3. Programación avanzada con JSP.

1. Introducción.

Como hemos visto, la tecnología JSP proporciona elementos de alto nivel que nos permiten desarrollar de forma sencilla aplicaciones web.

En éste capítulo veremos en detalle el uso de los elementos JSP, así como algunas técnicas de desarrollo de alto nivel que nos permitirán establecer una separación clara entre la presentación y el modelo de datos, ofreciendo escalabilidad y reutilización de código a las aplicaciones y separación de funciones en el equipo de desarrollo.

2. Directivas JSP.

Las directivas JSP son elementos declarativos interpretados por el servidor Java EE y que afectan a la generación de la página. JSP define tres tipos de directivas: *page*, *include* y *taglib*.

La directiva de página tiene la sintaxis **`<%@page atributo="valor" ... %>`** donde atributo puede ser:

- language: indica el lenguaje empleado en los elementos de secuencia de código. Su valor por defecto es “java”.

- session: se usa para establecer si la página usa una sesión. En tal caso, la página toma en su variable session la sesión actual, o crea una si no había ninguna sesión iniciada. El valor por defecto es “true”.

- contentType: especifica el tipo MIME del contenido devuelto. Su valor por defecto es “text/html”

- pageEncoding: indica la codificación de caracteres usada en la página.

- import: permite importar los paquetes java necesarios para el código de la página. Su valor será una lista de clausulas import separadas por comas. Se pueden especificar comodines, al igual que en las sentencias import de Java.

- buffer: especifica el tamaño del buffer de salida. Los valores válidos son de la forma “nnkb”, donde nn es el número de Kb empleados en el buffer. Su valor por defecto es “8kb”.

- autoflush: establece si el buffer debe vaciarse automáticamente cuando esté lleno. Su valor por defecto es “true”. Si se especifica “false” y el buffer se llena se lanzará una excepción por desbordamiento de buffer.

- isThreadSafe: indica si la página puede procesar peticiones simultáneas en múltiples subprocesos. Si se especifica "false" la página se ejecutará en un entorno de hilo único – sin multithreading -. El valor por defecto es "true".

- info: establece la cadena de información del servlet generado con el valor especificado.

- errorPage: indica la página de error asociada a la página actual. Si se produce alguna excepción sin tratar en ésta página, el servidor Java EE le cederá el control a la página indicada en éste atributo, enviándole la excepción.

- isErrorPage: establece si la página actual es una página de error. Su valor predeterminado es "false". Si especificamos "true", el objeto implícito exception estará disponible y contendrá el objeto de excepción lanzado por otra página.

- extends: si especificamos éste atributo, su valor contendrá el nombre completamente cualificado de la clase Java a la que debe extender el servlet generado. Dicha clase debe implementar la interfaz HttpJspPage. Normalmente no se utiliza.

- isELIgnored: sirve para especificar si queremos que se ignoren los elementos EL – lenguaje de expresiones -. Su valor predeterminado es false.

La directiva de inclusión tiene la sintaxis **<%@include file="path" ... %>** y sirve para fusionar, en el momento de la traducción, el contenido del fichero especificado por path con el código fuente de la página JSP. La ruta puede ser absoluta o relativa, y se interpreta de acuerdo al contexto del servlet. Hay que tener en cuenta que ésta fusión es estática, a diferencia de lo que ocurre con la acción <jsp:include>.

La directiva taglib permite importar bibliotecas de etiquetas personalizadas. Su sintaxis es **<%@taglib uri="tagLibraryURI" prefix="prefijo"%>**
donde el atributo uri indica la localización de un descriptor de biblioteca de etiquetas o TLD – Tag Library Descriptor -, y prefix indica el prefijo que se usará en ésta página para las etiquetas importadas desde esa biblioteca. Más adelante trataremos las etiquetas personalizadas y su uso.

3. Elementos de secuencias de código. Declaraciones, scriptlets y expresiones JSP.

Los elementos de secuencias de código contienen código ejecutable Java. Se pueden distinguir los siguientes:

Declaraciones: se delimitan por **<%!** y **%>** y contienen código Java que en el momento de la traducción se copia directamente dentro de la clase que implementa el servlet generado. Se usan para declarar elementos de programa como métodos auxiliares, variables o constantes, que se usarán en otros elementos de la página.

Scriptlets: se delimitan por **<%** y **%>** y contienen código Java que se copia directamente a la clase que implementa el servlet generado, justo en el lugar en el que aparece en la página JSP. Hay que tener en cuenta que a diferencia de lo que ocurre con las declaraciones, el código de los scriptlets se copia dentro del método *jspService()* - que es el método encargado de procesar la petición y generar la respuesta – , al igual que ocurre con el código correspondiente a otros elementos JSP, como el texto plano, etiquetas HTML, acciones JSP, expresiones, etc.

Objetos implícitos: son objetos accesibles mediante variables declaradas e inicializadas de forma implícita en el método *jspService()* y que son accesibles directamente desde cualquier elemento de secuencias de código excepto las declaraciones. Son los siguientes;

- **application**; hace referencia al contexto o ámbito de aplicación. Se obtiene directamente del objeto *servletContext* en el servlet generado. Algunos de sus métodos más importantes son *getAttribute()*, *setAttribute()*, *removeAttribute()*, *getContextPath()*, *getRealPath()*, etc. Es de tipo *javax.servlet.ServletContext*.

- **config**: es el objeto *servletConfig* del servlet generado. Algunos de sus métodos son *getInitParameter()*, *getServletName()*. Es de tipo *javax.servlet.servletConfig*.

- **exception**: sólo se puede usar en las páginas de error, y contiene la excepción lanzada por la página que causó el error. Es de tipo *java.lang.Throwable*.

- **out**: es el objeto que escribe la respuesta que se envía al cliente. Algunos de sus métodos son *flush()*, *print()*, *println()*, *clearBuffer()*, etc. Es de tipo *javax.servlet.jsp.JspWriter*.

- **pageContext**: hace referencia al contexto o ámbito de página. Algunos de sus métodos son *getAttribute()*, *setAttribute()*, *removeAttribute()*, *getRequest()*, *getResponse()*, etc. Es de tipo *javax.servlet.jsp.PageContext*.

- **request**: es el objeto de petición y sirve también para acceder al ámbito de petición. Algunos de sus métodos más importantes son *getParameter()*, *getCookies()*, *getHeader()*, *getAttribute()*, *setAttribute()*, *removeAttribute()*, etc. Es de tipo *javax.servlet.HttpServletRequest*.

- **response**: es el objeto de respuesta. Algunos de sus métodos son *addCookie()*, *addHeader()*, *getWriter()*, etc. Es de tipo *javax.servlet.HttpServletResponse*.

- **session**: hace referencia al ámbito de sesión. Algunos de sus métodos son *getAttribute()*, *setAttribute()*, *removeAttribute()*, *getId()*, *isNew()*, *invalidate()*, *setMaxInactiveInterval()*, etc. Es de tipo *javax.servlet.http.HttpSession*.

Es importante destacar que en JSP se definen cuatro contextos o ámbitos de declaración que sirven principalmente para almacenar y recuperar atributos mediante los métodos *setAttribute()*, *getAttribute()* y *removeAttribute()*, Los atributos son de tipo *Object*, lo que nos permite almacenar cualquier objeto en un atributo, desde un número hasta una colección. Dichos atributos se identifican en cada contexto por un nombre que le asignamos al crearlo y tienen un ciclo de vida que depende del contexto en el que están definidos. Los cuatro contextos son:

- **página(*pageContext*)**: es el que tiene un ciclo de vida más corto. Está limitado al procesamiento de la página. Cada vez que se accede a una página JSP se crea su contexto de página.

- **petición(*request*)**: está disponible mientras se procesa la petición. La principal diferencia con el anterior es que una petición puede ser procesada por varias páginas, ya que se pueden relanzar peticiones – <jsp:forward> - o incluir otras páginas - <jsp:include> - durante el procesamiento de la petición.

- **sesión(*session*)**: su ciclo de vida se extiende durante toda la sesión para un usuario, es decir, desde que se establece la sesión – normalmente cuando el usuario se conecta a la aplicación – hasta que se cierra la sesión, ya sea porque el usuario se desconecta o porque se ha cumplido el tiempo máximo de inactividad, etc. Es uno de los más utilizados ya que nos permite establecer y manejar atributos para cada usuario independientemente de los demás y durante el tiempo que dure su sesión.

- **aplicación(*application*)**: es el contexto con un ciclo de vida mayor: comienza cuando se inicia la aplicación y se extiende mientras ésta se mantenga activa. Los atributos definidos en el contexto de aplicación son comunes a todos los servlets y a todas las páginas de la aplicación.

Expresiones: se delimitan por **<%=** y **%>** y contienen código Java con una expresión que se evalúa en el momento de procesar la página y cuyo resultado se copia a la salida de la página. Dicha expresión debe devolver una cadena o cualquier valor que se pueda convertir en una cadena de forma automática – tal y como lo hace el método *println()* -. En la expresión se pueden usar los objetos implícitos y todos los valores derivados de ellos, así como variables o métodos creados en otros elementos JSP – como scriptlets, declaraciones, etc -.

A partir de la versión JSP 2.0 se introdujo un nuevo tipo de expresiones delimitadas por **${** y **}**, y que utiliza un lenguaje de expresiones, **EL** – expression language – definido a tal efecto. Éste lenguaje está orientado a acceder de una forma sencilla y flexible a elementos tales como parámetros de petición, propiedades de beans, etc. Los elementos que se pueden usar en el lenguaje de expresiones son:

- Variables: cuando usamos una variable, el servidor Java EE busca un atributo con ese nombre en los contextos de página, petición, sesión y aplicación sucesivamente hasta que la encuentre y devuelve su contenido. Si no encuentra el atributo en ningún contexto devolverá null. Si el nombre coincide con el de un objeto implícito, en lugar de buscar un atributo devolverá dicho objeto implícito. Para acceder a las propiedades de una variable se usa el operador **.** o **[]** indistintamente. El operador **[]** se puede usar también para acceder a los elementos de un array, colección o mapa.

- Objetos implícitos: el lenguaje de expresiones define los siguientes objetos implícitos:
  - **pageContext**: el contexto de página. Además proporciona acceso a varios objetos: servletContext, session, request y response.
  - **param**: un mapa para acceder a los parámetros de la petición. Devuelve un valor simple.
  - **paramValues**: un mapa para acceder a los parámetros de la petición. Devuelve un array de valores.
  - **header**: un mapa para acceder a las cabeceras de la petición. Devuelve un valor simple.
  - **headerValues**: un mapa para acceder a las cabeceras de la petición. Devuelve un array de valores.
  - **cookie**: un mapa para acceder a las cookies.
  - **initParam**: un mapa para acceder a los parámetros de inicialización.
  - **pageScope**: un mapa para acceder a las variables del contexto de página.
  - **requestScope**: un mapa para acceder a las variables del contexto de petición.
  - **sessionScope**: un mapa para acceder a las variables del contexto de sesión.
  - **applicationScope**: un mapa para acceder a las variables del contexto de aplicación.

- Literales: pueden ser booleanos - *true* y *false* -, enteros, de punto flotante y *null*. Se definen igual que en Java.

- Operadores: además de los operadores del lenguaje Java, se establecen algunos al estilo de los lenguajes de scripting, tales como lt (menor que), gt (mayor que), le (menor o igual), ge (mayor o igual), eq (igual), ne (distinto de), and, or, not, mod, empty. A continuación se muestra una relación de todos los operadores en orden de mayor a menor precedencia:
  - `[] .`
  - `()`
  - `-` (unario) `not ! empty`
  - `* / div % mod`
  - `+ -` (binario)
  - `< > <= >= lt gt le ge`
  - `== != eq ne`
  - `&& and`
  - `|| or`
  - `? :`

- Funciones: el lenguaje de expresiones permite usar funciones definidas en bibliotecas de etiquetas – taglibs -. La sintaxis para usar una función es f:funcion(parametros), donde f es el espacio de nombres asignado a la biblioteca que contiene la función. Más adelante veremos las bibliotecas de etiquetas y sus espacios de nombres.

- Llamadas a métodos con parámetros: a partir de la versión 2.2 del lenguaje de expresiones se pueden incluir llamadas a métodos arbitrarios de un bean, pasándoles una lista de parámetros tal y como se hace en lenguaje Java: metodo(parametro1, parametro2,...). Aunque no es lo más indicado desde el punto de vista de la separación de capas de la aplicación, en ocasiones resulta muy útil y puede ahorrar bastante código sin dañar significativamente la homogeneidad de la página.

4. Uso de JavaBeans para interactuar con la capa de negocio.

La mayoría de los elementos JSP de alto nivel, tales como el Lenguaje de Expresiones , algunas de las acciones JSP y muchas etiquetas de las bibliotecas de taglibs estándar están diseñados para trabajar con objetos JavaBeans, estableciendo así una comunicación normalizada y bien definida entre la capa Web y la capa de negocio. La comunicación con los JavaBeans se realiza en dos sentidos: estableciendo propiedades de los beans y consultando propiedades.

Los JavaBeans son componentes Java reutilizables que pueden ser manipulados por herramientas automáticas gracias a la normalización de sus métodos de acceso. Sirven esencialmente para almacenar y manipular datos.

Las normas para diseñar un JavaBean son las siguientes:

- La clase de be ser serializable, es decir, debe implementar la interfaz java.io.Serializable, con el fin de que las herramientas que manipulan beans sean capaces de almacenar y recuperar el estado de dichos objetos entre sesiones distintas. Ésta norma se relaja a menudo y muchas herramientas permiten que nos la saltemos.

- La clase debe tener un constructor por defecto, es decir, un constructor sin parámetros. De ésta forma las herramientas serán capaces de instanciar JavaBeans llamando a su constructor por defecto. En caso de no tenerlo, dichas herramientas no sabrían que valores enviar a los parámetros del constructor. Es importante destacar que además del constructor por defecto, los JavaBeans pueden tener otros constructores.

- Los métodos de acceso a las propiedades deben seguir el estándar getter/setter. Dicho de otro modo, para cada propiedad del bean, pongamos por caso la propiedad xxx el método de acceso para establecer la propiedad debe ser *public void setXxx(tipo parametro)*, y el método de consulta debe ser *public tipo getXxx()*, donde tipo es el tipo de la propiedad. Si la propiedad es booleana, el método getter se convierte en *public void isXxx()*. Las propiedades pueden ser de sólo lectura, cuando sólo existe el método getter, de sólo escritura, cuando sólo existe el setter o de lectura/escritura cuando existen los dos. Normalmente, los valores de las propiedades se almacenan en atributos privados, pero no tiene por qué ser así. Los métodos getter y setter, ademas de consultar o establecer valores pueden realizar otras cosas como validaciones, conversiones, etc.

5. Acciones estándar.

Son elementos JSP de alto nivel que en general, sirven para manipular otros objetos. Su sintaxis sigue la norma XML: <*accion* [*atributo*=”*valor*” ...]>...</*accion*> o <*accion* [*atributo*=”*valor*” ...]/> si la acción no tiene cuerpo. Las acciones se pueden dividir en tres grupos: manejo de beans, lanzamiento de peticiones y manejo de plugins.

Las acciones asociadas al manejo de beans son:

- **<jsp:useBean>**: declara un ejemplar de Java Bean y lo asocia a una variable. Su sintaxis es **`<jsp:useBean id=”nombre” [type=”tipo”] [class=”clase”] [beanName=”nombreDeBean”] [scope=”page|request|session|application”]>...</jsp:useBean>`**. Aunque los atributos class y type son opcionales, al menos uno de ellos debe aparecer, aunque también pueden estar ambos. Por otra parte, los atributos class y beanName son incompatibles, es decir, sólo puede aparecer uno de ellos. El atributo scope también es opcional y su valor predeterminado es page.

Ésta acción se usa para declarar un bean y así poder usarlo posteriormente en la página mediante otras acciones JSP.

Cuando declaramos un bean con ésta acción, el servidor Java EE declarará una variable con el nombre dado por el atributo id del tipo indicado por el atributo type o por class y seguidamente buscará un bean en el ámbito especificado por scope con el mismo nombre. Si lo encuentra, lo asignará a la variable declarada, y en caso contrario, instanciará un objeto de la clase especificada por el atributo class y se asignará.

Si la clase no se ha especificado y no se ha encontrado un bean en el ámbito correspondiente, se generará una excepción. En caso de que se especifique el atributo beanName en lugar de class, entonces si no se encuentra el bean se intentará deserializar desde un fichero cuyo nombre se obtendrá del atributo beanName – donde aparecerá el nombre de la clase completamente cualificada -, sustituyendo los puntos por barras de directorio, y se añadirá al final la extensión .ser.
Si la acción tiene un cuerpo – es decir, una o más líneas JSP entre la etiqueta de apertura y la de cierre - , éste será evaluado sólo en el caso de que se haya creado un nuevo bean. Dicho de otro modo, el cuerpo de la etiqueta servirá para la inicialización del bean.

- **<jsp:setProperty>**: Esteblece los valores de una o más propiedades de un bean declarado previamente. Su sintaxis es la siguiente: **`<jsp:setProperty name="id" asignar-propiedades/>`**, donde asignar-propiedades puede ser **`property=*|property="propiedad"|property="propiedad" param="parametro"|property="propiedad" value="valor"`**.
El atributo name se refiere al nombre asignado al bean mediante el atributo id de la acción <jsp:useBean>.

Las propiedades se pueden establecer desde los parámetros de la petición, ya sea especificando cada parámetro con param o buscando un parámetro que se llame igual que la propiedad, si sólo se especifica éste.

Además si se usa * para la propiedad, se intentarán establecer todas las propiedades del bean cuyos nombres coincidan con sendos parámetros de la petición. Por último, si se incluye el atributo value, su valor podrá ser un valor inmediato o una expresión que será evaluada.

En el caso de que asignemos valores de parámetros, si alguno de ellos es nulo o una cadena vacía, no se realizará la asignación. Ésta acción no puede tener cuerpo.

- **<jsp:getProperty>**: Devuelve el valor de la propiedad especificada. Su sintaxis es **`<jsp:getProperty name ="id" property="propiedad"/>`**, donde name indica el identificador asignado al bean y property el nombre de la propiedad a consultar.

Las acciones asociadas al lanzamiento de peticiones sirven para redirigir el control a otra página o servlet, ya sea definitivamente o para fusionar su respuesta con la de la página que realiza la llamada. En ninguno de éstos casos interviene el cliente. El traspaso de control se realiza íntegramente en el servidor, y se realiza pasándole al recurso receptor tanto la petición como la rspuesta. Las tres acciones disponibles a tal efecto son:

- **<jsp:forward>**: Reenvía la petición al recurso especificado. Su sintaxis es **<jsp:forward page="URL"/>**, donde page hace referencia al recurso de destino.
Su cuerpo puede contener acciones <jsp:param> si hay que enviar parámetros. Se puede usar una expresión para el atributo page, lo que permite decidir de forma dinámica el recurso al que se envía la petición.

Cuando se pasa el control al recurso de destino, la página de origen termina su ejecución, y si había algo escrito en el buffer de salida, se borra, ya que debe ser la página de destino la que genere la respuesta. En el caso de que ya se hubiesen enviado datos al cliente, se generaría una IllegalStateException.

- **<jsp:include>**: Invoca a otro recurso y fusiona su salida con la salida de la página que llama.

Cuando el recurso de destino es procesado, el control vuelve a la página de origen. Su sintaxis es **<jsp:include page="URL" [flush="true|false"]/>**.

Al igual que en la acción anterior, el atributo page indica el recurso a ser incluído. En éste caso se puede especificar el atributo flush, que indica si se debe vaciar el buffer, enviando su contenido al cliente, antes de incluir el contenido del otro recurso. Su valor por defecto es false. En éste caso también se pueden incluir acciones <jsp:param> en el cuerpo de la acción.

- **<jsp:param>**. Permite enviar parámetros a otros recursos. Su sintaxis es **<jsp:param name="nombre" value="valor"/>**.

Sólo existe una acción para el manejo de plugins:

- **<jsp:plugin>**. Sirve para generar el vínculo HTML apropiado para cargar un plugin, generalmente un applet. Su sintaxis es **<jsp:plugin type="applet|bean" code="objeto" codeBase="baseObjeto" {align="alineación"} {archive="listaDeArchivos"} {height="altura"} {hspace="espaciadoHorizontal"} {jreversion=""vorsion_jre} {name="nombreDeComponente"} {vspace="espaciadoVertical"} {width="anchura"} {nspluginurl="url"} {iepluginurl="url"} {<jsp:params> {<jsp:param name="nombre" value="valor"/>}...</jsp:params>}} </jsp:plugin>**

6. Bibliotecas de etiquetas personalizadas (taglibs).

JSP proporciona un mecanismo para ampliar las acciones estándar mediante las bibliotecas de etiquetas personalizadas o taglibs. Dichas bibliotecas contienen un conjunto de acciones que siguen la sintaxis XML: **<prefijo:etiqueta [atributo="valor"...]>** … **</prefijo:etiqueta>**. El prefijo indica el espacio de nombres que se utilizará para todas las etiquetas de una misma biblioteca y la etiqueta indicará la acción a realizar.

Cada etiqueta definirá un conjunto de atributos que podrán ser obligatorios u opcionales, y cuyos valores se especificarán entre comillas simples o dobles indistintamente. La etiqueta podrá contener un cuerpo, es decir un contenido entre la etiqueta de apertura y la de cierre. Éste contenido puede ser JSP o contenido específico de la etiqueta, como por ejemplo, una consulta SQL para etiquetas que realicen funciones de acceso a bases de datos. Si la etiqueta no tiene cuerpo, se podrá usar la sintaxis abreviada – con la barra de cierre al final de la etiqueta -.

Para usar una biblioteca de etiquetas en una página JSP, antes debemos declararla con una directiva taglib, con la sintaxis **<%@taglib uri="taglibURI" prefix="prefijo"%>**, donde taglibURI es la URL de un descriptor de biblioteca de etiquetas, o TLD, y prefix indica el prefijo o espacio de nombres que se usará en la página para todas las etiquetas de esa biblioteca.

El descriptor de biblioteca o TLD es un fichero en formato XML que contiene información de todas las etiquetas de la biblioteca, incluyendo sus atributos y la clase Java que implementa la funcionalidad de la etiqueta. La URL no tiene por qué ser una URL verdadera, sino que puede ser una URL ficticia que se usa como identificador y que estará definida en el fichero de despliegue de la aplicación web.

Para desarrollar una biblioteca de etiquetas, lo primero que debemos hacer es crear su descriptor TLD, con la información de todas las etiquetas, indicando si podrán tener cuerpo, sus atributos, tanto opcionales como obligatorios y su manejador – la clase Java que implementa su funcionalidad -.

Posteriormente crearemos los manejadores de etiqueta, que serán clases Java que heredan de la clase *TagSupport* si no tienen cuerpo o de *BodyTagSupport* en caso de que puedan contener cuerpo. Las clases *TagSupport* y *BodyTagSupport* pertenecen a la API Tag Handler y contienen los métodos que definen el ciclo de vida de la etiqueta, tales como *doStartTag()*, *doInitBody()*, *doAfterBody()*, *doEndTag()*. Dichos métodos contendrán el código ejecutable que implementará la funcionalidad de la etiqueta, y serán llamados desde el servlet generado por la página para procesar las distintas etapas del ciclo de vida de la etiqueta.

En dichos métodos podremos usar los objetos implícitos así como otras variables. Además, crearemos un método setter para establecer el valor de cada atributo. La API Tag Handler está disponible en el paquete *javax.servlet.jsp.tagext*. Para ver una explicación más detallada sobre el desarrollo de bibliotecas de etiquetas, se recomienda consultar algún tutorial al respecto, como http://docs.oracle.com/javaee/5/tutorial/doc/bnamu.html.

A partir de la versión 2.0 de JSP, disponemos de un conjunto de bibliotecas de etiquetas estándar conocido como JSTL – JSP Standard Tag Library -. JSTL proporciona un conjunto muy completo de bibliotecas de etiquetas, que en la mayoría de los casos son suficientes, junto al lenguaje de expresiones, EL, para implementar la funcionalidad de las páginas JSP, evitando el uso de scriptlets y ofreciendo así la posibilidad de escribir páginas JSP con una estructura más homogénea y fácil de comprender y mantener para los diseñadores Web.

Entre las bibliotecas JSTL se encuentran las siguientes:

- **Core**: Contiene acciones generales relacionadas con las estructuras de control, el manejo de datos y las excepciones.
- **Fmt**: Dirigida a la internacionalización y formato de datos.
- **SQL**: Orientada al acceso a bases de datos.
- **XML**: Para manejar documentos XML.

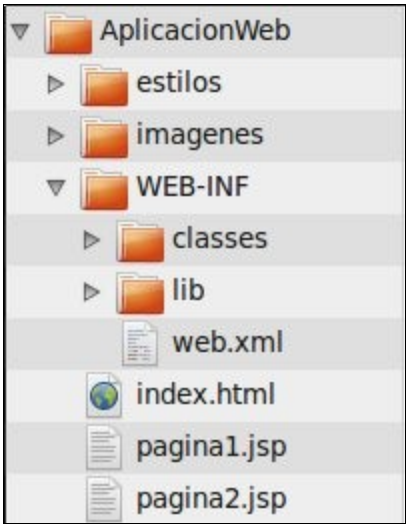
Para una referencia completa acerca de las bibliotecas JSTL, consultar el documento "JSTL Quick Reference" o alguna documentación similar. También se recomienda un tutorial como http://docs.oracle.com/javaee/5/tutorial/doc/bnamu.html.

7. Despliegue de aplicaciones Java EE.

Para ejecutar nuestras aplicaciones web Java – sin necesidad de un entorno de desarrollo – necesitamos *desplegarlas* en un servidor Java EE. Esto suele ser tan sencillo como copiar la estructura de directorios de nuestra aplicación web en el directorio de contenido del servidor Java EE. Es por ello que al servidor a veces también se le llama contenedor. En realidad, en lugar de copiar directamente la estructura de directorios de la aplicación, es preferible empaquetarla en un fichero .war – Web Application Resource -, que no es más que un fichero jar que contiene una aplicación web, y copiar éste fichero en el directorio de despliegue. Dicho directorio dependerá del servidor que estemos usando. En servidores Tomcat, por ejemplo, el directorio de despliegue será por defecto TOMCAT_BASE/webapps, donde TOMCAT_BASE es el directorio de instalación del servidor.

Por lo general, los servidores Java EE también proporcionan herramientas para desplegar aplicaciones de forma automática, tanto en forma de comandos como de herramientas de administración vía web. En ocasiones tendremos que realizar también tareas de administración en el propio servidor, ya sea para configurar opciones de seguridad, parámetros de red, etc. También para estas tareas podemos utilizar herramientas proporcionadas por el servidor o editar directamente los ficheros de configuración.

La estructura de directorios de una aplicación web Java es como se muestra a continuación:



En la imagen anterior podemos apreciar que el directorio raiz de la aplicación, AplicacionWeb, contiene directamente ficheros html y jsp, así como directorios de recursos, tales como estilos e imágenes, de la misma forma que en un servidor web normal. Además podemos tener cualquier estructura de subdirectorios para alojar las páginas web y jsp, así como otros recursos.

El directorio de la aplicación debe contener además un directorio llamado WEB-INF, que contendrá un fichero llamado web.xml, que es el descriptor de la aplicación. El directorio WEB-INF podrá contener también otros ficheros como por ejemplo, descriptores TLD, y los directorios classes y lib, que contendrán a su vez clases Java compiladas, con la estructura adecuada de paquetes y ficheros jar de bibliotecas, respectivamente. Entre las clases Java que se encuentran dentro del directorio classes pueden existir servlets, clases de utilidad, JavaBeans, etc. También podemos poner aquí, ficheros de propiedades, para contener, por ejemplo, los mensajes internacionalizados de nuestra aplicación.

Es importante destacar que el directorio WEB-INF se mantiene en privado, es decir, el servidor no publica su contenido, y por lo tanto, no atenderá a ninguna petición de un recurso que se encuentre dentro de él. Por ésta razón, dicho directorio es adecuado para almacenar cualquier contenido de la aplicación que deba protegerse.

El fichero web.xml es, como ya hemos mencionado, el descriptor de la aplicación web, y está en formato XML. Contiene una serie de secciones, muchas de ellas opcionales, que describen distintas partes de la aplicación. Algunas de las secciones más importantes son las que definen los servlets de la aplicación, así como su mapeo, el fichero de inicio –welcome file -, los descriptores TLD, la configuración de la sesión, la página de errror predeterminada, etc