Secondo Sprint

Da Applicazioni Console a Applicazioni Web

Finora, abbiamo sviluppato applicazioni Java console, un punto di partenza comune per molti sviluppatori. Queste applicazioni interagiscono con l'utente attraverso una semplice interfaccia a riga di comando (**CLI**). L'utente fornisce l'input come testo, l'applicazione elabora tale input e produce un output, sempre sotto forma di testo.

Questo approccio è fondamentale per apprendere i principi di base della programmazione in Java. Tuttavia, l'interfaccia utente e le capacità interattive delle applicazioni console sono piuttosto **limitate**. Quando si tratta di creare applicazioni più complesse e interattive, dobbiamo guardare oltre le applicazioni console e considerare altre forme di interfaccia utente.

Un'interfaccia utente comune per le applicazioni moderne è rappresentata dalle **applicazioni web**. Queste applicazioni, accessibili da qualsiasi dispositivo con una connessione Internet e un browser web, offrono maggiore flessibilità e potenza rispetto alle applicazioni console.

Tuttavia, per realizzare applicazioni web in Java, abbiamo bisogno di tecnologie aggiuntive rispetto a quelle utilizzate per le applicazioni console. In particolare, necessitiamo di un **web server** per servire le pagine web agli utenti e di **servlet** per gestire le **richieste** degli utenti e generare **risposte** dinamiche.

Introduzione ai Web Server

Un server web è un software che serve contenuti o servizi ai client attraverso il **protocollo HTTP**. Questi contenuti sono principalmente pagine web, ma possono anche essere **servizi RESTful**, file o qualsiasi tipo di risorsa accessibile via web.

Un web server accetta **richieste dai client** (tipicamente browser web), le elabora e invia una risposta appropriata. Questa risposta può essere una pagina web, un'immagine, un documento o un servizio web.

Tomcat

Tomcat è uno dei **server web** Java più popolari, ma è fondamentale notare che Tomcat è tecnicamente un container di servlet, il che significa che gestisce principalmente applicazioni Java basate su **servlet**. Può funzionare come server web **standalone**, ma è più conosciuto come server di applicazioni Java.

Cos'è Tomcat:

- **Container di Servlet**: Tomcat è principalmente un container di servlet. Gestisce e esegue applicazioni Java basate su **servlet** e **JSP** (JavaServer Pages). Mentre i servlet sono componenti Java che generano dinamicamente contenuto web, JSP permette di scrivere HTML incorporando frammenti di codice Java.

- **Server Web Leggero**: Oltre a gestire servlet e JSP, Tomcat può funzionare anche come server web tradizionale, servendo contenuti statici come HTML, CSS e immagini.

Più avanti vedremo come installare e configurare Tomcat sul vostro computer.

Servlet

Le servlet rappresentano uno degli elementi fondamentali nel panorama dello sviluppo di applicazioni web basate su Java. Sono, in sostanza, classi Java che estendono le **capacità dei server** che ospitano applicazioni accessibili tramite un modello **richiesta-risposta.**

Che cosa sono le Servlet?

Una servlet è una classe Java che gestisce le richieste HTTP, elabora dati, produce risposte e può inviare queste risposte al client (solitamente un browser web). Le servlet sono alla base di molte tecnologie Java Web, tra cui JSP (JavaServer Pages).

Ciclo di vita:

Le servlet hanno un ciclo di vita ben definito:

Inizializzazione (**init**): Viene chiamato una sola volta quando la servlet viene caricata. È utilizzato per definire configurazioni e risorse.

Elaborazione delle richieste (**service**): Per ogni richiesta HTTP, il server invoca il metodo service(). Questo metodo, a sua volta, chiama metodi come **doGet**() o **doPost**() in base al tipo di richiesta HTTP.

Distruzione (**destroy**): Quando una servlet viene rimossa dal servizio, viene chiamato il metodo destroy(). È l'ultimo metodo che viene eseguito e serve a rilasciare risorse o eseguire altre pulizie.

Thread-Safety:

Nel mondo delle servlet, ogni volta che un client (ad es., un browser) invia una richiesta a una servlet, il server web avvia un nuovo **thread** per gestire quella richiesta. Poiché le servlet sono di solito singole istanze che durano per tutta la durata dell'applicazione, ciò significa che una singola istanza di servlet potrebbe avere più thread che la stanno accedendo **contemporaneamente**.

Pensate ad una servlet come a un unico sportello bancario e ai thread come a più clienti che cercano di utilizzarlo allo stesso tempo. Se l'operatore dello sportello non è preparato a gestire più clienti contemporaneamente, potrebbero verificarsi errori o confusione.

Variabili di Istanza:

Le variabili di istanza sono quelle dichiarate al di fuori di qualsiasi metodo, costruttore o blocco, e sono accessibili da qualsiasi metodo della classe. Ecco perché sono particolarmente problematiche in un ambiente **multi-thread** come le servlet.

Se una servlet usa variabili di istanza per memorizzare **dati specifici di una richiesta** (ad es., dettagli dell'utente corrente, risultati di una query), si verificano problemi. Supponiamo che due utenti, Alice e Bob, inviino una richiesta alla stessa servlet nello stesso momento. Se la servlet memorizza il nome dell'utente corrente in una variabile di istanza, potrebbe succedere che la richiesta di Alice sovrascriva la richiesta di Bob, portando la servlet a pensare che entrambe le richieste provengano da Alice.

Dunque è sbagliato dichiarare un service come variabile d'istanza?

No,non è intrinsecamente errato, ma c'è un dettaglio cruciale da considerare.

Quando parliamo di thread-safety in servlet, ci riferiamo principalmente ai dati che possono cambiare e che sono **specifici** per una particolare **richiesta** o **sessione** utente. Se il **Service non mantiene** uno stato che può variare tra diverse richieste (ad es. non ha variabili di istanza che immagazzinano dati specifici della richiesta), allora è **sicuro** dichiararlo come variabile di istanza.

Perché usare le Servlet?

Portabilità: Le servlet sono indipendenti dalla piattaforma, il che significa che possono funzionare su **qualsiasi server** che supporti la specifica Servlet **Java**.

Gestione efficiente delle risorse: A differenza delle tradizionali CGI (Common Gateway Interface), dove un nuovo processo è creato per ogni richiesta, le servlet utilizzano un modello basato su thread, il che le rende molto più **efficienti** in termini di memoria e velocità.

Integrazione con Java: Le servlet sono naturalmente integrate con il linguaggio Java, consentendo l'accesso a un vasto ecosistema di librerie e framework.

Sicurezza: Java e i server basati su servlet offrono funzionalità di sicurezza robuste, tra cui autenticazione e autorizzazione.

JSP

Le JavaServer Pages (JSP) sono una tecnologia per sviluppare pagine web **dinamiche**. A differenza di una pagina HTML statica, una JSP può cambiare e adattarsi in base all'input dell'utente, alle decisioni del server, o a qualsiasi altra sorgente di dati ed inoltre consente di integrare all'interno della pagina HTML codice Java.

```
<html>
<head><title>Esempio JSP</title></head>
<body>
    Salve, <%= request.getParameter("nome") %>! Come stai oggi?
</body>
</html>
```

Quando si parla di JSP (JavaServer Pages), si potrebbe pensare che stiamo discutendo di un tipo completamente diverso di tecnologia rispetto alle Servlet, ma in realtà le JSP e le Servlet sono **strettamente collegate**. Le JSP, infatti, vengono **convertite** in Servlet quando vengono chiamate per la prima volta o quando vengono modificate.

Ecco come funziona:

Traduzione e Compilazione: Quando una richiesta per una JSP arriva al server per la prima volta, il server traduce la JSP in **codice sorgente** Java per una Servlet. Dopodiché, questo codice sorgente della Servlet viene **compilato** in una classe Java bytecode.

Esecuzione: Una volta che la JSP è stata **tradotta** in una Servlet e compilata, il server la esegue proprio come farebbe con qualsiasi altra Servlet. Ogni richiesta successiva alla stessa JSP farà semplicemente **riferimento** alla Servlet già tradotta e compilata, a meno che la JSP originale non venga modificata.

Vantaggio: Questo processo di traduzione e compilazione delle JSP in Servlet avviene dietro le quinte e in modo **trasparente** per gli sviluppatori, permettendo di scrivere **pagine web dinamiche** utilizzando una sintassi simile all'HTML, pur mantenendo la potenza e la flessibilità del Java.

In sostanza, mentre le Servlet sono puramente classi Java orientate all'elaborazione e alla gestione delle richieste HTTP, le JSP sono progettate come pagine web con codice Java incorporato, che vengono poi convertite in Servlet per **l'esecuzione**. Questo dà il meglio di entrambi i mondi: la **facilità d'uso** e la **leggibilità** delle pagine web e la **potenza** e la **flessibilità** delle Servlet Java.

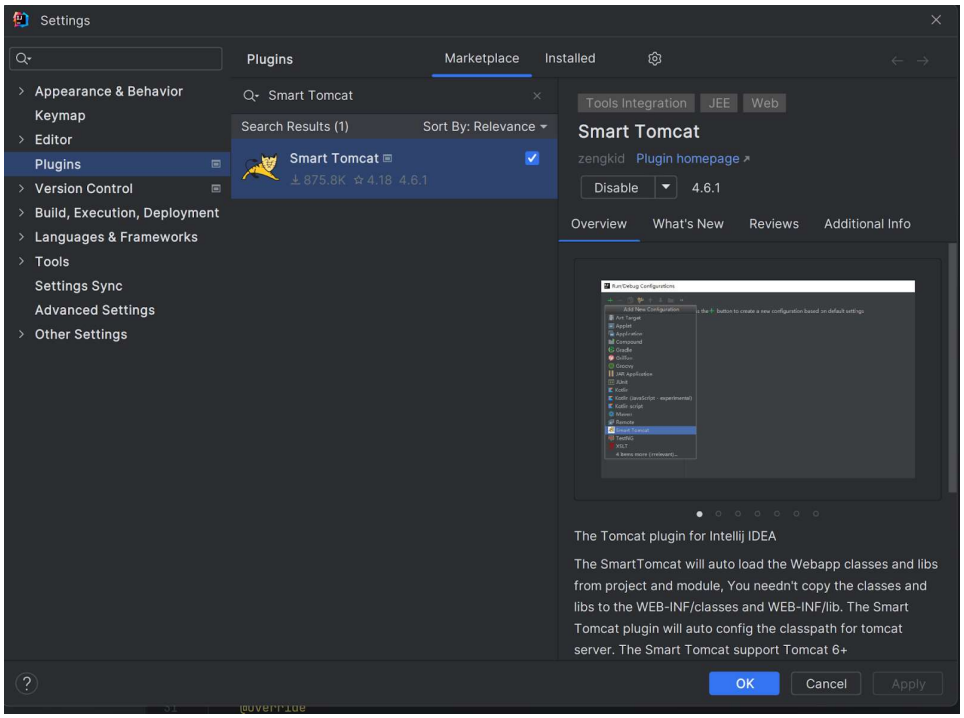
Setup Tomcat

Vediamo come andare a configurare Tomcat in modo da poter “runnare” la nostra applicazione.

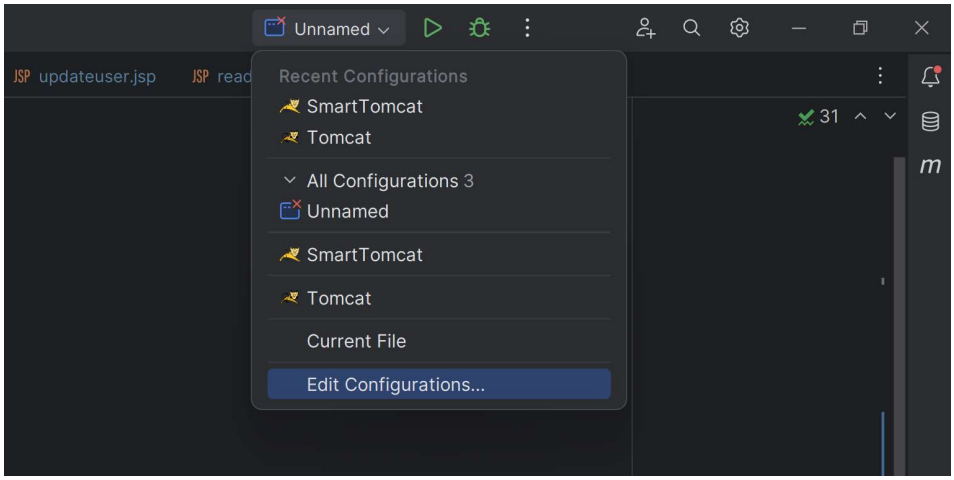
Come prima cosa andiamo a scaricare Tomcat dal seguente link https://tomcat.apache.org/download-90.cgi. In particolare scaricate questo link 32-bit/64-bit Windows Service Installer (pgp, sha512).

Una volta scaricato l’installer, completata l’installazione fate runnare il server Tomcat e aprite Intellij.

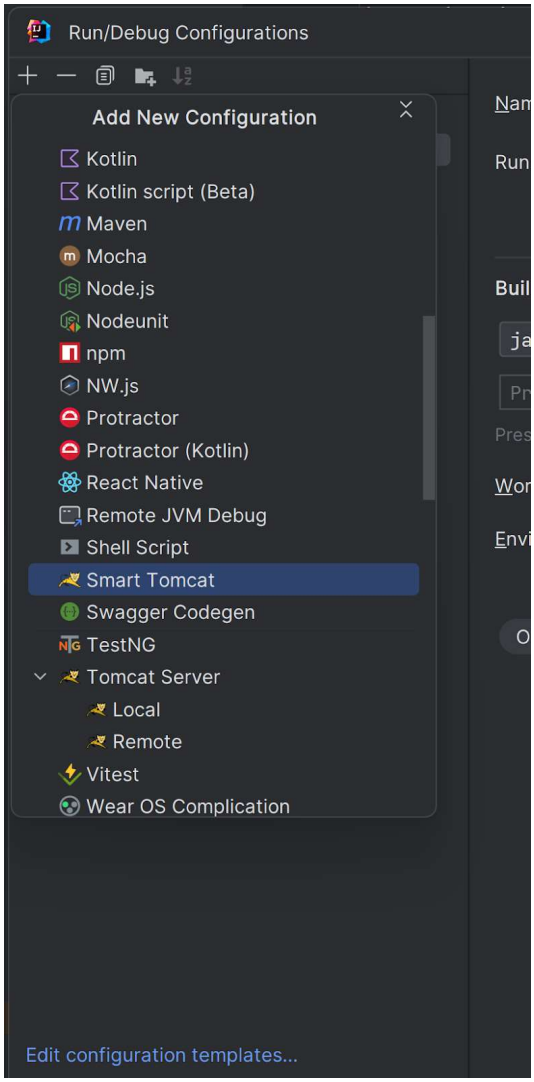
Arrivati su IntelliJ procediamo a scaricare il plugin “SmartTomcat” dalla sezione plugins.



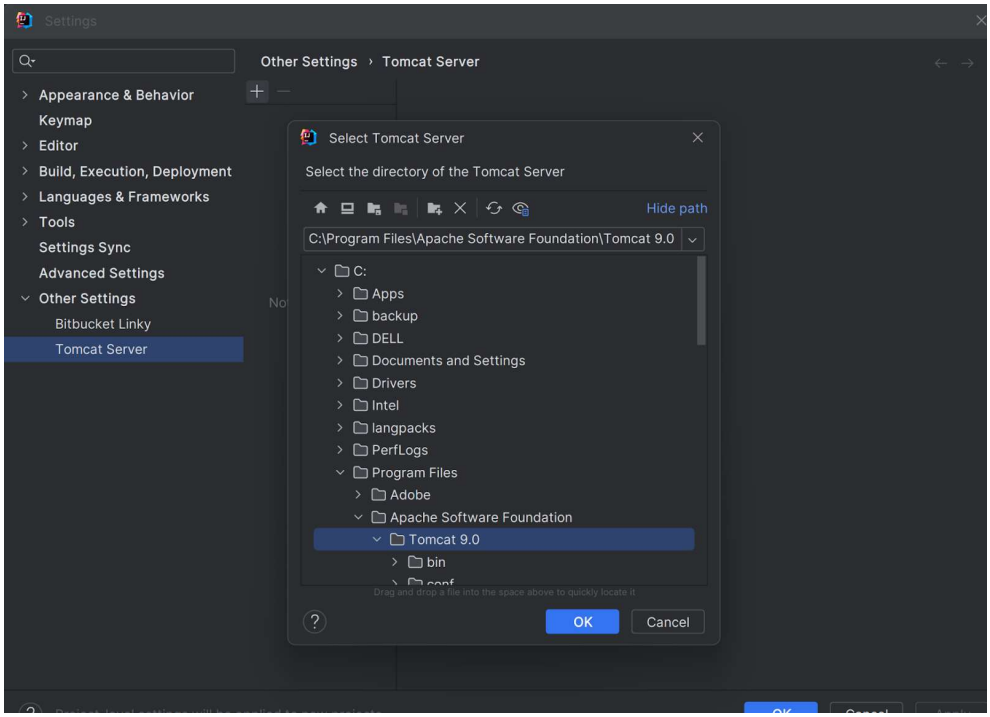
Scaricato ed installato il plugin procediamo a configurare Tomcat per runnare la nostra applicazione. Nella barra in alto andiamo a premere su Edit Configurations



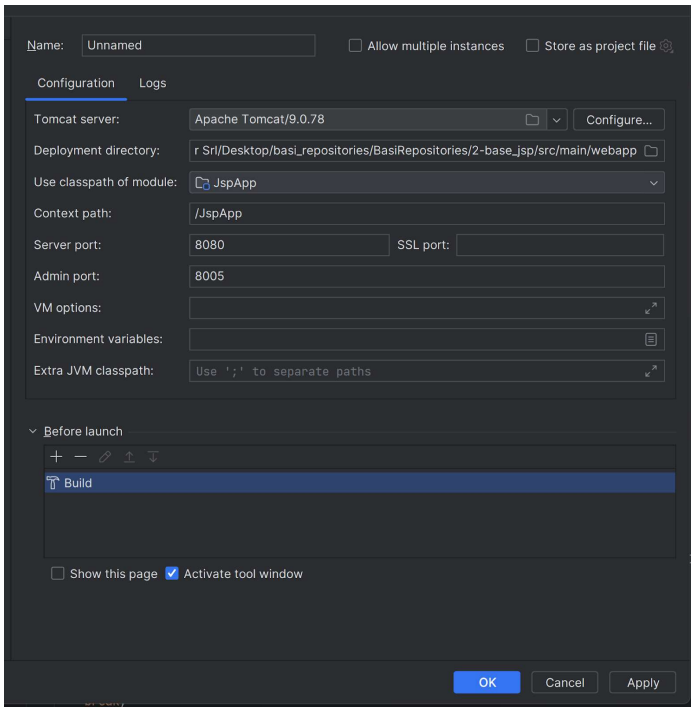
Vi si aprirà una schermata e dovrete premere sull'icona "+" e successivamente cercare nell'elenco delle configurazioni "Smart Tomcat"



A questo punto vi troverete nella schermata per configurare Tomcat. Come prima cosa premete su “Configure”, premete sul “+” e andate a selezionare la cartella di Tomcat, che si troverà nella cartella programmi del vostro PC

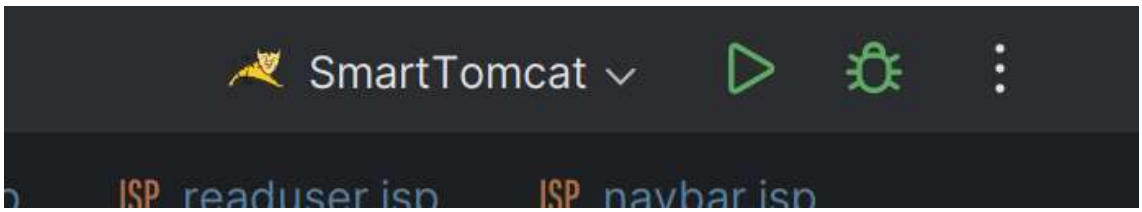


Vi dovreste ritrovare nella seguente situazione:

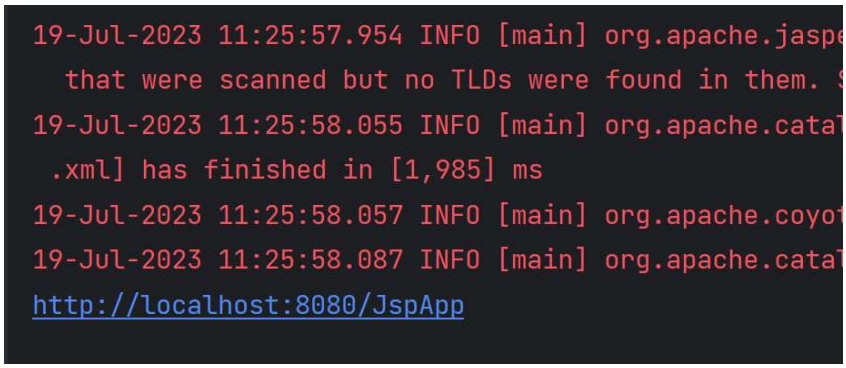


A questo punto avete terminato di configurare Tomcat e potete applicare le modifiche.

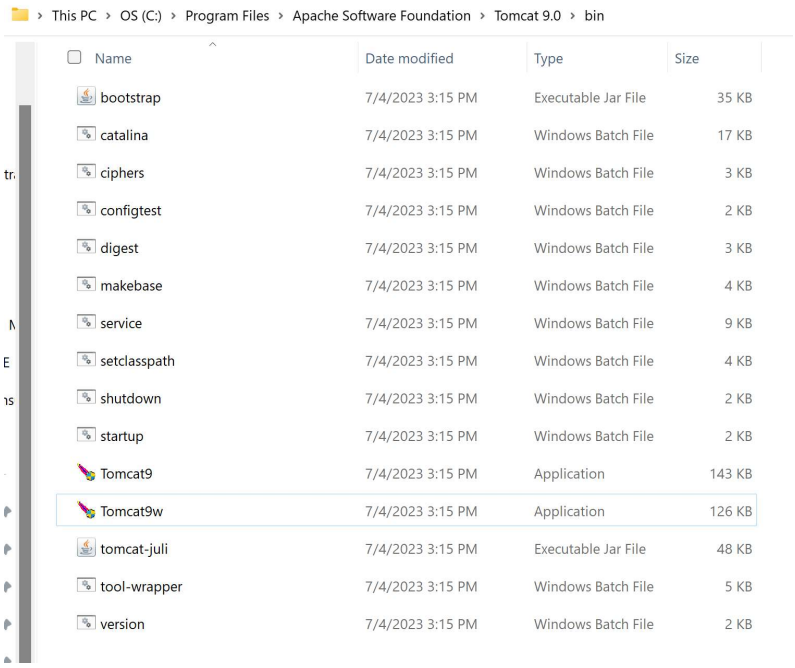
Ora premendo su “Play” Tomcat builderà la vostra applicazione per poi avviarla in localhost sulla porta 8080



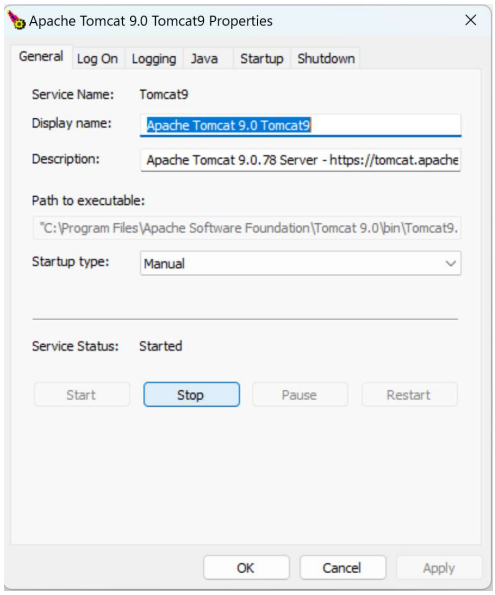
Sulla console vi comparirà il link della vostra applicazione, premendo su di esso vi si aprirà il browser e potrete iniziare ad utilizzare la vostra applicazione. (Non vi preoccupate di tutto quel rosso sulla console, è normale 😊)



Attenzione: Nel caso in cui doveste avere problemi ad avviare l'applicazione provate ad andare nella cartella "bin" di Tomcat e ad aprire "Tomcat9w"



A questo punto premete su "Stop" e aspettate che il server si stoppi e provate a riavviare l'applicazione su IntelliJ



Ora che abbiamo introdotto il concetto di applicazione Web, server, Servlet e Jsp e visto come configurare Tomcat andiamo ad approfondire il codice per comprendere meglio l'interazione tra questi componenti nel lato pratico.

Analisi del Codice

In questa sezione andremo ad analizzare il codice per comprendere come interagiscono Servlet e Jsp.

Come prima cosa andiamo ad analizzare il file **web.xml** all'interno della directory **webapp.**

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <session-config>
    <session-timeout>4000</session-timeout>
  </session-config>
</web-app>
```

Il file web.xml è il **Deployment Descriptor** di una web application basata su Java EE (Enterprise Edition). Esso fornisce informazioni al servlet container su come l'applicazione web deve essere **configurata** e come gestire le richieste.

Diamo un'occhiata più da vicino ai vari componenti di questo specifico file web.xml:

Dichiarazione XML:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Questa è una dichiarazione standard XML. Definisce la versione XML utilizzata (1.0) e l'encoding dei caratteri (in questo caso, UTF-8).

Elemento **web-app**:

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
```

Questo elemento rappresenta l'inizio del Deployment Descriptor. Contiene attributi che definiscono lo spazio dei nomi XML e lo schema (XSD) utilizzato. In particolare:

- **xmlns**: Definisce lo spazio dei nomi base per il file.
- **xmlns:xsi**: Definisce lo spazio dei nomi per l'XML Schema Instance.
- **xsi:schemaLocation**: Definisce la posizione dello schema XML utilizzato. Questo schema valida il file web.xml in base alle specifiche della versione 2.5 del servlet.
- **version**: Specifica la versione del servlet. In questo caso, è utilizzata la versione 2.5.

Elemento **welcome-file-list**:

```
<welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

Questo elemento definisce una lista di **file di benvenuto** per l'applicazione. Quando un utente visita l'URL base dell'applicazione (es. http://dominio/app/), il server cercherà in questa lista un file da mostrare come pagina iniziale. Nel nostro caso la prima pagina mostrata della nostra applicazione sarà il file **index.jsp**.

Elemento **session-config**:

```
<session-config>
   <session-timeout>60</session-timeout>
</session-config>
```

Questo elemento consente di configurare la sessione dell'utente. Il sotto-tag **session-timeout** definisce il tempo (in minuti) dopo il quale una sessione inattiva scadrà.

Il file si chiude con l'elemento </web-app>, che segna la fine del Deployment Descriptor.

In generale, il web.xml può essere molto più complesso e contenere molti altri elementi, a seconda delle necessità dell'applicazione. La versione fornita rappresenta una versione semplificata che copre solo alcune configurazioni di base.

Terminato di analizzare il web.xml possiamo procedere con l'analisi delle JSP e delle Servlet. Come descritto sul web.xml la pagina iniziale della nostra applicazione sarà la pagina “index.jsp”.

Gli elementi principali che possiamo trovare all’interno della pagina sono:

- **<link href="css/vittoriostyle.css" rel="stylesheet">**: Questa linea collega un foglio di stile CSS esterno chiamato vittoriostyle.css che si trova nella directory css. Questo foglio di stile definisce lo stile e la formattazione della pagina web.
- **<title>Login SAMPLE</title>**: Questo tag definisce il titolo della pagina web, che appare nella barra del titolo del browser.
- **<form class="login" action="LoginServlet" method="post">:** Questa è la definizione del form di login. L'attributo **action** indica che, quando l'utente invia il modulo, i dati verranno inviati al servlet chiamato **LoginServlet**. L'attributo **method="post"** indica che i dati saranno inviati utilizzando il metodo HTTP **POST**.
- I tag **<label> e <input>** sono utilizzati per creare un campo di input: L'attributo **name** dell'input è molto importante poiché determina come il valore dell'input verrà memorizzato nella request che verrà inviata al server. In questo caso abbiamo un campo input per lo **username** e uno per la **password**. L'input per la password possiede anche un altro attributo, **type="password",** che assicura che qualsiasi cosa digitata dall'utente nel campo sia mascherata.
- **<button type="submit" value="Login" name="pulsante">Login</button>**: Questo è il pulsante che l'utente può premere per **inviare** il modulo. Una volta premuto, i dati del modulo verranno inviati al LoginServlet, come specificato nell'attributo **action** del form, per l'elaborazione.

In sintesi, questa è una semplice pagina JSP per il login. Quando l'utente inserisce il proprio username e password e premere il pulsante "Login", i dati vengono inviati al LoginServlet per l'autenticazione.

Anche in questo caso andremo ad analizzare le parti principali della classe:

- **@WebServlet("/LoginServlet")**: Questa è un'annotazione che definisce l'URL pattern della servlet. Quindi, ogni volta che un'azione di un form o una richiesta AJAX punta all'URL "/LoginServlet", questa servlet sarà invocata. Ad esempio nel nostro form di login abbiamo specificato **action="LoginServlet" method="post",** dunque andremo a richiamare la classe **LoginServlet** ed il metodo **doPost.**
- **init():** Questo metodo viene chiamato una sola volta quando la servlet viene **inizializzata**. Crea una nuova istanza di LoginService, il servizio che utilizzeremo per effettuare il login. L'inizializzazione della servlet avviene prima che venga gestita qualsiasi richiesta da parte degli utenti.
- **doPost(HttpServletRequest request, HttpServletResponse response)**: Questo metodo viene chiamato quando la servlet riceve una **richiesta HTTP di tipo POST**. All'interno di questo metodo andremo a recuperare i parametri "username" e "password" dalla richiesta. A questo punto viene richiamato il servizio di login per verificare le credenziali. Se l'utente esiste **(userDTO != null)**, si salva l'utente nella **sessione** e si viene reindirizzati a una pagina appropriata in base al tipo di utente (admin o user). Se l'utente non esiste, setta un messaggio di errore e reindirizza alla pagina di login.

- **redirectToPage(HttpServletRequest request, HttpServletResponse response, String page)**: Questo è un metodo privato che facilita il reindirizzamento a una determinata pagina. Usa il **RequestDispatcher** per inoltrare la richiesta e la risposta alla pagina specificata.

Dunque facciamo un riassunto dell'interazione tra index.jsp e LoginServlet:

1. **Prima azione dell'utente:**
   L'utente apre la pagina **JSP di login** nel suo browser web. Questa pagina contiene un form che chiede all'utente di inserire il proprio "username" e "password".

2. **Compilazione del form:**
   Dopo aver inserito le proprie credenziali, l'utente fa clic sul pulsante "Login" del form. Quando questo accade, il **browser invia una richiesta HTTP di tipo POST** al server, indirizzando specificamente l'endpoint associato al LoginServlet, che è "/LoginServlet", come specificato nell'attributo "action" del form.

3. **Servlet riceve la richiesta:**
   La richiesta POST arriva al LoginServlet. Nel metodo **doPost** della servlet, vengono estratti i parametri "username" e "password" dalla richiesta HTTP grazie al metodo **request.getParameter.**

4. **Verifica delle credenziali:**
   Il LoginServlet chiama il servizio **LoginService** per verificare le credenziali fornite. Questo servizio richiama il LoginDAO per verificare che sul database sia presente un utente che corrisponde alle credenziali inserite.

5. **Risposta in base alla verifica:**
   - Se le credenziali sono corrette e l'utente esiste (userDTO != null), l'utente viene memorizzato nella **sessione** e, in base al suo tipo (usertype), viene reindirizzato a una pagina appropriata (ad esempio, "homeadmin.jsp" per gli admin o "homeuser.jsp" per gli utenti normali).
   - Se le credenziali sono errate o l'utente non esiste, viene impostato un messaggio di errore nella richiesta e l'utente viene reindirizzato alla pagina di login con un messaggio di errore appropriato.
6. **Fine dell'interazione:**
   Il browser dell'utente riceve la risposta dal server e visualizza la pagina appropriata all'utente.

In sintesi, la JSP di login funge da punto di inizio per l'utente per fornire le proprie credenziali, mentre il LoginServlet si occupa di elaborare queste credenziali, di

verificare la loro validità e di decidere dove reindirizzare l'utente in base ai risultati della verifica.

Utilizzo di codice Java all'interno di JSP:

Le JSP (JavaServer Pages) permettono di **incorporare** il codice Java direttamente nel markup HTML attraverso speciali tag e costrutti. Questi costrutti permettono agli sviluppatori di **generare dinamicamente** il contenuto HTML in base alle esigenze dell'applicazione.

Ecco alcuni dei costrutti comuni utilizzati in JSP per incorporare il codice Java:

Scriptlet: Si tratta di blocchi di codice Java inseriti tra **<% e %>**. Il codice all'interno di questi delimitatori viene eseguito ogni volta che la pagina JSP viene richiesta.

```
<%
    // Codice Java qui
%>
```

Espressioni: Queste sono utilizzate per **stampare valori** direttamente nel flusso di output della risposta. Le espressioni sono delimitate da **<%= e %>**.

```
<%= variabileJava %>
```

Dichiarazioni: Queste sono utilizzate per **dichiarare variabili** o metodi che saranno utilizzati in seguito nella JSP. Sono delimitate da **<%! e %>**.

```
<%!
    // Dichiarazione di variabili o metodi
%>
```

Direttive: Queste forniscono istruzioni alla JSP su come comportarsi durante la sua elaborazione. Alcune direttive comuni sono include, page e taglib.

```
<%@ page import="java.util.Date" %>
```

Andiamo ad analizzare un blocco di codice presente sulla pagine "**usermanager.jsp**":

```
<%
    for (UserDTO u : list) {
%>
<tr>
    <td><a href=UserServlet?mode=read&id=<%=u.getId()%>>
            <%=u.getUsername()%>
    </a></td>
    <td><%=u.getPassword()%></td>
    <td><%=u.getUsertype()%></td>
    <td><a href=UserServlet?mode=update&id=<%=u.getId()%>>Edit</a>
    </td>
    <td><a href=UserServlet?mode=delete&id=<%=u.getId()%>>Delete</a>
    </td>

</tr>
<%
    }
%>
```

Scriptlet iniziale:

<%
   for (UserDTO u : list) {
%>

Questo è uno scriptlet che inizia un ciclo for per iterare su una lista di oggetti UserDTO, che abbiamo ottenuto precedentemente tramite lo scriplet

<%
        List<UserDTO> list = (List<UserDTO>) request.getAttribute("users");
%>

Utilizzo delle espressioni per l'output:

<td>
        <a href=UserServlet?mode=read&id=<%=u.getId()%>>
                <%=u.getUsername()%>

</td>

In questo segmento:

u.getId() viene utilizzato per ottenere l'ID dell'utente corrente nel ciclo e viene inserito come parametro nella richiesta a UserServlet.
u.getUsername() viene utilizzato per ottenere il nome utente dell'utente corrente nel ciclo e visualizzarlo come testo del link.
Entrambe queste operazioni utilizzano espressioni per inserire direttamente il valore nel flusso di output HTML.
Altri valori mostrati utilizzando espressioni:

<td><%=u.getPassword()%></td>
<td><%=u.getUsertype()%></td>

Qui, mostriamo sia la password che il tipo di utente dell'utente corrente nel ciclo.

Link per l'aggiornamento e la cancellazione:

<td><a href=UserServlet?mode=update&id=<%=u.getId()%>>Edit</td>
<td><a href=UserServlet?mode=delete&id=<%=u.getId()%>>Delete</td>
In questi segmenti, vengono forniti link per modificare (Edit) o eliminare (Delete) l'utente corrente. L'ID dell'utente viene passato come parametro nella query URL.

Chiusura dello scriptlet:
<%
  }
%>
Qui, il ciclo for viene chiuso.

Differenze tra Console e Web:

- **Interfaccia Utente:**
  - Applicazioni Console: Utilizzano un'interfaccia basata sul testo. L'input viene ricevuto tramite righe di comando e l'output viene stampato direttamente nella console. Non c'è nessuna formattazione visuale.
  - Applicazioni Web: Offrono un'interfaccia grafica interattiva basata su browser. Nel tuo codice, l'interfaccia dell'utente viene definita attraverso elementi HTML come form, input e button.
- **Modalità di Interazione:**
  - Applicazioni Console: L'utente interagisce con il programma inserendo comandi o dati direttamente nella console. La risposta o l'output viene mostrato subito dopo.
  - Applicazioni Web: L'interazione avviene attraverso **richieste HTTP**. L'utente compila un form (come il form di login) e quando fa clic su "Submit", viene inviata una richiesta POST al server. Il Servlet poi elabora questa richiesta e invia una risposta, che può essere una nuova pagina web o un messaggio.
    N.B. Abbiamo analizzato solo il caso di una chiamata POST tramite form, ma nel codice troverete anche altre chiamate come ad esempio **href="UserServlet?mode=getall",** in questo caso stiamo inviando una richiesta di tipo **GET** a UserServlet passando come parametro **mode=getall**. In questo caso verrà richiamato il metodo **doGet** di UserServlet.
- **Architettura:**
  - Applicazioni Console: Sono generalmente monolitiche e eseguono tutto il loro codice su un singolo sistema o dispositivo.

    - Applicazioni Web: Funzionano su un'architettura **client-server.** Il client (browser) invia richieste al server, il server (dove risiede il Servlet) elabora la richiesta e invia indietro una risposta.
- **Stato e Sessioni:**
    - Applicazioni Console: Non hanno un concetto integrato di sessioni.
    - Applicazioni Web: Possono gestire sessioni e stato dell'utente. Nel nostro caso LoginServlet, una volta che un utente si autentica con successo, salva i suoi dettagli in una session (**request.getSession().setAttribute("user", userDTO)**). Questo consente di tracciare e mantenere le informazioni dell'utente attraverso diverse richieste e pagine.
- **Ambiente di Esecuzione:**
    - Applicazioni Console: Funzionano direttamente su un sistema operativo, senza l'esigenza di software aggiuntivi (come un browser).
    - Applicazioni Web: Richiedono un server web o un container servlet (come Tomcat) per eseguire codice server-side come i Servlet. Inoltre, necessitano di un browser per visualizzare e interagire con le pagine.

In sintesi, mentre le applicazioni console sono generalmente più semplici e dirette, le applicazioni web offrono una maggiore flessibilità e interattività, specialmente per un numero maggiore di utenti.