lastName = request.getParameter("lastname");

    PrintWriter out = response.getWriter();

    out.println("<h1>Ciao " + name + " " + lastName + "</h1>");
    out.println("<h2>Method: " + request.getMethod() + "</h2>");
}
```

servlet.java

← → C ⌂ ⓘ localhost:8080/web-app/servlet

**Ciao Umberto Emanuele**

**Method: POST**



# Request Dispatching



## Request Dispatching

Quando sviluppiamo una web application, difficilmente riusciamo a gestire tutte le funzionalità con una servlet.

Nel paradigma MVC, inoltre, le richieste arrivano ad uno o più controller che elaborano i dati e reindirizzano la **request** alla risorsa interessata (una JSP o un'altra Servlet).

Per gestire quest'attività di «dispatching» possiamo utilizzare l'interfaccia **javax.servlet.RequestDispatcher**.



## Request Dispatching

Per inoltrare la request possiamo fare in due modi:

**request.getServletContext().getRequestDispatcher(String jspPath)**, per reindirizzare la request ad una jsp;

**request.getServletContext().getNamedDispatcher(String servletName)**, per invocare un'altra servlet.

Entrambi i metodi ritornano un oggetto di tipo **RequestDispatcher**.



## Request Dispatching

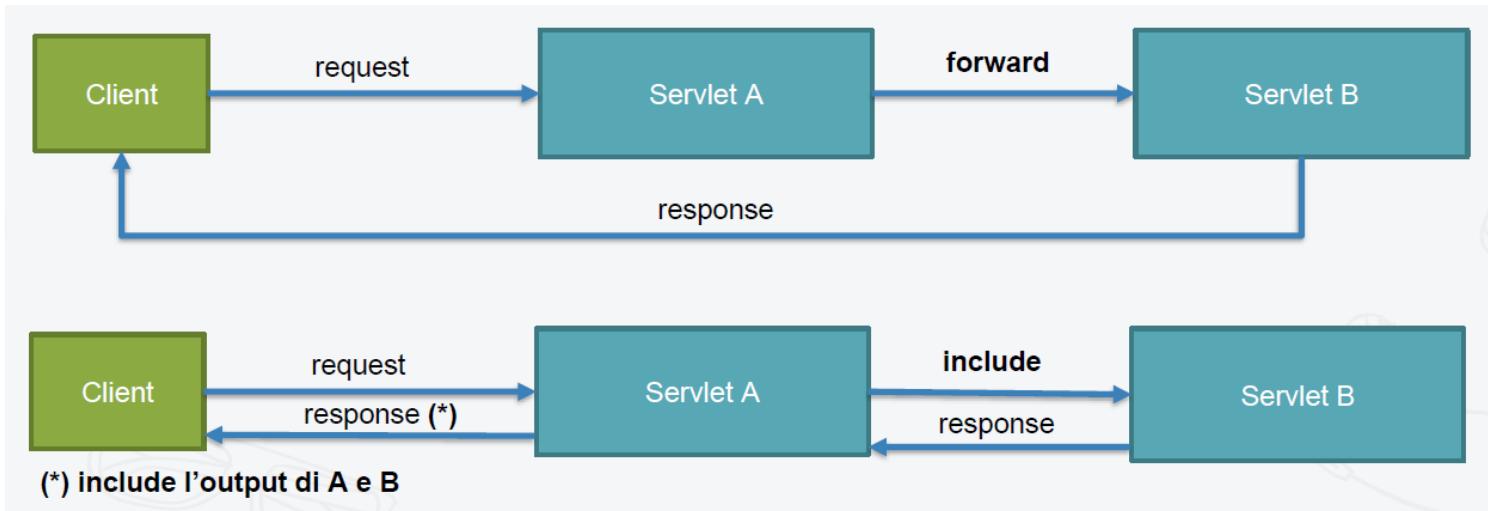
L'interfaccia **RequestDispatcher** mette a disposizione due metodi per inoltrare la richiesta alla nuova risorsa:

***forward(request, response)***: delega l'invio della risposta alla risorsa a cui ha inoltrato la request;

***include(request, response)***: per includere l'output ottenuto da un'altra risorsa web (ad es. un'altra servlet).



## Request Dispatching





## Request Dispatching

Creiamo una nuova **dispatcher**, e tre **jsp**  
che conterranno la nostra pagina HTML  
**header.jsp**, **body.jsp** e **footer.jsp**

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
    request.getServletContext().getRequestDispatcher("/jsp/header.jsp").include(request, response);  
    request.getServletContext().getRequestDispatcher("/jsp/body.jsp").include(request, response);  
    request.getServletContext().getRequestDispatcher("/jsp/footer.jsp").include(request, response);  
}
```



## Request Dispatching

Creiamo ancora e tre **jsp** che conterranno contenuto diverso per la nostra **dispatcher** **pagina1.jsp**, **pagina2.jsp** e **pagina3.jsp**

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

    request.getServletContext().getRequestDispatcher("/jsp/header.jsp").include(request, response);
    String pagina = request.getParameter("pagina");
    if(pagina != null && !pagina.trim().equals("")) {
        if(pagina.equals("1")) {
            request.getServletContext().getRequestDispatcher("/jsp/pagina1.jsp").include(request, response);
        } else if(pagina.equals("2")) {
            request.getServletContext().getRequestDispatcher("/jsp/pagina2.jsp").include(request, response);
        } else if(pagina.equals("3")) {
            request.getServletContext().getRequestDispatcher("/jsp/pagina3.jsp").include(request, response);
        } else {
            request.getServletContext().getRequestDispatcher("/jsp/body.jsp").include(request, response);
        }
    } else {
        request.getServletContext().getRequestDispatcher("/jsp/body.jsp").include(request, response);
    }
    request.getServletContext().getRequestDispatcher("/jsp/footer.jsp").include(request, response);
}
```



## Request Dispatching

Identico è il funzionamento utilizzando il **forward** deleghiamo la **response** ad un'altra pagina interna alla nostra applicazione.

```
request.getServletContext().getRequestDispatcher("/jsp/home.jsp").forward(request, response);
```



# HTTPSession



Sappiamo che il protocollo HTTP è **stateless**.

Questo vuol dire che il server, dopo aver soddisfatto la richiesta ricevuta, chiude la connessione con il client, senza conservare informazioni sul client stesso.





Per sopperire al limite del protocollo HTTP, i web server devono implementare un meccanismo per la gestione delle richieste effettuate dagli utenti.

### **SESSIONE HTTP**



## HttpSession

La sessione HTTP in Java viene gestita attraverso due componenti:

Il parametro **JSESSIONID** salvato all'interno di un cookie che rappresenta l'ID univoco di una sessione

L'interfaccia **javax.servlet.http.HttpSession**



## HttpSession

Quando una **web application Java** riceve una richiesta da un client, il **Servlet Engine** (Catalina) controlla se la **request** ricevuta contiene il **JSESSIONID**:

Se è presente, vuol dire che esiste già una sessione per il client

Se non è presente, il **ServletEngine** ne crea un uno nuovo e crea l'oggetto di tipo **HttpSession** associato ad esso. Di fatto, viene creata una nuova sessione HTTP.



### E se i cookie sono disabilitati?

Si utilizza l'**URL encoding** che consente di riscrivere le URL in modo che esse contengano al loro interno anche il session ID.

Con l'URL encoding, il SESSIONID viene passato sempre come parametro nella request!



Per recuperare l'oggetto **HttpSession**, l'interfaccia **HttpServletRequest** mette a disposizione il metodo **getSession()**.

```
HttpSession session = request.getSession();
```

Il metodo **getSession()** restituisce l'oggetto di tipo **HttpSession** se esiste una sessione attiva, altrimenti ne crea una nuova.



## HTTPSession

I principali metodi utilizzati sono:

**getId()**: restituisce il **JSESSIONID** associato alla sessione.

**setAttribute(String name, Object value)**: consente di inserire un nuovo oggetto in sessione, associandolo al nome specificato.

**getAttribute(String name)**: restituisce l'oggetto associato al nome indicato o null se non esiste alcun oggetto.



## HttpSession

**getAttributeNames()**: restituisce un **enumeration** che contiene i nomi di tutti gli attributi contenuti nell'oggetto **HttpSession**.

**removeAttribute(String name)**: rimuove un attributo dalla session se esiste, altrimenti non fa nulla.

**getLastAccessedTime()**: restituisce l'ultimo istante temporale (il timestamp...) in cui il cliente ha inviato una request, espresso in millisecondi.



## HTTPSession

**setMaxInactiveInterval(int interval)**: consente di impostare i secondi di inattività trascorsi i quali la sessione viene terminata. Il valore di default è di *1800 secondi* (30 minuti). Se interval è negative (ad es. -1) la sessione non verrà mai invalidate automaticamente, ma dovrà essere invalidate mediante l'invocazione del metodo ***invalidate()***.

**getMaxInactiveInterval()**: restituisce i secondi di inattività ammessi.



## HTTPSession

```
Dispatcher.java  * "pagina1.jsp"
 9
10  * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
11 */
12 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
13     request.getServletContext().getRequestDispatcher("/jsp/header.jsp").include(request, response);
14     String pagina = request.getParameter("pagina");
15     if(pagina != null && !pagina.trim().equals("")) {
16         if(pagina.equals("1")) {
17             List<String> carrello = (List<String>) request.getSession().getAttribute("carrello");
18             if(carrello == null) {
19                 carrello = new ArrayList<String>();
20                 request.getSession().setAttribute("carrello", carrello);
21             }
22             String articolo = request.getParameter("articolo");
23             if(articolo != null && !articolo.trim().equals("")) {
24                 carrello.add(articolo);
25             }
26
27             request.getServletContext().getRequestDispatcher("/jsp/pagina1.jsp").include(request, response);
28         } else if(pagina.equals("2")) {
29             request.getServletContext().getRequestDispatcher("/jsp/pagina2.jsp").include(request, response);
30         } else if(pagina.equals("3")) {
31             request.getServletContext().getRequestDispatcher("/jsp/pagina3.jsp").include(request, response);
32         } else {
33             request.getServletContext().getRequestDispatcher("/jsp/body.jsp").include(request, response);
34         }
35     } else {
36         request.getServletContext().getRequestDispatcher("/jsp/body.jsp").include(request, response);
37     }
38     request.getServletContext().getRequestDispatcher("/jsp/footer.jsp").include(request, response);
39 }
```

```
Dispatcher.java  * pagina1.jsp
 1 <%@page import="java.util.List"%>
 2 <h1>Sono nel Body della Pagina 1!</h1>
 3
 4<%>
 5
 6     List<String> carrello = (List<String>) request.getSession().getAttribute("carrello");
 7     if(carrello != null && carrello.size() > 0) {
 8         for(String articolo: carrello) {
 9             out.println(articolo + "<br>");
10     }
11     } else {
12         out.println("Non ci sono articoli nel carrello!!!");
13     }
14
15 %>
```



# Java Servlet Filters

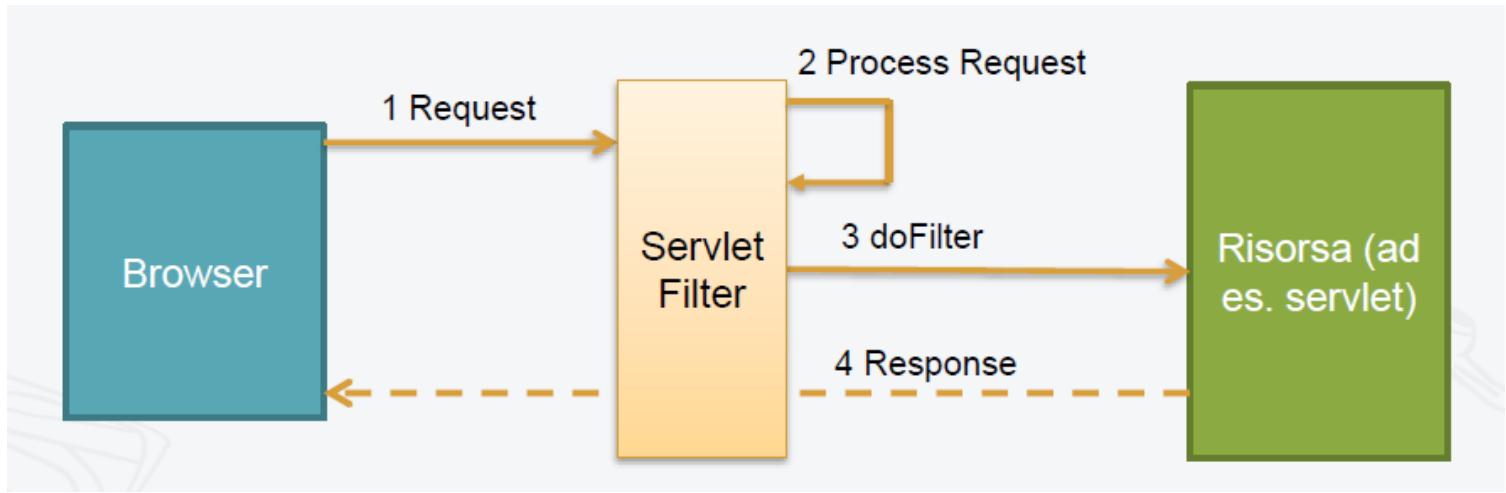


I filtri sono **oggetti di tipo javax.servlet.Filter** e rappresentano un potente meccanismo in grado di effettuare operazioni di **pre-processing** delle richieste e **post-processing** delle risposte.

I filtri intervengono, pertanto, prima che una richiesta raggiunga la servlet o appena dopo che la risposta esca dalla servlet.

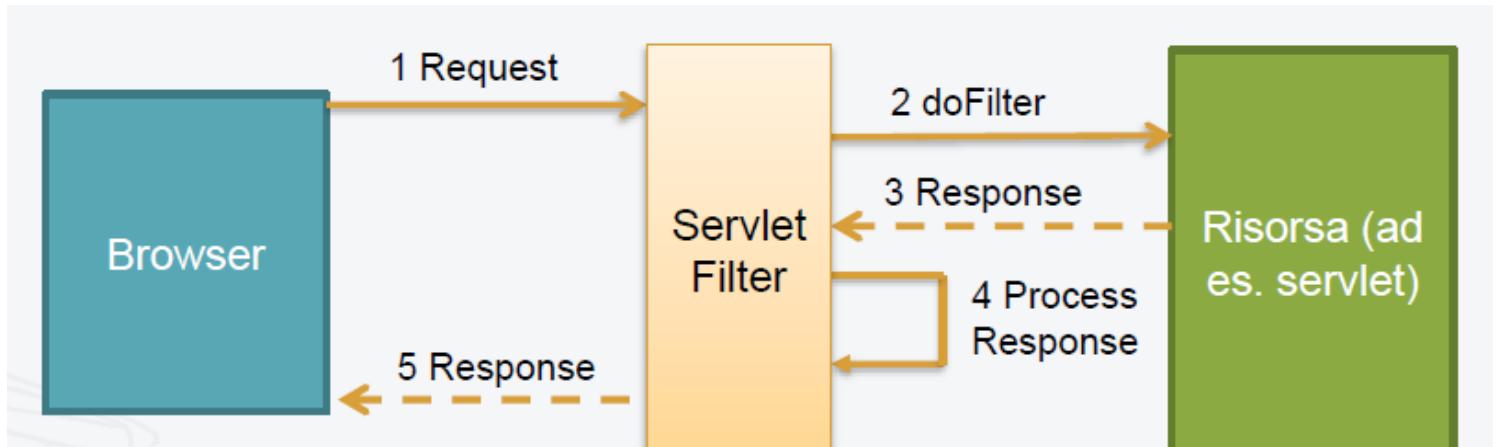


### Pre-processing





### Post-processing





# Come si implementa un filtro

I metodi definiti nell’interfaccia **javax.servlet.Filter**:

**init(FilterConfig filterConfig)**: invocato dal server, dopo la creazione dell’istanza del filtro e appena prima dell’attivazione dello stesso.

**doFilter(ServletRequest request, ServletResponse response, FilterChain chain)**: invocato dal container, contiene tutte le operazioni che dovremo effettuare quando il filtro intercetta una URL.

**destroy()**: invocato dal container terminare il ciclo di vita di un filtro.



## Java Servlet Filters

The screenshot shows the Eclipse IDE interface with the title "java - web-app/src/Dispatcher.java - Eclipse". The left sidebar displays the Project Explorer with a "src" folder containing "Dispatcher.java" and "pagina1.jsp". The main editor window shows the code for "Dispatcher.java". A context menu is open over the code, specifically over the line "protected void doGet(HttpServletRequest request, HttpServletResponse response) {". The context menu is titled "Dispatcher.java" and includes options like "New", "Go Into", "Open Type Hierarchy", "Copy", "Paste", "Delete", "Remove from Context", "Build Path", "Source", "Refactor", "Import...", "Export...", and "Refresh". A submenu is open under "New", listing "Project...", "Annotation", "Class", "Enum", "Interface", "Package", "HTML File", "JSP File", "Filter", "Listener", "Servlet", "Example...", and "Other...". To the right of the code editor, the generated Java code for a filter is shown:

```
public void doFilter(ServletRequest request, ServletResponse response) throws IOException, ServletException {
    HttpServletRequest req = (HttpServletRequest) request;
    req.setAttribute("log", "sono nel filtro");
    chain.doFilter(request, response);
}
```



# Java Servlet Listeners



I **ServletListeners** sono oggetti Java che vengono messi in ascolto su un particolare evento. Quando si verifica l'evento, intervengono per eseguire una serie di azioni.

Ogni **ServletListener** è associato a un tipo di evento (ad es. inizializzazione di una Servlet, aggiunta di un attributo alla request, etc...).



## Java Servlet Listeners

Ci sono 8 tipi di **listener** disponibili:

**ServletContextListener**: è in ascolto sull'evento **SessionContextEvent** che notifica quando una **Servlet** viene inizializzata o distrutta.

**ServletContextListener** è un'interfaccia che definisce due metodi:

- **contextDestroyed(ServletContextEvent e)**, eseguito quando l'applicazione viene distrutta
- **contextInitialized(ServletContextEvent e)**, eseguito quando l'applicazione viene inizializzata



## Java Servlet Listeners

**ServletContextAttributeListener:** è in ascolto sull'evento **SessionContextAttributetEvent** che notifica quando un oggetto viene aggiunto, rimosso o sostituito dalla **Servlet** (siamo a livello di applicazione non di richiesta!).

**ServletContextAttributeListener** è un'interfaccia che definisce tre metodi:

- **attributeAdded(ServletContextAttributeEvent e)**, eseguito quando un nuovo attributo viene aggiunto al servlet context.
- **attributeRemoved(ServletContextAttributeEvent e)**, eseguito quando un attributo viene rimosso dal servlet context.
- **attributeReplaced(ServletContextAttributeEvent e)**, eseguito quando un attributo viene sostituito nel servlet context.



## Java Servlet Listeners

**HttpSessionListener:** è in ascolto sull'evento **HttpSessionEvent** che notifica quando la sessione viene creata o distrutta.

**HttpSessionListener** è un'interfaccia che definisce due metodi:

- **sessionDestroyed(HttpSessionEvent e)**, eseguito quando la sessione viene distrutta
- **sessionCreated(HttpSessionEvent e)**, eseguito quando la sessione viene creata



## Java Servlet Listeners

**HttpSessionAttributeListener:** è in ascolto sull'evento **HttpSessionBindingEvent** che notifica quando un oggetto viene aggiunto, rimosso o sostituito dalla sessione.

**HttpSessionAttributeListener** è un'interfaccia che definisce tre metodi:

- **attributeAdded(HttpSessionBindingEvent e)**, eseguito quando un attributo viene aggiunto alla sessione.
- **attributeRemoved(HttpSessionBindingEvent e)**, eseguito quando un attributo viene rimosso dalla sessione.
- **attributeReplaced(HttpSessionBindingEvent e)**, eseguito quando un attributo viene sostituito nella sessione.



## Java Servlet Listeners

**ServletRequestListener**: è in ascolto sull'evento **ServletRequestEvent** che notifica quando viene creata o distrutta una request.

**ServletRequestListener** è un'interfaccia che definisce due metodi:

- **requestDestroyed(ServletRequestEvent e)**, eseguito quando la richiesta viene distrutta
- **requestInitialized(ServletRequestEvent e)**, eseguito quando la richiesta viene inizializzata



## Java Servlet Listeners

**ServletRequestAttributeListener:** è in ascolto sull'evento **ServletRequestAttributeEvent** che notifica quando un oggetto viene aggiunto, rimosso o sostituito dalla request.

**ServletRequestAttributeListener** è un'interfaccia che definisce tre metodi:

- **attributeAdded(ServletRequestAttributeEvent e):** eseguito quando viene aggiunto un attributo alla request.
- **attributeRemoved(ServletRequestAttributeEvent e):** eseguito quando un attributo viene rimosso dalla request.
- **attributeReplaced(ServletRequestAttributeEvent e):** eseguito quando un attributo viene sostituito sulla request.



## Java Servlet Listeners

**HttpSessionActivationListener:** è un listener che ascolta una sessione ed intercetta quando l'oggetto session migra da una macchina virtuale all'altra.

Questo listener è importante quando la web application viene deployata in ambiente distribuito (ad es. un cluster di server).



**HttpSessionBindingListener:** questo listener viene utilizzato per notificare ad un oggetto che è stato aggiunto o rimosso dalla sessione. La differenza con l'**HttpSessionAttributeListener** è che quest'ultimo viene invocato quando un oggetto viene aggiunto, rimosso o sostituito in sessione, mentre

**HttpSessionBindingListener** invia una notifica all'oggetto interessato dall'azione. Per avere efficacia, gli oggetti aggiunti alla sessione devono essere classi che implementano **HttpSessionBindingListener**.

# Java Servlet Listeners



java - Eclipse

File Edit Navigate Search Project Run Window Help

Project Explorer

- web-app
  - Deployment Descriptor: web-app
  - JAX-WS Web Services
  - Java Resources
    - src
      - (default package)
        - Dispatcher.java
        - LoginFilter.java
        - servlet.java
    - Libraries
    - JavaScript Resources
    - WebContent
      - jsp
        - body.jsp
        - footer.jsp
        - header.jsp
        - home.jsp
        - pagina1.jsp
        - pagina2.jsp
        - pagina3.jsp
    - META-INF
    - WEB-INF
    - index.html
  - Servers

Create Listener

Select the application lifecycle events to listen to.

Servlet context events

- Lifecycle *javax.servlet.ServletContextListener*
- Changes to attributes *javax.servlet.ServletContextAttributeListener*

HTTP session events

- Lifecycle *javax.servlet.http.HttpSessionListener*
- Changes to attributes *javax.servlet.http.HttpSessionAttributeListener*
- Session migration *javax.servlet.http.HttpSessionActivationListener*
- Object binding *javax.servlet.http.HttpSessionBindingListener*

Servlet request events

- Lifecycle *javax.servlet.ServletRequestListener*
- Changes to attributes *javax.servlet.ServletRequestAttributeListener*

Select All  Deselect All

?  < Back  Next >  Finish  Cancel



# Java Servlet Listeners

```
AttributoRequestListener.java ✘ Dispatcher.java  
1  
2  
3④ import javax.servlet.ServletRequestAttributeEvent;  
4  
5⑤ /**  
6   * Application Lifecycle Listener implementation class AttributoRequestListener  
7   */  
8  
9  @WebListener  
10 public class AttributoRequestListener implements ServletRequestAttributeListener {  
11  
12    * Default constructor. ◻  
13    public AttributoRequestlistener() {}  
14  
15    * @see ServletRequestAttributeListener#attributeRemoved(ServletRequestAttributeEvent)  
16    public void attributeRemoved(ServletRequestAttributeEvent arg0) {  
17      System.out.println("Oggetto rimosso dalla Request..." + arg0.getName());  
18    }  
19  
20    * @see ServletRequestAttributeListener#attributeAdded(ServletRequestAttributeEvent)  
21    public void attributeAdded(ServletRequestAttributeEvent arg0) {  
22      System.out.println("Oggetto aggiunto dalla Request..." + arg0.getName());  
23    }  
24  
25    * @see ServletRequestAttributeListener#attributeReplaced(ServletRequestAttributeEvent)  
26    public void attributeReplaced(ServletRequestAttributeEvent arg0) {  
27      System.out.println("Oggetto modificato dalla Request..." + arg0.getName());  
28  }
```

```
AttributoRequestListener.java ✘ Dispatcher.java ✘  
28①  /**  
29   * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)  
30   */  
31②  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
32  
33    request.getServletContext().getRequestDispatcher("/jsp/header.jsp").include(request, response);  
34    String pagina = request.getParameter("pagina");  
35    if(pagina != null && !pagina.trim().equals("")) {  
36      if(pagina.equals("1")) {  
37        request.setAttribute("oggetto", "test");  
38  
39        List<String> carrello = (List<String>).request.getSession().getAttribute("carrello");  
40        if(carrello == null) {  
41          carrello = new ArrayList<String>();  
42          request.getSession().setAttribute("carrello", carrello);  
43        }  
44        String articolo = request.getParameter("articolo");  
45        if(articolo != null && !articolo.trim().equals("")) {  
46          carrello.add(articolo);  
47        }  
48  
49        request.removeAttribute("oggetto");  
50        request.getServletContext().getRequestDispatcher("/jsp/pagina1.jsp").include(request, response);  
51      } else if(pagina.equals("2")) {  
52        request.getServletContext().getRequestDispatcher("/jsp/pagina2.jsp").include(request, response);  
53      } else if(pagina.equals("3")) {  
54        request.getServletContext().getRequestDispatcher("/jsp/pagina3.jsp").include(request, response);  
55      } else {  
56        request.getServletContext().getRequestDispatcher("/jsp/body.jsp").include(request, response);  
57      }  
58    }  
59  }
```

Oggetto aggiunto dalla Request...oggetto  
Oggetto rimosso dalla Request...oggetto



# JSP: Java Server Pages



## JSP: Java Server Pages

Le **Java Server Pages (JSP)** sono una tecnologia importantissima nello sviluppo di applicazioni web.

La principale caratteristica di una pagina **JSP** è che è costituita da un mix tra codice **HTML** e codice **Java**.

In sostanza, attraverso le JSP possiamo creare pagine che generano **dinamicamente codice HTML!**

Un file **JSP** deve avere estensione **.jsp**.

Il codice Java è racchiuso tra i simboli **<% e %>**



Come già anticipato, la caratteristica principale di una **JSP** è che può contenere un mix tra codice **HTML** e codice **Java**.

Il codice **Java**, per essere interpretato come tale, deve essere inserito all'interno dei simboli **<% %>**.

Questo blocco di codice Java viene detto **scriptlet**.



# Dichiarazione



Per **dichiarare** una variabile o un metodo all'interno di una JSP è necessario utilizzare la seguente sintassi:

### <% ! dichiarazione %>

```
<% ! String myVar = request.getParameter("myVar") %>

<% ! public String isPalindroma(String text){
    ...
    return ...;
}
%>
```

N.B. è sconsigliabile implementare la logica nelle pagine jsp



# Espressione



## JSP: Java Server Pages

Per inserire un'**espressione** all'interno di una JSP, la sintassi è:

**<%= espressione %>**

```
<% ! String myVar = request.getParameter("myVar") %>

<% ! public String isPalindroma(String text){
    ...
    return ...;
}
%>
...
<p>La parola è palindroma? <b><%=isPalindroma(myVar) %></b></p>
```

N.B. è sconsigliabile implementare la logica nelle pagine jsp



# Direttive



## JSP: Java Server Pages

Le **direttive** consentono di specificare alcune caratteristiche necessarie per il corretto funzionamento della pagina e sono:

**<%@ page**: consente di definire alcune impostazioni relative alla compilazione della pagina attraverso i suoi attributi. I principali attributi sono:

**language** specifica il linguaggio da utilizzare in fase di compilazione

Esempio: `<%@ page language="Java" %>`

**import** specifica i package da includere nella pagina

Esempio: `<%@ page import="java.util.List" %>`

**isThreadSafe** se impostato a **true** indica che la pagina è in grado di servire più richieste contemporaneamente (cioè viene creato un oggetto per ogni richiesta):

Esempio: `<%@ page isThreadSafe="true" %>`



## JSP: Java Server Pages

**<% include**: consente di includere altri file (jsp, html o file di testo) alla pagina. Questa direttiva consente di creare porzioni di codice HTML e di includerle all'interno di più pagine.

Esempio: **<%@ include file="fileB.jsp" %>**

**<% taglib**: consente di utilizzare all'interno di una JSP dei custom tag.

Esempio: **<%@taglib**

***uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>***



# Azioni



Le **azioni** consentono di incapsulare al meglio il codice **Java** presente in una **JSP**.

Le azioni principali sono:



## JSP: Java Server Pages

**<jsp:useBean>**: permette di associare ad una variabile script un'istanza di un JavaBean.

Un **JavaBean** è una classe Java che ha le seguenti caratteristiche, tra cui:

- La classe deve avere un costruttore senza argomenti
- Le sue proprietà devono essere accessibili usando **get**, **set**, **is**(per le variabili booleane)
- La classe deve essere **serializzabile**, implementando l'interfaccia **java.io.Serializable**(in questo modo la classe è in grado di salvare e ripristinare il suo stato in modo persistente)

```
<jsp:useBean id="myvar" class="it.test.Persona" scope="request" />
```

è l'analogo di

```
Persona p = (Persona) request.getAttribute("myvar");
if(p == null) {
    p = new Persona();
    request.setAttribute("myvar", p);
}
```

myvar è una variabile utilizzabile all'interno della jsp.



## JSP: Java Server Pages

**<jsp:setProperty>**: consente di assegnare un valore ad un parametro di un metodo di un Bean;

```
<jsp:setProperty name="myvar" property="username" value="paolo.preite" />  
        Mario.Rossi
```

è l'analogo di

```
Persona p = (Persona) request.getAttribute("myvar");  
p.setUsername("paolo.preite");
```

**<jsp:getProperty>**: consente di recuperare il valore di un parametro di un Bean;

```
<jsp:getProperty name="myvar" property="username" />
```

è l'analogo di

```
Persona p = (Persona) request.getAttribute("myvar");  
p.getUsername();
```



## JSP: Java Server Pages

**<jsp:param>**: consente di dichiarare ed inizializzare variabili in una jsp.

```
<jsp:params>
  <jsp:param name="nomeParametro" value="valore">
</jsp:params>
```

**<jsp:include>**: consente di includere una jsp in un'altra jsp

```
<jsp:include page="mypage.jsp">
  <jsp:param name="myvar" value="valoremyvar" />
</jsp:include>
```

La variabile myvar sarà visibile all'interno della jsp mypage.jsp...



# Oggetto request Nelle JSP



## JSP: Java Server Pages

L'oggetto **request** è di tipo **HttpServletRequest** e consente di accedere alle informazioni presenti nella richiesta **HTTP**.

Alcuni metodi utilizzati nelle **jsp** sono:

- **request.getParameter(String param)**
- **request.getAttribute(String param)**
- **request.getSession().getAttribute(String param)**
- **request.setAttribute(String param, Object obj)**
- **request.getSession().setAttribute(String param, Object obj)**
- **request.getCookies()**



# Oggetto response e out Nelle JSP



## JSP: Java Server Pages

L'oggetto **response** è di tipo **HttpServletResponse** e consente di gestire i contenuti da inviare al client.

L'oggetto **out** è di tipo **JspWriter** e consente, attraverso i metodi **print(Object o)** e **println(Object o)** di stampare del testo (semplice o html) nella **jsp**.

### Esempio

```
<%
    out.print("testo di esempio");

    out.println("testo di esempio"); /* a differenza di print, questo metodo stampa il testo e va a capo.*/

    out.print("<p>Testo paragrafo</p>");

%>
```



# Oggetto session Nelle JSP



L'oggetto **session** è di tipo **HttpSession** e contiene tutte le informazioni relative alla sessione del client.

Alcuni metodi utilizzati nelle jsp sono:

- **session.getAttribute(String param)**
- **session.setAttribute(String param, Object obj)**

Alla sessione è possibile accedere anche attraverso l'oggetto **request**

- **request.getSession().getAttribute(String param)**
- **request.getSession().setAttribute(String param, Object obj)**



# Oggetto application Nelle JSP



## JSP: Java Server Pages

L'oggetto **application** è di tipo **ServletContext** e consente di accedere alle variabili disponibili a livello di applicazione (quindi variabili indipendenti dall'i-esima **request** e dalla sessione del client).

Tali variabili, sono accessibili, finché l'applicazione è attiva ed accessibili in ogni **pagina o servlet**.

**application.setAttribute(String param, Object obj):** consente di impostare una variabile a livello di applicazione

**application.getAttribute(String param):** consente di accedere ad una variabile impostata a livello di applicazione

**application.getRealPath(String path):** restituisce il percorso assoluto del path specificato (ad es. `application.getRealPath("test.jsp")` potrebbe restituire `c:/tomcat/webapps/myapp/pages/test.jsp`).



# Gestione degli ERRORI



## JSP: Java Server Pages

Java è un linguaggio in grado di gestire in maniera efficace le **eccezioni** che possono inficiare la corretta esecuzione di un software.

Analogamente a quanto accade per **Java**, anche nelle **jsp** è possibile gestire varie tipologie di errori.

### Errori di compilazione

Si verificano quando il codice **jsp** viene tradotto in **servlet** e riguardano l'errata scrittura del codice Java. In questo caso il server invia al client il codice di **errore 500**.

### Errori durante l'elaborazione della richiesta

Si verificano quando viene eseguita la **jsp**(o la **servlet**) invocata. Questi errori sono legati all'errata scrittura del codice, cioè un bug applicativo... (ad es. accesso ad una variabile non inizializzata).



Nello sviluppo di applicazioni web, la gestione dell'errore viene eseguita direttamente dalla **servlet** generata dalla compilazione della pagina **jsp**.

Il nostro compito è quello di creare una pagina jsp che si occupi di gestire l'errore, evitando di visualizzare all'utente il tipo di errore (***per ovvii motivi di sicurezza e di estetica***).



# Come si crea una pagina di errore

Una pagina di errore è una **jsp** in cui si utilizza la direttiva **page** con l'attributo **isErrorHandler** impostato a **true**.

```
errorPage.jsp
<html>
  <body>
    <%@ page isErrorPage="true" %>
    <p>Ops...si è verificato un errore ☺ </p>
  </body>
</html>
```

Se volete far vedere il messaggio di errore (*a vostro rischio e pericolo ;-*)), sappiate che in tutte le jsp configurate come pagine di errore, è disponibile l'oggetto **exception** che consente di stampare il messaggio di errore attraverso l'invocazione del metodo `getMessage()`

```
<%= exception.getMessage()%>
```



### Come si usa la pagina di errore

Per fare in modo che, in caso di errore, l'utente venga reindirizzato nella pagina di errore creata, nelle **jsp** è necessario utilizzare la direttiva **page** con l'attributo **errorCode** che specifica la **jsp** che dovrà essere visualizzata in caso di errore. **myPage.jsp**

```
myPage.jsp
<html>
<body>
<%@ page errorCode="errorPage.jsp" %>
...contenuto pagina jsp...
</body>
</html>
```



# Tag Library



## Tag Library

Uno dei principi utilizzati nello sviluppo di software è il **disaccoppiamento** delle logiche di presentazione, business e data.

**Le taglibrary sono dei componenti**, da utilizzare all'interno dello strato di presentation(nelle jsp), che consentono di implementare numerose funzionalità (ad es. formattazione di testi e date, iterazione di elementi, et...).

**Un custom tag è un componente XML personalizzato**  
(ad es. <c:foreach> ... </c:foreach>)



## Tag Library

I principali vantaggi nell'usare **taglibrary** e **custom tag** sono:

- Riutilizzo all'interno di più pagine web
- Utilizzo dei **tag XML based** al posto di puro codice Java nelle scriptlet
- Gestione di logiche applicative riutilizzabili
- Eleganza nella produzione del codice
- Semplicità d'uso anche da parte di non sviluppatori



## Tag Library

Possiamo avere due tipi di tag:

**Tag senza corpo**, non possono contenere tra i tag di apertura e chiusura altri tag

```
<jsp:useBean id="nomevariabile" class="it.test.ClasseBean" scope="request" />
```

**Tag con corpo**, possono contenere tra i tag di apertura e chiusura altri tag

```
<jsp:forward page="nuovaUrl">
    <jsp:param name="nome" value="valore"/>
</jsp:forward>
```



### Tag senza corpo

La classe Java utilizzata per creare un tag senza corpo estende la classe **TagSupport**.

Alcuni metodi messi a disposizione dalla classe **TagSupport**, che noi possiamo **sovrascrivere**, sono:

- **int doEndTag()** eseguito dopo tutti gli altri metodi, generalmente ritorna il valore **int EVAL\_PAGE**
- **int doStartTag()** eseguito prima di tutti altri metodi, ritorna il valore **int SKIP\_BODY**. Per i tag senza corpo questo è il metodo principale da implementare che dovrà contenere la logica applicativa del tag
- **Object getValue(String k)** recupera il valore di un attributo definito nel tag.



### Tag con corpo

La classe Java utilizzata per creare un tag senza corpo estende la classe **BodyTagSupport**(che estende **TagSupport**...).

Alcuni metodi messi a disposizione dalla classe **BodyTagSupport**, che noi possiamo sovrascrivere, sono:

- **void doInitBody()** invocato solo una volta, prima della valutazione del corpo del tag. Il metodo è usato per l'inizializzazione di variabili
- **int doAfterBody()** invocato dopo che il corpo del tag è stato valutato. Se questo metodo ritorna **EVAL\_BODY\_TAG**, il corpo viene nuovamente valutato, se ritorna **SKIP\_BODY** non viene effettuata la nuova valutazione del corpo del tag



# Il file TLD (Tag Library Descriptor)

Il **TLD** è un file **xml** che contiene la descrizione di utilizzo della classe dei **custom tag**.

Il file **.tld** deve essere inserito nella cartella **WEB-INF** della **web application**(generalmente si usa una sottocartella **tlds** che conterrà tutti i file **.tld**)



## Tag Library

I principali elementi del file sono:

**<taglib>**: è il nodo root del documento XML che conterrà tutti gli altri nodi

**<tlib-version>**: definisce la versione delle specifiche taglib utilizzate

**<jsp-version>**: definisce la versione delle specifiche jsp utilizzate

**<short-name>**: contiene un nome descrittivo della custom tag library

**<tag>**: definisce un tag e contiene al suo interno:

**<name>**: è il nome del tag che utilizzeremo nella jsp

**<tag-class>**: è la classe (completa di package) che implementa il tag

**<attribute>**: definisce un attributo associato al tag



Il nodo **<attribute>** contiene al suo interno:

**<name>**: specifica il nome dell'attributo. Dal punto di vista della classe Java, è necessario implementare un metodo set con il nome indicato in questo elemento.

**<required>**: specifica se l'attributo è obbligatorio

**<rteprvalue>**: specifica se il valore dell'attributo è statico o dinamico, cioè generato utilizzando un'espressione (cioè `<%= ... %>`).



Per i tag con corpo, è necessario specificare anche il seguente elemento:

**<body-content>JSP</body-content>**: in questo modo il contenuto presente tra i tag di apertura e chiusura sarà utilizzabile nella classe Java che implementa il tag.



Per usare una **taglib** nella **jsp** è necessario definire la libreria di tag da utilizzare

```
<%@ tagliburi="/WEB-INF/tld/nomefile.tld" prefix="mytag" %>
```

A questo punto possiamo utilizzare il tag:

```
<mytag:nomeTag nomeAttributo="valore" />
```



# JSP Standard Tag Library



## JSP Standard Tag Library

### Cosa sono le JSTL - JSP Standard Tag Library

È un insieme di **JSP tag**, che implementano le principali funzionalità che abbiamo bisogno di utilizzare quando realizziamo un'applicazione web, ad esempio:

- Manipolazione di oggetti e stringhe
- Iterazione di liste
- Visualizzazione di parti di html in base a determinate condizioni
- Internazionalizzazione dei contenuti



Le JSTL sono raggruppate in 5 categorie:

**core, formattazione, SQL, XML, funzione**

Dove non arrivano le **JSTL** potete arrivare voi realizzando i vostri custom tag!



# Configurazione della web app per l'utilizzo di JSTL

Prima di tutto è necessario scaricare il **jar** contenente tutto il pacchetto **JSTL**.

Per scaricare il **jar** contenente le **JSTL** possiamo:

- Andare sul sito: <https://tomcat.apache.org/taglibs/standard/>
- Utilizzare la dependency **Maven**.

Il file **jar** andrà messo dentro **WEB-INF/lib** mentre i file **.tld** andranno messi dentro **WEB-INF** (generalmente in **WEB-INF/tld**).



## JSP Standard Tag Library

Se apriamo il file jar(file compresso), nella cartella **META-INF** troveremo tutti i file **.tld** da utilizzare con il rispettivo **URI**

Nome oggetto Dimensioni

- ..
- maven
- c.tld
- c-1\_0-rt.tld
- c-1.tld
- DEPENDENCIES
- fmt.tld
- fmt-1\_0-rt.tld
- fn.tld
- LICENSE
- MANIFEST.MF
- NOTICE
- permittedTaglibs.tld
- scriptfree.tld
- sql.tld
- sql-1\_0-rt.tld
- x.tld
- x-1\_0-rt.tld

Visualizza - c.tld

File Modifica Visualizza ?

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Licensed to the Apache Software Foundation (ASF) under one or more
contributor license agreements. See the NOTICE file distributed with
this work for additional information regarding copyright ownership.
The ASF licenses this file to You under the Apache License, Version 2.0
(the "License"); you may not use this file except in compliance with
the License. You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
-->
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml
version="2.1"

<description>JSTL 1.2 core library</description>
<display-name>JSTL core</display-name>
<tlib-version>1.2</tlib-version>
<short-name>c</short-name>
<uri>http://java.sun.com/jsp/jstl/core</uri>

<validator>
    <description>
        Provides core validation functions for JSTL core
    </description>
```



### Configurazione delle JSP

Nelle pagine **JSP**, se vogliamo utilizzare i tag **JSTL**, dobbiamo utilizzare la direttiva `<%@ taglib` in cui si desidera utilizzare una **taglibrary**, per esempio la **core**, si deve usare la direttiva **taglib** indicando:

- l'**uri** (che servirà al container per recuperare dalla mappa tutti i tag handler)
- il **prefix** che sarà utilizzato nella jsp per scrivere i tag



Ad esempio, se vogliamo utilizzare nella jsp i tag **core**, dobbiamo scrivere:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

e poi potremo utilizzare i tag nel seguente modo:

```
<c:if test="">...</c:if>
```



## JSP Standard Tag Library

```
taglib.jsp
1 <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2     pageEncoding="ISO-8859-1"%>
3
4 <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
5
6 <!DOCTYPE html>
7<html>
8<head>
9 <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
10 <title>Insert title here</title>
11 </head>
12<body>
13
14     <c:out value="Prova Stampa JAR "></c:out>
15
16 </body>
17 </html>
```



### I tag core

**<c:out>**: utilizzato per visualizzare i risultati di un'espressione (è l'analogo di <%= ...%>)

**<c:out value="string a o espressione" default="valore se value è null" />**

**<c: set>**: utilizzato per impostare delle variabili con relativo valore.

**<c:set var="nome della variabile"  
value="valore della variabile"  
scope="request, session, application"/>**



### I tag core

**<c:remove>**: rimuove una variabile dallo scope impostato

**<c:remove var="nome della variabile" scope="request, session, application"/>**

**<c:if>**: è l'analogo dello statement if(senza else...). Se l'espressione restituisce true il tag esegue il codice contenuto nel suo body.

**<c:if test="espressione"**

**var="nome della variabile dov'è salvato il risultato del test"**

**scope="dove sarà salvata la variabile, request session...">**

**</c:if>**



### I tag core

`<c:choose>`, `<c:when>`, `<c:otherwise>`: sono l'analogo dello statement switch/case.

```
<c:choose>
  <c:when test="espressione da verificare"/> ...
  </c:when>
  <c:when test=" espressione da verificare"/> ...
  </c:when>
  ...
  <c:otherwise> ...
  </c:otherwise>
</c:choose>
```



### I tag core

**<c:import>**: è l'analogo di **<jsp: include>** solo che consente anche di inserire URL assolute (<http://www.miosito.it>).

```
<c:import  
    url="urlpath"  
    var="variabile dove verrà salvata la url indicata"  
    scope="scope della variabile, request, session...«  
    context="opzionale...simbolo / seguito dal nome di una web application locale" />
```



### I tag core

**<c:forEach>**: itera liste di oggetti.

**<c:forEach**

*items="lista di oggetti"*

*begin="<int>" //dove partire*

*end="<int>" //dove terminare*

*step="avanzamento della lista, default è 1"*

*var="nome della variabile dove viene salvato l'i-esimo oggetto della lista"*

*varStatus="variabile contenente la posizione in cui si trova l'iterazione">*

*...*

**</c:forEach>**



### I tag core

**<c:forTokens>**: consente di iterare una stringa, specificando il separatore (convertendo, quindi, la stringa in array di stringhe...)

```
<c:forTokens items="stringa da iterare"
  delims="separatore"
  begin=<int> end=<int> step="avanzamento della lista, default è 1"
  var="nome della variabile dove viene salvato l'i-esimo oggetto della lista"
  varStatus="variabile contenente la posizione in cui si trova l'iterazione">
...
</forTokens>
```



### I tag core

**<c:url>**: formatta una stringa in URL.

```
<c:url  
    var="nome della variabile che conterrà la URL generata"  
    scope="scope della variabile"  
    value="stringa da convertire in URL"  
    context="opzionale...simbolo / seguito dal nome di una web application locale"/>
```

**<c:param>**: utilizzato nel **<c:url>** per specificare eventuali parametri.

```
<c:param name="nome della variabile" value="valore salvato nella variabile"/>
```



### I tag format

I tag format consentono di effettuare formattazione di date, numeri etc...

Per utilizzare i tag format, nelle JSP dobbiamo inserire la seguente direttiva:

```
<%@ tagliburi="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
```



## JSP Standard Tag Library

**<fmt:formatNumber>**: consente di formattare i numeri (in valuta, percentuali,etc...).

**<fmt:formatNumber value="valore numerico da formattare"**

**type**=*"tipo di numero da generare: NUMBER, CURRENCY, or PERCENT"*

**pattern**=*"formato del numero stampato (ad es. ###.##)"*

**currencyCode**=*"codice della valuta da utilizzare per formattare il valore"*

**currencySymbol**=*"simbolo della valuta da utilizzare per formattare il valore"*

**maxIntegerDigits**=*"numero massimo di interi da stampare"*

**minIntegerDigits**=*"numero minimo di interi da stampare"*

**maxFractionDigits**=*"numero massimo di cifre decimali da stampare"*

**minFractionDigits**=*"numero minimo di cifre decimali da stampare"*

**var**=*"variabile dove salvare il valore formattato"*

**scope**=*"scope della variabile" />*



## JSP Standard Tag Library

### Alcuni valori per definire il pattern

<b>0</b>	Rappresenta una cifra
<b>E</b>	Rappresenta il formato esponenziale
<b>#</b>	Rappresenta una cifra. Visualizza 0 se non ci sono cifre disponibili
.	Separatore dei decimali
,	Separatori delle migliaia
-	Utilizzare il prefisso negativo di default
%	Visualizza il numero a multipli di 100 e in formato percentuale
?	Visualizza il numero a multipli di 1000 e in formato per mille
<b>¤</b>	Rappresenta il simbolo della valuta. Questo elemento viene sostituito con la valuta correntemente impostata



## JSP Standard Tag Library

**<fmt:parseNumber>**: utilizzato per convertire stringhe (che possono essere numeri, percentuali, valuta) in numeri.

**<fmt:parseNumber value="stringa contenente il numero da elaborare"**  
**type="tipo di numero da generare: NUMBER, CURRENCY, or PERCENT"**  
**pattern="formato del numero stampato (ad es. ###.##)"**  
**parseLocale="Locale da utilizzare quando si fa il parsing della stringa"**  
**integerOnly="se true indica che la stringa verrà convertita in intero"**  
**var="variabile dove salvare il valore formattato"**  
**scope="scope della variabile"/>**



**<fmt:formatDate>**: utilizzato per formattare le date.

**<fmt:formatDate value="oggetto Date da formattare"**  
**type="tipo di output atteso DATE, TIME, or BOTH"**  
**dateStyle="FULL, LONG, MEDIUM, SHORT, or DEFAULT"**  
**timeStyle="FULL, LONG, MEDIUM, SHORT, or DEFAULT"**  
**pattern="formato personalizzato di visualizzazione della data"**  
**var="variabile dove salvare il valore formattato"**  
**scope="scope della variabile"/>**



## JSP Standard Tag Library

### Alcuni valori per definire il pattern

y	Anno	2002
M	Mese	Aprile oppure 04
d	Giorno del mese	20
h	Ora del giorno (12-hour time)	12
H	Ora del giorno (24-hour time)	0
m	Minuti	45
s	Secondi	52
S	Millisecondi	970
E	Giorno della settimana	Martedì
D	Giorno dell'anno	180

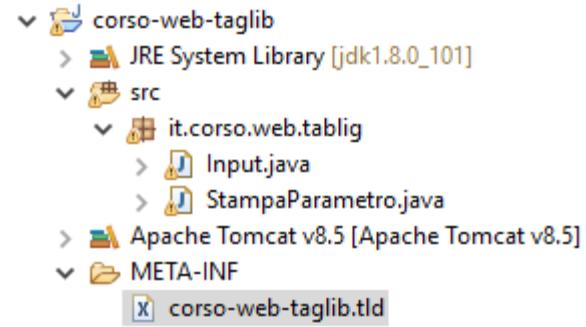


# Custom Tag Library



## Custom Tag Library

Per creare una **Custom Tag** ho bisogno di creare un nuovo progetto ed implementare le classi java dove ci sarà la logica della mia **Custon Tag Library**, infine creare un file tld che conterrà il codice XML della **taglib** personalizzata.





## Custom Tag Library

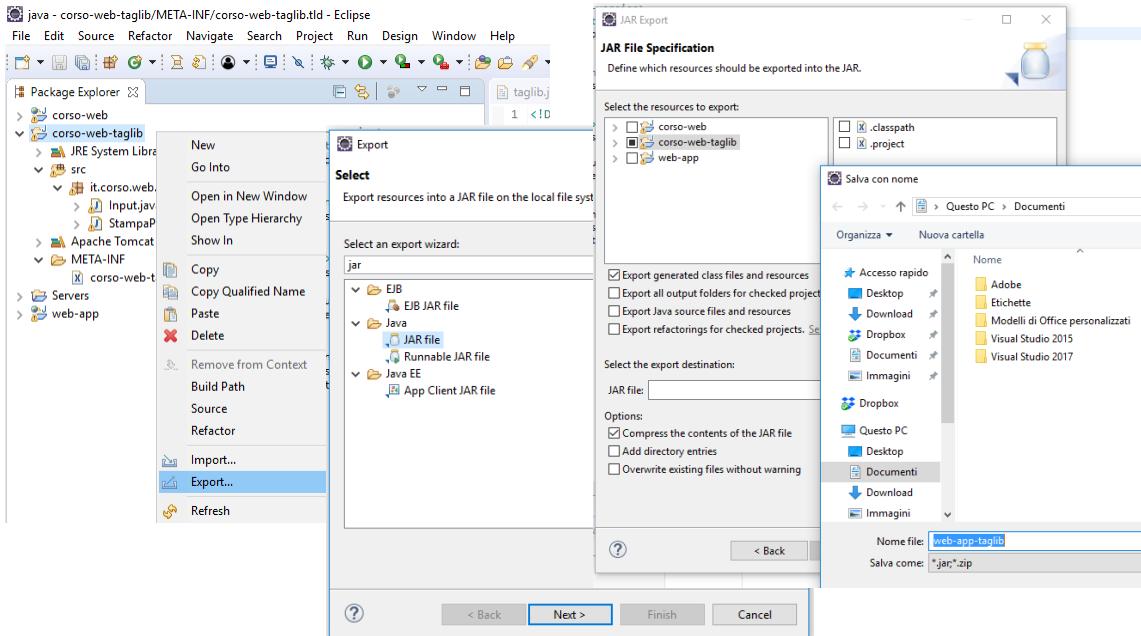
```
public class StampaParametro extends TagSupport {  
  
    @Override  
    public int doEndTag() throws JspException {  
        HttpServletRequest req = (HttpServletRequest)pageContext.getRequest();  
  
        try {  
            JspWriter writer = pageContext.getOut();  
            if(req.getParameter("utente") != null) {  
                writer.println("Ciao " + req.getParameter("utente"));  
            } else {  
                writer.print("Ciao, clicca qui per accedere!");  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return super.doEndTag();  
    }  
}
```

```
<!DOCTYPE taglib  
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"  
    "http://java.sun.com/j2ee/dtd/web-jsptaglibrary_1_2.dtd">  
<taglib  
    <tlib-version>1.0</tlib-version>  
    <jsp-version>1.2</jsp-version>  
    <short-name>web</short-name>  
    <uri>http://paolopreite.it/corso-web-taglib</uri>  
</taglib>  
<tag  
    <name>benvenuto</name>  
    <tag-class>it.corso.web.tablig.StampaParametro</tag-class>  
    </tag>  
<tag>  
    <name>input</name>  
    <tag-class>it.corso.web.tablig.Input</tag-class>  
    <attribute>  
        <name>nomeInput</name>  
        <required>true</required>  
    </attribute>  
    <attribute>  
        <name>valoreInput</name>  
        <required>false</required>  
        <rtextrvalue>true</rtextrvalue>  
    </attribute>  
    </tag>  
</taglib>
```



## Custom Tag Library

Creata una  
**Custom Tag**  
ho bisogno di  
esportarla in  
un file **.jar** poi  
importarla  
nei progetti.





# **JPA - Java Persistence API per l'interazione con i Database**



### Cos'è ORM?

L'**Object-Relational Mapping** (ORM) è una tecnica di programmazione che consente di gestire l'accesso in **scrittura/lettura** ad un database.

Attraverso l'**ORM** è possibile mappare tavole e record di un database relazionale attraverso oggetti software, nel caso di Java attraverso classi e oggetti.

L'**ORM** mette a disposizione tutte le operazioni per la persistenza dei dati, (le operazioni di **CRUD**).



Alcuni vantaggi nell'uso dell'ORM:

- **Riduzione del codice sorgente:** l'ORM implementa dei metodi che eseguono le operazioni di CRUD (non scriviamo più INSERT INTO ...)
- **Portabilità del software:** cambiando DBMS non devo riscrivere il software. Il CRUD è gestito a livello software. L'ORM si occupa per noi di convertire le operazioni in query SQL sulla base del DBMS utilizzato
- **Riduzione delle attività di test sulle operazioni di CRUD:** non scrivendo una query intera, l'errore a livello SQL è praticamente assente.
- **Caching dei dati:** l'ORM utilizza sistemi di cache che consentono di evitare l'accesso al DB ogni volta, migliorando le performance applicative



### Cos'è JPA?

**Java Persistence API (JPA)** è la specifica di JEE che consente di gestire la persistenza, l'accesso e la gestione dei dati con i database relazionali.

JPA include le funzionalità di un ORM e ne aggiunge di nuove!

JPA, come gli altri framework ORM, è un livello di astrazione al di sopra delle API JDBC.



Alla base di JPA c'è il concetto di **Entity** che ha le seguenti caratteristiche:

- È una normale classe Java che viene mappata su una o più tabelle
- Le variabili di istanza della classe vengono mappate con i campi delle tabelle
- Ogni istanza della classe corrisponde ad un record del database; i valori delle variabili di istanza sono i valori salvati sul database.



JPA può essere utilizzato in due modi:

- **Creazione di nuove strutture dati:** in questo caso, gli entity rappresentano la nuova struttura dati che verrà creata nel database.
- **Mappatura di un database esistente:** in questo caso, gli entity vengono mappati su un database esistente.



Per creare un Entity dobbiamo utilizzare obbligatoriamente due **annotation**:

**@Entity**: applicato sulla definizione della classe, indica che questa è mappata su una o più tabelle di un DB

**@Id**: indica quale variabile di istanza deve essere usata come primary key. Questa annotation può essere inserita sulla variabile di istanza o sul metodo get della variabile.

- se applichiamo **@Id** sul getter, tutte le altre annotation che interessano gli attributi della classe devono essere applicate sui relativi metodi get. (consigliato)
- se applichiamo **@Id** direttamente sulla variabile di istanza, stiamo indicando che è possibile accedere direttamente alle variabili di istanza. In quest'ultimo la variabile deve essere public ed è possibile omettere getter/setter.



## JPA, di default assume che:

- Il nome della tabella del DB coincide con il nome dell'Entity
- La tabella ha un campo per ciascuna variabile di istanza

Possiamo modificare questo comportamento di default, personalizzando il mapping sulla tabella e sulle sue colonne, attraverso le annotation **@Table** e **@Column**:

- **@Table(name="nomeTabella")**
- **@Column(name="nomeCampo")**

### Esempio

```
@Entity  
@Table(name="fattura")  
public class Fattura implements Serializable {  
    private long id;  
    private String numero;  
    private Date data;  
  
    @Id  
    public long getId() {  
        return this.id;  
    }  
  
    public void setId(long id) {  
        this.id = id;  
    }  
  
    @Column(name="numero_fattura")  
    public String getNumero() {  
        return this.numero;  
    }  
    ...  
}
```



## JPA - Java Persistence API

@Column può contenere altri attributi:

Attributo	Descrizione
<b>nullable</b>	Se impostato a false, indica che la colonna non ammette valori null (default=true).
<b>length</b>	Si può utilizzare per i tipi stringa ed indica la lunghezza massima.
<b>columnDefinition</b>	Se utilizzato, indica il tipo relazionale sulla colonna. (es. columnDefinition="VARCHAR(40)")
<b>precision e scale</b>	Si può utilizzare per personalizzare i numeri in virgola mobile.
<b>insertable e updatable</b>	Se impostati a false, indica che la colonna non deve essere inserita negli statement di insert e update (default=true).

### Esempio

```
@Entity  
@Table(name="fattura")  
public class Fattura implements Serializable {  
    private long id;  
    private String numero;  
    private Date data;  
  
    @Id  
    public long getId() {  
        return this.id;  
    }  
  
    public void setId(long id) {  
        this.id = id;  
    }  
  
    @Column(name="numero_fattura", nullable="false")  
    public String getNumero() {  
        return this.numero;  
    }  
    ...  
}
```



## Cos'è un DataSource

JEE mette a disposizione deli sviluppatori uno strumento molto utile che consente di gestire le connessioni verso il database.

**Questo strumento è il pool di connessioni (o data source).**



L'interfaccia Java che rappresenta un pool di connessioni si chiama **javax.sql.DataSource**.

Il ciclo di vita di un data source (attivazione e chiusura delle connessioni verso il database) è gestito dal server.

Attraverso il data source non dobbiamo preoccuparci di aprire la connessione al database e di chiuderla.



**ATTENZIONE:** un data source per funzionare ha comunque bisogno di utilizzare dei connettori verso i database (ad esempio, se create un data source per MySQL, dovete comunque utilizzare il **jdbc.jar** di MySQL)



In TomEE un DataSource si può dichiarare definendo un element **Resource** nel file **<tomee-home>/conf/tomee.xml** o nel file della web application **WEB-INF/resources.xml**

Nome	Ultima modifica
Catalina	23/09/2018 18:39
conf.d	23/09/2018 18:39
catalina.policy	20/06/2018 20:51
catalina.properties	20/06/2018 20:51
context.xml	20/06/2018 20:51
jaspic-providers.xml	20/06/2018 20:51
jaspic-providers.xsd	20/06/2018 20:51
logging.properties	02/09/2018 22:00
server.xml	02/09/2018 22:00
server.xml.original	02/09/2018 22:00
system.properties	02/09/2018 22:00
tomcat-users.xml	02/09/2018 22:00
tomcat-users.xml.original	02/09/2018 22:00
tomcat-users.xsd	20/06/2018 20:51
tomee.xml	02/09/2018 22:00
web.xml	02/09/2018 22:00



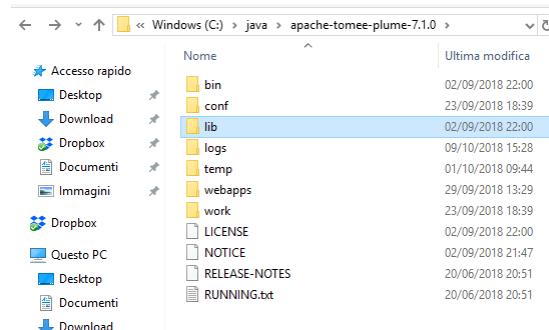
Qui trovate le configurazioni base di altri **datasource**(Oracle, OracleXA, PostgreSQL, ...)

```
<Resource id="ConnessioneMySQL" type="DataSource">
    JdbcDriver      com.mysql.jdbc.Driver
    JdbcUrl        jdbc:mysql://localhost/test
    UserName        test
    Password        test
</Resource>
```

<http://tomee.apache.org/common-datasource-configurations.html>



Scarichiamo ed installiamo un **database mysql**, scarichiamo ed istalliamo il **mysql-connector** nella cartella **lib** di tomee



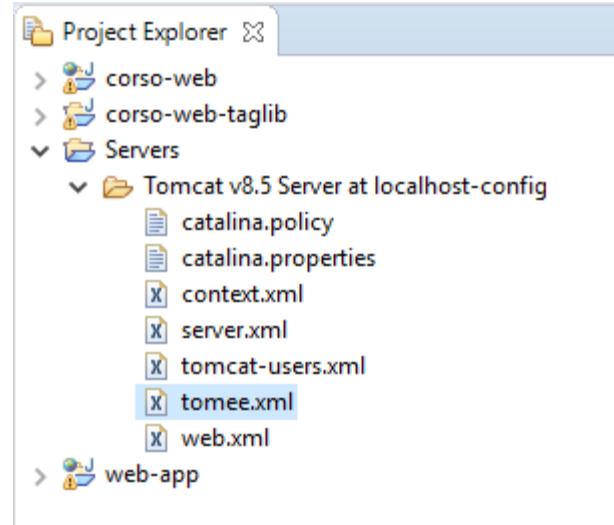


Apriamo ora la cartella **conf** e selezioniamo il file file **tomee.xml**

Windows (C:) > java > apache-tomee-plume-7.1.0		
	Nome	Ultima
pido	bin	02/09/2
d	conf	23/09/2
	lib	11/10/2
	logs	09/10/2
nti	temp	01/10/2
i	webapps	29/09/2
	work	23/09/2
:	LICENSE	02/09/2
	NOTICE	02/09/2
nti	RELEASE-NOTES	20/06/2
d	RUNNING.txt	20/06/2



Copiamo il file di configurazione **tomee.xml** nella cartella Tomcat di Servers presente in Eclipse





## JPA - Java Persistence API

```
1 <?xml version="1.0" encoding="UTF-8"?>
2<tomee>
3<Resource id="MySQLDatabase" type="DataSource">
4
5   JdbcDriver com.mysql.jdbc.Driver
6   JdbcUrl jdbc:mysql://localhost/persona
7   UserName root
8
9 </Resource>
10</tomee>
--
```



### Cos'è l'EntityManager

L'interfaccia **Entity Manager** si occupa di:

- persistenza delle entità
- effettuare le operazioni CRUD sulle entità

L'interfaccia Entity Manager è una componente fondamentale delle Java Persistence API perché consente di trasformare oggetti Java in record e viceversa, costituendo quindi un ponte la componente Object-Oriented e quella relazionale.



### Cos'è l'unità di persistenza?

L'unità di persistenza consente di specificare quale **data source** vogliamo utilizzare, le modalità di caching ed altre informazioni.

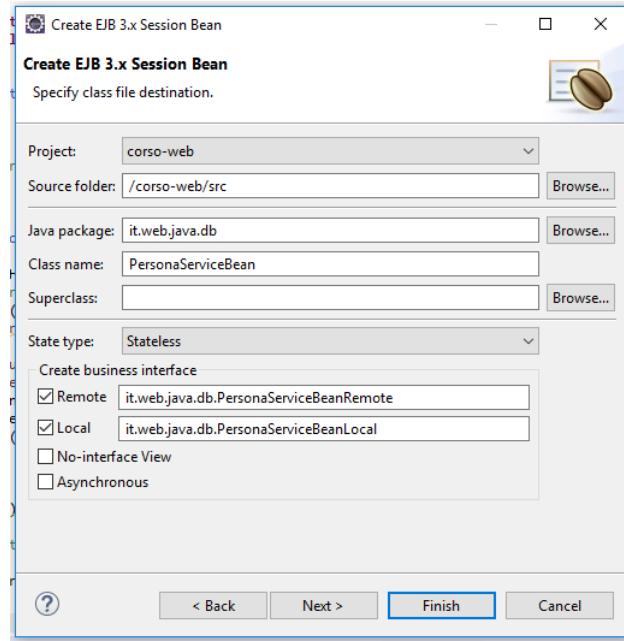
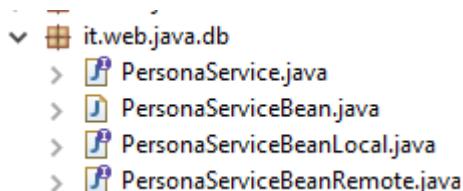
È definita nel file **persistence.xml**.

Il **persistence.xml** è un file di configurazione standard di JPA; deve essere inserito nella cartella WEB-INF del progetto web che contiene gli Entity

```
1<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="1.0">
2  <persistence-unit name="MySQLDatabase">
3    <jta-data-source>connessioneCorsoWeb</jta-data-source>
4
5    <properties>
6      <property name="openejb.jpa.auto-scan" value="true" />
7    </properties>
8  </persistence-unit>
9</persistence>
```



Creo ora un file **Session Bean** con due interfacce **Remote** e **Local** che estendono una nuova interfaccia **PersonaService**





# Aggiungere entità

Il metodo messo a disposizione dall'**Entity Manager** per l'inserimento di una nuova entità è **persist(Object o)**.

## Esempio

```
@Stateless  
public class ClienteBean implements ClienteLocal {  
    @PersistenceContext(unitName="nomeunita")  
    EntityManager em;  
  
    public void addCliente(Cliente c) {  
        em.persist(c);  
    }  
}
```



## Recuperare entità

Con JPA il modo più semplice per recuperare un'entità dal database, è attraverso la sua chiave primaria tramite il metodo **find(Class<T> entityClass, Object primaryKey)** dell'EntityManager:

- il primo parametro indica il tipo dell'entità
- il secondo parametro indica il valore della primary key dell'istanza da recuperare.

Se non esiste alcuna istanza associata alla chiave primaria specificata, il metodo restituisce **null**.



# Cancellare entità

Il metodo messo a disposizione dall'**Entity Manager** per la cancellazione di una entità è **remove(Object o)**.

## Esempio

```
@Stateless  
public class ClienteBean implements ClienteLocal {  
    @PersistenceContext(unitName="homeunita")  
    EntityManager em;  
  
    public void deleteCliente(Cliente c) {  
        em.remove(c);  
    }  
}
```



# Aggiornare entità

L'**EntityManager** mette a disposizione il metodo **merge(Object o)** per l'aggiornamento di un'entità.

## Esempio

```
@Stateless  
public class ClienteBean implements ClienteLocal {  
    @PersistenceContext(unitName="homeunita")  
    EntityManager em;  
  
    public void aggiornaCliente(Cliente c) {  
        em.merge(c);  
    }  
}
```



## Propagazione delle operazioni ad Entity collegate

Il comportamento standard dell'**Entity Manager** prevede che in presenza di relazioni tra più entità (**@OneToMany**, **@ManyToMany**, etc...) non venga effettuata alcuna operazione sulle entità collegate.

Se vogliamo propagare un'operazione su un entità (persist, merge, remove) anche alle entità collegate, nell'annotation che specifica la relazione, dobbiamo impostare il parametro **cascade** .



- **CascadeType.PERSIST**: propaga le operazioni di inserimento anche alle entità collegate
- **CascadeType.REMOVE**: propaga le operazioni di cancellazione anche alle entità collegate
- **CascadeType.MERGE**: propaga le operazioni di aggiornamento anche alle entità collegate
- **CascadeType.ALL**: propaga tutte le operazioni anche alle entità collegate

### Esempio

```
@Entity  
public class Utente {  
    @ManyToOne(cascade=CascadeType.MERGE)  
    public Account getAccount() { ... }  
}
```



## Mappare Entity su più tavole con JPA

Ci sono casi in cui abbiamo Entity che hanno attributi che devono essere mappati su più tavole.

### Esempio

- fattura(id\_fattura, numero, data\_fattura)
- cliente(id\_cliente, nome, cognome, email)

L'annotation **@SecondaryTable** specifica il nome della seconda tabella e della colonna (della seconda tabella) da usare per la Join tra le tavole.

Nell'annotation **@Column** dobbiamo specificare il parametro **table** per associare il campo alla seconda tabella.

### Esempio

```
@Entity  
@Table(name="fattura")  
@SecondaryTable(name="cliente",  
pkJoinColumns={@PrimaryKeyJoinColumn(name="id_cliente")})  
public class Fattura implements Serializable {  
    private Long idFattura;  
    private String numero;  
    private String dataFattura;  
    private String cognome;  
  
    @Column(name="cognome", table="cliente")  
    public String getCognome(){...}  
    ...  
}
```



Abbiamo visto che per definire più Entity dobbiamo usare l'annotation **@Entity**.

Quando progettiamo un database, quasi sicuramente avremo tavole legate tra loro. La relazione si gestisce attraverso la definizione delle **foreign key**.



Possiamo avere le seguenti relazioni:

- **Uno a uno**: a ciascun record di una tabella può essere associato un solo record di un'altra tabella
- **Uno a molti**: a ciascun record di una tabella possono essere associati più record di un'altra tabella
- **Molti a molti**: più record di una tabella sono associati a più record di un'altra tabella

Le stesse relazioni possono essere specificate anche con JPA, attraverso l'utilizzo di opportune **annotations**.



## Relazioni One-To-One

La relazione uno a uno è la più semplice e dice che ad un oggetto corrisponde solo un altro oggetto.

```
@Entity  
@Table(name="utente")  
public class Utente {  
    private Long id;  
    private String nome; ...  
    private Account account;  
  
    @Id  
    @Column(name="id_utente")  
    public long getId() {  
        return id;  
    }  
  
    @OneToOne  
    @JoinColumn(name="account_id")  
    public Account getAccount() {  
        return account;  
    }  
}
```

### Relazione bidirezionale

Dobbiamo inserire l'annotation `@OneToOne` su tutti e due gli Entity, definiamo una.

Vuol dire che da Utente posso sapere qual è il suo account e viceversa!

L'Entity Utente detiene il vincolo relazionale di foreign key (perché dica qual è la colonna che farà da foreign key).

### Relazione unidirezionale

Dobbiamo inserire le annotation `@OneToOne` e `@JoinColumn` solo su un Entity (ad es. solo su Utente o su Account).

### Relazione bidirezionale

```
@Entity  
@Table(name="account")  
public class Account {  
    private Long id;  
    private String username;  
    private String password;  
  
    private Utente utente;  
  
    @Id  
    @Column(name="id_account")  
    public long getId() {  
        return id;  
    }  
  
    @OneToOne(mappedBy="account")  
    public Utente getUtente() {  
        return utente;  
    }  
}
```



## One-To-Many-Many-To-One

Nella relazione uno a molti, un **EntityA** può avere più **EntityB** associate.

Nella relazione molti a uno, più **EntityA** possono essere collegate a un **EntityB**.

Ad esempio, un utente può avere più conti correnti bancari ed allo stesso tempo, più conti correnti possono appartenere ad un utente.



## One-To-Many-Many-To-One

A livello di database, se abbiamo due tavole **A** e **B**, questa relazione consiste nel creare una **foreign key** sulla tabella **B**, specificando che la **foreign key** può contenere valori duplicati. Per realizzare la relazione uno a molti in JPA, dobbiamo usare le annotation

**@OneToMany**

**@ManyToOne**

**@JoinColumn**.



## JPA - Java Persistence API

Tornando all'esempio (un utente può avere più conti correnti bancari), per rappresentare le strutture, dobbiamo creare le Entity **Utente** e **ContoCorrente**, usando le annotation come segue.

```
@Entity  
@Table(name="utente")  
public class Utente {  
    private Long id;  
    private String nome; ...  
    private List<ContoCorrente> conti;  
    @Id  
    @Column(name="id_utente")  
    public long getId() {  
        return id;  
    }  
    @OneToMany(mappedBy = "utente",  
               cascade = CascadeType.ALL)  
    public List<ContoCorrente> getContiCorrenti() {  
        return conti;  
    }  
}
```

### Relazione bidirezionale

Per ottenere una relazione bidirezionale dobbiamo aggiungere l'annotation `@OneToOne` sull'Entity Utente, specificando nel parametro `mappedBy`, il nome della variabile di istanza della classe ContoCorrente che viene usato per gestire la foreign key.

```
@Entity  
@Table(name="conto_corrente")  
public class ContoCorrente {  
    private Long id;  
    private String numero; ...  
    private Utente utente;  
    @Id  
    @Column(name="id_conto")  
    public long getId() {  
        return id;  
    }  
    @ManyToOne()  
    @JoinColumn(name = "id_utente")  
    public Utente getUtente() {  
        return utente;  
    }  
}
```



## Many-To-Many

Consideriamo il seguente caso: uno studente può iscriversi a più corsi e ad un corso possono iscriversi più persone.

In questo caso abbiamo una relazione **molti a molti** che deve essere gestita a livello di database, attraverso la creazione di una tabella che contenga due **foreignkey**:

La **foreignkey** che mappa la tabella studente

La **foreignkey** che mappa la tabella corso

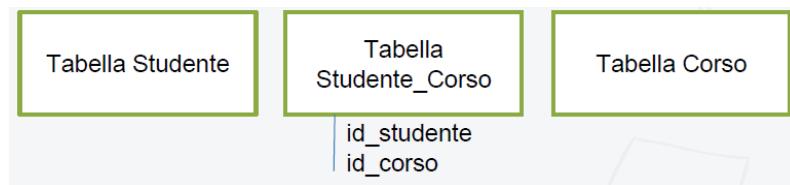


Per realizzare la relazione molti a molti in JPA, le annotation da usare nell'Entity che fa da owner sono:

**@ManyToMany**

**@JoinTable**, al cui interno dovremo valorizzare i seguenti parametri:

**name**(nome tabella...), **joinColumns**(nome della primary key della tabella owner), **inverseJoinColumns**(nome della primary key della seconda tabella)





## JPA - Java Persistence API

Tornando all'esempio (uno studente può iscriversi a più corsi online), per rappresentare le strutture, dobbiamo creare le Entity **Studente** e **Corso**.

```
@Entity
@Table(name="studente")
public class Studente {
    private Long id;
    private String nome; ...
    private List<Corso> corsi;

    @ManyToMany
    @JoinTable(
        name="studenti_corsi", joinColumns=@JoinColumn(name="studente", referencedColumnName="id_studente"),
        inverseJoinColumns=@JoinColumn(name="corsi", referencedColumnName="id_corso"))
    public Set<Corso> getCorsi() {
        return corsi;
    }
}
```

# **Java EE**

EMANUELE UMBERTO  
[www.web-passion.it](http://www.web-passion.it)  
[info@web-passion.it](mailto:info@web-passion.it)

